EDB Postgres Distributed for Kubernetes

Version 1

Built at 2024-05-15T14:51:31

1 EDB Postgres Distributed for Kubernetes

EDB Postgres Distributed for Kubernetes (`PGD4K`) is an operator designed to manage EDB Postgres Distributed (PGD) workloads on Kubernetes, with traffic routed by PGD Proxy.

The main custom resource that the operator provides is called `PGDGroup` .

Architectures can also be deployed across different Kubernetes clusters.

Before you start

EDB Postgres Distributed for Kubernetes provides you with a way to deploy EDB Postgres Distributed in a Kubernetes environment. Therefore, we recommend reading the EDB Postgres Distributed documentation.

To start working with EDB Postgres Distributed for Kubernetes, read the following in the PGD documentation:

- Terminology
- PGD overview
- Choosing your architecture
- Choosing a Postgres distribution

For advanced usage and maximum customization, it's also important to be familiar with the EDB Postgres for Kubernetes documentation, as described in Architecture.

Supported Kubernetes distributions

EDB Postgres Distributed for Kubernetes is available for:

- Kubernetes version 1.23 or later through a Helm chart
- Red Hat OpenShift version 4.10 or later only through the Red Hat OpenShift certified operator

Requirements

EDB Postgres Distributed for Kubernetes requires that the Kubernetes/OpenShift clusters hosting the distributed PGD cluster were prepared by you to cater for:

- The public key infrastructure (PKI) encompassing all the Kubernetes clusters the PGD global group is spread across. mTLS is required to authenticate and authorize all nodes in the mesh topology and guarantee encrypted communication.
- Networking infrastructure across all Kubernetes clusters involved in the PGD global group to ensure that each node can communicate with each other

EDB Postgres Distributed for Kubernetes also requires Cert Manager 1.10 or later.

> **About connectivity**
>
> See Connectivity for more information.

API reference

For a list of resources provided by EDB Postgres Distributed for Kubernetes, see the API reference.

Trademarks

2 EDB Postgres for Kubernetes Release notes

The EDB Postgres Distributed for Kubernetes documentation describes the major version of EDB Postgres Distributed for Kubernetes, including minor releases and patches. The release notes provide information on what is new in each release. For new functionality introduced in a minor or patch release, the content also indicates the release that introduced the feature.

Version	Release date
1.0.0	24 Apr 2024

2.1 EDB Postgres Distributed for Kubernetes 1.0.0 release notes

Released: 24 Apr 2024

This is the first major stable release of EDB Postgres Distributed for Kubernetes, a Kubernetes operator to deploy and manage EDB Postgres Distributed clusters.

Highlights of EDB Postgres Distributed for Kubernetes 1.0.0

The operator implements the `PGDGroup` custom resource in the API group `pgd.k8s.enterprisedb.io`. You can use this resource to create and manage EDB Postgres Distributed clusters inside Kubernetes with capabilities including:

- Deployment of EDB Postgres Distributed clusters with versions 5 and later.
- Additional self-healing capability on top of that of Postgres Distributed, such as recovery and restart of failed PGD nodes.
- Defined services that allow applications to connect to the write leader of each PGD group.

> **Note**
>
> The EDB Postgres Distributed for Kubernetes operator leverages EDB Postgres for Kubernetes (PG4K) and inherits many of that project's capabilities. EDB Postgres Distributed for Kubernetes version 1.0.0 is based, specifically, on release 1.22 of PG4K. See the PG4K release notes for more details.

Features

Component	Description
PGD4K	Deployment of EDB Postgres Distributed clusters with versions 5 and later inside Kubernetes
PGD4K	Self-healing capabilities such as recovery and restart of failed PGD nodes
PGD4K	Defined services that allow applications to connect to the write leader of each PGD group
PGD4K	Implementation of Raft subgroups
PGD4K	TLS connections and client certificate authentication
PGD4K	Continuous backup to an S3-compatible object store

3 Before you start

Before you get started, it's essential that you become familiar with some terminology that's specific to Kubernetes and PGD.

Kubernetes terminology

Node : A *node* is a worker machine in Kubernetes, either virtual or physical, where all services necessary to run pods are managed by the control plane nodes.

Pod : A *pod* is the smallest computing unit that can be deployed in a Kubernetes cluster and is composed of one or more containers that share network and storage.

Service : A *service* is an abstraction that exposes as a network service an application that runs on a group of pods and standardizes important features, such as service discovery across applications, load balancing, and failover.

Secret : A *secret* is an object that's designed to store small amounts of sensitive data, such as passwords, access keys, or tokens, for use within pods.

Storage class : A *storage class* allows an administrator to define the classes of storage in a cluster, including provisioner (such as AWS EBS), reclaim policies, mount options, volume expansion, and so on.

Persistent volume : A *persistent volume* (PV) is a resource in a Kubernetes cluster that represents storage that was either manually provisioned by an administrator or dynamically provisioned by a *storage class* controller. A PV is associated with a pod using a *persistent volume claim*, and its lifecycle is independent of any pod that uses it. Normally, a PV is a network volume, especially in the public cloud. A *local persistent volume* (LPV) is a persistent volume that exists only on the particular node where the pod that uses it is running.

Persistent volume claim : A *persistent volume claim* (PVC) represents a request for storage, which might include size, access mode, or a particular storage class. Similar to how a pod consumes node resources, a PVC consumes the resources of a PV.

Namespace : A *namespace* is a logical and isolated subset of a Kubernetes cluster and can be seen as a *virtual cluster* within the wider physical cluster. Namespaces allow administrators to create separated environments based on projects, departments, teams, and so on.

RBAC : *Role-based access control (RBAC)*, also known as *role-based security*, is a method used in computer systems security to restrict access to the network and resources of a system to authorized users only. Kubernetes has a native API to control roles at the namespace and cluster level and associate them with specific resources and individuals.

CRD : A *custom resource definition* (CRD) is an extension of the Kubernetes API and allows developers to create new data types and objects, *called custom resources*.

Operator : An *operator* is a Kubernetes software extension that automates those steps that are normally performed by a human operator when managing one or more applications or given services. An operator assists Kubernetes in making sure that the resource's defined state always matches the observed one.

`kubectl` : `kubectl` is the command-line tool used to manage a Kubernetes cluster.

EDB Postgres Distributed for Kubernetes requires a Kubernetes version supported by the community. See Supported releases for details.

PGD terminology

For more information, see Terminology in the PGD documentation.

Data node : A PGD database instance.

Failover : The automated process that recognizes a failure in a highly available database cluster and takes action to connect the application to another active database.

Switchover : A planned change in connection between the application and the active database node in a cluster, typically done for maintenance.

Write leader : In always-on architectures, a node is selected as the correct connection endpoint for applications. This node is called the write leader. The write leader is selected by consensus of a quorum of data nodes.

Cloud terminology

Region : A *region* in the cloud is an isolated and independent geographic area organized in *availability zones*. Zones within a region have very little round-trip network latency.

Zone : An *availability zone* in the cloud (also known as a *zone*) is an area in a region where resources can be deployed. Usually, an availability zone corresponds to a data center or an isolated building of the same data center.

What to do next

Now that you are familiar with the terminology, you can test EDB Postgres Distributed for Kubernetes on your laptop using a local cluster before deploying the operator in your selected cloud environment.

4 Use cases

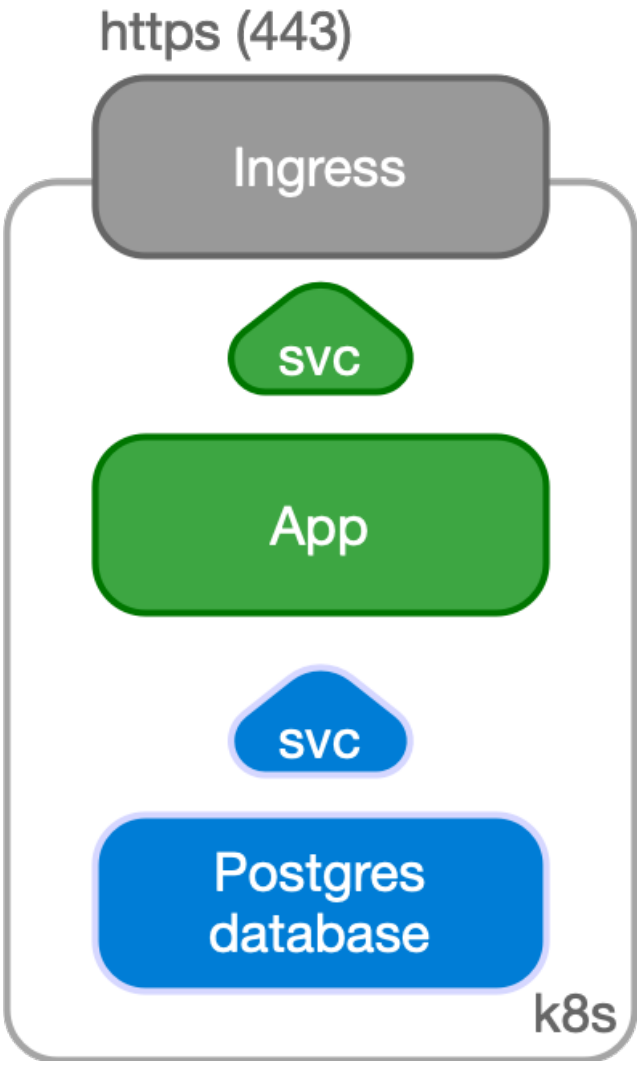
EDB Postgres Distributed for Kubernetes was designed to work with applications that reside in the same Kubernetes cluster for a full cloud native experience.

However, it might happen that, while the database can be hosted inside a Kubernetes cluster, applications can't be containerized at the same time and need to run in a traditional environment such as a VM.

The following is a summary of the basic considerations. See the EDB Postgres for Kubernetes documentation for more detail.

Case 1: Applications inside Kubernetes

In a typical situation, the application and the database run in the same namespace inside a Kubernetes cluster.



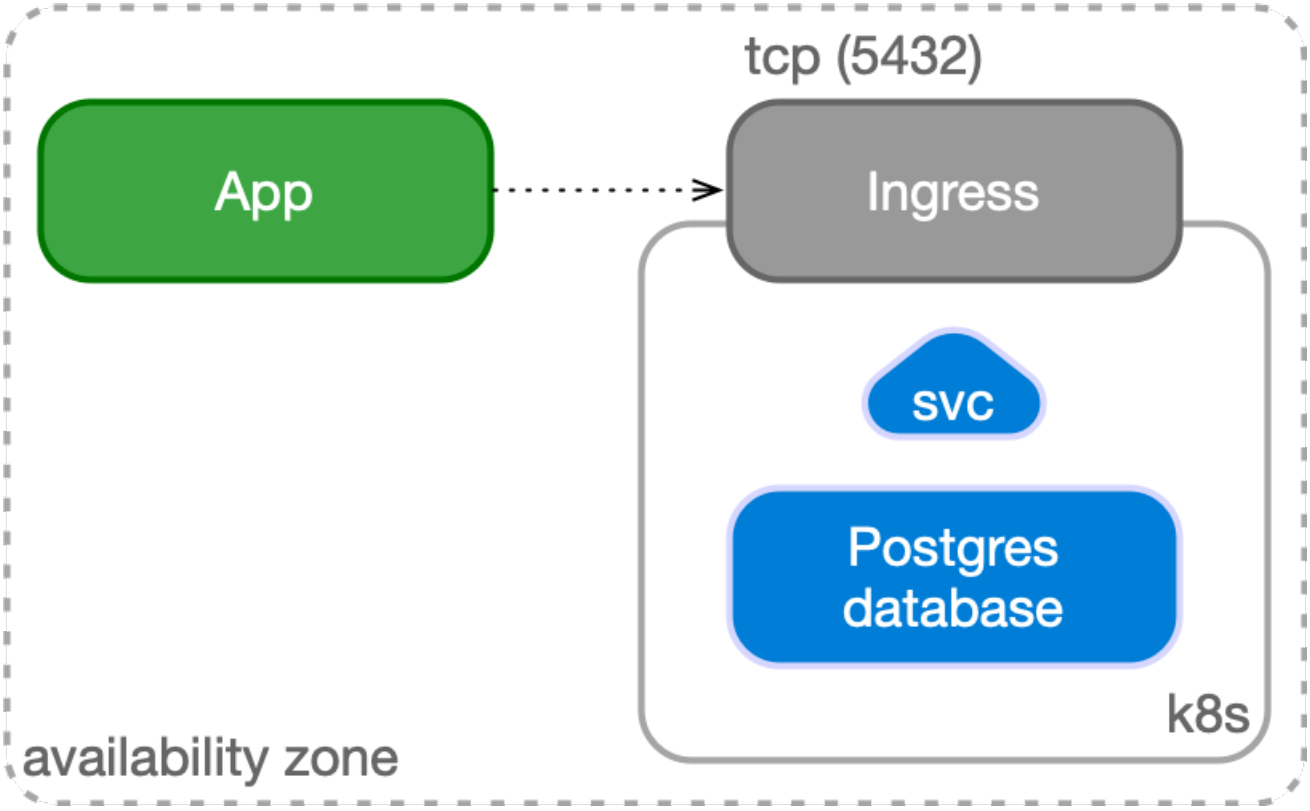
The application, normally stateless, is managed as a standard deployment, with multiple replicas spread over different Kubernetes nodes and internally exposed through a ClusterIP service.

The service is exposed externally to the end user through an Ingress and the provider's load balancer facility by way of HTTPS.

Case 2: Applications outside Kubernetes

Another possible use case is to manage your PGD database inside Kubernetes while having your applications outside of it, for example, in a virtualized environment. In this case, PGD is represented by an IP address or host name and a TCP port, corresponding to the defined Ingress resource in Kubernetes.

The application can still benefit from a TLS connection to PGD.



5 Architecture

Consider these main architectural aspects when deploying EDB Postgres Distributed in Kubernetes.

EDB Postgres Distributed for Kubernetes is a Kubernetes operator designed to deploy and manage EDB Postgres Distributed clusters running in private, public, hybrid, or multi-cloud environments.

Relationship with EDB Postgres Distributed

EDB Postgres Distributed (PGD) is a multi-master implementation of Postgres designed for high performance and availability. PGD generally requires deployment using Trusted Postgres Architect (TPA), a tool that uses Ansible to provision and deploy PGD clusters.

EDB Postgres Distributed for Kubernetes offers a different way of deploying PGD clusters, leveraging containers and Kubernetes. The advantages are that the resulting architecture:

- Is self-healing and robust.
- Is managed through declarative configuration.
- Takes advantage of the vast and growing Kubernetes ecosystem.

Relationship with EDB Postgres for Kubernetes

A PGD cluster consists of one or more *PGD groups*, each having one or more *PGD nodes*. A PGD node is a Postgres database. EDB Postgres Distributed for Kubernetes internally manages each PGD node using the `Cluster` resource as defined by EDB Postgres for Kubernetes, specifically a cluster with a single instance (that is, no replicas).

You can configure the single PostgreSQL instance created by each `Cluster` in the `.spec.cnp` section of the PGD Group spec.

In EDB Postgres Distributed for Kubernetes, as in EDB Postgres for Kubernetes, the underlying database implementation is responsible for data replication. However, it's important to note that failover and switchover work differently, entailing Raft election and nominating new write leaders. EDB Postgres for Kubernetes handles only the deployment and healing of data nodes.

Managing PGD using EDB Postgres Distributed for Kubernetes

The EDB Postgres Distributed for Kubernetes operator can manage the complete lifecycle of PGD clusters. As such, in addition to PGD nodes (represented as single-instance `Clusters`), it needs to manage other objects associated with PGD.

PGD relies on the Raft algorithm for distributed consensus to manage node metadata, specifically agreement on a *write leader*. Consensus among data nodes is also required for operations such as generating new global sequences or performing distributed DDL.

These considerations force additional actors in PGD above database nodes.

EDB Postgres Distributed for Kubernetes manages the following:

- Data nodes. A node is a database and is managed by EDB Postgres for Kubernetes, creating a `Cluster` with a single instance.
- Witness nodes are basic database instances that don't participate in data replication. Their function is to guarantee that consensus is possible in groups with an even number of data nodes or after network partitions. Witness nodes are also managed using a single-instance `Cluster` resource.
- PGD proxies act as Postgres proxies with knowledge of the write leader. PGD proxies need information from Raft to route writes to the current write leader.

Proxies and routing

PGD groups assume full mesh connectivity of PGD nodes. Each node must be able to connect to every other node using the appropriate connection string (a `libpq` -style DSN). Write operations don't need to be sent to every node. PGD takes care of replicating data after it's committed to one node.

For performance, we often recommend sending write operations mostly to a single node, the *write leader*. Raft is used to identify which node is the write leader and to hold metadata about the PGD nodes. PGD proxies are used to transparently route writes to write leaders and to quickly pivot to the new write leader in case of switchover or failover.

It's possible to configure *Raft subgroups*, each of which can maintain a separate write leader. In EDB Postgres Distributed for Kubernetes, a PGD group containing a PGD proxy comprises a Raft subgroup.

Two kinds of routing are available with PGD proxies:

- Global routing uses the top-level Raft group and maintains one global write leader.
- Local routing uses subgroups to maintain separate write leaders. Local routing is often used to achieve geographical separation of writes.

In EDB Postgres Distributed for Kubernetes, local routing is used by default, and a configuration option is available to select global routing.

For more information, see the PGD documentation of routing with Raft.

PGD architectures and high availability

EDB proposes several recommended architectures to make good use of PGD's distributed multi-master capabilities and to offer high availability.

The Always On architectures are built from either one group in a single location or two groups in two separate locations. See Choosing your architecture in the PGD documentation for more information.

Deploying PGD on Kubernetes

EDB Postgres Distributed for Kubernetes leverages Kubernetes to deploy and manage PGD clusters. As such, some adaptations are necessary to translate PGD into the Kubernetes ecosystem.

Images and operands

You can configure PGD to run one of three Postgres distributions. See the PGD documentation to understand the features of each distribution.

To function in Kubernetes, containers are provided for each Postgres distribution. These are the *operands*. In addition, the operator images are kept in those same repositories.

See EDB private image registries for details on accessing the images.

Kubernetes architecture

Some of the points of the PG4K document on Kubernetes architecture are reproduced here. See the PG4K documentation for details.

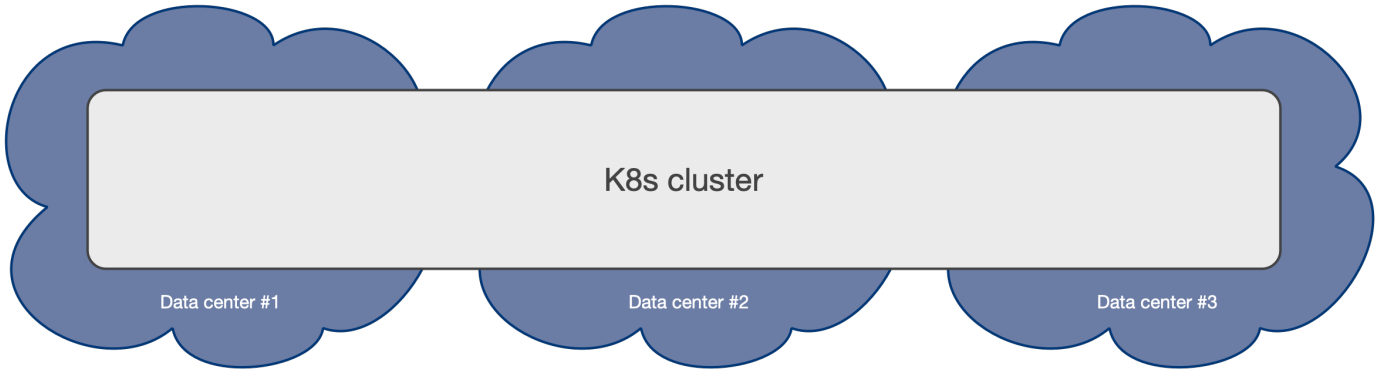
Kubernetes natively provides the possibility to span separate physical locations. These physical locations are also known as data centers, failure zones, or, more frequently, *availability zones*. They are connected to each other by way of redundant, low-latency, private network connectivity.

Being a distributed system, the recommended minimum number of availability zones for a *Kubernetes cluster* is three. This minimum makes the control plane resilient to the failure of a single zone. This means that each data center is active at any time and can run workloads simultaneously.

You can install EDB Postgres Distributed for Kubernetes in a single Kubernetes cluster or across multiple Kubernetes clusters.

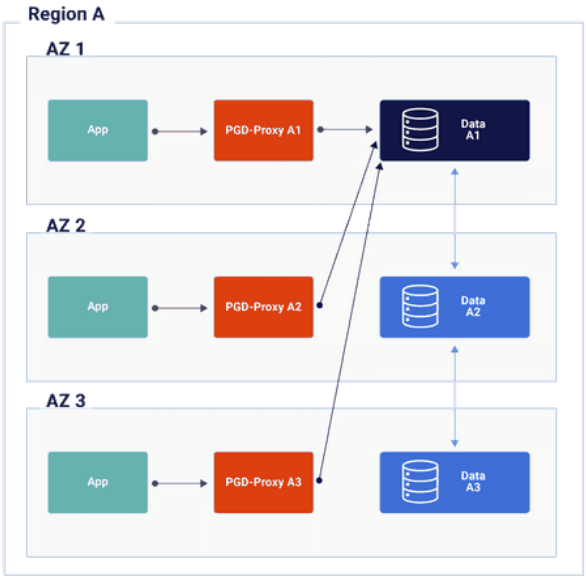
Single Kubernetes cluster

A multi-availability-zone Kubernetes architecture is typical of Kubernetes services managed by cloud providers. Such an architecture enables the EDB Postgres Distributed for Kubernetes and the EDB Postgres for Kubernetes operators to schedule workloads and nodes across availability zones, considering all zones active.



PGD clusters can be deployed in a single Kubernetes cluster and take advantage of Kubernetes availability zones to enable high-availability architectures, including the Always On recommended architectures.

You can realize the *Always On Single Location* architecture shown in Choosing your architecture in the PGD documentation on a single Kubernetes cluster with three availability zones.

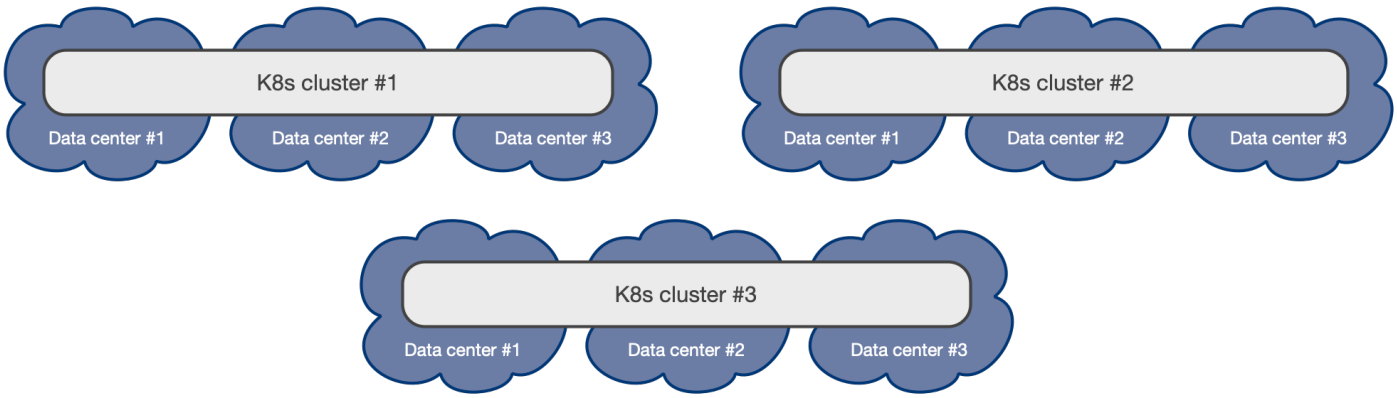


The EDB Postgres Distributed for Kubernetes operator can control the scheduling of pods (that is, which pods go to which data center) using affinity, tolerations, and node selectors, as is the case with EDB Postgres for Kubernetes. Individual scheduling controls are available for proxies as well as nodes.

See the Kubernetes documentation on scheduling, and Scheduling in the EDB Postgres for Kubernetes documentation for more information.

Multiple Kubernetes clusters

PGD clusters can also be deployed in multiple Kubernetes clusters that can reliably communicate with each other.



Always On multi-location PGD architectures can be realized on multiple Kubernetes clusters that meet the connectivity requirements.

For more information, see "Connectivity".

> **Regions and availability zones**
>
> When creating Kubernetes clusters in different regions or availability zones for cross-regional replication, ensure the clusters can communicate with each other by enabling network connectivity. Specifically, every service created with a `-node` or `-group` suffix must be discoverable by all other `-node` and `-group` services. You can achieve this by deploying a network connectivity application like Submariner on every cluster.

6 Installation

You can deploy EDB Postgres Distributed for Kubernetes using the provided Helm chart.

> **Info**
>
> For more details on the Helm chart, see the Helm chart repo documentation.

This section covers using `helm` to deploy of a set of default images with the latest available version. If want to specify a different operand or proxy image, see Identify your installation images and repositories, before continuing with the installation.

Prerequisites

- Install Helm. Follow these instructions to install it in your system.
- Install `kubectl`. See Install Tools for intructions.
- You have created at least one Kubernetes cluster.

Add the repository

Use `helm` to add the repository containing all images:

```
helm repo add edb \
  https://enterprisedb.github.io/edb-postgres-for-kubernetes-charts/
```

Deploy the images

> **Important**
>
> You need access to the private EDB repository where both the operator and operand images are stored. Access requires a valid EDB subscription plan.
>
> To identify your access credentials, see Accessing EDB private image registries.

Given that the container images for both the operator and the selected operand are in EDB's private registry, you need your credentials to enable `helm` to retrieve them.

Make sure to replace your repo and token in the following command:

```
helm upgrade --dependency-update \
  --install edb-pg4k-pgd \
  --wait \
  --namespace pgd-operator-system \
```

```
  --create-namespace \
  edb/edb-postgres-distributed-for-kubernetes \
  --set image.imageCredentials.username=<repository_name> \
  --set image.imageCredentials.password=<your_repository_token>
```

In particular:

- Set `<repository_name>` to the name of the repository, as explained in Which repository to choose?.
- Set `<your_repository_token>` to the repository token for your EDB account, as explained in How to retrieve the token.

Create a certificate issuer

Be sure to create a cert issuer before you start deploying PGD clusters. The Helm chart prompts you to do this, but in case you miss it, you can, for example, run:

```
kubectl apply -f \
  https://raw.githubusercontent.com/EnterpriseDB/edb-postgres-for-kubernetes-
charts/main/hack/samples/issuer-selfsigned.yaml
```

With the operators and a self-signed cert issuer deployed, you can start creating PGD clusters. See the Quick start for an example.

Red Hat OpenShift

If you're trying to install EDB Postgres Distributed for Kubernetes on Red Hat OpenShift, see Red Hat OpenShift, which contains information on the certified operator maintained by EDB.

7 Quick start

You can test an EDB Postgres Distributed (PGD) cluster on your laptop or computer using EDB Postgres Distributed for Kubernetes on a single local Kubernetes cluster built with Kind.

> **Warning**
>
> These instructions are only for demonstration, testing, and practice purposes and must not be used in production.

This quick start shows you how to start an EDB Postgres Distributed cluster on your local Kubernetes installation so you can experiment with it.

> **Important**
>
> To connect to the Kubernetes cluster, make sure that you have `kubectl` installed on your machine. See the Kubernetes documentation on installing `kubectl`.

Part 1 - Set up the local Kubernetes playground

Install Kind, a tool for running local Kubernetes clusters using Docker container nodes. (Kind stands for Kubernetes IN Docker.) If you already have access to a Kubernetes cluster, you can skip to Part 2.

Install Kind on your environment following the instructions in Kind Quick Start. Then, create a Kubernetes cluster:

```
kind create cluster --name
pgd
```

Part 2 - Install EDB Postgres Distributed for Kubernetes

After you have a Kubernetes installation up and running on your laptop, you can install EDB Postgres Distributed for Kubernetes.

See Installation for details.

Part 3 - Deploy a PGD cluster

As with any other deployment in Kubernetes, to deploy a PGD cluster you need to apply a configuration file that defines your desired `PGDGroup` resources that make up a PGD cluster.

Some sample files are included in the EDB Postgres Distributed for Kubernetes repository. The flexible_3regions.yaml manifest contains the definition of a PGD cluster with two data groups and a global witness node spread across three regions. Each data group consists of two data nodes and a local witness node.

> **Regions and availability zones**
>
> When creating Kubernetes clusters in different regions or availability zones for cross-regional replication, ensure the clusters can communicate with each other by enabling network connectivity. Specifically, every service created with a `-node` or `-group` suffix must be discoverable by all other `-node` and `-group` services. You can achieve this by deploying a network connectivity application like Submariner on every cluster.

> **Further reading**
>
> For more details about the available options, see the"API Reference" section.

You can deploy the `flexible-3-regions` example by saving it first and running:

```
kubectl apply -f flexible_3regions.yaml
```

You can check that the pods are being created using the `get pods` command:

```
kubectl get
pods
```

The pods are being created as part of PGD nodes. As described in Architecture, they're implemented on top of EDB Postgres for Kubernetes clusters.

You can list the clusters then, which shows the PGD nodes:

```
$ kubectl get
clusters
```

```
NAME          AGE     INSTANCES   READY   STATUS
PRIMARY
region-a-1    2m50s   1           1       Cluster in healthy state   region-a-1-
1
region-a-2    118s    1           1       Cluster in healthy state   region-a-2-
1
region-a-3    91s     1           1       Cluster in healthy state   region-a-3-
1
…
…
```

Ultimately, the PGD nodes are created as part of the PGD groups that make up your PGD cluster.

```
$ kubectl get
 pgdgroups
NAME        DATA INSTANCES   WITNESS INSTANCES   PHASE
AGE
region-a    2                1                   PGDGroup - Healthy
4m50s
region-b    2                1                   PGDGroup - Healthy
4m50s
region-c    0                1                   PGDGroup - Healthy
4m50s
```

Notice how the region-c group is only a witness node.

8 Managing EDB Postgres Distributed (PGD) databases

As described in the architecture document, EDB Postgres Distributed for Kubernetes is an operator created to deploy PGD databases. It provides an alternative over deployment with TPA, and by leveraging the Kubernetes ecosystem, it can offer self-healing and declarative control. The operator is also responsible of the backup and restore operations. See Backup.

However, many of the operations and control of PGD clusters aren't managed by the operator. The pods created by EDB Postgres Distributed for Kubernetes come with the PGD CLI installed. You can use this tool, for example, to execute a switchover.

PGD CLI

> **Warning**
>
> Don't use the PGD CLI to create and delete resources. For example, avoid the `create-proxy` and `delete-proxy` commands. Provisioning of resources is under the control of the operator, and manual creation and deletion isn't supported.

As an example, execute a switchover command.

We recommend that you use the PGD CLI from proxy pods. To find them, get a pod listing for your cluster:

```
kubectl get pods -n my-namespace

NAME                   READY   STATUS    RESTARTS      AGE
location-a-1-1         1/1     Running   0             2h
location-a-2-1         1/1     Running   0             2h
location-a-3-1         1/1     Running   0             2h
location-a-proxy-0     1/1     Running   0             2h
location-a-proxy-1     1/1     Running   0             2h
```

The proxy nodes have `proxy` in the name. Choose one, and get a command prompt in it:

```
kubectl exec -n my-namespace -ti location-a-proxy-0 -- bash
```

You now have a bash session open with the proxy pod. The `pgd` command is available:

```
pgd

Available Commands:
  check-health       Checks the health of the EDB Postgres Distributed cluster.
  <- snipped ->
  switchover         Switches over to new write leader.
  <- snipped ->
```

You can easily move your way through getting the information needed for the switchover:

```
pgd switchover --help

  $ pgd switchover --group-name group_a --node-name bdr-a1
  switchover is complete
```

```
pgd show-groups

Group      Group ID   Type   Parent Group Location Raft Routing Write Leader
-----      --------   ----   ------------ -------- ---- ------- ------------
world      3239291720 global                      true true    location-a-2
location-a 2135079751 data   world                 true true    location-a-1
```

```
pgd show-nodes
Node         Node ID    Group      Type Current State Target State Status Seq ID
----         -------    -----      ---- ------------- ------------ ------ ------
location-a-1 3165289849 location-a data ACTIVE        ACTIVE       Up     1
location-a-2 3266498453 location-a data ACTIVE        ACTIVE       Up     2
location-a-3 1403922770 location-a data ACTIVE        ACTIVE       Up     3
```

Accessing the database

In Use cases is a discussion on using the database within the Kubernetes cluster versus from outside. In Connectivity, you can find a discussion on services, which is relevant for accessing the database from applications.

However you implement your system, your applications must use the proxy service to connect to reap the benefits of PGD and of the increased self-healing capabilities added by the EDB Postgres Distributed for Kubernetes operator.

> **Important**
>
> As per the EDB Postgres for Kubernetes defaults, data nodes are created with a database called `app` and owned by a user named `app`, in contrast to the `bdrdb` database described in the EDB Postgres Distributed documentation. You can configure these values in the `cnp` section of the manifest. For reference, see Bootstrap in the EDB Postgres for Kubernetes documentation.

You might, however, want access to your PGD data nodes for administrative tasks, using the psql CLI.

You can get a pod listing for your PGD cluster and `kubectl exec` into a data node:

```
kubectl exec -n my-namespace -ti location-a-1-1 -- psql
```

In the familiar territory of psql, remember that the default created database is named `app` (see previous warning).

```
 postgres=# \c app
You are now connected to database "app" as user "postgres".
app=# \x
Expanded display is on.
app=# select * from bdr.node_summary;
-[ RECORD 1 ]----------------------------------------
node_name              | location-a-1
node_group_name        | location-a
interface_connstr      | host=location-a-1-node user=streaming_replica sslmode=verify-ca port=5432
sslkey=/controller/certificates/streaming_replica.key
sslcert=/controller/certificates/streaming_replica.crt sslrootcert=/controller/certificates/server-
ca.crt application_name=location-a-1 dbname=app
peer_state_name        | ACTIVE
peer_target_state_name | ACTIVE

<- snipped ->
```

For your applications, use the non-privileged role (`app` by default).

You need the user credentials, which are stored in a Kubernetes secret:

```
 kubectl get secrets

NAME                     TYPE                       DATA   AGE
<- snipped ->
location-a-app           kubernetes.io/basic-auth   2      2h
```

This secret contains the username and password needed for the Postgres DSN, encoded in base64:

```
 kubectl get secrets location-a-app -o yaml

apiVersion: v1
data:
  password: <base64-encoded-password>
  username: <base64-encoded-username>
kind: Secret
metadata:
  creationTimestamp: <timestamp>
  labels:

<- snipped ->
```

9 Backup on object stores

EDB Postgres Distributed for Kubernetes supports *online/hot backup* of PGD clusters through physical backup and WAL archiving on an object store. This means that the database is always up (no downtime required) and that point-in-time recovery (PITR) is available.

Common object stores

Multiple object stores are supported, such as AWS S3, Microsoft Azure Blob Storage, Google Cloud Storage, MinIO Gateway, or any S3-compatible provider. Given that EDB Postgres Distributed for Kubernetes configures the connection with object stores by relying on EDB Postgres for Kubernetes, see the EDB Postgres for Kubernetes cloud provider support documentation for more information.

> **Important**
>
> The EDB Postgres for Kubernetes documentation's Cloud Provider configuration section is available at `spec.backup.barmanObjectStore`. In EDB Postgres Distributed for Kubernetes examples, the object store section is at a different path: `spec.backup.configuration.barmanObjectStore`.

WAL archive

WAL archiving is the process that sends WAL files to the object storage, and it's essential to execute online/hot backups or PITR. In EDB Postgres Distributed for Kubernetes, each PGD node is set up to archive WAL files in the object store independently.

The WAL archive is defined in the PGD Group `spec.backup.configuration.barmanObjectStore` stanza, and is enabled as soon as a destination path and cloud credentials are set. You can choose to compress WAL files before they're uploaded and you can encrypt them. You can also enable parallel WAL archiving:

```
apiVersion:
pgd.k8s.enterprisedb.io/v1beta1
kind:
PGDGroup
[...]
spec:
  backup:
    configuration:
      barmanObjectStore:
        [...]
        wal:
          compression: gzip
          encryption:
AES256
          maxParallel: 8
```

For more information, see the EDB Postgres for Kubernetes WAL archiving documentation.

Scheduled backups

Scheduled backups are the recommended way to configure your backup strategy in EDB Postgres Distributed for Kubernetes. When the PGD group `spec.backup.configuration.barmanObjectStore` stanza is configured, the operator selects one of the PGD data nodes as the elected backup node for which it creates a `Scheduled Backup` resource.

The `.spec.backup.cron.schedule` field allows you to define a cron schedule specification, expressed in theGo `cron` package format.

```
apiVersion:
pgd.k8s.enterprisedb.io/v1beta1
kind:
PGDGroup
[...]
```

```
spec:
  backup:
    cron:
      schedule: "0 0 0 * *
*"
      backupOwnerReference: self
      suspend: false
      immediate: true
```

You can suspend scheduled backups if necessary by setting `.spec.backup.cron.suspend` to `true`. Setting this setting to `true` prevents any new backup from being scheduled.

If you want to execute a backup as soon as the `ScheduledBackup` resource is created, set `.spec.backup.cron.immediate` to `true`.

`.spec.backupOwnerReference` indicates the `ownerReference` to use in the created backup resources. The choices are:

- **none** — No owner reference for created backup objects.
- **self** — Sets the `ScheduledBackup` object as owner of the backup.
- **cluster** — Sets the cluster as owner of the backup.

> **Note**
>
> The EDB Postgres for Kubernetes `ScheduledBackup` object contains the `cluster` option to specify the cluster to back up. This option is currently not supported by EDB Postgres Distributed for Kubernetes and is ignored if specified.

If an elected backup node is deleted, the operator transparently elects a new backup node and reconciles the `ScheduledBackup` resource accordingly.

Retention policies

EDB Postgres Distributed for Kubernetes can manage the automated deletion of backup files from the backup object store using retention policies based on the recovery window. This process also takes care of removing unused WAL files and WALs associated with backups that are scheduled for deletion.

You can define your backups with a retention policy of 30 days:

```
apiVersion:
pgd.k8s.enterprisedb.io/v1beta1
kind:
PGDGroup
[...]
spec:
  backup:
    configuration:
      retentionPolicy: "30d"
```

For more information, see the EDB Postgres for Kubernetes retention policies in the EDB Postgres for Kubernetes documentation.

> **Important**
>
> Currently, the retention policy is applied only for the elected `Backup Node` backups and WAL files. Given that each other PGD node also archives its own WALs independently, it's your responsibility to manage the lifecycle of those WAL files, for example by leveraging the object storage data retention policy. Also, if you have an object storage data retention policy set up on every PGD node directory, make sure it's not overlapping or interfering with the retention policy managed by the operator.

Compression algorithms

Backups and WAL files are uncompressed by default. However, multiple compression algorithms are supported. For more information, see the EDB Postgres for Kubernetes compression algorithms documentation.

Tagging of backup objects

It's possible to specify tags as key-value pairs for the backup objects, namely base backups, WAL files, and history files. For more information, see the EDB Postgres for Kubernetes documentation about tagging of backup objects.

On-demand backups of a PGD node

A PGD node is represented as single-instance EDB Postgres for Kubernetes `Cluster` object. As such, if you need to, it's possible to request an on-demand backup of a specific PGD node by creating a EDB Postgres for Kubernetes `Backup` resource. To do that, see EDB Postgres for Kubernetes on-demand backups in the EDB Postgres for Kubernetes documentation.

> **Hint**
>
> You can retrieve the list of EDB Postgres for Kubernetes clusters that make up your PGD group by running `kubectl get cluster -l k8s.pgd.enterprisedb.io/group=my-pgd-group -n my-namespace`.

10 Recovery

In EDB Postgres Distributed for Kubernetes, recovery is available as a way to bootstrap a new PGD group starting from an available physical backup of a PGD node. The recovery can't be performed in place on an existing PGD group. EDB Postgres Distributed for Kubernetes also supports point-in-time recovery (PITR), which allows you to restore a PGD group up to any point in time, from the first available backup in your catalog to the last archived WAL. Having a WAL archive is mandatory in this case.

Prerequisite

Before recovering from a backup:

- Make sure that the PostgreSQL configuration (`.spec.cnp.postgresql.parameters`) of the recovered cluster is compatible with the original one from a physical replication standpoint.
- When recovering in a newly created namespace, first set up a cert-manager CA issuer before deploying the recovered PGD group.

For more information, see EDB Postgres for Kubernetes recovery - Additional considerations in the EDB Postgres for Kubernetes documentation.

Recovery from an object store

You can recover from a PGD node backup created by Barman Cloud and stored on supported object storage.

For example, given a PGD group `named` pgdgroup-example` with three instances with backups available, your object storage contains a directory for each node:

`pgdgroup-example-1`, `pgdgroup-example-2`, `pgdgroup-example-3`

This example defines a full recovery from the object store. The operator transparently selects the latest backup between the defined `serverNames` and replays up to the last available WAL.

```
apiVersion:
pgd.k8s.enterprisedb.io/v1beta1
kind:
PGDGroup
metadata:
  name: pgdgroup-restore
spec:
  [...]
  restore:
    serverNames:
      - pgdgroup-backup-1
      - pgdgroup-backup-2
      - pgdgroup-backup-3
    barmanObjectStore:
      destinationPath: "<destination path
here>"
      s3Credentials:
        accessKeyId:
          name: backup-storage-creds
          key: ID
        secretAccessKey:
          name: backup-storage-creds
          key:
KEY
      wal:
        compression: gzip
        encryption:
AES256
        maxParallel: 8
```

> **Important**
>
> Make sure to correctly configure the WAL section according to the source cluster. In the example, since the `pgdgroup-example` PGD group uses `compression` and `encryption`, make sure to set the proper parameters also in the PGD group that's being created by the `restore`.

> **Note**
>
> The example takes advantage of the parallel WAL restore feature, dedicating up to eight jobs to concurrently fetch the required WAL files from the archive. This feature can appreciably reduce the recovery time. Make sure that you plan ahead for this scenario and tune the value of this parameter for your environment. It makes a difference when you need it.

PITR from an object store

Instead of replaying all the WALs up to the latest one, after extracting a base backup, you can ask PostgreSQL to stop replaying WALs at any point in time. PostgreSQL uses this technique to achieve PITR. (The presence of a WAL archive is mandatory.)

This example defines a time-base target for the recovery:

```
apiVersion:
pgd.k8s.enterprisedb.io/v1beta1
kind:
PGDGroup
metadata:
  name: pgdgroup-restore
spec:
  [...]
  restore:
    recoveryTarget:
      targetTime: "2023-08-11 11:14:21.00000+02"
    serverNames:
      - pgdgroup-backup-1
      - pgdgroup-backup-2
      - pgdgroup-backup-3
    barmanObjectStore:
      destinationPath: "<destination path
here>"
      s3Credentials:
        accessKeyId:
          name: backup-storage-creds
          key: ID
        secretAccessKey:
          name: backup-storage-creds
          key:
KEY
      wal:
        compression: gzip
        encryption:
AES256
        maxParallel: 8
```

> **Important**
>
> PITR requires you to specify a `targetTime` recovery target by using the options described in Recovery targets. When you use `targetTime` or `targetLSN`, the operator selects the closest backup that was completed before that target. Otherwise, it selects the last available backup in chronological order between the specified `serverNames`.

Recovery from an object store specifying a `backupID`

The `.spec.restore.recoveryTarget.backupID` option allows you to specify a base backup from which to start the recovery process. By default, this value is empty. If you assign a value to it, the operator uses that backup as the base for the recovery. The value must be in the form of a Barman backup ID.

This example recovers a new PGD group from a specific backupID of the `pgdgroup-backup-1` PGD node:

```
apiVersion:
pgd.k8s.enterprisedb.io/v1beta1
kind:
PGDGroup
metadata:
  name: pgdgroup-restore
spec:
  [...]
```

```
  restore:
    recoveryTarget:
      backupID: 20230824T133000
    serverNames:
      - pgdgroup-backup-1
    barmanObjectStore:
      destinationPath: "<destination path
here>"
      s3Credentials:
        accessKeyId:
          name: backup-storage-creds
          key: ID
        secretAccessKey:
          name: backup-storage-creds
          key:
KEY
      wal:
        compression: gzip
        encryption:
AES256
        maxParallel: 8
```

> **Important**
>
> When a `backupID` is specified, make sure to define only the related PGD node in the `serverNames` option, and avoid defining the other ones.

> **Note**
>
> Defining a specific `backupID` is especially needed when using one of the following recovery targets: `targetName`, `targetXID`, and `targetImmediate`. In such cases, it's important to specify `backupID`, unless the last available backup in the catalog is okay.

Recovery targets

Beyond PITR are other recovery target criteria you can use. For more information on all the available recovery targets, seeEDB Postgres for Kubernetes recovery targets in the EDB Postgres for Kubernetes documentation.

11 Security

Security for EDB Postgres Distributed for Kubernetes is analyzed at three layers: code, container, and cluster.

> **Warning**
>
> In addition to security practices described here, you must perform regular InfoSec duties on your Kubernetes cluster. Familiarize yourself with Overview of Cloud Native Security in the Kubernetes documentation.

> **About the 4C's Security Model**
>
> See The 4C's Security Model in Kubernetes blog article for a better understanding and context of the approach EDB takes with security in EDB Postgres Distributed for Kubernetes.

Code

Source code of EDB Postgres Distributed for Kubernetes is systematically scanned for static analysis purposes, including security problems. EDB uses a popular open-source linter for Go called GolangCI-Lint directly in the CI/CD pipeline. GolangCI-Lint can run several linters on the same source code.

One of these is Golang Security Checker, or `gosec`. `gosec` is a linter that scans the abstract syntactic tree of the source against a set of rules aimed at discovering well-known vulnerabilities, threats, and weaknesses hidden in the code. These threads include hard-coded credentials, integer overflows, SQL injections, and others.

> **Important**
>
> A failure in the static code analysis phase of the CI/CD pipeline is a blocker for the entire delivery of EDB Postgres Distributed for Kubernetes, meaning that each commit is validated against all the linters defined by GolangCI-Lint.

Container

Every container image that's part of EDB Postgres Distributed for Kubernetes is built by way of CI/CD pipelines following every commit. Such images include not only those of the operator but also of the operands, specifically every supported PostgreSQL version. In the pipelines, images are scanned with:

- Dockle for best practices in terms of the container build process
- Clair for vulnerabilities found in both the underlying operating system and libraries and applications that they run

> **Important**
>
> All operand images are rebuilt once a day by our pipelines in case of security updates at the base image and package level, providing patch level updates for the container images that EDB distributes.

The following guidelines and frameworks were taken into account for container-level security:

- The Container Image Creation and Deployment Guide, developed by the Defense Information Systems Agency (DISA) of the United States Department of Defense (DoD)
- The CIS Benchmark for Docker, developed by the Center for Internet Security (CIS)

> **About the container-level security**
>
> See the Security and Containers in EDB Postgres Distributed for Kubernetes blog article for more information about the approach that EDB takes on security at the container level in EDB Postgres Distributed for Kubernetes.

Cluster

Security at the cluster level takes into account all Kubernetes components that form both the control plane and the nodes as well as the applications that run in the cluster, including PostgreSQL.

Role-based access control (RBAC)

The operator interacts with the Kubernetes API server with a dedicated service account called pgd-operator-controller-manager. In Kubernetes this account is installed by default in the `pgd-operator-system` namespace. A cluster role binds between this service account and the pgd-operator-controller-manager cluster role that defines the set of rules, resources, and verbs granted to the operator.

RedHat OpenShift directly manages the operator RBAC entities by way of Operator Lifecycle Manager (OLM). OLM allows you to grant permissions only where they're required, implementing the principle of least privilege.

> **Important**
>
> These permissions are exclusively reserved for the operator's service account to interact with the Kubernetes API server. They aren't directly accessible by the users of the operator that interact only with `PGDGroup` and `PGDGroupCleanup` resources.

The following are some examples and, most importantly, the reasons why EDB Postgres Distributed for Kubernetes requires full or partial management of standard Kubernetes namespaced resources.

`jobs` : The operator needs to handle jobs to manage different `PGDGroup` phases.

`poddisruptionbudgets` : The operator uses pod disruption budgets to make sure enough PGD nodes are kept active during maintenance operations.

`pods` : The operator needs to manage PGD nodes as a `Cluster` resource.

`secrets` : Unless you provide certificates and passwords to your data nodes, the operator adopts the "convention over configuration" paradigm by self-provisioning random-generated passwords and TLS certificates and by storing them in secrets.

`serviceaccounts` : The operator needs to create a service account to enable the `PGDGroup` recovery job to retrieve the backup objects from the object store where they reside.

`services` : The operator needs to control network access to the PGD cluster from applications and properly manage failover/switchover operations in an automated way.

`statefulsets` : The operator needs to manage PGD proxies.

`validatingwebhookconfigurations` and `mutatingwebhookconfigurations` : The operator injects its self-signed webhook CA into both webhook configurations, which are needed to validate and mutate all the resources it manages. For more details, see the Kubernetes documentation.

To see all the permissions required by the operator, you can run `kubectl describe clusterrole pgd-operator-manager-role`.

EDB Postgres Distributed for Kubernetes internally manages the PGD nodes using the `Cluster` resource as defined by EDB Postgres for Kubernetes. See the EDB Postgres for Kubernetes documentation for the list of permissions used by the EDB Postgres for Kubernetes operator service account.

Calls to the API server made by the instance manager

The instance manager, which is the entry point of the operand container, needs to make some calls to the Kubernetes API server to ensure that the status of some resources is correctly updated and to access the config maps and secrets that are associated with that Postgres cluster. Such calls are performed through a dedicated `ServiceAccount` created by the operator that shares the same PostgreSQL `Cluster` resource name.

> **Important**
>
> The operand can access only a specific and limited subset of resources through the API server. A service account is the recommended way to access the API server from within a pod. See the Kubernetes documentation for details.

See the EDB Postgres for Kubernetes documentation for more information on the instance manager.

Pod security policies

A pod security policy is the Kubernetes way to define security rules and specifications that a pod needs to meet to run in a cluster. For InfoSec reasons, every Kubernetes platform must implement them.

EDB Postgres Distributed for Kubernetes doesn't require privileged mode for containers execution. The PostgreSQL containers run as the postgres system user. No component requires running as root.

Likewise, volumes access doesn't require privileged mode or root privileges. Proper permissions must be assigned by the Kubernetes platform or administrators. The PostgreSQL containers run with a read-only root filesystem, that is, no writable layer.

The operator explicitly sets the required security contexts.

On Red Hat OpenShift, Cloud Native PostgreSQL runs in the `restricted` security context constraint, the most restrictive one. The goal is to limit the execution of a pod to a namespace allocated UID and SELinux context.

> **Security Context Constraints in OpenShift**
>
> For more information on security context constraints (SCC) in OpenShift, see the Managing SCC in OpenShift article.

> **Security context constraints and namespaces**
>
> As stated in the Openshift documentation, SCCs aren't applied in the default namespaces (`default` , `kube-system` , `kube-public` , `openshift-node` , `openshift-infra` , `openshift`). Don't use them to run pods. CNP clusters deployed in those namespaces will be unable to start due to missing SCCs.

Exposed ports

EDB Postgres Distributed for Kubernetes exposes ports at operator, instance manager, and operand levels, as shown in the table.

System	Port number	Exposing	Name	Certificates	Authentication
operator	9443	webhook server	`webhook-server`	TLS	Yes
operator	8080	metrics	`metrics`	no TLS	No
instance manager	9187	metrics	`metrics`	no TLS	No
instance manager	8000	status	`status`	no TLS	No
operand	5432	PostgreSQL instance	`postgresql`	optional TLS	Yes

PGD

The current implementation of EDB Postgres Distributed for Kubernetes creates passwords for the postgres superuser and the database owner.

As far as encryption of passwords is concerned, EDB Postgres Distributed for Kubernetes follows the default behavior of PostgreSQL: starting with PostgreSQL 14, `password_encryption` is by default set to `scram-sha-256` . On earlier versions, it's set to `md5` .

> **Important**
>
> See Connection DSNs and SSL in the PGD documentation for details.

You can disable management of the postgres user password using secrets by setting `enableSuperuserAccess` to `false` in the `cnp` section of

the spec.

> **Note**
>
> The operator supports toggling the `enableSuperuserAccess` option. When you disable it on a running cluster, the operator ignores the content of the secret. Remove it (if previously generated by the operator) and set the password of the postgres user to `NULL`, in effect disabling remote access through password authentication.

Storage

EDB Postgres Distributed for Kubernetes delegates encryption at rest to the underlying storage class. For data protection in production environments, we highly recommend that you choose a storage class that supports encryption at rest.

12 Connectivity

Information about secure network communications in a PGD cluster includes:

- Services
- Domain names resolution using fully qualified domain names (FQDN)
- TLS configuration

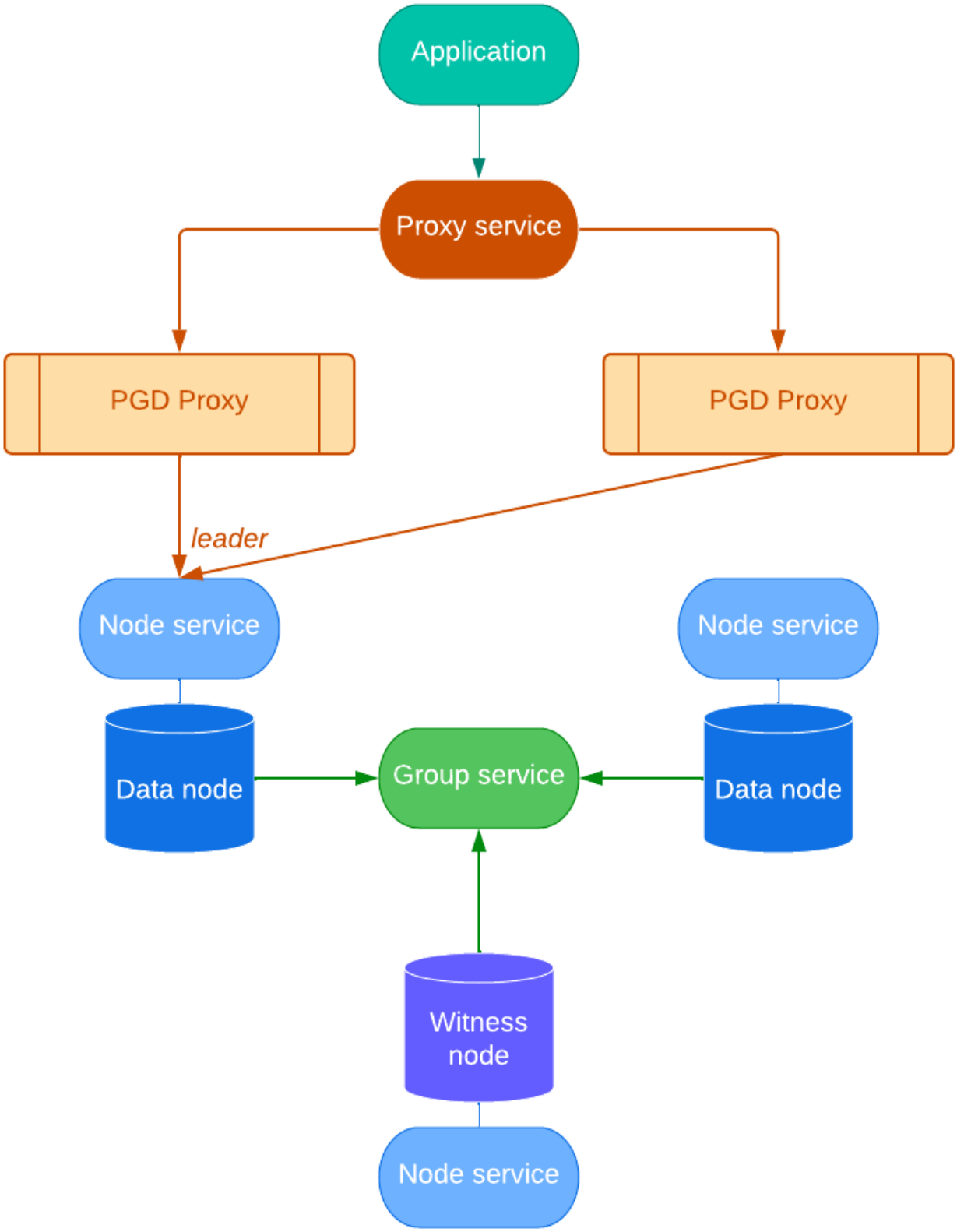
> **Notice**
>
> Although these topics might seem unrelated to each other, they all participate in the configuration of the PGD resources to make them universally identifiable and accessible over a secure network.

Services

Resources in a PGD cluster are accessible through Kubernetes services. Every PGD group manages several of them, namely:

- One service per node, used for internal communications (*node service*)
- A *group service* to reach any node in the group, used primarily by EDB Postgres Distributed for Kubernetes to discover a new group in the cluster
- A *proxy service* to enable applications to reach the write leader of the group transparently using PGD Proxy

For an example that uses these services, see Connecting an application to a PGD cluster.



Each service is generated from a customizable template in the `.spec.connectivity` section of the manifest.

All services must be reachable using their FQDN from all the PGD nodes in all the Kubernetes clusters. SeeDomain names resolution.

EDB Postgres Distributed for Kubernetes provides a service templating framework that gives you the availability to easily customize services at the following three levels:

Node Service Template : Each PGD node is reachable using a service that can be configured in the `.spec.connectivity.nodeServiceTemplate` section.

Group Service Template : Each PGD group has a group service that's a single entry point for the whole group and that can be configured in the `.spec.connectivity.groupServiceTemplate` section.

Proxy Service Template : Each PGD group has a proxy service to reach the group write leader through the PGD proxy and can be configured in the `.spec.connectivity.proxyServiceTemplate` section. This is the entry-point service for the applications.

You can use templates to create a LoadBalancer service or to add arbitrary annotations and labels to a service to integrate with other components available in the Kubernetes system (that is, to create external DNS names or tweak the generated load balancer).

Domain names resolution

EDB Postgres Distributed for Kubernetes ensures that all resources in a PGD group have a FQDN by adopting a convention that uses the PGD group name as a prefix for all of them.

As a result, it expects you to define the domain name of the PGD group. This can be done through the `.spec.connectivity.dns` section, which controls how the FQDN for the resources are generated with two fields:

- `domain` — Domain name for all the objects in the PGD group to use (mandatory).
- `hostSuffix` — Suffix to add to each service in the PGD group (optional).

TLS configuration

EDB Postgres Distributed for Kubernetes requires that resources in a PGD cluster communicate over a secure connection. It relies on PostgreSQL's native support for SSL connections to encrypt client/server communications using TLS protocols for increased security.

Currently, EDB Postgres Distributed for Kubernetes requires that cert-manager is installed. Cert-manager was chosen as the tool to provision dynamic certificates given that it's widely recognized as the standard in a Kubernetes environment.

The `spec.connectivity.tls` section describes how the communication between the nodes happens:

- `mode` is an enumeration describing how the server certificates are verified during PGD group nodes communication. It accepts the following values, as documented in SSL Support in the PostgreSQL documentation:
  - `verify-full`
  - `verify-ca`
  - `required`
- `serverCert` defines the server certificates used by the PGD group nodes to accept requests. The clients validate this certificate depending on the passed TLS mode. It accepts the same values as `mode`.
- `clientCert` defines the `streaming_replica` user certificate used by the nodes to authenticate each other.

Server TLS configuration

The server certificate configuration is specified in the `.spec.connectivity.tls.serverCert.certManager` section of the `PGDGroup`

custom resource.

The following assumptions were made for this section to work:

- An issuer `.spec.connectivity.tls.serverCert.certManager.issuerRef` is available for the domain `.spec.connectivity.dns.domain` and any other domain used by `.spec.connectivity.tls.serverCert.certManager.altDnsNames`.
- There's a secret containing the public certificate of the CA used by the issuer `.spec.connectivity.tls.serverCert.caCertSecret`.

The `.spec.connectivity.tls.serverCert.certManager` is used to create a per-node cert-manager certificate request. The resulting certificate is used by the underlying Postgres instance to terminate TLS connections.

The operator adds the following altDnsNames to the certificate:

- `$node$hostSuffix.$domain`
- `$groupName$hostSuffix.$domain`

> **Important**
>
> It's your responsibility to add to `.spec.connectivity.tls.serverCert.certManager.altDnsNames` any name required from the underlying networking architecture, for example, load balancers used by the user to reach the nodes.

Client TLS configuration

The operator requires client certificates to be dynamically provisioned using cert-manager (the recommended approach) or pre-provisioned using secrets.

Dynamic provisioning via cert-manager

The client certificates configuration is managed by the `.spec.connectivity.tls.clientCert.certManager` section of the `PGDGroup` custom resource. The following assumptions were made for this section to work:

- An issuer `.spec.connectivity.tls.clientCert.certManager.issuerRef` is available and signs a certificate with the common name `streaming_replica`.
- There's a secret containing the public certificate of the CA used by the issuer `.spec.connectivity.tls.clientCert.caCertSecret`.

The operator uses the configuration under `.spec.connectivity.tls.clientCert.certManager` to create a certificate request per the `streaming_replica` Postgres user. The resulting certificate is used to secure communication between the nodes.

Pre-provisioned certificates via secrets

Alternatively, you can specify a secret containing the pre-provisioned client certificate for the streaming replication user through the `.spec.connectivity.tls.clientCert.preProvisioned.streamingReplica.secretRef` option. The certificate lifecycle in this case is managed entirely by a third party, either manually or automated, by updating the content of the secret.

Connecting to a PGD cluster from an application

Connecting to a PGD group from an application running inside the same Kubernetes cluster or from outside the cluster is a simple procedure. In both cases, you connect to the proxy service of the PGD group as the `app` user. The proxy service is a LoadBalancer service that routes the connection to the write leader of the PGD group.

Connecting from inside the cluster

When connecting from inside the cluster, you can use the proxy service name to connect to the PGD group. The proxy service name is composed of the PGD group name and the optional host suffix defined in the `.spec.connectivity.dns` section of the `PGDGroup` custom resource.

For example, if the PGD group name is `my-group`, and the host suffix is `.my-domain.com`, the proxy service name is `my-group.my-domain.com`.

Before connecting, you need to get the password for the app user from the app user secret. The naming format of the secret is `my-group-app` for a PGD group named `my-group`.

You can get the username and password from the secret using the following commands:

```
kubectl get secret my-group-app -o jsonpath='{.data.username}' | base64 --
decode
kubectl get secret my-group-app -o jsonpath='{.data.password}' | base64 --
decode
```

With this, you have all the pieces for a connection string to the PGD group:

```
postgresql://<app-user>:<app-password>@<proxy-service-name>:5432/<database>
```

Or, for a `psql` invocation:

```
psql -U <app-user> -h <proxy-service-name>
<database>
```

Where `app-user` and `app-password` are the values you got from the secret, and `database` is the name of the database you want to connect to. (The default is `app` for the app user.)

Connecting from outside the Kubernetes cluster

When connecting from outside the Kubernetes cluster, in the general case, the Ingress resource or a load balancer is necessary. Check your cloud provider or local installation for more information about their behavior in your environment.

Ingresses and load balancers require a pod selector to forward connection to the PGD proxies. When configuring them, we suggest using the following labels:

- `k8s.pgd.enterprisedb.io/group` — Set the PGD group name.
- `k8s.pgd.enterprisedb.io/workloadType` — Set to `pgd-proxy`.

If using Kind or other solutions for local development, the easiest way to access the PGD group from outside is to use port forwarding to the proxy service. You can use the following command to forward port 5432 on your local machine to the proxy service:

```
kubectl port-forward svc/my-group.my-domain.com 5432:5432
```

Where `my-group.my-domain.com` is the proxy service name from the previous example.

13 Certificates

EDB Postgres Distributed for Kubernetes was designed to natively support TLS certificates. To set up an PGD cluster, each PGD node requires:

- A server certification authority (CA) certificate
- A server TLS certificate signed by the server CA
- A client CA certificate
- A streaming replication client certificate generated by the client CA

> **Note**
>
> You can find all the secrets used by each PGD node and the expiry dates in the cluster (PGD node) status.

EDB Postgres Distributed for Kubernetes is very flexible when it comes to TLS certificates. It operates primarily in two modes:

- **Operator managed** — Certificates are internally managed by the operator in a fully automated way and signed using a CA created by EDB Postgres Distributed for Kubernetes.
- **User provided** — Certificates are generated outside the operator and imported in the cluster definition as secrets. EDB Postgres Distributed for Kubernetes integrates itself with cert-manager.

For more information, see the EDB Postgres for Kubernetes documentation.

14 Client TLS/SSL connections

> **Certificates**
>
> See Certificates for more details on how EDB Postgres Distributed for Kubernetes supports TLS certificates.

The EDB Postgres Distributed for Kubernetes operator was designed to work with TLS/SSL for both encryption in transit and authentication on server and client sides. PGD nodes are created as cluster resources using the EDB Postgres for Kubernetes operator. This includes deploying a certification authority (CA) to create and sign TLS client certificates.

See the EDB Postgres for Kubernetes documentation for more information on issuers and certificates.

15 Declarative pausing and resuming

The *declarative pausing and resuming* feature enables saving CPU power by removing the database pods while keeping the database PVCs.

Declarative pausing and resuming leverages the hibernation functionality available for EDB Postgres for Kubernetes. For additional depth and an explanation of how hibernation works, see the Postgres for Kubernetes documentation on declarative hibernation.

Request pause by adding the `k8s.pgd.enterprisedb.io/pause` annotation in the desired PGD group.

For example:

```
kubectl annotate pgdgroup region-a
k8s.pgd.enterprisedb.io/pause=on
```

After a few seconds, the requested PGD group will be in paused state, with all the database pods removed:

```
kubectl get
pgdgroups

NAME       DATA INSTANCES   WITNESS INSTANCES   PHASE
AGE
region-a   2                1                   PGDGroup - Paused
25m
region-b   2                1                   PGDGroup - Healthy
25m
region-c   0                1                   PGDGroup - Healthy
25m
```

To resume a paused PGD group, set the annotation to `off`. Remember to add the `--overwrite` flag:

```
kubectl annotate pgdgroup region-a k8s.pgd.enterprisedb.io/pause=off --
overwrite
```

In a few seconds, you should see the nodes start resuming, and the pods to be re-created.

```
kubectl get
pgdgroups

NAME       DATA INSTANCES   WITNESS INSTANCES   PHASE
AGE
region-a   2                1                   Pause - resume nodes
1m
region-b   2                1                   PGDGroup - Healthy
25m
region-c   0                1                   PGDGroup - Healthy
25m
```

There are some requirements before the pause annotation can put the PGD group on Pause. Ideally, the PGD Group should be in Healthy state. Alternatively, if all the data nodes in the PGD Group are healthy at the individual level, Pause can also be initiated.

16 EDB private image registries

The images for the EDB Postgres Distributed for Kubernetes and EDB Postgres for Kubernetes operators, as well as various operands, are kept in private container image registries under `docker.enterprisedb.com`.

> **Important**
>
> Access to the private registries requires an account with EDB and is reserved for EDB customers with a valid subscription plan. Credentials are run through your EDB account. For trials, see Trials.

Which repository to choose?

EDB Postgres Distributed for Kubernetes is available as part of the Extreme High Availability Add-On on top of either the EDB Enterprise Plan or EDB Standard Plan.

Depending on your subscription plan, EDB Postgres Distributed for Kubernetes is in one of the following repositories.

Plan	Repository
EDB Standard Plan	`k8s_standard_pgd`

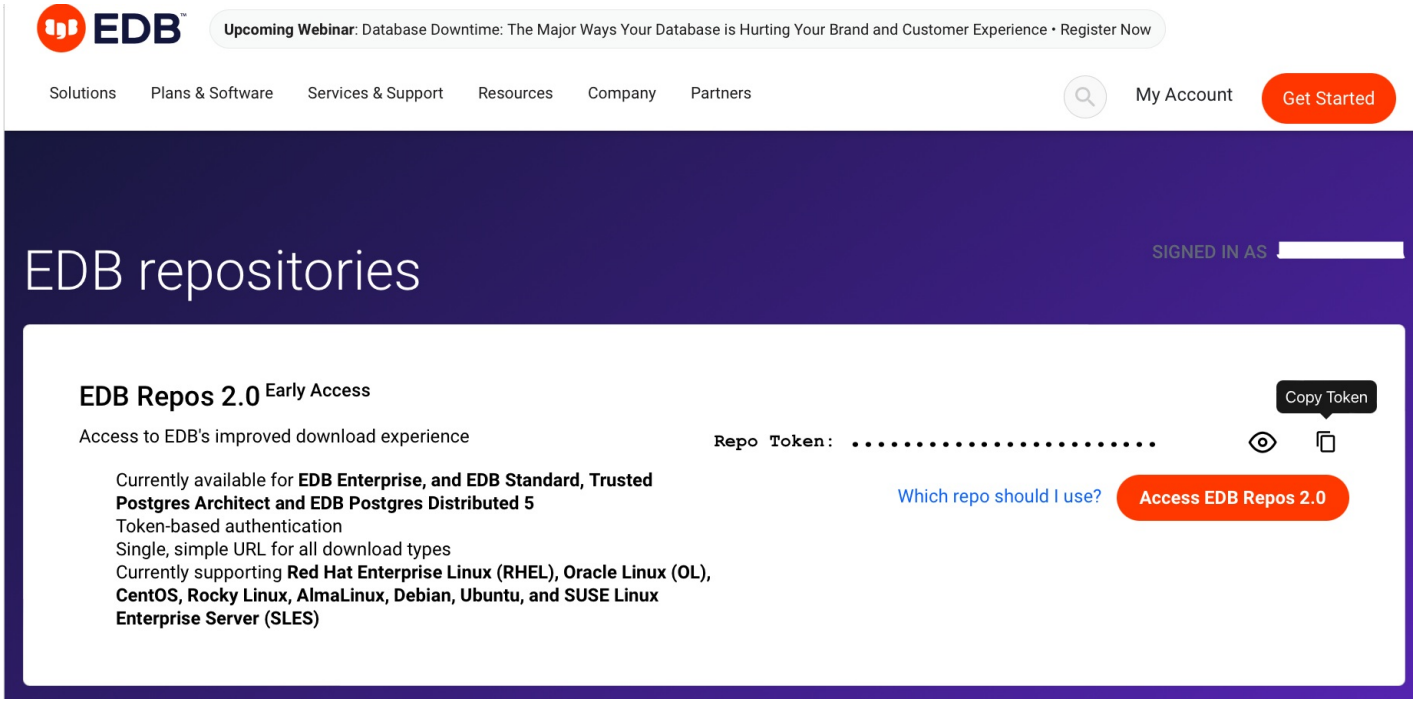
Plan	Repository
EDB EnterpriseDB Plan	`k8s_enterprise_pgd`

Use the name of the repository as the username when you log in to the EDB container registry, for example, through `docker login` or a `kubernetes.io/dockerconfigjson` pull secret.

> **Important**
>
> Each repository contains all the images you can access with your plan. You don't need to connect to different repositories to access different images, such as operator or operand images.

How to retrieve the token

In the repos page in EDB, is an EDB Repos 2.0 section where a repo token appears obscured.



Next to the repo token is a **Copy Token** button to copy the token and an eye icon for looking at the content of the token.

Use the repo token as the password when you log in to the EDB container registry.

Example with `docker login`

You can log in using Docker from your terminal. We suggest that you copy the repo token using **Copy Token**. The `docker` command prompts you for a username and a password.

The username is the repo you're trying to access, and the password is the token you just copied:

```
$ docker login docker.enterprisedb.com
Username: k8s_enterprise_pgd
```

```
Password:
Login Succeeded
```

Trials

If you're a trialist or a preview user, use `k8s_enterprise_pgd` as the name of the repository, and follow the instructions in How to retrieve the token for the token.

Operand images

EDB Postgres Distributed for Kubernetes is an operator that supports running EDB Postgres Distributed (PGD) version 5 on three PostgreSQL distributions:

- PostgreSQL
- EDB Postgres Advanced Server
- EDB Postgres Extended

> **Important**
>
> See Choosing a Postgres distribution in the PGD documentation for details and a comparison of PGD on the different supported PostgreSQL distributions.

Due to the immutable application container adoption in EDB operators, the operator expects for the container images to include all the binaries required to run the requested version of PGD on top of the required distribution and version of Postgres.

These images follow the requirements and the conventions described in Container image requirements in the EDB Postgres for Kubernetes documentation, adding the `bdr5` extension.

The table shows the image name prefix for each Postgres distribution.

Postgres distribution	Versions	Image name	Repositories
EDB Postgres Extended	15, 14	`edb-postgres-extended-pgd`	`k8s_standard_pgd`, `k8s_enterprise_pgd`
EDB Postgres Advanced	15, 14	`edb-postgres-advanced-pgd`	`k8s_enterprise_pgd`

> **Image naming**
>
> For more information on operand image naming and proxy image naming, see Identify your image name.

17 Predefined labels

These predefined labels are managed by the EDB Postgres Distributed for Kubernetes operator.

`k8s.pgd.enterprisedb.io/certificateType` : Indicates the type of the certificates. `replication` indicates a certificate to be used to authenticate the replication client. `server` indicates a certificate to be used for server authentication.

`k8s.pgd.enterprisedb.io/group` : Name of the PGDGroup that the resource belongs to. Added to cluster or instance resources.

`k8s.pgd.enterprisedb.io/isWitnessService` : Indicates a service is for a witness node.

`k8s.pgd.enterprisedb.io/type` : Type of the resource added to cluster or instance resources, usually `node` .

`k8s.pgd.enterprisedb.io/workloadType` : Indicates the workload type of the resource added to cluster or instance resources. `pgd-node-data` indicates data node; `pgd-node-witness` a witness node; `pgd-proxy` for PGD Proxy node; `proxy-svc` for PGD Proxy service; `group-svc` for PGD group service to communicate with any node in the PGDGroup; `node-svc` is a service created from the CNP service template; `scheduled-backup` is added to `scheduledBackup` resources; `bootstrap-cross-location-pgd-group` is added to the pod that creates a cross-location PGD group; `pgd-node-restore` is added to the pod that starts the node restore process.

Predefined annotations

`k8s.pgd.enterprisedb.io/dirtyMetadata` : Set in CNP cluster that have been generated from a backup and need to have their metadata cleaned up before creating the PGD node. This is written by the restore job.

`k8s.pgd.enterprisedb.io/hash` : Contains the hash of the used PGDGroup spec.

`k8s.pgd.enterprisedb.io/latestCleanupExecuted` : Set in the PGDGroup to indicate that the cleanup was executed.

`k8s.pgd.enterprisedb.io/node` : Contains the name of the node for which a certain certificate was generated. Added to the certificate resources.

`k8s.pgd.enterprisedb.io/noFinalizers` : Set in the PGDGroup with value `true` to skip the finalizer execution. For internal use only.

`k8s.pgd.enterprisedb.io/pause` : Set in the PGDGroup to pause a PGDGroup.

`k8s.pgd.enterprisedb.io/recoverabilityPointsByMethod` : Set in the PGDGroup to store the CNP cluster's first recoverability points by method in a tamper-proof place.

`k8s.pgd.enterprisedb.io/seedingServer` : Set in the PGDGroup to indicate to the operator which server to restore. This is written by the restore job.

`k8s.pgd.enterprisedb.io/seedingSnapshots` : Set in the PGDGroup to indicate to the operator which snapshots to restore. This is written by the restore job.

18 PGDGroup parting

Deletion and finalizers

When deleting a PGD Group, the operator will start parting every node in the group first. It will connect to an active instance and part every node in the target group. Once a node is parted, it will not participate in replication and consensus operations. To make sure the node is correctly parted before being deleted, the operator uses the `k8s.pgd.enterprisedb.io/partNodes` finalizer. Please refer to the kubernetes document on finalizers for context.

> **Note**

> If a namespace holding a PGD Group is deleted directly, we can't ensure the deleting and parting sequence is carried out correctly. Before deleting a namespace, it is recommended to delete all the contained PGD groups.

Time limit

When parting a node, the operator needs to connect to an active instance to execute the `bdr.part_node` function. To avoid this operation hanging, a time limit for the finalizer is used; by default, it is 300 seconds. After the time limit expires, the finalizer will be removed, and the node will be deleted anyway, potentially leaving stale metadata in the global PGD catalog. This time limit can be configured through `spec.failingFinalizerTimeLimitSeconds`, which is specified in seconds.

Skip finalizer

For testing purposes only, the operator also provides an annotation to skip the finalizer: `k8s.pgd.enterprisedb.io/noFinalizers`. When this annotation is added to a PGDGroup, the finalizer will be skipped when the PGDGroup is being deleted, and the nodes will not be parted from the PGD cluster.

PGDGroup cleanup

Cleanup parted node

Once the PGDGroup is deleted, its metadata will remain in the catalog in `PARTED` state in the `bdr.node_summary` table. The PGD4K operator defines a CRD named `PGDGroupCleanup` to help clean up the `PARTED` PGDGroup.

In the example below, the `PGDGroupCleanup` executes locally from `region-a`, and will clean up all of region-b, with the pre-requisite that all the nodes must be in the `PARTED` state.

```
apiVersion:
pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroupCleanup
metadata:
  name: region-b-cleanup
spec:
  executor: region-a
  target: region-b
```

Please note that if the target group (`region-b` in the example) contains nodes not in a `PARTED` state, the Group Cleanup will stop in phase `PGDGroupCleanup - Target PGDGroup is not parted, waiting for it to be parted before executing PGDGroupCleanup`. In cases of extreme need, we can add the `force` option.

> **Warning**
>
> Using `force` can leave the PGD cluster in an inconsistent state. Use it only to recover from failures in which you can't part the group nodes any other way.

```
apiVersion:
pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroupCleanup
metadata:
```

```
  name: region-b-cleanup
spec:
  force: true
  executor: region-a
  target: region-b
```

19 Red Hat OpenShift

EDB Postgres Distributed for Kubernetes is a certified operator that can be installed on OpenShift using a web interface.

Ensuring access to EDB private registry

> **Important**
>
> You need access to the private EDB repository where both the operator and operand images are stored. Access requires a valid EDB subscription plan. See Accessing EDB private image registries for details.

The OpenShift install uses pull secrets to access the operand and operator images, which are held in a private repository.

Once you have credentials to the private repo, you need to create two pull secrets in the `openshift-operators` namespace:

- `pgd-operator-pull-secret` for the EDB Postgres Distributed for Kubernetes operator images
- `postgresql-operator-pull-secret` for the EDB Postgres for Kubernetes operator images

You can create each secret using the `oc create` command:

```
oc create secret docker-registry pgd-operator-pull-secret
\
  -n openshift-operators --docker-server=docker.enterprisedb.com \
  --docker-username="@@REPOSITORY@@" \
  --docker-password="@@TOKEN@@"

oc create secret docker-registry postgresql-operator-pull-secret
\
  -n openshift-operators --docker-server=docker.enterprisedb.com \
  --docker-username="@@REPOSITORY@@" \
  --docker-password="@@TOKEN@@"
```

Where:

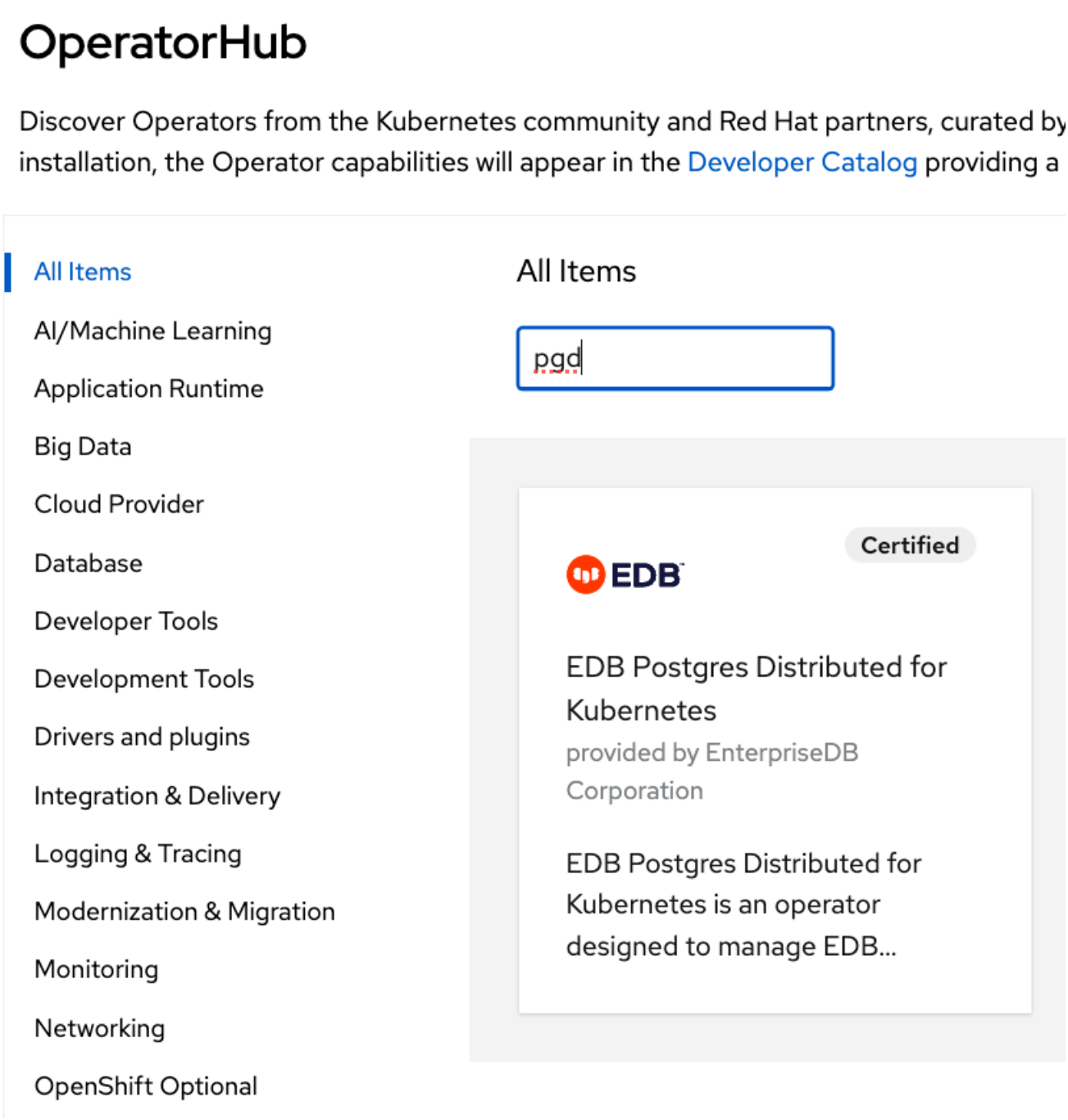
- `@@REPOSITORY@@` is the name of the repository, as explained in Which repository to choose?.
- `@@TOKEN@@` is the repository token for your EDB account, as explained in How to retrieve the token.

Installing the operator

The EDB Postgres Distributed for Kubernetes operator can be found in the Red Hat OperatorHub directly from your OpenShift dashboard.

1. From the hamburger menu, select **Operators > OperatorHub**.

2. In the web console, use the search box to filter the listing. For example, enter `EDB` or `pgd`:



3. Read the information about the operator and select **Install**.

4. In the Operator Installation page, select:

   - The installation mode. Cluster-wide is currently the only mode.
   - The update channel (currently **preview**).
   - The approval strategy, following the availability on the marketplace of a new release of the operator, certified by Red Hat:
     - **Automatic**: OLM upgrades the running operator with the new version.
     - **Manual**: OpenShift waits for human intervention by requiring an approval in the **Installed Operators** section.

Cluster-wide installation

With cluster-wide installation, you're asking OpenShift to install the operator in the default `openshift-operators` namespace and to make it available to all the projects in the cluster. This is the default and normally recommended approach to install EDB Postgres Distributed for Kubernetes.

From the web console, for **Installation mode**, select **All namespaces on the cluster (default)**.

On installation, the operator is visible in all namespaces. In case there were problems during installation, check the logs in any pods in the `openshift-operators` project on the **Workloads > Pods** page as you would with any other OpenShift operator.

> **Beware**
>
> By choosing the cluster-wide installation you, can't easily move to a single-project installation later.

Creating a PGD cluster

After the installation by OpenShift, the operator deployment is in the `openshift-operators` namespace. Notice the cert-manager operator was also installed, as was the EDB Postgres for Kubernetes operator (`postgresql-operator-controller-manager`).

```
$ oc get deployments -n openshift-
operators
NAME                                        READY   UP-TO-DATE   AVAILABLE
AGE
cert-manager-operator                       1/1     1            1
11m
pgd-operator-controller-manager             1/1     1            1
11m
postgresql-operator-controller-manager-1-20-0   1/1     1            1
23h
…
```

After checking that the `pgd-operator-controller-manager` deployment is READY, you can start creating PGD clusters. The EDB Postgres Distributed for Kubernetes repository contains some useful sample files.

You must deploy your PGD clusters on a dedicated namespace/project. The default namespace is reserved.

First, then, create a new namespace, and deploy a self-signed certificate `Issuer` in it:

```
oc create ns my-
namespace
oc apply -n my-namespace -f
\
  https://raw.githubusercontent.com/EnterpriseDB/edb-postgres-for-kubernetes-
charts/main/hack/samples/issuer-selfsigned.yaml
```

Using PGD in a single OpenShift cluster in a single region

Now you can deploy a PGD cluster, for example a flexible 3-region, which contains two data groups and a witness group. You can find the YAML manifest in the file `flexible_3regions.yaml` .

```
oc apply -f flexible_3regions.yaml -n my-
namespace
```

Your PGD groups start to come up:

```
$ oc get pgdgroups -n my-
namespace
NAME       DATA INSTANCES   WITNESS INSTANCES   PHASE                PHASE DETAILS
AGE
region-a   2                1                   PGDGroup - Healthy
23m
region-b   2                1                   PGDGroup - Healthy
23m
region-c   0                1                   PGDGroup - Healthy
23m
```

Using PGD in multiple OpenShift clusters in multiple regions

To deploy PGD in multiple OpenShift clusters in multiple regions, you must first establish a way for the PGD groups to communicate with each other. The recommended way of achieving this with multiple OpenShift clusters is to use Submariner. Configuring the connectivity is outside the scope of this documentation. However, once you've established connectivity between the OpenShift clusters, you can deploy PGD groups synced with one another.

> **Warning**
>
> This example assumes you're deploying three PGD groups, one in each OpenShift cluster, and that you established connectivity between the OpenShift clusters using Submariner.

Similar to the single-cluster example, this example creates two data PGD groups and one witness group. In contrast to that example, each group lives in a different OpenShift cluster.

In addition to basic connectivity between the OpenShift clusters, you need to ensure that each OpenShift cluster contains a certificate authority that's trusted by the other OpenShift clusters. This condition is required for the PGD groups to communicate with each other.

The OpenShift clusters can all use the same certificate authority, or each cluster can have its own certificate authority. Either way, you need to ensure that each OpenShift cluster's certificates trust the other OpenShift clusters' certificate authorities.

This example uses a self-signed certificate that has a single certificate authority used for all certificates on all the OpenShift clusters.

The example refers to the OpenShift clusters as `OpenShift Cluster A`, `OpenShift Cluster B`, and `OpenShift Cluster C`. In OpenShift, an installation of the EDB Postgres Distributed for Kubernetes operator from OperatorHub includes an installation of the cert-manager operator. We recommend creating and managing certificates with cert-manager.

1. Create a namespace to hold `OpenShift Cluster A`, and in it also create the needed objects for a self-signed certificate. Assuming that the PGD operator and the cert-manager are installed, you create a self-signed certificate `Issuer` in that namespace.

```
oc create ns pgd-
group
oc apply -n pgd-group -f \
  https://raw.githubusercontent.com/EnterpriseDB/edb-postgres-for-kubernetes-
charts/main/hack/samples/issuer-selfsigned.yaml
```

1. After a few moments, cert-manager creates the issuers and certificates. There are also now two secrets in the `pgd-group` namespace: `server-ca-key-pair` and `client-ca-key-pair`. These secrets contain the certificates and private keys for the server and client certificate authorities. You need to copy these secrets to the other OpenShift clusters before applying the `issuer-selfsigned.yaml` manifest. You can use the `oc get secret` command to get the contents of the secrets:

```
oc get secret server-ca-key-pair -n pgd-group -o yaml > server-ca-key-
pair.yaml
oc get secret client-ca-key-pair -n pgd-group -o yaml > client-ca-key-
pair.yaml
```

1. After removing the content specific to `OpenShift Cluster A` from these secrets (such as uid, resourceVersion, and timestamp), you can

switch context to `OpenShift Cluster B`. Then create the namespace, create the secrets in it, and only then apply the `issuer-selfsigned.yaml` file:

```
oc create ns pgd-
group
oc apply -n pgd-group -f server-ca-key-pair.yaml
oc apply -n pgd-group -f client-ca-key-pair.yaml
oc apply -n pgd-group -f \
  https://raw.githubusercontent.com/EnterpriseDB/edb-postgres-for-kubernetes-
charts/main/hack/samples/issuer-selfsigned.yaml
```

1. You can switch context to `OpenShift Cluster C` and repeat the same process followed for Cluster B:

```
oc create ns pgd-
group
oc apply -n pgd-group -f server-ca-key-pair.yaml
oc apply -n pgd-group -f client-ca-key-pair.yaml
oc apply -n pgd-group -f \
  https://raw.githubusercontent.com/EnterpriseDB/edb-postgres-for-kubernetes-
charts/main/hack/samples/issuer-selfsigned.yaml
```

1. On `OpenShift Cluster A`, you can create your first PGD group, called `region-a`. The YAML manifest for the PGD group is:

```
apiVersion:
pgd.k8s.enterprisedb.io/v1beta1
kind:
PGDGroup
metadata:
  name: region-a
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
      create: true
    discovery:
      - host: region-a-group.pgd-group.svc.clusterset.local
      - host: region-b-group.pgd-group.svc.clusterset.local
      - host: region-c-group.pgd-group.svc.clusterset.local
  cnp:
    storage:
      size:
1Gi
  connectivity:
    dns:
      domain: "pgd-group.svc.clusterset.local"
      additional:
        - domain:
alternate.domain
        - domain: my.domain
          hostSuffix: -
dc1
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
```

```
              kind:
Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind:
Issuer
              group: cert-manager.io
```

!!! Important The format of the hostnames in the `discovery` section differs from the single-cluster example. That's because Submariner is being used to connect the OpenShift clusters, and Submariner uses the `<service>.<ns>.svc.clusterset.local` domain to route traffic between the OpenShift clusters. `region-a-group` is the name of the service to be created for the PGD group named `region-a`.

1. Apply the `region-a` PGD group YAML:

```
oc apply -f region-a.yaml -n pgd-group
```

1. You can now switch context to `OpenShift Cluster B` and create the second PGD group. The YAML for the PGD group in Cluster B is as follows. The only difference is the `metadata.name`.

```
apiVersion:
pgd.k8s.enterprisedb.io/v1beta1
kind:
PGDGroup
metadata:
  name: region-b
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group.pgd-group.svc.clusterset.local
      - host: region-b-group.pgd-group.svc.clusterset.local
      - host: region-c-group.pgd-group.svc.clusterset.local
  cnp:
    storage:
      size:
1Gi
  connectivity:
    dns:
      domain: "pgd-group.svc.clusterset.local"
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind:
Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
```

```
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind:
Issuer
              group: cert-manager.io
```

1. Apply the `region-b` PGD group YAML:

```
oc apply -f region-b.yaml -n pgd-group
```

1. You can switch context to `OpenShift Cluster C` and create the third PGD group. The YAML for the PGD group is:

```
apiVersion:
pgd.k8s.enterprisedb.io/v1beta1
kind:
PGDGroup
metadata:
  name: region-c
spec:
  instances: 0
  proxyInstances: 0
  witnessInstances: 1
  pgd:
    parentGroup:
      name: world
    discovery:
      - host: region-a-group.pgd-group.svc.clusterset.local
      - host: region-b-group.pgd-group.svc.clusterset.local
      - host: region-c-group.pgd-group.svc.clusterset.local
  cnp:
    storage:
      size:
1Gi
  connectivity:
    dns:
      domain: "pgd-group.svc.clusterset.local"
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind:
Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind:
Issuer
              group: cert-manager.io
```

1. Apply the `region-c` PGD group YAML:

```
oc apply -f region-c.yaml -n pgd-group
```

Now you can switch context back to `OpenShift Cluster A` and check the status of the PGD group there:

```
oc get pgdgroup region-a -n pgd-
group
```

The PGD group is in the phase `PGD - Waiting for node discovery`.

After creating the PGD groups in each OpenShift cluster, which in turn creates the services for each node, you need to expose the services to the other OpenShift clusters. You can do this in various ways.

If you're using Submariner, you can do it using the `subctl` command. Run the `subctl export service` command for each service in the `pgd-group` namespace that has a `-group` or `-node` suffix. You can do this by running the following bash `for` loop on each cluster:

```
for service in $(oc get svc -n pgd-group --no-headers -o custom-columns="NAME:.metadata.name" | grep -E
'(-group|-node)$'); do

  subctl export service $service -n pgd-
group
done
```

After a few minutes, the status shows that the PGD group is healthy. Once each PGD group is healthy, you can write to the `app` database in either of the two data nodes: `region-a` or `region-b`. The data is replicated to the other data node.

20 Transparent Data Encryption (TDE)

> **Important**
>
> TDE is available *only* for operands that support it: EPAS versions 15 and newer, Postgres Extended versions 15 and newer.

Transparent Data Encryption, or TDE, is a technology used by several database vendors to **encrypt data at rest**, i.e. database files on disk. TDE does not however encrypt data in use.

TDE is included in EDB Postgres Advanced Server (EPAS) or EDB Postgres Extended, starting with version 15, and it is supported by EDB Postgres Distributed for Kubernetes.

> **Important**
>
> Before you proceed, please take some time to familiarize with theTDE feature in the EPAS documentation.

With TDE activated, both WAL files and files for tables will be encrypted. Data encryption/decryption is entirely transparent to the user, as it is managed by the database without requiring any application changes or updated client drivers.

The support for TDE on EDB Postgres Distributed for Kubernetes relies on the implementation from EDB Postgres for Kubernetes (PG4K). Please refer to the PG4K documentation for the full context.

We show now how to use TDE with a passphrase stored in a Kubernetes Secret, which will be used to encrypt the EPAS binary key.

> **EPAS documentation**
>
> Please refer to the EPAS documentation for details on the EPAS encryption key.

TDE on EDB Postgres Distributed for Kubernetes relies on the PG4K implementation. To activate TDE on a cluster, we use the `epas` section of the manifest, which is within the `cnp` section used for PG4K-level directives such as storage. Use the `tde` stanza to enable TDE, and set the name of

the Kubernetes secret holding the TDE encryption key.

The following YAML portion contains both a secret holding a passphrase (base-64 encoded), and the `epas` section activating TDE with the passphrase.

```
 ---
apiVersion: v1
kind:
Secret
metadata:
  name: tde-key
data:
  key:
bG9zcG9sbGl0b3NkaWNlbnBpb3Bpb3Bpb2N1YW5kb3RpZW5lbmhhbWJyZWN1YW5kb3RpZW5lbmZyaW8=

---
apiVersion:
pgd.k8s.enterprisedb.io/v1beta1
kind:
PGDGroup
[…]
spec:
  instances: 3
[…]
  cnp:
    postgresql:
      epas:
        tde:
          enabled: true
          secretKeyRef:
            name: tde-key
            key:
key
    storage:
      size:
1Gi
```

Again, please refer to the PG4K documentation for additional depth, including how to create the encryption secret and additional ways of using TDE.

As shown in the TDE feature documentation, the information will be encrypted at rest.

For example, open a `psql` terminal into one of your data nodes.

```
kubectl exec -ti <DATA-NODE> -- psql
app
```

and create a new table including a text column.

```
create table foo(bar int, baz
varchar);
insert into foo(bar, baz) values (1, 'hello'), (2,
'goodbye');
```

And then verify the location where the newly defined table is stored on disk:

```
select
pg_relation_filepath('foo');
 pg_relation_filepath
----------------------
 base/16385/16387
```

You can open a terminal on the same data node:

```
kubectl exec -ti <DATA-NODE> --
bash
```

and verify the file has been encrypted.

```
cd $PGDATA/base/16385
hexdump -C 16387 | grep hello
hexdump -C 16387 | grep goodbye
```

21 Examples of configuration

> **Important**
>
> The available examples are for demonstration and experimentation purposes only.

These examples are configuration files for setting up your EDB Postgres Distributed cluster in a Kubernetes environment.

Flexible 3 regions : `flexible_3regions.yaml` : A PGD cluster with two data groups and a global witness node spread across three regions, where each data groups consists of two data nodes and a local witness node.

For a list of available options, see the"API Reference" page.

23 API Reference

Package v1beta1 contains API Schema definitions for the pgd v1beta1 API group

Resource Types

- PGDGroup
- PGDGroupCleanup

CertificateKeystores

Appears in:

- CertificateSpec

CertificateKeystores configures additional keystore output formats to be created in the Certificate's output Secret.

Field	Description

Field	Description
`jks` *JKSKeystore*	JKS configures options for storing a JKS keystore in the `spec.secretName` Secret resource.
`pkcs12` *PKCS12Keystore*	PKCS12 configures options for storing a PKCS12 keystore in the `spec.secretName` Secret resource.

CertificatePrivateKey

Appears in:

- CertificateSpec

CertificatePrivateKey contains configuration options for private keys used by the Certificate controller. This allows control of how private keys are rotated.

Field	Description
`rotationPolicy` *PrivateKeyRotationPolicy*	RotationPolicy controls how private keys should be regenerated when a re-issuance is being processed. If set to Never, a private key will only be generated if one does not already exist in the target `spec.secretName`. If one does exists but it does not have the correct algorithm or size, a warning will be raised to await user intervention. If set to Always, a private key matching the specified requirements will be generated whenever a re-issuance occurs. Default is 'Never' for backward compatibility.
`encoding` *PrivateKeyEncoding*	The private key cryptography standards (PKCS) encoding for this certificate's private key to be encoded in. If provided, allowed values are `PKCS1` and `PKCS8` standing for PKCS#1 and PKCS#8, respectively. Defaults to `PKCS1` if not specified.
`algorithm` *PrivateKeyAlgorithm*	Algorithm is the private key algorithm of the corresponding private key for this certificate. If provided, allowed values are either `RSA`, `Ed25519` or `ECDSA` If `algorithm` is specified and `size` is not provided, key size of 256 will be used for `ECDSA` key algorithm and key size of 2048 will be used for `RSA` key algorithm. key size is ignored when using the `Ed25519` key algorithm.
`size` *int*	Size is the key bit size of the corresponding private key for this certificate. If `algorithm` is set to `RSA`, valid values are `2048`, `4096` or `8192`, and will default to `2048` if not specified. If `algorithm` is set to `ECDSA`, valid values are `256`, `384` or `521`, and will default to `256` if not specified. If `algorithm` is set to `Ed25519`, Size is ignored. No other values are allowed.

CertificateSpec

Appears in:

- CertManagerTemplate

CertificateSpec defines the desired state of Certificate. A valid Certificate requires at least one of a CommonName, DNSName, or URISAN to be valid.

Field	Description
`subject` *X509Subject*	Full X509 name specification (https://golang.org/pkg/crypto/x509/pkix/#Name).
`commonName` *string*	CommonName is a common name to be used on the Certificate. The CommonName should have a length of 64 characters or fewer to avoid generating invalid CSRs. This value is ignored by TLS clients when any subject alt name is set. This is x509 behaviour: https://tools.ietf.org/html/rfc6125#section-6.4.4
`duration` *Duration*	The requested 'duration' (i.e. lifetime) of the Certificate. This option may be ignored/overridden by some issuer types. If unset this defaults to 90 days. Certificate will be renewed either 2/3 through its duration or `renewBefore` period before its expiry, whichever is later. Minimum accepted duration is 1 hour. Value must be in units accepted by Go time.ParseDuration https://golang.org/pkg/time/#ParseDuration
`renewBefore` *Duration*	How long before the currently issued certificate's expiry cert-manager should renew the certificate. The default is 2/3 of the issued certificate's duration. Minimum accepted value is 5 minutes. Value must be in units accepted by Go time.ParseDuration https://golang.org/pkg/time/#ParseDuration
`dnsNames` *[]string*	DNSNames is a list of DNS subjectAltNames to be set on the Certificate.
`ipAddresses` *[]string*	IPAddresses is a list of IP address subjectAltNames to be set on the Certificate.
`uris` *[]string*	URIs is a list of URI subjectAltNames to be set on the Certificate.
`emailAddresses` *[]string*	EmailAddresses is a list of email subjectAltNames to be set on the Certificate.
`secretName` **[Required]** *string*	SecretName is the name of the secret resource that will be automatically created and managed by this Certificate resource. It will be populated with a private key and certificate, signed by the denoted issuer. IMPORTANT: this field was required in the original cert-manager API declaration
`keystores` *CertificateKeystores*	Keystores configures additional keystore output formats stored in the `secretName` Secret resource.
`issuerRef` **[Required]** *ObjectReference*	IssuerRef is a reference to the issuer for this certificate. If the `kind` field is not set, or set to `Issuer`, an Issuer resource with the given name in the same namespace as the Certificate will be used. If the `kind` field is set to `ClusterIssuer`, a ClusterIssuer with the provided name will be used. The `name` field in this stanza is required at all times.

Field	Description
`isCA` *bool*	IsCA will mark this Certificate as valid for certificate signing. This will automatically add the `cert sign` usage to the list of `usages`.
`usages` *[]KeyUsage*	Usages is the set of x509 usages that are requested for the certificate. Defaults to `digital signature` and `key encipherment` if not specified.
`privateKey` *CertificatePrivateKey*	Options to control private keys used for the Certificate.
`encodeUsagesInRequest` *bool*	EncodeUsagesInRequest controls whether key usages should be present in the CertificateRequest
`revisionHistoryLimit` *int32*	revisionHistoryLimit is the maximum number of CertificateRequest revisions that are maintained in the Certificate's history. Each revision represents a single `CertificateRequest` created by this Certificate, either when it was created, renewed, or Spec was changed. Revisions will be removed by oldest first if the number of revisions exceeds this number. If set, revisionHistoryLimit must be a value of `1` or greater. If unset (`nil`), revisions will not be garbage collected. Default value is `nil`.

ConditionStatus

(Alias of `string`)

ConditionStatus represents a condition's status.

JKSKeystore

Appears in:

- CertificateKeystores

JKSKeystore configures options for storing a JKS keystore in the `spec.secretName` Secret resource.

Field	Description
`create` **[Required]** *bool*	Create enables JKS keystore creation for the Certificate. If true, a file named `keystore.jks` will be created in the target Secret resource, encrypted using the password stored in `passwordSecretRef`. The keystore file will only be updated upon re-issuance. A file named `truststore.jks` will also be created in the target Secret resource, encrypted using the password stored in `passwordSecretRef` containing the issuing Certificate Authority

Field	Description
`passwordSecretRef` **[Required]** *SecretKeySelector*	PasswordSecretRef is a reference to a key in a Secret resource containing the password used to encrypt the JKS keystore.

KeyUsage

(Alias of `string`)

Appears in:

- CertificateSpec

KeyUsage specifies valid usage contexts for keys. See: https://tools.ietf.org/html/rfc5280#section-4.2.1.3

```
https://tools.ietf.org/html/rfc5280#section-4.2.1.12
```

Valid KeyUsage values are as follows: "signing", "digital signature", "content commitment", "key encipherment", "key agreement", "data encipherment", "cert sign", "crl sign", "encipher only", "decipher only", "any", "server auth", "client auth", "code signing", "email protection", "s/mime", "ipsec end system", "ipsec tunnel", "ipsec user", "timestamping", "ocsp signing", "microsoft sgc", "netscape sgc"

LocalObjectReference

Appears in:

- SecretKeySelector

LocalObjectReference is a reference to an object in the same namespace as the referent. If the referent is a cluster-scoped resource (e.g. a ClusterIssuer), the reference instead refers to the resource with the given name in the configured 'cluster resource namespace', which is set as a flag on the controller component (and defaults to the namespace that cert-manager runs in).

Field	Description
`name` **[Required]** *string*	Name of the resource being referred to. More info: https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names

ObjectReference

Appears in:

- CertificateSpec

ObjectReference is a reference to an object with a given name, kind and group.

Field	Description
`name` **[Required]** *string*	Name of the resource being referred to.
`group` *string*	Group of the resource being referred to.

PKCS12Keystore

Appears in:

- CertificateKeystores

PKCS12Keystore configures options for storing a PKCS12 keystore in the `spec.secretName` Secret resource.

Field	Description
`create` **[Required]** *bool*	Create enables PKCS12 keystore creation for the Certificate. If true, a file named `keystore.p12` will be created in the target Secret resource, encrypted using the password stored in `passwordSecretRef`. The keystore file will only be updated upon re-issuance. A file named `truststore.p12` will also be created in the target Secret resource, encrypted using the password stored in `passwordSecretRef` containing the issuing Certificate Authority
`passwordSecretRef` **[Required]** *SecretKeySelector*	PasswordSecretRef is a reference to a key in a Secret resource containing the password used to encrypt the PKCS12 keystore.

PrivateKeyAlgorithm

(Alias of `string`)

Appears in:

- CertificatePrivateKey

PrivateKeyAlgorithm represent a private key algorithm

PrivateKeyEncoding

(Alias of `string`)

Appears in:

- CertificatePrivateKey

PrivateKeyEncoding represent a private key encoding

PrivateKeyRotationPolicy

(Alias of `string`)

Appears in:

- CertificatePrivateKey

PrivateKeyRotationPolicy denotes how private keys should be generated or sourced when a Certificate is being issued.

SecretKeySelector

Appears in:

- JKSKeystore
- PKCS12Keystore

SecretKeySelector is a reference to a specific 'key' within a Secret resource. In some instances, `key` is a required field.

Field	Description
`LocalObjectReference` *LocalObjectReference*	(Members of `LocalObjectReference` are embedded into this type.) The name of the Secret resource being referred to.
`key` *string*	The key of the entry in the Secret resource's `data` field to be used. Some instances of this field may be defaulted, in others it may be required.

X509Subject

Appears in:

- CertificateSpec

X509Subject Full X509 name specification

Field	Description

Field	Description
`organizations` *[]string*	Organizations to be used on the Certificate.
`countries` *[]string*	Countries to be used on the Certificate.
`organizationalUnits` *[]string*	Organizational Units to be used on the Certificate.
`localities` *[]string*	Cities to be used on the Certificate.
`provinces` *[]string*	State/Provinces to be used on the Certificate.
`streetAddresses` *[]string*	Street addresses to be used on the Certificate.
`postalCodes` *[]string*	Postal codes to be used on the Certificate.
`serialNumber` *string*	Serial number to be used on the Certificate.

PGDGroup

PGDGroup is the Schema for the pgdgroups API

Field	Description
`apiVersion` **[Required]** string	`pgd.k8s.enterprisedb.io/v1beta1`
`kind` **[Required]** string	`PGDGroup`
`spec` **[Required]** *PGDGroupSpec*	No description provided.
`status` **[Required]** *PGDGroupStatus*	No description provided.

PGDGroupCleanup

PGDGroupCleanup is the Schema for the pgdgroupcleanups API

Field	Description

Field	Description
`apiVersion` **[Required]** string	`pgd.k8s.enterprisedb.io/v1beta1`
`kind` **[Required]** string	`PGDGroupCleanup`
`spec` **[Required]** *PGDGroupCleanupSpec*	No description provided.
`status` **[Required]** *PGDGroupCleanupStatus*	No description provided.

Backup

Appears in:

- PGDGroupSpec

Backup configures the backup of cnp-pgd nodes

Field	Description
`configuration` **[Required]** *BackupConfiguration*	The CNP configuration to be used for backup. ServerName value is reserved by the operator.
`cron` **[Required]** *ScheduledBackupSpec*	The scheduled backup for the data

BackupStatus

Appears in:

- PGDGroupStatus

BackupStatus contains the current status of the pgd backup

Field	Description
`clusterName` **[Required]** *string*	No description provided.
`scheduledBackupName` **[Required]** *string*	No description provided.
`scheduledBackupHash` **[Required]** *string*	No description provided.

CNPStatus

Appears in:

- PGDGroupStatus

CNPStatus contains any relevant status for the operator about CNP

Field	Description
`dataInstances` **[Required]** *int32*	No description provided.
`witnessInstances` **[Required]** *int32*	No description provided.
`firstRecoverabilityPointsByMethod` **[Required]** *map[string]RecoverabilityPointsByMethod*	The recoverability points by method, keyed per CNP clusterName nolint: lll
`firstRecoverabilityPoints` **[Required]** *map[string]string*	The recoverability points, keyed per CNP clusterName, as a date in RFC3339 format
`superUserSecretIsPresent` **[Required]** *bool*	No description provided.
`applicationUserSecretIsPresent` **[Required]** *bool*	No description provided.
`podDisruptionBudgetIsPresent` **[Required]** *bool*	No description provided.

CertManagerTemplate

Appears in:

- ClientCertConfiguration
- ServerCertConfiguration

CertManagerTemplate contains the data to generate a certificate request

Field	Description
`spec` **[Required]** *CertificateSpec*	The Certificate object specification
`metadata` **[Required]** *Metadata*	The label and annotations metadata

ClientCertConfiguration

Appears in:

- TLSConfiguration

ClientCertConfiguration contains the information to generate the certificate for the streaming_replica user

Field	Description
`caCertSecret` **[Required]** *string*	CACertSecret is the secret of the CA to be injected into the CloudNativePG configuration
`certManager` **[Required]** *CertManagerTemplate*	The cert-manager template used to generate the certificates
`preProvisioned` **[Required]** *ClientPreProvisionedCertificates*	PreProvisioned contains how to fetch the pre-generated client certificates

ClientPreProvisionedCertificates

Appears in:

- ClientCertConfiguration

ClientPreProvisionedCertificates instruct how to fetch the pre-generated client certificates

Field	Description
`streamingReplica` **[Required]** *PreProvisionedCertificate*	StreamingReplica the pre-generated certificate for 'streaming_replica' user

CnpBaseConfiguration

Appears in:

- CnpConfiguration
- PGDGroupSpec

CnpBaseConfiguration contains the configuration parameters that can be applied to both CNP Witness and Data nodes

Field	Description

Field	Description
`startDelay` **[Required]** *int32*	The time in seconds that is allowed for a PostgreSQL instance to successfully start up (default 3600)
`stopDelay` **[Required]** *int32*	The time in seconds that is allowed for a PostgreSQL instance node to gracefully shutdown (default 180)
`smartShutdownTimeout` *int32*	The time in seconds that controls the window of time reserved for the smart shutdown of Postgres to complete. Make sure you reserve enough time for the operator to request a fast shutdown of Postgres (that is: `stopDelay` - `smartShutdownTimeout`).
`storage` **[Required]** *StorageConfiguration*	Configuration of the storage of the instances
`walStorage` **[Required]** *StorageConfiguration*	Configuration of the WAL storage for the instances
`clusterMaxStartDelay` **[Required]** *int32*	The time in seconds that is allowed for a PostgreSQL instance to successfully start up (default 300)
`affinity` *AffinityConfiguration*	Affinity/Anti-affinity rules for Pods
`resources` *ResourceRequirements*	Resources requirements of every generated Pod. Please refer to https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ for more information.
`postgresql` *PostgresConfiguration*	Configuration of the PostgreSQL server
`monitoring` **[Required]** *MonitoringConfiguration*	The configuration of the monitoring infrastructure of this cluster
`logLevel` **[Required]** *string*	The instances' log level, one of the following values: error, warning, info (default), debug, trace
`serviceAccountTemplate` **[Required]** *ServiceAccountTemplate*	The service account template to be passed to CNP
`otel` **[Required]** *OTELConfiguration*	OpenTelemetry Configuration
`postInitSQL` **[Required]** *[]string*	List of SQL queries to be executed as a superuser immediately after a node has been created - to be used with extreme care (by default empty)
`postInitTemplateSQL` **[Required]** *[]string*	List of SQL queries to be executed as a superuser in the `template1` after a node has been created - to be used with extreme care (by default empty)

Field	Description
`seccompProfile` **[Required]** *SeccompProfile*	The SeccompProfile applied to every Pod and Container. Defaults to: `RuntimeDefault`
`metadata` **[Required]** *InheritedMetadata*	Metadata applied exclusively to the generated Cluster resources. Useful for applying AppArmor profiles.
`managed` **[Required]** *ManagedConfiguration*	The configuration that is used by the portions of PostgreSQL that are managed by the CNP instance manager

CnpConfiguration

Appears in:

- PGDGroupSpec

CnpConfiguration contains the configurations of the data nodes that will be injected into the resulting clusters composing the PGD group

Field	Description
`CnpBaseConfiguration` *CnpBaseConfiguration*	(Members of `CnpBaseConfiguration` are embedded into this type.) No description provided.
`enableSuperuserAccess` *bool*	When this option is enabled, the CNP operator will create or use the secret defined in the SuperuserSecret to allow superuser (postgres) access to the database. When this option is disabled on a running Group, the operator will ignore the content of the secret and set the password of the `postgres` user to `NULL`. Enabled by default.
`superuserSecret` *LocalObjectReference*	The secret containing the superuser password. A new secret will be created with a randomly generated password if not defined. This field is only allowed in the CNP Instances configuration. A Witness Node will always use the same SuperuserSecret as the other instances.

ConnectionString

(Alias of `map[string]string`)

Appears in:

- PgdConfiguration

ConnectionString represent the parameters to connect to a PostgreSQL cluster

ConnectivityConfiguration

Appears in:

- PGDGroupSpec

ConnectivityConfiguration describes how to generate the services and certificates for the PGDGroup

Field	Description
`dns` **[Required]** *RootDNSConfiguration*	Describes how the FQDN for the resources should be generated
`tls` **[Required]** *TLSConfiguration*	The configuration of the TLS infrastructure
`nodeServiceTemplate` **[Required]** *ServiceTemplate*	Instructs how to generate the service for each node
`groupServiceTemplate` **[Required]** *ServiceTemplate*	Instructs how to generate the service for the PGDGroup
`proxyServiceTemplate` **[Required]** *ServiceTemplate*	Instructs how to generate the service pointing to the PGD Proxy

ConnectivityStatus

Appears in:

- PGDGroupStatus

ConnectivityStatus contains any relevant status for the operator about Connectivity

Field	Description
`replicationTLSCertificate` **[Required]** *ReplicationCertificateStatus*	ReplicationTLSCertificate is the name of the replication TLS certificate, if we have it
`nodeTLSCertificates` **[Required]** *[]NodeCertificateStatus*	NodeTLSCertificates are the names of the certificates that have been created for the PGD nodes
`unusedCertificates` **[Required]** *[]string*	UnusedCertificates are the names of the certificates that we don't use anymore for the PGD nodes
`nodesWithoutCertificates` **[Required]** *[]string*	NodesWithoutCertificates are the names of the nodes which have not a server certificate

Field	Description
`nodesNeedingServiceReconciliation` **[Required]** *[]string*	NodesNeedingServiceReconciliation are the names of the nodes which have not a server certificate
`configurationHash` **[Required]** *string*	ConfigurationHash is the hash code of the connectivity configuration, used to check if we had a change in the configuration or not

DNSConfiguration

Appears in:

- RootDNSConfiguration

DNSConfiguration describes how the FQDN for the resources should be generated

Field	Description
`domain` **[Required]** *string*	Contains the domain name of by all services in the PGDGroup. It is responsibility of the user to ensure that the value specified here matches with the rendered nodeServiceTemplate and groupServiceTemplate
`hostSuffix` **[Required]** *string*	Contains an optional suffix to add to all the service names in the PGDGroup. The meaning of this setting it to allow the user to easily mark all the services created in a location for routing purpose (i.e., add a generic rule to CoreDNS to rewrite some service suffixes as local)

DiscoveryJobConfig

Appears in:

- PgdConfiguration

DiscoveryJobConfig contains a series of fields that configure the discovery job

Field	Description
`delay` **[Required]** *int*	Delay amount of time to sleep between retries, measured in seconds
`retries` **[Required]** *int*	Retries how many times the operation should be retried
`timeout` **[Required]** *int*	Timeout amount of time given to the operation to succeed, measured in seconds

InheritedMetadata

Appears in:

- CnpBaseConfiguration
- PGDGroupSpec

InheritedMetadata contains metadata to be inherited by all resources related to a Cluster

Field	Description
`labels` **[Required]** *map[string]string*	No description provided.
`annotations` **[Required]** *map[string]string*	No description provided.

Metadata

Appears in:

- CertManagerTemplate
- ServiceTemplate

Metadata is a structure similar to the metav1.ObjectMeta, but still parseable by controller-gen to create a suitable CRD for the user.

Field	Description
`labels` *map[string]string*	Map of string keys and values that can be used to organize and categorize (scope and select) objects. May match selectors of replication controllers and services. More info: http://kubernetes.io/docs/user-guide/labels
`annotations` *map[string]string*	Annotations is an unstructured key value map stored with a resource that may be set by external tools to store and retrieve arbitrary metadata. They are not queryable and should be preserved when modifying objects. More info: http://kubernetes.io/docs/user-guide/annotations

NodeCertificateStatus

Appears in:

- ConnectivityStatus

NodeCertificateStatus encapsulate the status of the server certificate of a CNP node

Field	Description
`ReplicationCertificateStatus` *ReplicationCertificateStatus*	(Members of `ReplicationCertificateStatus` are embedded into this type.) No description provided.
`nodeName` **[Required]** *string*	NodeName is the name of the CNP cluster using this certificate

NodeKindName

(Alias of `string`)

Appears in:

- NodeSummary

NodeKindName is a type containing the potential values of node_kind_name from bdr.node_summary

NodeSummary

Appears in:

- PGDGroupStatus

NodeSummary shows relevant info from bdr.node_summary

Field	Description
`node_name` **[Required]** *string*	Name of the node
`node_group_name` **[Required]** *string*	NodeGroupName is the name of the joined group
`peer_state_name` **[Required]** *string*	Consistent state of the node in human-readable form
`peer_target_state_name` **[Required]** *string*	State which the node is trying to reach (during join or promotion)
`node_kind_name` **[Required]** *NodeKindName*	The kind of node: witness or data

NodesExtensionsStatus

(Alias of `[]github.com/EnterpriseDB/pg4k-pgd/api/v1beta1.NodeExtensionStatus`)

NodesExtensionsStatus contains a list of NodeExtensionStatus entries

OTELConfiguration

Appears in:

- CnpBaseConfiguration

OTELConfiguration is the configuration for external openTelemetry

Field	Description
`metricsURL` **[Required]** *string*	The OpenTelemetry HTTP endpoint URL to accept metrics data
`traceURL` **[Required]** *string*	The OpenTelemetry HTTP endpoint URL to accept trace data
`traceEnable` **[Required]** *bool*	Whether to push trace data to OpenTelemetry traceUrl
`tls` **[Required]** *OTELTLSConfiguration*	TLSConfiguration provides the TLS certificate configuration when MetricsURL and TraceURL are using HTTPS

OTELTLSConfiguration

Appears in:

- OTELConfiguration

OTELTLSConfiguration contains the certificate configuration for TLS connections to openTelemetry

Field	Description
`caBundleSecretRef` **[Required]** *SecretKeySelector*	CABundleSecretRef is a reference to a secret field containing the CA bundle to verify the openTelemetry server certificate
`clientCertSecret` **[Required]** *LocalObjectReference*	ClientCertSecret is the name of the secret containing the client certificate used to connect to openTelemetry. It must contain both the standard "tls.crt" and "tls.key" files, encoded in PEM format.

PGDGroupCleanupSpec

Appears in:

- PGDGroupCleanup

PGDGroupCleanupSpec defines the desired state of PGDGroupCleanup

Field	Description
`executor` **[Required]** *string*	No description provided.
`target` **[Required]** *string*	No description provided.
`force` **[Required]** *bool*	Force will force the removal of the PGDGroup even if the target PGDGroup nodes are not parted

PGDGroupCleanupStatus

Appears in:

- PGDGroupCleanup

PGDGroupCleanupStatus defines the observed state of PGDGroupCleanup

Field	Description
`phase` **[Required]** *github.com/EnterpriseDB/pg4k-pgd/pkg/resources.OperatorPhaseCleanup*	No description provided.

PGDGroupSpec

Appears in:

- PGDGroup

PGDGroupSpec defines the desired state of PGDGroup

Field	Description
`imageName` **[Required]** *string*	Name of the container image, supporting both tags (`<image>:<tag>`) and digests for deterministic and repeatable deployments (`<image>:<tag>@sha256:<digestValue>`)

Field	Description
`imagePullPolicy` *PullPolicy*	Image pull policy. One of `Always`, `Never` or `IfNotPresent`. If not defined, it defaults to `IfNotPresent`. Cannot be updated. More info: https://kubernetes.io/docs/concepts/containers/images#updating-images
`imagePullSecrets` **[Required]** *[]LocalObjectReference*	The list of pull secrets to be used to pull operator and or the operand images
`inheritedMetadata` **[Required]** *InheritedMetadata*	Metadata that will be inherited by all objects related to the pgdGroup
`instances` **[Required]** *int32*	Number of instances required in the cluster
`proxyInstances` **[Required]** *int32*	Number of proxy instances required in the cluster
`witnessInstances` **[Required]** *int32*	Number of witness instances required in the cluster
`backup` **[Required]** *Backup*	The configuration to be used for backups in the CNP instances.
`restore` **[Required]** *Restore*	The configuration to restore this PGD group from an Object Store service
`cnp` **[Required]** *CnpConfiguration*	Instances configuration that will be injected into the CNP clusters that compose the PGD Group
`witness` **[Required]** *CnpBaseConfiguration*	WitnessInstances configuration that will be injected into the WitnessInstances CNP clusters If not defined, it will default to the Instances configuration
`pgd` **[Required]** *PgdConfiguration*	Pgd contains instructions to bootstrap this cluster
`pgdProxy` **[Required]** *PGDProxyConfiguration*	PGDProxy contains instructions to configure PGD Proxy
`connectivity` **[Required]** *ConnectivityConfiguration*	Configures the connectivity of the PGDGroup, like services and certificates that will be used.
`failingFinalizerTimeLimitSeconds` **[Required]** *int32*	The amount of seconds that the operator will wait in case of a failing finalizer. A finalizer is considered failing when the operator cannot reach any nodes of the PGDGroup

PGDGroupStatus

Appears in:

- PGDGroup

PGDGroupStatus defines the observed state of PGDGroup

Field	Description
`latestGeneratedNode` **[Required]** *int32*	ID of the latest generated node (used to avoid node name clashing)
`phase` **[Required]** *github.com/EnterpriseDB/pg4k-pgd/pkg/resources.OperatorPhase*	The initialization phase of this cluster
`phaseDetails` **[Required]** *string*	The details of the current phase
`phaseTroubleshootHints` **[Required]** *string*	PhaseTroubleshootHints general troubleshooting indications for the given phase
`phaseType` **[Required]** *github.com/EnterpriseDB/pg4k-pgd/pkg/resources.PhaseType*	PhaseType describes the phase category.
`conditions` **[Required]** *[]Condition*	Conditions for PGDGroup object
`nodes` **[Required]** *[]NodeSummary*	The list of summaries for the nodes in the group
`backup` **[Required]** *BackupStatus*	The node that is taking backups of this PGDGroup
`restore` **[Required]** *RestoreStatus*	The status of the restore process
`PGD` **[Required]** *PGDStatus*	Last known status of PGD
`CNP` **[Required]** *CNPStatus*	Last known status of CNP
`PGDProxy` **[Required]** *PGDProxyStatus*	Last known status of PGDProxy
`connectivity` **[Required]** *ConnectivityStatus*	Last known status of Connectivity
`pause` **[Required]** *PauseStatus*	Last known status of Pause

PGDNodeGroupEntry

Appears in:

- PGDStatus

PGDNodeGroupEntry shows information about the node groups available in the PGD configuration

Field	Description
`name` **[Required]** *string*	Name is the name of the node group
`enableProxyRouting` **[Required]** *bool*	EnableProxyRouting is true is the node group allows running PGD Proxies
`enableRaft` **[Required]** *bool*	EnableRaft is true if the node group has a subgroup raft instance
`routeWriterMaxLag` **[Required]** *int64*	RouteWriterMaxLag Maximum lag in bytes of the new write candidate to be selected as write leader, if no candidate passes this, there will be no writer selected automatically
`routeReaderMaxLag` **[Required]** *int64*	RouteReaderMaxLag Maximum lag in bytes for node to be considered viable read-only node
`routeWriterWaitFlush` **[Required]** *bool*	RouteWriterWaitFlush Whether to wait for replication queue flush before switching to new leader when using `bdr.routing_leadership_transfer()`

PGDNodeGroupSettings

Appears in:

- PgdConfiguration

PGDNodeGroupSettings contains the settings of the PGD Group

Field	Description
`routeWriterMaxLag` **[Required]** *int64*	RouteWriterMaxLag Maximum lag in bytes of the new write candidate to be selected as write leader, if no candidate passes this, there will be no writer selected automatically Defaults to -1
`routeReaderMaxLag` **[Required]** *int64*	RouteReaderMaxLag Maximum lag in bytes for node to be considered viable read-only node Defaults to -1
`routeWriterWaitFlush` **[Required]** *bool*	RouteWriterWaitFlush Whether to wait for replication queue flush before switching to new leader when using `bdr.routing_leadership_transfer()` Defaults to false

PGDProxyConfiguration

Appears in:

- PGDGroupSpec

PGDProxyConfiguration defines the configuration of PGD Proxy

Field	Description
`imageName` **[Required]** *string*	Name of the PGDProxy container image
`logLevel` **[Required]** *string*	The PGD Proxy log level, one of the following values: error, warning, info (default), debug, trace
`logEncoder` **[Required]** *string*	The format of the log output
`proxyAffinity` **[Required]** *Affinity*	ProxyAffinity/Anti-affinity rules for pods
`proxyNodeSelector` **[Required]** *map[string]string*	ProxyNodeSelector rules for pods
`proxyTolerations` **[Required]** *[]Toleration*	ProxyTolerations rules for pods
`proxyResources` *ResourceRequirements*	Defines the resources assigned to the proxy. If not defined uses defaults requests and limits values.

PGDProxyEntry

Appears in:

- PGDStatus

PGDProxyEntry shows information about the proxies available in the PGD configuration

Field	Description
`name` **[Required]** *string*	Name is the name of the proxy
`fallbackGroupNames` **[Required]** *[]string*	FallbackGroupNames are the names of the fallback groups configured for this proxy

Field	Description
`parentGroupName` **[Required]** *string*	ParentGroupName is the parent PGD group of this proxy
`maxClientConn` **[Required]** *int*	MaxClientConn maximum number of connections the proxy will accept
`maxServerConn` **[Required]** *int*	MaxServerConn maximum number of connections the proxy will make to the Postgres node
`serverConnTimeout` **[Required]** *int64*	ServerConnTimeout connection timeout for server connections in seconds
`serverConnKeepalive` **[Required]** *int64*	ServerConnKeepalive keepalive interval for server connections in seconds
`fallbackGroupTimeout` **[Required]** *int64*	FallbackGroupTimeout the interval after which the routing falls back to one of the fallback_groups
`consensusGracePeriod` **[Required]** *int64*	ConsensusGracePeriod the duration in seconds for which proxy continues to route even upon loss of a Raft leader.

PGDProxySettings

Appears in:

- PgdConfiguration

PGDProxySettings contains the settings of the proxy

Field	Description
`fallbackGroups` **[Required]** *[]string*	FallbackGroups is the list of groups the proxy should forward connection to when all the data nodes of this PGD group are not available
`maxClientConn` **[Required]** *int*	MaxClientConn maximum number of connections the proxy will accept. Defaults to 32767
`maxServerConn` **[Required]** *int*	MaxServerConn maximum number of connections the proxy will make to the Postgres node. Defaults to 32767
`serverConnTimeout` **[Required]** *int64*	ServerConnTimeout connection timeout for server connections in seconds. Defaults to 2
`serverConnKeepalive` **[Required]** *int64*	ServerConnKeepalive keepalive interval for server connections in seconds. Defaults to 10

Field	Description
`fallbackGroupTimeout` **[Required]** *int64*	FallbackGroupTimeout the interval after which the routing falls back to one of the fallback_groups. Defaults to 60
`consensusGracePeriod` **[Required]** *int64*	ConsensusGracePeriod the duration in seconds for which proxy continues to route even upon loss of a Raft leader. If set to 0s, proxy stops routing immediately. Defaults to 6

PGDProxyStatus

Appears in:

- PGDGroupStatus

PGDProxyStatus any relevant status for the operator about PGDProxy

Field	Description
`proxyInstances` **[Required]** *int32*	No description provided.
`writeLead` **[Required]** *string*	WriteLead is a reserved field for the operator, is not intended for external usage. Will be removed in future versions
`proxyHash` **[Required]** *string*	ProxyHash contains the hash we use to detect if we need to reconcile the proxies

PGDStatus

Appears in:

- PGDGroupStatus

PGDStatus any relevant status for the operator about PGD

Field	Description
`raftConsensusLastChangedStatus` **[Required]** *github.com/EnterpriseDB/pg4k-pgd/pkg/resources.PGDRaftStatus*	RaftConsensusLastChangedStatus indicates the latest reported status from bdr.monitor_group_raft
`raftConsensusLastChangedMessage` **[Required]** *string*	RaftConsensusLastChangedMessage indicates the latest reported message from bdr.monitor_group_raft

Field	Description
`raftConsensusLastChangedTimestamp` **[Required]** *string*	RaftConsensusLastChangedTimestamp indicates when the status and message were first reported
`registeredProxies` **[Required]** *[]PGDProxyEntry*	RegisteredProxies is the status of the registered proxies
`nodeGroup` **[Required]** *PGDNodeGroupEntry*	NodeGroup is the status of the node group associated with the PGDGroup

ParentGroupConfiguration

Appears in:

- PgdConfiguration

ParentGroupConfiguration contains the topology configuration of PGD

Field	Description
`name` **[Required]** *string*	Name of the parent group
`create` **[Required]** *bool*	Create is true when the operator should create the parent group if it doesn't exist

PauseStatus

Appears in:

- PGDGroupStatus

PauseStatus contains the information of group hibernating

Field	Description
`active` **[Required]** *bool*	Active indicates the PGDGroup is either: - in process of pausing - already paused - in process of resuming
`instances` **[Required]** *int32*	Instances is the number of paused PGD instances

Field	Description
`lastStartedTime` **[Required]** *Time*	LastStartedTime is the last time the PGDGroup started pausing
`lastCompletedTime` **[Required]** *Time*	LastCompletedTime is last time the PGDGroup completed pausing
`lastResumeStartedTime` **[Required]** *Time*	LastResumeStartedTime is the last time the PGDGroup started resuming
`lastResumeCompletedTime` **[Required]** *Time*	LastCompletedTime is last time the PGDGroup completed resuming

PgdConfiguration

Appears in:

- PGDGroupSpec

PgdConfiguration is the configuration of the PGD group structure

Field	Description
`parentGroup` **[Required]** *ParentGroupConfiguration*	ParentGroup configures the topology of the PGD group
`discovery` **[Required]** *[]ConnectionString*	The parameters we will use to connect to a node belonging to the parent PGD group. Even if provided, the following parameters will be overridden with default values: `application_name`, `sslmode`, `dbname` and `user`. The following parameters should not be provided nor used, as they are not even overridden with defaults: `sslkey`, `sslcert`, `sslrootcert`
`discoveryJob` **[Required]** *DiscoveryJobConfig*	DiscoveryJob the configuration of the PGD Discovery job
`databaseName` **[Required]** *string*	Name of the database used by the application. Default: `app`.
`ownerName` **[Required]** *string*	Name of the owner of the database in the instance to be used by applications. Defaults to the value of the `database` key.
`ownerCredentialsSecret` **[Required]** *LocalObjectReference*	Name of the secret containing the initial credentials for the owner of the user database. If empty a new secret will be created from scratch
`proxySettings` **[Required]** *PGDProxySettings*	Configuration for the proxy

Field	Description
`nodeGroupSettings` **[Required]** *PGDNodeGroupSettings*	Configuration for the PGD Group
`globalRouting` **[Required]** *bool*	GlobalRouting is true when global routing is enabled, and in this case the proxies will be created in the parent group
`mutations` **[Required]** *SQLMutations*	List of SQL mutations to apply to the node group

PreProvisionedCertificate

Appears in:

- ClientPreProvisionedCertificates

PreProvisionedCertificate contains the data needed to supply a pre-generated certificate

Field	Description
`secretRef` **[Required]** *string*	SecretRef a name pointing to a secret that contains a tls.crt and tls.key

RecoverabilityPointsByMethod

(Alias of `map[github.com/EnterpriseDB/cloud-native-postgres/api/v1.BackupMethod]k8s.io/apimachinery/pkg/apis/meta/v1.Time`)

Appears in:

- CNPStatus

RecoverabilityPointsByMethod contains the first recoverability points for a given backup method

ReplicationCertificateStatus

Appears in:

- ConnectivityStatus
- NodeCertificateStatus

ReplicationCertificateStatus encapsulate the certificate status

Field	Description
`name` **[Required]** *string*	Name is the name of the certificate
`hash` **[Required]** *string*	Hash is the hash of the configuration for which it has been generated
`isReady` **[Required]** *bool*	Ready is true when the certificate is ready
`preProvisioned` **[Required]** *bool*	PreProvisioned is true if the certificate is preProvisioned

Restore

Appears in:

- PGDGroupSpec

Restore configures the restore of a PGD group from an object store

Field	Description
`volumeSnapshots` *VolumeSnapshotsConfiguration*	The configuration for volumeSnapshot restore
`barmanObjectStore` **[Required]** *BarmanObjectStoreConfiguration*	The configuration for the barman-cloud tool suite
`recoveryTarget` **[Required]** *RecoveryTarget*	By default, the recovery process applies all the available WAL files in the archive (full recovery). However, you can also end the recovery as soon as a consistent state is reached or recover to a point-in-time (PITR) by specifying a `RecoveryTarget` object, as expected by PostgreSQL (i.e., timestamp, transaction Id, LSN, ...). More info: https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET
`serverNames` **[Required]** *[]string*	The list of server names to be used as a recovery origin. One of these servers will be elected as the seeding one when evaluating the recovery target, this option is only used when restore from barmanObjectStore.

RestoreStatus

Appears in:

- PGDGroupStatus

RestoreStatus contains the current status of the restore process

Field	Description
`serverName` **[Required]** *string*	The name of the server to be restored
`VolumeSnapshots` **[Required]** *[]VolumeSnapshotRestoreStatus*	selected volumeSnapshots to restore

RootDNSConfiguration

Appears in:

- ConnectivityConfiguration

RootDNSConfiguration describes how the FQDN for the resources should be generated

Field	Description
`DNSConfiguration` *DNSConfiguration*	(Members of `DNSConfiguration` are embedded into this type.) No description provided.
`additional` **[Required]** *[]DNSConfiguration*	AdditionalDNSConfigurations adds more possible FQDNs for the resources

SQLMutation

SQLMutation is a series of SQL statements to apply atomically

Field	Description
`isApplied` **[Required]** *[]string*	List of boolean-returning SQL queries. If any of them returns false the mutation will be applied
`exec` **[Required]** *[]string*	List of SQL queries to be executed to apply this mutation
`type` *SQLMutationType*	Type determines when the SQLMutation occurs. 'always': reconcile the mutation at each reconciliation cycle 'beforeSubgroupRaft': are executed only before the subgroupRaft is enabled If not specified, the Type defaults to 'always'.

SQLMutationType

(Alias of `string`)

Appears in:

- SQLMutation

SQLMutationType a supported type of SQL Mutation

SQLMutations

(Alias of `[]github.com/EnterpriseDB/pg4k-pgd/api/v1beta1.SQLMutation`)

Appears in:

- PgdConfiguration

SQLMutations A list of SQLMutation

ScheduledBackupSpec

Appears in:

- Backup

ScheduledBackupSpec defines the desired state of ScheduledBackup

Field	Description
`suspend` **[Required]** *bool*	If this backup is suspended or not
`immediate` **[Required]** *bool*	If the first backup has to be immediately start after creation or not
`schedule` **[Required]** *string*	The schedule does not follow the same format used in Kubernetes CronJobs as it includes an additional second specifier, see https://pkg.go.dev/github.com/robfig/cron#hdr-CRON_Expression_Format

Field	Description
`backupOwnerReference` **[Required]** *string*	Indicates which ownerReference should be put inside the created backup resources. - none: no owner reference for created backup objects (same behavior as before the field was introduced) - self: sets the Scheduled backup object as owner of the backup - cluster: set the cluster as owner of the backup
`target` **[Required]** *BackupTarget*	The policy to decide which instance should perform this backup. If empty, it defaults to `cluster.spec.backup.target`. Available options are empty string, `primary` and `prefer-standby`. `primary` to have backups run always on primary instances, `prefer-standby` to have backups run preferably on the most updated standby, if available.
`method` *BackupMethod*	The backup method to be used, possible options are `barmanObjectStore` and `volumeSnapshot`. Defaults to: `barmanObjectStore`.
`online` *bool*	Whether the default type of backup with volume snapshots is online/hot (`true`, default) or offline/cold (`false`) Overrides the default setting specified in the cluster field '.spec.backup.volumeSnapshot.online'
`onlineConfiguration` *OnlineConfiguration*	Configuration parameters to control the online/hot backup with volume snapshots Overrides the default settings specified in the cluster '.backup.volumeSnapshot.onlineConfiguration' stanza

ServerCertConfiguration

Appears in:

- TLSConfiguration

ServerCertConfiguration contains the information to generate the certificates for the nodes

Field	Description
`caCertSecret` **[Required]** *string*	CACertSecret is the secret of the CA to be injected into the CloudNativePG configuration
`certManager` **[Required]** *CertManagerTemplate*	The cert-manager template used to generate the certificates

ServiceTemplate

Appears in:

- ConnectivityConfiguration

ServiceTemplate is a structure that allows the user to set a template for the Service generation.

Field	Description
`metadata` *Metadata*	Standard object's metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata
`spec` *ServiceSpec*	Specification of the desired behavior of the service. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status
`updateStrategy` *ServiceUpdateStrategy*	UpdateStrategy indicates how to update the services generated by this template.

ServiceUpdateStrategy

(Alias of `string`)

Appears in:

- ServiceTemplate

ServiceUpdateStrategy defines the type for updating LoadBalancers. Allowed values are "patch" and "replace".

TLSConfiguration

Appears in:

- ConnectivityConfiguration

TLSConfiguration is the configuration of the TLS infrastructure used by PGD to connect to the nodes

Field	Description
`mode` **[Required]** *TLSMode*	No description provided.
`serverCert` **[Required]** *ServerCertConfiguration*	The configuration for the server certificates
`clientCert` **[Required]** *ClientCertConfiguration*	The configuration for the client certificates

TLSMode

(Alias of `string`)

Appears in:

- TLSConfiguration

TLSMode describes which mode should be used for the node to node communications

VolumeSnapshotRestoreStatus

Appears in:

- RestoreStatus

VolumeSnapshotRestoreStatus the volumeSnapshot to restore

Field	Description
`snapshotName` **[Required]** *string*	SnapshotName is the snapshot name to restore
`pvcRole` **[Required]** *github.com/EnterpriseDB/cloud-native-postgres/pkg/utils.PVCRole*	PVCRole is the pvcRole snapshot to restore

VolumeSnapshotsConfiguration

Appears in:

- Restore

VolumeSnapshotsConfiguration contains the configuration for the volumeSnapshots restore

Field	Description
`selector` **[Required]** *LabelSelector*	Label selector used to select the volumeSnapshot to restore

24 Supported versions

This page lists the status for currently supported releases of EDB Postgres Distributed for Kubernetes.

Support status of EDB Postgres for Kubernetes releases

Version	Currently Supported	Release Date	End of Life	Supported Kubernetes Versions	Supported OpenShift Versions	Supported Postgres versions
1.0	Yes	April 24, 2024	-	1.26 -> 1.29	4.12 -> 4.14	12 -> 16

The Postgres (operand) versions are limited to those supported by EDB Postgres Distributed (PGD).

> **Important**
>
> Please be aware that this page is informative only. The "Platform Compatibility" page from the EDB website contains the official list of supported software and Kubernetes distributions.

25 Known issues and limitations

These known issues and limitations are in the current release of EDB Postgres Distributed for Kubernetes.

Postgres major version upgrades

This version of EDB Postgres Distributed for Kubernetes **doesn't support** major version upgrades of Postgres.

Data migration

This version of EDB Postgres Distributed for Kubernetes **doesn't support** migrating from existing Postgres databases.

Connectivity with PgBouncer

EDB Postgres Distributed for Kubernetes **does not support** using PgBouncer to pool client connection requests. This limitation applies to both the open-source and EDB versions of PgBouncer.

Backup operations

To configure an EDB Postgres Distributed for Kubernetes environment, you must apply a `PGDGroup` YAML object to each Kubernetes cluster. Applying this object creates all necessary services for implementing a distributed architecture.

If you added a `spec.backup` section to this `PGDGroup` object with the goal of setting up a backup configuration, the backup will fail unless you also set the `spec.backup.cron.schedule` value.

Error output example:

```
The PGDGroup "region-a" is invalid: spec.backup.cron.schedule: Invalid value: "": Empty spec string
```

Workaround

To work around this issue, add a `spec.backup.cron.schedule` section with a schedule that meets your requirements, for example:

```
spec:
  instances: 3
  proxyInstances: 2
  pgd:
    parentGroup:
      create: true
      name: world
  backup:
    configuration:
      barmanObjectStore:
        ...
    cron:
      suspend: false
      immediate: true
      schedule: "0 */5 * * *
*"
```

Known issues and limitations in EDB Postgres Distributed

All issues and limitations known for the EDB Postgres Distributed version that you include in your deployment also affect your EDB Postgres Distributed for Kubernetes instance.

For example, if the EDB Postgres Distributed version you're using is 5.x, your EDB Postgres Distributed for Kubernetes instance will be affected by these 5.x known issues and 5.x limitations.