Введение в Kubernetes: Руководство для PostgreSQL DBA

Дополнительный материал для курса PostgreSQL Advanced

Дмитрий Золотов

2026-02-07

1 Введение в Kubernetes: Руководство для PostgreSQL DBA

Дополнительный материал к курсу PostgreSQL Advanced Автор: Дмитрий Золотов

1.1 Оглавление

1.2 Зачем DBA изучать Kubernetes

1.2.1 Почему это важно именно сейчас

Вы — опытный DBA. Вы знаете PostgreSQL от и до: умеете настраивать репликацию, оптимизировать запросы, восстанавливать базы после сбоев, планировать обновления. Возможно, вы даже задаётесь вопросом: **“Зачем мне ещё один инструмент? Я и так справляюсь.”**

Ответ простой: **инфраструктура вокруг вас уже изменилась.** Разработчики перешли на микросервисы, CI/CD-конвейеры, контейнеры и Kubernetes. Базы данных — последний оплот “классической” инфраструктуры. Но эта граница стирается.

Факты индустрии (2024):

- **72% организаций** управляют базами данных через Kubernetes (Data on Kubernetes Report 2024)
- **80% компаний** используют Kubernetes в production (CNCF Survey 2024)
- **90% технических лидеров** считают K8s готовым для stateful workloads

Вопрос больше не в том, **“стоит ли?”**, а в том, **“как правильно?”**

1.2.2 Что изменится в вашей работе

Было:

- Новое окружение PostgreSQL для тестирования: 2-4 часа ручной работы (поднять сервер, настроить сеть, установить PostgreSQL, настроить конфиги, настроить резервное копирование)
- Ночной failover: звонок в 3 AM, 1-2 часа работы, бессонная ночь
- Добавить read-replica: полдня конфигурации (клонирование, настройка репликации, проверка синхронизации)
- Обновление PostgreSQL на staging: планирование на выходные, простой сервиса
- Воспроизвести production в dev: “почти невозможно” из-за различий в конфигурациях

Стало (с Kubernetes):

- Новое окружение: `kubectl apply -f pg-test.yaml` — 30 секунд
- Failover: автоматически через Patroni, downtime < 30 секунд, alert утром “был failover, всё в порядке”
- Read-replica: `replicas: 3` в манифесте, применить, готово
- Обновление: rolling update без даунтайма (один за другим с проверками)
- Dev = Production: идентичные манифесты гарантируют одинаковую конфигурацию

1.2.3 Реальные компании, которые это делают

Zalando (Германия):

- Сотни PostgreSQL-кластеров в Kubernetes
- 5+ лет в production без серьёзных инцидентов
- Открытый Postgres Operator — индустриальный стандарт

Postgres Professional + Flant (Deckhouse):

- Стратегическое партнерство, 3 млрд руб. инвестиций
- 170+ production кластеров PostgreSQL в Kubernetes
- Российский рынок активно переходит на эту модель

IBM / EnterpriseDB → CloudNativePG:

- 132+ млн загрузок оператора
- CNCF Sandbox проект
- Enterprise-поддержка для критичных систем

Crunchy Data → PGO (Postgres Operator):

- Клиенты: IBM, SAS, Danfoss
- Сертифицирован для работы с OpenShift
- Автоматизация всего жизненного цикла PostgreSQL

1.2.4 Мотивация через боль

Типичные проблемы DBA, которые решает Kubernetes:

Проблема 1: “Снежинки” (snowflake servers)

У каждого сервера PostgreSQL — своя история. Кто-то когда-то что-то руками подкрутил. Конфиги различаются. Воспроизвести окружение — квест.

Решение K8s: Декларативные манифесты. Одна спецификация → одинаковые сервера. Версионирование через Git. Воспроизводимость гарантирована.

Проблема 2: Ночные инциденты

Сервер упал в 3 AM. Телефон звонит. Вы подключаетесь по SSH (если VPN работает), диагностируете, вручную делаете promote реплики, обновляете DNS, проверяете. Уже утро.

Решение K8s: Patroni + Kubernetes Operator следят за состоянием. Автоматический failover. Service переключает трафик. Вы получаете alert утром: “был failover, всё работает”.

Проблема 3: Масштабирование

Нужна ещё одна read-replica. Клонируем сервер, настраиваем репликацию, прописываем в конфигах приложения, тестируем. Полдня работы.

Решение K8s: Меняете `replicas: 2` на `replicas: 3`, применяете манифест. Готово.

Проблема 4: Тестовые окружения

“Дайте мне копию production для теста нового индекса.” Неделя согласований, ресурсы, настройка. А индекс можно было проверить за час.

Решение K8s: `kubectl apply -f pg-test.yaml` с теми же манифестами, но другим namespace. Удалили после теста — ресурсы освободились.

1.2.5 Что нужно понимать

Kubernetes — это инструмент, не серебряная пуля.

Для DBA это как `pg_basebackup`: не обязательно знать, но без него в современном мире сложнее работать.

Цель этого материала:

- Дать базовое понимание архитектуры Kubernetes
- Показать, как работают основные объекты
- Объяснить сетевую модель (критична для баз данных!)
- Научить запускать PostgreSQL в K8s (первые шаги)
- Подготовить к работе с production-операторами

Не цель:

- Превратить вас в DevOps-инженера
- Убедить, что K8s подходит для всех сценариев
- Скрыть ограничения и сложности

Давайте честно обсудим, когда Kubernetes **не подходит** для PostgreSQL.

1.3 Честный разговор о Kubernetes для баз данных

1.3.1 Когда Kubernetes НЕ подходит для PostgreSQL

1. Производительность

Проблема: Сетевое хранилище (Persistent Volumes через сеть) даёт overhead ~2x по latency по сравнению с bare metal (Data on Kubernetes Report).

Когда критично:

- Ultra-low-latency OLTP системы (< 1 мс на запрос)
- Высоконагруженные биржевые системы
- Real-time системы с жёсткими SLA по latency

Решение: Используйте bare metal или VM с локальными NVMe дисками.

2. Сложность мажорных обновлений

Проблема: `pg_upgrade` в Kubernetes сложнее, чем на bare metal. Нужно:

- Остановить оператор
- Создать резервную копию
- Вручную выполнить upgrade контейнера
- Обновить манифесты
- Проверить совместимость расширений

На bare metal: Останавливаете PostgreSQL, запускаете `pg_upgrade`, запускаете обратно.

Когда критично: Частые мажорные обновления (например, PostgreSQL 14 → 15 → 16 за год).

3. Требования к экспертизе

Реальность: Нужна экспертиза **и** в Kubernetes, **и** в PostgreSQL. Один человек редко владеет обоими на высоком уровне.

Когда критично:

- Малая команда (1-2 DBA без DevOps-поддержки)
- Нет времени на обучение
- Критичные системы без запаса на ошибки

4. Риск split-brain при двух нодах

Проблема: Kubernetes требует кворум для принятия решений. Двухнодная конфигурация PostgreSQL не может образовать кворум при сбое одной ноды.

Минимум: 3 ноды для production (Primary + 2 Replica).

Когда критично: Бюджет позволяет только 2 сервера.

5. Ожидания vs реальность

Миф: “Kubernetes превратит PostgreSQL в Amazon Aurora с бесконечным масштабированием!”

Реальность: Kubernetes — это оркестратор. Он автоматизирует управление, но не меняет архитектуру PostgreSQL. Запись всё равно идёт на Primary, реплики всё равно асинхронные.

1.3.2 Когда Kubernetes ПОДХОДИТ для PostgreSQL

✅ Подходит	❌ Не подходит
Много однотипных PostgreSQL-инстансов (> 5)	Единственная высоконагруженная OLTP-база
Нужна унификация dev/test/staging/prod	Требования к минимальной latency (< 1ms)
Команда уже использует K8s для приложений	Нет экспертизы ни в K8s, ни в контейнерах
Нужны автоматический failover и self-healing	Legacy-инфраструктура без планов контейнеризации
Микросервисная архитектура	Монолит + одна база на dedicated-сервере
Потребность в быстром создании окружений	Критичная система без толерантности к экспериментам

1.3.3 Гибридный подход

Рекомендация: Не обязательно переносить всё сразу.

Стратегия:

1. **Production critical OLTP** → оставить на bare metal / VM
2. **Аналитические реплики, staging, dev** → перенести в Kubernetes
3. **Новые микросервисы** → изначально в Kubernetes
4. **Постепенно:** накопить опыт, потом мигрировать критичные системы

1.3.4 Вывод

Kubernetes для PostgreSQL — это **выбор**, а не **императив**.

Вы должны принять **осознанное решение**, взвесив:

- Текущую инфраструктуру
- Компетенции команды
- Требования к производительности
- Бюджет и ресурсы

Этот материал даст вам базу для такого решения.

1.4 Архитектура Kubernetes глазами DBA

1.4.1 Аналогия с PostgreSQL

Kubernetes — сложная система. Но если вы знаете PostgreSQL, многие концепции покажутся знакомыми.

Kubernetes	PostgreSQL	Назначение
etcd	`pg_control` + WAL	Хранит текущее и желаемое состояние кластера
Controller Manager	`autovacuum`, `checkpointer`	Фоновые процессы, следящие за здоровьем
Scheduler	Query Planner	Выбирает оптимальный “план” размещения подов
kubelet	`postmaster`	Главный процесс на каждой ноде, запускает контейнеры
Pod	Backend process	Единица работы (один или несколько контейнеров)
Service	`pgbouncer` / VIP	Стабильная точка подключения
Namespace	Schema	Логическая группировка ресурсов
Deployment	`pg_basebackup` + `recovery.conf`	Декларативное описание желаемого состояния

1.4.2 Что такое Kubernetes на самом деле

Определение: Kubernetes (K8s) — это оркестратор контейнеров, который автоматизирует развёртывание, масштабирование и управление приложениями.

Ключевое слово: Декларативный подход.

Вы не говорите Kubernetes **“как”** делать. Вы говорите **“что”** хотите получить.

Пример:

```
# Вы говорите: "Хочу 3 экземпляра PostgreSQL версии 16"
replicas: 3
image: postgres:16
```

Kubernetes сам:

- Запустит 3 пода
- Распределит по нодам
- Будет следить за состоянием
- При падении пода — пересоздаст
- При падении ноды — перенесёт на другую

Сравните с императивным подходом (без K8s):

```
# Вы говорите КАК делать:
ssh server1 "docker run postgres:16"
ssh server2 "docker run postgres:16"
ssh server3 "docker run postgres:16"
# А если server1 упадёт? Нужен скрипт мониторинга, скрипт перезапуска...
```

1.4.3 Архитектура кластера

Kubernetes-кластер состоит из двух типов нод:

1. Control Plane (управляющая плоскость)

Это “мозг” кластера. Компоненты:

- **kube-apiserver** — единая точка входа. Все команды (`kubectl`) идут сюда. REST API для всех операций.
- **etcd** — распределённое key-value хранилище. Хранит всё состояние кластера. Критично для работы! Аналог `pg_control` + WAL.
- **kube-scheduler** — выбирает, на какой ноде запустить под. Учитывает ресурсы (CPU/RAM), affinity rules, taints/tolerations.
- **kube-controller-manager** — набор контроллеров (ReplicaSet Controller, Deployment Controller и т.д.). Каждый следит за своим типом объектов.
- **cloud-controller-manager** (опционально) — интеграция с облачными провайдерами (AWS, GCP, Azure) для LoadBalancer, Persistent Volumes.

2. Worker Nodes (рабочие ноды)

Это “мускулы” кластера. Здесь запускаются ваши приложения (поды). Компоненты:

- **kubelet** — агент на каждой ноде. Получает от API Server инструкции “запустить этот под” и выполняет через Container Runtime.
- **kube-proxy** — сетевой прокси. Настраивает iptables/IPVS правила для маршрутизации трафика к подам.
- **Container Runtime** — собственно запускает контейнеры (containerd, CRI-O).

1.4.4 Как работает развёртывание (на примере PostgreSQL)

Шаг 1: Вы создаёте Deployment

```
kubectl apply -f postgres-deployment.yaml
```

Шаг 2: API Server сохраняет в etcd

“Пользователь хочет Deployment с 3 репликами PostgreSQL.”

Шаг 3: Deployment Controller обнаруживает новый объект

Controller постоянно опрашивает API Server: “Есть ли новые Deployment?”

Видит новый Deployment → создаёт ReplicaSet.

Шаг 4: ReplicaSet Controller создаёт Pods

ReplicaSet Controller: “Нужно 3 пода, а сейчас 0. Создаю 3 пода.”

Поды создаются в etcd со статусом “Pending” (ожидание).

Шаг 5: Scheduler назначает поды на ноды

Scheduler смотрит на “Pending” поды и выбирает ноды: - Проверяет доступные ресурсы (CPU, RAM) - Учитывает affinity rules (например, “не размещать все реплики PostgreSQL на одной ноде”) - Обновляет в etcd: “Pod-1 → Node-1, Pod-2 → Node-2, Pod-3 → Node-3”

Шаг 6: Kubelet запускает контейнеры

Kubelet на каждой ноде следит за API Server: “Есть ли для меня новые поды?”

Видит назначение → скачивает образ `postgres:16` → запускает через Container Runtime.

Шаг 7: Контроллеры следят за здоровьем

ReplicaSet Controller постоянно проверяет: “Сколько подов запущено? Нужно 3.”

Если под упал → создаёт новый.

Если нода упала → Scheduler переназначает поды на другие ноды.

1.4.5 Что важно понять DBA

1. Kubernetes не знает про PostgreSQL

Kubernetes видит поды как чёрные ящики. Он не понимает, что такое Primary/Replica, не знает про репликацию, не умеет делать pg_basebackup.

Решение: Операторы PostgreSQL (Zalando, CloudNativePG, PGO) добавляют эту логику.

2. Состояние хранится в etcd

Если etcd сломается — кластер перестанет работать (как PostgreSQL без `pg_control`).

Production: etcd должен быть в кластере из 3-5 нод с резервными копиями.

3. Контроллеры работают по принципу “reconciliation loop”

Контроллер постоянно сравнивает: - **Желаемое состояние** (что написано в манифесте) - **Текущее состояние** (что реально запущено)

Если не совпадает → приводит к желаемому.

Аналогия в PostgreSQL: `autovacuum` постоянно проверяет таблицы и вычищает мёртвые строки.

4. Декларативность означает идемпотентность

```
kubectl apply -f deployment.yaml  # Первый раз – создаёт
kubectl apply -f deployment.yaml  # Второй раз – ничего не делает (уже создано)
kubectl apply -f deployment.yaml  # Изменили replicas: 5 – обновит
```

Это удобно для автоматизации: CI/CD может просто выполнять `kubectl apply` без проверки “а существует ли уже?”.

1.4.6 Scheduler: как Kubernetes размещает поды

Задача Scheduler: Выбрать оптимальную ноду для каждого пода.

Процесс принятия решения:

1. **Фильтрация (Filtering)** — исключить ноды, которые не подходят
2. **Ранжирование (Scoring)** — оценить оставшиеся ноды по приоритету
3. **Назначение (Binding)** — выбрать ноду с наивысшим баллом

1.4.6.1 Requests и Limits — основа для планирования

Requests (запросы): - Гарантированные ресурсы для пода - Scheduler использует requests для фильтрации нод - Если на ноде недостаточно свободных ресурсов → нода исключается

Limits (лимиты): - Максимальные ресурсы, которые под может использовать - Если под превышает CPU limit → throttling (замедление) - Если под превышает Memory limit → OOMKilled (pod terminated)

Пример для PostgreSQL:

```
resources:
  requests:
    memory: "4Gi"    # Гарантированно 4 GB RAM
    cpu: "2000m"     # Гарантированно 2 CPU cores (2000 millicores)
  limits:
    memory: "8Gi"    # Максимум 8 GB RAM
    cpu: "4000m"     # Максимум 4 CPU cores
```

Важно для DBA:

- **Requests < реальное потребление** → PostgreSQL будет тормозить из-за нехватки ресурсов
- **Limits слишком низкие** → OOMKilled во время пиковых нагрузок (потеря данных!)
- **Requests = Limits** (QoS Guaranteed) → лучший вариант для production БД

Quality of Service (QoS) классы:

QoS класс	Условие	Поведение при нехватке ресурсов
Guaranteed	requests = limits для всех контейнеров	Последний кандидат на eviction (выселение)
Burstable	requests < limits	Средний приоритет
BestEffort	Нет requests и limits	Первый кандидат на eviction

Для PostgreSQL всегда используйте Guaranteed!

1.4.6.2 nodeSelector — простое размещение

Назначение: Запустить под только на нодах с определёнными метками.

Пример: PostgreSQL только на нодах с SSD-дисками.

Шаг 1: Пометить ноду

```
kubectl label nodes worker-1 disk=ssd
kubectl label nodes worker-2 disk=ssd
```

Шаг 2: Указать в манифесте

```
spec:
  nodeSelector:
    disk: ssd  # Под запустится ТОЛЬКО на нодах с меткой disk=ssd
```

Использование: - Размещение на нодах с определённым типом дисков (SSD vs HDD) - Размещение на нодах с GPU (для ML-задач, но не для PostgreSQL) - Изоляция production и staging (разные ноды)

Недостаток: Слишком жёсткое правило. Если нет нод с меткой `disk=ssd` → под останется в состоянии Pending.

1.4.6.3 Affinity и Anti-Affinity — гибкое размещение

Node Affinity (привязка к ноде):

Более гибкая версия nodeSelector с двумя типами правил:

1. requiredDuringSchedulingIgnoredDuringExecution (жёсткое требование): - Под ОБЯЗАН запуститься на ноде, соответствующей правилу - Если нет подходящей ноды → Pending

2. preferredDuringSchedulingIgnoredDuringExecution (предпочтение): - Scheduler **предпочитает** ноды, соответствующие правилу - Если нет → запустит на любой подходящей

Пример для PostgreSQL Primary (предпочитает SSD, но может запуститься на HDD):

```
affinity:
  nodeAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 100  # Приоритет (1-100)
      preference:
        matchExpressions:
        - key: disk
          operator: In
          values:
          - ssd
          - nvme
```

Pod Affinity (привязка к подам):

Назначение: Запустить поды **рядом** с другими подами.

Пример: Read-реплики PostgreSQL рядом с приложением (уменьшить network latency).

```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: app
          operator: In
          values:
          - myapp
      topologyKey: kubernetes.io/hostname  # На той же ноде
```

topologyKey: - `kubernetes.io/hostname` — та же нода - `topology.kubernetes.io/zone` — та же зона доступности (AWS AZ, GCP Zone) - `topology.kubernetes.io/region` — тот же регион

Pod Anti-Affinity (размещение подальше):

Критически важно для PostgreSQL!

Назначение: Запустить поды **на разных нодах** (избежать single point of failure).

Пример: Primary и Replica PostgreSQL на разных нодах.

```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: app
          operator: In
          values:
          - postgresql
      topologyKey: kubernetes.io/hostname  # Разные ноды!
```

Результат: - `postgres-0` (Primary) → worker-1 - `postgres-1` (Replica) → worker-2 - `postgres-2` (Replica) → worker-3

Если worker-1 упадёт → Primary пропадёт, но Replica жива на worker-2 → автоматический failover.

Без Anti-Affinity: Все поды могут оказаться на одной ноде → нода упала → все реплики пропали → данных нет!

Best practice для PostgreSQL production:

```yaml
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchLabels:
          app: postgresql
          cluster: my-pg-cluster
      topologyKey: kubernetes.io/hostname
  nodeAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 100
      preference:
        matchExpressions:
        - key: disk
          operator: In
          values:
          - ssd
          - nvme
```

Означает: 1. Обязательно размещать на разных нодах (Anti-Affinity) 2. Предпочитать ноды с SSD/NVMe (Node Affinity)

1.4.6.4 Taints и Tolerations — резервирование нод

Назначение: Запретить размещение подов на определённых нодах (кроме подов с tolerations).

Пример: Выделенные ноды только для баз данных.

Шаг 1: Пометить ноду taint

```
kubectl taint nodes db-node-1 workload=database:NoSchedule
```

Результат: Обычные поды не смогут запуститься на `db-node-1`.

Шаг 2: Разрешить PostgreSQL

```yaml
tolerations:
- key: workload
  operator: Equal
  value: database
  effect: NoSchedule
```

Теперь: Только поды PostgreSQL (с toleration) могут запуститься на `db-node-1`.

Типы effects:

- **NoSchedule** — новые поды не размещаются

- **PreferNoSchedule** — Scheduler избегает, но может разместить
- **NoExecute** — существующие поды без toleration будут evicted (выселены)

Использование для DBA: - Выделенные ноды для production БД (изоляция от других нагрузок) - Ноды с особыми требованиями (compliance, security)

1.4.6.5 Другие факторы размещения

1. PersistentVolume locality

Если под использует локальный PersistentVolume → Scheduler обязан разместить под на ноде, где находится том.

2. Priority и Preemption

PriorityClass: Приоритет подов.

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: database-critical
value: 1000000  # Высокий приоритет
globalDefault: false
description: "Critical database workloads"
```

Использование:

```
spec:
  priorityClassName: database-critical
```

Поведение: - Если ресурсов не хватает → Kubernetes может **evict (выселить)** поды с низким приоритетом, чтобы освободить место для подов с высоким приоритетом - **Для PostgreSQL Primary** → всегда высокий приоритет!

3. Spread Constraints (равномерное распределение)

Назначение: Распределить поды равномерно по зонам доступности.

```
topologySpreadConstraints:
- maxSkew: 1  # Максимальная разница между зонами
  topologyKey: topology.kubernetes.io/zone
  whenUnsatisfiable: DoNotSchedule
  labelSelector:
    matchLabels:
      app: postgresql
```

Результат: - Зона A: 1 реплика - Зона B: 1 реплика - Зона C: 1 реплика

При сбое целой зоны → 2 реплики остаются живы.

1.5 Основные объекты и их применение

1.5.1 Namespace — изоляция окружений

Назначение: Логическая группировка ресурсов. Как Schema в PostgreSQL.

Типичное использование:

```yaml
# Создание namespace
apiVersion: v1
kind: Namespace
metadata:
  name: postgres-production
---
apiVersion: v1
kind: Namespace
metadata:
  name: postgres-staging
```

Зачем DBA:

- Изоляция окружений (prod, staging, dev)
- Квоты ресурсов (в staging не может съесть все ресурсы кластера)
- RBAC (кто имеет доступ к production, а кто только к dev)

Команды:

```
# Список namespace
kubectl get namespaces

# Создать namespace
kubectl create namespace postgres-test

# Переключить контекст на namespace (все команды будут в нём)
kubectl config set-context --current --namespace=postgres-test

# Удалить namespace (удалит ВСЕ ресурсы внутри!)
kubectl delete namespace postgres-test
```

1.5.2 Labels и Selectors — метаданные и фильтрация

Назначение: Key-value метки для объектов. Используются для фильтрации и связывания объектов.

Пример для PostgreSQL:

```
metadata:
  labels:
    app: postgresql          # Приложение
    role: primary            # Роль (primary / replica)
    version: "16"            # Версия PostgreSQL
    environment: production  # Окружение
    team: dba                # Команда-владелец
```

Селекторы (выборка по меткам):

```
# Все поды PostgreSQL
kubectl get pods -l app=postgresql

# Только primary
kubectl get pods -l app=postgresql,role=primary

# Только production
kubectl get pods -l environment=production

# Все реплики версии 16
kubectl get pods -l app=postgresql,role=replica,version=16
```

Зачем DBA:

Service использует селекторы для выбора подов:

```
# Service для подключения к Primary
selector:
  app: postgresql
  role: primary  # Трафик только на под с меткой role=primary
```

При failover (promote реплики в primary) достаточно **сменить метку** на новом primary — Service автоматически переключит трафик!

1.5.3 Pod — минимальная единица

Назначение: Группа из одного или нескольких контейнеров с общей сетью и volumes.

Важно: Pod — **эфемерный** (временный). Он может быть удалён и пересоздан в любой момент.

Типичный Pod для PostgreSQL:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: postgres-primary
  labels:
    app: postgresql
    role: primary
spec:
  containers:
  - name: postgres
    image: postgres:16
    env:
    - name: POSTGRES_PASSWORD
      value: "secretpassword"  # В production использовать Secret!
    - name: PGDATA
      value: /var/lib/postgresql/data/pgdata
    ports:
    - containerPort: 5432
    volumeMounts:
    - name: pgdata
      mountPath: /var/lib/postgresql/data
  volumes:
  - name: pgdata
    emptyDir: {}  # Временное хранилище (данные пропадут при удалении пода!)
```

Проблема: При удалении пода данные пропадут!

Решение: Использовать PersistentVolume (рассмотрим в следующем занятии).

Команды:

```
# Создать под
kubectl apply -f postgres-pod.yaml

# Список подов
kubectl get pods

# Детальная информация
kubectl describe pod postgres-primary

# Логи контейнера
kubectl logs postgres-primary

# Выполнить команду внутри пода
kubectl exec -it postgres-primary -- psql -U postgres

# Удалить под
kubectl delete pod postgres-primary
```

1.5.4 ReplicaSet — управление репликами

Назначение: Гарантирует, что всегда запущено N подов.

Пример:

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: postgres-replicas
spec:
  replicas: 3  # Всегда 3 пода
  selector:
    matchLabels:
      app: postgresql
      role: replica
  template:  # Шаблон пода (как в Pod выше)
    metadata:
      labels:
        app: postgresql
        role: replica
    spec:
      containers:
      - name: postgres
        image: postgres:16
        # ... (как в Pod)
```

Что делает ReplicaSet:

1. Следит за количеством подов с метками `app=postgresql, role=replica`

2. Если подов меньше 3 → создаёт новые
3. Если подов больше 3 → удаляет лишние
4. Если под упал → создаёт новый

Масштабирование:

```
# Увеличить до 5 реплик
kubectl scale replicaset postgres-replicas --replicas=5

# Уменьшить до 1
kubectl scale replicaset postgres-replicas --replicas=1
```

Проблема: ReplicaSet не умеет обновлять версию образа. Если изменить `image: postgres:17`, он не пересоздаст старые поды.

Решение: Используйте Deployment.

1.5.5 Deployment — production-ready управление

Назначение: Обёртка над ReplicaSet с возможностями обновления и отката.

Преимущества:

- Декларативные обновления (rolling update)
- Откат к предыдущим версиям
- Стратегии обновления (RollingUpdate, Recreate)
- История изменений

Пример:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: postgresql
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1  # Максимум 1 под может быть недоступен
      maxSurge: 1        # Максимум 1 дополнительный под при обновлении
  template:
    metadata:
      labels:
        app: postgresql
    spec:
      containers:
      - name: postgres
        image: postgres:16
        # ...
```

Обновление версии:

```bash
# Изменить образ на postgres:17
kubectl set image deployment/postgres-deployment postgres=postgres:17

# Kubernetes сделает:
# 1. Создаст новый под с postgres:17
# 2. Дождётся готовности (health check)
# 3. Удалит старый под с postgres:16
# 4. Повторит для всех подов (по одному)
```

Откат:

```bash
# Посмотреть историю
kubectl rollout history deployment/postgres-deployment

# Откатиться к предыдущей версии
kubectl rollout undo deployment/postgres-deployment

# Откатиться к конкретной ревизии
kubectl rollout undo deployment/postgres-deployment --to-revision=2
```

1.5.5.1 Стратегии обновления Deployment

1. RollingUpdate (по умолчанию)

Принцип: Постепенная замена старых подов на новые.

Параметры:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1  # Сколько подов может быть недоступно одновременно
    maxSurge: 1        # Сколько дополнительных подов создать сверх replicas
```

maxUnavailable (максимум недоступных):

- **Абсолютное число:** `maxUnavailable: 2` — максимум 2 пода недоступны
- **Процент:** `maxUnavailable: 25%` — максимум 25% от replicas

Пример: `replicas: 4`, `maxUnavailable: 1`

Процесс обновления:

```
Шаг 1: [Old] [Old] [Old] [Old]  → Удалить 1 старый
Шаг 2: [Old] [Old] [Old] [New]  → Создать 1 новый, дождаться Ready
Шаг 3: [Old] [Old] [New] [New]  → Удалить следующий старый
Шаг 4: [Old] [New] [New] [New]  → Создать новый, дождаться Ready
Шаг 5: [New] [New] [New] [New]  → Готово!
```

Доступность: Минимум 3 пода доступны всегда (4 - 1 maxUnavailable).

maxSurge (максимум дополнительных):

- **Абсолютное число:** `maxSurge: 2` — максимум +2 пода сверх replicas
- **Процент:** `maxSurge: 50%` — максимум +50% от replicas

Пример: `replicas: 4`, `maxSurge: 2`, `maxUnavailable: 0`

Процесс обновления:

```
Шаг 1: [Old] [Old] [Old] [Old]           → Создать 2 новых
Шаг 2: [Old] [Old] [Old] [Old] [New] [New]  → Дождаться Ready
Шаг 3: [New] [New] [Old] [Old] [New] [New]  → Удалить 2 старых
Шаг 4: [New] [New] [New] [New]           → Готово!
```

Доступность: Все 4 пода доступны всегда (maxUnavailable: 0).

Компромисс: Временно 6 подов вместо 4 → больше ресурсов.

Типичные конфигурации:

Сценарий	maxUnavailable	maxSurge	Поведение
Экономия ресурсов	1	0	Медленное обновление, без дополнительных подов
Быстрое обновление	25%	25%	Умеренная скорость, умеренные ресурсы
Нулевой downtime	0	1	Гарантия доступности, но +1 под временно
Агрессивное	50%	50%	Быстро, но много ресурсов

Health Checks и RollingUpdate:

Kubernetes **не удалит** старый под, пока новый под не пройдёт **Readiness Probe**.

Пример Readiness Probe для PostgreSQL:

```
readinessProbe:
  exec:
    command:
    - psql
    - -U
    - postgres
    - -c
    - SELECT 1
  initialDelaySeconds: 10  # Ждать 10 сек после старта
  periodSeconds: 5         # Проверять каждые 5 сек
  timeoutSeconds: 3
  failureThreshold: 3      # 3 неудачи → под НЕ готов
```

Если новый под не проходит Readiness Probe: - Старый под **не удаляется** - Обновление **останавливается** - Ручное вмешательство требуется

2. Recreate (полная пересборка)

Принцип: Удалить ВСЕ старые поды, затем создать новые.

Конфигурация:

```yaml
strategy:
  type: Recreate
```

Процесс обновления:

```
Шаг 1: [Old] [Old] [Old] [Old]  → Удалить ВСЕ старые
Шаг 2: []    []    []    []     → Downtime! Нет подов!
Шаг 3: [New] [New] [New] [New]  → Создать все новые
```

Downtime: Да, пока старые удаляются и новые стартуют.

Когда использовать:

- Приложение **не поддерживает** одновременную работу старой и новой версий (несовместимость схем БД)
- Shared state между подами (например, кластер с выборами лидера)
- Нужна гарантия “сначала остановить всё, потом запустить”

Для PostgreSQL:

- **RollingUpdate НЕ подходит для Primary!** (нельзя удалить Primary во время работы)
- **Recreate НЕ подходит для production!** (downtime)
- **Решение:** StatefulSet + Operator (управляет обновлением Primary корректно)

3. Blue-Green и Canary (через дополнительные инструменты)

Blue-Green Deployment:

- Создать полностью новое окружение (Green)
- Переключить трафик с старого (Blue) на новое (Green)
- Если проблемы → откатить трафик обратно

В Kubernetes: Создать второй Deployment, переключить Service selector.

Canary Deployment:

- Запустить небольшой процент подов с новой версией (canary)
- Если метрики хорошие → постепенно увеличивать процент
- Если плохие → откатить

В Kubernetes: Два Deployment (old + canary), Service балансирует трафик между ними.

Для advanced стратегий используйте: - **Argo Rollouts** — прогрессивная доставка (canary, blue-green) - **Flagger** — автоматический canary с метриками

1.5.5.2 Мониторинг обновления Deployment

Проверка статуса:

```
# Статус обновления
kubectl rollout status deployment/my-app

# История ревизий
kubectl rollout history deployment/my-app

# Детали конкретной ревизии
kubectl rollout history deployment/my-app --revision=3

# Пауза обновления (если проблемы)
kubectl rollout pause deployment/my-app

# Возобновить
kubectl rollout resume deployment/my-app
```

Автоматический откат при проблемах:

```
spec:
  progressDeadlineSeconds: 600  # Если обновление не завершится за 10 минут →
        Failed
  revisionHistoryLimit: 10      # Хранить 10 старых ревизий для отката
```

Важно для PostgreSQL:

Rolling Update подходит для **stateless** приложений (pgBouncer, pgAdmin).

Для **PostgreSQL с репликацией** нужна специальная логика: - Primary нельзя просто удалить и пересоздать - Нужен graceful failover перед обновлением Primary - Replica можно обновлять через RollingUpdate

Поэтому используют StatefulSet и Операторы.

1.5.6 Сравнительная таблица

Объект	Использование	Для PostgreSQL
Pod	Разовый запуск для теста	Только для экспериментов
ReplicaSet	Несколько одинаковых подов	Не подходит (нужен StatefulSet)
Deployment	Stateless приложения	Не подходит для Primary (подходит для pgBouncer, pgAdmin)
StatefulSet	Stateful приложения с уникальными идентификаторами	✅ Используется для PostgreSQL

1.6 Сетевое взаимодействие в кластере

1.6.1 Сетевая модель Kubernetes

Основные принципы:

1. **Каждый под получает свой IP-адрес**
2. **Поды на одной ноде** видят друг друга напрямую
3. **Поды на разных нодах** видят друг друга без NAT
4. **IP пода эфемерный** (меняется при пересоздании)

Распределение адресов:

- **Pod CIDR:** 10.244.0.0/16 (по умолчанию)
  - Node1: 10.244.0.0/24 (255 адресов для подов)
  - Node2: 10.244.1.0/24
  - Node3: 10.244.2.0/24
- **Service CIDR:** 10.96.0.0/12 (виртуальные IP для Service)

Проблема для баз данных:

Pod PostgreSQL получил IP `10.244.1.15`. Приложение подключилось. Под пересоздался → новый IP `10.244.1.23`. Приложение не может подключиться!

Решение: Service.

1.6.2 Service — стабильная точка доступа

Назначение: Service предоставляет постоянный IP и DNS-имя для группы подов.

Как работает:

```
apiVersion: v1
kind: Service
metadata:
  name: postgres-primary
spec:
  selector:
    app: postgresql
    role: primary  # Service найдёт ВСЕ поды с этими метками
  ports:
  - port: 5432          # Порт Service
    targetPort: 5432    # Порт контейнера
  type: ClusterIP       # Тип Service (см. ниже)
```

Service получает: - **Cluster IP:** 10.96.123.45 (стабильный, не меняется) - **DNS-имя:** `postgres-primary.default.svc.cluster.local`

Приложение подключается к Service → Service перенаправляет трафик на под с меткой `role=primary`.

Если под пересоздался:

- IP пода изменился
- Service автоматически обновил маршруты (через kube-proxy)
- Приложение продолжает подключаться по тому же DNS

1.6.3 Типы Service

1. ClusterIP (по умолчанию)

Назначение: Доступ только внутри кластера.

Использование: Межсервисное взаимодействие (приложение → PostgreSQL).

```
type: ClusterIP
```

2. NodePort

Назначение: Доступ извне кластера через порт ноды.

Как работает:

- Kubernetes открывает порт (например, 30432) на ВСЕХ нодах
- Трафик на `<any-node-ip>:30432` перенаправляется на Service

```
type: NodePort
ports:
- port: 5432
  nodePort: 30432  # Диапазон 30000-32767
```

Использование: Тестирование (для production не рекомендуется).

3. LoadBalancer

Назначение: Интеграция с облачным балансировщиком (AWS ELB, GCP LB).

Как работает:

- Kubernetes запрашивает у облака внешний IP
- Трафик на внешний IP → балансировщик → Service → поды

```
type: LoadBalancer
```

Использование: Production в облаке.

4. ExternalName (редко)

Назначение: CNAME-алиас на внешний DNS.

1.6.4 Endpoints — связь Service и Pods

Что это: Список IP:Port подов, соответствующих селектору Service.

Проверка:

```
# Посмотреть endpoints
kubectl get endpoints postgres-primary

# Детали
kubectl describe endpoints postgres-primary
```

Пример вывода:

```
NAME               ENDPOINTS
postgres-primary   10.244.1.15:5432
```

Если под упал:

```
NAME               ENDPOINTS
postgres-primary   <none>
```

Service не будет маршрутизировать трафик (нет живых подов).

1.6.5 DNS в Kubernetes

Каждый Service автоматически получает DNS-запись:

```
<service-name>.<namespace>.svc.cluster.local
```

Примеры:

- Service `postgres-primary` в namespace `default`:

```
postgres-primary.default.svc.cluster.local
```

- Service `postgres-replica` в namespace `production`:

```
postgres-replica.production.svc.cluster.local
```

Сокращения (внутри того же namespace):

```
# Полное имя
psql -h postgres-primary.default.svc.cluster.local -U postgres

# Сокращённое (если в том же namespace)
psql -h postgres-primary -U postgres
```

CoreDNS: Kubernetes использует CoreDNS для разрешения имён.

1.6.6 Практический пример для PostgreSQL

Архитектура:

```
┌─────────────────────────┐
| Application Pod         |
└─────────────┬───────────┘
              | psql -h postgres-primary
              ↓
┌─────────────────────────┐
| Service:                |
| postgres-primary        |
| ClusterIP: 10.96...     |
└─────────────┬───────────┘
              | Selector: role=primary
              ↓
┌─────────────────────────┐
| Pod (Primary)           |
| IP: 10.244.1.15         |
└─────────────────────────┘
```

Если Primary упал и был promote Replica:

1. Оператор меняет метку:

```
kubectl label pod postgres-replica-1 role=primary --overwrite
kubectl label pod postgres-primary role=replica --overwrite
```

2. Service автоматически переключает Endpoints на новый под

3. Приложение продолжает подключаться по тому же DNS

Downtime: Только время переключения (~1-5 секунд).

1.6.7 Ingress — HTTP-маршрутизация

Назначение: Маршрутизация внешнего HTTP/HTTPS трафика к Service.

Использование для PostgreSQL: Редко (PostgreSQL использует TCP, не HTTP). Но может использоваться для pgAdmin, pgWeb и других веб-интерфейсов.

Пример (pgAdmin через Ingress):

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: pgadmin-ingress
spec:
  rules:
  - host: pgadmin.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: pgadmin-service
            port:
              number: 80
```

Результат:

```
https://pgadmin.example.com → Ingress Controller → Service → pgAdmin Pod
```

Ingress Controller: Нужен отдельный контроллер (NGINX Ingress, Traefik). Он читает Ingress объекты и настраивает маршруты.

1.7 PostgreSQL в Kubernetes: первые шаги

1.7.1 Запуск PostgreSQL в Pod

Минимальный манифест:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pg-demo
  labels:
    app: postgresql
spec:
  containers:
  - name: postgres
    image: postgres:16
    env:
    - name: POSTGRES_PASSWORD
      value: "demo123"
    - name: PGDATA
      value: /var/lib/postgresql/data/pgdata
    ports:
    - containerPort: 5432
```

Запуск:

```
kubectl apply -f pg-pod.yaml
kubectl get pods
```

Подключение (через port-forward):

```
kubectl port-forward pg-demo 5432:5432
```

В другом терминале:

```
psql -h localhost -U postgres
```

Что произойдёт:

- PostgreSQL запустится
- Данные хранятся в `/var/lib/postgresql/data` внутри контейнера
- **При удалении пода данные пропадут!**

1.7.2 Создание Service

Манифест:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: pg-service
spec:
  selector:
    app: postgresql
  ports:
  - port: 5432
    targetPort: 5432
  type: ClusterIP
```

Проверка:

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

Подключение из другого пода:

```
# Запустить временный под с psql
kubectl run -it --rm psql --image=postgres:16 --restart=Never -- \
  psql -h pg-service -U postgres
```

1.7.3 Работа с данными

Создание базы и таблицы:

```
CREATE DATABASE testdb;
\c testdb
CREATE TABLE users(id serial PRIMARY KEY, name text);
INSERT INTO users(name) VALUES ('DBA в Kubernetes');
SELECT * FROM users;
```

Проверка персистентности:

```
# Удалить под
kubectl delete pod pg-demo

# Пересоздать
kubectl apply -f pg-pod.yaml

# Попробовать подключиться
kubectl port-forward pg-demo 5432:5432
psql -h localhost -U postgres -c '\l'
```

Результат: База `testdb` пропала! Данные потеряны.

1.7.4 Проблема: отсутствие персистентности

Почему:

- Pod использует `emptyDir` volume (временное хранилище)
- При удалении пода volume удаляется

Что нужно:

- **PersistentVolume (PV)** — абстракция над физическим хранилищем
- **PersistentVolumeClaim (PVC)** — запрос на хранилище
- **StatefulSet** — управление stateful приложениями

Пример PVC:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc
spec:
  accessModes:
  - ReadWriteOnce  # Один под может писать
  resources:
    requests:
      storage: 10Gi  # Размер
```

StatefulSet с PVC:

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
spec:
  serviceName: postgres
  replicas: 1
  selector:
    matchLabels:
      app: postgresql
  template:
    metadata:
      labels:
        app: postgresql
    spec:
      containers:
      - name: postgres
        image: postgres:16
        volumeMounts:
        - name: pgdata
          mountPath: /var/lib/postgresql/data
  volumeClaimTemplates:
  - metadata:
      name: pgdata
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 10Gi
```

Результат:

- Kubernetes создаст PVC автоматически
- Данные сохранятся даже при удалении пода
- Новый под подключит тот же PVC

Это мы рассмотрим подробно на следующем занятии.

1.8 Путь к production: что дальше

1.8.1 Что мы увидели и чего не хватает

Pod PostgreSQL работает, но:

✗ Нет persistent storage — данные пропадают при рестарте ✗ Нет репликации ✗ Нет автоматического failover ✗ Нет бэкапов ✗ Нет управления конфигурацией (`postgresql.conf`, `pg_hba.conf`)

Всё это решают:

1. StatefulSet + PersistentVolume

- Стабильные имена подов (`postgres-0` , `postgres-1`)
- Постоянные тома (данные переживают рестарт)
- Упорядоченное создание/удаление подов

2. ConfigMap / Secret

- ConfigMap для `postgresql.conf`
- Secret для паролей, сертификатов

3. Операторы PostgreSQL

Операторы добавляют знания о PostgreSQL в Kubernetes:

Zalando Postgres Operator:

- Автоматическая репликация через Patroni
- Автоматический failover
- Point-in-time recovery (PITR)
- Connection pooling (pgBouncer)
- Мониторинг (Prometheus экспортеры)

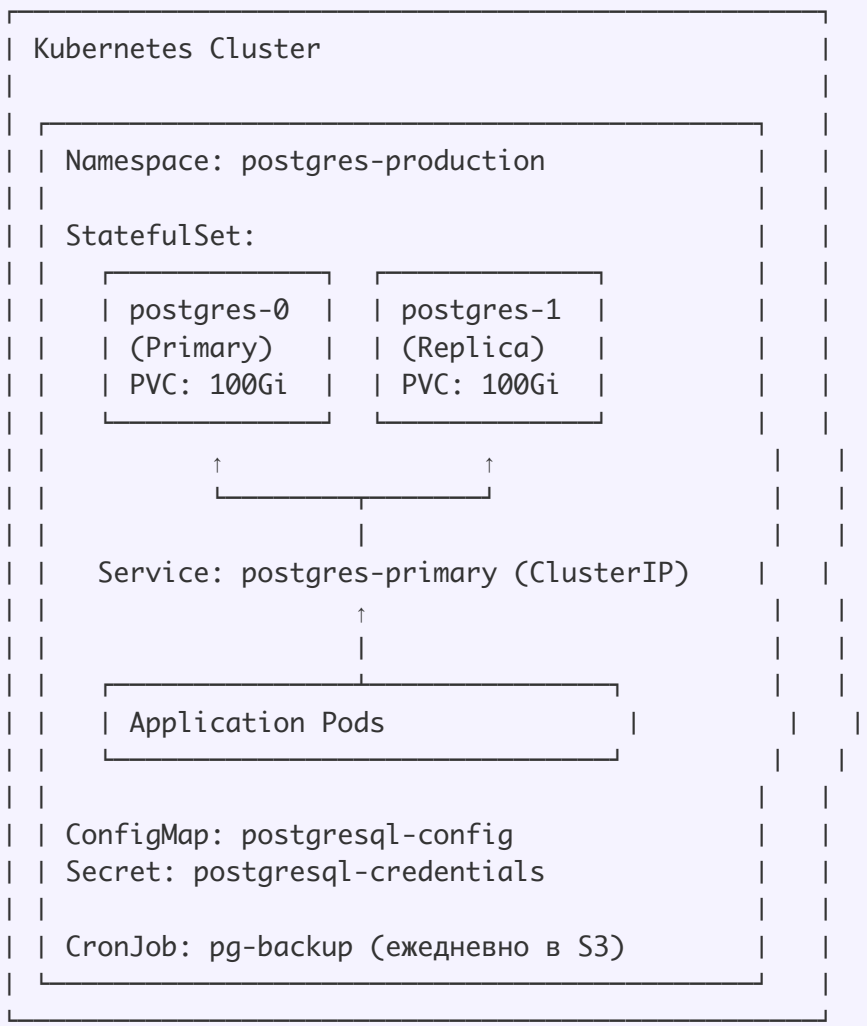
CloudNativePG:

- Декларативные манифесты
- Встроенные бэкапы (S3, Azure Blob)
- Rolling updates без простоя
- Read-only replicas

Crunchy Data PGO:

- Enterprise-поддержка
- Интеграция с Kubernetes RBAC
- TLS из коробки

1.8.2 Типичная production-архитектура



1.8.3 Операторы PostgreSQL: сравнение

Оператор	Плюсы	Минусы	Кому подходит
Zalando	Проверен временем (5+ лет production), Patroni из коробки, большое сообщество	Сложная конфигурация, много зависимостей	Опытные команды, высокие требования
CloudNativePG	Простота, хорошая документация, CNCF Sandbox	Моложе (но активно развивается)	Начинающие, средний уровень
Crunchy PGO	Enterprise-поддержка, OpenShift-сертификация	Платная поддержка (community версия есть)	Enterprise, требования к сертификации

1.8.4 Что изучить на следующем занятии

Тема: “Работа с хранилищами данных и конфигурациями”

План:

1. **PersistentVolume и PersistentVolumeClaim** — как правильно хранить данные PostgreSQL
2. **StatefulSet** — управление stateful приложениями (PostgreSQL, Kafka)
3. **ConfigMap и Secret** — управление конфигурацией и секретами
4. **InitContainers** — инициализация базы данных
5. **Операторы PostgreSQL** — развёртывание полноценного кластера с репликацией
6. **Бэкапы и восстановление** — стратегии резервного копирования

1.9 Практические рецепты и troubleshooting

1.9.1 Проверка состояния кластера

Базовые команды:

```
# Узлы кластера
kubectl get nodes

# Все поды во всех namespace
kubectl get pods --all-namespaces

# Поды с проблемами
kubectl get pods --field-selector=status.phase!=Running

# События (полезно при проблемах)
kubectl get events --sort-by='.lastTimestamp'
```

1.9.2 Диагностика пода PostgreSQL

Проверка статуса:

```
# Детальная информация
kubectl describe pod postgres-0

# Логи контейнера
kubectl logs postgres-0

# Логи с tail
kubectl logs -f postgres-0 --tail=50

# Логи предыдущего контейнера (если под перезапускался)
kubectl logs postgres-0 --previous
```

Подключение к поду:

```
# Запуск оболочки
kubectl exec -it postgres-0 -- /bin/bash

# Прямой psql
kubectl exec -it postgres-0 -- psql -U postgres

# Проверка конфигурации
kubectl exec -it postgres-0 -- cat /var/lib/postgresql/data/postgresql.conf
```

1.9.3 Частые проблемы

1. Pod в статусе “Pending”

Причины:

- Недостаточно ресурсов на нодах

- PVC не может быть привязан (нет доступного PV)
- Node selector не находит подходящую ноду

Диагностика:

```
kubectl describe pod postgres-0 | grep -A 10 Events
```

Решение:

```
# Проверить ресурсы нод
kubectl describe nodes | grep -A 5 "Allocated resources"

# Проверить PVC
kubectl get pvc
kubectl describe pvc postgres-pvc
```

2. Pod в статусе “CrashLoopBackOff”

Причина: Контейнер запускается и падает.

Диагностика:

```
# Логи
kubectl logs postgres-0 --previous

# Проверить readiness/liveness probes
kubectl describe pod postgres-0 | grep -A 5 Liveness
```

Частая причина для PostgreSQL:

- Неправильные permissions на PGDATA
- Конфликт портов
- Неверный POSTGRES_PASSWORD (если требуется конкретный формат)

Решение:

```
# Проверить права на volume
kubectl exec -it postgres-0 -- ls -la /var/lib/postgresql/data

# Если нужно, исправить
kubectl exec -it postgres-0 -- chown -R postgres:postgres /var/lib/postgresql/
        data
```

3. Невозможно подключиться к PostgreSQL

Проверка:

```
# Service существует?
kubectl get service postgres-primary

# Endpoints существуют?
kubectl get endpoints postgres-primary

# Под живой?
kubectl get pod postgres-0
```

Тест подключения:

```
# Port-forward
kubectl port-forward postgres-0 5432:5432

# В другом терминале
psql -h localhost -U postgres
```

Если через port-forward работает, а через Service нет:

```
# Проверить селектор Service
kubectl get service postgres-primary -o yaml | grep -A 5 selector

# Проверить метки пода
kubectl get pod postgres-0 --show-labels
```

4. Медленная работа PostgreSQL

Проверка I/O:

```
# Проверить latency диска
kubectl exec -it postgres-0 -- fio --name=randwrite --ioengine=libaio --
        iodepth=1 --rw=randwrite --bs=4k --direct=1 --size=1G --numjobs=1 --
        runtime=60
```

Проверка ресурсов:

```
# CPU/Memory пода
kubectl top pod postgres-0

# Requests/Limits
kubectl describe pod postgres-0 | grep -A 5 "Requests:\|Limits:"
```

Проверка сети:

```
# Latency между подами
kubectl exec -it app-pod -- ping postgres-primary
```

1.9.4 Полезные alias

Добавьте в `~/.bashrc` или `~/.zshrc`:

```
alias k='kubectl'
alias kgp='kubectl get pods'
alias kgs='kubectl get services'
alias kd='kubectl describe'
alias kl='kubectl logs'
alias kex='kubectl exec -it'
alias kpf='kubectl port-forward'
alias kctx='kubectl config use-context'
alias kns='kubectl config set-context --current --namespace'
```

1.9.5 Мониторинг PostgreSQL в Kubernetes

Минимальный набор метрик:

```
# Prometheus PostgreSQL Exporter
kubectl apply -f https://raw.githubusercontent.com/prometheus-operator/kube-
        prometheus/main/manifests/postgres-exporter-deployment.yaml
```

Ключевые метрики для DBA:

- `pg_up` — PostgreSQL доступен
- `pg_stat_database_tup_fetched` — строк прочитано
- `pg_stat_database_tup_inserted` — строк вставлено
- `pg_stat_replication_lag` — lag репликации
- `pg_locks_count` — количество блокировок
- `pg_database_size_bytes` — размер базы

1.10 Полезные ресурсы

1.10.1 Официальная документация

- **Kubernetes:** https://kubernetes.io/docs/

- **PostgreSQL Operators:**
  - Zalando: https://github.com/zalando/postgres-operator
  - CloudNativePG: https://cloudnative-pg.io/
  - Crunchy Data PGO: https://access.crunchydata.com/documentation/postgres-operator/

1.10.2 Сообщества

- **Data on Kubernetes (DoK):** https://dok.community/
  - Специализированное сообщество для баз данных в K8s
  - Ежемесячные вебинары с кейсами
  - Annual Report с бенчмарками и best practices
- **CNCF Slack:** https://cloud-native.slack.com/
  - Каналы: `#sig-storage`, `#postgres-operator`

1.10.3 Книги

- “Kubernetes in Action” (2nd Edition) — Marko Lukša
- “Production Kubernetes” — Josh Rosso et al.
- “PostgreSQL 14 Administration Cookbook” — Simon Riggs (раздел про K8s)

1.10.4 Инструменты

- **k9s** — TUI для управления кластером
- **Lens** — GUI для Kubernetes
- **Stern** — multi-pod tail логов
- **kubectx/kubens** — быстрое переключение контекстов и namespace

1.10.5 Тренировочные окружения

- **Minikube** — локальный кластер (Mac/Linux/Windows)
- **kind** — Kubernetes in Docker (быстрый старт)
- **k3s** — облегчённая версия (для edge/IoT)
- **Docker Desktop** — встроенный Kubernetes (Mac/Windows)

1.10.6 Онлайн-песочницы

- **Killercoda:** https://killercoda.com/kubernetes
- **Play with Kubernetes:** https://labs.play-with-k8s.com/

1.11 Заключение

1.11.1 Что вы теперь знаете

✅ **Мотивацию:** Почему Kubernetes становится стандартом для баз данных ✅ **Честные ограничения:** Когда K8s подходит, а когда нет ✅ **Архитектуру:** Как устроен кластер глазами DBA ✅ **Основные объекты:** Pod, ReplicaSet, Deployment, Service ✅ **Сетевую модель:** Как поды общаются между собой ✅ **Первые шаги с PostgreSQL:** Запуск и подключение ✅ **Путь к production:** Что нужно для реального использования

1.11.2 Что дальше

На следующем занятии:

- Persistent Volumes для надёжного хранения данных
- StatefulSet для управления PostgreSQL
- ConfigMap/Secret для конфигурации
- Развёртывание полноценного кластера с оператором

В самостоятельном изучении:

- Установите minikube локально
- Попробуйте запустить PostgreSQL по примерам из этого материала
- Поэкспериментируйте с масштабированием, удалением подов, Service

Для углублённого изучения:

- Пройдите официальный курс “Kubernetes for Beginners” (CNCF)
- Изучите один из операторов PostgreSQL (рекомендую CloudNativePG для начала)
- Подпишитесь на Data on Kubernetes Community для реальных кейсов

1.11.3 Последний совет

Kubernetes — это инструмент, не цель.

Не переносите PostgreSQL в K8s “потому что модно”. Делайте это, когда: - Это решает реальные проблемы вашей команды - У вас есть компетенция (или готовность учиться) - Вы понимаете trade-offs

Начните с экспериментов, накопите опыт, потом переносите критичные системы.

Удачи в изучении Kubernetes! Увидимся на следующем занятии.

Этот дополнительный материал подготовлен для курса “PostgreSQL Advanced” OTUS.