Chapter 5. Manage Storage for Application Configuration and Data

Abstract

Goal	Externalize application configurations in Kubernetes resources and provision storage volumes for persistent data files.
Sections	• Externalize the Configuration of Applications (and Guided Exercise) • Provision Persistent Data Volumes (and Guided Exercise) • Selecting a Storage Class for an Application (and Guided Exercise) • Manage Non-shared Storage with Stateful Sets (and Guided Exercise)
Lab	• Manage Storage for Application Configuration and Data

Externalize the Configuration of Applications

Objectives

- Configure applications by using Kubernetes secrets and configuration maps to initialize environment variables and to provide text and binary configuration files.

Configuring Kubernetes Applications

When an application is run in Kubernetes with a pre-existing image, the application uses the default configuration. This action is valid for testing purposes. However, for production environments, you might need to customize your applications before deploying them.

With Kubernetes, you can use manifests in `JSON` and `YAML` formats to specify the intended configuration for each application. 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 of a deployment:

```
apiVersion: apps/v1 (1)
kind: Deployment (2)
metadata: (3)
  name: hello-deployment
spec: (4)
  replicas: 1
  selector:
    matchLabels:
      app: hello-deployment
  template:
    metadata:
      labels:
        app: hello-deployment
    spec: (5)
      containers:
      - env: (6)
        - name: ENV_VARIABLE_1
          valueFrom:
            secretKeyRef:
              key: hello
              name: world
        image: quay.io/hello-image:latest
```

1. API version of the resource.
2. `Deployment` resource type.
3. In this section, you specify the metadata of your application, such as the name.
4. You can define the general configuration of the resource that is applied to the deployment, such as the number of replicas (pods), the selector label, and the template data.
5. In this section, you specify the configuration for your application, such as the image name, the container name, ports, environment variables, and more.
6. You can define the environment variables to configure your application needs.

Sometimes, your application requires using a combination of files. For example, at the time of creation, a database deployment must have preloaded databases and data. You most commonly configure applications by using environment variables, external files, or command-line arguments. This process of configuration externalization ensures that the application is portable across environments when the container image, external files, and environment variables are available in the environment where the application runs.

Kubernetes provides a mechanism to externalize the configuration of your applications by using configuration maps and secrets.

You can use configuration maps to inject containers with configuration data. The *ConfigMap* (configuration map) namespaced objects provide ways to inject configuration data into containers, which helps to maintain platform independence of the containers. These objects can store fine-grained information, such as individual properties, or coarse-grained information, such as entire configuration files or JSON blobs (JSON sections). The information in configuration maps does not require protection.

The following listing shows an example of a configuration map:

```
apiVersion: v1
kind: ConfigMap (1)
metadata:
  name: example-configmap
  namespace: my-app
data: (2)
  example.property.1: hello
  example.property.2: world
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
binaryData: (3)
  bar: R0lGODlhAgACAIABAO4AAP///yH5BAEHAAEALAAAAAACAAIAAAIDRAIFADs=
```

1. `ConfigMap` resource type.
2. Contains the configuration data.
3. Points to an encoded file in base64 that contains non-UTF-8 data, for example, a binary Java keystore file. Place a key followed by the encoded file.

Applications often require access to sensitive information. For example, a back-end web application requires access to database credentials to query a database. Kubernetes and OpenShift use secrets to hold sensitive information. For example, you can use secrets to store the following types of sensitive information:

- Passwords
- Sensitive configuration files
- Credentials to an external resource, such as an SSH key or OAuth token

The following listing shows an example of a secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: example-secret
  namespace: my-app
type: Opaque (1)
data: (2)
  username: bXl1c2VyCg==
  password: bXlQQDU1Cg==
stringData: (3)
  hostname: myapp.mydomain.com
  secret.properties: |
    property1=valueA
    property2=valueB
```

(1) Specifies the type of secret.

(2) Specifies the encoded string and data.

(3) Specifies the decoded string and data.

A secret is a namespaced object and it can store any type of data. Data in a secret is Base64-encoded, and is not stored in plain text. Secret data is not encrypted; you can decode the secret from Base64 format to access the original data. The following example shows the decoded values for the `username` and `password` objects from the `example-secret` secret:

```
[user@host ~]$ echo bXl1c2VyCg== | base64 --decode
myuser
[user@host ~]$ echo bXlQQDU1Cg== | base64 --decode
myP@55
```

Kubernetes and OpenShift support the following types of secrets:

- Opaque secrets: An opaque secret store key and value pairs that contain arbitrary values, and are not validated to conform to any convention for key names or values.
- Service account tokens: Store a token credential for applications that authenticate to the Kubernetes API.
- Basic authentication secrets: Store the needed credentials for basic authentication. The data parameter of the secret object must contain the user and the password keys that are encoded in the Base64 format.
- SSH keys: Store data that is used for SSH authentication.
- TLS certificates: Store a certificate and a key that are used for TLS.
- Docker configuration secrets: Store the credentials for accessing a container image registry.

When you store information in a specific secret resource type, Kubernetes validates that the data conforms to the type of secret.

> **NOTE**
>
> By default, configuration maps and secrets are not encrypted. To encrypt your secret data at rest, you must encrypt the Etcd database. When enabled, Etcd encrypts the following resources: secrets, configuration maps, routes, OAuth access tokens, and OAuth authorization tokens. Encrypting the Etcd database is outside the scope of the course.

Creating Secrets

If a pod requires access to sensitive information, then create a secret for the information before you deploy the pod. Both the `oc` and `kubectl` command-line tools provide the `create secret` command. Use one of the following commands to create a secret:

- Create a generic secret that contains key-value pairs from literal values that are typed on the command line:

```
[user@host ~]$ oc create secret generic secret_name \
--from-literal key1=secret1 \
--from-literal key2=secret2
```

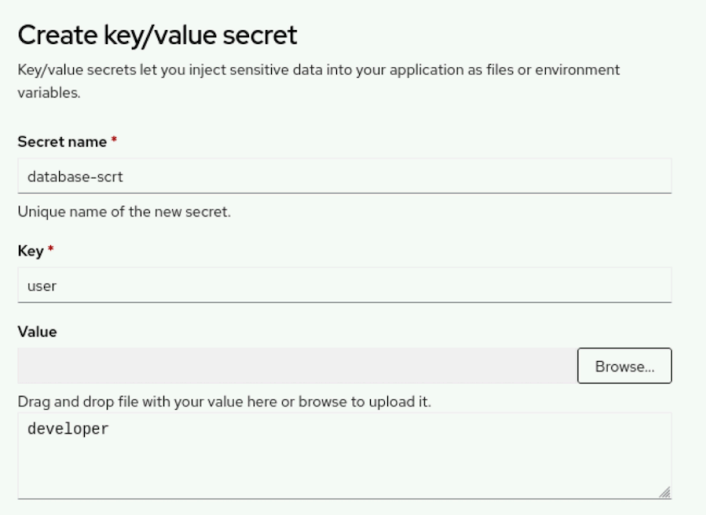
- Create a generic secret by using key names that are specified on the command line and values from files:

```
[user@host ~]$ kubectl create secret generic ssh-keys \
--from-file id_rsa=/path-to/id_rsa \
--from-file id_rsa.pub=/path-to/id_rsa.pub
```

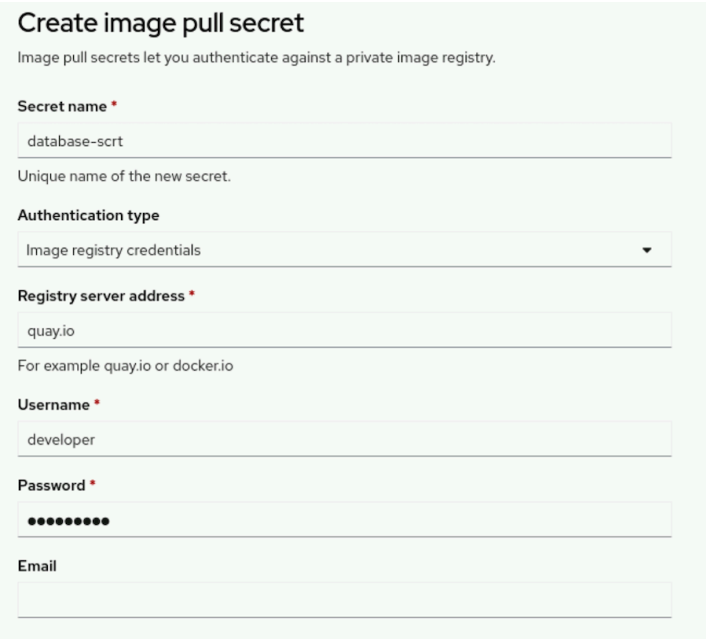
- Create a TLS secret that specifies a certificate and the associated key:

```
[user@host ~]$ oc create secret tls secret-tls \
--cert /path-to-certificate --key /path-to-key
```

To create an opaque secret from the web console, click the **Workloads** → **Secrets** menu. Click **Create** and select **Key/value secret**. Complete the form with the key name, and specify the value by writing it in the following section, or by extracting it from a file.



To create a secret from the web console that stores the credentials for accessing a container image registry, click the **Workloads** → **Secrets** menu. Click **Create** and select **Image pull secret**. Complete the form or upload a configuration file with the secret name, select the authentication type, and add the registry server address, the username, password, and email credentials.



Creating Configuration Maps

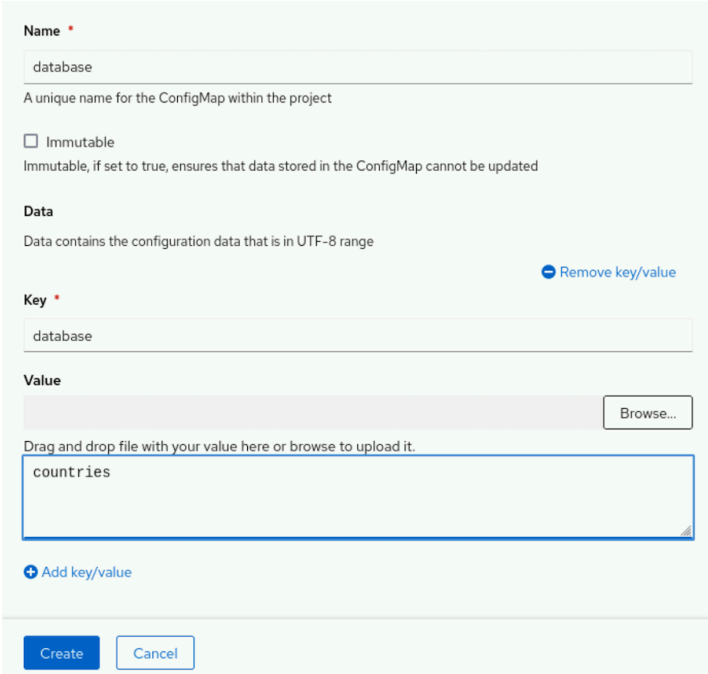
The syntax for creating a configuration map and for creating a secret closely match. You can enter key-value pairs on the command line, or use the content of a file as the value of a specified key. You can use either the `oc` or `kubectl` command-line tools to create a configuration map. The following command shows how to create a configuration map:

```
[user@host ~]$ kubectl create configmap my-config \
--from-literal key1=config1 --from-literal key2=config2
```

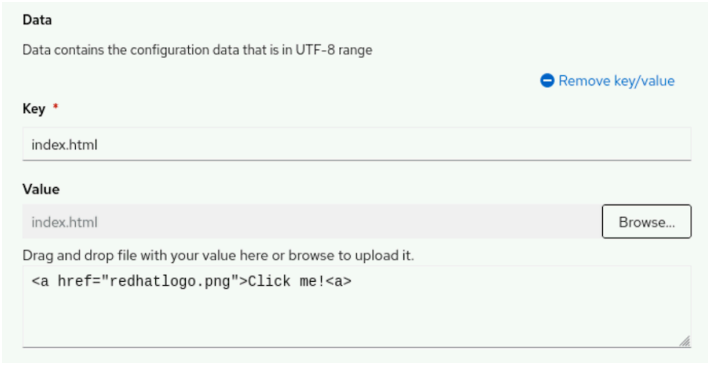
You can also use the `cm` shortname to create a configuration map.

```
[user@host ~]$ oc create cm my-config \
--from-literal key1=config1 --from-literal key2=config2
```

To create a configuration map from the web console, click the **Workloads** → **ConfigMaps** menu. Click **Create ConfigMap** and complete the configuration map by using the form view or the YAML view.



You can use files on each key that you add by clicking **Browse** beside the **Value** field. The **Key** field must be the name of the added file in the **Value** field.



> **NOTE**
>
> Use a binary data key instead of a data key if the file uses the binary format, such as a PNG file.

Using Configuration Maps and Secrets to Initialize Environment Variables

You can use configuration maps to populate individual environment variables that configure your application. Unlike secrets, the information in configuration maps does not require protection. The following listing shows an initialization example of environment variables:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-map-example
  namespace: example-app (1)
data:
  database.name: sakila (2)
  database.user: redhat (3)
```

(1) The project where the configuration map resides. `ConfigMap` objects can be referenced only by pods in the same project.

(2) Initializes the `database.name` variable to the `sakila` value.

3 Initializes the `database.user` variable to the `redhat` value.

You can then use the configuration map to populate environment variables for your application. The following example shows a pod resource that populates specific environment variables by using a configuration map.

```
apiVersion: v1
kind: Pod
metadata:
  name: config-map-example-pod
  namespace: example-app
spec:
  containers:
    - name: example-container
      image: registry.example.com/mysql-80:1-237
      command: [ "/bin/sh", "-c", "env" ]
      env: 1
        - name: MYSQL_DATABASE 2
          valueFrom:
            configMapKeyRef:
              name: config-map-example 3
              key: database.name 4
        - name: MYSQL_USER
          valueFrom:
            configMapKeyRef:
              name: config-map-example 5
              key: database.user 6
              optional: true 7
```

1 The attribute to specify environment variables for the pod.

2 The name of a pod environment variable where you are populating a key's value.

3 5 Name of the `ConfigMap` object to pull the environment variables from.

4 6 The environment variable to pull from the `ConfigMap` object.

7 Sets the environment variable as optional. The pod is started even if the specified `ConfigMap` object and keys do not exist.

The following example shows a pod resource that injects all environment variables from a configuration map:

```
apiVersion: v1
kind: Pod
metadata:
  name: config-map-example-pod2
  namespace: example-app
spec:
  containers:
    - name: example-container
      image: registry.example.com/mysql-80:1-237
      command: [ "/bin/sh", "-c", "env" ]
  envFrom: 1
    - configMapRef:
        name: config-map-example 2
  restartPolicy: Never
```

1 The attribute to pull all environment variables from a `ConfigMap` object.

2 The name of the `ConfigMap` object to pull environment variables from.

You can use secrets with other Kubernetes resources such as pods, deployments, builds, and more. You can specify secret keys or volumes with a mount path to store your secrets. The following snippet shows an example of a pod that populates environment variables with data from the `test-secret` Kubernetes secret:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "export" ]
      env: (1)
        - name: TEST_SECRET_USERNAME_ENV_VAR
          valueFrom: (2)
            secretKeyRef: (3)
              name: test-secret (4)
              key: username (5)
```

(1) Specifies the environment variables for the pod.

(2) Indicates the source of the environment variables.

(3) The `secretKeyRef` source object of the environment variables.

(4) Name of the secret, which must exist.

(5) The key that is extracted from the secret is the username for authentication.

In contrast with configuration maps, the values in secrets are always encoded (not encrypted), and their access is restricted to fewer authorized users.

Using Secrets and Configuration Maps as Volumes

To expose a secret to a pod, you must first create the secret in the same namespace, or project, as the pod. In the secret, assign each piece of sensitive data to a key. After creation, the secret contains key-value pairs.

The following command creates a generic secret that contains key-value pairs from literal values that are typed on the command line: `user` with the `demo-user` value, and `root_password` with the `zT1kTgk` value.

```
[user@host ~]$ oc create secret generic demo-secret \
--from-literal user=demo-user \
--from-literal root_password=zT1KTgk
```

You can also create a generic secret by specifying key names on the command line and values from files:

```
[user@host ~]$ oc create secret generic demo-secret \
--from-file user=/tmp/demo/user \
--from-file root_password=/tmp/demo/root_password
```

You can mount a secret to a directory within a pod. Kubernetes creates a file for each key in the secret that uses the name of the key. The content of each file is the decoded value of the secret. The following command shows how to mount secrets in a pod:

```
[user@host ~]$ oc set volume deployment/demo \ (1)
--add --type secret \ (2)
--secret-name demo-secret \ (3)
--mount-path /app-secrets (4)
```

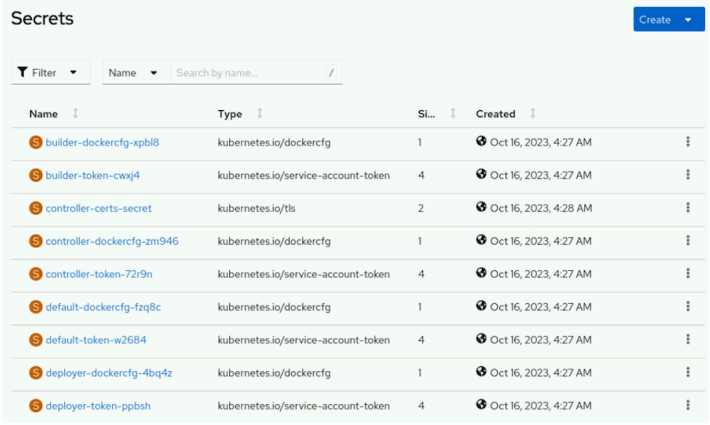
(1) Modify the volume configuration in the `demo` deployment.

(2) Add a new volume from a secret.

(3) Use the `demo-secret` secret.

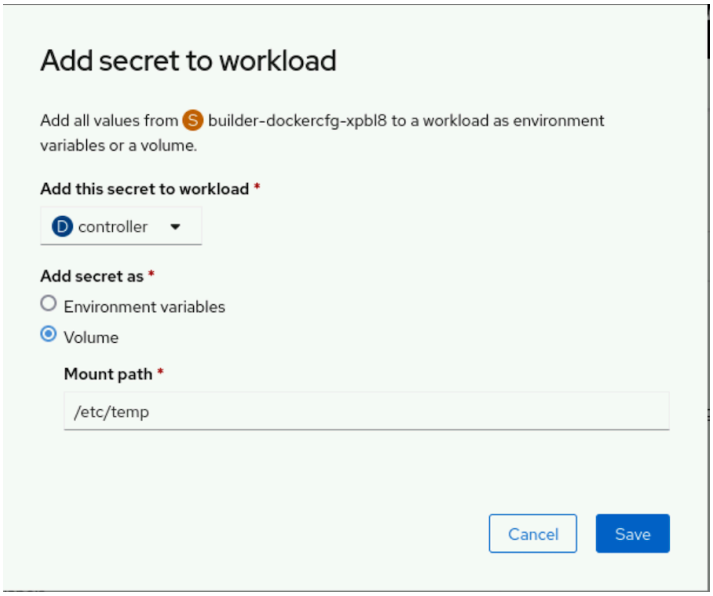
(4) Make the secret data available in the `/app-secrets` directory in the pod. The content of the `/app-secrets/user` file is `demo-user`. The content of the `/app-secrets/root_password` file is `zT1KTgk`.

To assign a secret as a volume to a deployment from the web console, list the available secrets from the **Workloads** → **Secrets** menu.



Select a secret and click **Add Secret to workload**.

Select the workload, choose the **Volume** option, and define the mount path for the secret.



Similar to secrets, you must first create a configuration map before a pod can consume it. The configuration map must exist in the same namespace, or project, as the pod. The following command shows how to create a configuration map from an external configuration file:

```
[user@host ~]$ oc create configmap demo-map \
--from-file=config-files/httpd.conf
```

You can similarly add a configuration map as a volume by using the following command:

```
[user@host ~]$ oc set volume deployment/demo \
--add --type configmap \
--configmap-name demo-map \
--mount-path /app-secrets
```

To confirm that the volume is attached to the deployment, use the following command:

```
[user@host ~]$ oc set volume deployment/demo
 demo
  configMap/demo-map as volume-du9in
    mounted at /app-secrets
```

You can also use the `oc set env` command to set application environment variables from either secrets or configuration maps. In some cases, you can modify the names of the keys to match the names of environment variables by using the `--prefix` option. In the following example, the `user` key from the `demo-secret` secret sets the `MYSQL_USER` environment variable, and the `root_password` key from the `demo-secret` secret sets the `MYSQL_ROOT_PASSWORD` environment variable. If the key name from the secret is lowercase, then the corresponding environment variable is converted to uppercase to match the pattern that the `--prefix` option defines.

```
[user@host ~]$ oc set env deployment/demo \
--from secret/demo-secret --prefix MYSQL_
```

> **NOTE**
>
> You cannot assign configuration maps by using the web console.

Updating Secrets and Configuration Maps

Secrets and configuration maps occasionally require updates. OpenShift provides the `oc extract` command to ensure that you have the latest data. You can save the data to a specific directory by using the `--to` option. Each key in the secret or configuration map creates a file with the same name as the key. The content of each file is the value of the associated key. If you run the `oc extract` command more than one time, then you must use the `--confirm` option to overwrite the existing files. You can also use the `--confirm` option to create the target directory for the extracted content.

```
[user@host ~]$ oc extract secret/demo-secrets -n demo \
--to /tmp/demo --confirm
[user@host ~]$ ls /tmp/demo/
user  root_password
[user@host ~]$ cat /tmp/demo/root_password
zT1KTgk
[user@host ~]$ echo k8qhcw3m0 > /tmp/demo/root_password
```

After updating the locally saved files, use the `oc set data` command to update the secret or configuration map. For each key that requires an update, specify the name of a key and the associated value. If a file contains the value, then use the `--from-file` option.

```
[user@host ~]$ oc set data secret/demo-secrets -n demo \
--from-file /tmp/demo/root_password
```

You must restart pods that use environment variables for the pods to read the updated secret or configuration map. Pods that use a volume mount to reference secrets or configuration maps receive the updates without a restart by using an eventually consistent approach. By default, the `kubelet` agent watches for changes to the keys and values that are used in volumes for pods on the node. The `kubelet` agent detects changes and propagates the changes to the pods to keep volume data consistent. Despite the automatic updates that Kubernetes provides, a restart of the pod is still required if the software reads configuration data only at startup time.

Deleting Secrets and Configuration Maps

Similar to other Kubernetes resources, you can use the `delete` command to delete secrets and configuration maps that are no longer needed or in use.

```
[user@host ~]$ kubectl delete secret/demo-secrets -n demo
```

```
[user@host ~]$ oc delete configmap/demo-map -n demo
```

> **REFERENCES**
>
> For more information, refer to the *Using Config Maps with Applications* chapter in the Red Hat OpenShift Container Platform 4.18 *Building Applications* documentation at https://docs.redhat.com/en/documentation/openshift_container_platform/4.18/html-single/building_applications/index#config-maps
>
> For more information, refer to *Providing Sensitive Data to Pods* in the Red Hat OpenShift Container Platform 4.18 *Working with Pods* documentation at https://docs.redhat.com/en/documentation/openshift_container_platform/4.18/html-single/nodes/index#nodes-pods-secrets

For more information, refer to the *Encrypting Etcd Data* chapter in the Red Hat OpenShift Container Platform 4.18 *Security and Compliance* documentation at https://docs.redhat.com/en/documentation/openshift_container_platform/4.18/html-single/security_and_compliance/index#about-etcd_encrypting-etcd