Installation

Contents

Deployment Requirements 2
Documentation 4
Create Project Structure 4
Configure Environment 4
Configure Operator Templates 5
    Storage 5
Operator Security 5
Create Security Resources 6
Deploy the Operator 6
pgo CLI Installation 6
Verify the Installation 7
Helm Chart 7
Quickstart Script 7
conf Directory 8
conf/postgres-operator/pgo.yaml 8
conf/postgres-operator Directory 8
conf/postgres-operator/cluster 8
Security 8

pgo.yaml Configuration **8**
Storage 9
Storage Configuration Examples 9
    HostPath Example 10
    NFS Example 10
    Storage Class Example 10
Container Resources 10
Miscellaneous (Pgo) 10
Storage Configuration Details 11
Container Resources Details 11
Overriding Storage Configuration Defaults 12
Using Storage Configurations for Disaster Recovery 12
Provisioning 12
Custom Resource Definitions 12
Event Listeners 14
REST API 14
Command Line Interface 14

Node Affinity . . . 14
Fail-over . . . 14
pgbackrest Integration . . . 14
pgbackrest Restore . . . 16
PGO Scheduler . . . 16
Schedule Expression Format . . . 17
pgBackRest Schedules . . . 17
pgBaseBackup Schedules . . . 17
Policy Schedules . . . 17
Custom Resource Definitions . . . 17
Considerations for Multi-zone Cloud Environments . . . 17
Kubernetes RBAC . . . 19
Operator RBAC . . . 20
Making Security Changes . . . 21
API Security . . . 21
Upgrading the Operator . . . 21
Upgrading to Version 3.5.0 From Previous Versions . . . 22
Documentation . . . 22
Hosting Hugo Locally (Optional) . . . 22
Contributing to the Documentation . . . 22

---

itle: "Crunchy Data Postgres Operator"

ate:

raft: false

Latest Release: 3.5.3

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

Use the postgres-operator to:

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

Deployment Requirements

The Operator deploys on Kubernetes and OpenShift clusters. Some form of storage is required, NFS, HostPath, and Storage Classes are currently supported.

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

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

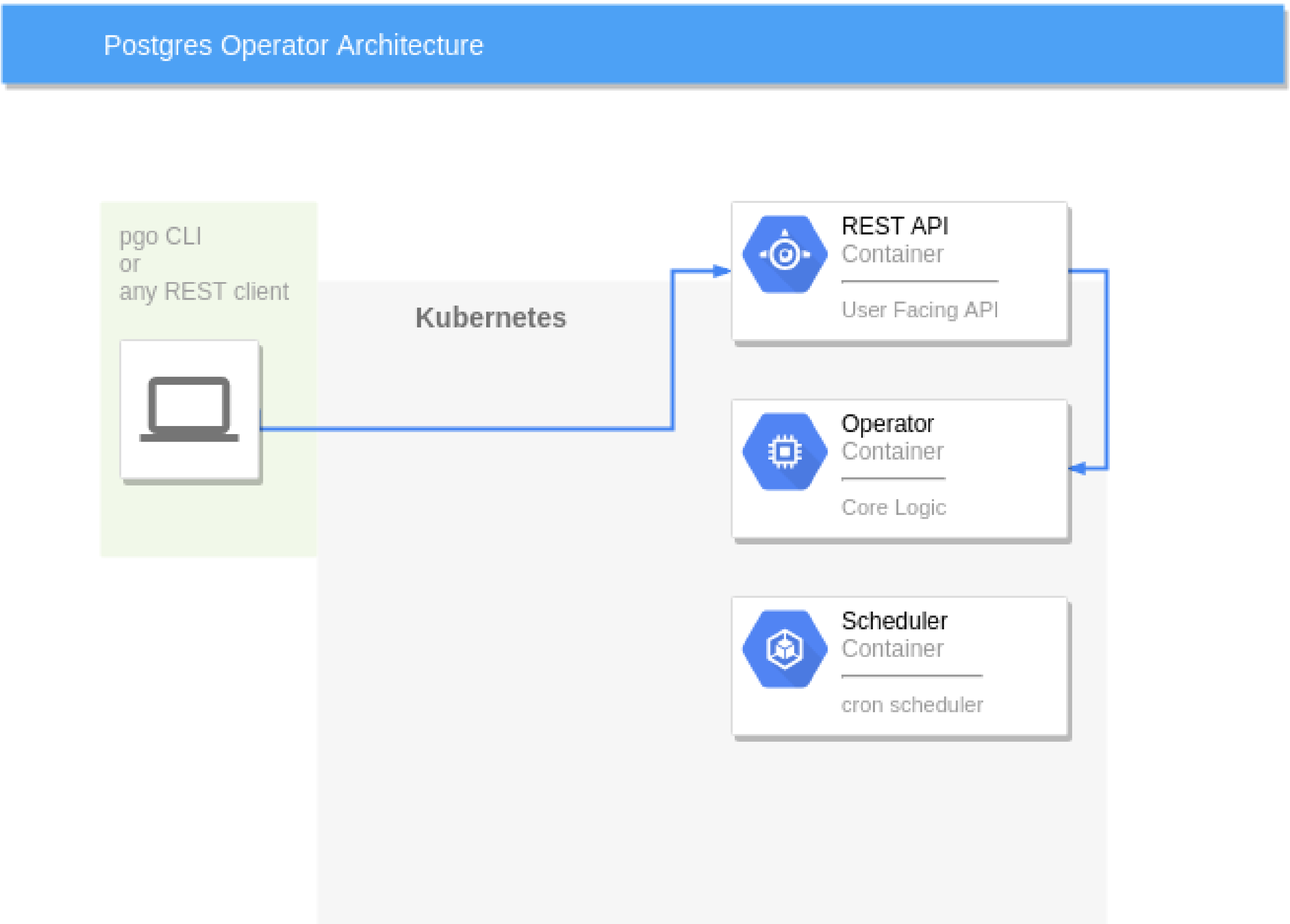

Figure 1: Architecture

Documentation

The following documentation is provided:

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

A full installation of the Operator includes the following steps:

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

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

The Operator can also be deployed with a sample Helm chart and also a *quickstart* script. Those installation methods don't provide the same level of customization that the installation provides but are alternatives. Crunchy also provides an Ansible playbook for Crunchy customers.

See below for details on the Helm and quickstart installation methods.

Create Project Structure

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

```
mkdir -p $HOME/odev/src/github.com/crunchydata $HOME/odev/bin $HOME/odev/pkg
cd $HOME/odev/src/github.com/crunchydata
git clone https://github.com/CrunchyData/postgres-operator.git
cd postgres-operator
git checkout 3.5.3
```

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

Configure Environment

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

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

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

```
make setup
```

{{% /notice %}}

In this example set of environment variables, the CO_NAMESPACE environment variable is set to *demo* as an example namespace in which the Operator will be deployed. See the Design section of documentation on the Operator namespace requirements.

Adjust the namespace value to suit your needs. There is a Makefile target you can run to create the *demo* namespace if you want:

```
make setupnamespace
```

Note, that command sets your Kubernetes context to be *demo* as well, so use with caution if you are using your system's main kubeconfig file.

Configure Operator Templates

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

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

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

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

Storage

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

```
PrimaryStorage: nfsstorage
ArchiveStorage: nfsstorage
BackupStorage: nfsstorage
ReplicaStorage: nfsstorage
Storage:
  hostpathstorage:
    AccessMode:  ReadWriteMany
    Size:  1G
    StorageType:  create
  nfsstorage:
    AccessMode:  ReadWriteMany
    Size:  1G
    StorageType:  create
    SupplementalGroups:  65534
  storageos:
    AccessMode:  ReadWriteOnce
    Size:  1G
    StorageType:  dynamic
    StorageClass:  fast
    Fsgroup:  26
```

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

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

As part of the Operator installation, you will need to adjust these storage settings to suit your deployment requirements.

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

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

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

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

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

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

Operator Security

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

There is a default set of Roles and Users defined respectively in the following files:

```
./conf/postgres-operator/pgouser
./conf/postgres-operator/pgorole
```

Adjust these settings to meet your local requirements.

Create Security Resources

The Operator installation requires Kubernetes administrators to create Resources required by the Operator. These resources are only allowed to be created by a cluster-admin user.

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

As part of the installation, download the *expenv* utility from the Releases page, add that to your path and as cluster admin, run the following Operator Makefile target:

```
make installrbac
```

That target will create the RBAC Resources required by the Operator. This set of Resources is created a single time unless a new Operator release requires these Resources to be recreated. Note that when you run *make installrbac* the set of keys used by the Operator REST API and also the pgbackrest ssh keys are generated. These keys are stored in the ConfigMap used by the Operator for securing connections.

Verify the Operator Custom Resource Definitions are created as follows:

```
kubectl get crd
```

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

Deploy the Operator

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

```
make deployoperator
```

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

This will create a postgres-operator Deployment along with a crunchy-scheduler Deployment, and a postgres-operator Service. So, Operator administrators needing to make changes to the Operator configuration would run this make target to pick up any changes to pgo.yaml or the Operator templates.

pgo CLI Installation

Most users will work with the Operator using the *pgo* CLI tool. That tool is downloaded from the GitHub Releases page for the Operator (https://github.com/crunchydata/postgres-operator/releases).

The *pgo* client is provided in Mac, Windows, and Linux binary formats, download the appropriate client to your local laptop or workstation to work with a remote Operator. Prior to using *pgo*, users testing the Operator on a single host can specify the *postgres-operator* URL as follows:

```
    $ kubectl get service postgres-operator
    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
```

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

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

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

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

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

Lastly, create a *.pgouser* file in your home directory with a credential known by the Operator (see your administrator for Operator credentials to use):

```
username:password
```

You can create this file as follows:

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

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

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

```
pgo version
```

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

Verify the Installation

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

You should see a pod running that contains the Operator:

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

That pod should show 2 of 2 containers in *running* state.

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

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

```
pgo version
```

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

Helm Chart

The Operator Helm chart is located in the following location: ./postgres-operator/chart

Modify the Helm templates to suit your requirements. The Operator templates in the *conf* directory are essentially the same as found in the Helm chart folder. Adjust as mentioned above to customize the installation.

Also, a pre-installation step is currently required prior to installing the Operator Helm chart. Specifically, the following script must be executed prior to installing the chart:

```
./postgres-operator/chart/gen-pgo-keys.sh
```

This script will generate any keys and certificates required to deploy the Operator, and will then place them in the proper directory within the Helm chart.

Quickstart Script

There is a *quickstart* script found in the following GitHub repository location which seeks to automate a simple Operator deployment onto an existing Kubernetes installation:

```
./examples/quickstart.sh
```

This script is a bash script and is intended to run on Linux hosts. The script will ask you questions related to your configuration and the proceed to execute commands to cause the Operator to be deployed. The quickstart script is meant for very simple deployments and to test the Operator and would not be typically used to maintain an Operator deployment.

Get a copy of the script as follows:

```
wget https://raw.githubusercontent.com/CrunchyData/postgres-operator/master/examples/quickstart.sh
chmod +x ./quickstart.sh
```

There are some prerequisites for running this script:

- a running Kubernetes system
- access to a Kube user account that has cluster-admin priviledges, this is required to install the Operator RBAC rules
- a namespace created to hold the Operator
- a Storage Class used for dynamic storage provisioning
- a Mac, Ubuntu, or Centos host to install from, this host and your terminal session should be configured to access your Kube cluster

The operator is template-driven; this makes it simple to configure both the client and the operator.

conf Directory

The Operator is configured with a collection of files found in the *conf* directory. These configuration files are deployed to your Kubernetes cluster when the Operator is deployed. Changes made to any of these configuration files currently require a redeployment of the Operator on the Kubernetes cluster.

The server components of the Operator include Role Based Access Control resources which need to be created a single time by a Kubernetes cluster-admin user. See the Installation section for details on installing a Postgres Operator server.

conf/postgres-operator/pgo.yaml

The *pgo.yaml* file sets many different Operator configuration settings and is described in the [pgo.yaml configuration]({{< ref “pgo-yaml-configuration.md” >}}) documentation section.

The *pgo.yaml* file is deployed along with the other Operator configuration files when you run:

```
make deployoperator
```

conf/postgres-operator Directory

Files within the *conf/postgres-operator* directory contain various templates that are used by the Operator when creating Kubernetes resources. In an advanced Operator deployment, administrators can modify these templates to add their own custom meta-data or make other changes to influence the Resources that get created on your Kubernetes cluster by the Operator.

conf/postgres-operator/cluster

Files within this directory are used specifically when creating PostgreSQL Cluster resources. Sidecar components such as pgBouncer and pgPool II templates are also located within this directory.

As with the other Operator templates, administrators can make custom changes to this set of templates to add custom features or metadata into the Resources created by the Operator.

Security

Security configuration is described in the Security section of this documentation.

pgo.yaml Configuration

The *pgo.yaml* file contains many different configuration settings as described in this section of the documentation.

The *pgo.yaml* file is broken into major sections as described below: ## Cluster

Setting	Definition
BasicAuth	if set to *true* will enable Basic Authentication
PrimaryNodeLabel	newly created primary deployments will specify this node label if specified, unless you override it using the –
ReplicaNodeLabel	newly created replica deployments will specify this node label if specified, unless you override it using the –n
CCPImagePrefix	newly created containers will be based on this image prefix (e.g. crunchydata), update this if you require a c
CCPImageTag	newly created containers will be based on this image version (e.g. centos7-11.3-2.3.2), unless you override it u

Setting	Definition
Port	the PostgreSQL port to use for new containers (e.g. 5432)
LogStatement	postgresql.conf log_statement value (required field)
LogMinDurationStatement	postgresql.conf log_min_duration_statement value (required field)
User	the PostgreSQL normal user name
Strategy	sets the deployment strategy to be used for deploying a cluster, currently there is only strategy *1*
Replicas	the number of cluster replicas to create for newly created clusters, typically users will scale up replicas on the
PgmonitorPassword	the password to use for pgmonitor metrics collection if you specify –metrics when creating a PG cluster
Metrics	boolean, if set to true will cause each new cluster to include crunchy-collect as a sidecar container for metrics
Badger	boolean, if set to true will cause each new cluster to include crunchy-pgbadger as a sidecar container for stat
Policies	optional, list of policies to apply to a newly created cluster, comma separated, must be valid policies in the c
PasswordAgeDays	optional, if set, will set the VALID UNTIL date on passwords to this many days in the future when creating
PasswordLength	optional, if set, will determine the password length used when creating passwords, defaults to 8
ArchiveMode	optional, if set to true will enable archive logging for all clusters created, default is false.
ArchiveTimeout	optional, if set, will determine the archive timeout setting used when ArchiveMode is true, defaults to 60 sec
ServiceType	optional, if set, will determine the service type used when creating primary or replica services, defaults to Cl
Backrest	optional, if set, will cause clusters to have the pgbackrest volume PVC provisioned during cluster creation
BackrestPort	currently required to be port 2022
Autofail	optional, if set, will cause clusters to be checked for auto failover in the event of a non-Ready status
AutofailReplaceReplica	optional, default is false, if set, will determine whether a replica is created as part of a failover to replace the

Storage

Setting	Definition
PrimaryStorage	required, the value of the storage configuration to use for the primary PostgreSQL deployment
XlogStorage	optional, the value of the storage configuration to use for the pgwal (archive) volume for the Postgres cont
BackupStorage	required, the value of the storage configuration to use for backups, including the storage for pgbackrest rep
ReplicaStorage	required, the value of the storage configuration to use for the replica PostgreSQL deployments
ReplicaStorage	required, the value of the storage configuration to use for the replica PostgreSQL deployments
BackrestStorage	required, the value of the storage configuration to use for the pgbackrest shared repository deployment cre
StorageClass	for a dynamic storage type, you can specify the storage class used for storage provisioning(e.g. standard, g
AccessMode	the access mode for new PVCs (e.g. ReadWriteMany, ReadWriteOnce, ReadOnlyMany). See below for des
Size	the size to use when creating new PVCs (e.g. 100M, 1Gi)
Storage.storage1.StorageType	supported values are either *dynamic*, *create*, if not supplied, *create* is used
Fsgroup	optional, if set, will cause a *SecurityContext* and *fsGroup* attributes to be added to generated Pod and De
SupplementalGroups	optional, if set, will cause a SecurityContext to be added to generated Pod and Deployment definitions
MatchLabels	optional, if set, will cause the PVC to add a *matchlabels* selector in order to match a PV, only useful when

Storage Configuration Examples

In *pgo.yaml*, you will need to configure your storage configurations depending on which storage you are wanting to use for Operator provisioning of Persistent Volume Claims. The examples below are provided as a sample. In all the examples you are free to change the *Size* to meet your requirements of Persistent Volume Claim size.

HostPath Example

HostPath is provided for simple testing and use cases where you only intend to run on a single Linux host for your Kubernetes cluster.

```
  hostpathstorage:
    AccessMode:  ReadWriteMany
    Size:  1G
    StorageType:  create
```

NFS Example

In the following NFS example, notice that the *SupplementalGroups* setting is set, this can be whatever GID you have your NFS mount set to, typically we set this *nfsnobody* as below. NFS file systems offer a *ReadWriteMany* access mode.

```
  nfsstorage:
    AccessMode:  ReadWriteMany
    Size:  1G
    StorageType:  create
    SupplementalGroups:  65534
```

Storage Class Example

In the following example, the important attribute to set for a typical Storage Class is the *Fsgroup* setting. This value is almost always set to *26* which represents the Postgres user ID that the Crunchy Postgres container runs as. Most Storage Class providers offer *ReadWriteOnce* access modes, but refer to your provider documentation for other access modes it might support.

```
  storageos:
    AccessMode:  ReadWriteOnce
    Size:  1G
    StorageType:  dynamic
    StorageClass:  fast
    Fsgroup:  26
```

Container Resources

Setting	Definition
DefaultContainerResource	optional, the value of the container resources configuration to use for all database containers, if not set, no re
DefaultLoadResource	optional, the value of the container resources configuration to use for pgo-load containers, if not set, no resou
DefaultLspvcResource	optional, the value of the container resources configuration to use for pgo-lspvc containers, if not set, no reso
DefaultRmdataResource	optional, the value of the container resources configuration to use for pgo-rmdata containers, if not set, no re
DefaultBackupResource	optional, the value of the container resources configuration to use for crunchy-backup containers, if not set, n
DefaultPgbouncerResource	optional, the value of the container resources configuration to use for crunchy-pgbouncer containers, if not se
DefaultPgpoolResource	optional, the value of the container resources configuration to use for crunchy-pgpool containers, if not set, n
RequestsMemory	request size of memory in bytes
RequestsCPU	request size of CPU cores
LimitsMemory	request size of memory in bytes
LimitsCPU	request size of CPU cores

Miscellaneous (Pgo)

Setting	Definition
PreferredFailoverNode	optional, a label selector (e.g. hosttype=offsite) that if set, will be used to pick the failover target which is runnin
LSPVCTemplate	the PVC lspvc template file that lists PVC contents
LoadTemplate	the load template file used for load jobs

Setting	Definition
COImagePrefix	image tag prefix to use for the Operator containers
COImageTag	image tag to use for the Operator containers
Audit	boolean, if set to true will cause each apiserver call to be logged with an *audit* marking

Storage Configuration Details

You can define n-number of Storage configurations within the *pgo.yaml* file. Those Storage configurations follow these conventions -

- they must have lowercase name (e.g. storage1)
- they must be unique names (e.g. mydrstorage, faststorage, slowstorage)

These Storage configurations are referenced in the BackupStorage, ReplicaStorage, and PrimaryStorage configuration values. However, there are command line options in the *pgo* client that will let a user override these default global values to offer you the user a way to specify very targeted storage configurations when needed (e.g. disaster recovery storage for certain backups).

You can set the storage AccessMode values to the following:

- *ReadWriteMany* - mounts the volume as read-write by many nodes
- *ReadWriteOnce* - mounts the PVC as read-write by a single node
- *ReadOnlyMany* - mounts the PVC as read-only by many nodes

These Storage configurations are validated when the *pgo-apiserver* starts, if a non-valid configuration is found, the apiserver will abort. These Storage values are only read at *apiserver* start time.

The following StorageType values are possible -

- *dynamic* - this will allow for dynamic provisioning of storage using a StorageClass.
- *create* - This setting allows for the creation of a new PVC for each PostgreSQL cluster using a naming convention of *clustername*. When set, the *Size*, *AccessMode* settings are used in constructing the new PVC.

The operator will create new PVCs using this naming convention: *dbname* where *dbname* is the database name you have specified. For example, if you run:

```
pgo create cluster example1
```

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

Note, when Storage Type is *create*, you can specify a storage configuration setting of *MatchLabels*, when set, this will cause a *selector* of *key=value* to be added into the PVC, this will let you target specific PV(s) to be matched for this cluster. Note, if a PV does not match the claim request, then the cluster will not start. Users that want to use this feature have to place labels on their PV resources as part of PG cluster creation before creating the PG cluster. For example, users would add a label like this to their PV before they create the PG cluster:

```
kubectl label pv somepv myzone=somezone
```

If you do not specify *MatchLabels* in the storage configuration, then no match filter is added and any available PV will be used to satisfy the PVC request. This option does not apply to *dynamic* storage types.

Example PV creation scripts are provided that add labels to a set of PVs and can be used for testing: `$COROOT/pv/create-pv-nfs-labels.sh` in that example, a label of **crunchyzone=red** is set on a set of PVs to test with.

The *pgo.yaml* includes a storage config named **nfsstoragered** that when used will demonstrate the label matching. This feature allows you to support n-number of NFS storage configurations and supports spreading a PG cluster across different NFS storage configurations.

Container Resources Details

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

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

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

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

```
Type     Reason            Age                From               Message
----     ------            ----               ----               -------
Warning  FailedScheduling  49s (x8 over 1m)   default-scheduler  No nodes are available that
    match all of the predicates: Insufficient memory (1).
```

Overriding Storage Configuration Defaults

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

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.

Using Storage Configurations for Disaster Recovery

A simple mechanism for partial disaster recovery can be obtained by leveraging network storage, Kubernetes storage classes, and the storage configuration options within the Operator.

For example, if you define a Kubernetes storage class that refers to a storage backend that is running within your disaster recovery site, and then use that storage class as a storage configuration for your backups, you essentially have moved your backup files automatically to your disaster recovery site thanks to network storage.

Provisioning

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

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

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

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

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

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*

Metadata about the Postgres cluster deployments are stored within these CRD resources which act as the source of truth for the Operator.

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

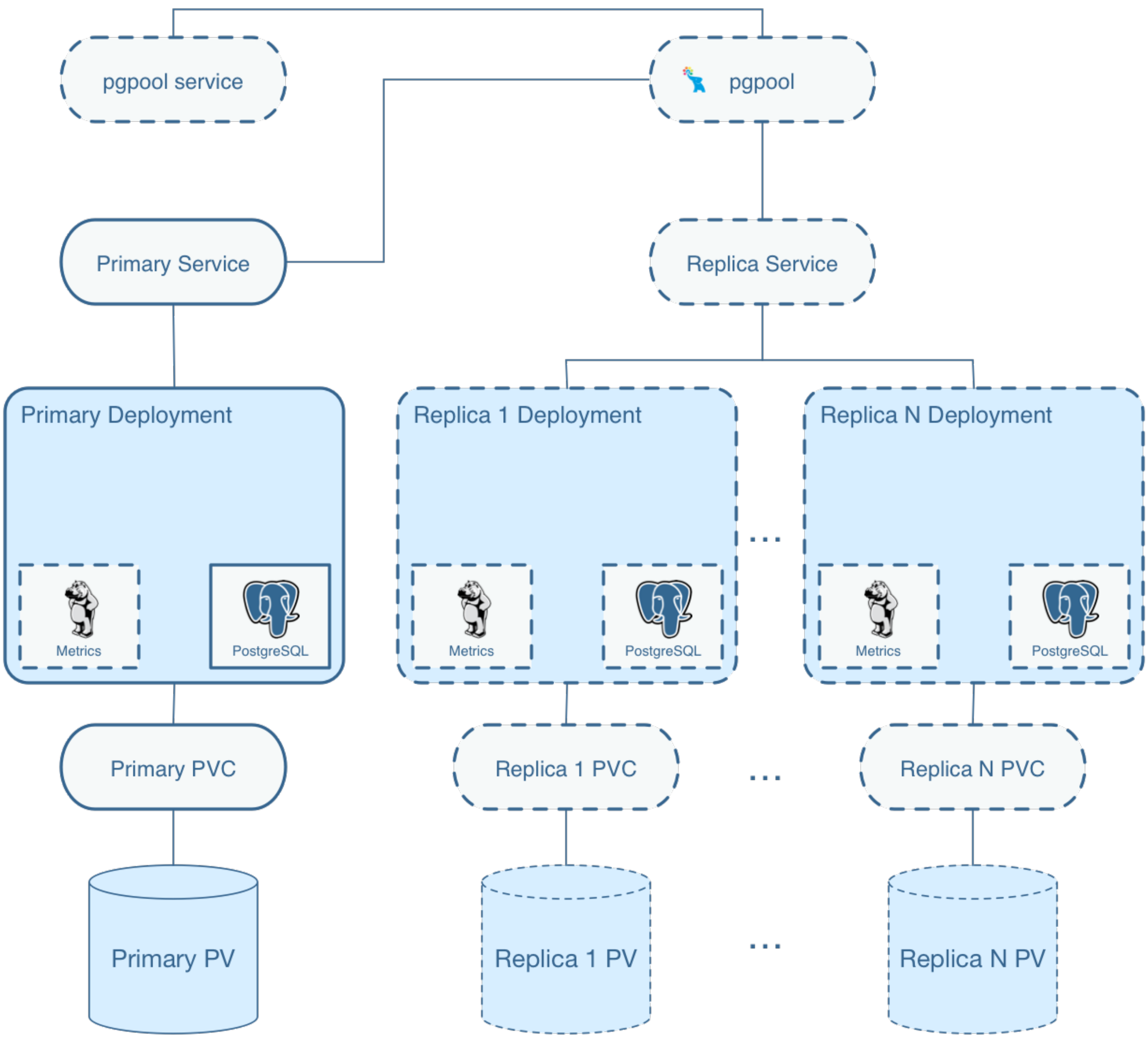

Figure 2: Reference

Event Listeners

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

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

REST API

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

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

Command Line Interface

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

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

Node Affinity

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

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

```
kubectl get nodes
```

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

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

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

When you scale up a Cluster and add a replica, the scaling will take into account the use of `--node-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

Fail-over

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

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

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

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

The fail-over logic includes:

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

pgbackrest Integration

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

The following diagrams depicts some of the integration features:

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

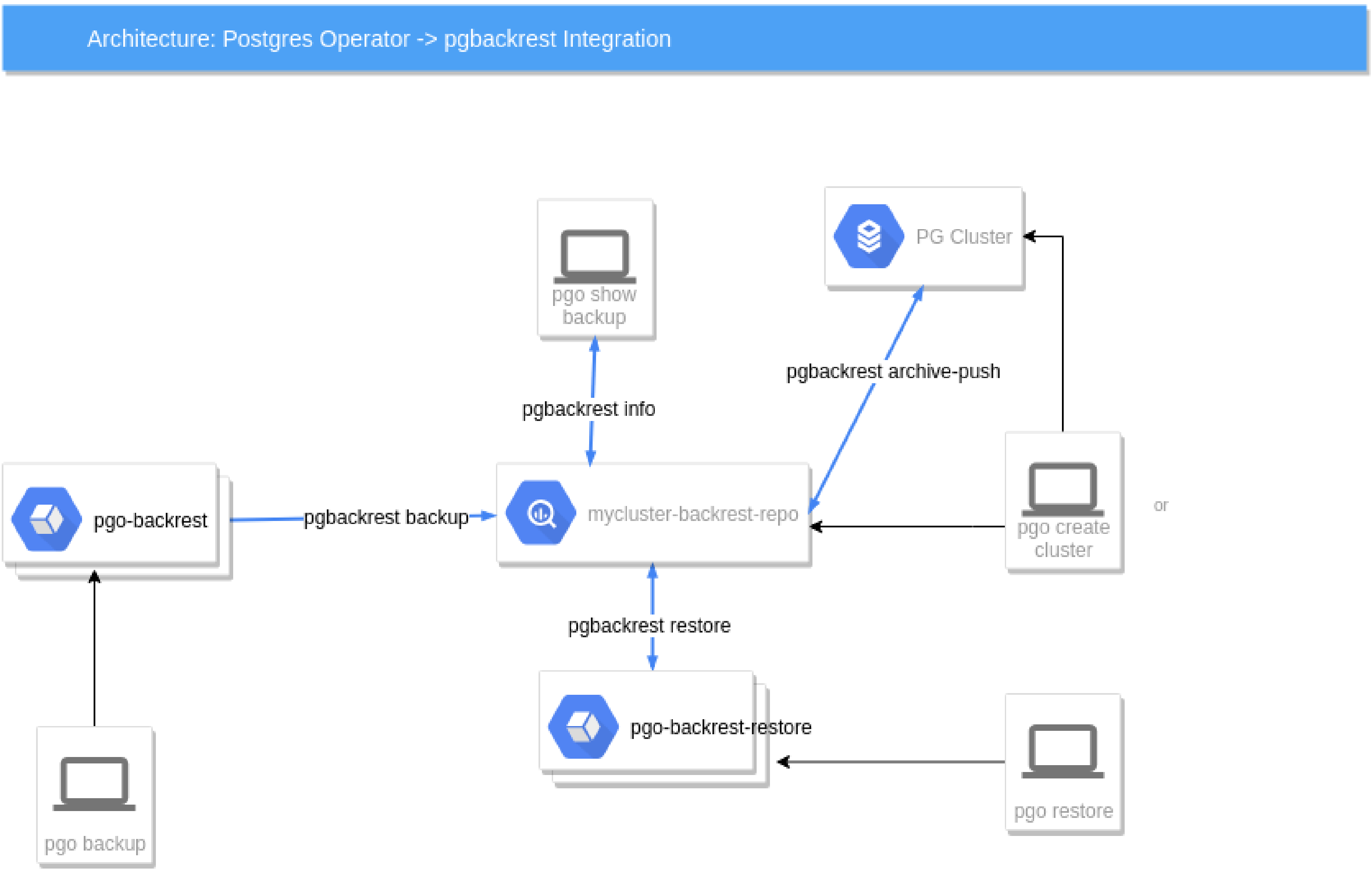

Figure 3: alt text

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

pgbackrest Restore

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

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

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

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

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

- if a schedule has been created for this PG cluster, delete that schedule prior to performing a restore
- after a restore, exec into the PG primary and make sure the database has fully recovered by looking at the database logs, if not recovered, you might have to run psql command *select pg_wal_replay_resume()* to complete the recovery, on PG 9.6/9.5 systems, the command you will use is *select pg_xlog_replay_resume()*.
- a pgBackRest restore is destructive in the sense that it deletes the existing primary deployment for the cluster prior to creating a new deployment containing the restored primary database. However, in the event that the pgBackRest restore job fails, the `pgo restore` command be can be run again, and instead of first deleting the primary deployment (since one no longer exists), a new primary will simply be created according to any options specified. Additionally, even though the original primary deployment will be deleted, the original primary PVC will remain.
- there is currently no Operator validation of user entered pgBackRest command options, you will need to make sure to enter these correctly, if not the pgBackRest restore command can fail.
- the restore workflow does not perform a backup after the restore nor does it verify that any replicas are in a working status after the restore, it is possible you might have to take actions on the replica to get them back to replicating with the new restored primary.
- pgbackrest.org suggests running a pgbackrest backup after a restore, this needs to be done by the DBA as part of a restore
- when performing a pgBackRest restore, the **node-label** flag can be utilized to target a specific node for both the pgBackRest restore job and the new (i.e. restored) primary deployment that is then created for the cluster. If a node label is not specified, the restore job will not target any specific node, and the restored primary deployment will inherit any node label's defined for the original primary deployment.

PGO Scheduler

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

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

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

Schedules are removed automatically when the configmaps are deleted.

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

Schedule Expression Format

Schedules are expressed using the following rules:

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

pgBackRest Schedules

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

pgBaseBackup Schedules

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

```
pgo backup mycluster
```

Policy Schedules

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

Custom Resource Definitions

The Operator makes use of custom resource definitions to maintain state and resource definitions as offered by the Operator.

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

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

Considerations for Multi-zone Cloud Environments

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

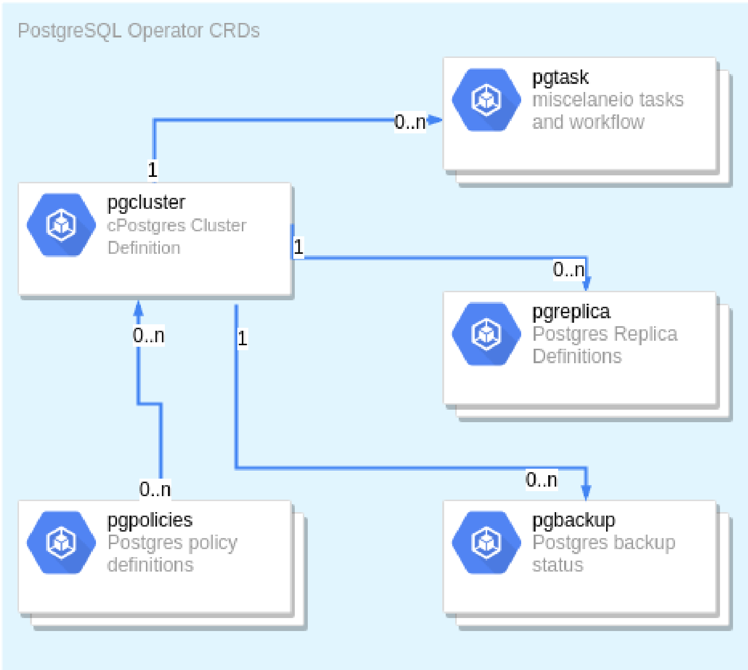

Figure 4: Reference

Default Behavior By default, the Kubernetes scheduler will ensure any pods created that claim a specific volume via a PVC are scheduled on a node in the same zone as that volume. This is part of the multi-zone support that is included in Kubernetes by default. However, when using dynamic provisioning, volumes are not provisioned in a topology-aware manner by default, which means a volume will not be provisioned according to the same scheduling requirements that will be placed on the pod that will be using it (e.g. it will not consider node selectors, resource requirements, pod affinity/anti-affinity, and various other scheduling requirements). Rather, PVC's are immediately bound as soon as they are requested, which means volumes are provisioned without knowledge of these scheduling requirements. This behavior is the result of the `volumeBindingMode` defined on the Storage Class being utilized to dynamically provision the volume, which is set to `Immediate` by default. This can be seen in the following Storage Class definition, which defines a Storage Class for a Google Cloud Engine Persistent Disk (GCE PD) that uses the default value of `Immediate` for its `volumeBindingMode`:

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

Unfortunately, using `Immediate` for the `volumeBindingMode` in a multi-zone cluster can result in undesired behavior when using the Operator, being that the scheduler will ignore any requested *(but not mandatory)* scheduling requirements if necessary to ensure the pod can be scheduled. Specifically, the scheduler will ultimately schedule the pod on a node in the same zone as the volume, even if another node was requested for scheduling that pod. For instance, a node label might be specified using the `--node-label` option when creating a cluster using the `pgo create cluster` command in order target a specific node (or nodes) for the deployment of that cluster. Within the Operator, a **node label** is implemented as a `preferredDuringSchedulingIgnoredDuringExecution` node affinity rule, which is an affinity rule that Kubernetes will attempt to adhere to when scheduling any pods for the cluster, but *will not guarantee* (more information on node affinity rules can be found here). Therefore, if the volume ends up in a zone other than the zone containing the node (or nodes) defined by the node label, the node label will be ignored, and the pod will be scheduled according to the zone containing the volume.

Topology Aware Volumes In order to overcome the behavior described above in a multi-zone cluster, volumes must be made topology aware. This is accomplished by setting the `volumeBindingMode` for the storage class to `WaitForFirstConsumer`, which delays the dynamic provisioning of a volume until a pod using it is created. In other words, the PVC is no longer bound as soon as it is requested, but rather waits for a pod utilizing it to be creating prior to binding. This change ensures that volume can take into account the scheduling requirements for the pod, which in the case of a multi-zone cluster means ensuring the volume is provisioned in the same zone containing the node where the pod has be scheduled. This also means the scheduler should no longer ignore a node label in order to follow a volume to another zone when scheduling a pod, since the volume will now follow the pod according to the pod's specificscheduling requirements. The following is an example of the the same Storage Class defined above, only with `volumeBindingMode` now set to `WaitForFirstConsumer`:

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

Additional Solutions If you are using a version of Kubernetes that does not support `WaitForFirstConsumer`, an alternate *(and now deprecated)* solution exists in the form of parameters that can be defined on the Storage Class definition to ensure volumes are provisioned in a specific zone (or zones). For instance, when defining a Storage Class for a GCE PD for use in Google Kubernetes Engine (GKE) cluster, the **zone** parameter can be used to ensure any volumes dynamically provisioned using that Storage Class are located in that specific zone. The following is an example of a Storage Class for a GKE cluster that will provision volumes in the **us-east1** zone:

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

Once storage classes have been defined for one or more zones, they can then be defined as one or more storage configurations within the pgo.yaml configuration file (as described in the PGO YAML configuration guide). From there those storage configurations can then be selected when creating a new cluster, as shown in the following example:

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

With this approach, the pod will once again be scheduled according to the zone in which the volume was provisioned. However, the zone parameters defined on the Storage Class bring consistency to scheduling by guaranteeing that the volume, and therefore also the pod using that volume, are scheduled in a specific zone as defined by the user, bringing consistency and predictability to volume provisioning and pod scheduling in multi-zone clusters.

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

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

Kubernetes RBAC

Install the requisite Operator RBAC resources, *as a Kubernetes cluster admin user*, by running a Makefile target:

```
make installrbac
```

This script creates the following RBAC resources on your Kubernetes cluster:

Setting	Definition
Custom Resource Definitions	pgbackups
	pgclusters
	pgpolicies
	pgreplicas
	pgtasks
	pgupgrades

Setting	Definition
Cluster Roles	pgopclusterrole
	pgopclusterrolecrd
	scheduler-sa
Cluster Role Bindings	pgopclusterbinding
	pgopclusterbindingcrd
	scheduler-sa
Service Account	scheduler-sa
	postgres-operator
	pgo-backrest
	scheduler-sa
Roles	pgo-role
	pgo-backrest-role
Role Bindings	pgo-backrest-role-binding

Operator RBAC

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

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

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

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

```
username:password
```

The users pgouser file can also be located at: */etc/pgo/pgouser* or it can be found at a path specified by the PGOUSER environment variable.

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

Permission	Description
ApplyPolicy	allow *pgo apply*
CreateBackup	allow *pgo backup*
CreateCluster	allow *pgo create cluster*
CreateFailover	allow *pgo failover*
CreatePgbouncer	allow *pgo create pgbouncer*
CreatePgpool	allow *pgo create pgpool*
CreatePolicy	allow *pgo create policy*
CreateSchedule	allow *pgo create schedule*
CreateUpgrade	allow *pgo upgrade*
CreateUser	allow *pgo create user*
DeleteBackup	allow *pgo delete backup*
DeleteCluster	allow *pgo delete cluster*
DeletePgbouncer	allow *pgo delete pgbouncer*
DeletePgpool	allow *pgo delete pgpool*
DeletePolicy	allow *pgo delete policy*

Permission	Description
DeleteSchedule	allow *pgo delete schedule*
DeleteUpgrade	allow *pgo delete upgrade*
DeleteUser	allow *pgo delete user*
DfCluster	allow *pgo df*
Label	allow *pgo label*
Load	allow *pgo load*
Reload	allow *pgo reload*
Restore	allow *pgo restore*
ShowBackup	allow *pgo show backup*
ShowCluster	allow *pgo show cluster*
ShowConfig	allow *pgo show config*
ShowPolicy	allow *pgo show policy*
ShowPVC	allow *pgo show pvc*
ShowSchedule	allow *pgo show schedule*
ShowUpgrade	allow *pgo show upgrade*
ShowWorkflow	allow *pgo show workflow*
Status	allow *pgo status*
TestCluster	allow *pgo test*
UpdateCluster	allow *pgo update cluster*
User	allow *pgo user*
Version	allow *pgo version*

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

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

Making Security Changes

The Operator today requires you to make Operator security changes in the pgouser and pgorole files, and for those changes to take effect you are required to re-deploy the Operator:

```
make deployoperator
```

This will recreate the *pgo-auth-secret* Secret that stores these files and is mounted by the Operator during its initialization.

API Security

The Operator REST API is secured with keys stored in the *pgo-auth-secret* Secret. Adjust the default keys to meet your security requirements using your own keys. The *pgo-auth-secret* Secret is created when you run:

```
make deployoperator
```

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

```
make installrbac
```

Upgrading the Operator

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

This section of the documentation shows specific steps required to the latest version from the previous version.

Upgrading to Version 3.5.0 From Previous Versions

- For clusters created in prior versions that used pgbackrest, you will be required to first create a pgbasebackup for those clusters, and after upgrading to Operator 3.5, you will need to restore those clusters from the pgbasebackup into a new cluster with *–pgbackrest* enabled, this is due to the new pgbackrest shared repository being implemented in 3.5. This is a breaking change for anyone that used pgbackrest in Operator versions prior to 3.5.
- The pgingest CRD is removed. You will need to manually remove it from any deployments of the operator after upgrading to this version. This includes removing ingest related permissions from the pgorole file. Additionally, the API server now removes the ingest related API endpoints.
- Primary and replica labels are only applicable at cluster creation and are not updated after a cluster has executed a failover. A new *service-name* label is applied to PG cluster components to indicate whether a deployment/pod is a primary or replica. *service-name* is also the label now used by the cluster services to route with. This scheme allows for an almost immediate failover promotion and avoids the pod having to be bounced as part of a failover. Any existing PostgreSQL clusters will need to be updated to specify them as a primary or replica using the new *service-name* labeling scheme.

- The autofail label was moved from deployments and pods to just the pgcluster CRD to support autofail toggling.
- The storage configurations in *pgo.yaml* support the MatchLabels attribute for NFS storage. This will allow users to have more than a single NFS backend,. When set, this label (key=value) will be used to match the labels on PVs when a PVC is created.
- The UpdateCluster permission was added to the sample pgorole file to support the new pgo update CLI command. It was also added to the pgoperm file.
- The pgo.yaml adds the PreferredFailoverNode setting. This is a Kubernetes selector string (e.g. key=value). This value if set, will cause fail-over targets to be preferred based on the node they run on if that node is in the set of *preferred*.
- The ability to select nodes based on a selector string was added. For this to feature to be used, multiple replicas have to be in a ready state, and also at the same replication status. If those conditions are not met, the default fail-over target selection is used.
- The pgo.yaml file now includes a new storage configuration, XlogStorage, which when set will cause the xlog volume to be allocated using this storage configuration. If not set, the PrimaryStorage configuration will be used.
- The pgo.yaml file now includes a new storage configuration, BackrestStorage, will cause the pgbackrest shared repository volume to be allocated using this storage configuration.
- The pgo.yaml file now includes a setting, AutofailReplaceReplica, which will enable or disable whether a new replica is created as part of a fail-over. This is turned off by default.

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

Documentation

The documentation website is generated using Hugo and https://pages.github.com/)GitHub Pages.

Hosting Hugo Locally (Optional)

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

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

```
cd $COROOT/hugo/
hugo server
```

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

Contributing to the Documentation

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

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

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

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

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

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

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

1. Create a Pull Request from your Fork to the Develop branch.
2. Update the checklists in the Pull Request Description.
3. Reference which issues this Pull Request is resolving.