Understanding ConfigMaps

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_COLOR: blue
  APP_MODE: prod
```

ConfigMaps are Kubernetes objects used to store non-confidential data in key-value pairs. They are designed to decouple environment-specific configurations from container images, making applications more portable.

ConfigMaps are useful for storing data like:

- **Application settings (log levels, feature flags, tuning parameters, etc.)**
- **Non-sensitive environment variables**
- **Command-line arguments**
- **Configuration files (XML, JSON, YAML, properties, etc.)**

Decoupling configuration from pods: By externalizing configuration into ConfigMaps, you can change an application ‘s configuration without rebuilding the container image.

Reducing image size: Separating config from your application images makes the images smaller and more focused.

Enabling configuration reuse: ConfigMaps can be shared across multiple Pods, enabling configuration reuse and standardization.

Simplifying Deployment: Using ConfigMaps, the same container image can be deployed across multiple environments (dev, test, prod) with different configurations.

Understanding Secrets

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secrets
type: Opaque
data:
  db-password: cGFzc3dvcmQ=
  license-key: QVNERiMxMjM0NTY3ODk=
```

Secrets are similar to ConfigMaps but are specifically designed for confidential data like passwords, tokens, certificates, and keys. They provide a more secure mechanism for storing sensitive information compared to including it in Pod manifests or container images.

When creating a Secret, Kubernetes stores the database64-encoded in the etcd backend. Secrets are not encrypted by default but can be encrypted at rest. Like ConfigMaps, Secrets can be exposed to Pods as files using a volume mount or as environment variables.

Etcd Storage: By default, Secret data is stored unencrypted in etcd. Anyone with access to etcd can see Secret data.

Base64 Encoding: Secret values are only base64-encoded, not encrypted. Base64 encoding is trivial to decode.

API Server Access: Anyone with API access can retrieve or modify Secrets. RBAC policies should be used to limit access.

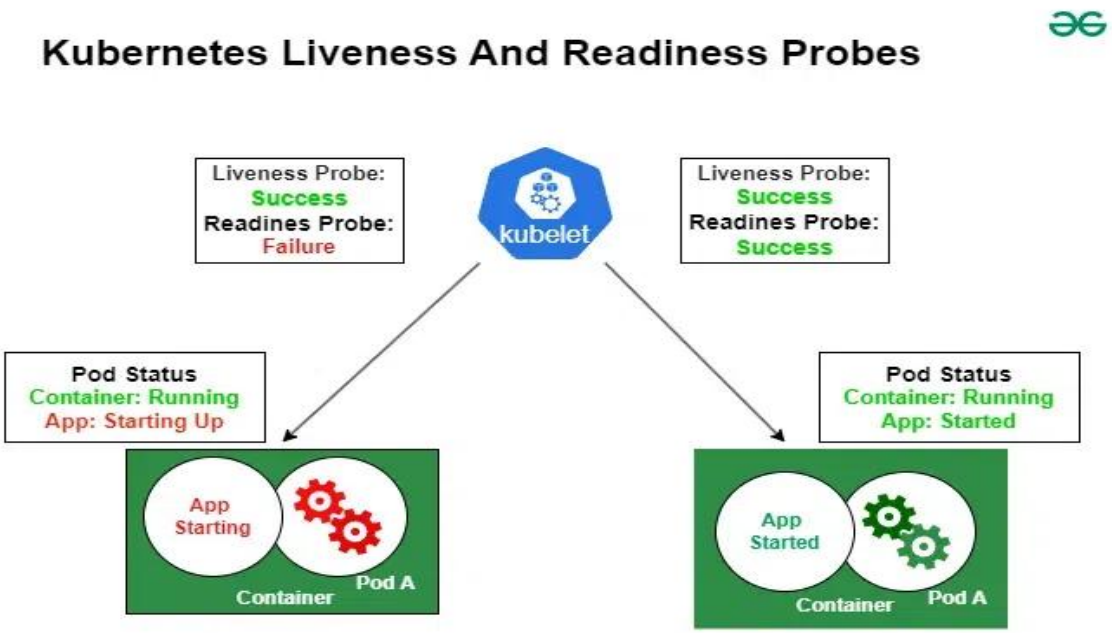
Application Security: Applications need to properly secure Secret data after reading it. Accidentally logging or exposing Secrets is a common security risk.

Aspect	ConfigMaps	Secrets
Purpose	Non-sensitive configuration data	Sensitive data (credentials, keys, etc.)
Data Format	Plain text (key-value pairs)	Base64-encoded
Encryption	Not encrypted	Can be encrypted at rest
Size Limit	No built-in limit	Limited to 1MB
Security	Anyone with access to the API server can view	Access controlled via RBAC

What is Kubernetes Liveness and Readiness Probes

In Kubernetes, Liveness and Readiness probes are usually used for monitoring the health of the pods and are quite similar as well. If the pod goes down or if the connection is interrupted these probes help efficiently to know it is down and also help recover it. They can be specified in the yaml files themselves.

These Kubernetes Liveness and Readiness are essential for monitoring the health of the pods. While the Liveness probes will determine if a pod is running, restarting it if necessary and Readiness probes will check if a pod is ready to serve the traffic. These can be specified directly in the YAML configuration files. It ensures automated recovery and smooth traffic management.



1. Liveness Probe

Liveness probes in kubernetes facilitate the application within a pod running as expected. If a liveness probe fails then it will that pod and then it will restart the pod according to the restart policy. It facilitates automatic recovery from the broken state and enhances the overall system reliability.

2. Readiness Probe

Readiness probes are determined if a pod is ready for accepting the incoming traffic. When a readiness probe fails, then it temporarily removes the probe from the service load balancer preventing it from receiving any requests unitl it is ready. These are particularly helpful during startup, maintenance or when scaling the applications.

Type of Probes

The next step is to define the probes that test readiness and liveness. There are three types of probes: HTTP, Command, and TCP. You can use any of them for liveness and readiness checks.

HTTP

HTTP probes are probably the most common type of custom liveness probe. Even if your app isn't an HTTP server, you can create a lightweight HTTP server inside your app to respond to the liveness probe. Kubernetes pings a path, and if it gets an HTTP response in the 200 or 300 range, it marks the app as healthy. Otherwise it is marked as unhealthy.

Command

For command probes, Kubernetes runs a command inside your container. If the command returns with exit code 0, then the container is marked as healthy. Otherwise, it is marked unhealthy. This type of probe is useful when you can't or don't want to run an HTTP server, but can run a command that can check whether or not your app is healthy.

TCP

The last type of probe is the TCP probe, where Kubernetes tries to establish a TCP connection on the specified port. If it can establish a connection, the container is considered healthy; if it can't it is considered unhealthy.

TCP probes come in handy if you have a scenario where HTTP probes or command probe don't work well. For example, a gRPC or FTP service is a prime candidate for this type of probe.

What is kubectl debug?

kubectl debug is a powerful command introduced to enhance Kubernetes troubleshooting. It lets you create an ephemeral container within an existing pod or even directly access a node, which makes debugging much more straightforward compared to the old days of accessing nodes manually or trying to add debug tools to existing containers.

Setting Up kubectl debug:

kubectl get nodes

Logging into a Node:

kubectl debug node/<node-name> -it --image=busybox

What is the kubectl describe command?

The kubectl describe command retrieves and displays detailed data from a single resource or multiple resources in your cluster. This command can be helpful when you are debugging specific issues in your Kubernetes workloads and need to see more than just a summary of your resources.

Here's the basic syntax:

kubectl describe <resource> <resource-name> [flags]

Name and Namespace: The name and namespace of the pod

Priority: The priority of the pod

Start Time: The timestamp indicating when the pod was created

Labels and Annotations: Any key-value pairs for identification (labels) and additional metadata (annotations) related to the pod

Node: The node where the pod is currently running

Status: The current state of the pod (e.g., Running, Pending, Failed)

IP Addresses: The pod's internal and external IP addresses

Containers: Each container's image, state, readiness probes, resource limits, and more — plus, events like crashes or restarts, including the last state of the container

Volumes: Information about any volumes attached to the pod

Events: Recent events that have affected the pod, such as scheduling issues, restarts, or failures

kubectl describe deployment:

kubectl describe deployment <deployment-name> -n <namespace>

kubectl port-forward

The kubectl port-forward command is a powerful tool that allows you to access internal Kubernetes cluster resources from your local machine. This command is particularly useful for debugging and troubleshooting applications running within a Kubernetes cluster without exposing them externally.

The basic syntax for the kubectl port-forward command is:

kubectl port-forward [resource-type]/[resource-name] [local-port]: [resource-port]

Here, [resource-type] can be a pod, deployment, service, or replicase, and [resource-name] is the name of the resource. [local-port] is the port on your local machine, and [resource-port] is the port on the resource within the cluster.

How to Set Up Port Forwarding With kubectl:

Setting up port forwarding with kubectl involves obtaining the resource's port data, executing the port-forward command, and connecting to the resource. Follow the steps below to complete the port forwarding procedure.

Step 1: View Available Resources

List the available resources by executing the relevant resource command. For example, to list all the services within a namespace, type the following:

kubectl -n [namespace] get svc

find the service you want to forward and note its name and the port number.

Listing the services running in the default namespace of the cluster.

```
marko@phoenixnap:~$ kubectl -n default get svc
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP   19m
nginx-service   ClusterIP   10.107.117.33   <none>        80/TCP    5m55s
marko@phoenixnap:~$
```

Step 2: Execute port-forward

Use the following command to access the resource in the cluster. For example, if the name of the service is nginx-service, and the port number is 80, expose Nginx on the local port 8080 by typing:

kubectl port-forward svc/nginx-service 8080:80

The Kubernetes API now listens to port 8080 and forwards data to the service port 80.

Port forwarding of the local port 8080 to the resource port 80.

```
marko@phoenixnap:~$ kubectl port-forward svc/nginx-service 8080:80
Forwarding from 127.0.0.1:8080 → 80
Forwarding from [::1]:8080 → 80
```

Step 3: Connect with Resource

Use the curl command to test if the service is available at localhost:8080:

curl localhost:8080

curl prints the HTML code of the Nginx page:

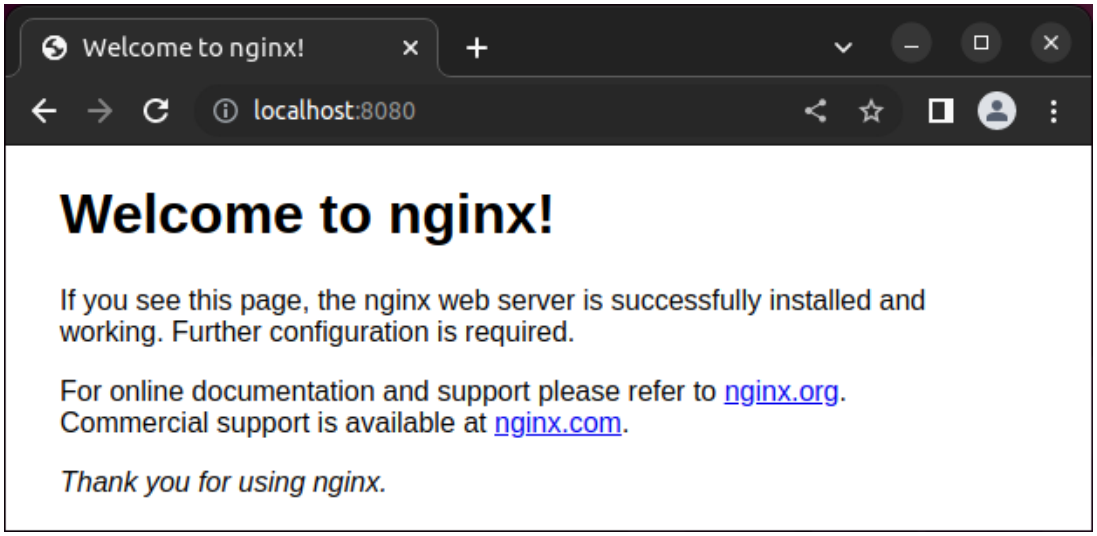
The curl command shows the NGINX test page.

Also, you can visit the same address in a web browser.

```
marko@phoenixnap:~$ curl localhost:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
marko@phoenixnap:~$
```

A web browser window showing the NGINX test page.



Once executed, the kubectl port-forward command actively runs in the terminal window. To issue other commands while port-forwarding is running, open another terminal instance.