
Table of Contents

1. Documentation	3
2. What is the Operator?	3
3. Design	3
4. Requirements	4
5. Installation	4
6. Configuration	4
7. Getting Started	5
8. Installation	5
9. Next Steps	5
10. Overview	5
11. Quickstart	6
11.1. GKE/PKS	6
11.2. Openshift Container Platform	7
12. Next Steps	7
13. Project Structure	7
14. Installation Prerequisites	8
15. Basic Installation	9
15.1. HostPath Persistent Volumes	9
15.2. NFS Persistent Volumes	9
16. Build Images & Deploy	9
16.1. Packaged Images	9
16.2. Build from Source	10
17. Makefile Targets	11
18. Next Steps	12
19. Helm Chart	12
20. Next Steps	12
21. Overview	12
22. Openshift Container Platform	12
23. Security Configuration	13
23.1. Kube RBAC	13
23.2. Basic Authentication	13
23.3. Configure TLS	14
23.4. pgo RBAC	14
23.5. REST API Configuration	16
23.6. PostgreSQL Operator Container Configuration	22
24. Bash Completion	22
25. REST API	23
26. Deploying pgPool	24
27. Storage Configuration	25
27.1. NFS	25
27.2. Dynamic	26
27.3. GKE	26
28. Verify Operator Status	27
29. Configure pgo Client	27
29.1. Running Kubernetes Locally	28
30. Verify pgo Client	29

31. Next Steps	29
32. Required Updates	31
32.1. Configuration File	32
32.2. Secrets	32
33. Required Updates	32
33.1. Configuration File	32
33.2. Container Resources	32
33.3. Kube RBAC	33
33.4. Application RBAC	33
33.5. User Creation	33
33.6. Replica CRD	34
34. First Steps	34
35. Cluster Names	34
36. General	34
36.1. Operator Version	34
36.2. Operator Status	34
36.3. Operator Configuration	35
36.4. Disk Capacity	35
37. Cluster Basics	35
37.1. Create Cluster	35
37.2. Delete Cluster	39
37.3. Show Cluster	40
37.4. Test Connection	41
38. Administration	42
38.1. Reload	42
38.2. Backups	42
38.3. Scheduling	44
38.4. Scaling Replicas	46
38.5. Manual Failover	48
38.6. Upgrading PostgreSQL	49
38.7. Labels	50
38.8. Creating SQL Policies	51
38.9. Loading Data	52
39. Authentication	54
39.1. Credential Management	54
40. pgbouncer Basics	56
41. pgpool Basics	56
41.1. Create pgpool	56
41.2. Delete pgpool	57
41.3. Workflow	57
42. Reference Architecture	58
43. Custom Resource Definitions	58
44. Command Line Interface	58
45. Operator Deployment	59
46. CLI Design	59
46.1. Verbs	59
47. Affinity	59
48. Debugging	60
49. Persistent Volumes	60

50. PostgreSQL Operator Deployment Strategies	60
50.1. Strategies	60
50.2. Specifying a Strategy	61
50.3. Strategy Template Files	61
50.4. Default Cluster Deployment Strategy (1)	61
50.5. Cluster Deletion	62
50.6. Custom Postgres Configurations	62
50.7. Metrics Collection	64
50.8. Manual Failover	64
50.9. Auto Failover	65

title: "Crunchy Data PostgreSQL Operator" date: 2018-04-23T14:52:09-07:00 draft: false

Latest Release: 3.4.0 2018-12-04

1. Documentation

Please view the official Crunchy Data PostgreSQL Operator documentation here [<https://crunchydata.github.io/postgres-operator/stable/>]. If you are interested in contributing or making an update to the documentation, please view the Contributing Guidelines [[/contributing/](#)].

2. What is the Operator?

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

Use the postgres-operator to -

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

3. Design

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

-
- adds Custom Resource Definitions for PostgreSQL to Kubernetes
 - adds controller logic that watches events on PostgreSQL resources
 - provides a command line client (**pgo**) and REST API for interfacing with the postgres-operator
 - provides for very customized deployments including container resources, storage configurations, and PostgreSQL custom configurations

More design information is found on the How It Works [/how-it-works/] page.

4. Requirements

The postgres-operator runs on any Kubernetes and Openshift platform that supports Custom Resource Definitions.

The Operator project builds and operates with the following containers -

- PVC Listing Container [<https://hub.docker.com/r/crunchydata/pgo-lspvc/>]
- Remove Data Container [<https://hub.docker.com/r/crunchydata/pgo-rmdata/>]
- postgres-operator Container [<https://hub.docker.com/r/crunchydata/postgres-operator/>]
- apiserver Container [<https://hub.docker.com/r/crunchydata/pgo-apiserver/>]
- file load Container [<https://hub.docker.com/r/crunchydata/pgo-load/>]
- backrest interface Container [<https://hub.docker.com/r/crunchydata/pgo-backrest/>]

This Operator is developed and tested on the following operating systems but is known to run on other operating systems -

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

5. Installation

To build and deploy the Operator on your Kubernetes system, follow the instructions documented on the Installation [/installation/] page.

If you're seeking to upgrade your existing Operator installation, please visit the Upgrading the Operator [/installation/upgrading-the-operator/] page.

6. Configuration

The operator is template-driven; this makes it simple to configure both the client and the operator. The configuration options are documented on the Configuration [/installation/configuration/] page.

7. Getting Started

postgres-operator commands are documented on the Getting Started [/getting-started/] page.

title: "Installation" date: 2018-04-24T18:27:02-07:00 draft: false weight: 10

Latest Release: 3.4.0 2018-12-04

8. Installation

For a quick deployment on either a GKE or OpenShift environment, visit the Quick Installation [/installation/quick-installation/] page.

For a manual installation of the Operator on either a Kubernetes or OpenShift environment, visit the Manual Installation [/installation/manual-installation/] page.

A Helm Chart [/installation/helm-chart/] is also provided.

If you're looking to upgrade a current PostgreSQL Operator installation, visit the Upgrading the Operator [/installation/upgrading-the-operator/] page.

There are many ways to configure the operator further. Some sample configurations are documented on the Configuration [/installation/configuration/] page. This includes setting up security and storage configurations for your environment.

After completing the installation steps, ensure you visit the Deployment [/installation/deployment/] page to deploy the Operator to your environment.

9. Next Steps

You may want to find out more information on how the operator is designed to work and deploy. This information can be found in the How It Works [/how-it-works/] page.

Information can be found on the full scope of commands on the Getting Started [/getting-started/] page.

title: "Quick Installation" date: 2018-04-26T15:22:14-07:00 draft: false weight: 10

Latest Release: 3.4.0 2018-12-04

10. Overview

There are currently **quickstart** script that seek to automate the deployment to popular Kubernetes environments -

- quickstart.sh [https://github.com/CrunchyData/postgres-operator/blob/master/examples/quickstart.sh]

The **quickstart** script will deploy the operator to a GKE Kube cluster or an Openshift Container Platform cluster. The **quickstart** script is intended to get you up and running quickly, for a typical more custom installation, the **manual** installation is recommended.

The script assumes you have a StorageClass defined for persistence.

Pre-compiled versions of the Operator **pgo** client are provided for the x86_64, Mac OSX, and Windows hosts.

11. Quickstart

11.1. GKE/PKS

The **quickstart.sh** script will allow users to set up the Postgres Operator quickly on GKE, PKS, and Openshift.

The script requires a few things in order to work -

- wget utility installed
- kubectl or oc utility installed
- StorageClass defined on your GKE instance

Executing the script will give you a default Operator deployment that assumes **dynamic** storage and a storage class named **standard**, user provided values are also allowed by the script to override these defaults.

The script performs the following -

- downloads the Operator configuration files
- sets the \$HOME/.pgouser file to default settings
- deploys the Operator Deployment
- sets your .bashrc to include the Operator environment variables
- sets your \$HOME/.bash_completion file to be the **pgo** bash_completion file

Note

You should copy the **quickstart.sh** script from github rather than cloning the entire github Operator repository!

A tip, if you want to set your Kube context to some particular namespace you can run commands similar to this to set it to a **demo** namespace if that namespace has already been created on your GKE cluster:

```
kubectl create -f $COROOT/examples/demo-namespace.json
kubectl config set-context demo --cluster=gke_crunchy-a-test_us-centrall1-a_usera-quickst
kubectl config use-context demo
```

For Mac and Windows users, pre-built **pgo** binaries are included in the operator release tar ball, you would download the pgo CLI binaries from the Releases page to your local machine as part of the quick installation:

-
- pgo-mac is the Mac binary
 - pgo.exe is the Windows binary
 - pgo is the Linux binary
 - expenv-mac is the expenv binary for Mac
 - expenv.exe is the expenv binary for Windows

Currently the quickstart scripts are meant for Linux installs, you will need to modify this script for Windows or Mac installs until we support and provide Windows and Mac installation scripts.

11.2. Openshift Container Platform

The script also is used for installing the operator on OCP.

12. Next Steps

Next, visit the Deployment [/installation/deployment/] page to deploy the Operator, verify the installation, and view various storage configurations.

title: "Manual Installation" date: 2018-04-26T15:22:21-07:00 draft: false weight: 20

Latest Release: 3.4.0 2018-12-04

13. Project Structure

First, define the following environment variables in **.bashrc**:

```
export GOPATH=$HOME/odev
export GOBIN=$GOPATH/bin
export PATH=$PATH:$GOBIN
export CO_NAMESPACE=demo
export CO_CMD=kubectl
export COROOT=$GOPATH/src/github.com/crunchydata/postgres-operator
export CO_IMAGE_PREFIX=crunchydata
export CO_BASEOS=centos7
export CO_VERSION=3.4.0
export CO_IMAGE_TAG=$CO_BASEOS-$CO_VERSION
```

```
# for the pgo CLI auth
export PGO_CA_CERT=$COROOT/conf/postgres-operator/server.crt
export PGO_CLIENT_CERT=$COROOT/conf/postgres-operator/server.crt
export PGO_CLIENT_KEY=$COROOT/conf/postgres-operator/server.key
```

```
# for crunchy-scheduler startup
export CCP_IMAGE_PREFIX=crunchydata
export CCP_IMAGE_TAG=centos7-10.6-2.2.0
```

```
# useful aliases
```

```
alias setip='export CO_APISERVER_URL=https://`kubectl get service postgres-operator -o=jsonpath='{.spec.ports[0].ipAddress}'`'
alias alog='kubectl logs `kubectl get pod --selector=name=postgres-operator -o jsonpath='{.items[0].metadata.name}'`'
alias olog='kubectl logs `kubectl get pod --selector=name=postgres-operator -o jsonpath='{.items[0].metadata.name}'`'
```

If you have access to the Crunchy RHEL images, you would change the above references to centos7 to rhel7.

When deploying on Openshift Container Platform, the CO_CMD environment variable should be:

```
export CO_CMD=oc
```

To perform an installation of the operator, first create the project structure as follows on your host, here we assume a local directory called **odev** -

```
. .bashrc
mkdir -p $HOME/odev/src $HOME/odev/bin $HOME/odev/pkg $GOPATH/src/github.com/crunchydata
```

Next, get a tagged release of the source code -

```
cd $GOPATH/src/github.com/crunchydata
git clone https://github.com/CrunchyData/postgres-operator.git
cd postgres-operator
git checkout 3.4.0
```

14. Installation Prerequisites

To run the operator and the **pgo** client, you will need the following -

- a running Kubernetes or OpenShift cluster
- the kubectl or oc clients installed in your PATH and configured to connect to the cluster (e.g. export KUBECONFIG=/etc/kubernetes/admin.conf)
- a Kubernetes namespace created and set to where you want the operator installed. For this install we assume a namespace of **demo** has been created.

```
kubectl create -f examples/demo-namespace.json
kubectl config set-context $(kubectl config current-context) --namespace=demo
kubectl config view -o "jsonpath={.contexts[?(@.name==\"$(kubectl config current-context)\"}]}"
```

On Openshift Container Platform, you would have a Project and User defined for installing the Operator.

Run the Makefile `setup` target to install dependencies.

```
make setup
```

Next, run the Makefile `installrbac` target as a user with cluster-admin privileges, not as a normal Kube or Openshift user. This target creates the RBAC roles and CRDs required by the Operator and is only required to be created one time.

For example, on an Openshift system you would run this target as follows using the system:admin Openshift user:

```
$ sudo su -
# oc login -u system:admin
# cd /home/oper
# . .bashrc
# export PATH=$PATH:/home/oper/odev/bin
# cd odev/src/github.com/crunchydata/postgres-operator
# make installrbac
```

On a Kube system, you would be connected as a cluster-admin user and just issue:

```
# cd /home/oper
# . .bashrc
# export PATH=$PATH:/home/oper/odev/bin
# cd odev/src/github.com/crunchydata/postgres-operator
make installrbac
```

15. Basic Installation

The basic pgo.yaml configuration specifies 3 different storage configurations: * hostpath * nfs (default) * storage-class

Storage configurations are documented here: [here](#) [/installation/configuration/_storage_configuration].

The default storage configuration used for creating Primary, Replica, and Backups is set to NFS in the default pgo.yaml file. Adjust this setting to meet your storage requirements.

Sample PV creation scripts are found in the following directory:

`examples/pv`

15.1. HostPath Persistent Volumes

The default Persistent Volume script assumes a default HostPath directory be created called **/data**:

```
sudo mkdir /data
sudo chmod 777 /data
```

Create some sample Persistent Volumes using the following script:

```
$COROOT/pv/create-pv.sh
```

15.2. NFS Persistent Volumes

The NFS Persistent Volume script assumes a default directory be created called **/nfsfileshare** as the NFS mount point on your system:

```
sudo ls /nfsfileshare
```

See the [crunchy-containers documentation](#) on how to install NFS on a centos/RHEL system if you want to use NFS for testing the operator.

Create some sample NFS Persistent Volumes using the following script:

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

16. Build Images & Deploy

16.1. Packaged Images

To pull prebuilt versions from Dockerhub of the **postgres-operator** containers, execute the following Makefile target -

```
make pull
```

To pull down the prebuilt **pgo** binaries, download the **tar.gz** release file from the following link -

- Github Releases [<https://github.com/CrunchyData/postgres-operator/releases>]
- extract (e.g. `tar xvzf postgres-operator.3.4.0.tar.gz`)

```
cd $HOME
tar xvzf ./postgres-operator.3.4.0.tar.gz
```

- copy **pgo** client to somewhere in your path (e.g. `cp pgo /usr/local/bin`)

Next, deploy the operator to your Kubernetes cluster -

```
cd $COROOT
make deployoperator
```

Warning

If you make configuration file changes you will need to re-run the `deployoperator` makefile target to re-deploy the Operator with the new configuration files.

16.2. Build from Source

The purpose of this section is to illustrate how to build the PostgreSQL Operator from source. These are considered advanced installation steps and should be primarily used by developers or those wishing a more precise installation method.

Requirements

The `postgres-operator` runs on any Kubernetes and OpenShift platform that supports Custom Resource Definitions. The Operator is tested on Kubeadm and OpenShift Container Platform environments.

The operator is developed with the Golang versions greater than or equal to version 1.8. See Golang website [<https://golang.org/dl/>] for details on installing golang.

The Operator project builds and operates with the following containers -

- PVC Listing Container [<https://hub.docker.com/r/crunchydata/pgo-lspvc/>]
- Remove Data Container [<https://hub.docker.com/r/crunchydata/pgo-rmdata/>]
- postgres-operator Container [<https://hub.docker.com/r/crunchydata/postgres-operator/>]
- apiserver Container [<https://hub.docker.com/r/crunchydata/pgo-apiserver/>]
- file load Container [<https://hub.docker.com/r/crunchydata/pgo-load/>]
- pgbackrest interface Container [<https://hub.docker.com/r/crunchydata/pgo-backrest/>]

This Operator is developed and tested on the following operating systems but is known to run on other operating systems -

- **CentOS 7**

- RHEL 7

17. Makefile Targets

The following table describes the Makefile targets -

Table 1. Makefile Targets

Target	Description
macpgo	build the Mac version of the pgo CLI binary
winpgo	build the Windows version of the pgo CLI binary
installrbac	only run once and by a cluster-admin user, this target creates the Operator CRDs and RBAC resources required by the Operator
setupnamespace	only run once, will create a namespace called demo
bounce	delete the Operator pod only, this is a way to upgrade the operator without a full redeploy, as the operator runs in a Deployment, a new pod will be created to replace the old one, a simple way to bounce the pod
deployoperator	deploy the Operator (apiserver and postgers-operator) to Kubernetes
all	compile all binaries and build all images
setup	fetch the dependent packages required to build with, and create Kube RBAC resources
main	compile the postgres-operator
pgo	build the pgo binary
clean	remove binaries and compiled packages, restore dependencies
operatorimage	compile and build the postgres-operator Docker image
apiserverimage	compile and build the apiserver Docker image
lsimage	build the lspvc Docker image
loadimage	build the file load Docker image
rmdataimage	build the data deletion Docker image

Target	Description
pgo-backrest-image	build the pgbacrest interface Docker image
release	build the postgres-operator release

18. Next Steps

Next, visit the Deployment [/installation/deployment/] page to deploy the Operator, verify the installation, and view various storage configurations.

title: "Helm Chart" date: 2018-04-26T15:24:16-07:00 draft: false weight: 30

Latest Release: 3.4.0 2018-12-04

19. Helm Chart

First, pull prebuilt versions from Dockerhub of the **postgres-operator** containers, specify the image versions, and execute the following Makefile target -

```
export CO_IMAGE_PREFIX=crunchydata
export CO_IMAGE_TAG=centos7-3.4.0
make pull
```

Then, build and deploy the operator using the provided Helm chart -

```
cd $COROOT/chart
helm install ./postgres-operator
helm ls
```

20. Next Steps

Next, visit the Deployment [/installation/deployment/] page to deploy the Operator, verify the installation, and view various storage configurations.

title: "Configuration" date: 2018-04-24T18:26:56-07:00 draft: false weight: 40

Latest Release: 3.4.0 2018-12-04

21. Overview

This document describes how to configure the operator beyond the default configurations in addition to detailing what the configuration settings mean.

22. Openshift Container Platform

To run the Operator on Openshift Container Platform note the following requirements -

- Openshift Container Platform 3.7 or greater is required due to the dependence on Custom Resource Definitions.

-
- The `CO_CMD` environment variable should be set to `oc` when operating in an Openshift environment.

23. Security Configuration

23.1. Kube RBAC

The `cluster-rbac.yaml` file is executed a single time when installing the Operator. This file, executed by a Kubernetes user with cluster-admin privileges, does the following:

- Creates Customer Resource Definitions
- Grants `get` access to Kube Node resources to the `postgres-operator` service account.

The `rbac.yaml` file is also executed a single time when installing the Operator. This file creates Role scoped privileges which are granted to the `postgres-operator` service account. The `postgres-operator` service account is used by the **apiserver** and **postgres-operator** containers to access Kubernetes resources.

Both of these RBAC files are executed by the `deploy/install-rbac.sh` script. It can also be installed through running `make installrbac` in the `$CCPROOT` directory.

Warning

The `CO_NAMESPACE` environment variable determines the namespace that is used within the deployment of the operator. If you are deploying to the **demo** namespace, the following should setting should be defined in your `.bashrc`: `export CO_NAMESPACE=demo`

See here [<https://kubernetes.io/docs/admin/authorization/rbac/>] for more details on how to enable RBAC roles and modify the scope of the permissions to suit your needs.

23.2. Basic Authentication

Basic authentication between the host and the `apiserver` is required. It will be necessary to configure the `pgo` client to specify a basic authentication username and password through the creation a file in the user's home directory named `.pgouser`. It will look similar to this, and contain only a single line -

```
username:password
```

The above excerpt specifies a username of **username** and a password of **password**. These values will be read by the **pgo** client and passed to the **apiserver** on each REST API call.

For the **apiserver**, a list of usernames and passwords is specified in the **pgo-auth-secret** Secret. The values specified in a deployment are found in the following location -

```
$COROOT/conf/postgres-operator/pgouser
```

The sample configuration for `pgouser` is as follows -

```
username:password:pgoadmin
testuser:testpass:pgoadmin
readonlyuser:testpass:pgoreader
```

Modify these values to be unique to your environment.

If the username and password passed by clients to the **apiserver** do not match, the REST call will fail and a log message will be produced in the **apiserver** container log. The client will receive a 401 HTTP status code if they are not able to authenticate.

If the `pgouser` file is not found in the **home** directory of the `pgo` user then the next searched location is `/etc/pgo/pgouser`. If the file is not found in either of the locations, the `pgo` client searches for the existence of a `PGOUSER` environment variable in order to locate a path to the basic authentication file.

Basic authentication can be entirely disabled by setting the `BasicAuth` setting in the `pgo.yaml` configuration file to `false`.

23.3. Configure TLS

TLS is used to secure communications to the **apiserver**. Sample keys and certifications that can be used by TLS are found here -

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

If you want to generate your own keys, you can use the script found in -

```
$COROOT/bin/make-certs.sh
```

The **pgo** client is required to use keys to connect to the **apiserver**. Specify the keys for `pgo` by setting the following environment variables -

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

You can also specify these credentials using the following command flags where you can reference they keys from any file path directly:

```
pgo version --pgo-ca-cert=/tmp/server.crt --pgo-client-cert=/tmp/server.crt --pgo-client-key=/tmp/server.key
```

The sample server keys are used as the client keys; adjust to suit security requirements.

For the **apiserver TLS configuration**, the keys are included in the **apiserver-conf-secret** Secret when the **apiserver** is deployed. See the `$COROOT/deploy/deploy.sh` script which is where the secret is created.

The **apiserver** listens on port 8443 (e.g. `https://postgres-operator:8443`) by default.

You can set `InsecureSkipVerify` to **true** by setting the `NO_TLS_VERIFY` environment variable in the `deployment.json` file to **true**. By default this value is set to **false** if you do not specify a value.

23.4. pgo RBAC

The `pgo` command line utility talks to the **apiserver** REST API instead of the Kubernetes API. It is therefore necessary for the `pgo` client to make use of RBAC configuration.

Starting in Release 3.0, the `/conf/postgres-operator/pgorole` is used to define some sample `pgo` roles, **pgadmin** and **pgreader**.

These roles are meant as examples that you can configure to suit security requirements as necessary. The **pgadmin** role grants a user authorization to all pgo commands. The **pgoreader** only grants access to pgo commands that display information such as `pgo show cluster`.

The `pgorole` file is read at start up time when the operator is deployed to the Kubernetes cluster.

Also, the `pgouser` file now includes the role that is assigned to a specific user as follows -

```
username:password:pgoadmin
testuser:testpass:pgoadmin
readonlyuser:testpass:pgoreader
```

The following list shows the current complete list of possible pgo permissions -

Table 2. pgo Permissions

Permission	Description
ShowSecrets	allow pgo show user
ShowCluster	allow pgo show cluster
CreateCluster	allow pgo create cluster
TestCluster	allow pgo test mycluster
ShowBackup	allow pgo show backup
CreateBackup	allow pgo backup mycluster
DeleteBackup	allow pgo delete backup mycluster
Label	allow pgo label
Load	allow pgo load
CreatePolicy	allow pgo create policy
DeletePolicy	allow pgo delete policy
ShowPolicy	allow pgo show policy
ApplyPolicy	allow pgo apply policy
ShowPVC	allow pgo show pvc
CreateUpgrade	allow pgo upgrade
ShowUpgrade	allow pgo show upgrade
DeleteUpgrade	allow pgo delete upgrade
CreateUser	allow pgo create user
CreateFailover	allow pgo failover
ShowConfig	allow pgo show config
User	allow pgo user
Version	allow pgo version

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

```
FATA[0000] Authentication Failed: 40
```

23.5. REST API Configuration

The postgres-operator pod includes the apiserver which is a REST API that pgo users are able to communicate with.

The apiserver uses the following configuration files found in `$COROOT/conf/postgres-operator` to determine how the Operator will provision PostgreSQL containers -

```
$COROOT/conf/postgres-operator/pgo.yaml
$COROOT/conf/postgres-operator/pgo.lspvc-template.json
$COROOT/conf/postgres-operator/pgo.load-template.json
```

Note that the default `pgo.yaml` file assumes you are going to use **HostPath** Persistent Volumes for your storage configuration. It will be necessary to adjust this file for NFS or other storage configurations. Some examples of how are listed in the manual installation document.

The version of PostgreSQL container the Operator will deploy is determined by the **CCPIImageTag** setting in the `$COROOT/conf/postgres-operator/pgo.yaml` configuration file. By default, this value is set to the latest release of the Crunchy Container Suite.

The default `pgo.yaml` configuration file, included in `$COROOT/conf/postgres-operator/pgo.yaml`, looks like this -

```
Cluster:
  PrimaryNodeLabel:
  ReplicaNodeLabel:
  CCPIImagePrefix:  crunchydata
  Metrics:  false
  Badger:  false
  CCPIImageTag:  centos7-10.6-2.2.0
  LogStatement:  none
  LogMinDurationStatement:  60000
  Port:  5432
  User:  testuser
  Database:  userdb
  PasswordAgeDays:  60
  PasswordLength:  8
  Strategy:  1
  Replicas:  0
  ArchiveMode:  false
  ArchiveTimeout:  60
  ServiceType:  ClusterIP
  Backrest:  false
  Autofail:  false
PrimaryStorage: hostpathstorage
BackupStorage: hostpathstorage
ReplicaStorage: hostpathstorage
Storage:
  hostpathstorage:
    AccessMode:  ReadWriteMany
    Size:  1G
    StorageType:  create
  nfsstorage:
    AccessMode:  ReadWriteMany
    Size:  1G
    StorageType:  create
    SupplementalGroups:  65534
```



```

storage2:
  AccessMode:  ReadWriteMany
  Size:  1G
  StorageType:  dynamic
  StorageClass:  gluster-heketi
  Fsgroup:  26
storage3:
  AccessMode:  ReadWriteOnce
  Size:  1G
  StorageType:  dynamic
  StorageClass:  rook-ceph-block
  Fsgroup:  26
DefaultContainerResources:
DefaultLoadResources:
DefaultLspvcResources:
DefaultRmdataResources:
DefaultBackupResources:
DefaultPgboouncerResources:
DefaultPgpoolResources:
ContainerResources:
  small:
    RequestsMemory:  512Mi
    RequestsCPU:  0.1
    LimitsMemory:  512Mi
    LimitsCPU:  0.1
  large:
    RequestsMemory:  2Gi
    RequestsCPU:  2.0
    LimitsMemory:  2Gi
    LimitsCPU:  4.0
Pgo:
  AutofailSleepSeconds:  9
  Audit:  false
  LSPVCTemplate:  /pgo-config/pgo.lspvc-template.json
  LoadTemplate:  /pgo-config/pgo.load-template.json
  COImagePrefix:  crunchydata
  COImageTag:  centos7-3.4.0-rc1

```

Values in the pgo configuration file have the following meaning:

Table 3. pgo Configuration File Definitions

Setting	Definition
BasicAuth	if set to true will enable Basic Authentication
Cluster.PrimaryNodeLabel	newly created primary deployments will specify this node label if specified, unless you override it using the --node-label command line flag, if not set, no node label is specified
Cluster.ReplicaNodeLabel	newly created replica deployments will specify this node label if specified, unless you override it using the --node-label command line flag, if not set, no node label is specified
Cluster.CCPIimageTag	newly created containers will be based on this image version (e.g. centos7-10.4-1.8.3), unless you override it using the --ccp-image-tag command line flag
Cluster.Port	the PostgreSQL port to use for new containers (e.g. 5432)

Setting	Definition
Cluster.LogStatement	postgresql.conf log_statement value (required field) (works with crunchy-postgres >= 2.2.0)
Cluster.LogMinDurationStatement	postgresql.conf log_min_duration_statement value (required field) (works with crunchy-postgres >= 2.2.0)
Cluster.User	the PostgreSQL normal user name
Cluster.Strategy	sets the deployment strategy to be used for deploying a cluster, currently there is only strategy 1
Cluster.Replicas	the number of cluster replicas to create for newly created clusters
Cluster.Metrics	boolean, if set to true will cause each new cluster to include crunchy-collect as a sidecar container for metrics collection, if set to false (default), users can still add metrics on a cluster-by-cluster basis using the pgo command flag --metrics
Cluster.Badger	boolean, if set to true will cause each new cluster to include crunchy-pgbadger as a sidecar container for static log analysis, if set to false (default), users can still add pgbadger on a cluster-by-cluster basis using the pgo create cluster command flag --pgbadger
Cluster.Policies	optional, list of policies to apply to a newly created cluster, comma separated, must be valid policies in the catalog
Cluster.PasswordAgeDays	optional, if set, will set the VALID UNTIL date on passwords to this many days in the future when creating users or setting passwords, defaults to 60 days
Cluster.PasswordLength	optional, if set, will determine the password length used when creating passwords, defaults to 8
Cluster.ArchiveMode	optional, if set to true will enable archive logging for all clusters created, default is false.
Cluster.ArchiveTimeout	optional, if set, will determine the archive timeout setting used when ArchiveMode is true, defaults to 60 seconds
Cluster.ServiceType	optional, if set, will determine the service type used when creating primary or replica services, defaults to ClusterIP if not set, can be overridden by the user on the command line as well
Cluster.Backrest	optional, if set, will cause clusters to have the pgbackrest volume PVC provisioned during cluster creation
Cluster.Autofail	optional, if set, will cause clusters to be checked for auto failover in the event of a non-Ready status
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 repo volumes

Setting	Definition
ReplicaStorage	required, the value of the storage configuration to use for the replica PostgreSQL deployments
Storage.storage1.StorageType	for a dynamic storage type, you can specify the storage class used for storage provisioning(e.g. standard, gold, fast)
Storage.storage1.AccessMode	the access mode for new PVCs (e.g. ReadWriteMany, ReadWriteOnce, ReadOnlyMany). See below for descriptions of these.
Storage.storage1.Size	the size to use when creating new PVCs (e.g. 100M, 1Gi)
Storage.storage1.StorageTypeSupport	supported values are either dynamic , create , if not supplied, create is used
Storage.storage1.Fsgroup	optional, if set, will cause a SecurityContext and fsGroup attributes to be added to generated Pod and Deployment definitions
Storage.storage1.SupplementalGroups	optional, if set, will cause a SecurityContext to be added to generated Pod and Deployment definitions
Storage.storage1.MatchLabels	optional, if set, will cause the PVC to add a matchlabels selector in order to match a PV, only useful when the StorageType is create , when specified a label of name=clustername is added to the PVC as a match criteria
DefaultContainerResource	optional, the value of the container resources configuration to use for all database containers, if not set, no resource limits or requests are added on the database container
DefaultLoadResource	optional, the value of the container resources configuration to use for pgo-load containers, if not set, no resource limits or requests are added on the database container
DefaultLspvcResource	optional, the value of the container resources configuration to use for pgo-lspvc containers, if not set, no resource limits or requests are added on the database container
DefaultRmdataResource	optional, the value of the container resources configuration to use for pgo-rmdata containers, if not set, no resource limits or requests are added on the database container
DefaultBackupResource	optional, the value of the container resources configuration to use for crunchy-backup containers, if not set, no resource limits or requests are added on the database container
DefaultPgbackerResource	optional, the value of the container resources configuration to use for crunchy-pgbacker containers, if not set, no resource limits or requests are added on the database container
DefaultPgpoolResource	optional, the value of the container resources configuration to use for crunchy-pgpool containers, if not set, no resource limits or requests are added on the database container
ContainerResources.small	request size of memory in bytes

Setting	Definition
ContainerResources.small	request size of CPU cores
ContainerResources.small	request size of memory in bytes
ContainerResources.small	request size of CPU cores
ContainerResources.large	request size of memory in bytes
ContainerResources.large	request size of CPU cores
ContainerResources.large	request size of memory in bytes
ContainerResources.large	request size of CPU cores
Pgo.LSPVCTemplate	the PVC lspvc template file that lists PVC contents
Pgo.LoadTemplate	the load template file used for load jobs
Pgo.COImagePrefix	image tag prefix to use for the Operator containers
Pgo.COImageTag	image tag to use for the Operator containers
Pgo.Audit	boolean, if set to true will cause each apiserver call to be logged with an audit marking

Storage Configurations

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
```

It will result in a PVC being created named **example1** and in the case of a backup job, the pvc is named **example1-backup**

There are currently 3 sample pgo configuration files provided for users to use as a starting configuration -

- `pgo.yaml.nfs` - this configuration specifies **create** storage to be used, this is used for NFS storage for example where you want to have a unique PVC created for each database
- `pgo.yaml.storageclass` - this configuration specifies **dynamic** storage to be used, namely a **storageclass** that refers to a dynamic provisioning storage such as StorageOS or Portworx, or GCE.

Note, when Storage Type is **create**, you can specify a storage configuration setting of **MatchLabels**, when set, this will cause a **selector** of **name=clustername** 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 name=myclustername
```

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.

Overriding Container Resources Configuration Defaults

In the **pgo.yaml** configuration file you have the option to configure a default container resources configuration that when set will add CPU and memory resource limits and requests values into each database container when the container is created.

You can also override the default value using the `--resources-config` command flag when creating a new cluster -

```
pgo create cluster testcluster --resources-config=large
```

Note, if you try to allocate more resources than your host or Kube cluster has available then you will see your pods wait in a **Pending** status. The output from a `kubectl describe pod` command will show output like this in this event -

```
Events:
  Type            Reason              Age             From              Message
  --            -
  Warning         FailedScheduling    49s (x8 over 1m)  default-scheduler  No nodes are available
```

Overriding Storage Configuration Defaults

```
pgo create cluster testcluster --storage-config=bigdisk
```

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
```

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
```

That example will create a backup and use the **offsitestorage** storage configuration for persisting the backup.

Disaster Recovery Using Storage Configurations

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.

23.6. PostgreSQL Operator Container Configuration

To enable **debug** level messages from the operator pod, set the `CRUNCHY_DEBUG` environment variable to **true** within its deployment file `deployment.json`.

Operator Templates

The database and cluster Kubernetes objects that get created by the operator are based on JSON templates that are added into the operator deployment by means of a mounted volume.

The templates are located in the `$COROOT/conf/postgres-operator` directory and are added into a config map which is mounted by the operator deployment.

24. Bash Completion

There is a bash completion file that is included for users to try located in the repository at `examples/pgo-bash-completion`. To use it -

```
cp $COROOT/examples/pgo-bash-completion /etc/bash_completion.d/pgo
su - $USER
```

25. REST API

Because the **apiserver** implements a REST API, it is possible to integrate with it using your own application code. To demonstrate this, the following **curl** commands show the API usage -

Note: Some setups may require the user to add *?version=x.x* to the end of the commands.

pgo version

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

pgo show policy <name>

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

pgo delete policy <name>

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

pgo show pvc <name>

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

pgo apply policy <name>

```
curl -v -X POST -u readonlyuser:testpass -H "Content-Type: application/json" --insecure l
```

pgo show ingest <name>

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

pgo label

```
curl -v -X POST -u readonlyuser:testpass -H "Content-Type: application/json" --insecure l
```

pgo load

```
curl -v -X POST -u readonlyuser:testpass -H "Content-Type: application/json" --insecure l
```

pgo user

```
curl -v -X POST -u readonlyuser:testpass -H "Content-Type: application/json" --insecure l
```

pgo users <name>

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

pgo delete user <name>

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

pgo show upgrade <name>

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

pgo delete upgrade <name>

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

pgo show cluster <name>

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

pgo delete cluster

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

pgo test <name>

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

pgo scale <name>

```
curl -v -X GET -u readonlyuser:testpass -H "Content-Type: application/json" --insecure h
```

26. Deploying pgPool

One option with pgo is enabling the creation of a pgpool deployment in addition to the PostgreSQL cluster. Running pgpool is a logical inclusion when the Kubernetes cluster includes both a primary database in addition to some number of replicas deployed. The current pgpool configuration deployed by the operator only works when both a primary and a replica are running.

When a user creates the cluster a command flag can be passed as follows to enable the creation of the pgpool deployment.

```
pgo create cluster cluster1 --pgpool
pgo scale cluster1
```

This will cause the operator to create a Deployment that includes the **crunchy-pgpool** container along with a replica. That container will create a configuration that will perform SQL routing to your cluster services, both for the primary and replica services.

Pgpool examines the SQL it receives and routes the SQL statement to either the primary or replica based on the SQL action. Specifically, it will send writes and updates to only the **primary** service. It will send read-only statements to the **replica** service.

When the operator deploys the pgpool container, it creates a secret (e.g. mycluster-pgpool-secret) that contains pgpool configuration files. It fills out templated versions of these configuration files specifically for this PostgreSQL cluster.

Part of the pgpool deployment also includes creating a `pool_passwd` file that will allow the **testuser** credential to authenticate to pgpool. Adding additional users to the pgpool configuration currently requires human intervention specifically creating a new pgpool secret and bouncing the pgpool pod to pick up the updated secret. Future operator releases will attempt to provide **pgo** commands to let you automate the addition or removal of a pgpool user.

Currently to update a pgpool user within the `pool_passwd` configuration file, it is necessary to copy the existing files from the secret to your local system, update the credentials in `pool_passwd` with the new user credentials, recreate the pgpool secret, and finally restart the pgpool pod to pick up the updated configuration files.

As an example -

```
kubectl cp demo/wed10-pgpool-6cc6f6598d-wcnmf:/pgconf/ /tmp/foo
```

That command gets a running set of secret pgpool configuration files and places them locally on your system for you to edit.

pgpool requires a specially formatted password credential to be placed into `pool_passwd`. There is a golang program included in `$COROOT/golang-examples/gen-pgpool-pass.go` that, when run, will generate the value to use within the **pgpool_passwd** configuration file.

```
go run $COROOT/golang-examples/gen-pgpool-pass.go
Enter Username: testuser
Enter Password:
Password typed: e99Mjt1dLz
hash of password is [md59c4017667828b33762665dc4558fbd76]
```

The value **md59c4017667828b33762665dc4558fbd76** is what you will use in the `pool_passwd` file.

Then, create the new secrets file based on those updated files -

```
$COROOT/bin/create-pgpool-secrets.sh
```

Lastly for pgpool to pick up the new secret file, delete the existing deployment pod -

```
kubectl get deployment wed-pgpool
kubectl delete pod wed10-pgpool-6cc6f6598d-wcnmf
```

The pgpool deployment will spin up another pgpool which will pick up the updated secret file.

27. Storage Configuration

Most users after they try out the operator will want to create a more customized installation and deployment of the operator using specific storage types.

The operator will work with HostPath, NFS, Dynamic, and GKE Storage.

27.1. NFS

To configure the operator to use NFS for storage, a sample **pgo.yaml.nfs** file is provided. Overlay the default `pgo.yaml` file with that file -

```
cp $COROOT/examples/pgo.yaml.nfs $COROOT/conf/postgres-operator/pgo.yaml
```

Then, in your `.bashrc` file, set the variable `CO_NFS_IP` to the IP address of your NFS server:

```
export CO_NFS_IP=192.168.2.14
```

Edit the **pgo.yaml** file to specify the NFS GID that is set for the NFS volume mount you will be using. The default value assumed is **nfsnobody** as the GID (65534). Update the value to meet your NFS security settings.

Finally, run the `$COROOT/pv/create-pv-nfs.sh` script to create persistent volumes based on your NFS settings.

27.2. Dynamic

To configure the operator to use Dynamic Storage classes for storage, a sample **pgo.yaml.storageclass** file is provided. Overlay the default **pgo.yaml** file with that file -

```
cp $COROOT/examples/pgo.yaml.storageclass $COROOT/conf/postgres-operator/pgo.yaml
```

Edit the **pgo.yaml** file to specify the storage class you will be using, the default value assumed is **standard** which is the name used by default within a GKE Kube cluster deployment. Update the value to match your storage classes.

Notice that the **FsGroup** setting is required for most block storage and is set to the value of **26** since the PostgreSQL container runs as UID **26**.

27.3. GKE

Some notes for setting up GKE for the Operator deployment.

Install Kubectl

On your host you will be working from, install the kubectl command -

<https://kubernetes.io/docs/tasks/tools/install-kubectl/>

GCP

- Select your project
- Create a Kube cluster in that project

By default a storage class called **standard** is created.

Install GCloud

To access the Kubernetes cluster, install the gcloud utility -

```
https://cloud.google.com/sdk/downloads
cd google-cloud-sdk
./install.sh
```

Configure Kubectl for Cluster Access

```
gcloud auth login
```

```
gcloud container clusters get-credentials jeff-quickstart --zone us-central1-a --project
```

```
kubectl get storageclass
```

title: "Deployment" date: 2018-04-26T15:26:40-07:00 draft: false weight: 50

Latest Release: 3.4.0 2018-12-04

This document details verifying the installation of the PostgreSQL Operator is successful, in addition to detailing some different storage configurations that can be made.

28. Verify Operator Status

To verify that the operator is deployed and running, run the following:

```
kubectl get pod --selector=name=postgres-operator
```

You should see output similar to this:

NAME	READY	STATUS	RESTARTS	AGE
postgres-operator-56598999cd-tbg4w	2/2	Running	0	1m

There are 2 containers in the operator pod, both should be **ready** as above.

When you first run the operator, it will create the required CustomResourceDefinitions. You can view these as follows -

```
kubectl get crd
```

The operator creates the following Custom Resource Definitions over time as the associated commands are triggered.

```
kubectl get crd
```

NAME	AGE
pgbackups.cr.client-go.k8s.io	2d
pgclusters.cr.client-go.k8s.io	2d
pgingests.cr.client-go.k8s.io	2d
pgpolicies.cr.client-go.k8s.io	2d
pgreplicas.cr.client-go.k8s.io	2d
pgtasks.cr.client-go.k8s.io	2d
pgupgrades.cr.client-go.k8s.io	2d

At this point, the server side of the operator is deployed and ready.

29. Configure pgo Client

The pgo command line client requires TLS for securing the connection to the operator's REST API. This configuration is performed as follows -

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

The pgo client uses Basic Authentication to authenticate to the operator REST API. For authentication, add the following `.pgouser` file to your `$HOME` -

```
echo "username:password" > $HOME/.pgouser
```

Roles are defined in a file called **pgorole**. This file defines each role and the permissions for that role. By default, two roles are defined as samples -

```
pgoadmin
pgoreader
```

This file, moved to your \$HOME folder, is optional. These default settings can be adjusted to meet local security requirements.

The format of this file is as follows -

```
rolename: permissionA, permissionB
```

These are defined in the following file -

```
$COROOT/conf/postgres-operator/pgrole
```

The complete set of permissions is documented in the Configuration [/installation/configuration/] document.

The **pgo** client needs the URL to connect to the operator.

Depending on your Kubernetes environment this can be done the following ways.

29.1. Running Kubernetes Locally

If your local host is not set up to resolve Kubernetes Service DNS names, you can specify the operator IP address as follows -

```
kubectl get service postgres-operator
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
postgres-operator   NodePort    10.109.184.8     <none>           8443:30894/TCP   5m

export CO_APISERVER_URL=https://10.109.184.8:8443
pgo version
```

Alternatively, an alias was set up in the .bashrc file earlier on, as follows:

```
alias setip='export CO_APISERVER_URL=https://`kubectl get service postgres-operator -o=j
```

This alias (setip) will set the CO_APISERVER_URL IP address for you.

Running Kubernetes Remotely

Port forwarding

Set up a port-forward tunnel from your host to the Kube remote host, specifying the operator pod -

```
kubectl get pod --selector=name=postgres-operator
NAME                                READY    STATUS    RESTARTS    AGE
postgres-operator-56598999cd-tbg4w  2/2     Running   0           8m

kubectl port-forward postgres-operator-56598999cd-tbg4w 8443:8443
```

In another terminal -

```
export CO_APISERVER_URL=https://127.0.0.1:8443
pgo version
```

Using an ingress

Ingresses allows you to access Kubernetes services through a controller.

First you will need to ensure a NGINX Ingress Controller is available in your Kubernetes cluster.

If you are using Minikube, you can easily deploy one using

```
minikube addons enable ingress
```

If not, please refer to the Nginx Ingress Controller's official documentation [<https://kubernetes.github.io/ingress-nginx/deploy/#bare-metal>] for its installation.

Once your controller is running, just deploy the ingress using

```
kubectl create -f $COROOT/deploy/ingress.yml
```

Due to the annotations used, please note this ingress is currently usable only with Nginx Ingress Controller.

Now you can use the address IP of the host where the nginx-ingress-controller pod is to connect to the pgo apiserver. The port will be 443 (and not 8443).

To retrieve the address ip:

```
kubectl get ingress postgres-operator -o jsonpath="{.status.loadBalancer.ingress[0].ip}"  
export CO_APISERVER_URL=https://`kubectl get ingress postgres-operator -o jsonpath="{.st
```

If you are using minikube, the address IP displayed is incorrect, just use:

```
minikube ip  
  
export CO_APISERVER_URL=https://`minikube ip`
```

30. Verify pgo Client

At this point you should be able to connect to the operator as follows -

```
pgo version  
pgo client version 3.4.0  
apiserver version 3.4.0
```

Operator commands are documented on the Getting Started [/getting-started/] page.

31. Next Steps

There are many ways to configure the operator further. Some sample configurations are documented on the Configuration [/installation/configuration/] page.

You may also want to find out more information on how the operator is designed to work and deploy. This information can be found in the How It Works [/how-it-works/] page.

Information can be found on the full scope of commands on the Getting Started [/getting-started/] page.

title: "Upgrading the Operator" date: 2018-04-24T18:27:30-07:00 draft: false weight: 60

Latest Release: 3.4.0 2018-12-04

- The `conf/apiserver/` configuration files were moved into the `conf/postgres-operator` directory to consolidate all config files into a single location. You will need to perform this step manually start with version 3.4.0 if you are running an existing Operator version prior to 3.4.0. The Helm chart is also updated to reflect this change. Starting with 3.4.0 there is a secret, **pgo-auth-secret** that holds authentication and authorization files used by the operator to authenticate REST clients. Also, the configuration files are stored in a configmap named **pgo-config**. Existing users will need to update their `deploy.sh` script and `deployment.yaml` files to pick up the the new naming conventions.
- new configuration settings were added into `pgo.yaml` to support resource configuration settings for the various **helper** containers. The new settings include `DefaultLoadResources`, `DefaultLspvcResources`, `DefaultRmdataResources`, `DefaultBackupResources`. You will need to add these manually into your existing `pgo.yaml` file if you want to make use of this feature.
- the `pgcluster` CRD was changed to remove the password fields, instead secret names are stored in the CRD to avoid having to have passwords in the CRD, the password fields are totally removed starting in this release. No changes are required for existing CRD resources, new CRDs that are created will not have the password fields.
- starting in 3.4, a new operator upgrade process is developed that eventually will handle various forms of automated upgrades depending on user settings and changes to the `postgres-operator` in between versions. When starting a new Operator it will scan the `pgcluster` and `pgreplica` instances and update the `pgo-version` to match the current operator version, it will also create a user label on the `pgreplica/pgcluster` to indicate the upgrade date. More advanced upgrade features are planned to be developed.
- in 3.4, the `pgo.yaml` `LogStatement` and `LogMinDurationStatement` settings are present, if not set, defaults are supplied for both. These settings let you define more precisely the degree of Postgres logging for any Postgres clusters created by the Operator. The `LogStatement` default is *none* and the `LogMinDurationStatement` defaults to 60000 (milliseconds). These settings greatly reduce the log file sizes and only will log statements that are longer running than 60000 milliseconds. If you want to see all statements logged, set `LogStatement` to *all*.
- in 3.4, the `pgbackup` CRD includes a new field called `BackupOpts`, used to hold the `pgbasebackup` command options which can now be passed in on the CLI backup command, no changes to existing `pgbackup` CRD resources is required.
- operator 3.3 introduced the alpha version of `pgbackrest` integration. Now in 3.4, the `pgbackrest` integration was changed so that users no longer have to specify a custom configuration file using **--custom-config**.
- the `pgo-backrest backup-job` is secured with a new service account named `pgo-backrest`, that SA is now included in the `rbac.yaml` file and is to be created by an a cluster admin user
- the permission `SHOW_WORKFLOW_PERM` was created and added to the default `pgrole` example, this permission lets users view workflow status, workflows are stored as `pgtask` CRDs

With 3.4, there is a global configmap created as part of the deployment process which will serve this same purpose, that is to indicate to the Postgres container that it must allocate the `pgbackrest` directories within the mounted `/backrestrepo` volume mount. This means however, that if you specify a global configuration file or specify your own custom configuration that you have to include the

`pgbackrest.conf` file and key within that configmap. The sample global custom configuration map, `pgo-custom-pg-config`, now includes `pgbackrest.conf` within it.

This new integration only works with `pgbackrest` v2.6 which is included into `crunchy-postgres` 2.2.0. This means that to use `pgbackrest`, you must run `crunchy-postgres` 2.2.0 or greater which will require users to upgrade the `pgo.yaml` to use the 2.2.0 `CCPImageTag`. `pgbackrest` commands will NOT work with clusters using older Postgres images.

The following changes are mandatory when upgrading to 3.3 from previous operator versions:

- The `MatchLabels` attribute was added to the `pgo.yaml` file as an optional storage configuration setting. You do not have to specify this setting, however, the operator-conf `ConfigMap` now has to include the `pvc-matchlabels.json` template file as required by this new feature. If you upgrade to 3.2, you will need to rebuild your **operator-conf** `ConfigMap` to include `pvc-matchlabels.json` and redeploy the Operator using the new `ConfigMap`.
- The `CCP_IMAGE_PREFIX`, `CO_IMAGE_PREFIX`, and `CO_IMAGE_TAG` environment variables are now pulled from the `pgo.yaml` configuration file that is mounted by both the `apiserver` and `operator` containers. To clean up an existing deployment, remove these environment variable definitions from your `deployment.yaml` file or Helm chart equivalent.
- The `ExternalIP` field was added to the `apiservermsgs.ShowClusterService` struct. This field is now passed back to `apiserver` clients in the REST API when viewing cluster details. For custom clients you might have written, you will see this new field in the REST message.
- Clusters that were created prior to 3.1 will need a new label to be applied. For primary deployments, apply the label `primary=true`. For example, `kubectl label deploy mycluster primary=true`. For replica deployments, specify `primary=false`. For example, `kubectl label deploy mycluster-xxxx primary=false`.
- The `collect.json` template now specifies a `pgmonitor` credential that must match the `PGMONITOR_PASSWORD` environment variable which was added into the `cluster-deployment-1.json` template. These changes were required to support `crunchy-collect` (2.1.0) changes that were introduced. Users should upgrade to `crunchy-collect:centos7-10.5-2.1.0` to use this feature. If you do not want to upgrade to this new metrics collector, you will need to retain and reuse the prior version of `collect.json` used by the Operator and make sure you deploy that version.

Clusters that were created prior to 3.1 will need a new label to be applied. For primary deployments, apply the label `primary=true`. For example, `kubectl label deploy mycluster primary=true`. For replica deployments, specify `primary=false`. For example, `kubectl label deploy mycluster-xxxx primary=false`.

For a full list of additions and revisions that occurred in the PostgreSQL Operator v2.5 release, please view the related release page here [<https://github.com/CrunchyData/postgres-operator/releases/tag/2.5>].

32. Required Updates

This section notes some required steps that will need to be taken in the process of upgrading from v2.4 to v2.5.

32.1. Configuration File

It will be necessary to update your existing `pgo.yaml` configuration file where the Storage Configuration sections are concerned. The updated file for v2.5 can be found here [<https://github.com/CrunchyData/postgres-operator/blob/2.5/conf/apiserver/pgo.yaml>]. The file contained within the local installation of the Operator is located by default in the following location -

```
$COROOT/conf/apiserver/pgo.yaml
```

32.2. Secrets

2.5 changed the names of the database credentials that are created by default in order to be consistent with the way new database credentials are named.

It will be necessary to run the following script to update your existing clusters. This script will essentially copy the existing secrets values and create new secrets with those same values but named to the new standard. Run the script by passing in the name of an existing cluster as a parameter.

```
$COROOT/bin/upgrade-secret.sh
```

For a full list of additions and revisions that occurred in the PostgreSQL Operator v2.5 release, please view the related release page here [<https://github.com/CrunchyData/postgres-operator/releases/tag/3.3.0>].

33. Required Updates

This section notes some required steps that will need to be taken in the process of upgrading from v2.5 to v2.6.

33.1. Configuration File

One update in v2.6 changed the `pgo.yaml` file through removing the Debug flag. The `Pgo.Debug` variable can now be removed from the `pgo.yaml` file as a result. The debug flag is now called `CRUNCHY_DEBUG` and is set in the `deployment.json` file as a default environment variable.

33.2. Container Resources

Release 2.6 added the concept of container resource configurations to the `pgo.yaml` file. In order to specify the optional container resource configurations, add a section as follows to your `pgo.yaml` file -

```
DefaultContainerResource: small
ContainerResources:
  small:
    RequestsMemory: 2Gi
    RequestsCPU: 0.5
    LimitsMemory: 2Gi
    LimitsCPU: 1.0
  large:
    RequestsMemory: 8Gi
    RequestsCPU: 2.0
```

```
LimitsMemory: 12Gi
LimitsCPU: 4.0
```

If these settings are set incorrectly or if the Kubernetes cluster cannot meet the defined memory and CPU requirements, deployments will go into a **pending** state.

33.3. Kube RBAC

Release 2.6 added a `rbac.yaml` file to capture the Kube RBAC rules. These RBAC rules allow the **apiserver** and **postgres-operator** containers access to the Kubernetes resources required for the operator to work. As part of the deployment process, it is necessary to execute the `rbac.yaml` file to set the roles and bindings required by the operator. Adjust this file to suit local security requirements.

33.4. Application RBAC

Release 2.6 added an RBAC capability to secure the **pgo** application. The **pgouser** now has a role appended at the end of each user definition as follows -

```
username:password:pgoadmin
testuser:testpass:pgoadmin
readonlyuser:testpass:pgoreader
```

These are defined in the following file -

```
$COROOT/conf/apiserver/pgouser
```

To match the behavior of the pre 2.6 releases, the **pgadmin** role is set on the previous user definitions, but a **readonlyuser** is now defined to test other role definitions. The roles are defined in a new file called **pgorole**. This file defines each role and the permissions for that role. By default, two roles are defined as samples -

```
pgoadmin
pgoreader
```

Adjust these default settings to meet local security requirements.

The format of this file is as follows -

```
rolename: permissionA, permissionB
```

These are defined in the following file -

```
$COROOT/conf/apiserver/pgorole
```

The complete set of permissions is documented in the Configuration [\[/installation/configuration/\]](#) document.

33.5. User Creation

Release 2.6 replaced the `pgo user --add` command with the `pgo create user` command to improve consistency across command usage. Any scripts written using the older style of command require an update to use the new command syntax.

33.6. Replica CRD

There is a new Kubernetes Custom Resource Definition that serves the purpose of holding replica information, called **pgreplicas**. This CRD is populated with the `pgo scale` command and is used to hold per-replica specific information such as the resource and storage configurations requested at run time.

title: "Getting Started" date: 2018-04-24T18:26:43-07:00 draft: false weight: 20

Latest Release: 3.4.0 2018-12-04

34. First Steps

Prior to using **pgo**, users will need to specify the **postgres-operator** URL as follows:

```
kubect1 get service postgres-operator
NAME                CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
postgres-operator   10.104.47.110    <none>           8443/TCP         7m
export CO_APISERVER_URL=https://10.104.47.110:8443
pgo version
```

35. Cluster Names

Many of the `pgo` commands take in a cluster name, in some cases the special name of **all** is accepted which will cause the command to be applied to all PostgreSQL clusters. For example:

```
pgo df all
```

36. General

36.1. Operator Version

This command makes it possible to see what version of the `pgo` client and `postgres-operator` you are running.

Syntax

```
$ pgo version
```

36.2. Operator Status

You can use the **pgo status** command to see overall `pgo` status. Selective metrics are displayed to provide some insights to the `pgo` user and administrator as to what is running currently in this namespace related to `pgo`.

Syntax

```
$ pgo status [FLAGS]
```

Flags

Name	Short-hand	Input	Usage
<code>--output=json</code>	<code>-o</code>	String json	The output format. Currently, json is the only supported value.

36.3. Operator Configuration

The `pgo show config` command displays the running operator configuration parameters that dictate the setup and user defined configuration of the operator. This command can be useful for sharing your configuration or verifying the setup is as expected.

Syntax

```
$ pgo show config
```

36.4. Disk Capacity

The `pgo df` command will let you see the disk capacity of a cluster's PVC versus that of the PostgreSQL data that has been written to disk. If the capacity is less than 50%, then the output is printed in red in order to alert the user. The listing is broken out by the cluster's Pods.

Syntax

```
$ pgo df NAME [FLAGS]
```

Flags

Name	Short-hand	Input	Usage
<code>--selector</code>	<code>-s</code>	String	The selector to use for cluster filtering.

Examples

Cluster Selectors

The `pgo df` command can either be run against a single cluster or against all clusters matching a selector:

```
pgo df mycluster
pgo df --selector=project=xrayapp
```

37. Cluster Basics

37.1. Create Cluster

The **create cluster** command will automatically provision a PostgreSQL cluster within Kubernetes or OpenShift using a Deployment.

Syntax

\$ pgo create cluster NAME [FLAGS]

Flags

Name	Short	Input	Usage
--archive	N/ A	N/ A	Enables archive logging for the database cluster.
--autofail	N/ A	N/ A	If set, will cause autofailover to be enabled on this cluster.
--backup-pvc	N/ A	String	The backup archive PVC to restore from.
--backup-path	N/ A	String	The backup archive path to restore from.
--ccp-image-tag	N/ A	String	The CCPIImageTag to use for cluster creation. If specified, overrides the pgo.yaml setting.
--custom-config	N/ A	String	The name of a configMap that holds custom PostgreSQL configuration files used to override defaults.
--labels	N/ A	String	The labels to apply to this cluster.
--metrics	N/ A	N/ A	Adds the crunchy-collect container to the database pod.
--node-label	N/ A	String	The node label (key) to use in placing the primary database. If not set, any node is used.
--password	N/ A	String	The password to use for initial database users.
--service-type	N/ A	String	The Service type to use for the PostgreSQL cluster. If not set, the pgo.yaml default will be used.
--pgbackrest	N/ A	N/ A	Enables a pgBackRest volume for the database pod.
--pgbackrest-restore-from	N/ A	N/ A	Only applies when creating a cluster from a pgbackrest restored PVC. This is the name of the cluster from which the restored PVC was created from and which the new cluster credentials will be based. This setting is required in the scenario.
--pgbadger	N/ A	N/ A	Adds the crunchy-pgbadger container to the database pod.
--pgpool	N/ A	N/ A	Adds the crunchy-pgpool container to the database pod.
--pgpool-secret	N/ A	String	The name of a pgpool secret to use for the pgpool configuration.
--policies	N/ A	String	The policies to apply when creating a cluster, comma separated.

Name	Shortname	Input	Usage
--replica-count	N/A	Int	The number of replicas to create as part of this cluster. After a cluster is created, you can also add replicas using the scale command.
--replica-storage-config	N/A	String	The name of a Storage config in pgo.yaml to use for the cluster replica storage.
--resources-config	N/A	String	The name of a container resource configuration in pgo.yaml that holds CPU and memory requests and limits.
--secret-from	N/A	String	The cluster name to use when restoring secrets.
--series	N/A	Int	The number of clusters to create in a series (default 1).
--storage-config	N/A	String	The name of a Storage config in pgo.yaml to use for the cluster storage.

Examples

Simple Creation

Create a single cluster:

```
pgo create cluster mycluster
```

Create a single cluster with a single replica:

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

Complex Creation

Create a series of clusters, specifying it as the xray project, with the xrayapp and rlspolicy policies added:

```
pgo create cluster mycluster --series=3 --labels=project=xray --policies=xrayapp,rlspoli
```

Image Version

New clusters typically pick up the container image version to use based on the pgo configuration file's CcpImageTag setting. You can override this value using the --ccp-image-tag command line flag:

```
pgo create cluster mycluster --ccp-image-tag=centos7-9.6.5-1.6.0
```

Metrics

Add the crunchy-collect [<https://crunchydata.github.io/crunchy-containers/stable/container-specifications/crunchy-collect/>] container from the Crunchy Container Suite to the database cluster pod and enable metrics collection on the database:

```
pgo create cluster mycluster --metrics
```

You can connect these containers to a metrics pipeline using Grafana [<https://grafana.com>] and Prometheus [<https://prometheus.io>] by following the example found in the Crunchy Container Suite

documentation [https://crunchydata.github.io/crunchy-containers/stable/getting-started/kubernetes-and-openshift/#_metrics_collection].

pgBadger

Add a pgBadger [<https://github.com/dalibo/pgbadger>] sidecar into the Postgres pod:

```
pgo create cluster mycluster --pgbadger
```

This command flag adds the crunchy-pgbadger [<https://crunchydata.github.io/crunchy-containers/stable/container-specifications/crunchy-pgbadger/>] container into the database pod. pgBadger reports can then be accessed through port 10000 at `/api/badgergenerate`.

pgPool II

By appending the `--pgpool` command line flag, you can add pgPool II [http://www.pgpool.net/mediawiki/index.php/Main_Page] to the database cluster. The container used for this functionality is the crunchy-pgpool [<https://crunchydata.github.io/crunchy-containers/stable/container-specifications/crunchy-pgpool/>] container image from the Crunchy Container Suite.

```
pgo create cluster mycluster --pgpool
```

Auto Failover

To enable auto failover on this cluster, use the following flag:

```
pgo create cluster mycluster --autofail
```

This flag, when set on the cluster, informs the operator to look or watch for NotReady events on this cluster. When those occur, it will create a failover state machine which acts as a timer for the cluster. If the timer expires, then a failover is triggered on the cluster turning one of the cluster replica pods into the replacement primary pod. See the How It Works [https://crunchydata.github.io/postgres-operator/stable/how-it-works/#_auto_failover] documentation for more details on auto failover.

pgBackRest

pgbackrest beta integration was implemented in version 3.4.0 of the Operator. NOTE: pgbackrest integration is still subject to change in upcoming releases.

The backrestrepo PVC, used by pgBackRest, has to be created on a RWX file system type in this release. pgBackRest is a more advanced backup and restore capability exposed by the Operator.

The pgBackRest support is enabled in a PG cluster by a user specifying the `--pgbackrest` command flag. To enable this feature for all PG clusters when created, you can specify a `pgbackrest` setting within the `pgo.yaml` configuration.

Create a PG cluster that enables pgBackRest specifically for that cluster:

```
pgo create cluster mycluster --pgbackrest
```

Setting this value will cause the Operator to create a PVC specifically dedicated for holding pgBackRest backups.

Create a pgBackRest backup:

```
pgo backup mycluster --backup-type=pgbackrest
```

You can also pass in pgbackrest backup command options:

```
pgo backup mycluster --backup-type=pgbackrest --pgbackrest-opts="--type=incr"
```

Note, you can not specify **--storage-config** flag when specifying a pgbackrest backup.

List pgBackRest information:

```
pgo show backup mycluster --backup-type=pgbackrest
```

Restore from an existing cluster into a newly created PVC:

```
pgo restore mycluster --to-pvc=restored
pgo create cluster restored --pgbackrest-restore-from=mycluster --pgbackrest
```

The pgBackRest backrestrepo PVCs are created using the pgo.yaml BackupStorage setting. Typically, this will be a RWX file system but if the file system is RWO the PVCs will be created without having write access and a backup and restore will fail. The RWX file system setup will allow you to restore from this PVC without having to shutdown the currently attached PostgreSQL cluster. Note that a cluster based off of the restored PVC has to attach the same pgbackrest repo used by the original cluster the restore was based off of.

37.2. Delete Cluster

The `delete cluster` command will by default delete all associated components of the selected cluster, but will not delete the data or the backups unless specified.

Syntax

```
$ pgo delete cluster NAME|all [FLAGS]
```

Flags

Name	Shorthand	Input	Usage
--delete-backups	-b	N/A	Causes the backups for this cluster to be removed permanently. This only is applicable with pgbasebackup backup volumes and does not remove pgbackrest repo volumes.
--delete-configs	-b	N/A	Causes the configuration maps for this cluster to be removed permanently.
--delete-data	-d	N/A	Causes the data for this cluster to be removed permanently.
--no-prompt	-n	N/A	No command line confirmation.
--selector	-s	String	The selector to use for cluster filtering.

Examples

Simple Deletion

Delete a single cluster:

```
pgo delete cluster mycluster
```

Note that this command will not remove the PVC associated with this cluster.

Complex Deletion

Selectors also apply to the delete command as follows:

```
pgo delete cluster --selector=project=xray
```

This command will cause any cluster matching the selector to be removed.

Delete Components, Data, & Backups

You can remove a cluster, it's data files, and all backups by running:

```
pgo delete cluster restoredb --delete-data --delete-backups --delete-configs
```

When you specify a destructive delete like above, you will be prompted to make sure this is what you want to do. If you don't want to be prompted you can enter the `--no-prompt` command line flag.

37.3. Show Cluster

The `show cluster` command allows you to view all the associated created components of a specific cluster or selection of clusters.

By default, you will be able to view the status of the created pod, the PVC, Deployment, Service, and Labels associated with the cluster, and any and all specified options (such as whether `crunchy_collect` is enabled).

Syntax

```
$ pgo show cluster NAME|all [FLAGS]
```

Flags

Name	Shorthand	Input	Usage
<code>--output=json</code>	<code>-o</code>	String json	The output format. Currently, json is the only supported value.
<code>--selector</code>	<code>-s</code>	String	The selector to use for cluster filtering.
<code>--ccp-image-tag</code>	<code>N/A</code>	String	Filter the results based on the PostgreSQL version of the cluster.

Examples

Simple Display

Show a single cluster:

```
pgo show cluster mycluster
```

Show All

Show all clusters available:

```
pgo show cluster all
```

Show Secrets

User credentials are generated through Kubernetes Secrets automatically for the **testuser**, **primaryuser** and **postgres** accounts. The generated passwords can be viewed by running the `pgo show user` command. More details are available on user management below.

```
pgo show user mycluster
```

Viewing Users With Passwords Set to Expire

To see user passwords that have expired past a certain number of days in the **mycluster** cluster:

```
pgo show user --expired=7 --selector=name=mycluster
```

Name	Short-hand	Input	Usage
--expired	N/A	String	

PostgreSQL Version

Filter the results based on the PostgreSQL version of the cluster with the `--ccp-image-tag` flag:

```
pgo show cluster all --ccp-image-tag=centos7-10.5-2.1.0
```

37.4. Test Connection

This command will test each service defined for the cluster using the `postgres`, `primary`, and `normal` user accounts defined for the cluster. The cluster credentials are accessed and used to test the database connections. The equivalent **psql** command is printed out as connections are tried, along with the connection status.

Syntax

```
$ pgo test NAME|all [FLAGS]
```

Flags

Name	Short-hand	Input	Usage
--output=json	-o	String json	The output format. Currently, json is the only supported value.
--selector	-s	String	The selector to use for cluster filtering.

Examples

Simple Test

Test the database connections to a cluster:

```
pgo test mycluster
```

Complex Test

Like other commands, you can use the selector to test a series of clusters or to test all available clusters:

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

38. Administration

38.1. Reload

The **reload** command will perform a reload on the specified PostgreSQL cluster.

Syntax

```
$ pgo reload NAME [FLAGS]
```

Flags

Name	Short	Input	Usage
--no-prompt	-n	N/ A	No command line confirmation.
--selector	-s	String	The selector to use for cluster filtering.

Examples

Simple Reload

Reload a single cluster:

```
pgo reload mycluster
```

38.2. Backups

The **backup** command will utilize the crunchy-backup [<https://crunchydata.github.io/crunchy-containers/stable/container-specifications/crunchy-backup/>] container to execute a full backup against another database container using the standard pg_basebackup utility that is included with PostgreSQL.

When you request a backup, **pgo** will prompt you if you want to proceed because this action will delete any existing backup job for this cluster that might exist. The backup files will still be left intact but the actual Kubernetes Job will be removed prior to creating a new Job with the same name.

Syntax

```
$ pgo backup NAME [FLAGS]
```

Flags

Name	Shortname	Input	Usage
--selector	-s	String	The selector to use for cluster filtering.
--pvc-name	N/A	String	The PVC name to use for the backup instead of the default.
--backup-type	N/A	String	The backup type to perform. Default is pgbasebackup, and both pgbasebackup and pgbackrest are valid backup types.
--backup-opts	N/A	String	The options to pass to pgbasebackup or pgbackrest, use appropriate command options depending on which type of backup you are performing.
--storage-config	N/A	String	The name of a Storage config in pgo.yaml to use for the cluster storage.

Examples

Simple Backup

You can start a backup job for a cluster as follows:

```
pgo backup mycluster
```

Show Backup

View the backup and backup status:

```
pgo show backup mycluster
```

Backup PVC Management

Note

pgo show pvc can run into file permission issues if you are trying to view a PVC that is on a RWO (read write once) file system (e.g. cloud storage, ceph, storageos, etc.). If another pod has the PVC mounted you will get timeout errors from the *pgo lspvc* command in the current 3.4.0 release.

View the PVC folder and the backups contained therein:

```
pgo show pvc mycluster-backup
pgo show pvc mycluster-backup --pvc-root=mycluster-backups
```

The output from this command is important in that it can let you copy/paste a backup snapshot path and use it for restoring a database or essentially cloning a database with an existing backup archive.

For example, to restore a database from a backup archive:

```
pgo create cluster restoredb --backup-path=mycluster-backups/2017-03-27-13-56-49 --backup
```

This will create a new database called **restoredb** based on the backup found in **mycluster-backups/2017-03-27-13-56-49** and the secrets of the **mycluster** cluster.

Override PVC

You can override the PVC used by the backup job with the following:

```
pgo backup mycluster --pvc-name=myremotepvc
```

This might be useful for special backup cases such as creating a backup on a disaster recovery PVC.

Delete Backup

To delete a backup enter the following:

```
pgo delete backup mycluster
```

When run, this command removes the PVC used for the backups, and runs the **rmdata** Job to physically perform data removal of that PVC's contents. It also removes the pgbackup CRD for this cluster that holds the last pg_basebackup results.

38.3. Scheduling

The `schedule` command will generate schedule configuration maps that are utilized by the crunchy-scheduler [<https://crunchydata.github.io/crunchy-containers/stable/container-specifications/crunchy-scheduler/>] container. This allows users to create automated, scheduled backups for their PostgreSQL clusters.

Currently only two types of backups are supported with the `schedule` command: `* pgBackRest *` and `pgBaseBackup`.

Crunchy Scheduler is a cron-like microservice that periodically queries Kubernetes for configuration maps with the label `crunchy-scheduler=true` in a specific namespace. After finding the schedule configs, the scheduler service will either exec into the container (`pgBackRest`) or create `pgBaseBackup` jobs for the configured schedule.

Note

in operator version 3.4.0, you are **REQUIRED**, a single time, to run a `pgbackrest` backup **PRIOR** to creating a `pgbackrest` schedule. This will not be a requirement in the 3.5.0 version of the Operator.

Syntax

```
$ pgo create schedule NAME [FLAGS]
```

Flags

Name	Shortname	Input	Output	Usage
--ccp-image-tag	-n	N/A		Image version to use for pgBaseBackup backup jobs. Defaults to what PGO is configured to use.
--no-prompt	-n	N/A		No command line confirmation.
--pgbackrest-backup-type		N/A	String	The type of pgBackRest backup to perform. There is no default and the following are valid: full, diff, incr
--pvc-name		N/A	String	The PVC name to use for the backup. Only used for pgBaseBackup schedule types and must be created prior to using.
--schedule		N/A	String	The schedule assigned to the cron task.
--schedule-type		N/A	String	The schedule type to perform. There is no default and both pgbasebackup and pgbackrest are valid schedule types.
--selector	-s		String	The selector to use for cluster filtering.

Examples

Creating pgBackRest Schedules

Create a pgBackRest `full` backup on Sunday at 1 a.m:

```
pgo create schedule --schedule="0 1 * * 7" --schedule-type=pgbackrest --pgbackrest-backup
```

Create a pgBackRest `diff` backup on Monday-Saturday at 1 a.m:

```
pgo create schedule --schedule="0 1 * * 1-6" --schedule-type=pgbackrest --pgbackrest-bac
```

Creating pgBaseBackup Schedules

Create a pgBaseBackup backup every day at 1 a.m:

```
pgo create schedule --schedule="0 1 * * *" --schedule-type=pgbasebackup --pvc-name=myclu
```

Creating Schedules Using Selectors

Using the `selector` flag, we can create schedules for all clusters that match a label:

```
pgo create schedule --schedule="0 1 * * *" --schedule-type=pgbasebackup --pvc-name=myclu
```

Show Schedules

View the schedules for cluster named `mycluster`:

```
pgo show schedule mycluster
```

View the schedules for all clusters with the label `env=test`:

```
pgo show schedule --selector=env=test
```

or for a particular cluster:

```
pgo show schedule --selector=pg-cluster=mycluster
```

Delete Schedules

To delete schedules for a specific cluster:

```
pgo delete schedule mycluster
```

To delete a schedule by name:

```
pgo delete schedule --schedule-name=mycluster-pgbackrest-full
```

To delete schedules for all clusters with the label `env=test`:

```
pgo delete schedule --selector=env=test
```

38.4. Scaling Replicas

When you create a Cluster, you will see in the output a variety of Kubernetes objects were created including:

- a Deployment holding the primary PostgreSQL database
- a Deployment holding the replica PostgreSQL database
- a service for the primary database
- a service for the replica databases

Since PostgreSQL is a single-primary database by design, the primary Deployment is set to a replica count of 1 and it can not scale beyond that.

With PostgreSQL, you can create any n-number of replicas each of which connect to the primary. This forms a streaming replication PostgreSQL cluster. The PostgreSQL replicas are read-only whereas the primary is read-write.

Syntax

```
$ pgo scale NAME [FLAGS]
```

Flags

Name	Short	Input	Usage
--service-type	N/A	String	The service type to use in the replica Service. If not set, the default in <code>pgo.yaml</code> will be used. Possible values include <code>LoadBalancer</code> , <code>ClusterIP</code> , and <code>NodePort</code> .
--ccp-image-tag	N/A	String	The <code>CCPImageTag</code> to use for cluster creation. If specified, overrides the <code>.pgo.yaml</code> setting.
--no-prompt	-n	N/A	No command line confirmation.

Name	Short	Input	Usage
--node-label	N/A	String	The node label (key) to use in placing the primary database. If not set, any node is used.
--replica-count	N/A	String	The replica count to apply to the clusters (default 1).
--resources-config	N/A	String	The name of a container resource configuration in pgo.yaml that holds CPU and memory requests and limits.
--storage-config	N/A	String	The name of a Storage config in pgo.yaml to use for the cluster storage.

Examples

Scaling Up

Create a Postgres replica:

```
pgo scale mycluster
```

Scale a Postgres replica to a certain number of replicas:

```
pgo scale mycluster --replica-count=3
```

The pgo scale command is additive, in that each time you execute it, another replica is created which is added to the Postgres cluster.

Scaling Down

You can cause a replica to be removed from a Postgres cluster by scaling down the replicas.

Syntax

```
$ pgo scaledown NAME [FLAGS]
```

Flags

Name	Short	Input	Usage
--query	N/A	N/A	Prints the list of targetable replica candidates.
--delete-data	-d	N/A	Causes the data for the scaled down replica to be removed permanently.
--target	N/A	String	The name of a replica to delete.

List the targetable replicas for a given cluster:

```
pgo scaledown mycluster --query
```

You can scale down a cluster as follows:

```
pgo scaledown mycluster --target=mycluster-replica-xxxx
```

Delete the PVC and associated data for the scaled down replica by using the `--delete-data` command flag:

```
pgo scaledown mycluster --target=mycluster-replica-xxxx --delete-data
```

Testing Replication

There are 2 service connections available to the PostgreSQL cluster. One is to the primary database which allows read-write SQL processing, and the other is to the set of read-only replica databases. The replica service performs round-robin load balancing to the replica databases.

You can connect to the primary database and verify that it is replicating to the replica databases as follows:

```
psql -h 10.107.180.159 -U postgres postgres -c 'table pg_stat_replication'
```

Specifying Nodes

The scale command will let you specify a `--node-label` flag which can be used to influence what Kube node the replica will be scheduled upon.

```
pgo scale mycluster --node-label=speed=fast
```

If you don't specify a `--node-label` flag, a node affinity rule of **NotIn** will be specified to **prefer** that the replica be schedule on a node that the primary is not running on.

Overriding Storage Defaults

You can also dictate what container resource and storage configurations will be used for a replica by passing in extra command flags:

```
pgo scale mycluster --storage-config=storage1 --resources-config=small
```

38.5. Manual Failover

Starting with Release 2.6, there is a manual failover command which can be used to promote a replica to a primary role in a PostgreSQL cluster.

This process includes the following actions:

- pick a target replica to become the new primary
- delete the current primary deployment to avoid user requests from going to multiple primary databases (split brain)
- promote the targeted replica using **pg_ctl promote**, this will cause PostgreSQL to go into read-write mode
- re-label the targeted replica to use the primary labels, this will match the primary service selector and cause new requests to the primary to be routed to the new primary (targeted replica)

Syntax

```
$ pgo failover NAME [FLAGS]
```

Flags

Name	Shortname	Input	Usage
<code>--no-prompt</code>	<code>-n</code>	N/ A	No command line confirmation.
<code>--query</code>	N/ A	N/ A	Prints the list of failover candidates.
<code>--target</code>	N/ A	String	The replica target which the failover will occur on.

Examples

Manual Failover

The command works like this:

```
pgo failover mycluster --query
```

That command will show you a list of replica targets you can choose to failover to. You will select one of those for the following command:

```
pgo failover mycluster --target=mycluster-abxq
```

There is a CRD called **pgtask** that will hold the failover request and also the status of that request. You can view the status by viewing it:

```
kubectl get pgtasks mycluster-failover -o yaml
```

Once completed, you will see a new replica has been started to replace the promoted replica, which happens automatically due to the re-label. The Deployment will recreate its pod because of this. The failover typically takes only a few seconds, however, the creation of the replacement replica can take longer depending on how much data is being replicated.

38.6. Upgrading PostgreSQL

The **upgrade** command will allow you to upgrade the PostgreSQL version of your cluster with the `pg_upgrade` utility. Minor or major upgrades are supported. The Crunchy Container Suite `crunchy-upgrade` [<https://crunchydata.github.io/crunchy-containers/stable/container-specifications/crunchy-upgrade/>] container is responsible for performing this task.

By default, it will request confirmation for the command as the operator deletes the existing containers of the database or cluster and recreates them using the currently defined PostgreSQL container image specified in the `pgo.yaml` configuration file or with a defined `--ccp-image-tag` flag. The database data files remain untouched throughout the upgrade.

Once the upgrade job is completed, the operator will create the original database or cluster container mounted with the new PVC which contains the upgraded database files.

As the upgrade is processed, the status of the **pgupgrade** CRD is updated to give the user some insight into how the upgrade is proceeding. Upgrades like this can take a long time if your database is large. The operator creates a watch on the upgrade job to know when and how to proceed.

Syntax

\$ pgo upgrade NAME [FLAGS]

Flags

Name	Short-hand	Input	Usage
--ccp-image-tag	N/A	String	The CCPIImageTag to use for cluster creation. If specified, overrides the pgo.yaml setting.

Examples

Minor Upgrade

Perform a minor PostgreSQL version upgrade:

```
pgo upgrade mycluster
```

Overriding Version

Override the CcpImageTag variable defined in the pgo.yaml configuration file:

```
pgo upgrade mycluster --ccp-image-tag=centos7-9.6.9-1.8.3
pgo upgrade mycluster --ccp-image-tag=centos7-9.6.9-1.8.3
```

Delete Upgrade

To remove an upgrade CRD, issue the following:

```
pgo delete upgrade
```

38.7. Labels

Labels can be applied to clusters and nested according to their type, with any string input being valid.

Syntax

\$ pgo label [NAME][all] [FLAGS]

Flags

Name	Short-hand	Input	Usage
--dry-run	N/A	N/A	Shows the clusters that the label would be applied to, without labelling them.
--label	N/A	String	The new label to apply for any selected or specified clusters.
--selector	-s	String	The selector to use for cluster filtering.

Examples

Applying Labels

You can apply a user defined label to a cluster as follows:

```
pgo label mycluster --label=env=research
```

Or if you wanted to apply it to a selection of clusters:

```
pgo label --label=env=research --selector=project=xray
pgo label all --label=env=research
```

In the first example, a label of **env=research** is applied to the cluster **mycluster**. The second example will apply the label to any clusters that have an existing label of **project=xray** applied or to all clusters.

Removing Labels

You can delete a user defined label from a cluster as follows:

```
pgo delete label mycluster --label=env=research
```

38.8. Creating SQL Policies

Policies are SQL files that can be applied to a single cluster, a selection of clusters, or to all newly created clusters by default.

They are automatically applied to any cluster you create if you define in your **pgo.yaml** configuration a **CLUSTER.POLICIES** value.

Policies are executed as the superuser or **postgres** user in PostgreSQL. These should therefore be exercised with caution.

Syntax

```
$ pgo create policy [NAME] [FLAGS]
```

Flags

Name	Short	Input	Usage
--in-file	N/ A	String	The policy file path to use for adding a policy.
--url	N/ A	N/ A	The url to use for adding a policy.

Examples

Creating Policies

To create a policy use the following syntax:

```
pgo create policy policy1 --in-file=/tmp/policy1.sql
pgo create policy policy1 --url=https://someurl/policy1.sql
```

When you execute this command, it will create a policy named **policy1** using the input file **/tmp/policy1.sql** as input. It will create on the server a PgPolicy CRD with the name **policy1** that you can examine as follows:

```
kubectl get pgpolicies policy1 -o json
```

Apply Policies

To apply an existing policy to a set of clusters, issue a command like this:

```
pgo apply policy1 --selector=name=mycluster
```

When you execute this command, it will look up clusters that have a label value of `name=mycluster` and then it will apply the **policy1** label to that cluster and execute the policy SQL against that cluster using the **postgres** user account.

Testing Policy Application

You can apply policies with a `--dry-run` flag applied to test which clusters the policy would be applied to without actually executing the SQL:

```
pgo apply policy1 --dry-run --selector=name=mycluster
```

Show Policies

To view policies, either all of them or a specific one:

```
pgo show policy all
pgo show policy somepolicy
```

Show Clusters with a Specific Policy

If you want to view the clusters than have a specific policy applied to them, you can use the `--selector` flag as follows to filter on a policy name (e.g. `policy1`):

```
pgo show cluster --selector=policy1=pgpolicy
```

Delete Policies

To delete a policy use the following form:

```
pgo delete policy policy1
pgo delete policy all
```

38.9. Loading Data

A CSV file loading capability is supported. This can be tested through creating a SQL Policy which will create a database table that will be loaded with the CSV data. The loading is based on a load definition found in the `sample-load-config.yaml` file. In that file, the data to be loaded is specified.

When the `pgo load` command is executed, Jobs will be created to perform the loading for each cluster that matches the selector filter.

The load configuration file has the following YAML attributes:

Attribute	Description
COImagePrefix	the pgo-load image prefix to use for the load job
COImageTag	the pgo-load image tag to use for the load job
DbDatabase	the database schema to use for loading the data
DbUser	the database user to use for loading the data
DbPort	the database port of the database to load
TableToLoad	the PostgreSQL table to load
FilePath	the name of the file to be loaded
FileType	either csv or json, determines the type of data to be loaded
PVCName	the name of the PVC that holds the data file to be loaded
SecurityContext	either fsGroup or SupplementalGroup values

For running the **pgo load** examples, you can create the **csv-pvc** PVC by running:

```
kubectl create -f examples/csv-pvc.json
```

Then you can copy sample load files as referenced by the examples into that PVC location (e.g. `/data` or `/nfsfileshare`).

Syntax

```
$ pgo load [FLAGS]
```

Flags

Name	Short	Input	Usage
<code>--load-config</code>	N/A	String	The load configuration to use that defines the load job.
<code>--policies</code>	N/A	String	The policies to apply before loading a file, comma separated.
<code>--selector</code>	-s	String	The selector to use for cluster filtering.

Examples

Loading CSV Files

Load a sample CSV file into a database as follows:

```
pgo load --load-config=$COROOT/examples/sample-load-config.yaml --selector=name=mycluster
```

Including Policies

If you include the **--policies** flag, any specified policies will be applied prior to the data being loaded. For example:

```
pgo load --policies="rlspolicy,xrayapp" --load-config=$COROOT/examples/sample-load-confi
```

39. Authentication

39.1. Credential Management

The `pgo user`, `pgo create user`, and `pgo delete user` commands are used to manage credentials for the PostgreSQL clusters.

Syntax

```
$ pgo user [FLAGS]
```

Flags

Name	Shorthand	Input	Usage
--change-password	N/A	String	Updates the password for a user on selective clusters.
--db	N/A	String	Grants the user access to a database.
--expired	N/A	String	Specifies number of days to check for expiring passwords when using --update-passwords flag to update passwords.
--selector	-s	String	The selector to use for cluster filtering.
--update-passwords	N/A	N/A	Performs password updating on expired passwords.
--password	N/A	N/A	Allows user to specify a password instead of using a generated password.
--valid-days	N/A	Int	Sets passwords for new users to X days (default 30).
--password-length	N/A	Int	When no password is provided, generates a password with this number of characters (default 12).

Examples

Basic User Creation

To create a new Postgres user assigned to the **mycluster** cluster, execute (password will be auto generated and 12 characters long):

```
pgo create user sally --selector=name=mycluster
```

Managed User Creation

To create a new Postgres user to the **mycluster** cluster that has credentials created with Kubernetes Secrets, use the **--managed** flag:

```
pgo create user sally --managed --selector=name=mycluster --password=somepass
```

A **managed** account is one that the Operator can manipulate as well; when you run `pgo test mycluster` the account is tested with the other default accounts, etc.

When you create a managed user, if pgpool is part of your cluster, then pgpool is reconfigured to pick up the new user.

Complex User Creation

In this example, a user named **user1** is created with a **valid until** password date set to expire in 30 days. That user will be granted access to the **userdb** database. This user account also will have an associated **Secret** created to hold the password that was generated for this user. Any clusters that match the selector value will have this user created on it.

```
pgo create user user1 --valid-days=30 --db=userdb --selector=name=xraydb1
```

Deleting Users

To delete a Postgres user in the **mycluster** cluster, execute:

```
pgo delete user sally --selector=name=mycluster
```

If pgpool is part of your cluster, deletion of a managed user will cause pgpool to be reconfigured to pick up the user deletion.

Change Password

To change the password for a user in the **mycluster** cluster (password will be auto generated and 12 characters long):

```
pgo user --change-password=sally --selector=name=mycluster
```

Or to change the password and set an expiration date:

```
pgo user --change-password=user1 --valid-days=10 --selector=name=xray1
```

In this example, a user named **user1** has its password changed to a generated value and the **valid until** expiration date set to 10 days from now. This command will take effect across all clusters that match the selector. If you specify **valid-days=-1** it will mean the password will not expire (e.g. infinity).

If pgpool is part of your cluster, changing a managed user password will cause pgpool to be reconfigured to pick up the password change.

Updating Expired Passwords

To update expired passwords in a cluster:

```
pgo user --update-passwords --selector=name=mycluster --expired=5
```

40. pgbouncer Basics

When a pgbouncer deployment is added into your cluster, it will cause the creation of a Secret that holds the pgbouncer configuration files: `* pg_hba.conf * pgbouncer.ini * users.txt`

Each user that is defined for your cluster is used to define the pgbouncer credentials, using the same password.

The pgbouncer configuration includes a connection to a database with the name of your cluster (e.g. mycluster) and also a database that connects to the cluster's replicas (e.g. mycluster-replica).

When you add a new user, it will cause the pgbouncer to be reconfigured and a new secret to be generated, the pgbouncer is restarted to pick up the new configuration file.

Adding a pgbouncer deployment into your PG cluster follows a sequence similar to this:

```
pgo create cluster mycluster --pgbouncer
```

You can also add pgbouncer after a cluster has been created:

```
pgo create pgbouncer mycluster
```

Note

currently you are required to have a replica in your PG cluster for the pgbouncer sidecar to effectively work, a replica is currently not automatically created when you create a PG cluster.

41. pgpool Basics

Adding a pgpool deployment into your PG cluster follows a sequence similar to this:

```
pgo create cluster mycluster
```

Then you will scale it up:

```
pgo scale mycluster
```

Then you will add managed users of your choice:

```
pgo create user somenewuser mycluster --managed
```

Then you will create a pgpool for the new cluster:

```
pgo create pgpool mycluster
```

This will create a pgpool user credential for each pgo managed user you have created.

41.1. Create pgpool

The `create pgpool` command will create a pgpool deployment that is part of a cluster.

Syntax

```
$ pgo create pgpool CLUSTERNAME [FLAGS]
```

Flags

Name	Short-hand	Input	Usage
--selector	-s	String	The selector to use for cluster filtering.

Examples

Simple Creation

Create a pgpool:

```
pgo create pgpool mycluster
```

Note

currently you are required to have a replica in your PG cluster for the pgpool sidecar to effectively work, a replica is currently not automatically created when you create a PG cluster.

41.2. Delete pgpool

The `delete pgpool` command will by delete the pgpool deployment that is part of a cluster.

Syntax

```
$ pgo delete pgpool CLUSTERNAME [FLAGS]
```

Flags

Name	Short-hand	Input	Usage
--selector	-s	String	The selector to use for cluster filtering.

Examples

Simple Deletion

Delete a pgpool:

```
pgo delete pgpool mycluster
```

41.3. Workflow

Starting with Release 3.4, there is a workflow concept that you can use to check the status of a cluster creation. When you create a cluster (e.g. `pgo create cluster`), you will see in the response a workflow ID. You can use that ID to check the status of the cluster creation.

Syntax

```
pgo show workflow ID
```

42. Reference Architecture

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

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

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

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

Notice that each cluster deployment gets its own unique Persistent Volumes. Each volume can use different storage configurations which is quite powerful.

43. Custom Resource Definitions

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

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

44. Command Line Interface

The pgo command line interface (CLI) is used by a normal end-user to create databases or clusters, or make changes to existing databases.

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

From the CLI, users can view existing clusters that were deployed using the CLI and Operator. Objects that were not previously created by the Crunchy Operator are now viewable from the CLI.

45. Operator Deployment

The PostgreSQL Operator runs within a Deployment in the Kubernetes cluster. An administrator will deploy the operator deployment using the provided script. Once installed and running, the Operator pod will start watching for certain defined events.

The operator watches for create/update/delete actions on the **pgcluster** custom resource definitions. When the CLI creates for example a new **pgcluster** custom resource definition, the operator catches that event and creates pods and services for that new cluster request.

46. CLI Design

The CLI uses the cobra package to implement CLI functionality like help text, config file processing, and command line parsing.

The **pgo** client is essentially a REST client which communicates to the **pgo-apiserver** REST server running within the Operator pod. In some cases you might want to split the apiserver out into its own Deployment but the default deployment has a consolidated pod that contains both the apiserver and operator containers simply for convenience of deployment and updates.

46.1. Verbs

A user works with the CLI by entering verbs to indicate what they want to do, as follows.

```
-  
pgo show cluster all  
pgo delete cluster db1 db2 db3  
pgo create cluster mycluster  
-
```

In the above example, the **show**, **backup**, **delete**, and **create** verbs are used. The CLI is case sensitive and supports only lowercase.

47. Affinity

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

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

```
kubectl get nodes
```

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

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

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

You can always view the actual node your cluster pod is scheduled on through the following command.

```
kubectl get pod -o wide
```

When you scale up a Cluster and add a replica, the scaling will take into account the use of `--node-name`. If it sees that a cluster was created with a specific node name, then the replica Deployment will add an affinity rule to attempt to schedule the replica on a different node than the node the primary is schedule on. This provides a simple version of high availability and causes the primary and replicas to not live on the same Kubernetes node.

48. Debugging

To see if the operator pod is running enter the following -

```
kubectl get pod -l 'name=postgres-operator'
```

To verify the operator is running and has deployed the Custom Resources execute the following -

```
kubectl get crd
```

The full list of CRDs that are created over time are shown below.

NAME	KIND
pgbackups.cr.client-go.k8s.io	CustomResourceDefinition.v1beta1.apiextensions.k8s.io
pgclusters.cr.client-go.k8s.io	CustomResourceDefinition.v1beta1.apiextensions.k8s.io
pgpolicies.cr.client-go.k8s.io	CustomResourceDefinition.v1beta1.apiextensions.k8s.io
pgpolicylogs.cr.client-go.k8s.io	CustomResourceDefinition.v1beta1.apiextensions.k8s.io
pgupgrades.cr.client-go.k8s.io	CustomResourceDefinition.v1beta1.apiextensions.k8s.io
pgtasks.cr.client-go.k8s.io	CustomResourceDefinition.v1beta1.apiextensions.k8s.io

49. Persistent Volumes

Currently, the operator does not delete persistent volumes by default. Instead, it deletes the claims on the volumes. Starting with release 2.4, the Operator will create Jobs that actually run **rm** commands on the data volumes before actually removing the Persistent Volumes if the user passes a `--delete-data`` flag when deleting a database cluster.

Likewise, if the user passes `--delete-backups` during cluster deletion a Job is created to remove all the backups for a cluster include the related Persistent Volume.

50. PostgreSQL Operator Deployment Strategies

This section describes the various deployment strategies offered by the operator. A deployment in this case is the set of objects created in Kubernetes when a custom resource definition of type **pgcluster** is created. CRDs are created by the pgo client command and acted upon by the postgres operator.

50.1. Strategies

To support different types of deployments, the operator supports multiple strategy implementations. Currently there is only a default **cluster** strategy.

In the future, more deployment strategies will be supported to offer users more customization to what they see deployed in their Kubernetes cluster.

Being open source, users can also write their own strategy!

50.2. Specifying a Strategy

In the pgo client configuration file, there is a `CLUSTER.STRATEGY`` setting. The current value of the default strategy is **1**. If you don't set that value, the default strategy is assumed. If you set that value to something not supported, the operator will log an error.

50.3. Strategy Template Files

Each strategy supplies its set of templates used by the operator to create new pods, services, etc.

When the operator is deployed, part of the deployment process is to copy the required strategy templates into a ConfigMap (**operator-conf**) that gets mounted into `/operator-conf` within the operator pod.

The directory structure of the strategy templates is as follows -

```
|-- backup-job.json
|-- cluster
|   |-- 1
|       |-- cluster-deployment-1.json
|       |-- cluster-replica-deployment-1.json
|       |-- cluster-service-1.json
|-- pvc.json
```

In this structure, each strategy's templates live in a subdirectory that matches the strategy identifier. The default strategy templates are denoted by the value of **1** in the directory structure above.

If you add another strategy, the file names **must** be unique within the entire strategy directory. This is due to the way the templates are stored within the ConfigMap.

50.4. Default Cluster Deployment Strategy (1)

Using the default cluster strategy, a **cluster** when created by the operator will create the following on a Kubernetes cluster -

- deployment running a Postgres **primary** container with replica count of 1
- service mapped to the **primary** Postgres database
- service mapped to the **replica** Postgres database
- PVC for the **primary** will be created if not specified in configuration, this assumes you are using a non-shared volume technology (e.g. Amazon EBS), if the `CLUSTER.PVC_NAME` value is set in your configuration then a shared volume technology is assumed (e.g. HostPath or NFS), if a PVC is created for the primary, the naming convention is **clustername** where clustername is the name of your cluster.

If you want to add a Postgres replica to a cluster, you will **scale** the cluster. For each **replica-count**, a Deployment will be created that acts as a PostgreSQL replica.

This is very different than using a StatefulSet to scale up PostgreSQL. Why would you do it this way? Imagine a case where you want different parts of your PostgreSQL cluster to use different storage configurations,. With this method, it can be done through using specific placement and deployments of each part of the cluster.

This same concept applies to node selection for the PostgreSQL cluster components. The Operator will let you define precisely which node that the PostgreSQL component should be placed upon using node affinity rules.

50.5. Cluster Deletion

When you run the following, the cluster and its services will be deleted. However, the data files and backup files will remain as well as the PVCs for this cluster.

```
pgo delete cluster mycluster
```

However, to remove the data files from the PVC you can pass the following flag -

```
--delete-data
```

This causes a workflow to be started to remove the data files on the primary cluster deployment PVC.

The following flag will cause **all** of the backup files to be removed.

```
--delete-backups
```

The data removal workflow includes the following steps -

- create a pgtask CRD to hold the PVC name and cluster name to be removed
- the CRD is watched, and on an ADD will cause a Job to be created that will run the **rmdata** container using the PVC name and cluster name as parameters which determine the PVC to mount, and the file path to remove under that PVC
- the **rmdata** Job is watched by the Operator, and upon a successful status completion the actual PVC is removed

This workflow insures that a PVC is not removed until all the data files are removed. Also, a Job was used for the removal of files since that can be a time consuming task.

The files are removed by the **rmdata** container which essentially issues the following command to remove the files -

```
rm -rf /pgdata/<some path>
```

50.6. Custom Postgres Configurations

Starting in release 2.5, users and administrators can specify a custom set of Postgres configuration files be used when creating a new Postgres cluster. The configuration files you can change include -

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

Different configurations for PostgreSQL might be defined for the following -

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

Global ConfigMap

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

This global configmap holds the pgbackrest.conf file, this is required for pgbackrest backups to work! This also applies to ANY custom configuration file you wish to use, it **MUST** contain a pgbackrest.conf file as a key. See the example for pgo-custom-pg-config for the pgbackrest.conf file and how to add it to your custom configuration ConfigMap.

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

```
$COROOT/examples/custom-config
```

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

Config Files Purpose

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

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

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

The **pgbackrest.conf** file is merely used to tell the Postgres container that it should allocate a pgbackrest configuration directory when initializing the container. The contents of this file do not

get inspected but the name has to be **pgbackrest.conf**. This requirement will change in upcoming operator releases.

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

Labeling

When a custom configMap is used in cluster creation, the Operator labels the primary Postgres Deployment with a label of **custom-config** and a value of what configMap was used when creating the database.

Commands coming in future releases will take advantage of this labeling.

50.7. Metrics Collection

If you add a `--metrics` flag to **pgo create cluster** it will cause the **crunchy-collect** container to be added to your Postgres cluster.

That container requires you run the **crunchy-metrics** containers as defined within the **crunchy-containers** project.

See the crunchy-containers Metrics example [https://crunchydata.github.io/crunchy-containers/stable/getting-started/kubernetes-and-openshift/#_metrics_and_performance] for more details on setting up the **crunchy-metrics** solution.

50.8. Manual Failover

With manual failover some key features include:

- when you perform a failover, a new replica is created to replace the replica that was promoted to even out the cluster to the original number of replicas
- when you perform a failover, the promoted replica is removed from the pgreplica CRD to represent the current **truth**

The `pgo failover --query` command will return a list of replica targets which you can select from. That list include the **Ready** status of the database as well as the Kube node name it is running on.

50.9. Auto Failover

Starting with release 3.1, there is an **auto** failover mechanism that can be leveraged by **pgo** users if enabled.

This feature will cause the operator to start a timer on a database primary that has received a **NotReady** status after the database has started. This can happen if for instance the primary database loses the connection to its database storage (e.g. gluster, NFS).

Once the timer is started, if the primary database does not get back to a **Ready** status within that timer period, a failover is triggered for this cluster. The failover target is selected by the auto failover logic.

The amount of time (in seconds) the auto failover timer will wait before triggering a failover is determined by the following **pgo.yaml** setting:

```
AutofailSleepSeconds: 9
```

If the above setting is not configured a default value of 30 seconds is chose.

The logic of auto failover works like this:

- the readiness probe on the primary database container is executed every few seconds to check the **readiness** of the database, this is what tells Kubernetes whether or not the container is **Ready** or **NotReady**.
- if a **NotReady** state is detected then that event is caught by the operator which is watching for database containers created by the operator
- upon a **NotReady** event, a timer is started for that database which acts as the final check as to if a failover is required for that database
- if the timer expires and the state is still **Not Ready** then the manual failover logic is executed for this cluster which causes a promotion of a replica to primary, and also creates a replacement replica
- only replica targets with a status of **Ready** will be used to select the target to promote

The readiness probe settings are defined in the following template:

```
conf/postgres-operator/cluster/1/cluster-deployment-1.json
```

The readiness probe settings determine how often the database check is performed. See the Kubernetes documentation on readiness probes for more details on these settings.