Percona Distribution for PostgreSQL Operator

Release 1.0.0

Percona LLC and/or its affiliates 2009-2021

Oct 07, 2021

CONTENTS

Kubernetes have added a way to manage containerized systems, including database clusters. This management is achieved by controllers, declared in configuration files. These controllers provide automation with the ability to create objects, such as a container or a group of containers called pods, to listen for an specific event and then perform a task.

This automation adds a level of complexity to the container-based architecture and stateful applications, such as a database. A Kubernetes Operator is a special type of controller introduced to simplify complex deployments. The Operator extends the Kubernetes API with custom resources.

The Percona Distribution for PostgreSQL Operator is based on best practices for configuration and setup of a Percona Distribution for PostgreSQL cluster. The benefits of the Operator are many, but saving time and delivering a consistent and vetted environment is key.

Part I

Requirements

CHAPTER

ONE

SYSTEM REQUIREMENTS

The Operator is validated for deployment on Kubernetes, GKE and EKS clusters. The Operator is cloud native and storage agnostic, working with a wide variety of storage classes, hostPath, and NFS.

1.1 Officially supported platforms

The following platforms were tested and are officially supported by the Operator 1.0.0:

- Google Kubernetes Engine (GKE) 1.17 - 1.21
- Amazon Elastic Container Service for Kubernetes (EKS) 1.21
- OpenShift 4.6 - 4.8

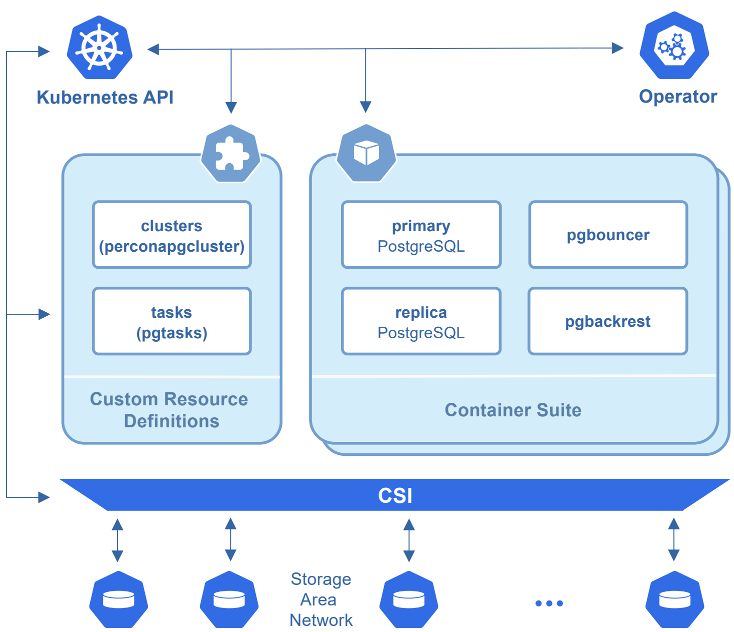
Other Kubernetes platforms may also work but have not been tested.

CHAPTER

TWO

DESIGN OVERVIEW

The Percona Distribution for PostgreSQL Operator automates and simplifies deploying and managing open source PostgreSQL clusters on Kubernetes. The Operator is based on CrunchyData's PostgreSQL Operator.

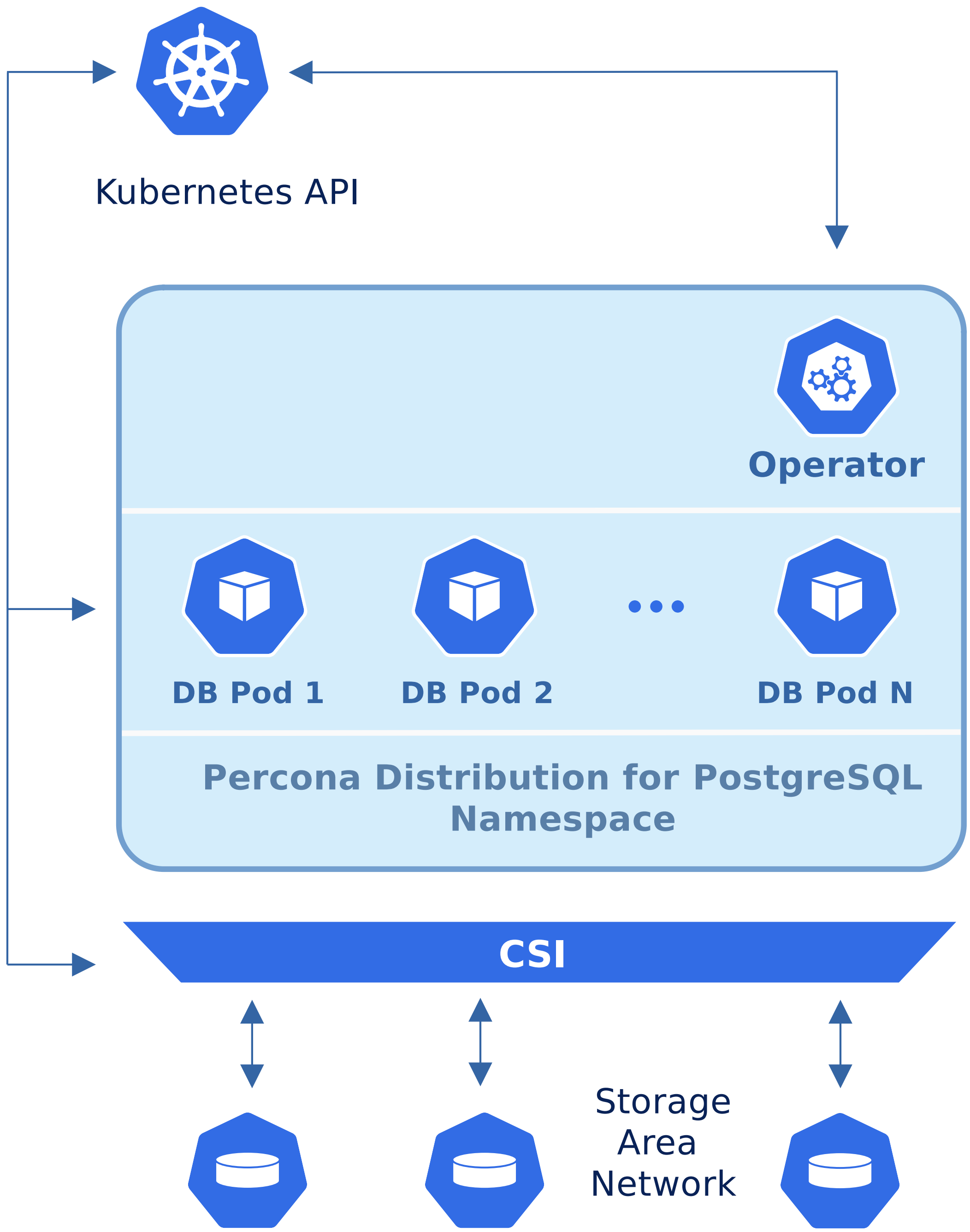


PostgreSQL containers deployed with the PostgreSQL Operator include the following components:

- The PostgreSQL database management system, including:
  - PostgreSQL Additional Supplied Modules,
  - pgAudit PostgreSQL auditing extension,
  - PostgreSQL set_user Extension Module,
  - wal2json output plugin,

- The pgBackRest Backup & Restore utility,
- The pgBouncer connection pooler for PostgreSQL,
- The PostgreSQL high-availability implementation based on the Patroni template,
- the pg_stat_monitor PostgreSQL Query Performance Monitoring utility,
- LLVM (for JIT compilation).

To provide high availability the Operator involves node affinity to run PostgreSQL Cluster instances on separate worker nodes if possible. If some node fails, the Pod with it is automatically re-created on another node.



To provide data storage for stateful applications, Kubernetes uses Persistent Volumes. A *PersistentVolumeClaim* (PVC) is used to implement the automatic storage provisioning to pods. If a failure occurs, the Container Storage Interface (CSI) should be able to re-mount storage on a different node.

The Operator functionality extends the Kubernetes API with Custom Resources Definitions. These CRDs provide extensions to the Kubernetes API, and, in the case of the Operator, allow you to perform actions such as creating a PostgreSQL Cluster, updating PostgreSQL Cluster resource allocations, adding additional utilities to a PostgreSQL cluster, e.g. pgBouncer for connection pooling and more.

When a new Custom Resource is created or an existing one undergoes some changes or deletion, the Operator automatically creates/changes/deletes all needed Kubernetes objects with the appropriate settings to provide a proper Percona PostgreSQL Cluster operation.

Following CRDs are created while the Operator installation:

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

Part II

Installation guide

CHAPTER

THREE

INSTALL PERCONA DISTRIBUTION FOR POSTGRESQL ON KUBERNETES

Following steps will allow you to install the Operator and use it to manage Percona Distribution for PostgreSQL in a Kubernetes-based environment.

1. First of all, clone the percona-postgresql-operator repository:

```
git clone -b v1.0.0 https://github.com/percona/percona-postgresql-operator
cd percona-postgresql-operator
```

Note: It is crucial to specify the right branch with `-b` option while cloning the code on this step. Please be careful.

2. The next thing to do is to add the `pgo` namespace to Kubernetes, not forgetting to set the correspondent context for further steps:

```
$ kubectl create namespace pgo
$ kubectl config set-context $(kubectl config current-context) --namespace=pgo
```

Note: To use different namespace, you should edit *all occurrences* of the `namespace: pgo` line in both `deploy/cr.yaml` and `deploy/operator.yaml` configuration files.

3. Deploy the operator with the following command:

```
$ kubectl apply -f deploy/operator.yaml
```

4. After the operator is started Percona Distribution for PostgreSQL can be created at any time with the following command:

```
$ kubectl apply -f deploy/cr.yaml
```

Creation process will take some time. The process is over when both operator and replica set pod have reached their Running status:

```
$ kubectl get pods
NAME                                              READY   STATUS      RESTARTS   AGE
backrest-backup-cluster1-j275w                    0/1     Completed   0          10m
cluster1-85486d645f-gpxzb                         1/1     Running     0          10m
cluster1-backrest-shared-repo-6495464548-c8wvl    1/1     Running     0          10m
```

(continues on next page)

(continued from previous page)

```
cluster1-pgbouncer-fc45869f7-s86rf                1/1     Running   0          10m
pgo-deploy-rhv6k                                  0/1     Completed 0          5m
postgres-operator-8646c68b57-z8m62                4/4     Running   1          5m
```

5. During previous steps, the Operator has generated several secrets, including the password for the `pguser` user, which you will need to access the cluster.

   Use `kubectl get secrets` command to see the list of Secrets objects (by default Secrets object you are interested in has `cluster1-pguser-secret` name). Then `kubectl get secret cluster1-pguser-secret -o yaml` will return the YAML file with generated secrets, including the password which should look as follows:

```
...
data:
  ...
  password: cGd1c2VyX3Bhc3N3b3JkCg==
```

   Here the actual password is base64-encoded, and `echo 'cGd1c2VyX3Bhc3N3b3JkCg==' | base64 --decode` will bring it back to a human-readable form (in this example it will be a `pguser_password` string).

6. Check connectivity to newly created cluster

```
$ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-
↪postgresql:13.2 --restart=Never -- bash -il
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer -p␣
↪5432 -U pguser pgdb
```

   This command will connect you to the PostgreSQL interactive terminal.

```
psql (13.2)
Type "help" for help.
pgdb=>
```

CHAPTER

FOUR

INSTALL PERCONA DISTRIBUTION FOR POSTGRESQL ON OPENSHIFT

Following steps will allow you to install the Operator and use it to manage Percona Distribution for PostgreSQL on Red Hat OpenShift platform. For more information on the OpenShift, see its official documentation.

Following steps will allow you to install the Operator and use it to manage Percona Distribution for PostgreSQL on OpenShift.

1. First of all, clone the percona-postgresql-operator repository:

```
git clone -b v1.0.0 https://github.com/percona/percona-postgresql-operator
cd percona-postgresql-operator
```

Note: It is crucial to specify the right branch with `-b` option while cloning the code on this step. Please be careful.

2. The next thing to do is to add the `pgo` namespace to Kubernetes, not forgetting to set the correspondent context for further steps:

```
$ oc create namespace pgo
$ oc config set-context $(kubectl config current-context) --namespace=pgo
```

Note: To use different namespace, you should edit *all occurrences* of the `namespace: pgo` line in both `deploy/cr.yaml` and `deploy/operator.yaml` configuration files.

3. Deploy the operator with the following command:

```
$ oc apply -f deploy/operator.yaml
```

4. After the operator is started Percona Distribution for PostgreSQL can be created at any time with the following command:

```
$ oc apply -f deploy/cr.yaml
```

Creation process will take some time. The process is over when both operator and replica set pod have reached their Running status:

```
$ oc get pods
NAME                                              READY   STATUS      RESTARTS   AGE
backrest-backup-cluster1-j275w                    0/1     Completed   0          10m
```

(continues on next page)

(continued from previous page)

```
cluster1-85486d645f-gpxzb                    1/1     Running   0          10m
cluster1-backrest-shared-repo-6495464548-c8wvl   1/1     Running   0          10m
cluster1-pgbouncer-fc45869f7-s86rf           1/1     Running   0          10m
pgo-deploy-rhv6k                             0/1     Completed 0          5m
postgres-operator-8646c68b57-z8m62           4/4     Running   1          5m
```

5. During previous steps, the Operator has generated several secrets, including the password for the `pguser` user, which you will need to access the cluster.

   Use `oc get secrets` command to see the list of Secrets objects (by default Secrets object you are interested in has `cluster1-pguser-secret` name). Then `kubectl get secret cluster1-pguser-secret -o yaml` will return the YAML file with generated secrets, including the password which should look as follows:

```
...
data:
  ...
  password: cGd1c2VyX3Bhc3N3b3JkCg==
```

   Here the actual password is base64-encoded, and `echo 'cGd1c2VyX3Bhc3N3b3JkCg==' | base64 --decode` will bring it back to a human-readable form (in this example it will be a `pguser_password` string).

6. Check connectivity to newly created cluster

```
$ oc run -i --rm --tty pg-client --image=perconalab/percona-distribution-
↪postgresql:13.2 --restart=Never -- bash -il
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer -p␣
↪5432 -U pguser pgdb
```

   This command will connect you to the PostgreSQL interactive terminal.

```
psql (13.2)
Type "help" for help.
pgdb=>
```

CHAPTER

FIVE

INSTALL PERCONA DISTRIBUTION FOR POSTGRESQL ON GOOGLE KUBERNETES ENGINE (GKE)

Following steps will allow you to install the Operator and use it to manage Percona Distribution for PostgreSQL with the Google Kubernetes Engine. The document assumes some experience with Google Kubernetes Engine (GKE). For more information on the GKE, see the Kubernetes Engine Quickstart.

5.1 Prerequisites

All commands from this quickstart can be run either in the **Google Cloud shell** or in **your local shell**.

To use *Google Cloud shell*, you need nothing but a modern web browser.

If you would like to use *your local shell*, install the following:

1. gcloud. This tool is part of the Google Cloud SDK. To install it, select your operating system on the official Google Cloud SDK documentation page and then follow the instructions.
2. kubectl. It is the Kubernetes command-line tool you will use to manage and deploy applications. To install the tool, run the following command:

```
$ gcloud auth login
$ gcloud components install kubectl
```

5.2 Configuring default settings for the cluster

You can configure the settings using the `gcloud` tool. You can run it either in the Cloud Shell or in your local shell (if you have installed Google Cloud SDK locally on the previous step). The following command will create a cluster named `my-cluster-1`:

```
$ gcloud container clusters create cluster-1 --project <project name> --zone us-central1-
↪a --cluster-version {{{gkerecommended}}} --machine-type n1-standard-4 --num-nodes=3
```

Note: You must edit the following command and other command-line statements to replace the `<project name>` placeholder with your project name. You may also be required to edit the *zone location*, which is set to `us-central1` in the above example. Other parameters specify that we are creating a cluster with 3 nodes and with machine type of 4 vCPUs and 45 GB memory.

You may wait a few minutes for the cluster to be generated, and then you will see it listed in the Google Cloud console (select *Kubernetes Engine* → *Clusters* in the left menu panel):

Now you should configure the command-line access to your newly created cluster to make `kubectl` be able to use it.

In the Google Cloud Console, select your cluster and then click the *Connect* shown on the above image. You will see the connect statement configures command-line access. After you have edited the statement, you may run the command in your local shell:

```
$ gcloud container clusters get-credentials cluster-1 --zone us-central1-a --project
↪<project name>
```

5.3 Installing the Operator

1. First of all, use your Cloud Identity and Access Management (Cloud IAM) to control access to the cluster. The following command will give you the ability to create Roles and RoleBindings:

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

   The return statement confirms the creation:

   ```
   clusterrolebinding.rbac.authorization.k8s.io/cluster-admin-binding created
   ```

2. Use the following `git clone` command to download the correct branch of the percona-postgresql-operator repository:

   ```
   git clone -b v1.0.0 https://github.com/percona/percona-postgresql-operator
   cd percona-postgresql-operator
   ```

3. The next thing to do is to add the `pgo` namespace to Kubernetes, not forgetting to set the correspondent context for further steps:

   ```
   $ kubectl create namespace pgo
   $ kubectl config set-context $(kubectl config current-context) --namespace=pgo
   ```

   Note: To use different namespace, you should edit *all occurrences* of the `namespace: pgo` line in both `deploy/cr.yaml` and `deploy/operator.yaml` configuration files.

4. Deploy the operator with the following command:

   ```
   $ kubectl apply -f deploy/operator.yaml
   ```

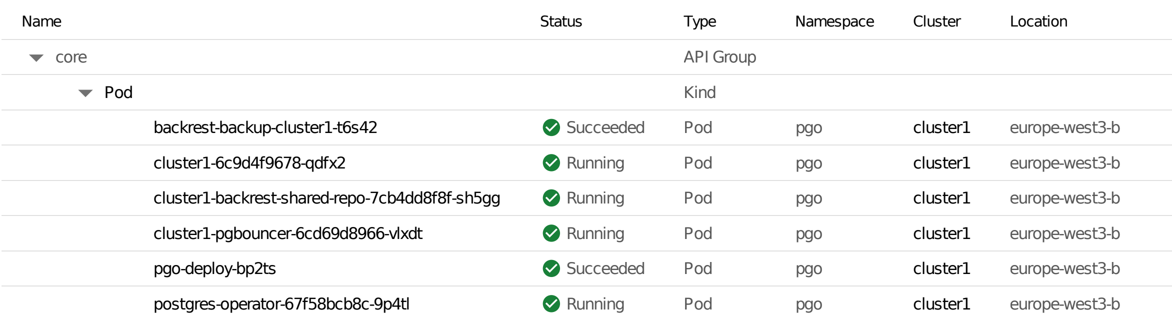
5. After the operator is started Percona Distribution for PostgreSQL can be created at any time with the following commands:

   ```
   $ kubectl apply -f deploy/cr.yaml
   ```

   Creation process will take some time. The process is over when the Operator and PostgreSQL Pods have reached their Running status:

```
$ kubectl get pods
NAME                                              READY   STATUS      RESTARTS   AGE
backrest-backup-cluster1-4nq2x                    0/1     Completed   0          10m
cluster1-6c9d4f9678-qdfx2                         1/1     Running     0          10m
cluster1-backrest-shared-repo-7cb4dd8f8f-sh5gg    1/1     Running     0          10m
cluster1-pgbouncer-6cd69d8966-vlxdt               1/1     Running     0          10m
pgo-deploy-bp2ts                                  0/1     Completed   0          5m
postgres-operator-67f58bcb8c-9p4tl                4/4     Running     1          5m
```

Also, you can see the same information when browsing Pods of your cluster in Google Cloud console via the *Object Browser*:



Name	Status	Type	Namespace	Cluster	Location
core		API Group			
Pod		Kind			
backrest-backup-cluster1-t6s42	Succeeded	Pod	pgo	cluster1	europe-west3-b
cluster1-6c9d4f9678-qdfx2	Running	Pod	pgo	cluster1	europe-west3-b
cluster1-backrest-shared-repo-7cb4dd8f8f-sh5gg	Running	Pod	pgo	cluster1	europe-west3-b
cluster1-pgbouncer-6cd69d8966-vlxdt	Running	Pod	pgo	cluster1	europe-west3-b
pgo-deploy-bp2ts	Succeeded	Pod	pgo	cluster1	europe-west3-b
postgres-operator-67f58bcb8c-9p4tl	Running	Pod	pgo	cluster1	europe-west3-b

6. During previous steps, the Operator has generated several secrets, including the password for the `pguser` user, which you will need to access the cluster.

   Use `kubectl get secrets` command to see the list of Secrets objects (by default Secrets object you are interested in has `cluster1-pguser-secret` name). Then `kubectl get secret cluster1-pguser-secret -o yaml` will return the YAML file with generated secrets, including the password which should look as follows:

```yaml
...
data:
  ...
  password: cGd1c2VyX3Bhc3N3b3JkCg==
```

   Here the actual password is base64-encoded, and `echo 'cGd1c2VyX3Bhc3N3b3JkCg==' | base64 --decode` will bring it back to a human-readable form (in this example it will be a `pguser_password` string).

7. Check connectivity to newly created cluster

```
$ kubectl run -i --rm --tty pg-client --image=perconalab/percona-distribution-
↪postgresql:13.2 --restart=Never -- bash -il
[postgres@pg-client /]$ PGPASSWORD='pguser_password' psql -h cluster1-pgbouncer -p␣
↪5432 -U pguser pgdb
```

   This command will connect you to the PostgreSQL interactive terminal.

```
psql (13.2)
Type "help" for help.
pgdb=>
```

CHAPTER

SIX

INSTALL PERCONA DISTRIBUTION FOR POSTGRESQL USING HELM

Helm is the package manager for Kubernetes. Percona Helm charts can be found in percona/percona-helm-charts repository in Github.

6.1 Pre-requisites

Install Helm following its official installation instructions.

Note: Helm v3 is needed to run the following steps.

6.2 Installation

1. Add the Percona's Helm charts repository and make your Helm client up to date with it:

```
$ helm repo add percona https://percona.github.io/percona-helm-charts/
$ helm repo update
```

2. Install the Percona Distribution for PostgreSQL Operator:

```
$ helm install my-operator percona/pg-operator --version 1.0.0
```

The `my-operator` parameter in the above example is the name of a new release object which is created for the Operator when you install its Helm chart (use any name you like).

Note: If nothing explicitly specified, `helm install` command will work with `default` namespace. To use different namespace, provide it with the following additional parameter: `--namespace my-namespace`.

3. Install PostgreSQL:

```
$ helm install my-db percona/pg-db --version 1.0.0 --namespace my-namespace
```

The `my-db` parameter in the above example is the name of a new release object which is created for the Percona Distribution for PostgreSQL when you install its Helm chart (use any name you like).

6.3 Installing Percona Distribution for PostgreSQL with customized parameters

The command above installs Percona Distribution for PostgreSQL with *default parameters*. Custom options can be passed to a `helm install` command as a `--set key=value[,key=value]` argument. The options passed with a chart can be any of the Operator's *Custom Resource options*.

The following example will deploy a Percona Distribution for PostgreSQL Cluster in the `pgdb` namespace, with enabled Percona Monitoring and Management (PMM) and 20 Gi storage for a Primary PostgreSQL node:

```
$ helm install my-db percona/pg-db --namespace pgdb \
  --set pgPrimary.volumeSpec.size=20Gi \
  --set pmm.enabled=true
```

Part III

Configuration and Management

CHAPTER SEVEN

PROVIDING BACKUPS

The Operator allows doing backups in two ways. *Scheduled backups* are configured in the deploy/cr.yaml file to be executed automatically in proper time. *On-demand backups* can be done manually at any moment.

The Operator uses the open source pgBackRest backup and restore utility. A special *pgBackRest repository* is created by the Operator along with creating a new PostgreSQL cluster to facilitate the usage of the pgBackRest features in it.

The Operator can store PostgreSQL backups on Amazon S3, any S3-compatible storage and Google Cloud Storage outside the Kubernetes cluster. Storing backups on Persistent Volume attached to the pgBackRest Pod is also possible. At PostgreSQL cluster creation time, you can specify a specific Storage Class for the pgBackRest repository. Additionally, you can also specify the type of the pgBackRest repository that can be used for backups:

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

The pgBackRest repository consists of the following Kubernetes objects:

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

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

The PostgreSQL Operator supports three types of pgBackRest backups:

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

The Operator also supports setting pgBackRest retention policies for backups. Backup retention can be controlled by the following pgBackRest options:

- `--repo1-retention-full` the number of full backups to retain,
- `--repo1-retention-diff` the number of differential backups to retain,
- `--repo1-retention-archive` how many sets of write-ahead log archives to retain alongside the full and differential backups that are retained.

You can set both backups type and retention policy when *Making on-demand backup*.

Also you should first configure the backup storage in the `deploy/cr.yaml` configuration file to have backups enabled.

7.1 Configuring the S3-compatible backup storage

In order to use S3-compatible storage for backups you need to provide some S3-related information, such as proper S3 bucket name, endpoint, etc. This information can be passed to pgBackRest via the following `deploy/cr.yaml` options in the `backup.storages` subsection:

- `bucket` specifies the AWS S3 bucket that should be utilized, for example `my-postgresql-backups-example`,
- `endpointUrl` specifies the S3 endpoint that should be utilized, for example `s3.amazonaws.com`,
- `region` specifies the AWS S3 region that should be utilized, for example `us-east-1`,
- `uriStyle` specifies whether `host` or `path` style URIs should be utilized,
- `verifyTLS` should be set to `true` to enable TLS verification or set to `false` to disable it,
- `type` should be set to `s3`.

You also need to supply pgBackRest with base64-encoded AWS S3 key and AWS S3 key secret stored along with other sensitive information in Kubernetes Secrets (e.g. encoding needed data with the `echo "string-to-encode" | base64` command). Edit the `deploy/backup/cluster1-backrest-repo-config-secret.yaml` configuration file: set there proper cluster name, AWS S3 key, and key secret:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: <cluster-name>-backrest-repo-config
type: Opaque
data:
  aws-s3-key: <base64-encoded-AWS-S3-key>
  aws-s3-key-secret: <base64-encoded-AWS-S3-key-secret>
```

When done, create the secret as follows:

```
$ kubectl apply -f deploy/backup/cluster1-backrest-repo-config-secret.yaml
```

Finally, create or update the cluster:

```
$ kubectl apply -f deploy/cr.yaml
```

7.2 Use Google Cloud Storage for backups

You can configure Google Cloud Storage as an object store for backups similarly to *S3 storage*.

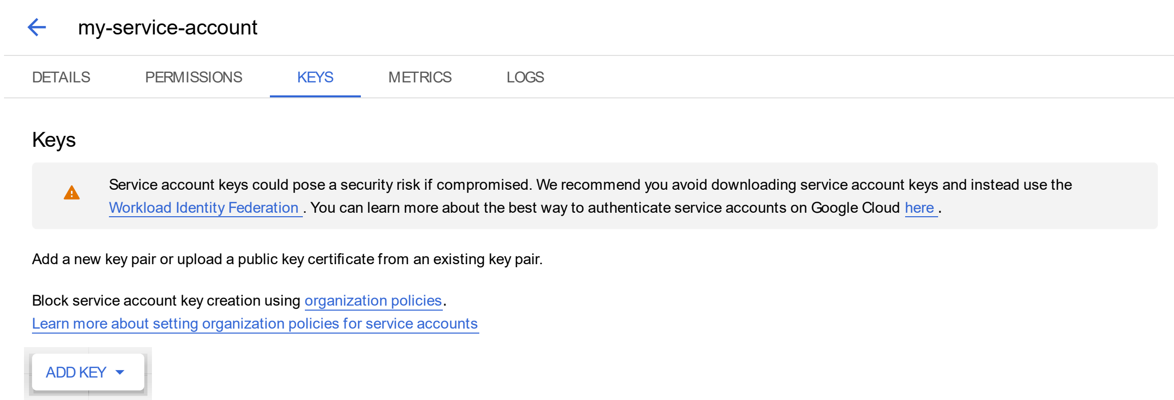
In order to use Google Cloud Storage (GCS) for backups you need to provide some GCS-related information, such as a proper GCS bucket name. This information can be passed to `pgBackRest` via the following options in the `backup.storages` subsection of the `deploy/cr.yaml` configuration file:

- `bucket` should contain the proper bucket name,
- `type` should be set to `gcs`.

The Operator will also need your service account key to access storage.

1. Create your service account key following the official Google Cloud instructions.
2. Export this key from your Google Cloud account.

   You can find your key in the Google Cloud console (select *IAM & Admin* → *Service Accounts* in the left menu panel, then click your account and open the *KEYS* tab):



   Click the *ADD KEY* button, chose *Create new key* and chose *JSON* as a key type. These actions will result in downloading a file in JSON format with your new private key and related information.

3. Now you should use a base64-encoded version of this file and to create the Kubernetes Secret. You can encode the file with the `base64 <filename>` command. When done, create the following yaml file with your cluster name and base64-encoded file contents:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: <cluster-name>-backrest-repo-config
type: Opaque
data:
  gcs-key: <base64-encoded-json-file-contents>
```

   When done, create the secret as follows:

```
$ kubectl apply -f ./my-gcs-account-secret.yaml
```

4. Finally, create or update the cluster:

```
$ kubectl apply -f deploy/cr.yaml
```

7.3 Scheduling backups

Backups schedule is defined in the backup section of the deploy/cr.yaml file. This section contains following subsections:

- storages subsection contains data needed to access the S3-compatible cloud to store backups.
- schedule subsection allows to actually schedule backups (the schedule is specified in crontab format).

Here is an example of deploy/cr.yaml which uses Amazon S3 storage for backups:

```
...
backup:
  ...
  schedule:
   - name: "sat-night-backup"
     schedule: "0 0 * * 6"
     keep: 3
     type: full
     storage: s3
  ...
```

The schedule is specified in crontab format as explained in *Custom Resource options*.

7.4 Making on-demand backup

To make an on-demand backup, the user should use a backup configuration file. The example of the backup configuration file is deploy/backup/backup.yaml.

The following keys are most important in the parameters section of this file:

- parameters.backrest-opts is the string with command line options which will be passed to pgBackRest, for example --type=full --repo1-retention-full=5,
- parameters.pg-cluster is the name of the PostgreSQL cluster to back up, for example cluster1.

When the backup options are configured, execute the actual backup command:

```
$ kubectl apply -f deploy/backup/backup.yaml
```

7.5 List existing backups

To get list of all existing backups in the pgBackrest repo, use the following command:

```
$ kubectl exec <name-of-backrest-shared-repo-pod>  -it -- pgbackrest info
```

7.6 Restore the cluster from a previously saved backup

The Operator supports the ability to perform a full restore on a PostgreSQL cluster as well as a point-in-time-recovery. There are two types of ways to restore a cluster:

- restore to a new cluster using the *pgDataSource.restoreFrom* option (and possibly, *pgDataSource.restoreOpts* for custom pgBackRest options),
- restore in-place, to an existing cluster (note that this is destructive).

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

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

To restore the previously saved backup the user should use a *backup restore* configuration file. The example of the backup configuration file is deploy/backup/restore.yaml.

The following keys are the most important in the parameters section of this file:

- `parameters.backrest-restore-from-cluster` specifies the name of a PostgreSQL cluster which will be restored. This includes stopping the database and recreating a new primary with the restored data (for example, `cluster1`),
- `parameters.backrest-restore-opts` specifies additional options for pgBackRest (for example, `--type=time --target="2021-04-16 15:13:32"` to perform a point-in-time-recovery),
- `parameters.backrest-storage-type` the type of the pgBackRest repository, (for example, `local`).

The actual restoration process can be started as follows:

```
$ kubectl apply -f deploy/backup/restore.yaml
```

To create a new PostgreSQL cluster from either the active one, or a former cluster whose pgBackRest repository still exists, use the *pgDataSource.restoreFrom* option.

The following example will create a new cluster named `cluster2` from an existing one named``cluster1``.

1. First, create the `cluster2-config-secrets.yaml` configuration file with the following content:

```yaml
apiVersion: v1
data:
  password: <base64-encoded-password-for-pguser->
  username: <base64-encoded-pguser-user-name>
kind: Secret
metadata:
  labels:
    pg-cluster: cluster2
```

(continues on next page)

(continued from previous page)

```
    vendor: crunchydata
  name: cluster2-pguser-secret
type: Opaque
---
apiVersion: v1
data:
  password: <base64-encoded-password-for-primaryuser>
  username: <base64-encoded-primaryuser-user-name>
kind: Secret
metadata:
  labels:
    pg-cluster: cluster2
    vendor: crunchydata
  name: cluster2-primaryuser-secret
type: Opaque
---
apiVersion: v1
data:
  password: <base64-encoded-password-for-postgres-user>
  username: <base64-encoded-pguser-postgres-name>
kind: Secret
metadata:
  labels:
    pg-cluster: cluster2
    vendor: crunchydata
  name: cluster2-postgres-secret
type: Opaque
```

2. When done, create the secrets as follows:

```
$ kubectl apply -f ./cluster2-config-secrets.yaml
```

3. Edit the `deploy/cr.yaml` configuration file:

   - set a new cluster name (`cluster2`),
   - set the option *pgDataSource.restoreFrom* to `cluster1`.

Create the cluster as follows:

```
$ kubectl apply -f deploy/cr.yaml
```

7.7 Delete a previously saved backup

The maximum amount of stored backups is controlled by the *backup.schedule.keep* option (only successful backups are counted). Older backups are automatically deleted, so that amount of stored backups do not exceed this number.

If you want to delete some backup manually, you need to delete both the `pgtask` object and the corresponding job itself. Deletion of the backup object can be done using the same YAML file which was used for the on-demand backup:

```
$ kubectl delete -f deploy/backup/backup.yaml
```

Deletion of the job which corresponds to the backup can be done using `kubectl delete jobs` command with the backup name:

```
$ kubectl delete jobs cluster1-backrest-full-backup
```

CHAPTER

EIGHT

UPDATE PERCONA DISTRIBUTION FOR POSTGRESQL OPERATOR

Percona Distribution for PostgreSQL Operator allows upgrades to newer versions. This includes upgrades of the Operator itself, and upgrades of the Percona Distribution for PostgreSQL.

Note: Only the incremental update to a nearest minor version of the Operator is supported. To update to a newer version, which differs from the current version by more than one, make several incremental updates sequentially.

The following steps will allow you to update both of them to current version (use the name of your cluster instead of the `<cluster-name>` placeholder).

1. Pause the cluster in order to stop all possible activities:

```
$ kubectl patch perconapgcluster/<cluster-name> --type json -p '[{"op": "replace", "path
↪": "/spec/pause", "value": true},{"op":"replace","path":"/spec/pgBouncer/size","value
↪":0}]'
```

1. Remove the old Operator and start the new Operator version:

```
$ kubectl delete \
    serviceaccounts/pgo-deployer-sa \
    clusterroles/pgo-deployer-cr \
    configmaps/pgo-deployer-cm \
    configmaps/pgo-config \
    clusterrolebindings/pgo-deployer-crb \
    jobs.batch/pgo-deploy \
    deployment/postgres-operator

$ kubectl create -f https://raw.githubusercontent.com/percona/percona-postgresql-
↪operator/v1.0.0/deploy/operator.yaml
$ kubectl wait --for=condition=Complete job/pgo-deploy --timeout=90s
```

1. Now you can switch the cluster to a new version:

```
$ kubectl patch perconapgcluster/<cluster-name> --type json -p '[{"op": "replace", "path
↪": "/spec/backup/backrestRepoImage", "value": "percona/percona-postgresql-operator:v1.
↪0.0-ppg13-pgbackrest-repo"},{"op":"replace","path":"/spec/backup/image","value":
↪"percona/percona-postgresql-operator:v1.0.0-ppg13-pgbackrest"},{"op":"replace","path":
↪"/spec/pgBadger/image","value":"percona/percona-postgresql-operator:v1.0.0-ppg13-
↪pgbadger"},{"op":"replace","path":"/spec/pgBouncer/image","value":"percona/percona-
↪postgresql-operator:v1.0.0-ppg13-pgbouncer"},{"op":"replace","path":"/spec/pgPrimary/
↪image","value":"percona/percona-postgresql-operator:v1.0.0-ppg13-postgres-ha"},{"op":
↪"replace","path":"/spec/userLabels/pgo-version","value":"v1.0.0"},{"op":"replace","path
↪":"/metadata/labels/pgo-version","value":"v1.0.0"},{"op": "replace", "path": "/spec/
↪pause", "value": false}]'
```

(continues on next page)

(continued from previous page)

```
```

Note: The above example is composed in asumption of using PostgreSQL 13 as a database management system. For PostgreSQL 12 you should change all occurrences of the `ppg13` substring to `ppg12`.

This will carry on the image update, cluster version update and the pause status switch.

1. Now you can enable the `pgbouncer` again:

```
$ kubectl patch perconapgcluster/<cluster-name --type json -p \
    '[
        {"op":"replace","path":"/spec/pgBouncer/size","value":1}
    ]'
```

Wait until the cluster is ready.

CHAPTER

NINE

PAUSE/RESUME POSTGRESQL CLUSTER

There may be external situations when it is needed to shutdown your PostgreSQL Cluster for a while and then start it back up (some works related to the maintenance of the enterprise infrastructure, etc.).

The `deploy/cr.yaml` file contains a special `spec.shutdown` key for this. Setting it to `true` gracefully stops the cluster:

```yaml
spec:
  .......
  shutdown: true
```

To start the cluster after it was shut down just revert the `spec.shutdown` key to `false`.

There is an option also to put the cluster into a read-only mode instead of completely shutting it down. This is done by a special `spec.standby` key, which should be set to `true` for read-only state or should be set to `false` for normal cluster operation:

```yaml
spec:
  .......
  standby: false
```

CHAPTER

TEN

TRANSPORT LAYER SECURITY (TLS)

The Percona Distribution for PostgreSQL Operator uses Transport Layer Security (TLS) cryptographic protocol for the following types of communication:

- Internal - communication between PostgreSQL instances in the cluster
- External - communication between the client application and the cluster

The internal certificate is also used as an authorization method for PostgreSQL Replica instances.

Currently, TLS security needs manual certificates generation.

You can also use pre-generated certificates available in the `deploy/ssl-secrets.yaml` file for test purposes, but we strongly recommend **avoiding their usage on any production system**!

The following subsections explain how to configure TLS security with the Operator yourself, as well as how to temporarily disable it if needed.

10.1 Generate certificates for the Operator

To generate certificates, follow these steps:

1. Provision a CA (Certificate authority) to generate TLS certificates,
2. Generate a CA key and certificate file with the server details,
3. Create the server TLS certificates using the CA keys, certs, and server details.

The set of commands generates certificates with the following attributes:

- `Server-pem` - Certificate
- `Server-key.pem` - the private key
- `ca.pem` - Certificate Authority

You should generate one set of certificates for external communications, and another set for internal ones.

Supposing that your cluster name is `cluster1`, you can use the following commands to generate certificates:

```
$ CLUSTER_NAME=cluster1
$ NAMESPACE=default
$ cat <<EOF | cfssl gencert -initca - | cfssljson -bare ca
{
  "CN": "*",
  "key": {
    "algo": "ecdsa",
    "size": 384
  }
}
EOF

$ cat <<EOF > ca-config.json
{
   "signing": {
     "default": {
        "expiry": "87600h",
        "usages": ["digital signature", "key encipherment", "content commitment"]
      }
   }
}
EOF

$ cat <<EOF | cfssl gencert -ca=ca.pem  -ca-key=ca-key.pem -config=./ca-config.json - |␣
↪cfssljson -bare server
{
   "hosts": [
     "localhost",
     "${CLUSTER_NAME}",
     "${CLUSTER_NAME}.${NAMESPACE}",
     "${CLUSTER_NAME}.${NAMESPACE}.svc.cluster.local",
     "${CLUSTER_NAME}-pgbouncer",
     "${CLUSTER_NAME}-pgbouncer.${NAMESPACE}",
     "${CLUSTER_NAME}-pgbouncer.${NAMESPACE}.svc.cluster.local",
     "*.${CLUSTER_NAME}",
     "*.${CLUSTER_NAME}.${NAMESPACE}",
     "*.${CLUSTER_NAME}.${NAMESPACE}.svc.cluster.local",
     "*.${CLUSTER_NAME}-pgbouncer",
     "*.${CLUSTER_NAME}-pgbouncer.${NAMESPACE}",
     "*.${CLUSTER_NAME}-pgbouncer.${NAMESPACE}.svc.cluster.local"
   ],
   "CN": "${CLUSTER_NAME}",
   "key": {
     "algo": "ecdsa",
     "size": 384
   }
}
EOF

$ kubectl create secret generic ${CLUSTER_NAME}-ssl-ca --from-file=ca.crt=ca.pem
$ kubectl create secret tls  ${CLUSTER_NAME}-ssl-keypair --cert=server.pem --key=server-
↪key.pem
```

When certificates are generated, set the following keys in the `deploy/cr.yaml` configuration file:

- `spec.sslCA` key should contain the name of the secret with TLS CA used for both connection encryption (external traffic), and replication (internal traffic),
- `spec.sslSecretName` key should contain the name of the secret created to encrypt **external** communications,
- `spec.secrets.sslReplicationSecretName` key should contain the name of the secret created to encrypt **internal** communications,
- `spec.tlsOnly` key should be set to `true` if you want to disable unencrypted communications.

Don't forget to apply changes as usual:

```
$ kubectl apply -f deploy/cr.yaml
```

10.1.1 Check connectivity to the cluster

You can check TLS communication with use of the `psql`, the standart interactive terminal-based front-end to PostgreSQL. The following command will spawn a new `pg-client` container, which includes needed command and can be used for the check (use your real cluster name instead of the `<cluster-name>` placeholder):

```
$ cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pg-client
spec:
  replicas: 1
  selector:
    matchLabels:
      name: pg-client
  template:
    metadata:
      labels:
        name: pg-client
    spec:
      containers:
        - name: pg-client
          image: perconalab/percona-distribution-postgresql:13.2
          imagePullPolicy: Always
          command:
          - sleep
          args:
          - "100500"
          volumeMounts:
            - name: ca
              mountPath: "/tmp/tls"
      volumes:
      - name: ca
        secret:
          secretName: <cluster_name>-ssl-ca
          items:
          - key: ca.crt
            path: ca.crt
```

(continues on next page)

(continued from previous page)

```
            mode: 0777
EOF
```

Now get shell access to the newly created container, and launch the PostgreSQL interactive terminal to check connectivity over the encrypted channel (please use real cluster-name, PostgreSQL user login and password):

```
$ kubectl exec -it deployment/pg-client -- bash -il
[postgres@pg-client /]$ PGSSLMODE=verify-ca PGSSLROOTCERT=/tmp/tls/ca.crt psql postgres:/
↪/<postgresql-user>:<postgresql-password>@<cluster-name>-pgbouncer.<namespace>.svc.
↪cluster.local
```

Now you should see the prompt of PostgreSQL interactive terminal:

```
psql (13.2)
Type "help" for help.
pgdb=>
```

10.2 Run Percona Distribution for PostgreSQL without TLS

Omitting TLS is also possible, but we recommend that you run your cluster with the TLS protocol enabled.

To disable TLS protocol (e.g. for demonstration purposes) set the `spec.tlsOnly` key to `false`, and and make sure that there are no certificate secrets configured in the ``deploy/cr.yaml` file.

CHAPTER

ELEVEN

MONITORING

Percona Monitoring and Management (PMM) provides an excellent solution to monitor Percona Distribution for PostgreSQL.

Note: Only PMM 2.x versions are supported by the Operator.

PMM is a client/server application. PMM Client runs on each node with the database you wish to monitor: it collects needed metrics and sends gathered data to PMM Server. As a user, you connect to PMM Server to see database metrics on a number of dashboards.

That's why PMM Server and PMM Client need to be installed separately.

11.1 Installing the PMM Server

PMM Server runs as a *Docker image*, a *virtual appliance*, or on an *AWS instance*. Please refer to the official PMM documentation for the installation instructions.

11.2 Installing the PMM Client

The following steps are needed for the PMM client installation in your Kubernetes-based environment:

1. The PMM client installation is initiated by updating the `pmm` section in the deploy/cr.yaml file.
   - set `pmm.enabled=true`
   - set the `pmm.serverHost` key to your PMM Server hostname,
   - check that the `serverUser` key contains your PMM Server user name (`admin` by default),
   - make sure the `pmmserver` key in the deploy/pmm-secret.yaml secrets file contains the password specified for the PMM Server during its installation.

     Apply changes with the `kubectl apply -f deploy/pmm-secret.yaml` command.

     Note: You use `deploy/pmm-secret.yaml` file to *create* Secrets Object. The file contains all values for each key/value pair in a convenient plain text format. But the resulting Secrets contain passwords stored as base64-encoded strings. If you want to *update* password field, you'll need to encode the value into base64 format. To do this, you can run `echo -n "password" | base64` in your local shell to get valid values. For example, setting the PMM Server user's password to *new_password*` in the `cluster1-pmm-secret` object can be done with the following command:

```
kubectl patch secret/cluster1-pmm-secret -p '{"data":{"pmmserver": '$(echo -n␣
↪new_password | base64)'}}'
```

When done, apply the edited `deploy/cr.yaml` file:

```
$ kubectl apply -f deploy/cr.yaml
```

2. Check that corresponding Pods are not in a cycle of stopping and restarting. This cycle occurs if there are errors on the previous steps:

```
$ kubectl get pods
$ kubectl logs cluster1-7b7f7898d5-7f5pz -c pmm-client
```

3. Now you can access PMM via *https* in a web browser, with the login/password authentication, and the browser is configured to show Percona Distribution for PostgreSQL metrics.

Part IV

Reference

CHAPTER TWELVE

CUSTOM RESOURCE OPTIONS

The Cluster is configured via the deploy/cr.yaml file.

The metadata part of this file contains the following keys:

- `name` (`cluster1` by default) sets the name of your Percona Distribution for PostgreSQL Cluster; it should include only URL-compatible characters, not exceed 22 characters, start with an alphabetic character, and end with an alphanumeric character;

The spec part of the deploy/cr.yaml file contains the following sections:

Key	Value type	Default	Description
pause	boolean	`false`	Pause/resume: setting it to `true` gracefully stops the cluster, and setting it to `false` after shut down starts the cluster back.
walStorage	*subdoc*		Write-ahead Log Storage Section
pmm	*subdoc*		Percona Monitoring and Management section
backup	*subdoc*		Section to configure backups and pgBackRest
pgBouncer	*subdoc*		The pgBouncer connection pooler section
pgReplicas	*subdoc*		Section required to manage the replicas within a PostgreSQL cluster
pgBadger	*subdoc*		The pgBadger PostgreSQL log analyzer section

Key	database
Value	string
Example	`pgdb`
Description	The name of a database that the PostgreSQL user can log into after the PostgreSQL cluster is created
Key	disableAutofail
Value	boolean
Example	`false`
Description	Turns high availability on or off. By default, every cluster can have high availability if there is at least one replica
Key	tlsOnly
Value	boolean
Example	`false`
Description	Enforece Operator to use only Transport Layer Security (TLS) for both internal and external communications

continues on next page

Table 1 – continued from previous page

Key	sslCA
Value	string
Example	`cluster1-ssl-ca`
Description	The name of the secret with TLS CA used for both connection encryption (external traffic), and replication (internal traffic)
Key	sslSecretName
Value	string
Example	`cluster1-ssl-keypair`
Description	The name of the secret created to encrypt external communications
Key	sslReplicationSecretName
Value	string
Example	`cluster1-ssl-keypair"`
Description	The name of the secret created to encrypt internal communications
Key	keepData
Value	boolean
Example	`true`
Description	If `true`, PVCs will be kept after the cluster deletion
Key	keepBackups
Value	boolean
Example	`true`
Description	If `true`, local backups will be kept after the cluster deletion
Key	pgDataSource.restoreFrom
Value	string
Example	`""`
Description	The name of a data source PostgreSQL cluster, which is used to *restore backup to a a new cluster*
Key	pgDataSource.restoreOpts
Value	string
Example	`""`
Description	Custom pgBackRest options to *restore backup to a a new cluster*
Key	pgPrimary.image
Value	string
Example	`perconalab/percona-postgresql-operator:main-ppg13-postgres-ha`
Description	The Docker image of the PostgreSQL Primary instance
Key	pgPrimary.volumeSpec.size
Value	int
Example	`1G`
Description	The Kubernetes PersistentVolumeClaim size for the PostgreSQL Primary storage
Key	pgPrimary.volumeSpec.accessmode
Value	string
Example	`ReadWriteOnce`
Description	The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL Primary storage

continues on next page

Table 1 – continued from previous page

Key	pgPrimary.volumeSpec.storagetype
Value	string
Example	`dynamic`
Description	Type of the PostgreSQL Primary storage provisioning: `create` (the default variant; used if storage is provisioned, e.g. using hostpath) or `dynamic` (for a dynamic storage provisioner, e.g. via a StorageClass)
Key	pgPrimary.volumeSpec.storageclass
Value	string
Example	`""`
Description	Optionally sets the Kubernetes storage class to use with the PostgreSQL Primary storage PersistentVolumeClaim
Key	pgPrimary.volumeSpec.matchLabels
Value	string
Example	`""`
Description	A PostgreSQL Primary storage label selector

12.1 Write-ahead Log Storage Section

The `walStorage` section in the deploy/cr.yaml file contains configuration options for PostgreSQL write-ahead logging.

Key	walStorage.volumeSpec.size
Value	int
Example	`1G`
Description	The Kubernetes PersistentVolumeClaim size for the PostgreSQL Write-ahead Log storage
Key	walStorage.volumeSpec.accessmode
Value	string
Example	`ReadWriteOnce`
Description	The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL Write-ahead Log storage
Key	walStorage.volumeSpec.storagetype
Value	string
Example	`dynamic`
Description	Type of the PostgreSQL Write-ahead Log storage provisioning: `create` (the default variant; used if storage is provisioned, e.g. using hostpath) or `dynamic` (for a dynamic storage provisioner, e.g. via a StorageClass)
Key	walStorage.volumeSpec.storageclass
Value	string
Example	`""`
Description	Optionally sets the Kubernetes storage class to use with the PostgreSQL Write-ahead Log storage PersistentVolumeClaim
Key	walStorage.volumeSpec.matchLabels
Value	string
Example	`""`
Description	A PostgreSQL Write-ahead Log storage label selector

12.2 Backup Section

The `backup` section in the deploy/cr.yaml file contains the following configuration options for the regular Percona Distribution for PostgreSQL backups.

Key	backup.image
Value	string
Example	`perconalab/percona-postgresql-operator:main-ppg13-pgbackrest`
Description	The Docker image for *pgBackRest*
Key	backup.backrestRepoImage
Value	string
Example	`perconalab/percona-postgresql-operator:main-ppg13-pgbackrest-repo`
Description	The Docker image for the *BackRest repository*
Key	backup.resources.requests.memory

continues on next page

Table 2 – continued from previous page

Value	int
Example	`48Mi`
Description	The Kubernetes memory requests for a pgBackRest container
Key	backup.resources.limits.cpu
Value	int
Example	`1`
Description	Kubernetes CPU limits for a pgBackRest container
Key	backup.resources.limits.memory
Value	int
Example	`64Mi`
Description	The Kubernetes memory limits for a pgBackRest container
Key	backup.volumeSpec.size
Value	int
Example	`1G`
Description	The Kubernetes PersistentVolumeClaim size for the pgBackRest Storage
Key	backup.volumeSpec.accessmode
Value	string
Example	`ReadWriteOnce`
Description	The Kubernetes PersistentVolumeClaim access modes for the pgBackRest Storage
Key	backup.volumeSpec.storagetype
Value	string
Example	`dynamic`
Description	Type of the pgBackRest storage provisioning: `create` (the default variant; used if storage is provisioned, e.g. using hostpath) or `dynamic` (for a dynamic storage provisioner, e.g. via a StorageClass)
Key	backup.volumeSpec.storageclass
Value	string
Example	`""`
Description	Optionally sets the Kubernetes storage class to use with the pgBackRest Storage PersistentVolumeClaim
Key	backup.volumeSpec.matchLabels
Value	string
Example	`""`
Description	A pgBackRest storage label selector
Key	backup.storages.<storage-name>.type
Value	string
Example	`s3`
Description	Type of the storage used for backups
Key	backup.storages.<storage-name>.endpointURL
Value	string
Example	`minio-gateway-svc:9000`

continues on next page

Table 2 – continued from previous page

Description	The endpoint URL of the S3-compatible storage to be used for backups (not needed for the original Amazon S3 cloud)
Key	backup.storages.<storage-name>.bucket
Value	string
Example	`""`
Description	The Amazon S3 bucket or Google Cloud Storage bucket name used for backups
Key	backup.storages.<storage-name>.region
Value	boolean
Example	`us-east-1`
Description	The AWS region to use for Amazon and all S3-compatible storages
Key	backup.storages.<storage-name>.uriStyle
Value	string
Example	`path`
Description	Optional parameter that specifies if pgBackRest should use the path or host S3 URI style
Key	backup.storages.<storage-name>.verifyTLS
Value	boolean
Example	`false`
Description	Enables or disables TLS verification for pgBackRest
Key	backup.storageTypes
Value	array
Example	`[ "s3" ]`
Description	The backup storage types for the pgBackRest repository
Key	backup.repoPath
Value	string
Example	`""`
Description	Custom path for pgBackRest repository backups
Key	backup.schedule.name
Value	string
Example	`sat-night-backup`
Description	The backup name
Key	backup.schedule.schedule
Value	string
Example	`0 0 * * 6`
Description	Scheduled time to make a backup specified in the crontab format
Key	backup.schedule.keep
Value	int
Example	`3`
Description	The amount of most recent backups to store. Older backups are automatically deleted. Set `keep` to zero or completely remove it to disable automatic deletion of backups
Key	backup.schedule.type

continues on next page

Table 2 – continued from previous page

Value	string
Example	`full`
Description	The *type* of the pgBackRest backup
Key	backup.schedule.storage
Value	string
Example	`local`
Description | The *type* of the pgBackRest repository	

12.3 PMM Section

The `pmm` section in the deploy/cr.yaml file contains configuration options for Percona Monitoring and Management.

Key	pmm.enabled
Value	boolean
Example	`false`
Description	Enables or disables monitoring Percona Distribution for PostgreSQL cluster with PMM
Key	pmm.image
Value	string
Example	`percona/pmm-client:2.21.0`
Description	Percona Monitoring and Management (PMM) Client Docker image
Key	pmm.serverHost
Value	string
Example	`monitoring-service`
Description	Address of the PMM Server to collect data from the cluster
Key	pmm.serverUser
Value	string
Example	`admin`
Description	The PMM Server User. The PMM Server password should be configured using Secrets
Key	pmm.pmmSecret
Value	string
Example	`cluster1-pmm-secret`
Description	Name of the Kubernetes Secret object for the PMM Server password
Key	pmm.resources.requests.memory
Value	string
Example	`200M`
Description	The Kubernetes memory requests for a PMM container
Key	pmm.resources.requests.cpu
Value	string
Example	`500m`
Description	Kubernetes CPU requests for a PMM container

continues on next page

Table 3 – continued from previous page

Key	pmm.resources.limits.cpu
Value	string
Example	`500m`
Description	Kubernetes CPU limits for a PMM container
Key	pmm.resources.limits.memory
Value	string
Example	`200M`
Description	The Kubernetes memory limits for a PMM container

12.4 pgBouncer Section

The `pgBouncer` section in the deploy/cr.yaml file contains configuration options for the pgBouncer connection pooler for PostgreSQL.

Key	pgBouncer.image
Value	string
Example	`perconalab/percona-postgresql-operator:main-ppg13-pgbouncer`
Description	Docker image for the pgBouncer connection pooler
Key	pgBouncer.size
Value	int
Example	`1G`
Description	The number of the pgBouncer Pods to provide connection pooling
Key	pgBouncer.resources.requests.cpu
Value	int
Example	`1`
Description	Kubernetes CPU requests for a pgBouncer container
Key	pgBouncer.resources.requests.memory
Value	int
Example	`128Mi`
Description	The Kubernetes memory requests for a pgBouncer container
Key	pgBouncer.resources.limits.cpu
Value	int
Example	`2`
Description	Kubernetes CPU limits for a pgBouncer container
Key	pgBouncer.resources.limits.memory
Value	int
Example	`512Mi`
Description	The Kubernetes memory limits for a pgBouncer container
Key	pgBouncer.expose.serviceType
Value	string

continues on next page

Table 4 – continued from previous page

Example	`ClusterIP`
Description	Specifies the type of Kubernetes Service for pgBouncer
Key	pgBouncer.expose.loadBalancerSourceRanges
Value	string
Example	`"10.0.0.0/8"`
Description	The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations)
Key	pgBouncer.expose.annotations
Value	label
Example	`pg-cluster-annot: cluster1`
Description	The Kubernetes annotations metadata for pgBouncer
Key	pgBouncer.expose.labels
Value	label
Example	`pg-cluster-label: cluster1`
Description	Set labels for the pgBouncer Service

12.5 pgReplicas Section

The `pgReplicas` section in the deploy/cr.yaml file stores information required to manage the replicas within a PostgreSQL cluster.

Key	pgReplicas.<replica-name>.size
Value	int
Example	`1G`
Description	The number of the PostgreSQL Replica Pods
Key	pgReplicas.<replica-name>.resources.requests.cpu
Value	int
Example	`1`
Description	Kubernetes CPU requests for a PostgreSQL Replica container
Key	pgReplicas.<replica-name>.resources.requests.memory
Value	int
Example	`128Mi`
Description	The Kubernetes memory requests for a PostgreSQL Replica container
Key	pgReplicas.<replica-name>.resources.limits.cpu
Value	int
Example	`2`
Description	Kubernetes CPU limits for a PostgreSQL Replica container
Key	pgReplicas.<replica-name>.resources.limits.memory
Value	int
Example	`512Mi`
Description	The Kubernetes memory limits for a PostgreSQL Replica container

continues on next page

Table 5 – continued from previous page

Key	pgReplicas.<replica-name>.volumeSpec.accessmode
Value	string
Example	`ReadWriteOnce`
Description	The Kubernetes PersistentVolumeClaim access modes for the PostgreSQL Replica storage
Key	pgReplicas.<replica-name>.volumeSpec.size
Value	int
Example	`1G`
Description	The Kubernetes PersistentVolumeClaim size for the PostgreSQL Replica storage
Key	pgReplicas.<replica-name>.volumeSpec.storagetype
Value	string
Example	`dynamic`
Description	Type of the PostgreSQL Replica storage provisioning: `create` (the default variant; used if storage is provisioned, e.g. using hostpath) or `dynamic` (for a dynamic storage provisioner, e.g. via a StorageClass)
Key	pgReplicas.<replica-name>.volumeSpec.storageclass
Value	string
Example	`standard`
Description	Optionally sets the Kubernetes storage class to use with the PostgreSQL Replica storage PersistentVolumeClaim
Key	pgReplicas.<replica-name>.volumeSpec.matchLabels
Value	string
Example	`""`
Description	A PostgreSQL Replica storage label selector
Key	pgReplicas.<replica-name>.labels
Value	label
Example	`pg-cluster-label: cluster1`
Description	Set labels for PostgreSQL Replica Pods
Key	pgReplicas.<replica-name>.annotations
Value	label
Example	`pg-cluster-annot: cluster1-1`
Description	The Kubernetes annotations metadata for PostgreSQL Replica
Key	pgReplicas.<replica-name>.expose.serviceType
Value	string
Example	`ClusterIP`
Description	Specifies the type of Kubernetes Service for for PostgreSQL Replica
Key	pgReplicas.<replica-name>.expose.loadBalancerSourceRanges
Value	string
Example	`"10.0.0.0/8"`
Description	The range of client IP addresses from which the load balancer should be reachable (if not set, there is no limitations)

continues on next page

Table 5 – continued from previous page

Key	pgReplicas.<replica-name>.expose.annotations
Value	label
Example	`pg-cluster-annot: cluster1`
Description	The Kubernetes annotations metadata for PostgreSQL Replica
Key	pgReplicas.<replica-name>.expose.labels
Value	label
Example	`pg-cluster-label: cluster1`
Description	Set labels for the PostgreSQL Replica Service

12.6 pgBadger Section

The `pgBadger` section in the deploy/cr.yaml file contains configuration options for the pgBadger PostgreSQL log analyzer.

Key	pgBadger.enabled
Value	boolean
Example	`false`
Description	Enables or disables the pgBadger PostgreSQL log analyzer
Key	pgBadger.image
Value	string
Example	`perconalab/percona-postgresql-operator:main-ppg13-pgbadger`
Description	pgBadger PostgreSQL log analyzer Docker image
Key	pgBadger.port
Value	int
Example	`10000`
Description	The port number for pgBadger

CHAPTER

THIRTEEN

PERCONA CERTIFIED IMAGES

Following table presents Percona's certified docker images to be used with the Percona Distribution for PostgreSQL Operator:

Image	Digest
percona/percona-postgresql-operator:1.0.0-pgo-deployer	sha256:48a16b95a307542282ec7fe6c99dd923db00b7b528dbcb9779ac4240fda7aa35
percona/percona-postgresql-operator:1.0.0-postgres-operator	sha256:cbeb03fd1ca5c57124272993c591bef8970a71a5ad128ce3f1d3af556382ca7e
percona/percona-postgresql-operator:1.0.0-pgo-scheduler	sha256:460508bcfde49b1c80b776b84269c7185d9929ee39164218591849c454bda3d2
percona/percona-postgresql-operator:1.0.0-pgo-rmdata	sha256:3b370aa297742a0c13650457b32ab9e698c24409d287887afdec6a6ccdeb1628
percona/percona-postgresql-operator:1.0.0-pgo-event	sha256:c4f9a3aa8039c69099fe0f65b4504e2edeb4eb8530df8e7491d91f663e0a1a1f
percona/percona-postgresql-operator:1.0.0-pgo-apiserver	sha256:3ef498def581f5944e09c6e964001f260e6acdb8b2ae52945e3929b7d5d07891
percona/percona-postgresql-operator:1.0.0-ppg12-pgbadger	sha256:476d3ca780878320632b5eef3c4b0fc9855d4226891f3b11c5fe91fc67192d84
percona/percona-postgresql-operator:1.0.0-ppg13-pgbadger	sha256:fe125347e73542a96aeddaba20de13d30933cc3a7835d29484491d2f86c157a4
percona/percona-postgresql-operator:1.0.0-ppg12-postgres-ha	sha256:ee234eef5f5e16f2eeaa894a1ab4593f52dcdc7507bd64245747d6617756ca26
percona/percona-postgresql-operator:1.0.0-ppg13-postgres-ha	sha256:61ee6450cf35ff6267ba766ec3ffe4b67cff483b449be9d5571fe5b66a4a739c
percona/percona-postgresql-operator:1.0.0-ppg12-pgbouncer	sha256:d52e4a335e9ed45ea55a9c96195d379c38bd28038cdc73df4b2645b513481c09
percona/percona-postgresql-operator:1.0.0-ppg13-pgbouncer	sha256:53d0036c9377a6af0a4e6f9271b0dd4f687b7d8968e2885c76d4add76ab01808
percona/percona-postgresql-operator:0.2.0-ppg12-pgbackrest	sha256:1d8699e1820bc656a6fcae1bb3baac3912a6c917b27a5b57a3b539a85c0f5273
percona/percona-postgresql-operator:1.0.0-ppg13-pgbackrest	sha256:d0c362c7dabfcce021a3cf4bb3a60439e991e8bc18f00c101aac6eab8e9b6743
percona/percona-postgresql-operator:1.0.0-ppg12-pgbackrest-repo	sha256:9b5d69880f6cedf5db028956d5783717786f178f1aa1e2bf25c2e20a412dabab
percona/percona-postgresql-operator:1.0.0-ppg13-pgbackrest-repo	sha256:5fd84c6fd7831284b5a48d698e57d2ce6cc2257a00cc6757408cf0321ee1df71

CHAPTER

FOURTEEN

PERCONA DISTRIBUTION FOR POSTGRESQL OPERATOR 1.0.0 RELEASE NOTES

14.1 *Percona Distribution for PostgreSQL Operator* 1.0.0

Date October 7, 2021

Installation Installing Percona Distribution for PostgreSQL Operator

Percona announces the general availability of Percona Distribution for PostgreSQL Operator 1.0.0.

The Percona Distribution for PostgreSQL Operator automates the lifecycle, simplifies deploying and managing open source PostgreSQL clusters on Kubernetes.

The Operator follows best practices for configuration and setup of the Percona Distribution for PostgreSQL. The Operator provides a consistent way to package, deploy, manage, and perform a backup and a restore for a Kubernetes application. Operators deliver automation advantages in cloud-native applications.

The advantages are the following:

- Deploy a Percona Distribution for PostgreSQL with no single point of failure and environment which can span multiple availability zones
- Modify the Percona Distribution for PostgreSQL size parameter to add or remove PostgreSQL instances
- Use single Custom Resource as a universal entry point to configure the cluster, similar to other Percona Operators
- Carry on semi-automatic upgrades of the Operator and PostgreSQL to newer versions
- Integrate with Percona Monitoring and Management (PMM) to seamlessly monitor your Percona Distribution for PostgreSQL
- Automate backups or perform on-demand backups as needed with support for performing an automatic restore
- Use cloud storage with S3-compatible APIs or Google Cloud for backups
- Use Transport Layer Security (TLS) for the replication and client traffic
- Support advanced Kubernetes features such as pod disruption budgets, node selector, constraints, tolerations, priority classes, and affinity/anti-affinity

Percona Distribution for PostgreSQL Operator is based on Postgres Operator developed by Crunchy Data.

14.1.1 Release Highlights

- It is now possible to *configure scheduled backups* following the declarative approach in the `deploy/cr.yaml` file, similar to other Percona Kubernetes Operators
- OpenShift compatibility allows *running Percona Distribution for PostgreSQL on Red Hat OpenShift Container Platform*
- For the first time, the main functionality of the Operator is covered by functional tests, which ensure the overall quality and stability

14.1.2 New Features and Improvements

- K8SPG-96: PMM Client container does not cause the crash of the whole database Pod if `pmm-agent` is not working properly
- K8SPG-86: The Operator *is now compatible* with the OpenShift platform
- K8SPG-62: Configuring *scheduled backups* through the main Custom Resource is now supported
- K8SPG-99, K8SPG-131: The Operator documentation was substantially improved, and now it covers among other things the usage of Transport Layer Security (TLS) for internal and external communications, and cluster upgrades

14.1.3 Supported Platforms

The following platforms were tested and are officially supported by Operator 1.0.0:

- OpenShift 4.6 - 4.8
- Google Kubernetes Engine (GKE) 1.17 - 1.21
- Amazon Elastic Container Service for Kubernetes (EKS) 1.21

This list only includes the platforms that the Operator is specifically tested on as a part of the release process. Other Kubernetes flavors and versions depend on the backward compatibility offered by Kubernetes itself.

14.2 *Percona Distribution for PostgreSQL Operator* 0.2.0

Date August 12, 2021

Installation Installing Percona Distribution for PostgreSQL Operator

Version 0.2.0 of the Percona Distribution for PostgreSQL Operator is a Beta release, and it is not recommended for production environments.

14.2.1 New Features and Improvements

- K8SPG-80: The Custom Resource structure was reworked to provide the same look and feel as in other Percona Operators. Read more about Custom Resource options in the *documentation* and review the default `deploy/cr.yaml` configuration file on GitHub.
- K8SPG-53: Merged upstream CrunchyData Operator v4.7.0 made it possible to use *Google Cloud Storage as an object store for backups* without using third-party tools
- K8SPG-42: There is no need to specify the name of the pgBackrest Pod in the backup manifest anymore as it is detected automatically by the Operator
- K8SPG-30: Replicas management is now performed through a main Custom Resource manifest instead of creating separate Kubernetes resources. This also adds the possibility of scaling up/scaling down replicas via the 'deploy/cr.yaml' configuration file
- K8SPG-66: Helm chart is now *officially provided with the Operator*

14.3 *Percona Distribution for PostgreSQL Operator* 0.1.0

Date May 10, 2021

Installation Installing Percona Distribution for PostgreSQL Operator

The Percona Operator is based on best practices for configuration and setup of a Percona Distribution for PostgreSQL on Kubernetes. The benefits of the Operator are many, but saving time and delivering a consistent and vetted environment is key.

Kubernetes provides users with a distributed orchestration system that automates the deployment, management, and scaling of containerized applications. The Operator extends the Kubernetes API with a new custom resource for deploying, configuring, and managing the application through the whole life cycle. You can compare the Kubernetes Operator to a System Administrator who deploys the application and watches the Kubernetes events related to it, taking administrative/operational actions when needed.

Version 0.1.0 of the Percona Distribution for PostgreSQL Operator is a tech preview release and it is not recommended for production environments.

You can install *Percona Distribution for PostgreSQL Operator* on Kubernetes, Google Kubernetes Engine (GKE), and Amazon Elastic Kubernetes Service (EKS) clusters. The Operator is based on Postgres Operator developed by Crunchy Data.

Here are the main differences between v 0.1.0 and the original Operator:

- Percona Distribution for PostgreSQL is now used as the main container image.
- It is possible to specify custom images for all components separately. For example, users can easily build and use custom images for one or several components (e.g. pgBouncer) while all other images will be the official ones. Also, users can build and use all custom images.
- All container images are reworked and simplified. They are built on Red Hat Universal Base Image (UBI) 8.
- The Operator has built-in integration with Percona Monitoring and Management v2.
- A build/test infrastructure was created, and we have started adding e2e tests to be sure that all pieces of the cluster work together as expected.
- We have phased out the `pgo` CLI tool, and the Custom Resource UX will be completely aligned with other Percona Operators in the following release.

Once Percona Operator is promoted to GA, users would be able to get the full package of services from Percona teams.

While the Operator is in its very first release, instructions on how to install and configure it are already available along with the source code hosted in our Github repository.

Help us improve our software quality by reporting any bugs you encounter using our bug tracking system.