Module K1

Découverte de CloudNativePG

25.07

Table des matières

Sur ce document . . . 1
Chers lectrices & lecteurs, . . . 1
À propos de DALIBO . . . 1
Remerciements . . . 2
Forme de ce manuel . . . 2
Licence Creative Commons CC-BY-NC-SA . . . 2
Marques déposées . . . 3
Versions de PostgreSQL couvertes . . . 3

1/ CloudNativePG **5**
1.1 Au menu . . . 6
1.1.1 Objectifs . . . 6
1.1.2 Opérateurs PostgreSQL . . . 6
1.2 L'opérateur CloudNativePG . . . 8
1.2.1 Ce que permet CloudNativePG . . . 8
1.2.2 Les images utilisées . . . 9
1.2.3 Versions supportées . . . 10
1.2.4 Installation de l'opérateur . . . 11
1.2.5 Installation de l'opérateur . . . 11
1.2.6 Montée de version de l'opérateur . . . 13
1.3 Gestion déclarative . . . 15
1.3.1 Principe . . . 15
1.3.2 Cluster . . . 15
1.3.3 Éléments initiaux . . . 16
1.3.4 Modification des éléments initiaux . . . 18
1.3.5 Database . . . 19
1.3.6 Role . . . 20
1.3.7 Tablespace . . . 22
1.4 Configuration de l'instance . . . 24
1.4.1 postgresql.conf . . . 24
1.4.2 pg_hba.conf et pg_ident.conf . . . 25
1.5 Administration de l'instance . . . 27
1.5.1 Administration de l'instance . . . 27
1.5.2 Plugin kubectl . . . 27
1.5.3 Connexion . . . 28
1.5.4 Traces . . . 29
1.5.5 Réplication . . . 30
1.5.6 Archivage . . . 32
1.5.7 Sauvegarde . . . 33
1.5.8 Sauvegarde sur stockage objets . . . 33
1.5.9 Sauvegarde par Volume Snapshot . . . 34
1.5.10 Backup . . . 34

1.5.11 ScheduledBackup . . . 35
1.5.12 Restauration . . . 36
1.5.13 Montée de version mineure de PostgreSQL . . . 38
1.5.14 Haute disponibilité et bascule . . . 39
1.5.15 Hibernation et fencing . . . 40
1.6 Conclusion . . . 43
1.6.1 Questions . . . 43
1.7 Quiz . . . 44
1.8 Travaux pratiques . . . 45
1.9 Prise en main du cluster Kubernetes (minikube) . . . 46
1.10 Installation de l'opérateur CloudNativePG . . . 47
1.11 Déploiement d'instances PostgreSQL . . . 48
1.12 Éléments initiaux . . . 50
1.12.1 Modifications de paramètres de configuration . . . 51
1.12.2 Créer un rôle . . . 51
1.12.3 Créer une base de données . . . 52
1.13 Déploiement d'une instance secondaire . . . 53
1.13.1 Configuration par défaut . . . 53
1.13.2 Emplacement des instances . . . 53
1.14 Tests de bascules . . . 54
1.14.1 Bascule manuelle . . . 54
1.14.2 Bascule automatique . . . 54
1.15 Mise en place d'une sauvegarde PITR . . . 55
1.15.1 Configuration . . . 55
1.15.2 Sauvegarde complète de l'instance . . . 56
1.15.3 Générer de la donnée . . . 57
1.16 Procéder à une restauration PITR . . . 58
1.17 Montée de version mineure de PostgreSQL . . . 60
1.17.1 Méthode *unsupervised* . . . 60
1.18 Montée de version de l'opérateur . . . 61
1.19 Exercices optionnels . . . 63
1.19.1 Mise en place d'un cluster Kubernetes (minikube) . . . 63
1.19.2 Mise en place de sauvegardes programmées . . . 63
1.19.3 Mettre en place une réplication synchrone . . . 63
1.19.4 Déploiement d'une application pgAdmin4 . . . 64
1.20 Travaux pratiques (solutions) . . . 66
1.21 Prise en main du cluster Kubernetes (minikube) . . . 67
1.22 Installation de l'opérateur CloudNativePG . . . 69
1.23 Déploiement d'instances PostgreSQL . . . 71
1.24 Éléments initiaux . . . 76
1.24.1 Modifications de paramètres de configuration . . . 79
1.24.2 Créer un rôle . . . 83
1.24.3 Créer une base de données . . . 85
1.25 Déploiement d'une instance secondaire . . . 87
1.25.1 Configuration par défaut . . . 87

1.25.2 Emplacement des instances . . . 90
1.26 Tests de bascules . . . 91
1.26.1 Bascule manuelle . . . 92
1.26.2 Bascule automatique . . . 92
1.27 Mise en place d'une sauvegarde PITR . . . 95
1.27.1 Configuration . . . 95
1.27.2 Sauvegarde complète de l'instance . . . 98
1.27.3 Générer de la donnée . . . 100
1.28 Procéder à une restauration PITR . . . 101
1.29 Montée de version mineure de PostgreSQL . . . 106
1.29.1 Méthode *unsupervised* . . . 106
1.29.2 Méthode *supervised* . . . 107
1.30 Montée de version de l'opérateur . . . 110
1.31 Exercices optionnels . . . 112
1.31.1 Mise en place d'un cluster Kubernetes (minikube) . . . 112
1.31.2 Mise en place de sauvegardes programmées . . . 114
1.31.3 Mettre en place une réplication synchrone . . . 116
1.31.4 Déploiement d'une application pgAdmin4 . . . 117

Les formations Dalibo **121**

Cursus des formations . . . 121
Les livres blancs . . . 122
Téléchargement gratuit . . . 122

Sur ce document

Formation	Module K1
Titre	Découverte de CloudNativePG
Révision	25.07
PDF	https://dali.bo/k1_pdf
EPUB	https://dali.bo/k1_epub
HTML	https://dali.bo/k1_html
Slides	https://dali.bo/k1_slides
TP	https://dali.bo/k1_tp
TP (solutions)	https://dali.bo/k1_solutions

Vous trouverez en ligne les différentes versions complètes de ce document.

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com[1] !

À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur https://dalibo.com/formations

[1] mailto:formation@dalibo.com

Remerciements

Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment :

Alexandre Anriot, Jean-Paul Argudo, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Ronan Dunklau, Vik Fearing, Stefan Fercot, Dimitri Fontaine, Pierre Giraud, Nicolas Gollet, Nizar Hamadi, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Jehan-Guillaume de Rorthais, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Arnaud de Vathaire, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées[3] par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

[3] https://www.postgresql.org/about/policies/trademarks/

1/ CloudNativePG

1.1 AU MENU

— L'opérateur CloudNativePG
— Fonctionnalités
— Fonctionnement

Différents sujets seront abordés pour présenter l'opérateur le mieux possible, que ce soit sur le projet CloudNativePG (historique, développement, CNCF…), sur ses fonctionnalités, ses mécanismes internes ou encore les images utilisées.

1.1.1 Objectifs

— Découverte de l'opérateur CloudNativePG
— Déploiement d'instances PostgreSQL via l'opérateur
— Tests et découvertes de fonctionnalités

L'objectif de ce module est la découverte de l'opérateur CloudNativePG mais également de sa prise en main. L'opérateur facilite le déploiement de clusters PostgreSQL dans Kubernetes. Une présentation générale de ce qu'est un opérateur et de son rôle sera faite pour bien poser le contexte.

Le TP permet d'installer l'opérateur CloudNativePG, de déployer un cluster PostgreSQL et d'effectuer différentes opérations comme la configuration d'une instance, la mise en place de sauvegardes ou de la restauration.

1.1.2 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

Pour rappel, un opérateur étend les possibilités d'un cluster Kubernetes grâce à la définition de `Custom Resource Definition`. Par exemple, l'opérateur CloudNativePG ajoute les ressources suivantes à l'API Kubernetes.

```
kubectl api-resources --api-group=postgresql.cnpg.io
NAME                 SHORTNAMES   APIVERSION             NAMESPACED   KIND
backups                           postgresql.cnpg.io/v1   true         Backup
clusterimagecatalogs              postgresql.cnpg.io/v1   false        ClusterImageCatalog
```

```
clusters                          postgresql.cnpg.io/v1   true         Cluster
databases                         postgresql.cnpg.io/v1   true         Database
imagecatalogs                     postgresql.cnpg.io/v1   true         ImageCatalog
poolers                           postgresql.cnpg.io/v1   true         Pooler
publications                      postgresql.cnpg.io/v1   true         Publication
scheduledbackups                  postgresql.cnpg.io/v1   true         ScheduledBackup
subscriptions                     postgresql.cnpg.io/v1   true         Subscription
```

Les opérateurs existants ont chacun leurs spécificités et particularités. Le site `operatorhub` [1] liste les principaux opérateurs qui existent.

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 le site Operator Framework [2] qui explique les différents niveaux existants.

[1] https://operatorhub.io/?keyword=postgresql
[2] https://sdk.operatorframework.io/docs/overview/operator-capabilities/

1.2 L'OPÉRATEUR CLOUDNATIVEPG

- https://cloudnative-pg.io
- Débuté par 2ndQuadrant puis EDB
- Libéré en 2022
- Projet Open Source, licence Apache 2.0
- Mode de gouvernance similaire au projet PostgreSQL
- Intégré à la Clou Native Computing Foundation (*Sandboxing*)
- Documentation https://cloudnative-pg.io/documentation/current/

Le projet a été initialement développé par la société 2ndQuadrant en 2019. La société a été rachetée par EnterpriseDB (EDB) en 2020. Le développement de cet opérateur a continué en interne avant que le projet ne soit rendu Open Source en 2022.

La gouvernance du projet se rapproche de celle de PostgreSQL avec une *core team* et l'ouverture à la contribution communautaire. L'intégration d'un nouveau mainteneur est soumise au vote des mainteneurs actuels du projet.

Le projet est bien engagé dans une démarche communautaire et Open Source avec notamment l'acceptation du projet au programme *Sandbox* de la Cloud Native Computing Foundation[3] en janvier 2025.

La documentation[4] du projet est particulièrement claire et fournie.

1.2.1 Ce que permet CloudNativePG

- Gestion déclarative
  - d'instances
  - de bases de données
  - `publication`
  - …
- Mise en place de réplication automatique
  - par *streaming replication*
- Archivage des journaux de transactions
- Sauvegardes PITR, sauvegardes planifiées
- Bascule automatique ou manuelle
- Haute disponibilité, hibernation, *fencing*, plugin `kubectl`…

D'une manière générale, si vous souhaitez déployer des instances PostgreSQL dans Kubernetes, nous

[3] https://www.cncf.io/
[4] https://cloudnative-pg.io/documentation/current/

vous recommandons d'utiliser un opérateur, quel qu'il soit. Les opérateurs sont spécialisés dans la gestion d'instances PostgreSQL. Nous déconseillons vivement le déploiement de conteneurs PostgreSQL sans opérateur, d'autant plus pour de la production.

Les fonctionnalités de CloudNativePG sont nombreuses. Il vous permet de gérer de manière déclarative un ensemble d'éléments PostgreSQL (bases, rôles, *tablespaces*…). Ils seront détaillés dans la suite du module.

Le déploiement d'instances, qu'elles soient primaires ou secondaires est très nettement facilité par l'opérateur : l'utilisateur de réplication est créé automatiquement, le déploiement d'un secondaire se fait en modifiant un seul champ dans la définition YAML, un slot de réplication est automatiquement créé pour protéger la réplication.

Les sauvegardes *PITR* sont supportées nativement et faites par l'outil `Barman Cloud` sur des stockages "cloud" (S3, GCP, Azure Blob).

La haute disponibilité est nativement supportée et gérée par l'intermédiaire des objets `Services` de Kubernetes et une utilisation astucieuse des `Labels` (un article[5] de blog a d'ailleurs été écrit à ce sujet.)

Toutes ces fonctionnalités sont très alléchantes, il n'en reste pas moins que de nombreux changements surviennent en déployant PostgreSQL sur une infrastructure conteneurisée comme Kubernetes. Un certain temps doit être alloué à l'étude de cette migration d'infrastructure tant les sujets impactés sont variés et importants : système de stockage, extensions PostgreSQL, images utilisées, accessibilité des instances, récupération des traces…

Des changements d'habitudes de travail auront nécessairement lieu et impliqueront également un accompagnement des équipes DBAs.

1.2.2 Les images utilisées

- Deux types d'images
  - `Pod` CloudNativePG
  - `ghcr.io/cloudnative-pg/cloudnative-pg:1.25.1`
  - `Pod` PostgreSQL
  - `ghcr.io/cloudnative-pg/postgresql:17.4`
- Images personnalisables

Il est nécessaire de distinguer deux types d'images. La première concerne l'opérateur CloudNativePG qui est une brique logicielle, développée en Go, et qui est mise à disposition sous forme d'image par l'équipe en charge du projet. Par défaut, l'image utilisée est `cloudnative-pg:1.25.0`. Elle se trouve sur la *registry* de Github (ghcr.io) associée au projet CloudNativePG.

[5] https://blog.dalibo.com/2025/01/17/cnpg-2.html

L'opérateur va se charger de déployer PostgreSQL pour nous. Ces déploiements se font à partir d'images également fournies par le projet. Elles se trouvent également sur cette même *registry* et leur nom est `postgresql:X.Y` ou `X` correspond à la version majeure de PostgreSQL et `Y` à la version mineure qui doit être déployée.

Cette image-là repose sur une autre image fournie quant à elle par la communauté Docker PostgreSQL[6]. Il est possible de voir cela dans le fichier Dockerfile[7] de l'image, avec la ligne `FROM postgres:%%POSTGRES_IMAGE_VERSION%%`. L'image qui contient PostgreSQL et utilisée par l'opérateur CloudNativePG est construite à partir d'une autre image.

Il vous est possible de créer vos propres images et de les utiliser avec CloudNativePG. Certains pré-requis sont nécessaires comme par exemple la présence dans le `PATH` des outils `initdb`, `pg_ctl` et d'exécutables Barman Cloud. Tous les pré-requis sont listés dans la documentation[8].

1.2.3 Versions supportées

- Versions PostgreSQL
  - 5 versions majeures supportées par an
- Versions de CloudNativePG
  - 2 versions majeures supportées par an
  - Uniquement des versions de PostgreSQL supportées
- Gérer les versions Kubernetes

Le *PostgreSQL Global Development Group* supporte chaque version majeure pendant une durée minimale de 5 ans. Par exemple, n'est plus supportée la version 12 depuis novembre 2024. Il n'y aura pour elle plus aucune mise à jour mineure, donc plus de correction de bug ou de faille de sécurité. Le support de la version 13 s'arrêtera en novembre 2025. Le support de la dernière version majeure, la 17, devrait durer jusqu'en 2029. Sur une année, il y a donc cinq versions de PostgreSQL simultanément supportées.

Cette politique de support de version est suivie par le projet CloudNativePG. L'opérateur permet de déployer uniquement des versions de PostgreSQL qui sont supportées par le *PGDG*.

En ce qui concerne les versions de l'opérateur, deux versions sont supportées simultanément. Actuellement ce sont les versions 1.25.x et 1.26.x. Chaque version est également supportée pour des versions spécifiques de Kubernetes.

La fréquence de montée de version sera donc plus élevée que d'habitude comme l'opérateur doit être mis à jour fréquemment. La durée de vie d'une version de CloudNativePG est de 6 mois.

N'hésitez pas à regarder la documentation à ce sujet : Supported releases[9].

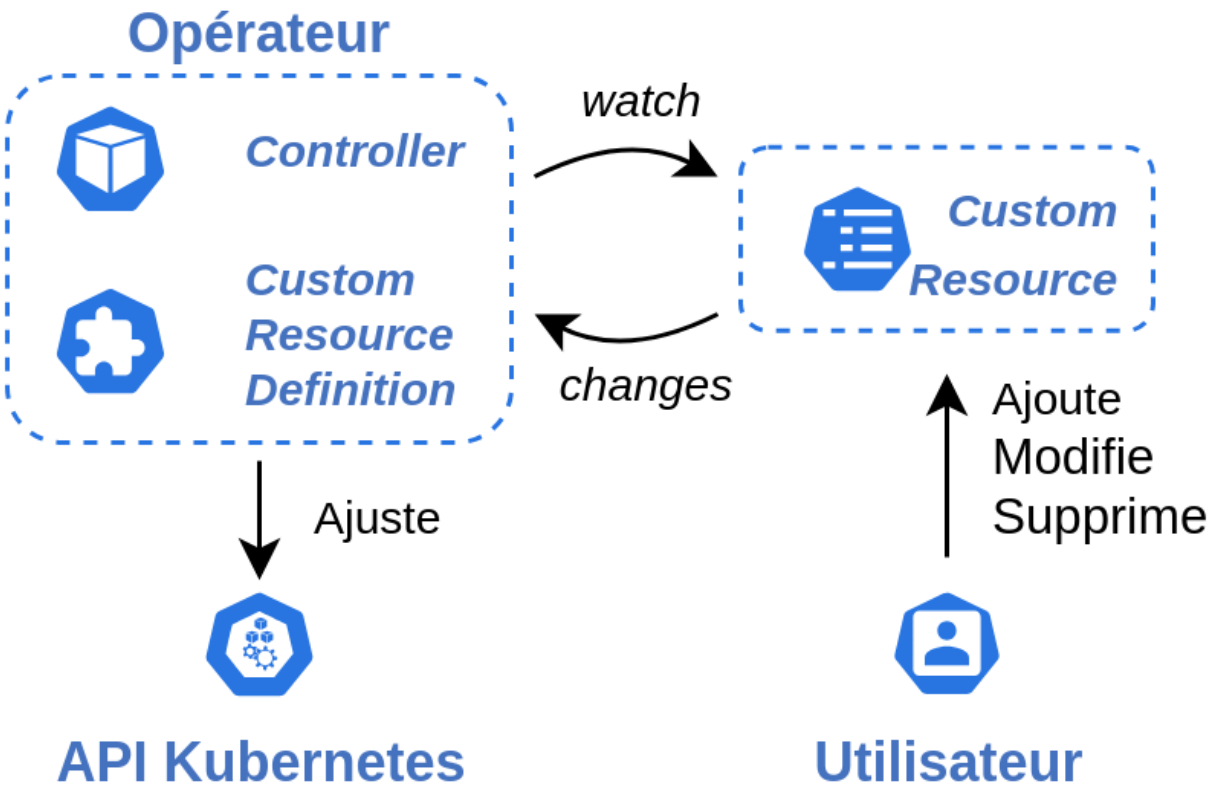
Il vous sera important d'avoir des créneaux de maintenance pour effectuer ces montées de versions.

[6] https://github.com/docker-library/postgres
[7] https://github.com/cloudnative-pg/postgres-containers/blob/main/Debian/Dockerfile.template
[8] https://cloudnative-pg.io/documentation/current/container_images/#
[9] https://cloudnative-pg.io/documentation/current/supported_releases/#

1.2.4 Installation de l'opérateur

- Deux éléments :
  - L'opérateur (code)
  - Les `Custom Resource Definitions` (extensions API)
- Installation :
  - *Manifest*
  - Helm Chart
  - OLM
- Un opérateur par cluster Kubernetes

1.2.5 Installation de l'opérateur



Un opérateur Kubernetes est, de manière simplifiée, composé de deux éléments :

- Un contrôleur ;
- Des `Custom Resource Definitions`.

Le premier élément n'est ni plus ni moins que le code de l'opérateur, son intelligence. Il embarque un ensemble de fonctions pour gérer convenablement PostgreSQL. L'opérateur CloudNativePG est écrit

en Go et se base sur le *framework* de développement de CLI Cobra[10]. Nous ne rentrerons pas dans le détail du développement de l'opérateur, mais sachez que pour chaque `Custom Resource` définie, il existe dans le code un *controller*. Cet élément est en charge de la gestion de chacune des ressources qui peuvent être gérées par l'opérateur (`Backup` , `Cluster` , etc).

Les `Custom Resource Definitions` contiennent la définition des nouvelles ressources Kubernetes qu'apporte l'opérateur. À partir du moment où les définitions sont déclarées dans le cluster Kubernetes, elles pourront être créées par un utilisateur. Ce sera alors l'opérateur qui saura les gérer (création, modification…).

L'installation de l'opérateur peut se faire de plusieurs manières différentes :

— soit en appliquant directement les fichiers YAML ;
— soit en utilisant le Helm Chart fourni par le projet ;
— soit via OLM (*Operator Lifecycle Manager*).

Les fichiers YAML se trouvent sur le `githubusercontent.com` du projet CloudNativePG. Pour la version 1.25.0 de l'opérateur, il est possible de les trouver ici : https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/release-1.25/releases/cnpg-1.25.0.yaml.

Pour installer l'opérateur sur un cluster Kubernetes auquel vous avez accès, rien de plus simple. Il suffit d'appeler `kubectl apply --server-side -f` suivi de l'URL.

```
kubectl apply --server-side \
-f  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-
 ↪  pg/main/releases/cnpg-1.25.0.yaml
```

Différentes ressources Kubernetes seront créées (`Namespace` , `ServiceAccount` , `Configmap` , `Deployment` etc). Toutes sont importantes évidemment, mais la `Deployment` nommée `cnpg-controller-manager` est la plus centrale puisqu'elle correspond concrètement à l'intelligence de l'opérateur CloudNativePG.

```
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
```

[10] https://cobra.dev/

```
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
```

L'autre possibilité est d'utiliser le *Helm Chart* proposé par le projet CloudNativePG (ou vos propres *Helm Chart* si vous êtes suffisamment à l'aise). Il est également disponible publiquement sur Github (voir https://github.com/cloudnative-pg/charts/tree/main/charts/cloudnative-pg). Il vous faudra avant tout ajouter le dépôt `cnpg` pour retrouver les `Charts` :

```
helm repo add cnpg https://cloudnative-pg.github.io/charts
```

Puis installer avec la commande `helm upgrade --install` :

```
helm upgrade --install cnpg \
  --namespace cnpg-system \
  --create-namespace \
  cnpg/cloudnative-pg
```

La dernière option, via OLM, nécessite l'installation du *manager* OLM dans votre cluster Kubernetes puis d'installer CloudNativePG via ce *manager*. Le choix de la méthode d'installation vous revient entièrement. Il doit être adapté à votre méthode de déploiement (à la main ? avec une chaîne CI/CD ?).

Quelque soit la manière dont est installé l'opérateur, vous ne pourrez installer qu'un seul opérateur par cluster Kubernetes.

1.2.6 Montée de version de l'opérateur

- L'opérateur et les `Custom Resource Definitions`
- L'`instance-manager`
- Redémarrage des instances
- Étalement des redémarrages dans le temps
  - `CLUSTERS_ROLLOUT_DELAY`
  - `INSTANCES_ROLLOUT_DELAY`

La mise à jour de l'opérateur se fait en plusieurs temps et impacte les instances PostgreSQL qu'il gère. Le principe de mise à jour est simple, il suffit d'appliquer les nouveaux manifestes de l'opérateur sur Kubernetes. Si vous avez installé l'opérateur avec *Helm*, mettez le à jour avec *Helm*. Même chose pour les autres méthodes de déploiement.

Ces nouveaux manifestes mettent à jour les `Custom Resource Definitions` (comme `Cluster`, `ScheduledBackup`, etc) et l'opérateur en tant que tel, c'est à dire le `Pod` qui contient le `controller`.

Cette première étape terminée, la seconde va être automatiquement déclenchée. Elle consiste à la mise à jour de tous les `Pods` des instances PostgreSQL déployées. En effet, il existe dans le `Pod` de

l'instance, un composant appelé `instance-manager`. Celui-ci est étroitement lié au `controller` CloudNativePG. Ils doivent être dans la même version. La mise à jour des instances suit le modèle de *Rolling Update* que nous verrons plus tard lorsque nous évoquerons la montée de version mineure d'une instance PostgreSQL.

Par défaut toutes les instances gérées par l'opérateur seront mises à jour en même temps. Ceci peut poser problème si vous avez de très nombreuses instances. Il est possible de répartir ces redémarrages dans le temps avec les paramètres :

— `CLUSTERS_ROLLOUT_DELAY` : permet de définir un délai entre le redémarrage de deux `Cluster` différents. Par défaut positionné à `0` ;
— `INSTANCES_ROLLOUT_DELAY` : permet de définir un délai entre le redémarrage de deux instances différentes qui font partie du même `Cluster`. Par défaut positionné à `0` ;

Ces paramètres là sont à définir dans le `Configmap` de configuration de l'opérateur, à savoir le `Configmap` `cnpg-controller-manager-config` qui doit être créé dans le même `Namespace` que l'opérateur.

Si il y a bien une chose à retenir, c'est qu'une mise à jour de l'opérateur déclenche la mise à jour d'un composant au sein des `Pods` PostgreSQL. Opération qui nécessite un redémarrage du `Pod`.

1.3 GESTION DÉCLARATIVE

1.3.1 Principe

— Fichiers YAML
— État désiré `<->` État actuel
— Boucle de réconciliation
— Nouvelles ressources créables dans Kubernetes

Le principe de la gestion déclarative est de décrire ce que l'on souhaite (une instance PostgreSQL, une sauvegarde, une base de données, etc) et de laisser le système, en l'occurrence Kubernetes et l'opérateur CloudNativePG, faire le travail de déploiement pour nous.

Leurs rôles est de faire en sorte que l'état d'une ressource corresponde toujours à l'état désiré (i.e défini dans le fichier YAML). C'est ce qui est appelé une **boucle de réconciliation**. L'opérateur suit les changements, voulus ou exceptionnels, qui ont lieu sur la ressource en question et réagira en conséquence.

Passons en revue quelques ressources qui sont désormais créables dans Kubernetes.

1.3.2 Cluster

— Définition minimale d'un `Cluster` PostgreSQL

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql
spec:
  instances: 1
  storage:
    size: 2Gi
  walStorage:
    size: 2Gi
```

Quelques lignes de YAML suffisent pour décrire un cluster PostgreSQL. Voici une explication des différents champs présents :

- `apiVersion` : la version de l'API de Kubernetes utilisée (ici celle apportée par l'opérateur CloudNativePG) ;
- `kind` : le type d'objet créé ; ici un cluster PostgreSQL (donc une ou plusieurs instances) ;
- `metadata` : des informations pour identifier l'objet, notamment son nom (`name`) ;
- `spec` : les spécifications de l'objet en question ;
  - `instances` : le nombre d'instances voulues (sera toujours un primaire, plus le reste en secondaire(s)) ;
  - `storage` : les informations sur le stockage souhaité pour `PGDATA` , par exemple la taille (`size`) des `Persistant Volumes` ;
  - `walStorage` : les informations sur le stockage souhaité pour les journaux de transactions.

Vous noterez qu'il n'est pas obligatoire d'indiquer quelle image sera utilisée. Par défaut, l'image utilisée pour ce `Pod` sera celle proposée par le projet CloudNativePG : `ghcr.io/cloudnative-pg/postgresql:X.Y` ou `X` représente le numéro de la dernière version majeure et `Y` le numéro de la dernière version mineure publiée par le projet PostgreSQL.

Une bonne pratique dans PostgreSQL souvent recommandée mais peu appliquée est d'avoir au moins deux espaces de stockage bien distincts :

- un pour les données ;
- un pour les journaux de transactions (WAL).

CloudNativePG respecte cette bonne pratique avec le paramètre `spec.walStorage` . Lorsque le cluster est créé, deux `Persistant Volumes` distincts sont créés automatiquement, un pour `PGDATA` et un pour les journaux de transactions (WAL). Si ce paramètre n'est pas utilisé, les journaux se trouveront dans le même volume que `PGDATA` , ce que nous ne recommandons pas.

De nombreux paramètres supplémentaires existent pour configurer nos instances PostgreSQL. Des subtilités existent aussi dans le déploiement du `Pod` , notamment sur le fait qu'un `init-container` est déployé avant le conteneur PostgreSQL. Tout ceci sera détaillé plus tard dans ce module.

1.3.3 Éléments initiaux

- PostgreSQL
  - Base de données `app`
  - Rôles `app` , `streaming_replica`
  - Règles `pg_hba`
- Kubernetes
  - `Secrets` : postgresql-app
  - `Services` : postgresql-r, postgresql-ro, postgresql-rw

Lorsque vous déployez une instance, plusieurs éléments sont automatiquement créés par l'opérateur, que ce soit dans l'instance avec la création de rôles ou d'une base de données, ou dans Kubernetes comme des `Secrets` ou des `Services`.

L'instance déployée est partiellement configurée pour fonctionner dès sa création. Plusieurs paramètres PostgreSQL sont déjà configurés, le fichier `pg_hba.conf` est en partie renseigné, des certificats sont générés, etc.

Deux nouveaux rôles existent : `app` et `streaming_replica`. Le premier est un rôle basique avec le droit de connexion. Le deuxième est un rôle utilisé par les instances secondaires lors de la mise en place de la réplication physique. Le rôle `postgres` existe lui aussi mais n'est pas créé par CloudNativePG.

Une base de données nommée `app` est d'office créée avec la configuration suivante :

- Name : `app`
- Owner : `app`
- Encoding : `UTF8`
- Locale Provider : `libc`
- Collate : `C`
- Locale Provider : `C`

Du côté de Kubernetes aussi, des ressources sont automatiquement créées. Certaines nous concernent directement en tant qu'administrateur PostgreSQL. Il y a notamment un `Secret` qui est créé et qui contient le mot de passe du rôle PostgreSQL `app`. D'autres objets comme des `Services` sont utilisés pour la connectivité de l'instance, ou des instances si des réplications existent.

Pour chaque cluster PostgreSQL déployé, trois services dédiés sont créés. Par exemple, pour le `Cluster` nommé `postgresql` :

- un service qui permet d'accéder au primaire : `postgresql-rw` qui est en lecture/écriture ;
- un service qui permet d'accéder uniquement aux secondaires : `postgresql-ro` qui sont en lecture seule ;
- un service qui permet d'accéder à toutes les instances : `postgresql-r`.

Nous verrons lorsque le sujet de la haute disponibilité sera abordé à quoi ces `Services` peuvent servir.

1.3.4 Modification des éléments initiaux

- Modification possible de certains éléments
- `spec.bootstrap.initdb`
- La commande `initdb` est utilisée

```
bootstrap:
  initdb:
    database: mabase
    owner: monrole
```

Les éléments déployés par défaut par CloudNativePG peuvent être modifiés si il ne vous conviennent pas. Cependant, vous devez le faire à l'initialisation de l'instance. Cela ne sera plus possible après coup.

La section `bootstrap` correspond à la manière dont est initiée l'instance. Par défaut c'est la méthode `initdb` qui est utilisée, mais nous verrons que d'autres méthodes peuvent être utlisées, notamment pour créer une instance à partir d'une sauvegarde.

Les changements que vous voulez apporter doivent être inscrits dans la section `bootstrap.initdb`. Dans l'exemple suivant, le nom de la base automatiquement créée et le nom du rôle sont modifiés par `mabase` et `monrole`.

```
bootstrap:
  initdb:
    database: mabase
    owner: monrole
```

De nombreuses autres options peuvent être passées à `initdb`, comme les plus notables :

- `dataChecksums` : pour activer les sommes de contrôle sur l'instance (`false` par défaut) ;
- `encoding` : pour choisir l'encodage des caractères (`UTF8` par défaut) ;
- `walSegmentSize` : si voulez changer la taille par défaut (16 Mo) des WALs.

1.3.5 Database

— v1.25.0+
— Définition minimale d'une `Database`

```
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: mabase
spec:
  name: mabase
  owner: monrole
  cluster:
    name: postgresql
```

Vous pouvez créer des ressources `Database` depuis la version 1.25.0 de l'opérateur. L'exemple ci-dessus permet de créer la base `mabase` dans l'instance PostgreSQL référencée par le paramètre `spec.cluster.name`. Le rôle `monrole` sera le propriétaire de cette base. Il doit exister dans l'instance pour que la création de cette ressource se fasse correctement. Autrement, un message d'erreur apparaîtra dans la description de la ressource. Par exemple :

```
kubectl get databases.postgresql.cnpg.io -o=custom-columns=MESSAGE:.message
```

```
MESSAGE
while creating database "mabase": ERROR: role "monrole" does not exist (SQLSTATE
↪  42704)
```

Il existe là aussi de nombreux paramètres pour configurer une ressource `Database`. En ce qui concerne l'exemple ci-dessus :

— `apiVersion` : la version de l'API de Kubernetes est utilisée (ici celle apportée par l'opérateur CloudNativePG) ;
— `kind` : le type d'objet créé, ici une base de données ;
— `metadata` : des informations pour identifier l'objet, notamment son nom (`name`) ;
— `spec` : les spécifications de l'objet en question ;
  — `name` : le nom de la base de données ;
  — `owner` : le nom du rôle PostgreSQL qui sera le propriétaire de la base de données, qui doit exister ;
  — `cluster` : le nom du cluster PostgreSQL dans laquelle doit être créée cette base de données.

Vous pouvez trouver de manière détaillée les autres paramètres sur cette page[11]. Voici d'autres paramètres de la section `spec` qui nous semblent intéressants :

— `encoding` : permet d'indiquer l'encodage de la base (UTF8, LATIN1…) ;

[11] https://cloudnative-pg.io/documentation/current/cloudnative-pg.v1/#postgresql-cnpg-io-v1-DatabaseSpec

- `template` : correspond au nom du modèle de base qui doit être utilisé pour la base créée ;
- `isTemplate` : permet d'indiquer si la base créée est un modèle ou pas ;
- `allowConnections` : permet d'indiquer s'il sera possible de se connecter à cette base ;
- `connectionLimit` : correspond au nombre de connexions simultanées autorisées à cette base ;
- `tablespace` : correspond au `TABLESPACE` où sera créée la base de données.

Ces paramètres ne sont ni plus ni moins que les options de la commande `CREATE DATABASE` de PostgreSQL.

Le renommage d'une base de données n'est pas possible.

1.3.6 Role

- Pas de `Custom Resource Definition`
- Déclaré dans la ressource `Cluster`
- `spec.managed.roles`

```yaml
spec: # Cluster
[…]
  managed:
    roles:
    - name: dalibo
      ensure: present
      comment: Support Account
      login: true
      superuser: true
      passwordSecret:
        name: dalibo-password
```

Les rôles PostgreSQL sont définis directement dans l'objet `Cluster`, dans la section `spec.managed.roles`. Il n'existe pas à l'heure actuelle de `Custom Resource` rôles.

- `roles` : est la liste des rôles qui doivent être présents dans l'instance ;
  - `name` : est évidemment le nom du rôle. N'oubliez pas le `-` en début de ligne indiquant qu'il s'agit d'un nouvel élément de la liste ;
  - `ensure` : positionné à `present` (valeur par défaut), l'opérateur vérifiera si le rôle existe et si ce n'est pas le cas, le créera. Le comportement est l'inverse s'il est positionné à `absent` ;
  - `comment` : simple champ commentaire ajouté au rôle si renseigné ;
  - `login` : indique si le rôle peut se connecter (`true`, valeur par défaut), `false` sinon ;
  - `superuser` : indique si le rôle est un super-utilisateur, `false` (valeur par défaut) ;
  - `passwordSecret.name` : indique le nom du `Secret` Kubernetes ou se trouve le mot de passe du rôle.

Vous pouvez trouver de manière détaillée les autres paramètres sur cette page[12]. Voici d'autres paramètres de la section `spec.managed.roles` qui nous semblent intéressants :

- `validUntil` : la date à partir de laquelle le rôle ne sera plus valide (exemple `validUntil: "2025-06-17T15:0`) ;
- `inRoles` : la liste des rôles auxquels le rôle créé doit appartenir ;
- `replication` : indique si le rôle doit avoir le rôle `REPLICATION`, par défaut à `false` par défaut.

Il est possible d'indiquer le mot de passe que doit avoir un nouveau rôle. Pour cela, il faut utiliser le paramètre `passwordSecret.name` qui doit contenir le nom d'un objet `Secret` dans le cluster Kubernetes. Il s'agit donc d'un pré-requis à la création d'un rôle avec mot de passe.

Dans l'exemple de la *slide*, le `Secret` `dalibo-password` doit être créé en amont.

```
apiVersion: v1
kind: Secret
type: kubernetes.io/basic-auth
metadata:
  name: dalibo-password
  labels:
    cnpg.io/reload: "true"
data:
  username: ZGFsaWJv
  password: Q0hBTkdFTUU=
```

Les champs `username` et `password` doivent contenir les valeurs encodées en BASE64. Cela peut se faire en ligne de commande avec `printf` et `base64` :

```
printf "dalibo" | base64
ZGFsaWJv
printf "CHANGEME" | base64
Q0hBTkdFTUU=
```

[12] https://cloudnative-pg.io/documentation/current/cloudnative-pg.v1/#postgresql-cnpg-io-v1-RoleConfiguration

1.3.7 Tablespace

— Pas de `Custom Resource Definition`
— Déclaré dans la ressource `Cluster`
— `spec.tablespaces`

```
spec: # Cluster
[…]
  tablespaces:
    - name: data
      storage:
        size: 1Gi
      owner: dalibo
    - name: fast
      storage:
        size: 2Gi
        storageClass: fast
      owner: dalibo
```

À l'image des rôles, les tablespaces sont également déclarés dans la ressource `Cluster`. Il est possible de renseigner une liste de tablespaces et de configurer chacun d'eux de manière spécifique.

— `tablespaces` : est la liste des tablespaces qui doivent être créés ;
  — `name` : est évidemment le nom du tablespace. N'oubliez pas le `-` en début de ligne qui indique qu'il s'agit d'un nouvel élément de la liste ;
  — `storage` : contient la configuration au niveau du stockage du tablespace ;
    — `size` : la taille du volume où sera créé le tablespace ;
    — `storageClass` : indique quelle classe de stockage doit être utilisée pour ce volume-là ;
  — `owner` : indique le nom du propriétaire de ce tablespace ;
  — `temporary` : permet d'indiquer si le tablespace doit être créé avec la clause `TEMPORARY`.

Si le propriétaire du tablespace n'est pas renseigné, l'utilisateur `app` le sera par défaut. Par défaut aussi, le tablespace est créé avec le paramètre `temporary` à `false`.

L'option `storageClass` est particulièrement intéressante pour configurer la classe de stockage sous-jacente au tablespace et donc, in fine, au volume. Cela vous permet d'utiliser des classes de stockage plus ou moins rapides selon vos besoins.

Lorsque le `Cluster` est créé, ou modifié, l'opérateur se charge de créer les objets `Persistant Volumes` / `Persistant Volume Claims` nécessaires et les associe au `Pod` de l'instance. Par exemple, lors d'un premier déploiement sans tablespace supplémentaire nous sommes dans la situation suivante avec un seul `PVC` et un seul `PV`.

```
kubectl get pvc
NAME           STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
postgresql-1   Bound    pvc-b0eb7368-0795-48ea-89be-88475dc2a486   2Gi        RWO            standard       <unset>                 3m8s
```

```
kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                 STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   A
pvc-67dda23b-5b3c-4e15-950d-681db723d62f    1Gi          RWO              Delete           Bound    default/postgresql-1-tbs-
data   standard       <unset>                          3m10s
```

Après l'ajout des deux tablespaces, la situation est la suivante, avec trois `PVC` et trois `PV`. Le premier couple `PV` / `PVC` correspond au tablespace par défaut.

```
kubectl get pvc
NAME                     STATUS   VOLUME                                     CAPACITY
 ↪  ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
postgresql-1             Bound    pvc-b0eb7368-0795-48ea-89be-88475dc2a486   2Gi
 ↪  RWO            standard       <unset>                 6m41s
postgresql-1-tbs-data    Bound    pvc-67dda23b-5b3c-4e15-950d-681db723d62f   1Gi
 ↪  RWO            standard       <unset>                 4m4s
postgresql-1-tbs-fast    Bound    pvc-934a3ab9-123e-481c-af44-d3b741961859   2Gi
 ↪  RWO            standard       <unset>                 4m4s

kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY
 ↪  STATUS   CLAIM                             STORAGECLASS   VOLUMEATTRIBUTESCLASS
 ↪  REASON   AGE
pvc-67dda23b-5b3c-4e15-950d-681db723d62f   1Gi        RWO            Delete
 ↪  Bound    default/postgresql-1-tbs-data     standard       <unset>
 ↪  3m59s
pvc-934a3ab9-123e-481c-af44-d3b741961859   2Gi        RWO            Delete
 ↪  Bound    default/postgresql-1-tbs-fast     standard       <unset>
 ↪  3m59s
pvc-b0eb7368-0795-48ea-89be-88475dc2a486   2Gi        RWO            Delete
 ↪  Bound    default/postgresql-1              standard       <unset>
 ↪  6m46s
```

Lors de l'ajout d'un ou de plusieurs tablespaces dans un `Cluster` en fonctionnement, le `Pod` PostgreSQL est redémarré, générant ainsi une interruption de service.

1.4 CONFIGURATION DE L'INSTANCE

- Déclaratif, tout se fait en YAML
- Accès direct à ces fichiers interdit !
  - `postgresql.conf`
  - `pg_hba.conf`
  - `pg_ident.conf`

Comme vous le savez, installer une instance PostgreSQL ne suffit pas. Il faut en plus la configurer. Généralement, la configuration se fait dans le fichier `postgresql.conf` et nécessite soit un rechargement, soit un redémarrage de l'instance selon le paramètre modifié.

Avec l'utilisation de CloudNativePG, il n'est plus possible de modifier directement le fichier de configuration. Tout se fait dans la définition YAML de l'instance. Voyons quels sont les changements auxquels s'attendre.

1.4.1 postgresql.conf

- Tous les paramètres ne sont pas modifiables
- `ALTER SYSTEM` désactivé
  - `allow_alter_system` à `false`
- Des vérifications sont mises en place
- Redémarrage ou rechargement automatique

Voici par exemple comment passer le paramètre `shared_buffers` à `1GB`, et activer la compression des journaux de transactions.

```
[…]
spec:
  postgresql:
    parameters:
      shared_buffers: "1GB"
      wal_compression: "on"
[…]
```

La configuration d'une instance se fait dans la section `.spec.postgresql.parameters` de la définition YAML du `Cluster`. Les noms des paramètres sont les mêmes que ceux de PostgreSQL. CloudNativePG ne les renomme pas d'une manière ou d'une autre.

Un bon nombre de paramètres PostgreSQL ne sont pas modifiables dans la section `.spec.postgresql.parameters` de l'instance. La liste est disponible dans la documentation[13].

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

Cette interdiction peut paraitre surprenante, d'autant plus qu'avec un logiciel open source et libre comme l'est PostgreSQL, nous nous attendons plutôt à être libres de nos mouvements. Le parti pris par les développeurs est que tous ces paramètres sont liés à des fonctionnalités dont l'opérateur a la charge (sauvegarde, réplication, archivage des journaux, gestion des traces, etc).

Certains d'entre eux, comme `allow_alter_system`, sont modifiables par l'utilisation d'éléments présents dans d'autres sections de configuration, comme par exemple `enableAlterSystem`, qui se trouve dans la section `.spec.postgresql`.

Des vérifications sont faites sur certains. Pour reprendre l'exemple de `shared_buffers`, si vous renseignez une valeur plus grande que `resources.requests.memory` (allouée au `Pod`), vous lèverez une alerte, et votre modification ne sera pas appliquée.

Exemple de message d'erreur remonté :

```
The Cluster "postgresql" is invalid: spec.resources.requests.memory:
Invalid value: "512Mi": Memory request is lower than PostgreSQL `shared_buffers`
↪  value
```

Vous n'êtes pas sans savoir que la modification de certains paramètres nécessite soit un rechargement de la configuration, soit un redémarrage de l'instance.

Par défaut l'opération de rechargement ou de redémarrage est déclenchée automatiquement lorsque le nouveau paramètre est appliqué. Concrètement, si vous modifiez `max_connections`, vos instances seront automatiquement redémarrées. Il est possible de configurer un délai avant que le redémarrage ne se fasse.

Ce dernier point sera détaillé lorsque l'on traitera de la stratégie de mise à jour que vous voulez mettre en place.

1.4.2 pg_hba.conf et pg_ident.conf

- Pré-configurés
  - `FIXED RULES`, `DEFAULT-RULES`
- Dans la définition du `Cluster`
  - `USER-DEFINED RULES`

Il existe trois sections dans le fichier `pg_hba.conf` :

- `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 par les administrateurs ;
- `DEFAULT RULES` : qui autorise par défaut toutes les connexions par mot de passe à toutes les bases de données.

Il existe deux sections dans le fichier `pg_ident.conf` :

— `FIXED RULES` : qui sont des règles fixées par l'opérateur. On retrouve une ligne concernant l'association de l'utilisateur système `postgres` au rôle `postgres` ;
— `USER-DEFINED RULES` : qui correspond aux règles qui seront créées par les administrateurs.

1.5 ADMINISTRATION DE L'INSTANCE

1.5.1 Administration de l'instance

- Plugin kubectl
- Connexion
- Traces
- Réplication
- Archivage
- Sauvegarde
- Restauration
- Montée de version mineure de PostgreSQL
- Haute disponibilité et bascule
- Hibernation et fencing

1.5.2 Plugin kubectl

- `kubectl cnpg`
- Écrit en Go
- Interaction avec l'opérateur ou l'instance
- Commandes :
  - `status`
  - `psql`
  - `promote`
  - `backup`
  - `logs`
  - `reload`
  - `restart`
  - …

Un plugin CloudNativePG (cnpg) existe pour l'outil `kubectl`. Il permet d'obtenir très simplement un ensemble d'informations sur un cluster PostgreSQL ou de se connecter à une instance, déclencher une sauvegarde, promouvoir un secondaire en primaire, etc. Il est écrit en Go et se base, comme le code de l'opérateur, sur le *framework* Cobra[14].

Il existe plusieurs méthodes d'installation : par script, par paquets `.rpm` ou `.deb` ou encore via `krew` ou `homebrew`. À vous de choisir ce qui convient le mieux à vos utilisateurs et administrateurs, qu'ils soient sur Linux, MacOS ou Windows.

[14] https://github.com/spf13/cobra

La liste des fonctionnalités offertes est assez longue. Elles couvrent des thèmes très variés allant de la simple connexion `psql` à une instance, à la création de publication pour des réplications logiques ou encore le déclenchement de tests de charges avec `fio` ou `pgbench`. Voici quelques unes des commandes qui paraissent essentielles à connaitre dans un premier temps :

- `kubectl cnpg status [cluster]` : génère un résumé sur l'état du cluster PostgreSQL (nombre d'instances, sauvegardes, secondaires, réplications…) ;
- `kubectl cnpg psql [cluster]` : permet de se connecter avec `psql` à l'instance primaire ;
- `kubectl cnpg logs [cluster]` : affiche les traces PostgreSQL. La commande `pretty` permet d'afficher les traces de manière plus lisible. L'outil `jq` peut également s'avérer utile ;
- `kubectl cnpg reload [cluster]` : déclenche une boucle de réconciliation pour prendre en compte les modifications apportées au cluster PostgreSQL ;
- `kubectl cnpg restart [cluster] [node]` : redémarre soit le cluster en entier, soit une seule instance si [node] est renseigné ;
- `kubectl cnpg promote [cluster] [node]` : promeut l'instance indiquée comme nouvelle primaire ;
- `kubectl cnpg backup [cluster]` : déclenche une sauvegarde physique de l'instance mentionnée. La configuration de la sauvegarde doit être faite dans la définition du cluster.

1.5.3 Connexion

- Plus d'accès au serveur
- Comme avec le PGaaS
- Accessibilité interne ou externe
- Outils
  - `kubectl` ou
  - `psql`
- DNS

En embarquant PostgreSQL dans un conteneur et dans Kubernetes, il ne vous sera plus possible d'accéder au serveur sur lequel est installé PostgreSQL. Cela vous demandera davantage de configuration et de connaissances (notamment sur les différentes couches qui existent dans Kubernetes).

Vous n'aurez plus accès au serveur sous-jacent, comme ce serait le cas avec une solution de PGaaS. Si vous gérez vous même votre cluster Kubernetes, il vous sera évidemment possible d'accéder aux nœuds *Workers*.

Il faut différencier deux types d'accès à une instance PostgreSQL :

1. les accès inités depuis un client déployé dans le cluster Kubernetes (interne) ;
2. et ceux initiés depuis un client en dehors du cluster Kubernetes (externe).

Dans le premier cas, l'application pourra accéder à l'instance PostgreSQL si les `Network Policies` l'autorisent. Dans le second cas, vos administrateurs Kubernetes devront mettre en place un *Load Balancer* en frontal du cluster pour autoriser les accès externes. Dès lors que votre instance est accessible depuis l'extérieur (interface et port exposés), vous pourrez vous y connecter avec `psql` par exemple. Dans le cas contraire, vous ne pourrez pas vous y connecter directement.

L'outil `kubectl` permet à des utilisateurs de se connecter à une instance spécifique. La commande `kubectl exec` permet d'exécuter une commande dans un conteneur. Par « chance », l'image utilisée pour déployer PostgreSQL contient `psql`. Dès lors que vous avez accès au cluster Kubernetes et que vous avez les bons droits pour le faire, `kubectl exec -it POD CONTAINER -- psql` vous permet de vous connecter à l'instance avec le rôle `postgres`.

```
kubectl exec -it postgresql-1 -c postgres -- psql
psql (17.2 (Debian 17.2-1.pgdg110+1))
Type "help" for help.

postgres=#
```

Le plugin `cnpg` permet la même chose avec sa commande `psql` :

```
kubectl cnpg psql postgresql
psql (17.2 (Debian 17.2-1.pgdg110+1))
Type "help" for help.

postgres=#
```

Au sein du cluster Kubernetes, vous pouvez utile le DNS attribué au `Pod` ou au `Service` pour vous connecter à une instance.

1.5.4 Traces

- Format `JSON`
- Un seul flux pour différents *logger*, pas que PostgreSQL
- Sortie standard du `Pod`
- Certains paramètres `log_*` non modifiables
- Outil de centralisation de traces obligatoire

Dès lors que vous déploierez PostgreSQL avec CloudNativePG, les traces que vous connaissez ne seront ni accessibles dans le fichier `postgresql.log` ni du même format.

Concernant les traces de l'opérateur, vous pouvez gérer le niveau de celles-ci avec l'argument `--log-level` du `Deployment` de l'opérateur. Les valeurs `error`, `warning`, `info`, `debug` et `trace` sont disponibles. La valeur par défaut est `info`.

Concernant le(s) `Pod(s)` PostgreSQL, les traces contiennent les traces de l'instance, mais également les traces d'autres éléments comme celles de l'`instance manager` ou encore de la solution de sau-

vegarde. Toutes les traces sont renvoyées sur la sortie standard du `Pod` au format `JSON` et sont mélangées. La clé `logger` de la trace `JSON` indique qui est responsable de cette ligne. Par exemple, la trace suivante a été générée par l'instance. On peut le voir avec le paramètre `logger` qui est à `postgres`.

```
{
  "level": "info",
  "ts": 1619781249.7188137,
  "logger": "postgres",
  "msg": "record",
  "record": {
    "log_time": "2021-04-30 11:14:09.718 UTC",
    "user_name": "",
  […]
  }
```

Concernant les paramètres de traces PostgreSQL, vous ne pouvez pas modifier les paramètres PostgreSQL suivants. CloudNativePG l'interdit.

```
log_destination
log_directory
log_file_mode
log_filename
log_rotation_age
log_rotation_size
log_truncate_on_rotation
logging_collector
```

Il est possible de suivre les traces d'un `Pod` en ligne de commande avec la commande `kubectl logs -f <cluster>` ou `kubectl cnpg logs cluster <cluster>` si vous avez installé le plugin `cnpg` pour `kubectl`.

Les traces ne sont pas persistées dans le `Pod`. Il est donc essentiel d'avoir une solution de centralisation des traces pour que vos administrateurs puissent y avoir accès. Des outils comme `Loki`[15], `Fluentd`[16].

1.5.5 Réplication

- Mise en place très simple
  - `instances: N`
  - `1` primaire et `N-1` secondaires
- *Streaming Replication*
- Répartition sur les nœuds
  - Configuration de l'affinité/anti-affinité
- Slot de réplication
- Asynchrone ou synchrone

[15] https://grafana.com/oss/loki/
[16] https://docs.fluentd.org/0.12

Le déploiement d'instances secondaires est très grandement facilité par CloudNativePG. En modifiant uniquement le nombre d'instances dans l'objet `Cluster`, une ou plusieurs nouvelles instances sera créée et configurée pour suivre le primaire grâce à la réplication physique (*Streaming Replication*). Un nouveau `Pod` sera donc créé, reprenant le nom du `Cluster` suivi d'un chiffre incrémenté de 1 pour chaque nouvelle instance. Avec la sortie de la commande `kubectl get pod` suivante, nous pouvons comprendre qu'au sein du cluster Kubernetes, deux `Pods` PostgreSQL existent et appartiennent au même `Cluster` (au sens de CloudNativePG) nommé `postgresql`.

```
kubectl get pod
NAME                  READY   STATUS    RESTARTS   AGE
postgresql-1          1/1     Running   0          14h
postgresql-2          1/1     Running   0          7h
```

Le chiffre qui suit le nom du cluster, ici `1` et `2`, n'indique PAS le rôle de l'instance (primaire ou secondaire). Se baser sur ce chiffre pour connaitre le rôle d'une instance est une erreur.

Il existe plusieurs méthodes pour retrouver l'instance primaire d'un cluster. Par exemple :

```
kubectl get cluster postgresql
NAME         AGE   INSTANCES   READY   STATUS                     PRIMARY
postgresql   14h   2           2       Cluster in healthy state   postgresql-1
```

L'opérateur et la configuration de base font en sorte que les instances secondaires soient réparties, si cela est possible, sur les différents *workers* qui composent le cluster Kubernetes. L'idée est de ne pas déployer au même endroit toutes les instances, auquel cas, en cas de panne, l'intégralité du `Cluster` PostgreSQL serait perdu.

Un slot de réplication sera automatiquement créé pour chaque nouveau secondaire. Le principal avantage est de ne plus avoir de décrochage du secondaire en conservant les journaux de transactions nécessaires. La conséquence directe est le risque d'accumulation de ces mêmes journaux qui pourraient saturer l'espace disque du primaire. Le nom du slot de réplication est automatiquement généré avec `cnpg` et le nom de l'instance. Voici un extrait de la vue `pg_stat_replication_slot` qui montre cela.

```
-[ RECORD 1 ]-------+-------------------
slot_name           | _cnpg_postgresql_2
plugin              |
slot_type           | physical
[…]
```

Vous trouverez davantage d'informations sur le concept de slot de réplication dans notre module W2B[17].

Par défaut, la réplication mise en place est asynchrone. Il est possible de mettre en place de la réplication synchrone.

[17] https://dali.bo/w2b_html

1.5.6 Archivage

- Activé par défaut (`archive_mode` à `on`)
  - Non modifiable
- `archive_command` : `/controller/manager wal-archive …`
  - Non modifiable
- Utilisé pour les sauvegardes *PITR*
- Barman Cloud, pour le moment

L'archivage des journaux de transaction est fortement conseillé lorsque qu'il est question de sauvegarde physique et permet notamment la mise en place de sauvegardes dites PITR (voir notre module I2[18]).

CloudNativePG active par défaut l'archivage des journaux et impose également la commande exécutée. L'archivage se fera forcément par l'intermédiaire de l'`instance manager`. Ce `manager` appelera *in fine* la commande d'archivage de Barman Cloud.

L'archive ne peut se faire que sur un stockage de type objet (type S3, Google Cloud Storage, Azure Blob Storage ou MinIO). Cet archivage est effectif dès lors que la section `barmanObjectStore` est définie dans la définition du `Cluster`. Voici un exemple tiré de la documentation officielle.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[…]
  backup:
    barmanObjectStore:
      destinationPath: "s3://<bucket>/<folder>/"
      endpointURL: "https://s3.<region>.scw.cloud"
      s3Credentials:
        accessKeyId:
          name: scaleway-api-secret
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: scaleway-api-secret
          key: ACCESS_SECRET_KEY
        region:
          name: scaleway-api-secret
          key: ACCESS_REGION
```

D'un fournisseur de stockage à un autre, les champs `destinationPath` et `endpointURL` peuvent changer. En plus de la connaissance du point d'entrée de votre solution de stockage, vous devez renseigner les informations de connexion dans la section `s3Credentials`. Ces informations là doivent être enregistrées dans un objet `Secret` (objet Kubernetes). Dans l'exemple, le nom de ce `Secret` est `scaleway`. Voici ce à quoi il pourrait ressembler.

```
apiVersion: v1
```

[18] https://dali.bo/i2_html

```
kind: Secret
metadata:
  name: scaleway-api-secret
type: Opaque
data:
  ACCESS_KEY_ID: bWEgY2xlIGQgYWNjZXMK
  ACCESS_REGION: ZnItcGFy
  ACCESS_SECRET_KEY: bW9uIHNlY3JldCBiaWVuIGdhcmRlCg==
```

La valeur de chacun des trois champs *data* doit être encodée en Base64.

1.5.7 Sauvegarde

- Sauvegarde physique uniquement
- 2 méthodes
  - Stockage objets
  - `Volume Snapshot`
- Objets
  - `Backup`
  - `ScheduledBackup`
- Configuration `spec.backup` au niveau du `Cluster`

L'opérateur sait gérer pour vous des sauvegardes dites physiques. C'est le seul type de sauvegarde que vous allez pouvoir déclencher via l'opérateur. Autrement dit, il existe une nouvelle ressource Kubernetes, appelée `Backup` qui correspond à une sauvegarde physique d'un `Cluster` PostgreSQL.

Les sauvegardes logiques (faites avec `pg_dump` ou `pg_dumpall` par exemple) pourront toujours se faire avec ces outils dès lors que votre instance est accessible.

Plusieurs méthodes de sauvegarde existent. Quelle que soit la méthode, la configuration se fait dans l'objet `Cluster` correspondant à vos instances. Certains paramètres peuvent également être renseignés dans les objets `Backup` ou `ScheduledBackup`. C'est ce que nous allons découvrir par la suite.

Aussi, vous pourrez effectuer ces sauvegardes à partir des instances secondaires pour décharger les instances primaires, et ce, quelle que soit la méthode choisie.

1.5.8 Sauvegarde sur stockage objets

- Méthode par défaut
- `spec.backup.barmanObjectStore`
- Barman Cloud
- Sauvegarde à chaud, *PITR*

La première méthode consiste à effectuer les sauvegardes sur un stockage objet de type S3, Azure Blob Storage, Google Cloud Storage ou MinIO. Cette méthode se repose à l'heure actuelle sur l'outil Barman Cloud. C'est le même fonctionnement que pour l'archivage.

Couplé avec l'archivage des journaux de transactions, vous obtenez une sauvegarde *PITR* fonctionnelle pour votre instance.

1.5.9 Sauvegarde par Volume Snapshot

- Fonctionnalité Kubernetes (API)
- `spec.backup.volumeSnapshot`
- Dépend de la `StorageClass` et du `Container Storage Interface`
- Sauvegarde à chaud ou à froid

La seconde méthode utilise quant à elle un mécanisme propre à l'API de Kubernetes, le `Volume Snapshot`. C'est une fonctionnalité qui doit être supportée par la `Storage Class` (et donc *in fine* par le `CSI`) avec laquelle les volumes de votre instance ont été créés.

Cette méthode va créer un objet `Volume Snapshot` dans votre cluster Kubernetes qui contiendra un instantané du `Persistent Volume` ciblé. Cette sauvegarde sera donc locale à votre système de stockage et non plus envoyée sur un stockage objet. Un *snapshot* sera créé pour chaque volume de votre instance (`storage` et `walStorage`).

Des mécanismes plus complexes comme les sauvegardes incrémentales ou différentielles sont possibles si la `Storage Class` le permet.

Couplé avec l'archivage des journaux de transactions, vous obtenez une sauvegarde *PITR* fonctionnelle pour votre instance. Les journaux de transaction sont quant à eux stockés sur un stockage objet.

1.5.10 Backup

- `Custom Resource Definition`
- Définit l'exécution d'une sauvegarde

```
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  name: masauvegarde
spec:
  cluster:
    name: postgresql
```

Voici un exemple d'un objet `Backup` qui, une fois créé dans le cluster Kubernetes, déclenchera une sauvegarde physique du `Cluster` nommé `postgresql`. Si rien n'est mentionné, la sauvegarde se fera sur un stockage objet.

Il existe d'autres paramètres de configuration, comme :

- `target` : qui indique à partir de quelle instance doit être faite la sauvegarde ;
  - `prefer-standby` : pour demander à la faire depuis un secondaire ;
  - `primary` : pour demander à la faire depuis le primaire.
- `method` : permet de définir par quel moyen la sauvegarde physique doit être faite ;
  - `barmanObjectStore` : avec l'outil Barman Cloud sur un stockage objet (type S3, Google Cloud Storage, Azure Blob Storage ou MinIO). C'est le choix par défaut ;
  - `volumeSnapshot` : en se basant sur la fonctionnalité de `Volume Snapshot`. Votre `CSI` doit supporter cette fonctionnalité pour utiliser cette méthode ;
  - `plugin` : si la sauvegarde se fait à partir d'un *plugin* autre que vous aurez déployé au préalable. Fonctionnalité encore en test.

Pour les sauvegardes qui utilisent la méthode `barmanObjectStore`, les informations sur l'emplacement de stockage seront reprises de la configuration `spec.backup.barmanObjectStore` du `Cluster` référencé dans le champ `spec.cluster.name`. Il y aura deux dossiers, un contenant les journaux archivés, un contenant les sauvegardes.

Pour les sauvegardes qui utilisent la méthode `volumeSnapshot`, la configuration sera récupérée depuis `spec.backup.volumeSnapshot`.

Par défaut, les sauvegardes se font à chaud. Seule la méthode par `Volume Snapshot` permet de faire des sauvegardes à froid (i.e instance arrêtée).

1.5.11 ScheduledBackup

- `Custom Resource Definition`
- Définit la planification de sauvegardes
- Une ressource `Backup` créée à chaque exécution

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: ScheduledBackup
metadata:
  name: masauvegardequotidienne
spec:
  schedule: "0 0 20 * * *"
  backupOwnerReference: self
  cluster:
    name: postgresql
```

Il existe une deuxième ressource appelée `ScheduledBackup` qui, comme son nom l'indique, permet

de déclencher une sauvegarde régulièrement. L'exemple donné correspond au déclenchement d'une sauvegarde quotidienne à 20h00 :00.

Quelques paramètres de configuration sont spécifiques à cet objet, comme :

- `schedule` : définit le moment où la sauvegarde sera déclenchée. Il y a bien six arguments, correspondant aux secondes, minutes, heures, jours du mois, mois et jour de la semaine ;
- `backupOwnerReference` : indique à quelle ressource sera rattachée cette sauvegarde ;
  - `self` : au `ScheduledBackup` qui a déclenché cette sauvegarde ;
  - `cluster` : au `Cluster` mentionné ;
  - `none` : à aucune ressource.

D'autres paramètres comme `method` ou `target` peuvent être renseignés dans un `ScheduledBackup`.

Le paramètre `backupOwnerReference` a un impact sur la manière dont sont conservés les objets Kubernetes `Backup`. Dans l'exemple ci-dessus, si l'objet `ScheduledBackup` `masauvegardequotidienne` est supprimé, tous les objets `Backup` qui auraient été créés par la sauvegarde planifiée seront supprimés. Dans le cas où `backupOwnerReference` est configuré à `cluster`, les objets `Backup` seront supprimés si l'objet `Cluster` est supprimé. À `none` ils seront tout le temps conservés. Notez bien qu'il s'agit bien des objets Kubernetes. Les sauvegardes qui se trouvent sur le stockage S3 par exemple, ne seront pas supprimées.

Il est tout à fait possible de combiner ces deux types de déclenchements (manuel ou régulier) de sauvegardes.

1.5.12 Restauration

- Création d'une nouvelle instance
- À partir d'une sauvegarde physique
- `spec.bootstrap.recovery`
- Plusieurs sources de restauration
- *PITR* supporté

La première chose à noter est qu'une restauration d'une instance avec CloudNativePG se fera toujours par la création d'une nouvelle instance. Autrement dit, il n'est pas possible de faire une restauration sur une instance déjà déployée. La restauration *in-place* n'est pas possible.

CloudNativePG se base sur une sauvegarde physique pour créer cette nouvelle instance. Le paramètre de configuration `spec.bootstrap.recovery` permet de configurer cette restauration. Lorsque ce paramétre est utilisée dans le fichier `YAML`, CloudNativePG comprend qu'il doit créer (`bootstrap`) une instance à partir d'une sauvegarde.

Comme l'`archive_command`, le paramètre `restore_command` est déjà positionné par l'opérateur et utilise l'`instance-manager`. Cette fois-ci c'est la commande `wal-restore` qui est utilisée.

```
/controller/manager wal-restore --log-destination /controller/log/postgres.json %f
↪ %p
```

La source utilisée pour la restauration peut être de nature différente selon la méthode (`method`) de la sauvegarde.

- Si vous repartez d'une sauvegarde faite sur un stockage objets, utilisez le paramètre `bootstrap.recovery.source` associé à `externalClusters`. La partie `barmanObjectStore` contiendra alors toutes les informations du stockage objets où se trouve la sauvegarde. Par exemple :

```
spec:
[…]

bootstrap:
  recovery:
    source: clusterBackup

externalClusters:
  - name: clusterBackup
    barmanObjectStore:
    […]
```

- Si vous repartez d'un `Volume Snapshot`, vous devrez utiliser `bootstrap.recovery.volumeSnapshots.storag`
  Dans le cas où le snapshot a été fait depuis une instance ayant deux espaces de stockage (`storage` et `walStorage`) vous devez également le renseigner.

Quelques éléments supplémentaires sont à prendre en compte. Tout d'abord, CloudNativePG part du principe que la base `app` existe dans la sauvegarde et qu'elle a pour propriétaire `app`. Si vous utilisez d'autres noms, vous devez le renseigner dans la partie `recovery`. Aussi, si vous souhaitez conserver un mot de passe en particulier pour le rôle par défaut, vous devez renseigner le `Secret` à utiliser. Autrement, CloudNativePG se chargera d'en générer un aléatoirement.

Par défaut, la sauvegarde se fera en rejouant l'intégralité des journaux de transactions disponibles sur la dernière `timeline`. Il est possible de faire une restauration de type *Point In Time Recovery* en renseignant le champs `bootstrap.recovery.recoveryTarget`. Par exemple :

```
[…]
      recoveryTarget:
        # Time base target for the recovery
        targetTime: "2023-08-11 11:14:21.00000+02"
```

D'autres cibles de restauration existent, comme :

- `targetTime` : l'horodatage auquel vous souhaitez restaurer votre instance ;
- `targetXID` : l'ID de transaction jusqu'auquel vous souhaitez restaurer votre instance ;
- `targetName` : le nom du point de restauration que vous aurez créé au préalable avec `pg_create_restore_point()` ;
- `targetLSN` : La position dans les journaux de transactions à laqulle arrêter la restauration ;
- `targetImmediate` : Indique si la restauration doit s'arrêter dès qu'un point de consistance est atteint.

1.5.13 Montée de version mineure de PostgreSQL

- Version PostgreSQL indiquée dans l'image
  - Modifier l'image pour monter de version
- Version mineure uniquement
- Mode *Rolling Update*
- `primaryUpdateStrategy` et `primaryUpdateMethod`
  - Définis dans le YAML du `Cluster`

Une montée de version consiste à la mise à jour des binaires. En mode conteneur, ou sur Kubernetes, il n'est pas envisageable de les mettre à jour via `apt` ou `yum`. Cela se fait en modifiant l'image utilisée dans la définition `YAML` de votre `Cluster` PostgreSQL. Une nouvelle image contenant PostgreSQL dans la version souhaitée sera alors téléchargée.

La modification de l'image entraine une montée de version de toutes les instances du `Cluster`. Cette montée de version se fera en mode *Rolling Update*. Les instances secondaires seront les premières à être mises à jour, une par une. L'instance primaire est la dernière à être mise à jour.

La mise à jour peut se faire automatiquement ou de manière supervisée selon le paramètre de `primaryUpdateStrategy` qui peut prendre deux valeurs.

- `unsupervised` : après que toutes les instances secondaires aient été mises à jour, l'instance primaire est automatiquement mise à jour. C'est la valeur par défaut ;
- `supervised` : le mécanisme de montée de version attend une opération manuelle pour effectuer la montée de version de l'instance primaire.

En mode `unsupervised`, le paramètre `primaryUpdateMethod` est pris en compte pour savoir comment procéder à la mise à jour de l'instance primaire. Il peut lui aussi prendre deux valeurs :

- `restart` : l'instance primaire est redémarrée. Il faudra attendre la fin de la mise à jour pour que le primaire soit de nouveau accessible. C'est la valeur par défaut ;
- `switchover` : une bascule sur un secondaire est effectuée et le primaire devient secondaire. Le nouveau primaire est déjà accessible comme il a déjà été mis à jour.

En mode `supervised`, vous devrez donc vous même procéder à ce redémarrage ou à ce *switchover* avec le plugin `cnpg`. Par exemple pour le *switchover* vous pouvez utiliser la commande suivante pour passer l'instance `postgresql-2` en tant que nouvelle primaire :

```
kubectl cnpg promote postgresql 2
```

Si au contraire, vous souhaitez procédre à un redémarrage du primair vous pouvez devrez la commande `restart` du plugin.

```
kubectl cnpg restart postgresql 1
```

À l'heure actuelle, la montée de version majeure en mode déclaratif n'est pas supportée par CloudNativePG.

1.5.14 Haute disponibilité et bascule

- Répartition sur les nœuds
- Bascule
  - Automatique
  - Manuelle

Nous l'avons vu, l'ajout de secondaire se fait très simplement en modifiant le paramètre `spec.instances`. Par défaut, CloudNativePG essaye de répartir les instances PostgreSQL d'un même `Cluster` sur des nœuds différents. Cette configuration est modifiable dans la partie `spec.affinity` de la définition `YAML`.

Voici quatre paramètres intéressants :

- `enablePodAntiAffinity` : par défaut à `true`, il indique si ce mécanisme d'affinité / anti-affinité doit être utilisé ou non ;
- `topologyKey` : indique sur quel paramètre va se reposer la répartition des `Pods`. Par défaut à `kubernetes.io/hostname`, le répartition se fera selon le nom du nœud. Il est possible d'utiliser d'autres valeurs comme `topology.kubernetes.io/zone` pour répartir les `Pods` non pas selon les nœuds mais sur des zones de disponibilité ;
- `podAntiAffinityType` : ce paramètre permet d'indiquer le type d'affinité qui doit être utilisé. À `prefered`, le *scheduler* de Kubernetes tente de respecter la règle d'affinité. Si il ne peut pas, il déploiera tout de même le `Pod` sur un nœud. À `required`, le `Pod` sera déployé uniquement si la règle est respectée.
- `nodeSelector` : vous permet de sélectionner les nœuds où pourra être déployé un `Pod`. Les nœuds qui possèdent ces *labels* seront les cibles potentielles du déploiement.

CloudNativePG gère nativement la bascule, qu'elle soit manuelle ou automatique. Par bascule, on entend la promotion d'un secondaire en primaire. Vous pouvez promouvoir une instance secondaire en primaire grâce au plugin `cnpg` pour `kubectl`.

```
kubectl cnpg promote CLUSTER CLUSTER-INSTANCE
```

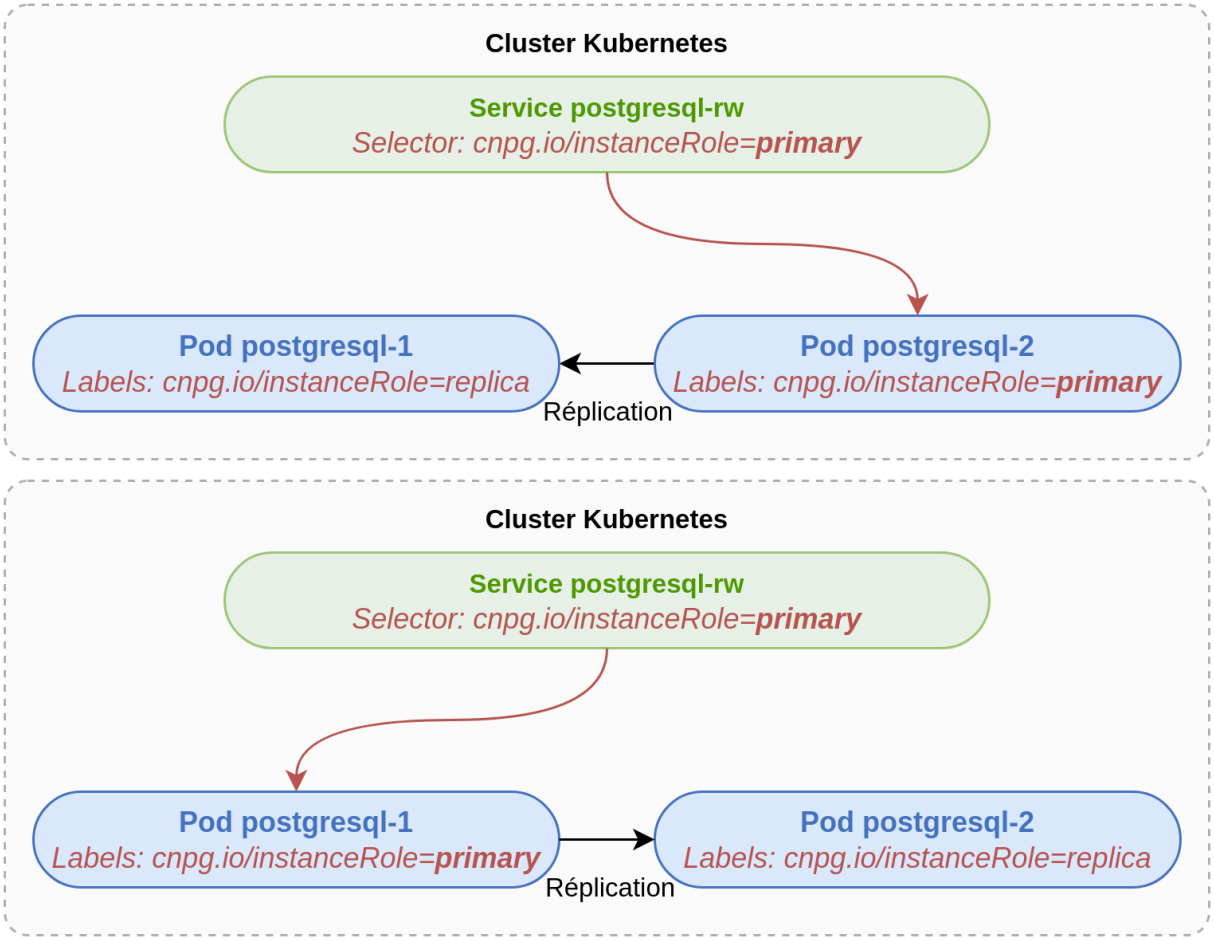
Il faut s'avoir qu'à chaque instance est associé un ou plusieurs *labels*. Celui qui nous intéresse particulièrement est `cnpg.io/instanceRole`. Chacun des `Pod` PostgreSQL possède ces *labels*, qui sont ajoutés automatiquement par l'opérateur CloudNativePG.

Ce *label* permet donc de connaitre le rôle de l'instance. Les `Services` Kubernetes, automatiquement créés, permettent de se connecter à un `Pod`. L'association `Service` - `Pod` se fait grâce à ces *labels*, et plus précisement grâce aux `Selector` créés sur le `Service`.

Lorsque la commande de promotion est lancée, en plus d'autres opérations qui sont faites au niveau de PostgreSQL, le *label* `cnpg.io/instanceRole` va être mis à jour sur chaque `Pod`.

L'instance qui était jusqu'à présent la secondaire, devient la nouvelle instance primaire. Le *label* `cnpg.io/instanceRole` est mis à jour. Ainsi, si vos applications utilisent le `Service` `postgresql-rw` (où `postgresql` est le nom du `Cluster`) elles devront uniquement initier une nouvelle connexion pour interagir avec la nouvelle instance primaire.

Les deux schémas si dessous montrent ce principe. Entre ces deux schémas, une promotion de l'instance `postgresql-1` en tant que nouveau primaire a eu lieu. L'instance `postgresql-2` est redevenue secondaire. Le `Service` `postgresql-rw` a suivi ce changement grâce au mécanisme de *Label - Selector*.



1.5.15 Hibernation et fencing

- Hibernation
  - Arrêter le `Pod` en conservant les volumes
  - Déclarative ou via le plugin `cnpg`
- Fencing
  - Arrêter uniquement le service `postmaster`

Le principe d'hibernation vous permet d'arrêter un `Cluster` PostgreSQL tout en conservant les volumes de données. Les `Pods` PostgreSQL seront arrêtés un à un en commençant par le primaire. Les `PVC` et `PV` de chaque instance, qu'elle soit primaire ou secondaire, existeront toujours.

La première méthode pour passer un `Cluster` en hibernation est de lui ajouter l'annotation `cnpg.io/hibernation=on`, avec par exemple, la commande suivante :

```
kubectl annotate cluster <cluster-name> --overwrite cnpg.io/hibernation=on
```

Voici un exemple de situation lorsqu'une instance est en hibernation.

```
$ kubectl get pod
No resources found in default namespace.

$ kubectl get pvc
NAME          STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
postgresql-1   Bound    pvc-bb811ce2-c4db-4f3d-8286-1187bcd91174   2Gi        RWO            standard       <unset>                 7m30s
postgresql-2   Bound    pvc-b8b35ea1-073c-4dd9-8ff1-3fc9d6f60418   2Gi        RWO            standard       <unset>                 2m2s
```

Il n'y a plus de `Pod` mais les `PVC` sont bien encore présents. Si vous souhaitez redéployer les `Pods` (primaire et secondaire) du `Cluster`, utilisez la même commande avec l'option `off` cette fois-ci.

Il est également possible d'hiberner un `Cluster` avec le plugin `cnpg` de `kubectl`. Une différence notable existe entre ces deux méthodes. Avec cette deuxième méthode, seul le `PVC` de l'instance primaire est conservé.

```
$ kubectl cnpg hibernate on postgresql
hibernation process starting...
waiting for the cluster to be fenced
cluster is now fenced, storing primary pg_controldata output
primary pg_controldata output fetched
annotating the PVC with the cluster manifest
PVC annotation complete
destroying the primary instance while preserving the pvc
Instance postgresql-1 of cluster postgresql has been destroyed and the PVC was kept
primary instance destroy completed
deleting the cluster resource
cluster resource deletion complete
Hibernation completed


$ kubectl get pvc
NAME          STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
postgresql-1   Bound    pvc-bb811ce2-c4db-4f3d-8286-1187bcd91174   2Gi        RWO            standard       <unset>                 10m
```

Lorsque le `Cluster` sortira d'hibernation (avec `kubectl cnpg hibernate off postgresql`) les `PVCs` nécessaires aux secondaires seront recréés. Selon la volumétrie des instances, cette copie pourra représenter beaucoup de volume et de temps.

Le *fencing* quant à lui permet d'arrêter le service `postmaster` de PostgreSQL sans pour autant arrêter le `Pod`. Cela revient à faire une arrêt propre de l'instance PostgreSQL au sein du `Pod`.

Il est possible de passer en *fencing* une instance spécifique (qu'elle soit primaire ou secondaire), une liste d'instances, ou toutes les instances d'un `Cluster`. Là encore, ce mécanisme est géré par une annotation, en l'occurence `cnpg.io/fencedInstances`. Par exemple, avec :

— `cnpg.io/fencedInstances: '["postgresql-1"]'`, seule cette instance sera en *fencing* ;

- `cnpg.io/fencedInstances: '["postgresql-1","postgresql-3"]'`, ces deux instances seront passées en *fencing* ;
- `cnpg.io/fencedInstances: '["*"]'`, toutes les instances du `Cluster` qui seront annotés passeront en *fencing*.

Vous pouvez, soit utiliser la commande `kubectl annotate …` soit le plugin `cnpg` avec par exemple `kubectl cnpg fencing on postgresql 1`.

Lorsqu'une instance est passée en *fencing*, le `Pod` ne sera plus marqué `READY`, comme le montre cet exemple.

```
$ kubectl cnpg fencing on postgresql 1
postgresql-1 fenced

$ kubectl get pod
NAME           READY   STATUS    RESTARTS   AGE
postgresql-1   0/1     Running   0          19m
postgresql-2   1/1     Running   0          19m
```

Le `Pod` est toujours accessible, permettant par exemple de procéder à du débogage.

1.6 CONCLUSION

— Un opérateur complet, open-source, communautaire
— Déploiement et configuration facilités
— Approche déclarative
— Nouveaux mécanismes et configuration à connaitre
— Connaissance de PostgreSQL nécessaire

À travers ce module, vous a été présenté l'opérateur CloudNativePG, son principe de fonctionnement, son installation et une grande partie de ce qu'il permet de faire. Cela représente une bonne découverte de l'opérateur, tant sur ses fonctionnalités que sur certains aspects critiques de son utilisation.

L'opérateur CloudNativePG est celui qui connait la plus forte adoption ces dernières années. Il est complet (niveau 5 sur le site https://operatorhub.io/), *open-source*, avec un souhait de gouvernance partagée. Il fait d'ailleurs partie du projet d'incubation de la_ Cloud Native Computing Foundation_, garant de certains principes *open-source*.

Le déploiement d'instances PostgreSQL est facilité par la nature déclarative de la gestion par CloudNativePG. En se reposant sur les fonctionnalitées de Kubernetes, l'opérateur permet la mise en place de mécanismes complexes comme la haute disponibilité ou la bascule automatique.

Des mécanismes propres à l'opérateur sont à connaitre, comme sa mise à jour et les conséquences sur les instances, les différents paramètres et spécifications utilisables dans les fichiers `YAML` ou encore ce qui est automatiquement créé par l'opérateur (rôle, base, `Secret`, etc).

L'opérateur se repose sur un certain nombre de concepts liés à PostgreSQL. Il s'agit donc de bien les connaître (réplication par flux, sauvegarde *PITR*, etc) pour comprendre comment l'opérateur les utilise ou les configure.

L'intégration de PostgreSQL dans Kubernetes est grandement facilité par CloudNativePG. En contrepartie, cela demande à un administrateur PostgreSQL de vraies connaissances sur Kubernetes (qu'est-ce qu'un `Pod` ? un `Service` ? un `Secret` ?) et de revoir sa manière de travailler avec son SGBD favori.

1.6.1 Questions

1.7 QUIZ

https://dali.bo/k1_quiz

1.8 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/k1_solutions.

1.9 PRISE 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.

Trouver la version de l'utilitaire `kubectl`.

Lister les nœuds du cluster Kubernetes.

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`.

1.10 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 la version 1.25.1 de l'opérateur avec la commande `kubectl apply -f` :

Lister les `Pods` présents dans le `namespace` `cnpg-system`.

Retrouver la liste des nouvelles ressources créées.

1.11 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
  walStorage:
    size: 5Gi
  affinity:
    enablePodAntiAffinity: true
    topologyKey: kubernetes.io/hostname
    podAntiAffinityType: preferred
  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 pour le PGDATA ;
- `walStorage` : Les informations sur le stockage souhaité pour les WALs ;
- `affinity` : Indique où et comment seront déployées les instances de ce `Cluster` ;
- `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.

> 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`.

> 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`.

> 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
```

> Se déconnecter de l'instance.

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

1.12 ÉLÉMENTS INITIAUX

But : Découvrir les éléments automatiquement créés par l'opérateur.

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

Bases de données

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

Rôles et `Secret`

Retrouver la liste des rôles dans l'instance déployée.

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

Récupérer la liste des `Secrets` du cluster.

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

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

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.

Retrouver les *labels* définis sur le `Pod` de votre instance.

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

1.12.1 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.

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`.

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.

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

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`;

1.12.2 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.

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

Vérifier que le rôle a bien été créé.

Encoder le nom du rôle en base64.

Encoder le mot de passe en base64.

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

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

Appliquer les modifications.

1.12.3 Créer une base de données

Créer une base de données `db1` dans l'instance `postgresql-demo`. Le propriétaire de cette base doit être le rôle `app`. Pour cela, créer un fichier `db1.yaml` contenant la définition d'une ressource `Database`.

Vérifier la présence de cette base de données dans l'instance.

1.13 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 en modifiant le paramètre `instances` dans la section `spec` de notre fichier yaml.

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

1.13.1 Configuration par défaut

Regardons la configuration qui est mise en place par défaut.

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

> Récupérer le contenu du paramètre `primary_conninfo`.

> Se connecter avec `psql` au primaire.

> Récupérer le contenu de la table `pg_stat_replication`.

> Récupérer le contenu de la table `pg_replication_slots`.

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

1.13.2 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.

1.14 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`.

1.14.1 Bascule manuelle

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

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

1.14.2 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*[19]).

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

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

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

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

1.15 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'éco-système PostgreSQL, cet outil nous permet de faire la sauvegarde physique et l'archivage des WALs. 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.

1.15.1 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`.

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
```

Créer le nouveau cluster PostgreSQL avec la commande :

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

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

1.15.2 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
```

Appliquer ce fichier avec `kubectl`.

Vérifier le statut de l'objet `Backup`.

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

1.15.3 Générer de la donnée

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

1.16 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).

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.

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.

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
```

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

1.17 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).

1.17.1 Méthode *unsupervised*

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

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`.

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.

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

Faite une bascule manuelle sur l'instance secondaire.

1.18 MONTÉE DE VERSION DE L'OPÉRATEUR

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

Nous avons déployé la version `1.25.1` 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/manager instance run--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 `1.25.2` de l'opérateur.

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

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

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

1.19 EXERCICES OPTIONNELS

But : Découvrir des fonctionnalités plus complexes.

1.19.1 Mise en place d'un cluster Kubernetes (minikube)

> Voir les explications dans les solutions du TP.

1.19.2 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`.

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

1.19.3 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
[...]
```

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

1.19.4 Déploiement d'une application pgAdmin4

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

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
```

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

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

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

> Créer une nouvelle connexion avec les informations suivantes : 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);`.

1.20 TRAVAUX PRATIQUES (SOLUTIONS)

1.21 PRISE EN MAIN DU CLUSTER KUBERNETES (MINIKUBE)

But : Prendre en main le cluster Kubernetes `k8s-demo`.

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: v1.31.0
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
```

1.22 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 la version 1.25.1.0 de l'opérateur avec la commande `kubectl apply -f` :

```
kubectl apply --server-side -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/release-
 ↪  1.25/releases/cnpg-1.25.1.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
```

1.23 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
  walStorage:
    size: 5Gi
  affinity:
    enablePodAntiAffinity: true
    topologyKey: kubernetes.io/hostname
    podAntiAffinityType: preferred
  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 pour le PGDATA ;
- `walStorage` : Les informations sur le stockage souhaité pour les WALs ;
- `affinity` : Indique où et comment seront déployées les instances de ce `Cluster` ;
- `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
  walStorage:
    size: 5Gi
  affinity:
    enablePodAntiAffinity: true
    topologyKey: kubernetes.io/hostname
    podAntiAffinityType: preferred
  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 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[20]) :

```
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
```

[20] 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.

> Se déconnecter de l'instance.

```
postgres=# \q
```

> 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":"2025-03-25T14:30:48.192432437Z","msg":"Starting CloudNativePG
 ↪  Instance Manager","logger":"instance-manager","logging_pod":"postgresql-demo-
 ↪  1","version":"1.25.1","build":{"Version":"1.25.1","Commit":"c56e00d4","Date":"2025-
 ↪  02-28"}}
{"level":"info","ts":"2025-03-25T14:30:48.192533981Z","msg":"Checking for free disk
 ↪  space for WALs before starting
 ↪  PostgreSQL","logger":"instance-manager","logging_pod":"postgresql-demo-1"}
{"level":"info","ts":"2025-03-25T14:30:48.203813591Z","msg":"starting tablespace
 ↪  manager","logger":"instance-manager","logging_pod":"postgresql-demo-1"}
```

Les logs de l'instances 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"
  }
}
```

1.24 ÉLÉMENTS INITIAUX

But : Découvrir les éléments automatiquement créés par l'opérateur.

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

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 -c "\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.

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 "VFdyejR-
 ↪  QbmY1RWMwVjFjUHlqYkdFZnI5RG52WE5YaXN0NUhIaFZkOENwSkpKOEthVkVLUkNxUGwweTRzaGlVbw=="
 ↪  | 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`.

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: 1.25.1
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.

1.24.1 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.y`

```
[...]

  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[21]).

1.24.2 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:
    roles:
    - name: dba
      ensure: present
```

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

```
    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=#
```

1.24.3 Créer une base de données

> Créer une base de données `db1` dans l'instance `postgresql-demo`. Le propriétaire de cette base doit être le rôle `app`. Pour cela, créer un fichier `db1.yaml` contenant la définition d'une ressource `Database`.

Voici la déclaration d'une telle base de données.

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

```
$ kubectl apply -f db1.yaml
database.postgresql.cnpg.io/db1 created
```

> Vérifier la présence de cette base de données dans l'instance.

Plusieurs possibilités. En voici une :

```
$ kubectl cnpg psql postgresql-demo -- -c "\l"
                                                    List of databases
   Name    |  Owner   | Encoding | Locale Provider | Collate | Ctype | Locale | ICU
 ↪  Rules |   Access privileges
-----------+----------+----------+-----------------+---------+-------+--------+-----
 ↪  ------+-----------------------
 app       | app      | UTF8     | libc            | C       | C     |        |
 db1       | 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
(5 rows)
```

1.25 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 en modifiant le 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

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*).

1.25.1 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 liée 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)
```

1.25.2 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>
```

1.26 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  1.25.1
 ↪  k8s-demo-m03
postgresql-demo-2  0/4050170    Standby (async)   OK      Burstable  1.25.1
 ↪  k8s-demo-m02
```

1.26.1 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  1.25.1
 ↪  k8s-demo-m02
postgresql-demo-1  0/60000A0    Standby (starting up)  OK      Burstable  1.25.1
 ↪  k8s-demo
```

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

1.26.2 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*[22]).

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

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

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

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

```
watch kubectl get pod
```

> 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  1.25.1
 ↪  k8s-demo
postgresql-demo-2  0/70000A0    Standby (file based)  OK      Burstable  1.25.1
 ↪  k8s-demo-m02
```

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

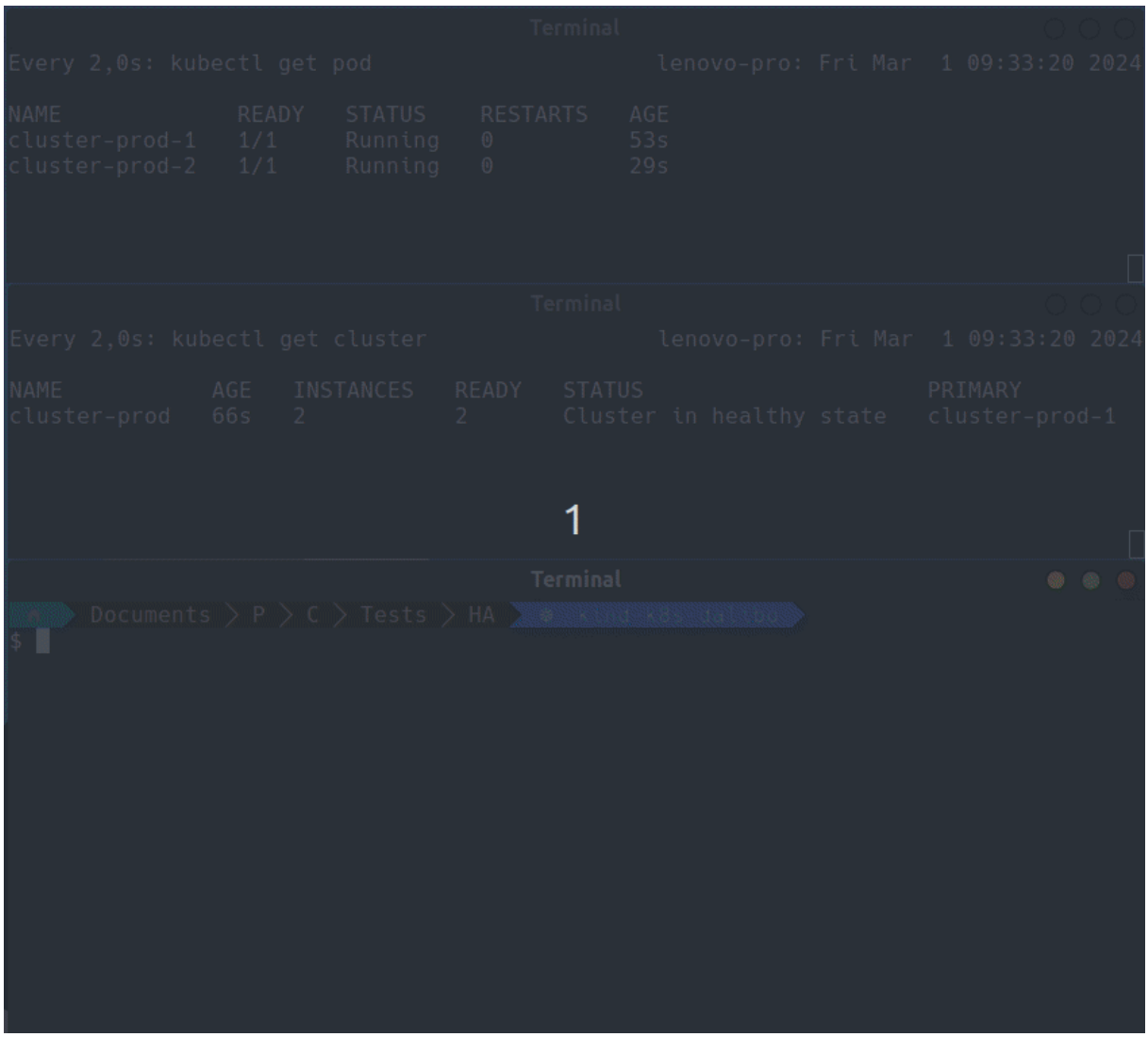


FIGURE 1/ .1 – bascule automatique

1.27 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'éco-système PostgreSQL, cet outil nous permet de faire la sauvegarde physique et l'archivage des WALs. 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.

1.27.1 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)

```
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 :

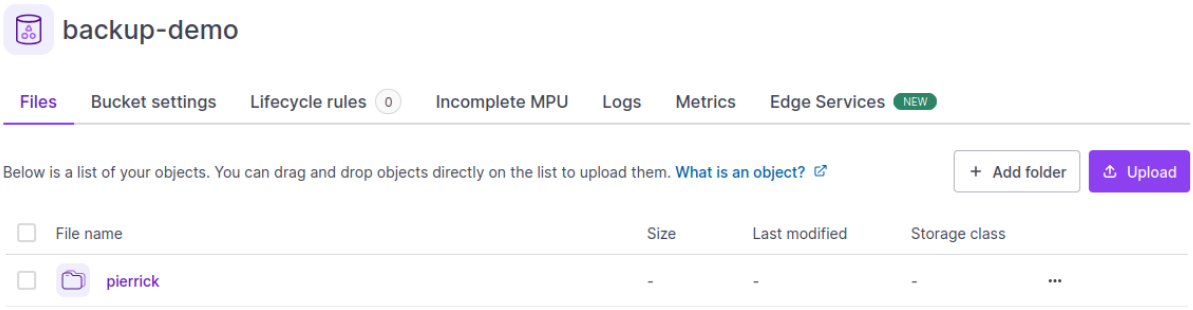
```
kubectl apply -f ~/postgresql-with-backup-demo.yaml
```

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

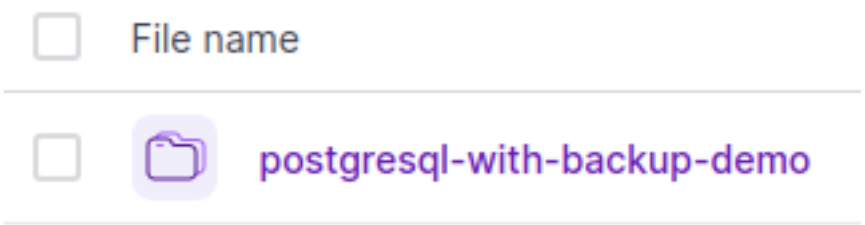
```
kubectl logs -f postgresql-with-backup-demo-1 | jq

[...]

{
  "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".

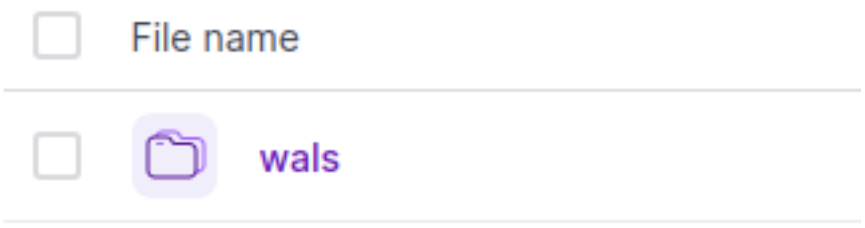
Pour ce TP, nous sommes passés par la solution `Object Storage` de Scaleway compatible S3. Voici un exemple de ce qu'il sera créé dans le `Bucket`.



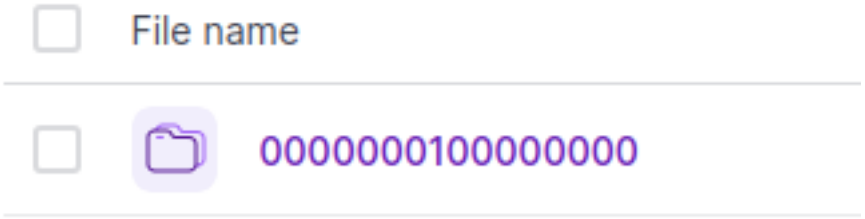
Le dossier `pierrick` est bien créé dans le `Bucket`.



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.

1.27.2 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
```

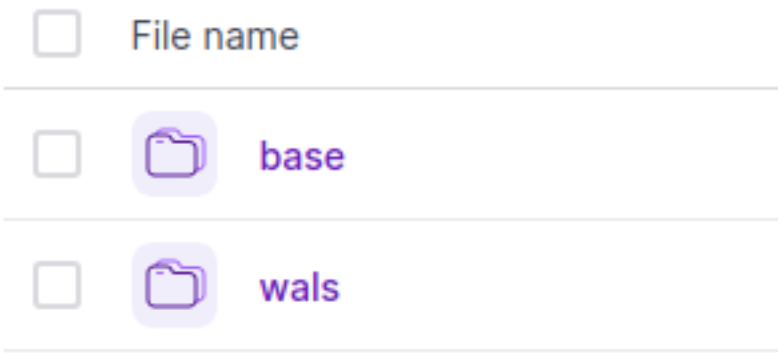
```
NAME           AGE     CLUSTER                       METHOD             PHASE       ERROR
first-backup   4m41s   postgresql-with-backup-demo   barmanObjectStore   completed
```

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

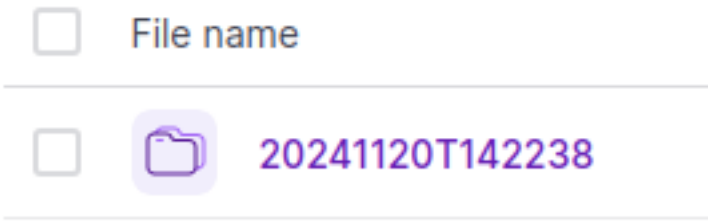
```
kubectl logs postgresql-with-backup-demo-1 | grep completed | jq
```

```
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"
}
```

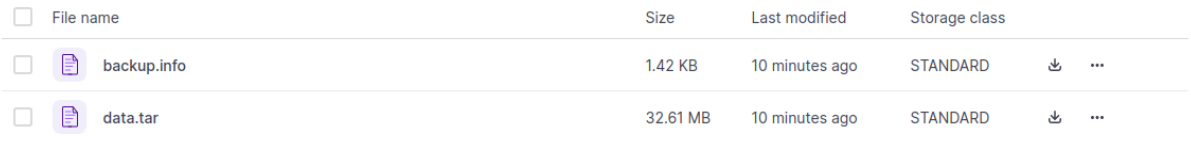
Au niveau de l'interface Scaleway, un nouveau dossier `base` est apparu à côté de `wals`.



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 WALs 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.

1.27.3 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`.

1.28 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 :

```yaml
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
```

1.29 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).

1.29.1 Méthode *unsupervised*

> 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     Terminatin
```

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`.

1.29.2 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 postgresq` qui donne l'état du cluster PostgreSQL, en particulier, avec la ligne `Primary instance:   postgresql-restored-der` ou encore dans le tableau.

```
Name                        Current LSN  Replication role  Status  QoS        Manager
 ↪  Version  Node
----                         -----------  ----------------  ------  ---
 ↪  ---------------  ----
postgresql-restored-demo-2  0/D001210    Primary           OK      Burstable  1.25.1
 ↪  k8s-demo-m03
postgresql-restored-demo-1  0/D001210    Standby (async)   OK      Burstable  1.25.1
 ↪  k8s-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]` ;

1.30 MONTÉE DE VERSION DE L'OPÉRATEUR

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

Nous avons déployé la version `1.25.1` 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 `1.25.2` de l'opérateur.

```
kubectl apply --server-side -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/release-
↪  1.25/releases/cnpg-1.25.2.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
```

1.31 EXERCICES OPTIONNELS

But : Découvrir des fonctionnalités plus complexes.

1.31.1 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]
 ↪  https://download.docker.com/linux/debian \
  $(. /etc/os-release && echo "$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 1.25.1) pour `kubectl` que nous utiliserons plus tard :

```
wget https://github.com/cloudnative-pg/cloudnative-
 ↪  pg/releases/download/v1.25.1/kubectl-cnpg_1.25.1_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*.

1.31.2 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.

1.31.3 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[23].

1.31.4 Déploiement d'une application pgAdmin4

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

> 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 :).

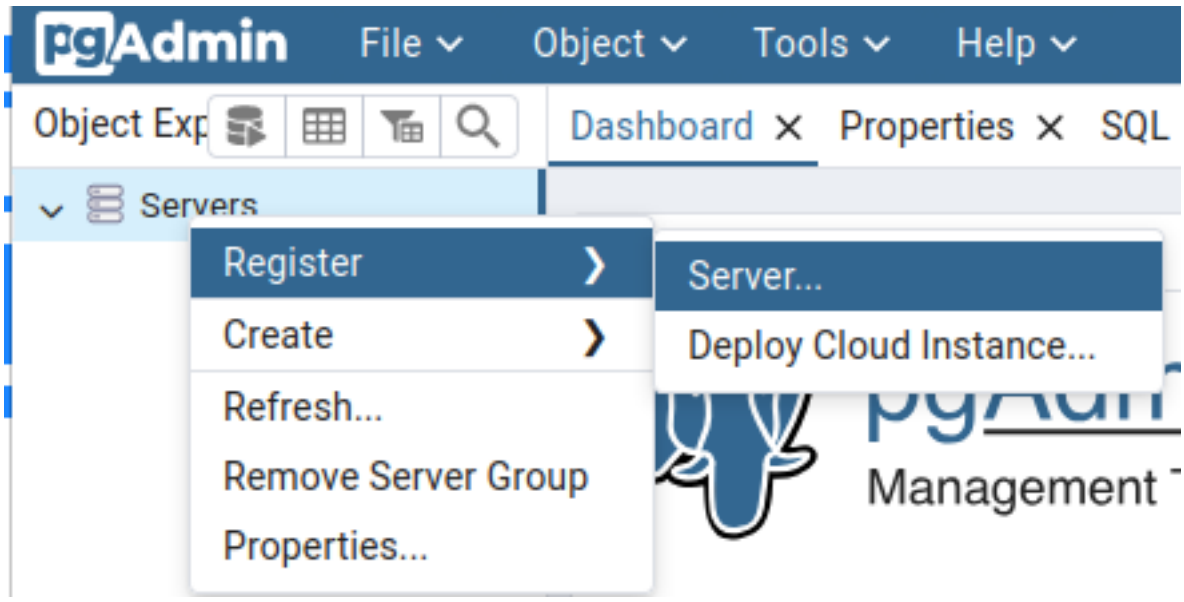
[23] https://cloudnative-pg.io/documentation/current/replication/#synchronous-replication

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
```

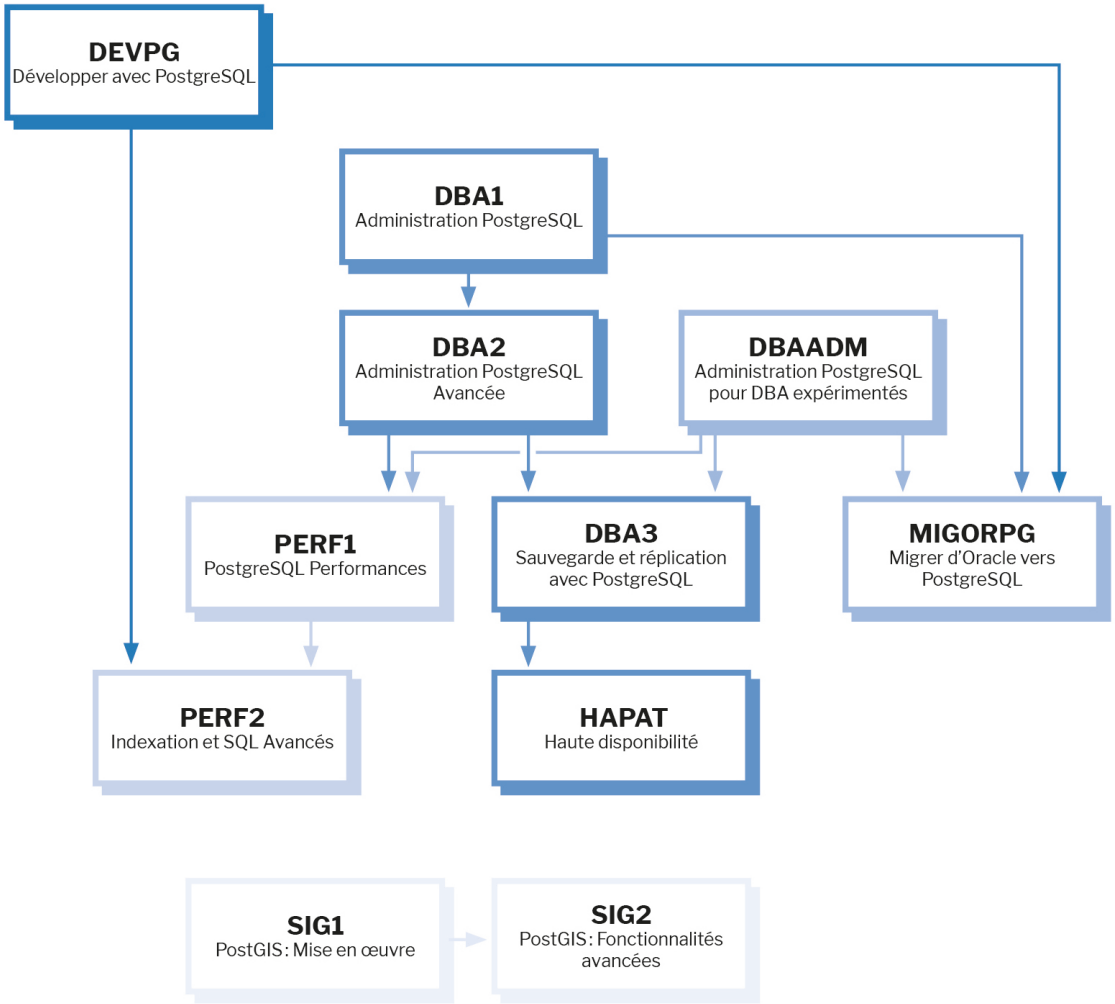
Vous avez maintenant l'application pgAdmin déployée dans Kubernetes qui a accès à l'instance `postgresql-demo`. L'exemple ci-dessus montre comment une application peut accéder à une base de données en utilisant la ressource `Service` prévue à cet effet. Application et base se trouvent toutes deux dans 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*.

Les formations Dalibo

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

- 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
- MIGORPG : Migrer d'Oracle à PostgreSQL
  https://dali.bo/migorpg
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

Les 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.