KUB 20: Ресурсы для управления подключаемыми томами

Описание:

Для постоянного сохранения данных в Kubernetes, как и в Docker, используются Volume. Но в отличие от Docker из-за того, что Kubernetes вместо перезапуска того же контейнера создает еще один, данные внутри контейнера теряются.
Кроме того, K8s поддерживает большое количество Volume, некоторые из них наследуются из Docker, а другие специфичны для Kubernetes. Разберем часть из них:

- gcePersistentDisk, awsElasticBlockStore, azureDisk — специфичны для соответствующих облачных провайдеров — позволяют подключать блочные диски как Volumes;
- nfs, iscsi — позволяют подключить соответствующие сетевые хранилища по общим протоколам;
- glusterfs, cephfs — подключают созданные блоки хранения в соответствующих кластерных системах хранения;
- csi — позволяет подключать Container Storage Interfaces хранилища;
- secret, configMap — приводят содержимое данных ресурсов в формат файлов, где имя ключа — это имя файла, а значение — содержимое файла (в случае секрета — в читаемом формате, а не в base64);
- hostPath — подключает директорию с хоста, где запущен под. Соответсвенно, до тех пор пока под запускается на той же ноде, данные сохраняются;
- local — схож с hostPath, но при рестарте пода Kubernetes запускает контейнер на той же ноде, что и раньше, поскольку запоминает, где Volume был создан. Однако при недоступности ноды есть способ потерять данные, так как под переедет на другую ноду;
- emptyDir — создает пустую директорию, подключенную к поду на конкретной ноде. То есть, данные в этой директории остаются до тех пор, пока не перезапускается под с переездом на другую ноду (при перезапусках контейнера данные сохраняются). У этого типа Volume отмечается особенность: дополнительно можно указать параметр medium: memory. В таком случае будет создан ram-диск;
- downwardAPI — это тип Volume, который позволяет монтировать значения метаданных пода внутри контейнера в виде файлов.

hostPath часто используют совместно с каким-то настроенным хранилищем на хосте, к примеру, с подключенным к определенной директории GlusterFS.
Для подключения Volume требуется определить 2 директивы в описании пода (или шаблона пода) — volumes (в общем описании вне контейнера) и volumeMounts (в описании контейнера).
Пример использования Volume на примере emptyDir:

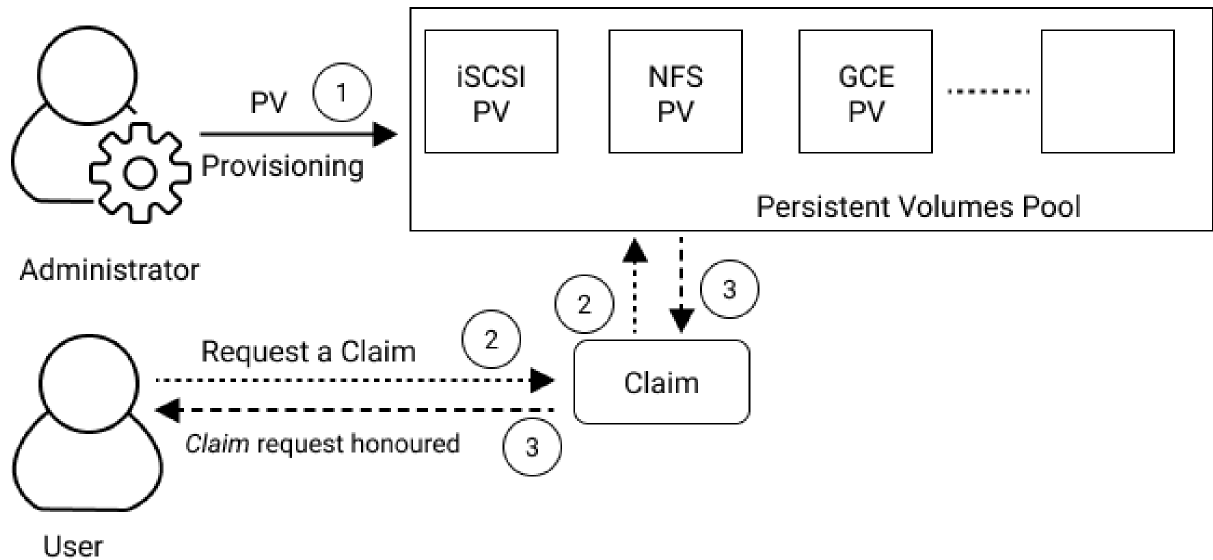
```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

В данном случае мы использовали Volume emptydir, но вместо него вы можете указать, например nfs и подключить внешний volume по nfs к вашему поду.

PersistentVolume

Обычно администратор хранилища выделяет пространство пользователям. А затем пользователи используют выделенное хранилище для своих нужд. В Kubernetes для этого служит PersistanceVolume. Эта абстракция предоставляет API для управления системами хранения (NFS, GlusterFS, CephFS, iSCSI — вот лишь небольшой и неполный список). В отличие от Docker, плагины для работы с разными системами хранения уже включены в код Kubernetes. Ничего дополнительного устанавливать не нужно (но расширять список можно при помощи CSI — Container Storage Interfaces — плагинов, которые предоставляют общий стандартный интерфейс для запросов к системам хранилища со стороны Kubernetes).

Каждый pod может запросить требуемое пространство из PV, используя Persistent Volume Claim (сокращенно — PVC).



При успешном запросе будет выделено требуемое пространство.

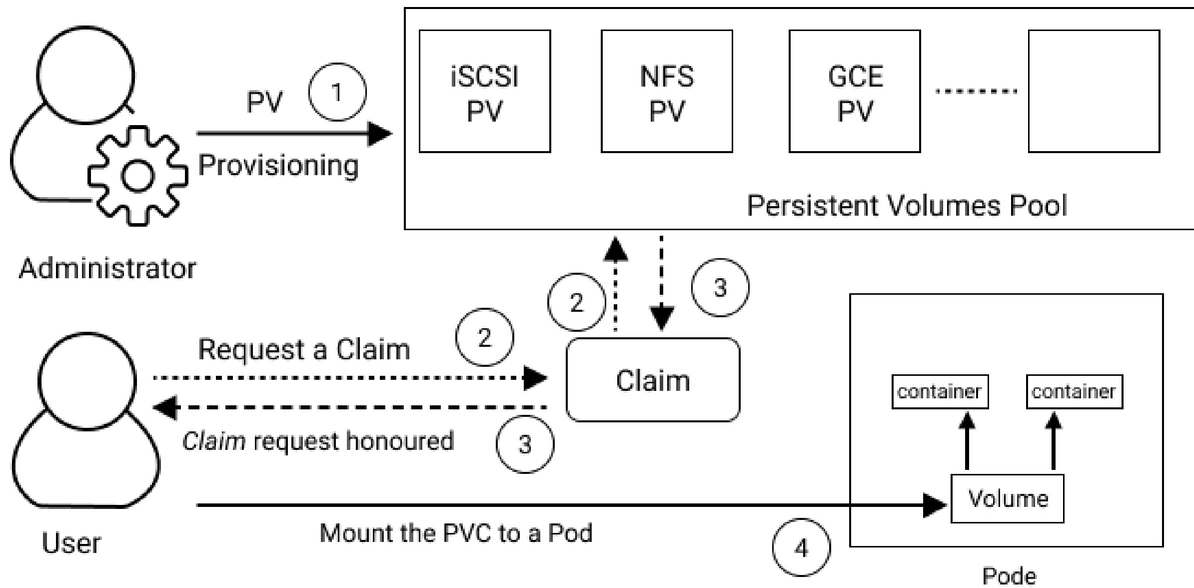


Схема работы выглядит так:

1. Вы создаете pvc - то есть запрос на диск и указываете там storage class, который хотели бы использовать.
2. PVC ищет уже созданные и свободные PersistentVolume с указанным storage class. Если он таковые находит - то подключает.
3. Если PVC не нашел свободный PV, то он обращается к provisioner'у, который указан в storage class'е и тот создает новый persistent volume.
4. PVC подключает к себе вновь созданный persistent volume.
5. При запуске пода kubelet подключает persistent volume к поду.

Для чего такие сложности? Кластер может использовать большое количество разных PV. Например, в облаке AWS есть десятки классов для дисков, все они дают разную производительность и стоят соответственно. Для логов можно использовать недорогие

шпиндельные диски, для cassandra — взять ssd. Kubernetes будет автоматически управлять всеми дисками, снимая с вас работу по созданию новых.
Ручное создание PV на примере локальных хранилищ. Создадим PV:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Это хранилище будет использовать папку /mnt/data.
Теперь создадим PVC:

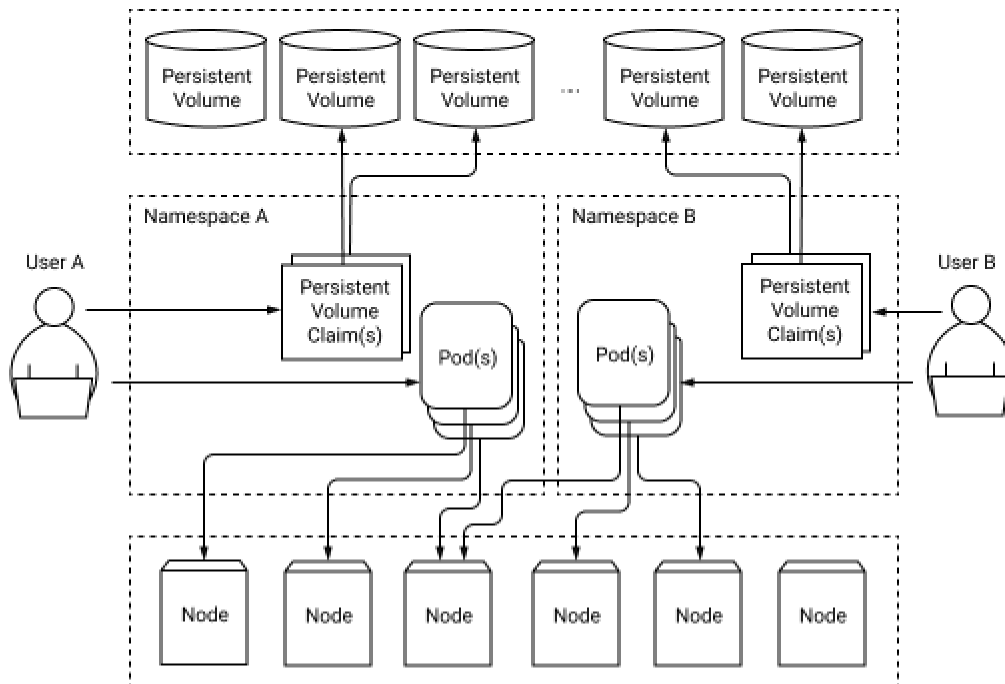
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Мы запросили 3 Гб из storageclass=manual, данный storageclass обслуживается только одним хранилищем (создали ранее). Из хранилища в 10 Гб будет запрошено 3 Гб.
А теперь подключим хранилище к приложению:

```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
```

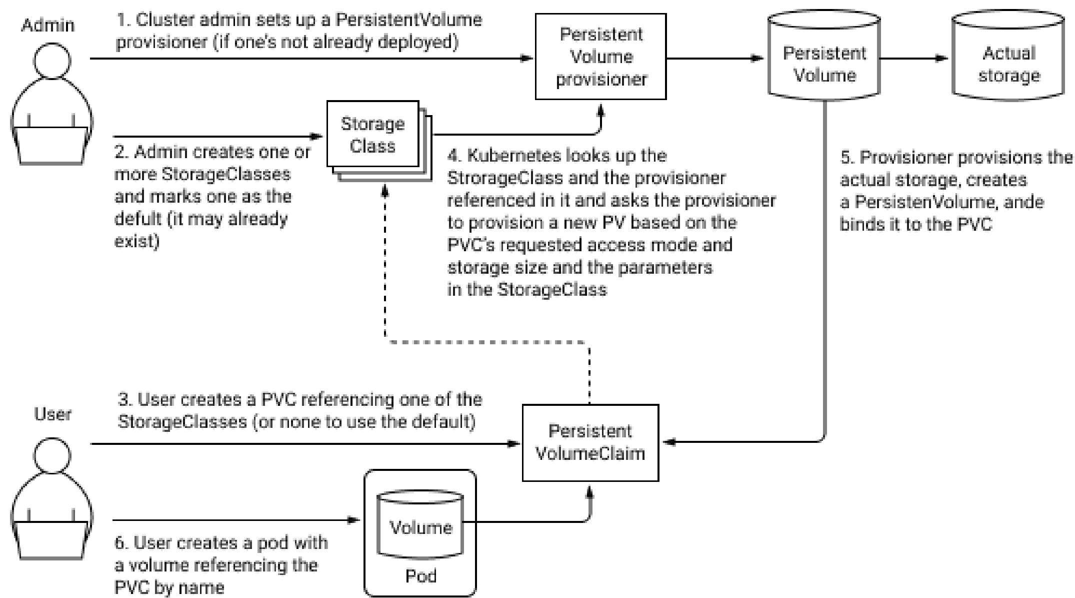
```yaml
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

Теперь при запуске контейнера nginx данный PV будет примонтирован по данному пути: /usr/share/nginx/html. Почти как docker bind mount. При добавлении файлов в эту папку их можно получить через nginx.



Provisioners

Мы почти все делали в ручном режиме. Для автоматического создания PV есть определенные типы Provisioner. Например, при запуске в облаке можно автоматически создавать PV. Это делается через абстракцию StorageClass.



StorageClass — это набор параметров для конкретного Provisioner. Provisioner — это программа, которая взаимодействует с инфраструктурой, создает диски и делает PV.
А теперь попробуем автоматический Provisioner local-path, который будет создавать директорию локально на ноде:

```
kubectl apply -f
https://raw.githubusercontent.com/rancher/local-path-provisioner
/master/deploy/local-path-storage.yaml
```

Будет создан storageclass local-path и provisioner для него в namespace local-path-provisioner. Точно также можем в ручном режиме создать PVC и pod:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: local-path-pvc
  namespace: default
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: local-path
  resources:
    requests:
      storage: 2Gi
---
apiVersion: v1
```

```
kind: Pod
metadata:
  name: volume-test
  namespace: default
spec:
  containers:
  - name: volume-test
    image: nginx:stable-alpine
    imagePullPolicy: IfNotPresent
    volumeMounts:
    - name: volv
      mountPath: /data
    ports:
    - containerPort: 80
  volumes:
  - name: volv
    persistentVolumeClaim:
      claimName: local-path-pvc
```

Самая важная часть — это persistent volume claim. Мы описали, что желаем получить 2 Гб диск класса local-path. При отправке данного yaml в API-server local-path-provisioner автоматически создаст PV, PVC, а kubelet подключит хранилище в pod.

Self Hosted provisioners

С облачными provisioners на самом деле все просто - они предустанавливаются в систему примерно так же, как мы с вами установили local-path provisioner. И дальше вы можете просто указывать нужный вам класс диска в pvc и cloud controller manager с удовольствием создаст вам диск и подключит его куда скажет kubernetes. Но что же делать с self-hosted решением?

На самом деле в Kubernetes из коробки встроены различные типы provisioners - например rbd или glusterfs. Полный список можно найти здесь. Для продакшен сред мы настоятельно рекомендуем использовать rbd - rados block device. Для этого вам потребуется установленный и настроенный сервер ceph, с которого вы сможете подключать блочные устройства. Настроить ceph можно, например по этой статье или воспользоваться официальной документацией.

К сожалению, настройка и управление ceph достаточно сложная задача, если вы не хотите потерять свои данные. Если говорить честно - то это тянет на отдельный практикум по ceph, после прохождения которого вы сможете очень просто создать storage class, который использует ceph:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
```

```
provisioner: kubernetes.io/rbd
parameters:
  monitors: 10.16.153.105:6789
  adminId: kube
  adminSecretName: ceph-secret
  adminSecretNamespace: kube-system
  pool: kube
  userId: kube
  userSecretName: ceph-secret-user
  userSecretNamespace: default
  fsType: ext4
  imageFormat: "2"
  imageFeatures: "layering"
```

В данном случае мы создаем новый storage class, который будет использовать встроенный provisioner kubernetes.io/rbd. Для него требуется указать некоторые параметры - такие как адрес monitor ceph сервера, имя администратора и его ключ, название пула, тип файловой системы и так далее. После этого вы сможете использовать данный storage class в pvc для автоматического создания блочных девайсов.
Давайте попробуем установить свой собственный provisioner - nfs - он будет автоматически создавать нам persistent volume из nfs сервера по запросу. Для этого мы воспользуемся официальной инструкцией, размещеной [здесь].
Для начала склонируем нужный репозиторий:

```
root@node1:~# git clone
https://github.com/kubernetes-sigs/nfs-subdir-external-provisioner
Cloning into 'nfs-subdir-external-provisioner'...
remote: Enumerating objects: 1292, done.
remote: Counting objects: 100% (99/99), done.
remote: Compressing objects: 100% (68/68), done.
remote: Total 1292 (delta 49), reused 70 (delta 30), pack-reused
1193
Receiving objects: 100% (1292/1292), 412.89 KiB | 4.35 MiB/s,
done.
Resolving deltas: 100% (691/691), done.
root@node1:~# cd nfs-subdir-external-provisioner/
root@node1:~/nfs-subdir-external-provisioner#
```

После этого изменим namespace, куда мы будем устанавливать provisioner и создадим его:

```
root@node1:~/nfs-subdir-external-provisioner# sed -i''
"s/namespace:.*/namespace: nfs-provisioner/g" ./deploy/rbac.yaml
./deploy/deployment.yaml
root@node1:~/nfs-subdir-external-provisioner# kubectl create ns
nfs-provisioner
namespace/nfs-provisioner created
root@node1:~/nfs-subdir-external-provisioner#
```

Теперь создадим сервисный аккаунт и назначим ему права для общения с Kubernetes:

```
root@node1:~/nfs-subdir-external-provisioner# kubectl apply -f
deploy/rbac.yaml
serviceaccount/nfs-client-provisioner created
clusterrole.rbac.authorization.k8s.io/nfs-client-provisioner-run
ner created
clusterrolebinding.rbac.authorization.k8s.io/run-nfs-client-prov
isioner created
role.rbac.authorization.k8s.io/leader-locking-nfs-client-provisi
oner created
rolebinding.rbac.authorization.k8s.io/leader-locking-nfs-client-
provisioner created
root@node1:~/nfs-subdir-external-provisioner#
```

Нам осталось только отредактировать наш deploy/deployment.yaml и изменить адрес нашего nfs сервера:

```
   spec:
      serviceAccountName: nfs-client-provisioner
      containers:
        - name: nfs-client-provisioner
          image:
k8s.gcr.io/sig-storage/nfs-subdir-external-provisioner:v4.0.2
          volumeMounts:
            - name: nfs-client-root
              mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: k8s-sigs.io/nfs-subdir-external-provisioner
            - name: NFS_SERVER
              value: 10.114.0.5
            - name: NFS_PATH
              value: /opt/nfs/kubernetes
      volumes:
```

```
    - name: nfs-client-root
      nfs:
        server: 10.114.0.5
        path: /opt/nfs/kubernetes
```

10.114.0.5 - адрес нашего nfs сервера, а /opt/nfs - директория, которую он экспортирует. Данному provisioner'у нужен подключенный volume с nfs для того, чтобы он мог создавать внутри директории, которые впоследствии будут использованы для persistent volume. То есть схема работы такая - мы запрашиваем persisten volume, provisioner создает директорию на nfs и после создает persistent volume, который использует это директорию. Запускаем его:

```
root@node1:~/nfs-subdir-external-provisioner# kubectl apply -f
deploy/deployment.yaml
deployment.apps/nfs-client-provisioner created
root@node1:~/nfs-subdir-external-provisioner#
```

Теперь можем создать наш storage class, который будет использоваться для запроса volumes:

```
root@node1:~/nfs-subdir-external-provisioner# kubectl apply -f
deploy/class.yaml
storageclass.storage.k8s.io/managed-nfs-storage created
root@node1:~/nfs-subdir-external-provisioner# kubectl get
storageclass
NAME                  PROVISIONER
RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
managed-nfs-storage
k8s-sigs.io/nfs-subdir-external-provisioner   Delete
Immediate           false                  6s
```

Storage class был успешно создан и использует он наш установленный provisioner. Давайте теперь попробуем создать простой pvc, который использует storage class managed-nfs-storage и создаст нам отдельный persistent volume:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-claim
spec:
  storageClassName: managed-nfs-storage
  accessModes:
    - ReadWriteMany
  resources:
    requests:
```

```
      storage: 1Mi
```

Применяем:

```
root@node1:~/nfs-subdir-external-provisioner# kubectl apply -f
deploy/test-claim.yaml
persistentvolumeclaim/test-claim unchanged
root@node1:~/nfs-subdir-external-provisioner# kubectl get pvc
NAME          STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS          AGE
test-claim    Bound    pvc-07a1fb0f-cddc-4daa-b23a-f54a0b169e8d
1Mi        RWX            managed-nfs-storage   7m11s
root@node1:~/nfs-subdir-external-provisioner# kubectl get pv
NAME                                       CAPACITY   ACCESS
MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS          REASON   AGE
pvc-07a1fb0f-cddc-4daa-b23a-f54a0b169e8d   1Mi        RWX
Delete           Bound    default/test-claim
managed-nfs-storage            94s
root@node1:~/nfs-subdir-external-provisioner#
```

Смотрите - наш volume успешно создался. Давайте посмотрим что произошло на nfs сервере в директории /opt/nfs:

```
root@data:~# ls -la /opt/nfs/
total 12
drwxr-xr-x 3 root root 4096 Apr 16 12:00 .
drwxr-xr-x 5 root root 4096 Apr 16 11:09 ..
drwxrwxrwx 2 root root 4096 Apr 16 12:00
default-test-claim-pvc-07a1fb0f-cddc-4daa-b23a-f54a0b169e8d
```

Как видно - Provisioner создал новую директорию, которая будет использоваться для persistent volume.

Полезные ссылки:

- Persistent Volumes (official docs)
- Container Storage Interface (CSI) for Kubernetes GA (official blog)
- Volumes (official docs)
- NFS Subdir provisioner

Задание:

1. Установите nfs subdir provisioner в namespace provisioner (обратите внимание, что для монтирования nfs на ноде должен быть установлен пакет nfs-common).
2. Создайте storage class nfs-sc, который будет использовать установленный provisioner.
3. Создайте pvc с именем nfs-data и размером 300Mb в namespace default.
4. Создайте под redis с образом redis:latest, который будет использовать созданный pvc и монтировать его в /data внутри контейнера. Redis должен быть создан в namespace default.
5. Отправьте задание на проверку.