Kubernetes Cluster Logging mit Loki

Die kontinuierliche Überwachung eines Kubernetes-Clusters erfordert ein effizientes Logmanagement, welches das zentrale Sammeln, Speichern und Analysieren von Protokolldaten ermöglicht. In Kubernetes-Umgebungen wird hierfür traditionell ein Stack aus Logstash oder Fluentd zur Datenerfassung, Elasticsearch als Speichersystem sowie Kibana oder Graylog zur Analyse und Visualisierung eingesetzt. Neben diesem etablierten Ansatz gewinnt der Loki-Grafana-Stack zunehmend an Bedeutung, da er eine ressourcenschonendere und leichter implementierbare Alternative darstellt.

2025/09/04 12:03 · marko

So versetzen Sie einen Cloud Native PostgreSQL Operator Cluster in den Wartungsmodus

Um einen von Cloud Native PostgreSQL (CNPG) Operator verwalteten PostgreSQL-Cluster vollständig in den Wartungsmodus zu versetzen, gibt es keine einzelne, allumfassende Funktion. Stattdessen können je nach Wartungsanforderung zwei Hauptmethoden angewendet werden: die Aktivierung eines Wartungsfensters für Knoten oder das „Fencing“ des gesamten Clusters.

Methode 1: Verwendung des Knoten-Wartungsfensters (nodeMaintenanceWindow)

Diese Methode ist ideal für geplante Wartungsarbeiten an den Kubernetes-Knoten, auf denen die PostgreSQL-Instanzen laufen. Sie informiert den Operator darüber, dass ein Knoten absichtlich außer Betrieb genommen wird, sodass der Operator keine automatischen Failover-Prozesse einleitet.

Anwendungsfall: Sie planen, die Kubernetes-Knoten zu aktualisieren oder andere Wartungsarbeiten auf der Infrastrukturebene durchzuführen.

Schritte:

1. Aktivieren Sie das Wartungsfenster für den Cluster. Dies geschieht durch Setzen von inProgress: true im nodeMaintenanceWindow-Abschnitt der Cluster-Konfiguration. Sie können auch festlegen, ob das persistente Volume (PVC) wiederverwendet werden soll (reusePVC).
   Mit kubectl patch:

```
kubectl patch cluster <cluster-name> -n <namespace> --type='merge' -p
'{"spec":{"nodeMaintenanceWindow":{"inProgress":true,"reusePVC":true}}}
'
```

YAML Beispiel:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: <cluster-name>
  namespace: <namespace>
spec:
  # ... andere Cluster-Spezifikationen
  nodeMaintenanceWindow:
    inProgress: true
    reusePVC: true # Empfohlen, um Datenverlust zu vermeiden
```

2. **Führen Sie die Wartungsarbeiten** an den Knoten durch (z. B. `kubectl drain <node-name>`). Der CNPG-Operator wird die Pods auf dem Knoten geordnet herunterfahren.
3. **Deaktivieren Sie das Wartungsfenster**, nachdem die Wartungsarbeiten abgeschlossen sind, um den normalen Betrieb wieder aufzunehmen.
   Mit `kubectl patch`**:**

```
kubectl patch cluster <cluster-name> -n <namespace> --type='merge' -p
'{"spec":{"nodeMaintenanceWindow":{"inProgress":false}}}'
```

Methode 2: Fencing des gesamten Clusters

„Fencing“ ist eine drastischere Maßnahme, die den PostgreSQL-Prozess auf einer oder allen Instanzen stoppt, ohne die Pods selbst zu löschen. Dies versetzt die Datenbank effektiv in einen Offline-Zustand und ist nützlich, um den gesamten Cluster für Untersuchungen oder andere kritische Eingriffe anzuhalten.

Anwendungsfall: Sie müssen den gesamten Datenbankbetrieb sofort unterbrechen, um beispielsweise ein Problem zu diagnostizieren, ohne dass der Operator versucht, den Cluster zu „reparieren“.

Schritte:

1. **Fencen Sie alle Instanzen** des Clusters. Dies wird durch Hinzufügen einer Annotation zum Cluster-Objekt erreicht.
   Mit `kubectl annotate`**:**

```
kubectl annotate cluster <cluster-name> -n <namespace>
cnpg.io/fencedInstances='["*"]'
```

   Das `["*"]` ist ein Wildcard-Zeichen, das alle Instanzen des Clusters betrifft.

2. **Überprüfen Sie den Status.** Die PostgreSQL-Prozesse in den Pods werden beendet, aber die Pods laufen weiter. Anwendungen können keine Verbindung zur Datenbank herstellen.
3. **Heben Sie das Fencing auf**, um den normalen Betrieb wiederherzustellen.

Mit kubectl annotate:

```
kubectl annotate cluster <cluster-name> -n <namespace>
cnpg.io/fencedInstances-
```

Das - am Ende des Befehls entfernt die Annotation.

Wichtige Überlegungen

- **Anwendungsverbindungen:** Bevor Sie den Cluster in einen Wartungsmodus versetzen, sollten Sie idealerweise die Anwendungen, die auf die Datenbank zugreifen, herunterfahren oder skalieren. Dies verhindert unerwartete Fehler und Verbindungsabbrüche auf der Anwendungsseite.
- **Auswirkungen von Fencing:** Wenn der primäre Server gefenct wird, findet **kein Failover** statt. Der Cluster ist für Schreibvorgänge nicht verfügbar, bis das Fencing aufgehoben wird.
- `kubectl cnpg` **Plugin:** Der CNPG-Operator verfügt über ein `kubectl`-Plugin, das einige dieser Operationen vereinfachen kann. Zum Beispiel kann der Befehl `kubectl cnpg maintenance` verwendet werden, um das `nodeMaintenanceWindow` zu setzen.

Zusammenfassend lässt sich sagen, dass das `nodeMaintenanceWindow` für geplante Infrastrukturwartungen gedacht ist, während das Fencing des gesamten Clusters die beste Methode ist, um den Datenbankbetrieb vollständig und sofort zu stoppen.

2025/08/10 16:15 · marko

Automatisiere Deine Git-Commit-Nachrichten mit ChatGPT

Das Erstellen aussagekräftiger und prägnanter Commit-Nachrichten ist ein wesentlicher Bestandteil eines guten Entwicklungsworkflows. Diese Nachrichten helfen dabei, Änderungen zu verfolgen, den Projektverlauf zu verstehen und mit Teammitgliedern zusammenzuarbeiten. Zugegeben: Das Schreiben von Commit-Nachrichten kann manchmal eine banale Aufgabe sein. In diesem Artikel zeigen ich Dir, wie Du mit ChatGPT von OpenAI automatisch Git-Commit-Nachrichten generieren lassen kannst.

Das Skript

```
#!/bin/bash

check_git_repo() {
    if ! git rev-parse --is-inside-work-tree >/dev/null 2>&1; then
        exit 1
```

```
    fi
}

check_changes() {
    if [ -z "$(git status --porcelain)" ]; then
        exit 0
    fi
}

generate_commit_message() {
    local diff_content=$(git diff --cached)
    local files_changed=$(git status --porcelain)

    echo -e "Files changed:\n$files_changed\n\nChanges:\n$diff_content" | \
        llm -m anthropic/claude-3-5-sonnet-latest \
        "Generate a git commit message for these changes. The message must
have:

        1. TITLE LINE: A specific, concise summary (max 50 chars) begin
without special characters
           that clearly describes the primary change or feature. This should
not be generic like
           'Update files' but rather describe the actual change like 'Add
user
           authentication to API endpoints'

        2. BLANK LINE

        3. DETAILED DESCRIPTION: A thorough explanation including:
           - What changes were made
           - Why they were necessary
           - Any important technical details
           - Breaking changes or important notes
           Wrap this at 72 chars.

        IMPORTANT:
        - Output ONLY the commit message
        - Make sure the title is specific to these changes
        - Focus on the what and why, not just the how"
}

# Main execution
main() {
    check_git_repo
    check_changes
    git add --all
    commit_message=$(generate_commit_message)
    git commit -m "$commit_message"
}
```

```
main "$@"
```

Aufschlüsselung

Repository-Validierung

```
check_git_repo() {
    if ! git rev-parse --is-inside-work-tree >/dev/null 2>&1; then
        exit 1
    fi
}
```

Diese Funktion stellt sicher, dass wir in einem Git-Repository arbeiten.

Änderungserkennung

```
check_changes() {
    if [ -z "$(git status --porcelain)" ]; then
        exit 0
    fi
}
```

Überprüft, ob tatsächlich Änderungen zum Festschreiben vorhanden sind.

KI-gestützte Nachrichtengenerierung

```
generate_commit_message() {
    local diff_content=$(git diff --cached)
    local files_changed=$(git status --porcelain)

    echo -e "Files changed:\n$files_changed\n\nChanges:\n$diff_content" | \
        llm -m anthropic/claude-3-5-sonnet-latest \
        "Generate a git commit message for these changes. The message must
have:

        1. TITLE LINE: A specific, concise summary (max 50 chars) that
clearly
           describes the primary change or feature. This should not be
generic like
           'Update files' but rather describe the actual change like 'Add
user
           authentication to API endpoints'

        2. BLANK LINE

        3. DETAILED DESCRIPTION: A thorough explanation including:
```

```
            - What changes were made
            - Why they were necessary
            - Any important technical details
            - Breaking changes or important notes
            Wrap this at 72 chars.

        IMPORTANT:
        - Output ONLY the commit message
        - Make sure the title is specific to these changes
        - Focus on the what and why, not just the how"
}
```

Hier geschieht die Magie – das Skript analysiert Deine Änderungen und verwendet KI, um eine aussagekräftige Commit-Nachricht zu generieren.

Das Skript verwendet Simon Willisons Kommandozeilentool llm , ein äußerst nützliches Dienstprogramm für die Interaktion mit verschiedenen KI-Modellen direkt von Deinem Terminal aus. Weitere Informationen dazu, wie Du es einrichtest und tatsächlich verwendest, findest Du in seiner Dokumentation.

Bitte beachte, dass ich in diesem Skript das Modell von Anthropic verwende. Das bedeutet, dass Du das Plugin „llm-anthropic“ einrichten musst.

Einrichten

Um das Skript am Ende auch auszuführen zu können, erstellst Du einfach eine ausführbare Commit-Datei und fügst diese in Dein Bin-Verzeichnis hinzu, sodass sie in Deinem PATH landet.

Vergiss nicht:

```
chmod +x ~/.local/bin/commit
```

um das Skript ausführbar zu machen. Ändere einfach den Pfad zum Skript, je nachdem, wo Du es speichern möchtest.

Juhuu

Nachdem Du hart an Deinem Code gearbeitet hast, musst Du ihn nur noch `**commit**` ausführen und erhälst eine von der KI generierte Commit-Nachricht.

2025/06/21 09:43 · marko

Automatisches Paperless-ngx Backup in

Kubernetes: CronJobs mit dynamischer Pod-Ermittlung

Manchmal ist es notwendig, in einem Kubernetes-Cluster regelmäßig Aufgaben auszuführen, die sich auf laufende Anwendungspods beziehen. Denken Sie an die Erstellung von Backups, das Generieren von Berichten oder das Ausführen von Wartungsskripten. Kubernetes bietet hierfür CronJobs an. Doch was, wenn der Ziel-Pod für Ihre Aufgabe nicht immer denselben Namen hat, weil er zum Beispiel bei jedem Deployment neu erstellt wird oder dynamisch skaliert wird?

Genau hier kommt die `dynamische Pod-Ermittlung in Kubernetes CronJobs` ins Spiel! In diesem Blogbeitrag zeigen wir Ihnen, wie Sie einen Kubernetes CronJob konfigurieren, der den zu interagierenden Pod dynamisch zur Laufzeit identifiziert. Dies macht Ihre Automatisierung robuster und wartungsfreundlicher, da Sie den CronJob nicht bei jeder Pod-Änderung anpassen müssen.

Warum dynamische Pod-Ermittlung?

Stellen Sie sich vor, Sie haben eine Anwendung wie Paperless-ngx, die in einem Kubernetes-Deployment läuft. Jedes Mal, wenn Sie ein Update bereitstellen oder die Anwendung neu startet, könnte der Pod, in dem die Hauptanwendung läuft, einen neuen Namen erhalten (z.B. `paperless-ngx-789abcde-xyz12`). Wenn Ihr CronJob fest an einen spezifischen Pod-Namen gebunden wäre, würde er fehlschlagen, sobald der Pod-Name sich ändert.

Durch die dynamische Ermittlung des Pod-Namens zur Laufzeit des CronJobs umgehen wir dieses Problem vollständig. Der CronJob sucht einfach nach einem Pod, der bestimmte Kriterien erfüllt (z.B. ein spezifisches Label), und führt den Befehl dann im `aktuell` gefundenen Pod aus.

Das Kernstück: Der Kubernetes CronJob

Das Herzstück unserer Lösung ist der Kubernetes CronJob. Er definiert den Zeitplan und die auszuführende Aufgabe. Werfen wir einen Blick auf die entscheidenden Teile des Manifests:

```
# ============================================================
#  ServiceAccount, Role, RoleBinding
# ============================================================
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kubectl-exec-sa
  namespace: paperless-production # <-- Anpassen
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
```

```
metadata:
  name: pod-exec-role
  namespace: paperless-production # <-- Anpassen
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list"]
  - apiGroups: [""]
    resources: ["pods/exec"]
    verbs: ["create"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-exec-binding
  namespace: paperless-production # <-- Anpassen
subjects:
  - kind: ServiceAccount
    name: kubectl-exec-sa
    namespace: paperless-production # <-- Anpassen
roleRef:
  kind: Role
  name: pod-exec-role
  apiGroup: rbac.authorization.k8s.io
---
# ======================================================================
#  CronJob (mit dynamischer Pod-Ermittlung)
# ======================================================================
apiVersion: batch/v1
kind: CronJob
metadata:
  name: run-document-exporter-dynamic
  namespace: paperless-production # <-- Anpassen
spec:
  timeZone: Europe/Berlin # Zeitzone für den Zeitplan

  schedule: "12 5 * * *" # Jeden Tag um 05:12 Uhr

  concurrencyPolicy: Forbid # Verhindert parallele Ausführungen
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 1

  jobTemplate:
    spec:
      template:
        spec:
          serviceAccountName: kubectl-exec-sa # Verwendet das ServiceAccount
für die Berechtigungen
          restartPolicy: OnFailure
          containers:
            - name: kubectl-executor
```

```
            image: bitnami/kubectl:1.29 # Ein Container mit kubectl
            command:
              - "sh"
              - "-c"
              - |
                set -e # Bricht bei Fehlern ab
                echo "Suche nach einem laufenden Pod mit dem Label-
Selector '$TARGET_POD_LABEL_SELECTOR' im Namespace
'$KUBERNETES_NAMESPACE'..."

                # Finde den Namen des ERSTEN laufenden Pods, der dem
Label-Selector entspricht.
                TARGET_POD_NAME=$(kubectl get pods --namespace
"$KUBERNETES_NAMESPACE" -l "$TARGET_POD_LABEL_SELECTOR" --field-
selector=status.phase=Running -o jsonpath='{.items[0].metadata.name}')

                # Überprüfe, ob ein Pod-Name gefunden wurde.
                if [ -z "$TARGET_POD_NAME" ]; then
                  echo "FEHLER: Kein laufender Pod mit dem Label-Selector
'$TARGET_POD_LABEL_SELECTOR' gefunden."
                  exit 1
                fi

                echo "Pod '$TARGET_POD_NAME' gefunden. Führe Befehl im
Container '$TARGET_CONTAINER_NAME' aus..."

                # Führe den eigentlichen Befehl aus, jetzt mit dem
dynamisch ermittelten Pod-Namen
                kubectl exec --namespace "$KUBERNETES_NAMESPACE"
"$TARGET_POD_NAME" -c "$TARGET_CONTAINER_NAME" -- /bin/sh -c
"document_exporter ../export -z --delete"

                echo "Befehlsausführung erfolgreich abgeschlossen."

            env:
              - name: TARGET_POD_LABEL_SELECTOR
                value: "app.kubernetes.io/name=paperless-ngx" # <--
Anpassen: Label des Ziel-Pods

              # - name: TARGET_CONTAINER_NAME
              #   value: "<NAME_DES_ZIEL-CONTAINERS_HIER_EINFUEGEN>" # <--
Anpassen (optional): Name des Ziel-Containers

              - name: KUBERNETES_NAMESPACE
                valueFrom:
                  fieldRef:
                    fieldPath: metadata.namespace # Der Namespace wird
automatisch bereitgestellt
```

Erläuterung der Schlüsselkomponenten

- `ServiceAccount, Role, RoleBinding`:
  - *`ServiceAccount`* (`kubectl-exec-sa`): Dieses Dienstkonto wird dem CronJob zugewiesen, um ihm die notwendigen Berechtigungen innerhalb des Clusters zu geben.
  - *`Role`* (`pod-exec-role`): Diese Rolle definiert die Berechtigungen. Hier erlauben wir dem ServiceAccount, Pods zu `get` (abzurufen) und zu `list` (aufzulisten) sowie `pods/exec` (`create`) zu nutzen, um Befehle in einem Pod auszuführen.
  - *`RoleBinding`* (`pod-exec-binding`): Dieses Binding verknüpft das `ServiceAccount` mit der `Role` im angegebenen `namespace`.
- `CronJob (run-document-exporter-dynamic)`:
  - *`namespace: paperless-production`*: Stellen Sie sicher, dass dieser Namespace zu Ihrem tatsächlichen Anwendungs-Namespace passt.
  - *`timeZone: Europe/Berlin`*: Wichtig für die korrekte Planung der Ausführung, insbesondere bei Sommer- und Winterzeitumstellungen.
  - *`schedule: „12 5 * * *“`*: Dies ist der Cron-String. In diesem Beispiel bedeutet er „jeden Tag um 5:12 Uhr“. Passen Sie ihn an Ihre Bedürfnisse an.
  - *`concurrencyPolicy: Forbid`*: Verhindert, dass neue Job-Instanzen gestartet werden, wenn eine vorherige Instanz noch läuft. Dies ist oft ratsam für Wartungsaufgaben.
  - *`successfulJobsHistoryLimit`* und *`failedJobsHistoryLimit`*: Legt fest, wie viele vergangene Job-Ausführungen im Kubernetes-API gespeichert werden. Nützlich für die Fehlerbehebung.
  - *`serviceAccountName: kubectl-exec-sa`*: Verbindet den Job mit dem zuvor definierten ServiceAccount.
  - *`image: bitnami/kubectl:1.29`*: Dieser Container enthält das `kubectl`-Tool, das wir für die Pod-Ermittlung und Befehlsausführung benötigen. Stellen Sie sicher, dass Sie eine passende Version für Ihre Kubernetes-Cluster-Version verwenden.
- `Dynamische Logik im` *`command`* `Sektion`:
  - *`set -e`*: Stellt sicher, dass das Skript sofort beendet wird, wenn ein Befehl fehlschlägt. Das ist eine gute Praxis für Shell-Skripte.
  - *`TARGET_POD_NAME=$(kubectl get pods … -o jsonpath='{.items[0].metadata.name}')`*: Dies ist der Kern der dynamischen Ermittlung.
    - `kubectl get pods`: Listet die Pods auf.
    - `–namespace „$KUBERNETES_NAMESPACE“`: Sucht im aktuellen Namespace.
    - `-l „$TARGET_POD_LABEL_SELECTOR“`: Filtert die Pods nach einem bestimmten Label. `Sie müssen den Wert für TARGET_POD_LABEL_SELECTOR anpassen`, um Ihre Ziel-Anwendung zu identifizieren (z.B. `app.kubernetes.io/name=paperless-ngx`).
    - `–field-selector=status.phase=Running`: Stellt sicher, dass nur `laufende` Pods berücksichtigt werden.
    - `-o jsonpath='{.items[0].metadata.name}`: Extrahiert den Namen des `ersten` gefundenen Pods aus der JSON-Ausgabe.
  - `Fehlerbehandlung`: Das Skript prüft, ob ein Pod-Name gefunden wurde. Wenn nicht, wird eine Fehlermeldung ausgegeben und das Skript mit `exit 1` beendet.
  - *`kubectl exec –namespace „$KUBERNETES_NAMESPACE“ „$TARGET_POD_NAME“ -c „$TARGET_CONTAINER_NAME“ – /bin/sh -c „document_exporter ../export -z –delete“`*: Sobald der Pod-Name ermittelt wurde, wird der eigentliche

Befehl (`document_exporter ../export -z –delete`) im Ziel-Pod ausgeführt.
- `TARGET_CONTAINER_NAME`: Optional können Sie auch den Namen des spezifischen Containers im Pod angeben, falls der Pod mehrere Container hat und der Befehl in einem bestimmten Container ausgeführt werden muss. Wenn nicht angegeben, versucht `kubectl exec` den Befehl im ersten Container des Pods auszuführen
- *`TARGET_POD_LABEL_SELECTOR`*: Diesen Wert `müssen Sie anpassen`, um das spezifische Label Ihrer Anwendung zu verwenden. Zum Beispiel: `value: „app=my-web-app“` oder `value: „app.kubernetes.io/name=my-application“`.
- *`KUBERNETES_NAMESPACE`*: Dieser Wert wird automatisch über die `Downward API` aus den Metadaten des Jobs bezogen, was die Konfiguration flexibler macht.

Anpassung und Bereitstellung

Bevor Sie dieses Manifest anwenden, stellen Sie sicher, dass Sie die folgenden Platzhalter in den Kommentaren (`# ←- Anpassen`) und den env-Variablen aktualisieren:

- *`namespace`*: Ersetzen Sie `paperless-production` durch den tatsächlichen Namespace Ihrer Anwendung.
- *`TARGET_POD_LABEL_SELECTOR`*: Dies ist das wichtigste Element. Ersetzen Sie `app.kubernetes.io/name=paperless-ngx` durch den Label-Selector, der Ihren Ziel-Pod eindeutig identifiziert. Sie können *kubectl get pods -n <your-namespace> –show-labels* verwenden, um die Labels Ihrer Pods zu überprüfen.
- *`TARGET_CONTAINER_NAME`* (optional): Wenn Ihr Ziel-Pod mehrere Container hat und Sie den Befehl in einem bestimmten Container ausführen müssen, uncommenten Sie diese Zeile und geben Sie den Containernamen an.

Sobald Sie die Anpassungen vorgenommen haben, können Sie das Manifest mit *kubectl apply -f your-cronjob-manifest.yaml* in Ihrem Kubernetes-Cluster bereitstellen.

Fazit

Die Verwendung der dynamischen Pod-Ermittlung in Kubernetes CronJobs ist eine elegante Lösung, um Ihre geplanten Aufgaben widerstandsfähiger gegen Änderungen in Ihrer Pod-Infrastruktur zu machen. Es reduziert den Wartungsaufwand und sorgt dafür, dass Ihre Automatisierung auch in dynamischen Umgebungen zuverlässig funktioniert.

2025/06/20 16:54 · marko

CloudNativePG 1.26 - PostgreSQL In-Place Major Upgrades

CloudNativePG 1.26 bringt eines der am meisten ersehnten Features: **deklarative In-Place-Major-Upgrades** für PostgreSQL unter Verwendung von `pg_upgrade`. Dieser neue Ansatz ermöglicht es, PostgreSQL-Cluster durch das Ändern des `imageName` in ihrer Konfiguration zu aktualisieren—genauso wie bei einem Minor-Version-Update. Obwohl eine kurze Downtime erforderlich ist, wird der betriebliche Aufwand erheblich gesenkt, was es zur idealen Lösung für die Verwaltung `großer Flotten von PostgreSQL-Datenbanken` in Kubernetes macht. In diesem Artikel werde ich die Funktionsweise untersuchen, die Vorteile und Einschränkungen darstellen und das Upgrade einer 2,2-TB-Datenbank erläutern.

CloudNativePG 1.26, welches Ende März released wurde, bringt eines der am meisten herbeigesehnten Features in der Geschichte des Projekts: In-Place-Upgrades der Hauptversion von PostgreSQL mit `pg_upgrade`.

Im Unterschied zu kleineren Upgrades, die hauptsächlich Patches einspielen, erfordern Hauptversion-Upgrades das Anpassen an Änderungen im internen Speicherformat, die durch die neue PostgreSQL-Version eingeführt wurden.

Dieses Feature steht nun in der Version 1.26.0 zur Erprobung bereit.

Eine Übersicht über die bestehenden Methoden

CloudNativePG bietet nun drei deklarative (ja, deklarative!) Upgrade-Methoden für Hauptversionen an. Zwei dieser Methoden erfordern die Einrichtung eines neuen Clusters und sind als **Blue/Green-Deployment-Strategien** bekannt.

Der erste Ansatz verwendet die `Importfunktion` von `pg_dump` und `pg_restore`. Dies ist für kleine Datenbanken praktisch und eignet sich gut zum Testen neuer Versionen, erfordert jedoch für den endgültigen Wechsel eine Ausfallzeit, weshalb es sich um ein Offline-Upgrade handelt.

Die zweite Methode nutzt die native logische Replikation von PostgreSQL, um Upgrades ohne Ausfallzeiten zu ermöglichen—also ein Online-Upgrade—unabhängig von der Größe der Datenbank. Diese Methode ist meine bevorzugte Vorgehensweise für Upgrades geschäftskritischer PostgreSQL-Datenbanken. Sie kann auch für die Migration aus externen Umgebungen nach Kubernetes eingesetzt werden (z.B. von Amazon RDS zu CloudNativePG).

Die dritte Methode, die im Fokus dieses Artikels steht, sind Offline-In-Place-Upgrades unter Verwendung von `pg_upgrade`, dem offiziellen Werkzeug von PostgreSQL für solche Aufgaben.

Der Anwendungsfall für In-Place-Hauptversion-Upgrades

Der Hauptgrund für die Implementierung dieser Funktion in Kubernetes besteht darin, den betrieblichen Unterschied zwischen kleinen und großen PostgreSQL-Upgrades für GitOps-Nutzer zu eliminieren. Mit diesem Ansatz erfordert das Upgrade lediglich das Anpassen der Clusterkonfiguration und das Aktualisieren des Images für alle Clusterkomponenten (primäre und Standby-Server). Das ist besonders auf großer Ebene nützlich—wenn du dutzende oder sogar hunderte PostgreSQL-Cluster in einem einzigen Kubernetes-Cluster verwaltest—wo Blue/Green-Upgrades betriebliche Herausforderungen mit sich bringen können.

Bevor du startest

In-Place-Hauptversion-Upgrades stehen in CloudNativePG 1.26.0 aktuell zur Anwendung bereit. Diese Funktion kann in jedem Kubernetes-Cluster getestet werden.

Um CloudNativePG 1.26.0 zu installieren:

```
kubectl apply --server-side -f
https://raw.githubusercontent.com/cloudnative-pg/cloudnative-
pg/main/releases/cnpg-1.26.0.yaml
```

Wie es funktioniert

CloudNativePG erlaubt dir, das PostgreSQL-Operand-Image auf zwei Arten anzugeben:

- mit der Option `.spec.imageName`
- mit Image-Katalogen (`ImageCatalog`- und `ClusterImageCatalog`-Ressourcen)

Dieser Artikel konzentriert sich auf die `imageName`-Methode, obwohl dieselben Prinzipien auch bei der Image-Katalog-Methode gelten.

Nehmen wir an, du hast ein PostgreSQL-Cluster, mit:

```
imageName: ghcr.io/cloudnative-pg/postgresql:13.20-minimal-bullseye
```

Dies bedeutet, dass dein Cluster das neueste verfügbare Container-Image für PostgreSQL 13 (Nebenversion 20) verwendet. Da PostgreSQL 13 im November dieses Jahres das Lebensende erreicht, entscheidest du dich für ein Upgrade auf PostgreSQL 17 mit dem Image `ghcr.io/cloudnative-pg/postgresql:17.4-minimal-bullseye`.

Durch die Aktualisierung des `imageName`-Feldes in der Cluster-Konfiguration initiiert CloudNativePG automatisch ein Upgrade auf eine neue Hauptversion.

Der Upgrade-Prozess

Der erste Schritt besteht darin, das PostgreSQL-Cluster sicher herunterzufahren, um vor dem Upgrade die Datenkonsistenz zu gewährleisten. Dies ist ein Offline-Vorgang, der einen Ausfall bedeutet, aber es ermöglicht, statische Datendateien mit voller Integrität zu modifizieren.

CloudNativePG aktualisiert dann den Status der Cluster-Ressource (kind: cluster), um das derzeit laufende Image vor dem Beginn des Upgrades zu protokollieren. Dies ist wesentlich für einen Rollback im Falle eines Fehlers (wird später im Artikel noch intensiver behandelt).

Danach startet CloudNativePG einen Kubernetes-Job, der für die Vorbereitung der PostgreSQL-Datendateien auf den Persistent Volume Claims (PVC) für die neue Hauptversion mit `pg_upgrade` zuständig ist:

- Der Job erstellt eine temporäre Kopie der alten PostgreSQL-Binärdateien.
- Er initialisiert ein neues PGDATA-Verzeichnis mit initdb für die Zielversion von PostgreSQL.
- Er überprüft das Upgrade-Erfordernis, indem er die auf der Festplatte gespeicherten PostgreSQL-Versionen vergleicht, um ungewollte Upgrades basierend auf Image-Tags zu verhindern.
- Er ordnet die WAL- und Tablespace-Volumes bei Bedarf automatisch neu zu.

An diesem Punkt wird der tatsächliche Upgrade-Prozess mit `pg_upgrade` und der `–link`-Option ausgeführt, um Hardlinks zu nutzen, was die Datenmigration erheblich beschleunigt und den Speicherplatzbedarf sowie die Festplatten-E/A minimiert.

Wenn das Upgrade erfolgreich abgeschlossen wird, ersetzt CloudNativePG die ursprünglichen PostgreSQL-Datenverzeichnisse durch die aktualisierten Versionen, zerstört die Persistent Volume Claims der Replikate und startet das Cluster neu.

Tritt jedoch ein Fehler bei pg_upgrade auf, musst du manuell zur vorherigen Hauptversion von PostgreSQL zurückkehren, indem du die Cluster-Spezifikation aktualisierst und den Upgrade-Job löschst. Wie bei jedem In-place-Upgrade besteht immer das Risiko eines Scheiterns. Um dies zu minimieren, ist es entscheidend, kontinuierliche Basissicherungen zu pflegen. Falls deine StorageClass Volumes-Snapshots unterstützt, solltest du erwägen einen solchen vor dem Upgrade zu erstellen, es ist eine einfache Vorsichtsmaßnahme die dich vor unerwarteten Problemen bewahren könnte.

Insgesamt verbessert dieser effiziente Ansatz die Effizienz und Zuverlässigkeit von In-place-Upgrades auf eine neue Hauptversion und macht PostgreSQL-Versionsübergänge in Kubernetes-Umgebungen handhabbarer.

Beispiel

Der beste Weg, dieses Feature zu verstehen, ist, es in der Praxis zu testen. Beginnen wir mit einem einfachen PostgreSQL-13-Cluster namens pg, definiert in der folgenden `pg.yaml`:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:13.20-minimal-bullseye
```

```
  instances: 3

  storage:
    size: 1Gi
  walStorage:
    size: 1Gi
```

Nach dem Erstellen des Clusters überprüfe seinen Status mit:

```
kubectl cnpg status pg
```

Du kannst die Version auch mit `psql` kontrollieren:

```
kubectl cnpg psql pg -- -qAt -c 'SELECT version()'
```

Es sollte eine ähnliche Ausgabe erscheinen:

```
PostgreSQL 13.20 (Debian 13.20-1.pgdg110+1) on x86_64-pc-linux-gnu, compiled
by gcc (Debian 10.2.1-6) 10.2.1 20210110, 64-bit
```

Lassen wir uns nun von PostgreSQL 13, das bald sein Lebensende erreicht, auf die neueste Minor-Version der aktuellsten Hauptversion upgraden. Dafür brauchst du lediglich das Feld `imageName` in deiner Konfiguration zu aktualisieren:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.4-minimal-bullseye
  instances: 3

  storage:
    size: 1Gi
  walStorage:
    size: 1Gi
```

Wende die Änderungen an, um den Major-Upgrade-Prozess zu starten:

```
kubectl apply -f pg.yaml
```

Sobald der Prozess abgeschlossen ist, überprüfe das Upgrade, indem du den Clusterstatus erneut kontrollierst. Deine Datenbank sollte nun unter PostgreSQL 17 laufen.

Wenn du die Version erneut prüfst, sollte eine ähnliche Ausgabe erscheinen:

```
PostgreSQL 17.4 (Debian 17.4-1.pgdg110+2) on x86_64-pc-linux-gnu, compiled
by gcc (Debian 10.2.1-6) 10.2.1 20210110, 64-bit
```

Wenn du jetzt `kubectl get pods` eingibst, wirst du feststellen, dass die Pods und PVCs mit den

Namen pg-2 und pg-3 nicht mehr vorhanden sind, da sie durch Sequenznummern 4 und 5 ersetzt wurden.

```
NAME   READY   STATUS    RESTARTS   AGE
pg-1   1/1     Running   0          62s
pg-4   1/1     Running   0          36s
pg-5   1/1     Running   0          15s
```

Einschränkungen und Vorbehalte

Wie du gerade gelernt hast, erfordert diese Implementierung, dass Replikate (replicas) neu erstellt werden, was derzeit nur mit pg_basebackup unterstützt wird, dies beeinträchtigt jedoch den Datenbankzugriff nicht. Wenn der primäre Knoten ausfällt, musst du bis zur Verfügbarkeit eines neuen Replikats aus dem neuesten Backup wiederherstellen. In den meisten Fällen stammt dieses Backup von der vorherigen PostgreSQL-Version, was bedeutet, dass du den Major-Upgrade-Prozess wiederholen musst.

Obwohl dieses Szenario unwahrscheinlich ist, ist es wichtig, das Risiko zu erkennen. In der Regel wird die Replikation innerhalb von Minuten abgeschlossen, je nach Komplexität der Datenbank (insbesondere der Anzahl der Tabellen).

Bei deutlich größeren Datenbanken sei darauf hingewiesen, dass der Cluster bis zur vollständigen Wiederherstellung der Replikation in einem eingeschränkten Zustand für hohe Verfügbarkeit verbleibt. Um das Risiko zu minimieren, empfehle ich dringend, so bald wie möglich nach Abschluss des Upgrades ein physisches Backup zu erstellen.

Ein weiterer wesentlicher Aspekt sind die Erweiterungen (extensions). Sie spielen eine entscheidende Rolle beim Upgrade-Prozess. Achte darauf, dass alle erforderlichen Erweiterunge, und deren jeweilige Versionen, im Operand-Image der Ziel PostgreSQL Version verfügbar sind. Sind Erweiterungen nicht vorhanden, wird das Upgrade fehlschlagen. Überprüfe deshalb immer die Kompatibilität der Erweiterungen im Vorfeld.

Test eines großen Datenbank-Upgrades

Um zu evaluieren, wie ein Major-Upgrade von PostgreSQL mit einer großen Datenbank umgeht, habe ich ins Rahmen meiner Tests eine 2,2 TB große PostgreSQL 16-Datenbank mit pgbench und einem Skalierungsfaktor von 150.000 erstellt. Hier ein Auszug aus dem cnpg status Befehl:

```
Cluster Summary
Name                 default/pg
System ID:           7487705689911701534
PostgreSQL Image:    ghcr.io/cloudnative-pg/postgresql:16
Primary instance:    pg-1
Primary start time:  2025-03-30 20:42:26 +0000 UTC (uptime 72h32m31s)
Status:              Cluster in healthy state
Instances:           1
Ready instances:     1
Size:                2.2T
```

```
Current Write LSN:   1D0/8000000 (Timeline: 1 - WAL File:
00000001000001D000000001)
<snip>
```

Ich habe dann ein Upgrade auf **PostgreSQL 17** angestoßen, welches in lediglich **33 Sekunden** abgeschlossen war, wobei der Cluster in weniger als einer Minute komplett einsatzbereit war. Hier die aktualisierte `cnpg status` Ausgabe:

```
Cluster Summary
Name                 default/pg
System ID:           7488830276033003555
PostgreSQL Image:    ghcr.io/cloudnative-pg/postgresql:17
Primary instance:    pg-1
Primary start time:  2025-03-30 20:42:26 +0000 UTC (uptime 72h44m45s)
Status:              Cluster in healthy state
Instances:           1
Ready instances:     1
Size:                2.2T
Current Write LSN:   1D0/F404F9E0 (Timeline: 1 - WAL File:
00000001000001D00000003D)
```

Da CloudNativePG die `–link` Option von PostgreSQL verwendet (die auf Hardlinks basiert), **hängt die Upgrade-Dauer vor allem von der Anzahl der Tabellen und nicht von der Größe der Datenbank ab**.

Schlussfolgerungen

In-place Major-Upgrades mit `pg_upgrade` bringen den herkömmlichen Upgrade-Pfad von PostgreSQL in Kubernetes, wodurch Nutzern eine deklarative Methode geboten wird, um mit minimalem Betriebsaufwand zwischen Hauptversionen zu wechseln. Diese Methode beinhaltet zwar Ausfallzeiten, eliminiert aber die Notwendigkeit von Blue/Green-Clustern und eignet sich somit besonders für Umgebungen, **die eine große Vielzahl kleiner bis mittelgroßer PostgreSQL-Instanzen verwalten**.

Bei einem erfolgreichen Upgrade erhältst du einen voll funktionsfähigen PostgreSQL-Cluster, wie wenn du `pg_upgrade` auf einer traditionellen VM oder einem physischen Server ausgeführt hättest. Bei einem Fehlschlag stehen Rollback-Optionen zur Verfügung—inklusive der Rückkehr zum ursprünglichen Manifest und Löschen des Upgrade-Jobs. Kontinuierliche Backups bieten ein zusätzliches Sicherheitsnetz.

Auch wenn In-place-Upgrades möglicherweise nicht meine bevorzugte Methode für geschäftskritische Datenbanken sind, stellen sie eine bedeutende Option für Teams dar, **die Betriebseinfachheit und Skalierbarkeit** über Null-Ausfallzeit-Upgrades priorisieren. Wie in den Tests aufgezeigt, wird die Upgrade-Dauer in erster Linie durch die Anzahl der Tabellen und nicht durch das Datenbankvolumen bestimmt, was diesen Ansatz auch für große Datensätze effizient macht.

2025/06/04 09:09 · marko

Ältere Einträge >>