Хранение данных

Введение

Введение

Основные понятия

Основные понятия

Persistent Volume (PV)

Persistent Volume Claim (PVC)

Generic Ephemeral Volumes

emptyDir

hostPath

ConfigMap

Secret

StorageClass

Container Storage Interface (CSI)

Persistent Volume

Динамические Persistent Volumes и стат

Жизненный цикл Persistent Volumes

Режимы до

Режимы досту

Режимы томов

Особенности х

Заключение

Руководства

Создание Storage Class типа (

Развертывание Volume Plugin

Создание Storage Class

Создание класса блочного хр

Развертывание плагина тома

Создание класса хранилища

Создание л

Общая информ

Развертывание

Создание Stor

Последующие

Создание Shared Storage Clas

Предварительные требования

Развертывание плагина Alauda Containe

Создание Shared Storage Class для NFS

Развертывание компонента Volume Snapsh

Развертывание через веб-консоль

Развертывание через YAML

Создание P

Предварительны

Пример Persist

Создание PV ч

Создание PV ч

Использование снимков томов

Предварительные требования

Пример custom resource (CR) VolumeSnapshot

Создание снимков томов через веб-консоль

Создание снимков томов через CLI

Создание persistent volume claims из снимков томов

Дополнительные ресурсы

·кон

ью С

Дополнительные ресурсы

Как сделать

Generic ephemeral volumes

- Пример эфемерных томов
- Основные характеристики
- Когда использовать Generic Ephemeral \
- Чем они отличаются от emptyDir?

Использование emptyDir

- Пример emptyDir
- Необязательная настройка medium
- Основные характеристики
- Распространённые сценарии использов

Настройка томов

- Предварительны
- Процедура
- Автоматизация

Руководство по аннотированию возможностей стороннего хранилища

1. Начало работы
2. Пример ConfigMap
3. Обновление существующих описаний возможностей
4. Совместимость с устаревшим форматом
5. Часто задаваемые вопросы

ка

- гия
- елы
- ание
- зап
- одн
- ава
- Плановая мигр
- Failover
- Failback (post-c

Устранение неполадок

Восстановление после ошибки расширения PVC

Процедура

Дополнительные советы

Объектное хранилище

Введение

Ограничения

Основные понятия

Введение в основные ресурсы и принци администраторов Kubernetes.

Обзор

Установка

Предварительны

Установка Alau

Руководства

Как сделать

Введение

Kubernetes предлагает гибкий и масштабируемый механизм хранения для управления сохранением данных в контейнеризованных средах. Абстрагируя ресурсы хранения, такие как Volumes, PersistentVolumes и PersistentVolumeClaims, Kubernetes отделяет приложения от базовых систем хранения, обеспечивая динамическое выделение ресурсов, автоматическое монтирование и сохранение данных между узлами.

Ключевые возможности включают поддержку множества систем хранения (например, локальные диски, NFS, облачные сервисы хранения), динамическое выделение, управление режимами доступа (такими как права на чтение/запись) и управление жизненным циклом — что удовлетворяет потребности stateful-приложений в хранении данных. Для корпоративных рабочих нагрузок, требующих высокой доступности, сохранения данных и изоляции между арендаторами, хранение в Kubernetes является важной базовой функцией.

Хранение в Kubernetes разработано для разработчиков, инженеров эксплуатации и платформенных команд, помогая им эффективно и безопасно управлять данными в контейнеризованных рабочих нагрузках.

Основные понятия

Основные понятия

Persistent Volume (PV)

Persistent Volume Claim (PVC)

Generic Ephemeral Volumes

emptyDir

hostPath

ConfigMap

Secret

StorageClass

Container Storage Interface (CSI)

Persistent Volume

Динамические Persistent Volumes и стат

Жизненный цикл Persistent Volumes

Режимы до

Режимы досту

Режимы томов

Особенности х

Заключение

Основные понятия

Хранение данных в Kubernetes основано на трёх ключевых концепциях: **PersistentVolume (PV)**, **PersistentVolumeClaim (PVC)** и **StorageClass**. Они определяют, как запрашивается, выделяется и настраивается хранилище внутри кластера. В основе работы часто лежат драйверы **CSI** (Container Storage Interface), которые обеспечивают фактическое предоставление и подключение хранилища. Давайте кратко рассмотрим каждый компонент и выделим роль CSI-драйвера.

Содержание

Persistent Volume (PV)
Persistent Volume Claim (PVC)
Generic Ephemeral Volumes
emptyDir
hostPath
ConfigMap
Secret
StorageClass
Container Storage Interface (CSI)

Persistent Volume (PV)

PersistentVolume (PV) — это часть хранилища в кластере, которая была выделена (либо статически администратором, либо динамически через StorageClass). Он представляет собой базовое хранилище — например, диск у облачного провайдера или файловую систему с сетевым доступом — и рассматривается как ресурс кластера, аналогично узлу.

Persistent Volume Claim (PVC)

PersistentVolumeClaim (PVC) — это запрос на хранилище. Пользователи определяют, сколько хранилища им нужно и режим доступа (например, чтение-запись). Если подходящий PV доступен или может быть динамически создан (через StorageClass), PVC связывается с этим PV. После связывания Pod'ы могут ссылаться на PVC для сохранения или совместного использования данных.

Generic Ephemeral Volumes

Generic Ephemeral Volumes для Kubernetes — это функция, введённая в Kubernetes, которая позволяет использовать CSI-управляемые `временные` тома в течение жизненного цикла Pod, аналогично `emptyDir`, но более мощная и позволяющая монтировать любой тип CSI-тома (с поддержкой снимков, масштабирования и т.д.).

Для дополнительной информации смотрите Generic ephemeral volumes

emptyDir

1. emptyDir — это временный том типа пустой директории.
2. Он создаётся при запуске Pod на узле, и хранилище располагается на локальной файловой системе этого узла (по умолчанию — диск узла).
3. При удалении Pod данные в emptyDir также удаляются.

Для дополнительной информации смотрите Using an emptyDir

hostPath

В Kubernetes том hostPath — это специальный тип тома, который отображает файл или директорию из файловой системы хоста непосредственно в контейнер Pod.

- Позволяет Pod получить доступ к файлам или директориям на узле-хосте.
- Полезен для:
  - Доступа к ресурсам уровня хоста
  - Отладки
  - Использования уже существующих данных на узле

ConfigMap

ConfigMap в Kubernetes — это объект API, используемый для хранения неконфиденциальных данных конфигурации в виде пар ключ-значение. Он позволяет отделить конфигурацию от кода приложения, делая приложения более переносимыми и удобными в управлении.

Secret

В Kubernetes Secret — это объект API, который хранит конфиденциальные данные, такие как:

- пароли
- OAuth-токены
- SSH-ключи
- TLS-сертификаты
- учётные данные баз данных

Secrets помогают защитить эти данные, избегая их прямого хранения в спецификациях Pod или образах контейнеров.

StorageClass

StorageClass описывает *как* тома должны динамически выделяться. Он соответствует конкретному провижионеру (часто CSI-драйверу) и может включать параметры, такие как уровни хранения, характеристики производительности или другие настройки бэкенда. Создавая несколько StorageClass, можно предложить разработчикам различные типы хранилища.

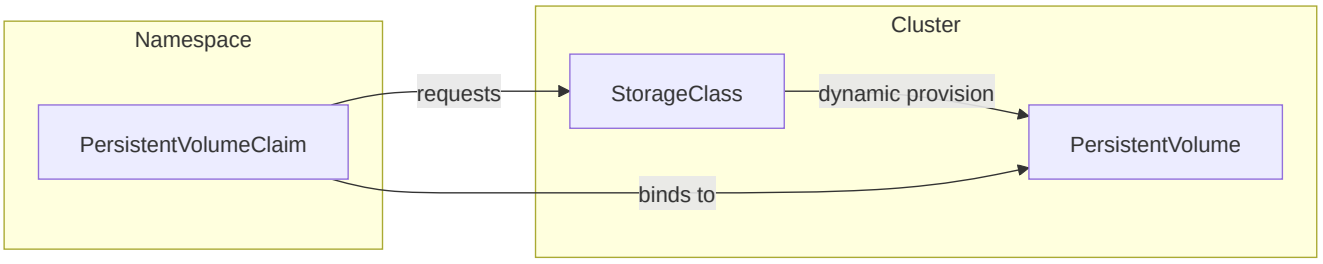

Диаграмма: Взаимосвязь между PVC, PV и StorageClass.

Container Storage Interface (CSI)

Container Storage Interface (CSI) — это стандартный API, который Kubernetes использует для интеграции с драйверами хранилища. Он позволяет сторонним поставщикам создавать плагины вне ядра Kubernetes, то есть вы можете устанавливать или обновлять драйверы хранилища без изменения самого Kubernetes.

CSI **драйвер** обычно состоит из двух компонентов:

1. **Компонент контроллера**: работает в кластере (часто как Deployment) и отвечает за операции высокого уровня, такие как **создание** или **удаление** томов. Для сетевого хранилища он также может управлять подключением и отключением томов к узлам.
2. **Компонент узла**: работает на каждом узле (часто как DaemonSet) и отвечает за **монтирование** и **отмонтирование** тома на конкретном узле. Он взаимодействует с kubelet, чтобы обеспечить доступность тома для Pod.

Когда пользователь создаёт PVC, ссылающийся на StorageClass с CSI-драйвером, драйвер отслеживает этот запрос и выделяет хранилище при необходимости динамического провижининга. После создания хранилища драйвер уведомляет Kubernetes, который создаёт соответствующий PV и связывает его с PVC. Когда Pod

использует этот PVC, компонент узла драйвера обрабатывает монтирование тома, делая хранилище доступным внутри контейнера.

Используя **PV**, **PVC**, **StorageClass** и **CSI**, Kubernetes обеспечивает мощный декларативный подход к управлению хранилищем. Администраторы могут определить один или несколько StorageClass для представления различных бэкендов или уровней производительности, а разработчики просто запрашивают хранилище через PVC — не заботясь о внутренней инфраструктуре.

Persistent Volume

PersistentVolume (PV) представляет собой объект API Kubernetes, отображающий связь с томами бекенд-хранилища в кластере Kubernetes. Это ресурс кластера, который создаётся и настраивается администраторами единообразно, отвечая за абстрагирование реальных ресурсов хранения и формирование инфраструктуры хранения кластера.

PersistentVolumes обладают жизненным циклом, независимым от Pod, что позволяет сохранять данные Pod постоянно.

Администраторы могут вручную создавать статические PersistentVolumes или генерировать динамические PersistentVolumes на основе классов хранения. Если разработчикам необходимо получить ресурсы хранения для приложений, они могут запросить их через PersistentVolumeClaims (PVC), которые сопоставляются и связываются с подходящими PersistentVolumes.

Содержание

Динамические Persistent Volumes и статические Persistent Volumes

Жизненный цикл Persistent Volumes

Динамические Persistent Volumes и статические Persistent Volumes

Платформа поддерживает управление двумя типами PersistentVolumes администраторами: динамическими и статическими Persistent Volumes.

- **Динамические Persistent Volumes**: Реализуются на основе классов хранения. Классы хранения создаются администраторами и представляют собой ресурс Kubernetes, описывающий категорию ресурсов хранения. Как только разработчик создаёт PersistentVolumeClaim, связанный с классом хранения, платформа динамически создаёт подходящий PersistentVolume согласно параметрам, настроенным в PersistentVolumeClaim и классе хранения, связывая его с PersistentVolumeClaim для динамического выделения ресурсов хранения.
- **Статические Persistent Volumes**: Persistent Volumes, создаваемые вручную администратором. В настоящее время поддерживается создание статических Persistent Volumes типа **HostPath** или **NFS shared storage**. Когда разработчики создают PersistentVolumeClaim без использования класса хранения, платформа сопоставляет и связывает подходящий статический PersistentVolume согласно параметрам, настроенным в PersistentVolumeClaim.
  - **HostPath**: Использует файловый каталог на хосте узла (локальное хранилище не поддерживается) в качестве бекенд-хранилища, например: `/etc/kubernetes`. Обычно применяется только для тестовых сценариев в кластере с одним вычислительным узлом.
  - **NFS Shared Storage**: Относится к Network File System, распространённому типу бекенд-хранилища для Persistent Volumes. Пользователи и программы могут обращаться к файлам на удалённых системах так, как если бы они были локальными файлами.

Жизненный цикл Persistent Volumes

1. **Provisioning (создание)**: Администраторы вручную создают статические Persistent Volumes. После создания Persistent Volume переходит в состояние **Available**; альтернативно, платформа динамически создаёт подходящие Persistent Volumes на основе PersistentVolumeClaims, связанных с классами хранения.
2. **Binding (связывание)**: Как только статический Persistent Volume сопоставлен и связан с PersistentVolumeClaim, он переходит в состояние **Bound**; динамические Persistent Volumes создаются динамически на основе запросов, соответствующих

PersistentVolumeClaims, и также переходят в состояние **Bound** после успешного создания.

3. **Using (использование)**: Разработчики связывают PersistentVolumeClaims с экземплярами контейнеров вычислительных компонентов, используя ресурсы бекенд-хранилища, отображённые Persistent Volumes.
4. **Releasing (освобождение)**: После удаления PersistentVolumeClaim разработчиками Persistent Volume освобождается.
5. **Reclaiming (восстановление)**: После освобождения Persistent Volume на нём выполняются операции восстановления в соответствии с параметрами политики восстановления Persistent Volume или класса хранения.

Режимы доступа и режимы томов

В Kubernetes PersistentVolumeClaims (PVC) и StorageClasses работают вместе для управления тем, как хранилище предоставляется и используется рабочими нагрузками. Два ключевых понятия в этой области — это **режимы доступа** и **режимы томов**. В этой статье рассматриваются эти понятия и подчеркивается, как различные системы хранения поддерживают их.

Содержание

Режимы доступа в Kubernetes
Режимы доступа по StorageClass
Режимы томов в Kubernetes
Режимы томов по StorageClass
Особенности хранения: снимки и расширение
Заключение

Режимы доступа в Kubernetes

Режимы доступа определяют, как том может быть смонтирован и использован подами. Основные режимы доступа:

- **ReadWriteOnce (RWO)**: том может быть смонтирован в режиме чтения-записи одним узлом.

- **ReadOnlyMany (ROX)**: том может быть смонтирован в режиме только для чтения несколькими узлами.
- **ReadWriteMany (RWX)**: том может быть смонтирован в режиме чтения-записи несколькими узлами.

Режимы доступа по StorageClass

Storage Class	Поддержка RWO	Поддержка ROX	Поддержка RWX
CephFS File Storage	Да	Нет	Да
CephRBD Block Storage	Да	Нет	Нет
TopoLVM	Да	Нет	Нет
NFS Shared Storage	Да	Нет	Да

Как показано выше, файловые системы хранения, такие как **CephFS** и **NFS**, поддерживают несколько одновременных операций записи или чтения, что делает их подходящими для сценариев совместного доступа. С другой стороны, блочные системы хранения, такие как **CephRBD** и **TopoLVM**, обеспечивают эксклюзивный доступ одному узлу за раз.

Режимы томов в Kubernetes

Режимы томов определяют, как данные представлены поду:

- **Filesystem**: том монтируется в под как файловая система.
- **Block**: том представлен как необработанное блочное устройство.

Режимы томов по StorageClass

Storage Class	Тип	Поддерживаемые режимы томов
CephFS File Storage	File Storage	Filesystem
CephRBD Block Storage	Block Storage	Filesystem, Block
TopoLVM	Block Storage	Filesystem, Block
NFS Shared Storage	File Storage	Filesystem

Блочные системы хранения, такие как **CephRBD** и **TopoLVM**, предлагают как доступ через файловую систему, так и через необработанный блочный доступ, обеспечивая гибкость для различных потребностей приложений. Файловые системы хранения, такие как **CephFS** и **NFS**, напротив, поддерживают только режим файловой системы.

Особенности хранения: снимки и расширение

Kubernetes также поддерживает расширенные возможности, такие как создание снимков томов и динамическое расширение PVC, в зависимости от используемого StorageClass.

Storage Class	Снимок тома	Расширение
CephFS File Storage	Поддерживается	Поддерживается
CephRBD Block Storage	Поддерживается	Поддерживается
TopoLVM	Поддерживается	Поддерживается
NFS Shared Storage	Не поддерживается	Не поддерживается

Снимки томов поддерживаются только для PVC, динамически созданных с использованием StorageClass. Эта функция полезна для резервного копирования и клонирования окружений.

Заключение

При настройке хранилища в Kubernetes понимание **режимов доступа** и **режимов томов** PVC и соответствующих **StorageClasses** критично для выбора правильного решения для вашей рабочей нагрузки. Файловые решения хранения, такие как CephFS и NFS, идеально подходят для сценариев совместного доступа, тогда как блочные системы хранения, такие как CephRBD и TopoLVM, превосходят в высокопроизводительных развертываниях на одном узле. Кроме того, поддержка таких функций, как снимки и расширение, значительно повышает гибкость хранения и стратегии управления данными.

Руководства

Создание Storage Class типа (

Развертывание Volume Plugin

Создание Storage Class

Создание класса блочного хр

Развертывание плагина тома

Создание класса хранилища

Создание л

Общая информ

Развертывание

Создание Stor

Последующие

Создание Shared Storage Clas

Предварительные требования

Развертывание плагина Alauda Containe

Создание Shared Storage Class для NFS

Развертывание компонента Volume Snapsh

Развертывание через веб-консоль

Развертывание через YAML

Создание P

Предварительны

Пример Persist

Создание PV ч

Создание PV ч

Использование снимков томов

Предварительные требования

Пример custom resource (CR) VolumeSnapshot

Создание снимков томов через веб-консоль

Создание снимков томов через CLI

Создание persistent volume claims из снимков томов

Дополнительные ресурсы

·кон

ью С

Дополнительные ресурсы

Создание Storage Class типа CephFS File Storage

CephFS file storage — это встроенная файловая система Ceph, которая предоставляет платформе метод доступа к хранилищу на основе Container Storage Interface (CSI), обеспечивая безопасный, надежный и масштабируемый общий файловый сервис, подходящий для сценариев, таких как совместное использование файлов и резервное копирование данных. Перед началом необходимо сначала создать Storage Class для CephFS file storage.

После привязки Storage Class в Persistent Volume Claim (PVC) платформа будет динамически создавать persistent volumes на узлах в соответствии с запросом на persistent volume для бизнес-приложений.

Содержание

Развертывание Volume Plugin

Создание Storage Class

Развертывание Volume Plugin

После нажатия **Deploy** на странице **Distributed Storage** выполните Создание Storage Service или Доступ к Storage Service.

Создание Storage Class

1. Перейдите в раздел **Administrator**.
2. В левой навигационной панели выберите **Storage Management** > **Storage Classes**.
3. Нажмите **Create Storage Class**.
   Примечание: Следующий пример приведён в форме; вы также можете создать Storage Class с помощью YAML.
4. Выберите **CephFS File Storage** и нажмите **Next**.
5. Настройте соответствующие параметры согласно следующим инструкциям.

Параметр	Описание
Reclaim Policy	Политика восстановления для persistent volumes. - Delete: При удалении persistent volume claim связанный persistent volume также будет удалён. - Retain: Связанный persistent volume останется, даже если persistent volume claim удалён.
Access Modes	Все режимы доступа, поддерживаемые текущим хранилищем. При объявлении persistent volumes можно выбрать только один из этих режимов. - ReadWriteOnce (RWO): Может быть смонтирован для чтения и записи одним узлом. - ReadWriteMany (RWX): Может быть смонтирован для чтения и записи несколькими узлами.
Allocate Project	Укажите проекты, которые могут использовать этот тип хранилища. Если в данный момент нет проектов, которым необходимо использовать этот тип хранилища, можно не выделять их сейчас и обновить позже.

Совет: Следующие параметры необходимо задать в распределённом хранилище, они будут применены здесь напрямую.

- Storage Cluster: Встроенный Ceph storage cluster в текущем кластере.

- Storage Pool: Логический раздел, используемый для хранения данных в storage cluster.

6. Нажмите **Create**.

Создание класса блочного хранилища CephRBD

CephRBD блочное хранилище — это встроенное блочное хранилище Ceph для платформы, предоставляющее метод доступа к хранилищу на основе Container Storage Interface (CSI), способный обеспечивать высокие IOPS и низкую задержку, что подходит для сценариев, таких как базы данных и виртуализация. Перед использованием необходимо создать класс блочного хранилища CephRBD.

После того как Persistent Volume Claim (PVC) будет привязан к классу хранилища, платформа динамически создаст Persistent Volume на основе Persistent Volume Claim для использования бизнес-приложениями.

Содержание

Развертывание плагина тома

Создание класса хранилища

Развертывание плагина тома

После нажатия **Deploy** на странице **Distributed Storage** создайте сервис хранения или получите доступ к сервису хранения.

Создание класса хранилища

1. Перейдите в раздел **Administrator**.
2. В левой навигационной панели нажмите **Storage Management** > **Storage Classes**.
3. Нажмите **Create Storage Class**.
   Примечание: Следующий пример приведён в форме, вы также можете выбрать YAML для выполнения операции.
4. Выберите **CephRBD Block Storage** и нажмите **Next**.
5. Настройте параметры согласно требованиям.

Параметр	Описание
File System	По умолчанию **EXT4** — журналируемая файловая система для Linux, способная обеспечивать хранение экстентов и обработку больших файлов. Вместимость файловой системы может достигать 1 EiB, поддерживаемый размер файла — до 16 TiB.
Reclaim Policy	Политика возврата для persistent volumes. - Delete: связанный persistent volume будет удалён вместе с persistent volume claim. - Retain: связанный persistent volume сохранится даже при удалении persistent volume claim.
Access Modes	Поддерживается только ReadWriteOnce (RWO): может быть смонтирован одним узлом в режиме чтения и записи.
Assign Project	Назначьте проекты, которые смогут использовать этот тип хранилища. Если в данный момент нет проектов, нуждающихся в этом типе хранилища, можно не назначать проект и обновить настройки позже.

Совет: Следующие параметры необходимо задать в распределённом хранилище, они будут применены здесь напрямую.

- Storage Cluster: встроенный Ceph storage cluster в текущем кластере.
- Storage Pool: логический раздел, используемый для хранения данных внутри storage cluster.

6. Нажмите **Create**.

Создание локального Storage Class TopoLVM

TopoLVM — это локальное решение для хранения на базе LVM, которое обеспечивает простые, удобные в обслуживании и высокопроизводительные локальные сервисы хранения, подходящие для сценариев, таких как базы данных и middleware. Перед использованием необходимо создать Storage Class TopoLVM.

После того как Persistent Volume Claim (PVC) будет привязан к этому Storage Class, платформа динамически создаст persistent volumes на узлах на основе PVC для использования бизнес-приложениями.

Содержание

- Общая информация
  - Преимущества использования
  - Сценарии использования
  - Ограничения и предостережения
- Развертывание Volume Plugin
- Создание Storage Class
- Последующие действия

Общая информация

Преимущества использования

- По сравнению с удалённым хранилищем (например, **NFS shared storage**): хранилище типа TopoLVM расположено локально на узле, что обеспечивает лучшие показатели IOPS и пропускной способности, а также меньшую задержку.
- По сравнению с hostPath (например, **local-path**): хотя оба варианта являются локальным хранилищем на узле, TopoLVM позволяет гибко планировать размещение контейнерных групп на узлах с достаточными доступными ресурсами, избегая ситуаций, когда контейнерные группы не могут запуститься из-за нехватки ресурсов.
- TopoLVM по умолчанию поддерживает автоматическое расширение томов. После изменения требуемой квоты хранения в Persistent Volume Claim расширение происходит автоматически без перезапуска контейнерной группы.

Сценарии использования

- Когда требуется только временное хранилище, например, для разработки и отладки.
- При высоких требованиях к I/O хранилища, например, для индексирования в реальном времени.

Ограничения и предостережения

Рекомендуется использовать локальное хранилище только для приложений, где возможно реализовать репликацию и резервное копирование данных на уровне приложения, например, MySQL. Избегайте потери данных из-за отсутствия гарантии сохранности данных в локальном хранилище.

Узнать больше ↗

Развертывание Volume Plugin

После нажатия кнопки deploy на открывшейся странице настройте локальное хранилище.

Создание Storage Class

1. Перейдите в раздел **Администратор**.
2. В левой навигационной панели выберите **Управление хранилищем** > **Storage Classes**.
3. Нажмите **Создать Storage Class**.
4. Выберите **Block Storage**.
5. Выберите **TopoLVM**, затем нажмите **Далее**.
6. Настройте параметры Storage Class, как описано ниже.
   Примечание: Следующий пример представлен в виде формы; вы также можете создать Storage Class с помощью YAML.

Параметр	Описание
Name	Имя Storage Class, которое должно быть уникальным в пределах текущего кластера.
Display Name	Имя, которое поможет вам идентифицировать или фильтровать Storage Class, например, описание на русском языке.
Device Class	Device Class — это способ классификации устройств хранения в TopoLVM, где каждый класс соответствует группе устройств с похожими характеристиками. Если нет специальных требований, используйте Device Class **Automatically Assigned**.
File System	• **XFS** — высокопроизводительная журналируемая файловая система, хорошо подходящая для параллельных I/O нагрузок, поддерживает работу с большими файлами и обеспечивает плавную передачу данных. • **EXT4** — журналируемая файловая система в Linux, предоставляющая extent-хранение файлов и поддержку

Параметр	Описание
	больших файлов, с максимальной ёмкостью файловой системы 1 EiB и максимальным размером файла 16 TiB.
Reclamation Policy	Политика освобождения persistent volumes. • Delete: связанный persistent volume будет удалён вместе с PVC. • Retain: связанный persistent volume останется даже после удаления PVC.
Access Mode	ReadWriteOnce (RWO): может быть смонтирован в режиме чтения-записи только одним узлом.
PVC Reconstruction	Поддержка реконструкции PVC между узлами. При включении необходимо настроить **Reconstruction Wait Time**. Если узел, на котором размещён PVC, созданный с этим Storage Class, выходит из строя, PVC автоматически восстанавливается на других узлах после указанного времени ожидания для обеспечения непрерывности работы. **Примечание**: • Восстановленный PVC не содержит исходных данных. • Убедитесь, что количество узлов хранения больше количества реплик экземпляров приложения, иначе это повлияет на реконструкцию PVC.
Allocated Projects	PVC этого типа можно создавать только в определённых проектах. Если проекты ещё не назначены, их можно **обновить позже**.

7. После проверки правильности конфигурации нажмите кнопку **Создать**.

Последующие действия

Когда всё будет готово, вы можете уведомить разработчиков о возможности использования функций TopoLVM. Например, создать Persistent Volume Claim и привязать его к Storage Class TopoLVM на странице **Storage** > **Persistent Volume Claims** в контейнерной платформе.

Создание Shared Storage Class для NFS

На основе community-драйвера NFS CSI (Container Storage Interface) предоставляется возможность доступа к нескольким NFS системам хранения или аккаунтам.

В отличие от традиционной клиент-серверной модели доступа к NFS, Shared Storage NFS использует community-плагин NFS CSI (Container Storage Interface), который лучше соответствует принципам проектирования Kubernetes и позволяет клиентам обращаться к нескольким серверам.

Содержание

Предварительные требования

Развертывание плагина Alauda Container Platform NFS CSI

Развертывание через веб-консоль

Развертывание через YAML

Создание Shared Storage Class для NFS

Предварительные требования

- Должен быть настроен NFS сервер, а также получены методы доступа к нему. В настоящее время платформа поддерживает три версии протокола NFS: `v3`, `v4.0` и `v4.1`. Вы можете выполнить команду `nfsstat -s` на стороне сервера для проверки версии.

Развертывание плагина Alauda Container Platform NFS CSI

Развертывание через веб-консоль

1. Войдите в систему как **Administrator**.
2. В левой навигационной панели нажмите **Storage** > **StorageClasses**.
3. Нажмите **Create StorageClass**.
4. Справа от **NFS CSI** нажмите Deploy, чтобы перейти на страницу **Plugins**.
5. Справа от плагина `Alauda Container Platform NFS CSI` нажмите ⋮ > **Install**.
6. Дождитесь, пока статус развертывания не изменится на **Deployment Successful**, после чего завершите развертывание.

Развертывание через YAML

См. Installing via YAML

`Alauda Container Platform NFS CSI` является **Non-config plugin**, имя модуля — `nfs`

Создание Shared Storage Class для NFS

1. Нажмите **Create Storage Class**.
   Примечание: Следующий контент представлен в виде формы, но вы также можете выполнить операцию с помощью YAML.
2. Выберите **NFS CSI** и нажмите **Next**.
3. Настройте параметры согласно следующим инструкциям.

Параметр	Описание
Name	Имя класса хранения. Должно быть уникальным в пределах текущего кластера.

Параметр	Описание
Service Address	Адрес доступа к NFS серверу. Например: `192.168.2.11`.
Path	Путь монтирования файловой системы NFS на сервере. Например: `/nfs/data`.
NFS Protocol Version	В настоящее время поддерживаются три версии: `v3`, `v4.0` и `v4.1`.
Reclaim Policy	Политика восстановления для persistent volume. - Delete: при удалении persistent volume claim будет также удалён связанный persistent volume. - Retain: даже при удалении persistent volume claim связанный persistent volume сохраняется.
Access Modes	Все режимы доступа, поддерживаемые текущим хранилищем. При последующем объявлении persistent volume можно выбрать только один из этих режимов для монтирования. - ReadWriteOnce (RWO): может быть смонтирован как для чтения и записи одним узлом. - ReadWriteMany (RWX): может быть смонтирован как для чтения и записи несколькими узлами. - ReadOnlyMany (ROX): может быть смонтирован только для чтения несколькими узлами.
Allocated Projects	Укажите проекты, которым разрешено использовать данный тип хранилища. Если в данный момент нет проектов, нуждающихся в этом типе хранилища, можно не выделять проекты сейчас и обновить позже.
subDir	Каждый PersistentVolumeClaim (PVC), созданный с использованием Shared Storage Class NFS, соответствует поддиректории внутри NFS шаринга. По умолчанию поддиректории именуются по шаблону `${pv.metadata.name}` (то есть именем PersistentVolume). Если имя по

Параметр	Описание
	умолчанию не подходит, можно настроить правила именования поддиректорий.

NOTE

Поле `subDir` поддерживает только следующие три переменные, которые автоматически разрешаются драйвером NFS CSI:

- `${pvc.metadata.namespace}`: Namespace PVC.
- `${pvc.metadata.name}`: Имя PVC.
- `${pv.metadata.name}`: Имя PV.

Правило именования `subDir` **ДОЛЖНО** гарантировать уникальность имён поддиректорий. В противном случае несколько PVC могут использовать одну и ту же поддиректорию, что приведёт к конфликтам данных.

Рекомендуемые конфигурации:

- `${pvc.metadata.namespace}_${pvc.metadata.name}_${pv.metadata.name}`
- `<cluster-identifier>_${pvc.metadata.namespace}_${pvc.metadata.name}_${pv.metadata.name}`

Предназначено для нескольких Kubernetes кластеров, использующих один и тот же NFS сервер. Такая конфигурация обеспечивает чёткое разделение кластеров за счёт включения идентификатора кластера (например, имени кластера) в правило именования поддиректорий.

Не рекомендуемые конфигурации:

- `${pvc.metadata.namespace}-${pvc.metadata.name}-${pv.metadata.name}` Не используйте дефис (-) в качестве разделителя, так как это может привести к неоднозначности имён поддиректорий. Например: если два PVC называются `ns-1/test` и `ns/1-test`, оба могут породить одинаковую поддиректорию `ns-1-test`.
- `${pvc.metadata.namespace}/${pvc.metadata.name}/${pv.metadata.name}` Не настраивайте `subDir` для создания вложенных директорий. Драйвер NFS CSI

удаляет только последний уровень директории `${pv.metadata.name}` при удалении PVC, оставляя родительские директории сиротами на NFS сервере.

4. После подтверждения правильности конфигурации нажмите **Create**.

Развертывание компонента Volume Snapshot

Volume snapshot — это снимок persistent volume, представляющий собой копию persistent volume в определённый момент времени. Если в кластере используются persistent volumes с поддержкой функции snapshot, можно развернуть компонент volume snapshot для включения этой возможности.

В настоящее время платформа поддерживает создание volume snapshots только для PVC, которые **динамически создаются** с использованием storage classes. На основе этих снимков можно создавать новые привязки PVC.

Совет: Режимы доступа, поддерживаемые при создании PVC из снимков, отличаются от тех, что поддерживаются при создании PVC с помощью storage classes, и выделены **жирным** в таблице ниже.

Storage Class, используемый для создания Volume Snapshots	**Single Node Read-Write (RWO)**	**Multi-Node Read-Only (ROX)**	**Multi-Node Read-Write (RWX)**
TopoLVM	Поддерживается	Не поддерживается	Не поддерживается
CephRBD Block Storage	Поддерживается	Не поддерживается	Не поддерживается
CephFS File Storage	Поддерживается	**Поддерживается**	Поддерживается

Содержание

Развертывание через веб-консоль

Развертывание через YAML

Развертывание через веб-консоль

1. Перейдите в раздел **Administrator**.
2. Нажмите **Marketplace** > **Cluster Plugins**, чтобы открыть страницу списка **Cluster Plugins**.
3. Найдите плагин кластера `Alauda Container Platform Snapshot Management`, нажмите **Install** и дождитесь успешного завершения развертывания.

Развертывание через YAML

См. Installing via YAML

`Alauda Container Platform Snapshot Management` — это **Non-config plugin**, имя модуля — `snapshot`

Создание PV

Ручное создание статического persistent volume типа **HostPath** или **NFS Shared Storage**.

- **HostPath**: Монтирует файловый каталог с хоста, на котором находится контейнер, в указанный путь внутри контейнера (соответствует Kubernetes HostPath), позволяя контейнеру использовать файловую систему хоста для постоянного хранения. Если хост становится недоступен, HostPath может стать недоступен.
- **NFS Shared Storage**: NFS Shared Storage использует общественный плагин хранения NFS CSI (Container Storage Interface), который более соответствует принципам проектирования Kubernetes, предоставляя возможности клиентского доступа для нескольких сервисов. Убедитесь, что в текущем кластере задеплоен **NFS storage plugin** перед использованием.

Содержание

Предварительные требования
Пример PersistentVolume
Создание PV через веб-консоль
Информация о хранилище
Создание PV через CLI
Режимы доступа
Политики возврата (Reclaim Policies)
Связанные операции
Дополнительные ресурсы

Предварительные требования

- Подтвердите размер создаваемого persistent volume и убедитесь, что бекенд-хранилище в данный момент имеет возможность предоставить соответствующий объем.
- Получите адрес доступа к бекенд-хранилищу, путь к монтируемой директории, учетные данные доступа (если требуются) и другую соответствующую информацию.

Пример PersistentVolume

```yaml
# example-pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 5Gi (1)
  accessModes:
    - ReadWriteOnce (2)
  persistentVolumeReclaimPolicy: Retain (3)
  storageClassName: manual (4)
  hostPath: (5)
    path: "/mnt/data"
```

1. Объем хранилища.
2. Способ монтирования тома.
3. Что происходит после удаления PVC (Retain, Delete, Recycle).
4. Имя StorageClass (для динамического связывания).
5. Тип бекенд-хранилища.

Создание PV через веб-консоль

1. Перейдите в раздел **Administrator**.
2. В левой навигационной панели нажмите **Storage Management** > **Persistent Volumes (PV)**.
3. Нажмите **Create Persistent Volume**.
4. Ознакомьтесь с инструкциями ниже и настройте параметры перед нажатием **Create**.

Информация о хранилище

Тип	Параметр	Описание
HostPath	Path	Путь к каталогу файлов на узле, который поддерживает том хранилища. Например: `/etc/kubernetes`.
NFS Shared Storage	Server Address	Адрес доступа к NFS-серверу.
	Path	Путь монтирования файловой системы NFS на серверном узле, например `/nfs/data`.
	NFS Protocol Version	В настоящее время платформа поддерживает версии протокола NFS: `v3`, `v4.0` и `v4.1`. Вы можете выполнить `nfsstat -s` на стороне сервера для просмотра информации о версиях.

Создание PV через CLI

```
kubectl apply -f example-pv.yaml
```

Режимы доступа

Режимы доступа persistent volume зависят от соответствующих параметров, заданных бекенд-хранилищем.

Режим доступа	Значение
ReadWriteOnce (RWO)	Может быть смонтирован для чтения и записи одним узлом.
ReadWriteMany (RWX)	Может быть смонтирован для чтения и записи несколькими узлами.
ReadOnlyMany (ROX)	Может быть смонтирован только для чтения несколькими узлами.

Политики возврата (Reclaim Policies)

Политика возврата	Значение
Delete	При удалении persistent volume claim одновременно удаляется связанный persistent volume, а также ресурс бекенд-хранилища. **Примечание**: политика возврата для PV типа NFS Shared Storage не поддерживает **Delete**.
Retain	Даже при удалении persistent volume claim связанный persistent volume и данные хранилища сохраняются. Требуется ручное управление данными и последующее удаление persistent volume.

Связанные операции

Вы можете нажать ⋮ справа на странице списка или нажать **Operations** в правом верхнем углу страницы деталей для обновления или удаления persistent volume по необходимости.

Удаление persistent volume применимо в следующих двух сценариях:

- Удаление несвязанного persistent volume: том не использовался для записи и больше не требуется для записи, что освобождает соответствующее пространство хранилища при удалении.

- Удаление persistent volume с политикой **Retain**: persistent volume claim был удалён, но из-за политики retain reclaim policy том не был удалён одновременно. Если данные persistent volume были сохранены в другое хранилище или больше не нужны, удаление также освободит соответствующее пространство хранилища.

Дополнительные ресурсы

- Creating PVCs

Создание PVC

Создайте PersistentVolumeClaim (PVC) и при необходимости задайте параметры для запрашиваемого PersistentVolume (PV).

Вы можете создать PersistentVolumeClaim либо через визуальную форму UI, либо с помощью пользовательского YAML-файла оркестрации.

Содержание

Предварительные требования

Пример PersistentVolumeClaim:

Создание Persistent Volume Claim через веб-консоль

Создание Persistent Volume Claim с помощью CLI

Операции

Расширение ёмкости PersistentVolumeClaim через веб-консоль

Расширение ёмкости Persistent Volume Claim с помощью CLI

Дополнительные ресурсы

Предварительные требования

Убедитесь, что в namespace достаточно оставшейся квоты **хранилища** для удовлетворения требуемого размера хранилища для этой операции создания.

Пример PersistentVolumeClaim:

```yaml
# example-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pvc
  namespace: k-1
  annotations: {}
  labels: {}
spec:
  storageClassName: cephfs
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 4Gi
```

Создание Persistent Volume Claim через веб-консоль

1. Перейдите в **Container Platform**.
2. В левой боковой панели выберите **Storage > PersistentVolumeClaims (PVC)**.
3. Нажмите **Create PVC**.
4. Настройте параметры по необходимости.

   Примечание: Следующий контент приведён в качестве примера с использованием формы; вы также можете переключиться в режим YAML для выполнения операции.

Параметр	Описание
Name	Имя PersistentVolumeClaim, которое должно быть уникальным в пределах текущего namespace.

Параметр	Описание
Creation Method	- Dynamic Creation: Динамически создаёт PersistentVolume на основе storage class и связывает его. - Static Binding: Выполняет сопоставление и связывание на основе настроенных параметров и существующих PersistentVolumes.
Storage Class	После выбора метода динамического создания платформа динамически создаст PersistentVolume в соответствии с описанием в указанном storage class.
Access Mode	- ReadWriteOnce (RWO): Может быть смонтирован одним узлом в режиме чтения-записи. - ReadWriteMany (RWX): Может быть смонтирован несколькими узлами в режиме чтения-записи. - ReadOnlyMany (ROX): Может быть смонтирован несколькими узлами в режиме только для чтения. **Совет**: Рекомендуется учитывать количество экземпляров workload, которые планируется привязать к текущему PersistentVolumeClaim, а также тип контроллера развертывания. Например, при создании мультиэкземплярного развертывания (Deployment), поскольку все экземпляры используют один PersistentVolumeClaim, не рекомендуется выбирать режим доступа RWO, который может быть подключён только к одному узлу.
Capacity	Размер запрашиваемого PersistentVolume.
Volume Mode	- Filesystem: Связывает PersistentVolume как файловую систему, монтируемую в Pod. Этот режим доступен для любого типа workload. - Block Device: Связывает PersistentVolume как блочное устройство, монтируемое в Pod. Этот режим доступен только для виртуальных машин.
More	- Labels - Annotations - Selector: После выбора метода статического связывания можно

Параметр	Описание
	использовать селектор для выбора PersistentVolumes с определёнными метками. Метки PersistentVolume могут использоваться для обозначения специальных атрибутов хранилища, таких как тип диска или географическое расположение.

5. Нажмите **Create**. Дождитесь, пока статус PersistentVolumeClaim не изменится на `Bound`, что означает успешное сопоставление PersistentVolume.

Создание Persistent Volume Claim с помощью CLI

```
kubectl apply -f example-pvc.yaml
```

Операции

- **Связывание PersistentVolumeClaim**: При создании приложений или workload, требующих постоянного хранения данных, связывайте PersistentVolumeClaim для запроса соответствующего PersistentVolume.
- **Создание PersistentVolumeClaim с использованием Volume Snapshots**: Это помогает создавать резервные копии данных приложений и восстанавливать их при необходимости, обеспечивая надёжность данных бизнес-приложений. Пожалуйста, обратитесь к разделу Using Volume Snapshots.
- **Удаление PersistentVolumeClaim**: Вы можете нажать кнопку **Actions** в правом верхнем углу страницы с деталями, чтобы удалить PersistentVolumeClaim при необходимости. Перед удалением убедитесь, что PersistentVolumeClaim не связан с какими-либо приложениями или workload и не содержит снимков томов. После удаления PersistentVolumeClaim платформа обработает PersistentVolume в соответствии с политикой рекламации, что может привести к очистке данных в PersistentVolume и освобождению ресурсов хранилища. Пожалуйста, действуйте осторожно с учётом требований безопасности данных.

Расширение ёмкости PersistentVolumeClaim через веб-консоль

1. В левой навигационной панели выберите Storage > Persistent Volume Claims (PVC).
2. Найдите нужный persistent volume claim и нажмите ⋮ > Expand.
3. Укажите новый размер.
4. Нажмите Expand. Процесс расширения может занять некоторое время, пожалуйста, будьте терпеливы.

Расширение ёмкости Persistent Volume Claim с помощью CLI

```
kubectl patch pvc example-pvc -n k-1 --type='merge' -p '{
  "spec": {
    "resources": {
      "requests": {
        "storage": "6Gi"
      }
    }
  }
}'
```

INFO

Если расширение PVC в Kubernetes не удалось, администраторы могут вручную восстановить состояние Persistent Volume Claim (PVC) и отменить запрос на расширение. См. Recover From PVC Expansion Failure

Дополнительные ресурсы

- How to Annotate Third-Party Storage Capabilities

Использование снимков томов

Снимок тома — это копия persistent volume claim (PVC) на определённый момент времени, которая может использоваться для настройки новых persistent volume claim (предварительное заполнение данными из снимка) или для отката существующих persistent volume claim к предыдущему состоянию, достигая эффекта резервного копирования данных приложения и их восстановления по необходимости, тем самым обеспечивая надёжность данных приложения.

Содержание

Предварительные требования
Пример custom resource (CR) VolumeSnapshot
Создание снимков томов через веб-консоль
Создание снимка тома на основе указанного persistent volume claim (PVC)
Создание снимков томов в произвольном порядке
Создание снимков томов через CLI
Создание persistent volume claims из снимков томов
Способ первый
Способ второй
Дополнительные ресурсы

Предварительные требования

- Администратор развернул компонент снимков томов **Snapshot Controller** для текущего кластера и включил функции, связанные со снимками, в кластере хранения.
- Persistent volume claim должен быть создан динамически, а его статус должен быть **Bound**.
- Storage class, связанный с persistent volume claim, должен поддерживать функциональность снимков, например, **CephRBD Built-in Storage**, **CephFS Built-in Storage** или **TopoLVM**.

Пример custom resource (CR) VolumeSnapshot

Этот пример создаёт снимок PVC с именем example-pvc с использованием CSI snapshot class.

```yaml
# example-snapshot.yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: example-pvc-20250527-111124
  namespace: k-1
  labels:
    snapshot.cpaas.io/sourcepvc: example-pvc
  annotations:
    cpaas.io/description: demo
spec:
  volumeSnapshotClassName: csi-cephfs-snapshotclass
  source:
    persistentVolumeClaimName: example-pvc
```

Создание снимков томов через веб-консоль

Создание снимка тома на основе указанного persistent volume claim (PVC)

Способ первый

1. Войдите в **Container Platform**.
2. В левой навигационной панели выберите **Storage > Persistent Volume Claims (PVC)**.
3. Нажмите ⋮ рядом с соответствующим persistent volume claim в списке и выберите **Create Volume Snapshot**.
4. Заполните описание снимка. Это описание поможет зафиксировать текущее состояние persistent volume, например *Before Application Upgrade*.
5. Нажмите **Create**. Время создания снимка зависит от состояния сети и объёма данных; пожалуйста, подождите.
   Когда статус снимка изменится на `Available`, это означает успешное создание.

Способ второй

1. Войдите в **Container Platform**.
2. В левой навигационной панели выберите **Storage > Persistent Volume Claims (PVC)**.
3. Кликните по имени persistent volume claim в списке.
4. Перейдите на вкладку **Volume Snapshots**.
5. Нажмите **Create Volume Snapshot** и настройте необходимые параметры.
6. Нажмите **Create**. Время создания снимка зависит от состояния сети и объёма данных; пожалуйста, подождите.
   Когда статус снимка изменится на `Available`, это означает успешное создание.

Создание снимков томов в произвольном порядке

1. Войдите в **Container Platform**.
2. В левой навигационной панели выберите **Storage > Volume Snapshots**.
3. Нажмите **Create Volume Snapshot** и настройте необходимые параметры.
4. Нажмите **Create**. Время создания снимка зависит от состояния сети и объёма данных; пожалуйста, подождите.
   Когда статус снимка изменится на `Available`, это означает успешное создание.

Создание снимков томов через CLI

```
kubectl apply -f example-snapshot.yaml
```

Создание persistent volume claims из снимков томов

В настоящее время платформа поддерживает создание снимков томов только с использованием PVC, созданных из storage class с **Dynamic Provisioning**. Вы можете создавать новые PVC на основе такого снимка и связывать их.

Примечание: Режимы доступа, поддерживаемые при создании PVC из снимка, отличаются от поддерживаемых при создании PVC из storage class, что выделено **жирным** в таблице.

Storage Class, используемый для создания снимков томов	**Single Node Read-Write (RWO)**	**Multi-Node Read-Only (ROX)**	**Multi-Node Read-Write (RWX)**
TopoLVM	Поддерживается	Не поддерживается	Не поддерживается
CephRBD Block Storage	Поддерживается	Не поддерживается	Не поддерживается
CephFS File Storage	Поддерживается	**Поддерживается**	Поддерживается

Способ первый

1. Войдите в **Container Platform**.
2. В левой навигационной панели выберите **Storage > Persistent Volume Claims (PVC)**.
3. Кликните по имени persistent volume claim в списке.
4. Перейдите на вкладку **Volume Snapshots**.

5. Нажмите ⋮ рядом с соответствующим снимком тома в списке и выберите **Create Persistent Volume Claim**, настроив необходимые параметры.
6. Нажмите **Create**.

Способ второй

1. Войдите в **Container Platform**.
2. В левой навигационной панели выберите **Storage > Volume Snapshots**.
3. Нажмите ⋮ рядом с соответствующим снимком тома в списке и выберите **Create Persistent Volume Claim**, настроив необходимые параметры.
4. Нажмите **Create**.

Дополнительные ресурсы

- Creating PVCs

Как сделать

Generic ephemeral volumes

- Пример эфемерных томов
- Основные характеристики
- Когда использовать Generic Ephemeral \
- Чем они отличаются от emptyDir?

Использование emptyDir

- Пример emptyDir
- Необязательная настройка medium
- Основные характеристики
- Распространённые сценарии использов

Настройка томов

- Предварителы
- Процедура
- Автоматизаци

Руководство по аннотированию возможностей стороннего хранилища

1. Начало работы
2. Пример ConfigMap
3. Обновление существующих описаний возможностей
4. Совместимость с устаревшим форматом
5. Часто задаваемые вопросы

ка

- гия
- елы
- ание
- зап
- одн
- е ава
- Плановая мигр
- Failover
- Failback (post-

Generic ephemeral volumes

Generic Ephemeral Volumes в Kubernetes — это функция, которая позволяет создавать эфемерные (временные) тома на уровне пода с использованием существующих StorageClasses и CSI-драйверов, без необходимости предварительного определения PersistentVolumeClaims (PVC).

Они сочетают гибкость динамического выделения с простотой объявления томов на уровне пода.

- Это временные тома, которые автоматически:
  - создаются при запуске Pod
  - удаляются при завершении Pod
- Используют те же базовые механизмы, что и PersistentVolumeClaim
- Требуют CSI (Container Storage Interface) драйвер с поддержкой динамического выделения

Содержание

Пример эфемерных томов

Основные характеристики

Когда использовать Generic Ephemeral Volumes

Чем они отличаются от emptyDir?

Пример эфемерных томов

Этот пример автоматически создаёт временный PVC для Pod с использованием указанного `StorageClass`.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ephemeral-demo
spec:
  containers:
    - name: app
      image: busybox
      command: ["sh", "-c", "echo hello > /data/hello.txt && sleep 3600"]
      volumeMounts:
        - mountPath: /data
          name: ephemeral-volume
  volumes:
    - name: ephemeral-volume
      ephemeral: 1
        volumeClaimTemplate:
          metadata:
            labels:
              type: temporary
          spec:
            accessModes: [ "ReadWriteOnce" ]
            resources:
              requests:
                storage: 1Gi
            storageClassName: standard
```

1 `Pod` создаст `PVC`, используя этот шаблон.

Основные характеристики

Feature	Description
Ephemeral	Том удаляется при удалении Pod

Feature	Description
Dynamic provisioning	Поддерживается любым CSI-драйвером с динамическим выделением
No separate PVC	VolumeClaim встроен непосредственно в спецификацию Pod
CSI-powered	Работает с любым совместимым CSI-драйвером (EBS, RBD, Longhorn и др.)

Когда использовать Generic Ephemeral Volumes

- Когда требуется временное хранилище с такими возможностями, как:
  - Изменяемый размер томов
  - Снимки (snapshots)
  - Шифрование
  - Хранилище, не привязанное к узлу (например, облачное блочное хранилище)
- Идеально подходит для:
  - Кэширования промежуточных данных
  - Временных рабочих каталогов
  - Конвейеров, AI/ML рабочих процессов

Чем они отличаются от emptyDir?

Feature	`emptyDir`	Generic Ephemeral Volume
Backing storage	Локальный диск или память узла	Любое хранилище с поддержкой CSI
Storage features	Базовые	Поддержка снимков, шифрования и др.

Feature	`emptyDir`	Generic Ephemeral Volume
Use case	Простое временное хранилище	Расширенные требования к эфемерному хранилищу
Reschedulable	Нет (привязан к узлу)	Да (если CSI-том можно присоединить)

Использование emptyDir

В Kubernetes emptyDir — это простой тип эфемерного тома, который предоставляет временное хранилище для пода на время его жизни. Он создаётся при назначении пода на узел и удаляется, когда под удаляется с этого узла.

Содержание

Пример emptyDir

Необязательная настройка medium

Основные характеристики

Распространённые сценарии использования

Пример emptyDir

Этот Pod создаёт временный том, смонтированный в /data, который используется контейнером.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-demo
spec:
  containers:
    - name: app
      image: busybox
      command: ["sh", "-c", "echo hello > /data/hello.txt && sleep 3600"]
      volumeMounts:
        - mountPath: /data
          name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

Необязательная настройка medium

Вы можете выбрать, где будут храниться данные:

```yaml
emptyDir:
  medium: "Memory"
```

Medium	Описание
(по умолчанию)	Использует диск узла, SSD или сетевое хранилище, в зависимости от окружения
`Memory`	Использует RAM (`tmpfs`) для более быстрого доступа (но данные нестабильны)

Основные характеристики

Особенность	Описание
Начинается пустым	Нет данных при создании
Совместное использование	Один и тот же том может использоваться несколькими контейнерами в поде
Удаляется вместе с подом	Том уничтожается при удалении пода
Локальный для узла	Том хранится на локальном диске или в памяти узла
Быстрый	Идеален для производительного временного пространства

Распространённые сценарии использования

- Кэширование промежуточных артефактов сборки
- Буферизация логов
- Временные рабочие директории
- Совместное использование данных между контейнерами в одном поде (например, сайдкары)

Настройка постоянного хранилища с использованием локальных томов

Alauda Container Platform может быть настроена с постоянным хранилищем с помощью локальных томов. Локальные постоянные тома позволяют получить доступ к локальным устройствам хранения, таким как диск или раздел, используя стандартный интерфейс persistent volume claim.

Локальные тома можно использовать без ручного назначения подов на узлы, так как система учитывает ограничения по узлам для томов. Однако локальные тома зависят от доступности базового узла и не подходят для всех приложений.

> **NOTE**
>
> Локальные тома могут использоваться только как статически созданные persistent volume.

Содержание

- Предварительные требования
- Процедура
  - Установка оператора Local Storage
  - Предоставление локальных томов с помощью оператора Local Storage
- Автоматизация обнаружения локальных устройств хранения
  - Предварительные требования
  - Процедура

Предварительные требования

- **Скачайте** установочный пакет **Alauda Build of LocalStorage**, соответствующий архитектуре вашей платформы.
- **Загрузите** установочный пакет **Alauda Build of LocalStorage** с помощью механизма Upload Packages.
- У вас есть локальный диск, который соответствует следующим условиям:
  - Он подключен к узлу.
  - Он не смонтирован.
  - Он не содержит разделов.

Процедура

1 Установка оператора Local Storage

1. Войдите в систему и перейдите на страницу **Administrator**.
2. Нажмите **Marketplace** > **OperatorHub**, чтобы перейти на страницу **OperatorHub**.
3. Найдите **Alauda Build of LocalStorage**, нажмите **Install** и перейдите на страницу **Install Alauda Build of LocalStorage**.
   Параметры конфигурации:

Параметр	Рекомендуемая конфигурация
Channel	Канал по умолчанию — `stable`.
Installation Mode	`Cluster`: Все пространства имён в кластере используют один экземпляр оператора для создания и управления, что снижает использование ресурсов.
Installation Place	Выберите `Recommended`, Namespace поддерживается только **acp-storage**.

Параметр	Рекомендуемая конфигурация
Upgrade Strategy	`Manual`: При наличии новой версии в Operator Hub требуется ручное подтверждение для обновления оператора до последней версии.

2 Предоставление локальных томов с помощью оператора Local Storage

Локальные тома не могут создаваться динамически. Вместо этого persistent volumes создаются оператором Local Storage. Провиженер локальных томов ищет устройства с файловой системой или блочные устройства по путям, указанным в определённом ресурсе.

1. Создайте ресурс локального тома. Этот ресурс должен определять узлы и пути к локальным томам.

> **NOTE**
>
> Не используйте разные имена storage class для одного и того же устройства. Это приведёт к созданию нескольких persistent volumes (PV).

Выполните команды на **контрольном узле** кластера.

```
cat << EOF | kubectl create -f -
apiVersion: "local.storage.openshift.io/v1"
kind: "LocalVolume"
metadata:
  name: "local-disks"
  namespace: "acp-storage" (1)
spec:
  nodeSelector: (2)
    nodeSelectorTerms:
    - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - worker-01
          - worker-02
          - worker-03
  storageClassDevices:
    - storageClassName: "local-sc" (3)
      forceWipeDevicesAndDestroyAllData: false (4)
      volumeMode: Filesystem (5)
      fsType: xfs (6)
      devicePaths: (7)
        - /path/to/device (8)
EOF
```

(1) Пространство имён, в котором установлен оператор Local Storage, по умолчанию `acp-storage`.

(2) Необязательно: селектор узлов, содержащий список узлов, к которым подключены локальные тома. В этом примере используются имена хостов узлов, полученные с помощью `kubectl get node`. Если значение не задано, оператор Local Storage попытается найти подходящие диски на всех доступных узлах.

(3) Имя storage class, используемого при создании объектов persistent volume. Оператор Local Storage автоматически создаёт storage class, если он отсутствует. Убедитесь, что используете storage class, который уникально идентифицирует этот набор локальных томов.

(4) Управляет тем, стирает ли оператор указанные устройства перед использованием. По умолчанию — "false". ВНИМАНИЕ: установка `forceWipeDevicesAndDestroyAllData: true` является разрушительной и

приведёт к удалению существующих таблиц разделов/подписей файловой системы и данных на этих устройствах. Используйте только если уверены, что устройства можно безопасно стереть.

5 Режим тома, либо `Filesystem`, либо `Block`, определяющий тип локальных томов.

6 Необязательно: файловая система, которая создаётся при первом монтировании локального тома. Этот параметр должен быть настроен только если `volumeMode` установлен в `Filesystem`.

7 Путь, содержащий список локальных устройств хранения для выбора.

8 Замените это значение на фактический путь к вашим локальным дискам в ресурсе `LocalVolume` по `by-id`, например `/dev/disk/by-id/wwn`. PV создаются для этих локальных дисков после успешного развертывания провиженера.

2. Проверьте, что persistent volumes созданы:
   Выполните команды на **контрольном узле** кластера.

```
kubectl get pv
```

Пример вывода:

```
NAME                CAPACITY   ACCESS MODES   RECLAIM POLICY   STA
TUS      CLAIM   STORAGECLASS   REASON   AGE
local-pv-n8xlrd2a   100Gi      RWO            Delete           Ava
ilable           local-sc                88m
local-pv-tc78vc73   100Gi      RWO            Delete           Ava
ilable           local-sc                82m
local-pv-q86px4df   100Gi      RWO            Delete           Ava
ilable           local-sc                48m
```

NOTE

Редактирование объекта LocalVolume не изменяет fsType или volumeMode существующих persistent volumes, так как это может привести к разрушительной операции.

3. Создание persistent volume claim для локального тома

   Выполните команды на **контрольном узле** кластера.

```
cat << EOF | kubectl create -f -
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: local-pvc-name 1
spec:
  accessModes:
  - ReadWriteOnce
  volumeMode: Filesystem 2
  resources:
    requests:
      storage: 100Gi 3
  storageClassName: local-sc 4
EOF
```

1. Имя PVC.
2. Тип PVC. По умолчанию `Filesystem`.
3. Объём хранилища, доступный для PVC.
4. Имя storage class, требуемого заявкой.

Автоматизация обнаружения локальных устройств хранения

Оператор Local Storage автоматизирует обнаружение локальных устройств хранения.

Предварительные требования

- Вы установили оператор Local Storage.

Процедура

1. Создайте объект LocalVolumeDiscovery

Выполните команды на **контрольном узле** кластера.

```
cat << EOF | kubectl create -f -
apiVersion: local.storage.openshift.io/v1alpha1
kind: LocalVolumeDiscovery
metadata:
  name: auto-discover-devices
  namespace: acp-storage
spec:
  nodeSelector: (1)
    nodeSelectorTerms:
      - matchExpressions:
          - key: kubernetes.io/hostname
            operator: In
            values:
              - worker-01
              - worker-02
              - worker-03
  tolerations: (2)
    - operator: Exists
EOF
```

(1) Необязательно: узлы, на которых должны выполняться политики автоматического обнаружения. Если значение не задано, оператор Local Storage попытается выполнить автоматическое обнаружение на всех доступных узлах.

(2) Необязательно: если указаны, tolerations — это список толерантностей, передаваемых демону LocalVolumeDiscovery.

2. Проверьте результат обнаружения

   Выполните команды на **контрольном узле** кластера.

```
kubectl -n acp-storage get localvolumediscoveryresult
```

Пример вывода:

```
NAME                           AGE
discovery-result-worker-01     21m
discovery-result-worker-02     21m
discovery-result-worker-03     21m
```

Для выбранных узлов в объекте LocalVolumeDiscovery создаётся отдельный объект LocalVolumeDiscoveryResult, в котором можно просмотреть обнаруженные блочные устройства на этом узле.

Настройка постоянного хранилища с использованием NFS

Кластеры Alauda Container Platform поддерживают постоянное хранилище с использованием NFS. Persistent Volumes (PV) и Persistent Volume Claims (PVC) обеспечивают уровень абстракции для предоставления и использования томов хранения в рамках проекта. Хотя детали конфигурации NFS можно встроить непосредственно в определение Pod, такой подход не создает том как отдельный, изолированный ресурс кластера, что увеличивает риск конфликтов.

Содержание

- Предварительные требования
- Процедура
  - Создайте определение объекта для PV
  - Проверьте, что PV был создан
  - Создайте PVC, ссылающийся на PV
  - Проверьте, что Persistent Volume Claim был создан
- Принудительное ограничение квот диска через разделённые экспорты
- Безопасность томов NFS
  - Идентификаторы групп
  - Идентификаторы пользователей
  - Настройки экспорта
- Освобождение ресурсов

Предварительные требования

- Хранилище должно существовать в базовой инфраструктуре до того, как его можно будет смонтировать как том в Alauda Container Platform.
- Для предоставления томов NFS требуется только список серверов NFS и путей экспорта.

Процедура

1 Создайте определение объекта для PV

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs-example 1
spec:
  capacity:
    storage: 1Gi 2
  accessModes:
  - ReadWriteOnce 3
  nfs: 4
    path: /tmp 5
    server: 10.0.0.3 6
  persistentVolumeReclaimPolicy: Retain 7
EOF
```

1 Имя тома.

2 Объем хранилища.

3 Хотя это кажется связанным с контролем доступа к тому, на самом деле это используется аналогично меткам и служит для сопоставления PVC с PV. В настоящее время правила доступа на основе accessModes не применяются.

4 Тип используемого тома, в данном случае плагин nfs.

5 Адрес сервера NFS.

6 Путь экспорта NFS.

7 Что происходит после удаления PVC (Retain, Delete, Recycle).

2 Проверьте, что PV был создан

Команда

```
kubectl get pv
```

Пример вывода

```
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATU
S   CLAIM   STORAGECLASS   REASON   AGE
pv-nfs-example   1Gi         RWO             Retain           Avail
able                                 10s
```

3 Создайте PVC, ссылающийся на PV

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-claim1
spec:
  accessModes:
  - ReadWriteOnce 1
  resources:
    requests:
      storage: 1Gi 2
  volumeName: pv-nfs-example 3
  storageClassName: ""
```

1 Режимы доступа не обеспечивают безопасность, а служат метками для сопоставления PV с PVC.

2 Этот запрос ищет PV с емкостью 1Gi или больше.

3 Имя PV, который будет использоваться.

4 **Проверьте, что Persistent Volume Claim был создан**

Команда

```
kubectl get pvc
```

Пример вывода

```
NAME            STATUS   VOLUME           CAPACITY   ACCESS MODE
S   STORAGECLASS   AGE
nfs-claim1      Bound    pv-nfs-example   1Gi        RWO
10s
```

Принудительное ограничение квот диска через разделённые экспорты

Для применения квот диска и ограничений по размеру можно использовать разделы диска. Назначьте каждый раздел как отдельную точку экспорта, при этом каждый экспорт соответствует отдельному PersistentVolume (PV).

Хотя Alauda Container Platform требует уникальных имен PV, ответственность за уникальность сервера и пути NFS для каждого экспорта лежит на администраторе.

Такой подход с разделами позволяет точно управлять емкостью. Разработчики запрашивают постоянное хранилище с указанием требуемого объема (например, 10Gi), и ACP сопоставляет запрос с PV, поддерживаемым разделом/экспортом, предлагающим как минимум такой объем. Обратите внимание: применение квоты распространяется на доступное пространство внутри назначенного раздела/экспорта.

Безопасность томов NFS

В этом разделе описываются механизмы безопасности томов NFS с акцентом на сопоставление прав доступа. Предполагается, что читатели имеют базовые знания о POSIX-правах, UID процессов и дополнительных группах.

Разработчики запрашивают NFS-хранилище через:

- Ссылку PersistentVolumeClaim (PVC) по имени, или
- Прямую конфигурацию плагина тома NFS в разделе volumes спецификации Pod.

На сервере NFS файл /etc/exports определяет правила экспорта для доступных директорий. Каждая экспортированная директория сохраняет свои исходные POSIX ID владельца и группы.

Ключевое поведение плагина NFS в Alauda Container Platform:

1. Монтирует тома в контейнеры, сохраняя точные POSIX-владельца и права из исходной директории
2. Запускает контейнеры без принудительного сопоставления UID процесса с владельцем монтирования — это намеренная мера безопасности

Например, рассмотрим NFS-директорию с такими атрибутами на сервере:

Команда

```
ls -l /share/nfs -d
```

Пример вывода

```
drwxrws---. nfsnobody 5555   /share/nfs
```

Команда

```
id nfsnobody
```

Пример вывода

```
uid=65534(nfsnobody) gid=65534(nfsnobody) groups=65534(nfsnobody)
```

Тогда контейнер должен либо запускаться с UID 65534, владельцем nfsnobody, либо иметь 5555 в дополнительных группах для доступа к директории.

> **NOTE**
>
> Примечание ID владельца 65534 приведён в качестве примера. Хотя root_squash NFS отображает root (uid 0) в nfsnobody (uid 65534), экспорты NFS могут иметь произвольные ID владельцев. Владелец 65534 не обязателен для экспортов NFS.

Идентификаторы групп

Рекомендуемое управление доступом к NFS (когда права экспорта фиксированы) Если изменить права на экспорт NFS невозможно, рекомендуемый способ управления доступом — через дополнительные группы.

Дополнительные группы в Alauda Container Platform — распространённый механизм контроля доступа к общему файловому хранилищу, например NFS.

В отличие от блочного хранилища: доступ к блочным томам (например, iSCSI) обычно управляется установкой значения fsGroup в securityContext пода. Этот подход использует изменение групповой принадлежности файловой системы при монтировании.

> **NOTE**
>
> Для доступа к постоянному хранилищу обычно предпочтительнее использовать дополнительные групповые ID, а не пользовательские ID.

Поскольку групповой ID в примере целевой NFS-директории равен 5555, под может определить этот групповой ID через supplementalGroups в securityContext пода. Например:

```
spec:
  containers:
    - name:
    ...
  securityContext: 1
    supplementalGroups: [5555] 2
```

1 securityContext должен быть определён на уровне пода, а не конкретного контейнера.

2 Массив GID, определённых для пода. В данном случае один элемент в массиве. Дополнительные GID разделяются запятыми.

Идентификаторы пользователей

UID могут быть определены в образе контейнера или в определении Pod.

> **NOTE**
>
> Обычно предпочтительнее использовать дополнительные групповые ID для доступа к постоянному хранилищу, а не пользовательские ID.

В приведённом выше примере целевой NFS-директории контейнеру нужно установить UID 65534, игнорируя пока групповые ID, поэтому в определение Pod можно добавить:

```
spec:
  containers: 1
  - name:
  ...
    securityContext:
      runAsUser: 65534 2
```

1 Под содержит определение securityContext, специфичное для каждого контейнера, и securityContext пода, который применяется ко всем контейнерам в поде.

2 65534 — это пользователь nfsnobody.

Настройки экспорта

Чтобы разрешить произвольным пользователям контейнера читать и записывать том, каждый экспортируемый том на сервере NFS должен соответствовать следующим условиям:

- Каждый экспорт должен быть экспортирован в следующем формате:

```
# замените 10.0.0.0/24 на доверенные CIDR/хосты
/<example_fs> 10.0.0.0/24(rw,sync,root_squash,no_subtree_check)
```

- Межсетевой экран должен быть настроен для разрешения трафика к точке монтирования.
  - Для NFSv4 настройте порт по умолчанию 2049 (nfs).

```
iptables -I INPUT 1 -p tcp --dport 2049 -j ACCEPT
```

  - Для NFSv3 необходимо настроить три порта: 2049 (nfs), 20048 (mountd) и 111 (portmapper).

```
iptables -I INPUT 1 -p tcp --dport 2049 -j ACCEPT
iptables -I INPUT 1 -p tcp --dport 20048 -j ACCEPT
iptables -I INPUT 1 -p tcp --dport 111 -j ACCEPT
```

- Экспорт и директория NFS должны быть настроены так, чтобы они были доступны целевым подам. Либо установите владельцем экспорта основной UID контейнера, либо предоставьте доступ группе пода через supplementalGroups, как показано выше в разделе про групповые ID.

Освобождение ресурсов

NFS реализует интерфейс Recyclable плагина Alauda Container Platform. Автоматические процессы обрабатывают задачи освобождения ресурсов на основе политик, установленных для каждого persistent volume.

По умолчанию PV настроены на Retain.

После удаления претензии на PVC и освобождения PV объект PV не должен повторно использоваться. Вместо этого следует создать новый PV с теми же основными параметрами тома, что и у оригинала.

Например, администратор создаёт PV с именем nfs1:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs1
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 192.168.1.1
    path: "/"
```

Пользователь создаёт PVC1, который связывается с nfs1. Затем пользователь удаляет PVC1, освобождая претензию на nfs1. В результате nfs1 становится Released. Если администратор хочет сделать тот же NFS-шар доступным, он должен создать новый PV с теми же деталями сервера NFS, но с другим именем PV:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs2
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 192.168.1.1
    path: "/"
```

Удаление оригинального PV и повторное создание его с тем же именем не рекомендуется. Попытки вручную изменить статус PV с Released на Available вызывают ошибки и могут привести к потере данных.

Настройка аварийного восстановления для PersistentVolumeClaim

Для обеспечения межкластерного аварийного восстановления для PVC приложения используйте **Alauda Build of VolSync**.

Содержание

Обзор
Терминология
Предварительные требования
Развертывание Alauda Build of VolSync
Настройка запланированной синхронизации
Создание Secret для Data Mover rsync-tls
Создание ресурса ReplicationDestination
Создание ресурса ReplicationSource
Проверка статуса синхронизации
Настройка однократной синхронизации
Создание ресурса ReplicationSource для однократной синхронизации
Проверка статуса синхронизации
Включение аварийного восстановления для PVC приложения
Развертывание stateful-приложения
Настройка аварийного восстановления PVC
Плановая миграция

Процедуры

Failover

Процедуры

Failback (post-disaster recovery)

Процедуры

Обзор

Alauda Build of VolSync — это Operator, который выполняет асинхронную репликацию persistent volumes внутри кластера или между кластерами. Репликация, предоставляемая VolSync, не зависит от системы хранения. Это позволяет выполнять репликацию в хранилища и из хранилищ, которые обычно не поддерживают удаленную репликацию. Кроме того, она может работать между разными типами (и поставщиками) хранилищ.

Терминология

Термин	Пояснение
Primary Cluster	Активная production-среда.
Secondary Cluster	Резервная площадка для восстановления; остается в режиме ожидания и готова принять нагрузку во время аварии.
Stateful Application	Приложения, использующие PVC для сохранения данных.
ReplicationSource	ReplicationSource — это ресурс VolSync, который можно использовать для определения исходного PVC и типа replication mover, что позволяет реплицировать или синхронизировать данные PVC в удаленное расположение.
ReplicationDestination	ReplicationDestination — это ресурс VolSync, который можно использовать для определения назначения репликации или синхронизации VolSync.

Термин	Пояснение
Data Movers	Data movers в VolSync отвечают за копирование данных из одного расположения в другое. Поддерживаемые movers: • Rclone • Restic • Rsync

Предварительные требования

- **Скачайте** установочный пакет **Alauda Build of VolSync**, соответствующий архитектуре вашей платформы.
- **Загрузите** установочный пакет **Alauda Build of VolSync** с помощью механизма Upload Packages на оба кластера: Primary и Secondary.
- На обоих кластерах — Primary и Secondary — развернут **Alauda Container Platform Snapshot Management**.
- Хранилище, используемое PVC, должно быть предоставлено через **CSI** и поддерживать функциональность **snapshot**.

Развертывание Alauda Build of VolSync

1. Войдите в систему и перейдите на страницу **Administrator**.
2. Нажмите **Marketplace** > **OperatorHub**, чтобы открыть страницу **OperatorHub**.
3. Найдите **Alauda Build of VolSync**, нажмите **Install** и перейдите на страницу **Install Alauda Build of VolSync**.
   Параметры конфигурации:

Parameter	Recommended Configuration
Channel	Канал по умолчанию — `stable`.
Installation Mode	`Cluster`: Все namespace в кластере используют один экземпляр Operator для создания и управления, что приводит к меньшему потреблению ресурсов.
Installation Place	Выберите `Recommended`. Поддерживается только Namespace **volsync-system**.
Upgrade Strategy	`Manual`: При появлении новой версии в Operator Hub для обновления Operator до последней версии требуется ручное подтверждение.

Настройка запланированной синхронизации

После настройки Scheduled Synchronization для PVC VolSync будет автоматически синхронизировать данные из `ReplicationSource` в `ReplicationDestination` с заданным интервалом.

В этом разделе описаны шаги настройки синхронизации данных из primary-кластера в secondary-кластер. Для синхронизации из secondary в primary адаптируйте приведенный ниже пример, поменяв местами роли кластеров (primary и secondary)

1 Создание Secret для Data Mover rsync-tls

Создайте Secret на обоих кластерах — **Primary** и **Secondary**; пропустите этот шаг, если Secret уже существует.

Команда

```
apiVersion: v1
data:
  psk.txt: <psk>
kind: Secret
metadata:
  name: <name>
  namespace: <namespace>
type: Opaque
```

Пример

```
apiVersion: v1
data:
  # echo -n 1:23b7395fafc3e842bd8ac0fe142e6ad1 | base64
  psk.txt: MToyM2I3Mzk1ZmFmYzNlODQyYmQ4YWMwZmUxNDJlNmFkMQ==
kind: Secret
metadata:
  name: volsync-rsync-tls
  namespace: default
type: Opaque
```

Параметры:

Parameter	Explanation
name	Имя Secret
namespace	Namespace Secret, должен совпадать с namespace приложения
psk	Это поле должно соответствовать формату, ожидаемому stunnel: `<id>:<не менее 32 шестнадцатеричных символов>`. Например, `1:23b7395fafc3e842bd8ac0fe142e6ad1`.

2 Создание ресурса ReplicationDestination

Создайте `ReplicationDestination` на кластере **Secondary**

Команда

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: rd-<pvc-name>
  namespace: <namespace>
spec:
  rsyncTLS:
    copyMethod: Snapshot
    destinationPVC: <pvc-name>
    keySecret: <key-secret>
    serviceType: <service-type>
    storageClassName: <storageclass-name>
    volumeSnapshotClassName: <volumesnapshotclass-name>
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
EOF
```

Пример

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: rd-pvc-01
  namespace: default
spec:
  rsyncTLS:
    copyMethod: Snapshot
    destinationPVC: pvc-01
    keySecret: volsync-rsync-tls
    serviceType: NodePort
    storageClassName: sc-cephfs
    volumeSnapshotClassName: csi-cephfs-snapshotclass
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
EOF
```

Параметры:

Parameter	Explanation
namespace	Namespace должен быть тем же, что и у приложения
pvc-name	Имя **существующего** PVC, используемого приложением
key-secret	Имя Secret, содержащего ключ TLS-PSK для аутентификации соединения с источником. Создайте его на Шаге 1
service-type	VolSync создает Service, чтобы источник мог подключиться к назначению. Это поле определяет тип этого Service. Допустимые значения: `ClusterIP`, `LoadBalancer` или `NodePort`.

Parameter	Explanation
storageclass-name	Имя storageclass, используемого PVC приложения
volumesnapshotclass-name	Имя volumesnapshotclass, соответствующего PVC приложения

NOTE

О типе Service

Если указан `ClusterIP`, Service получит IP-адрес, выделенный из пула адресов “cluster network”. По умолчанию этот набор адресов недоступен извне кластера, поэтому он плохо подходит для межкластерной репликации. Однако различные сетевые дополнения, такие как Submariner, объединяют cluster network, что делает этот вариант подходящим.

Если указан `LoadBalancer`, будет выделен внешний IP-адрес. Для этого требуется поддержка load balancer'ов на уровне кластера, например, реализованная различными облачными провайдерами, либо MetalLB в случае физических кластеров. Хотя это самый простой способ выделить доступный адрес в облачных средах, load balancer'ы обычно влекут за собой дополнительные расходы и имеют ограниченное количество.

3 Создание ресурса ReplicationSource

Создайте `ReplicationSource` на кластере **Primary**

Команда

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-<pvc-name>
  namespace: <namespace>
spec:
  rsyncTLS:
    address: <address>
    copyMethod: Snapshot
    keySecret: <key-secret>
    port: <port>
    storageClassName: <storageclass-name>
    volumeSnapshotClassName: <volumesnapshotclass-name>
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: <pvc-name>
  trigger:
    schedule: <schedule>
EOF
```

Пример

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-pvc-01
  namespace: default
spec:
  rsyncTLS:
    address: 192.168.129.201
    copyMethod: Snapshot
    keySecret: volsync-rsync-tls
    port: 30532
    storageClassName: sc-cephfs
    volumeSnapshotClassName: csi-cephfs-snapshotclass
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: pvc-01
  trigger:
    schedule: "*/10 * * * *"
EOF
```

Параметры:

Parameter	Explanation
namespace	Имя Namespace, должно совпадать с приложением
pvc-name	Имя PVC приложения
key-secret	Имя secret VolSync, созданного на Шаге 1
address	Указывает адрес SSH-сервера назначения репликации. Его можно взять напрямую из поля `.status.rsync.address` ресурса ReplicationDestination.

Parameter	Explanation
port	Порт Service для подключения к назначению
storageclass-name	Имя storageclass, используемого PVC приложения
volumesnapshotclass-name	Имя volumesnapshotclass, соответствующего PVC приложения
schedule	Расписание синхронизации, определяемое через cronspec, что делает его очень гибким. Можно указать как интервалы, так и конкретное время и/или дни.

4

Проверка статуса синхронизации

Проверьте синхронизацию из `ReplicationSource`

Команда

```
kubectl -n <namespace> get ReplicationSource <rs-name> -o jsonpath='{.status}'
```

Пример

```
kubectl -n default get ReplicationSource rs-pvc-01 -o jsonpath='{.status}'
```

Последняя синхронизация была завершена в `.status.lastSyncTime` и заняла `.status.lastSyncDuration` секунд.

Следующая запланированная синхронизация будет в `.status.nextSyncTime`.

Настройка однократной синхронизации

One-Time Synchronization запускается вручную. Это управляется путем задания уникальной строки для поля manual в спецификации trigger ресурса ReplicationSource. Задача синхронизации выполняется один раз сразу после применения конфигурации.

1 Создание ресурса ReplicationSource для однократной синхронизации

Команда

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-<pvc-name>-latest
  namespace: <namespace>
spec:
  rsyncTLS:
    address: <address>
    copyMethod: Snapshot
    keySecret: <key-secret>
    port: <port>
    storageClassName: <storageclass-name>
    volumeSnapshotClassName: <volumesnapshotclass-name>
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: <pvc-name>
  trigger:
    manual: <manual-id>
EOF
```

Пример

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-pvc-01-latest
  namespace: default
spec:
  rsyncTLS:
    address: 192.168.129.201
    copyMethod: Snapshot
    keySecret: volsync-rsync-tls
    port: 30532
    storageClassName: sc-cephfs
    volumeSnapshotClassName: csi-cephfs-snapshotclass
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: pvc-01
  trigger:
    manual: latest
```

Единственное отличие от Запланированной синхронизации состоит в том, что `.spec.trigger` следует установить в **manual**.

2 Проверка статуса синхронизации

Команда

```
kubectl -n <namespace> get ReplicationSource <rs-name> -o jsonpath='{.status.lastManualSync}'
```

Пример

```
kubectl -n default get ReplicationSource rs-pvc-01-latest -o jso
npath='{.status.lastManualSync}'
```

Если вывод совпадает с `<manual-id>`, синхронизация завершена.

Включение аварийного восстановления для PVC приложения

1 Развертывание stateful-приложения

1. Разверните stateful-приложения на кластере **Primary**

▶ Нажмите, чтобы просмотреть

2. Создайте PVC приложения на кластере **Secondary**

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-01
  namespace: default
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: sc-cephfs
  volumeMode: Filesystem
EOF
```

2 Настройка аварийного восстановления PVC

Настройте синхронизацию `Primary-to-Secondary`

См. Настройка запланированной синхронизации

Плановая миграция

Сценарий использования:

Перенос бизнес-сервисов с кластера Primary на кластер Secondary, пока оба кластера работают в штатном режиме.

Процедуры

1. **Снизьте количество pod'ов приложения**
   Снизьте количество всех pod'ов приложения, которые используют dr PVC, на кластере **Primary**.
2. **Удалите ресурс ReplicationSource**
   Удалите `ReplicationSource` на кластере **Primary**

   Команда

   ```
   kubectl -n <namespace> delete ReplicationSource <rs-name>
   ```

   Пример

   ```
   kubectl -n default delete ReplicationSource rs-pvc-01
   ```

3. **Создайте однократную синхронизацию**
   Запустите задачу синхронизации с кластера **Primary**, чтобы гарантировать, что данные на кластере **Secondary** находятся в состоянии `up-to-date`.
   Создайте `ReplicationSource` на кластере **Primary**
   См. Настройка однократной синхронизации
4. **Удалите однократную синхронизацию**

После завершения однократной синхронизации удалите ресурс One-Time `ReplicationSource`

Команда

```
kubectl -n <namespace> delete ReplicationSource <rs-name>
```

Пример

```
kubectl -n default delete ReplicationSource rs-pvc-01-latest
```

5. **Удалите ресурс ReplicationDestination**

   Удалите `ReplicationDestination` на кластере **Secondary**

   Команда

   ```
   kubectl -n <namespace> delete ReplicationDestination <rd-name>
   ```

   Пример

   ```
   kubectl -n default delete ReplicationDestination rd-pvc-01
   ```

6. **Увеличьте количество pod'ов приложения**

   Увеличьте количество всех pod'ов приложения, которые используют dr PVC, на кластере **Secondary**.

7. **Настройте синхронизацию secondary-to-primary**

   Настройте синхронизацию PVC для аварийного восстановления `Secondary-to-Primary`, создав `ReplicationDestination` на кластере **Primary** и `ReplicationSource` на кластере **Secondary**.

   См. Настройка запланированной синхронизации

Failover

Сценарий использования:

Переключение сервисов на кластер Secondary после внезапного отключения кластера Primary.

Процедуры

Чтобы обеспечить целостность данных (если в primary-кластере произойдет сбой во время синхронизации), выполните локальную синхронизацию на кластере **Secondary**. Используйте PVC, восстановленный из последнего snapshot PVC приложения, в качестве источника, а текущий PVC приложения — в качестве назначения для выполнения синхронизации данных.

1. **Восстановите PVC**
   Восстановите PVC из `ReplicationDestination` на кластере **Secondary**

   Команда

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: restored-<pvc-name>
  namespace: <namespace>
spec:
  accessModes: [<access-modes>]
  dataSourceRef:
    kind: ReplicationDestination
    apiGroup: volsync.backube
    name: <rd-name>
  resources:
    requests:
      storage: <pvc-size>
  storageClassName: <storageclass-name>
EOF
```

Пример

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: restored-pvc-01
  namespace: default
spec:
  accessModes: [ReadWriteMany]
  dataSourceRef:
    kind: ReplicationDestination
    apiGroup: volsync.backube
    name: rd-pvc-01
  resources:
    requests:
      storage: 10Gi
  storageClassName: sc-cephfs
EOF
```

2. **Создайте локальный ресурс ReplicationSource**

   Создайте ресурс `ReplicationSource` на кластере **Secondary**

Команда

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-<pvc-name>-local
  namespace: <namespace>
spec:
  rsyncTLS:
    address: <address>
    copyMethod: Snapshot
    keySecret: <key-secret>
    port: <port>
    storageClassName: <storageclass-name>
    volumeSnapshotClassName: <volumesnapshotclass-name>
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: restored-<pvc-name>
  trigger:
    manual: <manual-id>
EOF
```

Пример

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-pvc-01-local
  namespace: default
spec:
  rsyncTLS:
    address: 192.168.129.201
    copyMethod: Snapshot
    keySecret: volsync-rsync-tls
    port: 30532
    storageClassName: sc-cephfs
    volumeSnapshotClassName: csi-cephfs-snapshotclass
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: restored-pvc-01
  trigger:
    manual: latest
EOF
```

Параметры см. в Настройке однократной синхронизации

3. **Дождитесь завершения синхронизации**

Команда

```
kubectl -n <namespace> get ReplicationSource <rs-name> -o json
path='{.status.lastManualSync}'
```

Пример

```
kubectl -n default get ReplicationSource rs-pvc-01-local -o jsonpath='{.status.lastManualSync}'
```

Если вывод совпадает с `<manual-id>`, синхронизация завершена.

4. **Удалите локальный ReplicationSource**

   Удалите локальный `ReplicationSource` на кластере **Secondary**

   Команда

   ```
   kubectl -n <namespace> delete ReplicationSource <rs-name>
   ```

   Пример

   ```
   kubectl -n default delete ReplicationSource rs-pvc-01-local
   ```

5. **Удалите ReplicationDestination**

   Удалите `ReplicationDestination` на кластере **Secondary**

   Команда

   ```
   kubectl -n <namespace> delete ReplicationDestination <rd-name>
   ```

   Пример

   ```
   kubectl -n default delete ReplicationDestination rd-pvc-01
   ```

6. **Увеличьте количество pod'ов приложения**

   Увеличьте количество всех pod'ов приложения на кластере **Secondary**.

Failback (post-disaster recovery)

Сценарий использования:

Кластер Primary теперь восстановлен и работает, поэтому требуется возврат сервисов на него.

Процедуры

1. **Снизьте количество pod'ов приложения на кластере Primary**
   Когда кластер primary снова станет доступен, pod'ы приложения восстановятся автоматически. Однако сначала необходимо снизить количество экземпляров сервиса, чтобы остановить трафик. После синхронизации последних данных с кластера secondary на кластер primary приложение можно будет снова масштабировать вверх и возобновить нормальную работу.
2. **Удалите ReplicationSource на кластере Primary**
   Сначала нужно удалить `ReplicationSource`, созданный до отказа кластера **Primary**.

   Команда

   ```
   kubectl -n <namespace> delete ReplicationSource <rs-name>
   ```

   Пример

   ```
   kubectl -n default delete ReplicationSource rs-pvc-01
   ```

3. **Синхронизация последних данных с кластера Secondary**
   Настройте `Secondary-to-Primary` One-Time Synchronization.
   Создайте `ReplicationDestination` на кластере **Primary**, а затем создайте однократный `ReplicationSource` на кластере **Secondary**
   См. Настройка однократной синхронизации
4. **Удалите ReplicationDestination и ReplicationSource**

После синхронизации данных удалите однократные ресурсы

Удалите `ReplicationSource` на кластере **Secondary**

Команда

```
kubectl -n <namespace> delete ReplicationSource <rs-name>
```

Пример

```
kubectl -n default delete ReplicationSource rs-pvc-01-latest
```

Удалите `ReplicationDestination` на кластере **Primary**

Команда

```
kubectl -n <namespace> delete ReplicationDestination <rd-name>
```

Пример

```
kubectl -n default delete ReplicationDestination rd-pvc-01
```

5. **Миграция приложения**

   Снизьте количество pod'ов приложения на кластере **Secondary**

   Увеличьте количество pod'ов приложения на кластере **Primary**

6. **Настройте синхронизацию Primary-to-Secondary**

   См. Настройка запланированной синхронизации

Руководство по аннотированию возможностей стороннего хранилища

> **Обзор функции:** Добавляя ConfigMap **StorageDescription** в пространство имён `kube-public`, платформа автоматически определяет поддержку снимков (snapshot) для каждого стороннего StorageClass, а также поддерживаемые режимы томов и режимы доступа (включая режимы доступа, специфичные для блочных томов). На экране создания PVC будут отображаться только допустимые варианты, что поможет вам легко выбрать и использовать нужные функции хранилища.

Содержание

1. Начало работы
   1.1 Создание или обновление ConfigMap
   1.2 Заполнение поля `data`
   1.3 Применение конфигурации
2. Пример ConfigMap
3. Обновление существующих описаний возможностей
4. Совместимость с устаревшим форматом
5. Часто задаваемые вопросы

---

1. Начало работы

1.1 Создание или обновление ConfigMap

> **Важно:** Выполняйте следующую операцию **в пространстве имён** `kube-public`, иначе платформа не распознает возможности хранилища.

Отредактируйте или создайте ConfigMap, имя которого начинается с `sd-`, например `sd-capabilities-enhanced`:

```
kubectl -n kube-public edit configmap sd-capabilities-enhanced
```

Обязательная метка

```
metadata:
  labels:
    features.alauda.io/type: StorageDescription
```

1.2 Заполнение поля `data`

Каждый `key` соответствует `provisioner` StorageClass; значение — YAML-строка, описывающая его возможности. Основные поля:

Поле	Тип	Описание
`snapshot`	`Boolean`	Указывает, поддерживаются ли снимки томов
`volumeMode`	`List[String]`	Поддерживаемые режимы томов; минимум один из `Filesystem`, `Block`
`accessModes`	`List[String]`	Режимы доступа, доступные при `volumeMode` равном `Filesystem`
`blockAccessModes`	`List[String]`	Режимы доступа, специфичные для блочных томов (опционально)

> Если `blockAccessModes` не указаны, платформа использует `accessModes` для блочных томов.

1.3 Применение конфигурации

```
kubectl apply -f sd-capabilities-enhanced.yaml
```

После применения UI автоматически подстраивает доступные опции, например:

- При выборе режима тома **Block** выпадающий список режимов доступа заполняется значениями из `blockAccessModes`.
- Если `snapshot: true`, операции, связанные со снимками, становятся доступны на странице PVC.

2. Пример ConfigMap

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: sd-capabilities-enhanced
  namespace: kube-public
  labels:
    features.alauda.io/type: StorageDescription
data:
  storage.advanced-block-fs.com: |-
    snapshot: true
    volumeMode:
      - Filesystem
      - Block
    accessModes:
      - ReadWriteOnce
      - ReadOnlyMany
    blockAccessModes:
      - ReadWriteOnce
  storage.filesystem-basic.com: |-
    snapshot: false
    volumeMode:
      - Filesystem
    accessModes:
      - ReadWriteOnce
      - ReadWriteMany
```

3. Обновление существующих описаний возможностей

1. Найдите ключ `provisioner`, который хотите изменить.
2. Отредактируйте значения полей в соответствии с реальными возможностями.
3. Повторно примените ConfigMap с помощью `kubectl apply -f ...`. Платформа опрашивает обновления и автоматически обновляет UI; вы также можете обновить страницу браузера, чтобы сразу увидеть изменения.

4. Совместимость с устаревшим форматом

- Если отсутствует `blockAccessModes`, блочные тома наследуют `accessModes`.
- Удалять старые ConfigMap не нужно; просто добавьте новые поля для плавного обновления.

5. Часто задаваемые вопросы

Симптом	Возможная причина	Решение
Список режимов доступа пуст для блочных томов	`blockAccessModes` пуст и `accessModes` тоже пусты	Укажите хотя бы одно из двух
UI по-прежнему показывает устаревшие возможности	ConfigMap не сохранён или кэш браузера	Проверьте с помощью `kubectl get cm`, обновите страницу

Устранение неполадок

Восстановление после ошибки расширения PVC

Процедура

Дополнительные советы

Восстановление после ошибки расширения PVC

Когда расширение PVC в Kubernetes завершается неудачей, администраторы могут вручную восстановить состояние Persistent Volume Claim (PVC) и отменить запрос на расширение.

Содержание

Процедура

Дополнительные советы

Процедура

1. Измените политику восстановления (reclaim policy) Persistent Volume (PV), связанного с PVC, на `Retain`. Для этого отредактируйте соответствующий PV и установите поле `persistentVolumeReclaimPolicy` в значение `Retain`.
2. Удалите исходный PVC.
3. Вручную отредактируйте PV, чтобы удалить запись `claimRef` из его спецификации. Это гарантирует, что новый PVC сможет связаться с этим PV, изменив статус PV на `Available`.

4. Воссоздайте новый PVC с меньшим размером или размером, поддерживаемым базовым поставщиком хранилища.
5. Явно укажите поле `volumeName` в новом PVC, чтобы оно совпадало с именем исходного PV. Это обеспечит точное связывание нового PVC с указанным PV.
6. Наконец, восстановите исходную политику восстановления PV.

Дополнительные советы

- Убедитесь, что используемый `StorageClass` поддерживает расширение томов, установив `allowVolumeExpansion` в `true`.
- Выполняйте эти действия осторожно, чтобы избежать риска потери данных.

Объектное хранилище

Введение

Введение

Ограничения

Основные понятия

Основные понятия

Введение в основные ресурсы и принципы Container Object Storage Interface (COSI) для администраторов Kubernetes.

Обзор

Основные ресурсы

Взаимодействие ресурсов

Итог

Установка

Установка

Предварительные требования

Установка Alauda Container Platform COSI

Удаление

Руководства

Создание BucketClass для Ce

Предварительные требования

Шаг 1 – Подготовка кластера Ceph

Шаг 2 – Установка плагина COSI

Шаг 3 – Подготовка секрета с учётными

Шаг 4 – Создание BucketClass

Проверка и дальнейшие шаги

Создание BucketClass для Mi

Предварительные требования

Шаг 1 — Подготовка кластера MinIO

Шаг 2 — Подготовка секрета с учётными

Шаг 3 — Создание BucketClass

Проверка и дальнейшие шаги

Создание з

Требования

Процедура

Связанные дей

Как сделать

Управление доступом и квотами для COSI бакетов с помощью CephObjectStoreUser (драйвер Ceph)

Предварительные требования

Шаг 1 — Создайте CephObjectStoreUser (с возможностями и квотами)

Шаг 2 — Определите BucketClass, связанный с CephObjectStoreUser

Шаг 3 — Создайте бакет с помощью BucketClaim

Шаг 4 — Выдайте минимально необходимые учетные данные с помощью BucketAccessClass/BucketAccess

Шаг 5 — Анонимное публичное чтение (опционально)

Шаг 6 — Контроль квот: где применять и как изменять

Операции и устранение неполадок

Очистка

Введение

Container Object Storage Interface (COSI) — это нативный для Kubernetes фреймворк, разработанный для обеспечения стандартизированного и декларативного подхода к управлению сервисами объектного хранения, такими как AWS S3, MinIO и Ceph RGW, внутри кластеров Kubernetes. COSI расширяет модель хранения Kubernetes, чтобы поддерживать ресурсы объектного хранения таким образом, который является переносимым, масштабируемым и соответствует принципам Kubernetes.

COSI позволяет администраторам определять, предоставлять и использовать бакеты объектного хранения через знакомые API в стиле Kubernetes. Он упрощает интеграцию между приложениями и системами объектного хранения на бэкенде, автоматизируя жизненный цикл бакетов и их учетных данных доступа. С помощью COSI пользователи Kubernetes могут динамически запрашивать ресурсы объектного хранения, снижая ручные настройки и повышая операционную эффективность.

Используя COSI, предприятия могут:

- Стандартизировать предоставление объектного хранения в различных облачных и локальных средах.
- Динамически создавать и управлять бакетами через декларативные определения ресурсов.
- Бесшовно распространять учетные данные доступа к рабочим нагрузкам через Kubernetes Secrets.
- Согласовывать управление объектным хранением с паттернами постоянного хранения Kubernetes для единого опыта.

Содержание

Ограничения

Ограничения

- В настоящее время COSI находится в альфа-версии.
- На данный момент COSI поддерживает только драйверы Ceph RGW и MinIO.
- Интеграция с устаревшими бакетами объектного хранения может потребовать дополнительных ручных настроек.

Основные понятия

Содержание

Обзор

Основные ресурсы

1. BucketClass
2. Bucket
3. BucketClaim

Взаимодействие ресурсов

Итог

Обзор

Этот документ знакомит администраторов Kubernetes, знакомых с концепциями постоянного хранения, с основными ресурсами и принципами Container Object Storage Interface (COSI). COSI предоставляет декларативный механизм управления объектным хранилищем (таким как AWS S3, MinIO и Ceph RGW), аналогичный существующим подходам управления постоянным хранилищем в Kubernetes.

Мы рассмотрим три основных ресурса в COSI — **BucketClass, Bucket и BucketClaim** — проводя аналогии с ресурсами хранения Kubernetes для прояснения их взаимосвязей и функционала.

Основные ресурсы

COSI определяет три ключевых ресурса:

1. BucketClass

Область видимости: на уровне кластера
Аналог в Kubernetes: аналог StorageClass

BucketClass создаётся администраторами кластера для определения конкретных типов или уровней сервиса бакетов, включая региональное расположение, политики избыточности и уровни производительности.

Основные функции:

- Определяет политику удаления бакета (например, удалять ли сам бакет при удалении BucketClaim)
- Указывает драйвер COSI (driverName)
- Задаёт параметры, специфичные для поставщика

Пример YAML:

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
metadata:
  name: ceph-cosi-driver-class
deletionPolicy: Delete
driverName: ceph.objectstorage.k8s.io
parameters:
  objectStoreUserSecretName: rook-ceph-object-user-object-store-user-for-cosi
  objectStoreUserSecretNamespace: rook-ceph
```

2. Bucket

Область видимости: на уровне кластера
Аналог в Kubernetes: аналог PersistentVolume (PV)

Bucket представляет собой абстракцию реального бакета, существующего во внешней системе объектного хранения (например, AWS S3, MinIO, Ceph RGW) внутри Kubernetes.

Управление жизненным циклом:

- **Динамическое создание**: автоматически создаётся контроллером COSI при получении запроса BucketClaim.

3. BucketClaim

Область видимости: в пределах namespace
Аналог в Kubernetes: аналог PersistentVolumeClaim (PVC)

Ресурсы BucketClaim создаются разработчиками приложений в своих пространствах имён для запроса бакетов объектного хранилища.

Последовательность действий:

1. Пользователь создаёт BucketClaim, указывая BucketClass.
2. Контроллер COSI обнаруживает запрос и динамически создаёт бакет в бекенде объектного хранилища на основе определения BucketClass.
3. Создаётся соответствующий ресурс Bucket, который связывается с BucketClaim.
4. Генерируется Secret с учётными данными доступа к бакету, который автоматически монтируется в Pod’ы, запрашивающие бакет.

Пример YAML:

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClaim
metadata:
  name: my-app-bucket-claim
  namespace: my-app-ns
spec:
  bucketClassName: ceph-standard-replicated
  protocol:
    s3: {} # Defaults populated by the driver
```

Взаимодействие ресурсов

Ниже описан процесс динамического создания ресурсов COSI на практике:

1. **Администратор кластера** создаёт и поддерживает BucketClass.
2. **Пользователь namespace** создаёт BucketClaim с ссылкой на BucketClass.
3. **Контроллер COSI** обнаруживает BucketClaim и динамически создаёт бакет на основе определения BucketClass.
4. Контроллер создаёт соответствующий ресурс Bucket в Kubernetes.
5. BucketClaim и Bucket связываются.
6. Создаётся Secret с учётными данными для использования в Pod’ах.
7. Pod’ы монтируют Secret и получают доступ к объектному хранилищу.

Итог

Ресурс COSI	Область видимости	Аналог в Kubernetes	Назначение
BucketClass	На уровне кластера	StorageClass	Определяет типы бакетов и политики
Bucket	На уровне кластера	PersistentVolume (PV)	Абстракция Kubernetes для реального бакета
BucketClaim	В пределах namespace	PersistentVolumeClaim (PVC)	Запрос пользователя на ресурсы бакета

Используя стандартизированные API, предоставляемые COSI, администраторы Kubernetes могут декларативно и переносимо управлять ресурсами объектного хранилища, значительно повышая эффективность интеграции приложений с объектным хранилищем в кластерах Kubernetes.

Установка

Содержание

Предварительные требования

Установка Alauda Container Platform COSI

Ограничения и условия

Процедура

Удаление

Предварительные требования

1. **Скачайте** пакет плагина кластера **Alauda Container Platform COSI**, соответствующий архитектуре вашей платформы.
2. **Скачайте** пакет плагина кластера либо **Alauda Container Platform COSI for Ceph**, либо **Alauda Container Platform COSI for MinIO**, в зависимости от выбранного вами решения для объектного хранилища.
3. **Загрузите** скачанные пакеты плагинов с помощью функции **Upload Packages**.
4. **Установите** пакеты плагинов в целевой кластер с помощью механизма установки **Cluster Plugins**.

INFO

Upload Packages:

Platform Management > **Marketplace** > страница **Upload Packages**.

Нажмите **Help Document** справа, чтобы получить инструкции по публикации плагина кластера в нужном кластере. Для получения дополнительной информации обратитесь к разделу CLI.

Установка Alauda Container Platform COSI

Ограничения и условия

- Плагины применимы только к текущему кластеру. Необходимо устанавливать требуемые плагины отдельно на каждом кластере, где вы хотите включить функции COSI.
- Плагины **Alauda Container Platform COSI for Ceph** и **Alauda Container Platform COSI for MinIO** зависят от плагина **Alauda Container Platform COSI**. Убедитесь, что плагин **Alauda Container Platform COSI** установлен первым.

Процедура

1. Войдите в платформу и перейдите на страницу **Platform Management**.
2. Перейдите в **Marketplace** > **Cluster Plugins**, чтобы открыть список доступных плагинов кластера.
3. Выберите целевой кластер, в который хотите установить плагины.
4. Найдите плагин **Alauda Container Platform COSI** и выберите **Install** в меню ⋮ для начала установки.
5. Найдите и установите плагин **Alauda Container Platform COSI for Ceph** или **Alauda Container Platform COSI for MinIO** в зависимости от выбранного бекенда.
6. Дождитесь, пока статус всех плагинов не изменится на **Installed**.

Удаление

1. Войдите в платформу и перейдите на страницу **Platform Management**.
2. Перейдите в **Marketplace** > **Cluster Plugins**, чтобы открыть список установленных плагинов кластера.
3. Выберите целевой кластер, из которого хотите удалить плагины.
4. Найдите плагин, который хотите удалить, и выберите **Uninstall** в меню ⋮ для начала процесса удаления.
5. Дождитесь, пока статус плагина не изменится на **Ready**, что означает возможность повторной установки при необходимости.

Важно: Перед удалением плагина **Alauda Container Platform COSI** необходимо сначала удалить плагины **Alauda Container Platform COSI for Ceph** и/или **Alauda Container Platform COSI for MinIO**.

Руководства

Создание BucketClass для Ce

Предварительные требования
Шаг 1 – Подготовка кластера Ceph
Шаг 2 – Установка плагина COSI
Шаг 3 – Подготовка секрета с учётными
Шаг 4 – Создание BucketClass
Проверка и дальнейшие шаги

Создание BucketClass для Mi

Предварительные требования
Шаг 1 — Подготовка кластера MinIO
Шаг 2 — Подготовка секрета с учётными
Шаг 3 — Создание BucketClass
Проверка и дальнейшие шаги

Создание з

Требования
Процедура
Связанные дей

Создание BucketClass для Ceph RGW

Ceph Object Storage может быть предоставлено рабочим нагрузкам Kubernetes через **Container Object Storage Interface (COSI)**, обеспечивая высокомасштабируемое и эластичное хранилище для сценариев анализа больших данных, резервного копирования и восстановления, а также машинного обучения. Перед тем как пользователи смогут создавать bucket, необходимо создать *BucketClass*.

BucketClass — это шаблонный ресурс, который задаёт драйвер хранения, секрет аутентификации и политику удаления, применяемые ко всем bucket, созданным на его основе.

Содержание

- Предварительные требования
- Шаг 1 – Подготовка кластера Ceph
- Шаг 2 – Установка плагина COSI
- Шаг 3 – Подготовка секрета с учётными данными
  - Метод A – Автогенерация (Ceph под управлением Rook)
  - Метод B – Вручную (Внешний Ceph)
- Шаг 4 – Создание BucketClass
  - Вариант 1 – Через UI
  - Вариант 2 – YAML (подходит для GitOps)
- Проверка и дальнейшие шаги

Предварительные требования

Требование	Примечания
Запущенный кластер Ceph с включённым RGW (S3)	Подходит как внутренний (управляемый Rook), так и внешний кластер.
Плагины Alauda Container Platform COSI	Должны быть установлены **Alauda Container Platform COSI** *и* **Alauda Container Platform COSI for Ceph**.
Kubernetes Secret с учётными данными Ceph RGW	Подготавливается на **Шаге 3** ниже.

Шаг 1 – Подготовка кластера Ceph

Выберите **один** из вариантов:

Вариант	Описание
Внутренний Ceph	Кластер Ceph, развернутый и управляемый **внутри** платформы оператором Rook. Подробности см. в *create a storage service*.
Внешний Ceph	Отдельный кластер Ceph, доступный из сети платформы.

Шаг 2 – Установка плагина COSI

Установите следующие плагины кластера:

1. **Alauda Container Platform COSI**
2. **Alauda Container Platform COSI for Ceph**

> См. *Installing* для точных команд.

Шаг 3 – Подготовка секрета с учётными данными

COSI получает учётные данные RGW из Kubernetes **Secret**. Выберите **один** метод в зависимости от вашего развертывания Ceph.

Метод A – Автогенерация (Ceph под управлением Rook)

1. Создайте ресурс **CephObjectStoreUser** в пространстве имён *rook-ceph*:

```yaml
# ceph-object-store-user.yaml
apiVersion: ceph.rook.io/v1
kind: CephObjectStoreUser
metadata:
  name: user-for-cosi
  namespace: rook-ceph
spec:
  store: object-store              # имя вашего CephObjectStore
  capabilities:
    bucket: ["read", "write"]
    user:   ["read", "write"]
```

2. Примените манифест:

```
kubectl apply -f ceph-object-store-user.yaml
```

3. Получите имя автоматически созданного секрета (будет использоваться далее):

```
kubectl get cephobjectstoreuser user-for-cosi -n rook-ceph \
  -o jsonpath='{.status.info.secretName}'
```

Метод B – Вручную (Внешний Ceph)

1. Получите **AccessKey**, **SecretKey** и **RGW Endpoint**.

2. Создайте Secret в нужном проекте/пространстве имён и добавьте метку, чтобы UI мог его обнаружить:

```
kubectl create secret generic ceph-external-creds -n <YOUR_NAMESPACE> \
  --from-literal=AccessKey=<YOUR_ACCESS_KEY> \
  --from-literal=SecretKey=<YOUR_SECRET_KEY> \
  --from-literal=Endpoint=http://<YOUR_RGW_ENDPOINT>

kubectl label secret ceph-external-creds -n <YOUR_NAMESPACE> app=rook-ceph-rgw
```

> **Важно:** Метка `app=rook-ceph-rgw` обязательна для отображения секрета в UI платформы.

Шаг 4 – Создание BucketClass

Вариант 1 – Через UI

1. Перейдите в **Storage** → **Object StorageClass** и нажмите **Create Object StorageClass**.
2. Выберите драйвер **Ceph Object Storage**.
3. Заполните следующие поля:
   - **Deletion Policy** – как обрабатывается базовый bucket при удалении его BucketClaim (по умолчанию: `Delete`).
   - **Secret** – выберите секрет, подготовленный на *Шаге 3* (отображаются только секреты с меткой `app=rook-ceph-rgw`).
   - **Allocate Projects** – *(необязательно)* ограничение использования конкретными проектами.
4. Нажмите **Create**.

Вариант 2 – YAML (подходит для GitOps)

Создайте файл `ceph-bucketclass.yaml` с корректными ссылками на секрет:

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
metadata:
  name: ceph-cosi-driver
  labels:
    project.cpaas.io/ALL_ALL: "true"
driverName: ceph.objectstorage.k8s.io
deletionPolicy: Delete
parameters:
  objectStoreUserSecretName: <your-secret-name>
  objectStoreUserSecretNamespace: <your-secret-namespace>
```

Примените манифест:

```bash
kubectl apply -f ceph-bucketclass.yaml
```

Проверка и дальнейшие шаги

Проверьте BucketClass:

```bash
kubectl get bucketclass
```

После готовности BucketClass вы можете создавать ресурсы **Bucket** или **BucketClaim**, ссылающиеся на него, тем самым предоставляя S3-совместимое объектное хранилище для ваших приложений.

Создание BucketClass для MinIO

MinIO интегрируется с Kubernetes через **Container Object Storage Interface (COSI)**, обеспечивая масштабируемое объектное хранилище, совместимое с S3, для аналитики, резервного копирования и восстановления, а также для рабочих нагрузок ML/AI. Перед созданием бакетов необходимо определить *BucketClass*.

BucketClass — это шаблонный ресурс, который задаёт драйвер хранения, секрет аутентификации и политику удаления, применяемую ко всем бакетам, созданным на его основе.

Содержание

- Предварительные требования
- Шаг 1 — Подготовка кластера MinIO
- Шаг 2 — Подготовка секрета с учётными данными
- Шаг 3 — Создание BucketClass
  - Вариант 1 — через UI
  - Вариант 2 — YAML (GitOps-дружественный)
- Проверка и дальнейшие шаги

Предварительные требования

Требование	Примечания
Кластер MinIO готов к использованию	Подготовьте MinIO, следуя руководству по установке.
Плагины COSI для Alauda Container Platform	Должны быть установлены **acp-cosi** и **acp-cosi-minio**. См. Установка плагинов COSI для инструкций.
Kubernetes Secret с учётными данными MinIO	Подготовлен на **Шаге 2**.

Шаг 1 — Подготовка кластера MinIO

Убедитесь, что кластер MinIO установлен и доступен. Следуйте документации по установке MinIO для развертывания и настройки вашей среды MinIO.

Шаг 2 — Подготовка секрета с учётными данными

COSI получает учётные данные MinIO из Kubernetes **Secret**. Соберите следующие значения:

- `Endpoint` — например, `http://minio.minio-system.svc` или `https://minio.example.com:9000`
- `AccessKey`
- `SecretKey`

Создайте Secret в целевом namespace и добавьте метку для обнаружения в UI:

```
kubectl create secret generic minio-credentials -n <YOUR_NAMESPACE> \
  --from-literal=Endpoint=http://<YOUR_MINIO_ENDPOINT> \
  --from-literal=AccessKey=<YOUR_ACCESS_KEY> \
  --from-literal=SecretKey=<YOUR_SECRET_KEY>

kubectl label secret minio-credentials -n <YOUR_NAMESPACE> app=minio
```

Важно: Метка `app=minio` необходима для отображения секрета в UI платформы.

Примечание: Имена ключей **чувствительны к регистру** и должны быть точно `Endpoint`, `AccessKey` и `SecretKey`.

Если вы предпочитаете GitOps, можно определить Secret декларативно:

```
apiVersion: v1
kind: Secret
metadata:
  name: minio-credentials
  namespace: <YOUR_NAMESPACE>
  labels:
    app: minio
type: Opaque
stringData:
  Endpoint: http://<YOUR_MINIO_ENDPOINT>
  AccessKey: <YOUR_ACCESS_KEY>
  SecretKey: <YOUR_SECRET_KEY>
```

Шаг 3 — Создание BucketClass

Вариант 1 — через UI

1. Перейдите в **Storage** → **Object StorageClass** и нажмите **Create Object StorageClass**.
2. Выберите драйвер **MinIO Object Storage**.
3. Заполните следующие поля:

- **Deletion Policy** — как будет обрабатываться базовый бакет при удалении его BucketClaim (по умолчанию: `Delete`).
- **Secret** — выберите секрет, созданный на Шаге 2 (отображаются только секреты с меткой `app=minio`).
- **Allocate Projects** — необязательно: ограничить использование определёнными проектами.

4. Нажмите **Create**.

Вариант 2 — YAML (GitOps-дружественный)

Создайте файл `minio-bucketclass.yaml`. В примере ниже используется драйвер MinIO COSI и ссылка на секрет с правильными параметрами.

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
driverName: minio.objectstorage.k8s.io
metadata:
  labels:
    project.cpaas.io/name: null
    project.cpaas.io/ALL_ALL: "true"
  name: minio-bucket-class
  annotations:
    cpaas.io/display-name: BucketClass for MinIO
    cpaas.io/access-mode: ""
    cpaas.io/features: ""
parameters:
  providerSecretName: <your-secret-name>
  providerSecretNamespace: <your-secret-namespace>
deletionPolicy: Delete
```

Примените манифест:

```bash
kubectl apply -f minio-bucketclass.yaml
```

Проверка и дальнейшие шаги

Проверьте BucketClass:

```
kubectl get bucketclass
```

После готовности BucketClass создайте ресурсы **Bucket** или **BucketClaim**, ссылающиеся на него, чтобы обеспечить объектное хранилище, совместимое с S3, на базе MinIO.

Создание запроса на Bucket

Используйте **Bucket Request** для динамического создания bucket на основе **Object Storage Class** и автоматического связывания их.

Содержание

Требования

Процедура

Связанные действия

Требования

- **Установите** кластер **Alauda Container Platform COSI**.
- **Установите** пакет плагина кластера **Alauda Container Platform COSI for Ceph** или **Alauda Container Platform COSI for MinIO** в зависимости от выбранного решения для объектного хранилища.

Подробные шаги установки смотрите в разделе Installing.

Процедура

1. Переключитесь на вид **Container Platform**.

2. В левом навигационном меню выберите **Storage > Bucket Claims**.
3. Нажмите **Create a Bucket Request**.
4. Настройте параметры следующим образом.

Параметр	Описание
Name	Имя запроса на bucket.
Object Storage Class	Object Storage Class, используемый для динамического создания bucket и установления связи.

5. Нажмите **Create**. Дождитесь, пока статус не станет `Available`, что означает выполнение запроса и завершение связывания.

Связанные действия

На странице с деталями запроса на bucket нажмите **Actions** (в правом верхнем углу), чтобы при необходимости **Delete bucket policy**. **Внимание:** удаление политики bucket приведёт к очистке всех данных в bucket. Действуйте осторожно и убедитесь в наличии резервного копирования и соблюдении требований безопасности перед удалением.

Как сделать

Управление доступом и квотами для COSI бакетов с помощью CephObjectStoreUser (драйвер Ceph)

Предварительные требования

Шаг 1 — Создайте CephObjectStoreUser (с возможностями и квотами)

Шаг 2 — Определите BucketClass, связанный с CephObjectStoreUser

Шаг 3 — Создайте бакет с помощью BucketClaim

Шаг 4 — Выдайте минимально необходимые учетные данные с помощью BucketAccessClass/BucketAccess

Шаг 5 — Анонимное публичное чтение (опционально)

Шаг 6 — Контроль квот: где применять и как изменять

Операции и устранение неполадок

Очистка

Управление доступом и квотами для COSI бакетов с помощью CephObjectStoreUser (драйвер Ceph)

В этом руководстве показано, как администраторам Kubernetes сочетать **CephObjectStoreUser (COSU)**, **BucketClass/BucketClaim** и **BucketAccessClass/BucketAccess** для реализации **минимально необходимого доступа** и **контроля квот** для бакетов COSI на базе Ceph RGW.

> **Что вы создадите**
>
> 1. CephObjectStoreUser с явными возможностями и опциональными квотами на пользователя;
> 2. BucketClass, который указывает драйверу Ceph COSI, какие учетные данные COSU использовать;
> 3. Один или несколько BucketClaim для создания бакетов;
> 4. Тонко настроенные учетные данные на уровне нагрузки с использованием BucketAccessClass/BucketAccess (только чтение, только запись, чтение-запись), с опциональным анонимным чтением.

Содержание

Предварительные требования

Шаг 1 — Создайте CephObjectStoreUser (с возможностями и квотами)

Шаг 2 — Определите BucketClass, связанный с CephObjectStoreUser

Шаг 3 — Создайте бакет с помощью BucketClaim

Шаг 4 — Выдайте минимально необходимые учетные данные с помощью BucketAccessClass/BucketAccess

Пример `BucketAccessClass` (только чтение)

Создайте учетные данные с помощью `BucketAccess`

Шаг 5 — Анонимное публичное чтение (опционально)

Шаг 6 — Контроль квот: где применять и как изменять

Операции и устранение неполадок

Очистка

Предварительные требования

- Рабочий кластер Ceph с установленными RGW и Rook.
- Установленные плагины COSI.
- Привилегии администратора кластера (для создания ресурсов с областью действия кластера).

Шаг 1 — Создайте CephObjectStoreUser (с возможностями и квотами)

`CephObjectStoreUser` — это сервисный аккаунт, который драйвер использует для операций с бакетами в RGW. Он должен находиться в пространстве имен **`rook-ceph`** и ссылаться на ваш `CephObjectStore`.

```yaml
apiVersion: ceph.rook.io/v1
kind: CephObjectStoreUser
metadata:
  name: user-for-cosi
  namespace: rook-ceph
spec:
  # Целевой CephObjectStore
  store: object-store
  # Необходимые возможности для жизненного цикла COSI бакета
  capabilities:
    bucket: read, write
    user: read, write
  # Опциональные квоты на пользователя, применяемые RGW
  quotas:
    maxBuckets: 50        # ограничение на количество бакетов, которыми м
ожет владеть пользователь
    maxObjects: 500       # общее количество объектов во всех бакетах (пр
имер)
    maxSize: 100Gi        # общий логический размер во всех бакетах
  displayName: "User for COSI driver"
```

Получите сгенерированные ключи доступа (Rook создаст Secret для этого пользователя):

```bash
kubectl get cephobjectstoreuser user-for-cosi -n rook-ceph -o yaml
# Посмотрите status.info.secretName -> имя Secret с AccessKey/SecretKey
```

> Эти учетные данные COSU используются **драйвером** (не вашими приложениями) для создания/удаления бакетов от вашего имени.

Шаг 2 — Определите BucketClass, связанный с CephObjectStoreUser

`BucketClass` указывает драйверу, какой Secret COSU использовать при создании бакетов.

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
metadata:
  name: ceph-cosi-driver-class
deletionPolicy: Delete
# Должно совпадать с именем драйвера
driverName: ceph.objectstorage.k8s.io
parameters:
  # Ссылка на Secret, созданный для CephObjectStoreUser
  objectStoreUserSecretName: <secret-name-from-step-1>
  objectStoreUserSecretNamespace: rook-ceph
```

> `deletionPolicy` управляет физическим жизненным циклом бакета при удалении `BucketClaim` (`Delete` или `Retain`).

Шаг 3 — Создайте бакет с помощью BucketClaim

Создайте бакет в пространстве имен вашего приложения, ссылаясь на `BucketClass`.

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClaim
metadata:
  name: my-bucket-claim
  namespace: app-a
spec:
  bucketClassName: ceph-cosi-driver-class
  # Требуется API COSI (выберите нужный протокол данных)
  protocols:
  - S3
```

Дождитесь, пока `status.bucketReady: true`, и обратите внимание на `status.bucketName` — фактическое имя бакета в RGW.

Шаг 4 — Выдайте минимально необходимые учетные данные с помощью BucketAccessClass/BucketAccess

Определите шаблоны политик с помощью `BucketAccessClass` и создавайте учетные данные для каждой нагрузки через `BucketAccess`. Поддерживаемые политики: **`readonly`**, **`writeonly`**, **`readwrite`**. Анонимное чтение доступно при установке `parameters.anonymous: "true"` (строка).

Пример `BucketAccessClass` (только чтение)

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketAccessClass
metadata:
  name: ceph-readonly-access-class
authenticationType: KEY
driverName: ceph.objectstorage.k8s.io
parameters:
  policy: readonly
  anonymous: "false"
```

Создайте учетные данные с помощью `BucketAccess`

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketAccess
metadata:
  name: my-bucket-readonly-access
  namespace: app-a
spec:
  bucketAccessClassName: ceph-readonly-access-class
  bucketClaimName: my-bucket-claim
  credentialsSecretName: my-bucket-readonly-credentials
```

Драйвер создаст Secret с именем из `credentialsSecretName`. Раскодируйте `.data.BucketInfo` (base64), чтобы получить `secretS3.endpoint`, `accessKeyID` и

`accessSecretKey` для вашего S3 клиента.

> **Совет**: выдавайте **отдельные** учетные данные для каждого Deployment/Job, чтобы упростить ротацию и отзыв без сбоев у других нагрузок.

Шаг 5 — Анонимное публичное чтение (опционально)

Если требуется публичный хостинг статических ресурсов:

```
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketAccessClass
metadata:
  name: ceph-anonymous-readonly-class
authenticationType: KEY
driverName: ceph.objectstorage.k8s.io
parameters:
  policy: readonly
  anonymous: "true"
```

Свяжите его с `BucketAccess` для вашего `BucketClaim`. После предоставления доступа объекты будут доступны через неаутентифицированный HTTP `GET` (обеспечьте соответствующую сетевую доступность).

Шаг 6 — Контроль квот: где применять и как изменять

Область действия: блок `quotas` в **CephObjectStoreUser** применяется **на пользователя** и контролируется RGW для всех бакетов, принадлежащих этому пользователю.

- `maxBuckets`: верхний предел количества бакетов, которые пользователь может создать/владеть.

- `maxObjects`: максимальное количество объектов, которые пользователь может хранить (во всех бакетах).
- `maxSize`: общий разрешенный логический размер для пользователя.

Обновите квоты, отредактировав ресурс COSU:

```
kubectl -n rook-ceph edit cephobjectstoreuser user-for-cosi
# Измените spec.quotas.{maxBuckets,maxObjects,maxSize}; сохраните и выйдите.
# Rook выполнит согласование и применит новые квоты RGW пользователя.
```

> **Выбор архитектуры:** держите **квоты COSU** достаточно жесткими, чтобы ограничить зону поражения. Используйте **политики с минимальными правами** через BAC/BA, чтобы ограничить возможности конкретных учетных данных приложений **внутри** бакета.

Операции и устранение неполадок

- **Размещение в пространстве имен**: `CephObjectStoreUser` и его Secret должны находиться в `rook-ceph`. Ресурсы уровня приложения (`BucketClaim`, `BucketAccess`) — в пространстве имен приложения.
- **Политика не применяется**: проверьте `bucketAccessClassName` и `parameters.policy` в BAC (`readonly|writeonly|readwrite`).
- **Анонимное чтение не работает**: убедитесь, что `anonymous: "true"` — это **строка**, а не булево значение; проверьте доступность endpoint и путь HTTP (`/<bucket>/<object>`).
- **Не удается найти ключи**: проверьте Secret `BucketAccess` и раскодируйте `.data.BucketInfo`.
- **Бакет остается неготовым**: проверьте логи контроллера/драйвера (например, `kubectl -n cpaas-system logs deploy/ceph-cosi-driver -c ceph-cosi-driver`).
- **Ротация**: создайте новый `BucketAccess`, переключите нагрузки на новый Secret, затем удалите старый Secret/BA.

Очистка

- Удалите учетные данные нагрузки, удалив соответствующий `BucketAccess` и связанный Secret.
- Для удаления бакета удалите `BucketClaim` (поведение зависит от `BucketClass.deletionPolicy`). Если бэкенд отказывается удалять бакет из-за того, что он не пуст, сначала удалите объекты.