Índice

Sessão 7: Armazenamento no Kubernetes

1) Persistência de arquivos

1. Crie um pod com o nome `writer`, usando a imagem `busybox` e executando o comando `sleep 3600`.

   A seguir, execute um *shell* interativo no pod e crie o diretório novo `/data`. Dentro dele, crie um arquivo com o nome `hello` e conteúdo `world`.

   ▼ *Visualizar resposta*

   Vamos lá: primeiro, criamos o pod.

   ```
   # kubectl run writer --image=busybox -- sleep 3600
   pod/writer created
   ```

   Para criar o arquivo, iremos acessar o pod com um *shell* interativo:

   ```
   # kubectl exec -it writer -- /bin/sh
   ```

   Verifique que o contexto foi alterado corretamente:

   ```
   / # whoami ; hostname
   root
   writer
   ```

   Note que não existe ainda nenhum diretório `data` na raiz do sistema de arquivos. Vamos criá-lo.

```
/ # ls
bin   dev   etc   home  proc  root  sys   tmp   usr   var
```

```
/ # mkdir /data
```

Para criar o arquivo `hello`, basta redirecionar a saída do comando `echo`, assim:

```
/ # echo world > /data/hello
```

Vejamos se funcionou:

```
/ # cat /data/hello
world
```

2. Agora, remova o pod `writer`. O conteúdo do arquivo `/data/hello` permanece acessível?

   ▼ *Visualizar resposta*

   Antes de remover o pod, devemos sair do *shell* interativo dentro do pod `writer`. Verifique que o contexto foi alterado com sucesso.

   ```
   # whoami ; hostname
   root
   s2-master-1
   ```

   Agora, remova o pod.

   ```
   # kubectl delete pod writer
   pod "writer" deleted
   ```

   O conteúdo está inacessível, pois ele não foi armazenado permanentemente em nenhum lugar. De fato, o pod em si não pode ser acessado, uma vez que tenha sido removido.

3. Vamos solucionar isso. Primeiro, crie o diretório novo `/pods`.

   A seguir, execute o pod `writer` com as mesmas configurações utilizadas no passo (a), mas desta vez utilizando um volume do tipo `HostPath` que mapeie o diretório `/pods` no *node* para o diretório `/data` no contexto do pod.

   Finalmente, acesse um *shell* interativo no pod e crie o arquivo `/data/hello` com o conteúdo `world`, e verifique: o mesmo arquivo existe dentro da pasta `/pods` no *node*? Se não, porquê?

   ▼ *Visualizar resposta*

   Criar o diretório, evidentemente, é bastante simples.

```
# mkdir /pods
```

A seguir, criamos o pod `writer`. Desta vez, como queremos especificar um volume persistente, iremos defini-lo através de um arquivo YAML com o conteúdo que se segue:

```
apiVersion: v1
kind: Pod
metadata:
  name: writer
spec:
  containers:
  - image: busybox
    name: writer
    args:
    - sleep
    - "3600"
    volumeMounts:
    - mountPath: /data
      name: data-dir
  volumes:
  - name: data-dir
    hostPath:
      path: /pods
```

Crie o pod a partir do arquivo YAML:

```
# kubectl apply -f writer.yaml
pod/writer created
```

Agora, acessamos o pod com um *shell* interativo, como feito anteriormente.

```
# kubectl exec -it writer -- /bin/sh
```

```
/ # whoami ; hostname
root
writer
```

Note, desta vez, que a pasta `/data` já foi automaticamente criada pelo sistema — afinal, é sob esta pasta que o volume persistente encontra-se montado.

```
/ # ls -d /data/
/data/
```

Vamos criar o arquivo:

```
/ # echo world > /data/hello
```

Feito isso, encerre o *shell* interativo, voltando ao contexto no *host* `s2-master-1`. Verifique:

```
# whoami ; hostname
root
s2-master-1
```

Finalmente, acesse o arquivo criado dentro do pod:

```
# cat /pods/hello
cat: /pods/hello: No such file or directory
```

Ué... não funcionou? Iremos investigar, a seguir.

4. Onde terá ido parar o arquivo? Para isso, é importante perceber que volumes do tipo `hostPath` armazenam arquivos na hierarquia de diretórios do *node* em que o pod está executando.

   Sabendo disso, responda: em qual *node* o pod `writer` está executando?

   Acesse esse *node* via SSH e determine se o caminho `/pods` existe, bem como se o arquivo `/pods/hello` foi criado e possui o conteúdo esperado.

   ▼ *Visualizar resposta*

   Como dito no enunciado, arquivos armazenados em volumes do tipo `hostPath` são gravados na hierarquia de diretórios do *node* em que o pod está executando. Sabendo disso, fica a pergunta: em que *node* está executando o pod `writer`?

   ```
   # kubectl get pod writer -o custom-
   columns=NAME:.metadata.name,NODE:.spec.nodeName
   NAME     NODE
   writer   s2-node-1
   ```

   Ah, isso explica tudo! Acesse o *node* `s2-node-1` via SSH, via `vagrant ssh` ou usando o comando que segue:

   ```
   # sudo -u vagrant ssh -i /home/vagrant/.ssh/tmpkey vagrant@s2-node-1

   (...)

   vagrant@s2-node-1:~$
   ```

   Uma vez dentro do *host* `s2-node-1`, eleve privilégio para `root`...

```
vagrant@s2-node-1:~$ sudo -i
```

```
root@s2-node-1:~# whoami ; hostname
root
s2-node-1
```

E liste o diretório `/pods`. Note que, agora sim, conseguimos visualizar o arquivo criado através do pod `writer`.

```
root@s2-node-1:~# ls /pods/
hello
```

E o seu conteúdo? Será o mesmo?

```
root@s2-node-1:~# cat /pods/hello
world
```

5. Encerre a sessão SSH no *host* `s2-node-1` antes de prosseguir, garantindo que você está logado na máquina `s2-master-1`, como o usuário `root`.

```
# whoami ; hostname
root
s2-master-1
```

2) Volumes persistentes e *claims*

A solução empregada na atividade (1) possui alguns problemas, que iremos atacar em ordem.

O primeiro desses problemas é que o arquivo YAML que define o pod `writer` possui um grande acoplamento entre a definição dos métodos de provisionamento e consumo do *storage*.

Para solucionar isso o Kubernetes provê objetos do tipo **PersistentVolume** (PVs), que podem ser provisionados manualmente por um administrador ou dinamicamente via StorageClasses. PVs são entidades de armazenamento plugáveis, assim como os volumes que utilizamos na atividade anterior, mas possuem um ciclo de vida independente dos pods que o utilizam.

Para utilizar PVs temos objetos do tipo **PersistentVolumeClaim** (PVCs), que consistem em requisições de armazenamento por um usuário. Fazendo um paralelo, assim como pods consomem recursos de *nodes*, como CPU e memória, PVCs consomem recursos de PVs. PVCs podem requisitar especificidades como tamanho a armazenar e tipos de acesso (`ReadWriteOnce`, `ReadOnlyMany` ou `ReadWriteMany`).

1. Como visto acima, diversos tipos de acesso são suportados por PVs e PVCs: `ReadWriteOnce`, `ReadOnlyMany` ou `ReadWriteMany`. Consulte a documentação do Kubernetes e responda: o que

significam cada um desses modos? Quais suas abreviações, ao utilizar a linha de comando?

▼ *Visualizar resposta*

Como documentado em https://kubernetes.io/docs/concepts/storage/persistent-volumes/#access-modes , temos três modos de acesso possíveis para PVs:

- `ReadWriteOnce`: neste modo, o volume pode ser montado como leitura-escrita por um único *node*.
- `ReadOnlyMany`: neste modo, o volume pode ser montado como somente-leitura por múltiplos *nodes*.
- `ReadWriteMany`: neste modo, o volume pode ser montado como leitura-escrita por múltiplos *nodes*.

Note que a definição dos modos acima especifica que os volumes serão montados por ***nodes***, e não por **pods**. Outro aspecto relevante é que um volume só pode ser montado utilizando um desses modos de acesso, mesmo que múltiplos modos sejam suportados.

Quanto às abreviações ao utilizar a linha de comando, estas são:

- `RWO`: `ReadWriteOnce`
- `ROX`: `ReadOnlyMany`
- `RWX`: `ReadWriteMany`

2. Vamos começar pelo PersistentVolume. Crie um PV com o nome `pv-data` e tipo `hostPath`, requisitando um espaço de 200Mi e modo de acesso `ReadWriteMany`. Utilize o mesmo caminho do volume especificado na atividade anterior.

   A seguir, verifique o funcionamento de sua configuração com o comando `kubectl get pv`.

   ▼ *Visualizar resposta*

   O arquivo YAML de definição do PV é relativamente simples, como visto abaixo.

   ```yaml
   apiVersion: v1
   kind: PersistentVolume
   metadata:
     name: pv-data
   spec:
     capacity:
       storage: 200Mi
     accessModes:
       - ReadWriteMany
     hostPath:
       path: "/pods"
   ```

   Para criá-lo, basta utilizar `kubectl apply` (ou `create`).

   ```
   # kubectl apply -f pv-data.yaml
   ```

```
persistentvolume/pv-data created
```

Para visualizar os dados do objeto utilize `kubectl get pv`:

```
# kubectl get pv pv-data
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
STORAGECLASS   REASON   AGE
pv-data   200Mi      RWX            Retain           Available
36s
```

3. Agora, para o PersistentVolumeClaim. Crie o PVC `pvc-data`, com requerimento de armazenamento de 100Mi e modo de acesso `ReadWriteOnce`.

   A seguir, verifique: quais são os estados do PVC `pvc-data` e do PV `pv-data`. Porquê?

   ▼ *Visualizar resposta*

   O arquivo YAML que define o PVC é ainda mais simples que o anterior. Veja:

   ```
   apiVersion: v1
   kind: PersistentVolumeClaim
   metadata:
     name: pvc-data
   spec:
     accessModes:
       - ReadWriteOnce
     resources:
       requests:
         storage: 100Mi
   ```

   A seguir, criamos o objeto...

   ```
   # kubectl apply -f pvc-data.yaml
   persistentvolumeclaim/pvc-data created
   ```

   E verificamos seu estado. Note que ele se encontra como `Pending`.

   ```
   # kubectl get pvc pvc-data
   NAME       STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
   pvc-data   Pending                                                     5s
   ```

   O PV, por outro lado, ainda é marcado como `Available`.

   ```
   # kubectl get pv pv-data
   NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
   STORAGECLASS   REASON   AGE
   ```

```
pv-data   200Mi      RWX            Retain           Available
5m24s
```

Ao investigar os eventos do PVC, o motivo fica claro: nenhum PV é identificado como disponível para atender os requisitos do PVC. Mas porquê?

```
# kubectl describe pvc pvc-data | tail -n1
  Normal  FailedBinding  9s (x6 over 82s)  persistentvolume-controller  no
persistent volumes available for this claim and no storage class is set
```

4. Corrija o problema identificado com o PVC `pvc-data`.

A seguir, verifique o estado do PVC `pvc-data` e do PV `pv-data` para garantir o funcionamento de sua configuração.

▼ *Visualizar resposta*

A razão para o erro identificado no passo anterior é que o PV e o PVC não possuem modos de acesso compatíveis: enquanto o PV possui o modo `ReadWriteMany`, o PVC exige o modo `ReadWriteOnce`. Para corrigir isso, basta editar o arquivo YAML de definição do PVC:

```
# sed -i 's/ReadWriteOnce/ReadWriteMany/' pvc-data.yaml
```

Agora, delete o objeto e recrie-o.

```
# kubectl delete pvc pvc-data ; kubectl apply -f pvc-data.yaml
persistentvolumeclaim "pvc-data" deleted
persistentvolumeclaim/pvc-data created
```

Ao visualizar o estado do PVC, imediatamente notamos que ele está marcado como `Bound`:

```
# kubectl get pvc pvc-data
NAME       STATUS   VOLUME    CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc-data   Bound    pv-data   200Mi      RWX                           8s
```

Semelhantemente, o PV foi atualizado e agora possui o mesmo estado.

```
# kubectl get pv pv-data
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS   REASON   AGE
pv-data   200Mi      RWX            Retain           Bound    default/pvc-data
9m15s
```

Note, ainda, que o espaço disponível para o PVC `pvc-data` é de 200Mi, e não os 100Mi solicitados originalmente. Dado que o PV oferece um espaço maior que o solicitado pelo PVC,

o Kubernetes identifica que é possível atender o requisito do *claim* sem maiores problemas. Se a situação fosse invertida, contudo (isto é, o PVC solicitando um espaço de armazenamento superior ao que é disponibilizado pelo PV), a requisição não poderia ser atendida.

5. Faça com que o pod `writer` utilize o PVC `pvc-data` criado no passo anterior, substituindo o volume anteriormente configurado.

   Após a recriação do pod, crie o arquivo novo `/data/pvc-hello` com o conteúdo `world`, e valide sua presença no sistema de arquivos do *node* hospedeiro.

   ▼ *Visualizar resposta*

   Para fazer a alteração solicitada, basta editar a seção `spec.volumes` do arquivo YAML. Seu conteúdo deverá ficar assim:

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: writer
   spec:
     containers:
     - image: busybox
       name: writer
       args:
       - sleep
       - "3600"
       volumeMounts:
       - mountPath: /data
         name: data-dir
     volumes:
     - name: data-dir
       persistentVolumeClaim:
         claimName: pvc-data
   ```

   Remova o pod, e recrie-o usando o novo arquivo de definição.

   ```
   # kubectl delete pod writer ; kubectl apply -f writer.yaml
   pod "writer" deleted
   pod/writer created
   ```

   Agora, utilize o `kubectl exec` para criar o arquivo solicitado. Veja que podemos realizar essa ação sem necessariamente iniciar um *shell* interativo, com o comando que se segue.

   ```
   # kubectl exec -it writer -- /bin/sh -c 'echo world > /data/pvc-hello'
   ```

   Como estabelecido, o pod `writer` está executando no *node* `s2-node-1` (devido ao *node* `s2-master-1` possuir um *taint* aplicado, como vimos na sessão 3). Para maior agilidade, podemos iniciar uma sessão SSH e verificar o conteúdo do arquivo criado pelo comando anterior em

um *one-liner*. Veja:

```
# sudo -u vagrant ssh -i /home/vagrant/.ssh/tmpkey vagrant@s2-node-1 cat
/pods/pvc-hello
world
```

6. Considere a política de recuperação (*Reclaim Policy*) do PV `pv-data`. O que ocorreria com esse PV caso o PVC `pvc-data` fosse removido?

   Quais outras políticas de recuperação poderiam ter sido escolhidas?

   ▼ *Visualizar resposta*

   A política de recuperação do PV pode ser visualizada com os comandos `kubectl get` ou `kubectl describe`. Indo direto ao ponto, podemos usar JSONPath para buscar apenas o campo relevante:

   ```
   # kubectl get pv pv-data -o custom-
   columns=NAME:.metadata.name,RECLAIMPOL:.spec.persistentVolumeReclaimPolicy
   NAME      RECLAIMPOL
   pv-data   Retain
   ```

   Quanto aos demais *reclaim policies* possíveis, estes estão documentados em https://kubernetes.io/docs/concepts/storage/persistent-volumes/#reclaiming . Vejamos quais são eles:

   - ***Retain***: este modo permite a recuperação manual do recurso. Quanto o PVC é removido, o PV permanece inalterado e o volume é marcado como "liberado". Ele ainda não pode ser utilizado por outro *claim*, contudo, já que os dados do *claim* anterior ainda não foram tratados.
   - ***Delete***: neste modo, a deleção do PVC ocasiona a remoção também do PV e do *asset* de armazenamento em infraestrutura externa suportada, como AWS EBS, GCE PD ou volume Cinder.
   - ***Recycle***: neste modo, o sistema realiza uma limpeza básica (`rm -rf /${VOLUME}/*`) e o marca como disponível para outro *claim*.

7. Vamos testar! Remova o PVC `pvc-data` e verifique: o que acontece? Porquê?

   ▼ *Visualizar resposta*

   Vamos lá:

   ```
   # kubectl delete pvc pvc-data
   persistentvolumeclaim "pvc-data" deleted
   <HANGUP>
   ```

   Após esse comando, o PVC fica travado em estado `Terminating`, e o *shell* fica indisponível. Isto ocorre porque o PVC ainda está sendo utilizado por um pod ativo (no caso, o pod `writer`).

8. Encerre o processo de remoção do PVC e, a seguir, remova o pod `writer`. O que acontece com o PVC `pvc-data` e o PV `pv-data`?

   Verifique o conteúdo do diretório `/pods` no *node* hospedeiro: o que aconteceu com os dados armazenados nesse local?

   ▼ *Visualizar resposta*

   Para retomar o controle do *shell*, digite `CTRL + C`. Em seguida, remova o pod:

   ```
   # kubectl delete pod writer
   pod "writer" deleted
   ```

   Note que o PVC é removido imediatamente:

   ```
   # kubectl get pvc pvc-data
   Error from server (NotFound): persistentvolumeclaims "pvc-data" not found
   ```

   O PV, por outro lado, fica com o estado *Released* (liberado) — como seria de se esperar para o *access mode* `Retain`, explicado anteriormente.

   ```
   # kubectl get pv pv-data
   NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM
   STORAGECLASS   REASON   AGE
   pv-data   200Mi      RWX            Retain           Released   default/pvc-data
   23m
   ```

   De fato, os dados escritos pelo pod `writer` ainda permanecem acessíveis no volume persistente, como constatado pelo comando abaixo:

   ```
   # sudo -u vagrant ssh -i /home/vagrant/.ssh/tmpkey vagrant@s2-node-1 cat
   /pods/pvc-hello
   world
   ```

9. Antes de prosseguir, remova o PV `pv-data`.

   ```
   # kubectl delete pv pv-data
   persistentvolume "pv-data" deleted
   ```

3) *Storage Classes*

StorageClasses são formas dos administradores de um *cluster* Kubernetes descreverem as classes que armazenamento oferecidas pelo ambiente. Essas classes podem mapear níveis de qualidade de serviço, políticas de backup, velocidade de acesso ou quaisquer outros atributos arbitrários definidos pelos administradores.

Uma funcionalidade interessante provida por StorageClasses é o fato de que podemos provisionar volumes de forma dinâmica, isto é, sob demanda. Sem esse tipo de recurso os administradores devem, primeiro, providenciar volumes manualmente em seus ambiente de *storage* ou no provedor de *cloud* e, segundo, criar objetos do tipo PersistentVolume para representá-los no Kubernetes. Com o provisionamento dinâmico, esses recursos são criados quando solicitados pelo usuário.

3.1) Provisionamento manual

Para melhor ambientação com StorageClasses iremos primeiramente trabalhar com o *provisioner* `Local`, que não suporta provisionamento dinâmico. Não obstante, ainda neste caso é vantajoso trabalhar com StorageClasses em lugar de PVs diretamente, por motivos que veremos nos passo a seguir.

1. Antes de criar o StorageClass propriamente dito, é importante saber que além do seu *provisioner* (o *plugin* utilizado para provisionar PVs, como NFS, CephFS, Glusterfs ou AWSElasticBlockStore), um outro parâmetro a ser definido é o modo de associação (*Volume Binding Mode*) a ser utilizado.

   Como estabelecido anteriormente, nesta atividade iremos utilizar o StorageClass com um *provisioner* do tipo `Local`. Pesquise na documentação oficial do Kubernetes, e responda: quais *Volume Binding Modes* são suportados por esse *provisioner*? O que cada um deles faz?

   ▼ *Visualizar resposta*

   Como documentado em https://kubernetes.io/docs/concepts/storage/storage-classes/#volume-binding-mode , há dois *Volume Binding Modes* disponíveis: `Immediate` e `WaitForFirstConsumer`:

   - `Immediate`: por padrão, este modo indica que o *binding* do volume ocorrerá assim que o PVC for criado. Assim, PVs serão associados ou provisionados sem conhecimento dos requerimentos de agendamento de pods. Para *backends* de armazenamento restritos por topologia ou inacessíveis a todos os *nodes* do *cluster*, isso pode resultar em pods não-agendáveis.
   - `WaitForFirstConsumer`: a situação acima pode ser solucionada com o modo `WaitForFirstConsumer`, que atrasa a associação e criação de PVs até que um pod que utilize o PVC seja criado. PVs serão, então, selecionados ou provisionados considerando a topologia especificada pelos requisitos de agendamento do pod, incluindo aspectos como requerimento de recursos, seletores de *nodes*, afinidade e anti-afinidade de pods e *taints* e *tolerations*.

2. Agora que os *Volume Binding Modes* válidos para o *provisioner* `Local` são conhecidos, vamos ao trabalho.

   Crie um StorageClass com o nome `sc-data`, *provisoner* do tipo `Local` e *Volume Binding Mode* `WaitForFirstConsumer`. Verifique o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   O arquivo YAMl que define o StorageClass é bastante simples, e mostrado a seguir.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: sc-data
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

Crie-o com `kubectl apply`:

```
# kubectl apply -f sc-data.yaml
storageclass.storage.k8s.io/sc-data created
```

E, finalmente, verifique o sucesso na criação do objeto.

```
# kubectl get sc
NAME      PROVISIONER                    RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
sc-data   kubernetes.io/no-provisioner   Delete          WaitForFirstConsumer
false                  3s
```

3. Qual é a política de recuperação (*Reclaim Policy*) do StorageClass `sc-data`? Porquê?

   ▼ *Visualizar resposta*

   Esse dado é mostrado no último comando do passo acima. Podemos também buscar o campo específico:

   ```
   # kubectl get sc sc-data -o custom-
   columns=NAME:.metadata.name,RECLAIMPOL:.reclaimPolicy
   NAME      RECLAIMPOL
   sc-data   Delete
   ```

   Como documentado em https://kubernetes.io/docs/concepts/storage/storage-classes/#reclaim-policy , PVs dinamicamente criados por um StorageClass terão sua política de recuperação atribuída por herança. Caso nenhuma política seja especificada quando da criação do StorageClass (exatamente o que fizemos, no passo anterior), o *reclaim policy* assumirá o valor padrão `Delete`.

4. Como visto anteriormente o StorageClass do tipo `Local` não suporta provisionamento dinâmico — assim sendo, iremos fazê-lo manualmente.

   Crie o PV `sc-pv-data` com tamanho de 250 Mi, modo de acesso `ReadWriteOnce`, política de recuperação `Retain` e o mesmo caminho usado anteriormente, `/pods`. Evidentemente, faça com que esse PV utilize o StorageClass `sc-data`.

   Mais um detalhe: algo que foi omitido na atividade anterior, mas que é bastante relevante, é que PVs do tipo `local` devem explicitamente ajustar a afinidade em relação a *nodes* — assim,

pods que utilizem esses PVs serão agendados apenas em *nodes* que podem ser selecionados. Essa características não é tão relevante no momento, já que nosso *cluster* possui apenas um *node* agendável (a máquina `s2-node-1`), mas deve ser considerado em *cluster* maiores. Essa característica é documentada neste link: https://kubernetes.io/docs/concepts/storage/persistent-volumes/#node-affinity

▼ *Visualizar resposta*

O arquivo YAML que define o PV é mostrado abaixo. Note o uso do atributo `spec.nodeAffinity` para garantir o *node* de agendamento do PV, bem como a customização do *reclaim policy* via `spec.persistentVolumeReclaimPolicy`.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: sc-pv-data
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 250Mi
  local:
    path: /pods
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - s2-node-1
  persistentVolumeReclaimPolicy: Retain
  storageClassName: sc-data
```

Crie o objeto:

```
# kubectl apply -f sc-pv-data.yaml
persistentvolume/sc-pv-data created
```

E verifique o funcionamento de sua configuração.

```
# kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
STORAGECLASS   REASON   AGE
sc-pv-data   250Mi      RWO            Retain           Available           sc-
data                4s
```

Note que mesmo com o *reclaim policy* do StorageClass `sc-data` sendo igual a `Delete`,

conseguimos suplantá-lo com uma configuração específica no arquivo YAML que define o PV manualmente provisionado, como objetivado.

5. Perfeito! Agora, crie o PVC `sc-pvc-data`; utilize as mesmas características (`accessMode` e tamanho de armazenamento) configurados para o PV `sc-pv-data`. Não se esqueça de mencionar o uso do StorageClass apropriado.

   Qual é o estado desse PVC após sua criação? Porquê?

   ▼ *Visualizar resposta*

   Vamos lá:

   ```
   apiVersion: v1
   kind: PersistentVolumeClaim
   metadata:
     name: sc-pvc-data
   spec:
     storageClassName: sc-data
     accessModes:
       - ReadWriteOnce
     resources:
       requests:
         storage: 250Mi
   ```

   ```
   # kubectl apply -f sc-pvc-data.yaml
   persistentvolumeclaim/sc-pvc-data created
   ```

   Note que o estado do PVC, após sua criação, é marcado como `Pending`.

   ```
   # kubectl get pvc sc-pvc-data
   NAME          STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
   sc-pvc-data   Pending                                      sc-data        30s
   ```

   Isto se deve ao uso do *volume binding mode* `WaitForFirstConsumer`, atribuído ao StorageClass. De fato, ao observarmos os eventos do PVC, é exatamente este o motivo pelo qual seu estado é marcado como pendente: ele está aguardando a criação do primeiro consumidor antes de associar-se a um PV.

   ```
   # kubectl describe pvc sc-pvc-data | tail -n1
     Normal  WaitForFirstConsumer  12s (x15 over 3m30s)  persistentvolume-controller
   waiting for first consumer to be created before binding
   ```

6. Finalmente, recrie o pod `writer`, desta vez utilizando o PVC `sc-pvc-data` criado no passo anterior.

   Após a criação do pod, crie o arquivo novo `/data/sc-pvc-hello` com o conteúdo `world`, e valide sua presença no sistema de arquivos do *node* hospedeiro.

Finalmente, verifique: qual o estado do PV `sc-pv-data`? E quanto ao PVC `sc-pvc-data`?

▼ *Visualizar resposta*

Para criar o pod segundo as especificações, basta editar o nome do PVC utilizado via `sed`.

```
# sed -i 's/pvc-data/sc-pvc-data/' writer.yaml
```

Em seguida, criamos o pod...

```
# kubectl apply -f writer.yaml
pod/writer created
```

E o arquivo solicitado:

```
# kubectl exec -it writer -- /bin/sh -c 'echo world > /data/sc-pvc-hello'
```

Verifique que o arquivo foi, de fato, escrito no volume persistente no *node* `s2-node-1`.

```
# sudo -u vagrant ssh -i /home/vagrant/.ssh/tmpkey vagrant@s2-node-1 cat
/pods/sc-pvc-hello
world
```

Uma vez criado o pod, note que o PV é marcado como `Bound`, assim como o PVC. Este é o comportamento esperado ao utilizar o *volume binding mode* `WaitForFirstConsumer`.

```
# kubectl get pv sc-pv-data
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS   REASON   AGE
sc-pv-data   250Mi      RWO            Retain           Bound    default/sc-pvc-
data   sc-data                 11m
```

```
# kubectl get pvc
NAME          STATUS   VOLUME    CAPACITY   ACCESS MODES   STORAGECLASS   AGE
sc-pvc-data   Bound    pv-data   250Mi      RWO            sc-data        7m55s
```

7. Antes de partir para a próxima atividade, remova todos os objetos criados até aqui. Iremos arquitetar uma maneira melhor de lidar com a persistência de dados, a seguir.

```
# kubectl delete pod writer ; kubectl delete pvc sc-pvc-data ; kubectl delete pv
sc-pv-data ; kubectl delete sc sc-data
pod "writer" deleted
persistentvolumeclaim "sc-pvc-data" deleted
persistentvolume "sc-pv-data" deleted
```

```
storageclass.storage.k8s.io "sc-data" deleted
```

3.2) Provisionamento dinâmico

Muito embora a configuração realizada na atividade anterior tenha funcionado, ela ainda não é ideal.

Imagine, por exemplo, que o pod `writer` fosse agendado em outro *node*: o que ocorreria nesse caso? Como o PV foi configurado com afinidade específica para o *node* `s2-node-1`, o *cluster* teria problemas em agendar o pod.

Considere, ainda, um serviço em que múltiplos pods precisam ler o mesmo dado — como uma pasta compartilhada ou um *cluster* de banco de dados. Ora, como o armazenamento configurado até aqui foi totalmente local, é fácil imaginar que dados escritos em um dos *nodes* não seria propagado para os demais.

Finalmente, note que tivemos que criar manualmente o PV antes que pudéssemos associá-lo a um PVC, e então a um pod. Essa configuração é bastante envolvida, e nada prática.

Para resolver essas questões, iremos implementar nesta atividade o provisionamento dinâmico de PVs. Para garantir que os dados fiquem sincronizados entre os diferentes *nodes* do *cluster*, utilizaremos uma solução de armazenamento distruído simples — neste caso o *Network File System* (NFS). Vamos lá?

1. Antes de mais nada, temos que implementar o servidor NFS — felizmente, isso é bastante simples. Iremos criar um servidor NFS na máquina `s2-master-1`, servindo o diretório `/pods`.

   Note que esse diretório foi criado anteriormente, mas nunca foi de fato utilizado porque todos os pods foram agendados no *host* `s2-node-1` até aqui:

   ```
   # whoami ; hostname
   root
   s2-master-1
   ```

   ```
   # ls -1a /pods
   .
   ..
   ```

   Instale os pacotes necessário ao funcionamento do NFS do lado do servidor:

   ```
   # apt install -y nfs-kernel-server nfs-common portmap
   ```

   Iremos, a seguir, realizar uma configuração simples (porém insegura) de um servidor NFS para testar o provisionamento dinâmico de volumes.

   

   Como mencionado, a configuração utilizada aqui foi propositalmente

> simplificada para acelerar a implantação do serviço e focar no tema-alvo da sessão, que é a gestão de armazenamento no Kubernetes.
>
> A configuração detalhada de permissionamento, tanto no nível do SO, serviço e ambiente de orquestração de containers aumentaria significativamente a complexidade de vários elementos, incluindo os arquivos YAML de definição de objetos, tornando a atividade consideralmente mais complexa. Nesse caso, a atividade provavelmente teria que ser realizada de forma guiada (isto é, fornecendo ao aluno as perguntas e também respostas, juntamente com comandos), alterando a didática utilizada até aqui.
>
> Em um ambiente de produção, é crítico que o permissionamento de diretórios e controle de quais usuários podem ler e escrever arquivos seja cuidadosamente planejado. Assim, ao realizar este tipo de implantação em sua organização, tenha atenção ao fazer os ajustes necessários.
>
> Como um desafio, sugere-se que o aluno tente adaptar este roteiro para controlar apropriadamente as permissões de leitura e escrita para os diferentes pods e seus usuários/grupos efetivos. Todos os conhecimentos necessários para este fim já foram tratados até aqui, neste curso. Você está preparado?

Vamos lá. Popule o arquivo `/etc/exports` com o diretório que será exportado pelo servidor NFS:

```
# echo '/pods *(rw,sync,no_subtree_check,no_root_squash,no_all_squash,insecure)' >>
/etc/exports
```

Atualize a lista de *exports* do servidor:

```
# exportfs -rv
exporting *:/pods
```

E verifique seu funcionamento:

```
# showmount -e
Export list for s2-master-1:
/pods *
```

Perfeito. Vamos agora verificar o funcionamento de nossa configuração: acesse o *host* `s2-node-1` e veja se ele consegue montar e visualizar o diretório compartilhado via NFS. Comece efetuando login no *host*, como o usuário `root`:

```
# sudo -u vagrant ssh -i /home/vagrant/.ssh/tmpkey vagrant@s2-node-1

(...)
```

```
vagrant@s2-node-1:~$
```

```
vagrant@s2-node-1:~$ sudo -i
```

```
root@s2-node-1:~# hostname ; whoami
s2-node-1
root
```

Teste a montagem do diretório remoto em qualquer local, por exemplo, na pasta `/mnt`. Note que utilizaremos o endereço IP do *host* `s2-master-1`, e não seu nome de domínio — isto se deve ao fato de que este *hostname* é definido através do arquivo `/etc/hosts` no *node* `s2-node-1` apenas, e não em um servidor DNS, e é portanto inválido dentro do contexto de pods criados dentro do *cluster*.

```
root@s2-node-1:~# mount -t nfs 192.168.68.20:/pods /mnt
```

```
root@s2-node-1:~# mount | grep '^192.168.68.20:/pods'
192.168.68.20:/pods on /mnt type nfs4
(rw,relatime,vers=4.2,rsize=524288,wsize=524288,namlen=255,hard,proto=tcp,timeo=600
,retrans=2,sec=sys,clientaddr=192.168.68.25,local_lock=none,addr=192.168.68.20)
```

Tudo certo! Antes de prosseguir, desmonte o diretório.

```
root@s2-node-1:~# umount /mnt
```

2. Vamos agora realizar a implantação do *NFS-Client Provisioner* (https://github.com/kubernetes-sigs/nfs-subdir-external-provisioner), uma solução criada para suportar o provisionamento dinâmico de PVs no Kubernetes em um servidor NFS preexistente — como o que criamos no passo anterior.

   Primeiro, garanta que você está na máquina `s2-master-1`, como usuário `root`.

   ```
   # hostname ; whoami
   s2-master-1
   root
   ```

   Vamos agora criar um ServiceAccount para o serviço de provisionamento, juntamente com um ClusterRole, ClusterRoleBinding, Role e RoleBinding. Observe que o arquivo utilizado no comando a seguir configura o provisionador para funcionamento no namespace *default* — para alterar isso, basta editar o arquivo YAML antes de aplicá-lo ao *cluster*.

   ```
   # kubectl apply -f https://raw.githubusercontent.com/kubernetes-sigs/nfs-subdir-
   ```

```
external-provisioner/master/deploy/rbac.yaml
serviceaccount/nfs-client-provisioner created
clusterrole.rbac.authorization.k8s.io/nfs-client-provisioner-runner created
clusterrolebinding.rbac.authorization.k8s.io/run-nfs-client-provisioner created
role.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner created
rolebinding.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner created
```

Vamos verificar o funcionamento do comando:

```
# kubectl get clusterrole,clusterrolebinding,role,rolebinding | grep nfs | cut -d'
' -f1
clusterrole.rbac.authorization.k8s.io/nfs-client-provisioner-runner
clusterrolebinding.rbac.authorization.k8s.io/run-nfs-client-provisioner
role.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner
rolebinding.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner
```

A seguir criaremos o StorageClass que irá consumir o recurso de armazenamento NFS. Usaremos como base o arquivo YAML disponível no mesmo GitHub do projeto *NFS-Client Provisioner*, apenas editando o nome do *provisioner* para um nome mais em linha com o recurso que configuramos no passo (a).

```
# curl -s https://raw.githubusercontent.com/kubernetes-sigs/nfs-subdir-external-
provisioner/master/deploy/class.yaml | \
  sed 's/^\(provisioner:\).*/\1 nfs.contorq.com/' | \
  kubectl apply -f -
storageclass.storage.k8s.io/nfs-client created
```

Verifique o sucesso da configuração:

```
# kubectl get sc nfs-client
NAME         PROVISIONER       RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
nfs-client   nfs.contorq.com   Delete          Immediate           false
66s
```

Finalmente, iremos criar o deployment: este será o serviço que efetivamente irá monitorar por novos PVCs e criar PVs correspondentes automaticamente, alocando-os dentro do servidor NFS. Novamente, teremos que adaptar o nome do *provisioner* (assim como feito na criação do StorageClass, acima), bem como o endereço IP do servidor NFS e o caminho exportado por ele.

```
# curl -s https://raw.githubusercontent.com/kubernetes-sigs/nfs-subdir-external-
provisioner/master/deploy/deployment.yaml | \
  sed '/PROVISIONER_NAME/{n;s/^\([[:space:]]*value:\).*/\1 nfs.contorq.com/;}' | \
  sed '/NFS_SERVER/{n;s/^\([[:space:]]*value:\).*/\1 192.168.68.20/;}' | \
  sed 's/^\([[:space:]]*server:\).*/\1 192.168.68.20/' | \
  sed '/NFS_PATH/{n;s/^\([[:space:]]*value:\).*/\1 \/pods/;}' | \
```

```
  sed 's/^\([[:space:]]*path:\).*/\1 \/pods/' | \
  kubectl apply -f -
deployment.apps/nfs-client-provisioner created
```

A seguir, verifique se o deployment, e seu pod correspondente, foram criados.

```
# kubectl get deploy,pod
NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nfs-client-provisioner   1/1     1            1           29s

NAME                                         READY   STATUS    RESTARTS   AGE
pod/nfs-client-provisioner-85f8f98c95-fs8pv   1/1     Running   0          29s
```

3. Excelente! Hora de testar o ambiente: primeiro, verifique que não existe nenhum PersistentVolume ou PersistentVolumeClaim no ambiente.

   A seguir, crie o PVC `dynamic-pvc-data` que utilize o StorageClass criado no passo anterior, usando o `accessMode ReadWriteMany` e tamanho de 500Mi.

   Finalmente, cheque se o PVC foi criado, bem como um PV correspondente.

   ▼ *Visualizar resposta*

   Vamos verificar a existência de PVs e PVCs remanescentes:

   ```
   # kubectl get pv,pvc
   No resources found
   ```

   Agora, para o PVC: utilize o arquivo YAML que se segue.

   ```yaml
   apiVersion: v1
   kind: PersistentVolumeClaim
   metadata:
     name: dynamic-pvc-data
   spec:
     storageClassName: nfs-client
     accessModes:
       - ReadWriteMany
     resources:
       requests:
         storage: 500Mi
   ```

   Crie o objeto com `kubectl apply`:

   ```
   # kubectl apply -f dynamic-pvc-data.yaml
   persistentvolumeclaim/dynamic-pvc-data created
   ```

A seguir, verifique o estado do PVC e seu PV correspondente. Note que o estado do PVC criado é *Bound* — isto se deve ao *volume binding mode* utilizado no StorageClass criado no passo anterior, que é `Immediate`.

```
# kubectl get pvc dynamic-pvc-data
NAME                STATUS   VOLUME                                     CAPACITY
ACCESS MODES   STORAGECLASS          AGE
dynamic-pvc-data    Bound    pvc-13a54631-3798-41ba-909d-24ae94e3f262   500Mi
RWX            nfs-client   48s
```

```
# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM
POLICY   STATUS   CLAIM                        STORAGECLASS          REASON   AGE
pvc-13a54631-3798-41ba-909d-24ae94e3f262   500Mi      RWX            Delete
Bound    default/dynamic-pvc-data   nfs-client           61s
```

4. Agora, recrie o pod `writer`, desta vez instruindo-o a utilizar o PVC `dynamic-pvc-data` criado no passo anterior.

   Após a criação do pod, crie o arquivo novo `/data/dynamic-pvc-hello` com o conteúdo `world`. Em seguida, responda: onde é garantida a persistência desse arquivo? Aponte o local onde o arquivo foi criado, no servidor NFS.

   Outra pergunta: em qual *node* foi agendado o pod `writer`? Em que aspecto isso difere da configuração realizada na atividade anterior.

   ▼ *Visualizar resposta*

   Novamente, para alterar a configuração do pod `writer` basta utilizar o comando `sed` e editar o nome do PVC em uso.

   ```
   # sed -i 's/sc-pvc-data/dynamic-pvc-data/' writer.yaml
   ```

   Crie o pod e escreva o arquivo solicitado.

   ```
   # kubectl apply -f writer.yaml
   pod/writer created
   ```

   ```
   # kubectl exec -it writer -- /bin/sh -c 'echo world > /data/dynamic-pvc-hello'
   ```

   Dentro da pasta compartilhada pelo servidor NFS, `/pods`, note que um diretório foi criado automaticamente:

   ```
   # ls /pods
   ```

```
default-dynamic-pvc-data-pvc-13a54631-3798-41ba-909d-24ae94e3f262
```

Dentro dele está o arquivo criado pelo pod `writer`, com o conteúdo que se espera.

```
# cat /pods/default-dynamic-pvc-data-pvc-13a54631-3798-41ba-909d-
24ae94e3f262/dynamic-pvc-hello
world
```

Veja que o pod `writer` está de fato sendo executado pelo *node* `s2-node-1`, mas seus dados persistentes estão sendo armazenados no servidor NFS `s2-master-1`. Isto comprova que nossa configuração funcionou, e que agora os locais de execução do pod e armazenamento de seus dados são independentes.

```
# kubectl get pod writer -o custom-
columns=NAME:.metadata.name,NODE:.spec.nodeName
NAME     NODE
writer   s2-node-1
```

5. Finalmente, remova o pod `writer`. O que acontece com o PVC `dynamic-pvc-data`? E quanto ao seu PV correspondente?

   A seguir, remova o PVC `dynamic-pvc-data`. O que acontece com o PV associado? E quanto aos dados armazenados no servidor NFS?

   ▼ *Visualizar resposta*

   Vamos remover o pod:

   ```
   # kubectl delete pod writer
   pod "writer" deleted
   ```

   Note que o PVC e o PV permanecem no sistema, bem como os dados criados pelo pod.

   ```
   # kubectl get pvc,pv
   NAME                                     STATUS   VOLUME
   CAPACITY   ACCESS MODES   STORAGECLASS          AGE
   persistentvolumeclaim/dynamic-pvc-data   Bound    pvc-13a54631-3798-41ba-909d-
   24ae94e3f262   500Mi      RWX           nfs-client   4m27s

   NAME                                                        CAPACITY   ACCESS
   MODES   RECLAIM POLICY   STATUS   CLAIM                     STORAGECLASS
   REASON   AGE
   persistentvolume/pvc-13a54631-3798-41ba-909d-24ae94e3f262   500Mi      RWX
   Delete           Bound    default/dynamic-pvc-data   nfs-client           4m27s
   ```

   Agora, remova o PVC:

```
# kubectl delete pvc dynamic-pvc-data
persistentvolumeclaim "dynamic-pvc-data" deleted
```

Como o *Reclaim Policy* está configurado como `Delete`, o PV também é removido no processo.

```
# kubectl get pvc,pv
No resources found
```

De igual forma, os dados armazenados no servidor NFS também são removidos, quando da exclusão do PVC `dynamic-pvc-data`.

```
# ls /pods
```

6. As atividades anteriores resolveram dois dos problemas apontados no motivador desta atividade:

   - Agora, não é mais necessário criar manualmente um PV — o provisionamento dinâmico do StorageClass `nfs-client` faz com que PVs sejam criados automaticamente assim que um PVC é solicitado.
   - Adicionalmente, verificamos que mesmo que o pod e o dado não residam no mesmo *node*, seu agendamento pod e acesso aos dados persistentes é realizado com sucesso.

   Resta, portanto, validar apenas uma questão: será que a solução implementada funciona caso dois pods diferentes precisem ler/escrever no mesmo volume persistente? O atendimento desse requisito é frequentemente necessário, especialmente em aplicações distribuídas como *clusters* de bancos de dados, por exemplo.

   Vamos verificar! Crie o PVC `novel` usando o StorageClass `nfs-client`, com tamanho de 200Mi.

   A seguir, crie o deployment `author`, com 2 réplicas e usando a imagem `busybox`, executando o comando `sleep 3600`. Utilize a estratégia de deployment `Recreate`. Usando anti-afinidade de pods e *tolerations*, garanta que os pods desse deployment executem em *nodes* diferentes do *cluster*. Finalmente, faça com que os pods desse deployment montem um volume persistente usando o PVC `novel` no diretório `/story`.

   Usando um dos pods do deployment `author`, crie um arquivo novo com qualquer conteúdo dentro do diretório `/story`; a seguir, acesse a outra réplica do deployment e verifique que seu conteúdo está acessível sob o mesmo diretório.

   ▼ *Visualizar resposta*

   Uma vez que as réplicas do deployment serão executadas em *nodes* distintos, queremos que o PV provisionado dinamicamente pelo PVC `novel` seja acessado por todos *nodes* ao mesmo tempo. Como visto anteriormente, o modo que permite leitura-escrita simultânea nesse caso é `ReadWriteMany`.

   Sabendo disso, crie o PVC com o arquivo YAML que se segue.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: novel
spec:
  storageClassName: nfs-client
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 200Mi
```

```
# kubectl apply -f novel.yaml
persistentvolumeclaim/novel created
```

Agora, para o deployment. Note o uso de *tolerations* e anti-afinidade de pods no arquivo YAML abaixo para atingir os objetivos especificados pelo enunciado, bem como a customização da estratégia do deployment.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: author
spec:
  selector:
    matchLabels:
      app: author
  replicas: 2
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: author
    spec:
      tolerations:
      - key: "node-role.kubernetes.io/master"
        operator: "Exists"
        effect: "NoSchedule"
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - author
```

```
30            topologyKey: "kubernetes.io/hostname"
31      containers:
32      - name: author
33        image: busybox
34        args:
35        - sleep
36        - "3600"
37        volumeMounts:
38        - mountPath: /story
39          name: story-dir
40      volumes:
41      - name: story-dir
42        persistentVolumeClaim:
43          claimName: novel
```

```
# kubectl apply -f author.yaml
deployment.apps/author created
```

Verifique o estado do deployment, constatando que os pods estão operacionais e executando em *nodes* distintos.

```
# kubectl get deploy author
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
author   2/2     2            2           103s
```

```
# kubectl get pod -l app=author -o custom-
columns=NAME:.metadata.name,NODE:.spec.nodeName
NAME                      NODE
author-7f85d48844-6qcd6   s2-master-1
author-7f85d48844-998vx   s2-node-1
```

Agora criaremos o arquivo `/story/chapter1` com o conteúdo `Once upon a time⋯` em um dos pods acima:

```
# kubectl exec -it author-7f85d48844-6qcd6 -- /bin/sh -c 'echo "Once upon a
time..." > /story/chapter1'
```

No outro pod, listamos o diretório, constatando que o arquivo de fato encontra-se acessível...

```
# kubectl exec -it author-7f85d48844-998vx -- /bin/sh -c 'ls /story'
chapter1
```

E que seu conteúdo é o mesmo que foi inserido originalmente.

```
# kubectl exec -it author-7f85d48844-998vx -- /bin/sh -c 'cat /story/chapter1'
Once upon a time...
```

Listando a pasta-raiz do compartilhamento NFS, fica claro que um diretório foi criado para amazenar os dados persistentes, e dentro dele encontra-se o arquivo criado pelo pod.

```
# ls /pods/
default-novel-pvc-49b725f6-9d5a-4d5a-927a-2e8044e232c3
```

```
# cat /pods/default-novel-pvc-49b725f6-9d5a-4d5a-927a-2e8044e232c3/chapter1
Once upon a time...
```

3.3) Múltiplas utilizações concorrentes de volumes

Agora, para um cenário de uso mais complexo. Imagine que vários pods precisam acessar o mesmo PV, como vimos no último passo da atividade anterior, mas que os dados desses pods devem ser individualizados.

Você pode estar se perguntando: em que cenário isso seria necessário?

Simples! Suponha, por exemplo, que queremos apontar um analisador de registros de log para um diretório em que diversos pods escrevem seus eventos. Ora, é bastante provável que esses pods escrevam seus registros em arquivos com o mesmo nome (digamos, `app.log` ou `mail.log`), então não faz nenhum sentido — e nem sequer é possível — que eles sejam criados no mesmo diretório.

Para cenários como esses, *subPaths* são ideais. Esse atributo permite que um mesmo volume seja utilizado múltiplas vezes em um único pod, ou que variáveis e expressões regulares sejam utilizada para customizar o caminho de escrita e leitura em um volume. Vamos testar esse recurso nesta atividade.

1. Crie um novo PVC, `events`, com tamanho de 100Mi. Ele será utilizado para armazenar os eventos gerados por múltiplos pods.

   A seguir, edite o deployment `author` e faça com que o diretório `/log` seja montado sob o PV criado no passo anterior. Do ponto de vista do servidor NFS, os arquivos criados por um pod devem ser armazenados dentro do diretório `/pods/${EVENT_PV}/${POD_NAME}`, sendo `${EVENT_PV}` o PersistentVolume criado automaticamente pelo PVC `events` e `${POD_NAME}` o *hostname* do pod em questão.

   Finalmente, entre em cada um dos pods do deployment e crie um arquivo de eventos no caminho `/log/app.log` com qualquer conteúdo. Verifique o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   A atividade é relativamente complexa; vamos lá. Para criar o PVC `events`, podemos utilizar o arquivo YAML do PVC `novel` como base e editar os atributos-chave via `sed`. Veja:

```
# cat novel.yaml | sed 's/novel/events/' | sed 's/200/100/' | kubectl apply -f -
persistentvolumeclaim/events created
```

A seguir, edite o deployment `author`.

```
# kubectl edit deploy author
```

Para escrever em um diretório com o nome do pod, primeiro precisamos determinal qual é ele — felizmente, isso pode ser feito através de *Downward API environment variables*, documentadas em https://kubernetes.io/docs/tasks/inject-data-application/environment-variable-expose-pod-information/ . Na seção `.spec.template.spec.containers.env`, insira o excerto abaixo.

```
env:
- name: POD_NAME
  valueFrom:
    fieldRef:
      apiVersion: v1
      fieldPath: metadata.name
```

Com a variável `${POD_NAME}` definida, podemos utilizá-la na seção `.spec.template.spec.containers.volumeMounts`, especificando o atributo `subPathExpr`. Esta funcionalidade é documentada em https://kubernetes.io/docs/concepts/storage/volumes/#using-subpath-with-expanded-environment-variables . Utilize o excerto abaixo:

```
volumeMounts:
- mountPath: /log
  name: log-dir
  subPathExpr: $(POD_NAME)
```

Finalmente, na seção `.spec.template.spec.volumes`, adicione o PVC criado anteriormente.

```
volumes:
- name: log-dir
  persistentVolumeClaim:
    claimName: events
```

Saia e salve o arquivo. Os pods do deployment serão encerrados e recriados — devido à estratégia de deployment `Recreate`. Após algum tempo, ambos deverão estar ativos:

```
# kubectl get pod -l app=author
NAME                      READY   STATUS    RESTARTS   AGE
author-5dbfb76865-p9m2z   1/1     Running   0          35s
```

```
author-5dbfb76865-qw7r5   1/1     Running   0          35s
```

Entre em cada um dos pods acima e crie o arquivo `/log/app.log`, com um conteúdo qualquer.

```
# kubectl exec -it author-5dbfb76865-p9m2z -- /bin/sh -c 'echo event1 >
/log/app.log'
```

```
# kubectl exec -it author-5dbfb76865-qw7r5 -- /bin/sh -c 'echo event2 >
/log/app.log'
```

Vamos ver o que aconteceu com o diretório compartilhado via NFS.

```
# ls /pods
default-events-pvc-2bf969eb-d07e-4a6c-b9a0-1c0c979d6632
default-novel-pvc-49b725f6-9d5a-4d5a-927a-2e8044e232c3
```

Temos a nova pasta `default-events-pvc-2bf969eb-d07e-4a6c-b9a0-1c0c979d6632`, que referencia o PVC `events`. Dentro dela, foram criadas duas pastas — com o nome dos pods pertencentes ao deployment `author`, como esperaríamos.

```
# ls -1 /pods/default-events-pvc-2bf969eb-d07e-4a6c-b9a0-1c0c979d6632/
author-5dbfb76865-p9m2z
author-5dbfb76865-qw7r5
```

Listando recursivamente esse diretório, notamos que cada um desses diretórios possui um arquivo `app.log` dentro de si.

```
# ls -R /pods/default-events-pvc-2bf969eb-d07e-4a6c-b9a0-1c0c979d6632/
/pods/default-events-pvc-2bf969eb-d07e-4a6c-b9a0-1c0c979d6632/:
author-5dbfb76865-p9m2z  author-5dbfb76865-qw7r5

/pods/default-events-pvc-2bf969eb-d07e-4a6c-b9a0-1c0c979d6632/author-5dbfb76865-
p9m2z:
app.log

/pods/default-events-pvc-2bf969eb-d07e-4a6c-b9a0-1c0c979d6632/author-5dbfb76865-
qw7r5:
app.log
```

Usando um *loop* `for` combinado com o comando `find`, é trivial imprimir o conteúdo desses arquivos na tela e constatar que seu conteúdo é o mesmo que foi inserido dentro do contexto dos pods.

```
# for file in $( find /pods/default-events-pvc-2bf969eb-d07e-4a6c-b9a0-
1c0c979d6632/ -type f -print ); do echo -e "\n$file\n$( cat $file )"; done

/pods/default-events-pvc-2bf969eb-d07e-4a6c-b9a0-1c0c979d6632/author-5dbfb76865-
qw7r5/app.log
event2

/pods/default-events-pvc-2bf969eb-d07e-4a6c-b9a0-1c0c979d6632/author-5dbfb76865-
p9m2z/app.log
event1
```

2. Finalmente, remova todos os objetos criados nesta sessão para liberar recursos do *cluster*:

```
# kubectl delete deploy,pvc,pv --all
deployment.apps "author" deleted
deployment.apps "nfs-client-provisioner" deleted
persistentvolumeclaim "events" deleted
persistentvolumeclaim "novel" deleted
persistentvolume "pvc-2bf969eb-d07e-4a6c-b9a0-1c0c979d6632" deleted
persistentvolume "pvc-49b725f6-9d5a-4d5a-927a-2e8044e232c3" deleted
```

```
# kubectl delete sc --all
storageclass.storage.k8s.io "nfs-client" deleted
```

```
# kubectl delete role leader-locking-nfs-client-provisioner ; kubectl delete
rolebindings.rbac.authorization.k8s.io leader-locking-nfs-client-provisioner
role.rbac.authorization.k8s.io "leader-locking-nfs-client-provisioner" deleted
rolebinding.rbac.authorization.k8s.io "leader-locking-nfs-client-provisioner"
deleted
```

```
# kubectl delete clusterrole nfs-client-provisioner-runner ; kubectl delete
clusterrolebinding run-nfs-client-provisioner
clusterrole.rbac.authorization.k8s.io "nfs-client-provisioner-runner" deleted
clusterrolebinding.rbac.authorization.k8s.io "run-nfs-client-provisioner" deleted
```

ENTREGA DA TAREFA

Para que seja considerada entregue você deve anexar a esta atividade no AVA uma imagem (nos formatos .png ou .jpg) do terminal com a saída do comando `ls` que mostra que dos arquivos `app.log` **diferentes** foram criados em pastas nos diretórios `/pods/*events*/author*`. Isto irá comprovar o funcionamento da utilização concorrente de volumes no *cluster*.

Utilize como referência a saída de comando mostrada na atividade 7.3.3 (a) deste

> roteiro.