PGSession 2025

# couverte de l’opérateur Kubernetes CloudNative


0.1

# Contents

# 1/ Introduction

## 1.1 OBJECTIF DE L'ATELIER

- Découverte et prise en main de l'opérateur CloudNativePG
- Déploiement d'instances PostgreSQL via l'opérateur
- Tests et découvertes de fonctionnalités


L'objectif de cet atelier est de découvrir les opérateurs PostgreSQL sur Kubernetes et en particulier l'opérateur CloudNativePG. Une présentation générale est faite pour évoquer certains aspects des opérateurs et de leur utilisation.

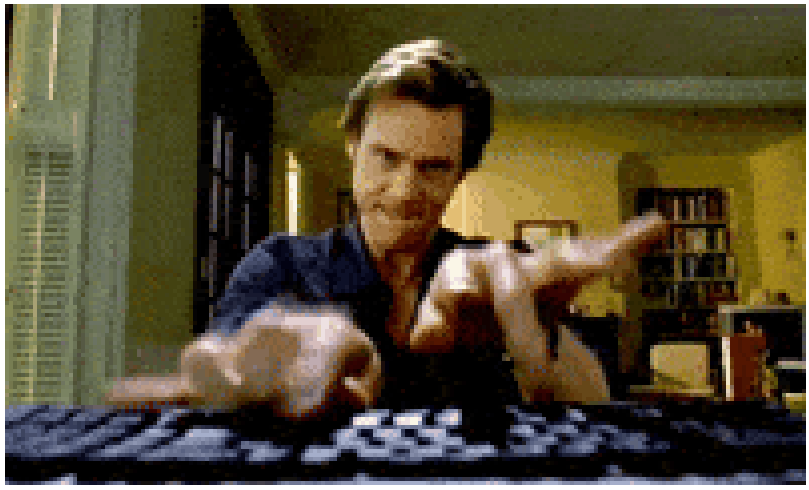
Le TP vous permettra d'installer l'opérateur CloudNativePG, de déployer un cluster PostgreSQL et d'effectuer différentes opérations comme la mise en place de sauvegardes S3 et la restauration.

---

## 1.2 DÉROULÉ DE L'ATELIER

Durée : ~ 3 heures
Pause : 11h00

1. Quelques mots sur PostgreSQL dans Kubernetes
2. L'opérateur CloudNativePG
3. **Travaux pratiques** : accompagné, avoir le même rythme.



---

## 1.3 PRÉREQUIS DE L'ATELIER

- Un terminal Linux
- Une VM Debian Bookworm ou équivalente (mise à disposition par Dalibo)
- Un peu de compétences Linux, PostgreSQL, image Docker et Kubernetes :)
- Outils utilisés :
  - ssh, minikube, kubectl, vi (nano, emacs ou autre), …

Des connaissances de base pour chacun des éléments présents dans cette liste sont suffisantes pour pouvoir suivre le contenu proposé dans cet atelier sans difficulté.

---

# 2/ Kubernetes

## 2.1 PRÉSENTATION

- Plateforme de déploiement de conteneurs libre et *Open Source*
- Cloud Native Computing Foundation (CNCF)
- Plusieurs distributions existent (K3s, AKS, GKE, Rancher, Openshift, **Minikube**…)
- Déploiements applicatifs, auto-scaling, redéploiement automatique, *load balancing*, …

Kubernetes est de plus en plus présent dans les infrastructures techniques des départements/services IT. Grâce à ses fonctionnalités, comme l'*auto-scaling* ou le redéploiement automatique, il est plébiscité pour le déploiement des applications dites *stateless*.

De plus en plus d'outils liés à la donnée sont déployés dans Kubernetes. Les systèmes de gestion de bases de données "classiques" n'y coupent pas.

---

## 2.2 DES DONNÉES DANS KUBERNETES ?!

- Pas une évidence
- Habituellement du *stateless*
- Apparition des `StatefulSet`
- Couche supplémentaire (complexité ?)
- Effet de mode ? Pas que !

Choisissez surtout une solution qui correspond à vos besoins !

Historiquement, le déploiement d'instances PostgreSQL ou autre système de gestion de bases de données (SGBD) en général, dans Kubernetes, peut sembler tout sauf évident, voire même contre-intuitif. Cependant, l'évolution de Kubernetes, avec l'apparition des `StatefulSet`, et surtout des opérateurs, offrent aujourd'hui des solutions viables pour le déploiement d'applications *stateful*.

La simplification des déploiements et les fonctionnalités proposées vont de pair avec une couche d'abstraction et de complexité supplémentaire.

---

## 2.3 DÉPLOYER POSTGRESQL DANS KUBERNETES

- Nécessite de choisir une image Docker contenant PostgreSQL
- Dockerhub[1] ? image maison ?

Vient ensuite le déploiement :

- Manuel
- `StatefulSet`
- Helm Chart
- Opérateur (leurs propres images)

Il existe différentes possibilités pour déployer une instance PostgreSQL dans Kubernetes. Cela peut très bien se faire manuellement (i.e avec votre propre image et avec un déploiement), via un `StatefulSet`, avec un Helm Chart (sorte de définition packagée d'une application sur Kubernetes) ou enfin grâce à un opérateur.

Un opérateur apporte de nombreuses fonctionnalités et est spécifique à PostgreSQL, contrairement à un `StatefulSet` qui reste très général. Les Helm Chart automatisent beaucoup de choses mais n'apportent pas autant de fonctionnalités que les opérateurs (notamment concernant la partie automatisation). Ces derniers apportent une vraie plus-value sur la gestion de PostgreSQL à titre d'exemples :

- Gestion des différents volumes (PGDATA, WAL, …) ;
- Gestion des opérations de bascules (manuelles ou automatiques) ;
- Mise en place de réplications ;
- …

## 2.4 OPÉRATEURS POSTGRESQL

- Extension des fonctionnalités et des ressources de Kubernetes
- Plusieurs opérateurs (~8) pour PostgreSQL sur https://operatorhub.io
- Maturité variable en fonction des projets

https://sdk.operatorframework.io/docs/overview/operator-capabilities/

Un opérateur étend les possibilités d'un cluster Kubernetes grâce à la définition de `CustomResourceDefinition`. Par exemple pour CloudNativePG :

```
kubectl api-resources | grep cnpg
backups                    postgresql.cnpg.io/v1       true         Backup
clusterimagecatalogs       postgresql.cnpg.io/v1       false        ClusterImageCatalog
clusters                   postgresql.cnpg.io/v1       true         Cluster
imagecatalogs              postgresql.cnpg.io/v1       true         ImageCatalog
poolers                    postgresql.cnpg.io/v1       true         Pooler
scheduledbackups           postgresql.cnpg.io/v1       true         ScheduledBackup
```

Les opérateurs existants ont chacun leurs spécificités et particularités. Le site `operatorhub` liste ceux qui existent : https://operatorhub.io/?keyword=postgresql

Leur maturité est différente, allant du niveau 1, correspondant à des opérateurs facilitant l'installation et la configuration, jusqu'au niveau 5 où un opérateur se doit de pouvoir faire face à des situations plus complexes (*auto-healing*, bascule automatique, …). Voir à ce propos : https://sdk.operatorframework.io/docs/overview/operator-capabilities/

## 2.5

# 3/ L’opérateur CloudNativePG

CloudNativePG is the Kubernetes operator that covers the full lifecycle of a highly available PostgreSQL database cluster with a primary/standby architecture, using native streaming replication.

---

## 3.1 LE PROJET


- https://cloudnative-pg.io/
- Débuté par 2ndQuadrant puis EDB. Enfin libéré en 2022
- Projet open source, licence Apache 2.0
- Mode de gouvernance similaire à PostgreSQL

Le projet a été initialement développé par EDB puis libéré en 2022 pour la communauté. La gouvernance du projet se rapproche de celle du projet PostgreSQL avec une *core team* et l'ouverture à la contribution.

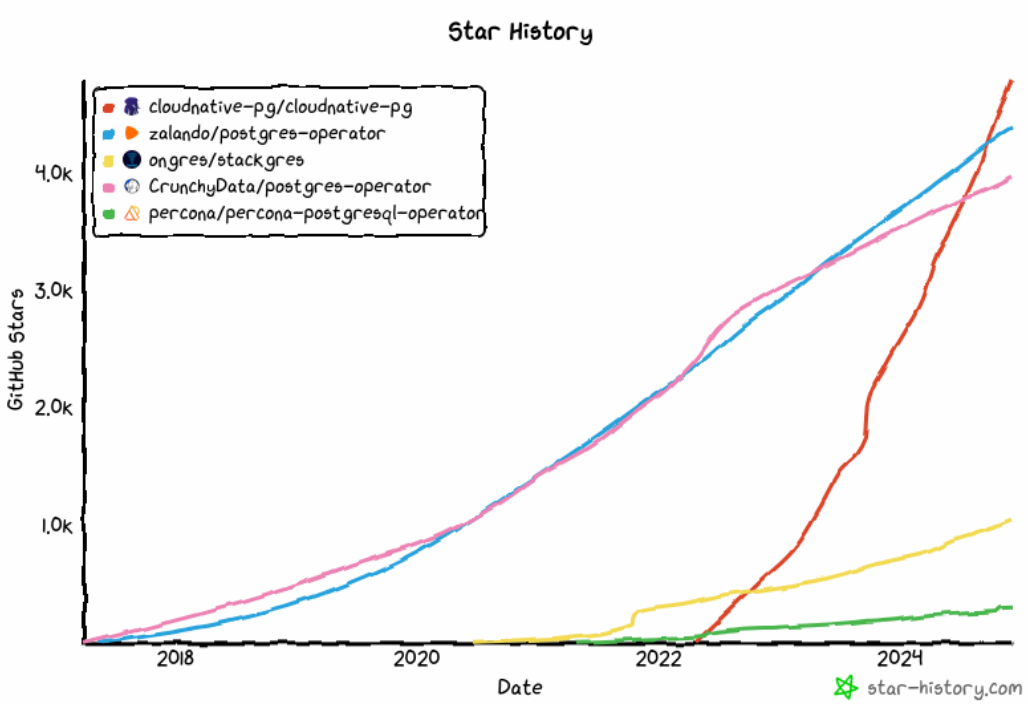
Le projet est bien engagé dans une démarche communautaire et open source avec une intention d'être incubé au sein du projet CNCF.

---

## 3.2 L'ADOPTION (1)

- De plus en plus cité, des présentations (KubeCon, PGConfEU)
- Tentatives d'incubation CNCF (en cours)
- Grand attrait sur Github

## 3.3 L'ADOPTION (2)




Les étoiles ne sont pas un argument "d'autorité"

## 3.4 CE QUE PERMET CLOUDNATIVEPG

- Déploiement d'instance PostgreSQL facilité
- Mise en place de réplication automatique
- Sauvegardes PITR, planifiées
- Bascule (automatique ou manuelle)
- Haute disponibilité
- *Hibernation*, *Fencing*
- Plugin `kubectl`

## 3.5 ATTENTION, TOUT N'EST PAS SIMPLE (1)

- Connaissances Kubernetes
  - Même en tant que DBA
- Connaissances PostgreSQL
  - Même en tant qu'admin K8S
- Changements d'habitudes et d'outils
  - Plus de *SSH* sur la machine où se trouve l'instance
  - Configuration dans les ressources Kubernetes

Les échanges entre administrateurs Kubernetes et PostgreSQL sont à renforcer pour qu'ils puissent se comprendre. Laisser la gestion d'instances PostgreSQL aux administrateurs Kubernetes, sous prétexte que tout se fait via un opérateur, et donc sans impliquer un DBA, n'est pas une solution viable. PostgreSQL est très spécifique et demande une vraie connaissance de son fonctionnement. Un DBA saura comprendre ce que fait un opérateur et réagira correctement en cas de problème.

Une montée en compétences et connaissances sont donc nécessaires pour les différents administrateurs.

Des changements d'habitudes de travail auront nécessairement lieu et impliquent également un accompagnement des équipes DBAs. L'exemple le plus parlant est l'absence d'accès SSH au serveur où est déployée l'instance, ou encore le fait de devoir passer par Kubernetes pour configurer l'instance. Certains opérateurs imposent des outils (notamment pour la partie sauvegarde) qui doivent être connus des DBAs.

## 3.6 ATTENTION, TOUT N'EST PAS SIMPLE (2)

- Couche(s) d'abstraction(s) supplémentaire(s)
  - Debug plus long / compliqué
  - Avoir les bons outils
- Jongler avec les versions
  - Kubernetes : 3 supportées
  - CloudNativePG : 2 supportées
  - PostgreSQL : 5 supportées

Le déploiement de PostgreSQL dans Kubernetes a des conséquences à bien avoir en tête. Celles-ci ne sont pas insurmontables. Il faut juste en avoir conscience.

La première à citer est la couche d'abstraction supplémentaire apportée par Kubernetes et l'opérateur. L'empilement de couches rendra par exemple le débug probablement plus long sans les bons outils de monitoring.

La gestion des versions se voit complexifiée avec un "jonglage" à faire pour avoir un bon alignement des versions supportées de PostgreSQL, de l'opérateur et de Kubernetes. La fréquence des différentes *releases* est assez élevée.

---

# 4/ Travaux pratiques

Lien du TP :

https://dali.bo/ws_cnpg_2025

---

## 4.1 PRIS EN MAIN DU CLUSTER KUBERNETES (MINIKUBE)

**Le cluster minikube est déjà déployé sur la machine virtuelle que vous allez utiliser. Les étapes nécessaires à sa mise en place se trouvent dans la partie optionnelle à la fin du TP.**

Ouvrir un terminal et se connecter en SSH à l'environnement qui vous a été attribué.

```
ssh dalibo@<IP> -p <PORTSSH>
```

Trouver la version de l'utilitaire `kubectl`.

```
kubectl version
```

```
Client Version: v1.32.0
Kustomize Version: v5.5.0
Server Version: v1.31.0
```

L'utilitaire `kubectl` vous permet d'interagir avec le cluster Kubernetes déployé.

Lister les nœuds du cluster Kubernetes.

```
kubectl get nodes
```

```
NAME           STATUS   ROLES           AGE   VERSION
k8s-demo       Ready    control-plane   57m   v1.31.0
k8s-demo-m02   Ready    <none>          56m   v1.31.0
k8s-demo-m03   Ready    <none>          55m   v1.31.0
```

Cet utilitaire sait avec quel cluster Kubernetes interagir grâce au fichier `~/.kube/config` (qui se trouve dans le répertoire de l'utilisateur `dalibo`).

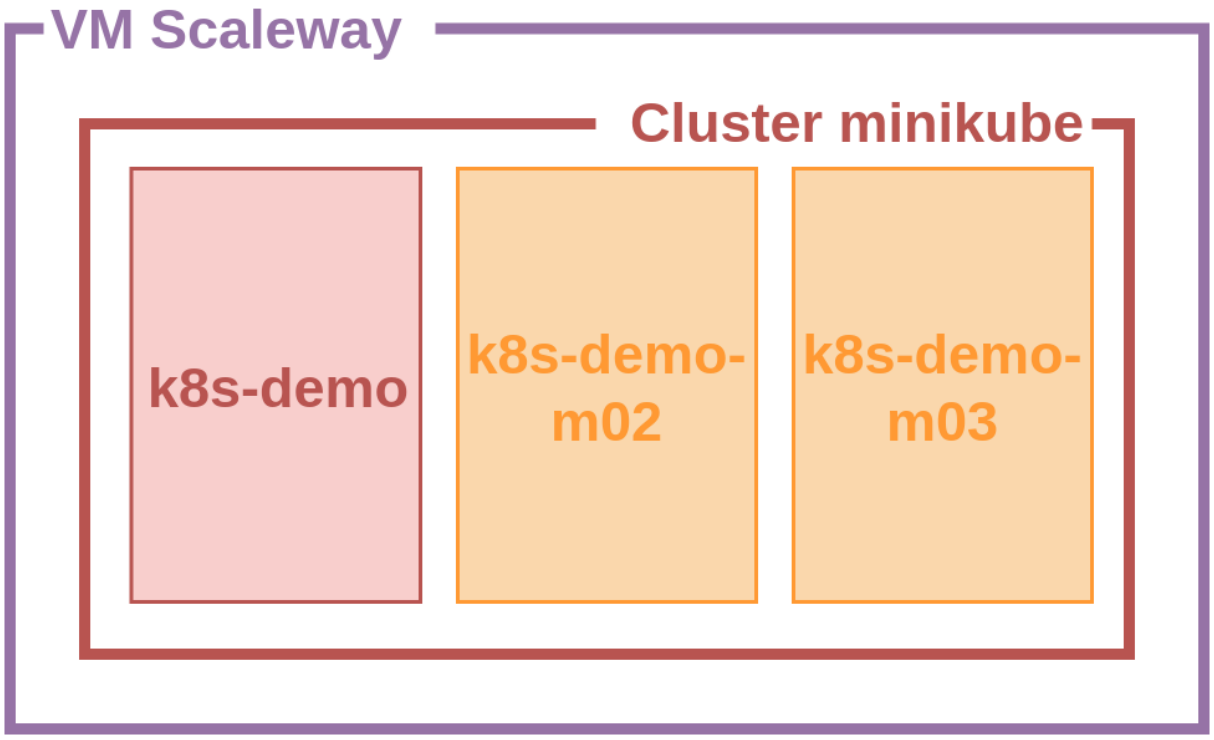
Afficher le contenu du fichier `~/.kube/config`.

```
cat ~/.kube/config
```

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /home/dalibo/.minikube/ca.crt
    extensions:
    - extension:
        last-update: Fri, 29 Nov 2024 08:31:07 UTC
        provider: minikube.sigs.k8s.io
        version: v1.34.0
```

```
      name: cluster_info
    server: https://192.168.49.2:8443
  name: k8s-demo
contexts:
- context:
    cluster: k8s-demo
    extensions:
    - extension:
        last-update: Fri, 29 Nov 2024 08:31:07 UTC
        provider: minikube.sigs.k8s.io
        version: v1.34.0
      name: context_info
    namespace: default
    user: k8s-demo
  name: k8s-demo
current-context: k8s-demo
kind: Config
preferences: {}
users:
- name: k8s-demo
  user:
    client-certificate: /home/dalibo/.minikube/profiles/k8s-demo/client.crt
    client-key: /home/dalibo/.minikube/profiles/k8s-demo/client.key
```

Schématiquement, nous nous trouvons dans cette situation :



## 4.2 INSTALLATION DE L'OPÉRATEUR CLOUDNATIVEPG

**But** : Installer l'opérateur dans le cluster `k8s-demo`.

Il existe plusieurs méthodes pour installer l'opérateur : soit en appliquant directement les fichiers YAML soit en utilisant le `Helm Chart` fourni par le projet. Pour cet atelier, nous utiliserons la première méthode, plus simple et rapide.

Installer l'opérateur avec la commande `kubectl apply -f` :

```
kubectl apply --server-side -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/release-
 ↪  1.24/releases/cnpg-1.24.0.yaml

namespace/cnpg-system serverside-applied
customresourcedefinition.apiextensions.k8s.io/backups.postgresql.cnpg.io
 ↪  serverside-applied
customresourcedefinition.apiextensions.k8s.io/clusterimagecatalogs.postgresql.cnpg.io
 ↪  serverside-applied
customresourcedefinition.apiextensions.k8s.io/clusters.postgresql.cnpg.io
 ↪  serverside-applied
customresourcedefinition.apiextensions.k8s.io/imagecatalogs.postgresql.cnpg.io
 ↪  serverside-applied
customresourcedefinition.apiextensions.k8s.io/poolers.postgresql.cnpg.io
 ↪  serverside-applied
customresourcedefinition.apiextensions.k8s.io/scheduledbackups.postgresql.cnpg.io
 ↪  serverside-applied
serviceaccount/cnpg-manager serverside-applied
clusterrole.rbac.authorization.k8s.io/cnpg-manager serverside-applied
clusterrolebinding.rbac.authorization.k8s.io/cnpg-manager-rolebinding
 ↪  serverside-applied
configmap/cnpg-default-monitoring serverside-applied
service/cnpg-webhook-service serverside-applied
deployment.apps/cnpg-controller-manager serverside-applied
mutatingwebhookconfiguration.admissionregistration.k8s.io/cnpg-mutating-webhook-
 ↪  configuration serverside-applied
validatingwebhookconfiguration.admissionregistration.k8s.io/cnpg-validating-webhook-
 ↪  configuration serverside-applied
```

Les fichiers seront récupérés depuis internet et appliqués sur votre cluster `k8s-demo`. Pour rappel, l'outil `kubectl` sait avec quel cluster Kubernetes interagir grâce au fichier *kubeconfig*.

Ces fichiers là contiennent la définition de différents ressources :

- Un `Namespace`;

- Une `CustomResourceDefinition` pour les différentes ressources que l'opérateur va gérer (`Backup`, `Cluster`, …)
- Mais aussi un `ServiceAccount`, un`ClusterRoleBinding` et surtout un déploiement du `controller` CloudNativePG.

Par défaut, le *Controller*, cerveau de l'opérateur, sera déployé dans le *Namespace* `cnpg-system`, créé lors de l'installation du l'opérateur. Ce controller n'est ni plus ni moins qu'une application. On peut voir le `controller` avec la commande `kubectl get pods` et en indiquant le bon `Namespace` :

> Lister les Pods présents dans le `namespace cnpg-system` :

```
kubectl get pods -n cnpg-system
```

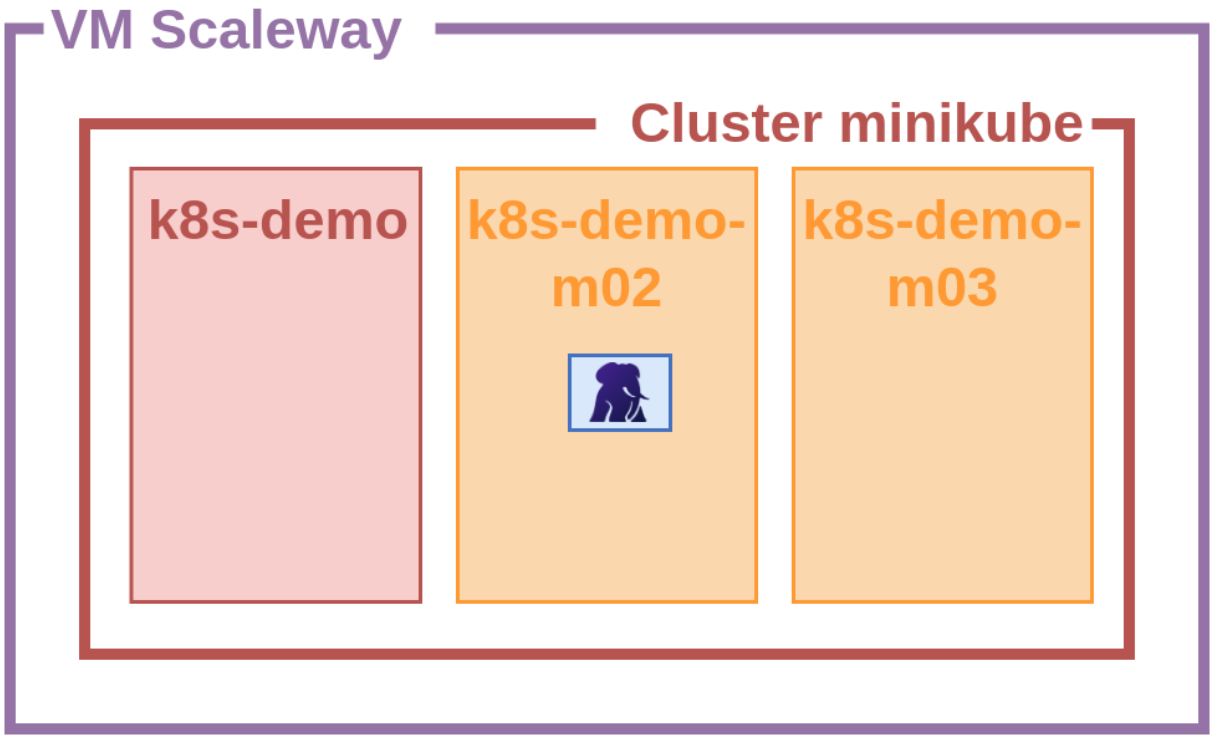
```
NAME                                       READY   STATUS    RESTARTS   AGE
cnpg-controller-manager-7fc549dc69-xq7gq   1/1     Running   0          11s
```

> Retrouver la liste des nouvelles ressources créées.

```
kubectl api-resources --api-group postgresql.cnpg.io
```

```
backups                                         postgresql.cnpg.io/v1             true
 ↪  Backup
clusterimagecatalogs                              postgresql.cnpg.io/v1
 ↪  false        ClusterImageCatalog
clusters                                        postgresql.cnpg.io/v1             true
 ↪  Cluster
imagecatalogs                                   postgresql.cnpg.io/v1             true
 ↪  ImageCatalog
poolers                                         postgresql.cnpg.io/v1             true
 ↪  Pooler
scheduledbackups                                  postgresql.cnpg.io/v1
 ↪  true         ScheduledBackup
```

Schématiquement, nous nous trouvons dans cette situation :




Notez que le `controller` CloudNativePg ici représenté, peut être démarré indifféremment sur l'un ou l'autre des nœuds. Voir la sortie de la commande `kubectl get pods -n cnpg-system -o wide`.

---

## 4.3 DÉPLOIEMENT D'INSTANCES POSTGRESQL

**But** : Déployer un cluster PostgreSQL mono-instance, s'y connecter et suivre les traces de l'opérateur et de l'instance.

Voici un exemple de fichier YAML très simple qui permet de déployer une instance PostgreSQL en version 17.0 avec 5 Go de volume associé.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-demo
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.0
  instances: 1
  storage:
    size: 5Gi
  resources:
    requests:
      memory: "256Mi"
      cpu: "0.5"
    limits:
      memory: "512Mi"
      cpu: "1"
```

Quelques informations supplémentaires sur le contenu de ce fichier :

- `apiVersion` : La version de l'API de Kubernetes est utilisée ;
- `kind` : Le type d'objet créé ;
- `metadata` : Des informations pour identifier l'objet ;
- `spec` : La définition de l'objet en question ("l'état désiré") ;
- `imageName` : Le nom de l'image utilisée ;
- `instances` : Le nombre d'instances voulues (sera toujours 1 primaire + le reste en secondaire(s) ;)
- `storage` : Les informations sur le stockage souhaité ;
- `resources` : L'indication de *requests* et *limits* sur la RAM et CPU.

Créer le fichier `postgresql-demo.yaml` dans le home directory de `dalibo` et copier le contenu YAML ci-dessus :

```
$ cat <<'EOF' > ~/postgresql-demo.yaml
apiVersion: postgresql.cnpg.io/v1
```

```
kind: Cluster
metadata:
  name: postgresql-demo
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.0
  instances: 1
  storage:
    size: 5Gi
  resources:
    requests:
      memory: "256Mi"
      cpu: "0.5"
    limits:
      memory: "512Mi"
      cpu: "1"
EOF
```

> Dans un autre terminal sur la VM, suivre les traces du `controller` avec la commande `kubectl logs -f -n cnpg-system <POD>` et l'utilitaire `jq`. Pour retrouver le nom du Pod du controlleur, vous pouvez utiliser `kubectl get pod -A`.

```
kubectl logs -f -n cnpg-system cnpg-controller-manager-7fc549dc69-8v8xw | jq
```

> Retourner dans l'ancienne session SSH et créer l'instance PostgreSQL à partir du fichier `~/postgresql-demo.yaml` avec `kubectl`. En parallèle regarder ce qu'il se passe dans les traces du `controller` :

```
kubectl apply -f ~/postgresql-demo.yaml
```

```
cluster.postgresql.cnpg.io/postgresql-demo created
```

Une instance PostgreSQL est désormais en train d'être déployée par l'opérateur. Vous avez décrit ce que vous souhaitiez avoir, l'opérateur fait le reste. Plusieurs choses se passent lorsque vous appliquez ce fichier avec `kubectl`. Tout d'abord l'opérateur va déployer un premier Pod appelé `<clusterName>-1-initdb-<random>`. `initdb` devrait vous faire penser à la commande à exécuter lorsque vous devez créer une instance manuellement par exemple.

```
kubectl get pods --watch
```

```
NAME                              READY   STATUS    RESTARTS   AGE   IP       NODE
 ↪  NOMINATED NODE   READINESS GATES
postgresql-demo-1-initdb-5pndc    0/1     Pending   0          2s    <none>   <none>
 ↪  <none>           <none>
```

Ce Pod là se repose sur une image qui doit être téléchargée. C'est pour cela que vous devez avoir autorisé l'accès vers internet (ou votre dépôt local d'images) à votre cluster. Lorsque celle-ci est

récupérée, le Pod est “amorcé” et les conteneurs d’initialisation sont déployés, comme on peut le voir avec cette seconde remontée. Ici il existe un conteneur d’initialisation mais aucun n’est terminé.

```
NAME                              READY   STATUS     RESTARTS   AGE   IP       NODE
 ↪  NOMINATED NODE   READINESS GATES
postgresql-demo-1-initdb-5pndc    0/1     Init:0/1   0          13s   <none>
 ↪  k8s-demo   <none>           <none>
```

Au fur et à mesure, le Pod passe par d’autres états …

```
NAME                              READY   STATUS            RESTARTS   AGE   IP
 ↪  NODE       NOMINATED NODE   READINESS GATES
postgresql-demo-1-initdb-5pndc    0/1     PodInitializing   0          24s
 ↪  10.244.228.68   k8s-demo   <none>           <none>
```

… jusqu’à l’état `Running`. À cette étape-ci, le Pod va notamment initialiser l’instance avec la création de l’arborescence du PGDATA.

```
NAME                              READY   STATUS    RESTARTS   AGE   IP
 ↪  NODE       NOMINATED NODE   READINESS GATES
postgresql-demo-1-initdb-5pndc    1/1     Running   0          35s   10.244.228.68
 ↪  k8s-demo   <none>           <none>
```

Enfin, lorsque cette étape est terminée, l’opérateur CloudNativePG déploie un autre Pod qui cette fois-ci ne porte plus le mot `initdb`. Un numéro est ajouté à la fin du nom. Une adresse IP est attribuée à ce Pod (IP privée RFC 1918[1]) :

```
kubectl get pods -o wide
```

```
NAME                READY   STATUS    RESTARTS   AGE   IP              NODE
 ↪  NOMINATED NODE   READINESS GATES
postgresql-demo-1   1/1     Running   0          10m   10.244.228.69   k8s-demo
 ↪  <none>           <none>
```

Votre Pod est prêt et donc votre instance aussi !

> Se connecter à l’instance et vérifier la version de celle-ci. Vous pouvez utiliser `kubectl exec […]` comme ceci :

```
kubectl exec -it postgresql-demo-1 -- psql
```

ou, via le plugin :

```
kubectl cnpg psql postgresql-demo
```

[1] https://datatracker.ietf.org/doc/html/rfc1918

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
psql (17.0 (Debian 17.0-1.pgdg110+1))
Type "help" for help.

postgres=# select version()\gx
-[ RECORD 1 ]
----------------
version | PostgreSQL 17.0 (Debian 17.0-1.pgdg110+1) on aarch64-unknown-linux-gnu,
 ↪  compiled by gcc (Debian 10.2.1-6) 10.2.1 20210110, 64-bit

postgres=# exit
```

Pour quitter `psql`, vous pouvez utiliser `control+d`, `\q` ou `exit`.

La commande `kubectl exec -it` permet d'exécuter un programme au sein du Pod. L'outil `psql` étant présent dans l'image, cela est possible. Essayez avec `vim`, qui lui n'est pas présent dans l'image, un message d'erreur apparaîtra.

> Suivre les traces de l'instance avec la commande `kubectl logs -f postgresql-demo-1`.

```
kubectl logs -f postgresql-demo-1
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
{"level":"info","ts":"2024-11-18T14:33:11.503705829Z","msg":"Starting CloudNativePG
 ↪  Instance Manager","logger":"instance-manager","logging_pod":"postgresql-demo-
 ↪  1","version":"1.24.0","build":{"Version":"1.24.0","Commit":"3f96930d","Date":"2024-
 ↪  10-16"}}
{"level":"info","ts":"2024-11-18T14:33:11.504063871Z","msg":"Checking for free disk
 ↪  space for WALs before starting
 ↪  PostgreSQL","logger":"instance-manager","logging_pod":"postgresql-demo-1"}
{"level":"info","ts":"2024-11-18T14:33:11.639359824Z","msg":"starting tablespace
 ↪  manager","logger":"instance-manager","logging_pod":"postgresql-demo-1"}
```

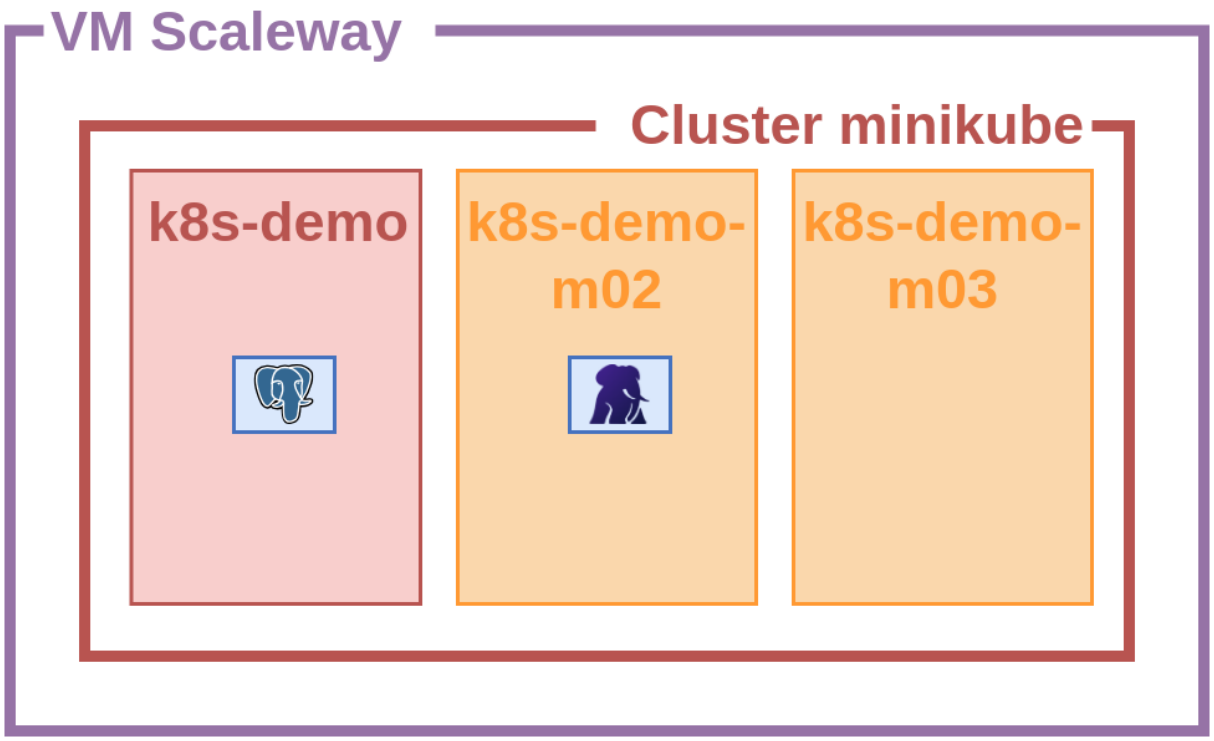
Les logs de l'instance sont récupérés au format JSON, et sont en l'état peu exploitable. Pour les lire plus facilement, vous pouvez utiliser l'outil `jq`.

```
kubectl logs -f postgresql-demo-1 | jq
```

```
{
  "level": "info",
  "ts": "2024-11-19T09:43:46.00225746Z",
  "logger": "postgres",
  "msg": "record",
  "logging_pod": "postgresql-demo-1",
  "record": {
    "log_time": "2024-11-19 09:43:46.002 UTC",
    "process_id": "20",
    "session_id": "673c5dd1.14",
```

```
    "session_line_num": "6",
    "session_start_time": "2024-11-19 09:43:45 UTC",
    "transaction_id": "0",
    "error_severity": "LOG",
    "sql_state_code": "00000",
    "message": "database system is ready to accept connections",
    "backend_type": "postmaster",
    "query_id": "0"
  }
}
```


Schématiquement, nous nous trouvons dans cette situation :



## 4.4 QU'EST-CE QUI EST CRÉÉ ?

**But** : Découvrir ce qui est automatiquement créé.

Avec quelques lignes de yaml et commandes, une instance PostgreSQL est déployée et accessible. De nombreuses choses sont créées automatiquement pour nous. Voyons de quoi il s'agit.

### 4.4.1 Bases de données

Retrouver la liste des bases de données dans l'instance déployée.

La meta-commande `\l` de psql vous permet de récupérer la liste des bases.

```
kubectl exec -it postgresql-demo-1 -- psql -l
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
                                                List of databases
   Name    |  Owner   | Encoding | Locale Provider | Collate | Ctype | Locale | ICU
    ↪  Rules |   Access privileges
-----------+----------+----------+-----------------+---------+-------+--------+-----
 ↪  ------+-----------------------
 app       | app      | UTF8     | libc            | C       | C     |        |
 postgres  | postgres | UTF8     | libc            | C       | C     |        |
 template0 | postgres | UTF8     | libc            | C       | C     |        |
  ↪  | =c/postgres          +
           |          |          |                 |         |       |        |
            ↪  | postgres=CTc/postgres
 template1 | postgres | UTF8     | libc            | C       | C     |        |
  ↪  | =c/postgres          +
           |          |          |                 |         |       |        |
            ↪  | postgres=CTc/postgres
(4 rows)
```

Par défaut, une base de données app est créée dans l'instance PostgreSQL.

### 4.4.2 Rôles et `Secret`

Retrouver la liste des rôles dans l'instance déployée.

La meta-commande `\du` de psql vous permet de récupérer la liste des rôles.

```
kubectl cnpg psql postgresql-demo -- -c '\du'
```

```
                                 List of roles
     Role name     |                         Attributes
-------------------+------------------------------------------------------------
 app               |
 postgres          | Superuser, Create role, Create DB, Replication, Bypass RLS
 streaming_replica | Replication
```

Par défaut deux rôles sont créés : `app` et `streaming_replica`. Dans la liste des bases de données, on peut d'ailleurs voir que le rôle `app` est propriétaire de la base `app`.

> Se connecter à la base de données `app` avec le rôle `app`.

```
kubectl exec -it postgresql-demo-1 -- psql -U app
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
psql: error: connection to server on socket "/controller/run/.s.PGSQL.5432" failed:
 ↪  FATAL:  Peer authentication failed for user "app"
command terminated with exit code 2
```

L'authentification du rôle `app` avec la méthode `peer` ne peut pas se faire. Mais alors comment se connecter avec `app` ? En sachant que `listen_addresses` est positionné à `*` par défaut, une solution pour tester la connexion est de passer par la pile TCP/IP classique en utilisant l'option `-h` de `psql`.

```
kubectl exec -it postgresql-demo-1 -- psql -U app -h 127.0.0.1
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
Password for user app:
```

Il faut comprendre que le `127.0.0.1` fait référence à l'adresse localhost du Pod. On demande à `psql`, via `kubectl`, de se connecter sur l'interface `localhost` du Pod… mais il nous faut le mot de passe de `app`… où le trouver ?

CloudNativePG crée automatiquement un `Secret` qui contient des informations de connexion, notamment le mot de passe de `app`.

> Récupérer la liste des `Secrets` du cluster.

```
kubectl get secrets
```

```
NAME                          TYPE                       DATA   AGE
postgresql-demo-app           kubernetes.io/basic-auth   9      39m
postgresql-demo-ca            Opaque                     2      39m
postgresql-demo-replication   kubernetes.io/tls          2      39m
postgresql-demo-server        kubernetes.io/tls          2      39m
```

> Récupérer le mot de passe présent dans le `Secret` postgresql-demo-app.

```
kubectl get secret postgresql-demo-app -o json | jq '.data.password'
```

```
"VFdyejRQbmY1RWMwVjFjUHlqYkdFZnI5RG52WE5YaXN0NUhIaFZkOENwSkpKOEthVkVLUkNxUGwweTRzaGlVbw=="
```

Ou bien sans `jq`, avec une commande `kubectl` un peu plus poussée :

```
$ kubectl get secret postgresql-demo-app --no-headers -o
 ↪  custom-columns=Passwd:.data.password
VFdyejRQbmY1RWMwVjFjUHlqYkdFZnI5RG52WE5YaXN0NUhIaFZkOENwSkpKOEthVkVLUkNxUGwweTRzaGlVbw==
```

Le résultat est encodé en base64. Il faut donc le décoder avec l'une des commandes suivantes :

```
$ echo "VFdye-
 ↪  jRQbmY1RWMwVjFjUHlqYkdFZnI5RG52WE5YaXN0NUhIaFZkOENwSkpKOEthVkVLUkNxUGwweTRzaGlVbw=="
 ↪  | base64 -d
TWrz4Pnf5Ec0V1cPyjbGEfr9DnvXNXist5HHhVd8CpJJJ8KaVEKRCqPl0y4shiUo

$ kubectl get secret postgresql-demo-app --no-headers -o
 ↪  custom-columns=Passwd:.data.password | base64 -d
TWrz4Pnf5Ec0V1cPyjbGEfr9DnvXNXist5HHhVd8CpJJJ8KaVEKRCqPl0y4shiUo
```

> Se connecter à l'instance avec l'utilisateur app.

```
kubectl exec -it postgresql-demo-1 -- psql -U app -h 127.0.0.1
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
Password for user app:
psql (17.0 (Debian 17.0-1.pgdg110+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off,
 ↪  ALPN: postgresql)
Type "help" for help.

app=>
```

L'astuce d'utiliser `kubectl` et `psql` avec l'option `-h` permet à des administrateurs de se connecter, mais cela n'est pas envisageable pour des applications. Les applications doivent passer par les objets `Services`.

### 4.4.3 Services

Un `Service` est une couche d'abstraction qui permet d'accéder à un ensemble de `Pods` spécifiques. L'association `Service` - `Pods` se fait via des *labels*. Un *label* est une étiquette, un *tag*, apposée à une ressource.

> Retrouver la liste des `Services` dans le cluster Kubernetes.

Comme toutes les autres ressources Kubernetes, vous pouvez récupérer les objets `Services` avec `get`.

```
kubectl get svc
```

```
NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
kubernetes           ClusterIP   10.96.0.1        <none>        443/TCP    6h24m
postgresql-demo-r    ClusterIP   10.105.219.134   <none>        5432/TCP   6h11m
postgresql-demo-ro   ClusterIP   10.105.155.153   <none>        5432/TCP   6h11m
postgresql-demo-rw   ClusterIP   10.105.191.44    <none>        5432/TCP   6h11m
```

À chaque cluster PostgreSQL déployé, trois services sont créés :

- Un service qui permet d'accéder au primaire : `postgresql-demo-rw` qui est en lecture/écriture;
- Un service qui permet d'accéder uniquement au(x) secondaire(s) : `postgresql-demo-ro` qui sont en lecture seule;
- Un service qui permet d'accéder à toutes les instances : `postgresql-demo-r`.

> Retrouver les *labels* définis sur le Pod de votre instance :

```
kubectl get pod postgresql-demo-1 --show-labels
```

```
NAME                READY   STATUS    RESTARTS   AGE   LABELS
postgresql-demo-1   1/1     Running   0          26h
 ↪  cnpg.io/cluster=postgresql-demo,cnpg.io/instanceName=postgresql-demo-
 ↪  1,cnpg.io/instanceRole=primary,cnpg.io/podRole=instance,role=primary
```

> Retrouver la description du `Service` `postgresql-demo-ro` et retrouver la partie `Selector` qui indique avec quel(s) Pod(s) sera associé ce `Service`.

```
kubectl describe service postgresql-demo-ro
```

```
Name:                     postgresql-demo-ro
Namespace:                default
Labels:                   cnpg.io/cluster=postgresql-demo
Annotations:              cnpg.io/operatorVersion: 1.24.0
Selector:                 cnpg.io/cluster=postgresql-demo,cnpg.io/instanceRole=replica
Type:                     ClusterIP
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.105.155.153
IPs:                      10.105.155.153
Port:                     postgres  5432/TCP
TargetPort:               5432/TCP
Endpoints:                10.244.112.199:5432
Session Affinity:         None
Internal Traffic Policy:  Cluster
Events:                   <none>
```

Lorsqu'une bascule a lieu, les *labels* des `Pods` sont mis à jour et l'association `Service` - `Pod` est automatiquement adaptée. De ce fait, si vos applications utilisent bien le nom du `Service` dans les informations de connexion, elles seront automatiquement redirigées vers la nouvelle instance primaire par exemple.

### 4.4.4 pg_hba

Le fichier `pg_hba.conf` est un passage obligatoire pour la bonne configuration de votre instance. Le modifier directement n'est plus possible. Voyons tout de même ce qu'il contient par défaut et comment configurer l'accès à l'instance via l'opérateur.

> Retrouver le contenu du fichier `pg_hba.conf` de l'instance.

Voici quelques méthodes possibles :

- Avec `cat` depuis le `Pod` : `kubectl exec -it postgresql-demo-1 -- cat /var/lib/postgresql/data/pgdata/pg_hba.conf`;
- Avec `psql` et la table `pg_hba_file_rules` : `kubectl exec -it postgresql-demo-1 \ -- psql -c 'select type, database, user_name, address, netmask, auth_method, options from pg_hba_file_rules ORDER BY rule_number'`
- Avec la documentation[2]

Il existe trois sections dans ce fichier :

- `FIXED RULES` : qui sont des règles fixées par l'opérateur (on retrouve une règle concernant la réplication par exemple);
- `USER-DEFINED RULES` : qui correspond aux règles qui seront créées;
- `DEFAULT RULES` : qui autorise par défaut toutes les connexions par mots de passe à toutes les bases de données.

[2] https://cloudnative-pg.io/documentation/current/postgresql_conf/#the-pg_hba-section

## 4.5 OPÉRATIONS BASIQUES

**But** : Créer, configurer et manipuler notre instance.

### 4.5.1 Modifications de paramètres de configuration

Installer une instance PostgreSQL ne suffit pas. Il faut en plus la configurer. Habituellement, la configuration se fait dans le fichier `postgresql.conf` et nécessite soit un rechargement, soit un redémarrage de l'instance selon le paramètre modifié. Nous allons voir comment le faire sur notre instance `postgresql-demo-1`.

> Dupliquer le fichier `~/postgresql-demo.yaml` pour conserver une copie de la définition initiale de l'instance.

```
cp ~/postgresql-demo.yaml ~/postgresql-demo.bckp
```

> Modifier le fichier `~/postgresql-demo.yaml` et ajouter la section `postgresql.parameters` comme dans l'exemple. Nous allons tout d'abord modifier les paramètres `shared_buffers` et `max_connection`.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-demo
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.0
  instances: 1
  storage:
    size: 5Gi
  postgresql:
    parameters:
      shared_buffers: 256MB
      max_connections: '10'
  resources:
    requests:
      memory: "256Mi"
      cpu: "0.5"
    limits:
      memory: "512Mi"
      cpu: "1"
```

> Suivre les traces du Pod et de l'instance avec `kubectl logs -f postgresql-demo-1 | jq`:

> Utiliser `kubectl apply -f ~/postgresql-demo.yaml` appliquer les modifications.

Dans les traces du Pod, certains messages indiquent très clairement ce qu'il va se passer. Les champs `msg` et `message` sont les plus intéressants. Par exemple, celui-ci qui indique qu'un rechargement de la configuration est nécessaire.


```
{
  "level": "info",
  "ts": "2024-11-19T10:16:20.260220994Z",
  "msg": "Requesting configuration reload",
  "logger": "instance-manager",
  "logging_pod": "postgresql-demo-1",
  "controller": "instance-cluster",
  "controllerGroup": "postgresql.cnpg.io",
  "controllerKind": "Cluster",
  "Cluster": {
    "name": "postgresql-demo",
    "namespace": "default"
  },
  "namespace": "default",
  "name": "postgresql-demo",
  "reconcileID": "ee16f0eb-c352-41e9-a955-0587219b4d7b",
  "pgdata": "/var/lib/postgresql/data/pgdata"
}
```


Ou encore celui-ci, quelques lignes plus loin, qui indique que le paramètre modifié implique qu'un redémarrage de l'instance est nécessaire.


```
{
  "level": "info",
  "ts": "2024-11-19T10:16:20.269390325Z",
  "logger": "postgres",
  "msg": "record",
  "logging_pod": "postgresql-demo-1",
  "record": {
    "log_time": "2024-11-19 10:16:20.263 UTC",
    "process_id": "21",
    "session_id": "673c655c.15",
    "session_line_num": "9",
    "session_start_time": "2024-11-19 10:15:56 UTC",
    "transaction_id": "0",
    "error_severity": "LOG",
```

```
    "sql_state_code": "55P02",
    "message": "parameter \"shared_buffers\" cannot be changed without restarting
     ↪  the server",
    "backend_type": "postmaster",
    "query_id": "0"
  }
}
```


À la toute fin, votre instance est de nouveau accessible comme l'indique la ligne JSON :


```
{
  "level": "info",
  "ts": "2024-11-19T10:16:20.811302094Z",
  "logger": "postgres",
  "msg": "record",
  "logging_pod": "postgresql-demo-1",
  "record": {
    "log_time": "2024-11-19 10:16:20.811 UTC",
    "process_id": "204",
    "session_id": "673c6574.cc",
    "session_line_num": "6",
    "session_start_time": "2024-11-19 10:16:20 UTC",
    "transaction_id": "0",
    "error_severity": "LOG",
    "sql_state_code": "00000",
    "message": "database system is ready to accept connections",
    "backend_type": "postmaster",
    "query_id": "0"
  }
}
```


Il faut donc comprendre que dès qu'une modification est apportée à la configuration, par défaut, l'opérateur CloudNativePG va faire en sorte de la prendre immédiatement en compte. Un rechargement de la configuration sera effectué si le paramètre ne nécessite pas le redémarrage de l'instance. Si un redémarrage est effectué, alors les connexions seront coupées et devront être refaites à l'instance PostgreSQL par les applications. Vous devez donc bien savoir ce que vous devez faire. Des précautions sont donc plus que nécessaires.

> Modifier le paramètre `work_mem` et réappliquer la définition YAML avec `kubectl apply -f ~/postgresql-demo.yaml`.

```
[...]
```

```
  postgresql:
    parameters:
      shared_buffers: 256MB
      max_connections: '10'
```

```
      work_mem: '8MB'
[...]
```

Le paramètre `work_mem` ne nécessite pas un redémarrage de l'instance. Voici un exemple de trace obtenue lors de la modification de ce paramètre.


```
{
  "level": "info",
  "ts": "2024-11-29T06:02:03Z",
  "logger": "postgres",
  "msg": "record",
  "logging_pod": "postgresql-demo-1",
  "record": {
    "log_time": "2024-11-29 06:02:03.388 UTC",
    "process_id": "936",
    "session_id": "67495814.3a8",
    "session_line_num": "7",
    "session_start_time": "2024-11-29 05:58:44 UTC",
    "transaction_id": "0",
    "error_severity": "LOG",
    "sql_state_code": "00000",
    "message": "received SIGHUP, reloading configuration files",
    "backend_type": "postmaster",
    "query_id": "0"
  }
}
```


La ligne `message` indique que seul un rechargement de la configuration a été nécessaire.

> Vérifier que la modification a bien été prise en compte. Vous pouvez le voir dans les traces ou alors directement en vous connectant à l'instance et en utilisant `show work_mem` dans le prompt `psql`;

Dans les traces, regarder le champ `message`.


```
{
  "level": "info",
  "ts": "2024-11-29T06:02:03Z",
  "logger": "postgres",
  "msg": "record",
  "logging_pod": "postgresql-demo-1",
  "record": {
    "log_time": "2024-11-29 06:02:03.390 UTC",
    "process_id": "936",
    "session_id": "67495814.3a8",
    "session_line_num": "8",
    "session_start_time": "2024-11-29 05:58:44 UTC",
```

```
    "transaction_id": "0",
    "error_severity": "LOG",
    "sql_state_code": "00000",
    "message": "parameter \"work_mem\" changed to \"8MB\"",
    "backend_type": "postmaster",
    "query_id": "0"
  }
}
```

En lignes de commande :

```
kubectl exec -it postgresql-demo-1 -- psql
```

ou, via le plugin :

```
kubectl cnpg psql postgresql-demo
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
psql (17.0 (Debian 17.0-1.pgdg110+1))
Type "help" for help.

postgres=# show work_mem ;
 work_mem
----------
 8MB
(1 row)
```

Certains paramètres PostgreSQL ne sont pas modifiables. C'est le parti pris des développeurs de CloudNativePG. La liste se trouve dans la documentation du projet (ici[3]).

### 4.5.2 Créer un rôle

Il existe plusieurs méthodes pour créer un rôle dans une instance. L'ordre SQL `CREATE ROLE ...` peut évidemment être utilisé, mais pour cet exemple, nous allons passer par la méthode déclarative et demander à l'opérateur de faire en sorte que le rôle soit présent dans l'instance.

> Créer un rôle dba ayant les droits SUPERUSER dans l'instance.

L'ajout d'un rôle se fait avec la section `managed` du fichier yaml. Par exemple dans notre fichier `~/postgresql-demo.yaml`, cela donnerait :

```
[...]
spec:
[...]
  managed:
```

[3] https://cloudnative-pg.io/documentation/current/postgresql_conf/#fixed-parameters

```
    roles:
    - name: dba
      ensure: present
      comment: Administrateur
      login: true
      superuser: true
```

> Appliquer la modification avec `kubectl apply -f ~/postgresql-demo.yaml`.

> Vérifier que le rôle a bien été créé.

```
kubectl exec -it postgresql-demo-1 -- psql -c '\du'
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
                                   List of roles
     Role name     |                         Attributes
-------------------+------------------------------------------------------------
 app               |
 dba               | Superuser
 postgres          | Superuser, Create role, Create DB, Replication, Bypass RLS
 streaming_replica | Replication
```

Le rôle est bien créé mais il n'a actuellement pas de mot de passe configuré.

Si vous souhaitez en ajouter un, vous pouvez le faire de plusieurs manières :

- en exécutant la requête `ALTER ROLE ... SET PASSWORD ...` ;
- en utilisant `\password <user>` (la préférer à `ALTER ROLE...`);
- en demandant à CloudNativePG de le faire. Cela nécessite la création d'un objet `Secret`.

C'est cette dernière méthode que nous allons suivre. Pour cela, le mot de passe n'est jamais passé en clair dans le fichier yaml. Il est en fait nécessaire de créer un objet `Secret` qui contiendra ce mot de passe encodé en base64 ainsi que le nom du rôle. C'est ce `Secret` là qui sera utilisé dans le fichier yaml.

> Encoder le nom du rôle en base64.

```
printf "dba" | base64
ZGJh
```

> Encoder le mot de passe en base64.

Vous pouvez ajouter un espace avant `echo` pour que la commande n'apparaisse pas dans l'historique de l'utilisateur `dalibo`.

```
printf "ilovemydba" | base64
aWxvdmVteWRiYQ==
```

> Créer un fichier `~/secret.yaml` avec le contenu suivant puis créer le `Secret`.

```
apiVersion: v1
data:
  username: ZGJh
  password: aWxvdmVteWRiYQ==
kind: Secret
metadata:
  name: secret-password-dba
  labels:
    cnpg.io/reload: "true"
type: kubernetes.io/basic-auth
```

```
kubectl apply -f ~/secret.yaml
```

```
secret/secret-password-dba created
```

> Ajouter ce mot de passe à la définition du rôle dba dans le fichier `~/postgresql-demo.yaml`, via l'information `passwordSecret`.

```
[...]
  managed:
    roles:
    - name: dba
      ensure: present
      comment: Administrateur
      login: true
      superuser: true
      passwordSecret:
        name: secret-password-dba
```

> Appliquer les modifications.

```
kubectl apply -f ~/postgresql-demo.yaml
```

```
cluster.postgresql.cnpg.io/postgresql-demo configured
```

Le rôle dba peut désormais se connecter avec son super mot de passe. Par exemple :

```
kubectl exec -it postgresql-demo-1 -- psql -d postgres -U dba -h 127.0.0.1
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
Password for user dba:
psql (17.0 (Debian 17.0-1.pgdg110+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off,
 ↪  ALPN: postgresql)
Type "help" for help.

postgres=#
```

### 4.5.3 Créer une base de données

**Au moment de l'écriture du *workshop* la version 1.25 n'était pas encore disponible …**

**Cette version permet désormais, via la nouvelle CRD `Database`, de manipuler et gérer des bases de données de manière déclarative avec l'opérateur.**

```
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: db1
spec:
  name: db1
  owner: app
  cluster:
    name: cluster-dalidemo
```

**Pour les besoins du TP, nous resteront en version 1.24.0.**

Jusqu'en 1.24.0, il n'était pas possible de créer une base de données dans une instance de manière déclarative via le fichier yaml.

Il est toujours possible de créer des bases de données supplémentaires plus tard en utilisant l'ordre SQL `CREATE DATABASE`.

CNPG offre tout de même deux possibilités supplémentaires intéressantes. Voici quelques explications à titre indicatif (pas besoin de les faire dans le cadre du *workshop*) :

1. Vous pouvez modifier la base de données créée automatiquement (par défaut `app`). Pour cela, il faut modifier sa définition dans la section `bootstrap` avant de créer le cluster. Par exemple :

   ```
   [...]
   bootstrap:
     initdb:
       database: app
       owner: app
       secret:
         name: app-secret
   [...]
   ```

2. Si vous souhaitez conserver la base `app` et en créer une supplémentaire, il existe le paramètre `postInitSQL` dans la section `initdb` qui permet d'exécuter des ordres SQL dès la fin de création de l'instance. Vous pouvez alors indiquer un ordre SQL `CREATE DATABASE ...`. Les requêtes indiquées à ce niveau seront exécutées **uniquement** et **une seule fois** après la création de l'instance. Autrement dit, si la modification est apportée a une instance déjà créée, rien ne se passera.

```
[...]
spec:
[...]
  bootstrap:
   initdb:
     postInitSQL:
       - create database my_database
```

## 4.6 DÉPLOIEMENT D'UNE INSTANCE SECONDAIRE

**But** : Déployer une instance secondaire dans le cluster `postgresql-demo`.

Notre instance actuellement déployée ne possède pas de secondaire. L'ajout de secondaire se fait facilement via la modification du paramètre `instances` dans la section `spec` de notre fichier yaml.

> Déployer un secondaire à votre instance en modifiant le paramètre `instances` à 2.

```
[...]
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.0
  instances: 2
[...]

kubectl apply -f ~/postgresql-demo.yaml
```

Un second Pod va être déployé.

```
kubectl get pod | grep demo
postgresql-demo-1              1/1     Running    0          73m
postgresql-demo-2-join-h2vvw   0/1     Init:0/1   0          38s

kubectl get pod | grep demo

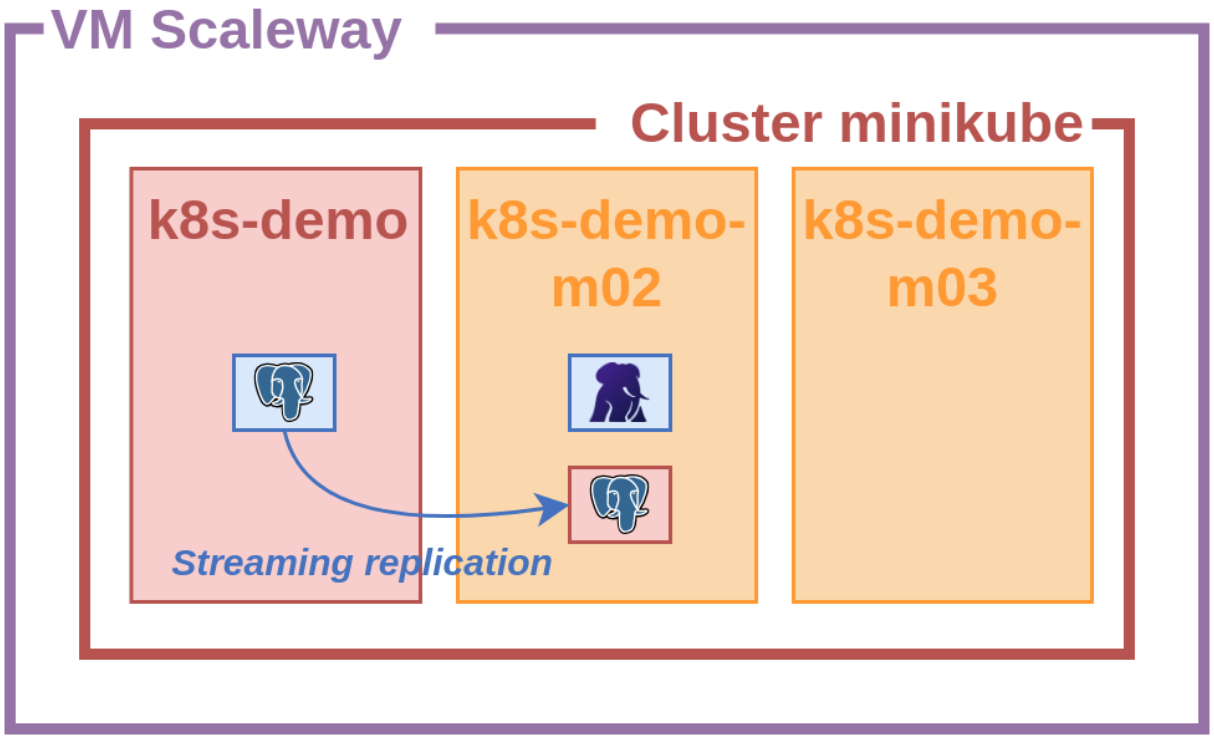
postgresql-demo-1              1/1     Running           0          73m
postgresql-demo-2-join-h2vvw   0/1     PodInitializing   0          58s

kubectl get pod | grep demo

``` sh
postgresql-demo-1              1/1     Running     0          73m
postgresql-demo-2              1/1     Running     0          31s
```

Et voilà, un secondaire a été créé ! L'opérateur CloudNativePG s'assure de tout configurer : ajout du paramètre `primary_conninfo`, création du fichier `standby.signal`, mise à jour de `pg_hba.conf` etc. Le secondaire se connecte alors au primaire en utilisant la réplication physique native de PostgreSQL (*Streaming Replication*).

Schématiquement, nous nous trouvons dans cette situation :
```




### 4.6.1 Configuration par défaut

Regardons la configuration qui est mise en place par défaut.

> Se connecter avec `psql` au secondaire nouvellement créé.

```
kubectl exec -it postgresql-demo-2 -- psql
```

> Récupérer le contenu du paramètre `primary_conninfo`.

```
postgres=# \x
Expanded display is on.
postgres=# show primary_conninfo ;

-[ RECORD 1 ]----+---------------------
primary_conninfo | host=postgresql-demo-rw user=streaming_replica [...]
```

*La sortie a été mise en forme pour plus de lisibilité.*

```
host=postgresql-demo-rw
user=streaming_replica
port=5432
sslkey=/controller/certificates/streaming_replica.key
sslcert=/controller/certificates/streaming_replica.crt
```

```
sslrootcert=/controller/certificates/server-ca.crt
application_name=postgresql-demo-2
sslmode=verify-ca
```

Le secondaire utilise le `Service` `postgresql-demo-rw` pour accéder à l'instance primaire. Vous comprendrez qu'une résolution DNS doit se faire pour retrouver l'adresse IP associée. La réplication utilise l'utilisateur dédié `streaming_replica` créé par CloudNativePG lors du déploiement de la première instance. L'authentification se fait par certificat. Le paramètre `application_name` permet d'indiquer un nom d'application dans les informations liées la connexion.

> Se connecter avec `psql` au primaire.

```
kubectl exec -it postgresql-demo-1 -- psql
```

ou, via le plugin :

```
kubectl cnpg psql postgresql-demo
```

> Récupérer le contenu de la table `pg_stat_replication`.

```
postgres=# select * from pg_stat_replication\gx
-[ RECORD 1 ]----+------------------------------
pid              | 2370
usesysid         | 16388
usename          | streaming_replica
application_name | postgresql-demo-2
client_addr      | 10.244.41.198
client_hostname  |
client_port      | 35530
backend_start    | 2024-11-29 07:47:00.777328+00
backend_xmin     |
state            | streaming
sent_lsn         | 0/6000060
write_lsn        | 0/6000060
flush_lsn        | 0/6000060
replay_lsn       | 0/6000060
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
reply_time       | 2024-11-29 08:51:20.1176+00
```

Par défaut, c'est une réplication asynchrone qui est créée.

> Récupérer le contenu de la table `pg_replication_slots`.

```
postgres=# select * from pg_replication_slots \gx

-[ RECORD 1 ]-------+------------------------
slot_name           | _cnpg_postgresql_demo_2
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | t
active_pid          | 2370
xmin                |
catalog_xmin        |
restart_lsn         | 0/6000060
confirmed_flush_lsn |
wal_status          | reserved
safe_wal_size       |
two_phase           | f
inactive_since      |
conflicting         |
invalidation_reason |
failover            | f
synced              | f
```

Par défaut, CloudNativePG crée automatiquement un slot de réplication pour sécuriser la réplication. Son nom permet de savoir facilement à quoi elle correspond. Le slot de réplication garantit au secondaire que son primaire ne recyclera pas les journaux dont il aura encore besoin. Le secondaire peut donc prendre un retard conséquent sans risque de décrochage. Attention à l'accumulation des *WALs* qu'il peut y avoir sur le primaire en cas de retard ou de problème (coupure réseau, crash secondaire, etc).

> Vérifier qu'une table créée sur le primaire soit bien présente sur le secondaire.

Sur le primaire :

```
kubectl exec -it postgresql-demo-1 -- psql -c "create table ma_table (i int);" app

Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
CREATE TABLE
```

Sur le secondaire :

```
kubectl exec -it postgresql-demo-2 -- psql -c "\dt" app
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
          List of relations
 Schema |    Name    | Type  |  Owner
--------+------------+-------+----------
 public | ma_table | table | postgres
(1 row)
```

### 4.6.2 Emplacement des instances

Par défaut, l'opérateur CloudNativePG veille à déployer les instances PostgreSQL sur des nœuds différents afin de garantir la disponibilité. Cela permet de réduire les risques liés à un incident en s'assurant que toutes les instances ne sont pas affectées simultanément. Cette configuration permet également la répartition de la charge entre plusieurs nœuds pour des opérations de lecture.

> Trouver le nom du nœud où est déployée chaque instance.

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

```
NAME                        READY   STATUS    RESTARTS   AGE    IP              NODE
 ↪  NOMINATED NODE   READINESS GATES
postgresql-demo-1           1/1     Running   0          3h3m   10.244.228.71
 ↪  k8s-demo       <none>           <none>
postgresql-demo-2           1/1     Running   0          109m   10.244.41.198
 ↪  k8s-demo-m03   <none>           <none>
```

---

## 4.7 TESTS DE BASCULES

**But** : Tester et comprendre le mécanisme de bascule entre primaire et secondaire.

Trouver quelle est l'instance primaire du cluster `postgresql-demo`.

Vous devriez trouver que l'instance `postgresql-demo-1` est l'instance primaire.

```
kubectl cnpg status postgresql-demo
```

```
Cluster Summary
Name                 default/postgresql-demo
System ID:           7445242620478582804
PostgreSQL Image:    ghcr.io/cloudnative-pg/postgresql:17.0
Primary instance:    postgresql-demo-1
Primary start time:  2024-12-06 10:24:03 +0000 UTC (uptime 2m3s)
Status:              Cluster in healthy state
Instances:           2
Ready instances:     2
Size:                94M
Current Write LSN:   0/4050170 (Timeline: 1 - WAL File: 000000010000000000000004)

Continuous Backup status
Not configured

Physical backups
Name  Phase  Started at  Total  Transferred  Progress  Tablespaces
----  -----  ----------  -----  -----------  --------  -----------

Streaming Replication status
Replication Slots Enabled
Name               Sent LSN   Write LSN  Flush LSN  Replay LSN  Write Lag        Flush
 ↪  Lag        Replay Lag       State      Sync State  Sync Priority  Replication Slot
----               --------   ---------  ---------  ----------  ---------
 ↪  ---------        ----------       -----      ----------  -------------
 ↪  -----------------
postgresql-demo-2  0/4050170  0/4050170  0/4050170  0/4050170   00:00:00.001012
 ↪  00:00:00.007096  00:00:00.011149  streaming  async       0              active

Instances status
Name               Current LSN  Replication role  Status  QoS        Manager Version
 ↪  Node
----               -----------  ----------------  ------  ---        ---------------
 ↪  ----
```

```
postgresql-demo-1  0/4050170    Primary           OK      Burstable  1.24.0
 ↪  k8s-demo-m03
postgresql-demo-2  0/4050170    Standby (async)   OK      Burstable  1.24.0
 ↪  k8s-demo-m02
```

### 4.7.1 Bascule manuelle

> Promouvoir l'instance `postgresql-demo-2` comme nouveau primaire.

Attention, il y a bien un espace entre le nom du cluster et l'identifiant de l'instance.

```
kubectl cnpg promote postgresql-demo 2
```

```
{"level":"info","ts":"2024-12-06T10:40:18.767552528Z","msg":"Cluster is not
 ↪  healthy"}
Node postgresql-demo-2 in cluster postgresql-demo will be promoted
```

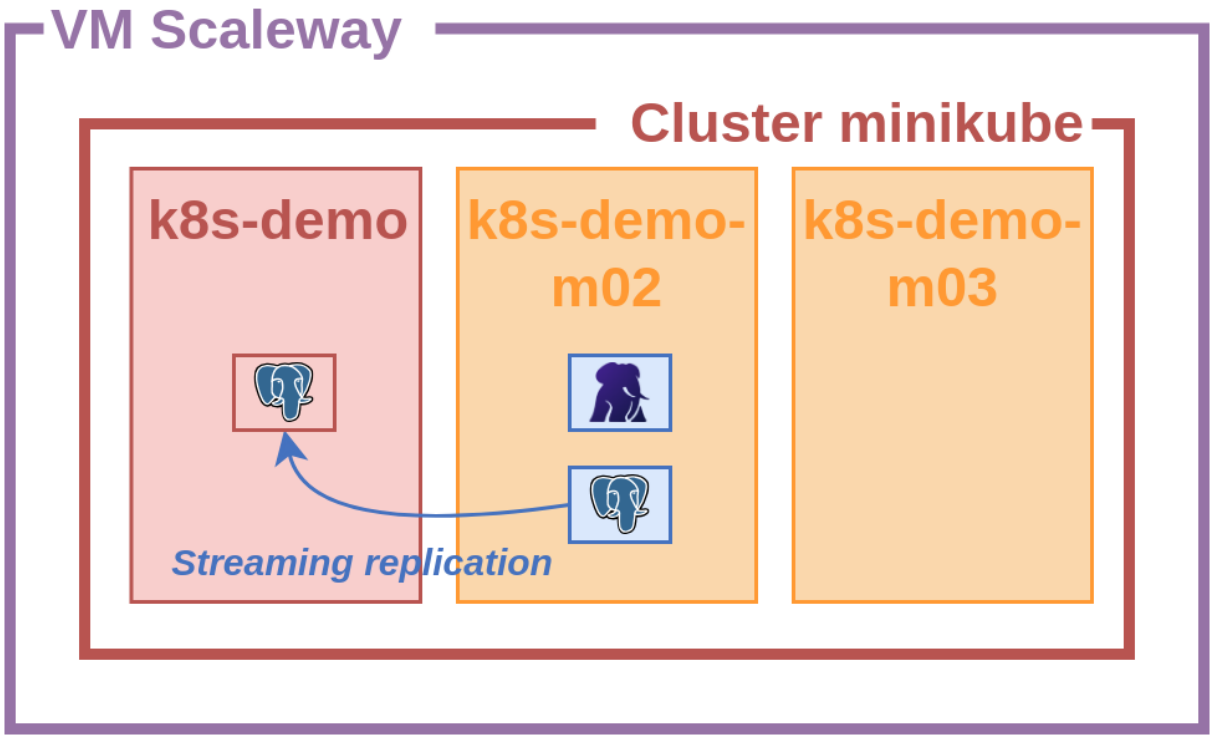
> Vérifier que le cluster est en bonne santé et que `postgresql-demo-2` est désormais l'instance primaire.

```
kubectl cnpg status postgresql-demo
```

```
[...]
Instances status
Name               Current LSN  Replication role  Status  QoS        Manager Version
 ↪  Node
----               -----------  ----------------  ------  ---        ---------------
 ↪  ----
postgresql-demo-2  0/6006778    Primary                OK      Burstable  1.24.0
 ↪  k8s-demo-m02
postgresql-demo-1  0/60000A0    Standby (starting up)  OK      Burstable  1.24.0
 ↪  k8s-demo
```

Les applications auront évidemment une coupure réseau, comme il y a une bascule.

Schématiquement, nous nous trouvons dans cette situation :




### 4.7.2 Bascule automatique

Lorsqu'une erreur survient sur le primaire le mode *failover* va être déclenché. Ce mécanisme sera démarré après une certaine durée modifiable via le paramètre `.spec.failoverDelay` (par défaut à 0) dans la définition du cluster PostgreSQL.

L'erreur peut être, par exemple, un problème sur le volume associé, le Pod primaire qui serait supprimé, le conteneur PostgreSQL qui serait KO, etc… (voir la documentation sur les *probes*[4]).

> Modifier le paramètre `.spec.failoverDelay` à 10 secondes.

```
[...]
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.0
  instances: 2
  failoverDelay: 10
[...]
```

> Dans une fenêtre, lancer la commande `watch kubectl get pod`.

```
watch kubectl get pod
```

[4] https://cloudnative-pg.io/documentation/current/instance_manager/#startup-liveness-and-readiness-probes

> Dans une autre fenêtre, détruire le Pod correspond à l'instance primaire. Regarder comment réagit le cluster.

```
kubectl delete pod postgresql-demo-2
```

```
pod "postgresql-demo-2" deleted
```

Une bascule sur le seul Pod disponible est faite après 10 secondes. L'instance `postgresql-demo-1` est automatiquement promue primaire. Dans la foulée, un Pod est recréé pour retrouver la situation initiale.

```
Instances status
Name               Current LSN  Replication role  Status  QoS         Manager Version
 ↪  Node
----               -----------  ----------------  ------  ---         ---------------
 ↪  ----
postgresql-demo-1  0/7001080    Primary               OK      Burstable  1.24.0
 ↪  k8s-demo
postgresql-demo-2  0/70000A0    Standby (file based)  OK      Burstable  1.24.0
 ↪  k8s-demo-m02
```

Voici un exemple avec un délai `.spec.failoverDelay` à 20 secondes.

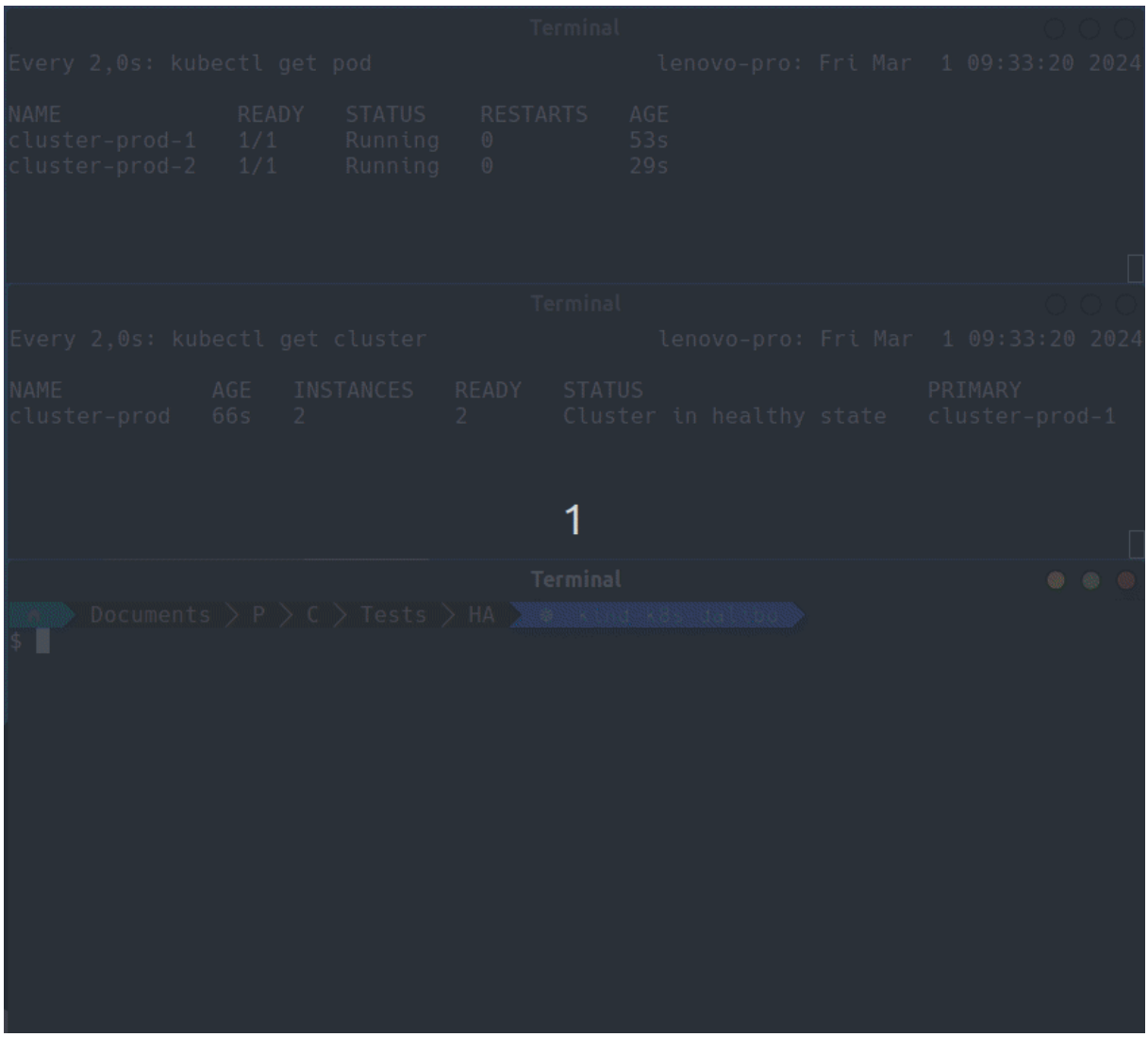
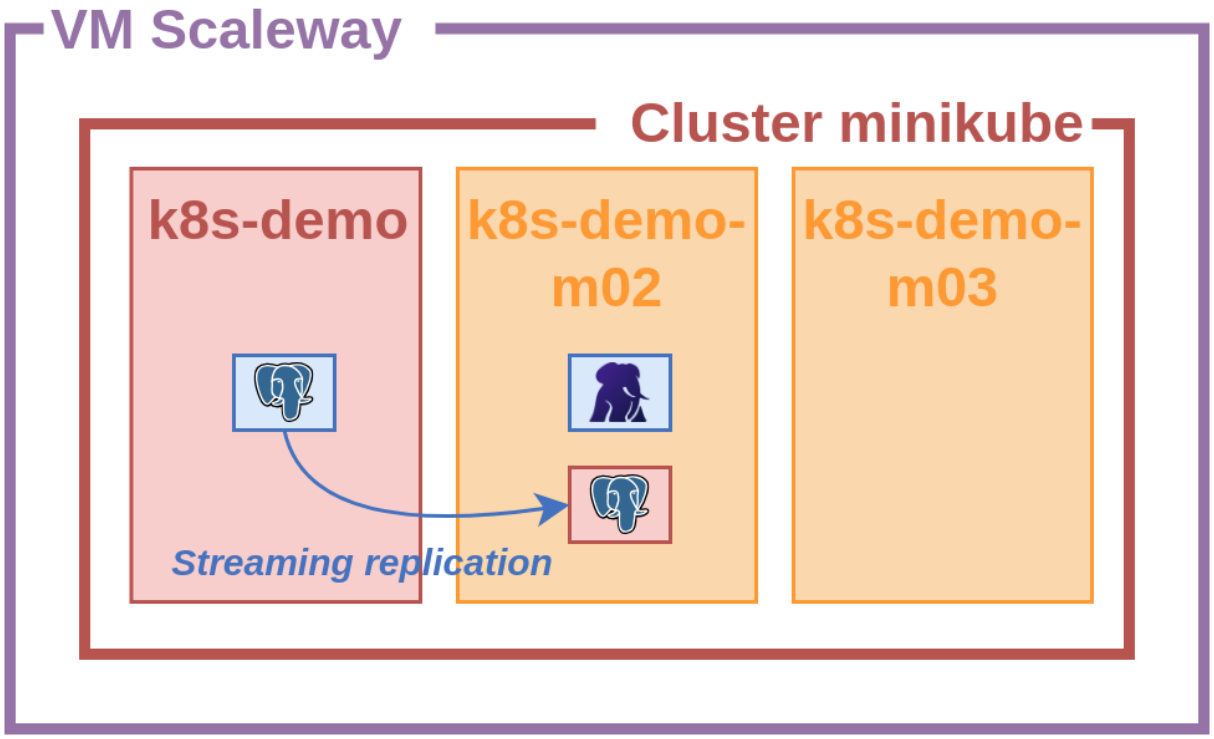



**Figure 4/ .1:** bascule automatique

Schématiquement, nous nous trouvons dans la situation précédente :


VM Scaleway
Cluster minikube
k8s-demo
k8s-demo-m02
k8s-demo-m03
Streaming replication

## 4.8 MISE EN PLACE D'UNE SAUVEGARDE PITR

**But** : Mettre en place une sauvegarde PITR sur un stockage S3 (archivage et sauvegarde complète).

Comme vous le savez certainement, il existe le concept de sauvegarde physique PITR comme mécanisme de sauvegarde d'une instance. Pour mettre en place cela, il est d'abord nécessaire de faire une sauvegarde physique de l'arborescence de l'instance. Ceci peut être fait à chaud. Le second élément essentiel est l'archivage des journaux de transactions (WAL) qui seront rejoués après une restauration pour rétablir un état cohérent.

En déployant une instance avec CloudNativePG, la seule solution de sauvegarde PITR utilisable est `Barman`. Très connu dans l'écosystème PostgreSQL, cet outil nous permet de faire la sauvegarde physique et l'archivage des WAL. Les commandes passées pour la mettre en place le seront de manière automatique mais une configuration doit être rajoutée dans le fichier YAML de définition.

### 4.8.1 Configuration

Créer le fichier `~/s3-creds.yaml` avec le contenu suivant.

```
---
apiVersion: v1
kind: Secret
metadata:
  name: s3-creds
type: Opaque
data:
  ACCESS_KEY_ID: U0NXQkhBWlFWMzk4N004Q1kxWEM=
  ACCESS_REGION: ZnItcGFy
  ACCESS_SECRET_KEY: MDVlZDFhZjMtNzc4Ni00MjE2LTlhZWYtOTQ5MmM3YzRjMzJh
```

Créer le `Secret` dans votre cluster Kubernetes avec la commande `kubectl apply -f ~/s3-creds.yaml`.

Ce `Secret` contient les informations de la clé API qui permettra de s'authentifier au `Bucket` et de déposer les WAL et sauvegardes.

Pour cette partie du TP, nous allons créer une autre instance PostgreSQL (`postgresql-with-backup-demo`), donc un objet de type `Cluster` et un nouveau nom.

Créer le fichier `~/postgresql-with-backup-demo.yaml` avec le contenu suivant.

N'oubliez pas de modifier **CHANGEME** dans le `destinationPath` en gardant bien le dernier `/` (mettre quelque chose de reconnaissable et unique)

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-with-backup-demo
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.0
  instances: 1
  storage:
    size: 5Gi
  postgresql:
    parameters:
      shared_buffers: "256MB"
      max_connections: "10"
      work_mem: "8MB"
      archive_timeout: "20min"
  resources:
    requests:
      memory: "256Mi"
      cpu: "0.5"
    limits:
      memory: "512Mi"
      cpu: "1"
  backup:
    barmanObjectStore:
      destinationPath: "s3://demo-cnpg/CHANGEME/"
      endpointURL: "https://s3.fr-par.scw.cloud"
      s3Credentials:
        accessKeyId:
          name: s3-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: s3-creds
          key: ACCESS_SECRET_KEY
        region:
          name: s3-creds
          key: ACCESS_REGION
      wal:
        compression: gzip
```

La configuration des paramètres `endpointURL` et `destinationPath` devra être adaptée selon votre fournisseur de stockage S3. Les paramètres ci-dessus fonctionnent bien avec Scaleway. Faites vraiment attention, vous risquerez de perdre beaucoup de temps … vraiment :).

> Créer le nouveau cluster PostgreSQL avec la commande :

```
kubectl apply -f ~/postgresql-with-backup-demo.yaml
```

> Vérifier dans les traces de cette nouvelle instance que l'archivage se passe correctement. Vous devriez voir des lignes comme `"msg": "Archived WAL file"`.

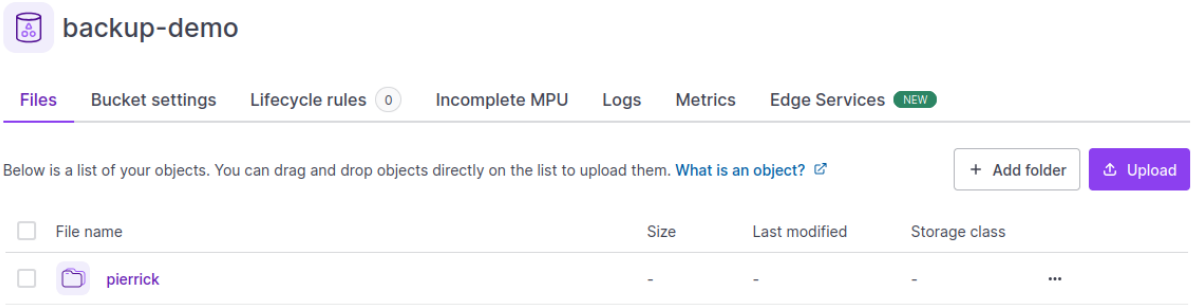
```
kubectl logs -f postgresql-with-backup-demo-1 | jq
```

```
[...]
```

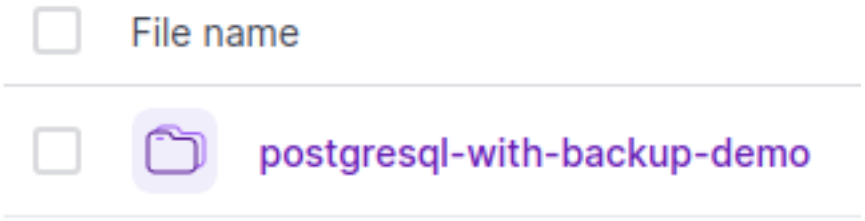
```
{
  "level": "info",
  "ts": "2024-11-20T13:29:44.953496179Z",
  "logger": "wal-archive",
  "msg": "Archived WAL file",
  "logging_pod": "postgresql-with-backup-demo-1",
  "walName": "pg_wal/000000010000000000000003",
  "startTime": "2024-11-20T13:29:44.115115707Z",
  "endTime": "2024-11-20T13:29:44.953469619Z",
  "elapsedWalTime": 0.838353912
}
```

> Demandez nous de vous montrer sur l'interface Scaleway le `Bucket` et le dossier qui vous "appartient".

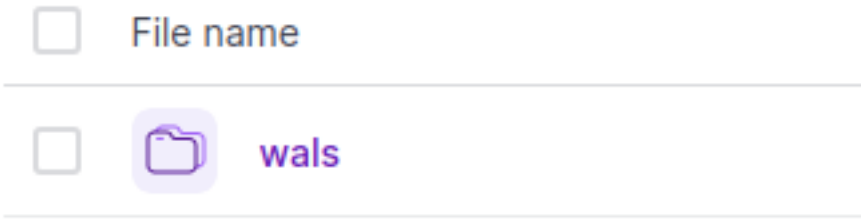
Pour ce TP, nous sommes passés par la solution `Object Storage` de Scaleway compatible S3. Voici un exemple de ce qu'il sera créé dans le `Bucket`.



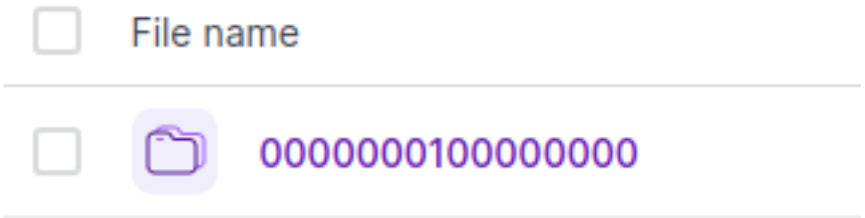

Le dossier `pierrick` est bien créé dans le `Bucket`.




On y retrouve dedans un dossier avec le nom du cluster PostgreSQL …




… qui contient lui-même un dossier `wals`.




Les journaux (WAL) sont enregistrés dans des dossiers qui reprennent la `timeline` de l'instance.

| File name | Size | Last modified | Storage class |
|---|---|---|---|
| 000000010000000000000001.gz | 2.31 MB | 20 minutes ago | STANDARD |
| 000000010000000000000002.gz | 122.91 KB | 15 minutes ago | STANDARD |
| 000000010000000000000003.gz | 16.5 KB | 10 minutes ago | STANDARD |

Et enfin, dans ce dernier dossier, se trouvent les journaux de transaction compressés. C'est un super point de départ. Cependant, pour le moment, il n'est pas possible de faire quelconque restauration comme il nous manque une sauvegarde complète de l'instance.

### 4.8.2 Sauvegarde complète de l'instance

> Créer le fichier `~/letsbackup.yaml` avec le contenu suivant :

```
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  name: first-backup
```

```
spec:
  cluster:
    name: postgresql-with-backup-demo
```

Il faut donner un nom à cet objet `Backup` et le nom du cluster PostgreSQL que l'on souhaite sauvegarder.

> Appliquer ce fichier avec `kubectl` :

```
kubectl apply -f ~/letsbackup.yaml
```

```
backup.postgresql.cnpg.io/first-backup created
```

> Vérifier le statut de l'objet `Backup` :

```
kubectl get backup
```

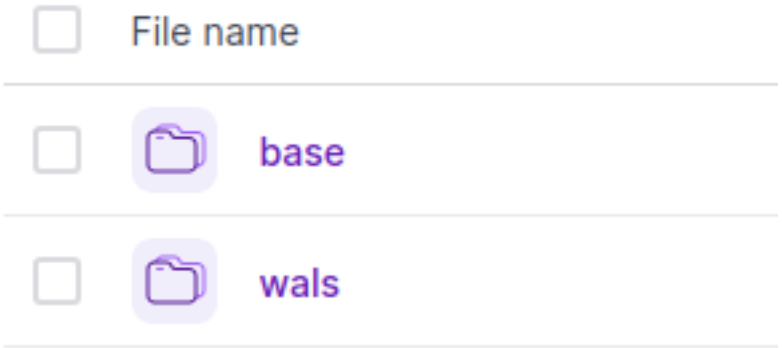
```
NAME           AGE     CLUSTER                      METHOD              PHASE       ERROR
first-backup   4m41s   postgresql-with-backup-demo   barmanObjectStore   completed
```

> Chercher dans les traces de l'instance une preuve que la sauvegarde complète s'est bien déroulée.

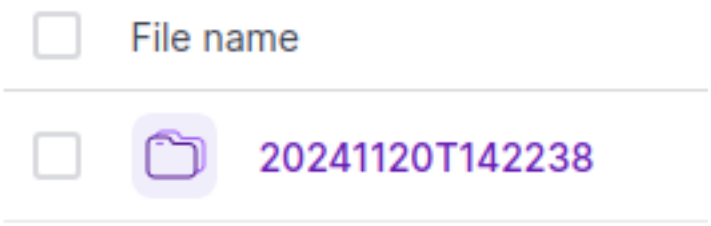
```
kubectl logs postgresql-with-backup-demo-1 | grep completed | jq
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
{
  "level": "info",
  "ts": "2024-11-20T14:22:46.968654454Z",
  "msg": "Backup completed",
  "backupName": "first-backup",
  "backupNamespace": "first-backup",
  "logging_pod": "postgresql-with-backup-demo-1"
}
```

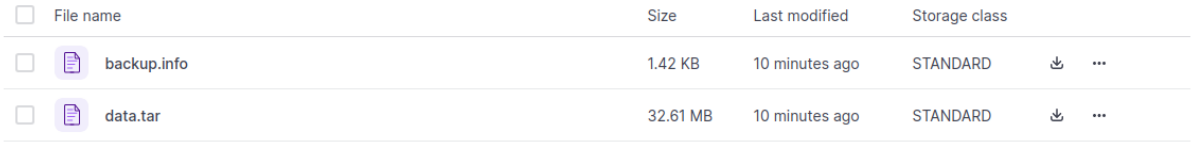
Au niveau de l'interface Scaleway, un nouveau dossier `base` est apparu à côté de `wals`.




Il contient toutes les sauvegardes faites jusqu'à présent.




La sauvegarde physique se trouve dans ce dossier et comporte un fichier d'informations et une archive `tar`.




Incroyable ! Nous avons une sauvegarde et un archivage des WAL qui semblent se dérouler correctement. Mais, qu'est ce qui se cache derrière cela ?

La première chose que nous pouvons chercher à savoir par exemple, est quel outil est utilisé pour archiver les journaux. Le paramètre `archive_command` nous donne un début de réponse.

```
kubectl exec -it postgresql-with-backup-demo-1 -- psql -c "SHOW archive_command"
```

```
                                archive_command
-------------------------------------------------------------------------------
 /controller/manager wal-archive --log-destination /controller/log/postgres.json %p
(1 row)
```

Un outil appelé `manager` présent dans le conteneur est utilisé avec l'option `wal-archive` suivie de plusieurs paramètres. %p est un *placeholders* qui permet d'indiquer le WAL courant. En regardant dans le code de l'opérateur, on peut retrouver facilement la trace du `manager` (voir par exemple le fichier https://github.com/cloudnative-pg/cloudnative-pg/blob/main/cmd/manager/main.go ou encore https://github.com/cloudnative-pg/cloudnative-pg/blob/main/internal/cmd/manager/walarchive/cmd.go#L17). La lecture du code nous fait comprendre que c'est *in fine* `Barman` qui est utilisé et plus exactement `barman-cloud`. Cet outil est installé au sein de l'image utilisée.

### 4.8.3 Générer de la donnée

> Se connecter à l'instance. Créer une table et insérer quelques données.

```
kubectl exec -it postgresql-with-backup-demo-1 -- psql
```

ou, via le plugin :

```
kubectl cnpg psql postgresql-with-backup-demo
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
psql (17.0 (Debian 17.0-1.pgdg110+1))
Type "help" for help.

postgres=# create table t1 (i int);
CREATE TABLE
postgres=# insert into t1 select generate_series(1, 100);
INSERT 0 100
postgres=# checkpoint ;
CHECKPOINT
```

Ne pas oublier d'exécuter l'ordre `CHECKPOINT` qui permettra de forcer la synchronisation des données sur disque et la création d'un point de cohérence sans attendre l'expiration de `checkpoint_timeout`.

---

## 4.9 PROCÉDER À UNE RESTAURATION PITR

**But** : Restaurer notre instance depuis la sauvegarde PITR existante.

Les restaurations se feront dans une autre instance PostgreSQL. Ce ne sera pas une restauration “in-place”.

Maintenant qu’une instance est déployée et qu’une sauvegarde a été faite, attardons-nous sur les manières qui existent pour restaurer une instance.

Aussi, c’est l’occasion de faire un petit rappel ! N’oubliez pas de tester vos procédures de restauration fréquemment !

> Simuler un crash. Détruire l’instance `postgresql-with-backup-demo` (Nous sommes bien évidemment ici dans un exercice de destruction maîtrisée par des professionnels).

```
kubectl delete -f ~/postgresql-with-backup-demo.yaml
```

> Créer un nouveau fichier `~/postgresql-restored-demo.yaml` avec le contenu suivant. L’idée est de créer une nouvelle instance `postgresql-restored-demo` et d’indiquer avec la section `bootstrap` qu’elle doit démarrer à partir d’une sauvegarde.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-restored-demo
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.0
  instances: 1
  storage:
    size: 5Gi
  postgresql:
    parameters:
      shared_buffers: "256MB"
      max_connections: "10"
      work_mem: "8MB"
      archive_timeout: "20min"
  resources:
    requests:
      memory: "256Mi"
      cpu: "0.5"
    limits:
      memory: "512Mi"
```

```
      cpu: "1"
  bootstrap:
    recovery:
      backup:
        name: first-backup
```

Le nom de la sauvegarde est indiqué dans l'attribut `name:` de la section `bootstrap.recovery.backup`. Pour déterminer quel `Backup` doit être restauré, vous pouvez en retrouver la liste avec :

```
kubectl get backup
```

Et même retrouver toutes les informations à propos de lui avec :

```
kubectl describe backup first-backup
```

> Créer votre nouvelle instance avec `kubectl apply -f ~/postgresql-restored-demo.yaml`. Lorsque l'instance est prête, s'y connecter et vérifier que les données s'y trouvent bien.

```
kubectl apply -f  ~/postgresql-restored-demo.yaml
```

```
kubectl exec -it postgresql-restored-demo-1 -- psql -c "select count(*) from t1;"
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
 count
-------
   100
(1 row)
```

La méthode que nous venons de suivre, présuppose que vous ayez accès à l'objet `Backup` créé dans le cluster Kubernetes. Mais qu'en est-il si c'est tout le cluster Kubernetes qui est en panne et doit être recréé ? Dans ce cas-là, l'objet `Backup` n'existe plus.

CloudNativePG propose une autre méthode pour restaurer une instance sans l'objet `Backup`. La configuration de la nouvelle instance doit contenir une section `externalClusters` qui contiendra les informations pour retrouver la sauvegarde.

> Créer le fichier `~/postgresql-externalcluster-demo.yaml` et ajouter le contenu suivant :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-external-cluster-demo
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.0
  instances: 1
```

```
  storage:
    size: 5Gi
  postgresql:
    parameters:
      shared_buffers: "256MB"
      max_connections: "10"
      work_mem: "8MB"
      archive_timeout: "20min"
  resources:
    requests:
      memory: "256Mi"
      cpu: "0.5"
    limits:
      memory: "512Mi"
      cpu: "1"

  bootstrap:
    recovery:
      source: postgresql-with-backup-demo

  externalClusters:
    - name: postgresql-with-backup-demo
      barmanObjectStore:
        destinationPath: "s3://demo-cnpg/CHANGEME/"
        endpointURL: "https://s3.fr-par.scw.cloud"
        s3Credentials:
          accessKeyId:
            name: s3-creds
            key: ACCESS_KEY_ID
          secretAccessKey:
            name: s3-creds
            key: ACCESS_SECRET_KEY
          region:
            name: s3-creds
            key: ACCESS_REGION
        wal:
          compression: gzip
```

Le paramètre de la section `bootstrap` est passé de `backup` à `recovery` avec comme paramètre le nom de la sauvegarde présente à récupérer (ici `postgresql-with-backup-demo`).

> Créer cette nouvelle instance et vérifier que les données dans la table `t1` soient bien présentes.

```
kubectl apply -f ~/postgresql-externalcluster-demo.yaml
```

```
cluster.postgresql.cnpg.io/postgresql-external-cluster-demo created
```

Dans les traces du Pod intermédiaire, on peut voir le début de la restauration :

```
{
  "level": "info",
  "ts": "2024-11-25T07:55:13.340842047Z",
  "msg": "Target backup found",
  "logging_pod": "postgresql-external-cluster-demo-1-full-recovery",
  "backup": {
    "backup_name": "backup-20241122160755",
    "backup_label": "'START WAL LOCATION: 0/3000028 (file
     ↪  000000010000000000000003)\\nCHECKPOINT LOCATION: 0/3000080\\nBACKUP METHOD:
     ↪  streamed\\nBACKUP FROM: primary\\nSTART TIME: 2024-11-22 16:07:56
     ↪  UTC\\nLABEL: Barman backup cloud 20241122T160755\\nSTART TIMELINE: 1\\n'",
    "begin_time": "Fri Nov 22 16:07:55 2024",
    "end_time": "Fri Nov 22 16:07:58 2024",
    "BeginTime": "2024-11-22T16:07:55Z",
    "EndTime": "2024-11-22T16:07:58Z",
    "begin_wal": "000000010000000000000003",
    "end_wal": "000000010000000000000003",
    "begin_xlog": "0/3000028",
    "end_xlog": "0/3000158",
    "systemid": "7440134501184790547",
    "backup_id": "20241122T160755",
    "error": "",
    "timeline": 1
  }
}

{
  "level": "info",
  "ts": "2024-11-25T07:55:13.340842047Z",
  "msg": "Target backup found",
  "logging_pod": "postgresql-external-cluster-demo-1-full-recovery",
  "backup": {
    "backup_name": "backup-20241122160755",
    "backup_label": "'START WAL LOCATION: 0/3000028 (file
     ↪  000000010000000000000003)\\nCHECKPOINT LOCATION: 0/3000080\\nBACKUP METHOD:
     ↪  streamed\\nBACKUP FROM: primary\\nSTART TIME: 2024-11-22 16:07:56
     ↪  UTC\\nLABEL: Barman backup cloud 20241122T160755\\nSTART TIMELINE: 1\\n'",
    "begin_time": "Fri Nov 22 16:07:55 2024",
    "end_time": "Fri Nov 22 16:07:58 2024",
    "BeginTime": "2024-11-22T16:07:55Z",
    "EndTime": "2024-11-22T16:07:58Z",
    "begin_wal": "000000010000000000000003",
    "end_wal": "000000010000000000000003",
    "begin_xlog": "0/3000028",
    "end_xlog": "0/3000158",
    "systemid": "7440134501184790547",
    "backup_id": "20241122T160755",
    "error": "",
```

```
    "timeline": 1
  }
}
```

Les données sont bien là :

```
kubectl exec -it postgresql-external-cluster-demo-1 -- psql -c 'select count(*) from
 ↪  t1;'

Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
 count
-------
   100
(1 row)
```

Dans cet exemple, la restauration s'est faite sur le même cluster Kubernetes. Dans le cas où vous devez la faire ailleurs, n'oubliez pas de recréer le `Secret` qui contient les informations de l'API Key nécessaire à l'accès au stockage S3.

todo expliquer que la restauration doit se faire sur une autre instance ! attention si on veut garder le même nom.

Supprimer les instances qui ne vont plus nous servir par la suite.

```
kubectl delete -f postgresql-externalcluster-demo.yaml
kubectl delete -f postgresql-demo.yaml
```

Il ne doit rester que le cluster PostgreSQL `postgresql-restored-demo`.

```
kubectl get pod
```

```
NAME                          READY   STATUS    RESTARTS      AGE
postgresql-restored-demo-1    1/1     Running   2 (26m ago)   2d15h
```

## 4.10 MONTÉE DE VERSION MINEURE DE POSTGRESQL

**But** : Effectuer une montée de version mineure de PostgreSQL.

La version de PostgreSQL est indiquée dans le nom et le tag de l'image déployée. La modification de celle-ci entraine une montée de version de l'instance. Cette montée de version peut se faire automatiquement ou de manière supervisée (appelée "manuelle" dans la documentation).

### 4.10.1 Méthode automatique

> Dans une autre session SSH, lancer la commande `watch kubectl get pods` pour voir ce qu'il se passe pendant la montée de version.

Vous devriez voir la chose suivante avec un rafraichissement toutes les 2 secondes.

```
Every 2.0s: kubectl get pods         scw-boring-keller: Mon Nov 25 08:40:03 2024

NAME                          READY   STATUS    RESTARTS      AGE
postgresql-restored-demo-1    1/1     Running   2 (46m ago)   2d16h
```

> Modifier la version de PostgreSQL de `17.0` à `17.1` dans le fichier `~/postgresql-restored-demo.yaml` et appliquer la modification avec `kubectl apply`.

```
[...]
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.1
[...]
```

Le Pod de l'instance en version 17.0 est supprimé.

```
NAME                          READY   STATUS
postgresql-restored-demo-1    1/1     Terminating
```

Un nouveau pod est créé.

```
NAME                          READY   STATUS            RESTARTS   AGE
postgresql-restored-demo-1    0/1     PodInitializing   0          5s
```

Peu de temps après, il passe à l'état `Running`.

```
NAME                          READY   STATUS    RESTARTS   AGE
postgresql-restored-demo-1    1/1     Running   0          22s
```

Un `select version()` indique que nous sommes bien passés en version 17.1.

Par défaut, une instance PostgreSQL est déployée de telle sorte que la montée de version se fasse automatiquement, c'est-à-dire que l'opérateur arrête puis redémarre l'instance tout seul. Voyons maintenant le cas d'une montée de version en mode `supervised`.

### 4.10.2 Méthode *supervised*

Le paramètre `primaryUpdateStrategy` permet de définir la stratégie à suivre lors d'une mise à jour de l'instance primaire. Il est positionné par défaut à `unsupervised`. C'est le comportement que nous venons de voir avec l'exemple précédent.

> Positionner ce paramètre-là à `supervised`, modifier la version en la passant de 17.1 à 17.2 et tenter de faire la montée de version.

```
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.2
  instances: 1
  primaryUpdateStrategy: supervised
```

```
kubectl apply -f ~/postgresql-restored-demo.yaml
```

Aïe …

```
The Cluster "postgresql-restored-demo" is invalid: spec.primaryUpdateStrategy:
Invalid value: "supervised": supervised update strategy is not allowed for clusters
with a single instance
```

Nous venons de découvrir une première subtilité. Ce paramètre n'est en réalité pas utilisable avec une seule instance. En effet ce paramètre permet de contrôler la manière dont est redémarrée l'instance primaire après que **tous** les secondaires ont été mis à jour. Comme ici, nous n'avons que le primaire de déployé.

Ceci nous permet donc de voir redéployer un secondaire. Rien de plus simple.

> Ajouter un secondaire à votre cluster PostgreSQL en modifiant la ligne `instances` du fichier `postgresql-restored-demo.yaml` et en appliquant la modification.

```
[...]
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.2
  instances: 2
[...]
```

Un premier Pod *join* va être créé puis le Pod de l'instance secondaire sera finalement déployé. Nous nous retrouvons donc avec deux Pods reprenant le nom du cluster, incrémentés de 1.

```
kubectl get pod
```

```
NAME                                  READY   STATUS      RESTARTS   AGE
postgresql-restored-demo-1            1/1     Running     0          61m
postgresql-restored-demo-2            1/1     Running     0          20s
```

Comme nous avions modifié la version de l'image en 17.2, l'instance secondaire qui vient d'être déployée est bien dans cette version. La version du primaire est quant à elle restée en 17.1, ce qui est normal comme nous sommes dans une montée de version supervisée.

```
# primaire
kubectl exec -it postgresql-restored-demo-1 -- psql -c 'show server_version;'
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
         server_version
--------------------------------
 17.1 (Debian 17.1-1.pgdg110+1)
(1 row)

# secondaire
kubectl exec -it postgresql-restored-demo-2 -- psql -c 'show server_version;'
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
         server_version
--------------------------------
 17.2 (Debian 17.2-1.pgdg110+1)
(1 row)
```

Il nous reste donc à signifier à CloudNativePG que le primaire peut être mis à jour à son tour. Pour cela, il faut passer par le *plugin* CloudNativePG de `kubectl` et procéder à une bascule sur le secondaire qui deviendra le nouveau primaire avec la ligne de commande : `kubectl cnpg promote [cluster] [new_primary]`.

> Faite une bascule manuelle sur l'instance secondaire.

```
kubectl cnpg promote postgresql-restored-demo postgresql-restored-demo-2
```

L'ancien secondaire est désormais primaire comme on peut le voir dans la sortie de `kubectl cnpg status postgresql-restored-demo` qui donne l'état du cluster PostgreSQL, en particulier, avec la ligne `Primary instance: postgresql-restored-demo-2`, ou encore dans le tableau.

```
Name                  Current LSN  Replication role  Status  QoS      Manager Version
----                  -----------  ----------------  ------  ---
---------------  ----
postgresql-restored-demo-2  0/D001210   Primary        OK    Burstable  1.24.0
demo-m03
postgresql-restored-demo-1  0/D001210   Standby (async)  OK    Burstable  1.24.0
dem
```

Les deux instances sont bien en version 17.2.

```
kubectl exec -it postgresql-restored-demo-1 -- psql -c 'show server_version;'
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
         server_version
--------------------------------
 17.2 (Debian 17.2-1.pgdg110+1)
(1 row)
```

```
kubectl exec -it postgresql-restored-demo-2 -- psql -c 'show server_version;'
```

```
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
         server_version
--------------------------------
 17.2 (Debian 17.2-1.pgdg110+1)
(1 row)
```

Il existe d'autres cas où le redémarrage des instances est nécessaire. Par exemple, si un paramètre comme `max_connections` ou `shared_buffers` a été modifié. Dans ce cas-là, si vous faites toujours une montée de version supervisée, il vous est possible de ne pas faire de bascule mais simplement de redémarrer l'instance primaire avec la ligne de commande `kubectl cnpg restart [cluster] [current_primary];`

## 4.11 MONTÉE DE VERSION DE L'OPÉRATEUR

**But** : Effectuer une montée de version de l'opérateur CNPG.

Nous avons déployé la version `1.24.0` de l'opérateur. Nous allons nous intéresser à la manière de le mettre à jour.

Lorsqu'une nouvelle version de l'opérateur est déployée, un nouveau Pod se crée. Lorsque celui-ci est prêt, l'ancien opérateur est tout simplement supprimé. Cette mise à jour déclenche également la mise à jour d'un composant présentant dans les `Pods` des instances PostgreSQL.

Lorsqu'un Pod PostgreSQL est déployé, un `InitContainer` est créé en amont et permet de récupérer du code correspondant au `manager`. Il permet de contrôler l'instance, son cycle, ses redémarrages, etc. La version de ce manager est étroitement liée à la version de l'opérateur. Pour information, c'est ce processus qui va lancer PostgreSQL et qui aura le `pid` 1 dans le Pod.

```
cat /proc/1/cmdline
/controller/managerinstancerun--status-port-tls--log-level=info
```

**Attention** si vous avez utilisé votre opérateur pour déployer plusieurs instances PostgreSQL, lorsque vous mettez à jour l'opérateur, **tous** les `Pods` seront, mis à jour en même temps (ou quasiment). Il y aura donc une coupure de service pour chaque instance. C'est le fonctionnement par défaut.

> Avoir deux nouvelles connexions ssh à votre machine virtuelle et passer sous l'utilisateur `dalibo`.

> Dans la première console, lancer la commande `watch` suivante :

```
watch kubectl get pod
```

> Dans la seconde console, lancer la commande `watch` suivante :

```
watch kubectl get pod -n cnpg-system
```

> Dans une autre console, appliquer les fichiers yaml correspondant à la version `1.24.1` de l'opérateur.

```
kubectl apply --server-side -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/release-
↪  1.24/releases/cnpg-1.24.1.yaml
```

> Regarder ce qui se passe au niveau des différents `Pods` (opérateur et PostgreSQL)

Les instances PostgreSQL déployées par l'opérateur sont redémarrées lors d'une montée de version de l'opérateur. Selon la configuration des instances, une opération manuelle sera nécessaire pour mettre à jour le primaire.

Il existe une méthode pour éviter ce comportement. Cependant elle ne garantit pas le critère immuable que devrait suivre un Pod.

À titre d'information, voici la méthode à suivre pour y parvenir. Pour cela il faut modifier la configuration de l'opérateur en passant le paramètre `ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES` à `true`. Cela permettra de mettre à jour le `manager` sans pour autant redémarrer le Pod complet. Cette configuration doit être faite dans un objet `ConfigMap`.

> Créer le fichier `~/config-cnpg.yaml` avec le contenu suivant et appliquer le avec `kubectl`.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cnpg-controller-manager-config
  namespace: cnpg-system
data:
  ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES: 'true'
```

```
kubectl apply -f ~/config-cnpg.yaml
```

> Redémarrer l'opérateur pour la bonne prise en compte de la nouvelle configuration.

```
kubectl rollout restart deployment \
    -n cnpg-system \
    cnpg-controller-manager
```

```
deployment.apps/cnpg-controller-manager restarted
```

---

## 4.12 EXERCICES OPTIONNELS

**But** : Découvrir des fonctionnalités plus complexes.

### 4.12.1 Mise en place d'un cluster Kubernetes (minikube)

**But** : Mettre en place un cluster Kubernetes avec minikube et installer les outils complémentaires.**

> Toutes les commandes seront exécutées en étant connecté avec l'utilisateur `dalibo` qui possède les droits `sudo`.

> Il est tout d'abord nécessaire d'installer `Docker` qui sera reconnu par `minikube` comme *runtime* (répondez Y(es) à toutes les questions) :

```
# Add Docker's official GPG key:
sudo apt update
sudo apt install -y ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/debian/gpg -
o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
 "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt update

# Install it
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-
buildx-plugin docker-compose-plugin
```

> Installer l'outil `jq` et `yamllint` que nous utiliserons plus tard :

```
sudo apt install -y jq yamllint
```

> Installer le plugin CNPG (en version 1.24.0) pour `kubectl` que nous utiliserons plus tard :

```
wget https://github.com/cloudnative-pg/cloudnative-
 ↪  pg/releases/download/v1.24.0/kubectl-cnpg_1.24.0_linux_x86_64.deb
 ↪  --output-document kube-plugin.deb

sudo dpkg -i kube-plugin.deb
```

> Ajouter l'utilisateur `dalibo` au groupe `docker` :

```
sudo usermod -aG docker dalibo && newgrp docker
```

(voir aussi https://docs.docker.com/engine/install/debian/#installation-methods )

> Installer l'outil `minikube` :

```
curl -LO
 ↪  https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64
```

> Créer un cluster Kubernetes nommé `k8s-demo` avec un *Master* et deux *Workers* grâce à la commande `minikube`. L'option `-p` permet de nommer le cluster en question et l'option `--cni` permet d'indiquer quel *Container Network Interface* utiliser.

```
minikube start -p k8s-demo --network-plugin=cni --cni=calico
minikube node add -p k8s-demo
minikube node add -p k8s-demo
```

> Pour notre cluster de démo, l'ajout d'`addons` pour le stockage est nécessaire. Passer les commandes suivantes.

```
minikube addons enable volumesnapshots -p k8s-demo
minikube addons enable csi-hostpath-driver -p k8s-demo

minikube addons disable storage-provisioner -p k8s-demo
minikube addons disable default-storageclass -p k8s-demo
```

> Pour interagir avec un cluster Kubernetes, l'outil `kubectl` est fait pour ça. Installer le avec :

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s
 ↪  https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl
kubectl version --client
```

Lorsque `minikube` crée un cluster, il génère automatiquement un fichier *kubeconfig* qu'il place dans `~/.kube/config` et qui stocke toutes les informations de connexions au cluster Kubernetes.

> Regarder le contenu du fichier `~/.kube/config`.

```
cat ~/.kube/config
```

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /home/dalibo/.minikube/ca.crt
    extensions:
    - extension:
        last-update: Fri, 29 Nov 2024 08:31:07 UTC
        provider: minikube.sigs.k8s.io
        version: v1.34.0
      name: cluster_info
    server: https://192.168.49.2:8443
  name: k8s-demo
contexts:
- context:
    cluster: k8s-demo
    extensions:
    - extension:
        last-update: Fri, 29 Nov 2024 08:31:07 UTC
        provider: minikube.sigs.k8s.io
        version: v1.34.0
      name: context_info
    namespace: default
    user: k8s-demo
  name: k8s-demo
current-context: k8s-demo
kind: Config
preferences: {}
users:
- name: k8s-demo
  user:
    client-certificate: /home/dalibo/.minikube/profiles/k8s-demo/client.crt
    client-key: /home/dalibo/.minikube/profiles/k8s-demo/client.key
```

> À partir de là, vous avez un cluster Kubernetes multi-nœuds qui tourne sur la machine qui est à votre disposition.

> Vous pouvez vérifier le nombre de nœuds déployés avec la commande `kubectl get nodes` :

```
kubectl get nodes
```

```
NAME           STATUS   ROLES           AGE     VERSION
k8s-demo       Ready    control-plane   3m48s   v1.31.0
k8s-demo-m02   Ready    <none>          3m22s   v1.31.0
k8s-demo-m03   Ready    <none>          2m59s   v1.31.0
```

> Modifier la `StorageClass` par défaut avec la commande suivante :

```
kubectl patch storageclass csi-hostpath-sc -p '{"metadata":
 ↪  {"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
```

Une `StorageClass`, ou classe de stockage en bon français, est un objet Kubernetes qui indique les caractéristiques d'un stockage disponible. Ici, nous définition la classe `csi-hostpath-sc` comme celle par défaut. Elle permet de créer de volumes sur les *hosts*.

### 4.12.2 Mise en place de sauvegardes programmées

Il est possible de programmer des sauvegardes régulières avec la ressource `ScheduledBackup`.

Voici un exemple de définition qui permet de déclencher une sauvegarde appelée `backup-every-day` tous les jours à 16h00 pour le cluster `postgresql-with-backup-demo` :

```
apiVersion: postgresql.cnpg.io/v1
kind: ScheduledBackup
metadata:
  name: backup-every-day
spec:
  schedule: "0 0 16 * * *"
  backupOwnerReference: self
  cluster:
    name: postgresql-with-backup-demo
```

Attention, l'option `schedule` prend bien six paramètres (le premier étant les secondes), contrairement au `CronJob` dans Kubernetes ou aux lignes de `/etc/crontab` qui en prenne que cinq.

> Créer le fichier `~/backup-every-day.yaml` avec le contenu ci-dessus en modifiant l'heure d'exécution pour que la sauvegarde s'exécute dans 5 à 10 minutes.

> Créer l'objet `ScheduledBackup` avec `kubectl`.

```
kubectl apply -f backup-every-day.yaml
```

```
scheduledbackup.postgresql.cnpg.io/backup-every-day created
```

> Suivez les traces de l'instance avec `kubectl cnpg logs cluster postgresql-with-backup-demo | jq`. Vous devriez voir le déclenchement de la sauvegarde.

```
kubectl cnpg logs cluster postgresql-with-backup-demo | jq
```


```
{
  "level": "info",
  "ts": "2024-12-04T15:52:00Z",
  "msg": "WAL archiving is working",
  "logging_pod": "postgresql-with-backup-demo-1"
}
{
  "level": "info",
  "ts": "2024-12-04T15:52:00Z",
  "msg": "Starting barman-cloud-backup",
  "backupName": "backup-every-day-20241204155200",
  "backupNamespace": "backup-every-day-20241204155200",
  "logging_pod": "postgresql-with-backup-demo-1",
  "options": [
    "--user",
    "postgres",
    "--name",
    "backup-20241204155200",
    "--endpoint-url",
    "https://s3.fr-par.scw.cloud",
    "--cloud-provider",
    "aws-s3",
    "s3://backup-demo/pierrick/",
    "postgresql-with-backup-demo"
  ]
}
```


Vous verrez alors la sauvegarde sur votre emplacement de stockage S3.

### 4.12.3 Mettre en place une réplication synchrone

Il existe plusieurs méthodes pour mettre en place une réplication synchrone. Le but ici n'est pas de les évoquer ni de les comparer, mais simplement de voir le principe de la configuration.

Pour l'exemple, nous mettrons en place la méthode par `Quorum`.

> Modifier la configuration de l'instance en rajoutant le bloc suivant à votre fichier `~/postgresql-demo.yaml`.

```
[...]
  postgresql:
```

```
    synchronous:
      method: any
      number: 1
[...]
```

```
kubectl apply -f postgresql.yaml
```

```
cluster.postgresql.cnpg.io/postgresql-demo configured
```

> Vérifier que la réplication est synchrone en regardant le champ `sync_state` de la vue `pg_stat_replication`.

```
postgres=# select * from pg_stat_replication\gx
-[ RECORD 1 ]----+------------------------------
pid              | 2370
usesysid         | 16388
usename          | streaming_replica
application_name | postgresql-demo-2
client_addr      | 10.244.41.198
client_hostname  |
client_port      | 35530
backend_start    | 2024-11-29 07:47:00.777328+00
backend_xmin     |
state            | streaming
sent_lsn         | 0/C000000
write_lsn        | 0/C000000
flush_lsn        | 0/C000000
replay_lsn       | 0/C000000
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 1
sync_state       | quorum
reply_time       | 2024-11-29 09:58:35.787111+00
```

`sync_state` est bien à `quorum`.

Plus d'informations sur la documentation[5].

### 4.12.4 Déploiement d'une application pgAdmin4

Pour voir comment une application peut se connecter à une instance, nous allons déployer pgAdmin dans le cluster Kubernetes.

[5] https://cloudnative-pg.io/documentation/current/replication/#synchronous-replication

> Ouvrir une nouvelle session SSH. Passer en tant qu'utilisateur `dalibo`.

> Créer le fichier `~/pgadmin.yaml` avec le contenu suivant et le déployer dans le cluster Kubernetes.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pgadmin
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pgadmin
  template:
    metadata:
      labels:
        app: pgadmin
    spec:
      containers:
        - name: pgadmin
          image: dpage/pgadmin4
          ports:
            - containerPort: 80
          env:
            - name: PGADMIN_DEFAULT_EMAIL
              value: admin@example.com
            - name: PGADMIN_DEFAULT_PASSWORD
              value: admin
```

```
kubectl apply -f ~/pgadmin.yaml
```

```
deployment.apps/pgadmin created
```

> Récupérer le nom du Pod pgAdmin déployé, et lancer la commande suivante :

```
kubectl port-forward --address 0.0.0.0 pgadmin-*****-***** 8888:80
```

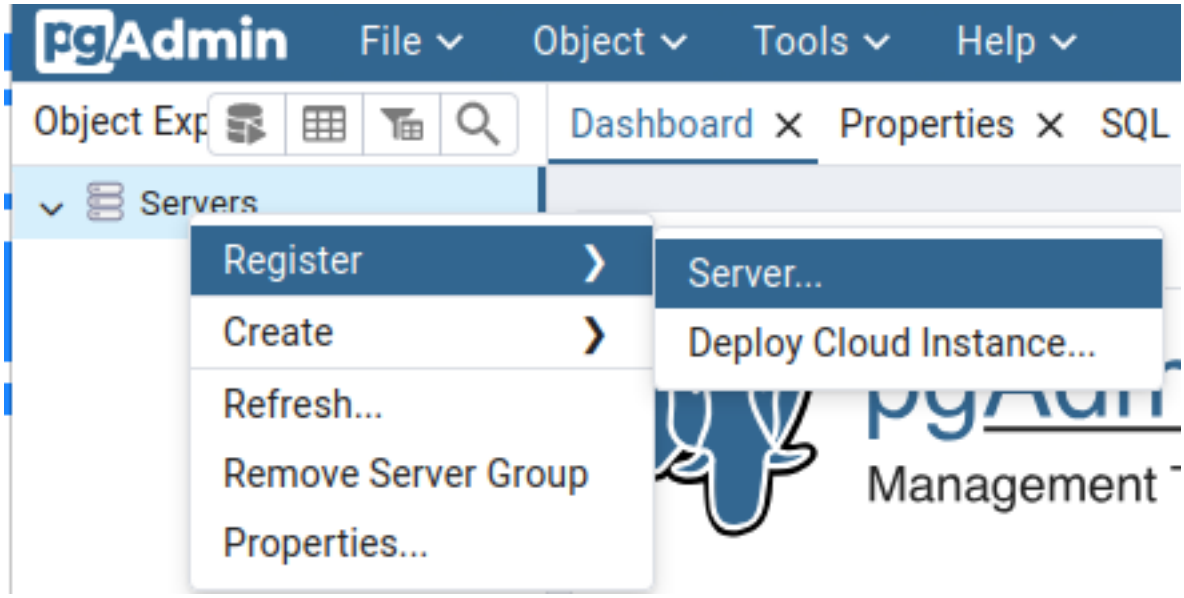
Cette commande permet de *forwarder* le trafic entrant sur le port TCP 8888 de la machine vers le port 80 du Pod pgAdmin, rendant ainsi accessible l'application. Cette méthode reste valide pour des démonstrations, n'allez pas mettre ça en production :).

> Accéder à l'interface de pgAdmin via votre navigateur `http://adresseippublique:8888` et connectez vous à l'interface (admin@example.com / admin).

L'adresse IP publique de la machine peut être retrouvée avec la commande suivante :

```
ip -f inet addr show ens2
2: ens2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group
 ↪  default qlen 1000
    altname enp0s2
    inet 51.158.67.253/32 metric 100 scope global dynamic ens2
       valid_lft 842sec preferred_lft 842sec
```


Créer une nouvelle connexion avec les informations suivantes :




- Name : `postgresql-demo` (*Onglet General*) ;
- Host name/address : `postgresql-demo-rw` (*Onglet Connection*) ;
- Port : `5432` ;

- Username : app;
- Password : celui récupéré dans le `Secret`;
- Cliquer sur `Save`.

> Créer une nouvelle connexion avec cette fois-ci `postgresql-demo-ro` comme paramètre `Host name/address` et créer une table `CREATE TABLE ma_table (i int);`.

Cette commande ne pourra pas être exécutée comme le `Service` renvoie sur une instance secondaire qui est nécessairement en lecture seule. Le message d'erreur sera :

```
CREATE TABLE ma_table (i int);

ERROR:  cannot execute CREATE TABLE in a read-only transaction

SQL state: 25006
```

Vous avez maintenant l'application pgAdmin déployée dans Kubernetes qui a accès à l'instance `postgresql-demo`. L'exemple ci-dessus montre comment une application peut accéder à une base de données en utilisant la ressource `Service` prévue à cet effet. Application et base se trouvent toutes deux dans Kubernetes. Accéder à l'instance PostgreSQL depuis une application externe (i.e déployée ailleurs) est plus complexe, demande le déploiement d'autres ressources … mais ne sera pas traité dans ce *workshop*.

---

# 5/ Conclusion

- Échanges sur PostgreSQL dans Kubernetes
- Prise en main de l'opérateur CloudNativePG
- Attentions changements qu'implique un déploiement sur Kubernetes (et pas que DBA !)
- Choisissez une solution qui répond à vos besoins !

# 6/ Références

- CloudNativePG : https://cloudnative-pg.io
- Dalibo : https://dalibo.com

---

# 7/ Remerciements


- Alain Lesage
- David Bidoc
- Robin Portigliatti
- Jehan-Guillaume de Rorthais
- Pierrick Chovelon

# Notes

# Notes

# Notes

# Nos autres publications

## FORMATIONS

- DBA1 : Administration PostgreSQL
  https://dali.bo/dba1
- DBA2 : Administration PostgreSQL avancé
  https://dali.bo/dba2
- DBA3 : Sauvegarde et réplication avec PostgreSQL
  https://dali.bo/dba3
- DEVPG : Développer avec PostgreSQL
  https://dali.bo/devpg
- PERF1 : PostgreSQL Performances
  https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés
  https://dali.bo/perf2
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

## LIVRES BLANCS

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

## TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.

# 8/ DALIBO, L'Expertise PostgreSQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.