

PostgreSQL mit Zalando Operator in Kubernetes bereitstellen

In dieser Anleitung erfahren Sie, wie Sie mit dem [Zalando Postgres-Operator](#) Postgres-Cluster in bereitstellen.

[PostgreSQL](#) ist ein leistungsstarkes, objektrelationales Open-Source-Datenbanksystem, das über mehrere Jahrzehnte hinweg aktiv entwickelt wurde und sich einen guten Ruf für Zuverlässigkeit, Robustheit von Features und Leistung verdient hat.

Vorteile

Zalando bietet folgende Vorteile:

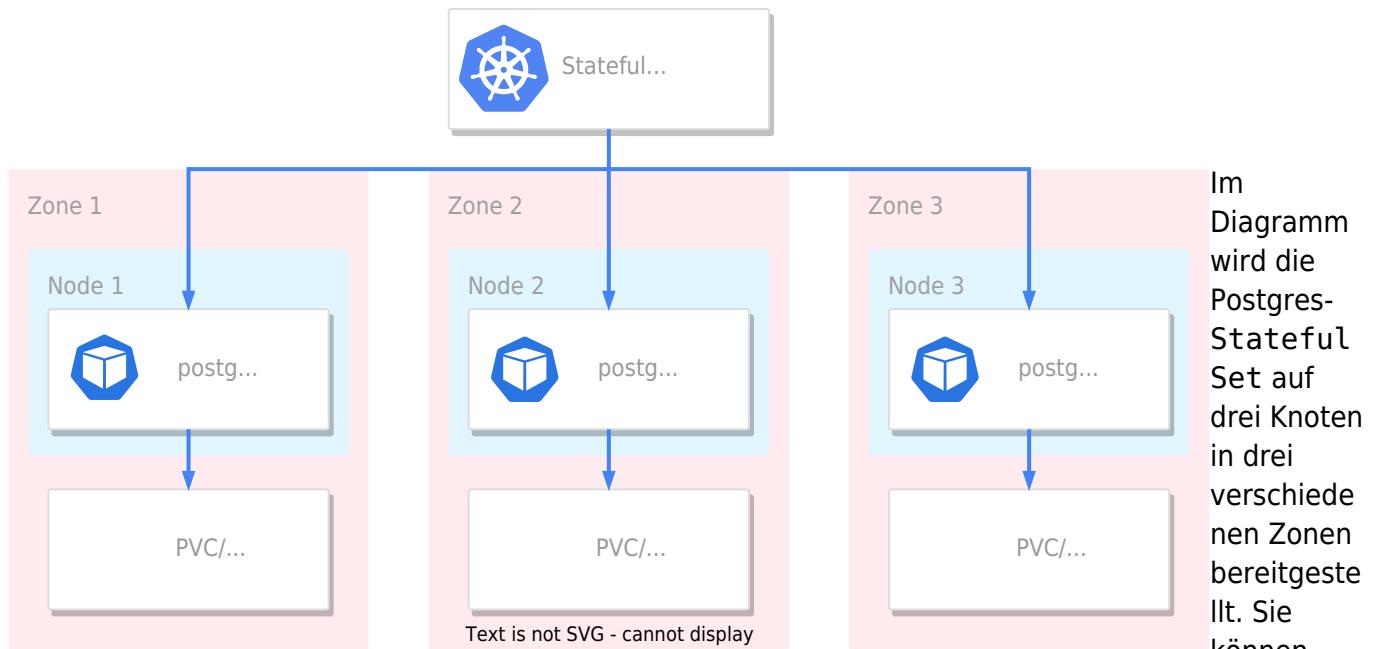
- Eine deklarative und Kubernetes-native Methode zum Verwalten und Konfigurieren der PostgreSQL-Cluster
- Hochverfügbarkeit von [Patroni](#)
- Unterstützung der Sicherungsverwaltung mithilfe von [Cloud Storage-Buckets](#)
- Rolling Updates für Postgres-Clusteränderungen, einschließlich schneller Nebenversionsaktualisierungen
- Deklarative [Nutzerverwaltung](#) mit Passworterstellung und -rotation mithilfe benutzerdefinierter Ressourcen
- Unterstützung für [TLS](#), Zertifikatsrotation und [Verbindungs pools](#)
- [Clusterklonen](#) und Datenreplikation

Bereitstellungsarchitektur

In dieser Anleitung verwenden Sie den Zalando Postgres-Operator, um einen hochverfügbaren Postgres-Cluster bereitzustellen und zu konfigurieren. Der Cluster hat ein Leader-Replikat und zwei schreibgeschützte Standby-Replikate, die von [Patroni](#) verwaltet werden. Patroni ist eine von Zalando verwaltete Open-Source-Lösung, um Postgres Hochverfügbarkeits- und automatische Failover-Funktionen zu bieten. Wenn ein Leader ausfällt, wird ein Standby-Replikat automatisch zu einer Leader-Rolle hochgestuft.

Außerdem stellen Sie einen hochverfügbaren Kubernetes-Cluster für Postgres bereit, wobei mehrere Kubernetes-Knoten über verschiedene Verfügbarkeitszonen verteilt sind. Diese Konfiguration sorgt für Fehlertoleranz, Skalierbarkeit und geografische Redundanz. Damit können Rolling Updates und Wartungen durchgeführt werden, während SLAs für Verfügbarkeit und Verfügbarkeit bereitgestellt werden.

Das folgende Diagramm zeigt einen Postgres-Cluster, der auf mehreren Knoten und Zonen in einem Kubernetes-Cluster ausgeführt wird:



steuern, wie Kubernetes auf Knoten bereitgestellt wird. Dazu legen Sie den erforderlichen Pod-Regeln für [Affinität und Anti-Affinität](#) für die [postgresql](#) benutzerdefinierte Ressourcenspezifikation. Wenn eine Zone gemäß der empfohlenen Konfiguration ausfällt, verschiebt Kubernetes die Pods auf andere verfügbare Knoten in Ihrem Cluster. Zum Speichern von Daten verwenden Sie SSD-Laufwerke (premium-rwo StorageClass). Diese werden in den meisten Fällen für stark ausgelastete Datenbanken aufgrund ihrer niedrigen Latenz und hohen IOPS empfohlen.

Zalando-Operator in Ihrem Cluster bereitstellen

Stellen Sie den Zalando-Operator mithilfe eines Helm-Diagramms in Ihrem Kubernetes-Cluster bereit.

1. Fügen Sie das Helm-Diagramm-Repository des Zalando-Operators hinzu:

```
helm repo add postgres-operator-charts
https://opensource.zalando.com/postgres-operator/charts/postgres-operator
```

2. Erstellen Sie einen Namespace für den Zalando-Operator und den Postgres-Cluster:

```
kubectl create ns postgres
kubectl create ns zalando
```

3. Stellen Sie den Zalando-Operator mit dem Helm-Befehlszeilentool bereit:

```
helm install postgres-operator postgres-operator-charts/postgres-operator -n zalando \
  --set configKubernetes.enable_pod_antiaffinity=true \
  --set configKubernetes.pod_antiaffinity_preferred_during_scheduling=true \
  --set
```

```
configKubernetes.pod_antiaffinity_topology_key="topology.kubernetes.io/
zone" \
--set configKubernetes.spilo_fsgroup="103"
```

Sie können die podAntiAffinity-Einstellungen nicht direkt für die benutzerdefinierte Ressource konfigurieren, die den Postgres-Cluster darstellt. Legen Sie stattdessen global die podAntiAffinity-Einstellungen für alle Postgres-Cluster in den Operatoreinstellungen fest.

- Prüfen Sie den Bereitstellungsstatus des Zalando-Operators mit Helm:

```
helm ls -n zalando
```

Die Ausgabe sieht in etwa so aus:

NAME	STATUS	CHART	NAMESPACE	REVISION	APP VERSION	UPDATED
postgres-operator	postgres-operator-0200 CEST	zalando	zalando	1	2023-10-13 16:04:13.945614	1.10.1

Postgres bereitstellen

Die grundlegende Konfiguration für die Postgres-Clusterinstanz umfasst die folgenden Komponenten:

- Drei Postgres-Replikate: ein Leader und zwei Standby-Replikate.
- CPU-Ressourcenzuweisung: eine CPU-Anfrage und zwei CPU-Limits mit 4 GB Speicheranfragen und -Limits.
- Toleranzen, nodeAffinities und topologySpreadConstraints, die für jede Arbeitslast konfiguriert sind und eine ordnungsgemäße Verteilung auf Kubernetes-Knoten mithilfe ihrer jeweiligen Knotenpools und verschiedenen Verfügbarkeitszonen gewährleisten.

Diese Konfiguration stellt die minimale Einrichtung dar, die zum Erstellen eines produktionsfertigen Postgres-Clusters erforderlich ist.

Das folgende Manifest beschreibt einen Postgres-Cluster: databases/postgres-zalando/manifests/01-basic-cluster/my-cluster.yaml

```
apiVersion: "acid.zalan.do/v1"
kind: postgresql
metadata:
  name: my-cluster
spec:
  dockerImage: ghcr.io/zalando/spilo-15:3.0-p1
  teamId: "my-team"
  numberInstances: 3
  users:
    mydatabaseowner:
    - superuser
    - createdb
    myuser: []
```

```
databases:
  mydatabase: mydatabaseowner
postgresql:
  version: "15"
parameters:
  shared_buffers: "32MB"
  max_connections: "10"
  log_statement: "all"
  password_encryption: scram-sha-256
volume:
  size: 5Gi
  storageClass: premium-rwo
enableShmVolume: true
podAnnotations:
  cluster-autoscaler.kubernetes.io/safe-to-evict: "true"
tolerations:
- key: "app.stateful/component"
  operator: "Equal"
  value: "postgres-operator"
  effect: NoSchedule
nodeAffinity:
  preferredDuringSchedulingIgnoredDuringExecution:
- weight: 1
  preference:
    matchExpressions:
    - key: "app.stateful/component"
      operator: In
      values:
      - "postgres-operator"
resources:
  requests:
    cpu: "1"
    memory: 4Gi
  limits:
    cpu: "2"
    memory: 4Gi
sidecars:
- name: exporter
  image: quay.io/prometheuscommunity/postgres-exporter:v0.14.0
  args:
  - --collector.stat_statements
  ports:
  - name: exporter
    containerPort: 9187
    protocol: TCP
  resources:
    limits:
      cpu: 500m
      memory: 256M
```

```

  requests:
    cpu: 100m
    memory: 256M
  env:
    - name: "DATA_SOURCE_URI"
      value: "localhost/postgres?sslmode=require"
    - name: "DATA_SOURCE_USER"
      value: "${POSTGRES_USER}"
    - name: "DATA_SOURCE_PASS"
      value: "${POSTGRES_PASSWORD}"

```

Dieses Manifest hat die folgenden Felder:

- `spec.teamId` ist ein Präfix für die von Ihnen ausgewählten Clusterobjekte
- `spec.number0fInstances` ist die Gesamtzahl der Instanzen für einen Cluster
- `spec.users` ist die Nutzerliste mit **Berechtigungen**
- `spec.databases` ist die Datenbankliste im Format `dbname: ownername`
- `spec.postgresql` sind die Postgres-Parameter
- `spec.volume` sind die Parameter für den nichtflüchtigen Speicher
- `spec.tolerations` ist die Toleranz-Pod-Vorlage, mit der Cluster-Pods auf pool-postgres-Knoten geplant werden können
- `spec.nodeAffinity` ist die Pod-Vorlage `nodeAffinity`, die Kubernetes mitteilt, dass Cluster-Pods lieber auf pool-postgres-Knoten geplant werden sollen.
- `spec.resources` sind Anfragen und Limits für Cluster-Pods
- `spec.sidecars` ist eine Liste der Sidecar-Container, die `postgres-exporter` enthält

Weitere Informationen finden Sie in der Postgres-Dokumentation unter [Referenz zu Clustermanifesten](#).

Einfachen Postgres-Cluster erstellen

1. Erstellen Sie mithilfe der grundlegenden Konfiguration einen neuen Postgres-Cluster:

```
kubectl apply -n postgres -f manifests/01-basic-cluster/my-cluster.yaml
```

Mit diesem Befehl wird eine benutzerdefinierte PostgreSQL-Ressource des Zalando-Operators erstellt:

- * CPU- und Speicheranforderungen und -limits
- * Markierungen und Affinitäten zum Verteilen der bereitgestellten Pod-Replikate auf Kubernetes-Knoten.
- * Eine Datenbank
- * Zwei Nutzer mit Datenbankinhaberberechtigungen
- * Ein Nutzer ohne Berechtigungen

2. Warten Sie, bis Kubernetes die erforderlichen Arbeitslasten gestartet hat:

```
kubectl wait pods -l cluster-name=my-cluster --for condition=Ready --timeout=300s -n postgres
```

Die Verarbeitung dieses Befehls kann einige Minuten dauern.

3. Prüfen Sie, ob Kubernetes die Postgres-Arbeitslasten erstellt hat:

```
kubectl get pod,svc,statefulset,deploy,pdb,secret -n postgres
```

Die Ausgabe sieht in etwa so aus:

NAME	READY	STATUS	RESTARTS	AGE
pod/my-cluster-0	1/1	Running	0	6m41s
pod/my-cluster-1	1/1	Running	0	5m56s
pod/my-cluster-2	1/1	Running	0	5m16s
pod/postgres-operator-db9667d4d-rgcs8	1/1	Running	0	12m
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	
PORT(S)	AGE			
service/my-cluster	ClusterIP	10.52.12.109	<none>	5432/TCP 6m43s
service/my-cluster-config	ClusterIP	None	<none>	<none> 5m55s
service/my-cluster-repl	ClusterIP	10.52.6.152	<none>	5432/TCP 6m43s
service/postgres-operator	ClusterIP	10.52.8.176	<none>	8080/TCP 12m
NAME	READY	AGE		
statefulset.apps/my-cluster	3/3	6m43s		
NAME	READY	UP-TO-DATE	AVAILABLE	
AGE				
deployment.apps/postgres-operator	1/1	1	1	12m
NAME	MIN	AVAILABLE	MAX	
UNAVAILABLE	ALLOWED	DISRUPTIONS	AGE	
poddisruptionbudget.policy/postgres-my-cluster-pdb	1		N/A	0 6m44s
NAME	TYPE			
DATA	AGE			
secret/my-user.my-cluster.credentials.postgresql.acid.zalan.do	Opaque			2 6m45s
secret/postgres.my-cluster.credentials.postgresql.acid.zalan.do	Opaque	2		6m44s
secret/sh.helm.release.v1.postgres-operator.v1				helm.sh/release.v1 1 12m
secret/standby.my-cluster.credentials.postgresql.acid.zalan.do	Opaque			2 6m44s
secret/zalando.my-cluster.credentials.postgresql.acid.zalan.do	Opaque			

2 6m44s

Der Operator erstellt die folgenden Ressourcen:

- Ein Postgres-StatefulSet, das drei Pod-Replikate für Postgres steuert
- Einen PodDisruptionBudgets, wodurch mindestens ein verfügbares Replikat garantiert wird
- Den `my-cluster`-Dienst, der nur auf das Leader-Replikat ausgerichtet ist
- Den `my-cluster-repl`-Dienst, der den Postgres-Port für eingehende Verbindungen und für die Replikation zwischen Postgres-Replikaten verfügbar macht
- Den monitorlosen Dienst `my-cluster-config` zum Abrufen der Liste der laufenden Postgres-Pod-Replikate
- Secrets mit Nutzeranmeldedaten für den Zugriff auf die Datenbank und die Replikation zwischen Postgres-Knoten

Bei Postgres authentifizieren

Sie können Postgres-Nutzer erstellen und ihnen Datenbankberechtigungen zuweisen. Das folgende Manifest beschreibt beispielsweise eine benutzerdefinierte Ressource, die Nutzer und Rollen zuweist:

```
apiVersion: "acid.zalan.do/v1"
kind: postgresql
metadata:
  name: my-cluster
spec:
  ...
  users:
    mydatabaseowner:
      - superuser
      - createdb
    myuser: []
  databases:
    mydatabase: mydatabaseowner
```

In diesem Manifest:

- Der `mydatabaseowner`-Nutzer hat die Rollen `SUPERUSER` und `CREATEDB`, die uneingeschränkte Administratorrechte gewähren (Verwaltung der Postgres-Konfiguration, neue Datenbanken, Tabellen und Nutzer erstellen, usw.).
- Dem Nutzer `myuser` wurden keine Rollen zugewiesen. Dies folgt der `Best Practice` zur Verwendung von `SUPERUSER`, um Nutzer mit den geringsten Berechtigungen zu erstellen. Detaillierte Rechte werden `myuser` von `mydatabaseowner` gewährt. Aus Sicherheitsgründen sollten Sie `myuser`-Anmeldedaten nur für Clientanwendungen freigeben.

Passwörter speichern

Verwenden Sie die [empfohlene Methode zum Speichern von Passwörtern](#) `scram-sha-256`. Das folgende Manifest beschreibt beispielsweise eine benutzerdefinierte Ressource, die die `scram-`

sha-256-Verschlüsselung mit dem Feld `postgresql.parameters.password_encryption` angibt:

```
apiVersion: "acid.zalan.do/v1"
kind: postgresql
metadata:
  name: my-cluster
spec:
  ...
  postgresql:
    parameters:
      password_encryption: scram-sha-256
```

Nutzeranmeldedaten rotieren

Sie können [Nutzeranmeldedaten rotieren](#), die in Kubernetes-Secrets mit Zalando gespeichert sind. Das folgende Manifest beschreibt beispielsweise eine benutzerdefinierte Ressource, die die Rotation von Nutzeranmeldedaten mithilfe des Felds `usersWithSecretRotation` definiert:

```
apiVersion: "acid.zalan.do/v1"
kind: postgresql
metadata:
  name: my-cluster
spec:
  ...
  usersWithSecretRotation:
    - myuser
    - myanotheruser
    - ...
```

Authentifizierungsbeispiel: Verbindung zu Postgres herstellen

In diesem Abschnitt erfahren Sie, wie Sie einen Postgres-Beispielclient bereitstellen und mit dem Passwort aus einem Kubernetes-Secret eine Verbindung zur Datenbank herstellen.

1. Führen Sie den Client-Pod aus, um mit Ihrem Postgres-Cluster zu interagieren:

```
kubectl apply -n postgres -f manifests/02-auth/client-pod.yaml
```

Die Anmeldedaten der Nutzer `myuser` und `mydatabaseowner` werden aus den zugehörigen Secrets übernommen und als Umgebungsvariablen im Pod bereitgestellt.

2. Stellen Sie eine Verbindung zum Pod her, wenn er bereit ist:

```
kubectl wait pod postgres-client --for=condition=Ready --timeout=300s -
```

```
n postgres
kubectl exec -it postgres-client -n postgres -- /bin/bash
```

3. Stellen Sie eine Verbindung zu Postgres her und versuchen Sie, eine neue Tabelle mit den myuser-Anmelde- und -Berechtigungsdaten zu erstellen:

```
PGPASSWORD=$CLIENTPASSWORD psql \
-h my-cluster \
-U $CLIENTUSERNAME \
-d mydatabase \
-c "CREATE TABLE test (id serial PRIMARY KEY, randomdata VARCHAR ( 50 ) NOT NULL);"
```

Der Befehl sollte mit einem Fehler wie diesem fehlschlagen:

```
ERROR: permission denied for schema public
LINE 1: CREATE TABLE test (id serial PRIMARY KEY, randomdata VARCHAR...
```

Der Befehl schlägt fehl, da sich Nutzer ohne zugewiesene Berechtigungen standardmäßig nur bei Postgres anmelden und Datenbanken auflisten können.

4. Erstellen Sie eine Tabelle mit mydatabaseowner-Anmelde- und -Berechtigungsdaten und gewähren Sie myuser **alle** Berechtigungen für die Tabelle:

```
PGPASSWORD=$OWNERPASSWORD psql \
-h my-cluster \
-U $OWNERUSERNAME \
-d mydatabase \
-c "CREATE TABLE test (id serial PRIMARY KEY, randomdata VARCHAR ( 50 ) NOT NULL);GRANT ALL ON test TO myuser;GRANT ALL ON SEQUENCE test_id_seq TO myuser;"
```

Die Ausgabe sieht in etwa so aus:

```
CREATE TABLE
GRANT
GRANT
```

5. Fügen Sie zufällige Daten mit den Anmelde- und -Berechtigungsdaten myuser in die Tabelle ein:

```
for i in {1..10}; do
DATA=$(tr -dc A-Za-z0-9 </dev/urandom | head -c 13)
PGPASSWORD=$CLIENTPASSWORD psql \
-h my-cluster \
-U $CLIENTUSERNAME \
-d mydatabase \
-c "INSERT INTO test(randomdata) VALUES ('$DATA');"
done
```

Die Ausgabe sieht in etwa so aus:

```
INSERT 0 1
```

6. Rufen Sie die eingefügten Werte ab:

```
PGPASSWORD=$CLIENTPASSWORD psql \
-h my-cluster \
-U $CLIENTUSERNAME \
-d mydatabase \
-c "SELECT * FROM test;"
```

Die Ausgabe sieht in etwa so aus:

id	randomdata
1	jup9HYsAjwtW4
2	9rLAyBlcpLgNT
3	wcXSqxb5Yz75g
4	KoDRSrx3muD6T
5	b9atC7RPai7En
6	20d7kC8E6Vt1V
7	GmgNxaWbkevGq
8	BkTwFWH6hWC7r
9	nkLXHclkqaqkqy
10	HEebZ9Lp71Nm3
(10 rows)	

7. Beenden Sie die Pod-Shell:

```
exit
```

