Manage Non-shared Storage with Stateful Sets

Objectives

- Deploy applications that scale without sharing storage.

Application Clustering

Clustering applications, such as MySQL and Cassandra, typically require persistent storage. This storage maintains the integrity of the data and files that the application uses. When many applications require persistent storage at the same time, multi-disk provisioning might not be possible because of limited resources.

Shared storage solves this problem by allocating resources from a single device to multiple services.

File Storage

File storage solutions provide a familiar directory structure. Using file storage is ideal when applications generate or consume typical volumes of organized data. Applications that use file-based implementations are prevalent, easy to manage, and provide an affordable storage solution.

File-based solutions are a good fit for data backup, archiving, file sharing, and collaboration services because of their reliability. Most data centers provide file storage solutions, such as a *network-attached storage* (NAS) cluster, for these scenarios.

NAS is a file-based storage architecture that makes stored data accessible to networked devices. NAS gives networks a single access point for storage with built-in security, management, and fault-tolerant capabilities. Networks can run multiple data transfer protocols. Two fundamental protocols are *internet protocol (IP)* and *transmission control protocol (TCP)*.

The files that are transferred by using these protocols can be formatted with one of the following protocols:

- *Network File System* (NFS): This protocol enables remote hosts to mount file systems over a network and interact with those file systems as though they are mounted locally.
- *Server Message Block* (SMB): This protocol implements an application-layer network protocol to access resources on a server, such as file shares and shared printers.

NAS solutions can provide file-based storage to applications within the same data center. This approach applies to the following application architectures:

- Web server content
- File-sharing services

- FTP storage
- Backup archives

These applications take advantage of data reliability and the ease of file sharing that is available by using file storage. Additionally, for file storage data, the OS and file system handle the locking and caching of the files.

Although familiar and prevalent, file storage solutions are not ideal for all application scenarios. One pitfall of file storage is poor handling of large data sets or unstructured data.

Block Storage

Block storage solutions, such as Storage Area Network (SAN) and iSCSI technologies, provide access to raw block devices for application storage. These block devices function as independent storage volumes, such as the physical drives in servers, and typically require formatting and mounting for application access.

Using block storage is ideal when applications require faster access for optimizing computationally heavy data workloads. Applications that use block-level storage implementations gain efficiencies by communicating at the raw device level, instead of relying on operating system layer access.

Block-level approaches enable data distribution on blocks across the storage volume. Blocks also use basic metadata, including a unique identification number for each block of data, for quick retrieval and reassembly of blocks for reading.

SAN and iSCSI technologies provide applications with block-level volumes from network-based storage pools. Using block-level access to storage volumes is common for the following application architectures:

- SQL databases (single-node access)
- Virtual machines (multinode access)
- High-performance data access
- Server-side processing applications
- Multiple block device RAID configurations

Application storage that uses several block devices in a RAID configuration benefits from the data integrity and performance that the various arrays provide.

With Red Hat OpenShift Container Platform, you can create customized storage classes for your applications. With NAS and SAN storage technologies, OpenShift applications can use either the NFS protocol for file-based storage or a block-level protocol for block storage.

Stateful Sets

A stateful application acts according to past states or transactions, which affect the current and future states of the application. Using a stateful application simplifies recovery from failures by starting from a certain point in time.

A stateful set is the representation of a set of pods with consistent identities. These identities are defined as a network with a single stable DNS, hostname, and storage from as many volume claims as the stateful set specifies. A stateful set guarantees that a given network identity maps to the same storage identity.

Deployments represent a set of containers within a pod. Each deployment can have many active replicas, depending on the user specification. These replicas can be scaled up or down, as needed. A replica set is a native Kubernetes API object that ensures that the specified number of pod replicas are running. Deployments are used for stateless applications by default, and they can be used for stateful applications by attaching a persistent volume. All pods in a deployment use the same PVC.

In contrast with deployments, stateful set pods do not share a persistent volume. Instead, each stateful set pod has its own unique persistent volume. Pods are created without a replica set, and each replica records its own transactions. Each replica has its own identifier, which is maintained in any rescheduling. You must configure application-level clustering so that stateful set pods have the same data.

Stateful sets are the best option for applications, such as databases, that require consistent identities and non-shared persistent storage.

Headless Services for Stateful Sets

Stateful sets require a headless service to provide the stable network identity for its pods. Unlike a typical service that gets a single stable IP address (`clusterIP`) for load balancing, a headless service does not have a `clusterIP` address.

You create a headless service by explicitly setting its `clusterIP` field to `None`. For a DNS lookup on a headless service, the DNS server returns the IP addresses of all the individual pods that the service selects. This feature allows clients to connect directly to a specific pod instead of going through a load balancer.

For a StatefulSet, this behavior is critical. When you associate a stateful set with a headless service, each pod in the set gets a unique and predictable DNS entry. This DNS entry follows the format `<pod-name>.<service-name>.<namespace>.svc.cluster.local`. For example, a pod named `dbserver-0` that is governed by a headless service named `mysql-headless` can be reached reliably at the `dbserver-0.mysql-headless` FQDN.

This stable identity enables peer discovery and direct communication between pods in a clustered application. These capabilities are crucial for applications such as distributed databases, where specific pods must be addressed directly for operations such as writing or replication.

Working with Stateful Sets

With Kubernetes, you can use manifest files to specify the intended configuration of a stateful set. You can define the name of the application, labels, the image source, storage, environment variables, and more. The following snippet shows an example of a `YAML` manifest file for a stateful set named `dbserver`:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: dbserver (1)
spec:
  selector:
    matchLabels:
      app: database (2)
  replicas: 3 (3)
  serviceName: mysql (4)
  template:
    metadata:
      labels:
        app: database (5)
    spec:
      containers:
      - env: (6)
        - name: MYSQL_USER
          valueFrom:
            secretKeyRef:
              key: user
              name: sakila-cred
        image: registry.ocp4.example.com:8443/redhattraining/mysql-app:v1 (7)
        name: database (8)
        ports: (9)
        - containerPort: 3306
          name: database
        volumeMounts: (10)
        - mountPath: /var/lib/mysql
          name: data
      terminationGracePeriodSeconds: 10
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOncePod" ] (11)
      storageClassName: "lvms-vg1" (12)
      resources:
        requests:
          storage: 1Gi (13)
```

(1) Name of the stateful set.

(2) (5) Application labels.

(3) Number of replicas.

(4) Name of the service that governs the stateful set. This service must exist before the stateful set.

(6) Environment variables, which can be defined explicitly or by using a secret object.

(7) Image source.

8. Container name.

9. Container ports.

10. Mount path information for the persistent volumes for each replica. Each persistent volume has the same configuration.

11. The access mode of the persistent volume. The valid values are `ReadWriteOncePod`, `ReadWriteOnce`, `ReadWriteMany`, and `ReadOnlyMany`. The `ReadWriteOncePod` mode is recommended for production use.

12. The storage class that the persistent volume uses.

13. Size of the persistent volume.

> **NOTE**
>
> Stateful sets can be created only by using manifest files.
> The `oc` and `kubectl` CLIs do not have commands to create stateful sets imperatively.

The following snippet shows an example `YAML` manifest for a headless service named `mysql` for the `dbserver` stateful set:

```
apiVersion: v1
kind: Service
metadata:
  name: mysql 1
  labels:
    app: database
spec:
  ports:
  - port: 3306
    name: mysql
  clusterIP: None 2
  selector:
    app: database 3
```

1. The name of the headless service.

2. Set the `clusterIP` field to `None` to make the service headless.

3. The service selector must match the labels of the pods that the stateful set created.

You create the `mysql` headless service by using the `oc create` command:

```
[user@host ~]$ oc create -f headless-service.yml
```

You can verify the creation of the service by running the following command:

```
[user@host ~]$ oc get svc
NAME            TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
service/mysql   ClusterIP   None         <none>        3306/TCP   5s
```

You can create the `dbserver` stateful set by using the `oc create` command:

```
[user@host ~]$ oc create -f statefulset-dbserver.yml
```

You can verify the creation of the stateful set by running the following command. In the following example, the stateful set has three replica pods, and all three pods are ready.

```
[user@host ~]$ oc get statefulset
NAME       READY   AGE
dbserver   3/3     6s
```

The stateful set controller assigns each pod a unique and stable name in the `<statefulset-name>-<ordinal-index>` format. The controller starts pods one at a time, sorted by their ordinal index, and it waits until each pod reports being ready before starting the next one.

```
[user@host ~]$ oc get pods
NAME         READY   STATUS    RESTARTS   AGE
dbserver-0   1/1     Running   0          85s
dbserver-1   1/1     Running   0          82s
dbserver-2   1/1     Running   0          79s
```

You can get the service endpoints, which are the pods' IP addresses by running the following command:

```
[user@host ~]$ oc get endpoints
NAME       ENDPOINTS                                         AGE
mysql      10.8.0.45:3306,10.8.0.54:3306,10.8.0.61:3306   2m
```

You resolve the pods' IP addresses by using the headless service; for example, to resolve the IP address for the `dbserver-0` pod, you use the `dbserver-0.mysql` hostname.

```
[user@host ~]$ oc rsh dbserver-0 curl -v \
  telnet://dbserver-0.mysql:3306
Rebuilt URL to: telnet://dbserver-0.mysql:3306/
Trying 10.8.0.45...
TCP_NODELAY set
Connected to dbserver-0.mysql (10.8.0.45) port 3306 (#0)
```

The stateful set creates three PVCs, one PVC for each replica pod. Each PVC has a unique and stable name in the `data-<statefulset-name>-<ordinal-index>` format.

```
[user@host ~]$ oc get pvc
NAME              STATUS   VOLUME      CAPACITY ACCESS MODES ...
data-dbserver-0   Bound    pvc-c28...  1Gi      RWO          ...
data-dbserver-1   Bound    pvc-ddb...  1Gi      RWO          ...
data-dbserver-2   Bound    pvc-830...  1Gi      RWO          ...
```

Each pod attaches its associated PVC. You can check the PVC that is attached to each replica pod in the stateful set by running the following commands:

```
[user@host ~]$ oc describe pod dbserver-0
...output omitted...
Volumes:
  data:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the sa
me namespace)
    ClaimName:  data-dbserver-0
...output omitted...
```

```
[user@host ~]$ oc describe pod dbserver-1
...output omitted...
Volumes:
  data:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the sa
me namespace)
    ClaimName:  data-dbserver-1
...output omitted...
```

```
[user@host ~]$ oc describe pod dbserver-2
...output omitted...
Volumes:
  data:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the sa
me namespace)
    ClaimName:  data-dbserver-2
...output omitted...
```

The data is replicated between the three pods at the application level. For example, in the `dbserver` stateful set, you can configure the first pod as the primary, and the other two pods as replicas. The primary pod handles read and write requests, and the replica pods sync with the primary pod for data replication.

NOTE

Configuring a replicated stateful set application is outside the scope of this course.

You can update the number of replicas of the stateful set by using the `oc scale` command. The following command adds one more pod to the stateful set. A fourth pod is created if the third pod is up and running.

```
[user@host ~]$ oc scale statefulset/dbserver --replicas 4
NAME          READY    STATUS     RESTARTS  ...
dbserver-0    4/4      Running    0         ...
```

To delete the stateful set, use the `delete statefulset` command:

```
[user@host ~]$ oc delete statefulset dbserver
statefulset.apps "dbserver" deleted
```

The PVCs are not deleted after the execution of the `oc delete statefulset` command:

```
[user@host ~]$ oc get pvc
NAME               STATUS   VOLUME       CAPACITY ACCESS MODES ...
data-dbserver-0    Bound    pvc-c28...  1Gi      RWO          ...
data-dbserver-1    Bound    pvc-ddb...  1Gi      RWO          ...
data-dbserver-2    Bound    pvc-830...  1Gi      RWO          ...
```

You can create a stateful set from the web console. Go to the **Workloads** → **StatefulSets** menu. Click **Create StatefulSet** and customize the YAML manifest.

REFERENCES

Kubernetes Documentation - StatefulSets

Kubernetes Documentation - Run a Replicated Stateful Application

For more information about network-attached storage, refer to *What Is Network-attached Storage?* at https://www.redhat.com/en/topics/data-storage/network-attached-storage#how-does-it-work