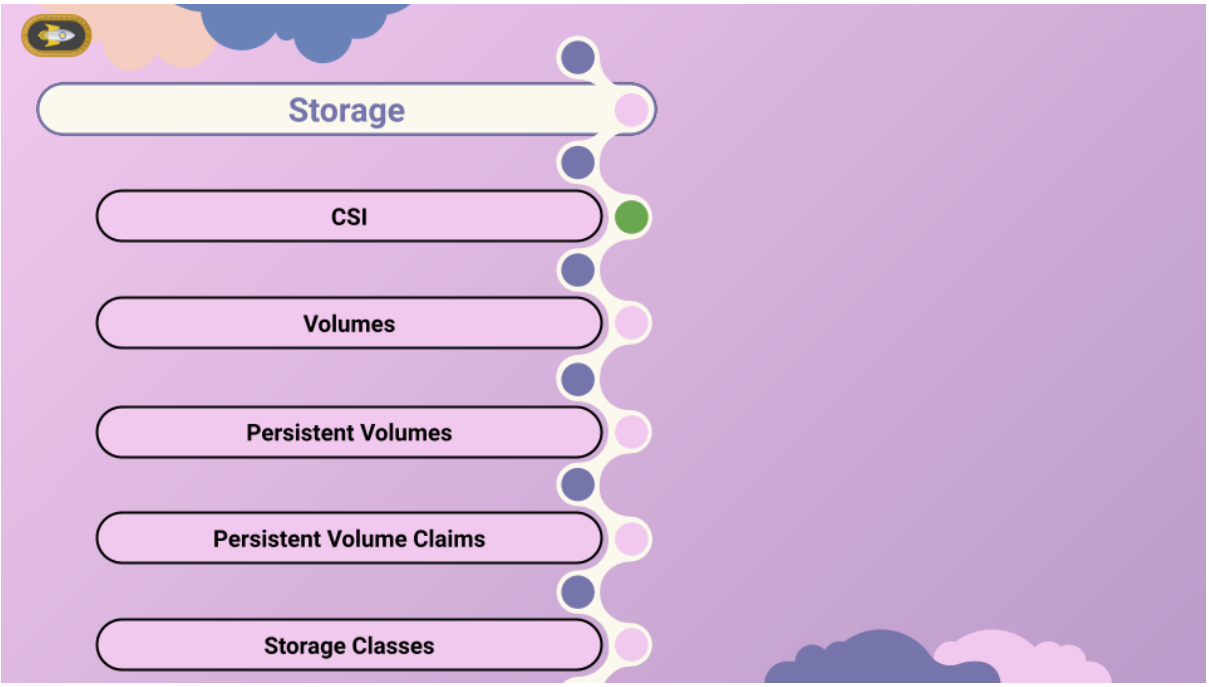


Привет. Мы изучаем хранение в Kubernetes.

Для понимания хранилища в инструментах оркестрации контейнеров, вроде Kubernetes, сначала требуется понять принципы хранилищ, реализованные в контейнерных движках. Следующие две лекции посвящены этому.

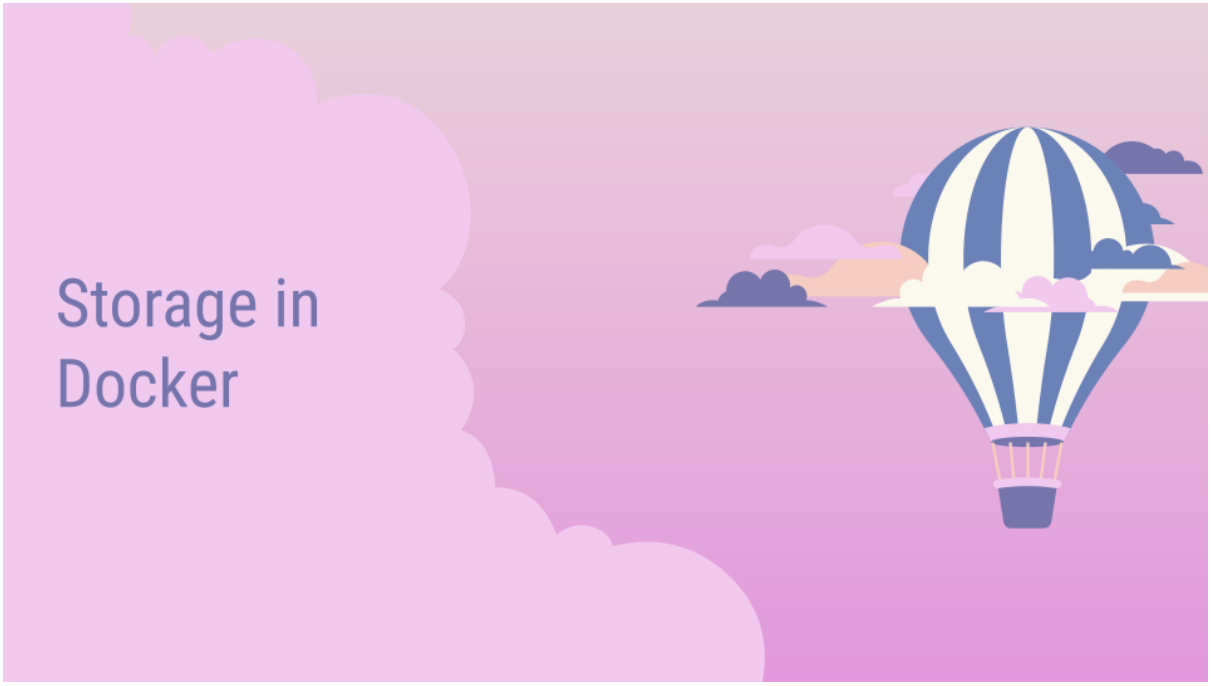
Понимание того, как Docker организует свой storage, значительно упростит понимание основ того, как это работает в Kubernetes.

В Docker есть две концепции, которые ты должен знать - это драйвера хранилища (storage drivers) и подключаемых драйвера тома (volume driver plugins).

В следующем видео мы поговорим о storage drivers.
Это то, что мы обсуждали в курсе Docker.

Так что, если ты уже прошел тот курс, не стесняйся пропустить следующее видео. Или ты можешь остаться и освежить свою память.

После этого мы поговорим о volume drivers. Увидимся!



Привет и добро пожаловать на лекцию.

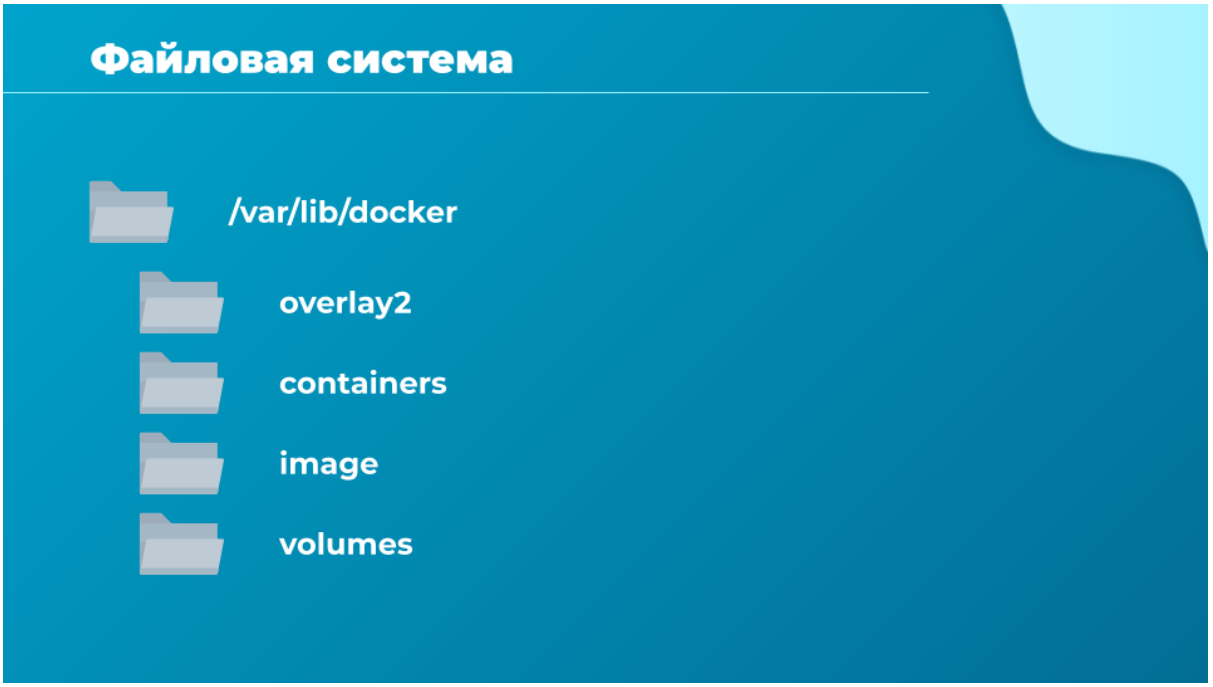
В этой лекции мы изучаем продвинутые концепции Docker, а именно поговорим драйверах хранилища Docker и файловых системах.

Мы посмотрим, где и как Docker хранит данные, и как он управляет файловыми системами контейнеров.

Давай начнем с того, как Docker хранит данные в локальной файловой системе.

Когда мы устанавливаем Docker в систему, он создает некоторую структуру папок в каталоге /var/lib/docker.

Там находится несколько папок, с названиями overlay2, containers, image, volumes и т. д.



Здесь Docker по умолчанию хранит все свои данные.

Когда я говорю «данные», я имею в виду файлы, связанные с образами и контейнерами, запущенными на этом докер-хосте.

Например, все файлы, относящиеся к контейнерам, хранятся в папке containers, а файлы, связанные с образами, хранятся в папке image.

Любые тома, созданные для докер-контейнеров, создаются в папке volumes.

Что ж, пока не беспокойся об этом. Мы вернемся к этому чуть позже. А пока давай просто разберемся, где Docker хранит свои файлы и в каком формате.
Итак, как именно Docker хранит файлы образа и контейнера?

Чтобы это хорошо понять, нужно вникнуть в многоуровневую или слоеную архитектуру Docker.

Давай быстро вспомним то, что мы узнали об этом.

Когда Docker создает образы, он создает их в многоуровневой архитектуре.
Каждая строка инструкции в Dockerfile создает новый слой в докер-образе с фиксацией изменений от предыдущего уровня.

Например, первый уровень - это базовая операционная система Ubuntu, за которой следует вторая инструкция, которая создает второй слой, который устанавливает все пакеты APT.
Затем третья инструкция создает третий уровень, который занимается пакетами python, за которым следует четвертый уровень, который копирует исходный код.
И, наконец, пятый слой, который обновляет точку входа образа.
Поскольку каждый слой сохраняет только изменения из предыдущего, это отражается и на размере.
Если мы посмотрим на базовый размер образа, он имеет размер около 80 мегабайт.
Пакеты apt, которые были установлены составляют около 450 МБ, а оставшиеся слои малы, чтобы понять преимущества этой многоуровневой архитектуры.

Теперь посмотрим второе приложение, в котором есть другой Dockerfile, но он очень похож на наше первое приложение.
В нем используется тот же базовый образ, что и раньше, это ubuntu и оно использует те же зависимости Python и Flask, но имеет в себе другой исходный код для создания образа и также другую точку входа.

Когда я запускаю команду docker build, чтобы создать новый образ для этого приложения, первые три уровня обоих приложений будут одинаковы и Docker не будет создавать эти первые три уровня.
Вместо этого он повторно применит одинаковые, уже созданные первые три слоя, которые он сделал для первого приложения.
Их он возьмет из кеша, в который ложит все слои, когда-либо собранные на докер-хосте.

Далее, он создаст только последние два слоя с новым исходным кодом и новой точкой входа. Вновь собранные слои он также положит в кэш. Таким образом Docker создает образы быстрее и эффективно экономит дисковое пространство.

Это также применимо, если требуется обновить код своего приложения.

Всякий раз, когда нужно обновлить исходный код, например app.py, как в нашем случае, Docker просто повторно использует все предыдущие слои из кеша и быстро перестроит образ приложения, обновив слои с исходным кодом и следующие за ним. Таким образом, это сохранит нам много времени во время перестроек и обновлений образов.

Для наглядности, давай разместим слои снизу вверх, чтобы мы могли все это лучше понять.

Внизу у нас есть базовый уровень ubuntu, затем пакеты, затем зависимости, а затем исходный код приложения и точка входа.

Все эти слои создаются, когда мы запускаем команду docker build.
Формируется окончательный образ Docker, состоящий из этих слоев.

После завершения создания образа мы не можем изменять содержимое этих слоев, они доступны только для чтения, а изменить их возможно только через запуск новой сборки.

Когда мы запускаем контейнер, основанный на этом образе при помощи команды docker run, Docker создает контейнер на основе этих слоев, а далее создает новый слой поверх слоев образа, с которого можно не только читать, но и в который можно писать.

Слой с возможностью записи используется для хранения и изменения данных, созданных контейнером.
Таких данных, как логи, временные файлы и прочие вещи, необходимые для работы приложения.
В него также попадает любой файл, измененный пользователем в этом контейнере.
Срок жизни этого слоя ограничен временем, пока жив контейнер.
Когда контейнер разрушается, этот слой со всеми сохраненные в нем изменения уничтожается.

Помни, что одни и те же слои образа делятся между всеми контейнерами, запущенными с использованием этого образа.
Скажем, я зайду в созданный контейнер и создам там новый файл с именем temp.txt.
Этот файл будет создан на уровне контейнера, который доступен для чтения и записи.

Как мы только что сказали, что файлы в слоях образа доступны только для чтения, что означает, что мы не можем ничего редактировать в этих слоях.

Теперь давай возьмем пример кода нашего приложения, поскольку мы запекли наш код в образ. Код является частью одного их слоев образа, и, следовательно он будет доступен только для чтения после запуска контейнера.

Что, если я хочу изменить исходный код, скажем для какого-то теста?

Вспоминаем, что один и тот же слой образа может использоваться несколькими контейнерами, созданными из этого образа.
Значит ли это, что я не могу изменить этот файл внутри контейнера?

Нет, я все еще могу изменить этот файл, но прежде, чем я сохраню измененный файл, Docker автоматически создает копию файла на уровне чтения и записи, и затем я буду изменять и работать уже с другой версией файла, которая уже будет находится в другом слое - в слое контейнера, слое чтения и записи.

Все будущие изменения будут внесены в эту копию файла на уровне чтения-записи.

Это называется механизмом копирования при записи (copy-on-write).

Слои образа доступны только для чтения, это означает, что файлы в этих слоях не будут изменены на самом деле, образ будет оставаться неизменным все время, пока ты не перестроишь его с помощью команды docker build.

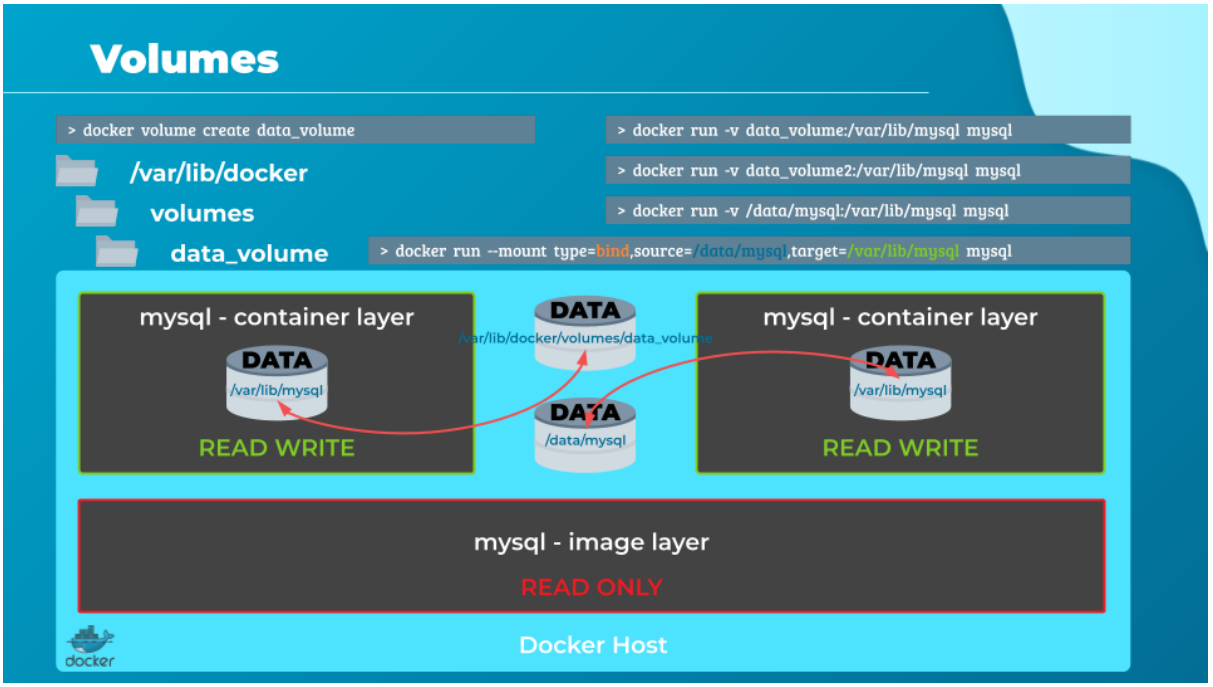
Итак, пока файлы не изменились, их не существует в слое контейнера. Они все где-то в слоях образа. Что происходит, когда мы избавляемся от контейнера?

Все данные, которые хранились на уровне контейнера, также удаляются.

Изменения, которые мы внесли в app.py и новый временный файл, который мы создали, также будут удалены.

А что, если мы хотим сохранить эти данные?

Например, мы работали с базой данных и хотели бы сохранить данные, созданные контейнером.
Для этих целей мы можем добавить в контейнер постоянный том (volume).



Чтобы сделать это, сначала создадим том с помощью команды docker volume create.

Когда мы запускаем команду docker volume create data_volume, она создает папку с именем data_volume в каталоге /var/lib/docker/volumes.

Затем, когда я запускаю докер-контейнер с помощью команды docker run, я могу смонтировать этот том внутрь контейнера для чтения и записи, используя опцию -v, как ты видишь на экране.

Я указал в docker run -v имя моего вновь созданного тома, за которым идет двоеточие и местоположение внутри моего контейнера, которым является местоположением по умолчанию, где MySQL хранит данные, т.е. /var/lib/mysql.
Далее указал имя образа.

Теперь Docker создаст новый контейнер и смонтирует созданный ранее том в папку /var/lib/mysql внутри контейнера, и все данные, записанные базой данных, фактически будут хранится в томе, созданном на докер-хосте.
Даже если контейнер разрушится, данные остаются целыми.

А что, если мы не запускали команду docker volume create для создания тома, прежде чем сделали docker run?

Например, я запускаю команду docker run для создания нового экземпляра контейнера mysql с параметром монтирования тома data_volume_2, но том я еще не создал.
В таком случае Docker автоматически создаст том с именем data_volume_2 и подключит его к контейнеру.

Мы можем найти все эти созданные тома, если просмотрим содержимое папки /var/lib/docker/volume.

Это называется volume mounting, поскольку мы монтируем том, созданный Docker в папке /var/lib/docker/volume.

Но что, если мы уже разместили данные в другом месте и не хотим их переносить в папку /var/lib/docker/volume?
Например, у нас есть директория в хранилище докер-хоста в размещении /data, и мы хотели бы хранить данные базы данных в качестве тома в этой папке, а не в директории томов docker по умолчанию.

В этом случае мы также запустим контейнер с помощью команды docker run -v.
Но в этом случае мы предоставим полный путь в папку, которую хотим смонтировать.

Это /data/mysql, и Docker создаст контейнер и подключит папку к контейнеру. Это называется bind mounting.

Таким образом, существует два типа монтирования: монтирование тома и монтирование с привязкой. Volume mounting монтирует том из каталога томов, а bind mounting монтирует каталог из любого места на докер-хосте.

И напоследок отмечу, что использование -v - это старый стиль. Новый способ - использовать опцию --mount. Это считается более предпочтительным способом, поскольку он более подробный, хотя на практике я его встречаю реже.

В этом случае, мы должны указать каждый параметр в формате «ключ равно значение».

Например, предыдущая команда может быть записана с параметром --mount, и далее указав опции source и target.

Type этом случае - bind, источник - это местоположение на моем хосте, а цель - это местоположение в моем контейнере.

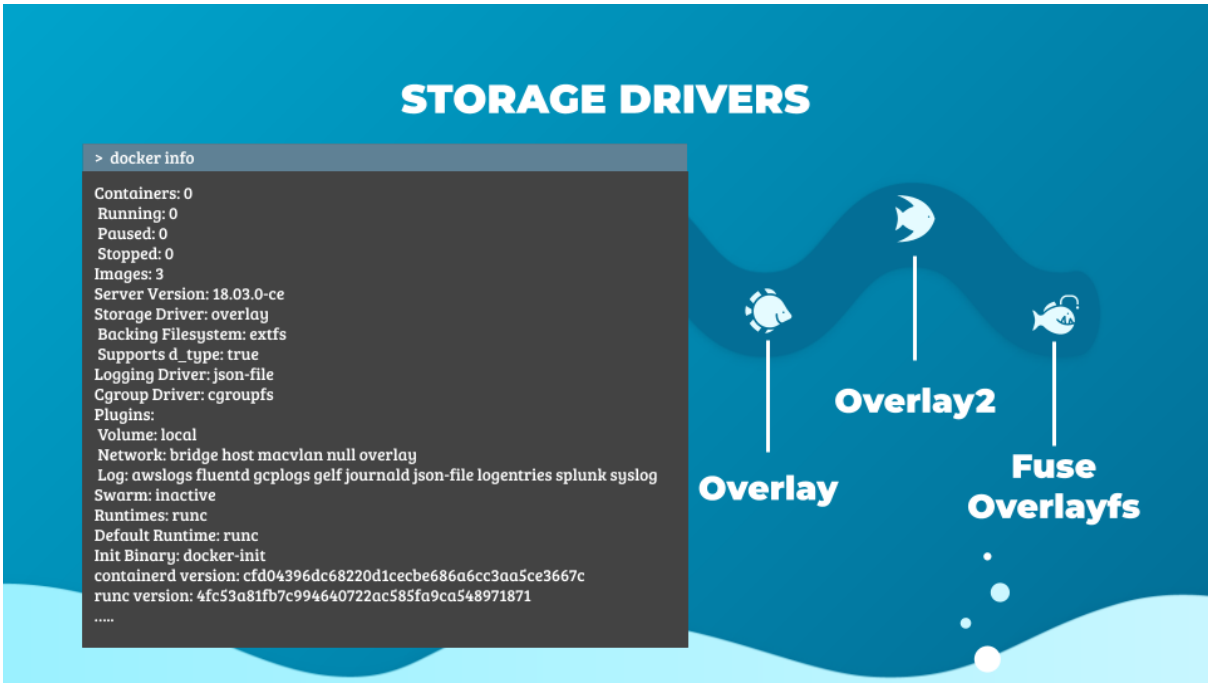
Ок, а кто несет ответственность за выполнение всех этих операций. Сохранение многоуровневой архитектуры. Создание слоя с возможностью записи, перемещение файлов между слоями для возможности копирования и записи и т. д.

Это storage drivers. Таким образом, Docker использует драйверы хранилища для обеспечения многоуровневой архитектуры.

Некоторые из распространенных драйверов хранения: AUFS, BTRFS, ZFS, device-mapper, overlay и overlay 2, fuse overlayfs. Некоторые из них широко используются, некоторые уже устарели, но могут встретиться в каких-то легаси нагрузках.

Выбор драйвера хранилища зависит от используемой ОС.

Например, у Ubuntu.



Драйвер хранилища по умолчанию для нее - это overlay2. Но до 18 версии Docker это был aufs, и он был недоступен в таких операционных системах, таких как Fedora или Cent OS, и в них приходилось использовать device-mapper.

Docker сам постарается выбрать лучший драйвер хранилища, который доступен в конкретной операционной системе. Но это не всегда работает хорошо. В случае проблем нужно настраивать руками, изучив соответствующие issues по конкретной ОС и ее ядру. Где-то до 18 года это было серьезной проблемой, с появлением overlay2 дела стали обстоять гораздо лучше.

Различные драйверы хранилища также обеспечивают разные характеристики производительности и стабильности, поэтому ты можешь выбрать тот, который соответствует потребностям твоего приложения и соответствии требования твоей организации.
Если хочешь быть осведомленным о каком-либо из этих драйверов хранения, начни с документации Docker, а далее погрузись в десятки issues на гитхаб докера.

Это все из концепций хранения в Docker.

Увидимся на следующей лекции.

Привет и добро пожаловать.

Итак, в предыдущей лекции мы обсуждали драйверы хранилища Эти storage drivers помогают управлять хранилищем в образах и контейнерах.

В предыдущей лекции мы также кратко коснулись томов. Мы узнали, что если нам нужно сохранить данные в хранилище, то нужно создать тома.

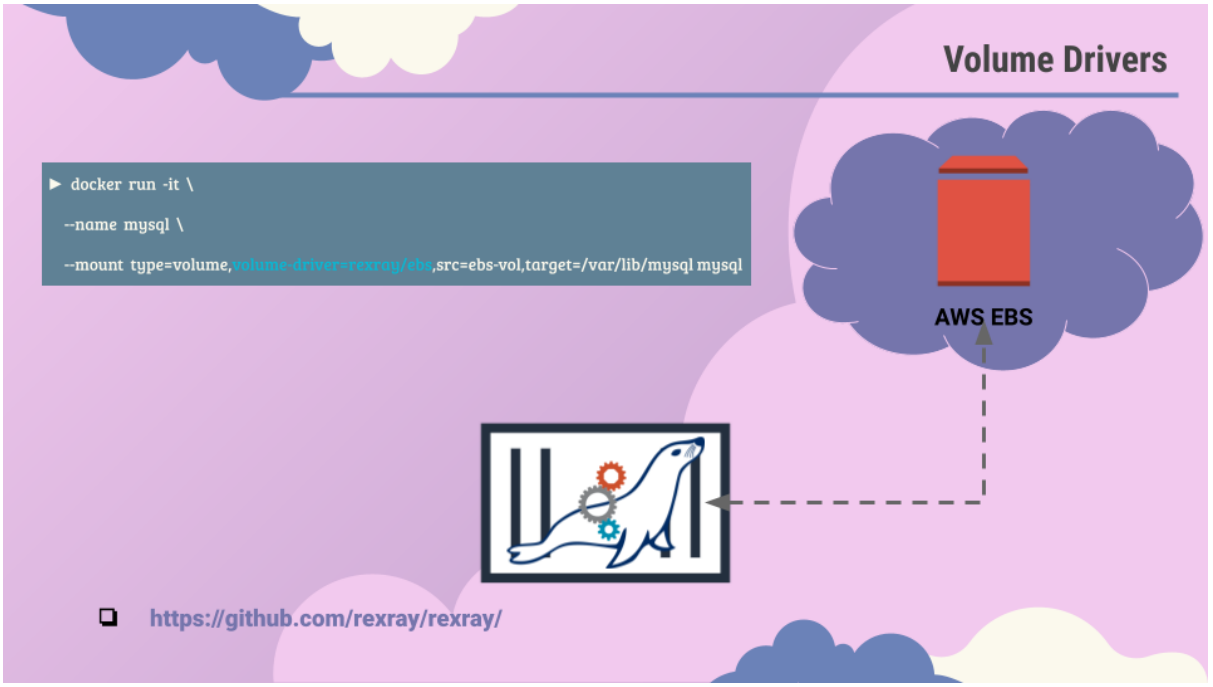
Помни, что тома обрабатываются не драйверами хранилища. Их обрабатывают volume driver plugins. Такой volume driver plugin по умолчанию это local.

Плагин локального тома помогает создать volume на докер-хосте и хранить его данные в каталоге /var/lib/docker/volumes.

Существует множество других volume driver plugins, которые позволяют создавать тома на сторонних решениях, таких как Azure File Storage, Convoy, DigitalOcean Block Storage, Flocker, gce persistent disk, GlusterFS, RexRay, Portworx, vSphere Storage, NetApp.

Это лишь некоторые из многих, всех сложно запомнить, это постоянно развивается.

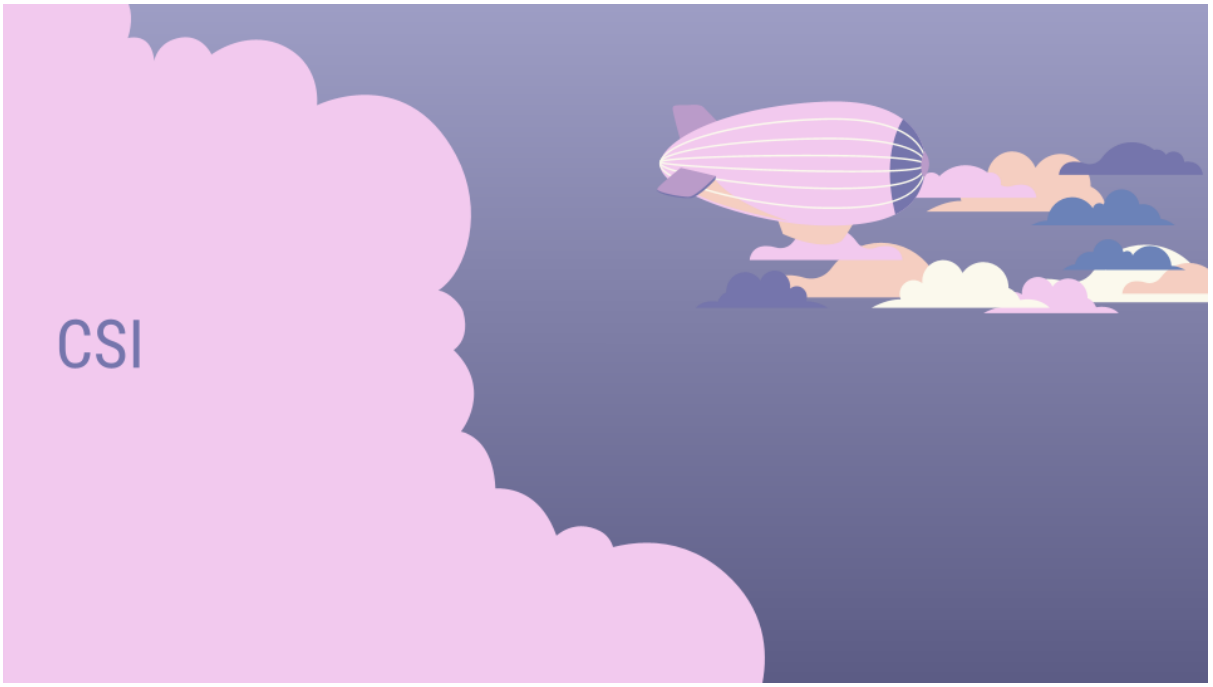
Некоторые из этих драйверов томов поддерживают различных поставщиков хранилищ, например, драйвер хранилища Rexray можно использовать для предоставления хранилища на массивах хранения от AWS EBS, EFS, S3FS, от Dell EMC, таких как Isilon и ScaleIO, от Google Persistent Disk или Cinder от Open Stack.



Когда мы запускаем докер-контейнер, мы можем использовать определенный драйвер тома, например rexray/ebs, для предоставления контейнеру тома из Amazon EBS.

Это создаст контейнер и присоединит том из облака, а когда контейнер выйдет, все наши данные будут в безопасности в облаке.

В следующих лекциях мы узнаем больше о томах в Kubernetes. Жду тебя на них.



Привет и добро пожаловать.
Давай теперь посмотрим на интерфейс контейнерного хранилища - container storage interface или короче CSI.

В прошлом Kubernetes использовал Docker в качестве единственного движка среды выполнения контейнеров. В то время весь код для работы с Docker был встроен в исходный код Kubernetes. Со временем появилась необходимость добавить возможность работать с другими движками, такими как cri-o и rkt.
Было важно открыть и расширить поддержку работы с различными container runtimes и не зависеть от исходного кода Kubernetes.

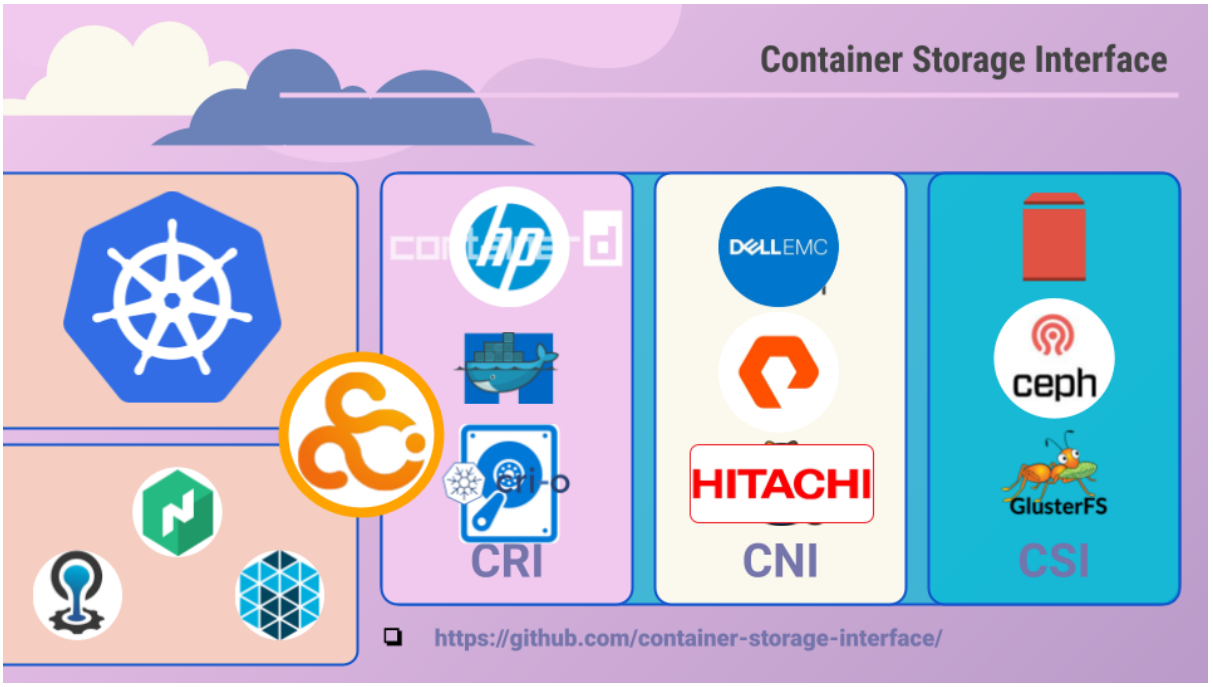
Так появился container runtime interface, CRI или интерфейс среды выполнения контейнера.

Container runtime interface - это стандарт, который определяет, как решение для оркестрации, вроде Kubernetes, будет взаимодействовать со средами выполнения контейнера, такими как Docker.

Таким образом, в будущем, если будет разработан какой-либо новый container runtime, он может просто следовать стандартам CRI.

И эта новая среда выполнения контейнера будет работать без необходимости разработки и добавления в себя какого-то дополнительного функционала под Kubernetes или добавления чего-то к исходному коду Kubernetes.

В данный момент официально поддерживаются 3 движка: Docker, Cri-o и Containerd. А с версии 1.23 дефолтным runtime станет containerd.



Точно так же, для расширения поддержки различных сетевых решений был представлен сетевой интерфейс контейнера - CNI.

Теперь любые новые поставщики сетевых услуг могут просто разработать свой плагин на основе стандартов CNI и это их решение будет работать с Kubernetes.

Как ты можешь догадаться, интерфейс контейнерного хранилища был разработан для поддержки нескольких решений хранения. С CSI теперь мы можем написать свои собственные драйверы для собственного хранилища для работы с Kubernetes.

AWS EBS, Ceph, GlusterFS, Dell EMC, Portworx, Hitachi, HP, NetApp, Azure Disk и др.

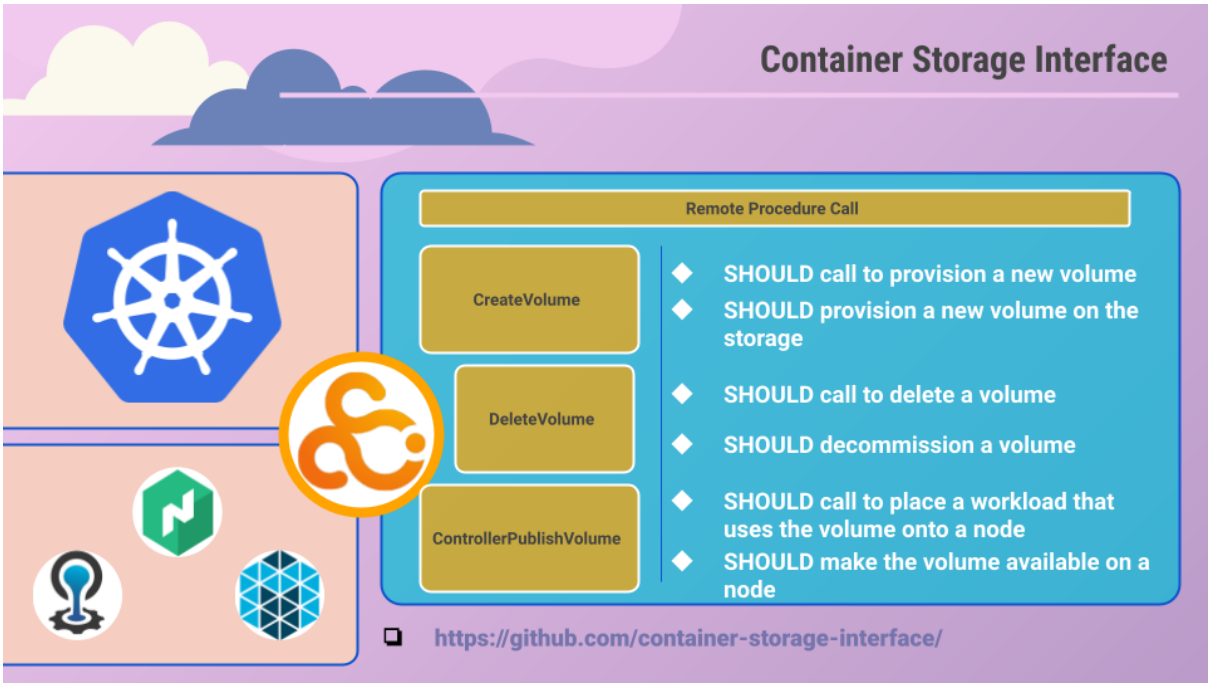
У каждого свои драйверы CSI.

Обрати внимание, что CSI не является специфическим стандартом Kubernetes. Это универсальный стандарт и он позволяет любому инструменту оркестрации контейнеров работать с любым поставщиком хранилища с поддерживаемым плагином.

В настоящее время полностью приняли 4 оркестратора:

- Kubernetes
- Cloud Foundry
- Nomad
- Mesos

Итак, вот как выглядит CSI: он определяет набор RPC или удаленных вызовов процедур, которые будут инициироваться оркестратором контейнеров, и они должны быть реализованы в storage drivers.



Например, CSI говорит, что когда создается POD, и ему требуется volume, то оркестратор контейнеров в данном случае, Kubernetes, должен вызвать RPC `CreateVolume` и передать в него набор деталей, таких как имя этого volume.

Storage driver должен реализовывать этот удаленный вызов и обработать запрос, предостав новый том в области хранения и вернуть результат операции.

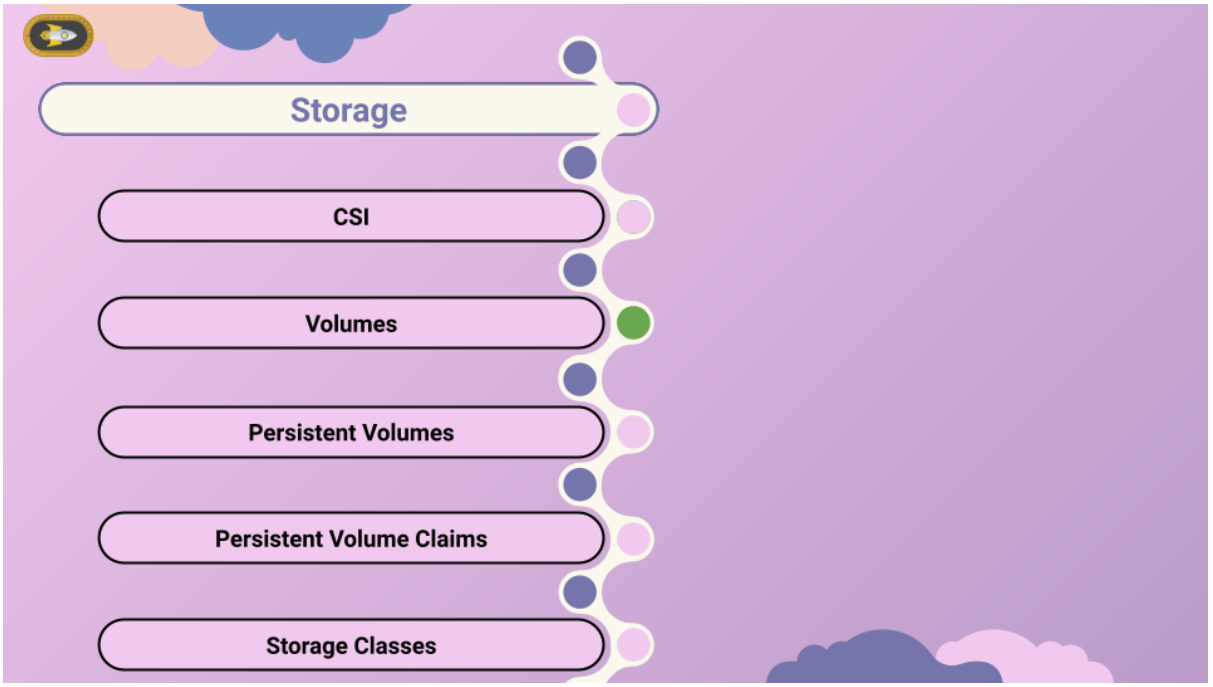
Точно так же оркестратор контейнеров должен осуществить вызов RPC `DeleteVolume`, когда volume должен быть удален, а драйвер хранилища должен содержать в себе реализацию действия вывода тома из массива хранения при выполнении этого вызова.

В спецификации CSI точно указано, какие параметры должны быть отправлены оркестратором, что именно может получать драйвер и какие коды ошибок они должны предоставлять.

Если тебе интересно, ты можешь просмотреть все эти детали в спецификации CSI на GitHub, ссылку я привожу.

На этом пока все об интерфейсе контейнерного хранилища.

Увидимся на следующей лекции.

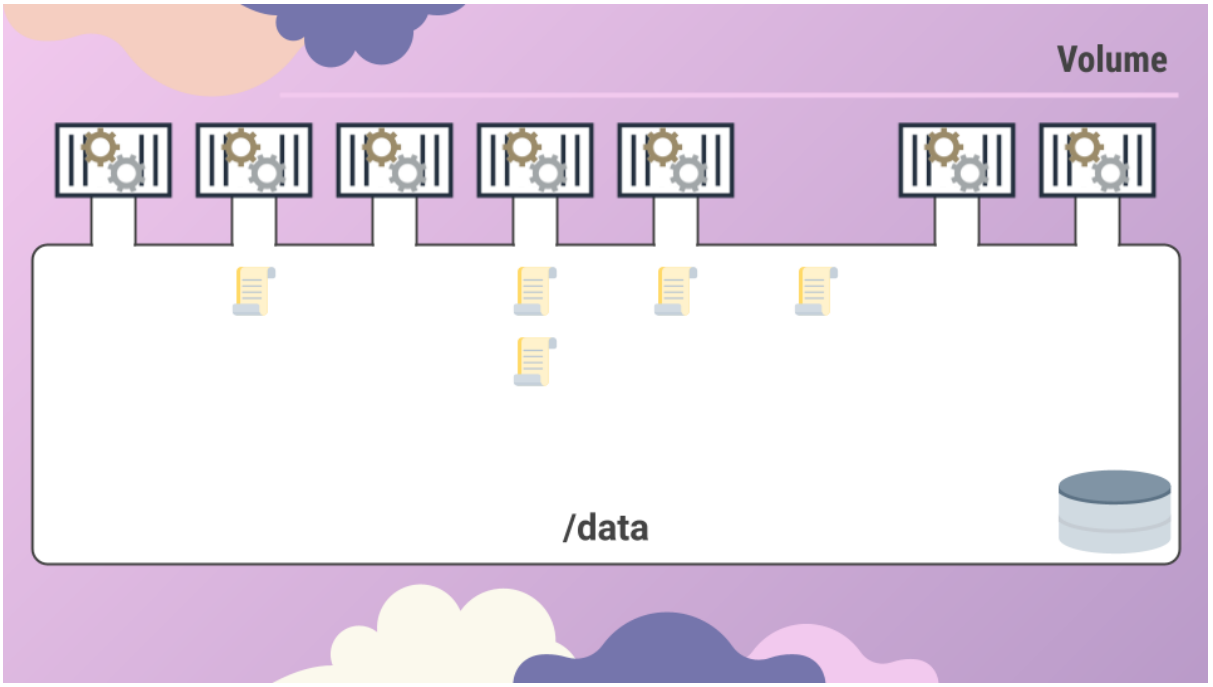


Привет и добро пожаловать на лекцию о томах.

Прежде чем мы перейдем к постоянным томам - persistence volumes, давай начнем с томов в Kubernetes.

Сначала посмотрим на тома в Docker. Контейнеры Docker должны быть временными по своей природе, что означает, что они предназначены для работы только в течение небольшого периода времени. Они вызываются, когда требуется обработать данные, и уничтожаются после завершения.

То же самое верно и для данных в контейнере: данные уничтожаются вместе с контейнером. Для сохранения данных, обрабатываемых контейнерами, мы присоединяем тома к контейнерам при их создании.



Эти данные, обрабатываемые контейнером, теперь помещаются в volume, тем самым сохраняясь навсегда. Когда контейнер будет удален, сгенерированные или обработанные им данные останутся.

Итак, как это работает в мире Kubernetes.

Как и в Docker, POD, созданный в Kubernetes, временный по своей природе.

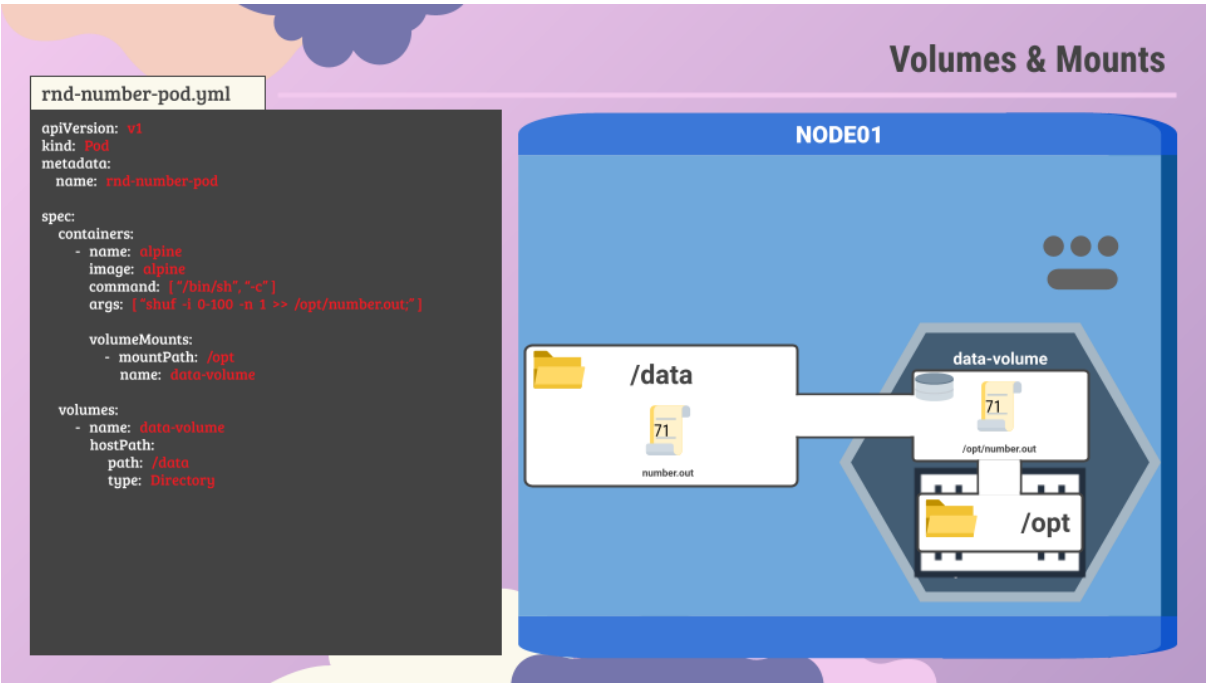
Когда создается POD для обработки данных, а затем удаляется, обрабатываемые им данные также удаляются.
Для сохранения мы прикрепляем volume к POD. Данные, сгенерированные POD, теперь хранятся в томе, и даже после удаления POD данные остаются. Давай посмотрим на простую реализацию volumes.

У нас есть кластер Kubernetes с одним узлом.

Мы создаем простой POD, который генерирует случайное число от 1 до 100 и записывает его в файл по адресу /opt/number.out, на этом его работа заканчивается и POD удаляется. Разумеется он удалится вместе с этим случайным числом. Чтобы сохранить номер, сгенерированный POD мы создаем volume, и этому volume требуется хранилище.

При создании тома мы можем настроить его хранилище разными способами.
Мы немного рассмотрим различные параметры, а пока просто настроим его для использования каталога на хосте.

В этом случае я указываю путь /data в поле hostPath.



Таким образом, любые файлы, созданные в этом volume, будут храниться каталоге /data на этом узле. После создания volume для доступа к нему из контейнера мы монтируем volume в каталог внутри контейнера.

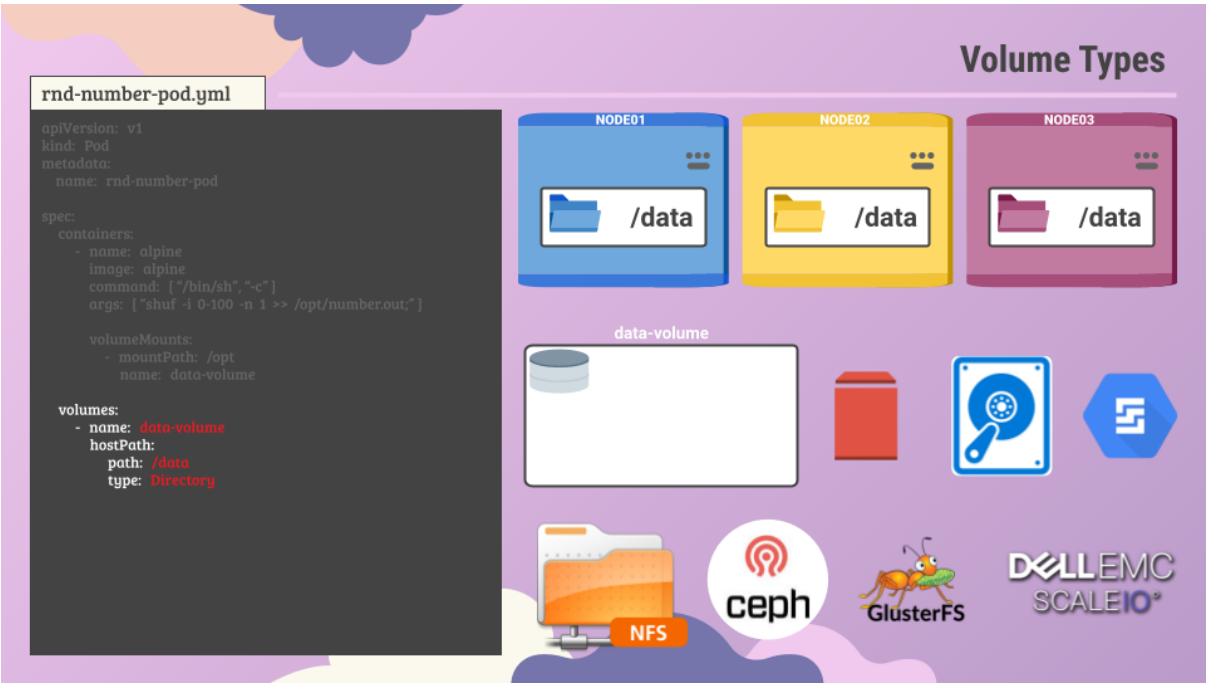
Мы используем поле volumeMounts в каждом контейнере для монтирования тома /data в каталог /opt внутри контейнера.

Теперь случайное число будет записано в директорию /opt, смонтированную внутри контейнера. И этот каталог связан через том data-volume с каталогом /data на хосте. Т.е. эта папка /opt контейнера фактически является папкой /data файловой системы ноды.

Когда POD удаляется, файл со случайным номером остается в папке на хосте. Ведь вместе с POD удалилась связь между каталогом /opt и /data, но сам каталог /data никто не удалял.

Давай сделаем шаг назад и посмотрим на варианты хранения томов.

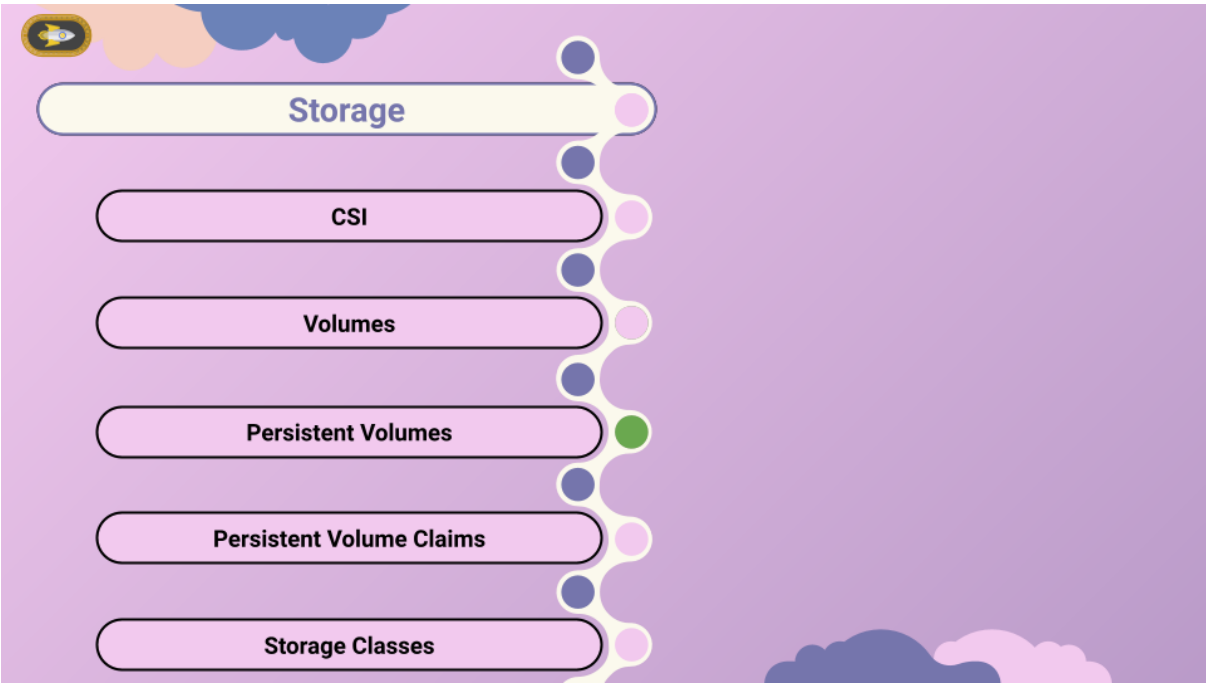
Мы просто использовали параметр hostPath для настройки каталога, и этим выделили место на хосте для размещения тома.
И это отлично работает на одном узле. Но что, если кластер многоузловой? Такой подход будет не рекомендован.
Это связано с тем, что PODs будут использовать каталог /data на всех нодах, но будут полагать, что это один и тот же каталог. Следовательно и ожидать, что все данные положенные туда сохранятся. Но это не так, будет мозаика из файлов, созданных отработавшими PODs, и разбросанная по папкам /data всех нодах кластера.
Потому, что эти volumes создавались на каждой ноде, где запускался POD.



Это можно изменить, используя какое-нибудь решение для внешнего кластерного хранилища с репликацией.
Kubernetes поддерживает несколько типов стандартных решений для хранения, таких как NFS, CephFS, glusterFS, ScaleIO, Flocker, FibreChannel, CephFS или облачные решения, такие как AWS EBS, Azure Disk или File или Google Persistent Disk.

Например, чтобы настроить том AWS Elastic Block Store в качестве хранилища или тома, мы заменяем в volumes поле hostPath на awsElasticBlockStore, туда добавляем парамерт volumeID и fsType. Теперь этот том будет вести в хранилище, находящееся в AWS EBS. Разумеется для нормальной работы нужно будет настроить доступ.

Ну вот и все о volumes в Kubernetes. Теперь мы готовы перейти к обсуждению постоянных томов.



Привет и добро пожаловать на эту лекцию о постоянных томах - persistent volumes.

На последней лекции мы говорили о томах.

Теперь поговорим о Persistent Volumes в Kubernetes. Когда мы создавали volumes в предыдущем разделе, мы настраивали volumes в файле определения POD.

Это значит, что вся информация о конфигурации, необходимая для настройки хранилища для тома помещается в файл определения POD.

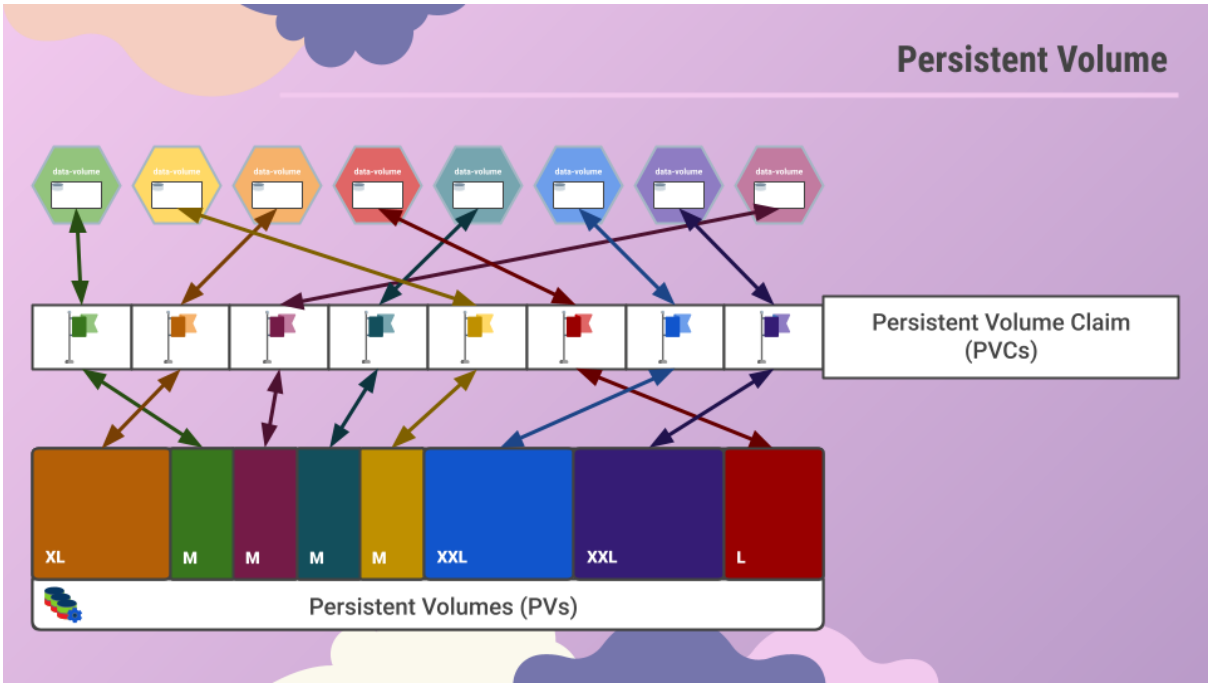
В распределенной среде с большим количеством пользователей, для них будет развернуто множество PODs. Пользователям каждый раз придется настраивать хранилище для каждого POD.

Какое бы решение для хранения ни использовалось, пользователи, развертывающие PODs, должны будут прописывать эти настройки в каждый свой файл определений.

И при каждом внесении изменений в свою среду пользователю придется снова внести изменения в файлы определений своих приложений.

Это не кажется легким делом. Вместо этого мы бы хотели управлять хранилищем более централизованно. Например, чтобы оно было настроено так, чтобы администратор мог создать большой пул хранилищ, а затем порезать на кусочки того объема, о котором попросят пользователи.

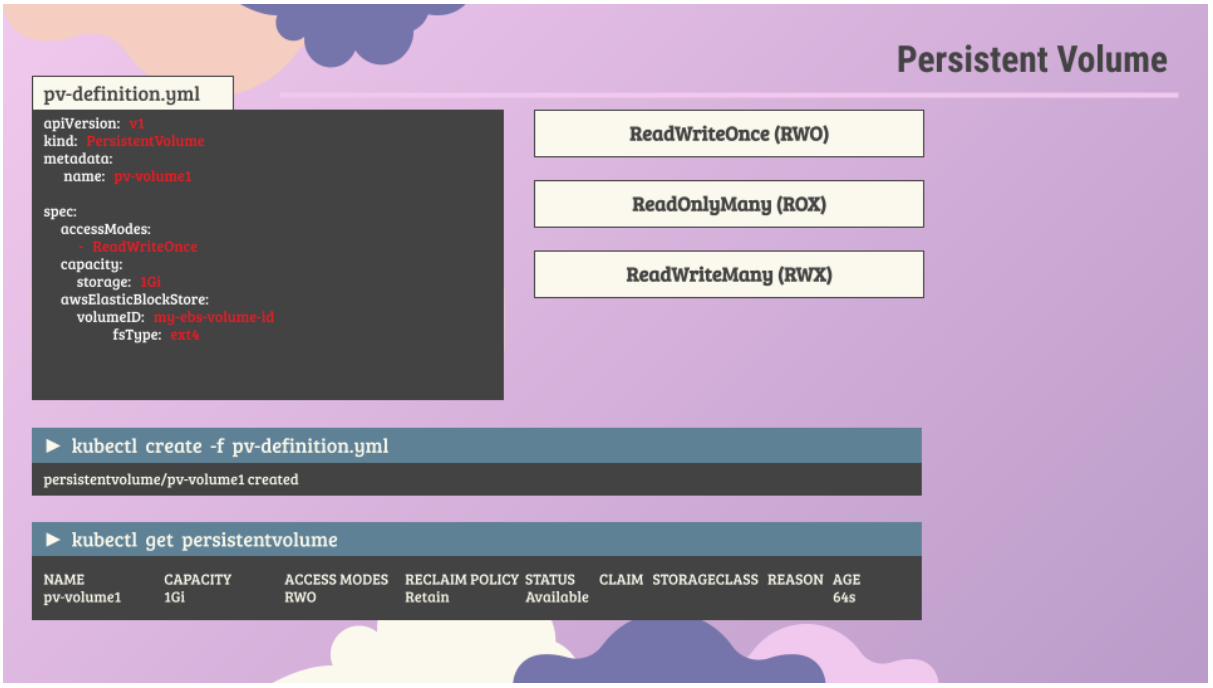
И вот в этом нам могут помогут persistent volumes или короче PV.



Постоянный том - это пул томов хранения - storage volumes на уровне кластера, настроенный администратором для использования пользователями, развертывающими приложения в кластере.

Теперь пользователи могут выбирать хранилище из этого пула с помощью persistent volume claims или PVC.

Давай теперь создадим persistent volume.



Мы начнем с базового шаблона объекта Kubernetes:

apiVersion, kind, metadata, spec.

Версия API - v1, kind - PersistentVolume, в метаданных определим name - pv-volume1.
В разделе спецификаций укажем режимы доступа.

Этот accessModes определяет, как volume должен быть работать на хостах.
Это похоже, как в операционной системе монтируют устройство в режим только для чтения или в режим чтения-записи.
Поддерживаемые значения для accessModes:

- ReadWriteOnce доступны только для чтения.
- ReadOnlyMany доступны для записи одному
- ReadWriteMany доступны для записи многим

Далее в разделе capacity указываем объем хранилища, который должен будет зарезервирован для этого persistent volume. Здесь установлен в 1 Гибибайт.

Далее идет тип тома.

Здесь идет параметр hostPath, который использует хранилище из локального каталога этой ноды.

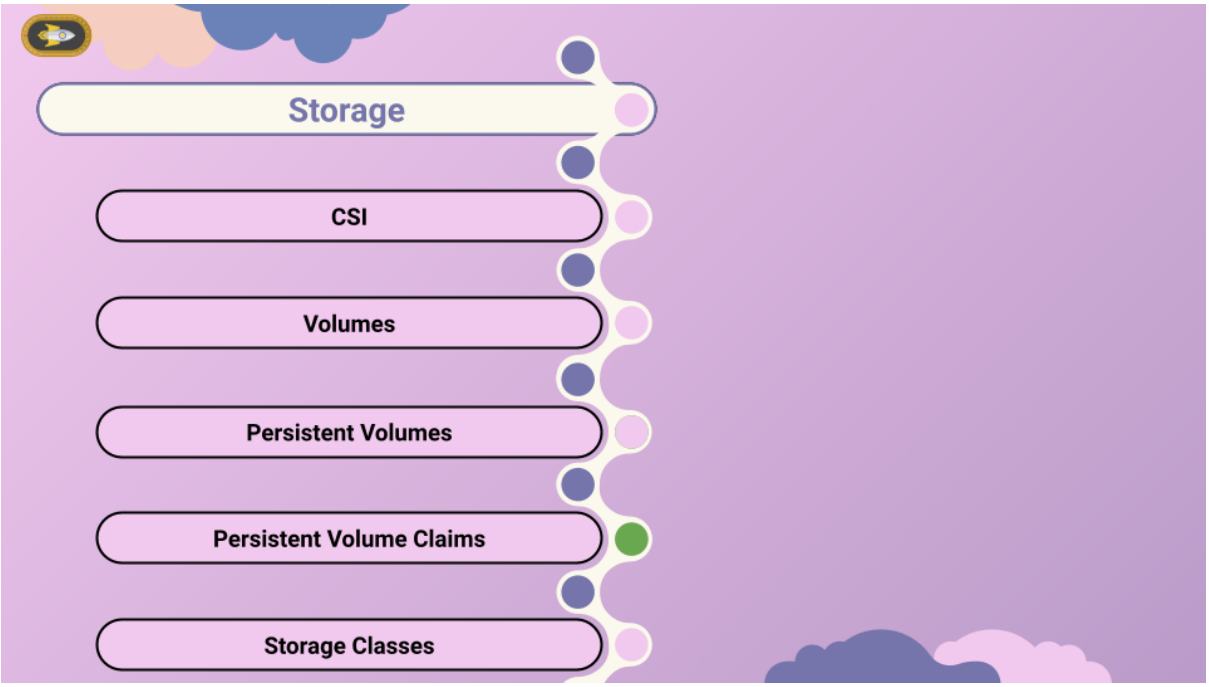
Помни, что эту опцию не стоит использовать в производственной среде, локальное хранилище ноды не самое надежное место.

Чтобы создать persistent volume, выполним команду kubectrl create.

Выведем список созданных томов командой kubectrl get persistentvolume.

Теперь заменим hostPath одним из поддерживаемых решений хранения, как мы видели в предыдущей лекции, а именно, AWS elastic blocks store.

Это все о persistent volumes в этой лекции. В следующей мы рассмотрим, как эти нарезанные кусочки будут раздаваться пользователю с помощью persistent volume claims. Увидимся!

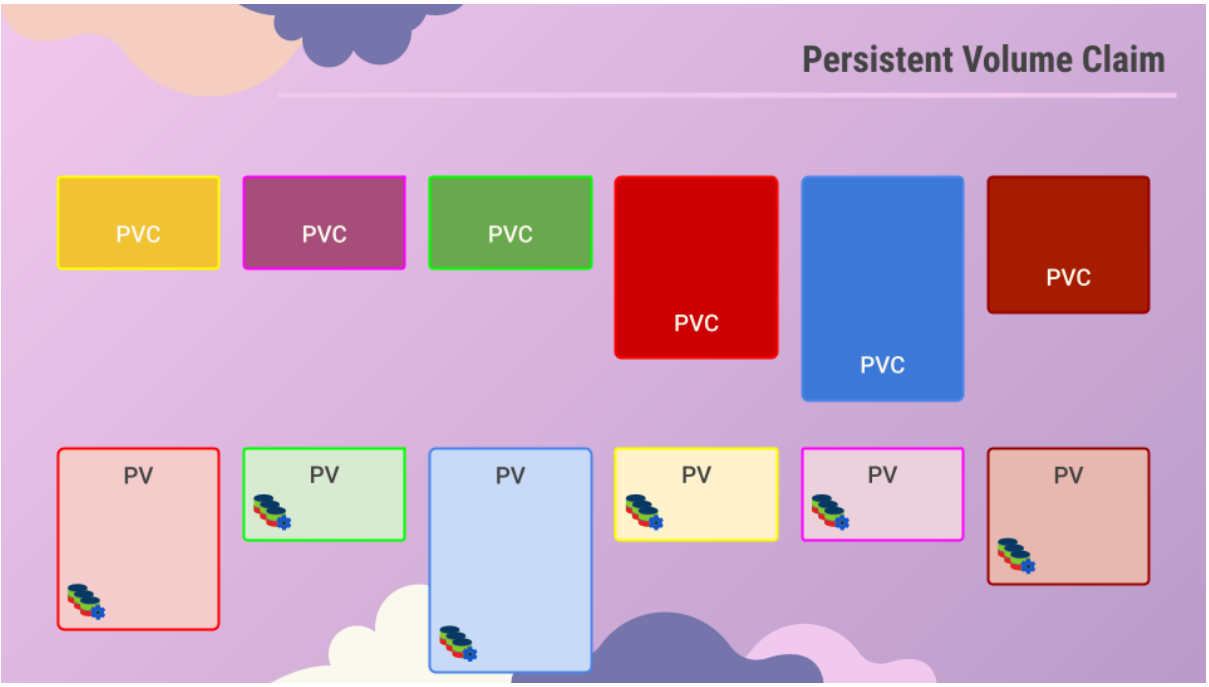


Привет и добро пожаловать на лекцию о Persistent Volume Claims в Kubernetes.

В предыдущей лекции мы создали Persistent Volume.
Теперь создадим Persistent Volume Claim, чтобы сделать хранилище доступным для ноды.

PV и PVC - это два отдельных объекта внутри namespace.

Администратор создает набор постоянных томов, а пользователь создает заявку на постоянный том, т.е. Persistent Volume Claim для пользования хранилищем.

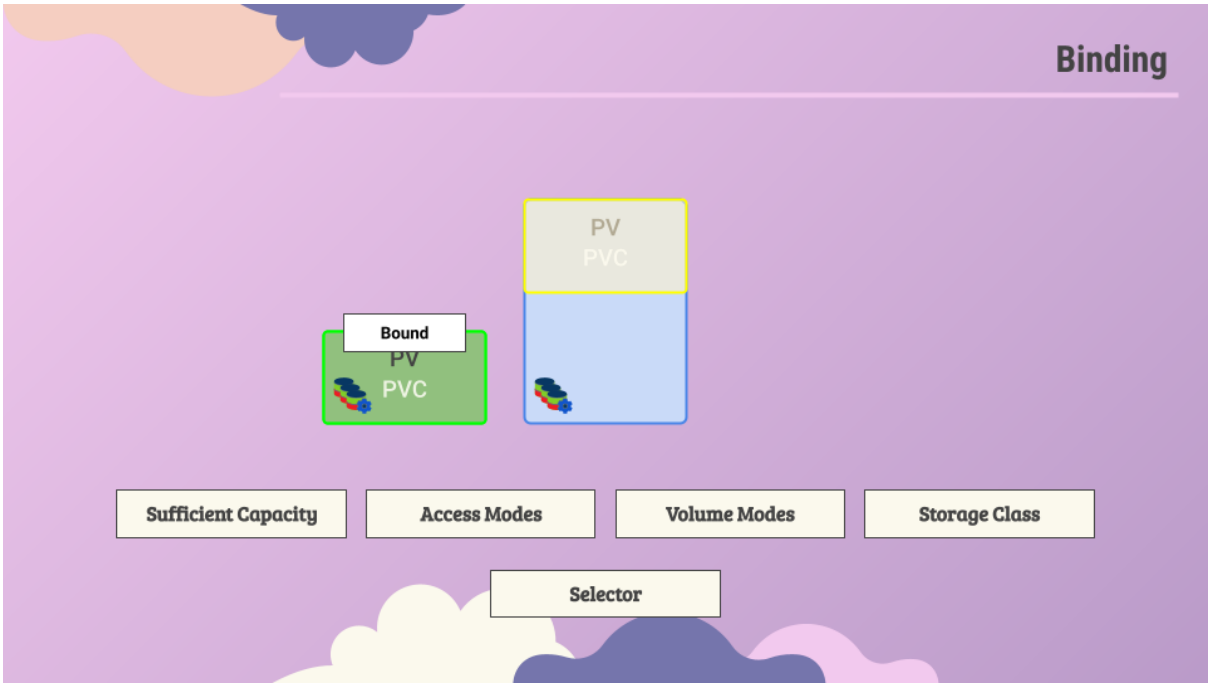


После создания PVC Kubernetes привязывает PV к этим заявкам на основе требований запроса и свойств тома. Каждая Persistent Volume Claim привязывается к одному из наиболее подходящих Persistent Volumes. Во время процесса привязки Kubernetes пытается найти постоянный том с достаточной емкостью в соответствии с запросом и любыми другими свойствами Persistent Volume, такими как режим доступа, режим тома, класс хранения и т. д.

Однако, если вдруг наличествуют несколько возможных совпадений для Persistent Volume Claim, а ты бы хотел связать его с конкретным Persistent Volume, то можешь просто использовать метки и селекторы для привязки PVC к нужным PV.

Наконец, обрати внимание, что меньшая запросу на емкость PVC может быть связана с PV бОльшей емкости, если другие критерии соответствуют и нет лучших вариантов.

Между заявками и томами существует взаимно однозначная связь, поэтому никакие другие PVC не могут использовать оставшуюся емкость PV.



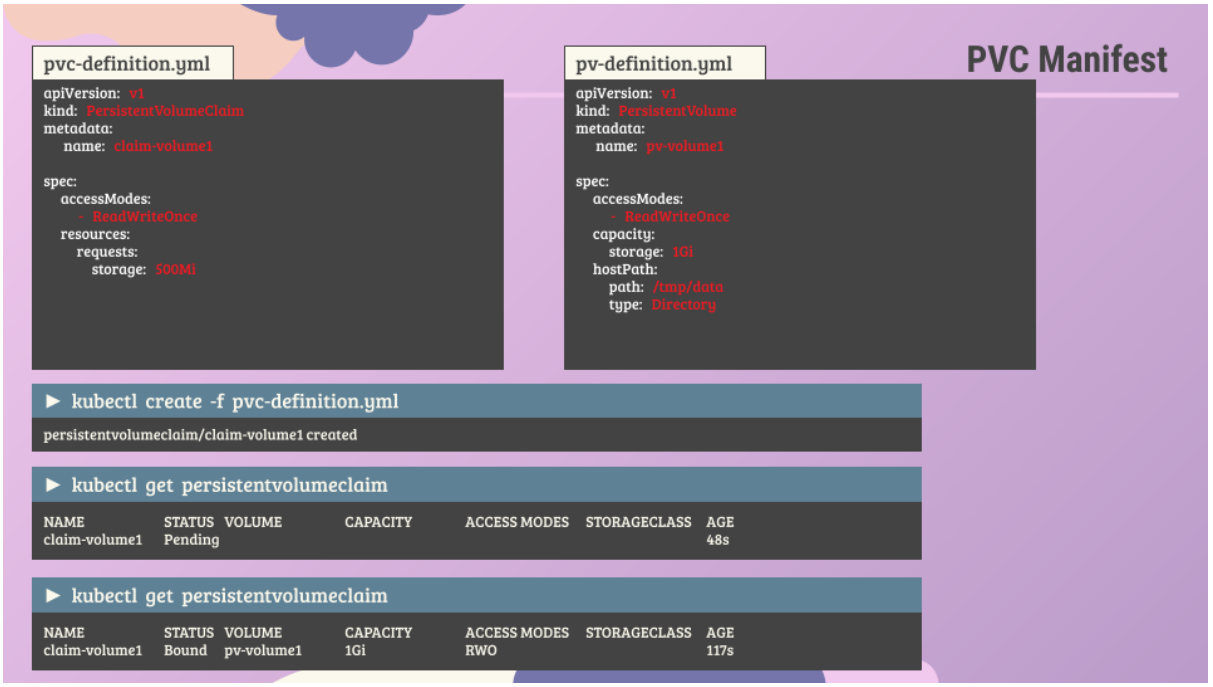
Если доступных томов нет, Persistent Volume Claim будет оставаться в состоянии ожидания - pending - до тех пор, пока новые PV не станут доступными для кластера.

Как только новые тома появятся или освободятся старые заявка будет автоматически привязана - bound - к новому доступному тому.

Теперь давай создадим Persistent Volume Claim.

Мы начинаем с пустого шаблона, в котором указаны apiVersion, kind, metadata и spec.

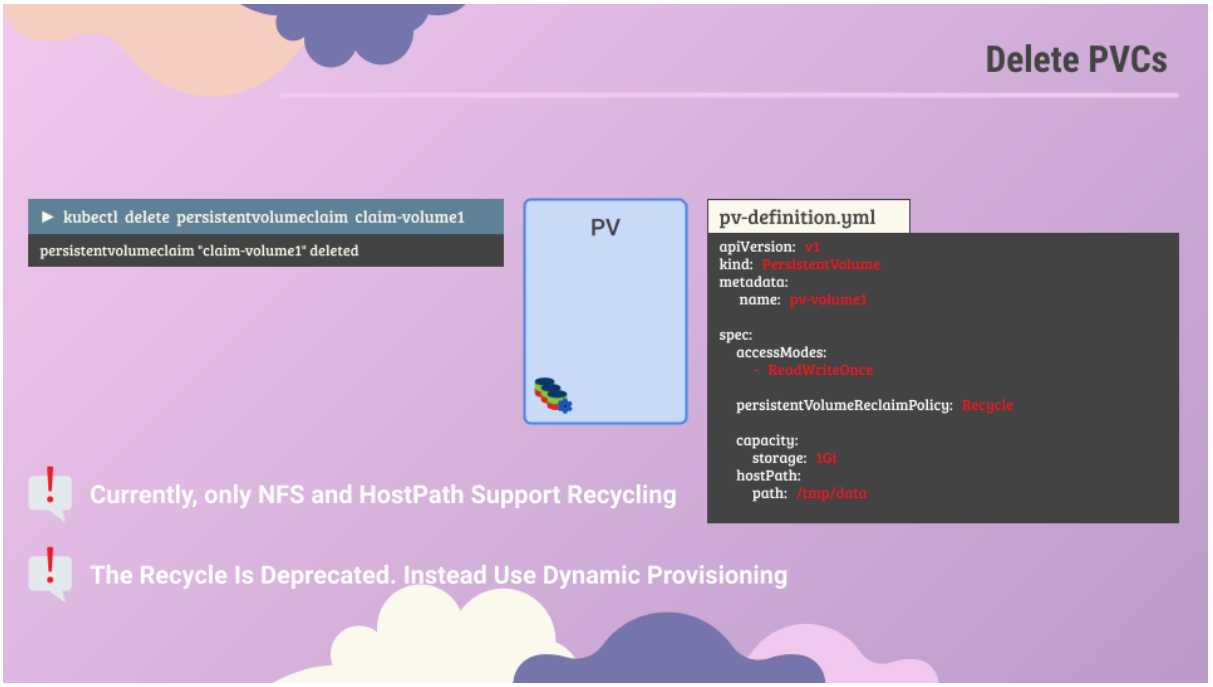
apiVersion - v1, kind - PersistentVolumeClaim.
Назовем ее claim-volume1.



В спецификации установим для параметра accessModes значение ReadWriteOnce. Установим в поле resources параметр requests для запроса хранилища в 500 мебибайт.

Создадим claim с помощью команды kubectl create. Чтобы просмотреть созданную PVC, запусти команду kubectl get persistentvolumeclaim.
Мы видим claim в состоянии ожидания.

При создании PVC Kubernetes просматривает ранее созданный volume. Режимы доступа совпадают. Запрашиваемая емкость составляет 500 мебибайт, но хранилища настроен на 1 гибибайт.
Поскольку других томов нет в наличии, Persistent Volume Claim будет привязана к этому PV. Когда мы снова запускаем команду get persistentvolumeclaim мы увидим, что PVC привязана к PV, который мы создали.



Все получилось.
Чтобы удалить PVC, запусти команду kubectl delete persistentvolumeclaim
Но что произойдет с постоянным томом, который занимала до удаления PVC?

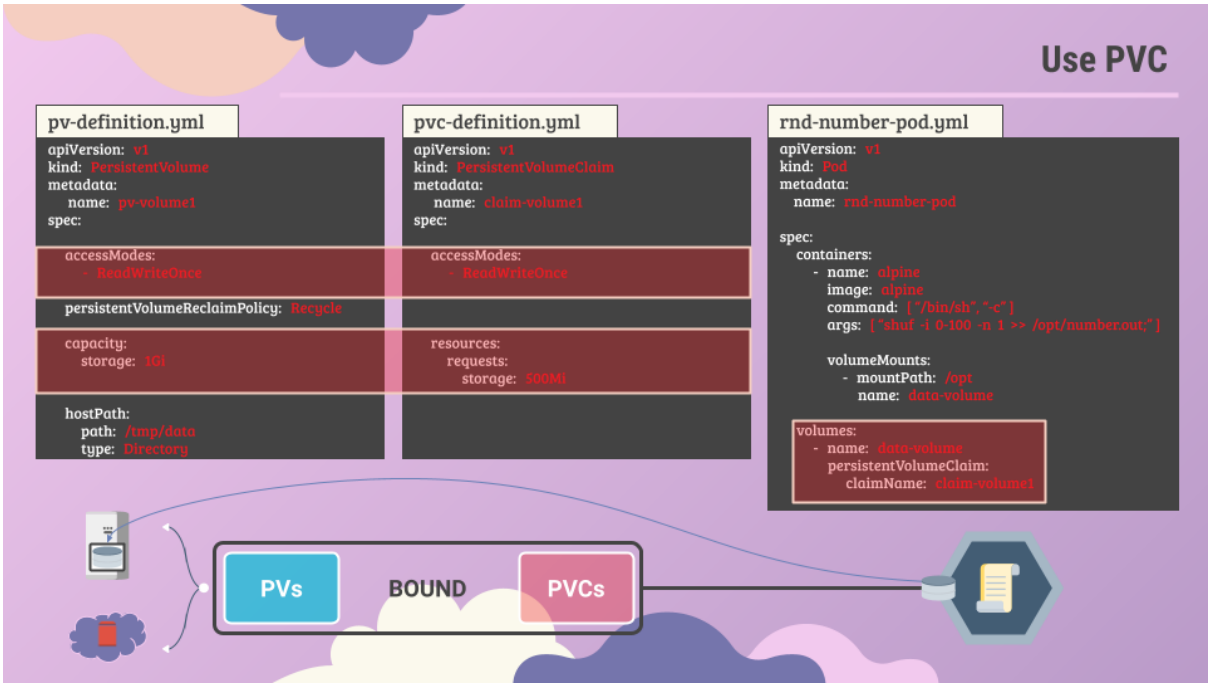
Ты можешь установить поведение PV для таких случаев с помощью параметра `persistentVolumeReclaimPolicy`
Если его значение Retain означает PV будет оставаться до тех пор, пока администратор не удалит его вручную и не пересозаст заново.
Сам постоянный том в это время будет недоступен для повторного использования другими PVC.

Значение `Delete` автоматически удаляет PV и данные после отцепления PVC.
Таким образом, как только PVC будет удален, PV тоже будет удален, что освободит место на конечном устройстве хранения.

Третий вариант - `Recycle`.

В этом случае данные в томе будут очищены, потом том станет доступным для других Persistent Volume Claims.

Политика Recycle считается deprecated, ей на смену пришло dynamic provisioning с помощью StorageClasses.



Итак, как это работает.

В нашем POD смонтирован внутрь каталог, в котором подразумевается постоянное хранение.

У нас есть некоторое количество аппаратных мощностей для хранения. Эти мощности мы делим на кусочки исходя из размера, месторасположения и типа. Для этого создаем набор PV, указывая в них эти параметры.

После этого создаем PVC. Она ищет подходящий PV по таким параметрам: accessMode, volumeMode, resources, labels, storageClass. Также можно явно связать PV и PVC по имени, чтобы именно эти две стали связанными.

Итак, половинки нашли друг друга и стали связанными - bound.

После создания PVC используем ее в файле определения POD, указав имя PVC в разделе persistentVolumeClaim раздела volumes следующим образом.

Мы задекларировали том с именем data-volume с указанием ссылки на PVC. Теперь создадим POD.

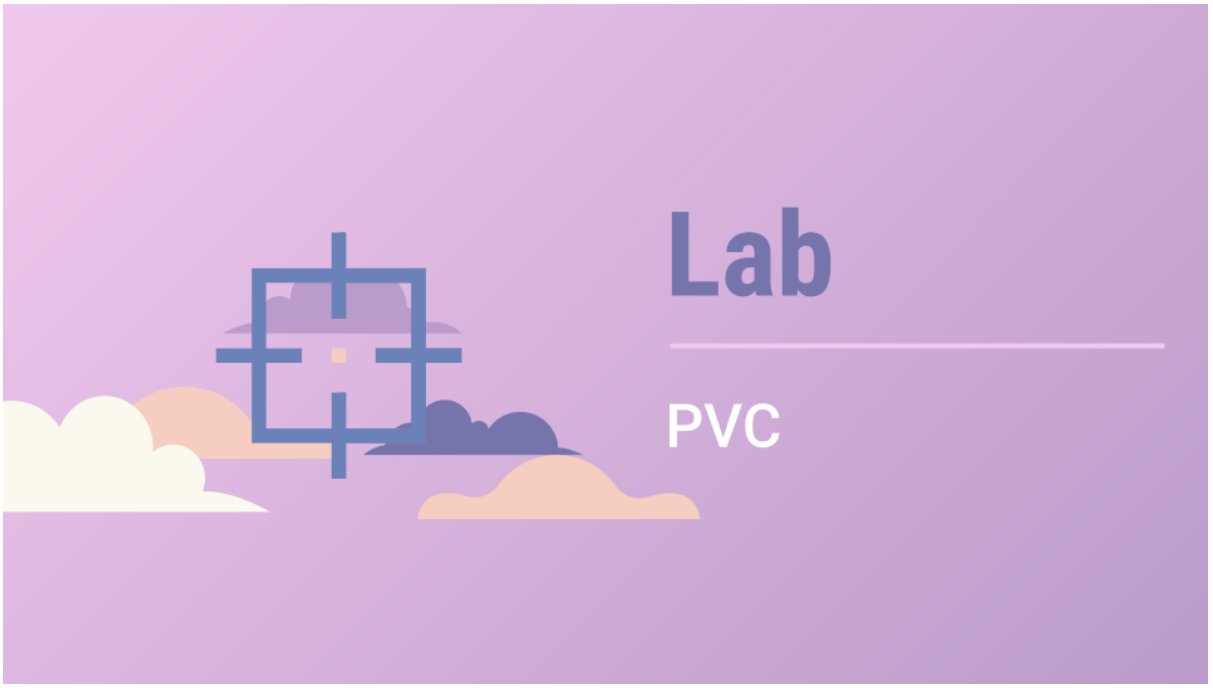
Приложение из POD будет писать данные в каталог /opt, который фактически будет размещаться там, где мы описали это в pv-volume1. А чтобы POD нашел этот кусочек хранения, том внутри POD связан с PVC с именем claim-volume1.

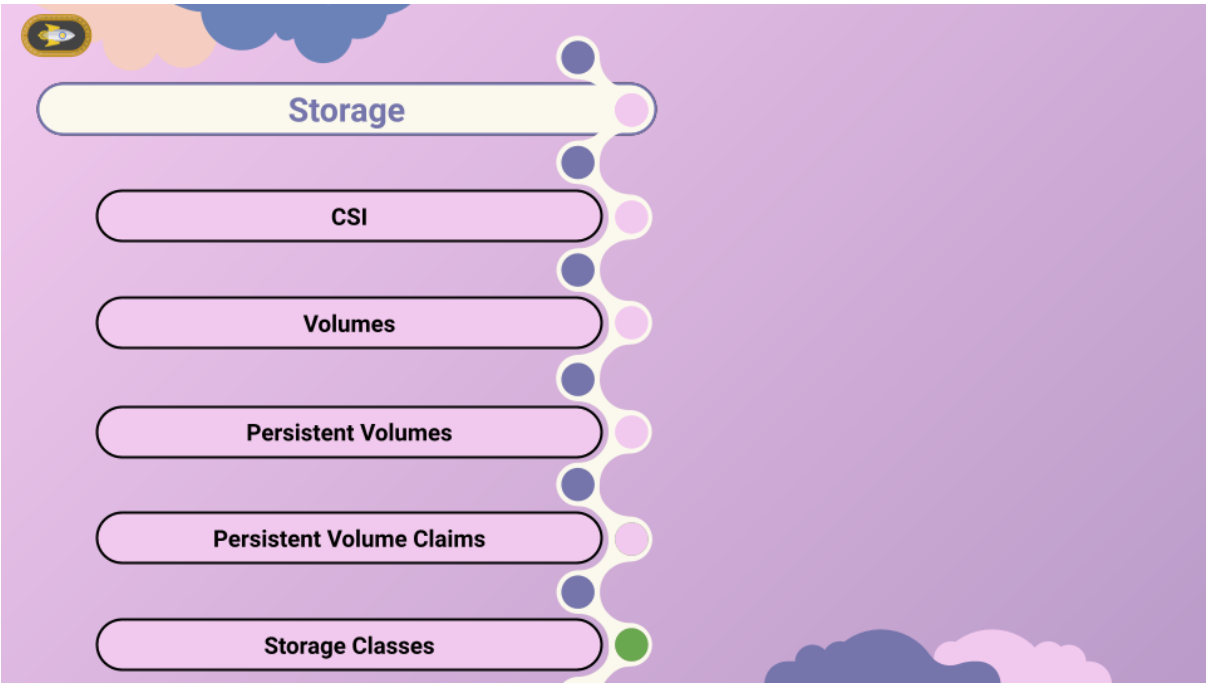
То же самое верно для ReplicaSets или Deployments. Добавь это в раздел template для POD в Deployment или ReplicaSet.

На этом лекция закончена.

Перейди в раздел практики, попробуй себя в настройке и устранении неполадок PV и PVC в Kubernetes.

Жду тебя на следующей лекции!





Привет и добро пожаловать.

В этой лекции мы рассмотрим классы хранения - Storage Classes.

В предыдущих лекциях мы обсуждали, как создать PV, а затем создать PVC, чтобы чтобы разместиться в этом хранилище, а затем использовать PVC в файлах определений PODs в качестве volumes.

В этом случае мы создаем PVC для подключения хранилища Google Persistent Disk. Она будет цепляться за этот PV.
Но проблема здесь в том, что перед его созданием мы должны будем создать хранилище в Google Cloud.

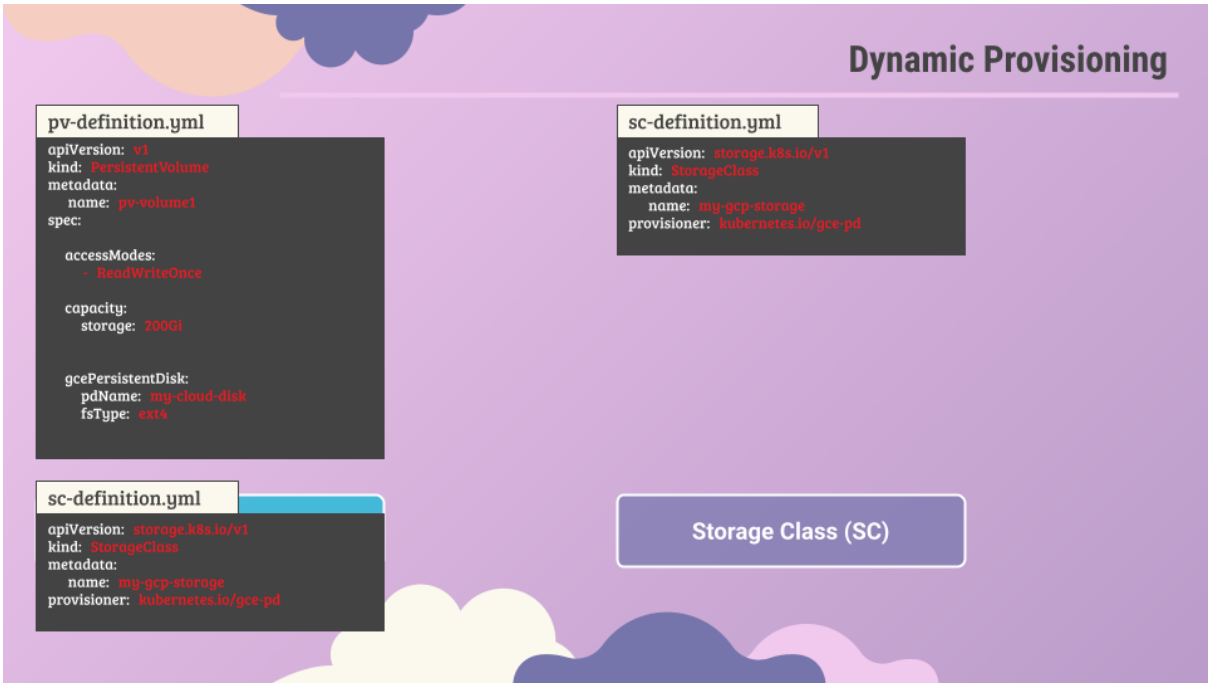
И каждый раз, когда приложению потребуется хранилище, нам придется сначала вручную подготовить disk в gcloud compute engine, а затем вручную создать файл определения PersistentVolume с указанием имени этого созданного диска.

Это называется томами со статической подготовкой - Static Provisioning of Volumes.

Было бы неплохо, если бы volume был подготовлен автоматически, когда это требуется приложению.

Вот именно здесь вступают в игру Storage Classes.

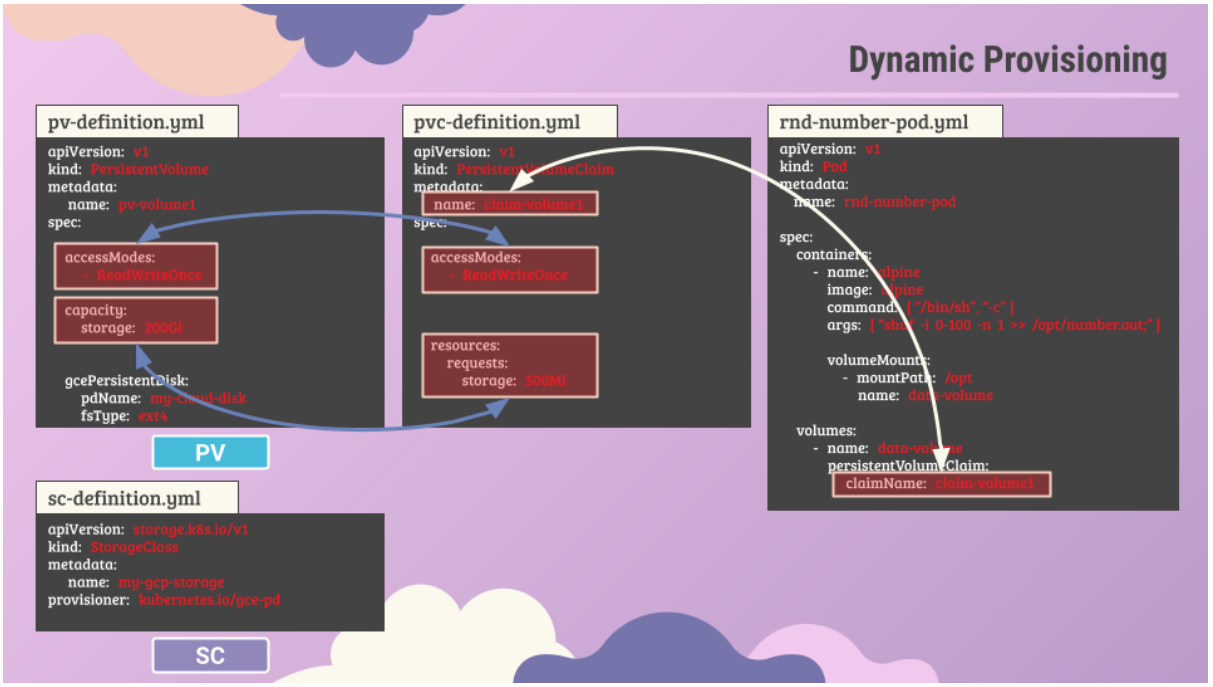
С помощью классов хранилища ты можешь определить provisioner, вроде дискового хранилища Google, который может автоматически провижинить экземпляр хранения в Google Cloud и присоединять его к PODs при создании PVC.



Это называется динамической подготовкой томов - Dynamic Provisioning of Volumes.

Мы делаем это, создавая объект StorageClass.
Его apiVersion - storage.k8s.io/v1, kind - StorageClass. Укажем имя и provisioner. Т.е. укажем какой компонент будет заниматься обработкой создания PV для вновь созданных PVC. У нас это будет kubernetes.io/gce-pd.

Возвращаясь к нашему исходному состоянию, когда у нас есть POD, использующий ссылку на PVC для своей volume, а эта PVC привязана к PV, теперь у нас появляется есть StorageClass.

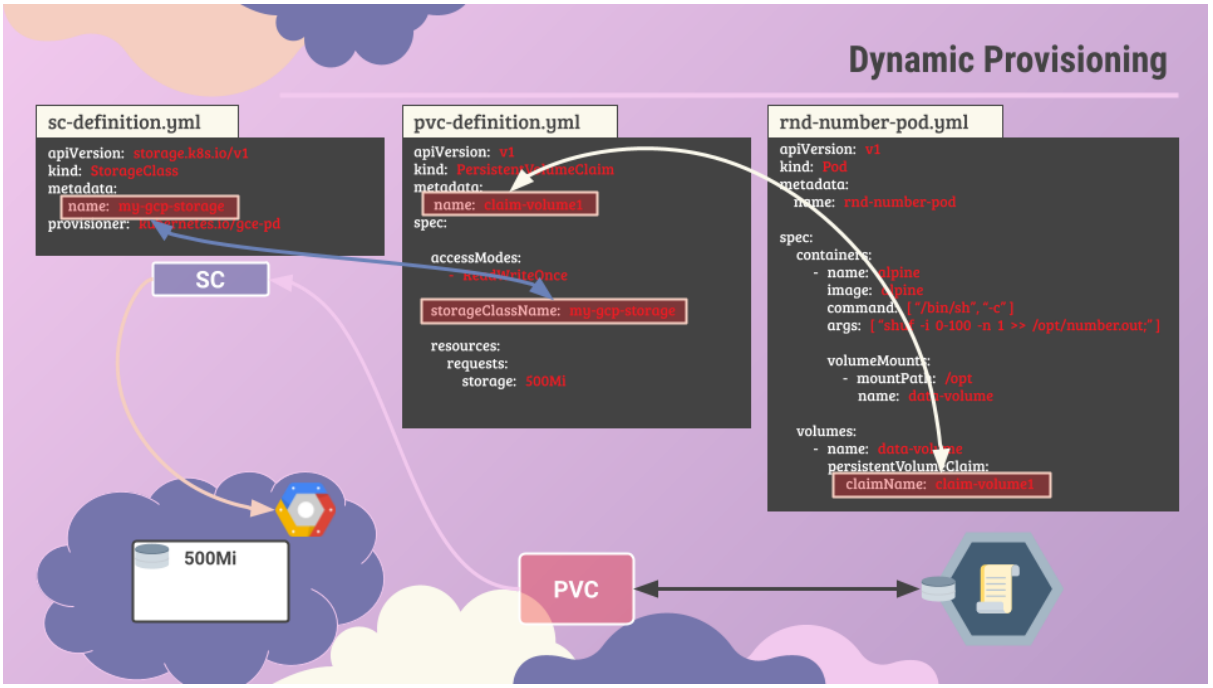


Таким образом, нам больше не нужно определение PV, потому что PV и любое связанное с ним хранилище будут созданы автоматически при создании PVC исходя из условий, определенных в StorageClass.

Чтобы PVC использовала определенный нами StorageClass, мы указываем параметр StorageClassName в определении PVC.

Вот так PVC понимает, какой StorageClass использовать. В следующий раз, когда PVC будет создана, связанный с ней класс хранилища использует заданного в нем provisioner для подготовки нового диска требуемого размера в GCP. Затем создаст PV, после чего привяжет PVC к этому persistent volume.

Помни, что он по-прежнему создает PV, просто тебе больше не нужно вручную создавать эти PV. За тебя это автоматически делает StorageClass.



Мы использовали GCE-provisioner для создания тома в GCP. Есть много других provisioners, например, для AWS EBS, AzureFile, AzureDisk, CephFS, ScaleIO и так далее.

С каждым из них мы можем передавать дополнительные параметры, такие как тип требуемого диска, нужный тип репликации и т. д., Эти параметры очень специфичны для каждого отдельного provisioner.
Для gce-pd они такие:

- type может быть pd-standard или pd-ssd
- replication-type или без репликации или региональная репликация

Итак, как видишь, мы можем создавать разные классы хранения, каждый из которых использует разные типы дисков и количество дисков.
И следовательно у классов будет разный Performance, Reliability и Cost.

Например, для разработки и проверки гипотез мы определили cheap-test класс, у него медленные HDD и нет репликации, но он самый дешевый.

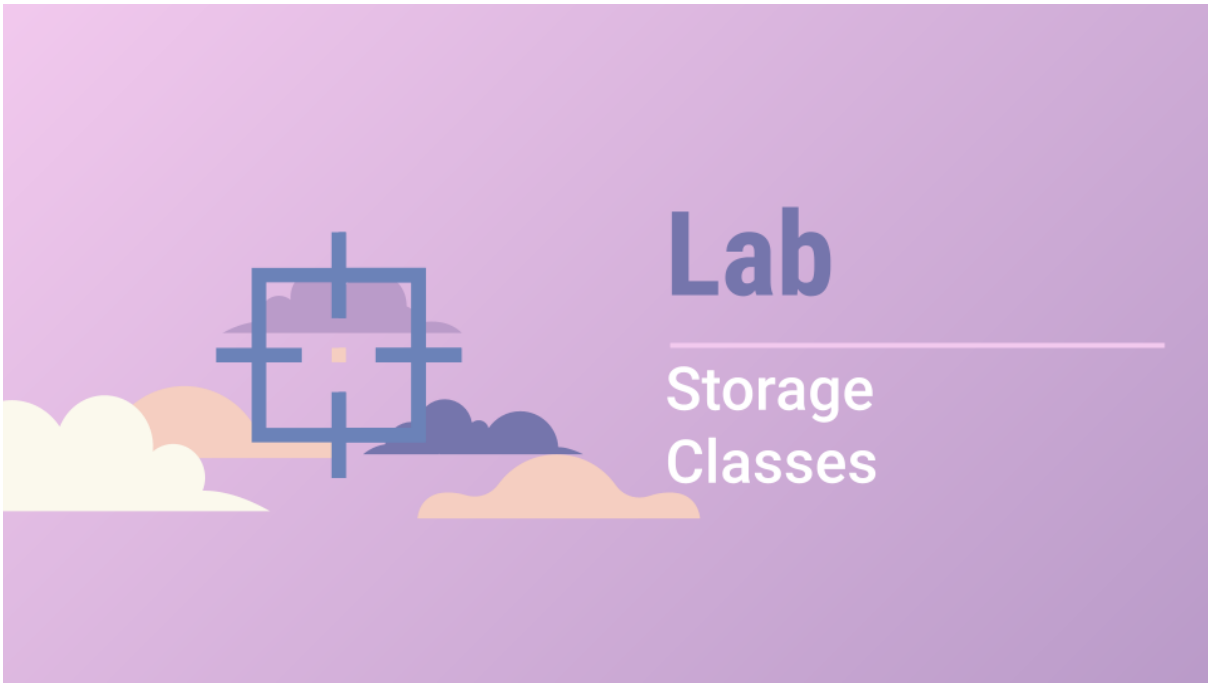
Для тестирования выкаток в stage среде, приближенной к боевой, мы определим StorageClass fast-test, он будет на более быстрых SSD, но надежность тут не требуется.

Для прода мы определим свой класс, и там будут уже соблюдены требования и по надежности, и по перфомансу.

Надеюсь теперь тебе понятна логика создателей Kubernetes, когда они назвали этот механизм Storage Classes. Это как в самолете: для разных клиентов - разные классы обслуживания.

В следующий раз, когда будем создавать PVC, нам можно просто указать StorageClass, который требуется для наших томов.

Пройди практическое упражнение далее, а я жду тебя на следующей лекции.


Lab
Storage
Classes