Crunchy PostgreSQL Operator

Contents

Crunchy PostgreSQL Operator

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

Latest Release: {{< param operatorVersion >}}

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({{< relref "/architecture/provisioning.md" >}}) Create, Scale, & Delete PostgreSQL clusters with ease, while fully customizing your Pods and PostgreSQL configuration!

[High Availability]({{< relref "/architecture/high-availability/_index.md" >}}) 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.

Support for [standby PostgreSQL clusters]({{< relref "/architecture/high-availability/multi-cluster-kubernetes.md" >}}) that work both within an across [multiple Kubernetes clusters]({{< relref "/architecture/high-availability/multi-cluster-kubernetes.md" >}}).

[Disaster Recovery]({{< relref "/architecture/disaster-recovery.md" >}}) 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!

TLS Secure communication between your applications and data servers by enabling TLS for your PostgreSQL servers, including the ability to enforce that all of your connections to use TLS.

Monitoring({{< relref "/architecture/monitoring.md" >}}) [Track the health of your PostgreSQL clusters]({{< relref "/architecture/monitoring.md" >}}) 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 or backups with `pgo create cluster --restore-from`.

Connection Pooling Use pgBouncer({{< relref "tutorial/pgbouncer.md" >}}) for connection pooling.

Affinity and Tolerations Have your PostgreSQL clusters deployed to Kubernetes Nodes of your preference with [node affinity]({{< relref "architecture/high-availability/*index.md">}}#node-affinity), or designate which nodes Kubernetes can schedule PostgreSQL instances to with [tolerations]({{< relref "architecture/high-availability/*index.md">}}#tolerations).

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
- [Customize your PostgreSQL configuration]({{< relref "/advanced/custom-configuration.md" >}})
- Bring your own trusted certificate authority (CA) for use with the Operator API server
- Override your PostgreSQL configuration for each cluster

How it Works

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.

Included Components

PostgreSQL containers deployed with the PostgreSQL Operator include the following components:

- PostgreSQL
- PostgreSQL Contrib Modules
- PL/Python + PL/Python 3
- PL/Perl
- pgAudit
- pgAudit Analyze
- pgnodemx
- set_user
- wal2json
- pgBackRest
- pgBouncer
- pgAdmin 4
- pgMonitor
- Patroni
- LLVM (for JIT compilation)

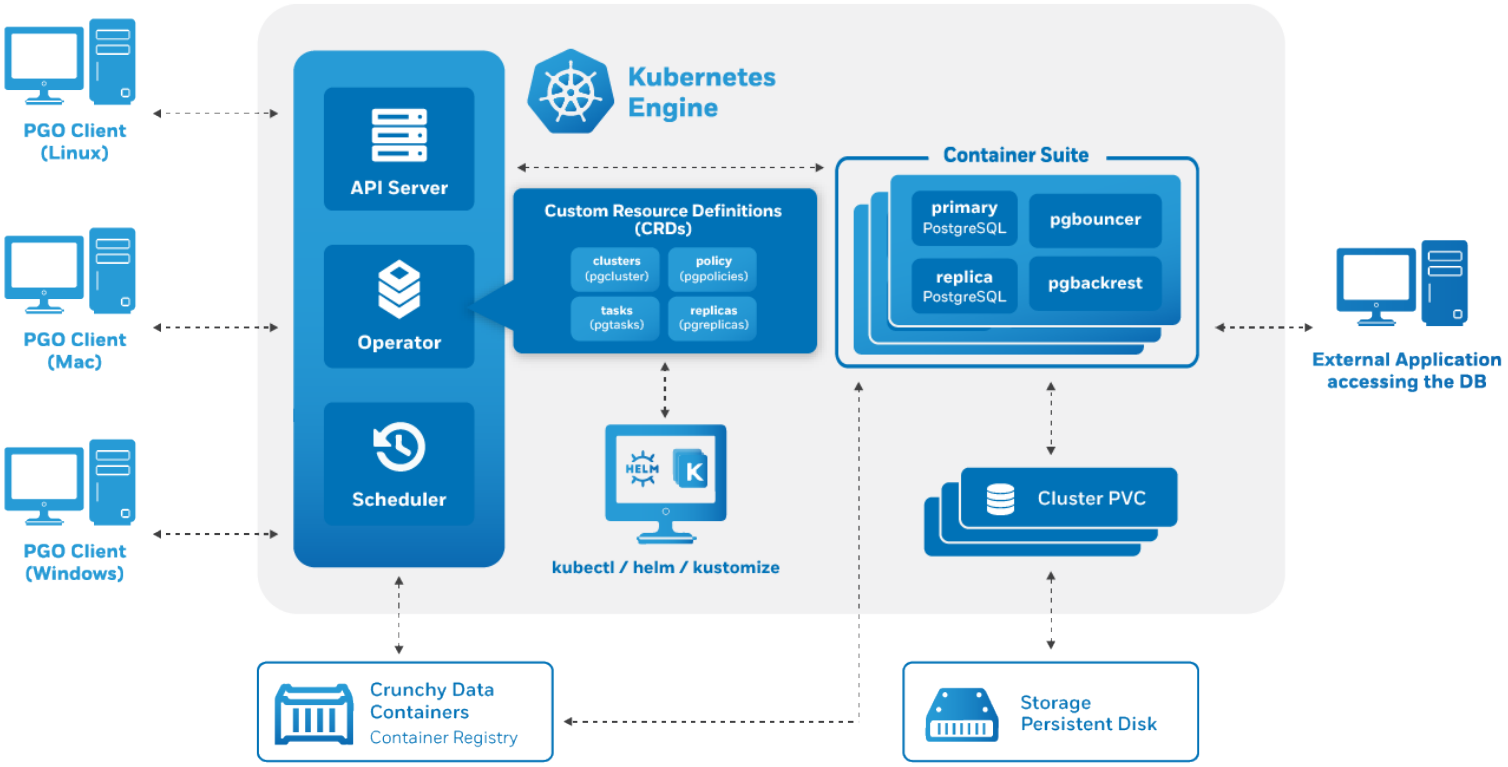

Figure 1: Architecture

In addition to the above, the geospatially enhanced PostgreSQL + PostGIS container adds the following components:

- PostGIS
- pgRouting
- PL/R

PostgreSQL Operator Monitoring({{< relref “architecture/monitoring/_index.md” >}}) uses the following components:

- pgMonitor
- Prometheus
- Grafana
- Alertmanager

Additional containers that are not directly integrated with the PostgreSQL Operator but can work alongside it include:

- pgPool II
- pg_upgrade
- pgBench

For more information about which versions of the PostgreSQL Operator include which components, please visit the [compatibility]({{< relref “configuration/compatibility.md” >}}) section of the documentation.

Supported Platforms

The Crunchy PostgreSQL Operator maintains backwards compatibility to Kubernetes 1.11 and is tested is tested against the following Platforms:

- Kubernetes 1.17+
- Openshift 4.4+

- OpenShift 3.11
- Google Kubernetes Engine (GKE), including Anthos
- Amazon EKS
- Microsoft AKS
- VMware Enterprise PKS 1.3+

This list only includes the platforms that the PostgreSQL Operator is specifically tested on as part of the release process: the PostgreSQL Operator works on other Kubernetes distributions as well. ## Storage

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

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

and more. We have had reports of people using the PostgreSQL Operator with other Storage Classes as well.

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. Storage is a rapidly evolving field in Kubernetes and we will continue to adapt the PostgreSQL Operator to modern Kubernetes storage standards.

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 the PostgreSQL Operator Installer
- Installation via a Marketplace
- Installation via [Operator Lifecycle Manager]({{< relref "/installation/other/operator-hub.md" >}})
- Installation via [Google Cloud Marketplace]({{< relref "/installation/other/google-cloud-marketplace.md" >}})

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. You can find out more information about how to get started with one of those installers in the Installation({{< relref "/installation/_index.md" >}}) section.

PostgreSQL Operator Installer

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

Installation

Install the PostgreSQL Operator

On environments that have a default storage class set up (which is most modern Kubernetes environments), the below command should work:

```
kubectl create namespace pgo
kubectl apply -f https://raw.githubusercontent.com/CrunchyData/postgres-operator/v{{< param
    operatorVersion >}}/installers/kubectl/postgres-operator.yml
```

This will launch the `pgo-deployer` container that will run the various setup and installation jobs. This can take a few minutes to complete depending on your Kubernetes cluster.

If your install is unsuccessful, you may need to modify your configuration. Please read the "Troubleshooting" section. You can still get up and running fairly quickly with just a little bit of configuration.

Install the pgo Client

During or after the installation of the PostgreSQL Operator, download the pgo client set up script. This will help set up your local environment for using the PostgreSQL Operator:

```
curl https://raw.githubusercontent.com/CrunchyData/postgres-operator/v{{< param operatorVersion
    >}}/installers/kubectl/client-setup.sh > client-setup.sh
chmod +x client-setup.sh
```

When the PostgreSQL Operator is done installing, run the client setup script:

```
./client-setup.sh
```

This will download the pgo client and provide instructions for how to easily use it in your environment. It will prompt you to add some environmental variables for you to set up in your session, which you can do with the following commands:

```
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.key"
export PGO_APISERVER_URL='https://127.0.0.1:8443'
export PGO_NAMESPACE=pgo
```

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.key"
export PGO_APISERVER_URL='https://127.0.0.1:8443'
export PGO_NAMESPACE=pgo
EOF

source ~/.bashrc
```

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

Post-Installation Setup

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 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 {{< param operatorVersion >}}
pgo-apiserver version {{< param operatorVersion >}}
```

Create a PostgreSQL Cluster

The quickstart installation method creates a namespace called `pgo` where the PostgreSQL Operator manages PostgreSQL clusters. Try creating a PostgreSQL cluster called `hippo`:

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

Alternatively, because we set the [`PGO_NAMESPACE`]({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}#general-notes-on-using-the-pgo-client) environmental variable in our `.bashrc` file, we could omit the `-n` flag from the `pgo create cluster`({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}) 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`({{< relref "pgo-client/reference/pgo_test.md" >}}) command:

```
pgo test -n pgo 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 pgo hippo`.

Connect to a PostgreSQL Cluster

By default, the PostgreSQL Operator creates a database inside the cluster with the same name of the cluster, in this case, `hippo`. Below demonstrates how we can connect to `hippo`.

How Users Work

You can get information about the users in your cluster with the [`pgo show user`]({{< relref "pgo-client/reference/pgo_show_user.md" >}}) command:

```
pgo show user -n pgo hippo
```

This will give you all the unprivileged, non-system PostgreSQL users for the `hippo` PostgreSQL cluster, for example:

```
CLUSTER USERNAME PASSWORD                  EXPIRES STATUS ERROR
------- -------- ------------------------- ------- ------ -----
hippo   testuser datalake                  never   ok
```

To get the information about all PostgreSQL users that the PostgreSQL Operator is managing, you will need to use the `--show-system-accounts` flag:

```
pgo show user -n pgo hippo --show-system-accounts
```

which returns something similar to:

```
CLUSTER USERNAME       PASSWORD                 EXPIRES STATUS ERROR
------- -------------- ------------------------ ------- ------ -----
hippo   postgres       <REDACTED>               never   ok
hippo   primaryuser    <REDACTED>               never   ok
hippo   testuser       datalake                 never   ok
```

The `postgres` user represents the database superuser and has every privilege granted to it. The PostgreSQL Operator securely interfaces through the `postgres` account to perform certain actions, such as managing users.

The `primaryuser` is the used for replication and [high availability]({{< relref "architecture/high-availability/_index.md" >}}). You should never need to interface with this user account.

Connecting via `psql`

Let's see how we can connect to `hippo` using `psql`, the command-line tool for accessing PostgreSQL. Ensure you have installed the `psql` client.

The PostgreSQL Operator creates a service with the same name as the cluster. See for yourself! Get a list of all of the Services available in the `pgo` namespace:

```
kubectl -n pgo get svc

NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
                         AGE
hippo                         ClusterIP   10.96.218.63    <none>        2022/TCP,5432/TCP
               59m
hippo-backrest-shared-repo    ClusterIP   10.96.75.175    <none>        2022/TCP
                         59m
postgres-operator             ClusterIP   10.96.121.246   <none>        8443/TCP,4171/TCP,4150/TCP
     71m
```

Let's connect the `hippo` cluster. First, in a different console window, set up a port forward to the `hippo` service:

```
kubectl -n pgo port-forward svc/hippo 5432:5432
```

You can connect to the database with the following command, substituting `datalake` for your actual password:

```
PGPASSWORD=datalake psql -h localhost -p 5432 -U testuser hippo
```

You should then be greeted with the PostgreSQL prompt:

```
psql ({{< param postgresVersion >}})
Type "help" for help.

hippo=>
```

Connecting via [pgAdmin 4]({{< relref "architecture/pgadmin4.md" >}})

[pgAdmin 4]({{< relref "architecture/pgadmin4.md" >}}) is a graphical tool that can be used to manage and query a PostgreSQL database from a web browser. The PostgreSQL Operator provides a convenient integration with pgAdmin 4 for managing how users can log into the database.

To add pgAdmin 4 to `hippo`, you can execute the following command:

```
pgo create pgadmin -n pgo hippo
```

It will take a few moments to create the pgAdmin 4 instance. The PostgreSQL Operator also creates a pgAdmin 4 service. See for yourself! Get a list of all of the Services available in the `pgo` namespace:

```
kubectl -n pgo get svc

NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
                         AGE
hippo                         ClusterIP   10.96.218.63    <none>        2022/TCP,5432/TCP
               59m
hippo-backrest-shared-repo    ClusterIP   10.96.75.175    <none>        2022/TCP
                         59m
```

```
hippo-pgadmin                 ClusterIP   10.96.165.27    <none>        5050/TCP
                    5m1s
postgres-operator             ClusterIP   10.96.121.246   <none>        8443/TCP,4171/TCP,4150/TCP
     71m
```

Let's connect to our `hippo` cluster via pgAdmin 4! In a different terminal, set up a port forward to pgAdmin 4:

```
kubectl -n pgo port-forward svc/hippo-pgadmin 5050:5050
```

Navigate your browser to http://localhost:5050 and use your database username (`testuser`) and password (e.g. `datalake`) to log in. Though the prompt says "email address", using your PostgreSQL username will work:

Figure 2: pgAdmin 4 Login Page

(There are occasions where the initial credentials do not properly get set in pgAdmin 4. If you have trouble logging in, try running the command `pgo update user -n pgo hippo --username=testuser --password=datalake`).

Once logged into pgAdmin 4, you will be automatically connected to your database. Explore pgAdmin 4 and run some queries!

For more information, please see the section on [pgAdmin 4]({{< relref "architecture/pgadmin4.md" >}}).

Troubleshooting

Installation Failures

Some Kubernetes environments may require you to customize the configuration for the PostgreSQL Operator installer. The below provides a guide on the common parameters that require modification, though this may vary based on your installation. For a full reference, please visit the Installation({{< relref "/installation/_index.md" >}}) section.

If you already attempted to install the PostgreSQL Operator and that failed, the easiest way to clean up that installation is to delete the Namespace that you attempted to install the PostgreSQL Operator into. **Note: This deletes all of the other objects in the Namespace, so please be sure this is OK!**

To delete the namespace, you can run the following command:

```
kubectl delete namespace pgo
```

Get the PostgreSQL Operator Installer Manifest You will need to download the PostgreSQL Operator Installer manifest to your environment, which you can do with the following command:

```
curl https://raw.githubusercontent.com/CrunchyData/postgres-operator/v{{< param operatorVersion
   >}}/installers/kubectl/postgres-operator.yml > postgres-operator.yml
```

Configure the PostgreSQL Operator Installer There are many [configuration parameters]({{< relref "/installation/configuration.md">}}) to help you fine tune your installation, but there are a few that you may want to change to get the PostgreSQL Operator to run in your environment. Open up the `postgres-operator.yml` file and edit a few variables.

Find the `pgo_admin_password` variable. This is the password you will use with the [`pgo` client]({{< relref "/installation/pgo-client" >}}) to manage your PostgreSQL clusters. The default is `password`, but you can change it to something like `hippo-elephant`.

You may also need to set the storage default storage classes that you would like the PostgreSQL Operator to use. These variables are called `primary_storage`, `replica_storage`, `backup_storage`, and `backrest_storage`. There are several storage configurations listed out in the configuration file under the heading `storage[1-9]_name`. Find the one that you want to use, and set it to that value.

For example, if your Kubernetes environment is using NFS storage, you would set these variables to the following:

```
backrest_storage: "nfsstorage"
backup_storage: "nfsstorage"
primary_storage: "nfsstorage"
replica_storage: "nfsstorage"
```

If you are using either Openshift or CodeReady Containers and you have a `restricted` Security Context Constraint, you will need to set `disable_fsgroup` to `true` in order to deploy the PostgreSQL Operator.

For a full list of available storage types that can be used with this installation method, please review the [configuration parameters]({{< relref "/installation/configuration.md">}}).

When you are done editing the file, you can install the PostgreSQL Operator by running the following commands:

```
kubectl create namespace pgo
kubectl apply -f postgres-operator.yml
```

The PostgreSQL Operator provides functionality that lets you run your own database-as-a-service: from deploying PostgreSQL clusters with [high availability]({{< relref "architecture/high-availability/_index.md" >}}), to a [full stack monitoring]({{< relref "architecture/high-availability/_index.md" >}}) solution, essential [disaster recovery and backup tools]({{< relref "architecture/disaster-recovery.md" >}}), the ability to secure your cluster with TLS, and much more!

What's more, you can manage your PostgreSQL clusters with the convenient [`pgo` client]({{< relref "pgo-client/_index.md" >}}) or by interfacing directly with the PostgreSQL Operator [custom resources]({{< relref "custom-resources/_index.md" >}}).

Given the robustness of the PostgreSQL Operator, we think it's helpful to break down the functionality in this step-by-step tutorial. The tutorial covers the essential functions the PostgreSQL Operator can perform and covers many common basic and advanced use cases.

So what are you waiting for? Let's [get started]({{< relref "tutorial/getting-started.md" >}})!

Installation

If you have not installed the PostgreSQL Operator yet, we recommend you take a look at our [quickstart]({{< relref "quickstart/_index.md" >}}) or the [installation]({{< relref "installation/_index.md" >}}) sections.

Customizing an Installation

How to customize a PostgreSQL Operator installation is a lengthy topic. The details are covered in the [installation]({{< relref "installation/postgres-operator.md" >}}) section, as well as a list of all the [configuration variables]({{< relref "installation/configuration.md" >}}) available.

Setup the `pgo` Client

This tutorial will be using the [`pgo` client]({{< relref "pgo-client/_index.md" >}}) to interact with the PostgreSQL Operator. Please follow the instructions in the [quickstart]({{< relref "quickstart/_index.md" >}}) or the [installation]({{< relref "installation/pgo-client.md" >}}) sections for how to configure the `pgo` client.

The PostgreSQL Operator and `pgo` client are designed to work in a [multi-namespace deployment environment]({{< relref "architecture/namespace.md" >}}) and many `pgo` commands require that the namespace flag (`-n`) are passed into it. You can use the `PGO_NAMESPACE` environmental variable to set which namespace a `pgo` command can use. For example:

```
export PGO_NAMESPACE=pgo
pgo show cluster --all
```

would show all of the PostgreSQL clusters deployed to the `pgo` namespace. This is equivalent to:

```
pgo show cluster -n pgo --all
```

(Note: `-n` takes precedence over `PGO_NAMESPACE`.)

For convenience, we will use the `pgo` namespace created as part of the [quickstart]({{< relref "quickstart/_index.md" >}}) in this tutorial. In the shell that you will be executing the `pgo` commands in, run the following command:

```
export PGO_NAMESPACE=pgo
```

Next Steps

Before proceeding, please make sure that your `pgo` client setup can communicate with your PostgreSQL Operator. In a separate terminal window, set up a port forward to your PostgreSQL Operator:

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

The [`pgo version`]({{< relref "pgo-client/reference/pgo_version.md" >}}) command is a great way to check connectivity with the PostgreSQL Operator, as it is a very simple, safe operation. Try it out:

```
pgo version
```

If it is working, you should see results similar to:

```
pgo client version {{< param operatorVersion >}}
pgo-apiserver version {{< param operatorVersion >}}
```

Note that the version of the `pgo` client **must** match that of the PostgreSQL Operator.

You can also use the `pgo version` command to check the version specifically for the `pgo` client. This command only runs locally, i.e. it does not make any requests to the PostgreSQL Operator. For example:

```
pgo version --client
```

which yields results similar to:

```
pgo client version {{< param operatorVersion >}}
```

Alright, we're now ready to start our journey with the PostgreSQL Operator!

If you came here through the [quickstart]({{< relref "quickstart/_index.md" >}}), you may have already [created a cluster]({{< relref "quickstart/_index.md" >}}#create-a-postgresql-cluster), in which case, feel free to skip ahead, or read onward for a more in depth look into cluster creation!

Create a PostgreSQL Cluster

Creating a cluster is simple with the `pgo create cluster`({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}) command:

```
pgo create cluster hippo
```

with output similar to:

```
created cluster: hippo
workflow id: 25c870a0-5d27-42c2-be00-92f0ba8768e7
database name: hippo
users:
    username: testuser password: securerandomlygeneratedpassword
```

This creates a new PostgreSQL cluster named `hippo` with a database in it named `hippo`. This operation may take a few moments to complete. Note the name of the database user (`testuser`) and password (`securerandomlygeneratedpassword`) for when we connect to the PostgreSQL cluster.

To make it easier to copy and paste statements used throughout this tutorial, you can set the password of `testuser` as part of creating the PostgreSQL cluster:

```
pgo create cluster hippo --password=securerandomlygeneratedpassword
```

You can check on the status of the cluster creation using the `pgo test`({{< relref "pgo-client/reference/pgo_test.md" >}}) 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`.

For example, when the `hippo` cluster is ready,

```
pgo test hippo
```

will yield output similar to:

```
cluster : hippo
    Services
        primary (10.96.179.126:5432): UP
    Instances
        primary (hippo-57675d4f8f-wwx64): UP
```

The Create Cluster Process

So what just happened? Let's break down what occurs during the create cluster process.

1. First, `pgo` client creates an entry in the PostgreSQL Operator [pgcluster custom resource definition]({{< relref "custom-resources/_index.md" >}}) with the attributes desired to create the cluster. In the case above, this fills in the name of the cluster (`hippo`) and leverages a lot of defaults from the [PostgreSQL Operator configuration]({{< relref "configuration/pgo-yaml-configuration.md" >}}). We'll discuss more about the PostgreSQL Operator configuration later in the tutorial.
2. Once the custom resource is added, the PostgreSQL Operator begins provisioning the PostgreSQL instace and a pgBackRest repository which is used to store backups. The following actions occur as part of this process:

- Creating persistent volume claims (PVCs) for the PostgreSQL instance and the pgBackRest repository.
- Creating services that provide a stable network interface for connecting to the PostgreSQL instance and pgBackRest repository.
- Creating deployments that house each PostgreSQL instance and pgBackRest repository. Each of these is responsible for one Pod.
- The PostgreSQL Pod, when it is started, provisions a PostgreSQL database and performs other bootstrapping functions, such as creating `testuer`.
- The pgBackRest Pod, when it is started, initializes a pgBackRest repository. Note that the pgBackRest repository is not yet ready to start taking backups, but will be after the next step!

3. When the PostgreSQL Operator detects that the PostgreSQL and pgBackRest deployments are up and running, it creates a Kubernetes Job to create a pgBackRest stanza. This is necessary as part of intializing the pgBackRest repository to accept backups from our PostgreSQL cluster.
4. When the PostgreSQL Operator detects that the stanza creation is completed, it will take an initial backup of the cluster.

In order for a PostgreSQL cluster to be considered successfully created, all of these steps need to succeed. You can connect to the PostgreSQL cluster after step two completes, but note for the cluster to be considered "healthy", you need for pgBackRest to finish initializig.

You may ask yourself, "wait, why do I need for the pgBackRest repository to be initialized for a cluster to be successfully created?" That is a good question! The reason is that pgBackRest plays a fundamental role in both the [disaster recovery]({{< relref "architecture/disaster-recovery.md" >}}) AND [high availability]({{< relref "architecture/high-availability/_index.md" >}}) system with the PostgreSQL Operator, particularly around self-healing.

What Is Created?

There are several Kubernetes objects that are created as part of the `pgo create cluster` command, including:

- A Deployment representing the primary PostgreSQL instance
- A PVC that persists the data of this instance
- A Service that can connect to this instance
- A Deployment representing the pgBackRest repository
- A PVC that persists the data of this repository
- A Service that can connect to this repository
- Secrets representing the following three user accounts:
- `postgres`: the database superuser for the PostgreSQL cluster. This is in a secret called `hippo-postgres-secret`.
- `primaryuser`: the replication user. This is used for copying data between PostgreSQL instance. You should not need to login as this user. This is in a secret called `hippo-primaryuser-secret`.
- `testuser`: the regular user account. This user has access to log into the `hippo` database that is created. This is the account you want to give out to your user / application. In a later section, we will see how we can change the default user that is created. This is in a secret called `hippo-testuser-secret`, where `testuser` can be substituted for the name of the user account.
- ConfigMaps, including:
- `hippo-pgha-config`, which allows you to [customize the configuration of your PostgreSQL cluster]({{< relref "advanced/custom-configuration.md">}}). We will cover more about this topic in later sections.
- `hippo-config` and `hippo-leader`, which are used by the high availability system. You should not modify these ConfigMaps.

Each deployment contains a single Pod. **Do not scale the deployments!**: further into the tutorial, we will cover some commands that let you scale your PostgreSQL cluster.

Some Job artifacts may be left around after the cluster creation process completes, including the stanza creation job (`hippo-stanza-create`) and initial backup job (`backrest-backup-hippo`). If the jobs completed successfully, you can safely delete these objects.

Create a PostgreSQL Cluster With Monitoring

The PostgreSQL Operator Monitoring({{< relref "architecture/monitoring.md" >}}) stack provides a convenient way to gain insights into the availabilty and performance of your PostgreSQL clusters. In order to collect metrics from your PostgreSQL clusters, you have to enable the `crunchy-postgres-exporter` sidecar alongside your PostgreSQL cluster. You can do this with the `--metrics` flag on `pgo create cluster`({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}):

```
pgo create cluster hippo --metrics
```

Note that the `--metrics` flag just enables a sidecar that can be scraped. You will need to install the [monitoring stack]({{< relref "installation/metrics/_index.md" >}}) separately, or tie it into your existing monitoring infrastructure.

Troubleshooting

PostgreSQL / pgBackRest Pods Stuck in `Pending` Phase

The most common occurrence of this is due to PVCs not being bound. Ensure that you have configure your [storage options]({{< relref "installation/configuration.md" >}}#storage-settings) correctly for your Kubernetes environment, if for some reason you cannot use your default storage class or it is unavailable.

Also ensure that you have enough persistent volumes available: your Kubernetes administrator may need to provision more.

`stanza-create` Job Never Finishes

The most common occurrence of this is due to the Kubernetes network blocking SSH connections between Pods. Ensure that your Kubernetes networking layer allows for SSH connections over port 2022 in the Namespace that you are deploying your PostgreSQL clusters into.

PostgreSQL Pod reports "Authentication Failed for `ccp_monitoring`"

This is a temporary error that occurs when a new PostgreSQL cluster is first initialized with the `--metrics` flag. The `crunchy-postgres-expo` container within the PostgreSQL Pod may be ready before the container with PostgreSQL is ready. If a message in your logs further down displays a timestamp, e.g.:

```
              now
-------------------------------
2020-11-10 08:23:15.968196-05
```

Then the `ccp_monitoring` user is properly reconciled with the PostgreSQL cluster.

If the error message does not go away, this could indicate a few things:

- The PostgreSQL instance has not initialized. Check to ensure that PostgreSQL has successfully started.
- The password for the `ccp_monitoring` user has changed. In this case you will need to update the Secret with the monitoring credentials.

Next Steps

Once your cluster is created, the next step is to [connect to your PostgreSQL cluster]({{< relref "tutorial/connect-cluster.md" >}}). You can also [learn how to customize your PostgreSQL cluster]({{< relref "tutorial/customize-cluster.md" >}})!

Naturally, once the [PostgreSQL cluster is created]({{< relref "tutorial/create-cluster.md" >}}), you may want to connect to it. You can get the credentials of the users of the cluster using the [`pgo show user`]({{< relref "pgo-client/reference/pgo_show_user.md" >}}) command, i.e.:

```
pgo show user hippo
```

yields output similar to:

```
CLUSTER USERNAME PASSWORD                          EXPIRES STATUS ERROR
------- -------- ------------------------------- ------- ------ -----
hippo   testuser securerandomlygeneratedpassword  never   ok
```

If you need to get the password of one of the system or privileged accounts, you will need to use the `--show-system-accounts` flag, i.e.:

```
pgo show user hippo --show-system-accounts
```

```
CLUSTER USERNAME    PASSWORD                          EXPIRES STATUS ERROR
------- ----------- ------------------------------- ------- ------ -----
hippo   postgres    B>xy}9+7wTVp)gkntf}X|H@N          never   ok
hippo   primaryuser ^zULckQy-\KPws:2UoC+szXl          never   ok
hippo   testuser    securerandomlygeneratedpassword   never   ok
```

Let's look at three different ways we can connect to the PostgreSQL cluster.

Connecting via `psql`

Let's see how we can connect to `hippo` using `psql`, the command-line tool for accessing PostgreSQL. Ensure you have installed the `psql` client.

The PostgreSQL Operator creates a service with the same name as the cluster. See for yourself! Get a list of all of the Services available in the `pgo` namespace:

```
kubectl -n pgo get svc

NAME                           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
                        AGE
hippo                          ClusterIP   10.96.218.63    <none>        2022/TCP,5432/TCP
              59m
hippo-backrest-shared-repo     ClusterIP   10.96.75.175    <none>        2022/TCP
                       59m
postgres-operator              ClusterIP   10.96.121.246   <none>        8443/TCP,4171/TCP,4150/TCP
     71m
```

Let's connect the `hippo` cluster. First, in a different console window, set up a port forward to the `hippo` service:

```
kubectl -n pgo port-forward svc/hippo 5432:5432
```

You can connect to the database with the following command, substituting `datalake` for your actual password:

```
PGPASSWORD=datalake psql -h localhost -p 5432 -U testuser hippo
```

You should then be greeted with the PostgreSQL prompt:

```
psql ({{< param postgresVersion >}})
Type "help" for help.

hippo=>
```

Connecting via [pgAdmin 4]({{< relref "architecture/pgadmin4.md" >}})

[pgAdmin 4]({{< relref "architecture/pgadmin4.md" >}}) is a graphical tool that can be used to manage and query a PostgreSQL database from a web browser. The PostgreSQL Operator provides a convenient integration with pgAdmin 4 for managing how users can log into the database.

To add pgAdmin 4 to `hippo`, you can execute the following command:

```
pgo create pgadmin -n pgo hippo
```

It will take a few moments to create the pgAdmin 4 instance. The PostgreSQL Operator also creates a pgAdmin 4 service. See for yourself! Get a list of all of the Services available in the `pgo` namespace:

```
kubectl -n pgo get svc

NAME                           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
                        AGE
```

```
hippo                          ClusterIP   10.96.218.63    <none>        2022/TCP,5432/TCP
              59m
hippo-backrest-shared-repo     ClusterIP   10.96.75.175    <none>        2022/TCP
                        59m
hippo-pgadmin                  ClusterIP   10.96.165.27    <none>        5050/TCP
                        5m1s
postgres-operator              ClusterIP   10.96.121.246   <none>        8443/TCP,4171/TCP,4150/TCP
     71m
```

Let's connect to our `hippo` cluster via pgAdmin 4! In a different terminal, set up a port forward to pgAdmin 4:

```
kubectl -n pgo port-forward svc/hippo-pgadmin 5050:5050
```

Navigate your browser to http://localhost:5050 and use your database username (`testuser`) and password (e.g. `datalake`) to log in. Though the prompt says "email address", using your PostgreSQL username will work:

Figure 3: pgAdmin 4 Login Page

(There are occasions where the initial credentials do not properly get set in pgAdmin 4. If you have trouble logging in, try running the command `pgo update user -n pgo hippo --username=testuser --password=datalake`).

Once logged into pgAdmin 4, you will be automatically connected to your database. Explore pgAdmin 4 and run some queries!

Connecting from a Kubernetes Application

Within a Kubernetes Cluster

Connecting a Kubernetes application that is within the same cluster that your PostgreSQL cluster is deployed in is as simple as understanding the default Kubernetes DNS system. A cluster created by the PostgreSQL Operator automatically creates a Service of the same name (e.g. `hippo`).

Following the example we've created, the hostname for our PostgreSQL cluster is `hippo.pgo` (or `hippo.pgo.svc.cluster.local`). To get your exact DNS resolution rules, you may need to consult with your Kubernetes administrator.

Knowing this, we can construct a Postgres URI that contains all of the connection info:

```
postgres://testuser:securerandomlygeneratedpassword@hippo.jkatz.svc.cluster.local:5432/hippo
```

which breaks down as such:

- `postgres`: the scheme, i.e. a Postgres URI
- `testuser`: the name of the PostgreSQL user
- `securerandomlygeneratedpassword`: the password for `testuser`
- `hippo.jkatz.svc.cluster.local`: the hostname
- `5432`: the port
- `hippo`: the database you want to connect to

If your application or connection driver cannot use the Postgres URI, the above should allow for you to break down the connection string into its appropriate components.

Outside a Kubernetes Cluster

To connect to a database from an application that is outside a Kubernetes cluster, you will need to set one of the following:

- A Service type of `LoadBalancer` or `NodePort`
- An Ingress. The PostgreSQL Operator does not provide any management for Ingress types.

To have the PostgreSQL Operator create a Service that is of type `LoadBalancer` or `NodePort`, you can use the `--service-type` flag as part of creating a PostgreSQL cluster, e.g.:

```
pgo create cluster hippo --service-type=LoadBalancer
```

You can also set the `ServiceType` attribute of the [PostgreSQL Operator configuration]({{< relref "configuration/pgo-yaml-configuration.md" >}}) to provide a default Service type for all PostgreSQL clusters that are created.

Next Steps

We've created a cluster and we've connected to it! Now, let's [learn what customizations we can make as part of the cluster creation process]({{< relref "tutorial/customize-cluster.md" >}}).

The PostgreSQL Operator makes it very easy and quick to [create a cluster]({{< relref "tutorial/create-cluster.md" >}}), but there are possibly more customizations you want to make to your cluster. These include:

- Resource allocations (e.g. Memory, CPU, PVC size)
- Sidecars (e.g. Monitoring({{< relref "architecture/monitoring.md" >}}), pgBouncer({{< relref "tutorial/pgbouncer.md" >}}), [pgAdmin 4]({{< relref "architecture/pgadmin4.md" >}}))
- High Availability (e.g. adding replicas)
- Specifying specific PostgreSQL images (e.g. one with PostGIS)
- Specifying a Pod anti-affinity and Node affinity
- Enable and/or require TLS for all connections
- [Custom PostgreSQL configurations]({{< relref "advanced/custom-configuration.md" >}})

and more.

There are an abundance of ways to customize your PostgreSQL clusters with the PostgreSQL Operator. You can read about all of these options in the `pgo create cluster`({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}) reference.

The goal of this section is to present a few of the common actions that can be taken to help create the PostgreSQL cluster of your choice. Later sections of the tutorial will cover other topics, such as creating a cluster with TLS or tablespaces.

Create a PostgreSQL Cluster With Monitoring

The PostgreSQL Operator Monitoring({{< relref "architecture/monitoring.md" >}}) stack provides a convenient way to gain insights into the availabilty and performance of your PostgreSQL clusters. In order to collect metrics from your PostgreSQL clusters, you have to enable the `crunchy-postgres-exporter` sidecar alongside your PostgreSQL cluster. You can do this with the `--metrics` flag on `pgo create cluster`({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}):

```
pgo create cluster hippo --metrics
```

Note that the `--metrics` flag just enables a sidecar that can be scraped. You will need to install the [monitoring stack]({{< relref "installation/metrics/_index.md" >}}) separately, or tie it into your existing monitoring infrastructure.

If you have an exiting cluster that you would like to add metrics collection to, you can use the `--enable-metrics` flag on the [`pgo update cluster`]({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}) command:

```
pgo update cluster hippo --enable-metrics
```

Customize PVC Size

Databases come in all different sizes, and those sizes can certainly change over time. As such, it is helpful to be able to specify what size PVC you want to store your PostgreSQL data.

Customize PVC Size for PostgreSQL

The PostgreSQL Operator lets you choose the size of your "PostgreSQL data directory" (aka "PGDATA" directory) using the `--pvc-size` flag. The PVC size should be selected using standard Kubernetes resource units, e.g. `20Gi`.

For example, to create a PostgreSQL cluster that has a data directory that is `20Gi` in size:

```
pgo create cluster hippo --pvc-size=20Gi
```

Customize PVC Size for pgBackRest

You can also specify the PVC size for the [pgBackRest repository]({{< relref "architecture/disaster-recovery.md" >}}) with the `--pgbackrest-pvc-size`. pgBackRest is used to store all of your backups, so you want to size it so that you can meet your backup retention policy.

For example, to create a pgBackRest repository that has a PVC sized to `100Gi` in size:

```
pgo create cluster hippo --pgbackrest-pvc-size=100Gi
```

Customize CPU / Memory

Databases have different CPU and memory requirements, often which is dictated by the amount of data in your working set (i.e. actively accessed data). Kubernetes provides several ways for Pods to manage CPU and memory resources:

- CPU & Memory Requests
- CPU & Memory Limits

A CPU or Memory Request tells Kubernetes to ensure that there is *at least* that amount of resource available on the Node to schedule a Pod to.

A CPU Limit tells Kubernetes to not let a Pod exceed utilizing that amount of CPU. A Pod will only be allowed to use that maximum amount of CPU. Similarly, a Memory limit tells Kubernetes to not let a Pod exceed a certain amount of Memory. In this case, if Kubernetes detects that a Pod has exceed a Memory limit, it will try to terminate any processes that are causing the limit to be exceed. We mention this as, prior to cgroups v2, Memory limits can potentially affect PostgreSQL availability and we advise to use them carefully.

The below goes into how you can customize the CPU and memory resources that are made available to the core deployment Pods with your PostgreSQL cluster. Customizing CPU and memory does add more resources to your PostgreSQL cluster, but to fully take advantage of additional resources, you will need to customize your PostgreSQL configuration and tune parameters such as `shared_buffers` and others.

Customize CPU / Memory for PostgreSQL

The PostgreSQL Operator provides several flags for `pgo create cluster`({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}) to help manage resources for a PostgreSQL instance:

- `--cpu`: Specify the CPU Request for a PostgreSQL instance
- `--cpu-limit`: Specify the CPU Limit for a PostgreSQL instance
- `--memory`: Specify the Memory Request for a PostgreSQL instance
- `--memory-limit`: Specify the Memory Limit for a PostgreSQL instance

For example, to create a PostgreSQL cluster that makes a CPU Request of 2.0 with a CPU Limit of 4.0 and a Memory Request of 4Gi with a Memory Limit of 6Gi:

```
pgo create cluster hippo \
  --cpu=2.0 --cpu-limit=4.0 \
  --memory=4Gi --memory-limit=6Gi
```

Customize CPU / Memory for Crunchy PostgreSQL Exporter Sidecar

If you deploy your PostgreSQL cluster with monitoring, you may want to adjust the resources of the `crunchy-postgres-exporter` sidecar that runs next to each PostgreSQL instnace. You can do this with the following flags:

- `--exporter-cpu`: Specify the CPU Request for a `crunchy-postgres-exporter` sidecar
- `--exporter-cpu-limit`: Specify the CPU Limit for a `crunchy-postgres-exporter` sidecar
- `--exporter-memory`: Specify the Memory Request for a `crunchy-postgres-exporter` sidecar
- `--exporter-memory-limit`: Specify the Memory Limit for a `crunchy-postgres-exporter` sidecar

For example, to create a PostgreSQL cluster with a metrics sidecar with custom CPU and memory requests + limits, you could do the following:

```
pgo create cluster hippo --metrics \
  --exporter-cpu=0.5 --exporter-cpu-limit=1.0 \
  --exporter-memory=256Mi --exporter-memory-limit=1Gi
```

Customize CPU / Memory for pgBackRest

You can also customize the CPU and memory requests and limits for pgBackRest with the following flags:

- `--pgbackrest-cpu`: Specify the CPU Request for pgBackRest
- `--pgbackrest-cpu-limit`: Specify the CPU Limit for pgBackRest
- `--pgbackrest-memory`: Specify the Memory Request for pgBackRest
- `--pgbackrest-memory-limit`: Specify the Memory Limit for pgBackRest

For example, to create a PostgreSQL cluster with custom CPU and memory requests + limits for pgBackRest, you could do the following:

```
pgo create cluster hippo \
  --pgbackrest-cpu=0.5 --pgbackrest-cpu-limit=1.0 \
  --pgbackrest-memory=256Mi --pgbackrest-memory-limit=1Gi
```

Create a High Availability PostgreSQL Cluster

[High availability]({{< relref “architecture/high-availability/_index.md” >}}) allows you to deploy PostgreSQL clusters with redundancy that allows them to be accessible by your applications even if there is a downtime event to your primary instance. The PostgreSQL clusters use the distributed consensus storage system that comes with Kubernetes so that availability is tied to that of your Kubenretes clusters. For an in-depth discussion of the topic, please read the [high availability]({{< relref “architecture/high-availability/_index.md” >}}) section of the documentation.

To create a high availability PostgreSQL cluster with one replica, you can run the following command:

```
pgo create cluster hippo --replica-count=1
```

You can scale up and down your PostgreSQL cluster with the [`pgo scale`]({{< relref “pgo-client/reference/pgo_scale.md” >}}) and `pgo scaledown`({{< relref “pgo-client/reference/pgo_scaledown.md” >}}) commands.

Set Tolerations for a PostgreSQL Cluster

Tolerations help with the scheduling of Pods to appropriate nodes. There are many reasons that a Kubernetes administrator may want to use tolerations, such as restricting the types of Pods that can be assigned to particular nodes.

The PostgreSQL Operator supports adding tolerations to PostgreSQL instances using the `--toleration` flag. The format for adding a toleration is as such:

```
rule:Effect
```

or

```
rule
```

where a `rule` can represent existence (e.g. `key`) or equality (`key=value`) and `Effect` is one of `NoSchedule`, `PreferNoSchedule`, or `NoExecute`. For more information on how tolerations work, please refer to the Kubernetes documentation.

You can assign multiple tolerations to a PostgreSQL cluster.

For example, to add two tolerations to a new PostgreSQL cluster, one that is an existence toleration for a key of `ssd` and the other that is an equality toleration for a key/value pair of `zone`/`east`, you can run the following command:

```
pgo create cluster hippo \
  --toleration=ssd:NoSchedule \
  --toleration=zone=east:NoSchedule
```

Tolerations can be updated on an existing cluster using the [`pgo update cluster`]({{ relref “pgo-client/reference/pgo_update_cluster.md” }}) command. For example, to add a toleration of `zone=west:NoSchedule` and remove the toleration of `zone=east:NoSchedule`, you could run the following command:

```
pgo update cluster hippo \
  --toleration=zone=west:NoSchedule \
  --toleration=zone-east:NoSchedule-
```

You can also add or edit tolerations directly on the `pgclusters.crunchydata.com` custom resource and the PostgreSQL Operator will roll out the changes to the appropriate instances.

Customize PostgreSQL Configuration

PostgreSQL provides a lot of different knobs that can be used to fine tune the configuration for your workload. While you can [customize your PostgreSQL configuration]({{< relref "advanced/custom-configuration.md" >}}) after your cluster has been deployed, you may also want to load in your custom configuration during initialization.

The PostgreSQL Operator uses Patroni to help manage cluster initialization and high availability. To understand how to build out a configuration file to be used to customize your PostgreSQL cluster, please review the Patroni documentation.

For example, let's say we want to create a PostgreSQL cluster with `shared_buffers` set to `2GB`, `max_connections` set to `30` and `password_encryption` set to `scram-sha-256`. We would create a configuration file that looks similar to:

```
---
bootstrap:
  dcs:
    postgresql:
      parameters:
        max_connections: 30
        shared_buffers: 2GB
        password_encryption: scram-sha-256
```

Save this configuration in a file called `postgres-ha.yaml`.

Next, create a ConfigMap called `hippo-custom-config` like so:

```
kubectl -n pgo create configmap hippo-custom-config --from-file=postgres-ha.yaml
```

You can then have you new PostgreSQL cluster use `hippo-custom-config` as part of its cluster initialization by using the `--custom-config` flag of `pgo create cluster`:

```
pgo create cluster hippo --custom-config=hippo-custom-config
```

After your cluster is initialized, [connect to your cluster]({{< relref "tutorial/connect-cluster.md" >}}) and confirm that your settings have been applied:

```
SHOW shared_buffers;

 shared_buffers
----------------
 2GB
```

Troubleshooting

PostgreSQL Pod Can't Be Scheduled

There are many reasons why a PostgreSQL Pod may not be scheduled:

- **Resources are unavailable**. Ensure that you have a Kubernetes Node with enough resources to satisfy your memory or CPU Request.
- **PVC cannot be provisioned**. Ensure that you request a PVC size that is available, or that your PVC storage class is set up correctly.
- **Node affinity rules cannot be satisfied**. If you assigned a node label, ensure that the Nodes with that label are available for scheduling. If they are, ensure that there are enough resources available.
- **Pod anti-affinity rules cannot be satisfied**. This most likely happens when [pod anti-affinity]({{< relref "architecture/high-availability/_index.md" >}}#how-the-crunchy-postgresql-operator-uses-pod-anti-affinity) is set to `required` and there are not enough Nodes available for scheduling. Consider adding more Nodes or relaxing your anti-affinity rules.

PostgreSQL Pod reports "Authentication Failed for `ccp_monitoring`"

This is a temporary error that occurs when a new PostgreSQL cluster is first initialized with the `--metrics` flag. The `crunchy-postgres-exp` container within the PostgreSQL Pod may be ready before the container with PostgreSQL is ready. If a message in your logs further down displays a timestamp, e.g.:

```
              now
-------------------------------
2020-11-10 08:23:15.968196-05
```

Then the `ccp_monitoring` user is properly reconciled with the PostgreSQL cluster.

If the error message does not go away, this could indicate a few things:

- The PostgreSQL instance has not initialized. Check to ensure that PostgreSQL has successfully started.
- The password for the `ccp_monitoring` user has changed. In this case you will need to update the Secret with the monitoring credentials.

PostgreSQL Pod Not Scheduled to Nodes Matching Tolerations

While Kubernetes Tolerations allow for Pods to be scheduled to Nodes based on their taints, this does not mean that the Pod *will* be assigned to those nodes. To provide Kubernetes scheduling guidance on where a Pod should be assigned, you must also use Node Affinity({{< relref "architecture/high-availability/_index.md" >}}#node-affinity).

Next Steps

As mentioned at the beginning, there are a lot more customizations that you can make to your PostgreSQL cluster, and we will cover those as the tutorial progresses! This section was to get you familiar with some of the most common customizations, and to explore how many options `pgo create cluster` has!

Now you have your PostgreSQL cluster up and running and using the resources as you see fit. What if you want to make changes to the cluster? We'll explore some of the commands that can be used to update your PostgreSQL cluster!

You've done it: your application is a huge success! It's so successful that you database needs more resources to keep up with the demand. How do you add more resources to your PostgreSQL cluster?

The PostgreSQL Operator provides several options to [update a cluster's]({{< relref "pgo-client/reference/pgo_update_cluster.md" >}}) resource utilization, including:

- Resource allocations (e.g. Memory, CPU, PVC size)
- Tablespaces
- Annotations
- Availability options
- Configuration({{< relref "advanced/custom-configuration.md" >}})

and more. There are additional actions that can be taken as well outside of the update process, including [scaling a cluster]({{< relref "architecture/high-availability/_index.md" >}}), adding a pgBouncer or [pgAdmin 4]({{< relref "architecture/pgadmin4.md" >}}) Deployment, and more.

The goal of this section is to present a few of the common actions that can be taken to update your PostgreSQL cluster so it has the resources and configuration that you require.

Update CPU / Memory

You can update the CPU and memory resources available to the Pods in your PostgreSQL cluster by using the [`pgo update cluster`]({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}) command. By using this method, the PostgreSQL instances are safely shut down and the new resources are applied in a rolling fashion (though we caution that a brief downtime may still occur).

Customizing CPU and memory does add more resources to your PostgreSQL cluster, but to fully take advantage of additional resources, you will need to [customize your PostgreSQL configuration]({{< relref "advanced/custom-configuration.md" >}}) and tune parameters such as `shared_buffers` and others.

Customize CPU / Memory for PostgreSQL

The PostgreSQL Operator provides several flags for [`pgo update cluster`]({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}) to help manage resources for a PostgreSQL instance:

- `--cpu`: Specify the CPU Request for a PostgreSQL instance
- `--cpu-limit`: Specify the CPU Limit for a PostgreSQL instance
- `--memory`: Specify the Memory Request for a PostgreSQL instance
- `--memory-limit`: Specify the Memory Limit for a PostgreSQL instance

For example, to update a PostgreSQL cluster that makes a CPU Request of 2.0 with a CPU Limit of 4.0 and a Memory Request of 4Gi with a Memory Limit of 6Gi:

```
pgo update cluster hippo \
  --cpu=2.0 --cpu-limit=4.0 \
  --memory=4Gi --memory-limit=6Gi
```

Customize CPU / Memory for Crunchy PostgreSQL Exporter Sidecar

If your PostgreSQL cluster has monitoring, you may want to adjust the resources of the `crunchy-postgres-exporter` sidecar that runs next to each PostgreSQL instnace. You can do this with the following flags:

- `--exporter-cpu`: Specify the CPU Request for a `crunchy-postgres-exporter` sidecar
- `--exporter-cpu-limit`: Specify the CPU Limit for a `crunchy-postgres-exporter` sidecar
- `--exporter-memory`: Specify the Memory Request for a `crunchy-postgres-exporter` sidecar
- `--exporter-memory-limit`: Specify the Memory Limit for a `crunchy-postgres-exporter` sidecar

For example, to update a PostgreSQL cluster with a metrics sidecar with custom CPU and memory requests + limits, you could do the following:

```
pgo update cluster hippo \
  --exporter-cpu=0.5 --exporter-cpu-limit=1.0 \
  --exporter-memory=256Mi --exporter-memory-limit=1Gi
```

Customize CPU / Memory for pgBackRest

You can also customize the CPU and memory requests and limits for pgBackRest with the following flags:

- `--pgbackrest-cpu`: Specify the CPU Request for pgBackRest
- `--pgbackrest-cpu-limit`: Specify the CPU Limit for pgBackRest
- `--pgbackrest-memory`: Specify the Memory Request for pgBackRest
- `--pgbackrest-memory-limit`: Specify the Memory Limit for pgBackRest

For example, to update a PostgreSQL cluster with custom CPU and memory requests + limits for pgBackRest, you could do the following:

```
pgo update cluster hippo \
  --pgbackrest-cpu=0.5 --pgbackrest-cpu-limit=1.0 \
  --pgbackrest-memory=256Mi --pgbackrest-memory-limit=1Gi
```

Customize PostgreSQL Configuration

PostgreSQL provides a lot of different knobs that can be used to fine tune the configuration for your workload. While you can [customize your PostgreSQL configuration]({{< relref "advanced/custom-configuration.md" >}}) after your cluster has been deployed, you may also want to load in your custom configuration during initialization.

The configuration can be customized by editing the `<clusterName>-pgha-config` ConfigMap. For example, with the `hippo` cluster:

```
kubectl -n pgo edit configmap hippo-pgha-config
```

We recommend that you read the section on how to [customize your PostgreSQL configuration]({{< relref "advanced/custom-configuration.md" >}}) to find out how to customize your configuration.

Troubleshooting

Configuration Did Not Update

Any updates to a ConfigMap may take a few moments to propagate to all of your Pods. Once it is propagated, the PostgreSQL Operator will attempt to reload the new configuration on each Pod.

If the information has propagated but the Pods have not been reloaded, you can force an explicit reload with the [`pgo reload`]({{< relref "pgo-client/reference/pgo_reload.md" >}}) command:

```
pgo reload hippo
```

Some customized configuration settings can only be applied to your PostgreSQL cluster after it is restarted. For example, to restart the `hippo` cluster, you can use the [`pgo restart`]({{< relref "pgo-client/reference/pgo_restart.md" >}}) command:

```
pgo restart hippo
```

Next Steps

We've seen how to create, customize, and update a PostgreSQL cluster with the PostgreSQL Operator. What about [deleting a PostgreSQL cluster]({{< relref "tutorial/delete-cluster.md" >}})?

There are many reasons you may want to delete a PostgreSQL cluster, and a few different questions to consider, such as do you want to permanently delete the data or save it for later use?

The PostgreSQL Operator offers several different workflows for deleting a cluster, from wiping all assets, to keeping PVCs of your data directory, your backup repository, or both.

Delete Everything

Deleting everything in a PostgreSQL cluster is a simple as using the `pgo delete cluster`({{< relref "pgo-client/reference/pgo_delete_cluste >}}) command. For example, to delete the `hippo` cluster:

```
pgo delete cluster hippo
```

This command launches a Job that uses the `pgo-rmdata` container to delete all of the Kubernetes objects associated with this PostgreSQL cluster. Once the `pgo-rmdata` Job finishes executing, all of your data, configurations, etc. will be removed.

Keep Backups

If you want to keep your backups, which can be used to [restore your PostgreSQL cluster at a later time]({{< relref "architecture/disaster-recovery.md">}}#restore-to-a-new-cluster) (a popular method for cloning and having sample data for your development team to use!), use the `--keep-backups` flag! For example, to delete the `hippo` PostgreSQL cluster but keep all of its backups:

```
pgo delete cluster hippo --keep-backups
```

This keeps the pgBackRest PVC which follows the pattern `<clusterName>-hippo-pgbr-repo` (e.g. `hippo-pgbr-repo`) and any PVCs that were created using the `pgdump` method of [`pgo backup`]({{< relref "pgo-client/reference/pgo_backup.md">}}).

Keep the PostgreSQL Data Directory

You may also want to delete your PostgreSQL cluster data directory, which is the core of your database, but remove any actively running Pods. This can be accomplished with the `--keep-data` flag. For example, to keep the data directory of the `hippo` cluster:

```
pgo delete cluster hippo --keep-data
```

Once the `pgo-rmdata` Job completes, your data PVC for `hippo` will still remain, but you will be unable to access it unless you attach it to a new PostgreSQL instance. The easiest way to access your data again is to create a PostgreSQL cluster with the same name:

```
pgo create cluster hippo
```

and the PostgreSQL Operator will re-attach your PVC to the newly running cluster.

Next Steps

We've covered the fundamental lifecycle elements of the PostgreSQL Operator, but there is much more to learn! If you're curious about how things work in the PostgreSQL Operator and how to perform daily tasks, we suggest you continue with the following sections:

- [Architecture]({{< relref "architecture/_index.md" >}})
- [Common `pgo` Client Tasks]({{< relref "pgo-client/common-tasks.md" >}})

The tutorial will now go into some more advanced topics. Up next, learn how to [secure connections to your PostgreSQL clusters with TLS]({{< relref "tutorial/tls.md" >}}).

TLS allows secure TCP connections to PostgreSQL, and the PostgreSQL Operator makes it easy to enable this PostgreSQL feature. The TLS support in the PostgreSQL Operator does not make an opinion about your PKI, but rather loads in your TLS key pair that you wish to use for the PostgreSQL server as well as its corresponding certificate authority (CA) certificate. Both of these Secrets are required to enable TLS support for your PostgreSQL cluster when using the PostgreSQL Operator, but it in turn allows seamless TLS support.

Prerequisites

There are three items that are required to enable TLS in your PostgreSQL clusters:

- A CA certificate
- A TLS private key
- A TLS certificate

There are a variety of methods available to generate these items: in fact, Kubernetes comes with its own certificate management system! It is up to you to decide how you want to manage this for your cluster. The PostgreSQL documentation also provides an example for how to generate a TLS certificate as well.

To set up TLS for your PostgreSQL cluster, you have to create two Secrets: one that contains the CA certificate, and the other that contains the server TLS key pair.

First, create the Secret that contains your CA certificate. Create the Secret as a generic Secret, and note that the following requirements **must** be met:

- The Secret must be created in the same Namespace as where you are deploying your PostgreSQL cluster
- The `name` of the key that is holding the CA **must** be `ca.crt`

There are optional settings for setting up the CA secret:

- You can pass in a certificate revocation list (CRL) for the CA secret by passing in the CRL using the `ca.crl` key name in the Secret.

For example, to create a CA Secret with the trusted CA to use for the PostgreSQL clusters, you could execute the following command:

```
kubectl create secret generic postgresql-ca -n pgo --from-file=ca.crt=/path/to/ca.crt
```

To create a CA Secret that includes a CRL, you could execute the following command:

```
kubectl create secret generic postgresql-ca -n pgo \
  --from-file=ca.crt=/path/to/ca.crt \
  --from-file=ca.crl=/path/to/ca.crl
```

Note that you can reuse this CA Secret for other PostgreSQL clusters deployed by the PostgreSQL Operator.

Next, create the Secret that contains your TLS key pair. Create the Secret as a a TLS Secret, and note the following requirement must be met:

- The Secret must be created in the same Namespace as where you are deploying your PostgreSQL cluster

```
kubectl create secret tls hippo-tls-keypair -n pgo \
  --cert=/path/to/server.crt \
  --key=/path/to/server.key
```

Now you can create a TLS-enabled PostgreSQL cluster!

Create a Postgres Cluster with TLS

Using the above example, to create a TLS-enabled PostgreSQL cluster that can accept both TLS and non-TLS connections, execute the following command:

```
pgo create cluster hippo \
  --server-ca-secret=postgresql-ca \
  --server-tls-secret=hippo-tls-keypair
```

Including the `--server-ca-secret` and `--server-tls-secret` flags automatically enable TLS connections in the PostgreSQL cluster that is deployed. These flags should reference the CA Secret and the TLS key pair Secret, respectively.

If deployed successfully, when you connect to the PostgreSQL cluster, assuming your `PGSSLMODE` is set to `prefer` or higher, you will see something like this in your `psql` terminal:

```
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression:
   off)
```

Force TLS For All Connections

There are many environments where you want to force all remote connections to occur over TLS, for example, if you deploy your PostgreSQL cluster's in a public cloud or on an untrusted network. The PostgreSQL Operator lets you force all remote connections to occur over TLS by using the `--tls-only` flag.

For example, using the setup above, you can force TLS in a PostgreSQL cluster by executing the following command:

```
pgo create cluster hippo \
  --tls-only \
  --server-ca-secret=postgresql-ca --server-tls-secret=hippo-tls-keypair
```

If deployed successfully, when you connect to the PostgreSQL cluster, assuming your `PGSSLMODE` is set to `prefer` or higher, you will see something like this in your `psql` terminal:

```
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression:
    off)
```

If you try to connect to a PostgreSQL cluster that is deployed using the `--tls-only` with TLS disabled (i.e. `PGSSLMODE=disable`), you will receive an error that connections without TLS are unsupported.

TLS Authentication for Replicas

PostgreSQL supports certificate-based authentication, which allows for PostgreSQL to authenticate users based on the common name (CN) in a certificate. Using this feature, the PostgreSQL Operator allows you to configure PostgreSQL replicas in a cluster to authenticate using a certificate instead of a password.

To use this feature, first you will need to set up a Kubernetes TLS Secret that has a CN of `primaryuser`. If you do not wish to have this as your CN, you will need to map the CN of this certificate to the value of `primaryuser` using a pg_ident username map, which you can configure as part of a [custom PostgreSQL configuration]({{< relref "/advanced/custom-configuration.md" >}}).

You also need to ensure that the certificate is verifiable by the certificate authority (CA) chain that you have provided for your PostgreSQL cluster. The CA is provided as part of the `--server-ca-secret` flag in the `pgo create cluster`({{< relref "/pgo-client/reference/pgo_create_cluster.md" >}}) command.

To create a PostgreSQL cluster that uses TLS authentication for replication, first create Kubernetes Secrets for the server and the CA. For the purposes of this example, we will use the ones that were created earlier: `postgresql-ca` and `hippo-tls-keypair`. After generating a certificate that has a CN of `primaryuser`, create a Kubernetes Secret that references this TLS keypair called `hippo-tls-replication-keypair`:

```
kubectl create secret tls hippo-tls-replication-keypair -n pgo \
  --cert=/path/to/replication.crt \
  --key=/path/to/replication.key
```

We can now create a PostgreSQL cluster and allow for it to use TLS authentication for its replicas! Let's create a PostgreSQL cluster with two replicas that also requires TLS for any connection:

```
pgo create cluster hippo \
  --tls-only \
  --server-ca-secret=postgresql-ca \
  --server-tls-secret=hippo-tls-keypair \
  --replication-tls-secret=hippo-tls-replication-keypair \
  --replica-count=2
```

By default, the PostgreSQL Operator has each replica connect to PostgreSQL using a PostgreSQL TLS mode of `verify-ca`. If you wish to perform TLS mutual authentication between PostgreSQL instances (i.e. certificate-based authentication with SSL mode of `verify-full`), you will need to create a [PostgreSQL custom configuration]({{< relref "/advanced/custom-configuration.md" >}}).

Troubleshooting

Replicas Cannot Connect to Primary

If your primary is forcing all connections over TLS, ensure that your replicas are connecting with a `sslmode` of `prefer` or higher.

If using TLS authentication with your replicas, ensure that the common name (`CN`) for the replicas is `primaryuser` or that you have set up an entry in `pg_ident` that provides a mapping from your `CN` to `primaryuser`.

Next Steps

You've now secured connections to your database. However, how do you scale and pool your PostgreSQL connections? Learn how to [set up and configure pgBouncer]({{< relref "tutorial/pgbouncer.md" >}})!

pgBouncer is a lightweight connection poooler and state manager that provides an efficient gateway to metering connections to PostgreSQL. The PostgreSQL Operator provides an integration with pgBouncer that allows you to deploy it alongside your PostgreSQL cluster.

This tutorial covers how you can set up pgBouncer, functionality that the PostgreSQL Operator provides to manage it, and more.

Setup pgBouncer

pgBouncer lives as an independent Deployment next to your PostgreSQL cluster but, thanks to the PostgreSQL Operator, is synchronized with various aspects of your environment.

There are two ways you can set up pgBouncer for your cluster. You can add pgBouncer when you create your cluster, e.g.:

```
pgo create cluster hippo --pgbouncer
```

or after your PostgreSQL cluster has been provisioned with the [`pgo create pgbouncer`]({{< relref "pgo-client/reference/pgo_create_pgbou
>}}):

```
pgo create pgbouncer hippo
```

There are several managed objects that are created alongside the pgBouncer Deployment, these include:

- The pgBouncer Deployment itself
- One or more pgBouncer Pods
- A pgBouncer ConfigMap, e.g. `hippo-pgbouncer-cm` which has two entries:
- `pgbouncer.ini`, which is the configuration for the pgBouncer instances
- `pg_hba.conf`, which controls how clients can connect to `pgBouncer`
- A pgBouncer Secret e.g. `hippo-pgbouncer-secret`, that contains the following values:
- `password`: the password for the `pgbouncer` user. The `pgbouncer` user is described in more detail further down.
- `users.txt`: the description for how the `pgbouncer` user and only the `pgbouncer` user can explicitly connect to a pgBouncer instance.

The `pgbouncer` user

The `pgbouncer` user is a special type of PostgreSQL user that is solely for the administration of pgBouncer. It performs several roles, including:

- Securely load PostgreSQL user credentials into pgBouncer so pgBouncer can perform authentication and connection forwarding
- The ability to log into `pgBouncer` itself for administration, introspection, and looking at statistics

The pgBouncer user **is not meant to be used to log into PostgreSQL directly**: the account is given permissions for ad hoc tasks. More information on how to connect to pgBouncer is provided in the next section.

Connect to a Postgres Cluster Through pgBouncer

Connecting to a PostgreSQL cluster through pgBouncer is similar to how you [connect to PostgreSQL directly]({{< relref "tutorial/connect-cluster.md">}}), but you are connecting through a different service. First, note the types of users that can connect to PostgreSQL through `pgBouncer`:

- Any regular user that's created through [`pgo create user`]({{< relref "pgo-client/reference/pgo_create_user.md" >}}) or a user that is not a system account.
- The `postgres` superuser

The following example will follow similar steps for how you would connect to a [Postgres Cluster via `psql`]({{< relref "tutorial/connect-cluster.md">}}#connection-via-psql), but applies to all other connection methods.

First, get a list of Services that are available in your namespace:

```
kubectl -n pgo get svc
```

You should see a list similar to:

```
NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
                        AGE
hippo                         ClusterIP   10.96.104.207   <none>        2022/TCP,5432/TCP
              12m
hippo-backrest-shared-repo    ClusterIP   10.96.134.253   <none>        2022/TCP
                        12m
hippo-pgbouncer               ClusterIP   10.96.85.35     <none>        5432/TCP
                        11m
```

We are going to want to create a port forward to the `hippo-pgbouncer` service. In a separate terminal window, run the following command:

```
kubectl -n pgo port-forward svc/hippo-pgbouncer 5432:5432
```

Recall in the [earlier part of the tutorial]({{< relref "tutorial/connect-cluster.md">}}) that we created a user called `testuser` with a password of `securerandomlygeneratedpassword`. We can the connect to PostgreSQL via pgBouncer by executing the following command:

```
PGPASSWORD=securerandomlygeneratedpassword psql -h localhost -p 5432 -U testuser hippo
```

You should then be greeted with the PostgreSQL prompt:

```
psql ({{< param postgresVersion >}})
Type "help" for help.

hippo=>
```

Validation: Did this actually work?

This looks just like how we connected to PostgreSQL before, so how do we know that we are connected to PostgreSQL via pgBouncer? Let's log into pgBoucner as the `pgbouncer` user and demonstrate this.

In another terminal window, get the credential for the pgBouncer user. This can be done with the [`pgo show pgbouncer`]({{< relref "pgo-client/reference/pgo_show_pgbouncer.md" >}}) command:

```
pgo show pgbouncer hippo
```

which yields something that looks like:

```
CLUSTER SERVICE          USERNAME  PASSWORD                  CLUSTER IP  EXTERNAL IP
------- ---------------- --------- ------------------------- ----------- -----------
hippo   hippo-pgbouncer pgbouncer randompassword            10.96.85.35
```

Copy the actual password and log into pgbouncer with the following command:

```
PGPASSWORD=randompassword psql -h localhost -p 5432 -U pgbouncer pgbouncer
```

You should see something similar to this:

```
psql (13.1, server 1.14.0/bouncer)
Type "help" for help.

pgbouncer=#
```

In the `pgboucner` terminal, run the following command. This will show you the overall connection statistics for pgBouncer:

```
SHOW stats;
```

Success, you have connected to pgBouncer!

Setup pgBouncer with TLS

Similarly to how you can [setup TLS for PostgreSQL]({{< relref "tutorial/tls.md" >}}), you can set up TLS connections for pgBouncer. To do this, the PostgreSQL Operator takes the following steps:

- Ensuring TLS communication between a client (e.g. `psql`, your application, etc.) and pgBouncer
- Ensuring TLS communication between pgBouncer and PostgreSQL

When TLS is enabled, the PostgreSQL Operator configures pgBouncer to require each client to use TLS to communicate with pgBouncer. Additionally, the PostgreSQL Operator requires that pgBouncer and the PostgreSQL cluster share the same certificate authority (CA) bundle, which allows for pgBouncer to communicate with the PostgreSQL cluster using PostgreSQL's `verify-ca` SSL mode.

The below guide will show you how set up TLS for pgBouncer.

Prerequisites

In order to set up TLS connections for pgBouncer, you must first [enable TLS on your PostgreSQL cluster]({{< relref "tutorial/tls.md" >}}).

For the purposes of this exercise, we will re-use the Secret TLS keypair `hippo-tls-keypair` that was created for the PostgreSQL server. This is only being done for convenience: you can substitute `hippo-tls-keypair` with a different TLS key pair as long as it can be verified by the certificate authority (CA) that you selected for your PostgreSQL cluster. Recall that the certificate authority (CA) bundle is stored in a Secret named `postgresql-ca`.

Create pgBouncer with TLS

Knowing that our TLS key pair is stored in a Secret called `hippo-tls-keypair`, you can setup pgBouncer with TLS using the following command:

```
pgo create pgbouncer hippo --tls-secret=hippo-tls-keypair
```

And that's it! So long as the prerequisites are satisfied, this will create a pgBouncer instance that is TLS enabled.

Don't believe it? Try logging in. First, ensure you have a port-forward from pgBouncer to your host machine:

```
kubectl -n pgo port-forward svc/hippo-pgbouncer 5432:5432
```

Then, connect to the pgBouncer instances:

```
PGPASSWORD=securerandomlygeneratedpassword psql -h localhost -p 5432 -U testuser hippo
```

You should see something similar to this:

```
psql (13.1)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

hippo=>
```

Still don't believe it? You can verify your connection using the PostgreSQL `get_backend_pid()` function and the `pg_stat_ssl` monitoring view:

```
hippo=> SELECT * FROM pg_stat_ssl WHERE pid = pg_backend_pid();
  pid  | ssl | version |         cipher         | bits | compression | client_dn | client_serial |
     issuer_dn
-------+-----+---------+------------------------+------+-------------+-----------+---------------+---
 15653 | t   | TLSv1.3 | TLS_AES_256_GCM_SHA384 |  256 | f           |           |               |
(1 row)
```

Create a PostgreSQL cluster with pgBouncer and TLS

Want to create a PostgreSQL cluster with pgBouncer with TLS enabled? You can with the `pgo create cluster`({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}) command and using the `--pgbouncer-tls-secret` flag. Using the same Secrets that were created in the [creating a PostgreSQL cluster with TLS]({{ relref "tutorial/tls.md" }}) tutorial, you can create a PostgreSQL cluster with pgBouncer and TLS with the following command:

```
pgo create cluster hippo \
  --server-ca-secret=postgresql-ca \
  --server-tls-secret=hippo-tls-keypair \
  --pgbouncer \
  --pgbouncer-tls-secret=hippo-tls-keypair
```

Customize CPU / Memory for pgBouncer

Provisioning

The PostgreSQL Operator provides several flags for `pgo create cluster`({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}) to help manage resources for pgBouncer:

- `--pgbouncer-cpu`: Specify the CPU Request for pgBouncer

- `--pgbouncer-cpu-limit`: Specify the CPU Limit for pgBouncer
- `--pgbouncer-memory`: Specify the Memory Request for pgBouncer
- `--pgbouncer-memory-limit`: Specify the Memory Limit for pgBouncer

Additional, the PostgreSQL Operator provides several flags for [`pgo create pgbouncer`]({{< relref "pgo-client/reference/pgo_create_pgbou
>}}) to help manage resources for pgBouncer:

- `--cpu`: Specify the CPU Request for pgBouncer
- `--cpu-limit`: Specify the CPU Limit for pgBouncer
- `--memory`: Specify the Memory Request for pgBouncer
- `--memory-limit`: Specify the Memory Limit for pgBouncer

To create a pgBouncer Deployment that makes a CPU Request of 1.0 with a CPU Limit of 2.0 and a Memory Request of 64Mi with a Memory Limit of 256Mi:

```
pgo create pgbouncer hippo \
  --cpu=1.0 --cpu-limit=2.0 \
  --memory=64Mi --memory-limit=256Mi
```

Updating

You can also add more memory and CPU resources to pgBouncer with the [`pgo update pgbouncer`]({{< relref "pgo-client/reference/pgo_up
>}}) command, including:

- `--cpu`: Specify the CPU Request for pgBouncer
- `--cpu-limit`: Specify the CPU Limit for pgBouncer
- `--memory`: Specify the Memory Request for pgBouncer
- `--memory-limit`: Specify the Memory Limit for pgBouncer

For example, to update a pgBouncer to a CPU Request of 2.0 with a CPU Limit of 3.0 and a Memory Request of 128Mi with a Memory Limit of 512Mi:

```
pgo update pgbouncer hippo \
  --cpu=2.0 --cpu-limit=3.0 \
  --memory=128Mi --memory-limit=512Mi
```

Scaling pgBouncer

You can add more pgBouncer instances when provisioning pgBouncer and to an existing pgBouncer Deployment.

Provisioning

To add pgBouncer instances when creating a PostgreSQL cluster, use the `--pgbouncer-replicas` flag on `pgo create cluster`. For example, to add 2 replicas:

```
pgo create cluster hippo --pgbouncer --pgbouncer-replicas=2
```

If adding a pgBouncer to an already provisioned PostgreSQL cluster, use the `--replicas` flag on `pgo create pgbouncer`. For example, to add a pgBouncer instance with 2 replicas:

```
pgo create pgbouncer hippo --replicas=2
```

Updating

To update pgBouncer instances to scale the replicas, use the `pgo update pgbouncer` command with the `--replicas` flag. This flag can scale pgBouncer up and down. For example, to run 3 pgBouncer replicas:

```
pgo update pgbouncer hippo --replicas=3
```

Rotate pgBouncer Password

If you wish to rotate the pgBouncer password, you can use the `--rotate-password` flag on `pgo update pgbouncer`:

```
pgo update pgbouncer hippo --rotate-password
```

This will change the pgBouncer password and synchronize the change across all pgBouncer instances.

Next Steps

Now that you have connection pooling set up, let's create a [high availability PostgreSQL cluster]({{< relref "tutorial/high-availability.md" >}})!

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.

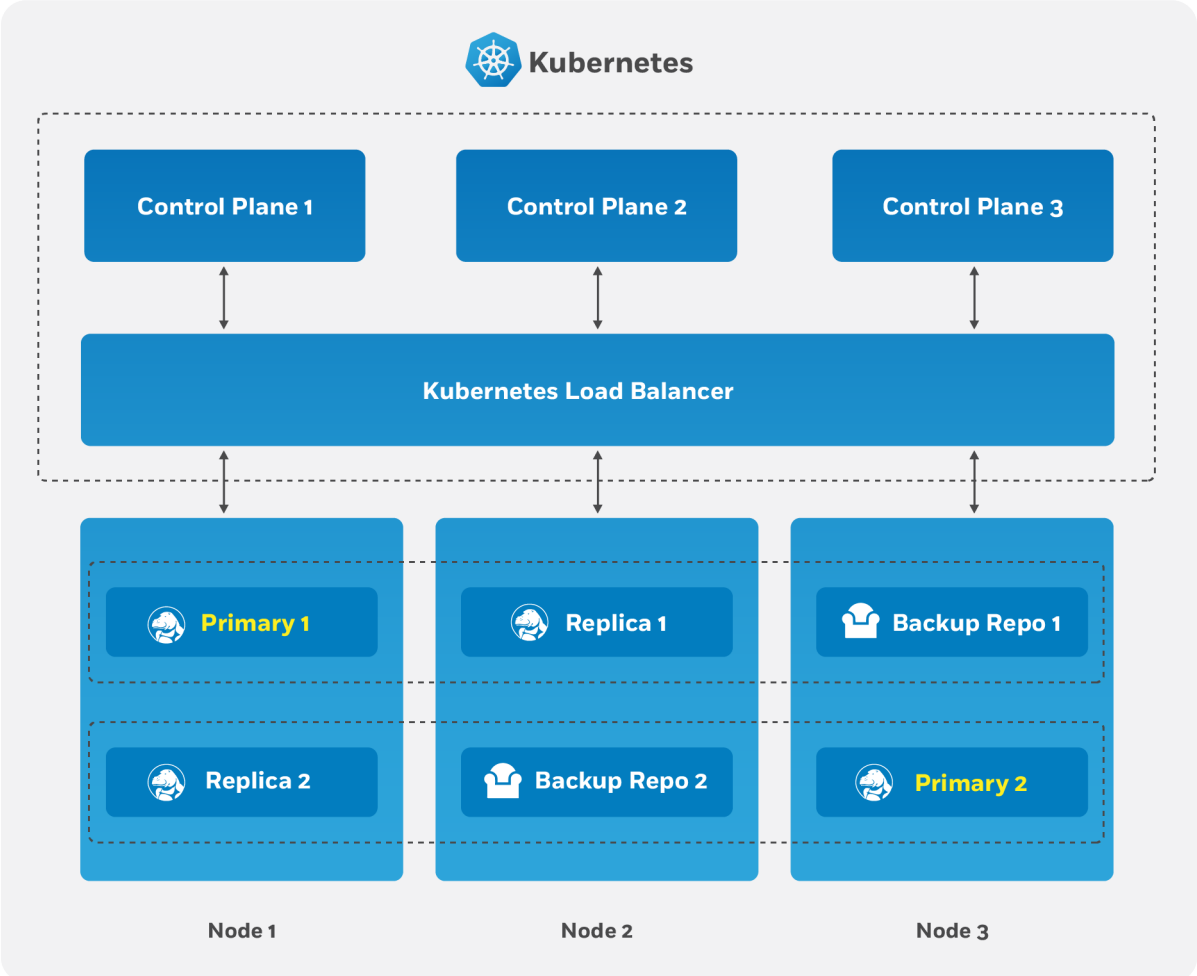

Figure 4: PostgreSQL Operator High-Availability Overview

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!

This tutorial will cover the "howtos" of high availbility. For more information on the topic, please review the detailed [high availability architecture]({{< relref "architecture/high-availability/_index.md" >}}) section.

Create a HA PostgreSQL Cluster

High availability is enabled in the PostgreSQL Operator by default so long as you have more than one replica. To create a high availability PostgreSQL cluster, you can execute the following command:

```
pgo create cluster hippo --replica-count=1
```

Scale a PostgreSQL Cluster

You can scale an existing PostgreSQL cluster to add HA to it by using the [`pgo scale`]({{< relref "pgo-client/reference/pgo_scale.md">}}) command:

```
pgo scale hippo
```

Scale Down a PostgreSQL Cluster

To scale down a PostgreSQL cluster, you will have to provide a target of which instance you want to scale down. You can do this with the `pgo scaledown`({{< relref "pgo-client/reference/pgo_scaledown.md">}}) command:

```
pgo scaledown hippo --query
```

which will yield something similar to:

```
Cluster: hippo
REPLICA                 STATUS      NODE        REPLICATION LAG         PENDING RESTART
hippo-ojnd              running     node01                0 MB                   false
```

Once you have determined which instance you want to scale down, you can run the following command:

```
pgo scaledown hippo --target=hippo-ojnd
```

Manual Failover

Each PostgreSQL cluster will manage its own availability. If you wish to manually fail over, you will need to use the [`pgo failover`]({{< relref "pgo-client/reference/pgo_failover.md">}}) command.

There are two ways to issue a manual failover to your PostgreSQL cluster:

1. Allow for the PostgreSQL Operator to select the best replica candidate for failover.
2. Select your own replica candidate for failover.

Both methods are detailed below.

Manual Failover - PostgreSQL Operator Candidate Selection

To have the PostgreSQL Operator select the best replica candidate for failover, all you need to do is execute the following command:

```
pgo failover hippo
```

The PostgreSQL Operator will determine which is the best replica candidate to fail over to, and take into account factors such as replication lag and current timeline.

Manual Failover - Manual Selection

If you wish to have your cluster manually failover, you must first query your determine which instance you want to fail over to. You can do so with the following command:

```
pgo failover hippo --query
```

which will yield something similar to:

```
Cluster: hippo
REPLICA                 STATUS      NODE        REPLICATION LAG         PENDING RESTART
hippo-ojnd              running     node01                0 MB                   false
```

Once you have determine your failover target, you can run the following command:

```
pgo failover hippo --target==hippo-ojnd
```

Synchronous Replication

If you have a [write sensitive workload and wish to use synchronous replication]({{< relref "architecture/high-availability/_index.md" >}}#synchronous-replication-guarding-against-transactions-loss), you can create your PostgreSQL cluster with synchronous replication turned on:

```
pgo create cluster hippo --sync-replication
```

Please understand the tradeoffs of synchronous replication before using it.

Pod Anti-Affinity and Node Affinity

To learn how to use pod anti-affinity and node affinity, please refer to the [high availability architecture documentation]({{< relref "architecture/high-availability/_index.md" >}}).

Tolerations

If you want to have a PostgreSQL instance use specific Kubernetes tolerations, you can use the `--toleration` flag on [`pgo scale`]({{< relref "pgo-client/reference/pgo_scale.md">}}). Any tolerations added to the new PostgreSQL instance fully replace any tolerations available to the entire cluster.

For example, to assign equality toleration for a key/value pair of `zone/west`, you can run the following command:

```
pgo scale hippo --toleration=zone=west:NoSchedule
```

For more information on the PostgreSQL Operator and tolerations, please review the [high availability architecture documentation]({{< relref "architecture/high-availability/_index.md" >}}).

Troubleshooting

No Primary Available After Both Synchronous Replication Instances Fail

Though synchronous replication is available for guarding against transaction loss for [write sensitive workloads]({{< relref "architecture/high-availability/_index.md" >}}#synchronous-replication-guarding-against-transactions-loss), by default the high availability systems prefers availability over consistency and will continue to accept writes to a primary even if a replica fails. Additionally, in most scenarios, a system using synchronous replication will be able to recover and self heal should a primary or a replica go down.

However, in the case that both a primary and its synchronous replica go down at the same time, a new primary may not be promoted. To guard against transaction loss, the high availability system will not promote any instances if it cannot determine if they had been one of the synchronous instances. As such, when it recovers, it will bring up all the instances as replicas.

To get out of this situation, inspect the replicas using `pgo failover --query` to determine the best candidate (typically the one with the least amount of replication lag). After determining the best candidate, promote one of the replicas using `pgo failover --target` command.

If you are still having issues, you may need to execute into one of the Pods and inspect the state with the `patronictl` command.

A detailed breakdown of this case be found here.

Next Steps

Backups, restores, point-in-time-recoveries: [disaster recovery]({{< relref "architecture/disaster-recovery.md" >}}) is a big topic! We'll learn about you can [perform disaster recovery]({{< relref "tutorial/disaster-recovery.md" >}}) and more in the PostgreSQL Operator.

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

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 [creating a PostgreSQL cluster]({{< relref "tutorial/create-cluster.md" >}}) section.

For more information on how disaster recovery in the PostgreSQL Operator works, please see the [disaster recovery architecture]({{< relref "architecture/disaster-recovery.md">}}) section.

Creating a Backup

The PostgreSQL Operator uses the open source pgBackRest backup and recovery utility for managing backups and PostgreSQL archives. pgBackRest has several types of backups that you can take:

- Full: Back up the entire database
- Differential: Create a backup of everything since the last full back up was taken
- Incremental: Back up everything since the last backup was taken, whether it was full, differential, or incremental

When a new PostgreSQL cluster is provisioned by the PostgreSQL Operator, a full pgBackRest backup is taken by default.

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

```
pgo backup hippo
```

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.

Creating a Full Backup

You can create a full backup using the following command:

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

Creating a Differential Backup

You can create a differential backup using the following command:

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

Creating an Incremental Backup

You can create a differential backup using the following command:

```
pgo backup hippo --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]({{< relref "architecture/disaster-recovery.md">}}#using-s3) in the architecture section.

Set 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 hippo --backup-opts="--type=full --repo1-retention-full=7"
```

pgBackRest also supports time-based retention. Please review the pgBackRest documentation for more information.

Schedule Backups

It is good practice to take backups regularly. The PostgreSQL Operator allows you to schedule backups to occur automatically.

The PostgreSQL Operator comes with a 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.

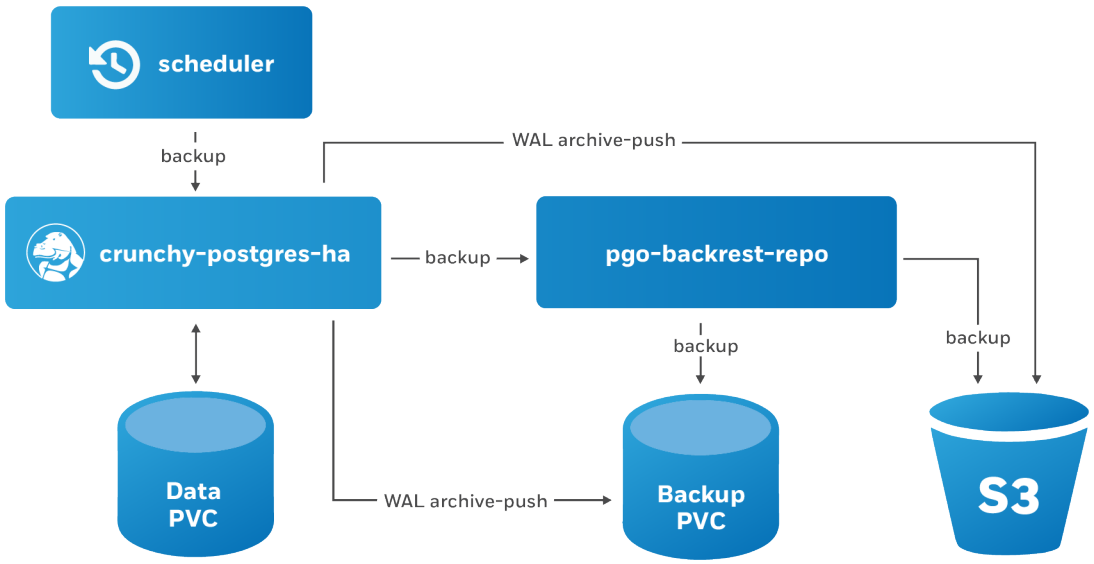

Figure 5: PostgreSQL Operator Schedule Backups

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

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

To schedule an incremental backup once every 3 hours:

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

You can also add the backup retention settings to these commands.

View Backups

You can view all of the available backups in your pgBackRest repository with the `pgo show backup` command:

```
pgo show backup hippo
```

Restores

The PostgreSQL Operator supports the ability to perform a full restore on a PostgreSQL cluster (i.e. a "clone" or "copy") as well as a point-in-time-recovery. There are two types of ways to restore a cluster:

- Restore to a new cluster using the `--restore-from` flag in the `pgo create cluster`({{< relref "/pgo-client/reference/pgo_create_clus >}}) command. This is effectively a clone or a copy.
- Restore in-place using the [`pgo restore`]({{< relref "/pgo-client/reference/pgo_restore.md" >}}) command. Note that this is **destructive**.

It is typically better to perform a restore to a new cluster, particularly when performing a point-in-time-recovery, as it can allow you to more effectively manage your downtime and avoid making undesired changes to your production data.

Additionally, the "restore to a new cluster" technique works so long as you have a pgBackRest repository available: the pgBackRest repository does not need to be attached to an active cluster! For example, if a cluster named `hippo` was deleted as such:

```
pgo delete cluster hippo --keep-backups
```

you can create a new cluster from the backups like so:

```
pgo create cluster datalake --restore-from=hippo
```

Below provides guidance on how to perform a restore to a new PostgreSQL cluster both as a full copy and to a specific point in time. Additionally, it also shows how to restore in place to a specific point in time.

Restore to a New Cluster (aka "copy" or "clone")

Restoring to a new PostgreSQL cluster allows one to take a backup and create a new PostgreSQL cluster that can run alongside an existing PostgreSQL cluster. There are several scenarios where using this technique is helpful:

- Creating a copy of a PostgreSQL cluster that can be used for other purposes. Another way of putting this is "creating a clone."
- Restore to a point-in-time and inspect the state of the data without affecting the current cluster

and more.

Restore Everything To create a new PostgreSQL cluster from a backup and restore it fully, you can execute the following command:

```
pgo create cluster datalake --restore-from=hippo
```

Partial Restore / Point-in-time-Recovery (PITR) To create a new PostgreSQL cluster and restore it to specific point-in-time (e.g. before a key table was dropped), you can use the following command, substituting the time that you wish to restore to:

```
pgo create cluster datalake \
  --restore-from hippo \
  --restore-opts "--type=time --target='2019-12-31 11:59:59.999999+00'"
```

When the restore is complete, the cluster is immediately available for reads and writes. To inspect the data before allowing connections, add pgBackRest's `--target-action=pause` option to the `--restore-opts` parameter.

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

Restore in-place

Restoring a PostgreSQL cluster in-place is a **destructive** action that will perform a recovery on your existing data directory. This is accomplished using the [`pgo restore`]({{< relref "/pgo-client/reference/pgo_restore.md" >}}) command. The most common scenario is to restore the database to a specific point in time.

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 31, 2019 at 11:59pm:

```
pgo restore hippo --pitr-target="2019-12-31 11:59:59.999999+00" \
  --backup-opts="--type=time"
```

When the restore is complete, the cluster is immediately available for reads and writes. To inspect the data before allowing connections, add pgBackRest's `--target-action=pause` option to the `--backup-opts` parameter.

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

Deleting a Backup

You typically do not want to delete backups. Instead, it's better to set a backup retention policy as part of scheduling your ackups.

However, there are situations where you may want to explicitly delete backups, in particular, if you need to reclaim space on your backup disk or if you accidentally created too many backups.

If you delete a backup that is *not* set to expire, you may be unable to meet your retention requirements. If you are deleting backups to free space, it is recommended to delete your oldest backups first.

In these cases, a backup can be deleted using the [`pgo delete backup`]({{< relref "pgo-client/reference/pgo_delete_backup.md" >}}) command. You must specify a specific backup to delete using the `--target` flag. You can get the backup names from the [`pgo show backup`]({{< relref "pgo-client/reference/pgo_show_backup.md" >}}) command.

Let's say that the `hippo` cluster currently has a set of backups that look like this, obtained from running the `pgo show backup hippo` command:

```
cluster: hippo
storage type: posix

stanza: db
    status: ok
    cipher: none

    db (current)
        wal archive min/max (12-1)

        full backup: 20201220-171801F
            timestamp start/stop: 2020-12-20 17:18:01 +0000 UTC / 2020-12-20 17:18:10 +0000 UTC
            wal start/stop: 000000010000000000000002 / 000000010000000000000002
            database size: 31.3MiB, backup size: 31.3MiB
            repository size: 3.8MiB, repository backup size: 3.8MiB
            backup reference list:

        incr backup: 20201220-171801F_20201220-171939I
            timestamp start/stop: 2020-12-20 17:19:39 +0000 UTC / 2020-12-20 17:19:41 +0000 UTC
            wal start/stop: 000000010000000000000005 / 000000010000000000000005
            database size: 31.3MiB, backup size: 216.3KiB
            repository size: 3.8MiB, repository backup size: 25.9KiB
            backup reference list: 20201220-171801F

        incr backup: 20201220-171801F_20201220-172046I
            timestamp start/stop: 2020-12-20 17:20:46 +0000 UTC / 2020-12-20 17:23:29 +0000 UTC
            wal start/stop: 00000001000000000000000A / 00000001000000000000000A
            database size: 65.9MiB, backup size: 37.5MiB
            repository size: 7.7MiB, repository backup size: 4.3MiB
            backup reference list: 20201220-171801F, 20201220-171801F_20201220-171939I

        full backup: 20201220-201305F
            timestamp start/stop: 2020-12-20 20:13:05 +0000 UTC / 2020-12-20 20:13:15 +0000 UTC
            wal start/stop: 00000001000000000000000F / 00000001000000000000000F
            database size: 65.9MiB, backup size: 65.9MiB
            repository size: 7.7MiB, repository backup size: 7.7MiB
            backup reference list:
```

Note that the backup targets can be found after the backup type, e.g. `20201220-171801F` or `20201220-171801F_20201220-172046I`.

One can delete the oldest backup, in this case `20201220-171801F`, by running the following command:

```
pgo delete backup hippo --target=20201220-171801F
```

You can then verify the backup is deleted with `pgo show backup hippo`:

```
cluster: hippo
storage type: posix

stanza: db
    status: ok
    cipher: none

    db (current)
        wal archive min/max (12-1)

        full backup: 20201220-201305F
            timestamp start/stop: 2020-12-20 20:13:05 +0000 UTC / 2020-12-20 20:13:15 +0000 UTC
            wal start/stop: 000000010000000000000000F / 000000010000000000000000F
            database size: 65.9MiB, backup size: 65.9MiB
            repository size: 7.7MiB, repository backup size: 7.7MiB
            backup reference list:
```

Note that deleting the oldest backup also had the effect of deleting all of the backups that depended on it. This is a feature of pgBackRest!

Next Steps

There are cases where you may want to take [logical backups]({{< relref "tutorial/pgdump.md" >}}), aka `pg_dump` / `pg_dumpall`. Let's learn how to do that with the PostgreSQL Operator!

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 the `postgres` database, you can run the following command:

```
pgo backup hippo --backup-type=pgdump
```

To create a logical backup of a specific database, you can use the `--database` flag, as in the following command:

```
pgo backup hippo --backup-type=pgdump --database=hippo
```

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 hippo --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 hippo --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 with the `--backup-type=pgdump` flag:

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

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 get the timestamp from the `pgo show backup --backup-type=pgdump` command.

You can restore a logical backup using the following command:

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

To restore to a specific database, add the `--pgdump-database` flag to the command from above:

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

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

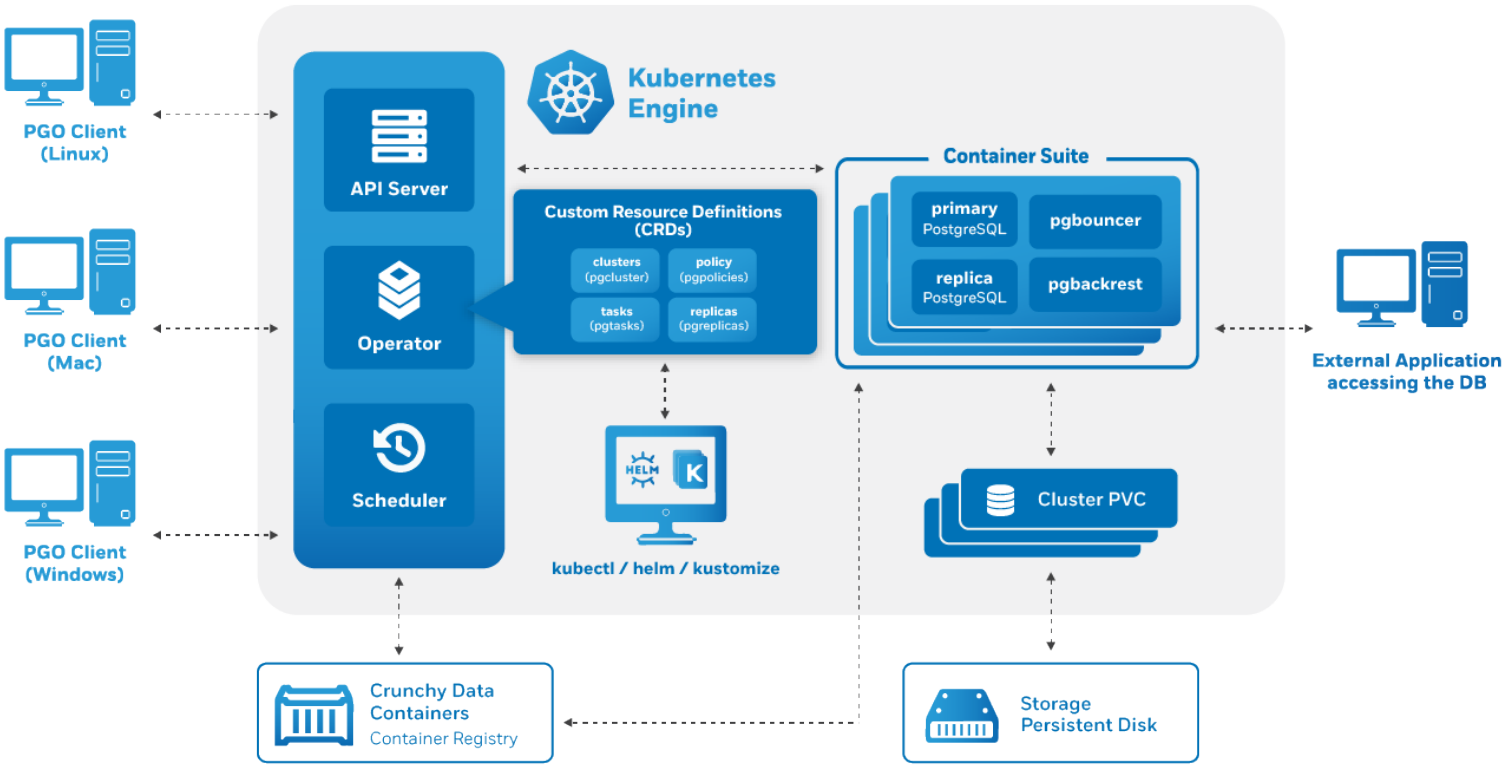

Figure 6: Operator Architecture with CRDs

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. take a backup) and tracks the state of said task through 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, etc.)

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:

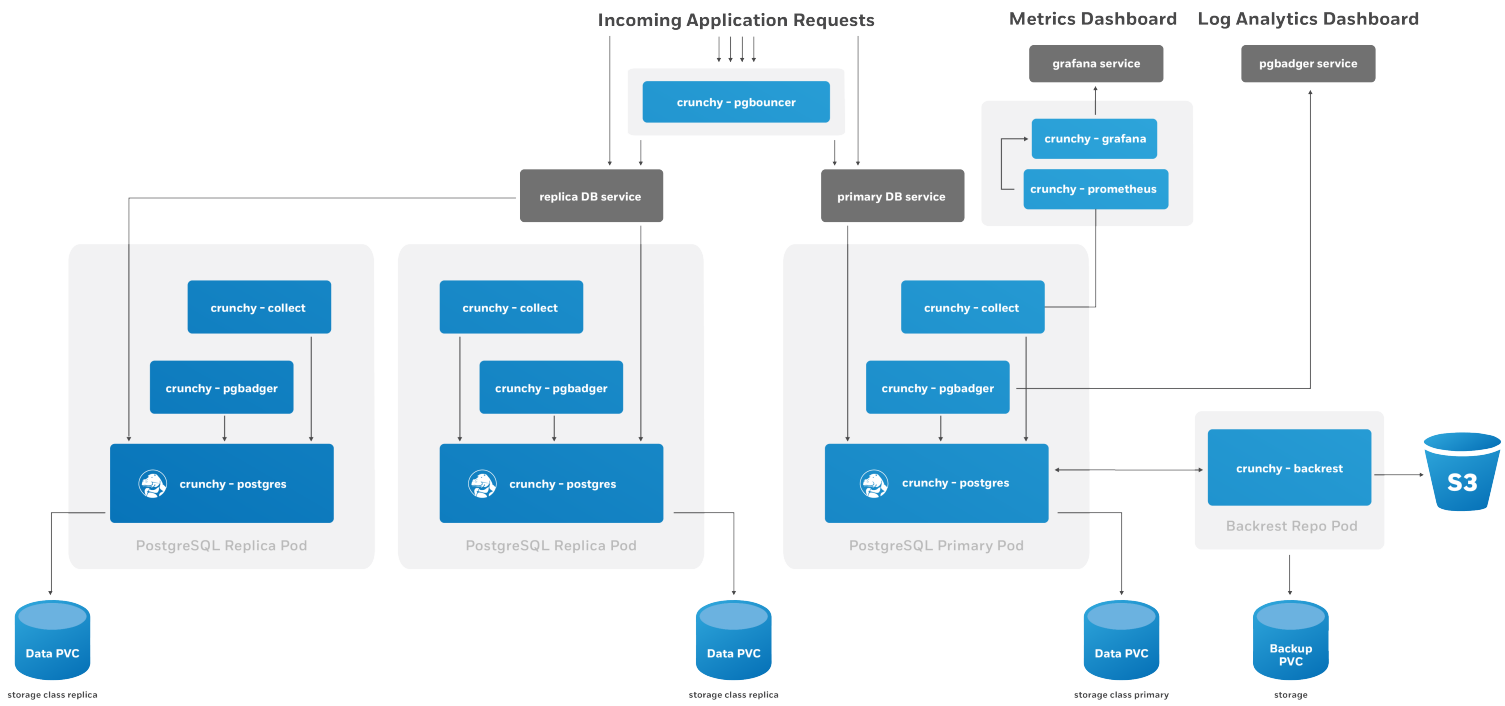

Figure 7: PostgreSQL HA Cluster

We can accomplish that with a single command:

```
pgo create cluster hacluster --replica-count=1 --metrics --pgbackrest-storage-type="posix,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 environment with a command similar to below:

```
pgo create cluster mixed --replica-count=1 \
  --storage-config=fast --memory=32Gi --cpu=8.0 \
  --replica-storage-config=standard
```

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
```

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 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?

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:

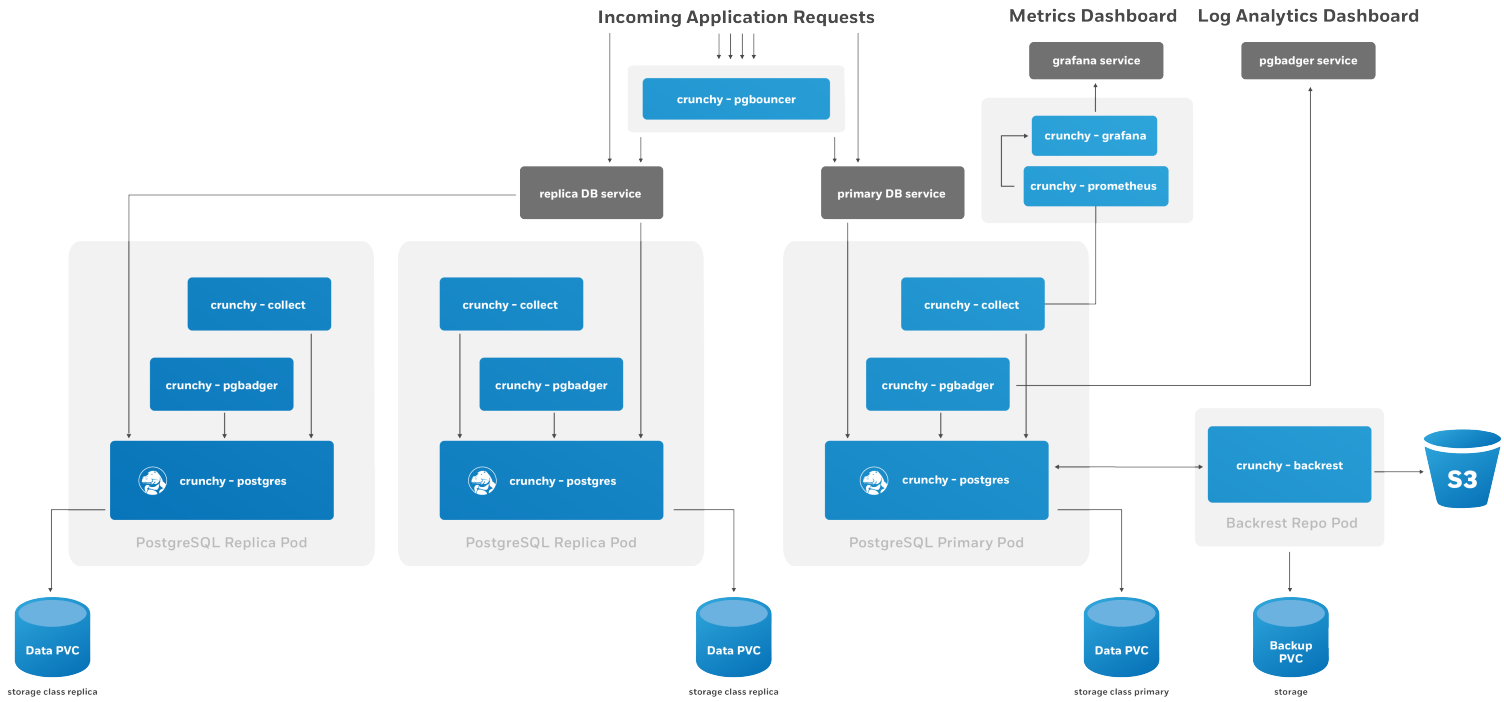

Figure 8: PostgreSQL HA Cluster

- 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 backups and perform full and point-in-time restores

The pgBackRest repository can be configured to use storage that resides within the Kubernetes cluster (the `posix` option), Amazon S3 or a storage system that uses the S3 protocol (the `s3` option), or both (`posix,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 `exporter`, uses the `crunchy-postgres-exporter` 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, an `exporter` 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

Custom Configuration({{< relref "/advanced/custom-configuration.md" >}})

PostgreSQL workloads often need tuning and additional configuration in production environments, and the PostgreSQL Operator allows for this via its ability to manage [custom PostgreSQL configuration]({{< relref "/advanced/custom-configuration.md" >}}).

The custom configuration can be edit from a ConfigMap that follows the pattern of `<clusterName>-pgha-config`, where `<clusterName>` would be `hippo` in `pgo create cluster hippo`. When the ConfigMap is edited, the changes are automatically pushed out to all of the PostgreSQL instances within a cluster.

For more information on how this works and what configuration settings are editable, please visit the "[Custom PostgreSQL configuration]({{< relref"/advanced/custom-configuration.md" >}})" section of the documentation.

Provisioning Using a Backup from an Another PostgreSQL Cluster

When provisioning a new PostgreSQL cluster, it is possible to bootstrap the cluster using an existing backup from either another PostgreSQL cluster that is currently running, or from a PostgreSQL cluster that no longer exists (specifically a cluster that was deleted using the `keep-backups` option, as discussed in section Deprovisioning below). This is specifically accomplished by performing a `pgbackrest restore` during cluster initialization in order to populate the initial `PGDATA` directory for the new cluster using the contents of a backup from another cluster.

To leverage this capability, the name of the cluster containing the backup that should be utilzed when restoring simply needs to be specified using the `restore-from` option when creating a new cluster:

```
pgo create cluster mycluster2 --restore-from=mycluster1
```

By default, pgBackRest will restore the latest backup available in the repository, and will replay all available WAL archives. However, additional pgBackRest options can be specified using the `restore-opts` option, which allows the restore command to be further tailored and customized. For instance, the following demonstrates how a point-in-time restore can be utilized when creating a new cluster:

```
pgo create cluster mycluster2 \
  --restore-from=mycluster1 \
  --restore-opts="--type=time --target='2020-07-02 20:19:36.13557+00'"
```

Additionally, if bootstrapping from a cluster the utilizes AWS S3 storage with pgBackRest (or a cluster that utilized AWS S3 storage in the case of a former cluster), you can also also specify `s3` as the repository type in order to restore from a backup stored in an S3 storage bucket:

```
pgo create cluster mycluster2 \
  --restore-from=mycluster1 \
  --restore-opts="--repo-type=s3"
```

When restoring from a cluster that is currently running, the new cluster will simply connect to the existing pgBackRest repository host for that cluster in order to perform the pgBackRest restore. If restoring from a former cluster that has since been deleted, a new pgBackRest repository host will be deployed for the sole purpose of bootstrapping the new cluster, and will then be destroyed once the restore is complete. Also, please note that it is only possible for one cluster to bootstrap from another cluster (whether running or not) at any given time.

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 backups 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:

- `posix`: 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
- `posix,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.

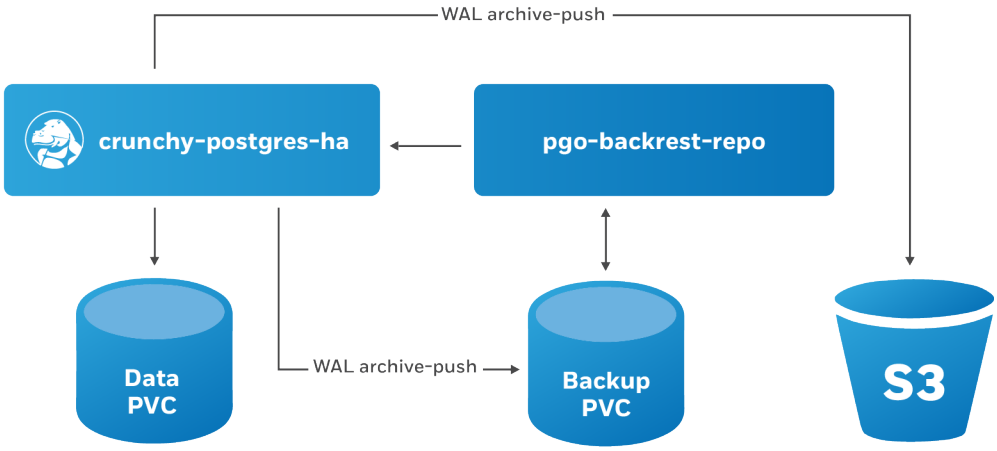

Figure 9: PostgreSQL Operator pgBackRest Integration

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. There are two types of ways to restore a cluster:

- Restore to a new cluster using the `--restore-from` flag in the `pgo create cluster`({{< relref "/pgo-client/reference/pgo_create_clus >}}) command.
- Restore in-place using the [`pgo restore`]({{< relref "/pgo-client/reference/pgo_restore.md" >}}) command. Note that this is **destructive**.

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.

The following explains how to perform restores based on the restoration method you chose.

Restore to a New Cluster

Restoring to a new PostgreSQL cluster allows one to take a backup and create a new PostgreSQL cluster that can run alongside an existing PostgreSQL cluster. There are several scenarios where using this technique is helpful:

- Creating a copy of a PostgreSQL cluster that can be used for other purposes. Another way of putting this is "creating a clone."
- Restore to a point-in-time and inspect the state of the data without affecting the current cluster

and more.

Restoring to a new cluster can be accomplished using the `pgo create cluster`({{< relref "/pgo-client/reference/pgo_create_cluster.md" >}}) command with several flags:

- `--restore-from`: specifies the name of a PostgreSQL cluster (either one that is active, or a former cluster whose pgBackRest repository still exists) to restore from.
- `--restore-opts`: used to specify additional options, similar to the ones that are passed into `pgbackrest restore`.

One can copy an entire PostgreSQL cluster into a new cluster with a command as simple as the one below:

```
pgo create cluster newcluster --restore-from oldcluster
```

To perform a point-in-time-recovery, you have to pass in the pgBackRest `--type` and `--target` options, where `--type` indicates the type of recovery to perform, and `--target` indicates the point in time to recover to:

```
pgo create cluster newcluster \
  --restore-from oldcluster \
  --restore-opts "--type=time --target='2019-12-31 11:59:59.999999+00'"
```

Note that when using this method, the PostgreSQL Operator can only restore one cluster from each pgBackRest repository at a time. Using the above example, one can only perform one restore from `oldcluster` at a given time.

When using the restore to a new cluster method, the PostgreSQL Operator takes the following actions:

- After running the normal cluster creation tasks, the PostgreSQL Operator creates a "bootstrap" job that performs a pgBackRest restore to the newly created PVC.
- The PostgreSQL Operator kicks off the new PostgreSQL cluster, which enters into recovery mode until it has recovered to a specified point-in-time or finishes replaying all available write-ahead logs.
- When this is done, the PostgreSQL cluster performs its regular operations when starting up.

Restore in-place

Restoring a PostgreSQL cluster in-place is a **destructive** action that will perform a recovery on your existing data directory. This is accomplished using the [`pgo restore`]({{< relref "/pgo-client/reference/pgo_restore.md" >}}) command.

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

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.

- 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 --enable-autofail
```

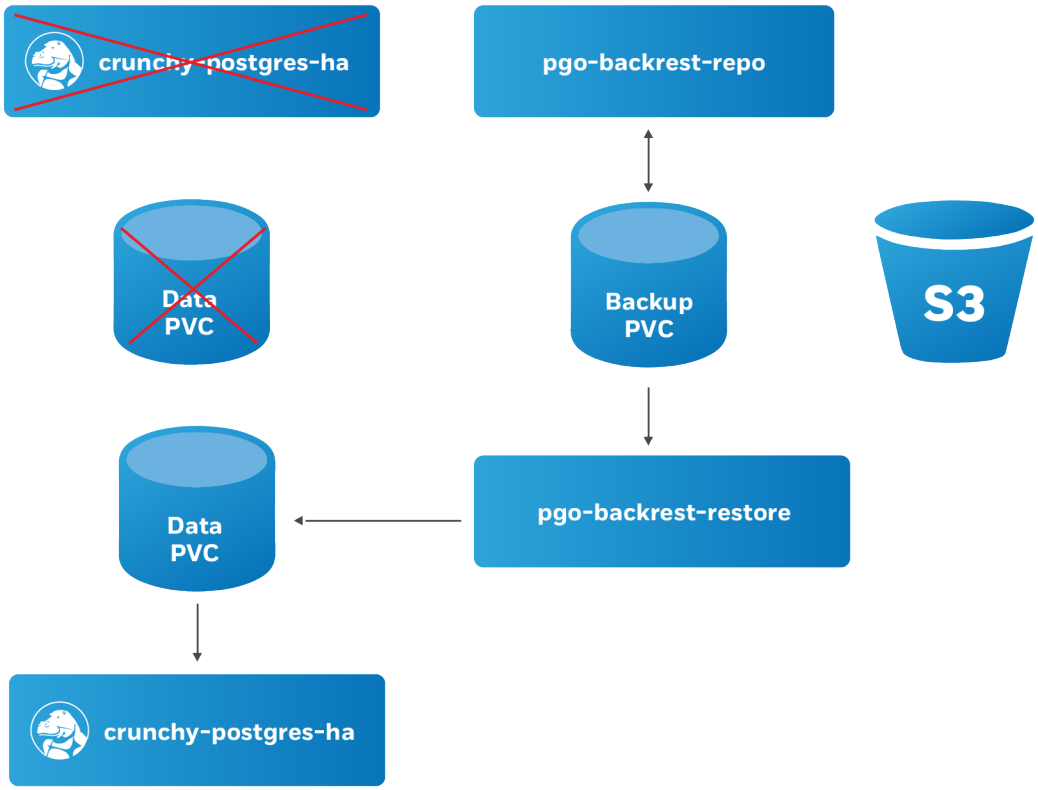

Figure 10: PostgreSQL Operator Restore Step 1

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.

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"
```

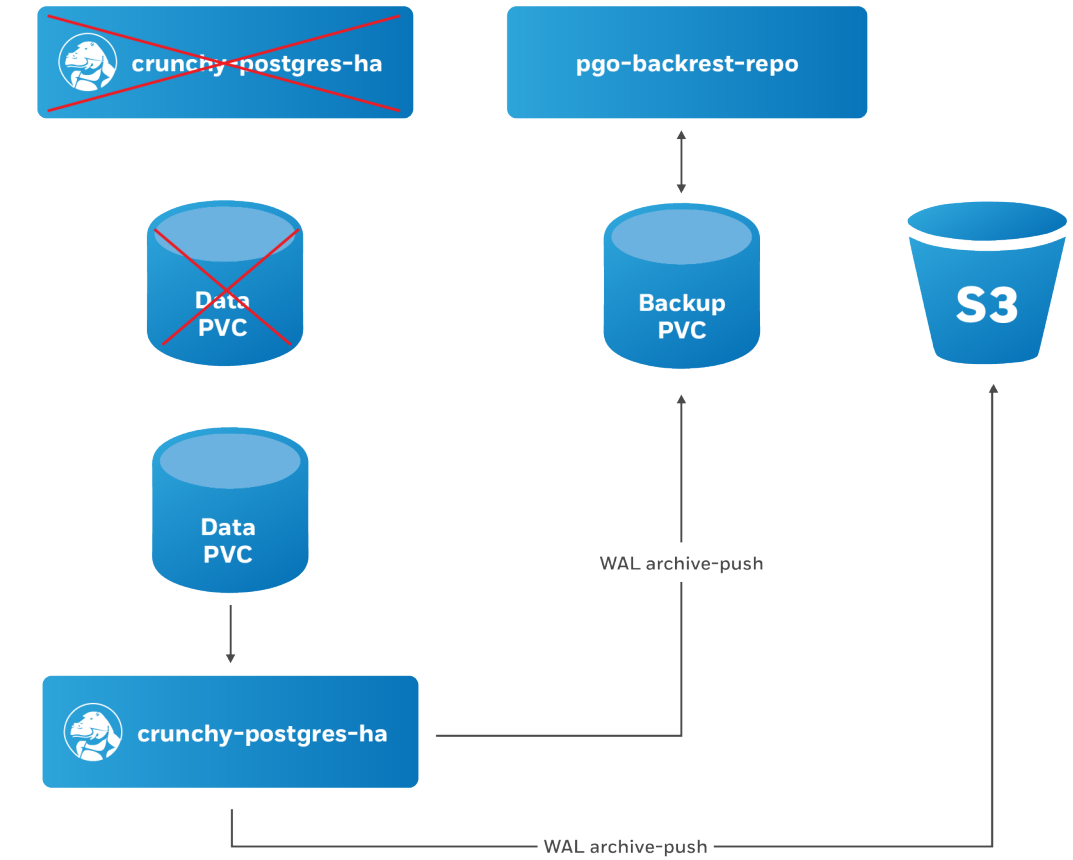

Figure 11: PostgreSQL Operator Restore Step 2

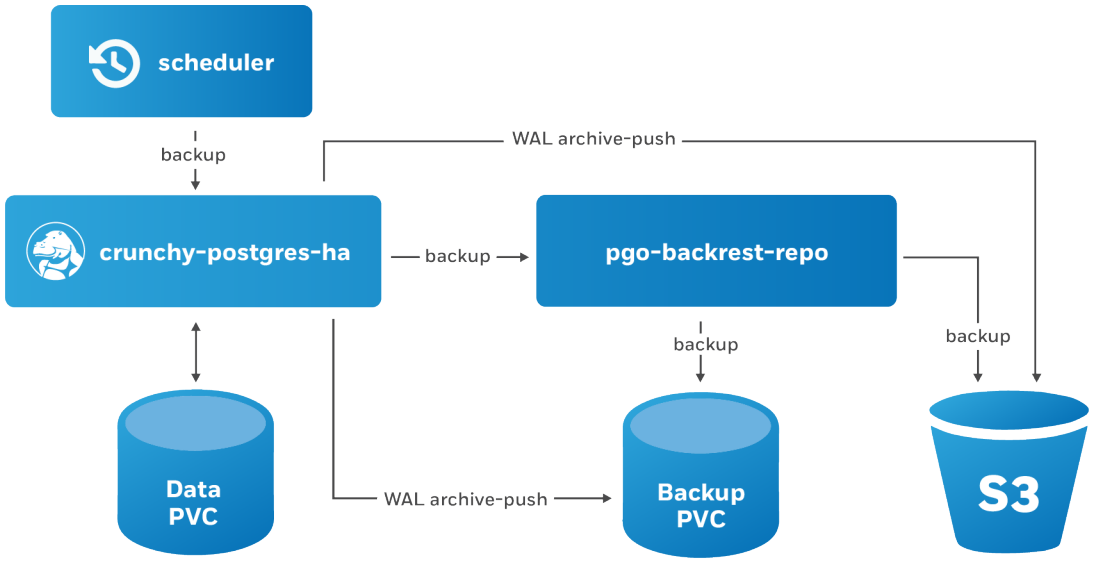

Figure 12: PostgreSQL Operator Schedule Backups

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
  BackrestS3URIStyle: host
  BackrestS3VerifyTLS: true
```

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
- `--pgbackrest-s3-uri-style` - specifies whether “host” or “path” style URIs should be utilized
- `--pgbackrest-s3-verify-tls` - set this value to “true” to enable TLS verification

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 `posix,s3`.

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

Deleting a Backup

If you delete a backup that is *not* set to expire, you may be unable to meet your retention requirements. If you are deleting backups to free space, it is recommended to delete your oldest backups first.

A backup can be deleted using the [`pgo delete backup`]({{< relref “pgo-client/reference/pgo_delete_backup.md” >}}) command. You must specify a specific backup to delete using the `--target` flag. You can get the backup names from the [`pgo show backup`]({{< relref “pgo-client/reference/pgo_show_backup.md” >}}) command.

For example, using a PostgreSQL cluster called `hippo`, pretend there is an example pgBackRest repository in the state shown after running the `pgo show backup hippo` command:

```
cluster: hippo
storage type: posix

stanza: db
    status: ok
```

```
    cipher: none

    db (current)
        wal archive min/max (12-1)

        full backup: 20201220-171801F
            timestamp start/stop: 2020-12-20 17:18:01 +0000 UTC / 2020-12-20 17:18:10 +0000 UTC
            wal start/stop: 000000010000000000000002 / 000000010000000000000002
            database size: 31.3MiB, backup size: 31.3MiB
            repository size: 3.8MiB, repository backup size: 3.8MiB
            backup reference list:

        incr backup: 20201220-171801F_20201220-171939I
            timestamp start/stop: 2020-12-20 17:19:39 +0000 UTC / 2020-12-20 17:19:41 +0000 UTC
            wal start/stop: 000000010000000000000005 / 000000010000000000000005
            database size: 31.3MiB, backup size: 216.3KiB
            repository size: 3.8MiB, repository backup size: 25.9KiB
            backup reference list: 20201220-171801F

        incr backup: 20201220-171801F_20201220-172046I
            timestamp start/stop: 2020-12-20 17:20:46 +0000 UTC / 2020-12-20 17:23:29 +0000 UTC
            wal start/stop: 00000001000000000000000A / 00000001000000000000000A
            database size: 65.9MiB, backup size: 37.5MiB
            repository size: 7.7MiB, repository backup size: 4.3MiB
            backup reference list: 20201220-171801F, 20201220-171801F_20201220-171939I

        full backup: 20201220-201305F
            timestamp start/stop: 2020-12-20 20:13:05 +0000 UTC / 2020-12-20 20:13:15 +0000 UTC
            wal start/stop: 00000001000000000000000F / 00000001000000000000000F
            database size: 65.9MiB, backup size: 65.9MiB
            repository size: 7.7MiB, repository backup size: 7.7MiB
            backup reference list:
```

The backup targets can be found after the backup type, e.g. `20201220-171801F` or `20201220-171801F_20201220-172046I`.

One can delete the oldest backup, in this case `20201220-171801F`, by running the following command:

```
pgo delete backup hippo --target=20201220-171801F
```

Verify the backup is deleted with `pgo show backup hippo`:

```
cluster: hippo
storage type: posix

stanza: db
    status: ok
    cipher: none

    db (current)
        wal archive min/max (12-1)

        full backup: 20201220-201305F
            timestamp start/stop: 2020-12-20 20:13:05 +0000 UTC / 2020-12-20 20:13:15 +0000 UTC
            wal start/stop: 00000001000000000000000F / 00000001000000000000000F
            database size: 65.9MiB, backup size: 65.9MiB
            repository size: 7.7MiB, repository backup size: 7.7MiB
            backup reference list:
```

(Note: this had the net effect of expiring all of the incremental backups associated with the full backup that as deleted. This is a feature of pgBackRest).

While having [high availability]({{< relref "architecture/high-availability/_index.md" >}}) and [disaster recovery]({{< relref "architecture/disaster-recovery.md" >}}) systems in place helps in the event of something going wrong with your PostgreSQL cluster, monitoring helps you anticipate problems before they happen. Additionally, monitoring can help you diagnose and resolve additional issues that may not result in downtime, but cause degraded performance.

There are many different ways to monitor systems within Kubernetes, including tools that come with Kubernetes itself. This is by no means to be a comprehensive on how to monitor everything in Kubernetes, but rather what the PostgreSQL Operator provides to give you an [out-of-the-box monitoring solution]({{< relref "installation/metrics/_index.md" >}}).

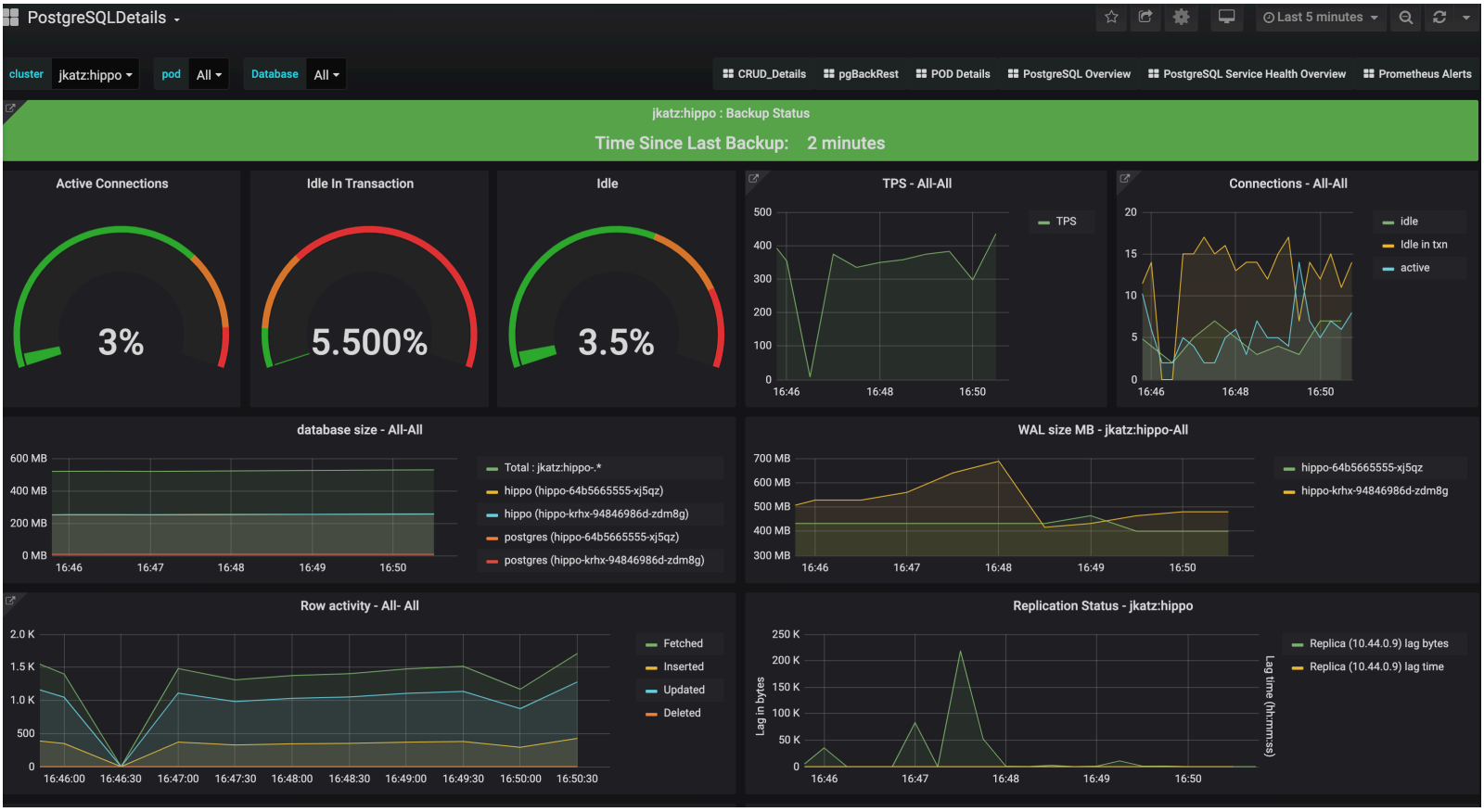

Figure 13: PostgreSQL Operator Monitoring

Getting Started

If you want to install the metrics stack, please visit the [installation]({{< relref "installation/metrics/_index.md" >}}) instructions for the PostgreSQL Operator Monitoring({{< relref "installation/metrics/_index.md" >}}) stack.

Once the metrics stack is set up, you will need to deploy your PostgreSQL clusters with monitoring enabled. To do so, you will need to use the `--metrics` flag as part of the `pgo create cluster`({{< relref "pgo-client/reference/pgo_create_cluster.md" >}}) command, for example:

```
pgo create cluster --metrics hippo
```

If you have already created a cluster and want to add metrics collection to it, you can use the `--enable-metrics` flag as part of the [`pgo update cluster`]({{< relref "pgo-client/reference/pgo_update_cluster.md" >}}) command, for example:

```
pgo update cluster --enable-metrics hippo
```

Components

The PostgreSQL Operator Monitoring({{< relref "installation/metrics/_index.md" >}}) stack is made up of several open source components:

- pgMonitor, which provides the core of the monitoring infrastructure including the following components:
- postgres_exporter, which provides queries used to collect metrics information about a PostgreSQL instance.
- Prometheus, a time-series database that scrapes and stores the collected metrics so they can be consumed by other services.
- Grafana, a visualization tool that provides charting and other capabilities for viewing the collected monitoring data.
- Alertmanager, a tool that can send alerts when metrics hit a certain threshold that require someone to intervene.
- pgnodemx, a PostgreSQL extension that is able to pull container-specific metrics (e.g. CPU utilization, memory consumption) from the container itself via SQL queries.

Visualizations

Below is a brief description of all the visualizations provided by the PostgreSQL Operator Monitoring({{< relref "installation/metrics/_index.md" >}}) stack. Some of the descriptions may include some directional guidance on how to interpret the charts, though this is only to provide a starting point: actual causes and effects of issues can vary between systems.

Many of the visualizations can be broken down based on the following groupings:

- Cluster: which PostgreSQL cluster should be viewed
- Pod: the specific Pod or PostgreSQL instance

Overview

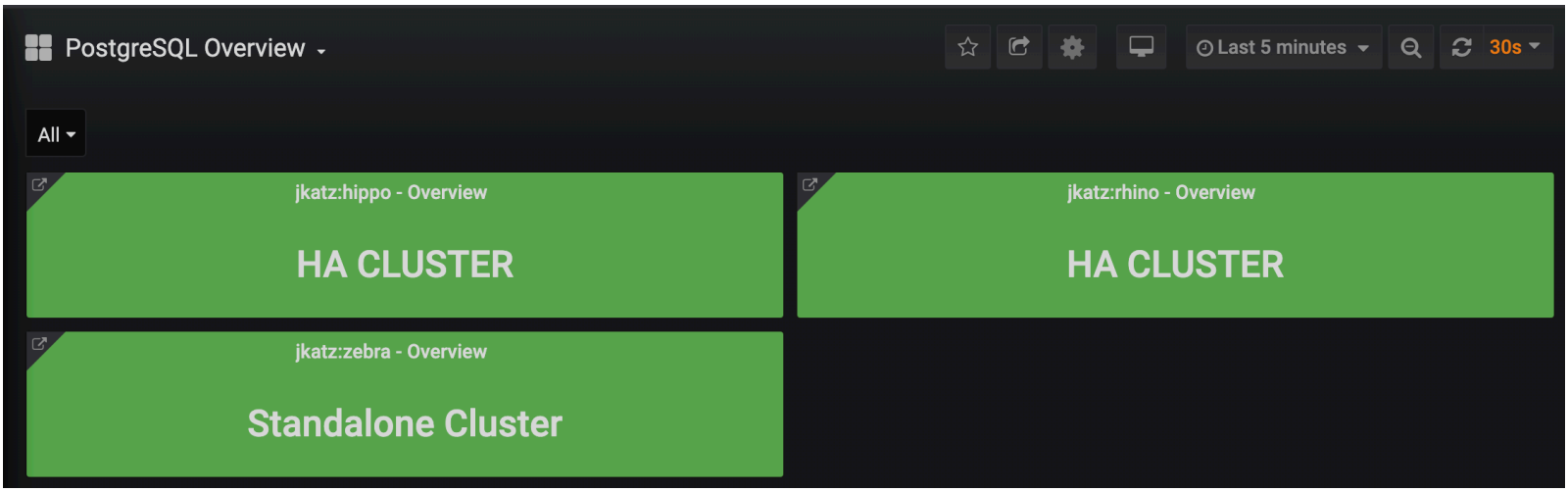

Figure 14: PostgreSQL Operator Monitoring - Overview

The overview provides an overview of all of the PostgreSQL clusters that are being monitoring by the PostgreSQL Operator Monitoring stack. This includes the following information:

- The name of the PostgreSQL cluster and the namespace that it is in
- The type of PostgreSQL cluster (HA [high availability] or standalone)
- The status of the cluster, as indicate by color. Green indicates the cluster is available, red indicates that it is not.

Each entry is clickable to provide additional cluster details.

PostgreSQL Details

The PostgreSQL Details view provides more information about a specific PostgreSQL cluster that is being managed and monitored by the PostgreSQL Operator. These include many key PostgreSQL-specific metrics that help make decisions around managing a PostgreSQL cluster. These include:

- Backup Status: The last time a backup was taken of the cluster. Green is good. Orange means that a backup has not been taken in more than a day and may warrant investigation.
- Active Connections: How many clients are connected to the database. Too many clients connected could impact performance and, for values approaching 100%, can lead to clients being unable to connect.
- Idle in Transaction: How many clients have a connection state of "idle in transaction". Too many clients in this state can cause performance issues and, in certain cases, maintenance issues.
- Idle: How many clients are connected but are in an "idle" state.
- TPS: The number of "transactions per second" that are occurring. Usually needs to be combined with another metric to help with analysis. "Higher is better" when performing benchmarking.
- Connections: An aggregated view of active, idle, and idle in transaction connections.
- Database Size: How large databases are within a PostgreSQL cluster. Typically combined with another metric for analysis. Helps keep track of overall disk usage and if any triage steps need to occur around PVC size.
- WAL Size: How much space write-ahead logs (WAL) are taking up on disk. This can contribute to extra space being used on your data disk, or can give you an indication of how much space is being utilized on a separate WAL PVC. If you are using replication slots, this can help indicate if a slot is not being acknowledged if the numbers are much larger than the `max_wal_size` setting (the PostgreSQL Operator does not use slots by default).
- Row Activity: The number of rows that are selected, inserted, updated, and deleted. This can help you determine what percentage of your workload is read vs. write, and help make database tuning decisions based on that, in conjunction with other metrics.
- Replication Status: Provides guidance information on how much replication lag there is between primary and replica PostgreSQL instances, both in bytes and time. This can provide an indication of how much data could be lost in the event of a failover.

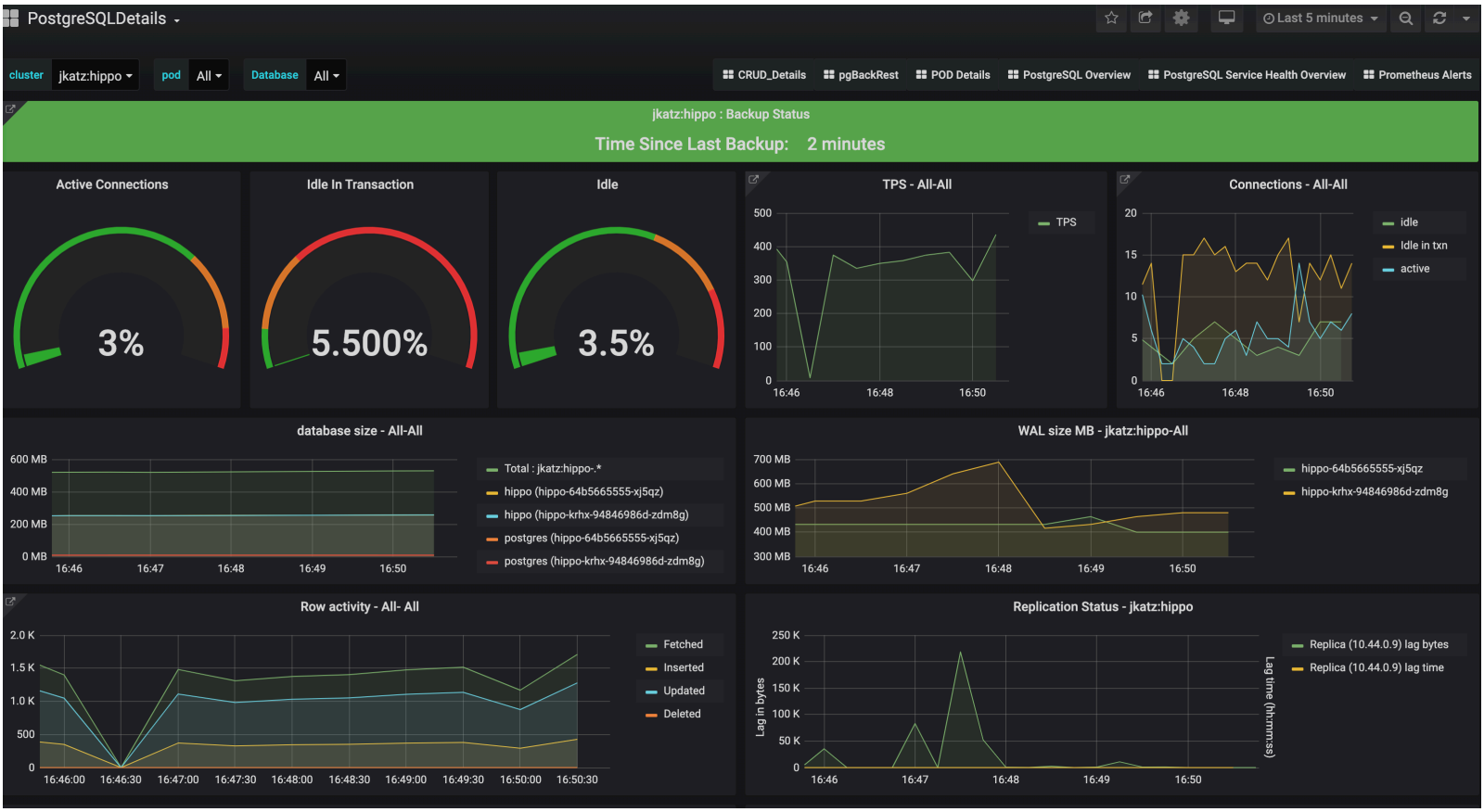

Figure 15: PostgreSQL Operator Monitoring - Cluster Cluster Details

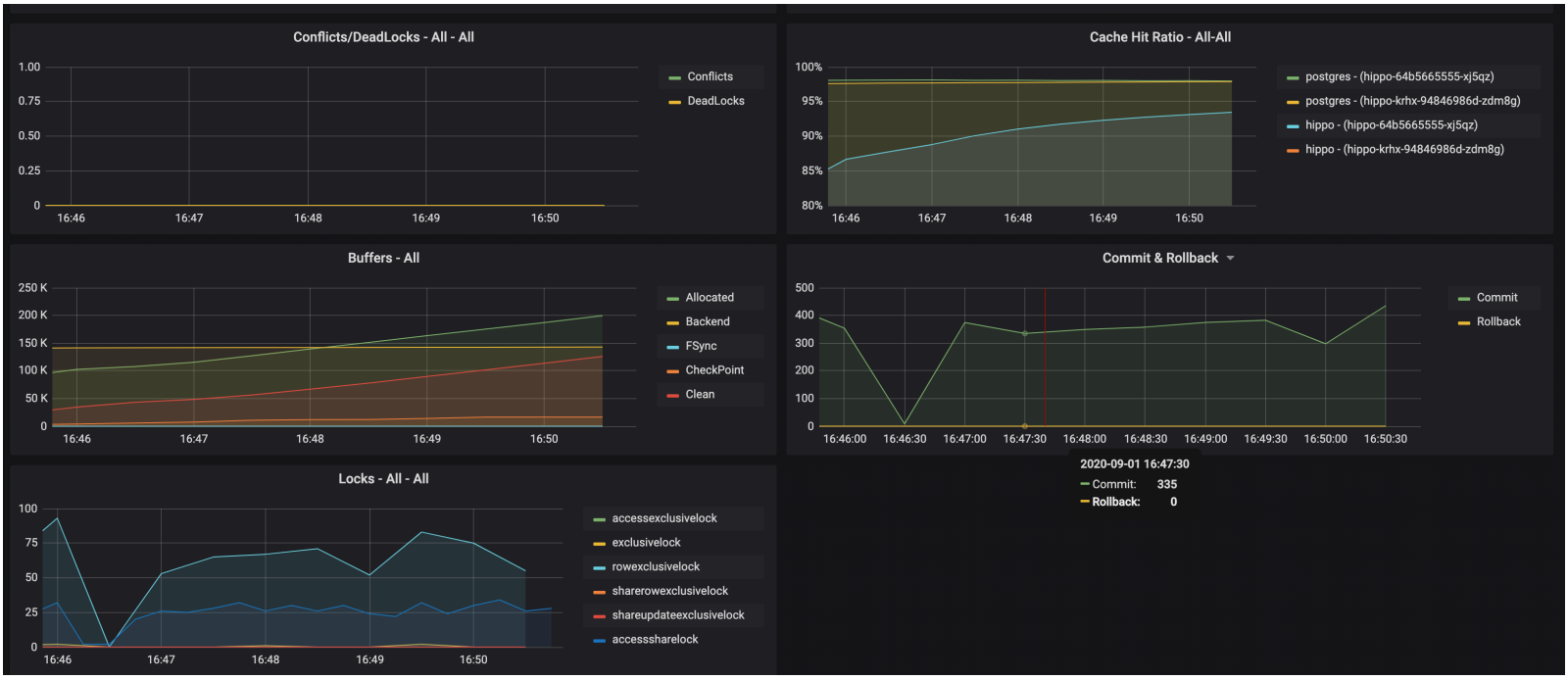

Figure 16: PostgreSQL Operator Monitoring - Cluster Cluster Details 2

- Conflicts / Deadlocks: These occur when PostgreSQL is unable to complete operations, which can result in transaction loss. The goal is for these numbers to be `0`. If these are occurring, check your data access and writing patterns.
- Cache Hit Ratio: A measure of how much of the "working data", e.g. data that is being accessed and manipulated, resides in memory. This is used to understand how much PostgreSQL is having to utilize the disk. The target number of this should be as high as possible. How to achieve this is the subject of books, but certain takes efforts on your applications use PostgreSQL.
- Buffers: The buffer usage of various parts of the PostgreSQL system. This can be used to help understand the overall throughput between various parts of the system.
- Commit & Rollback: How many transactions are committed and rolled back.
- Locks: The number of locks that are present on a given system.

Pod Details

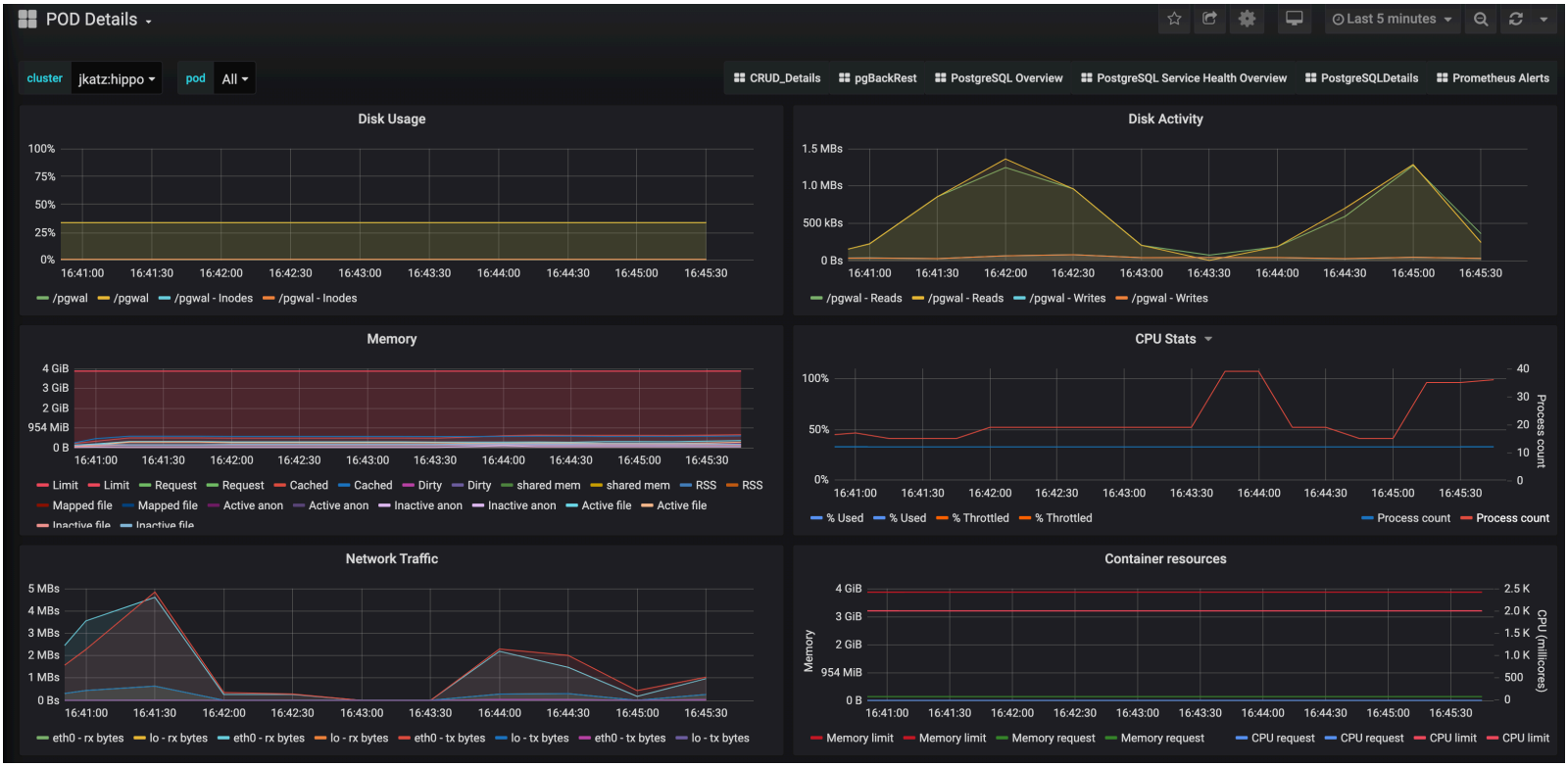

Figure 17: PostgreSQL Operator Monitoring - Pod Details

Pod details provide information about a given Pod or Pods that are being used by a PostgreSQL cluster. These are similar to "operating system" or "node" metrics, with the differences that these are looking at resource utilization by a container, not the entire node.

It may be helpful to view these metrics on a "pod" basis, by using the Pod filter at the top of the dashboard.

- Disk Usage: How much space is being consumed by a volume.
- Disk Activity: How many reads and writes are occurring on a volume.
- Memory: Various information about memory utilization, including the request and limit as well as actually utilization.
- CPU: The amount of CPU being utilized by a Pod
- Network Traffic: The amount of networking traffic passing through each network device.
- Container ResourceS: The CPU and memory limits and requests.

PostgreSQL Service Health Overview

The Service Health Overview provides information about the Kubernetes Services that sit in front of the PostgreSQL Pods. This provides information about the status of the network.

- Saturation: How much of the available network to the Service is being consumed. High saturation may cause degraded performance to clients or create an inability to connect to the PostgreSQL cluster.
- Traffic: Displays the number of transactions per minute that the Service is handling.
- Errors: Displays the total number of errors occurring at a particular Service.
- Latency: What the overall network latency is when interfacing with the Service.

Figure 18: PostgreSQL Operator Monitoring - Service Health Overview

Alerts

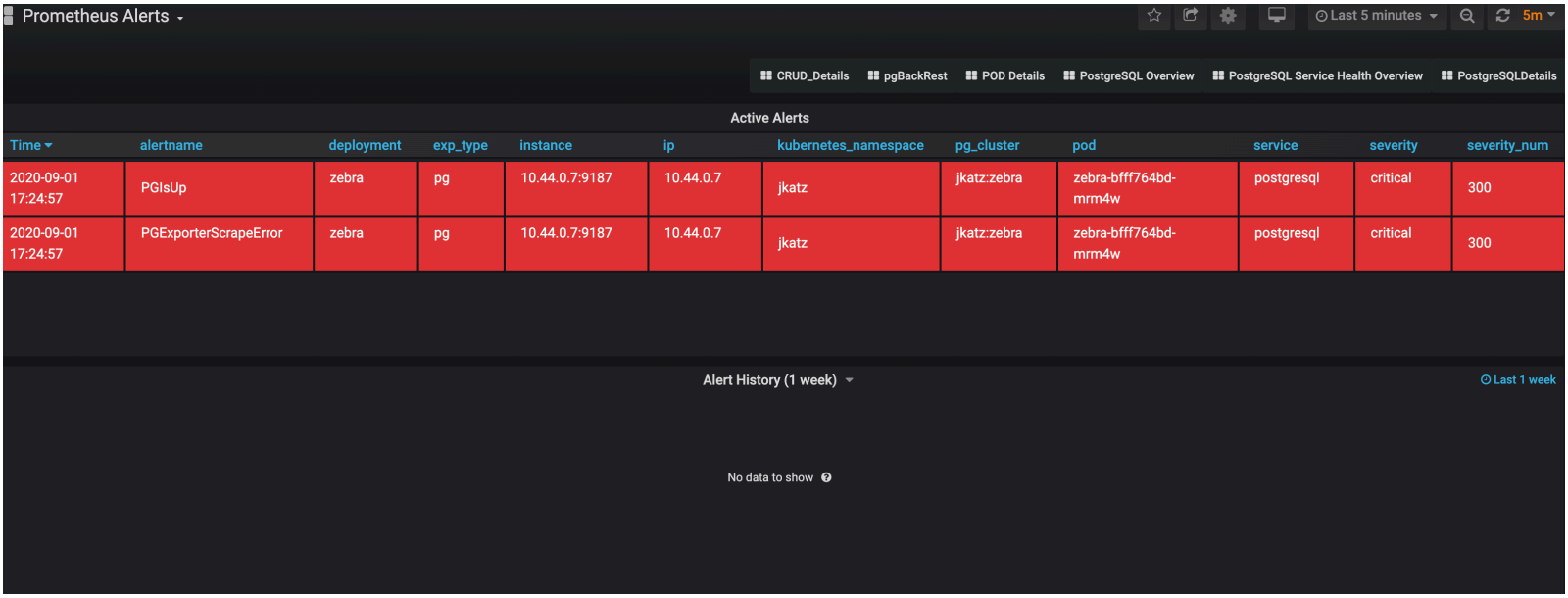

Figure 19: PostgreSQL Operator Monitoring - Alerts

Alerting lets one view and receive alerts about actions that require intervention, for example, a HA cluster that cannot self-heal. The alerting system is powered by Alertmanager.

The alerts that come installed by default include:

- `PGExporterScrapeError`: The Crunchy PostgreSQL Exporter is having issues scraping statistics used as part of the monitoring stack.
- `PGIsUp`: A PostgreSQL instance is down.
- `PGIdleTxn`: There are too many connections that are in the "idle in transaction" state.
- `PGQueryTime`: A single PostgreSQL query is taking too long to run. Issues a warning at 12 hours and goes critical after 24.
- `PGConnPerc`: Indicates that there are too many connection slots being used. Issues a warning at 75% and goes critical above 90%.
- `PGDiskSize`: Indicates that a PostgreSQL database is too large and could be in danger of running out of disk space. Issues a warning at 75% and goes critical at 90%.
- `PGReplicationByteLag`: Indicates that a replica is too far behind a primary instance, which coul risk data loss in a failover scenario. Issues a warning at 50MB an goes critical at 100MB.

- `PGReplicationSlotsInactive`: Indicates that a replication slot is inactive. Not attending to this can lead to out-of-disk errors.
- `PGXIDWraparound`: Indicates that a PostgreSQL instance is nearing transaction ID wraparound. Issues a warning at 50% and goes critical at 75%. It's important that you vacuum your database to prevent this.
- `PGEmergencyVacuum`: Indicates that autovacuum is not running or cannot keep up with ongoing changes, i.e. it's past its "freeze" age. Issues a warning at 110% and goes critical at 125%.
- `PGArchiveCommandStatus`: Indicates that the archive command, which is used to ship WAL archives to pgBackRest, is failing.
- `PGSequenceExhaustion`: Indicates that a sequence is over 75% used.
- `PGSettingsPendingRestart`: Indicates that there are settings changed on a PostgreSQL instance that requires a restart.

Optional alerts that can be enabled:

- `PGMinimumVersion`: Indicates if PostgreSQL is below a desired version.
- `PGRecoveryStatusSwitch_Replica`: Indicates that a replica has been promoted to a primary.
- `PGConnectionAbsent_Prod`: Indicates that metrics collection is absent from a PostgresQL instance.
- `PGSettingsChecksum`: Indicates that PostgreSQL settings have changed from a previous state.
- `PGDataChecksum`: Indicates that there are data checksum failures on a PostgreSQL instance. This could be a sign of data corruption.

You can modify these alerts as you see fit, and add your own alerts as well! Please see the [installation instructions]({{< relref "installation/metrics/_index.md" >}}) for general setup of the PostgreSQL Operator Monitoring stack.

Kubernetes Namespaces and the PostgreSQL Operator

The PostgreSQL Operator leverages Kubernetes Namespaces to react to actions taken within a Namespace to keep its PostgreSQL clusters deployed as requested. Early on, the PostgreSQL Operator was scoped to a single namespace and would only watch PostgreSQL clusters in that Namspace, but since version 4.0, it has been expanded to be able to manage PostgreSQL clusters across multiple namespaces.

The following provides more information about how the PostgreSQL Operator works with namespaces, and presents several deployment patterns that can be used to deploy the PostgreSQL Operator.

Namespace Operating Modes

The PostgreSQL Operator can be run with various Namespace Operating Modes, with each mode determining whether or not certain namespace capabilities are enabled for the PostgreSQL Operator installation. When the PostgreSQL Operator is run, the Kubernetes environment is inspected to determine what cluster roles are currently assigned to the `pgo-operator` `ServiceAccount` (i.e. the `ServiceAccount` running the Pod the PostgreSQL Operator is deployed within). Based on the `ClusterRoles` identified, one of the namespace operating modes described below will be enabled for the [PostgreSQL Operator Installation]({{< relref "installation" >}}). Please consult the installation section for more information on the available settings.

`dynamic`

Enables full dynamic namespace capabilities, in which the Operator can create, delete and update any namespaces within a Kubernetes cluster. With `dynamic` mode enabled, the PostgreSQL Operator can respond to namespace events in a Kubernetes cluster, such as when a namespace is created, and take an appropriate action, such as adding the PostgreSQL Operator controllers for the newly created namespace.

The following defines the namespace permissions required for the `dynamic` mode to be enabled:

```
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: pgo-cluster-role
rules:
  - apiGroups:
      - ''
    resources:
      - namespaces
    verbs:
      - get
      - list
      - watch
      - create
```

```
      - update
      - delete
```

`readonly`

In `readonly` mode, the PostgreSQL Operator is still able to listen to namespace events within a Kubernetes cluster, but it can no longer modify (create, update, delete) namespaces. For example, if a Kubernetes administrator creates a namespace, the PostgreSQL Operator can respond and create controllers for that namespace.

The following defines the namespace permissions required for the `readonly` mode to be enabled:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: pgo-cluster-role
rules:
  - apiGroups:
      - ''
    resources:
      - namespaces
    verbs:
      - get
      - list
      - watch
```

`disabled`

`disabled` mode disables namespace capabilities namespace capabilities within the PostgreSQL Operator altogether. While in this mode the PostgreSQL Operator will simply attempt to work with the target namespaces specified during installation. If no target namespaces are specified, then the Operator will be configured to work within the namespace in which it is deployed. Since the Operator is unable to dynamically respond to namespace events in the cluster, in the event that target namespaces are deleted or new target namespaces need to be added, the PostgreSQL Operator will need to be re-deployed.

Please note that it is important to redeploy the PostgreSQL Operator following the deletion of a target namespace to ensure it no longer attempts to listen for events in that namespace.

The `disabled` mode is enabled the when the PostgreSQL Operator has not been assigned namespace permissions.

RBAC Reconciliation

By default, the PostgreSQL Operator will attempt to reconcile RBAC resources (ServiceAccounts, Roles and RoleBindings) within each namespace configured for the PostgreSQL Operator installation. This allows the PostgreSQL Operator to create, update and delete the various RBAC resources it requires in order to properly create and manage PostgreSQL clusters within each targeted namespace (this includes self-healing RBAC resources as needed if removed and/or misconfigured).

In order for RBAC reconciliation to function properly, the PostgreSQL Operator ServiceAccount must be assigned a certain set of permissions. While the PostgreSQL Operator is not concerned with exactly how it has been assigned the permissions required to reconcile RBAC in each target namespace, the various [installation methods]({{< relref “installation” >}}) supported by the PostgreSQL Operator install a recommended set permissions based on the specific Namespace Operating Mode enabled (see section Namespace Operating Modes({{< relref “#namespace-operating-modes” >}}) above for more information regarding the various Namespace Operating Modes available).

The following section defines the recommended set of permissions that should be assigned to the PostgreSQL Operator ServiceAccount in order to ensure proper RBAC reconciliation based on the specific Namespace Operating Mode enabled. Please note that each PostgreSQL Operator installation method handles the initial configuration and setup of the permissions shown below based on the Namespace Operating Mode configured during installation.

`dynamic` Namespace Operating Mode

When using the `dynamic` Namespace Operating Mode, it is recommended that the PostgreSQL Operator ServiceAccount be granted permissions to manage RBAC inside any namespace in the Kubernetes cluster via a ClusterRole. This allows for a fully-hands off approach to managing RBAC within each targeted namespace space. In other words, as namespaces are added and removed post-installation of the PostgreSQL Operator (e.g. using `pgo create namespace` or `pgo delete namespace`), the Operator is able to automatically reconcile RBAC in those namespaces without the need for any external administrative action and/or resource creation.

The following defines ClusterRole permissions that are assigned to the PostgreSQL Operator ServiceAccount via the various Operator installation methods when the `dynamic` Namespace Operating Mode is configured:

```
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: pgo-cluster-role
rules:
  - apiGroups:
      - ''
    resources:
      - serviceaccounts
    verbs:
      - get
      - create
      - update
      - delete
  - apiGroups:
      - rbac.authorization.k8s.io
    resources:
      - roles
      - rolebindings
    verbs:
      - get
      - create
      - update
      - delete
  - apiGroups:
      - ''
    resources:
      - configmaps
      - endpoints
      - pods
      - pods/exec
      - secrets
      - services
      - persistentvolumeclaims
    verbs:
      - get
      - list
      - watch
      - create
      - patch
      - update
      - delete
      - deletecollection
  - apiGroups:
    - ''
    resources:
      - pods/log
    verbs:
      - get
      - list
      - watch
  - apiGroups:
      - apps
    resources:
      - deployments
      - replicasets
    verbs:
      - get
      - list
      - watch
      - create
```

```
      - patch
      - update
      - delete
      - deletecollection
  - apiGroups:
      - batch
    resources:
      - jobs
    verbs:
      - get
      - list
      - watch
      - create
      - patch
      - update
      - delete
      - deletecollection
  - apiGroups:
      - crunchydata.com
    resources:
      - pgclusters
      - pgpolicies
      - pgreplicas
      - pgtasks
    verbs:
      - get
      - list
      - watch
      - create
      - patch
      - update
      - delete
      - deletecollection
```

`readonly` & `disabled` Namespace Operating Modes

When using the `readonly` or `disabled` Namespace Operating Modes, it is recommended that the PostgreSQL Operator ServiceAccount be granted permissions to manage RBAC inside of any configured namespaces using local Roles within each targeted namespace. This means that as new namespaces are added and removed post-installation of the PostgreSQL Operator, an administrator must manually assign the PostgreSQL Operator ServiceAccount the permissions it requires within each target namespace in order to successfully reconcile RBAC within those namespaces.

The following defines the permissions that are assigned to the PostgreSQL Operator ServiceAccount in each configured namespace via the various Operator installation methods when the `readonly` or `disabled` Namespace Operating Modes are configured:

```
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: pgo-local-ns
  namespace: targetnamespace
rules:
  - apiGroups:
      - ''
    resources:
      - serviceaccounts
    verbs:
      - get
      - create
      - update
      - delete
  - apiGroups:
      - rbac.authorization.k8s.io
    resources:
      - roles
```

```
      - rolebindings
    verbs:
      - get
      - create
      - update
      - delete
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pgo-target-role
  namespace: targetnamespace
rules:
- apiGroups:
    - ''
  resources:
    - configmaps
    - endpoints
    - pods
    - pods/exec
    - pods/log
    - replicasets
    - secrets
    - services
    - persistentvolumeclaims
  verbs:
    - get
    - list
    - watch
    - create
    - patch
    - update
    - delete
    - deletecollection
- apiGroups:
    - apps
  resources:
    - deployments
  verbs:
    - get
    - list
    - watch
    - create
    - patch
    - update
    - delete
    - deletecollection
- apiGroups:
    - batch
  resources:
    - jobs
  verbs:
    - get
    - list
    - watch
    - create
    - patch
    - update
    - delete
    - deletecollection
- apiGroups:
    - crunchydata.com
  resources:
    - pgclusters
    - pgpolicies
```

```
    - pgtasks
    - pgreplicas
  verbs:
    - get
    - list
    - watch
    - create
    - patch
    - update
    - delete
    - deletecollection
```

Disabling RBAC Reconciliation

In the event that the reconciliation behavior discussed above is not desired, it can be fully disabled by setting `DisableReconcileRBAC` to `true` in the `pgo.yaml` configuration file. When reconciliation is disabled using this setting, the PostgreSQL Operator will not attempt to reconcile RBAC in any configured namespace. As a result, any RBAC required by the PostreSQL Operator a targeted namespace must be manually created by an administrator.

Please see the the [`pgo.yaml` configuration guide]({{< relref "configuration/pgo-yaml-configuration.md" >}}), as well as the documentation for the various [installation methods]({{< relref "installation" >}}) supported by the PostgreSQL Operator, for guidance on how to properly configure this setting and therefore disable RBAC reconciliation.

Namespace Deployment Patterns

There are several different ways the PostgreSQL Operator can be deployed in Kubernetes clusters with respect to Namespaces.

One Namespace: PostgreSQL Operator + PostgreSQL Clusters

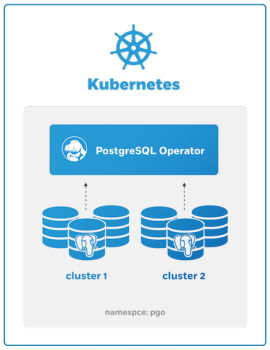

Figure 20: PostgreSQL Operator Own Namespace Deployment

This patterns is great for testing out the PostgreSQL Operator in development environments, and can also be used to keep your entire PostgreSQL workload within a single Kubernetes Namespace.

This can be set up with the `disabled` Namespace mode.

Single Tenant: PostgreSQL Operator Separate from PostgreSQL Clusters

The PostgreSQL Operator can be deployed into its own namespace and manage PostgreSQL clusters in a separate namespace.

This can be set up with either the `readonly` or `dynamic` Namespace modes.

Multi Tenant: PostgreSQL Operator Managing PostgreSQL Clusters in Multiple Namespaces

The PostgreSQL Operator can manage PostgreSQL clusters across multiple namespaces which allows for multi-tenancy.

This can be set up with either the `readonly` or `dynamic` Namespace modes.

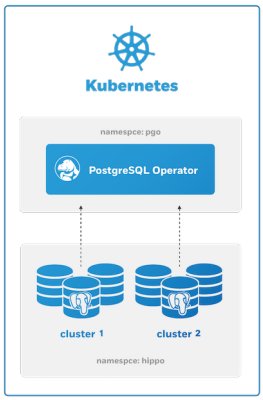

Figure 21: PostgreSQL Operator Single Namespace Deployment

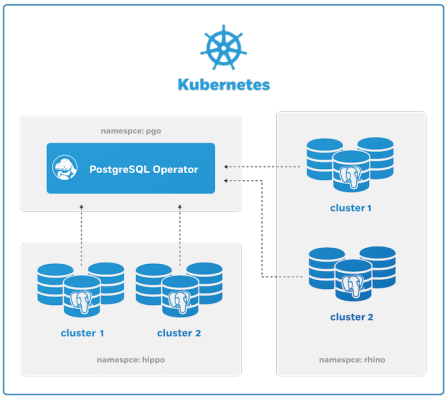

Figure 22: PostgreSQL Operator Multi Namespace Deployment

[pgo client]({{< relref "/pgo-client/_index.md" >}}) and Namespaces

The [pgo client]({{< relref "/pgo-client/_index.md" >}}) needs to be aware of the Kubernetes Namespaces it is issuing commands to. This can be accomplish with the `-n` flag that is available on most PostgreSQL Operator commands. For example, to create a PostgreSQL cluster called `hippo` in the `pgo` namespace, you would execute the following command:

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

For convenience, you can set the `PGO_NAMESPACE` environmental variable to automatically use the desired namespace with the commands.

For example, to create a cluster named `hippo` in the `pgo` namespace, you could do the following

```
# this export only needs to be run once per session
export PGO_NAMESPACE=pgo

pgo create cluster hippo
```

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 svc/postgres-operator 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/pkg/events/eventt

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, Tools, 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. The container supports PostgreSQL tools by running in different modes, more information on running modes can be found in the Crunchy Container documentation. The PostgreSQL operator uses the following running modes:
  - **pgdump** (MODE: pgdump) running in pgdump mode, the image executes either a pg_dump or pg_dumpall database backup against another PostgreSQL database.
  - **pgrestore** (MODE: pgrestore) running in pgrestore mode, the 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.
  - **sqlrunner** (MODE: sqlrunner) running in sqlrunner mode, the image will use `psql` to issue specified queries, defined in SQL files, to a PostgreSQL container database.
- **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-postgres-ha). pgBackRest is a high performance backup and restore utility for PostgreSQL. The crunchy-postgres-ha container executes the pgBackRest utility, allowing FULL and DELTA restore capability.

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 pgBackRest backups within a single namespace.

Metrics and Monitoring

- **Metrics Collection** (crunchy-postgres-exporter). The crunchy-postgres-exporter 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** (grafana). Hosts an open source web-based graphing dashboard called Grafana. Provides visual dashboards for monitoring PostgreSQL clusters, specifically using Crunchy PostgreSQL Exporter data stored within Prometheus.
- **Prometheus** (prometheus). Prometheus is a multi-dimensional time series data model with an elastic query language. It is used in collaboration with the Crunchy PostgreSQL Exporter and Grafana to provide and store metrics.
- **Alertmanager** (alertmanager). Handles alerts sent by Prometheus by deduplicating, grouping, and routing them to reciever integrations.

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 or restored cluster
- Deprovisioining: Tablespaces are deleted when a PostgreSQL instance or cluster is deleted

Adding Tablespaces to a New Cluster

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

```
pgo create cluster hacluster \
    --tablespace=name=tablespace1:storageconfig=storageconfigname \
    --tablespace=name=tablespace2:storageconfig=storageconfigname
```

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 \
    --tablespace=name=faststorage1:storageconfig=nfsstorage \
    --tablespace=name=faststorage2:storageconfig=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;
```

Adding Tablespaces to Existing Clusters

You can also add a tablespace to an existing PostgreSQL cluster with the `pgo update cluster` command. Adding a tablespace to a cluster uses a similar syntax to creating a cluster with tablespaces, for example:

```
pgo update cluster hacluster \
    --tablespace=name=tablespace3:storageconfig=storageconfigname
```

NOTE: This operation can cause downtime. In order to add a tablespace to a PostgreSQL cluster, persistent volume claims (PVCs) need to be created and mounted to each PostgreSQL instance in the cluster. The act of mounting a new PVC to a Kubernetes Deployment causes the Pods in the deployment to restart.

When the operation completes, the tablespace will be set up and accessible to use within the PostgreSQL cluster.

Removing Tablespaces

Removing a tablespace is a nontrivial operation. PostgreSQL does not provide a `DROP TABLESPACE .. CASCADE` command that would drop any associated objects with a tablespace. Additionally, the PostgreSQL documentation covering the `DROP TABLESPACE` command goes on to note:

> A tablespace can only be dropped by its owner or a superuser. The tablespace must be empty of all database objects before it can be dropped. It is possible that objects in other databases might still reside in the tablespace even if no objects in the current database are using the tablespace. Also, if the tablespace is listed in the temp_tablespaces setting of any active session, the DROP might fail due to temporary files residing in the tablespace.

Because of this, and to avoid a situation where a PostgreSQL cluster is left in an inconsistent state due to trying to remove a tablespace, the PostgreSQL Operator does not provide any means to remove tablespaces automatically. If you do need to remove a tablespace from a PostgreSQL deployment, we recommend following this procedure:

1. As a database administrator:
2. Log into the primary instance of your cluster.
3. Drop any objects that reside within the tablespace you wish to delete. These can be tables, indexes, and even databases themselves
4. When you believe you have deleted all objects that depend on the tablespace you wish to remove, you can delete this tablespace from the PostgreSQL cluster using the `DROP TABLESPACE` command.
5. As a Kubernetes user who can modify Deployments and edit an entry in the pgclusters.crunchydata.com CRD in the Namespace that the PostgreSQL cluster is in:
6. For each Deployment that represents a PostgreSQL instance in the cluster (i.e. `kubectl -n <TARGET_NAMESPACE> get deployments --selector=pgo-pg-database=true,pg-cluster=<CLUSTER_NAME>`), edit the Deployment and remove the Volume and VolumeMount entry for the tablespace. If the tablespace is called `hippo-ts`, the Volume entry will look like: “‘yaml

- name: tablespace-hippo-ts persistentVolumeClaim: claimName: -tablespace-hippo-ts `and the VolumeMount entry will look like:`yaml
- mountPath: /tablespaces/hippo-ts name: tablespace-hippo-ts “‘

2. Modify the CR entry for the PostgreSQL cluster and remove the `tablespaceMounts` entry. If your PostgreSQL cluster is called `hippo`, then the name of the CR entry is also called `hippo`. If your tablespace is called `hippo-ts`, then you would remove the YAML stanza called `hippo-ts` from the `tablespaceMounts` entry.

More Information

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

pgAdmin 4 is a popular graphical user interface that makes it easy to work with PostgreSQL databases from both a desktop or web-based client. With its ability to manage and orchestrate changes for PostgreSQL users, the PostgreSQL Operator is a natural partner to keep a pgAdmin 4 environment synchronized with a PostgreSQL environment.

The PostgreSQL Operator lets you deploy a pgAdmin 4 environment alongside a PostgreSQL cluster and keeps users' database credentials synchronized. You can simply log into pgAdmin 4 with your PostgreSQL username and password and immediately have access to your databases.

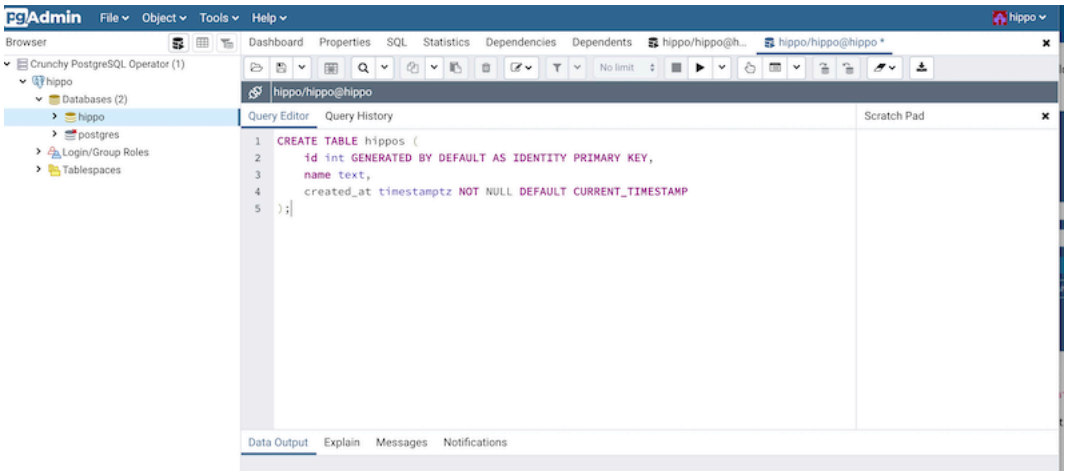

Figure 23: pgAdmin 4 Query

Deploying pgAdmin 4

For example, let's use a PostgreSQL cluster called hippo `hippo` that has a user named `hippo` with password `datalake`:

```
pgo create cluster hippo --username=hippo --password=datalake
```

After the PostgreSQL cluster becomes ready, you can create a pgAdmin 4 deployment with the [`pgo create pgadmin`]({{< relref "/pgo-client/reference/pgo_create_pgadmin.md" >}}) command:

```
pgo create pgadmin hippo
```

This creates a pgAdmin 4 deployment unique to this PostgreSQL cluster and synchronizes the PostgreSQL user information into it. To access pgAdmin 4, you can set up a port-forward to the Service, which follows the pattern `<clusterName>-pgadmin`, to port `5050`:

```
kubectl port-forward svc/hippo-pgadmin 5050:5050
```

Point your browser at `http://localhost:5050` and use your database username (e.g. `hippo`) and password (e.g. `datalake`) to log in. Though the prompt says "email address", using your PostgreSQL username will work.

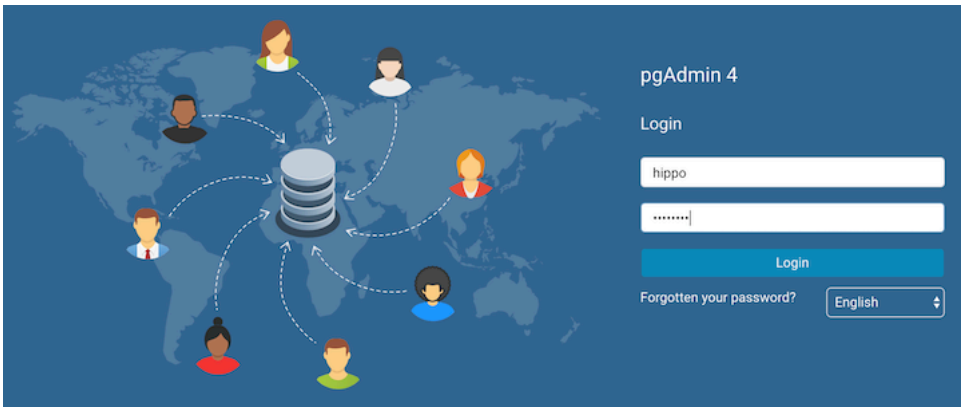

Figure 24: pgAdmin 4 Login Page

(**Note**: if your password does not appear to work, you can retry setting up the user with the [`pgo update user`]({{< relref "/pgo-client/reference/pgo_update_user.md" >}}) command: `pgo update user hippo --password=datalake`)

User Synchronization

The [`pgo create user`]({{< relref "/pgo-client/reference/pgo_create_user.md" >}}), [`pgo update user`]({{< relref "/pgo-client/reference >}}), and [`pgo delete user`]({{< relref "/pgo-client/reference/pgo_delete_user.md" >}}) commands are synchronized with the pgAdmin 4 deployment. Note that if you use `pgo create user` without the `--managed` flag prior to deploying pgAdmin 4, then the user's credentials will not be synchronized to the pgAdmin 4 deployment. However, a subsequent run of `pgo update user --password` will synchronize the credentials with pgAdmin 4.

Deleting pgAdmin 4

You can remove the pgAdmin 4 deployment with the [`pgo delete pgadmin`]({{< relref "/pgo-client/reference/pgo_delete_pgadmin.md" >}}) command.

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.

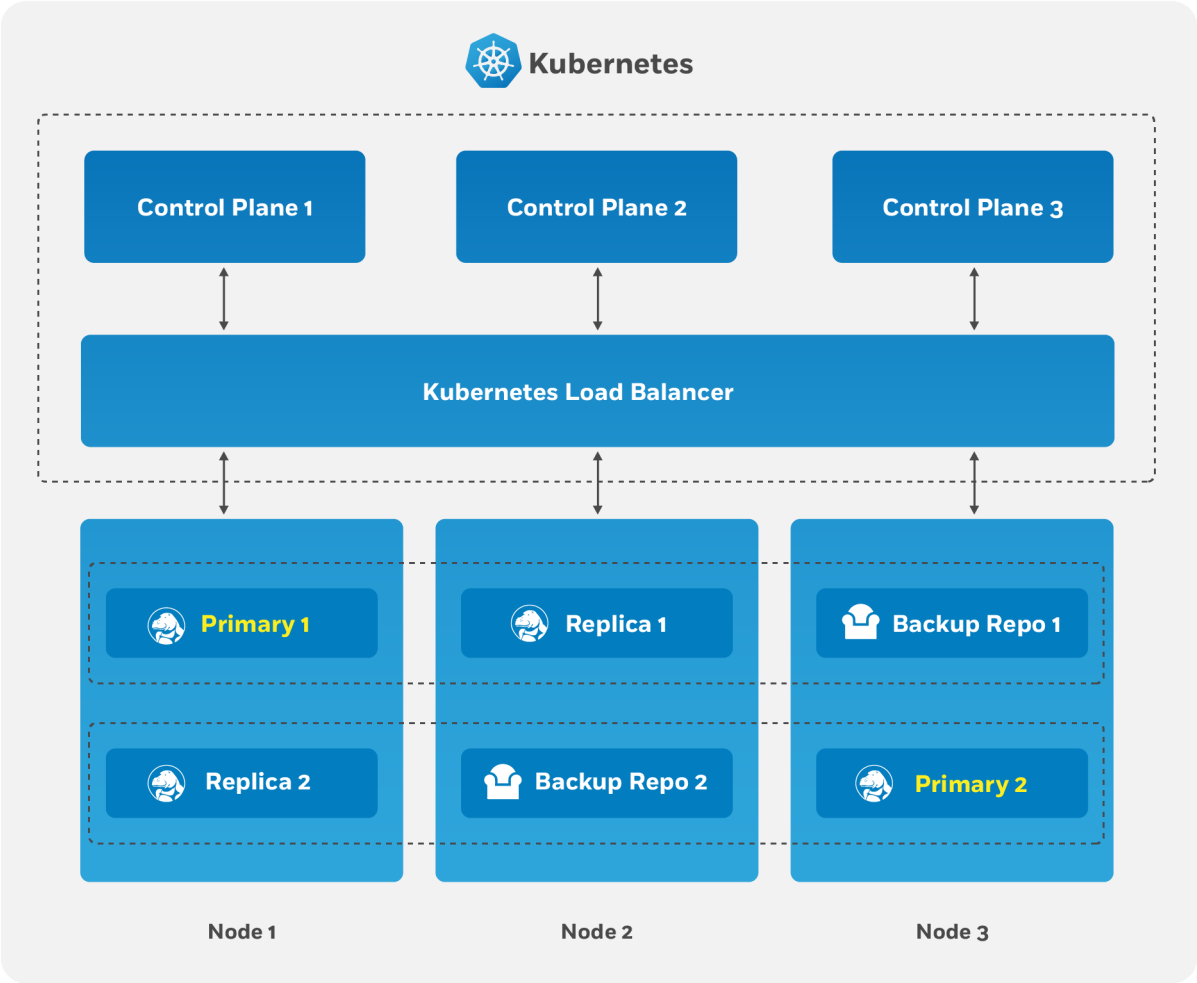

Figure 25: PostgreSQL Operator High-Availability Overview

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 repository 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: PostgreSQL has to wait for a transaction to be committed on all synchronous replicas, and 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
```

By default, node affinity 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.

The PostgreSQL Operator supports two different types of node affinity:

- `preferred`
- `required`

which can be selected with the `--node-affinity-type` flag, e.g:

```
pgo create cluster hippo \
  --node-label=region=us-east-1 --node-affinity-type=required
```

When creating a cluster, the node affinity rules will be applied to the primary and any other PostgreSQL instances that are added. If you would like to specify a node affinity rule for a specific instance, you can do so with the [`pgo scale`]({{< relref "pgo-client/reference/pgo_scale.md">}}) command and the `--node-label` and `--node-affinity-type` flags, i.e:

```
pgo scale cluster hippo \
  --node-label=region=us-south-1 --node-affinity-type=required
```

Tolerations

Kubernetes Tolerations can help with the scheduling of Pods to appropriate nodes. There are many reasons that a Kubernetes administrator may want to use tolerations, such as restricting the types of Pods that can be assigned to particular Nodes. Reasoning and strategy for using taints and tolerations is outside the scope of this documentation.

The PostgreSQL Operator supports the setting of tolerations across all PostgreSQL instances in a cluster, as well as for each particular PostgreSQL instance within a cluster. Both the `pgo create cluster`({{< relref "pgo-client/reference/pgo_create_cluster.md">}}) and [`pgo scale`]({{< relref "pgo-client/reference/pgo_scale.md">}}) commands support the `--toleration` flag, which allows for one or more tolerations to be added to a PostgreSQL cluster. Values accepted by the `--toleration` use the following format:

```
rule:Effect
```

where a `rule` can represent existence (e.g. `key`) or equality (`key=value`) and `Effect` is one of `NoSchedule`, `PreferNoSchedule`, or `NoExecute`. For more information on how tolerations work, please refer to the Kubernetes documentation.

For example, to add two tolerations to a new PostgreSQL cluster, one that is an existence toleration for a key of `ssd` and the other that is an equality toleration for a key/value pair of `zone/east`, you can run the following command:

```
pgo create cluster hippo \
  --toleration=ssd:NoSchedule \
  --toleration=zone=east:NoSchedule
```

For another example, to assign equality toleration for a key/value pair of `zone/west` to a new instance in the `hippo` cluster, you can run the following command:

```
pgo scale hippo --toleration=zone=west:NoSchedule
```

Tolerations can be updated on an existing cluster. You can do this by either modifying the `pgclusters.crunchydata.com` and `pgreplicas.crunchydata.com` custom resources directly, e.g. via the `kubectl edit` command, or with the [`pgo update cluster`]({{ relref "pgo-client/reference/pgo_update_cluster.md" }}) command. Using the `pgo update cluster` command, a toleration can be removed by adding a `-` at the end of the toleration effect.

For example, to add a toleration of `zone=west:NoSchedule` and remove the toleration of `zone=east:NoSchedule`, you could run the following command:

```
pgo update cluster hippo \
  --toleration=zone=west:NoSchedule \
  --toleration=zone-east:NoSchedule-
```

Once the updates are applied, the PostgreSQL Operator will roll out the changes to the appropriate instances.

Rolling Updates

During the lifecycle of a PostgreSQL cluster, there are certain events that may require a planned restart, such as an update to a "restart required" PostgreSQL configuration setting (e.g. `shared_buffers`) or a change to a Kubernetes Deployment template (e.g. [changing the memory request]({{< relref "tutorial/customize-cluster.md">}}#customize-cpu-memory)). Restarts can be disruptive in a high availability deployment, which is why many setups employ a "rolling update" strategy (aka a "rolling restart") to minimize or eliminate downtime during a planned restart.

Because PostgreSQL is a stateful application, a simple rolling restart strategy will not work: PostgreSQL needs to ensure that there is a primary available that can accept reads and writes. This requires following a method that will minimize the amount of downtime when the primary is taken offline for a restart.

The PostgreSQL Operator provides a mechanism for rolling updates implicitly on certain operations that change the Deployment templates (e.g. memory updates, CPU updates, adding tablespaces, modifiny annotations) and explicitly through the [`pgo restart`]({{< relref "pgo-client/reference/pgo_restart.md">}}) command with the `--rolling` flag. The PostgreSQL Operator uses the following algorithm to perform the rolling restart to minimize any potential interruptions:

1. Each replica is updated in sequential order. This follows the following process:
2. The replica is explicitly shut down to ensure any outstanding changes are flushed to disk.
3. If requested, the PostgreSQL Operator will apply any changes to the Deployment.
4. The replica is brought back online. The PostgreSQL Operator waits for the replica to become available before it proceeds to the next replica.
5. The above steps are repeated until all of the replicas are restarted.
6. A controlled switchover is performed. The PostgreSQL Operator determines which replica is the best candidate to become the new primary. It then demotes the primary to become a replica and promotes the best candidate to become the new primary.
7. The former primary follows a process similar to what is described in step 1.

The downtime is thus constrained to the amount of time the switchover takes.

A rolling update strategy will be used if any of the following changes are made to a PostgreSQL cluster, either through the `pgo update` command or from a modification to the custom resource:

- Memory resource adjustments
- CPU resource adjustments
- Custom annotation changes
- Enabling/disabling the monitoring sidecar on a PostgreSQL cluster (`--metrics`)
- Enabling/disabling the pgBadger sidecar on a PostgreSQL cluster (`--pgbadger`)
- Tablespace additions
- Toleration modifications

Advanced [high-availability]({{< relref "/architecture/high-availability/_index.md" >}}) and [disaster recovery]({{< relref "/architecture/disaster-recovery.md" >}}) strategies involve spreading your database clusters across multiple data centers to help maximize uptime. In Kubernetes, this technique is known as "federation". Federated Kubernetes clusters are able to communicate with each other, coordinate changes, and provide resiliency for applications that have high uptime requirements.

As of this writing, federation in Kubernetes is still in ongoing development and is something we monitor with intense interest. As Kubernetes federation continues to mature, we wanted to provide a way to deploy PostgreSQL clusters managed by the PostgreSQL Operator that can span multiple Kubernetes clusters. This can be accomplished with a few environmental setups:

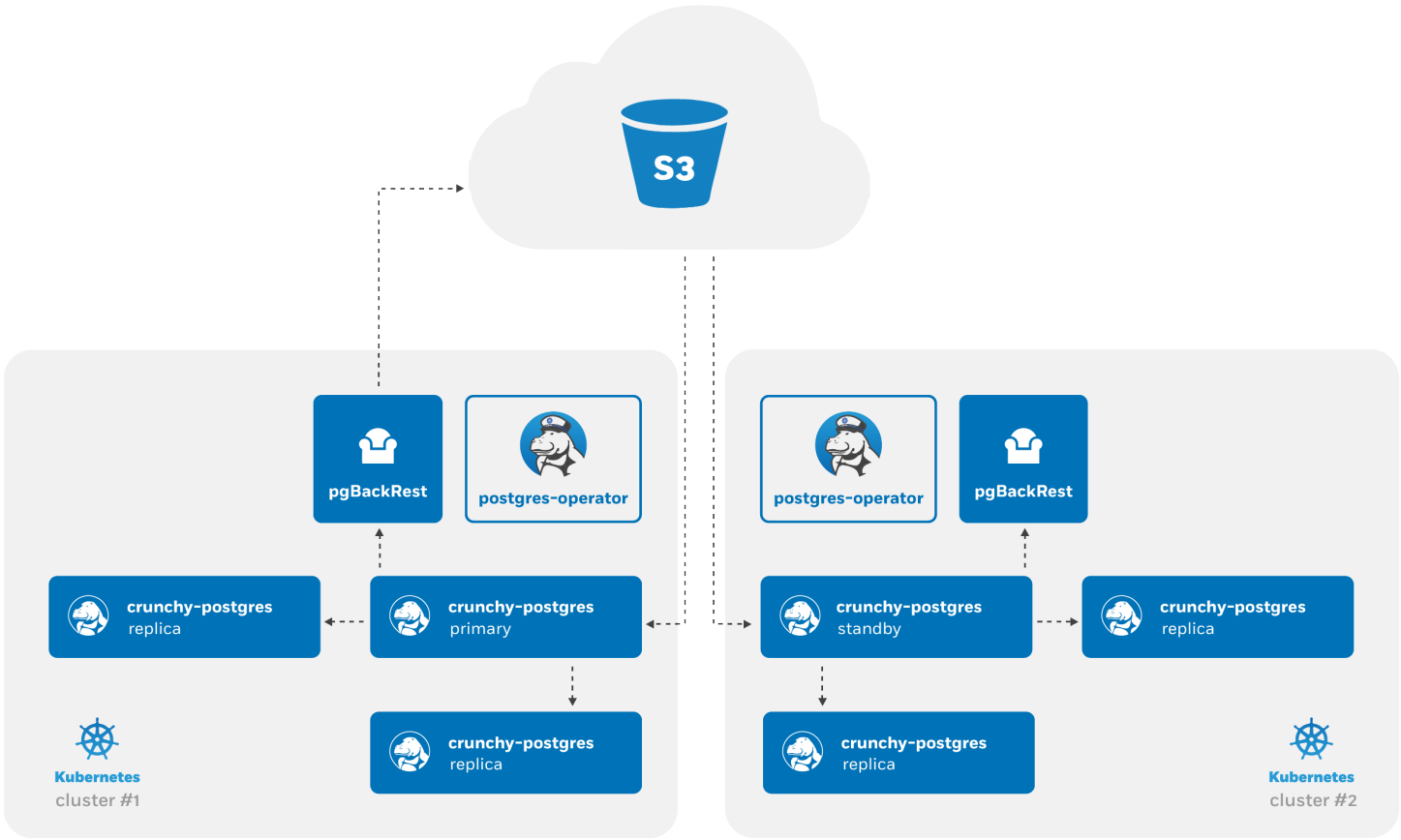

Figure 26: PostgreSQL Operator High-Availability Overview

- Two Kubernetes clusters
- S3, or an external storage system that uses the S3 protocol

At a high-level, the PostgreSQL Operator follows the "active-standby" data center deployment model for managing the PostgreSQL clusters across Kuberntetes clusters. In one Kubernetes cluster, the PostgreSQL Operator deploy PostgreSQL as an "active" PostgreSQL cluster, which means it has one primary and one-or-more replicas. In another Kubernetes cluster, the PostgreSQL cluster is deployed as a "standby" cluster: every PostgreSQL instance is a replica.

A side-effect of this is that in each of the Kubernetes clusters, the PostgreSQL Operator can be used to deploy both active and standby PostgreSQL clusters, allowing you to mix and match! While the mixing and matching may not ideal for how you deploy your PostgreSQL clusters, it does allow you to perform online moves of your PostgreSQL data to different Kubernetes clusters as well as manual online upgrades.

Lastly, while this feature does extend high-availability, promoting a standby cluster to an active cluster is **not** automatic. While the PostgreSQL clusters within a Kubernetes cluster do support self-managed high-availability, a cross-cluster deployment requires someone to specifically promote the cluster from standby to active.

Standby Cluster Overview

Standby PostgreSQL clusters are managed just like any other PostgreSQL cluster that is managed by the PostgreSQL Operator. For example, adding replicas to a standby cluster is identical to before: you can use [`pgo scale`]({{< relref "/pgo-client/reference/pgo_scale.md" >}}).

As the architecture diagram above shows, the main difference is that there is no primary instance: one PostgreSQL instance is reading in the database changes from the S3 repository, while the other replicas are replicas of that instance. This is known as cascading replication. replicas are cascading replicas, i.e. replicas replicating from a database server that itself is replicating from another database server.

Because standby clusters are effectively read-only, certain functionality that involves making changes to a database, e.g. PostgreSQL user changes, is blocked while a cluster is in standby mode. Additionally, backups and restores are blocked as well. While pgBackRest does support backups from standbys, this requires direct access to the primary database, which cannot be done until the PostgreSQL Operator supports Kubernetes federation. If a blocked function is called on a standby cluster via the [`pgo` client]({{< relref "/pgo-client/_index.md">}}) or a direct call to the API server, the call will return an error.

Key Commands

`pgo create cluster`**({{< relref "/pgo-client/reference/pgo_create_cluster.md" >}})** This first step to creating a standby PostgreSQL cluster is…to create a PostgreSQL standby cluster. We will cover how to set this up in the example below, but wanted to provide some of the standby-specific flags that need to be used when creating a standby cluster. These include:

- `--standby`: Creates a cluster as a PostgreSQL standby cluster
- `--password-superuser`: The password for the `postgres` superuser account, which performs a variety of administrative actions.
- `--password-replication`: The password for the replication account (`primaryuser`), used to maintain high-availability.
- `--password`: The password for the standard user account created during PostgreSQL cluster initialization.
- `--pgbackrest-repo-path`: The specific pgBackRest repository path that should be utilized by the standby cluster. Allows a standby cluster to specify a path that matches that of the active cluster it is replicating.
- `--pgbackrest-storage-type`: Must be set to `s3`
- `--pgbackrest-s3-key`: The S3 key to use
- `--pgbackrest-s3-key-secret`: The S3 key secret to use
- `--pgbackrest-s3-bucket`: The S3 bucket to use
- `--pgbackrest-s3-endpoint`: The S3 endpoint to use
- `--pgbackrest-s3-region`: The S3 region to use

If you do not want to set the user credentials, you can retrieve them at a later time by using the [`pgo show user`]({{< relref "/pgo-client/reference/pgo_show_user.md" >}}) command with the `--show-system-accounts` flag, e.g.

```
pgo show user --show-system-accounts hippo
```

With respect to the credentials, it should be noted that when the standby cluster is being created within the same Kubernetes cluster AND it has access to the Kubernetes Secret created for the active cluster, one can use the `--secret-from` flag to set up the credentials.

[pgo update cluster]({{< relref "/pgo-client/reference/pgo_update_cluster.md" >}}) [pgo update cluster]({{< relref "/pgo-client/reference/pgo_update_cluster.md" >}}) is responsible for the promotion and disabling of a standby cluster, and contains several flags to help with this process:

- `--enable-standby`: Enables standby mode in a cluster for a cluster. This will bootstrap a PostgreSQL cluster to become aligned with the current active cluster and begin to follow its changes.
- `--promote-standby`: Enables standby mode in a cluster. This is a destructive action that results in the deletion of all PVCs for the cluster (data will be retained according Storage Class and/or Persistent Volume reclaim policies). In order to allow the proper deletion of PVCs, the cluster must also be shutdown.
- `--shutdown`: Scales all deployments for the cluster to 0, resulting in a full shutdown of the PG cluster. This includes the primary, any replicas, as well as any supporting services (pgBackRest and pgBouncer if enabled).
- `--startup`: Scales all deployments for the cluster to 1, effectively starting a PG cluster that was previously shutdown. This includes the primary, any replicas, as well as any supporting services (pgBackRest and pgBouncer if enabled). The primary is brought online first in order to maintain a consistent primary/replica architecture across startups and shutdowns.

Creating a Standby PostgreSQL Cluster

Let's create a PostgreSQL deployment that has both an active and standby cluster! You can try this example either within a single Kubernetes cluster, or across multuple Kubernetes clusters.

First, deploy a new active PostgreSQL cluster that is configured to use S3 with pgBackRest. For example:

```
pgo create cluster hippo --pgbouncer --replica-count=2 \
  --pgbackrest-storage-type=posix,s3 \
  --pgbackrest-s3-key=<redacted> \
  --pgbackrest-s3-key-secret=<redacted> \
  --pgbackrest-s3-bucket=watering-hole \
  --pgbackrest-s3-endpoint=s3.amazonaws.com \
  --pgbackrest-s3-region=us-east-1 \
  --password-superuser=supersecrethippo \
  --password-replication=somewhatsecrethippo \
  --password=opensourcehippo
```

(Replace the placeholder values with your actual values. We are explicitly setting all of the passwords for the primary cluster to make it easier to run the example as is).

The above command creates an active PostgreSQL cluster with two replicas and a pgBouncer deployment. Wait a few moments for this cluster to become live before proceeding.

Once the cluster has been created, you can then create the standby cluster. This can either be in another Kubernetes cluster or within the same Kubernetes cluster. If using a separate Kubernetes cluster, you will need to provide the proper passwords for the superuser and replication accounts. You can also provide a password for the regular PostgreSQL database user created during cluster initialization to ensure the passwords and associated secrets across both clusters are consistent.

(If the standby cluster is being created using the same PostgreSQL Operator deployment (and therefore the same Kubernetes cluster), the `--secret-from` flag can also be used in lieu of these passwords. You would specify the name of the cluster [e.g. `hippo`] as the value of the `--secret-from` variable.)

With this in mind, create a standby cluster similar to this below:

```
pgo create cluster hippo-standby --standby --pgbouncer --replica-count=2 \
  --pgbackrest-storage-type=s3 \
  --pgbackrest-s3-key=<redacted> \
  --pgbackrest-s3-key-secret=<redacted> \
  --pgbackrest-s3-bucket=watering-hole \
  --pgbackrest-s3-endpoint=s3.amazonaws.com \
  --pgbackrest-s3-region=us-east-1 \
  --pgbackrest-repo-path=/backrestrepo/hippo-backrest-shared-repo \
  --password-superuser=supersecrethippo \
  --password-replication=somewhatsecrethippo \
  --password=opensourcehippo
```

(If you are unsure of your credentials, you can use `pgo show user hippo --show-system-accounts` to retrieve them).

Note the use of the `--pgbackrest-repo-path` flag as it points to the name of the pgBackRest repository that is used for the original `hippo` cluster.

At this point, the standby cluster will bootstrap as a standby along with two cascading replicas. pgBouncer will be deployed at this time as well, but will remain non-functional until `hippo-standby` is promoted. To see that the Pod is indeed a standby, you can check the logs.

```
kubectl logs hippo-standby-dcff544d6-s6d58…

Thu Mar 19 18:16:54 UTC 2020 INFO: Node standby-dcff544d6-s6d58 fully initialized for cluster
   standby and is ready for use
2020-03-19 18:17:03,390 INFO: Lock owner: standby-dcff544d6-s6d58; I am standby-dcff544d6-s6d58
2020-03-19 18:17:03,454 INFO: Lock owner: standby-dcff544d6-s6d58; I am standby-dcff544d6-s6d58
2020-03-19 18:17:03,598 INFO: no action.  i am the standby leader with the lock
2020-03-19 18:17:13,389 INFO: Lock owner: standby-dcff544d6-s6d58; I am standby-dcff544d6-s6d58
2020-03-19 18:17:13,466 INFO: no action.  i am the standby leader with the lock
```

You can also see that this is a standby cluster from the [`pgo show cluster`]({{< relref "/pgo-client/reference/pgo_show_cluster.md" >}}) command.

```
pgo show cluster hippo

cluster : standby (crunchy-postgres-ha:{{< param centosBase >}}-{{< param postgresVersion >}}-{{<
   param operatorVersion >}})
       standby : true
```

Promoting a Standby Cluster

There comes a time where a standby cluster needs to be promoted to an active cluster. Promoting a standby cluster means that a PostgreSQL instance within it will become a priary and start accepting both reads and writes. This has the net effect of pushing WAL (transaction archives) to the pgBackRest repository, so we need to take a few steps first to ensure we don't accidentally create a split-brain scenario.

First, if this is not a disaster scenario, you will want to "shutdown" the active PostgreSQL cluster. This can be done with the `--shutdown` flag:

```
pgo update cluster hippo --shutdown
```

The effect of this is that all the Kubernetes Deployments for this cluster are scaled to 0. You can verify this with the following command:

```
kubectl get deployments --selector pg-cluster=hippo

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
hippo                           0/0     0            0           32m
hippo-backrest-shared-repo      0/0     0            0           32m
hippo-kvfo                      0/0     0            0           27m
hippo-lkge                      0/0     0            0           27m
hippo-pgbouncer                 0/0     0            0           31m
```

We can then promote the standby cluster using the `--promote-standby` flag:

```
pgo update cluster hippo-standby --promote-standby
```

This command essentially removes the standby configuration from the Kubernetes cluster's DCS, which triggers the promotion of the current standby leader to a primary PostgreSQL instance. You can view this promotion in the PostgreSQL standby leader's (soon to be active leader's) logs:

```
kubectl logs hippo-standby-dcff544d6-s6d58…

2020-03-19 18:28:11,919 INFO: Reloading PostgreSQL configuration.
server signaled
2020-03-19 18:28:16,792 INFO: Lock owner: standby-dcff544d6-s6d58; I am standby-dcff544d6-s6d58
2020-03-19 18:28:16,850 INFO: Reaped pid=5377, exit status=0
2020-03-19 18:28:17,024 INFO: no action.  i am the leader with the lock
2020-03-19 18:28:26,792 INFO: Lock owner: standby-dcff544d6-s6d58; I am standby-dcff544d6-s6d58
2020-03-19 18:28:26,924 INFO: no action.  i am the leader with the lock
```

As pgBouncer was enabled for the cluster, the `pgbouncer` user's password is rotated, which will bring pgBouncer online with the newly promoted active cluster. If pgBouncer is still having trouble connecting, you can explicitly rotate the password with the following command:

```
pgo update pgbouncer --rotate-password hippo-standby
```

With the standby cluster now promoted, the cluster with the original active PostgreSQL cluster can now be turned into a standby PostgreSQL cluster. This is done by deleting and recreating all PVCs for the cluster and re-initializing it as a standby using the S3 repository. Being that this is a destructive action (i.e. data will only be retained if any Storage Classes and/or Persistent Volumes have the appropriate reclaim policy configured) a warning is shown when attempting to enable standby.

```
pgo update cluster hippo --enable-standby
Enabling standby mode will result in the deletion of all PVCs for this cluster!
Data will only be retained if the proper retention policy is configured for any associated storage
   classes and/or persistent volumes.
Please proceed with caution.
WARNING: Are you sure? (yes/no): yes
updated pgcluster hippo
```

To verify that standby has been enabled, you can check the DCS configuration for the cluster to verify that the proper standby settings are present.

```
kubectl get cm hippo-config -o yaml | grep standby
    %f
        \"%p\""},"use_pg_rewind":true,"use_slots":false},"standby_cluster":{"create_replica_methods":[
```

Also, the PVCs for the cluster should now only be a few seconds old, since they were recreated.

```
kubectl get pvc --selector pg-cluster=hippo
NAME                  STATUS   VOLUME           CAPACITY AGE
hippo                 Bound    crunchy-pv251    1Gi      33s
hippo-kvfo            Bound    crunchy-pv174    1Gi      29s
hippo-lkge            Bound    crunchy-pv228    1Gi      26s
hippo-pgbr-repo       Bound    crunchy-pv295    1Gi      22s
```

And finally, the cluster can be restarted:

```
pgo update cluster hippo --startup
```

At this point, the cluster will reinitialize from scratch as a standby, just like the original standby that was created above. Therefore any transactions written to the original standby, should now replicate back to this cluster.

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
4.6.0	4.6.0	13.1	2.31
		12.5	2.31
		11.10	2.31
		10.15	2.31
		9.6.20	2.31
4.5.1	4.5.1	13.1	2.29
		12.5	2.29
		11.10	2.29
		10.15	2.29
		9.6.20	2.29
		9.5.24	2.29
4.5.0	4.5.0	13.0	2.29
		12.4	2.29
		11.9	2.29
		10.14	2.29
		9.6.19	2.29
		9.5.23	2.29

Operator Release	Container Release	Postgres	PgBackrest Version
4.4.1	4.4.1	12.4	2.27
		11.9	2.27
		10.14	2.27
		9.6.19	2.27
		9.5.23	2.27
4.4.0	4.4.0	12.3	2.27
		11.8	2.27
		10.13	2.27
		9.6.18	2.27
		9.5.22	2.27
4.3.2	4.3.2	12.3	2.25
		11.8	2.25
		10.13	2.25
		9.6.18	2.25
		9.5.22	2.25
4.3.1	4.3.1	12.3	2.25
		11.8	2.25
		10.13	2.25
		9.6.18	2.25
		9.5.22	2.25
4.3.0	4.3.0	12.2	2.25
		11.7	2.25
		10.12	2.25
		9.6.17	2.25
		9.5.21	2.25
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

Operator Release	Container Release	Postgres	PgBackrest Version
		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
		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 privileged Kubernetes 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 create a `pgo-config` ConfigMap using the configuration files that are included in the Operator container.

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
```

Config Directory

Files within the *PGO_CONF_DIR* 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
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. {{< param centosBase >}}-{{< param
Port	the PostgreSQL port to use for new containers (e.g. 5432)

Setting	Definition
PGBadgerPort	the port used to connect to pgbadger (e.g. 10000)
ExporterPort	the port used to connect to postgres exporter (e.g. 9187)
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
Metrics	boolean, if set to true will cause each new cluster to include crunchy-postgres-exporter as a sidecar contai
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
DefaultInstanceMemory	string, matches a Kubernetes resource value. If set, it is used as the default value of the memory request f
DefaultBackrestMemory	string, matches a Kubernetes resource value. If set, it is used as the default value of the memory request f
DefaultPgBouncerMemory	string, matches a Kubernetes resource value. If set, it is used as the default value of the memory request f
DisableFSGroup	If set to `true`, this will disable the use of the fsGroup for the containers related to PostgreSQL, which is n

Storage

Setting	Definition
PrimaryStorage	required, the value of the storage configuration to use for the primary PostgreSQL deployment
BackupStorage	required, the value of the storage configuration to use for backups, including the storage for pgbackrest rep
ReplicaStorage	required, the value of the storage configuration to use for the replica PostgreSQL deployments
BackrestStorage	required, the value of the storage configuration to use for the pgbackrest shared repository deployment cre
WALStorage	optional, the value of the storage configuration to use for PostgreSQL Write Ahead Log
StorageClass	optional, for a dynamic storage type, you can specify the storage class used for storage provisioning (e.g. s
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
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

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
```

Miscellaneous (Pgo)

Setting	Definition
Audit	boolean, if set to true will cause each apiserver call to be logged with an *audit* marking
ConfigMapWorkerCount	The number of workers created for the worker queue within the ConfigMap controller (defaults to 2)
ControllerGroupRefreshInterval	The refresh interval for any per-namespace controller with a refresh interval (defaults to 60 seconds)
DisableReconcileRBAC	Whether or not to disable RBAC reconciliation in targeted namespaces (defaults to `false`)
NamespaceRefreshInterval	The refresh interval for the namespace controller (defaults to 60 seconds)
NamespaceWorkerCount	The number of workers created for the worker queue within the Namespace controller (defaults to 2)
PgclusterWorkerCount	The number of workers created for the worker queue within the PGCluster controller (defaults to 1)
PGOImagePrefix	image tag prefix to use for the Operator containers
PGOImageTag	image tag to use for the Operator containers
PGReplicaWorkerCount	The number of workers created for the worker queue within the PGReplica controller (defaults to 1)
PGTaskWorkerCount	The number of workers created for the worker queue within the PGTask controller (defaults to 1)

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.

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
```

There are several different ways to install and deploy the PostgreSQL Operator based upon your use case.

For the vast majority of use cases, we recommend using the PostgreSQL Operator Installer({{< relref "/installation/postgres-operator.md" >}}), which uses the `pgo-deployer` container to set up all of the objects required to run the PostgreSQL Operator.

For advanced use cases, such as for development, one may want to set up a [development environment]({{< relref "/contributing/developer-setup.md" >}}) that is created using a series of scripts controlled by the Makefile.

Before selecting your installation method, it's important that you first read the [prerequisites]({{< relref "/installation/prerequisites.md" >}}) for your deployment environment to ensure that your setup meets the needs for installing the PostgreSQL Operator.

Prerequisites

The following is required prior to installing PostgreSQL Operator.

Environment

The PostgreSQL Operator is tested in the following environments:

- Kubernetes v1.13+
- Red Hat OpenShift v3.11+
- Red Hat OpenShift v4.4+
- Amazon EKS
- VMWare Enterprise PKS 1.3+
- IBM Cloud Pak Data

IBM Cloud Pak Data If you install the PostgreSQL Operator, which comes with Crunchy PostgreSQL for Kubernetes, on IBM Cloud Pak Data, please note the following additional requirements:

- Cloud Pak Data Version 2.5
- Minimum Node Requirements (Cloud Paks Cluster): 3
- Crunchy PostgreSQL for Kuberentes (Service):
- Minimum CPU Requirements: 0.2 CPU
- Minimum Memory Requirements: 120MB
- Minimum Storage Requirements: 5MB

Note: PostgreSQL clusters deployed by the PostgreSQL Operator with Crunchy PostgreSQL for Kubernetes are workload dependent. As such, users should allocate enough resources for their PostgreSQL clusters.

Client Interfaces

The PostgreSQL Operator installer will install the [pgo client]({{< relref "/pgo-client/_index.md" >}}) interface to help with using the PostgreSQL Operator. However, it is also recommend that you have access to `kubectl` or `oc` and are able to communicate with the Kubernetes or OpenShift cluster that you are working with.

Ports

There are several application ports to note when using the PostgreSQL Operator. These ports allow for the [pgo client]({{< relref "/pgo-client/_index.md" >}}) to interface with the PostgreSQL Operator API as well as for users of the event stream to connect to `nsqd` and `nsqdadmin`:

Container	Port
API Server	8443
nsqadmin	4151
nsqd	4150

If you are using these services, ensure your cluster administrator has given you access to these ports.

Application Ports

The PostgreSQL Operator deploys different services to support a production PostgreSQL environment. Below is a list of the applications and their default Service ports.

Service	Port
PostgreSQL	5432
pgbouncer	5432
pgBackRest	2022
postgres-exporter	9187
pgbadger	10000

The PostgreSQL Operator Installer

Quickstart

If you believe that all the default settings in the installation manifest work for you, you can take a chance by running the manifest directly from the repository:

```
kubectl create namespace pgo
kubectl apply -f https://raw.githubusercontent.com/CrunchyData/postgres-operator/v{{< param
    operatorVersion >}}/installers/kubectl/postgres-operator.yml
```

However, we still advise that you read onward to see how to properly configure the PostgreSQL Operator.

Overview

The PostgreSQL Operator comes with a container called `pgo-deployer` which handles a variety of lifecycle actions for the PostgreSQL Operator, including:

- Installation
- Upgrading
- Uninstallation

After configuring the Job template, the installer can be run using `kubectl apply` and takes care of setting up all of the objects required to run the PostgreSQL Operator.

The installation manifest, called `postgres-operator.yaml`, is available in the `installers/kubectl/postgres-operator.yml` path in the PostgreSQL Operator repository.

Requirements

RBAC

The `pgo-deployer` requires a ServiceAccount and ClusterRoleBinding to run the installation job. Both of these resources are already defined in the `postgres-operator.yml`, but can be updated based on your specific environmental requirements.

By default, the `pgo-deployer` uses a ServiceAccount called `pgo-deployer-sa` that has a ClusterRoleBinding (`pgo-deployer-crb`) with several ClusterRole permissions. This is required to create the Custom Resource Definitions that power the PostgreSQL Operator. While the PostgreSQL Operator itself can be scoped to a specific namespace, you will need to have `cluster-admin` for the initial deployment, or privileges that allow you to install Custom Resource Definitions. The required list of privileges are available in the postgres-operator.yml file:

https://raw.githubusercontent.com/CrunchyData/postgres-operator/v{{< param operatorVersion >}}/installers/kubectl/postgres-operator.yml

If you have already configured the ServiceAccount and ClusterRoleBinding for the installation process (e.g. from a previous installation), then you can remove these objects from the `postgres-operator.yml` manifest.

Config Map

The `pgo-deployer` uses a Kubernetes ConfigMap to pass configuration options into the installer. The ConfigMap is defined in the `postgres-operator.yaml` file and can be updated based on your configuration preferences.

Namespaces

By default, the installer will run in the `pgo` Namespace. This can be updated in the `postgres-operator.yml` file. **Please ensure that this namespace exists before the job is run**.

For example, to create the `pgo` namespace:

```
kubectl create namespace pgo
```

The PostgreSQL Operator has the ability to manage PostgreSQL clusters across multiple Kubernetes Namespaces, including the ability to add and remove Namespaces that it watches. Doing so does require the PostgreSQL Operator to have elevated privileges, and as such, the PostgreSQL Operator comes with three "namespace modes" to select what level of privileges to provide:

- `dynamic`: The default is the default mode. This enables full dynamic Namespace management capabilities, in which the PostgreSQL Operator can create, delete and update any Namespaces within the Kubernetes cluster, while then also having the ability to create the Roles, RoleBindings andService Accounts within those Namespaces for normal operations. The PostgreSQL Operator can also listen for Namespace events and create or remove controllers for various Namespaces as changes are made to Namespaces from Kubernetes and the PostgreSQL Operator's management.
- `readonly`: In this mode, the PostgreSQL Operator is able to listen for namespace events within the Kubernetes cluster, and then manage controllers as Namespaces are added, updated or deleted. While this still requires a ClusterRole, the permissions mirror those of a "read-only" environment, and as such the PostgreSQL Operator is unable to create, delete or update Namespaces itself nor create RBAC that it requires in any of those Namespaces. Therefore, while in readonly, mode namespaces must be preconfigured with the proper RBAC as the PostgreSQL Operator cannot create the RBAC itself.
- `disabled`: Use this mode if you do not want to deploy the PostgreSQL Operator with any ClusterRole privileges, especially if you are only deploying the PostgreSQL Operator to a single namespace. This disables any Namespace management capabilities within the PostgreSQL Operator and will simply attempt to work with the target Namespaces specified during installation. If no target Namespaces are specified, then the Operator will be configured to work within the namespace in which it is deployed. As with the readonly mode, while in this mode, Namespaces must be preconfigured with the proper RBAC, since the PostgreSQL Operator cannot create the RBAC itself.

Configuration - postgres-operator.yml

The postgres-operator.yml file contains all of the configuration parameters for deploying the PostgreSQL Operator. The example file contains defaults that should work in most Kubernetes environments, but it may require some customization.

For a detailed description of each configuration parameter, please read the [PostgreSQL Operator Installer Configuration Reference](<{{< relref "/installation/configuration.md">}}>)

Configuring to Update and Uninstall The deploy job can be used to perform different deployment actions for the PostgreSQL Operator. When you run the job it will install the operator by default but you can change the deployment action to uninstall or update. The DEPLOY_ACTION environment variable in the postgres-operator.yml file can be set to install, update, and uninstall.

Image Pull Secrets

If you are pulling the PostgreSQL Operator images from a private registry, you will need to setup an imagePullSecret with access to the registry. The image pull secret will need to be added to the installer service account to have access. The secret will need to be created in each namespace that the PostgreSQL Operator will be using.

After you have configured your image pull secret in the Namespace the installer runs in (by default, this is pgo), add the name of the secret to the job yaml that you are using. You can update the existing section like this:

```
apiVersion: v1
kind: ServiceAccount
metadata:
    name: pgo-deployer-sa
    namespace: pgo
imagePullSecrets:
  - name: <image_pull_secret_name>
```

If the service account is configured without using the job yaml file, you can link the secret to an existing service account with the kubectl or oc clients.

```
# kubectl
kubectl patch serviceaccount <deployer-sa> -p '{"imagePullSecrets": [{"name": "myregistrykey"}]}'
   -n <install-namespace>

# oc
oc secrets link <registry-secret> <deployer-sa> --for=pull --namespace=<install-namespace>
```

Installation

Once you have configured the PostgreSQL Operator Installer to your specification, you can install the PostgreSQL Operator with the following command:

```
kubectl apply -f /path/to/postgres-operator.yml
```

Install the [pgo Client]({{< relref "/installation/pgo-client" >}})

To use the [pgo Client]({{< relref "/installation/pgo-client" >}}), there are a few additional steps to take in order to get it to work with you PostgreSQL Operator installation. For convenience, you can download and run the client-setup.sh script in your local environment:

```
curl https://raw.githubusercontent.com/CrunchyData/postgres-operator/v{{< param operatorVersion
   >}}/installers/kubectl/client-setup.sh > client-setup.sh
chmod +x client-setup.sh
./client-setup.sh
```

Running this script can cause existing pgo client binary, pgouser, client.crt, and client.key files to be overwritten.

The client-setup.sh script performs the following tasks:

- Sets $PGO_OPERATOR_NAMESPACE to pgo if it is unset. This is the default namespace that the PostgreSQL Operator is deployed to
- Checks for valid Operating Systems and determines which pgo binary to download
- Creates a directory in $HOME/.pgo/$PGO_OPERATOR_NAMESPACE (e.g. /home/hippo/.pgo/pgo)
- Downloads the pgo binary, saves it to in $HOME/.pgo/$PGO_OPERATOR_NAMESPACE, and sets it to be executable

- Pulls the TLS keypair from the PostgreSQL Operator `pgo.tls` Secret so that the `pgo` client can communicate with the PostgreSQL Operator. These are saved as `client.crt` and `client.key` in the `$HOME/.pgo/$PGO_OPERATOR_NAMESPACE` path.
- Pulls the `pgouser` credentials from the `pgouser-admin` secret and saves them in the format `username:password` in a file called `pgouser`
- `client.crt`, `client.key`, and `pgouser` are all set to be read/write by the file owner. All other permissions are removed.
- Sets the following environmental variables with the following values:

```
export PGOUSER=$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/pgouser
export PGO_CA_CERT=$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/client.crt
export PGO_CLIENT_CERT=$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/client.crt
export PGO_CLIENT_KEY=$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/client.key
```

For convenience, after the script has finished, you can permanently at these environmental variables to your environment:

```
cat <<EOF >> ~/.bashrc
export PATH="$HOME/.pgo/$PGO_OPERATOR_NAMESPACE:$PATH"
export PGOUSER="$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/pgouser"
export PGO_CA_CERT="$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/client.crt"
export PGO_CLIENT_CERT="$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/client.crt"
export PGO_CLIENT_KEY="$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/client.key"
EOF
```

By default, the `client-setup.sh` script targets the user that is stored in the `pgouser-admin` secret in the `pgo` (`$PGO_OPERATOR_NAMESPACE`) Namespace. If you wish to use a different Secret, you can set the `PGO_USER_ADMIN` environmental variable.

For more detailed information about [installing the `pgo` client]({{< relref "/installation/pgo-client" >}}), please see [Installing the `pgo` client]({{< relref "/installation/pgo-client" >}}).

Verify the Installation

One way to verify the installation was successful is to execute the [`pgo version`]({{< relref "/pgo-client/reference/pgo_version.md" >}}) command.

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
```

Next, in another console window, set the following environment variable to configure the API server address:

```
cat <<EOF >> ${HOME?}/.bashrc
export PGO_APISERVER_URL="https://127.0.0.1:8443"
EOF
```

Apply those changes to the current session by running:

```
source ${HOME?}/.bashrc
```

Now run the `pgo version` command:

```
pgo version
```

If successful, you should see output similar to this:

```
pgo client version {{< param operatorVersion >}}
pgo-apiserver version {{< param operatorVersion >}}
```

Post-Installation

To clean up the installer artifacts, you can simply run:

```
kubectl delete -f /path/to/postgres-operator.yml
```

Note that if you still have the ServiceAccount and ClusterRoleBinding in there, you will need to have elevated privileges.

Installing the PostgreSQL Operator Monitoring Infrastructure

Please see the [PostgreSQL Operator Monitoring installation section]({{< relref "/installation/metrics" >}}) for instructions on how to install the PostgreSQL Operator Monitoring infrastructure.

Install the PostgreSQL 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

To authenticate with the Crunchy PostgreSQL Operator API:

- 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.key ${HOME?}/.pgo/client.key && chmod 400 ${HOME?}/.pgo/client.key
```

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.key"
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 -n pgo svc/postgres-operator 8443:8443

# If deployed to OpenShift
oc port-forward -n pgo svc/postgres-operator 8443:8443
```

In the above examples, you can substitute pgo for the namespace that you deployed the PostgreSQL Operator into.

Note: The port-forward will be required for the duration of using the PostgreSQL client.

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 deploy/pgo-client -- pgo version

# or

kubectl exec -it -n pgo deploy/pgo-client 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.key "%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.key"
```

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 -n pgo svc/postgres-operator 8443:8443

# If deployed to OpenShift
oc port-forward -n pgo svc/postgres-operator 8443:8443
```

In the above examples, you can substitute `pgo` for the namespace that you deployed the PostgreSQL Operator into.

Note: The port-forward will be required for the duration of using the PostgreSQL client.

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.

PostgreSQL Operator Installer Configuration

When installing the PostgreSQL Operator you have many configuration options, these options are listed in this section.

General Configuration

These variables affect the general configuration of the PostgreSQL Operator.

Name	Default	Rec
archive_mode	true	**Re**
archive_timeout	60	**Re**
backrest_aws_s3_bucket		
backrest_aws_s3_endpoint		
backrest_aws_s3_key		
backrest_aws_s3_region		
backrest_aws_s3_secret		
backrest_aws_s3_uri_style		
backrest_aws_s3_verify_tls		
backrest_port	2022	**Re**
badger	false	**Re**
ccp_image_prefix	registry.developers.crunchydata.com/crunchydata	**Re**
ccp_image_pull_secret		
ccp_image_pull_secret_manifest		
ccp_image_tag	{{< param centosBase >}}-{{< param postgresVersion >}}-{{< param operatorVersion >}}	**Re**

Name	Default	Re
create_rbac	true	**Re**
crunchy_debug	false	
db_name		
db_password_age_days	0	
db_password_length	24	
db_port	5432	**Re**
db_replicas	0	**Re**
db_user	testuser	**Re**
default_instance_memory	128Mi	
default_pgbackrest_memory	48Mi	
default_pgbouncer_memory	24Mi	
delete_operator_namespace	false	
delete_watched_namespaces	false	
disable_auto_failover	false	
disable_fsgroup	false	
exporterport	9187	**Re**
metrics	false	**Re**
namespace	pgo	
namespace_mode	dynamic	
pgbadgerport	10000	**Re**
pgo_add_os_ca_store	false	**Re**
pgo_admin_password	examplepassword	
pgo_admin_perms	*	**Re**
pgo_admin_role_name	pgoadmin	**Re**
pgo_admin_username	admin	**Re**
pgo_apiserver_port	8443	
pgo_apiserver_url	https://postgres-operator	
pgo_client_cert_secret	pgo.tls	
pgo_client_container_install	false	
pgo_client_install	true	
pgo_client_version	{{< param operatorVersion >}}	**Re**
pgo_cluster_admin	false	**Re**
pgo_disable_eventing	false	
pgo_disable_tls	false	
pgo_image_prefix	registry.developers.crunchydata.com/crunchydata	**Re**
pgo_image_pull_secret		
pgo_image_pull_secret_manifest		
pgo_image_tag	{{< param centosBase >}}-{{< param operatorVersion >}}	**Re**
pgo_installation_name	devtest	**Re**
pgo_noauth_routes		
pgo_operator_namespace	pgo	**Re**
pgo_tls_ca_store		
pgo_tls_no_verify	false	
reconcile_rbac	true	

Name	Default	Re
scheduler_timeout	3600	**Re**
service_type	ClusterIP	
sync_replication	false	

Storage Settings

The store configuration options defined in this section can be used to specify the storage configurations that are used by the PostgreSQL Operator.

Storage Configuration Options

Kubernetes and OpenShift offer support for a wide variety of different storage types and we provide suggested configurations for different environments. These storage types can be modified or removed as needed, while additional storage configurations can also be added to meet the specific storage requirements for your PostgreSQL clusters.

The following storage variables are utilized to add or modify operator storage configurations in the with the installer:

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>_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.

Example Storage Configuration

```
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.

PostgreSQL Cluster Storage Defaults

You can specify the default storage to use for PostgreSQL, pgBackRest, and other elements that require storage that can outlast the lifetime of a Pod. While the PostgreSQL Operator defaults to using default to work with the default storage class available in your environment.

Name	Default	Required	Description
backrest_storage	default	**Required**	Set the value of the storage configuration to use for the pgbackrest shared repository deploym
backup_storage	default	**Required**	Set the value of the storage configuration to use for backups, including the storage for pgback
primary_storage	default	**Required**	Set to configure which storage definition to use when creating volumes used by PostgreSQL p
replica_storage	default	**Required**	Set to configure which storage definition to use when creating volumes used by PostgreSQL r
wal_storage			Set to configure which storage definition to use when creating volumes used for PostgreSQL W

```
backrest_storage: default
backup_storage: default
primary_storage: default
replica_storage: default
```

With the configuration shown above, the default storage class available in the deployment environment is used.

Considerations for Multi-Zone Cloud Environments

When using the Operator in a Kubernetes cluster consisting of nodes that span multiple zones, special consideration must be taken 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.

Default Storage Configuration Types

Name	Value
storage1_name	default
storage1_access_mode	ReadWriteOnce
storage1_size	1G
storage1_type	dynamic

Default StorageClass

Name	Value
storage2_name	hostpathstorage
storage2_access_mode	ReadWriteMany
storage2_size	1G
storage2_type	create

Host Path Storage

Name	Value
storage3_name	nfsstorage
storage3_access_mode	ReadWriteMany
storage3_size	1G
storage3_type	create
storage3_supplemental_groups	65534

NFS Storage

Name	Value
storage4_name	nfsstoragered
storage4_access_mode	ReadWriteMany
storage4_size	1G
storage4_match_labels	crunchyzone=red
storage4_type	create
storage4_supplemental_groups	65534

NFS Storage Red

Name	Value
storage5_name	storageos
storage5_access_mode	ReadWriteOnce
storage5_size	5Gi
storage5_type	dynamic
storage5_class	fast

StorageOS

Name	Value
storage6_name	primarysite
storage6_access_mode	ReadWriteOnce
storage6_size	4G
storage6_type	dynamic
storage6_class	primarysite

Primary Site

Name	Value
storage7_name	alternatesite
storage7_access_mode	ReadWriteOnce
storage7_size	4G
storage7_type	dynamic
storage7_class	alternatesite

Alternate Site

Name	Value
storage8_name	gce
storage8_access_mode	ReadWriteOnce
storage8_size	300M
storage8_type	dynamic

Name	Value
storage8_class	standard

GCE

Name	Value
storage9_name	rook
storage9_access_mode	ReadWriteOnce
storage9_size	1Gi
storage9_type	dynamic
storage9_class	rook-ceph-block

Rook

Pod Anti-affinity Settings

This will set the default pod anti-affinity for the deployed PostgreSQL clusters. Pod Anti-Affinity is set to determine where the PostgreSQL Pods are deployed relative to each other There are three levels:

- required: Pods *must* be scheduled to different Nodes. If a Pod cannot be scheduled to a different Node from the other Pods in the anti-affinity group, then it will not be scheduled.
- preferred (default): Pods *should* be scheduled to different Nodes. There is a chance that two Pods in the same anti-affinity group could be scheduled to the same node
- disabled: Pods do not have any anti-affinity rules

The `POD_ANTI_AFFINITY` label sets the Pod anti-affinity for all of the Pods that are managed by the Operator in a PostgreSQL cluster. In addition to the PostgreSQL Pods, this also includes the pgBackRest repository and any pgBouncer pods. By default, the pgBackRest and pgBouncer pods inherit the value of `POD_ANTI_AFFINITY`, but one can override the default by setting the `POD_ANTI_AFFINITY_PGBACKREST` and `POD_ANTI_AFFINITY_PGBOUNCER` variables for pgBackRest and pgBouncer respectively

Name	Default	Required	Description
`pod_anti_affinity`	preferred		This will set the default pod anti-affinity for the deployed PostgreSQL clusters
`pod_anti_affinity_pgbackrest`			This will set the default pod anti-affinity for the pgBackRest pods.
`pod_anti_affinity_pgbouncer`			This will set the default pod anti-affinity for the pgBouncer pods.

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:

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 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 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 configuration file for each deployment of the operator.

For each operator deployment the following 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:

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

```
# Config A
pgo_operator_namespace: 'pgo'
namespace: 'pgo'
...
```

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

```
# Config 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.

Though the years, we have built up several other methods for installing the PostgreSQL Operator. The next few sections provide some alternative ways of deploying the PostgreSQL Operator. Some of these methods are deprecated and may be removed in a future release.

A full installation of the Operator includes the following steps:

- get the Operator project
- 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 - Get Project

The Operator project is hosted on GitHub. You can get a copy using `git clone`:

```
git clone -b v{{< param operatorVersion >}} https://github.com/CrunchyData/postgres-operator.git
cd postgres-operator
```

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 ./examples/envs.sh >> $HOME/.bashrc
source $HOME/.bashrc
```

Default Installation - Namespace Creation

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.

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.

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.

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 *PGO_CONF_DIR* 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
WALStorage: gce
BackupStorage: gce
ReplicaStorage: gce
  gce:
    AccessMode:  ReadWriteOnce
```

```
    Size:  1G
    StorageType:  dynamic
    StorageClass:  standard
```

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*, and *StorageClass* 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:

```
admin: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.

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

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 administrator 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
```

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.

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 PostgreSQL Operator Helm Chart

Overview

The PostgreSQL Operator comes with a container called `pgo-deployer` which handles a variety of lifecycle actions for the PostgreSQL Operator, including:

- Installation
- Upgrading
- Uninstallation

After configuring the `values.yaml` file with you configuration options, the installer will be run using the `helm` command line tool and takes care of setting up all of the objects required to run the PostgreSQL Operator.

The `postgres-operator` Helm chart is available in the Helm directory in the PostgreSQL Operator repository.

Requirements

RBAC

The Helm chart will create the ServiceAccount, ClusterRole, and ClusterRoleBinding that are required to run the `pgo-deployer`. If you have already configured the ServiceAccount and ClusterRoleBinding for the installation process (e.g. from a previous installation), you can disable their creation using the `rbac.create` and `serviceAccount.create` variables in the `values.yaml` file. If these options are disabled, you must provide the name of your preconfigured ServiceAccount using `serviceAccount.name`.

Namespace

In order to install the PostgreSQL Operator using the Helm chart you will need to first create the namespace in which the `pgo-deployer` will be run. By default, it will run in the namespace that is provided to `helm` at the command line.

```
kubectl create namespace <namespace>
helm install postgres-operator -n <namespace> /path/to/chart_dir
```

The PostgreSQL Operator has the ability to manage PostgreSQL clusters across multiple Kubernetes Namespaces, including the ability to add and remove Namespaces that it watches. Doing so does require the PostgreSQL Operator to have elevated privileges, and as such, the PostgreSQL Operator comes with three "namespace modes" to select what level of privileges to provide. Detailed information about these "namespace modes" can be found in the Namespace(<{{< relref "/installation/postgres-operator.md" >}}>) section here.

Config Map

The `pgo-deployer` uses a Kubernetes ConfigMap to pass configuration options into the installer. The values in your `values.yaml` file will be used to populate the configuation options in the ConfigMap.

Configuration - `values.yaml`

The `values.yaml` file contains all of the configuration parametes for deploying the PostgreSQL Operator. The values.yaml file contains the defaults that should work in most Kubernetes environments, but it may require some customization.

For a detailed description of each configuration parameter, please read the [PostgreSQL Operator Installer Configuration Reference](<{{< relref "/installation/configuration.md">}}>)

Installation

Once you have configured the PostgreSQL Operator Installer to your specification, you can install the PostgreSQL Operator with the following command:

```
helm install <name> -n <namespace> /path/to/chart_dir
```

Take note of the `name` used when installing, this `name` will be used to upgrade and uninstall the PostgreSQL Operator.

Install the [`pgo` Client]({{< relref "/installation/pgo-client" >}})

To use the [`pgo` Client]({{< relref "/installation/pgo-client" >}}), there are a few additional steps to take in order to get it to work with your PostgreSQL Operator installation. For convenience, you can download and run the `client-setup.sh` script in your local environment:

```
curl
   https://raw.githubusercontent.com/CrunchyData/postgres-operator/master/installers/kubectl/client-s
   > client-setup.sh
chmod +x client-setup.sh
./client-setup.sh
```

Running this script can cause existing `pgo` client binary, `pgouser`, `client.crt`, and `client.key` files to be overwritten.

The `client-setup.sh` script performs the following tasks:

- Sets `$PGO_OPERATOR_NAMESPACE` to `pgo` if it is unset. This is the default namespace that the PostgreSQL Operator is deployed to
- Checks for valid Operating Systems and determines which `pgo` binary to download
- Creates a directory in `$HOME/.pgo/$PGO_OPERATOR_NAMESPACE` (e.g. `/home/hippo/.pgo/pgo`)
- Downloads the `pgo` binary, saves it to in `$HOME/.pgo/$PGO_OPERATOR_NAMESPACE`, and sets it to be executable
- Pulls the TLS keypair from the PostgreSQL Operator `pgo.tls` Secret so that the `pgo` client can communicate with the PostgreSQL Operator. These are saved as `client.crt` and `client.key` in the `$HOME/.pgo/$PGO_OPERATOR_NAMESPACE` path.
- Pulls the `pgouser` credentials from the `pgouser-admin` secret and saves them in the format `username:password` in a file called `pgouser`
- `client.crt`, `client.key`, and `pgouser` are all set to be read/write by the file owner. All other permissions are removed.
- Sets the following environmental variables with the following values:

```
export PGOUSER=$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/pgouser
export PGO_CA_CERT=$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/client.crt
export PGO_CLIENT_CERT=$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/client.crt
export PGO_CLIENT_KEY=$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/client.key
```

For convenience, after the script has finished, you can permanently add these environmental variables to your environment:

```
cat <<EOF >> ~/.bashrc
export PATH="$HOME/.pgo/$PGO_OPERATOR_NAMESPACE:$PATH"
export PGOUSER="$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/pgouser"
export PGO_CA_CERT="$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/client.crt"
export PGO_CLIENT_CERT="$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/client.crt"
export PGO_CLIENT_KEY="$HOME/.pgo/$PGO_OPERATOR_NAMESPACE/client.key"
EOF
```

By default, the `client-setup.sh` script targets the user that is stored in the `pgouser-admin` secret in the `pgo` (`$PGO_OPERATOR_NAMESPACE`) Namespace. If you wish to use a different Secret, you can set the `PGO_USER_ADMIN` environmental variable.

For more detailed information about [installing the `pgo` client]({{< relref "/installation/pgo-client" >}}), please see [Installing the `pgo` client]({{< relref "/installation/pgo-client" >}}).

Verify the Installation

One way to verify the installation was successful is to execute the [`pgo version`]({{< relref "/pgo-client/reference/pgo_version.md" >}}) command.

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
```

In another console window, run the `pgo version` command:

```
pgo version
```

If successful, you should see output similar to this:

```
pgo client version {{< param operatorVersion >}}
pgo-apiserver version {{< param operatorVersion >}}
```

Upgrade and Uninstall

Once install has be completed using Helm, it will also be used to upgrade and uninstall your PostgreSQL Operator.

The `name` and `namespace` in the following sections should match the options provided at install.

Upgrade

To make changes to your deployment of the PostgreSQL Operator you will use the `helm upgrade` command. Once the configuration changes have been made to you `values.yaml` file, you can run the following command to implement them in the deployment:

```
helm upgrade <name> -n <namespace> /path/to/updated_chart
```

Uninstall

To uninstall the PostgreSQL Operator you will use the `helm uninstall` command. This will uninstall the operator and clean up resources used by the `pgo-deployer`.

```
helm uninstall <name> -n <namespace>
```

Debugging

When the `pgo-deployer` job does not complete successfully, the resources that are created and normally cleaned up by Helm will be left in your Kubernetes cluster. This will allow you to use the failed job and its logs to debug the issue. The following command will show the logs for the `pgo-deployer` job:

```
kubectl logs -n <namespace> job.batch/pgo-deploy
```

You can also view the logs as the job is running by using the `kubectl -f` follow flag:

```
kubectl logs -n <namespace> job.batch/pgo-deploy -f
```

These logs will provide feedback if there are any misconfigurations in your install. Once you have finished debugging the failed job and fixed any configuration issues, you can take steps to re-run your install, upgrade, or uninstall. By running another command the resources from the failed install will be cleaned up so that a successfull install can run.

The PostgreSQL Operator is installed as part of Crunchy PostgreSQL for GKE that is available in the Google Cloud Marketplace.

Step 1: Install

Install Crunchy PostgreSQL for GKE to a Google Kubernetes Engine cluster using Google Cloud Marketplace.

Step 2: Verify Installation

Install `kubectl` using the `gcloud components` command of the Google Cloud SDK or by following the Kubernetes documentation.

Using the `gcloud` utility, ensure you are logged into the GKE cluster in which you installed the PostgreSQL Operator, and see that it is running in the namespace in which you installed it. For example, in the `pgo` namespace:

```
kubectl -n pgo get deployments,pods
```

If successful, you should see output similar to this:

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

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

Step 3: Install the PostgreSQL Operator User Keys

You will need to get TLS keys used to secure the Operator REST API. Again, in the `pgo` namespace:

```
kubectl -n pgo get secret pgo.tls -o 'go-template={{ index .data "tls.crt" | base64decode }}' >
   /tmp/client.crt
kubectl -n pgo get secret pgo.tls -o 'go-template={{ index .data "tls.key" | base64decode }}' >
   /tmp/client.key
```

Step 4: 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 5: 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 3 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=pgo
```

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=pgo
EOF

source ~/.bashrc
```

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

Step 6: 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 {{< param operatorVersion >}} of the PostgreSQL Operator, you must use the `pgo` for {{< param operatorVersion >}}.

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 7: 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 {{< param operatorVersion >}}
pgo-apiserver version {{< param operatorVersion >}}
```

Step 8: 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 9: 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`.

If your Kubernetes cluster is already running the Operator Lifecycle Manager, the PostgreSQL Operator can be installed as part of Crunchy PostgreSQL for Kubernetes that is available in OperatorHub.io.

Before You Begin

There are some optional Secrets you can add before installing the PostgreSQL Operator into your cluster.

Secrets (optional)

If you plan to use AWS S3 to store backups and would like to have the keys available for every backup, you can create a Secret as described below:

```
kubectl -n "$PGO_OPERATOR_NAMESPACE" create secret generic pgo-backrest-repo-config \
  --from-literal=aws-s3-key="<your-aws-s3-key>" \
  --from-literal=aws-s3-key-secret="<your-aws-s3-key-secret>"
```

Certificates (optional)

The PostgreSQL Operator has an API that uses TLS to communicate securely with clients. If you have a certificate bundle validated by your organization, you can install it now. If not, the API will automatically generate and use a self-signed certificate.

```
kubectl -n "$PGO_OPERATOR_NAMESPACE" create secret tls pgo.tls \
  --cert=/path/to/server.crt \
  --key=/path/to/server.key
```

Installation

Create an `OperatorGroup` and a `Subscription` in your chosen namespace. Make sure the `source` and `sourceNamespace` match the CatalogSource from earlier.

```
kubectl -n "$PGO_OPERATOR_NAMESPACE" create -f- <<YAML
---
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: postgresql
spec:
  targetNamespaces: ["$PGO_OPERATOR_NAMESPACE"]

---
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: postgresql
spec:
  name: postgresql
  channel: stable
  source: operatorhubio-catalog
  sourceNamespace: olm
  startingCSV: postgresoperator.v{{< param operatorVersion >}}
YAML
```

After You Install

Once the PostgreSQL Operator is installed in your Kubernetes cluster, you will need to do a few things to use the [PostgreSQL Operator Client]({{< relref "/pgo-client/_index.md" >}}).

Install the first set of client credentials and download the `pgo` binary and client certificates.

```
PGO_CMD=kubectl ./deploy/install-bootstrap-creds.sh
PGO_CMD=kubectl ./installers/kubectl/client-setup.sh
```

The client needs to be able to reach the PostgreSQL Operator API from outside the Kubernetes cluster. Create an external service or forward a port locally.

```
kubectl -n "$PGO_OPERATOR_NAMESPACE" expose deployment postgres-operator --type=LoadBalancer

export PGO_APISERVER_URL="https://$(
  kubectl -n "$PGO_OPERATOR_NAMESPACE" get service postgres-operator \
    -o jsonpath="{.status.loadBalancer.ingress[*]['ip','hostname']}"
):8443"
```

or

```
kubectl -n "$PGO_OPERATOR_NAMESPACE" port-forward deployment/postgres-operator 8443

export PGO_APISERVER_URL="https://127.0.0.1:8443"
```

Verify connectivity using the `pgo` command.

```
pgo version
# pgo client version {{< param operatorVersion >}}
# pgo-apiserver version {{< param operatorVersion >}}
```

Crunchy Data PostgreSQL Operator Playbooks

The Crunchy Data PostgreSQL Operator Playbooks contain Ansible roles for installing and managing the [Crunchy Data PostgreSQL Operator]({{< relref "/installation/other/ansible/installing-operator.md" >}}).

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
- support a variety of deployment models

Resources

- Ansible
- Crunchy Data
- 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 2.9.0+

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

Permissions

The installation of the Crunchy PostgreSQL Operator requires elevated privileges, as the following objects need to be created:

- Custom Resource Definitions
- Cluster RBAC for using one of the multi-namespace modes
- Create required namespaces

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.

Obtaining Operator Ansible Role

- Clone the postgres-operator project

GitHub Installation

All necessary files (inventory.yaml, values.yaml, main playbook and roles) can be found in the `installers/ansible` directory in the source code.

Configuring the Inventory File

The `inventory.yaml` file included with the PostgreSQL Operator Playbooks allows installers to configure how Ansible will connect to your Kubernetes cluster. This file should contain the following connection variables:

You will have to uncomment out either the `kubernetes` or `openshift` variables if you are being using them for your environment. Both sets of variables cannot be used at the same time. The unused variables should be left commented out or removed.

Name	Default	Required	Description
`kubernetes_context`		**Required**, if deploying to Kubernetes	When deploying to Kubernetes, set to configure the co
`openshift_host`		**Required**, if deploying to OpenShift	When deploying to OpenShift, set to configure the hos
`openshift_password`		**Required**, if deploying to OpenShift	When deploying to OpenShift, set to configure the pas
`openshift_skip_tls_verify`		**Required**, if deploying to OpenShift	When deploying to Openshift, set to ignore the integri
`openshift_token`		**Required**, if deploying to OpenShift	When deploying to OpenShift, set to configure the tok
`openshift_user`		**Required**, if deploying to OpenShift	When deploying to OpenShift, set to configure the use

To retrieve the `kubernetes_context` value for Kubernetes installs, run the following command:

```
kubectl config current-context
```

Configuring - `values.yaml`

The `values.yaml` file contains all of the configuration parameters for deploying the PostgreSQL Operator. The example file contains defaults that should work in most Kubernetes environments, but it may require some customization.

For a detailed description of each configuration parameter, please read the [PostgreSQL Operator Installer Configuration Reference](<{{< relref "/installation/configuration.md">}}>)

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][ansible-prerequisites] we can now install the PostgreSQL Operator.

The commands should be run in the directory where the Crunchy PostgreSQL Operator playbooks are stored. See the `installers/ansible` directory in the Crunchy PostgreSQL Operator project for the inventory file, values 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.yaml --tags=install --ask-become-pass 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.yaml --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.yaml --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.

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

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.key"
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 -n pgo svc/postgres-operator 8443:8443

# If deployed to OpenShift
oc port-forward -n pgo svc/postgres-operator 8443:8443
```

You can subsitute `pgo` in the above examples with the namespace that you deployed the PostgreSQL Operator into.

On a separate terminal verify the PostgreSQL client 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.

[ansible-prerequisites]: {{< relref "/installation/other/ansible/prerequisites.md" >}}

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 `values.yaml`
- Reinstall the `pgo` client if a new version is specified

The following assumes the proper [prerequisites are satisfied][ansible-prerequisites] 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, values 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.yaml --tags=update --ask-become-pass 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.yaml --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.yaml --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.key"
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 -n pgo svc/postgres-operator 8443:8443

# If deployed to OpenShift
oc port-forward -n pgo svc/postgres-operator 8443:8443
```

In the above examples, you can substitute `pgo` for the namespace that you deployed the PostgreSQL Operator into.

On a separate terminal verify the PostgreSQL Operator client can communicate with the 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.

[ansible-prerequisites]: {{< relref "/installation/other/ansible/prerequisites.md" >}}

Uninstalling PostgreSQL Operator

The following assumes the proper [prerequisites are satisfied][ansible-prerequisites] 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.yaml --tags=uninstall --ask-become-pass main.yml
```

Deleting `pgo` Client

If variable `pgo_client_install` is set to `true` in the `values.yaml` 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
```

[ansible-prerequisites]: {{< relref "/installation/other/ansible/prerequisites.md" >}}

The PostgreSQL Operator Monitoring infrastructure is a fully integrated solution for monitoring and visualizing metrics captured from PostgreSQL clusters created using the PostgreSQL Operator. By leveraging pgMonitor to configure and integrate the various tools, components and metrics needed to effectively monitor PostgreSQL clusters, the PostgreSQL Operator Monitoring infrastructure provides an powerful and easy-to-use solution to effectively monitor and visualize pertinent PostgreSQL database and container metrics. Included in the monitoring infrastructure are the following components:

- pgMonitor - Provides the configuration needed to enable the effective capture and visualization of PostgreSQL database metrics using the various tools comprising the PostgreSQL Operator Monitoring infrastructure
- Grafana - Enables visual dashboard capabilities for monitoring PostgreSQL clusters, specifically using Crunchy PostgreSQL Exporter data stored within Prometheus
- Prometheus - A multi-dimensional data model with time series data, which is used in collaboration with the Crunchy PostgreSQL Exporter to provide and store metrics
- Alertmanager - Handles alerts sent by Prometheus by deduplicating, grouping, and routing them to reciever integrations.

When installing the monitoring infrastructure, various configuration options and settings are available to tailor the installation according to your needs. For instance, custom dashboards and datasources can be utilized with Grafana, while custom scrape configurations can be utilized with Promtheus. Please see the [monitoring configuration reference](<{{< relref "/installation/metrics/metrics-configuration.md">}}>) for additional details.

By leveraging the various installation methods described in this section, the PostgreSQL Operator Metrics infrastructure can be deployed alongside the PostgreSQL Operator. There are several different ways to install and deploy the PostgreSQL Operator Monitoring infrastructure based upon your use case.

For the vast majority of use cases, we recommend using the PostgreSQL Operator Monitoring Installer({{< relref "/installation/metrics/postg
operator-metrics.md" >}}), which uses the `pgo-deployer` container to set up all of the objects required to run the PostgreSQL Operator Monitoring infrastructure.
Additionally, Ansible(<{{< relref "/installation/metrics/metrics-configuration.md">}}>) and Helm(<{{< relref "/installation/metrics/other/ansible">}}>) installers are available.

Before selecting your installation method, it's important that you first read the [prerequisites]({{< relref "/installation/metrics/metrics-prerequisites.md" >}}) for your deployment environment to ensure that your setup meets the needs for installing the PostgreSQL Operator Monitoring infrastructure.

Prerequisites

The following is required prior to installing PostgreSQL Operator Monitoring.

Environment

PostgreSQL Operator Monitoring is tested in the following environments:

- Kubernetes v1.13+
- Red Hat OpenShift v3.11+
- Red Hat OpenShift v4.3+
- VMWare Enterprise PKS 1.3+

Application Ports

The PostgreSQL Operator Monitoring installer deploys different services as needed to support PostgreSQL Operator Monitoring collection and monitoring. Below is a list of the applications and their default Service ports.

Service	Port
Grafana	3000
Prometheus	9090
Alertmanager	9093

PostgreSQL Operator Monitoring Installer

Quickstart

If you believe that all the default settings in the installation manifest work for you, you can take a chance by running the metrics manifest directly from the repository:

```
kubectl create namespace pgo
```

```
kubectl apply -f https://raw.githubusercontent.com/CrunchyData/postgres-operator/v{{< param
    operatorVersion >}}/installers/metrics/kubectl/postgres-operator-metrics.yml
```

However, we still advise that you read onward to see how to properly configure the PostgreSQL Operator Monitoring infrastructure.

Overview

The PostgreSQL Operator comes with a container called `pgo-deployer` which handles a variety of lifecycle actions for the PostgreSQL Operator Monitoring infrastructure, including:

- Installation
- Upgrading
- Uninstallation

After configuring the Job template, the installer can be run using `kubectl apply` and takes care of setting up all of the objects required to run the PostgreSQL Operator.

The installation manifest, called `postgres-operator-metrics.yml`, is available in the `installers/metrics/kubectl/postgres-operator-` path in the PostgreSQL Operator repository.

Requirements

RBAC

The `pgo-deployer` requires a ServiceAccount and ClusterRoleBinding to run the installation job. Both of these resources are already defined in the `postgres-operator-metrics.yml`, but can be updated based on your specific environmental requirements.

By default, the `pgo-deployer` uses a ServiceAccount called `pgo-metrics-deployer-sa` that has a ClusterRoleBinding (`pgo-metrics-deploy` with several ClusterRole permissions. This ClusterRole is needed for the initial configuration and deployment of the various applications comprising the monitoring infrastructure. This includes permissions to create:

- RBAC for use by Prometheus and/or Grafana
- The metrics namespace

The required list of privileges are available in the postgres-operator-metrics.yml file:

https://raw.githubusercontent.com/CrunchyData/postgres-operator/v{{< param operatorVersion >}}/installers/metrics/kubectl/postgres-operator-metrics.yml

If you have already configured the ServiceAccount and ClusterRoleBinding for the installation process (e.g. from a previous installation), then you can remove these objects from the `postgres-operator-metrics.yml` manifest.

Config Map

The `pgo-deployer` uses a Kubernetes ConfigMap to pass configuration options into the installer. The ConfigMap is defined in the `postgres-operator-metrics.yaml` file and can be updated based on your configuration preferences.

Namespaces

By default, the PostgreSQL Operator Monitoring installer will run in the `pgo` Namespace. This can be updated in the `postgres-operator-me` file. **Please ensure that this namespace exists before the job is run**.

For example, to create the `pgo` namespace:

```
kubectl create namespace pgo
```

Configuration - `postgres-operator-metrics.yml`

The `postgres-operator-metrics.yml` file contains all of the configuration parameters for deploying PostgreSQL Operator Monitoring. The example file contains defaults that should work in most Kubernetes environments, but it may require some customization.

For a detailed description of each configuration parameter, please read the [PostgreSQL Operator Monitoring Installer Configuration Reference](<{{< relref "/installation/metrics/metrics-configuration.md">}}>)

Configuring to Update and Uninstall The deploy job can be used to perform different deployment actions for the PostgreSQL Operator Monitoring infrastructure. When you run the job it will install the monitoring infrastructure by default but you can change the deployment action to uninstall or update. The `DEPLOY_ACTION` environment variable in the `postgres-operator-metrics.yml` file can be set to `install-metrics`, `update-metrics`, and `uninstall-metrics`.

Image Pull Secrets

If you are pulling PostgreSQL Operator Monitoring images from a private registry, you will need to setup an imagePullSecret with access to the registry. The image pull secret will need to be added to the installer service account to have access. The secret will need to be created in each namespace that the PostgreSQL Operator will be using.

After you have configured your image pull secret in the Namespace the installer runs in (by default, this is `pgo`), add the name of the secret to the job yaml that you are using. You can update the existing section like this:

```
apiVersion: v1
kind: ServiceAccount
metadata:
    name: pgo-metrics-deployer-sa
    namespace: pgo
imagePullSecrets:
  - name: <image_pull_secret_name>
```

If the service account is configured without using the job yaml file, you can link the secret to an existing service account with the `kubectl` or `oc` clients.

```
# kubectl
kubectl patch serviceaccount <deployer-sa> -p '{"imagePullSecrets": [{"name": "myregistrykey"}]}'
   -n <install-namespace>

# oc
oc secrets link <registry-secret> <deployer-sa> --for=pull --namespace=<install-namespace>
```

Installation

Once you have configured the PostgreSQL Operator Monitoring installer to your specification, you can install the PostgreSQL Operator Monitoring infrastructure with the following command:

```
kubectl apply -f /path/to/postgres-operator-metrics.yml
```

Post-Installation

To clean up the installer artifacts, you can simply run:

```
kubectl delete -f /path/to/postgres-operator-metrics.yml
```

Note that if you still have the ServiceAccount and ClusterRoleBinding in there, you will need to have elevated privileges.

PostgreSQL Operator Monitoring Installer Configuration

When installing the PostgreSQL Operator Monitoring infrastructure you have various configuration options available, which are defined below.

General Configuration

These variables affect the general configuration of PostgreSQL Operator Monitoring.

Name	Default	Required	Description
`alertmanager_log_level`	info		Set the log level for Alertmanager logging.
`alertmanager_service_type`	ClusterIP	**Required**	How to expose the Alertmanager service.
`alertmanager_storage_access_mode`	ReadWriteOnce	**Required**	Set to the access mode used by the configured storage class for

Name	Default	Required	Description
alertmanager_storage_class_name	fast		Set to the name of the storage class used when creating Alertm
alertmanager_supplemental_groups	65534		Set to configure any supplemental groups that should be added
alertmanager_volume_size	1Gi	**Required**	Set to the size of persistent volume to create for Alertmanager.
create_rbac	true	**Required**	Set to true if the installer should create the RBAC resources re
db_port	5432	**Required**	Set to configure the PostgreSQL port used by all PostgreSQL c
delete_metrics_namespace	false		Set to configure whether or not the metrics namespace (defined
disable_fsgroup	false		Set to `true` for deployments where you do not want to have the
grafana_admin_password	admin	**Required**	Set to configure the login password for the Grafana administrat
grafana_admin_username	admin	**Required**	Set to configure the login username for the Grafana administrat
grafana_install	true	**Required**	Set to true to install Grafana to visualize metrics.
grafana_service_type	ClusterIP	**Required**	How to expose the Grafana service.
grafana_storage_access_mode	ReadWriteOnce	**Required**	Set to the access mode used by the configured storage class for
grafana_storage_class_name	fast		Set to the name of the storage class used when creating Grafan
grafana_supplemental_groups	65534		Set to configure any supplemental groups that should be added
grafana_volume_size	1Gi	**Required**	Set to the size of persistent volume to create for Grafana.
metrics_image_pull_secret			Name of a Secret containing credentials for container image reg
metrics_image_pull_secret_manifest			Provide a path to the image Secret manifest to be installed in t
metrics_namespace	1G	**Required**	The namespace that should be created (if it doesn't already exi
pgbadgerport	10000	**Required**	Set to configure the port used by pgbadger in any PostgreSQL
prometheus_install	false	**Required**	Set to true to install Promotheus in order to capture metrics ex
prometheus_service_type	true	**Required**	How to expose the Prometheus service.
prometheus_storage_access_mode	ReadWriteOnce	**Required**	Set to the access mode used by the configured storage class for
prometheus_storage_class_name	fast		Set to the name of the storage class used when creating Promet
prometheus_supplemental_groups	65534		Set to configure any supplemental groups that should be added
prometheus_volume_size	1Gi	**Required**	Set to the size of persistent volume to create for Prometheus.

Custom Configuration

When installing the PostgreSQL Operator Monitoring infrastructure, it is possible to further customize the various Deployments included (e.g. Alertmanager, Grafana, and/or Prometheus) using custom configuration files. Specifically, by pointing the PostgreSQL Operator Monitoring installer to one or more ConfigMaps containing any desired custom configuration settings, those settings will then be applied during configuration and installation of the PostgreSQL Operator Monitoring infrastructure.

The specific custom configuration settings available are as follows:

Name	Default	Required	Description
alertmanager_custom_config	alertmanager-config		The name of a ConfigMap containing a custom `alertma`
alertmanager_custom_rules_config	alertmanager-rules-config		The name of a ConfigMap container custom alerting rule
grafana_datasources_custom_config	grafana-datasources		The name of a ConfigMap containing custom Grafana da
grafana_dashboards_custom_config	grafana-dashboards		The name of a ConfigMap containing custom Grafana da
prometheus_custom_configmap	crunchy-prometheus		The name of a ConfigMap containing a custom `prometh`

Please note that when using custom ConfigMaps per the above configuration settings, any defaults for the specific configuration being customized are no longer applied.

Using Alertmanager

The Alertmanager deployment requires a custom configuration file to configure reciever integrations that are supported by Prometheus Alertmanager. The installer will create a configmap containing an example Alertmanager configuration file created by the pgMonitor project, this file can be found in the pgMonitor repository. This example file, along with the Alertmanager configuration docs, will help you to configure alerting for you specific use cases.

Alertmanager cannot be installed without also deploying the Crunchy Prometheus deployment. Once both are deployed, Prometheus is automatically configured to send alerts to the Alertmanager.

Using RedHat Certified Containers & Custom Images

By default, the PostgreSQL Operator Monitoring installer will deploy the official Grafana and Prometheus containers that are publically available on dockerhub:

- https://hub.docker.com/r/grafana/grafana
- https://hub.docker.com/r/prom/prometheus
- https://hub.docker.com/r/prom/alertmanager

However, if RedHat certified containers are needed, the following certified images have also been verified with the PostgreSQL Operator Metric infrastructure, and can therefore be utilized instead:

- https://catalog.redhat.com/software/containers/openshift4/ose-grafana/5cdc17d55a13467289f58321
- https://catalog.redhat.com/software/containers/openshift4/ose-prometheus/5cdc1e585a13467289f5841a
- https://catalog.redhat.com/software/containers/openshift4/ose-prometheus-alertmanager/5cdc1cfbbed8bd5717d60b17

The following configuration settings can be applied to properly configure the image prefix, name and tag as needed to use the RedHat certified containers:

Name	Default	Required	Description
`alertmanager_image_prefix`	prom	**Required**	Configure the image prefix to use for the Alertmanager container.
`alertmanager_image_name`	alertmanager	**Required**	Configure the image name to use for the Alertmanager container.
`alertmanager_image_tag`	v0.21.0	**Required**	Configures the image tag to use for the Alertmanager container.
`grafana_image_prefix`	grafana	**Required**	Configures the image prefix to use for the Grafana container.
`grafana_image_name`	grafana	**Required**	Configures the image name to use for the Grafana container.
`grafana_image_tag`	6.7.5	**Required**	Configures the image tag to use for the Grafana container.
`prometheus_image_prefix`	prom	**Required**	Configures the image prefix to use for the Prometheus container.
`prometheus_image_name`	promtheus	**Required**	Configures the image name to use for the Prometheus container.
`prometheus_image_tag`	v2.24.0	**Required**	Configures the image tag to use for the Prometheus container.

Additionally, these same settings can be utilized as needed to support custom image names, tags, and additional container registries.

Helm Only Configuration Settings

When using Helm, the following settings can be defined to control the image prefix and image tag utilized for the `pgo-deployer` container that is run to install, update or uninstall the PostgreSQL Operator Monitoring infrastructure:

Name	Default	Required	Description
`pgo_image_prefix`	registry.developers.crunchydata.com/crunchydata	**Required**	Configures the image prefix used by the
`pgo_image_tag`	{{< param centosBase >}}-{{< param operatorVersion >}}	**Required**	Configures the image tag used by the `pg`

This section provides additional methods for installing the PostgreSQL Operator Metrics infrastructure.

The PostgreSQL Operator Monitoring Helm Chart

Overview

The PostgreSQL Operator comes with a container called `pgo-deployer` which handles a variety of lifecycle actions for the PostgreSQL Operator Monitoring infrastructure, including:

- Installation
- Upgrading
- Uninstallation

After configuring the `values.yaml` file with you configuration options, the installer will be run using the `helm` command line tool and takes care of setting up all of the objects required to run the PostgreSQL Operator.

The PostgreSQL Operator Monitoring Helm chart is available in the Helm directory in the PostgreSQL Operator repository.

Requirements

RBAC

The Helm chart will create the ServiceAccount, ClusterRole, and ClusterRoleBinding that are required to run the `pgo-deployer`. If you have already configured the ServiceAccount and ClusterRoleBinding for the installation process (e.g. from a previous installation), you can disable their creation using the `rbac.create` and `serviceAccount.create` variables in the `values.yaml` file. If these options are disabled, you must provide the name of your preconfigured ServiceAccount using `serviceAccount.name`.

Namespace

In order to install the PostgreSQL Operator using the Helm chart you will need to first create the namespace in which the `pgo-deployer` will be run. By default, it will run in the namespace that is provided to `helm` at the command line.

```
kubectl create namespace <namespace>
helm install postgres-operator-metrics -n <namespace> /path/to/chart_dir
```

Config Map

The `pgo-deployer` uses a Kubernetes ConfigMap to pass configuration options into the installer. The values in your `values.yaml` file will be used to populate the configuation options in the ConfigMap.

Configuration - `values.yaml`

The `values.yaml` file contains all of the configuration parameters for deploying the PostgreSQL Operator Monitoring infrastructure. The values.yaml file contains the defaults that should work in most Kubernetes environments, but it may require some customization.

For a detailed description of each configuration parameter, please read the [PostgreSQL Operator Monitoring Installer Configuration Reference](<{{< relref "/installation/metrics/metrics-configuration.md">}}>)

Installation

Once you have configured the PostgreSQL Operator Monitoring installer to your specification, you can install the PostgreSQL Operator Monitoring infrastructure with the following command:

```
helm install <name> -n <namespace> /path/to/chart_dir
```

Take note of the `name` used when installing, this `name` will be used to upgrade and uninstall the PostgreSQL Operator.

Upgrade and Uninstall

Once install has be completed using Helm, it will also be used to upgrade and uninstall your PostgreSQL Operator.

The `name` and `namespace` in the following sections should match the options provided at install.

Upgrade

To make changes to your deployment of the PostgreSQL Operator you will use the `helm upgrade` command. Once the configuration changes have been made to you `values.yaml` file, you can run the following command to implement them in the deployment:

```
helm upgrade <name> -n <namespace> /path/to/updated_chart
```

Uninstall

To uninstall the PostgreSQL Operator you will use the `helm uninstall` command. This will uninstall the operator and clean up resources used by the `pgo-deployer`.

```
helm uninstall <name> -n <namespace>
```

Debugging

When the `pgo-deployer` job does not complete successfully, the resources that are created and normally cleaned up by Helm will be left in your Kubernetes cluster. This will allow you to use the failed job and its logs to debug the issue. The following command will show the logs for the `pgo-deployer` job:

```
kubectl logs -n <namespace> job.batch/pgo-metrics-deploy
```

You can also view the logs as the job is running by using the `kubectl -f` follow flag:

```
kubectl logs -n <namespace> job.batch/pgo-metrics-deploy -f
```

These logs will provide feedback if there are any misconfigurations in your install. Once you have finished debugging the failed job and fixed any configuration issues, you can take steps to re-run your install, upgrade, or uninstall. By running another command the resources from the failed install will be cleaned up so that a successfull install can run.

Crunchy Data PostgreSQL Operator Monitoring Playbooks

The Crunchy Data PostgreSQL Operator Monitoring Playbooks contain Ansible roles for installing and managing the [Crunchy Data PostgreSQL Operator Monitoring infrastructure]({{< relref "/installation/other/ansible/installing-operator.md" >}}).

Features

The playbooks provided allow users to:

- install PostgreSQL Operator Monitoring on Kubernetes and OpenShift
- install PostgreSQL Operator from a Linux, Mac or Windows (Ubuntu subsystem) host
- support a variety of deployment models

Resources

- Ansible
- Crunchy Data
- Crunchy Data PostgreSQL Operator Project

Prerequisites

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

- postgres-operator playbooks source code for the target version
- Ansible 2.9.0+

Kubernetes Installs

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

OpenShift Installs

- OpenShift v3.11+
- 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

Permissions

The installation of the Crunchy PostgreSQL Operator Monitoring infrastructure requires elevated privileges, as the following objects need to be created:

- RBAC for use by Prometheus and/or Grafana
- The metrics namespace

Obtaining Operator Ansible Role

- Clone the postgres-operator project

GitHub Installation

All necessary files (inventory.yaml, values.yaml, main playbook and roles) can be found in the `installers/metrics/ansible` directory in the source code.

Configuring the Inventory File

The `inventory.yaml` file included with the PostgreSQL Operator Monitoring Playbooks allows installers to configure how Ansible will connect to your Kubernetes cluster. This file should contain the following connection variables:

You will have to uncomment out either the `kubernetes` or `openshift` variables if you are being using them for your environment. Both sets of variables cannot be used at the same time. The unused variables should be left commented out or removed.

Name	Default	Required	Description
`kubernetes_context`		**Required**, if deploying to Kubernetes	When deploying to Kubernetes, set to configure the co
`openshift_host`		**Required**, if deploying to OpenShift	When deploying to OpenShift, set to configure the hos
`openshift_password`		**Required**, if deploying to OpenShift	When deploying to OpenShift, set to configure the pas
`openshift_skip_tls_verify`		**Required**, if deploying to OpenShift	When deploying to Openshift, set to ignore the integri
`openshift_token`		**Required**, if deploying to OpenShift	When deploying to OpenShift, set to configure the tok
`openshift_user`		**Required**, if deploying to OpenShift	When deploying to OpenShift, set to configure the use

To retrieve the `kubernetes_context` value for Kubernetes installs, run the following command:

```
kubectl config current-context
```

Configuring - `values.yaml`

The `values.yaml` file contains all of the configuration parameters for deploying the PostgreSQL Operator Monitoring infrastructure. The example file contains defaults that should work in most Kubernetes environments, but it may require some customization.

For a detailed description of each configuration parameter, please read the [PostgreSQL Operator Installer Metrics Configuration Reference](<{{< relref "/installation/metrics/metrics-configuration.md">}}>)

Installing the Monitoring Infrastructure

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][ansible-prerequisites] we can now install the PostgreSQL Operator.

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.yaml --tags=install-metrics 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.yaml --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.yaml --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 <metrics_namespace>
kubectl get pods -n <metrics_namespace>

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

Verify Alertmanager

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

```
# If deployed to Kubernetes
kubectl port-forward -n <METRICS_NAMESPACE> svc/crunchy-alertmanager  9093:9093

# If deployed to OpenShift
oc port-forward -n <METRICS_NAMESPACE> svc/crunchy-alertmanager 9093:9093
```

In a browser navigate to `http://127.0.0.1:9093` to access the Alertmanager dashboard.

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 -n <METRICS_NAMESPACE> svc/crunchy-grafana 3000:3000

# If deployed to OpenShift
oc port-forward -n <METRICS_NAMESPACE> svc/crunchy-grafana 3000:3000
```

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

No metrics will be scraped if no exporters are available. To create a PostgreSQL cluster with metric exporters, run the following command following installation of the PostgreSQL Operator:

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

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 -n <METRICS_NAMESPACE> svc/crunchy-prometheus  9090:9090

# If deployed to OpenShift
oc port-forward -n <METRICS_NAMESPACE> svc/crunchy-prometheus 9090:9090
```

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

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>
```

[ansible-prerequisites]: {{< relref "/installation/metrics/other/ansible/metrics-prerequisites.md" >}}

Updating the Monitoring Infrastructure

Updating the PostgreSQL Operator Monitoring infrastrcutre is essential to the lifecycle management of the service. Using the `update-metrics` flag will:

- Update and redeploy the monitoring infrastructure deployments
- Recreate configuration maps and/or secrets used by the monitoring infrastructure
- Remove any deprecated objects
- Allow administrators to change settings configured in the `values.yaml`

The following assumes the proper [prerequisites are satisfied][ansible-prerequisites] 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, values 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.yaml --tags=update --ask-become-pass 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.yaml --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.yaml --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 <metrics_namespace>
kubectl get pods -n <metrics_namespace>

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

Verify Alertmanager

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

```
# If deployed to Kubernetes
kubectl port-forward -n <METRICS_NAMESPACE> svc/crunchy-alertmanager  9093:9093

# If deployed to OpenShift
oc port-forward -n <METRICS_NAMESPACE> svc/crunchy-alertmanager 9093:9093
```

In a browser navigate to `http://127.0.0.1:9093` to access the Alertmanager dashboard.

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 -n <METRICS_NAMESPACE> svc/crunchy-grafana 3000:3000

# If deployed to OpenShift
oc port-forward -n <METRICS_NAMESPACE> svc/crunchy-grafana 3000:3000
```

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

No metrics will be scraped if no exporters are available. To create a PostgreSQL cluster with metric exporters, run the following command following installation of the PostgreSQL Operator:

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

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 -n <METRICS_NAMESPACE> svc/crunchy-prometheus  9090:9090

# If deployed to OpenShift
oc port-forward -n <METRICS_NAMESPACE> svc/crunchy-prometheus 9090:9090
```

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

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>
```

[ansible-prerequisites]: {{< relref "/installation/metrics/other/ansible/metrics-prerequisites.md" >}}

Uninstalling the Monitoring Infrastructure

The following assumes the proper [prerequisites are satisfied][ansible-prerequisites] we can now uninstall the PostgreSQL Operator Monitoring infrastructure.

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

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

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

[ansible-prerequisites]: {{< relref "/installation/metrics/other/ansible/metrics-prerequisites.md" >}}

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

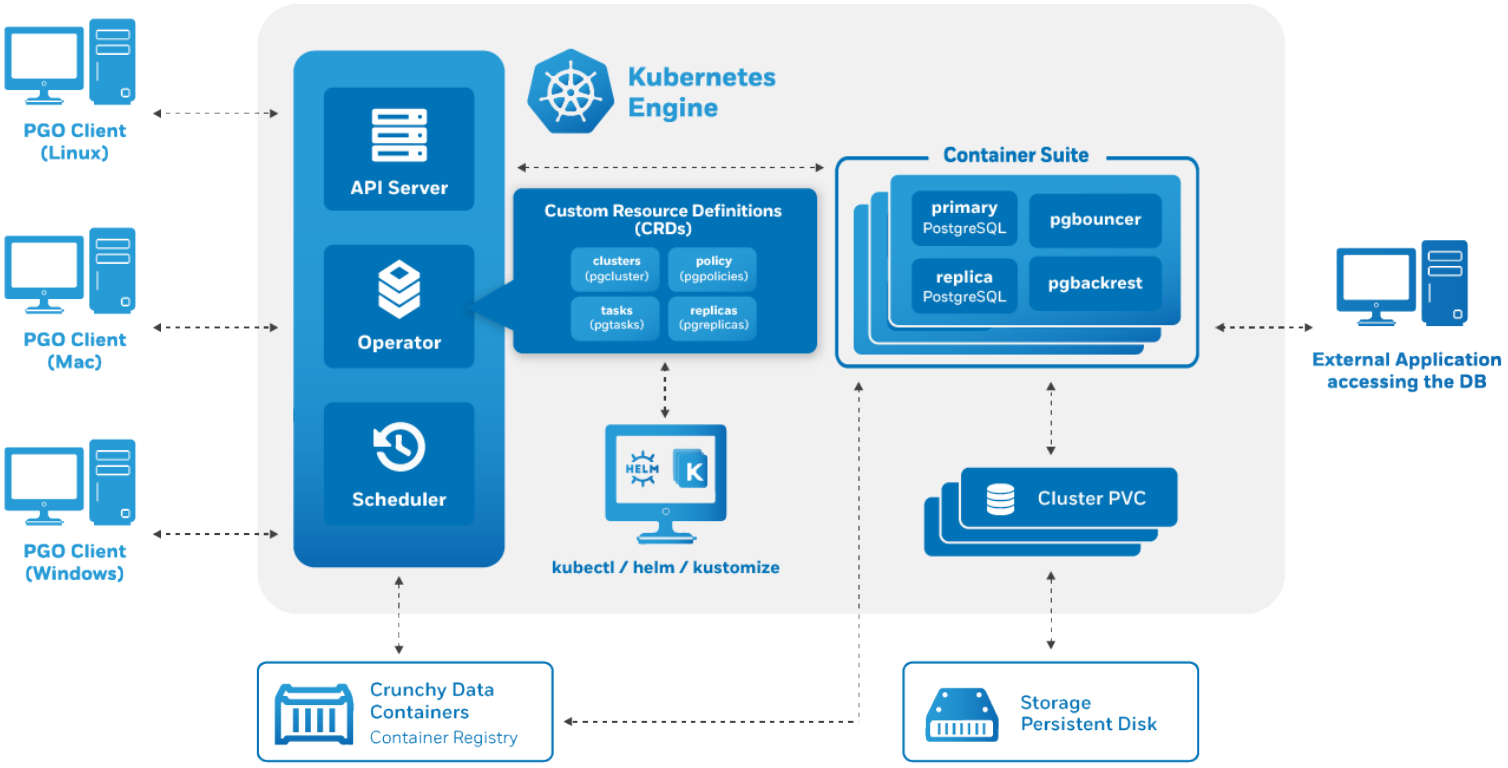

Figure 27: Architecture

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 install the PostgreSQL Operator to a namespace called `pgo`. 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 `pgo` namespace in the examples below. For even more convenience, we recommend setting `pgo` 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=pgo
```

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 pgo
```

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 have a label mat
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.
create	`pgo create cluster mycluster`	Create an Operator resource type (e.g. cluster, policy, schedule, us
delete	`pgo delete cluster mycluster`	Delete an Operator resource type (e.g. cluster, policy, user, schedu
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).
reload	`pgo reload mycluster`	Perform a `pg_ctl` reload command on a Postgres cluster(s).
restore	`pgo restore mycluster`	Perform a `pgbackrest` or `pgdump` restore on a Postgres cluster.

Operation	Syntax	Description
scale	`pgo scale mycluster`	Create a Postgres replica(s) for a given Postgres cluster.
scaledown	`pgo scaledown mycluster --query`	Delete a replica from a Postgres cluster.
show	`pgo show cluster mycluster`	Display Operator resource information (e.g. cluster, user, policy, sc
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, or namespace.
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. Note that the
`--debug`	Enable additional output for debugging.
`--disable-tls`	Disable TLS authentication to the Postgres Operator.
`--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.
`PGO_APISERVER_URL`	The URL for the PostgreSQL Operator apiserver that will process the request from the pgo client. Note that the U
`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.

- Common pgo Client Tasks
- pgo Client Reference

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 install the PostgreSQL Operator to a namespace called pgo. 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 pgo namespace in the examples below. For even more convenience, we recommend setting pgo 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=pgo
```

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 pgo
```

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 {{< param operatorVersion >}}
pgo-apiserver version {{< param operatorVersion >}}
```

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: {{< param centosBase >}}-{{< param postgresVersion >}}-{{< param operatorVersion >}}
  Policies: ""
  Metrics: false
  Badger: false
  Port: "5432"
  PGBadgerPort: "10000"
  ExporterPort: "9187"
  User: testuser
  Database: userdb
  PasswordAgeDays: "60"
  PasswordLength: "8"
  Replicas: "0"
  ServiceType: ClusterIP
  BackrestPort: 2022
  Backrest: true
  BackrestS3Bucket: ""
  BackrestS3Endpoint: ""
  BackrestS3Region: ""
  BackrestS3URIStyle: ""
  BackrestS3VerifyTLS: true
  DisableAutofail: false
  DisableReplicaStartFailReinit: false
  PodAntiAffinity: preferred
  SyncReplication: false
Pgo:
  Audit: false
  PGOImagePrefix: crunchydata
  PGOImageTag: {{< param centosBase >}}-{{< param operatorVersion >}}
PrimaryStorage: nfsstorage
BackupStorage: nfsstorage
ReplicaStorage: nfsstorage
BackrestStorage: nfsstorage
Storage:
  nfsstorage:
    AccessMode: ReadWriteMany
    Size: 1G
    StorageType: create
    StorageClass: ""
    SupplementalGroups: "65534"
    MatchLabels: ""
```

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: admin
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 Different PVC Sizes You can also create a PostgreSQL cluster with an arbitrary PVC size using the `pgo create cluster` command. For example, if you want to create a PostgreSQL cluster with with a 128GB PVC, you can use the following command:

```
pgo create cluster hacluster --pvc-size=128Gi
```

The above command sets the PVC size for all PostgreSQL instances in the cluster, i.e. the primary and replicas.

This also extends to the size of the pgBackRest repository as well, if you are using the local Kubernetes cluster storage for your backup repository. To create a PostgreSQL cluster with a pgBackRest repository that uses a 1TB PVC, you can use the following command:

```
pgo create cluster hacluster --pgbackrest-pvc-size=1Ti
```

Specify CPU / Memory for a PostgreSQL Cluster To specify the amount of CPU and memory to request for a PostgreSQL cluster, you can use the `--cpu` and `--memory` flags of the `pgo create cluster` command. Both of these values utilize the Kubernetes quantity format for specifying how to allocate resources.

For example, to create a PostgreSQL cluster that requests 4 CPU cores and has 16 gibibytes of memory, you can use the following command:

```
pgo create cluster hacluster --cpu=4 --memory=16Gi
```

Create a PostgreSQL Cluster with PostGIS To create a PostgreSQL cluster that uses the geospatial extension PostGIS, you can execute the following command, updated with your desired image tag. In the example below, the cluster will use PostgreSQL {{< param postgresVersion >}} and PostGIS {{< param postgisVersion >}}:

```
pgo create cluster hagiscluster \
  --ccp-image=crunchy-postgres-gis-ha \
  --ccp-image-tag={{< param centosBase >}}-{{< param postgresVersion >}}-{{< param postgisVersion
     >}}-{{< param operatorVersion >}}
```

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 --tablespace=name=ts1:storageconfig=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 \
    --tablespace=name=ts1:storageconfig=gce \
    --tablespace=name=ts2:storageconfig=gce
```

Tablespaces are immediately available once the PostgreSQL cluster is provisioned. For example, to create a table using the tablespace `ts1`, you can run the following SQL on your PostgreSQL cluster:

```
CREATE TABLE sensor_data (
  id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  sensor1 numeric,
  sensor2 numeric,
  sensor3 numeric,
  sensor4 numeric
)
TABLESPACE ts1;
```

You can also create tablespaces that have different sized PVCs from the ones defined in the storage specification. For instance, to create two tablespaces, one that uses a 10GiB PVC and one that uses a 20GiB PVC, you can execute the following command:

```
pgo create cluster hactsluster \
    --tablespace=name=ts1:storageconfig=gce:pvcsize=10Gi \
    --tablespace=name=ts2:storageconfig=gce:pvcsize=20Gi
```

Create a PostgreSQL Cluster Using a Backup from Another PostgreSQL Cluster It is also possible to create a new PostgreSQL Cluster using a backup from another PostgreSQL cluster. To do so, simply specify the cluster containing the backup that you would like to utilize using the `restore-from` option:

```
pgo create cluster hacluster2 --restore-from=hacluster1
```

When using this approach, a `pgbackrest restore` will be performed using the pgBackRest repository for the `restore-from` cluster specified in order to populate the initial `PGDATA` directory for the new PostgreSQL cluster. By default, pgBackRest will restore to the latest backup available and replay all WAL. However, a `restore-opts` option is also available that allows the `restore` command to be further customized, e.g. to perform a point-in-time restore and/or restore from an S3 storage bucket:

```
pgo create cluster hacluster2 \
  --restore-from=hacluster1 \
  --restore-opts="--repo-type=s3 --type=time --target='2020-07-02 20:19:36.13557+00'"
```

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:{{< param centosBase >}}-{{< param postgresVersion
    >}}-{{< param operatorVersion >}})
    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 : archive-timeout=60 deployment-name=hacluster pg-cluster=hacluster
       crunchy-pgha-scope=hacluster pgo-version={{< param operatorVersion >}}
       current-primary=hacluster name=hacluster pgouser=admin
       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 (i.e. a “clone” or “copy”) as well as a point-in-time-recovery. There are two types of ways to restore a cluster:

- Restore to a new cluster using the `--restore-from` flag in the `pgo create cluster`({{< relref “/pgo-client/reference/pgo_create_clus >}}) command. This is effectively a clone or a copy.
- Restore in-place using the [`pgo restore`]({{< relref “/pgo-client/reference/pgo_restore.md” >}}) command. Note that this is **destructive**.

It is typically better to perform a restore to a new cluster, particularly when performing a point-in-time-recovery, as it can allow you to more effectively manage your downtime and avoid making undesired changes to your production data.

Additionally, the “restore to a new cluster” technique works so long as you have a pgBackRest repository available: the pgBackRest repository does not need to be attached to an active cluster! For example, if a cluster named `hippo` was deleted as such:

```
pgo delete cluster hippo --keep-backups
```

you can create a new cluster from the backups like so:

```
pgo create cluster datalake --restore-from=hippo
```

Below provides guidance on how to perform a restore to a new PostgreSQL cluster both as a full copy and to a specific point in time. Additionally, it also shows how to restore in place to a specific point in time.

Restore to a New Cluster (aka "copy" or "clone") Restoring to a new PostgreSQL cluster allows one to take a backup and create a new PostgreSQL cluster that can run alongside an existing PostgreSQL cluster. There are several scenarios where using this technique is helpful:

- Creating a copy of a PostgreSQL cluster that can be used for other purposes. Another way of putting this is "creating a clone."
- Restore to a point-in-time and inspect the state of the data without affecting the current cluster

and more.

Full Restore To create a new PostgreSQL cluster from a backup and restore it fully, you can execute the following command:

```
pgo create cluster newcluster --restore-from=oldcluster
```

Point-in-time-Recovery (PITR) To create a new PostgreSQL cluster and restore it to specific point-in-time (e.g. before a key table was dropped), you can use the following command, substituting the time that you wish to restore to:

```
pgo create cluster newcluster \
  --restore-from oldcluster \
  --restore-opts "--type=time --target='2019-12-31 11:59:59.999999+00'"
```

When the restore is complete, the cluster is immediately available for reads and writes. To inspect the data before allowing connections, add pgBackRest's `--target-action=pause` option to the `--restore-opts` parameter.

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

Restore in-place Restoring a PostgreSQL cluster in-place is a **destructive** action that will perform a recovery on your existing data directory. This is accomplished using the [`pgo restore`]({{< relref "/pgo-client/reference/pgo_restore.md" >}}) command. The most common scenario is to restore the database to a specific point in time.

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 31, 2019 at 11:59pm:

```
pgo restore hacluster --pitr-target="2019-12-31 11:59:59.999999+00" \
  --backup-opts="--type=time"
```

When the restore is complete, the cluster is immediately available for reads and writes. To inspect the data before allowing connections, add pgBackRest's `--target-action=pause` option to the `--backup-opts` parameter.

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

Using this technique, after a restore is complete, you will need to re-enable high availability on the PostgreSQL cluster manually. You can re-enable high availability by executing the following command:

```
pgo update cluster hacluster --enable-autofail
```

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 the 'postgres' database, you can run the following command:

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

To create a logical backup of a specific database, you can use the `--database` flag, as in the following command:

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

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
```

To restore to a specific database, add the `--pgdump-database` flag to the command from above:

```
pgo restore hacluster --backup-type=pgdump --backup-pvc=hacluster-pgdump-pvc \
  --pgdump-database=mydb --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. There are two ways to issue a manual failover to your PostgreSQL cluster:

1. Allow for the PostgreSQL Operator to select the best replica candidate to failover to
2. Select your own replica candidate to failover to.

To have the PostgreSQL Operator select the best replica candidate for failover, all you need to do is execute the following command:

```
pgo failover hacluster
```

If you wish to have your cluster manually failover, you must 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 hacluster --query
```

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.

Both methods perform the failover immediately upon execution.

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.

Monitoring

PostgreSQL Metrics via pgMonitor

You can view metrics about your PostgreSQL cluster using PostgreSQL Operator Monitoring({{< relref "/installation/metrics" >}}), which uses open source pgMonitor. First, you need to install the PostgreSQL Operator Monitoring({{< relref "/installation/metrics" >}}) stack for your PostgreSQL Operator environment.

After that, you need to ensure that you deploy the `crunchy-postgres-exporter` with each PostgreSQL cluster that you deploy:

```
pgo create cluster hippo --metrics
```

For more information on how monitoring with the PostgreSQL Operator works, please see the Monitoring({{< relref "/architecture/monitoring.md" >}}) section of the documentation.

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
```

Cluster Maintenance & Resource Management

There are several operations that you can perform to modify a PostgreSQL cluster over its lifetime.

Modify CPU / Memory for a PostgreSQL Cluster As database workloads change, it may be necessary to modify the CPU and memory allocation for your PostgreSQL cluster. The PostgreSQL Operator allows for this via the `--cpu` and `--memory` flags on the `pgo update cluster` command. Similar to the create command, both flags accept values that follow the Kubernetes quantity format.

For example, to update a PostgreSQL cluster to use 8 CPU cores and has 32 gibibytes of memory, you can use the following command:

```
pgo update cluster hacluster --cpu=8 --memory=32Gi
```

The resource allocations apply to all instances in a PostgreSQL cluster: this means your primary and any replicas will have the same cluster resource allocations. Be sure to specify resource requests that your Kubernetes environment can support.

NOTE: This operation can cause downtime. Modifying the resource requests allocated to a Deployment requires that the Pods in a Deployment must be restarted. Each PostgreSQL instance is safely shutdown using the "fast" shutdown method to help ensure it will not enter crash recovery mode when a new Pod is created.

When the operation completes, each PostgreSQL instance will have the new resource allocations.

Adding a Tablespace to a Cluster Based on your workload or volume of data, you may wish to add a tablespace to your PostgreSQL cluster.

You can add a tablespace to an existing PostgreSQL cluster with the `pgo update cluster` command. Adding a tablespace to a cluster uses a similar syntax to creating a cluster with a tablespace, for example:

```
pgo update cluster hacluster \
    --tablespace=name=tablespace3:storageconfig=storageconfigname
```

NOTE: This operation can cause downtime. In order to add a tablespace to a PostgreSQL cluster, persistent volume claims (PVCs) need to be created and mounted to each PostgreSQL instance in the cluster. The act of mounting a new PVC to a Kubernetes Deployment causes the Pods in the deployment to restart.

Each PostgreSQL instance is safely shutdown using the "fast" shutdown method to help ensure it will not enter crash recovery mode when a new Pod is created.

When the operation completes, the tablespace will be set up and accessible to use within the PostgreSQL cluster.

For more information on tablespaces, please visit the tablespace section of the documentation.

Clone a PostgreSQL Cluster

You can create a copy of an existing PostgreSQL cluster in a new PostgreSQL cluster by using the `pgo create cluster`({{< relref "/pgo-client/reference/pgo_create_cluster.md" >}}) command with the `--restore-from` flag (and, if needed, `--restore-opts`). The command copies the pgBackRest repository from either an active PostgreSQL cluster, or a pgBackRest repository that exists from a former cluster that was deleted using `pgo delete cluster --keep-backups`.

You can clone a PostgreSQL cluster by running the following command:

```
pgo create cluster newcluster --restore-from=oldcluster
```

By leveraging `pgo create cluster`, you are able to copy the data from a PostgreSQL cluster while creating the topology of a new cluster the way you want to. For instance, if you want to copy data from an existing cluster that does not have metrics to a new cluster that does, you can accomplish that with the following command:

```
pgo create cluster newcluster --restore-from=oldcluster --metrics
```

Clone a PostgreSQL Cluster to Different PVC Size

You can have a cloned PostgreSQL cluster use a different PVC size, which is useful when moving your PostgreSQL cluster to a larger PVC. For example, to clone a PostgreSQL cluster to a 256GiB PVC, you can execute the following command:

```
pgo create cluster bighippo --restore-from=hippo  --pvc-size=256Gi
```

You can also have the cloned PostgreSQL cluster use a larger pgBackRest backup repository by setting its PVC size. For example, to have a cloned PostgreSQL cluster use a 1TiB pgBackRest repository, you can execute the following command:

```
pgo create cluster bighippo --restore-from=hippo --pgbackrest-pvc-size=1Ti
```

Enable TLS

TLS allows secure TCP connections to PostgreSQL, and the PostgreSQL Operator makes it easy to enable this PostgreSQL feature. The TLS support in the PostgreSQL Operator does not make an opinion about your PKI, but rather loads in your TLS key pair that you wish to use for the PostgreSQL server as well as its corresponding certificate authority (CA) certificate. Both of these Secrets are required to enable TLS support for your PostgreSQL cluster when using the PostgreSQL Operator, but it in turn allows seamless TLS support.

Setup

There are three items that are required to enable TLS in your PostgreSQL clusters:

- A CA certificate
- A TLS private key
- A TLS certificate

There are a variety of methods available to generate these items: in fact, Kubernetes comes with its own certificate management system! It is up to you to decide how you want to manage this for your cluster. The PostgreSQL documentation also provides an example for how to generate a TLS certificate as well.

To set up TLS for your PostgreSQL cluster, you have to create two Secrets: one that contains the CA certificate, and the other that contains the server TLS key pair.

First, create the Secret that contains your CA certificate. Create the Secret as a generic Secret, and note that the following requirements **must** be met:

- The Secret must be created in the same Namespace as where you are deploying your PostgreSQL cluster
- The `name` of the key that is holding the CA **must** be `ca.crt`

There are optional settings for setting up the CA secret:

- You can pass in a certificate revocation list (CRL) for the CA secret by passing in the CRL using the `ca.crl` key name in the Secret.

For example, to create a CA Secret with the trusted CA to use for the PostgreSQL clusters, you could execute the following command:

```
kubectl create secret generic postgresql-ca --from-file=ca.crt=/path/to/ca.crt
```

To create a CA Secret that includes a CRL, you could execute the following command:

```
kubectl create secret generic postgresql-ca \
  --from-file=ca.crt=/path/to/ca.crt \
  --from-file=ca.crl=/path/to/ca.crl
```

Note that you can reuse this CA Secret for other PostgreSQL clusters deployed by the PostgreSQL Operator.

Next, create the Secret that contains your TLS key pair. Create the Secret as a a TLS Secret, and note the following requirement must be met:

- The Secret must be created in the same Namespace as where you are deploying your PostgreSQL cluster

```
kubectl create secret tls hacluster-tls-keypair \
  --cert=/path/to/server.crt \
  --key=/path/to/server.key
```

Now you can create a TLS-enabled PostgreSQL cluster!

Create a TLS Enabled PostgreSQL Cluster

Using the above example, to create a TLS-enabled PostgreSQL cluster that can accept both TLS and non-TLS connections, execute the following command:

```
pgo create cluster hacluster-tls \
  --server-ca-secret=postgresql-ca \
  --server-tls-secret=hacluster-tls-keypair
```

Including the `--server-ca-secret` and `--server-tls-secret` flags automatically enable TLS connections in the PostgreSQL cluster that is deployed. These flags should reference the CA Secret and the TLS key pair Secret, respectively.

If deployed successfully, when you connect to the PostgreSQL cluster, assuming your `PGSSLMODE` is set to `prefer` or higher, you will see something like this in your `psql` terminal:

```
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression:
   off)
```

Force TLS in a PostgreSQL Cluster

There are many environments where you want to force all remote connections to occur over TLS, for example, if you deploy your PostgreSQL cluster's in a public cloud or on an untrusted network. The PostgreSQL Operator lets you force all remote connections to occur over TLS by using the `--tls-only` flag.

For example, using the setup above, you can force TLS in a PostgreSQL cluster by executing the following command:

```
pgo create cluster hacluster-tls-only \
  --tls-only \
  --server-ca-secret=postgresql-ca --server-tls-secret=hacluster-tls-keypair
```

If deployed successfully, when you connect to the PostgreSQL cluster, assuming your `PGSSLMODE` is set to `prefer` or higher, you will see something like this in your `psql` terminal:

```
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression:
   off)
```

If you try to connect to a PostgreSQL cluster that is deployed using the `--tls-only` with TLS disabled (i.e. `PGSSLMODE=disable`), you will receive an error that connections without TLS are unsupported.

TLS Authentication for PostgreSQL Replication

PostgreSQL supports certificate-based authentication, which allows for PostgreSQL to authenticate users based on the common name (CN) in a certificate. Using this feature, the PostgreSQL Operator allows you to configure PostgreSQL replicas in a cluster to authenticate using a certificate instead of a password.

To use this feature, first you will need to set up a Kubernetes TLS Secret that has a CN of `primaryuser`. If you do not wish to have this as your CN, you will need to map the CN of this certificate to the value of `primaryuser` using a pg_ident username map, which you can configure as part of a [custom PostgreSQL configuration]({{< relref "/advanced/custom-configuration.md" >}}).

You also need to ensure that the certificate is verifiable by the certificate authority (CA) chain that you have provided for your PostgreSQL cluster. The CA is provided as part of the `--server-ca-secret` flag in the `pgo create cluster`({{< relref "/pgo-client/reference/pgo_create_cluster.md" >}}) command.

To create a PostgreSQL cluster that uses TLS authentication for replication, first create Kubernetes Secrets for the server and the CA. For the purposes of this example, we will use the ones that were created earlier: `postgresql-ca` and `hacluster-tls-keypair`. After generating a certificate that has a CN of `primaryuser`, create a Kubernetes Secret that references this TLS keypair called `hacluster-tls-replication-keypair`:

```
kubectl create secret tls hacluster-tls-replication-keypair \
  --cert=/path/to/replication.crt \
  --key=/path/to/replication.key
```

We can now create a PostgreSQL cluster and allow for it to use TLS authentication for its replicas! Let's create a PostgreSQL cluster with two replicas that also requires TLS for any connection:

```
pgo create cluster hippo \
  --tls-only \
  --server-ca-secret=postgresql-ca \
  --server-tls-secret=hacluster-tls-keypair \
  --replication-tls-secret=hacluster-tls-replication-keypair \
  --replica-count=2
```

By default, the PostgreSQL Operator has each replica connect to PostgreSQL using a PostgreSQL TLS mode of `verify-ca`. If you wish to perform TLS mutual authentication between PostgreSQL instances (i.e. certificate-based authentication with SSL mode of `verify-full`), you will need to create a [PostgreSQL custom configuration]({{< relref "/advanced/custom-configuration.md" >}}).

Custom PostgreSQL Configuration({{< relref "/advanced/custom-configuration.md" >}})

Customizing PostgreSQL configuration is currently not subject to the `pgo` client, but given it is a common question, we thought it may be helpful to link to how to do it from here. To find out more about how to [customize your PostgreSQL configuration]({{< relref "/advanced/custom-configuration.md" >}}), please refer to the Custom PostgreSQL Configuration({{< relref "/advanced/custom-configuration.md" >}}) section of the documentation.

pgAdmin 4: PostgreSQL Administration

pgAdmin 4 is a popular graphical user interface that lets you work with PostgreSQL databases from both a desktop or web-based client. In the case of the PostgreSQL Operator, the pgAdmin 4 web client can be deployed and synchronized with PostgreSQL clusters so that users can administrate their databases with their PostgreSQL username and password.

For example, let's work with a PostgreSQL cluster called `hippo` that has a user named `hippo` with password `datalake`, e.g.:

```
pgo create cluster hippo --username=hippo --password=datalake
```

Once the `hippo` PostgreSQL cluster is ready, create the pgAdmin 4 deployment with the [`pgo create pgadmin`]({{< relref "/pgo-client/reference/pgo_create_pgadmin.md" >}}) command:

```
pgo create pgadmin hippo
```

This creates a pgAdmin 4 deployment unique to this PostgreSQL cluster and synchronizes the PostgreSQL user information into it. To access pgAdmin 4, you can set up a port-forward to the Service, which follows the pattern `<clusterName>-pgadmin`, to port `5050`:

```
kubectl port-forward svc/hippo-pgadmin 5050:5050
```

Point your browser at `http://localhost:5050` and use your database username (e.g. `hippo`) and password (e.g. `datalake`) to log in.

Figure 28: pgAdmin 4 Login Page

(Note: if your password does not appear to work, you can retry setting up the user with the [`pgo update user`]({{< relref "/pgo-client/reference/pgo_update_user.md" >}}) command: `pgo update user hippo --password=datalake`)

The `pgo create user`, `pgo update user`, and `pgo delete user` commands are synchronized with the pgAdmin 4 deployment. Any user with credentials to this PostgreSQL cluster will be able to log in and use pgAdmin 4:

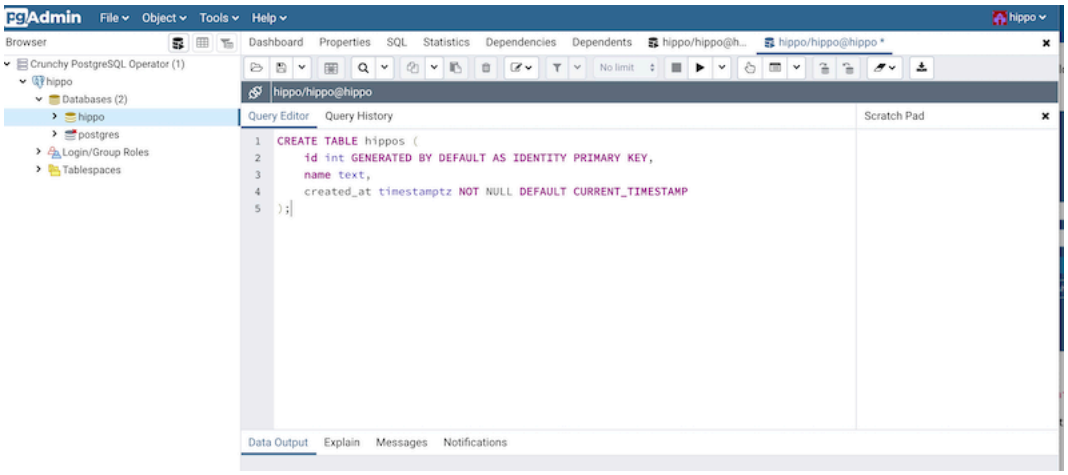

Figure 29: pgAdmin 4 Query

You can remove the pgAdmin 4 deployment with the [`pgo delete pgadmin`]({{< relref "/pgo-client/reference/pgo_delete_pgadmin.md" >}}) command.

For more information, please read the [pgAdmin 4 Architecture]({{< relref "/architecture/pgadmin4.md" >}}) section of the documentation.

Standby Clusters: Multi-Cluster Kubernetes Deployments

A [standby PostgreSQL cluster]({{< relref "/architecture/high-availability/multi-cluster-kubernetes.md" >}}) can be used to create an advanced high-availability set with a PostgreSQL cluster running in a different Kubernetes cluster, or used for other operations such as migrating from one PostgreSQL cluster to another. Note: this is not [high availability]({{< relref "/architecture/high-availability/_index.md" >}}) per se: a high-availability PostgreSQL cluster will automatically fail over upon a downtime event, whereas a standby PostgreSQL cluster must be explicitly promoted.

With that said, you can run multiple PostgreSQL Operators in different Kubernetes clusters, and the below functionality will work!

Below are some commands for setting up and using standby PostgreSQL clusters. For more details on how standby clusters work, please review the section on [Kubernetes Multi-Cluster Deployments]({{< relref "/architecture/high-availability/multi-cluster-kubernetes.md" >}}).

Creating a Standby Cluster

Before creating a standby cluster, you will need to ensure that your primary cluster is created properly. Standby clusters require the use of S3 or equivalent S3-compatible storage system that is accessible to both the primary and standby clusters. For example, to create a primary cluster to these specifications:

```
shell pgo create cluster hippo --pgbouncer --replica-count=2 \ --pgbackrest-storage-type=posix,s3 \ --pgbackrest-s3-
\ --pgbackrest-s3-key-secret=<redacted> \ --pgbackrest-s3-bucket=watering-hole \ --pgbackrest-s3-endpoint=s3.amazona
\ --pgbackrest-s3-region=us-east-1 \ --pgbackrest-s3-uri-style=host \ --pgbackrest-s3-verify-tls=true \ --password-s
\ --password-replication=somewhatsecrethippo \ --password=opensourcehippo
```

Before setting up the standby PostgreSQL cluster, you will need to wait a few moments for the primary PostgreSQL cluster to be ready. Once your primary PostgreSQL cluster is available, you can create a standby cluster by using the following command:

```
pgo create cluster hippo-standby --standby --replica-count=2 \
  --pgbackrest-storage-type=s3 \
  --pgbackrest-s3-key=<redacted> \
  --pgbackrest-s3-key-secret=<redacted> \
  --pgbackrest-s3-bucket=watering-hole \
  --pgbackrest-s3-endpoint=s3.amazonaws.com \
  --pgbackrest-s3-region=us-east-1 \
  --pgbackrest-s3-uri-style=host \
  --pgbackrest-s3-verify-tls=true \
  --pgbackrest-repo-path=/backrestrepo/hippo-backrest-shared-repo \
  --password-superuser=supersecrethippo \
  --password-replication=somewhatsecrethippo \
  --password=opensourcehippo
```

If you are unsure of your user credentials form the original `hippo` cluster, you can retrieve them using the [`pgo show user`]({{< relref "/pgo-client/reference/pgo_show_user.md" >}}) command with the `--show-system-accounts` flag:

```
pgo show user hippo --show-system-accounts
```

The standby cluster will take a few moments to bootstrap, but it is now set up!

Promoting a Standby Cluster

Before promoting a standby cluster, it is first necessary to shut down the primary cluster, otherwise you can run into a potential "split-brain" scenario (if your primary Kubernetes cluster is down, it may not be possible to do this).

To shutdown, run the following command:

```
pgo update cluster hippo --shutdown
```

Once it is shut down, you can promote the standby cluster:

```
pgo update cluster hippo-standby --promote-standby
```

The standby is now an active PostgreSQL cluster and can start to accept writes.

To convert the previous active cluster into a standby cluster, you can run the following command:

```
pgo update cluster hippo --enable-standby
```

This will take a few moments to make this PostgreSQL cluster into a standby cluster. When it is ready, you can start it up with the following command:

```
pgo update cluster hippo --startup
```

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
```

Custom Annotations

There are a variety of reasons why one may want to add additional Annotations to the Deployments, and by extension Pods, managed by the PostgreSQL Operator:

- External applications that extend functionality via details in an annotation
- Tracking purposes for an internal application

etc.

As such the `pgo` client allows you to manage your own custom annotations on the Operator. There are four different ways to add annotations:

- On PostgreSQL instances
- On pgBackRest instances
- On pgBouncer instances
- On all of the above

The custom annotation feature follows the same syntax as Kubernetes for adding and removing annotations, e.g.:

`--annotation=name=value`

would add an annotation called `name` with a value of `value`, and:

`--annotation=name-`

would remove an annotation called `name`

Adding an Annotation

There are two ways to add an Annotation during the lifecycle of a PostgreSQL cluster:

- Cluster creation: (`pgo create cluster`({{< relref "/pgo-client/reference/pgo_create_cluster.md" >}}))
- Updating a cluster: ([`pgo update cluster`]({{< relref "/pgo-client/reference/pgo_update_cluster.md" >}}))

There are several flags available for managing Annotations, i.e.:

- `--annotation`: adds an Annotation to all managed Deployments (PostgreSQL, pgBackRest, pgBouncer)
- `--annotation-postgres`: adds an Annotation only to PostgreSQL Deployments
- `--annotation-pgbackrest`: adds an Annotation only to pgBackrest Deployments
- `--annotation-pgbouncer`: adds an Annotation only to pgBouncer Deployments

To add an Annotation with key `hippo` and value `awesome` to all of the managed Deployments when creating a cluster, you would run the following command:

```
pgo create cluster hippo --annotation=hippo=awesome
```

To add an Annotation with key `elephant` and value `cool` to only the PostgreSQL Deployments when creating a cluster, you would run the following command:

```
pgo create cluster hippo --annotation-postgres=elephant=cool
```

To add an Annotation to all the managed Deployments in an existing cluster, you can use the `pgo update cluster` command:

```
pgo update cluster hippo --annotation=zebra=nice
```

Adding Multiple Annotations

There are two syntaxes you could use to add multiple Annotations to a cluster:

```
pgo create cluster hippo --annotation=hippo=awesome,elephant=cool
```

or

```
pgo create cluster hippo --annotation=hippo=awesome --annotation=elephant=cool
```

Updating Annotations

To update an Annotation, you can use the [`pgo update cluster`]({{< relref "/pgo-client/reference/pgo_update_cluster.md" >}}) command and reference the original Annotation key. For intance, if I wanted to update the `hippo` annotation to be `rad`:

```
pgo update cluster hippo --annotation=hippo=rad
```

Removing Annotations

To remove an Annotation, you need to add a `-` to the end of the Annotation name. For example, to remove the `hippo` annotation:

```
pgo update cluster hippo --annotation=hippo-
```

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

Please see the [tutorial on pgBouncer]({{< relref "tutorial/pgbouncer.md" >}}).

Query Analysis via pgBadger

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

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

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.

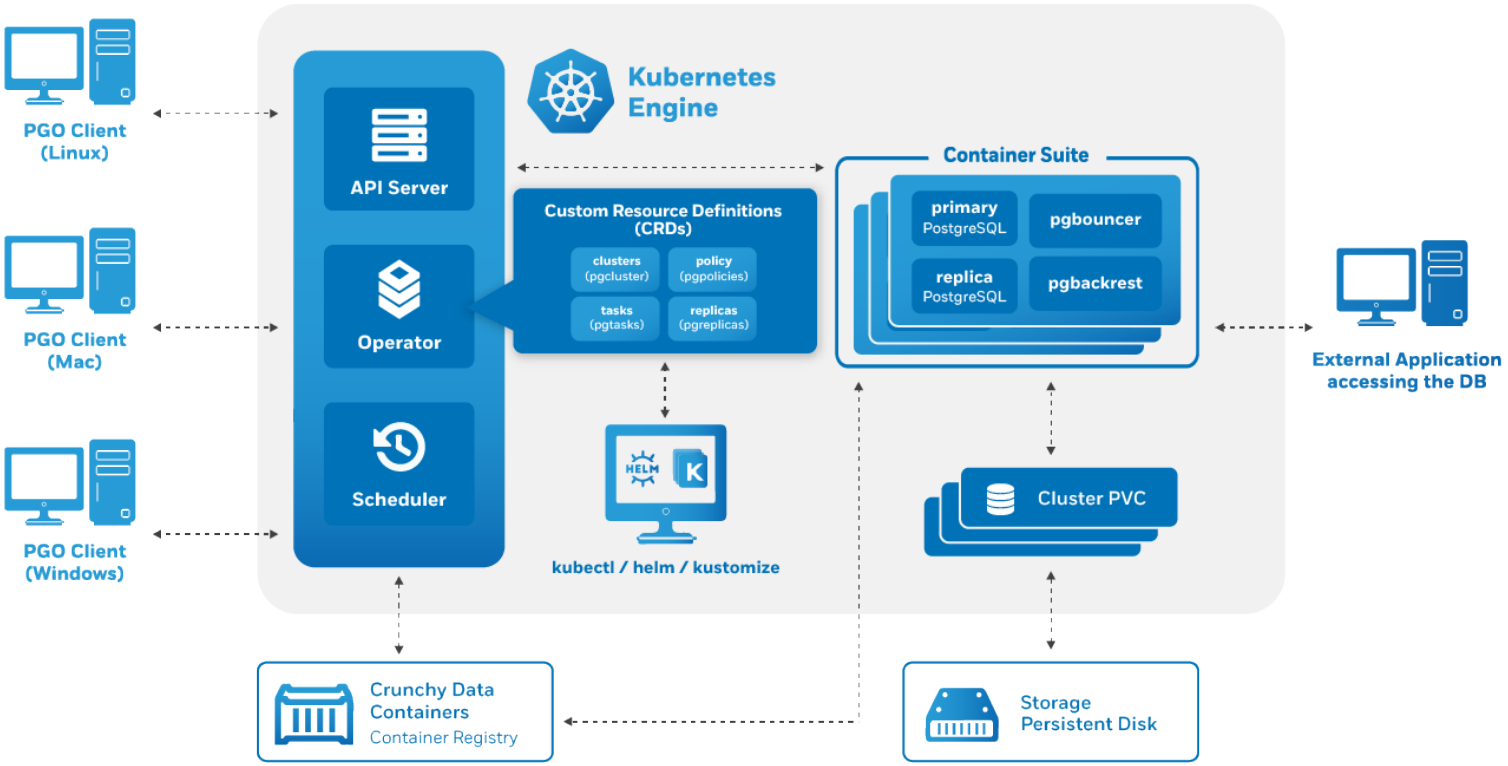

Figure 30: Operator Architecture with CRDs

As discussed in the [architecture overview]({{< relref "/architecture/overview.md" >}}), the heart of the [PostgreSQL Operator]({{< relref "_index.md" >}}), and any Kubernetes Operator, is one or more Custom Resources Definitions, also known as "CRDs". CRDs provide extensions to the Kubernetes API, and, in the case of the PostgreSQL Operator, allow you to perform actions such as:

- Creating a PostgreSQL Cluster
- Updating PostgreSQL Cluster resource allocations
- Add additional utilities to a PostgreSQL cluster, e.g. pgBouncer({{< relref "/pgo-client/reference/pgo_create_pgbouncer.md" >}}) for connection pooling and more.

The PostgreSQL Operator provides the [pgo client]({{< relref "/pgo-client/_index.md" >}}) as a convenience for interfacing with the CRDs, as manipulating the CRDs directly can be a tedious process. For example, there are several Kubernetes objects that need to be set up prior to creating a `pgcluster` custom resource in order to successfully deploy a new PostgreSQL cluster.

The Kubernetes community trend has been to move towards supporting a "custom resource only" workflow for using Operators, and this is something that the PostgreSQL Operator aims to do as well. Certain workflows are fully driven by Custom Resources (e.g. creating a PostgreSQL cluster), while others still need to interface through the [pgo client]({{< relref "/pgo-client/_index.md" >}}) (e.g. adding a PostgreSQL user).

The following sections will describe the functionality that is available today when manipulating the PostgreSQL Operator Custom Resources directly.

Custom Resource Workflows

Create a PostgreSQL Cluster

The fundamental workflow for interfacing with a PostgreSQL Operator Custom Resource Definition is for creating a PostgreSQL cluster. There are several that a PostgreSQL cluster requires to be deployed, including:

- Secrets
- Information for setting up a pgBackRest repository
- PostgreSQL superuser bootstrap credentials
- PostgreSQL replication user bootstrap credentials
- PostgresQL standard user bootstrap credentials

Additionally, if you want to add some of the other sidecars, you may need to create additional secrets.

The good news is that if you do not provide these objects, the PostgreSQL Operator will create them for you to get your Postgres cluster up and running!

The following goes through how to create a PostgreSQL cluster called `hippo` by creating a new custom resource.

```
# this variable is the name of the cluster being created
export pgo_cluster_name=hippo
# this variable is the namespace the cluster is being deployed into
export cluster_namespace=pgo

cat <<-EOF > "${pgo_cluster_name}-pgcluster.yaml"
apiVersion: crunchydata.com/v1
kind: Pgcluster
metadata:
  annotations:
    current-primary: ${pgo_cluster_name}
  labels:
    crunchy-pgha-scope: ${pgo_cluster_name}
    deployment-name: ${pgo_cluster_name}
    name: ${pgo_cluster_name}
    pg-cluster: ${pgo_cluster_name}
    pgo-version: {{< param operatorVersion >}}
    pgouser: admin
  name: ${pgo_cluster_name}
  namespace: ${cluster_namespace}
spec:
  BackrestStorage:
    accessmode: ReadWriteMany
    matchLabels: ""
    name: ""
    size: 1G
    storageclass: ""
    storagetype: dynamic
    supplementalgroups: ""
  PrimaryStorage:
    accessmode: ReadWriteMany
    matchLabels: ""
    name: ${pgo_cluster_name}
    size: 1G
```

```
    storageclass: ""
    storagetype: dynamic
    supplementalgroups: ""
  ReplicaStorage:
    accessmode: ReadWriteMany
    matchLabels: ""
    name: ""
    size: 1G
    storageclass: ""
    storagetype: dynamic
    supplementalgroups: ""
  annotations: {}
  ccpimage: crunchy-postgres-ha
  ccpimageprefix: registry.developers.crunchydata.com/crunchydata
  ccpimagetag: {{< param centosBase >}}-{{< param postgresVersion >}}-{{< param operatorVersion >}}
  clustername: ${pgo_cluster_name}
  database: ${pgo_cluster_name}
  exporterport: "9187"
  limits: {}
  name: ${pgo_cluster_name}
  namespace: ${cluster_namespace}
  pgDataSource:
    restoreFrom: ""
    restoreOpts: ""
  pgbadgerport: "10000"
  pgoimageprefix: registry.developers.crunchydata.com/crunchydata
  podAntiAffinity:
    default: preferred
    pgBackRest: preferred
    pgBouncer: preferred
  port: "5432"
  tolerations: []
  user: hippo
  userlabels:
    pgo-version: {{< param operatorVersion >}}
EOF

kubectl apply -f "${pgo_cluster_name}-pgcluster.yaml"
```

And that's all! The PostgreSQL Operator will go ahead and create the cluster.

As part of this process, the PostgreSQL Operator creates several Secrets that contain the credentials for three user accounts that must be present in order to bootstrap a PostgreSQL cluster. These are:

- A PostgreSQL superuser
- A replication user
- A standard PostgreSQL user

The Secrets represent the following PostgreSQL users and can be identified using the below patterns:

PostgreSQL User	Type	Secret Pattern	Notes
`postgres`	Superuser	`<clusterName>-postgres-secret`	This is the PostgreSQL superuser account. Using the above exa
`primaryuser`	Replication	`<clusterName>-primaryuser-secret`	This is for the managed replication user account for maintainin
User	User	`<clusterName>-<User>-secret`	This is an unprivileged user that should be used for most opera

To extract the user credentials so you can log into the database, you can use the following JSONPath expression:

```
# namespace that the cluster is running in
export cluster_namespace=pgo
# name of the cluster
export pgo_cluster_name=hippo
# name of the user whose password we want to get
```

```
export pgo_cluster_username=hippo

kubectl -n "${cluster_namespace}" get secrets \
  "${pgo_cluster_name}-${pgo_cluster_username}-secret" -o "jsonpath={.data['password']}" | base64
     -d
```

Customizing User Credentials If you wish to set the credentials for these users on your own, you have to create these Secrets *before* creating a custom resource. The below example shows how to create the three required user accounts prior to creating a custom resource. Note that if you omit any of these Secrets, the Postgres Operator will create it on its own.

```
# this variable is the name of the cluster being created
pgo_cluster_name=hippo
# this variable is the namespace the cluster is being deployed into
cluster_namespace=pgo

# this is the superuser secret
kubectl create secret generic -n "${cluster_namespace}" "${pgo_cluster_name}-postgres-secret" \
  --from-literal=username=postgres \
  --from-literal=password=Supersecurepassword*

# this is the replication user secret
kubectl create secret generic -n "${cluster_namespace}" "${pgo_cluster_name}-primaryuser-secret" \
  --from-literal=username=primaryuser \
  --from-literal=password=Anothersecurepassword*

# this is the standard user secret
kubectl create secret generic -n "${cluster_namespace}" "${pgo_cluster_name}-hippo-secret" \
  --from-literal=username=hippo \
  --from-literal=password=Moresecurepassword*


kubectl label secrets -n "${cluster_namespace}" "${pgo_cluster_name}-postgres-secret"
   "pg-cluster=${pgo_cluster_name}"
kubectl label secrets -n "${cluster_namespace}" "${pgo_cluster_name}-primaryuser-secret"
   "pg-cluster=${pgo_cluster_name}"
kubectl label secrets -n "${cluster_namespace}" "${pgo_cluster_name}-hippo-secret"
   "pg-cluster=${pgo_cluster_name}"
```

Create a PostgreSQL Cluster With Backups in S3

A frequent use case is to create a PostgreSQL cluster with S3 or a S3-like storage system for storing backups. This requires adding a Secret that contains the S3 key and key secret for your account, and adding some additional information into the custom resource.

Step 1: Create the pgBackRest S3 Secrets As mentioned above, it is necessary to create a Secret containing the S3 key and key secret that will allow a user to create backups in S3.

The below code will help you set up this Secret.

```
# this variable is the name of the cluster being created
pgo_cluster_name=hippo
# this variable is the namespace the cluster is being deployed into
cluster_namespace=pgo
# the following variables are your S3 key and key secret
backrest_s3_key=yours3key
backrest_s3_key_secret=yours3keysecret

kubectl -n "${cluster_namespace}" create secret generic "${pgo_cluster_name}-backrest-repo-config"
   \
  --from-literal="aws-s3-key=${backrest_s3_key}" \
  --from-literal="aws-s3-key-secret=${backrest_s3_key_secret}"

unset backrest_s3_key
unset backrest_s3_key_secret
```

Step 2: Create the PostgreSQL Cluster With the Secrets in place. It is now time to create the PostgreSQL cluster.

The below manifest references the Secrets created in the previous step to add a custom resource to the `pgclusters.crunchydata.com` custom resource definition. There are some additions in this example specifically for storing backups in S3.

```
# this variable is the name of the cluster being created
export pgo_cluster_name=hippo
# this variable is the namespace the cluster is being deployed into
export cluster_namespace=pgo
# the following variables store the information for your S3 cluster. You may
# need to adjust them for your actual settings
export backrest_s3_bucket=your-bucket
export backrest_s3_endpoint=s3.region-name.amazonaws.com
export backrest_s3_region=region-name

cat <<-EOF > "${pgo_cluster_name}-pgcluster.yaml"
apiVersion: crunchydata.com/v1
kind: Pgcluster
metadata:
  annotations:
    current-primary: ${pgo_cluster_name}
  labels:
    crunchy-pgha-scope: ${pgo_cluster_name}
    deployment-name: ${pgo_cluster_name}
    name: ${pgo_cluster_name}
    pg-cluster: ${pgo_cluster_name}
    pgo-version: {{< param operatorVersion >}}
    pgouser: admin
  name: ${pgo_cluster_name}
  namespace: ${cluster_namespace}
spec:
  BackrestStorage:
    accessmode: ReadWriteMany
    matchLabels: ""
    name: ""
    size: 1G
    storageclass: ""
    storagetype: dynamic
    supplementalgroups: ""
  PrimaryStorage:
    accessmode: ReadWriteMany
    matchLabels: ""
    name: ${pgo_cluster_name}
    size: 1G
    storageclass: ""
    storagetype: dynamic
    supplementalgroups: ""
  ReplicaStorage:
    accessmode: ReadWriteMany
    matchLabels: ""
    name: ""
    size: 1G
    storageclass: ""
    storagetype: dynamic
    supplementalgroups: ""
  annotations: {}
  backrestStorageTypes:
  - s3
  backrestS3Bucket: ${backrest_s3_bucket}
  backrestS3Endpoint: ${backrest_s3_endpoint}
  backrestS3Region: ${backrest_s3_region}
  backrestS3URIStyle: ""
  backrestS3VerifyTLS: ""
  ccpimage: crunchy-postgres-ha
  ccpimageprefix: registry.developers.crunchydata.com/crunchydata
  ccpimagetag: {{< param centosBase >}}-{{< param postgresVersion >}}-{{< param operatorVersion >}}
```

```
  clustername: ${pgo_cluster_name}
  database: ${pgo_cluster_name}
  exporterport: "9187"
  limits: {}
  name: ${pgo_cluster_name}
  namespace: ${cluster_namespace}
  pgDataSource:
    restoreFrom: ""
    restoreOpts: ""
  pgbadgerport: "10000"
  pgoimageprefix: registry.developers.crunchydata.com/crunchydata
  podAntiAffinity:
    default: preferred
    pgBackRest: preferred
    pgBouncer: preferred
  port: "5432"
  tolerations: []
  user: hippo
  userlabels:
    pgo-version: {{< param operatorVersion >}}
EOF

kubectl apply -f "${pgo_cluster_name}-pgcluster.yaml"
```

Create a PostgreSQL Cluster with TLS

There are three items that are required to enable TLS in your PostgreSQL clusters:

- A CA certificate
- A TLS private key
- A TLS certificate

It is possible [create a PostgreSQL cluster with TLS]({{< relref "tutorial/tls.md" >}}) using a custom resource workflow with the prerequisite of ensuring the above three items are created.

For a detailed explanation for how TLS works with the PostgreSQL Operator, please see the [TLS tutorial]({{< relref "tutorial/tls.md" >}}).

Step 1: Create TLS Secrets There are two Secrets that need to be created:

1. A Secret containing the certificate authority (CA). You may only need to create this Secret once, as a CA certificate can be shared amongst your clusters.
2. A Secret that contains the TLS private key & certificate.

This assumes that you have already generated your TLS certificates where the CA is named `ca.crt` and the server key and certificate are named `server.key` and `server.crt` respectively.

Substitute the correct values for your environment into the environmental variables in the example below:

```
# this variable is the name of the cluster being created
export pgo_cluster_name=hippo
# this variable is the namespace the cluster is being deployed into
export cluster_namespace=pgo
# this is the local path to where you stored the CA and server key and certificate
export cluster_tls_asset_path=/path/to

# create the CA secret. if this already exists, it's OK if it fails
kubectl create secret generic postgresql-ca -n "${cluster_namespace}" \
  --from-file="ca.crt=${cluster_tls_asset_path}/ca.crt"

# create the server key/certificate secret
kubectl create secret tls "${pgo_cluster_name}-tls-keypair" -n "${cluster_namespace}" \
  --cert="${cluster_tls_asset_path}/server.crt" \
  --key="${cluster_tls_asset_path}/server.key"
```

Step 2: Create the PostgreSQL Cluster The below example uses the Secrets created in the previous step and creates a TLS-enabled PostgreSQL cluster. Additionally, this example sets the `tlsOnly` attribute to `true`, which requires all TCP connections to occur over TLS:

```
# this variable is the name of the cluster being created
export pgo_cluster_name=hippo
# this variable is the namespace the cluster is being deployed into
export cluster_namespace=pgo

cat <<-EOF > "${pgo_cluster_name}-pgcluster.yaml"
apiVersion: crunchydata.com/v1
kind: Pgcluster
metadata:
  annotations:
    current-primary: ${pgo_cluster_name}
  labels:
    crunchy-pgha-scope: ${pgo_cluster_name}
    deployment-name: ${pgo_cluster_name}
    name: ${pgo_cluster_name}
    pg-cluster: ${pgo_cluster_name}
    pgo-version: {{< param operatorVersion >}}
    pgouser: admin
  name: ${pgo_cluster_name}
  namespace: ${cluster_namespace}
spec:
  BackrestStorage:
    accessmode: ReadWriteMany
    matchLabels: ""
    name: ""
    size: 1G
    storageclass: ""
    storagetype: dynamic
    supplementalgroups: ""
  PrimaryStorage:
    accessmode: ReadWriteMany
    matchLabels: ""
    name: ${pgo_cluster_name}
    size: 1G
    storageclass: ""
    storagetype: dynamic
    supplementalgroups: ""
  ReplicaStorage:
    accessmode: ReadWriteMany
    matchLabels: ""
    name: ""
    size: 1G
    storageclass: ""
    storagetype: dynamic
    supplementalgroups: ""
  annotations: {}
  ccpimage: crunchy-postgres-ha
  ccpimageprefix: registry.developers.crunchydata.com/crunchydata
  ccpimagetag: {{< param centosBase >}}-{{< param postgresVersion >}}-{{< param operatorVersion >}}
  clustername: ${pgo_cluster_name}
  database: ${pgo_cluster_name}
  exporterport: "9187"
  limits: {}
  name: ${pgo_cluster_name}
  namespace: ${cluster_namespace}
  pgDataSource:
    restoreFrom: ""
    restoreOpts: ""
  pgbadgerport: "10000"
  pgoimageprefix: registry.developers.crunchydata.com/crunchydata
  podAntiAffinity:
    default: preferred
```

```
    pgBackRest: preferred
    pgBouncer: preferred
  port: "5432"
  tls:
    caSecret: postgresql-ca
    tlsSecret: ${pgo_cluster_name}-tls-keypair
  tlsOnly: true
  user: hippo
  userlabels:
    pgo-version: {{< param operatorVersion >}}
EOF

kubectl apply -f "${pgo_cluster_name}-pgcluster.yaml"
```

Modify a Cluster

There following modification operations are supported on the `pgclusters.crunchydata.com` custom resource definition:

Modify Resource Requests & Limits Modifying the `resources`, `limits`, `backrestResources`, `backRestLimits`, `pgBouncer.resource` or `pgbouncer.limits` will cause the PostgreSQL Operator to apply the new values to the affected Deployments.

For example, if we wanted to make a memory request of 512Mi for the `hippo` cluster created in the previous example, we could do the following:

```
# this variable is the name of the cluster being created
export pgo_cluster_name=hippo
# this variable is the namespace the cluster is being deployed into
export cluster_namespace=pgo

kubectl edit pgclusters.crunchydata.com -n "${cluster_namespace}" "${pgo_cluster_name}"
```

This will open up your editor. Find the `resources` block, and have it read as the following:

```
resources:
  memory: 256Mi
```

The PostgreSQL Operator will respond and modify the PostgreSQL instances to request 256Mi of memory.

Be careful when editing these values for a variety of reasons, mainly that modifying these values will cause the Pods to restart, which in turn will create potential downtime events. It's best to modify the values for a deployment group together and not mix and match, i.e.

- PostgreSQL instances: `resources`, `limits`
- pgBackRest: `backrestResources`, `backrestLimits`
- pgBouncer: `pgBouncer.resources`, `pgBouncer.limits`

Scale

Once you have created a PostgreSQL cluster, you may want to add a replica to create a high-availability environment. Replicas are added and removed using the `pgreplicas.crunchydata.com` custom resource definition. Each replica must have a unique name, e.g. `hippo-rpl1` could be one unique replica for a PostgreSQL cluster.

Using the above example cluster, `hippo`, let's add a replica called `hippo-rpl1` using the configuration below. Be sure to change the `replicastorage` block to match the storage configuration for your environment:

```
# this variable is the name of the cluster being created
export pgo_cluster_name=hippo
# this helps to name the replica, in this case "rpl1"
export pgo_cluster_replica_suffix=rpl1
# this variable is the namespace the cluster is being deployed into
export cluster_namespace=pgo

cat <<-EOF > "${pgo_cluster_name}-${pgo_cluster_replica_suffix}-pgreplica.yaml"
apiVersion: crunchydata.com/v1
kind: Pgreplica
metadata:
```

```
  labels:
    name: ${pgo_cluster_name}-${pgo_cluster_replica_suffix}
    pg-cluster: ${pgo_cluster_name}
    pgouser: admin
  name: ${pgo_cluster_name}-${pgo_cluster_replica_suffix}
  namespace: ${cluster_namespace}
spec:
  clustername: ${pgo_cluster_name}
  name: ${pgo_cluster_name}-${pgo_cluster_replica_suffix}
  namespace: ${cluster_namespace}
  replicastorage:
    accessmode: ReadWriteMany
    matchLabels: ""
    name: ${pgo_cluster_name}-${pgo_cluster_replica_suffix}
    size: 1G
    storageclass: ""
    storagetype: dynamic
    supplementalgroups: ""
  tolerations: []
  userlabels:
    pgo-version: {{< param operatorVersion >}}
EOF

kubectl apply -f "${pgo_cluster_name}-${pgo_cluster_replica_suffix}-pgreplica.yaml"
```

Add this time, removing a replica must be handled through the [pgo client]({{< relref "/pgo-client/common-tasks.md#high-availability-scaling-up-down">}}).

Monitoring

To enable the [monitoring]({{< relref "/architecture/monitoring.md">}}) (aka metrics) sidecar using the `crunchy-postgres-exporter` container, you need to set the `exporter` attribute in `pgclusters.crunchydata.com` custom resource.

Add a Tablespace

Tablespaces can be added during the lifetime of a PostgreSQL cluster (tablespaces can be removed as well, but for a detailed explanation as to how, please see the Tablespaces({{< relref "/architecture/tablespaces.md">}}) section).

To add a tablespace, you need to add an entry to the `tablespaceMounts` section of a custom entry, where the key is the name of the tablespace (unique to the `pgclusters.crunchydata.com` custom resource entry) and the value is a storage configuration as defined in the `pgclusters.crunchydata.com` section above.

For example, to add a tablespace named `lake` to our `hippo` cluster, we can open up the editor with the following code:

```
# this variable is the name of the cluster being created
export pgo_cluster_name=hippo
# this variable is the namespace the cluster is being deployed into
export cluster_namespace=pgo

kubectl edit pgclusters.crunchydata.com -n "${cluster_namespace}" "${pgo_cluster_name}"
```

and add an entry to the `tablespaceMounts` block that looks similar to this, with the addition of the correct storage configuration for your environment:

```
tablespaceMounts:
  lake:
    accessmode: ReadWriteMany
    matchLabels: ""
    size: 5Gi
    storageclass: ""
    storagetype: dynamic
    supplementalgroups: ""
```

pgBouncer

pgBouncer is a PostgreSQL connection pooler and state manager that can be useful for high-availability setups as well as managing overall performance of a PostgreSQL cluster. A pgBouncer deployment for a PostgreSQL cluster can be fully managed from a `pgclusters.crunchydata.com` custom resource.

For example, to add a pgBouncer deployment to our `hippo` cluster with two instances and a memory limit of 36Mi, you can edit the custom resource:

```
# this variable is the name of the cluster being created
export pgo_cluster_name=hippo
# this variable is the namespace the cluster is being deployed into
export cluster_namespace=pgo

kubectl edit pgclusters.crunchydata.com -n "${cluster_namespace}" "${pgo_cluster_name}"
```

And modify the `pgBouncer` block to look like this:

```
pgBouncer:
  limits:
    memory: 36Mi
  replicas: 2
```

Likewise, to remove pgBouncer from a PostgreSQL cluster, you would set `replicas` to `0`:

```
pgBouncer:
  replicas: 0
```

Start / Stop a Cluster

A PostgreSQL cluster can be start and stopped by toggling the `shutdown` parameter in a `pgclusters.crunchydata.com` custom resource. Setting `shutdown` to `true` will stop a PostgreSQL cluster, whereas a value of `false` will make a cluster available. This affects all of the associated instances of a PostgreSQL cluster.

Manage Annotations

Kubernetes Annotations can be managed for PostgreSQL, pgBackRest, and pgBouncer Deployments, as well as being able to apply Annotations across all three. This is done via the `annotations` block in the `pgclusters.crunchydata.com` custom resource definition. For example, to apply Annotations in the `hippo` cluster, some that are global, some that are specific to each Deployment type, you could do the following.

First, start editing the `hippo` custom resource:

```
# this variable is the name of the cluster being created
export pgo_cluster_name=hippo
# this variable is the namespace the cluster is being deployed into
export cluster_namespace=pgo

kubectl edit pgclusters.crunchydata.com -n "${cluster_namespace}" "${pgo_cluster_name}"
```

In the `hippo` specification, add the annotations block similar to this (note, this explicitly shows that this is the `spec` block. **Do not modify the `annotations` block in the `metadata` section**).

```
spec:
  annotations:
    global:
      favorite: hippo
    backrest:
      chair: comfy
    pgBouncer:
      pool: swimming
    postgres:
      elephant: cool
```

Save your edits, and in a short period of time, you should see these annotations applied to the managed Deployments.

Delete a PostgreSQL Cluster

A PostgreSQL cluster can be deleted by simply deleting the `pgclusters.crunchydata.com` resource.

It is possible to keep both the PostgreSQL data directory as well as the pgBackRest backup repository when using this method by setting the following annotations on the `pgclusters.crunchydata.com` custom resource:

- `keep-backups`: indicates to keep the pgBackRest PVC when deleting the cluster.
- `keep-data`: indicates to keep the PostgreSQL data PVC when deleting the cluster.

PostgreSQL Operator Custom Resource Definitions

There are several PostgreSQL Operator Custom Resource Definitions (CRDs) that are installed in order for the PostgreSQL Operator to successfully function:

- `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. take a backup) and tracks the state of said task through 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.

Below takes an in depth look for what each attribute does in a Custom Resource Definition, and how they can be used in the creation and update workflow.

Glossary

- `create`: if an attribute is listed as `create`, it means it can affect what happens when a new Custom Resource is created.
- `update`: if an attribute is listed as `update`, it means it can affect the Custom Resource, and by extension the objects it manages, when the attribute is updated.

`pgclusters.crunchydata.com`

The `pgclusters.crunchydata.com` Custom Resource Definition is the fundamental definition of a PostgreSQL cluster. Most attributes only affect the deployment of a PostgreSQL cluster at the time the PostgreSQL cluster is created. Some attributes can be modified during the lifetime of the PostgreSQL cluster and make changes, as described below.

Attribute	Action	Description
annotations	**Specification (Spec)**	Specify Kubernetes Annotations that can be applied to the different deployments managed by t
backrestConfig	`create`	Optional references to pgBackRest configuration files
backrestLimits	`create`, `update`	Specify the container resource limits that the pgBackRest repository should use. Follows the Ku
backrestRepoPath	`create`	Optional reference to the location of the pgBackRest repository.
BackrestResources	`create`, `update`	Specify the container resource requests that the pgBackRest repository should use. Follows the
backrestS3Bucket	`create`	An optional parameter that specifies a S3 bucket that pgBackRest should use.
backrestS3Endpoint	`create`	An optional parameter that specifies the S3 endpoint pgBackRest should use.
backrestS3Region	`create`	An optional parameter that specifies a cloud region that pgBackRest should use.
backrestS3URIStyle	`create`	An optional parameter that specifies if pgBackRest should use the `path` or `host` S3 URI style.
backrestS3VerifyTLS	`create`	An optional parameter that specifies if pgBackRest should verify the TLS endpoint.
BackrestStorage	`create`	A specification that gives information about the storage attributes for the pgBackRest repositor
backrestStorageTypes	`create`	An optional parameter that takes an array of different repositories types that can be used to st
ccpimage	`create`	The name of the PostgreSQL container image to use, e.g. `crunchy-postgres-ha` or `crunchy-p`

Attribute	Action	Description
ccpimageprefix	`create`	If provided, the image prefix (or registry) of the PostgreSQL container image, e.g. `registry.de`
ccpimagetag	`create`	The tag of the PostgreSQL container image to use, e.g. `{{< param centosBase >}}-{{< para`
clustername	`create`	The name of the PostgreSQL cluster, e.g. `hippo`. This is used to group PostgreSQL instances (
customconfig	`create`	If specified, references a custom ConfigMap to use when bootstrapping a PostgreSQL cluster. F
database	`create`	The name of a database that the PostgreSQL user can log into after the PostgreSQL cluster is
disableAutofail	`create`, `update`	If set to true, disables the high availability capabilities of a PostgreSQL cluster. By default, eve
exporter	`create`,`update`	If `true`, deploys the `crunchy-postgres-exporter` sidecar for metrics collection
exporterLimits	`create`, `update`	Specify the container resource limits that the `crunchy-postgres-exporter` sidecar uses when i
exporterport	`create`	If `Exporter` is `true`, then this specifies the port that the metrics sidecar runs on (e.g. `9187`)
exporterResources	`create`, `update`	Specify the container resource requests that the `crunchy-postgres-exporter` sidecar uses whe
limits	`create`, `update`	Specify the container resource limits that the PostgreSQL cluster should use. Follows the Kube
name	`create`	The name of the PostgreSQL instance that is the primary. On creation, this should be set to be
namespace	`create`	The Kubernetes Namespace that the PostgreSQL cluster is deployed in.
nodeAffinity	`create`	Sets the node affinity rules for the PostgreSQL cluster and associated PostgreSQL instances. Ca
pgBadger	`create`,`update`	If `true`, deploys the `crunchy-pgbadger` sidecar for query analysis.
pgbadgerport	`create`	If the `PGBadger` label is set, then this specifies the port that the pgBadger sidecar runs on (e.g.
pgBouncer	`create`, `update`	If specified, defines the attributes to use for the pgBouncer connection pooling deployment that
pgDataSource	`create`	Used to indicate if a PostgreSQL cluster should bootstrap its data from a pgBackRest repositor
pgoimageprefix	`create`	If provided, the image prefix (or registry) of any PostgreSQL Operator images that are used for
podAntiAffinity	`create`	A required section. Sets the [pod anti-affinity rules]({{< relref "/architecture/high-availability/

Storage Specification The storage specification is a spec that defines attributes about the storage to be used for a particular function of a PostgreSQL cluster (e.g. a primary instance or for the pgBackRest backup repository). The below describes each attribute and how it works.

Attribute	Action	Description
accessmode	`create`	The name of the Kubernetes Persistent Volume Access Mode to use.
matchLabels	`create`	Only used with `StorageType` of `create`, used to match a particular subset of provisioned Persistent Volu
name	`create`	Only needed for `PrimaryStorage` in `pgclusters.crunchydata.com`.Used to identify the name of the Post
size	`create`	The size of the Persistent Volume Claim (PVC). Must use a Kubernetes resource value, e.g. `20Gi`.
storageclass	`create`	The name of the Kubernetes StorageClass to use.
storagetype	`create`	Set to `create` if storage is provisioned (e.g. using `hostpath`). Set to `dynamic` if using a dynamic storage p
supplementalgroups	`create`	If provided, a comma-separated list of group IDs to use in case it is needed to interface with a particular s

Node Affinity Specification Sets the [node affinity]({{< relref "/architecture/high-availability/_index.md#node-affinity" >}}) for the PostgreSQL cluster and associated deployments. Follows the Kubernetes standard format for setting node affinity, including `preferred` and `required` node affinity.

To set node affinity for a PostgreSQL cluster, you will need to modify the `default` attribute in the node affinity specification. As mentioned above, the values that `default` accepts match what Kubernetes uses for node affinity.

For a detailed explanation for node affinity works. Please see the [high-availability]({{< relref "/architecture/high-availability/_index.md#no affinity" >}}) documentation.

Attribute	Action	Description
default	`create`	The default pod anti-affinity to use for all PostgreSQL instances managed in a given PostgreSQL cluster. Can be ov

Pod Anti-Affinity Specification Sets the [pod anti-affinity]({{< relref "/architecture/high-availability/_index.md#how-the-crunchy-postgresql-operator-uses-pod-anti-affinity" >}}) for the PostgreSQL cluster and associated deployments. Each attribute can contain one of the following values:

- `required`
- `preferred` (which is also the recommended default)
- `disabled`

For a detailed explanation for how this works. Please see the [high-availability]({{< relref "/architecture/high-availability/_index.md#how-the-crunchy-postgresql-operator-uses-pod-anti-affinity" >}}) documentation.

Attribute	Action	Description
default	`create`	The default pod anti-affinity to use for all Pods managed in a given PostgreSQL cluster.
pgBackRest	`create`	If set to a value that differs from `Default`, specifies the pod anti-affinity to use for just the pgBackRest repository
pgBouncer	`create`	If set to a value that differs from `Default`, specifies the pod anti-affinity to use for just the pgBouncer Pods.

PostgreSQL Data Source Specification This specification is used when one wants to bootstrap the data in a PostgreSQL cluster from a pgBackRest repository. This can be a pgBackRest repository that is attached to an active PostgreSQL cluster or is kept around to be used for spawning new PostgreSQL clusters.

Attribute	Action	Description
restoreFrom	`create`	The name of a PostgreSQL cluster, active or former, that will be used for bootstrapping the data of a new Postgr
restoreOpts	`create`	Additional pgBackRest restore options that can be used as part of the bootstrapping operation, for example, poin

TLS Specification The TLS specification makes a reference to the various secrets that are required to enable TLS in a PostgreSQL cluster. For more information on how these secrets should be structured, please see [Enabling TLS in a PostgreSQL Cluster]({{< relref "/pgo-client/common-tasks.md#enable-tls" >}}).

Attribute	Action	Description
caSecret	`create`	A reference to the name of a Kubernetes Secret that specifies a certificate authority for the PostgreSQL
replicationTLSSecret	`create`	A reference to the name of a Kubernetes TLS Secret that contains a keypair for authenticating the replic
tlsSecret	`create`	A reference to the name of a Kubernetes TLS Secret that contains a keypair that is used for the Postgre

pgBouncer Specification The pgBouncer specification defines how a pgBouncer deployment can be deployed alongside the PostgreSQL cluster. pgBouncer is a PostgreSQL connection pooler that can also help manage connection state, and is helpful to deploy alongside a PostgreSQL cluster to help with failover scenarios too.

Attribute	Action	Description
limits	`create`, `update`	Specify the container resource limits that the pgBouncer Pods should use. Follows the Kubernetes definit
replicas	`create`, `update`	The number of pgBouncer instances to deploy. Must be set to at least `1` to deploy pgBouncer. Setting to
resources	`create`, `update`	Specify the container resource requests that the pgBouncer Pods should use. Follows the Kubernetes defi
serviceType	`create`, `update`	Sets the Kubernetes Service type to use for the cluster. If not set, defaults to the `ServiceType` set for th
tlsSecret	`create`	A reference to the name of a Kubernetes TLS Secret that contains a keypair that is used for the pgBounc

Annotations Specification The `pgcluster.crunchydata.com` specification contains a block that allows for custom Annotations to be applied to the Deployments that are managed by the PostgreSQL Operator, including:

- PostgreSQL
- pgBackRest
- pgBouncer

This also includes the option to apply Annotations globally across the three different deployment groups.

Attribute	Action	Description
backrest	`create`, `update`	Specify annotations that are only applied to the pgBackRest deployments
global	`create`, `update`	Specify annotations that are applied to the PostgreSQL, pgBackRest, and pgBouncer deployments
pgBouncer	`create`, `update`	Specify annotations that are only applied to the pgBouncer deployments
postgres	`create`, `update`	Specify annotations that are only applied to the PostgreSQL deployments

`pgreplicas.crunchydata.com`

The `pgreplicas.crunchydata.com` Custom Resource Definition contains information pertaning to the structure of PostgreSQL replicas associated within a PostgreSQL cluster. All of the attributes only affect the replica when it is created.

Attribute	Action	Description
clustername	**Specification** (`Spec`)	The name of the PostgreSQL cluster, e.g. `hippo`. This is used to group PostgreSQL instances (primary, replicas)
name	`create`	The name of this PostgreSQL replica. It should be unique within a `ClusterName`.
namespace	`create`	The Kubernetes Namespace that the PostgreSQL cluster is deployed in.
nodeAffinity	`create`	Sets the [node affinity rules]({{< relref "/architecture/high-availability/_index.md#node-affinity" >}}) for this P

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 available 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*

Permission	Description
CreateBackup	allow *pgo backup*
CreateCluster	allow *pgo create cluster*
CreateDump	allow *pgo create pgdump*
CreateFailover	allow *pgo failover*
CreatePgAdmin	allow *pgo create pgadmin*
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*
DeleteCluster	allow *pgo delete cluster*
DeletePgAdmin	allow *pgo delete pgadmin*
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*
Label	allow *pgo label*
Reload	allow *pgo reload*
Restart	allow *pgo restart*
Restore	allow *pgo restore*
RestoreDump	allow *pgo restore* for pgdumps
ShowBackup	allow *pgo show backup*
ShowCluster	allow *pgo show cluster*
ShowConfig	allow *pgo show config*
ShowPgAdmin	allow *pgo show pgadmin*
ShowPgBouncer	allow *pgo show pgbouncer*
ShowPolicy	allow *pgo show policy*
ShowPVC	allow *pgo show pvc*
ShowSchedule	allow *pgo show schedule*
ShowNamespace	allow *pgo show namespace*
ShowSystemAccounts	allows commands with the`--show-system-accounts` flag to return system account information (e.g. the `postgre`
ShowUpgrade	allow *pgo show upgrade*
ShowWorkflow	allow *pgo show workflow*
Status	allow *pgo status*
TestCluster	allow *pgo test*
UpdatePgBouncer	allow *pgo update pgbouncer*
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

For a list of the RBAC required to install the PostgreSQL Operator, please view the `postgres-operator.yml` file:

https://raw.githubusercontent.com/CrunchyData/postgres-operator/v{{< param operatorVersion >}}/installers/kubectl/postgres-operator.yml

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

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

Also, the specific Cluster Roles installed depends on the Namespace Mode enabled via the `PGO_NAMESPACE_MODE` environment variable when running `make installrbac`. Please consult the Namespace documentation for more information regarding the Namespace Modes available, including the specific `ClusterRoles` required to enable each mode.

The crunchy-postgres-exporter 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.

The crunchy-postgres-exporter container uses pgMonitor for advanced metric collection. It is required that the `crunchy-postgres-ha` container has the `PGMONITOR_PASSWORD` environment variable to create the appropriate user (`ccp_monitoring`) to collect metrics.

Custom queries to collect metrics can be specified by the user. By mounting a **queries.yml** file to */conf* on the container, additional metrics can be specified for the API to collect. For an example of a queries.yml file, see here

Packages

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

- PostgreSQL ({{< param postgresVersion13 >}}, {{< param postgresVersion12 >}}, {{< param postgresVersion11 >}}, {{< param postgresVersion10 >}}, {{< param postgresVersion96 >}} and {{< param postgresVersion95 >}})
- CentOS 8 - publicly available
- UBI 7, UBI 8 - customers only
- PostgreSQL Exporter

Environment Variables

Required

Name	Default	Description
EXPORTER_PG_PASSWORD	none	Provides the password needed to generate the PostgreSQL URL required by the Postgr

Optional

Name	Default	Description
EXPORTER_PG_USER	ccp_monitoring	Provides the username needed to generate the PostgreSQL URL required by
EXPORTER_PG_HOST	127.0.0.1	Provides the host needed to generate the PostgreSQL URL required by the P
EXPORTER_PG_PORT	5432	Provides the port needed to generate the PostgreSQL URL required by the P
EXPORTER_PG_DATABASE	postgres	Provides the name of the database used to generate the PostgreSQL URL re
DATA_SOURCE_NAME	None	Explicitly defines the URL for connecting to the PostgreSQL database (must
CRUNCHY_DEBUG	FALSE	Set this to true to enable debugging in logs. Note: this mode can reveal secr
POSTGRES_EXPORTER_PORT	9187	Set the postgres-exporter port to listen on for web interface and telemetry.

Viewing Cluster Metrics

To view a particular cluster's available metrics in a local browser window, port forwarding can be set up as follows. For a pgcluster, `mycluster`, deployed in the `pgouser1` namespace, use

```
# If deployed to Kubernetes
kubectl port-forward -n pgouser1 svc/mycluster 9187:9187

# If deployed to OpenShift
oc port-forward -n pgouser1 svc/mycluster 9187:9187
```

Then, in your local browser, go to `http://127.0.0.1:9187/metrics` to view the available metrics for that cluster.

Crunchy Postgres Exporter Metrics Detail

Below are details on the various metrics available from the crunchy-postgres-exporter container. The name, SQL query and metric details are given for each available item.

{{< exporter_metrics >}}

pgnodemx

In addition to the metrics above, the pgnodemx PostgreSQL extension provides SQL functions to allow the capture of node OS metrics via SQL queries. For more information, please see the pgnodemx project page:

https://github.com/CrunchyData/pgnodemx

{{< pgnodemx_metrics >}}

Custom PostgreSQL Configuration

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

- postgres-ha.yaml
- 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 *postgres-ha.yaml* file is the main configuration file that allows for the configuration of a wide variety of tuning parameters for you PostgreSQL cluster. This includes various PostgreSQL settings, e.g. those that should be applied to files such as `postgresql.conf`, `pg_hba.conf` and `pg_ident.conf`, as well as tuning parameters for the High Availability features included in each cluster. The various configuration settings available can be found here

The *setup.sql* file is a SQL file that is executed following the initialization of a new PostgreSQL cluster, specifically after *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.

Create a PostgreSQL Cluster With Custom Configuration

The PostgreSQL Operator allows for a PostgreSQL cluster to be created with a customized configuration. To do this, one must create a ConfigMap with an entry called `postgres-ha.yaml` that contains the custom configuration. The custom configuration follows the Patorni YAML format. Note that parameters that are placed in the `bootstrap` section are applied once during cluster initialization. Editing these values in a working cluster require following the Modifying PostgreSQL Cluster Configuration section.

For example, let's say we want to create a PostgreSQL cluster with `shared_buffers` set to `2GB`, `max_connections` set to `30` and `password_encryption` set to `scram-sha-256`. We would create a configuration file that looks similar to:

```
---
bootstrap:
  dcs:
    postgresql:
      parameters:
        max_connections: 30
        shared_buffers: 2GB
        password_encryption: scram-sha-256
```

Save this configuration in a file called `postgres-ha.yaml`.

Create a ConfigMap like so:

```
kubectl -n pgo create configmap hippo-custom-config --from-file=postgres-ha.yaml
```

You can then have you new PostgreSQL cluster use `hippo-custom-config` as part of its cluster initialization by using the `--custom-config` flag of `pgo create cluster`:

```
pgo create cluster hippo -n pgo --custom-config=hippo-custom-config
```

After your cluster is initialized, [connect to your cluster]({{< relref "tutorial/connect-cluster.md" >}}) and confirm that your settings have been applied:

```
SHOW shared_buffers;

 shared_buffers
----------------
 2GB
```

Modifying PostgreSQL Cluster Configuration

Once a PostgreSQL cluster has been initialized, its configuration settings can be updated and modified as needed. This done by modifying the `<clusterName>-pgha-config` ConfigMap that is created for each individual PostgreSQL cluster.

The `<clusterName>-pgha-config` ConfigMap is populated following cluster initializtion, specifically using the baseline configuration settings used to bootstrap the cluster. Therefore, any customiztions applied using a custom `postgres-ha.yaml` file as described in the **Custom PostgreSQL Configuration** section above will also be included when the ConfigMap is populated.

The various configuration settings available for modifying and updating and cluster's configuration can be found here. Please proceed with caution when modiying configuration, especially those settings applied by default by Operator. Certain settings are required for normal operation of the Operator and the PostgreSQL clusters it creates, and altering these settings could result in expected behavior.

Types of Configuration

Within the `<clusterName>-pgha-config` ConfigMap are two forms of configuration:

- **Distributed Configuration Store (DCS):** Cluster-wide configuration settings that are applied to all database servers in the PostgreSQL cluster
- **Local Database:** Configuration settings that are applied individually to each database server (i.e. the primary and each replica) within the cluster.

The DCS configuration settings are stored within the `<clusterName>-pgha-config` ConfigMap in a configuration named `<clusterName>-dcs` while the local database configurations are stored in one or more configurations named `<serverName>-local-config` (with one local configuration for the primary and each replica within the cluster). Please note that as described here, certain settings can only be applied via the DCS to ensure they are uniform among the primary and all replicas within the cluster.

The following is an example of the both the DCS and primary configuration settings as stored in the `<clusterName>-pgha-config` ConfigMap for a cluster named `mycluster`. Please note the `mycluster-dcs-config` configuration defining the DCS configuration for `mycluster`, along with the `mycluster-local-config` configuration defining the local configuration for the database server named `mycluster`, which is the current primary within the PostgreSQL cluster.

```
$ kubectl describe cm mycluster-pgha-config
Name:         mycluster-pgha-config
Namespace:    pgouser1
Labels:       pg-cluster=mycluster
              pgha-config=true
              vendor=crunchydata
Annotations:  <none>

Data
====
mycluster-dcs-config:
----
postgresql:
  parameters:
```

```
    archive_command: source /opt/cpm/bin/pgbackrest/pgbackrest-set-env.sh && pgbackrest
      archive-push "%p"
    archive_mode: true
    archive_timeout: 60
    log_directory: pg_log
    log_min_duration_statement: 60000
    log_statement: none
    max_wal_senders: 6
    shared_buffers: 128MB
    shared_preload_libraries: pgaudit.so,pg_stat_statements.so
    temp_buffers: 8MB
    unix_socket_directories: /tmp
    wal_level: logical
    work_mem: 4MB
  recovery_conf:
    restore_command: source /opt/cpm/bin/pgbackrest/pgbackrest-set-env.sh && pgbackrest
      archive-get %f "%p"
  use_pg_rewind: true

mycluster-local-config:
----
postgresql:
  callbacks:
    on_role_change: /opt/cpm/bin/callbacks/pgha-on-role-change.sh
  create_replica_methods:
  - pgbackrest
  - basebackup
  pg_hba:
  - local all postgres peer
  - host replication primaryuser 0.0.0.0/0 md5
  - host all primaryuser 0.0.0.0/0 reject
  - host all all 0.0.0.0/0 md5
  pgbackrest:
    command: /opt/cpm/bin/pgbackrest/pgbackrest-create-replica.sh
    keep_data: true
    no_params: true
  pgbackrest_standby:
    command: /opt/cpm/bin/pgbackrest/pgbackrest-create-replica.sh
    keep_data: true
    no_master: 1
    no_params: true
  pgpass: /tmp/.pgpass
  remove_data_directory_on_rewind_failure: true
  use_unix_socket: true
```

Updating Configuration Settings

In order to update a cluster's configuration settings and then apply those settings (e.g. to the DCS and/or any individual database servers), the DCS and local configuration settings within the `<clusterName>-pgha-config` ConfigMap can be modified. This can be done using the various commands available using the `kubectl` client (or the `oc` client if using OpenShift) for modifying Kubernetes resources. For instance, the following command can be utilized to open the ConfigMap in a local text editor, and then update the various cluster configurations as needed:

```
kubectl edit configmap mycluster-pgha-config
```

Once the `<clusterName>-pgha-config` ConfigMap has been updated, any changes made will be detected by the Operator, and then applied to the DCS and/or any individual database servers within the cluster.

PostgreSQL Configuration In order to update the `postgresql.conf` file for a one of more database servers, the `parameters` section of either the DCS and/or a local database configuration can be updated, e.g.:

```
----
postgresql:
  parameters:
    max_wal_senders: 10
```

The various key/value pairs provided within the `parameters` section result in the configuration of the same settings within the `postgresql.conf` file. Please note that settings applied locally to a database server take precedence over those set via the DCS (with the exception being those that must be set via the DCS, as described here).

Also, please note that `pg_hba` and `pg_ident` sections exist to update both the `pg_hba.conf` and `pg_ident.conf` PostgreSQL configuration files as needed.

A Note on Customizing `authentication` One of the blocks that can be modified in a `local` database setting is the `authentication` block. This can be useful for setting customizations such as TLS connection requirements (`sslmode`). However, one should take care when modifying this block, as modifying certain parameters can interfere with the management features that the PostgreSQL Operator provides.

In particular, one should **not** customize the `username` or `password` attributes within this section as that will interface with the PostgreSQL Operator. Additionally, is using the built-in support for certificate-based authentication for replication users, you should not modify the `sslcert`, `sslkey`, `sslrootcert`, and `sslcrl` entries in the `replication` block of the `authentication` block.

Restarting Database Servers

Changes to certain settings may require one or more PostgreSQL databases within the cluster to be restarted. This can be accomplished using the `pgo restart` command included with the `pgo` client. To detect if a restart is needed for a instance within a cluster called `mycluster` after making a configuration change, the `query` flag can be utilized with the `pgo restart` command as follows:

```
$ pgo restart mycluster2 --query

Cluster: mycluster2
INSTANCE                ROLE            STATUS          NODE            REPLICATION LAG
   PENDING RESTART
mycluster               primary         running         node01                    0 MB
                  false
mycluster-ambq          replica         running         node01                    0 MB
                   true
```

Here we can see that the `mycluster-ambq` instance (i.e. the sole replica in cluster `mycluster`) is pending a restart, as shown by the `PENDING RESTART` column. A restart can then be requested as follows:

```
$ pgo restart mycluster --target mycluster-ambq
WARNING: Are you sure? (yes/no): yes
Successfully restarted instance mycluster
```

It is also possible to target multiple instances at the same time:

```
$ pgo restart mycluster --target mycluster --target mycluster-ambq
WARNING: Are you sure? (yes/no): yes
Successfully restarted instance mycluster
Successfully restarted instance mycluster-ambq
```

Or if no target is specified, the all instances within the cluster will be restarted:

```
$ pgo restart mycluster
WARNING: Are you sure? (yes/no): yes
Successfully restarted instance mycluster
Successfully restarted instance mycluster-ambq
```

Refreshing Configuration Settings

If necessary, it is possible to refresh the configuration stored within the `<clusterName>-pgha-config` ConfigMap with a fresh copy of either the DCS configuration and/or the configuration for one or more local database servers. This is specifically done by fully deleting a configuration from the `<clusterName>-pgha-config` ConfigMap. Once a configuration has been deleted, the Operator will detect this and refresh the ConfigMap with a fresh copy of that specific configuration.

For instance, the following `kubectl patch` command can be utilized to remove the `mycluster-dcs-config` configuration from the example above, causing that specific configuration to be refreshed with a fresh copy of the DCS configuration settings for `mycluster`:

```
kubectl patch configmap mycluster-pgha-config \
  --type='json' -p='[{"op": "remove", "path": "/data/mycluster-dcs-config"}]'
```

Custom pgBackRest Configuration

Users can configure pgBackRest by passing the name of an existing ConfigMap to the `--pgbackrest-custom-config` flag when creating a PostgreSQL cluster. The entire contents of that ConfigMap appear as files in pgBackRest's `config-include-path` directory.

Regardless of the flags passed at creation, every PostgreSQL cluster is automatically configured to read from a ConfigMap named `<clusterName>-config-backrest` and a Secret named `<clusterName>-config-backrest`. These objects can be populated either before or *after* a PostgreSQL cluster is created. The entire contents of each appear as files in pgBackRest's `config-include-path` directory.

Though the above is very flexible, not all pgBackRest settings can be managed this way. There are a few that are always overridden by the PostgreSQL Operator (the path to the PostgreSQL data directory, for example).

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.

Get API Server Version

```
curl --cacert $PGO_CA_CERT --key $PGO_CLIENT_KEY --cert $PGO_CA_CERT \
-u admin: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, and the name of the new cluster.

Create Cluster

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

Delete Cluster

```
curl --cacert $PGO_CA_CERT --key $PGO_CLIENT_KEY --cert $PGO_CA_CERT \
-u admin:examplepassword -H "Content-Type:application/json" --insecure \
-X POST --data \
  '{"ClientVersion":"{{< param operatorVersion >}}",
  "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 Behavior 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 behavior 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 behavior 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 Crunchy PostgreSQL Operator

There are two methods for upgrading your existing deployment of the PostgreSQL Operator.

If you are upgrading from PostgreSQL Operator 4.1.0 or later, you are encouraged to use the Automated Upgrade Procedure. This method simplifies the upgrade process, as well as maintains your existing clusters in place prior to their upgrade.

For versions before 4.1.0, please see the appropriate manual procedure.

Automated PostgreSQL Operator Upgrade Procedure

The automated upgrade to a new release of the PostgreSQL Operator comprises two main steps:

- Upgrading the PostgreSQL Operator itself
- Upgrading the existing PostgreSQL Clusters to the new release

The first step will result in an upgraded PostgreSQL Operator that is able to create and manage new clusters as expected, but will be unable to manage existing clusters until they have been upgraded. The second step upgrades the clusters to the current Operator version, allowing them to once again be fully managed by the Operator.

The automated upgrade procedure is designed to facilate the quickest and most efficient method to the current release of the PostgreSQL Operator. However, as with any upgrade, there are several considerations before beginning.

Considerations

1. Versions Supported - This upgrade currently supports cluster upgrades from PostgreSQL Operator version 4.1.0 and later.
2. PostgreSQL Major Version Requirements - The underlying PostgreSQL major version must match between the old and new clusters. For example, if you are upgrading a 4.1.0 version of the PostgreSQL Operator and the cluster is using PostgreSQL 11.5, your upgraded clusters will need to use container images with a later minor version of PostgreSQL 11. Note that this is not a requirement for new clusters, which may use any currently supported version. For more information, please see the [Compatibility Requirements]({{< relref "configuration/compatibility.md" >}}).
3. Cluster Downtime - The re-creation of clusters will take some time, generally on the order of minutes but potentially longer depending on the operating environment. As such, the timing of the upgrade will be an important consideration. It should be noted that the upgrade of the PostgreSQL Operator itself will leave any existing cluster resources in place until individual pgcluster upgrades are performed.
4. Destruction and Re-creation of Certain Resources - As this upgrade process does destroy and recreate most elements of the cluster, unhealthy Kubernetes or Openshift environments may have difficulty recreating the necessary elements. Node availability, necessary PVC storage allocations and processing requirements are a few of the resource considerations to make before proceeding.
5. Compatibility with Custom Configurations - Given the nearly endless potential for custom configuration settings, it is important to consider any resource or implemenation that might be uniquely tied to the current PostgreSQL Operator version.
6. Storage Requirements - An essential part of both the automated and manual upgrade procedures is the reuse of existing PVCs. As such, it is essential that the existing storage settings are maintained for any upgraded clusters.
7. As opposed to the manual upgrade procedures, the automated upgrade is designed to leave existing resources (such as CRDs, config maps, secrets, etc) in place whenever possible to minimize the need for resource re-creation.
8. Metrics - While the PostgreSQL Operator upgrade process will not delete an existing Metrics Stack, it does not currently support the upgrade of existing metrics infrastructure.

NOTE: As with any upgrade procedure, it is strongly recommended that a full logical backup is taken before any upgrade procedure is started. Please see the Logical Backups section of the Common Tasks page for more information.

Automated Upgrade when using the PostgreSQL Operator Installer (`pgo-deployer`), Helm or Ansible

For all existing PostgreSQL Operator deployments that were installed using the Ansible installation method, the PostgreSQL Operator Installer or the Helm Chart Installation of the PostgreSQL Operator, the upgrade process is straightforward.

First, you will copy your existing configuration file (whether inventory, postgres-operator.yml, values.yaml, etc, depending on method and version) as a backup for your existing settings. You will reference these settings, but you will need to use the updated version of this file for the current version of PostgreSQL Operator.

In all three cases, you will need to use the relevant update functionality available with your chosen installation method. For all three options, please keep the above Considerations in mind, particularly with regard to the version and storage requirements listed.

PostgreSQL Operator Installer For existing PostgreSQL Operator deployments that were installed using the PostgreSQL Operator Installer, you will check out the appropriate release tag and update your the new configuration files. After this, you will need to update your Operator installation using the `DEPLOY_ACTION` method described in the Configuring to Update and Uninstall section of the documentation.

Please note, you will need to ensure that you have executed the post-installation cleanup between each `DEPLOY_ACTION` activity.

Helm For existing PostgreSQL Operator deployments that were installed using the Helm installer, you will check out the appropriate release tag and update your the new configuration files. Then you will need to use the `helm upgrade` command as described in the Helm Upgrade section of the Helm installation documentation.

Ansible For existing PostgreSQL Operator deployments that were installed using Ansible, you will first need to check out the appropriate release tag of the Operator. Then please follow the [Update Instructions]({{< relref "installation/other/ansible/updating-operator.md" >}}), being sure to update the new inventory file with your required settings.

Wrapping Up the PostgreSQL Operator Upgrade Once the upgrade is complete, you should now see the PostgreSQL Operator pods are up and ready. It is strongly recommended that you create a test cluster to validate proper functionality before moving on to the Automated Cluster Upgrade section below.

Automated Upgrade when using a Bash installation of the PostgreSQL Operator

Like the Ansible procedure given above, the Bash upgrade procedure for upgrading the PostgreSQL Operator will require some manual configuration steps before the upgrade can take place. These updates will be made to your user's environment variables and the pgo.yaml configuration file.

PostgreSQL Operator Configuration Updates To begin, you will need to make the following updates to your existing configuration.

Bashrc File Updates First, you will make the following updates to your $HOME/.bashrc file.

When upgrading from version 4.1.X, in `$HOME/.bashrc`

Add the following variables:

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

# Disable default inclusion of OS trust in PGO clients
export EXCLUDE_OS_TRUST=false
```

Then, for either 4.1.X or 4.2.X,

Update the `PGO_VERSION` variable to `{{< param operatorVersion >}}`

Finally, source this file with

```
source $HOME/.bashrc
```

PostgreSQL Operator Configuration File updates Next, you will and save a copy of your existing pgo.yaml file (`$PGOROOT/conf/post`
as pgo_old.yaml or similar.

Once this is saved, you will checkout the current release of the PostgreSQL Operator and update the pgo.yaml for the current version, making sure to make updates to the CCPImageTag and storage settings in line with the Considerations given above.

Upgrading the Operator Once the above configuration updates are completed, the PostgreSQL Operator can be upgraded. To help ensure that needed resources are not inadvertently deleted during an upgrade of the PostgreSQL Operator, a helper script is provided. This script provides a similar function to the Ansible installation method's 'update' tag, where the Operator is undeployed, and the designated namespaces, RBAC rules, pods, etc are redeployed or recreated as appropriate, but required CRDs and other resources are left in place.

To use the script, execute:

```
$PGOROOT/deploy/upgrade-pgo.sh
```

This script will undeploy the current PostgreSQL Operator, configure the desired namespaces, install the RBAC configuration, deploy the new Operator, and, attempt to install a new PGO client, assuming default location settings are being used.

After this script completes, it is strongly recommended that you create a test cluster to validate the Operator is functioning as expected before moving on to the individual cluster upgrades.

PostgreSQL Operator Automated Cluster Upgrade

Previously, the existing cluster upgrade focused on updating a cluster's underlying container images. However, due to the various changes in the PostgreSQL Operator's operation between the various versions (including numerous updates to the relevant CRDs, integration of Patroni for HA and other significant changes), updates between PostgreSQL Operator releases required the manual deletion of the existing clusters while preserving the underlying PVC storage. After installing the new PostgreSQL Operator version, the clusters could be recreated manually with the name of the new cluster matching the existing PVC's name.

The automated upgrade process provides a mechanism where, instead of being deleted, the existing PostgreSQL clusters will be left in place during the PostgreSQL Operator upgrade. While normal Operator functionality will be restricted on these existing clusters until they are upgraded to the currently installed PostgreSQL Operator version, the pods, services, etc will still be in place and accessible via other methods (e.g. kubectl, service IP, etc).

To upgrade a PostgreSQL cluster using the standard (`crunchy-postgres-ha`) image, you can run the following command:

```
pgo upgrade mycluster
```

If you are using the PostGIS-enabled image (i.e. `crunchy-postgres-gis-ha`) or any other custom images, you will need to add the `--ccp-image-tag`:

```
pgo upgrade --ccp-image-tag={{< param centosBase >}}-{{< param postgresVersion >}}-{{< param
    postgisVersion >}}-{{< param operatorVersion >}} mygiscluster
```

Where `{{< param postgresVersion >}}` is the PostgreSQL version, `{{< param postgisVersion >}}` is the PostGIS version and `{{< param operatorVersion >}}` is the PostgreSQL Operator version. Please note, no tag validation will be performed and additional steps may be required to upgrade your PostGIS extension implementation. For more information on PostGIS upgrade considerations, please see PostGIS Upgrade Documentation.

This will follow a similar process to the documented manual process, where the pods, deployments, replicasets, pgtasks and jobs are deleted, the cluster's replicas are scaled down and replica PVCs deleted, but the primary PVC and backrest-repo PVC are left in place. Existing services for the primary, replica and backrest-shared-repo are also kept and will be updated to the requirements of the current version. Configmaps and secrets are kept except where deletion is required. For a cluster 'mycluster', the following configmaps will be deleted (if they exist) and recreated:

```
mycluster-leader
mycluster-pgha-default-config
```

along with the following secret:

```
mycluster-backrest-repo-config
```

The pgcluster CRD will be read, updated automatically and replaced, at which point the normal cluster creation process will take over. The end result of the upgrade should be an identical numer of pods, deployments, replicas, etc with a new pgbackrest backup taken, but existing backups left in place.

Finally, to disable PostgreSQL version checking during the upgrade, such as for when container images are re-tagged and no longer follow the standard version tagging format, use the "ignore-validation" flag:

```
pgo upgrade mycluster --ignore-validation
```

That will allow the upgrade to proceed, regardless of the tag values. Please note, the underlying image must still be chosen in accordance with the Considerations listed above.

Upgrade Guidance for PostgreSQL Operator Monitoring

Migration to Upstream Containers

The Crunchy PostgreSQL Monitoring infrastructure now uses upstream Prometheus and Grafana containers. By default the installers will deploy the monitoring infrastructure using images from Docker Hub but can easily be updated to point to a Red Hat certified container repository. The Red Hat certified image catalog can be found here and the Docker Hub images can be found at the following links:

- https://hub.docker.com/r/prom/prometheus
- https://hub.docker.com/r/grafana/grafana
- https://hub.docker.com/r/prom/alertmanager

These containers are configurable through Kubernetes ConfigMaps and the updated PostgreSQL Operator Monitoring installers. Once deployed Prometheus and Grafana will be populated with resource data from metrics-enabled PostgreSQL clusters.

New Monitoring Features

Alerting

The updated PostgreSQL Operator Monitoring Infrastructure supports deployment of Prometheus Alertmanager. This deployment uses upstream Prometheus Alertmanager images that can be installed and configured with the metrics installers and Kubernetes ConfigMaps.

Updated pgMonitor

Prometheus and Grafana have been updated to include a default configuration from pgMonitor that is tailored for container-based PostgreSQL deployments. This updated configuration will show container specific resource information from your metrics-enabled PostgreSQL clusters. By default the metrics infrastructure will include:

- New Grafana dashboards tailored for container-based PostgreSQL deployments
- Container specific operating system metrics
- General PostgreSQL alerting rules.

Updated Monitoring Installer

The installer for the PostgreSQL Operating Monitoring infrastructure has been split out into a separate set of installers. With each installer (Ansible({{< relref "/installation/metrics/other/ansible" >}}), a [Kubectl job]({{< relref "installation/metrics/postgres-operator-metrics" >}}), or Helm({{< relref "/installation/metrics/other/helm-metrics" >}})) you will be able to apply custom configurations through Kubernetes ConfigMaps. This includes:

- Custom Grafana dashboards and datasources
- Custom Prometheus scrape configuration
- Custom Prometheus alerting rules
- Custom Alertmanager notification configuration

Updating from Pre-4.5.0 Monitoring

Ensure that you have a copy of any install or custom configurations you have applied to your previous metrics install.

You can upgrade the Grafana and Prometheus deployments in place by using the new installers. After you have updated the PostgreSQL Operator and configured the `values.yaml`, run the [metrics update]({{< relref "/installation/metrics/other/ansible/updating-metrics" >}}). This will replace the old deployments while keeping your pvcs in place.

To make use of the updated exporter queries you must update the PostgreSQL Operator and [upgrade]({{< relref "/upgrade/automatedupgrade" >}}) your cluster.

Manually Upgrading the Operator and PostgreSQL Clusters

In the event that the automated upgrade cannot be used, below are manual upgrade procedures for both PostgreSQL Operator 3.5 and 4.0 releases. These procedures will require action by the Operator administrator of your organization in order to upgrade to the current release of the Operator. Some upgrade steps are still automated within the Operator, but not all are possible with this upgrade method. As such, the pages below show the specific steps required to upgrade different versions of the PostgreSQL Operator depending on your current environment.

NOTE: If you are upgrading from Crunchy PostgreSQL Operator version 4.1.0 or later, the Automated Upgrade Procedure is recommended. If you are upgrading PostgreSQL 12 clusters, you MUST use the Automated Upgrade Procedure.

When performing a manual upgrade, it is recommended to upgrade to the latest PostgreSQL Operator available.

[Manual Upgrade - PostgreSQL Operator 3.5]({{< relref "upgrade/manual/upgrade35.md" >}})

[Manual Upgrade - PostgreSQL Operator 4]({{< relref "upgrade/manual/upgrade4.md" >}})

Upgrading the Crunchy PostgreSQL Operator from Version 3.5 to {{< param operatorVersion >}}

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

Crunchy PostgreSQL High Availability Containers Using the PostgreSQL Operator {{< param operatorVersion >}} 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 {{< param operatorVersion >}} of the PostgreSQL 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 and image tag, as with the command:

```
pgo create cluster mygiscluster --ccp-image=crunchy-postgres-gis-ha --ccp-image-tag={{< param
    centosBase >}}-{{< param postgresVersion >}}-{{< param postgisVersion >}}-{{< param
    operatorVersion >}}
```

Where `{{< param postgresVersion >}}` is the PostgreSQL version, `{{< param postgisVersion >}}` is the PostGIS version and `{{< param operatorVersion >}}` is the PostgreSQL Operator version. Please note, no tag validation will be performed and additional steps may be required to upgrade your PostGIS extension implementation. For more information on PostGIS upgrade considerations, please see PostGIS Upgrade Documentation.

NOTE: As with any upgrade procedure, it is strongly recommended that a full logical backup is taken before any upgrade procedure is started. Please see the Logical Backups section of the Common Tasks page for more information.

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

- The code for the latest PostgreSQL Operator available
- The latest client binary

Step 1 Create a new Linux user with the same permissions as the existing user used to install the Crunchy PostgreSQL Operator. This is necessary to avoid any issues with environment variable differences between 3.5 and {{< param operatorVersion >}}.

Step 2 For the cluster(s) you wish to upgrade, record the cluster details provided by

```
pgo show cluster <clustername>
```

so that your new clusters can be recreated with the proper settings.

Also, you will need to note the name of the primary PVC. If it does not exactly match the cluster name, you will need to recreate your cluster using the primary PVC name as the new cluster name.

For example, given the following output:

```
$ pgo show cluster mycluster

cluster : mycluster (crunchy-postgres:centos7-11.5-2.4.2)
    pod : mycluster-7bbf54d785-pk5dq (Running) on kubernetes1 (1/1) (replica)
    pvc : mycluster
    pod : mycluster-ypvq-5b9b8d645-nvlb6 (Running) on kubernetes1 (1/1) (primary)
    pvc : mycluster-ypvq
...
```

the new cluster's name will need to be "mycluster-ypvq"

Step 3 NOTE: Skip this step if your primary PVC still matches your original cluster name, or if you do not have pgBackrestBackups you wish to preserve for use in the upgraded cluster.

Otherwise, noting the primary PVC name mentioned in Step 2, run

```
kubectl exec mycluster-backrest-shared-repo-<id> -- bash -c "mv
   /backrestrepo/mycluster-backrest-shared-repo /backrestrepo/mycluster-ypvq-backrest-shared-repo"
```

where "mycluster" is the original cluster name, "mycluster-ypvq" is the primary PVC name and "mycluster-backrest-shared-repo-" is the pgBackRest shared repo pod name.

Step 4 For the cluster(s) you wish to upgrade, scale down any replicas, if necessary, then delete the cluster

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

If there are any remaining jobs for this deleted cluster, use

```
kubectl -n <namespace> delete job <jobname>
```

to remove the job and any associated "Completed" pods.

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

Step 5 Delete the 3.5.x version of the operator by executing:

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

Step 6 Log in as your new Linux user and install the {{< param operatorVersion >}} PostgreSQL Operator as described in the [Bash Installation Procedure]({{< relref "installation/other/bash.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.

We strongly recommend that you create a test cluster before proceeding to the next step.

Step 7 Once the Operator is installed and functional, create a new {{< param operatorVersion >}} cluster matching the cluster details recorded in Step 1. Be sure to use the primary PVC name (also noted in Step 1) and the same major PostgreSQL version as was used previously. This will allow the new clusters to utilize the existing PVCs.

NOTE: If you have existing pgBackRest backups stored that you would like to have available in the upgraded cluster, you will need to follow the PVC Renaming Procedure.

A simple example is given below, but more information on cluster creation can be found here

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

Step 8 Manually update the old leftover Secrets to use the new label as defined in {{< param operatorVersion >}}:

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

Step 9 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 10 Scale up to the required number of replicas, as needed.

Congratulations! Your cluster is upgraded and ready to use!

pgBackRest Repo PVC Renaming

If the pgcluster you are upgrading has an existing pgBackRest repo PVC that you would like to continue to use (which is required for existing pgBackRest backups to be accessible by your new cluster), the following renaming procedure will be needed.

Step 1 To start, if your current cluster is “mycluster”, the pgBackRest PVC created by version 3.5 of the Postgres Operator will be named “mycluster-backrest-shared-repo”. This will need to be renamed to “mycluster-pgbr-repo” to be used in your new cluster.

To begin, save the output from

```
kubectl -n <namespace> describe pvc mycluster-backrest-shared-repo
```

for later use when recreating this PVC with the new name. In this output, note the “Volume” name, which is the name of the underlying PV.

Step 2 Now use

```
kubectl -n <namespace> get pv <PV name>
```

to check the “RECLAIM POLICY”. If this is not set to “Retain”, edit the “persistentVolumeReclaimPolicy” value so that it is set to “Retain” using

```
kubectl -n <namespace> patch pv <PV name> --type='json' -p='[{"op": "replace", "path":
   "/spec/persistentVolumeReclaimPolicy", "value":"Retain"}]'
```

Step 3 Now, delete the PVC:

```
kubectl -n <namespace> delete pvc mycluster-backrest-shared-repo
```

Step 4 You will remove the “claimRef” section of the PV with

```
kubectl -n <namespace> patch pv <PV name> --type=json -p='[{"op": "remove", "path":
   "/spec/claimRef"}]'
```

which will make the PV “Available” so it may be reused by the new PVC.

Step 5 Now, create a file with contents similar to the following:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: mycluster-pgbr-repo
  namespace: demo
spec:
  storageClassName: ""
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 2Gi
  volumeMode: Filesystem
  volumeName: "crunchy-pv156"
```

where name matches your new cluster (Remember that this will need to match the “primary PVC” name identified in Step 2 of the upgrade procedure!) and namespace, storageClassName, accessModes, storage, volumeMode and volumeName match your original PVC.

Step 6 Now you can use the new file to recreate your PVC using

```
kubectl -n <namespace> create -f <filename>
```

To check that your PVC is “Bound”, run

```
kubectl -n <namespace> get pvc mycluster-pgbr-repo
```

Congratulations, you have renamed your PVC! Once the PVC Status is “Bound”, your cluster can be recreated. If you altered the Reclaim Policy on your PV in Step 1, you will want to reset it now.

Manual PostgreSQL Operator 4 Upgrade Procedure

Below are the procedures for upgrading to version {{< param operatorVersion >}} of the Crunchy PostgreSQL Operator using the Bash or Ansible installation methods. This version of the PostgreSQL Operator has several fundamental changes to the existing PGCluster structure and deployment model. Most notably for those upgrading from 4.1 and below, all PGClusters use the new Crunchy PostgreSQL HA container in place of the previous Crunchy PostgreSQL containers. The use of this new container is a breaking change from previous versions of the Operator did not use the HA containers.

NOTE: If you are upgrading from Crunchy PostgreSQL Operator version 4.1.0 or later, the Automated Upgrade Procedure is recommended. If you are upgrading PostgreSQL 12 clusters, you MUST use the Automated Upgrade Procedure.

Crunchy PostgreSQL High Availability Containers Using the PostgreSQL Operator {{< param operatorVersion >}} 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 {{< param operatorVersion >}} of the PostgreSQL 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 and image tag, as with the command:

```
pgo create cluster mygiscluster --ccp-image=crunchy-postgres-gis-ha --ccp-image-tag={{< param
    centosBase >}}-{{< param postgresVersion >}}-{{< param postgisVersion >}}-{{< param
    operatorVersion >}}
```

Where `{{< param postgresVersion >}}` is the PostgreSQL version, `{{< param postgisVersion >}}` is the PostGIS version and `{{< param operatorVersion >}}` is the PostgreSQL Operator version. Please note, no tag validation will be performed and additional steps may be required to upgrade your PostGIS extension implementation. For more information on PostGIS upgrade considerations, please see PostGIS Upgrade Documentation.

NOTE: As with any upgrade procedure, it is strongly recommended that a full logical backup is taken before any upgrade procedure is started. Please see the Logical Backups section of the Common Tasks page for more information.

The Ansible installation upgrade procedure is below. Please click here for the Bash installation upgrade procedure.

Ansible Installation Upgrade Procedure

Below are the procedures for upgrading the PostgreSQL Operator and PostgreSQL clusters using the Ansible installation method.

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

- The latest {{< param operatorVersion >}} 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 1 For the cluster(s) you wish to upgrade, record the cluster details provided by

```
pgo show cluster <clustername>
```

so that your new clusters can be recreated with the proper settings.

Also, you will need to note the name of the primary PVC. If it does not exactly match the cluster name, you will need to recreate your cluster using the primary PVC name as the new cluster name.

For example, given the following output:

```
$ pgo show cluster mycluster

cluster : mycluster (crunchy-postgres:centos7-11.5-2.4.2)
        pod : mycluster-7bbf54d785-pk5dq (Running) on kubernetes1 (1/1) (replica)
        pvc : mycluster
        pod : mycluster-ypvq-5b9b8d645-nvlb6 (Running) on kubernetes1 (1/1) (primary)
        pvc : mycluster-ypvq
...
```

the new cluster's name will need to be "mycluster-ypvq"

Step 2 NOTE: Skip this step if your primary PVC still matches your original cluster name, or if you do not have pgBackrestBackups you wish to preserve for use in the upgraded cluster.

Otherwise, noting the primary PVC name mentioned in Step 2, run

```
kubectl exec mycluster-backrest-shared-repo-<id> -- bash -c "mv
   /backrestrepo/mycluster-backrest-shared-repo /backrestrepo/mycluster-ypvq-backrest-shared-repo"
```

where “mycluster” is the original cluster name, “mycluster-ypvq” is the primary PVC name and “mycluster-backrest-shared-repo-” is the pgBackRest shared repo pod name.

Step 3 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

For 4.2:

```
pgo delete cluster <clustername> --keep-backups --keep-data
```

For 4.0 and 4.1:

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

and then, for all versions, delete the “backrest-repo-config” secret, if it exists:

```
kubectl delete secret <clustername>-backrest-repo-config
```

If there are any remaining jobs for this deleted cluster, use

```
kubectl -n <namespace> delete job <jobname>
```

to remove the job and any associated “Completed” pods.

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

Step 4 Save a copy of your current inventory file with a new name (such as `inventory.backup`) and checkout the latest {{< param operatorVersion >}} tag of the Postgres Operator.

Step 5 Update the new inventory file with the appropriate values for your new Operator installation, as described in the [Ansible Install Prerequisites]({{< relref “installation/other/ansible/prerequisites.md” >}}) and the [Compatibility Requirements Guide]({{< relref “configuration/compatibility.md” >}}).

Step 6 Now you can upgrade your Operator installation and configure your connection settings as described in the [Ansible Update Page]({{< relref “installation/other/ansible/updating-operator.md” >}}).

Step 7 Verify the Operator is running:

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

And that it is upgraded to the appropriate version

```
pgo version
```

We strongly recommend that you create a test cluster before proceeding to the next step.

Step 8 Once the Operator is installed and functional, create a new {{< param operatorVersion >}} cluster matching the cluster details recorded in Step 1. Be sure to use the primary PVC name (also noted in Step 1) and the same major PostgreSQL version as was used previously. This will allow the new clusters to utilize the existing PVCs.

NOTE: If you have existing pgBackRest backups stored that you would like to have available in the upgraded cluster, you will need to follow the PVC Renaming Procedure.

A simple example is given below, but more information on cluster creation can be found here

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

Step 9 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 10 Scale up to the required number of replicas, as needed.

Congratulations! Your cluster is upgraded and ready to use!

Bash Installation Upgrade Procedure

Below are the procedures for upgrading the PostgreSQL Operator and PostgreSQL clusters using the Bash installation method.

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

- The code for the latest release of the PostgreSQL Operator
- The latest 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 1 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 2 For the cluster(s) you wish to upgrade, record the cluster details provided by

```
pgo show cluster <clustername>
```

so that your new clusters can be recreated with the proper settings.

Also, you will need to note the name of the primary PVC. If it does not exactly match the cluster name, you will need to recreate your cluster using the primary PVC name as the new cluster name.

For example, given the following output:

```
$ pgo show cluster mycluster

cluster : mycluster (crunchy-postgres:centos7-11.5-2.4.2)
        pod : mycluster-7bbf54d785-pk5dq (Running) on kubernetes1 (1/1) (replica)
        pvc : mycluster
        pod : mycluster-ypvq-5b9b8d645-nvlb6 (Running) on kubernetes1 (1/1) (primary)
        pvc : mycluster-ypvq
...
```

the new cluster's name will need to be "mycluster-ypvq"

Step 3 NOTE: Skip this step if your primary PVC still matches your original cluster name, or if you do not have pgBackrestBackups you wish to preserve for use in the upgraded cluster.

Otherwise, noting the primary PVC name mentioned in Step 2, run

```
kubectl exec mycluster-backrest-shared-repo-<id> -- bash -c "mv
   /backrestrepo/mycluster-backrest-shared-repo /backrestrepo/mycluster-ypvq-backrest-shared-repo"
```

where "mycluster" is the original cluster name, "mycluster-ypvq" is the primary PVC name and "mycluster-backrest-shared-repo-" is the pgBackRest shared repo pod name.

Step 4 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

For 4.2:

```
pgo delete cluster <clustername> --keep-backups --keep-data
```

For 4.0 and 4.1:

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

and then, for all versions, delete the "backrest-repo-config" secret, if it exists:

```
kubectl delete secret <clustername>-backrest-repo-config
```

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

Step 5 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 6 For versions 4.0, 4.1 and 4.2, update environment variables in the bashrc:

```
export PGO_VERSION={{< param operatorVersion >}}
```

NOTE: This will be the only update to the bashrc file for 4.2.

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 PostgreSQL Operator 4.0, 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, 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 PostgreSQL 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=""
```

Step 7 Source the updated bash file:

```
source ~/.bashrc
```

Step 8 Ensure you have checked out the latest {{< param operatorVersion >}} version of the source code and update the pgo.yaml file in `$PGOROOT/conf/postgres-operator/pgo.yaml`

You will want to use the {{< param operatorVersion >}} pgo.yaml file and update custom settings such as image locations, storage, and resource configs.

Step 9 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:

```
PGOADMIN_USERNAME=admin
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/other/bash.md" >}}). Once these accounts are created, you can change this file to login in via the PGO CLI as that user.

Step 10 Install the {{< param operatorVersion >}} Operator:

Setup the configured namespaces:

```
make setupnamespaces
```

Install the RBAC configurations:

```
make installrbac
```

Deploy the PostgreSQL Operator:

```
make deployoperator
```

Verify the Operator is running:

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

Step 11 Next, update the PGO client binary to {{< param operatorVersion >}} by replacing the existing 4.X binary with the latest {{< param operatorVersion >}} binary available.

You can run:

```
which pgo
```

to ensure you are replacing the current binary.

Step 12 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 13 The Operator is now upgraded to {{< param operatorVersion >}} and all users and roles have been recreated. Verify this by running:

```
pgo version
```

We strongly recommend that you create a test cluster before proceeding to the next step.

Step 14 Once the Operator is installed and functional, create a new {{< param operatorVersion >}} cluster matching the cluster details recorded in Step 1. Be sure to use the same name and the same major PostgreSQL version as was used previously. This will allow the new clusters to utilize the existing PVCs. A simple example is given below, but more information on cluster creation can be found here

NOTE: If you have existing pgBackRest backups stored that you would like to have available in the upgraded cluster, you will need to follow the PVC Renaming Procedure.

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

Step 15 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 16 Scale up to the required number of replicas, as needed.

Congratulations! Your cluster is upgraded and ready to use!

pgBackRest Repo PVC Renaming

If the pgcluster you are upgrading has an existing pgBackRest repo PVC that you would like to continue to use (which is required for existing pgBackRest backups to be accessible by your new cluster), the following renaming procedure will be needed.

Step 1 To start, if your current cluster is "mycluster", the pgBackRest PVC created by version 3.5 of the Postgres Operator will be named "mycluster-backrest-shared-repo". This will need to be renamed to "mycluster-pgbr-repo" to be used in your new cluster.

To begin, save the output description from the pgBackRest PVC:

In 4.0:

```
kubectl -n <namespace> describe pvc mycluster-backrest-shared-repo
```

In 4.1 and later:

```
kubectl -n <namespace> describe pvc mycluster-pgbr-repo
```

for later use when recreating this PVC with the new name. In this output, note the "Volume" name, which is the name of the underlying PV.

Step 2 Now use

```
kubectl -n <namespace> get pv <PV name>
```

to check the "RECLAIM POLICY". If this is not set to "Retain", edit the "persistentVolumeReclaimPolicy" value so that it is set to "Retain" using

```
kubectl -n <namespace> patch pv <PV name> --type='json' -p='[{"op": "replace", "path":
   "/spec/persistentVolumeReclaimPolicy", "value":"Retain"}]'
```

Step 3 Now, delete the PVC:

In 4.0:

```
kubectl -n <namespace> delete pvc mycluster-backrest-shared-repo
```

In 4.1 and later:

```
kubectl -n <namespace> delete pvc mycluster-pgbr-repo
```

Step 4 You will remove the "claimRef" section of the PV with

```
kubectl -n <namespace> patch pv <PV name> --type=json -p='[{"op": "remove", "path":
   "/spec/claimRef"}]'
```

which will make the PV "Available" so it may be reused by the new PVC.

Step 5 Now, create a file with contents similar to the following:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: mycluster-pgbr-repo
  namespace: demo
spec:
  storageClassName: ""
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 2Gi
  volumeMode: Filesystem
  volumeName: "crunchy-pv156"
```

where name matches your new cluster (Remember that this will need to match the "primary PVC" name identified in Step 2 of the upgrade procedure!) and namespace, storageClassName, accessModes, storage, volumeMode and volumeName match your original PVC.

Step 6 Now you can use the new file to recreate your PVC using

```
kubectl -n <namespace> create -f <filename>
```

To check that your PVC is "Bound", run

```
kubectl -n <namespace> get pvc mycluster-pgbr-repo
```

Congratulations, you have renamed your PVC! Once the PVC Status is "Bound", your cluster can be recreated. If you altered the Reclaim Policy on your PV in Step 1, you will want to reset it now.

The PostgreSQL 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 8 or RHEL 8 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
`PGOROOT`	$HOME/postgres-operator | Operator repository location `PGO_CONF_DIR` | PGOROOT/installers/ansible/roles/pgo-operator/

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

Other requirements

- The development host has `git` installed and has cloned the postgres-operator repository. Makefile targets below are run from the repository directory.
- Deploying the Operator will require deployment access to a Kubernetes or OpenShift cluster
- Once you have cloned the git repository, you will need to download the CentOS repository files and GPG keys and place them in the `$PGOROOT/conf` directory. You can do so with the following code:

```
cd $PGOROOT
curl
   https://api.developers.crunchydata.com/downloads/repo/rpm-centos/postgresql12/crunchypg12.repo
   > conf/crunchypg12.repo
```

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

Building

Dependencies

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

```
make setup
```

The setup target ensures the presence of:

- go compiler version 1.13+
- NSQ messaging binaries
- `buildah` OCI image building tool version 1.14.9+

Code Generation

Code generation is leveraged to generate the clients and informers utilized to interact with the various Custom Resources (e.g. `pgclusters`) comprising the PostgreSQL Operator declarative API. Code generation is provided by the Kubernetes code-generator project, and the following Make target is included within the PostgreSQL Operator project to update that code as needed:

```
# Update any generated code:
make generate
```

Therefore, in the event that a Custom Resource defined within the PostgreSQL Operator API (`$PGOROOT/pkg/apis/crunchydata.com`) is updated, the `verify-codegen` target will indicate that an update is needed, and the `update-codegen` target should then be utilized to generate the updated code prior to compiling.

Compile

Please be sure to have your GPG Key and `.repo` file in the `conf` directory before proceeding.

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`

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

Deployment

Now that you have built the PostgreSQL Operator images, you can now deploy them to your Kubernetes cluster by following the [Bash Installation Guide]({{< relref “installation/other/bash.md” >}}).

Testing

Once the PostgreSQL Operator is deployed, you can run the end-to-end regression test suite interface with the PostgreSQL client. You need to ensure that the `pgo` client executable is in your `$PATH`. The test suite can be run using the following commands:

```
cd $PGOROOT/testing/pgo_cli
GO111MODULE=on go test -count=1 -parallel=2 -timeout=30m -v .
```

For more information, please follow the testing README in the source repository.

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.

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/docs/
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.

Crunchy Data announces the release of the PostgreSQL Operator 4.6.0 on January 22, 2021. You can get started with the PostgreSQL Operator with the following commands:

```
kubectl create namespace pgo
kubectl apply -f
    https://raw.githubusercontent.com/CrunchyData/postgres-operator/v4.6.0/installers/kubectl/postgres
```

The PostgreSQL Operator is released in conjunction with the Crunchy Container Suite.

The PostgreSQL Operator 4.6.0 release includes the following software versions upgrades:

- pgBackRest is now at version 2.31
- pgnodemx is now at version 1.0.3
- Patroni is now at version 2.0.1
- pgBadger is now at 11.4

The monitoring stack for the PostgreSQL Operator uses upstream components as opposed to repackaging them. These are specified as part of the PostgreSQL Operator Installer. We have tested this release with the following versions of each component:

- Prometheus: 2.24.0
- Grafana: 6.7.5
- Alertmanager: 0.21.0

This release of the PostgreSQL Operator drops support for PostgreSQL 9.5, which goes EOL in February 2021.

PostgreSQL Operator is tested against Kubernetes 1.17 - 1.20, OpenShift 3.11, OpenShift 4.4+, Google Kubernetes Engine (GKE), Amazon EKS, Microsoft AKS, and VMware Enterprise PKS 1.3+, and works on other Kubernetes distributions as well.

Major Features

Rolling Updates

During the lifecycle of a PostgreSQL cluster, there are certain events that may require a planned restart, such as an update to a "restart required" PostgreSQL configuration setting (e.g. `shared_buffers`) or a change to a Kubernetes Deployment template (e.g. changing the memory request). Restarts can be disruptive in a high availability deployment, which is why many setups employ a "rolling update" strategy (aka a "rolling restart") to minimize or eliminate downtime during a planned restart.

Because PostgreSQL is a stateful application, a simple rolling restart strategy will not work: PostgreSQL needs to ensure that there is a primary available that can accept reads and writes. This requires following a method that will minimize the amount of downtime when the primary is taken offline for a restart.

This release introduces a mechanism for the PostgreSQL Operator to perform rolling updates implicitly on certain operations that change the Deployment templates and explicitly through the `pgo restart` command with the `--rolling` flag. Some of the operations that will trigger a rolling update include:

- Memory resource adjustments
- CPU resource adjustments
- Custom annotation changes
- Tablespace additions
- Adding/removing the metrics sidecar to a PostgreSQL cluster

Please reference the documentation for more details on rolling updates.

Pod Tolerations

Kubernetes Tolerations can help with the scheduling of Pods to appropriate Nodes based upon the taint values of said Nodes. For example, a Kubernetes administrator may set taints on Nodes to restrict scheduling to just the database workload, and as such, tolerations must be assigned to Pods to ensure they can actually be scheduled on thos nodes.

This release introduces the ability to assign tolerations to PostgreSQL clusters managed by the PostgreSQL Operator. Tolerations can be assigned to every instance in the cluster via the `tolerations` attribute on a `pgclusters.crunchydata.com` custom resource, or to individual instances using the `tolerations` attribute on a `pgreplicas.crunchydata.com` custom resource.

Both the `pgo create cluster` and `pgo scale` commands support the `--toleration` flag, which can be used to add one or more tolerations to a cluster. Values accepted by the `--toleration` flag use the following format:

```
rule:Effect
```

where a `rule` can represent existence (e.g. `key`) or equality (`key=value`) and `Effect` is one of `NoSchedule`, `PreferNoSchedule`, or `NoExecute`, e.g:

```
pgo create cluster hippo \
  --toleration=ssd:NoSchedule \
  --toleration=zone=east:NoSchedule
```

Tolerations can also be added and removed from an existing cluster using the `pgo update cluster` , command e.g:

```
pgo update cluster hippo \
  --toleration=zone=west:NoSchedule \
  --toleration=zone=east:NoSchedule-
```

or by modifying the `pgclusters.crunchydata.com` custom resource directly.

For more information on how tolerations work, please refer to the Kubernetes documentation.

Node Affinity Enhancements

Node affinity has been a feature of the PostgreSQL Operator for a long time but has received some significant improvements in this release.

It is now possible to control the node affinity across an entire PostgreSQL cluster as well as individual PostgreSQL instances from a custom resource attribute on the `pgclusters.crunchydata.com` and `pgreplicas.crunchydata.com` CRDs. These attributes use the standard Kubernetes specifications for node affinity and should be familiar to users who have had to set this in applications.

Additionally, this release adds support for both "preferred" and "required" node affinity definitions. Previously, one could achieve required node affinity by modifying a template in the `pgo-config` ConfigMap, but this release makes this process more straightforward.

This release introduces the `--node-affinity-type` flag for the `pgo create cluster`, `pgo scale`, and `pgo restore` commands that allows one to specify the node affinity type for PostgreSQL clusters and instances. The `--node-affinity-type` flag accepts values of `preferred` (default) and `required`. Each instance in a PostgreSQL cluster will inherit its node affinity type from the cluster (`pgo create cluster`) itself, but the type of an individual instance (`pgo scale`) will supersede that value.

The `--node-affinity-type` must be combined with the `--node-label` flag.

TLS for pgBouncer

Since 4.3.0, the PostgreSQL Operator has had support for TLS connections to PostgreSQL clusters and an improved integration with pgBouncer, used for connection pooling and state management. However, the integration with pgBouncer did not support TLS directly: it could be achieved through modifying the pgBouncer Deployment template.

This release brings TLS support for pgBouncer to the PostgreSQL Operator, allowing for communication over TLS between a client and pgBouncer, and pgBouncer and a PostgreSQL server. In other words, the following is now support:

`Client` <= TLS => `pgBouncer` <= TLS => `PostgreSQL`

In other words, to use TLS with pgBouncer, all connections from a client to pgBouncer and from pgBouncer to PostgreSQL **must** be over TLS. Effectively, this is "TLS only" mode if connecting via pgBouncer.

In order to deploy pgBouncer with TLS, the following preconditions must be met:

- TLS **MUST** be enabled within the PostgreSQL cluster.
- pgBouncer and the PostgreSQL **MUST** share the same certificate authority (CA) bundle.

You must have a Kubernetes TLS Secret containing the TLS keypair you would like to use for pgBouncer.

You can enable TLS for pgBouncer using the following commands:

- `pgo create pgbouncer --tls-secret`, where `--tls-secret` specifies the location of the TLS keypair to use for pgBouncer. You **must** already have TLS enabled in your PostgreSQL cluster.
- `pgo create cluster --pgbouncer --pgbouncer-tls-secret`, where `--tls-secret` specifies the location of the TLS keypair to use for pgBouncer. You **must** also specify `--server-tls-secret` and `--server-ca-secret`.

This adds an attribute to the `pgclusters.crunchydata.com` Customer Resource Definition in the `pgBouncer` section called `tlsSecret`, which will store the name of the TLS secret to use for pgBouncer.

By default, connections coming into pgBouncer have a PostgreSQL SSL mode of `require` and connections going into PostgreSQL using `verify-ca`.

Enable/Disable Metrics Collection for PostgreSQL Cluster

A common case is that one creates a PostgreSQL cluster with the Postgres Operator and forget to enable it for monitoring with the `--metrics` flag. Prior to this release, adding the `crunchy-postgres-exporter` to an already running PostgreSQL cluster presented challenges.

This release brings the `--enable-metrics` and `--disable-metrics` introduces to the `pgo update cluster` flags that allow for monitoring to be enabled or disabled on an already running PostgreSQL cluster. As this involves modifying Deployment templates, this action triggers a rolling update that is described in the previous section to limit downtime.

Metrics can also be enabled/disabled using the `exporter` attribute on the `pgclusters.crunchydata.com` custom resource.

This release also changes the management of the PostgreSQL user that is used to collect the metrics. Similar to pgBouncer, the PostgreSQL Operator fully manages the credentials for the metrics collection user. The `--exporter-rotate-password` flag on `pgo update cluster` can be used to rotate the metric collection user's credentials.

Container Image Reduction & Reorganization

Advances in Postgres Operator functionality have allowed for a culling of the number of required container images. For example, functionality that had been broken out into individual container images (e.g. `crunchy-pgdump`) is now consolidated within the `crunchy-postgres` and `crunchy-postgres-ha` containers.

Renamed container images include:

- `pgo-backrest` => `crunchy-pgbackrest`
- `pgo-backrest-repo` => `crunchy-pgbackrest-repo`

Removed container images include:

- `crunchy-admin`
- `crunchy-backrest-restore`
- `crunchy-backup`
- `crunchy-pgbasebackup-restore`
- `crunchy-pgbench`
- `crunchy-pgdump`
- `crunchy-pgrestore`
- `pgo-sqlrunner`
- `pgo-backrest-repo-sync`
- `pgo-backrest-restore`

These changes also include overall organization and build performance optimizations around the container suite.

Breaking Changes

- Metrics collection can now be enabled/disabled using the `exporter` attribute on `pgclusters.crunchydata.com`. The previous method to do so, involving a label buried within a custom resource, no longer works.
- pgBadger can now be enabled/disabled using the `pgBadger` attribute on `pgclusters.crunchydata.com`. The previous method to do so, involving a label buried within a custom resource, no longer works.
- Several additional labels on the `pgclusters.crunchydata.com` CRD that had driven behavior have been moved to attributes. These include:
  - `autofail`, which is now represented by the `disableAutofail` attribute.
  - `service-type`, which is now represented by the `serviceType` attribute.
  - `NodeLabelKey`/`NodeLabelValue`, which is now replaced by the `nodeAffinity` attribute.
  - `backrest-storage-type`, which is now represented with the `backrestStorageTypes` attribute.
- The `--labels` flag on `pgo create cluster` is removed and replaced with the `--label`, which can be specified multiple times. The API endpoint for `pgo create cluster` is also modified: labels must now be passed in as a set of key-value pairs. Please see the "Features" section for more details.
- The API endpoints for `pgo label` and `pgo delete label` is modified to accept a set of key/value pairs for the values of the `--label` flag. The API parameter for this is now called `Labels`. The `pgo upgrade` command will properly moved any data you have in these labels into the correct attributes. You can read more about how to use the various CRD attributes in the Custom Resources section of the documentation.

- The `rootsecretname`, `primarysecretname`, and `usersecretname` attributes on the `pgclusters.crunchydata.com` CRD have been removed. Each of these represented managed Secrets. Additionally, if the managed Secrets are not created at cluster creation time, the Operator will now generate these Secrets.
- The `collectSecretName` attribute on `pgclusters.crunchydata.com` has been removed. The Secret for the metrics collection user is now fully managed by the PostgreSQL Operator.
- There are changes to the `exporter.json` and `cluster-deployment.json` templates that reside within the `pgo-config` ConfigMap that could be breaking to those who have customized those templates. This includes removing the opening comma in the `exporter.json` and removing unneeded match labels on the PostgreSQL cluster Deployment. This is resolved by following the standard upgrade procedure.(https://access.crunchydata.com/documentation/postgres-operator/latest/upgrade/), and only affects new clusters and existing clusters that wish to use the enable/disable metric collection feature. The `affinity.json` entry in the `pgo-config` ConfigMap has been removed in favor of the updated node affinity support.
- Failovers can no longer be controlled by creating a `pgtasks.crunchydata.com` custom resource.
- Remove the `PgMonitorPassword` attribute from `pgo-deployer`. The metric collection user password is managed by the PostgreSQL Operator.
- Policy creation only supports the method of creating the policy from a file/ConfigMap.
- Any pgBackRest variables of the format `PGBACKREST_REPO_` now follow the format `PGBACKREST_REPO1_` to be consistent with what pgBackRest expects.

Features

- Monitoring can now be enabled/disabled during the lifetime of a PostgreSQL cluster using the `pgo update --enable-metrics` and `pgo update --disable-metrics` flag. This can also be modified directly on a custom resource.
- The Service Type of a PostgreSQL cluster can now be updated during the lifetime of a cluster with `pgo update cluster --service-type`. This can also be modified directly on a custom resource.
- The Service Type of pgBouncer can now be independently controlled and set with the `--service-type` flag on `pgo create pgbouncer` and `pgo update pgbouncer`. This can also be modified directly on a custom resource.
- pgBackRest delta restores, which can efficiently restore data as it determines which specific files need to be restored from backup, can now be used as part of the cluster creation method with `pgo create cluster --restore-from`. For example, if a cluster is deleted as such:

```
pgo delete cluster hippo --keep-data --keep-backups
```

It can subsequently be recreated using the delta restore method as such:

```
pgo create cluster hippo --restore-from=hippo
```

Passing in the `--process-max` option to `--restore-opts` can help speed up the restore process based upon the amount of CPU you have available. If the delta restore fails, the PostgreSQL Operator will attempt to perform a full restore.

- `pgo restore` will now first attempt a pgBackRest delta restore, which can significantly speed up the restore time for large databases. Passing in the `--process-max` option to `--backup-opts` can help speed up the restore process based upon the amount of CPU you have available.
- A pgBackRest backup can now be deleted with `pgo delete backup`. A backup name must be specified with the `--target` flag. Please refer to the documentation for how to use this command.
- `pgo create cluster` now accepts a `--label` flag that can be used to specify one or more custom labels for a PostgreSQL cluster. This replaces the `--labels`flag.
- `pgo label` and `pgo delete label` can accept a `--label` flag specified multiple times.
- pgBadger can now be enabled/disabled during the lifetime of a PostgreSQL cluster using the `pgo update --enable-pgbadger` and `pgo update --disable-pgbadger` flag. This can also be modified directly on a custom resource.
- Managed PostgreSQL system accounts and now have their credentials set and rotated with `pgo update user` by including the `--set-system-account-password` flag. Suggested by (@srinathganesh).

Changes

- If not provided at installation time, the Operator will now generate its own `pgo-backrest-repo-config` Secret.
- The `local` storage type option for pgBackRest is deprecated in favor of `posix`, which matches the pgBackRest term. `local` will still continue to work for backwards compatibility purposes.
- PostgreSQL clusters using multi-repository (e.g. `posix` + `s3` at the same time) archiving will now, by default, take backups to both repositories when `pgo backup` is used without additional options.
- If not provided a cluster creation time, the Operator will now generate the PostgreSQL user Secrets required for bootstrap, including the superuser (`postgres`), the replication user (`primaryuser`), and the standard user.
- `crunchy-postgres-exporter` now exposes several pgMonitor metrics related to `pg_stat_statements`.

- When using the `--restore-from` option on `pgo create cluster` to create a new PostgreSQL cluster, the cluster bootstrap Job is now automatically removed if it completes successfully.
- The `pgo failover` command now works without specifying a target: the candidate to fail over to will be automatically selected.
- For clusters that have no healthy instances, `pgo failover` can now force a promotion using the `--force` flag. A `--target` flag must also be specified when using `--force`.
- If a predefined custom ConfigMap for a PostgreSQL cluster (`-pgha-config`) is detected at bootstrap time, the Operator will ensure it properly initializes the cluster.
- Deleting a `pgclusters.crunchydata.com` custom resource will now properly delete a PostgreSQL cluster. If the `pgclusters.crunchyd` custom resource has the annotations `keep-backups` or `keep-data`, it will keep the backups or keep the PostgreSQL data directory respectively. Reported by Leo Khomenko (@lkhomenk).
- PostgreSQL JIT compilation is explicitly disabled on new cluster creation. This prevents a memory leak that has been observed on queries coming from the metrics exporter.
- The credentials for the metrics collection user are now available with `pgo show user --show-system-accounts`.
- The default user for executing scheduled SQL policies is now the Postgres superuser, instead of the replication user.
- Add the `--no-prompt` flag to `pgo upgrade`. The mechanism to disable the prompt verification was already in place, but the flag was not exposed. Reported by (@devopsevd).
- Remove certain characters that causes issues in shell environments from consideration when using the random password generator, which is used to create default passwords or with `--rotate-password`.
- Allow for the `--link-map` attribute for a pgBackRest option, which can help with the restore of an existing cluster to a new cluster that adds an external WAL volume.
- Remove the long deprecated `archivestorage` attribute from the `pgclusters.crunchydata.com` custom resource definition. As this attribute is not used at all, this should have no effect.
- The `ArchiveMode` parameter is now removed from the configuration. This had been fully deprecated for awhile.
- Add an explicit size limit of `64Mi` for the `pgBadger` ephemeral storage mount. Additionally, remove the ephemeral storage mount for the `/recover` mount point as that is not used. Reported by Pierre-Marie Petit (@pmpetit).
- New PostgreSQL Operator deployments will now generate ECDSA keys (P-256, SHA384) for use by the API server.

Fixes

- Ensure custom annotations are applied if the annotations are supposed to be applied globally but the cluster does not have a pgBouncer Deployment.
- Fix issue with UBI 8 / CentOS 8 when running a pgBackRest bootstrap or restore job, where duplicate "repo types" could be set. Specifically, the ensures the name of the repo type is set via the `PGBACKREST_REPO1_TYPE` environmental variable. Reported by Alec Rooney (@alrooney).
- Fix issue where `pgo test` would indicate every Service was a replica if the cluster name contained the word `replica` in it. Reported by Jose Joye (@jose-joye).
- Do not consider Evicted Pods as part of `pgo test`. This eliminates a behavior where faux primaries are considered as part of `pgo test`. Reported by Dennis Jacobfeuerborn (@dennisjac).
- Fix `pgo df` to not fail in the event it tries to execute a command within a dangling container from the bootstrap process when `pgo create cluster --restore-from` is used. Reported by Ignacio J.Ortega (@IJOL).
- `pgo df` will now only attempt to execute in running Pods, i.e. it does not attempt to run in evicted Pods. Reported by (@kseswar).
- Ensure the sync replication ConfigMap is removed when a cluster is deleted.
- Fix crash in shutdown logic when attempting to shut down a cluster where no primaries exist. Reported by Jeffrey den Drijver (@JeffreyDD).
- Fix syntax in recovery check command which could lead to failures when manually promoting a standby cluster. Reported by (@SockenSalat).
- Fix potential race condition that could lead to a crash in the Operator boot when an error is issued around loading the `pgo-config` ConfigMap. Reported by Aleksander Roszig (@AleksanderRoszig).
- Do not trigger a backup if a standby cluster fails over. Reported by (@aprilito1965).
- Ensure pgBouncer Secret is created when adding it to a standby cluster.
- Generally improvements to initialization of a standby cluster.
- Remove legacy `defaultMode` setting on the volume instructions for the pgBackRest repo Secret as the `readOnly` setting is used on the mount itself. Reported by (@szhang1).
- Ensure proper label parsing based on Kubernetes rules and that it is consistently applied across all functionality that uses labels. Reported by José Joye (@jose-joye).
- The logger no longer defaults to using a log level of `DEBUG`.
- Autofailover is no longer disabled when an `rmdata` Job is run, enabling a clean database shutdown process when deleting a PostgreSQL cluster.
- Allow for `Restart` API server permission to be explicitly set. Reported by Aleksander Roszig (@AleksanderRoszig).
- Update `pgo-target` permissions to match expectations for modern Kubernetes versions.
- Major upgrade container now includes references for `pgnodemx`.

- During a major upgrade, ensure permissions are correct on the old data directory before running `pg_upgrade`.
- The metrics stack installer is fixed to work in environments that may not have connectivity to the Internet ("air gapped"). Reported by (@eliranw).

Crunchy Data announces the release of the PostgreSQL Operator 4.5.1 on November 13, 2020.

The PostgreSQL Operator is released in conjunction with the Crunchy Container Suite.

PostgreSQL Operator 4.5.1 release includes the following software versions upgrades:

- PostgreSQL is now at versions 13.1, 12.5, 11.10, 10.15, 9.6.20, and 9.5.24.
- Patroni is now at version 2.0.1.
- PL/Perl can now be used in the PostGIS-enabled containers.

Changes

- Simplified creation of a PostgreSQL cluster from a `pgcluster` resource. A user no longer has to provide a pgBackRest repository Secret: the Postgres Operator will now automatically generate this.
- The exposed ports for Services associated with a cluster is now available from the `pgo show cluster` command.
- If the `pgo-config` ConfigMap is not created during the installation of the Postgres Operator, the Postgres Operator will generate one when it initializes.
- Providing a value for `pgo_admin_password` in the installer is now optional. If no value is provided, the password for the initial administrative user is randomly generated.
- Added an example for how to create a PostgreSQL cluster that uses S3 for pgBackRest backups via a custom resource.

Fixes

- Fix readiness check for a standby leader. Previously, the standby leader would not report as ready, even though it was. Reported by Alec Rooney (@alrooney).
- Proper determination if a `pgcluster` custom resource creation has been processed by its corresponding Postgres Operator controller. This prevents the custom resource from being run by the creation logic multiple times.
- Prevent `initdb` (cluster reinitialization) from occurring if the PostgreSQL container cannot initialize while bootstrapping from an existing PGDATA directory.
- Fix issue with UBI 8 / CentOS 8 when running a pgBackRest bootstrap or restore job, where duplicate "repo types" could be set. Specifically, the ensures the name of the repo type is set via the `PGBACKREST_REPO1_TYPE` environmental variable. Reported by Alec Rooney (@alrooney).
- Ensure external WAL and Tablespace PVCs are fully recreated during a restore. Reported by (@aurelien43).
- Ensure `pgo show backup` will work regardless of state of any of the PostgreSQL clusters. This pulls the information directly from the pgBackRest Pod itself. Reported by (@saltenhub).
- Ensure that sidecars (e.g. metrics collection, pgAdmin 4, pgBouncer) are deployable when using the PostGIS-enabled PostgreSQL image. Reported by Jean-Denis Giguère (@jdenisgiguere).
- Allow for special characters in pgBackRest environmental variables. Reported by (@SockenSalat).
- Ensure password for the `pgbouncer` administrative user stays synchronized between an existing Kubernetes Secret and PostgreSQL should the pgBouncer be recreated.
- When uninstalling an instance of the Postgres Operator in a Kubernetes cluster that has multiple instances of the Postgres Operator, ensure that only the requested instance to be uninstalled is the one that's uninstalled.
- The logger no longer defaults to using a log level of `DEBUG`.

Crunchy Data announces the release of the PostgreSQL Operator 4.5.0 on October 2, 2020.

The PostgreSQL Operator is released in conjunction with the Crunchy Container Suite.

The PostgreSQL Operator 4.5.0 release includes the following software versions upgrades:

- Add support for PostgreSQL 13.
- pgBackRest is now at version 2.29.
- postgres_exporter is now at version 0.8.0
- pgMonitor support is now at 4.4
- pgnodemx is now at version 1.0.1
- wal2json is now at version 2.3
- Patroni is now at version 2.0.0

Additionally, PostgreSQL Operator 4.5.0 introduces support for the CentOS 8 and UBI 8 base container images. In addition to using the newer operating systems, this enables support for TLS 1.3 when connecting to PostgreSQL. This release also moves to building the containers using Buildah 1.14.9.

The monitoring stack for the PostgreSQL Operator has shifted to use upstream components as opposed to repackaging them. These are specified as part of the PostgreSQL Operator Installer. We have tested this release with the following versions of each component:

- Prometheus: 2.20.0
- Grafana: 6.7.4
- Alertmanager: 0.21.0

PostgreSQL Operator is tested with Kubernetes 1.15 - 1.19, OpenShift 3.11+, OpenShift 4.4+, Google Kubernetes Engine (GKE), Amazon EKS, and VMware Enterprise PKS 1.3+.

Major Features

PostgreSQL Operator Monitoring

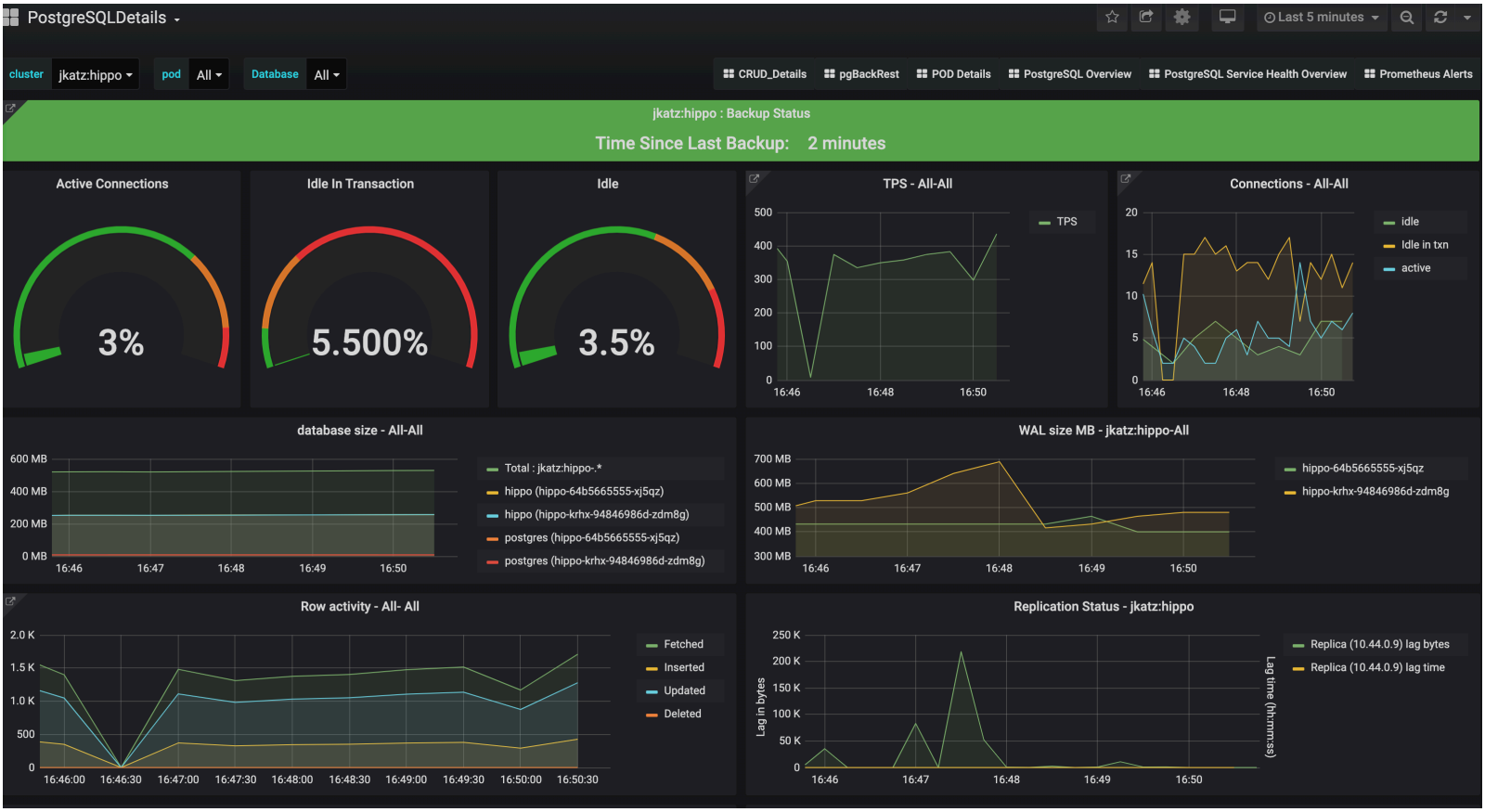

Figure 31: PostgreSQL Operator Monitoring

This release makes several changes to the PostgreSQL Operator Monitoring solution, notably making it much easier to set up a turnkey PostgreSQL monitoring solution with the PostgreSQL Operator using the open source pgMonitor project.

pgMonitor combines insightful queries for PostgreSQL with several proven tools for statistics collection, data visualization, and alerting to allow one to deploy a turnkey monitoring solution for PostgreSQL. The pgMonitor 4.4 release added support for Kubernetes environments, particularly with the pgnodemx that allows one to get host-like information from the Kubernetes Pod a PostgreSQL instance is deployed within.

PostgreSQL Operator 4.5 integrates with pgMonitor to take advantage of its Kubernetes support, and provides the following visualized metrics out-of-the-box:

- Pod metrics (CPU, Memory, Disk activity)
- PostgreSQL utilization (Database activity, database size, WAL size, replication lag)
- Backup information, including last backup and backup size
- Network utilization (traffic, saturation, latency)
- Alerts (uptime et al.)

More metrics and visualizations will be added in future releases. You can further customize these to meet the needs for your enviornment.

PostgreSQL Operator 4.5 uses the upstream packages for Prometheus, Grafana, and Alertmanager. Those using earlier versions of monitoring provided with the PostgreSQL Operator will need to switch to those packages. The tested versions of these packages for PostgreSQL Operator 4.5 include:

- Prometheus (2.20.0)
- Grafana (6.7.4)
- Alertmanager (0.21.0)

You can find out how to install PostgreSQL Operator Monitoring in the installation section:

https://access.crunchydata.com/documentation/postgres-operator/latest/latest/installation/metrics/

Customizing pgBackRest via ConfigMap

pgBackRest powers the disaster recovery capabilities of PostgreSQL clusters deployed by the PostgreSQL Operator. While the PostgreSQL Operator provides many toggles to customize a pgBackRest configuration, it can be easier to do so directly using the pgBackRest configuration file format.

This release adds the ability to specify the pgBackRest configuration from either a ConfigMap or Secret by using the `pgo create cluster --pgbackrest-custom-config` flag, or by setting the `BackrestConfig` attributes in the `pgcluster` CRD. Setting this allows any pgBackRest resource (Pod, Job etc.) to leverage this custom configuration.

Note that some settings will be overriden by the PostgreSQL Operator regardless of the settings of a customized pgBackRest configuration file due to the nature of how the PostgreSQL instances managed by the Operator access pgBackRest. However, these are typically not the settings that one wants to customize.

Apply Custom Annotations to Managed Deployments

It is now possible to add custom annotations to the Deployments that the PostgreSQL Operator manages. These include:

- PostgreSQL instances
- pgBackRest repositories
- pgBouncer instances

Annotations are applied on a per-cluster basis, and can be set either for all the managed Deployments within a cluster or individual Deployment groups. The annotations can be set as part of the `Annotations` section of the pgcluster specification.

This also introduces several flags to the `pgo` client that help with the management of the annotations. These flags are available on `pgo create cluster` and `pgo update cluster` commands and include:

- `--annotation` - apply annotations on all managed Deployments
- `--annotation-postgres` - applies annotations on all managed PostgreSQL Deployments
- `--annotation-pgbackrest` - applies annotations on all managed pgBackRest Deployments
- `--annotation-pgbouncer` - applies annotations on all managed pgBouncer Deployments

These flags work similarly to how one manages annotations and labels from `kubectl`. To add an annotation, one follows the format:

```
--annotation=key=value
```

To remove an annotation, one follows the format:

```
--annotation=key-
```

Breaking Changes

- The `crunchy-collect` container, used for metrics collection is renamed to `crunchy-postgres-exporter`
- The `backrest-restore-<fromClusterName>-to-<toPVC>` pgtask has been renamed to `backrest-restore-<clusterName>`. Additionally, the following parameters no longer need to be specified for the pgtask:
- pgbackrest-stanza
- pgbackrest-db-path
- pgbackrest-repo-path
- pgbackrest-repo-host

- backrest-s3-verify-tls
- When a restore job completes, it now emits the message `restored Primary created` instead of `restored PVC created`.
- The `toPVC` parameter has been removed from the restore request endpoint.
- Restore jobs using `pg_restore` no longer have `from-<pvcName>` in their names.
- The `pgo-backrest-restore` container has been retired.
- The `pgo load` command has been removed. This also retires the `pgo-load` container.
- The `crunchy-prometheus` and `crunchy-grafana` containers are now removed. Please use the corresponding upstream containers.

Features

- The metrics collection container now has configurable resources. This can be set as part of the custom resource workflow as well as from the `pgo` client when using the following command-line arguments:
- CPU resource requests:
  - `pgo create cluster --exporter-cpu`
  - `pgo update cluster --exporter-cpu`
- CPU resource limits:
  - `pgo create cluster --exporter-cpu-limit`
  - `pgo update cluster --exporter-cpu-limit`
- Memory resource requests:
  - `pgo create cluster --exporter-memory`
  - `pgo update cluster --exporter-memory`
- Memory resource limits:
  - `pgo create cluster --exporter-memory-limit`
  - `pgo update cluster --exporter-memory-limit`
- Support for TLS 1.3 connections to PostgreSQL when using the UBI 8 and CentOS 8 containers
- Added support for the `pgnodemx` extension which makes container-level metrics (CPU, memory, storage utilization) available via a PostgreSQL-based interface.

Changes

- The PostgreSQL Operator now supports the default storage class that is available within a Kubernetes cluster. The installers are updated to use the default storage class by default.
- The `pgo restore` methodology is changed to mirror the approach taken by `pgo create cluster --restore-from` that was introduced in the previous release. While `pgo restore` will still perform a "restore in-place", it will now take the following actions:
- Any existing persistent volume claims (PVCs) in a cluster removed.
- New PVCs are initialized and the data from the PostgreSQL cluster is restored based on the parameters specified in `pgo restore`.
- Any customizations for the cluster (e.g. custom PostgreSQL configuration) will be available.
- This also fixes several bugs that were reported with the `pgo restore` functionality, some of which are captured further down in these release notes.
- Connections to pgBouncer can now be passed along to the default `postgres` database. If you have a pre-existing pgBouncer Deployment, the most convenient way to access this functionality is to redeploy pgBouncer for that PostgreSQL cluster (`pgo delete pgbouncer` + `pgo create pgbouncer`). Suggested by (@lgarcia11).
- The Downward API is now available to PostgreSQL instances.
- The pgBouncer `pgbouncer.ini` and `pg_hba.conf` have been moved from the pgBouncer Secret to a ConfigMap whose name follows the pattern `<clusterName>-pgbouncer-cm`. These are mounted as part of a project volume in conjunction with the current pgBouncer Secret.
- The `pgo df` command will round values over 1000 up to the next unit type, e.g. `1GiB` instead of `1024MiB`.

Fixes

- Ensure that if a PostgreSQL cluster is recreated from a PVC with existing data that it will apply any custom PostgreSQL configuration settings that are specified.
- Fixed issues with PostgreSQL replica Pods not becoming ready after running `pgo restore`. This fix is a result of the change in methodology for how a restore occurs.
- The `pgo scaledown` now allows for the removal of replicas that are not actively running.
- The `pgo scaledown --query` command now shows replicas that may not be in an active state.

- The pgBackRest URI style defaults to `host` if it is not set.
- pgBackRest commands can now be executed even if there are multiple pgBackRest Pods available in a Deployment, so long as there is only one "running" pgBackRest Pod. Reported by Rubin Simons (@rubin55).
- Ensure pgBackRest S3 Secrets can be upgraded from PostgreSQL Operator 4.3.
- Ensure pgBouncer Port is derived from the cluster's port, not the Operator configuration defaults.
- External WAL PVCs are only removed for the replica they are targeted for on a scaledown. Reported by (@dakine1111).
- When deleting a cluster with the `--keep-backups` flag, ensure that backups that were created via `--backup-type=pgdump` are retained.
- Return an error if a cluster is not found when using `pgo df` instead of timing out.
- pgBadger now has a default memory limit of 64Mi, which should help avoid a visit from the OOM killer.
- The Postgres Exporter now works if it is deployed in a TLS-only environment, i.e. the `--tls-only` flag is set. Reported by (@shuhanfan).
- Fix `pgo label` when applying multiple labels at once.
- Fix `pgo create pgorole` so that the expression `--permissions=*` works.
- The `operator` container will no longer panic if all Deployments are scaled to `0` without using the `pgo update cluster <mycluster> --shutdown` command.

Crunchy Data announces the release of the PostgreSQL Operator 4.4.1 on August 17, 2020.

The PostgreSQL Operator is released in conjunction with the Crunchy Container Suite.

The PostgreSQL Operator 4.4.1 release includes the following software versions upgrades:

- The PostgreSQL containers now use versions 12.4, 11.9, 10.14, 9.6.19, and 9.5.23

PostgreSQL Operator is tested with Kubernetes 1.13 - 1.18, OpenShift 3.11+, OpenShift 4.3+, Google Kubernetes Engine (GKE), and VMware Enterprise PKS 1.3+.

Fixes

- The pgBackRest URI style defaults to `host` if it is not set.
- Fix `pgo label` when applying multiple labels at once.
- pgBadger now has a default memory limit of 64Mi, which should help avoid a visit from the OOM killer.
- Fix `pgo create pgorole` so that the expression `--permissions=*` works.

Crunchy Data announces the release of the PostgreSQL Operator 4.4.0 on July 17, 2020.

The PostgreSQL Operator is released in conjunction with the Crunchy Container Suite.

The PostgreSQL Operator 4.4.0 release includes the following software versions upgrades:

- PostGIS 3.0 is now supported. There is now a manual upgrade path between PostGIS containers.
- pgRouting is now included in the PostGIS containers.
- pgBackRest is now at version 2.27.
- pgBouncer is now at version 1.14.

PostgreSQL Operator is tested with Kubernetes 1.15 - 1.18, OpenShift 3.11+, OpenShift 4.4+, Google Kubernetes Engine (GKE), and VMware Enterprise PKS 1.3+.

Major Features

- Create New PostgreSQL Clusters from pgBackRest Repositories
- Improvements to RBAC Reconciliation.
- TLS Authentication for PostgreSQL Instances.
- A Helm Chart is now available and support for deploying the PostgreSQL Operator.

Create New PostgreSQL Clusters from pgBackRest Repositories

A technique frequently used in PostgreSQL data management is to have a pgBackRest repository that can be used to create new PostgreSQL clusters. This can be helpful for a variety of purposes:

- Creating a development or test database from a production data set
- Performing a point-in-time-restore on a database that is separate from the primary database

and more.

This can be accomplished with the following new flags on `pgo create cluster`:

- `--restore-from`: used to specify the name of the pgBackRest repository to restore from via the name of the PostgreSQL cluster (whether the PostgreSQL cluster is active or not).
- `--restore-opts`: used to specify additional options like the ones specified to `pgbackrest restore` (e.g. `--type` and `--target` if performing a point-in-time-recovery).

Only one restore can be performed against a pgBackRest repository at a given time.

RBAC Reconciliation

PostgreSQL Operator 4.3 introduced a change that allows for the Operator to manage the role-based access controls (RBAC) based upon the Namespace Operating mode that is selected. This ensures that the PostgreSQL Operator is able to function correctly within the Namespace or Namespaces that it is permitted to access. This includes Service Accounts, Roles, and Role Bindings within a Namespace.

PostgreSQL Operator 4.4 removes the requirements of granting the PostgreSQL Operator `bind` and `escalate` privileges for being able to reconcile its own RBAC, and further defines which RBAC is specifically required to use the PostgreSQL Operator (i.e. the removal of wildcard `*` privileges). The permissions that the PostgreSQL Operator requires to perform the reconciliation are assigned when it is deployed and is a function of which `NAMESPACE_MODE` is selected (`dynamic`, `readonly`, or `disabled`).

This change renames the `DYNAMIC_RBAC` parameter in the installer to `RECONCILE_RBAC` and is set to `true` by default.

For more information on how RBAC reconciliation works, please visit the RBAC reconciliation documentation.

TLS Authentication for PostgreSQL Instances

Certificate-based authentication is a powerful PostgreSQL feature that allows for a PostgreSQL client to authenticate using a TLS certificate. While there are a variety of permutations for this can be set up, we can at least create a standardized way for enabling the replication connection to authenticate with a certificate, as we do have a known certificate authority.

PostgreSQL Operator 4.4 introduces the `--replication-tls-secret` flag on the `pgo create cluster` command, which, if specified and if the prerequisites are specified (`--server-tls-secret` and `--server-ca-secret`), then the replication account ("primaryuser") is configured to use certificate-based authentication. Combine with `--tls-only` for powerful results.

Note that the common name (CN) on the certificate MUST be "primaryuser", otherwise one must specify a mapping in a `pg_ident` configuration block to map to "primary" user.

When mounted to the container, the connection `sslmode` that the replication user uses is set to `verify-ca` by default. We can make that guarantee based on the certificate authority that is being mounted. Using `verify-full` would cause the Operator to make assumptions about the cluster that we cannot make, and as such a custom `pg_ident` configuration block is needed for that. However, using `verify-full` allows for mutual authentication between primary and replica.

Breaking Changes

- The parameter to set the RBAC reconciliation settings is renamed to `RECONCILE_RBAC` (from `DYNAMIC_RBAC`).

Features

- Added support for using the URI path style feature of pgBackRest. This includes:
- Adding the `BackrestS3URIStyle` configuration parameter to the PostgreSQL Operator ConfigMap (`pgo.yaml`), which accepts the values of `host` or `path`.
- Adding the `--pgbackrest-s3-uri-style` flag to `pgo create cluster`, which accepts values of `host` or `path`.
- Added support to disable TLS verification when connecting to a pgBackRest repository. This includes:
- Adding the `BackrestS3VerifyTLS` configuration parameter to the PostgreSQL Operator ConfigMap (`pgo.yaml`). Defaults to `true`.
- Adding the `--pgbackrest-s3-verify-tls` flag to `pgo create cluster`, which accepts values of `true` or `false`.
- Perform a `pg_dump` from a specific database using the `--database` flag when using `pgo backup` with `--backup-type=pgdump`.
- Restore a `pg_dump` to a specific database using the `--pgdump-database` flag using `pgo restore` when `--backup-type=pgdump` is specified.

- Allow for support of authentication parameters in the `pgha-config` (e.g. `sslmode`). See the documentation for words of caution on using these.
- Add the `--client` flag to `pgo version` to output the client version of `pgo`.
- A Helm Chart using Helm v3 is now available.

Changes

- `pgo clone` is now deprecated. For a better cloning experience, please use `pgo create cluster --restore-from`
- The PostgreSQL cluster scope is now utilized to identify and sync the ConfigMap responsible for the DCS for a PostgreSQL cluster.
- The `PGMONITOR_PASSWORD` is now populated by an environmental variable secret. This environmental variable is only set on a primary instance as it is only needed at the time a PostgreSQL cluster is initialized.
- Remove "Operator Start Time" from `pgo status` as it is more convenient and accurate to get this information from `kubectl` and the like, and it was not working due to RBAC privileges. (Reported by @mw-0).
- Removed unused pgcluster attributes `PrimaryHost` and `SecretFrom`.
- `pgo-rmdata` container no longer runs as the `root` user, but as `daemon` (UID 2)
- Remove dependency on the `expenv` binary that was included in the PostgreSQL Operator release. All `expenv` calls were either replaced with the native `envsubst` program or removed.

Fixes

- Add validation to ensure that limits for CPU/memory are greater-than-or-equal-to the requests. This applies to any command that can set a limit/request.
- Ensure PVC capacities are being accurately reported when using `pgo show cluster`
- Ensure WAL archives are pushed to all repositories when pgBackRest is set to use both a local and a S3-based repository
- Silence expected error conditions when a pgBackRest repository is being initialized.
- Deployments with `pgo-deployer` using the default file with `hostpathstorage` will now successfully deploy PostgreSQL clusters without any adjustments.
- Add the `watch` permissions to the `pgo-deployer` ServiceAccount.
- Ensure the PostgreSQL Operator can be uninstalled by adding `list` verb ClusterRole privileges to several Kubernetes objects.
- Ensure `client-setup.sh` executes to completion if existing PostgreSQL Operator credentials exist that were created by a different installation method.
- Ensure `client-setup.sh` works with when there is an existing `pgo` client in the install path.
- Update the documentation to properly name `CCP_IMAGE_PULL_SECRET_MANIFEST` and `PGO_IMAGE_PULL_SECRET_MANIFEST` in the `pgo-deployer` configuration.
- Bring up the correct number of pgBouncer replicas when `pgo update cluster --startup` is issued.
- Fixed issue where `pgo scale` would not work after `pgo update cluster --shutdown` and `pgo update cluster --startup` were run.
- Ensure `pgo scaledown` deletes external WAL volumes from the replica that is removed.
- Fix for PostgreSQL cluster startup logic when performing a restore.
- Several fixes for selecting default storage configurations and sizes when using the `pgo-deployer` container. These include #1, #4, and #8.
- Do not consider non-running Pods as primary Pods when checking for multiple primaries (Reported by @djcooklup).
- Fix race condition that could occur while `pgo upgrade` was running while a HA configuration map attempted to sync. (Reported by Paul Heinen @v3nturetheworld).
- The custom setup example was updated to reflect the current state of bootstrapping the PostgreSQL container.
- Silence "ConfigMap not found" error messages that occurred during PostgreSQL cluster initialization, as these were not real errors.
- Fix an issue with controller processing, which could manifest in PostgreSQL clusters not being deleted.
- Eliminate `gcc` from the `postgres-ha` and `pgadmin4` containers.

Crunchy Data announces the release of the PostgreSQL Operator 4.3.3 on August 17, 2020.

The PostgreSQL Operator is released in conjunction with the Crunchy Container Suite.

The PostgreSQL Operator 4.3.3 release includes the following software versions upgrades:

- The PostgreSQL containers now use versions 12.4, 11.9, 10.14, 9.6.19, and 9.5.23
- pgBouncer is now at version 1.14.

PostgreSQL Operator is tested with Kubernetes 1.13 - 1.18, OpenShift 3.11+, OpenShift 4.3+, Google Kubernetes Engine (GKE), and VMware Enterprise PKS 1.3+.

Changes

- Perform a `pg_dump` from a specific database using the `--database` flag when using `pgo backup` with `--backup-type=pgdump`.
- Restore a `pg_dump` to a specific database using the `--pgdump-database` flag using `pgo restore` when `--backup-type=pgdump` is specified.
- Add the `--client` flag to `pgo version` to output the client version of `pgo`.
- The PostgreSQL cluster scope is now utilized to identify and sync the ConfigMap responsible for the DCS for a PostgreSQL cluster.
- The `PGMONITOR_PASSWORD` is now populated by an environmental variable secret. This environmental variable is only set on a primary instance as it is only needed at the time a PostgreSQL cluster is initialized.
- Remove "Operator Start Time" from `pgo status` as it is more convenient and accurate to get this information from `kubectl` and the like, and it was not working due to RBAC privileges. (Reported by @mw-0).
- `pgo-rmdata` container no longer runs as the `root` user, but as `daemon` (UID 2)
- Remove dependency on the `expenv` binary that was included in the PostgreSQL Operator release. All `expenv` calls were either replaced with the native `envsubst` program or removed.

Fixes

- Add validation to ensure that limits for CPU/memory are greater-than-or-equal-to the requests. This applies to any command that can set a limit/request.
- Ensure WAL archives are pushed to all repositories when pgBackRest is set to use both a local and a S3-based repository
- Silence expected error conditions when a pgBackRest repository is being initialized.
- Add the `watch` permissions to the `pgo-deployer` ServiceAccount.
- Ensure `client-setup.sh` works with when there is an existing `pgo` client in the install path
- Ensure the PostgreSQL Operator can be uninstalled by adding `list` verb ClusterRole privileges to several Kubernetes objects.
- Bring up the correct number of pgBouncer replicas when `pgo update cluster --startup` is issued.
- Fixed issue where `pgo scale` would not work after `pgo update cluster --shutdown` and `pgo update cluster --startup` were run.
- Ensure `pgo scaledown` deletes external WAL volumes from the replica that is removed.
- Fix for PostgreSQL cluster startup logic when performing a restore.
- Do not consider non-running Pods as primary Pods when checking for multiple primaries (Reported by @djcooklup).
- Fix race condition that could occur while `pgo upgrade` was running while a HA configuration map attempted to sync. (Reported by Paul Heinen @v3nturetheworld).
- Silence "ConfigMap not found" error messages that occurred during PostgreSQL cluster initialization, as these were not real errors.
- Fix an issue with controller processing, which could manifest in PostgreSQL clusters not being deleted.
- Eliminate `gcc` from the `postgres-ha` and `pgadmin4` containers.
- Fix `pgo label` when applying multiple labels at once.

Crunchy Data announces the release of the PostgreSQL Operator 4.3.2 on May 27, 2020.

The PostgreSQL Operator is released in conjunction with the Crunchy Container Suite.

Version 4.3.2 of the PostgreSQL Operator contains bug fixes to the installer container and changes to how CPU/memory requests and limits can be specified.

PostgreSQL Operator is tested with Kubernetes 1.13 - 1.18, OpenShift 3.11+, OpenShift 4.3+, Google Kubernetes Engine (GKE), and VMware Enterprise PKS 1.3+.

Changes

Resource Limit Flags

PostgreSQL Operator 4.3.0 introduced some new options to tune the resource requests for PostgreSQL instances under management and other associated deployments, including pgBackRest and pgBouncer. From some of our learnings of running PostgreSQL in Kubernetes, we heavily restricted how the limits on the Pods could be set, and tied them to be the same as the requests.

Due to feedback from a variety of sources, this caused more issues than it helped, and as such, we decided to introduce a breaking change into a patch release and remove the `--enable-*-limit` and `--disable-*-limit` series of flags and replace them with flags that allow you to specifically choose CPU and memory limits.

This release introduces several new flags to various commands, including:

- `pgo create cluster --cpu-limit`
- `pgo create cluster --memory-limit`

- `pgo create cluster --pgbackrest-cpu-limit`
- `pgo create cluster --pgbackrest-memory-limit`
- `pgo create cluster --pgbouncer-cpu-limit`
- `pgo create cluster --pgbouncer-memory-limit`
- `pgo update cluster --cpu-limit`
- `pgo update cluster --memory-limit`
- `pgo update cluster --pgbackrest-cpu-limit`
- `pgo update cluster --pgbackrest-memory-limit`
- `pgo create pgbouncer --cpu-limit`
- `pgo create pgbouncer --memory-limit`
- `pgo update pgbouncer --cpu-limit`
- `pgo update pgbouncer --memory-limit`

Additionally, these values can be modified directly in a pgcluster Custom Resource and the PostgreSQL Operator will react and make the modifications.

Other Changes

- The `pgo-deployer` container can now run using an aribtrary UID.
- For deployments of the PostgreSQL Operator using the `pgo-deployer` container to OpenShift 3.11 environments, a new template YAML file, `postgresql-operator-ocp311.yml` is provided. This YAML file requires that the `pgo-deployer` is run with `cluster-admin` role for OpenShift 3.11 environments due to the lack of support of the `escalate` RBAC verb. Other environments (e.g. Kubernetes, OpenShift 4+) still do not require `cluster-admin`.
- Allow for the resumption of download the `pgo` client if the `client-setup.sh` script gets interrupted. Contributed by Itay Grudev (@itay-grudev).

Fixes

- The `pgo-deployer` container now assigns the required Service Account all the appropriate `get` RBAC privileges via the `postgres-operator.yml` file that it needs to properly install. This allows the `install` functionality to properly work across multiple runs.
- For OpenShift deploymments, the `pgo-deployer` leverages version 4.4 of the `oc` client.
- Use numeric UIDs for users in the PostgreSQL Operator management containers to support `MustRunAsNonRoot` Pod Security Policies and the like. Reported by Olivier Beyler (@obeyler).

Crunchy Data announces the release of the PostgreSQL Operator 4.3.1 on May 18, 2020.

The PostgreSQL Operator is released in conjunction with the Crunchy Container Suite.

The PostgreSQL Operator 4.3.1 release includes the following software versions upgrades:

- The PostgreSQL containers now use versions 12.3, 11.8, 10.13, 9.6.18, and 9.5.22

PostgreSQL Operator is tested with Kubernetes 1.13 - 1.18, OpenShift 3.11+, OpenShift 4.3+, Google Kubernetes Engine (GKE), and VMware Enterprise PKS 1.3+.

Changes

Initial Support for SCRAM

SCRAM is a password authentication method in PostgreSQL that has been available since PostgreSQL 10 and is considered to be superior to the `md5` authentication method. The PostgreSQL Operator now introduces support for SCRAM on the `pgo create user` and `pgo update user` commands by means of the `--password-type` flag. The following values for `--password-type` will select the following authentication methods:

- `--password-type=""`, `--password-type="md5"` => md5
- `--password-type="scram"`, `--password-type="scram-sha-256"` => SCRAM-SHA-256

In turn, the PostgreSQL Operator will hash the passwords based on the chosen method and store the computed hash in PostgreSQL.

When using SCRAM support, it is important to note the following observations and limitations:

- When using one of the password modifications commands on `pgo update user` (e.g. `--password`, `--rotate-password`, `--expires`) with the desire to keep the persisted password using SCRAM, it is necessary to specify the "–password-type=scram-sha-256" directive.
- SCRAM does not work with the current pgBouncer integration with the PostgreSQL Operator. pgBouncer presently supports only one password-based authentication type at a time. Additionally, to enable support for SCRAM, pgBouncer would require a list of plaintext passwords to be stored in a file that is accessible to it. Future work can evaluate how to leverage SCRAM support with pgBouncer.

`pgo restart` and `pgo reload`

This release introduces the `pgo restart` command, which allow you to perform a PostgreSQL restart on one or more instances within a PostgreSQL cluster.

You restart all instances at the same time using the following command:

```
pgo restart hippo
```

or specify a specific instance to restart using the `--target` flag (which follows a similar behavior to the `--target` flag on `pgo scaledown` and `pgo failover`):

```
pgo restart hippo --target=hippo-abcd
```

The restart itself is performed by calling the Patroni `restart` REST endpoint on the specific instance (primary or replica) being restarted.

As with the `pgo failover` and `pgo scaledown` commands it is also possible to specify the `--query` flag to query instances available for restart:

```
pgo restart mycluster --query
```

With the new `pgo restart` command, using `--query` flag with the `pgo failover` and `pgo scaledown` commands include the `PENDING RESTART` information, which is now returned with any replication information.

This release allows for the `pgo reload` command to properly reloads all instances (i.e. the primary and all replicas) within the cluster.

Dynamic Namespace Mode and Older Kubernetes Versions

The dynamic namespace mode (e.g. `pgo create namespace` + `pgo delete namespace`) provides the ability to create and remove Kubernetes namespaces and automatically add them unto the purview of the PostgreSQL Operator. Through the course of fixing usability issues with working with the other namespaces modes (`readonly`, `disabled`), a change needed to be introduced that broke compatibility with Kubernetes 1.12 and earlier.

The PostgreSQL Operator still supports managing PostgreSQL Deployments across multiple namespaces in Kubernetes 1.12 and earlier, but only with `readonly` mode. In `readonly` mode, a cluster administrator needs to create the namespace and the RBAC needed to run the PostgreSQL Operator in that namespace. However, it is now possible to define the RBAC required for the PostgreSQL Operator to manage clusters in a namespace via a ServiceAccount, as described in the Namespace section of the documentation.

The usability change allows for one to add namespace to the PostgreSQL Operator's purview (or deploy the PostgreSQL Operator within a namespace) and automatically set up the appropriate RBAC for the PostgreSQL Operator to correctly operate.

Other Changes

- The RBAC required for deploying the PostgreSQL Operator is now decomposed into the exact privileges that are needed. This removes the need for requiring a `cluster-admin` privilege for deploying the PostgreSQL Operator. Reported by (@obeyler).
- With namespace modes `disabled` and `readonly`, the PostgreSQL Operator will now dynamically create the required RBAC when a new namespace is added if that namespace has the RBAC defined in `local-namespace-rbac.yaml`. This occurs when `PGO_DYNAMIC_NAMESPACE` is set to `true`.
- If the PostgreSQL Operator has permissions to manage it's own RBAC within a namespace, it will now reconcile and auto-heal that RBAC as needed (e.g. if it is invalid or has been removed) to ensure it can properly interact with and manage that namespace.
- Add default CPU and memory limits for the metrics collection and pgBadger sidecars to help deployments that wish to have a Pod QoS of `Guaranteed`. The metrics defaults are 100m/24Mi and the pgBadger defaults are 500m/24Mi. Reported by (@jose-joye).
- Introduce `DISABLE_FSGROUP` option as part of the installation. When set to `true`, this does not add a FSGroup to the Pod Security Context when deploying PostgreSQL related containers or pgAdmin 4. This is helpful when deploying the PostgreSQL Operator in certain environments, such as OpenShift with a `restricted` Security Context Constraint. Defaults to `false`.
- Remove the custom Security Context Constraint (SCC) that would be deployed with the PostgreSQL Operator, so now the PostgreSQL Operator can be deployed using default OpenShift SCCs (e.g. "restricted", though note that `DISABLE_FSGROUP` will need to be set to `true` for that). The example PostgreSQL Operator SCC is left in the `examples` directory for reference.
- When `PGO_DISABLE_TLS` is set to `true`, then `PGO_TLS_NO_VERIFY` is set to `true`.

- Some of the `pgo-deployer` environmental variables that we not needed to be set by a user were internalized. These include `ANSIBLE_CONFIG` and `HOME`.
- When using the `pgo-deployer` container to install the PostgreSQL Operator, update the default watched namespace to `pgo` as the example only uses this namespace.

Fixes

- Fix for cloning a PostgreSQL cluster when the pgBackRest repository is stored in S3.
- The `pgo show namespace` command now properly indicates which namespaces a user is able to access.
- Ensure the `pgo-apiserver` will successfully run if `PGO_DISABLE_TLS` is set to `true`. Reported by (@zhubx007).
- Prevent a run of `pgo-deployer` from failing if it detects the existence of dependent cluster-wide objects already present.
- Deployments with `pgo-deployer` using the default file with `hostpathstorage` will now successfully deploy PostgreSQL clusters without any adjustments.
- Ensure image pull secrets are attached to deployments of the `pgo-client` container.
- Ensure `client-setup.sh` executes to completion if existing PostgreSQL Operator credentials exist that were created by a different installation method
- Update the documentation to properly name `CCP_IMAGE_PULL_SECRET_MANIFEST` and `PGO_IMAGE_PULL_SECRET_MANIFEST` in the `pgo-deployer` configuration.
- Several fixes for selecting default storage configurations and sizes when using the `pgo-deployer` container. These include #1, #4, and #8 in the `STORAGE` family of variables.
- The custom setup example was updated to reflect the current state of bootstrapping the PostgreSQL container.

Crunchy Data announces the release of the PostgreSQL Operator 4.3.0 on May 1, 2020.

The PostgreSQL Operator is released in conjunction with the Crunchy Container Suite.

The PostgreSQL Operator 4.3.0 release includes the following software versions upgrades:

- The PostgreSQL containers now use versions 12.2, 11.7, 10.12, 9.6.17, and 9.5.21
- This now includes support for using the JIT compilation feature introduced in PostgreSQL 11
- PostgreSQL containers now support PL/Python3
- pgBackRest is now at version 2.25
- Patroni is now at version 1.6.5
- postgres_exporter is now at version 0.7.0
- pgAdmin 4 is at 4.18

PostgreSQL Operator is tested with Kubernetes 1.13 - 1.18, OpenShift 3.11+, OpenShift 4.3+, Google Kubernetes Engine (GKE), and VMware Enterprise PKS 1.3+.

Major Features

- Standby Clusters + Multi-Kubernetes Deployments({{< relref "/architecture/high-availability/multi-cluster-kubernetes.md" >}})
- [Improved custom configuration for PostgreSQL clusters]({{< relref "/advanced/custom-configuration.md" >}})
- Installation via the `pgo-deployer` container({{< relref "/installation/postgres-operator/_index.md" >}})
- [Automatic Upgrades of the PostgreSQL Operator via `pgo upgrade`]({{< relref "/upgrade/_index.md" >}})
- Set [custom PVC sizes]({{< relref "pgo-client/common-tasks/_index.md" >}}#create-a-postgresql-cluster-with-different-pvc-sizes) for PostgreSQL clusters on creation and clone
- Support for PostgreSQL Tablespaces({{< relref "/architecture/tablespaces.md" >}})
- The ability to specify an external volume for write-ahead logs (WAL)
- [Elimination of `ClusterRole` requirement]({{< relref "/architecture/namespace.md" >}}) for using the PostgreSQL Operator
- [Easy TLS-enabled PostgreSQL cluster creation]({{< relref "pgo-client/common-tasks/_index.md" >}}#enable-tls)
- All Operator commands now support TLS-only PostgreSQL workflows
- Feature Preview: [pgAdmin 4 Integration + User Synchronization]({{< relref "/architecture/pgadmin4.md" >}})

Standby Clusters + Multi-Kubernetes Deployments

A key component of building database architectures that can ensure continuity of operations is to be able to have the database available across multiple data centers. In Kubernetes, this would mean being able to have the PostgreSQL Operator be able to have the PostgreSQL

Operator run in multiple Kubernetes clusters, have PostgreSQL clusters exist in these Kubernetes clusters, and only ensure the "standby" deployment is promoted in the event of an outage or planned switchover.

As of this release, the PostgreSQL Operator now supports standby PostgreSQL clusters that can be deployed across namespaces or other Kubernetes or Kubernetes-enabled clusters (e.g. OpenShift). This is accomplished by leveraging the PostgreSQL Operator's support for [pgBackRest]({{< relref "/architecture/disaster-recovery.md" >}}) and leveraging an intermediary, i.e. S3, to provide the ability for the standby cluster to read in the PostgreSQL archives and replicate the data. This allows a user to quickly promote a standby PostgreSQL cluster in the event that the primary cluster suffers downtime (e.g. data center outage), for planned switchovers such as Kubernetes cluster maintenance or moving a PostgreSQL workload from one data center to another.

To support standby clusters, there are several new flags available on `pgo create cluster` that are required to set up a new standby cluster. These include:

- `--standby`: If set, creates the PostgreSQL cluster as a standby cluster.
- `--pgbackrest-repo-path`: Allows the user to override the `pgBackRest` repository path for a cluster. While this setting can now be utilized when creating any cluster, it is typically required for the creation of standby clusters as the repository path will need to match that of the primary cluster.
- `--password-superuser`: When creating a standby cluster, allows the user to specify a password for the superuser that matches the superuser account in the cluster the standby is replicating from.
- `--password-replication`: When creating a standby cluster, allows the user to specify a password for the replication user that matches the superuser account in the cluster the standby is replicating from.

Note that the `--password` flag must be used to ensure the password of the main PostgreSQL user account matches that of the primary PostgreSQL cluster, if you are using Kubernetes to manage the user's password.

For example, if you have a cluster named `hippo` and wanted to create a standby cluster called `hippo` and assuming the S3 credentials are using the defaults provided to the PostgreSQL Operator, you could execute a command similar to:

```
pgo create cluster hippo-standby --standby \
  --pgbackrest-repo-path=/backrestrepo/hippo-backrest-shared-repo
  --password-superuser=superhippo
  --password-replication=replicahippo
```

To shutdown the primary cluster (if you can), you can execute a command similar to:

```
pgo update cluster hippo --shutdown
```

To promote the standby cluster to be able to accept write traffic, you can execute the following command:

```
pgo update cluster hippo-standby --promote-standby
```

To convert the old primary cluster into a standby cluster, you can execute the following command:

```
pgo update cluster hippo --enable-standby
```

Once the old primary is converted to a standby cluster, you can bring it online with the following command:

```
pgo update cluster hippo --startup
```

For information on the architecture and how to [set up a standby PostgreSQL cluster]({{< relref "/architecture/high-availability/multi-cluster-kubernetes.md" >}}), please refer to the [documentation]({{< relref "/architecture/high-availability/multi-cluster-kubernetes.md" >}}).

At present, streaming replication between the primary and standby clusters are not supported, but the PostgreSQL instances within each cluster do support streaming replication.

Installation via the `pgo-deployer` container

Installation, alongside upgrading, have long been two of the biggest challenges of using the PostgreSQL Operator. This release makes improvements on both (with upgrading being described in the next section).

For installation, we have introduced a new container called [`pgo-deployer`]({{< relref "/installation/postgres-operator/_index.md" >}}). For environments that use hostpath storage (e.g. minikube), [installing the PostgreSQL Operator]({{< relref "/installation/postgres-operator/_index.md" >}}) can be as simple as:

```
kubectl create namespace pgo
kubectl apply -f
   https://raw.githubusercontent.com/CrunchyData/postgres-operator/v4.3.0/installers/kubectl/postgres
```

The `pgo-deployer` container can be configured by a manifest called `postgres-operator.yml` and provides a set of [environmental variables]({{< relref "/installation/configuration/_index.md" >}}) that should be familiar from using the [other installers]({{< relref "/installation/other/_index.md" >}}).

The `pgo-deployer` launches a Job in the namespace that the PostgreSQL Operator will be installed into and sets up the requisite Kubernetes objects: CRDs, Secrets, ConfigMaps, etc.

The `pgo-deployer` container can also be used to uninstall the PostgreSQL Operator. For more information, please see the [installation documentation]({{< relref "/installation/_index.md" >}}).

Automatic PostgreSQL Operator Upgrade Process

One of the biggest challenges to using a newer version of the PostgreSQL Operator was upgrading from an older version.

This release introduces the ability to [automatically upgrade from an older version of the Operator]({{< relref "/upgrade/_index.md" >}}) (as early as 4.1.0) to the newest version (4.3.0) using the [`pgo upgrade`]({{< relref "/pgo-client/reference/pgo_upgrade.md" >}}) command.

The `pgo upgrade` command follows a process similar to the [manual PostgreSQL Operator upgrade]({{< relref "/upgrade/manual/upgrade4.md" >}}) process, but instead automates it.

To find out more about how to upgrade the PostgreSQL Operator, please review the [upgrade documentation]({{< relref "/upgrade/_index.md" >}}).

Improved Custom Configuration for PostgreSQL Clusters

The ability to customize the configuration for a PostgreSQL cluster with the PostgreSQL Operator can now be easily modified by making changes directly to the ConfigMap that is created with each PostgreSQL cluster. The ConfigMap, which follows the pattern `<clusterName>-pgha-config` (e.g. `hippo-pgha-config` for `pgo create cluster hippo`), manages the user-facing configuration settings available for a PostgreSQL cluster, and when modified, it will automatically synchronize the settings across all primaries and replicas in a PostgreSQL cluster.

Presently, the ConfigMap can be edited using the `kubectl edit cm` command, and future iterations will add functionality to the PostgreSQL Operator to make this process easier.

Customize PVC Size on PostgreSQL cluster Creation & Clone

The PostgreSQL Operator provides the ability to set customization for how large the PVC can be via the "storage config" options available in the PostgreSQL Operator configuration file (aka `pgo.yaml`). While these provide a baseline level of customizability, it is often important to be able to set the size of the PVC that a PostgreSQL cluster should use at cluster creation time. In other words, users should be able to choose exactly how large they want their PostgreSQL PVCs ought to be.

PostgreSQL Operator 4.3 introduces the ability to set the PVC sizes for the PostgreSQL cluster, the pgBackRest repository for the PostgreSQL cluster, and the PVC size for each tablespace at cluster creation time. Additionally, this behavior has been extended to the clone functionality as well, which is helpful when trying to resize a PostgreSQL cluster. Here is some information on the flags that have been added:

pgo create cluster

`--pvc-size` - sets the PVC size for the PostgreSQL data directory `--pgbackrest-pvc-size` - sets the PVC size for the PostgreSQL pgBackRest repository

For tablespaces, one can use the `pvcsize` option to set the PVC size for that tablespace.

pgo clone cluster

`--pvc-size` - sets the PVC size for the PostgreSQL data directory for the newly created cluster `--pgbackrest-pvc-size` - sets the PVC size for the PostgreSQL pgBackRest repository for the newly created cluster

Tablespaces

Tablespaces can be used to spread out PostgreSQL workloads across multiple volumes, which can be used for a variety of use cases:

- Partitioning larger data sets

- Putting data onto archival systems
- Utilizing hardware (or a storage class) for a particular database object, e.g. an index

and more.

Tablespaces can be created via the `pgo create cluster` command using the `--tablespace` flag. The arguments to `--tablespace` can be passed in using one of several key/value pairs, including:

- `name` (required) - the name of the tablespace
- `storageconfig` (required) - the storage configuration to use for the tablespace
- `pvcsize` - if specified, the size of the PVC. Defaults to the PVC size in the storage configuration

Each value is separated by a `:`, for example:

```
pgo create cluster hacluster --tablespace=name=ts:storageconfig=nfsstorage
```

All tablespaces are mounted in the `/tablespaces` directory. The PostgreSQL Operator manages the mount points and persistent volume claims (PVCs) for the tablespaces, and ensures they are available throughout all of the PostgreSQL lifecycle operations, including:

- Provisioning
- Backup & Restore
- High-Availability, Failover, Healing
- Clone

etc.

One additional value is added to the pgcluster CRD:

- TablespaceMounts: a map of the name of the tablespace and its associated storage.

Tablespaces are automatically created in the PostgreSQL cluster. You can access them as soon as the cluster is initialized. For example, using the tablespace created above, you could create a table on the tablespace `ts` with the following SQL:

```
CREATE TABLE (id int) TABLESPACE ts;
```

Tablespaces can also be added to existing PostgreSQL clusters by using the `pgo update cluster` command. The syntax is similar to that of creating a PostgreSQL cluster with a tablespace, i.e.:

```
pgo update cluster hacluster --tablespace=name=ts2:storageconfig=nfsstorage
```

As additional volumes need to be mounted to the Deployments, this action can cause downtime, though the expectation is that the downtime is brief.

Based on usage, future work will look to making this more flexible. Dropping tablespaces can be tricky as no objects must exist on a tablespace in order for PostgreSQL to drop it (i.e. there is no DROP TABLESPACE .. CASCADE command).

Easy TLS-Enabled PostgreSQL Clusters

Connecting to PostgreSQL clusters is a typical requirement when deploying to an untrusted network, such as a public cloud. The PostgreSQL Operator makes it easy to enable TLS for PostgreSQL. To do this, one must create two secrets prior: one containing the trusted certificate authority (CA) and one containing the PostgreSQL server's TLS keypair, e.g.:

```
kubectl create secret generic postgresql-ca --from-file=ca.crt=/path/to/ca.crt
kubectl create secret tls hippo-tls-keypair \
  --cert=/path/to/server.crt \
  --key=/path/to/server.key
```

From there, one can create a PostgreSQL cluster that supports TLS with the following command:

```
pgo create cluster hippo-tls \
  --server-ca-secret=postgresql-ca \
  --server-tls-secret=hippo-tls-keypair
```

To create a PostgreSQL cluster that **only** accepts TLS connections and rejects any connection attempts made over an insecure channel, you can use the `--tls-only` flag on cluster creation, e.g.:

```
pgo create cluster hippo-tls \
  --tls-only \
  --server-ca-secret=postgresql-ca \
  --server-tls-secret=hippo-tls-keypair
```

External WAL Volume

An optimization used for improving PostgreSQL performance related to file system usage is to have the PostgreSQL write-ahead logs (WAL) written to a different mounted volume than other parts of the PostgreSQL system, such as the data directory.

To support this, the PostgreSQL Operator now supports the ability to specify an external volume for writing the PostgreSQL write-head log (WAL) during cluster creation, which carries through to replicas and clones. When not specified, the WAL resides within the PGDATA directory and volume, which is the present behavior.

To create a PostgreSQL cluster to use an external volume, one can use the `--wal-storage-config` flag at cluster creation time to select the storage configuration to use, e.g.

```
pgo create cluster --wal-storage-config=nfsstorage hippo
```

Additionally, it is also possible to specify the size of the WAL storage on all newly created clusters. When in use, the size of the volume can be overridden per-cluster. This is specified with the `--wal-storage-size` flag, i.e.

```
pgo create cluster --wal-storage-config=nfsstorage --wal-storage-size=10Gi hippo
```

This implementation does not define the WAL volume in any deployment templates because the volume name and mount path are constant.

Elimination of `ClusterRole` Requirement for the PostgreSQL Operator

PostgreSQL Operator 4.0 introduced the ability to manage PostgreSQL clusters across multiple Kubernetes Namespaces. PostgreSQL Operator 4.1 built on this functionality by allowing users to dynamically control which Namespaces it managed as well as the PostgreSQL clusters deployed to them. In order to leverage this feature, one must grant a `ClusterRole` level permission via a ServiceAccount to the PostgreSQL Operator.

There are a lot of deployment environments for the PostgreSQL Operator that only need for it to exists within a single namespace and as such, granting cluster-wide privileges is superfluous, and in many cases, undesirable. As such, it should be possible to deploy the PostgreSQL Operator to a single namespace without requiring a `ClusterRole`.

To do this, but maintain the aforementioned Namespace functionality for those who require it, PostgreSQL Operator 4.3 introduces the ability to opt into deploying it with minimum required `ClusterRole` privileges and in turn, the ability to deploy the PostgreSQL Operator without a `ClusterRole`. To do so, the PostgreSQL Operator introduces the concept of "namespace operating mode" which lets one select the type deployment to create. The namespace mode is set at the install time for the PostgreSQL Operator, and files into one of three options:

- `dynamic`: **This is the default**. This enables full dynamic Namespace management capabilities, in which the PostgreSQL Operator can create, delete and update any Namespaces within the Kubernetes cluster, while then also having the ability to create the Roles, Role Bindings and Service Accounts within those Namespaces for normal operations. The PostgreSQL Operator can also listen for Namespace events and create or remove controllers for various Namespaces as changes are made to Namespaces from Kubernetes and the PostgreSQL Operator's management.
- `readonly`: In this mode, the PostgreSQL Operator is able to listen for namespace events within the Kubernetetes cluster, and then manage controllers as Namespaces are added, updated or deleted. While this still requires a `ClusterRole`, the permissions mirror those of a "read-only" environment, and as such the PostgreSQL Operator is unable to create, delete or update Namespaces itself nor create RBAC that it requires in any of those Namespaces. Therefore, while in `readonly`, mode namespaces must be preconfigured with the proper RBAC as the PostgreSQL Operator cannot create the RBAC itself.
- `disabled`: Use this mode if you do not want to deploy the PostgreSQL Operator with any `ClusterRole` privileges, especially if you are only deploying the PostgreSQL Operator to a single namespace. This disables any Namespace management capabilities within the PostgreSQL Operator and will simply attempt to work with the target Namespaces specified during installation. If no target Namespaces are specified, then the Operator will be configured to work within the namespace in which it is deployed. As with the `readonly` mode, while in this mode, Namespaces must be pre-configured with the proper RBAC, since the PostgreSQL Operator cannot create the RBAC itself.

Based on the installer you use, the variables to set this mode are either named:

- PostgreSQL Operator Installer: `NAMESPACE_MODE`
- Developer Installer: `PGO_NAMESPACE_MODE`
- Ansible Installer: `namespace_mode`

Feature Preview: pgAdmin 4 Integration + User Synchronization

pgAdmin 4 is a popular graphical user interface that lets you work with PostgreSQL databases from both a desktop or web-based client. With its ability to manage and orchestrate changes for PostgreSQL users, the PostgreSQL Operator is a natural partner to keep a pgAdmin 4 environment synchronized with a PostgreSQL environment.

This release introduces an integration with pgAdmin 4 that allows you to deploy a pgAdmin 4 environment alongside a PostgreSQL cluster and keeps the user's database credentials synchronized. You can simply log into pgAdmin 4 with your PostgreSQL username and password and immediately have access to your databases.

For example, let's there is a PostgreSQL cluster called `hippo` that has a user named `hippo` with password `datalake`:

```
pgo create cluster hippo --username=hippo --password=datalake
```

After the PostgreSQL cluster becomes ready, you can create a pgAdmin 4 deployment with the [`pgo create pgadmin`]({{< relref "/pgo-client/reference/pgo_create_pgadmin.md" >}}) command:

```
pgo create pgadmin hippo
```

This creates a pgAdmin 4 deployment unique to this PostgreSQL cluster and synchronizes the PostgreSQL user information into it. To access pgAdmin 4, you can set up a port-forward to the Service, which follows the pattern `<clusterName>-pgadmin`, to port `5050`:

```
kubectl port-forward svc/hippo-pgadmin 5050:5050
```

Point your browser at `http://localhost:5050` and use your database username (e.g. `hippo`) and password (e.g. `datalake`) to log in.

(Note: if your password does not appear to work, you can retry setting up the user with the [`pgo update user`]({{< relref "/pgo-client/reference/pgo_update_user.md" >}}) command: `pgo update user hippo --password=datalake`)

The `pgo create user`, `pgo update user`, and `pgo delete user` commands are synchronized with the pgAdmin 4 deployment. Note that if you use `pgo create user` without the `--managed` flag prior to deploying pgAdmin 4, then the user's credentials will not be synchronized to the pgAdmin 4 deployment. However, a subsequent run of `pgo update user --password` will synchronize the credentials with pgAdmin 4.

You can remove the pgAdmin 4 deployment with the [`pgo delete pgadmin`]({{< relref "/pgo-client/reference/pgo_delete_pgadmin.md" >}}) command.

We have released the first version of this change under "feature preview" so you can try it out. As with all of our features, we open to feedback on how we can continue to improve the PostgreSQL Operator.

Enhanced `pgo df`

`pgo df` provides information on the disk utilization of a PostgreSQL cluster, and previously, this was not reporting accurate numbers. The new `pgo df` looks at each PVC that is mounted to each PostgreSQL instance in a cluster, including the PVCs for tablespaces, and computers the overall utilization. Even better, the data is returned in a structured format for easy scraping. This implementation also leverages Golang concurrency to help compute the results quickly.

Enhanced pgBouncer Integration

The pgBouncer integration was completely rewritten to support the TLS-only operations via the PostgreSQL Operator. While most of the work was internal, you should now see a much more stable pgBouncer experience.

The pgBouncer attributes in the `pgclusters.crunchydata.com` CRD are also declarative and any updates will be reflected by the PostgreSQL Operator.

Additionally, a few new commands were added:

- `pgo create pgbouncer --cpu` and `pgo update pgbouncer --memory` resource request flags for settings container resources for the pgBouncer instances. For CPU, this will also set the limit.
- `pgo create pgbouncer --enable-memory-limit` sets the Kubernetes resource limit for memory
- `pgo create pgbouncer --replicas` sets the number of pgBouncer Pods to deploy with a PostgreSQL cluster. The default is `1`.
- `pgo show pgbouncer` shows information about a pgBouncer deployment
- `pgo update pgbouncer --cpu` and `pgo update pgbouncer --memory` resource request flags for settings container resources for the pgBouncer instances after they are deployed. For CPU, this will also set the limit.
- `pgo update pgbouncer --disables-memory-limit` and `pgo update pgbouncer --enable-memory-limit` respectively unset and set the Kubernetes resource limit for memory
- `pgo update pgbouncer --replicas` sets the number of pgBouncer Pods to deploy with a PostgreSQL cluster.
- `pgo update pgbouncer --rotate-password` allows one to rotate the service account password for pgBouncer

Rewritten pgo User Management commands

The user management commands were rewritten to support the TLS only workflow. These commands now return additional information about a user when actions are taken. Several new flags have been added too, including the option to view all output in JSON. Other flags include:

- `pgo update user --rotate-password` to automatically rotate the password
- `pgo update user --disable-login` which disables the ability for a PostgreSQL user to login
- `pgo update user --enable-login` which enables the ability for a PostgreSQL user to login
- `pgo update user --valid-always` which sets a password to always be valid, i.e. it has no expiration
- `pgo show user` does not show system accounts by default now, but can be made to show the system accounts by using `pgo show user --show-system-accounts`

A major change as well is that the default password expiration function is now defaulted to be unlimited (i.e. never expires) which aligns with typical PostgreSQL workflows.

Breaking Changes

- `pgo create cluster` will now set the default database name to be the name of the cluster. For example, `pgo create cluster hippo` would create the initial database named `hippo`.
- The `Database` configuration parameter in `pgo.yaml` (`db_name` in the Ansible inventory) is now set to `""` by default.
- the `--password`/`-w` flag for `pgo create cluster` now only sets the password for the regular user account that is created, not all of the system accounts (e.g. the `postgres` superuser).
- A default `postgres-ha.yaml` file is no longer is no longer created by the Operator for every PostgreSQL cluster.
- "Limit" resource parameters are no longer set on the containers, in particular, the PostgreSQL container, due to undesired behavior stemming from the host machine OOM killer. Further details can be found in the original pull request.
- Added `DefaultInstanceMemory`, `DefaultBackrestMemory`, and `DefaultPgBouncerMemory` options to the `pgo.yaml` configuration to allow for the setting of default memory requests for PostgreSQL instances, the pgBackRest repository, and pgBouncer instances respectively.
- If unset by either the PostgreSQL Operator configuration or one-off, the default memory resource requests for the following applications are:
- PostgreSQL: The installers default to 128Mi (suitable for test environments), though the "default of last resort" is 512Mi to be consistent with the PostgreSQL default shared memory requirement
- pgBackRest: 48Mi
- pgBouncer: 24Mi
- Remove the `Default...ContainerResources` set of parameters from the `pgo.yaml` configuration file.
- The `pgbackups.crunchydata.com`, deprecated since 4.2.0, has now been completely removed, along with any code that interfaced with it.
- The `PreferredFailoverFeature` is removed. This had not been doing anything since 4.2.0, but some of the legacy bits and configuration were still there.
- `pgo status` no longer returns information about the nodes available in a Kubernetes cluster
- Remove `--series` flag from `pgo create cluster` command. This affects API calls more than actual usage of the `pgo` client.
- `pgo benchmark`, `pgo show benchmark`, `pgo delete benchmark` are removed. PostgreSQL benchmarks with `pgbench` can still be executed using the `crunchy-pgbench` container.
- `pgo ls` is removed.
- The API that is used by `pgo create cluster` now returns its contents in JSON. The output now includes information about the user that is created.
- The API that is used by `pgo show backup` now returns its contents in JSON. The output view of `pgo show backup` remains the same.
- Remove the `PreferredFailoverNode` feature, as it had already been effectively removed.
- Remove explicit `rm` calls when cleaning up PostgreSQL clusters. This behavior is left to the storage provisioner that one deploys with their PostgreSQL instances.
- Schedule backup job names have been shortened, and follow a pattern that looks like `<clusterName>-<backupType>-sch-backup`

Features

- Several additions to `pgo create cluster` around PostgreSQL users and databases, including:
- `--ccp-image-prefix` sets the `CCPImagePrefix` that specifies the image prefix for the PostgreSQL related containers that are deployed by the PostgreSQL Operator

- `--cpu` flag that sets the amount of CPU to use for the PostgreSQL instances in the cluster. This also sets the limit. `---database` / `-d` flag that sets the name of the initial database created.
- `--enable-memory-limit`, `--enable-pgbackrest-memory-limit`, `--enable-pgbouncer-memory-limit` enable the Kubernetes memory resource limit for PostgreSQL, pgBackRest, and pgBouncer respectively
- `--memory` flag that sets the amount of memory to use for the PostgreSQL instances in the cluster
- `--user` / `-u` flag that sets the PostgreSQL username for the standard database user
- `--password-length` sets the length of the password that should be generated, if `--password` is not set.
- `--pgbackrest-cpu` flag that sets the amount of CPU to use for the pgBackRest repository
- `--pgbackrest-memory` flag that sets the amount of memory to use for the pgBackRest repository
- `--pgbackrest-s3-ca-secret` specifies the name of a Kubernetes Secret that contains a key (`aws-s3-ca.crt`) to override the default CA used for making connections to a S3 interface
- `--pgbackrest-storage-config` lets one specify a different storage configuration to use for a local pgBackRest repository
- `--pgbouncer-cpu` flag that sets the amount of CPU to use for the pgBouncer instances
- `--pgbouncer-memory` flag that sets the amount of memory to use for the pgBouncer instances
- `--pgbouncer-replicas` sets the number of pgBouncer Pods to deploy with the PostgreSQL cluster. The default is `1`.
- `--pgo-image-prefix` sets the `PGOImagePrefix` that specifies the image prefix for the PostgreSQL Operator containers that help to manage the PostgreSQL clusters
- `--show-system-accounts` returns the credentials of the system accounts (e.g. the `postgres` superuser) along with the credentials for the standard database user
- `pgo update cluster` now supports the `--cpu`, `--disable-memory-limit`, `--disable-pgbackrest-memory-limit`, `--enable-memory` `--enable-pgbackrest-memory-limit`, `--memory`, `--pgbackrest-cpu`, and `--pgbackrest-memory` flags to allow PostgreSQL instances and the pgBackRest repository to have their resources adjusted post deployment
- Added the `PodAntiAffinityPgBackRest` and `PodAntiAffinityPgBouncer` to the `pgo.yaml` configuration file to set specific Pod anti-affinity rules for pgBackRest and pgBouncer Pods that are deployed along with PostgreSQL clusters that are managed by the Operator. The default for pgBackRest and pgBouncer is to use the value that is set in `PodAntiAffinity`.
- `pgo create cluster` now supports the `--pod-anti-affinity-pgbackrest` and `--pod-anti-affinity-pgbouncer` flags to specifically overwrite the pgBackRest repository and pgBouncer Pod anti-affinity rules on a specific PostgreSQL cluster deployment, which overrides any values present in `PodAntiAffinityPgBackRest` and `PodAntiAffinityPgBouncer` respectfully. The default for pgBackRest and pgBouncer is to use the value for pod anti-affinity that is used for the PostgreSQL instances in the cluster.
- One can specify the "image prefix" (e.g. `crunchydata`) for the containers that are deployed by the PostgreSQL Operator. This adds two fields to the pgcluster CRD: `CCPImagePrefix` and 'PGOImagePrefix
- Specify a different S3 Certificate Authority (CA) with `pgo create cluster` by using the `--pgbackrest-s3-ca-secret` flag, which refers to an existing Secret that contains a key called `aws-s3-ca.crt` that contains the CA. Reported by Aurelien Marie @(aurelien-marie)
- `pgo clone` now supports the `--enable-metrics` flag, which will deploy the monitoring sidecar along with the newly cloned PostgreSQL cluster.
- The pgBackRest repository now uses ED25519 SSH key pairs.
- Add the `--enable-autofail` flag to `pgo update` to make it clear how the autofailover mechanism can be re-enabled for a PostgreSQL cluster.

Changes

- Remove `backoffLimit` from Jobs that can be retried, which is most of them.
- POSIX shared memory is now used for the PostgreSQL Deployments.
- Increase the number of namespaces that can be watched by the PostgreSQL Operator.
- The number of unsupported pgBackRest flags on the deny list has been reduced.
- The liveness and readiness probes for a PostgreSQL cluster now reference the `/opt/cpm/bin/health`
- `wal_level` is now defaulted to `logical` to enable logical replication
- `archive_timeout` is now a default setting in the `crunchy-postgres-ha` and `crunchy-postgres-ha-gis` containers and is set to `60`
- `ArchiveTimeout`, `LogStatement`, `LogMinDurationStatement` are removed from `pgo.yaml`, as these can be customized either via a custom `postgresql.conf` file or `postgres-ha.yaml` file
- Quoted identifiers for the database name and user name in bootstrap scripts for the PostgreSQL containers
- Password generation now leverages cryptographically secure random number generation and uses the full set of typeable ASCII characters
- The `node` ClusterRole is no longer used
- The names of the scheduled backups are shortened to use the pattern `<clusterName>-<backupType>-sch-backup`
- The PostgreSQL Operator now logs its timestamps using RFC3339 formatting as implemented by Go
- SSH key pairs are no longer created as part of the Operator installation process. This was a legacy behavior that had not been removed
- The `pv/create-pv-nfs.sh` has been modified to create persistent volumes with their own directories on the NFS filesystems. This

better mimics production environments. The older version of the script still exists as `pv/create-pv-nfs-legacy.sh`
- Load pgBackRest S3 credentials into environmental variables as Kubernetes Secrets, to avoid revealing their contents in Kubernetes commands or in logs
- Update how the pgBackRest and pgMonitor pamareters are loaded into Deployment templates to no longer use JSON fragments
- The `pgo-rmdata` Job no longer calls the `rm` command on any data within the PVC, but rather leaves this task to the storage provisioner
- Remove using `expenv` in the `add-targeted-namespace.sh` script

Fixes

- Ensure PostgreSQL clusters can be successfully restored via `pgo restore` after 'pgo scaledown' is executed
- Allow the original primary to be removed with `pgo scaledown` after it is failed over
- The replica Service is now properly managed based on the existence of replicas in a PostgreSQL cluster, i.e. if there are replicas, the Service exists, if not, it is removed
- Report errors in a SQL policy at the time `pgo apply` is executed, which was the previous behavior. Reported by José Joye (@jose-joye)
- Ensure all replicas are listed out via the `--query` flag in `pgo scaledown` and `pgo failover`. This now follows the pattern outlined by the Kubernetes safe random string generator
- Default the recovery action to "promote" when performing a "point-in-time-recovery" (PITR), which will ensure that a PITR process completes
- The `stanza-create` Job now waits for both the PostgreSQL cluster and the pgBackRest repository to be ready before executing
- Remove `backoffLimit` from Jobs that can be retried, which is most of them. Reported by Leo Khomenko (@lkhomenk)
- The `pgo-rmdata` Job will not fail if a PostgreSQL cluster has not been properly initialized
- Fixed a separate `pgo-rmdata` crash related to an improper SecurityContext
- The `failover` ConfigMap for a PostgreSQL cluster is now removed when the cluster is deleted
- Allow the standard PostgreSQL user created with the Operator to be able to create and manage objects within its own user schema. Reported by Nicolas HAHN (@hahnn)
- Honor the value of "PasswordLength" when it is set in the pgo.yaml file for password generation. The default is now set at `24`
- Do not log pgBackRest environmental variables to the Kubernetes logs
- By default, exclude using the trusted OS certificate authority store for the Windows pgo client.
- Update the `pgo-client` imagePullPolicy to be `IfNotPresent`, which is the default for all of the managed containers across the project
- Set `UsePAM yes` in the `sshd_config` file to fix an issue with using SSHD in newer versions of Docker
- Only add Operator labels to a managed namespace if the namespace already exists when executing the `add-targeted-namespace.sh` script

Crunchy Data announces the release of the PostgreSQL Operator 4.2.2 on February, 18, 2020.

The PostgreSQL Operator 4.2.2 release provides bug fixes and continued support to the 4.2 release line.

This release includes updates for several packages supported by the PostgreSQL Operator, including:

- The PostgreSQL containers now use versions 12.2, 11.7, 10.12, 9.6.17, and 9.5.21
- The PostgreSQL containers now support PL/Python3
- Patroni is updated to version 1.6.4

The PostgreSQL Operator is released in conjunction with the Crunchy Container Suite.

PostgreSQL Operator is tested with Kubernetes 1.13+, OpenShift 3.11+, Google Kubernetes Engine (GKE), and VMware Enterprise PKS 1.3+.

Changes since 4.2.1

- Added the `--enable-autofail` flag to `pgo update` to make it clear how the auto-failover mechanism can be re-enabled for a PostgreSQL cluster.
- Remove using `expenv` in the `add-targeted-namespace.sh` script

Fixes since 4.2.1

- Ensure PostgreSQL clusters can be successfully restored via `pgo restore` after 'pgo scaledown' is executed

- Ensure all replicas are listed out via the `--query` flag in `pgo scaledown` and `pgo failover`. This now follows the pattern outlined by the Kubernetes safe random string generator (#1247)
- Honor the value of "PasswordLength" when it is set in the pgo.yaml file for password generation. The default is now set at `24`
- Set `UsePAM yes` in the `sshd_config` file to fix an issue with using SSHD in newer versions of Docker
- The backup task listed in the `pgtask` CRD is now only deleted if one already exists
- Ensure that a successful "rmdata" Job does not delete all cluster pgtasks listed in the CRD after a successful run
- Only add Operator labels to a managed namespace if the namespace already exists when executing the `add-targeted-namespace.sh` script
- Remove logging of PostgreSQL user credentials in the PostgreSQL Operator logs
- Consolidation of the Dockerfiles for RHEL7/UBI7 builds
- Several fixes to the documentation (#1233)

Crunchy Data announces the release of the PostgreSQL Operator 4.2.1 on January, 16, 2020.

The PostgreSQL Operator 4.2.1 provides bug fixes and continued support to the 4.2 release line.

The PostgreSQL Operator is released in conjunction with the Crunchy Container Suite.

PostgreSQL Operator is tested with Kubernetes 1.13+, OpenShift 3.11+, Google Kubernetes Engine (GKE), and VMware Enterprise PKS 1.3+.

Fixes

- Ensure Pod labels are updated after failover (#1218)
- Fix for scheduled tasks to continue executing even after `pgo delete schedule` is called (#1215)
- Ensure `pgo restore` carries through the `--node-label` to the new primary (#1206)
- Fix for displaying incorrect replica names with the `--query` flag on `pgo scaledown/pgo failover` after a failover occurred
- Fix for default CA exclusion for the Windows-based [pgo client]({{< relref "pgo-client/_index.md" >}})
- Fix a race condition where the `pgo-rmdata` job could fail when doing its final pass on deleting PVCs. This went unnoticed as it was the final task to occur
- Fix image pull policy for the `pgo-client` container to match the project default (`IfNotPresent`)
- Update the "Create CRD Example" to reference the `crunchy-postgres-ha` container
- Update comments used for the API documentation generation via Swagger
- Update the directions for running the PostgreSQL Operator from the GCP Marketplace deployer
- Updated OLM installer example and added generation script
- Update the "cron" package to `v3.0.1`

Crunchy Data announces the release of the PostgreSQL Operator 4.2.0 on December, 31, 2019.

The focus of the 4.2.0 release of the PostgreSQL Operator was on the resiliency and uptime of the PostgreSQL clusters that the PostgreSQL Operator manages, with an emphasis on high-availability and removing the Operator from being a single-point-of-failure in the HA process. This release introduces support for a distributed-consensus based high-availability approach using Kubernetes distributed consensus store as the backing, which, in other words, allows for the PostgreSQL clusters to manage their own availability and **not** the PostgreSQL Operator. This is accomplished by leveraging the open source high-availability framework Patroni as well as the open source, high-performant PostgreSQL disaster recovery management tool pgBackRest.

To accomplish this, we have introduced a new container called `crunchy-postgres-ha` (and for geospatial workloads, `crunchy-postgres-gis-`**If you are upgrading from an older version of the PostgreSQL Operator, you will need to modify your installation to use these containers**.

Included in the PostgreSQL Operator 4.2.0 introduces the following new features:

- An improved PostgreSQL HA (high-availability) solution using distributed consensus that is backed by Kubernetes. This includes:
- Elimination of the PostgreSQL Operator as the arbiter that decides when a cluster should fail over
- Support for Pod anti-affinity, which indicates to Kubernetes schedule pods (e.g. PostgreSQL instances) on separate nodes
- Failed primaries now automatically heal, which significantly reduces the time in which they can rejoin the cluster.
- Introduction of synchronous replication for workloads that are sensitive to transaction loss (with a tradeoff of performance and potentially availability)
- Standardization of physical backups and restores on pgBackRest, with native support for `pg_basebackup` removed.
- Introduction of the ability to clone PostgreSQL clusters using the `pgo clone` command. This feature copies the pgBackRest repository from a cluster and creates a new, single instance primary as its own cluster.
- Allow one to use their own certificate authority (CA) when interfacing with the Operator API, and to specify usage of the CA from the `pgo` command-line interface (CLI)

The container building process has been optimized, with build speed ups reported to be 70% faster.

The Postgres Operator 4.2.0 release also includes the following software versions upgrades:

- The PostgreSQL containers now use versions 12.1, 11.6, 10.11, 9.6.16, and 9.5.20.
- pgBackRest is upgraded to use version 2.20
- pgBouncer is upgraded to use version 1.12
- Patroni uses version 1.6.3

PostgreSQL Operator is tested with Kubernetes 1.13 - 1.15, OpenShift 3.11+, Google Kubernetes Engine (GKE), and VMware Enterprise PKS 1.3+. We have taken steps to ensure the PostgreSQL Operator is compatible with Kubernetes 1.16+, but did not test thoroughly on it for this release. Cursory testing indicates that the PostgreSQL Operator is compatible with Kubernetes 1.16 and beyond, but we advise that you run your own tests.

Major Features

High-Availability & Disaster Recovery

PostgreSQL Operator 4.2.0 makes significant enhancements to the high-availability and disaster recovery capabilities of the PostgreSQL Operator by moving to a distributed-consensus based model for maintaining availability, standardizing around pgBackRest for backups and restores, and removing the Operator itself as a single-point-of-failure in relation to PostgreSQL cluster resiliency.

As the high-availability environment introduced by PostgreSQL Operator 4.2.0 is now the default, setting up a HA cluster is as easy as:

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

If you wish to disable high-availability for a cluster, you can use the following command:

```
pgo create cluster boringcluster --disable-autofail
```

New Required HA PostgreSQL Containers: `crunchy-postgres-ha` and `crunchy-postgres-gis-ha`

Using the PostgreSQL Operator 4.2.0 requires replacing your `crunchy-postgres` and `crunchy-postgres-gis` containers with the `crunchy-postgres-ha` and `crunchy-postgres-gis-ha` containres 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.

Upgrading to these new containers is as simple as modifying your CRD `ccpimage` parameter to use `crunchy-postgres-ha` to use the HA enabled containers. Please see our upgrade instructions to select your preferred upgrade strategy.

pgBackRest Standardization

pgBackRest is now the only backup and restore method supported by the PostgreSQL Operator. This has allowed for the following features:

- Faster creation of new replicas when a scale up request is made
- Automatic healing of PostgreSQL instances after a failover event, leveraging the pgBackRest delta restore feature. This allows for a significantly shorter healing process
- The ability to clone PostgreSQL clusters

As part of this standardization, one change to note is that after a PostgreSQL cluster is created, the PostgreSQL Operator will schedule a full backup of the cluster. This is to ensure that a new replica can be created from a pgBackRest backup. If this initial backup fails, no new replicas will be provisioned.

When upgrading from an earlier version, please ensure that you have at least one pgBackRest full backup in your backup repository.

Pod Anti-Affinity

PostgreSQL Operator 4.2.0 adds support for Kubernetes pod anti-affinity, which provides guidance on how Kubernetes should schedule pods relative to each other. This is helpful in high-availability architectures to ensure that PostgreSQL pods are spread out in order to withstand node failures. For example, in a setup with two PostgreSQL instances, you would not want both instances scheduled to the same node: if that node goes down or becomes unreachable, then your cluster will be unavailable!

The way the PostgreSQL Operator uses pod anti-affinity is that it tries to ensure that **none** of the managed pods within the same cluster are scheduled to the same node. These include:

- Any PostgreSQL instances
- The pod that manages pgBackRest repository
- If deployed, any pgBouncer pods

This helps improve the likelihood that a cluster can remain up even if a downtime event occurs.

There are three options available for pod anti-affinity:

- `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. 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.
- `disabled`: Pod anti-affinity is not used.

These options can be combined with the existing node affinity functionality (`--node-label`) to group the scheduling of pods to particular node labels!

Synchronous Replication

PostgreSQL Operator 4.2 introduces support for synchronous replication by leveraging the "synchronous mode" functionality provided by Patroni. Synchronous replication 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.

Updated pgo CLI Flags

- `pgo create` now has a CLI flag for pod anti-affinity called `--pod-anti-affinity`, which accepts the values `required`, `preferred`, and `disabled`
- `pgo create --sync-replication` specifies to create a PostgreSQL HA cluster with synchronous replication

Global Configuration

To support high-availability there are some new settings that you can manage from your `pgo.yaml` file:

- `DisableAutofail` - when set to `true`, this will disable the new HA functionality in any newly created PostgreSQL clusters. By default, this is `false`.
- `DisableReplicaStartFailReinit` - when set to `true`, this will disable attempting to re-provision a PostgreSQL replica when it is stuck in a "start failed" state. By default, this `false`.
- `PodAntiAffinity` - Determines the type of pod anti-affinity rules to apply to the pods within a newly PostgreSQL cluster. If set to `required`, pods within a PostgreSQL cluster **must** be scheduled on different nodes, otherwise a pod will fail to schedule. If set to `preferred`, Kubernetes will make a best effort to schedule pods of the same PostgreSQL cluster on different nodes. If set to `disabled`, this feature is disabled. By default, this is `preferred`.
- `SyncReplication` - If set to `true`, enables synchronous replication in newly created PostgreSQL clusters. Default to `false`.

pgo clone

PostgreSQL Operator 4.2.0 introduces the ability to clone the data from one PostgreSQL cluster into a brand new PostgreSQL cluster. The command to do so is simple:

```
pgo clone oldcluster newcluster
```

After the command is executed, the PostgreSQL Operator checks to see if a) the `oldcluster` exists and b) the `newcluster` does not exist. If both of these conditions hold, the PostgreSQL Operator creates two new PVCs the match the specs of the `oldcluster` PostgreSQL data PVC (`PrimaryStorage`) and its pgBackRest repository PVC (`BackrestStorage`).

If these PVCs are successfully created, the PostgreSQL Operator will copy the contents of the pgBackRest repository from the `oldcluster` to the one setup for the `newcluster` by means of a Kubernetes Job that is running `rsync` provided by the `pgo-backrest-repo-sync` container. We are able to do this because all changes to the pgBackRest repository are atomic.

If this successfully completes, the PostgreSQL Operator then runs a pgBackRest restore job to restore the PostgreSQL cluster. On a successful restore, the new PostgreSQL cluster is then scheduled and runs in recovery mode until it reaches a consistent state, and then comes online as a brand new cluster

To optimize the time it takes to restore for a clone, we recommend taking a backup of the cluster you want to clone. You can do this with the `pgo backup` command, and choose if you want to take a full, differential, or incremental backup.

Future work will be focused on additional options, such as being able to clone a PostgreSQL cluster to a particular point-in-time (so long as the backup is available to support it) and supporting other `pgo create` flags.

Schedule Backups With Retention Policies

While the PostgreSQL Operator has had the ability to schedule full, incremental, and differential pgBackRest backups for awhile, it has not been possible to set the retention policy on these backups. Backup retention policies allow users to manage their backup storage whle maintaining enough backups to be able to recover to a specific point-in-time, or perform forensic work on data in a particular state.

For example, one can schedule a full backup to occur nightly at midnight and keep up to 21 full backups (e.g. a 21 day retention policy):

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

Breaking Changes

Feature Removals

- Physical backups using `pg_basebackup` are no longer supported. Any command-line option that references using this method has been removed. The API endpoints where one can specify a `pg_basebackup` remain, but will be removed in a future release (likely the next one).
- Removed the `pgo-lspvc` container. This container was used with the `pgo show pvc` and performed searches on the mounted filesystem. This would cause issues both on environments that could not support a PVC being mounted multiple times, and for underlying volumes that contained numerous files. Now, `pgo show pvc` only lists the PVCs associated with the PostgreSQL clusters managed by the PostgreSQL Operator.
- Native support for pgpool has been removed.

Command Line (`pgo`)

`pgo create cluster`

- The `--pgbackrest` option is removed as it is no longer needed. pgBackRest is enabled by default

`pgo delete cluster`

The default behavior for `pgo delete cluster` has changed so that **all backups and PostgreSQL data are deleted by default**.

To keep a backup after a cluster is deleted, one can use the `--keep-backups` flag with `pgo delete cluster`, and to keep the PostgreSQL data directory, one can specify the `--keep-data` flag. There is a plan to remove the `--keep-data` flag in a future release, though this has not been determined yet.

The `-b`, `--delete-backups`, `-d`, and `--delete-data` flags are all deprecated and will be removed in the next release.

`pgo scaledown`

With this release, `pgo scaledown` will **delete the PostgreSQL data directory of the replica by default.** To keep the PostgreSQL directory after the replica has scaled down, one can use the `--keep-data` flag.

`pgo test`

`pgo test` is optimized to provide faster results about the availability of a PostgreSQL cluster. Instead of attempting to make PostgreSQL connections to each PostgreSQL instance with each user, `pgo test` now checks the availability of the service endpoints for each PostgreSQL cluster as well as the output of the PostgreSQL readiness checks, which check the connectivity of a PostgreSQL cluster.

Both the API and the output of `pgo test` are modified for this optimization.

Additional apiserver Changes

- An authorization failure in the `apiserver` (i.e. not having the correct RBAC permission for a `pgouser`) will return a status code of `403` instead of `401`
- The pgorole permissions now support the `"*"` permission to specify *all* pgorole RBAC permissions are granted to a pgouser. Users upgrading from an earlier version should note this change if they want to assign their users to access new features.

Additional Features

pgo (Operator CLI)

- Support the pgBackRest options for backup retention, including `--repo1-retention-full`, `--repo1-retention-diff`, `--repo1-retention-archive`, `--repo1-retention-archive-type`, which can be added in the `--backup-opts` flag in the `pgo backup` command. For example:

```
# create a pgBackRest incremental backup with one full backup being retained and two differential
    backups being retained, along with incremental backups associated with each
pgo backup mycluster --backup-type="pgbackrest" --backup-opts="--type=incr
    --repo1-retention-diff=2 --repo1-retention-full=1"

# create a pgBackRest full backup where 2 other full backups are retained, with WAL archive
    retained for full and differential backups
pgo backup mycluster --backup-opts="--type=full --repo1-retention-full=2
    --repo1-retention-archive=4 --repo1-retention-archive-type=diff"
```

- Allow for users to define S3 credentials and other options for pgBackRest backups on a per-cluster basis, in addition to leveraging the globally provided templates. This introduces the following flags on the `pgo create cluster` command:
- `--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
- Add the `--disable-tls` flag to the `pgo` command-line client, as to be compatible with the Operator API server that is deployed with `DISABLE_TLS` enabled.
- Improved output for the `pgo scaledown --query` and and `pgo failover --query` commands, including providing easy-to-understand results on replication lag
- Containerized `pgo` via the `pgo-client` container. This can be installed from the Ansible installer using the `pgo_client_container_ins` flag, and it installs into the same namespace as the PostgreSQL Operator. You can connect to the container via `kubectl exec` and execute pgo commands!

Builds

- Refactor the Dockerfiles to rely on a "base" definition for ease of management and to ensure consistent builds across all container images during a full `make`
- Selecting which image builder to use is now argument based using the `IMGBUILDER` environmental variable. Default is `buildah`
- Optimize `yum clean` invocation to occur on same line as the `RUN`, which leads to smaller image builds.

Installation

- Add the `pgo_noauth_routes` (Ansible) / `NOAUTH_ROUTES` (Bash) configuration variables to disable TLS/BasicAuth authentication on particular API routes. This defaults to `'/health'`
- Add the `pgo_tls_ca_store` Ansible / `TLS_CA_TRUST` (Bash) configuration variables to specify a PEM encoded list of trusted certificate authorities (CA) for the Operator to use when authenticating API requests over TLS
- Add the `pgo_add_os_ca_store` / `ADD_OS_TRUSTSTORE` (Bash) to specify to use the trusted CA that is provided by the operating system. Defaults to `true`

Configuration

- Enable individual ConfigMap files to be customized without having to upload every single ConfigMap file available in `pgo-config`. Patch by Conor Quin (@Conor-Quin)
- Add `EXCLUDE_OS_TRUST` environmental variable that allows the `pgo` client to specify that it does not want to use the trusted certificate authorities (CA) provided by the operating system.

Miscellaneous

- Migrated Kubernetes API groups using API version `extensions/v1beta1` to their respective updated API groups/versions. This improves compatibility with Kubernetes 1.16 and beyond. Original patch by Lucas Bickel (@hairmare)
- Add a Kubernetes Service Account to every Pod that is managed by the PostgreSQL Operator
- Add support for private repositories using `imagePullSecret`. This can be configured during installation by setting the `pgo_image_pull_secret` and `pgo_image_pull_secret_manifest` in the inventory file using Ansible installer, or with the `PGO_IMAGE_PULL_SECRET` and `PGO_IMAGE_PULL_SECRET_MANIFEST` environmental variables using the Bash installer. The "pull secret" is the name of the pull secret, whereas the manifest is what is used to define the secret
- The pgorole permissions now support the `"*"` permission to specify *all* pgorole RBAC permissions are granted to a pgouser
- Policies that are executed using `pgo apply` and `pgo create cluster --policies` are now executed over a UNIX socket directly on the Pod of the primary PostgreSQL instance. Reported by @yuanlinios
- A new sidecar, `crunchyadm`, is available for running managemnet commands on a PostgreSQL cluster. As this is experimental, this feature is disabled by default.

Fixes

- Update the YAML library to v2.2.4 to mitigate issues presented in CVE-2019-11253
- Specify the `pgbackrest` user by its ID number (2000) in the backrest-repo container image so that containers instantiated with the `runAsNonRoot` option enabled recognize the `pgbackrest` user as non-root.
- Ensure any Kubernetes Secret associated with a PostgreSQL backup is deleted when the `--delete-backups` flag is specified on `pgo delete cluster`
- The pgBouncer pod can now support connecting to databases that are added after a PostgreSQL cluster is deployed
- Remove misleading error messages from the logs that were caused by the readiness/liveness probes on the `apiserver` and `event` containers in the `postgres-operator` pod
- Several fixes to the cleanup of a PostgreSQL cluster after a deletion event (e.g. `pgo delete cluster`) to ensure data is safely removed. This includes ensuring schedules managed by `pgo schedule` are removed, as well as PostgreSQL cluster and backup data
- Skip the HTTP Basic Authorization check if the `BasicAuth` parameter in `pgo.yaml` is set to `false`
- Ensure all available backup types are displayed in the `pgo schedule` are listed (full, incr, diff)
- Ensure schedule tasks create with `pgo create schedule` are deleted when `pgo delete cluster` is called
- Fix the missing readiness/liveness probes used to check the status of the `apiserver` and `event` containers in the `postgres-operator` pod
- Remove misleading error messages from the logs that were caused by the readiness/liveness probes on the `apiserver` and `event` containers in the `postgres-operator` pod
- Fix a race condition where the `pgo-rmdata` job could fail when doing its final pass on deleting PVCs. This became noticeable after adding in the task to clean up any configmaps that a PostgreSQL cluster relied on

- Improved logging around authorization failures in the apiserver

This release was to update the supported PostgreSQL versions to 12.2, 11.7, 10.12, 9.6.17, and 9.5.21

Crunchy Data announces the release of PostgreSQL Operator 4.1.1 on November, 22, 2019.

Postgres Operator 4.1.1 provide bug fixes and continued support to Postgres Operator 4.1 as well as continued compatibility with newer versions of PostgreSQL.

The PostgreSQL Operator is released in conjunction with the Crunchy Container Suite.

The Postgres Operator 4.1.1 release includes the following software versions upgrades:

- The PostgreSQL now uses versions 12.1, 11.6, 10.11, 9.6.16, and 9.5.20.

Postgres Operator is tested with Kubernetes 1.13 - 1.15, OpenShift 3.11+, Google Kubernetes Engine (GKE), and VMware Enterprise PKS 1.3+. At present, Postgres Operator 4.1 is **not** compatible with Kubernetes 1.16.

Fixes

- Add the `--disable-tls` flag to the `pgo` command-line client, as to be compatible with the Operator API server that is deployed with `DISABLE_TLS` enabled. This is backported due to this functionality being missed in the 4.1 release.
- Update the YAML library to v2.2.4 to mitigate issues presented in CVE-2019-11253
- Specify the `pgbackrest` user by its ID number (2000) in the backrest-repo container image so that containers instantiated with the `runAsNonRoot` option enabled recognize the `pgbackrest` user as non-root.
- Ensure any Kubernetes Secret associated with a PostgreSQL backup is deleted when the `--delete-backups` flag is specified on `pgo delete cluster`
- Enable individual ConfigMap files to be customized without having to upload every single ConfigMap file available in `pgo-config`. Patch by Conor Quin (@Conor-Quin)
- Skip the HTTP Basic Authorization check if the `BasicAuth` parameter in `pgo.yaml` is set to `false`

Crunchy Data announces the release of PostgreSQL Operator 4.1 on October 15, 2019.

In addition to new features, such as dynamic namespace manage by the Operator and the ability to subscribe to a stream of lifecycle events that occur with PostgreSQL clusters, this release adds many new features and bug fixes.

The Postgres Operator 4.1 release also includes the following software versions upgrades:

- The PostgreSQL now uses versions 11.5, 10.10, 9.6.15, and 9.5.19. The PostgreSQL container now includes support for PL/Python.
- pgBackRest is now 2.17
- pgMonitor now uses version 3.2

To build Postgres Operator 4.1, you will need to utilize buildah version 1.9.0 and above.

Postgres Operator is tested with Kubernetes 1.13 - 1.15, OpenShift 3.11+, Google Kubernetes Engine (GKE), and VMware Enterprise PKS 1.3+. At present, Postgres Operator 4.1 is **not** compatible with Kubernetes 1.16.

Major Features

Dynamic Namespace Management

Postgres Operator 4.1 introduces the ability to dynamically management Kubernetes namespaces from the Postgres Operator itself. Kubernetes namespaces provide the ability to isolate workloads within a Kubernetes cluster, which can provide additional security and privacy amongst users.

The previous version of the Postgres Operator allowed users to add Kubernetes namespaces to which the Postgres Operator could deploy and manage clusters. Postgres Operator 4.1 expands on this ability by allowing the Operator to dynamically add and remove namespaces using the `pgo create namespace` and `pgo delete namespace` commands.

This allows for several different deployment patterns for PostgreSQL clusters, including:

- Deploying PostgreSQL clusters within a single namespace for an individual user
- Deploying a PostgreSQL cluster into its own namespace

Note that deleting a namespace in Kubernetes deletes all of the objects that reside within that namespace, **including active PostgreSQL clusters**. Ensure that you wish to delete everything inside a namespace before executing `pgo delete namespace`.

This has also lead to a change in terms of how role-based access control (RBAC) is handled. Traditionally, RBAC permissions we added to the `ClusterRole` objects, but in order to support dynamic namespace management, the RBAC has been moved to the `Role` objects.

If you would like to use the dynamic namespace feature Kubernetes 1.11 and OpenShift 3.11, you will also need to utilize the `add-targeted-namespace.sh` script that is bundled with Postgres Operator 4.1. To add a namespace to dynamically to your Postgres Operator deployment in Kubernetes 1.11, you first need to create the namespace with `kubectl` (e.g. `kubectl create namespace yournamespace`) and then run the `add-targeted-namespace.sh` script (`./add-targeted-namespace.sh yournamespace`).

Lifecycle Events

Postgres Operator 4.1 now provides PostgreSQL lifecycle events that occur during the operation of a cluster. Lifecycle events include things such as when a cluster is provisioned, a replica is added, a backup is taken, a cluster fails over, etc. Each deployed PostgreSQL cluster managed by the PostgreSQL Operator will report back to the Operator around these lifecycle events via the NSQ distributed messaging platform.

You can subscribe to lifecycle events by topic using the `pgo watch` command. For subscribe to all events for clusters under management, you can run `pgo watch alltopic`. Eventing can be disabled using the `DISABLE_EVENTING` environmental variable within the `postgres-operator` deployment.

For more information, please read the [Eventing]({{< relref "/architecture/eventing.md" >}}) section of the documentation.

Breaking Changes

Containers

- The `node_exporter` container is no longer shipped with the PostgreSQL Operator. A detailed explanation of how node-style metrics are handled is available in the "Additional Features" section.

API

- The `pgo update cluster` API endpoint now uses a HTTP `POST` instead of `GET`
- The user management endpoints (e.g. `pgo create user`) now use a HTTP `POST` instead of a `GET`.

Command-line interface

- Removed the `-db` flag from `pgo create user` and `pgo update user`
- Removed `--update-password` flag from the `pgo user` command

Installation

- Changed the Ansible installer to use `uninstall` and `uninstall-metrics` tags instead of `deprovision` and `deprovision-metrics` respectively

Builds

- The `Makefile` now uses `buildah` for building the containers instead of `Docker`. The PostgreSQL Operator can be built with buildah v1.9.0 and above.

Additional Features

General PostgreSQL Operator Features

- PostgreSQL Operator users and roles can now be dynamically managed (i.e. `pgouser` and `pgorole`)
- Readiness probes have been added to the `apiserver` and `scheduler` and is now included in the new `event` container. The `scheduler` uses a special `heartbeat` task to provide its readiness.

- Added the `DISABLE_TLS` environmental variable for `apiserver`, which allows the API server to run over HTTP.
- Added the `NOAUTH_ROUTES` environmental variable for `apiserver`, which allows useres to bypass TLS authentication on certain routes (e.g. `/health`). At present, only `/health` can be used in this variable.
- Services ports for the postgres_exporter and pgBadger are now templated so a user can now customize them beyond the defaults.

PostgreSQL Upgrade Management

- The process to perform a minor upgrade on a PostgreSQL deployment was modified in order to minimize downtime. Now, when a `pgo upgrade cluster` command is run, the PostgreSQL Operator will upgrade all the replicas to the desired version of PostgreSQL before upgrading the primary container. If `autofail` is enabled, the PostgreSQL Operator will failover to a pod that is already updated to a newer version, which minimizes downtime and allows the cluster to upgrade to the desired, updated version of PostgreSQL.
- `pgo upgrade` now supports the `--all` flag, which will upgrade every single PostgreSQL cluster managed by the PostgreSQL Operator (i.e. `pgo upgrade --all`)

PostgreSQL User Management

- All user passwords are now loaded in from Kubernetes Secrets.
- `pgo create user --managed` now supports any acceptable password for PostgreSQL
- Improved error message for calling the equivalent `pgo show user` command when interfacing with the API directly and there are no clusters found for th euser.

Monitoring

- Updated the Grafana dashboards to use those found in pgMonitor v3.2
- The `crunchy-collect` container now connects to PostgreSQL using a password that is stored in a Kubernetes secret
- Introduced support for collecting host-style metrics via the cAdvisor installations that are installed and running on each Kubelet. This requires for the `ClusterRole` to have the `nodes` and `nodes/metrics` resources granted to it.

Logging

- Updated logging to provide additional details of where issues occurred, including file and line number where the issue originated.

Installation

- The Ansible installer `uninstall` tag now has the option of preserving portions of the previous installation
- The Ansible installer supports NFS and hostpath storage options
- The Ansible installer can now set the fsgroup for the `metrics` tag
- The Ansible installer now has the same configuration options as the bash installer
- The Ansible installer now supports a separate RBAC installation
- Add a custom security context constraint (SCC) to the Ansible and bash installers that is applied to pods created by the Operator. This makes it possible to customize the control permissions for the PostgreSQL cluster pods managed by the Operator

Fixes

- Fixed a bug where `testuser` was always created even if the username was modified in the `pgo.yaml`
- Fixed the `--expired` flag for `pgo show user` to show the number of days until a user's password expires
- Fixed the workflow for `pgo benchmark` jobs to show completion
- Modify the create a cluster via a custom resource definition (CRD) to use pgBackRest
- Fixed an issue with the `pgpool` label when a `pg_dump` is performed by calling the REST API
- Fixed the `pgo load` example, which previous used a hardcoded namespace. This has changed with the support of dynamic namespaces.

There are a few options available for community support of the PostgreSQL Operator:

- **If you believe you have found a bug** or have a detailed feature request: please open an issue on GitHub. The PostgreSQL Operator community and the Crunchy Data team behind the PostgreSQL Operator is generally active in responding to issues.
- **For general questions or community support**: please join the PostgreSQL Operator community mailing list at https://groups.google.com/a/crunchydata.com/forum/#!forum/postgres-operator/join,

In all cases, please be sure to provide as many details as possible in regards to your issue, including:

- Your Platform (e.g. Kubernetes vX.YY.Z)
- Operator Version (e.g. {{< param centosBase >}}-{{< param operatorVersion >}})
- A detailed description of the issue, as well as steps you took that lead up to the issue
- Any relevant logs
- Any additional information you can provide that you may find helpful

For production and commercial support of the PostgreSQL Operator, please contact Crunchy Data at info@crunchydata.com for information regarding an Enterprise Support Subscription.