Configure application

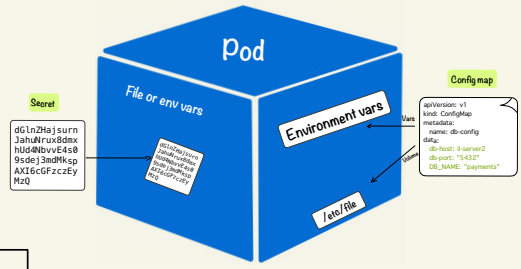
Kubernetes provides several ways to configure applications, including using ConfigMaps, environment variables, and Secrets.

ConfigMaps are Kubernetes resources that can be used to store configuration data as key-value pairs. You can create a ConfigMap with the desired configuration data, and then reference it in your Deployment or Pod specification using the 'configMapKeyRef' field or mount it directly to the pod.

You can create a ConfigMap using the 'kubectl create configmap' command, or by defining a YAML file.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-configmap
data:
  DB_HOST: "mydbhost"
  DB_PORT: "5432"
  DB_NAME: "mydb"
```

```
K get cm
k describe cm db-config
```



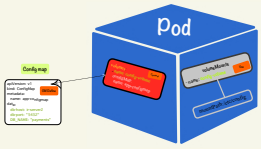
How you can use a ConfigMap to store configuration data

Environment Variables

You can store environment variables in a ConfigMap and use them to configure your application

Configuration Files

You can store configuration files in a ConfigMap and mount them as volumes in your container



we define a volume named 'config-volume' that maps to the 'app-configmap' ConfigMap using the 'configMap' field. We then mount this volume into the container using the 'volumeMounts' field, which specifies that the volume should be mounted at the path '/etc/config' in the container

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: myimage
    volumeMounts:
    - name: config-volume
      mountPath:/etc/config
  volumes:
  - name: config-volume
    configMap:
      name: app-configmap
```

Now, any configuration files that are stored in the 'app-configmap' ConfigMap can be accessed by the application running in the container at the '/etc/config' path

Command-Line Arguments

You can store command-line arguments in a ConfigMap and use them to configure your application

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: myimage
    command: ["/bin/myapp"]
    args: ["--config", "/etc/myapp/config.yaml"]
    volumeMounts:
    - name: config-volume
      mountPath: /etc/myapp/
  volumes:
  - name: config-volume
    configMap:
      name: my-configmap
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  config.yaml: |
    setting1: value1
    setting2: value2
```

Environment variables can be used to pass configuration information to the container, such as database connection strings or API keys. You can define environment variables in the Deployment or Pod specification using the 'env' field, the 'envFrom' field, and the 'valueFrom' field

The 'env' field is used to define individual environment variables for a container. You can define the name and value of each environment variable using the 'name' and 'value' fields, respectively

```yaml
apiVersion: apps/v1
kind: Deployment
...
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: web
        image: myapp:latest
        ports:
        - containerPort: 80
        env:
        - name: DB_HOST
          value: "il-server2"
        - name: DB_PORT
          value: "5432"
        - name: DB_NAME
          value: "payments"
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  DB_HOST: "il-server2"
  DB_PORT: "5432"
  DB_NAME: "payments"
```

The 'envFrom' field is used to define environment variables for a container based on a ConfigMap or Secret. You can specify the name of the ConfigMap or Secret using the 'configMapRef' or 'secretRef' fields, respectively

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: web
        image: myapp:latest
        ports:
        - containerPort: 80
        envFrom:
        - configMapRef:
            name: db-config
        -secretRef:
            name: db-secrets
```

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: db-secrets
type: Opaque
data:
  DB_USER: dXNlcg==
  DB_PASSWORD: cGFzc3dvcmQ=
```

The 'valueFrom' field is used to define environment variables for a container based on a field in another resource, such as a ConfigMap or Secret. You can specify the name of the resource and the field using the 'configMapKeyRef' or 'secretKeyRef' fields, respectively

```yaml
apiVersion: apps/v1
kind: Deployment
...
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: web
        image: myapp:latest
        ports:
        - containerPort: 80
        env:
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: db-config
              key: db-host
        - name: DB_PORT
          valueFrom:
            configMapKeyRef:
              name: db-config
              key: db-port
```

The 'name' field in the 'configMapKeyRef' field specifies the name of the ConfigMap, and the 'key' field specifies the name of the key within the ConfigMap to use as the value for the environment variable

Because we only want to add specific variables from a ConfigMap to a container, we use the 'valueFrom' field

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  db-host: il-server2
  db-port: "5432"
  DB_NAME: "payments"
```

Secrets are similar to ConfigMaps, but are used to store sensitive information such as passwords, tokens or API keys. You can create a Secret with the desired sensitive information, and then reference it in your Deployment or Pod specification using the 'secretKeyRef' field.

you can also create a secret by running the kubectl create secret command

```
kubectl create secret generic db-secret --from-literal=username=myuser --from-literal=password=mypassword
```

This command will create a secret named db-secret with two key-value pairs: username and password

```
arye@dev: kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-abc12   kubernetes.io/service-account-token   3      4d
db-secret             Opaque                                2      2h
```

The DATA column shows the number of data items (key-value pairs) in each secret

To use a secret in a pod, you can mount it as a volume or use it as an environment variable

```
spec:
  containers:
  - name: my-container
    image: my-image
    volumeMounts:
    - name: secret-volume
      mountPath: /etc/myapp/secret
      readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: db-secret
```

To update a secret, you can use the kubectl edit secret command or edit the yaml file directly and apply the changes

```
kubectl edit secret db-secret
```

```
arye@dev: kubectl describe secret db-secret
Name:         db-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  16 bytes
username:  6 bytes
```

By default, the values of the key-value pairs in a secret are base64-encoded to provide a basic level of obfuscation. To decode the values, you can use the base64 command

```
kubectl get secret db-secret -o jsonpath='{.data.password}' | base64 --decode
```

There are several types of secrets in Kubernetes, including:

1. Opaque: This is the default secret type in Kubernetes. It can be used to store any arbitrary data and is encoded in base64.
2. TLS: This type of secret is used to store TLS certificates and keys. It contains two keys: tls.crt and tls.key.
3. Docker-registry: This type of secret is used to authenticate with a Docker registry. It contains the username and password for the registry.
4. SSH: This type of secret is used to store SSH keys. It contains the private key and the public key.
5. Service account: This type of secret is automatically created by Kubernetes when a service account is created. It contains a token that can be used to authenticate the service account.

Warning

Kubernetes Secrets use base64 encoding to obfuscate the sensitive data, it is important to note that base64 encoding is not a form of encryption and can be easily decoded

two solutions to solve this problem

Using external encryption tools or key management systems to secure sensitive data before storing it in Kubernetes Secrets can enhance security.(HashiCorp Vault, Azure Key Vault, and AWS Key Management Service)

you can use access controls to limit who can access the sensitive data. K8s provides various mechanisms for controlling access, such as RBAC and network policies, that can be used to limit access to sensitive data to only authorized users and applications

Application Lifecycle Management

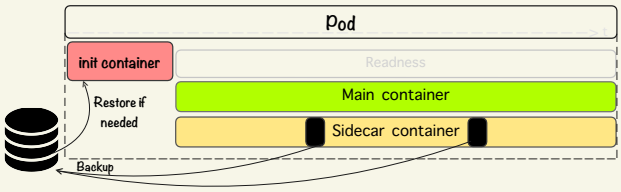
initContainer

An init container is a special type of container that runs before the main container(s) in a pod. The purpose of an init container is to perform some initialization or setup tasks that are required before the main container(s) can start running. Init containers are defined in the same YAML file as the pod specification, alongside the main container(s). They can be used to perform tasks such as setting up a database schema, downloading necessary files, or waiting for a specific service to become available

Init containers have their own lifecycle, and they are considered successful if they complete their tasks without error. If an init container fails, Kubernetes will attempt to restart it until it succeeds, which ensures that the main container(s) in the pod are not started until the initialization tasks are complete

Example 1

```
apiVersion: v1
kind: Pod
metadata:
  name: my-webapp-pod
spec:
  initContainers:
  - name: redis-setup
    image: redis:latest
    command: ["sh", "-c"]
    args:
    - |
      redis-cli ping || exit 1
      redis-cli config set maxmemory 1gb
      redis-cli config set maxmemory-policy allkeys-lru
      redis-cli config set save ""
  containers:
  - name: webapp
    image: my-webapp-image
    ports:
    - containerPort: 80
    env:
    - name: REDIS_HOST
      value: redis-service
    - name: REDIS_PORT
      value: "6379"
```



The init container uses the Redis image and runs a shell command that performs the following tasks:

- Check if the Redis server is running by pinging it
- Set the maximum memory limit to 1 gigabyte
- Set the eviction policy to "allkeys-lru"
- Disable automatic snapshots by setting the save policy to an empty string

After completing its tasks, the init container exits and is terminated. The main container then starts running and serves the web application for the duration of the Pod's lifecycle

Example 2

```
apiVersion: v1
kind: Pod
metadata:
  name: mysql-db
spec:
  containers:
  - name: mysql
    image: mysql:5.7
    env:
      - name: MYSQL_ROOT_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-secrets
            key: password
  initContainers:
  - name: migrate-db
    image: mysql:5.7
    command: ['sh', '-c', 'mysql -h ${DB_HOST} -u root -p${DB_PASSWORD} ${DB_NAME} < /migrations/migrate.sql']
    env:
      - name: DB_HOST
        value: 127.0.0.1
      - name: DB_NAME
        value: mydb
      - name: DB_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-secrets
            key: password
    volumeMounts:
      - name:migrations
        mountPath: /migrations
  volumes:
  - name: migrations
    configMap:
      name: db-migrations
```

When the Pod is started, the migrate-db container runs first and performs the database migration. Once the migration is complete, the mysql-db container starts and runs the application, which now uses the migrated database

Pod lifecycle

Here are the key phases in the lifecycle of a Pod in Kubernetes:

Once a pod's containers are scheduled to run on a node, the pod enters the running phase

When a pod is created, it enters the pending phase. During this phase, the Kubernetes scheduler assigns the pod to a node and the container images are pulled from the container registry. The pod remains in the pending phase until all of its containers are ready and scheduled to run on a node

When a container in a pod completes its task successfully, the container enters the Succeeded phase and the pod is considered to have completed its task

When a container in a pod fails or crashes, the container enters the Failed phase