AUTOMATE CONFIGURATION AND DEPLOYMENT

Training Material

TABLE OF CONTENTS

Chapter 1: Deploying Applications with Kubernetes

1.1 Introduction to Kubernetes Deployments and Pods

- **Kubernetes Deployments**: Deployments in Kubernetes are used to define the desired state for applications, ensuring the correct number of Pods are running. Deployments help manage scaling, rolling updates, and rollbacks, ensuring that applications are always available and meet the defined specifications.
- **Kubernetes Pods**: Pods are the smallest deployable unit in Kubernetes, typically used to run containers. A Pod encapsulates a container (or multiple containers) and storage, and provides a network interface. Pods are ephemeral and are usually replaced by Kubernetes when required.

1.2 Creating and Managing Deployments in Kubernetes

- **Creating a Deployment**:
    - Use kubectl create deployment to create deployments:

kubectl create deployment <name> --image=<image-name>

    - YAML-based deployment definition allows more control. Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
```

```
metadata:
  labels:
    app: my-app
spec:
  containers:
  - name: my-container
    image: my-image
    ports:
    - containerPort: 80
```

- **Managing Deployments**:
  - **Scaling**: You can scale deployments up or down using kubectl scale:

```
kubectl scale deployment my-deployment --replicas=5
```

  - **Rolling Updates**: Kubernetes supports automatic rolling updates to update application versions without downtime.
  - **Rollback**: Use kubectl rollout undo to roll back to a previous version if there is an issue with a deployment.

1.3 Configuring Multi-Container Pods

- **Multi-Container Pods**: Kubernetes allows multiple containers to run in a single Pod. This is useful for tightly coupled applications, such as a web server and a logging agent or a database and cache.
- **Use Cases**: Common patterns include:
  - **Sidecar pattern**: A helper container that complements the main application container (e.g., logging, monitoring).
  - **Ambassador pattern**: A proxy container that manages communication to the external services.
  - **Adapter pattern**: A container that modifies data for consumption by the main application.

1.4 Scaling Applications with ReplicaSets

- **ReplicaSets**: ReplicaSets ensure that a specified number of Pods are running at any given time. It works behind the scenes to manage the number of Pods in a Deployment.
    - **Scaling**: Scaling up or down the number of replicas:

kubectl scale --replicas=5 deployment/my-deployment

    - **Horizontal Pod Autoscaling**: Automatically adjusts the number of Pods based on CPU utilization or custom metrics.

Chapter 2: Managing Configuration in Kubernetes

2.1 Using ConfigMaps to Store and Manage Configuration Data

- **ConfigMaps**: Kubernetes ConfigMaps allow you to store configuration data separately from application code. This makes it easier to manage different configurations across various environments (development, testing, production).
    - **Creating ConfigMap**: You can create a ConfigMap from a literal value, file, or directory:

kubectl create configmap my-config --from-literal=key=value

    - **Using ConfigMaps**: Inject ConfigMap data into Pods either as environment variables or mount them as volumes in a container:

```
apiVersion: v1
kind: Pod
metadata:
  name: config-pod
spec:
  containers:
```

```
- name: app
  image: my-app
  envFrom:
  - configMapRef:
      name: my-config
```

2.2 Using Secrets for Sensitive Information

- **Secrets**: Kubernetes Secrets are used to store sensitive information such as passwords, API keys, or certificates. Secrets are base64-encoded and can be injected into Pods either as environment variables or volumes.
    - **Creating Secrets**:

```
kubectl create secret generic my-secret --from-literal=password=myPassword
```

    - **Using Secrets in Pods**: Similar to ConfigMaps, Secrets can be mounted as volumes or passed as environment variables to containers.

2.3 Injecting Configuration Data into Pods

- **Using ConfigMaps and Secrets**: Configuration data can be injected into Pods via environment variables or mounted as volumes:
    - **Environment Variables**:

```
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
  - name: app-container
    image: app-image
```

```
  envFrom:
  - configMapRef:
      name: my-config
```

- **Volumes**: Mount a ConfigMap or Secret as a file inside the container:

```
volumes:
- name: config-volume
  configMap:
    name: my-config
```

2.4 Environment-Specific Configuration with Kubernetes

- **Environment-Specific Configurations**: Different configurations may be needed for different environments (development, staging, production).
  - Use different namespaces for each environment or configure different ConfigMaps and Secrets for each environment.
  - Tools like Helm or Kustomize can manage multiple environments efficiently.

Chapter 3: Exposing Applications in Kubernetes

3.1 Understanding Kubernetes Services (ClusterIP, NodePort, LoadBalancer)

- **ClusterIP**: The default service type, which exposes the service within the Kubernetes cluster. It is only accessible internally.
- **NodePort**: Exposes the service on a static port on each node, making the service accessible externally via nodeIP:nodePort.
- **LoadBalancer**: Provision an external load balancer to expose the service to the outside world. Typically used in cloud environments.
  - Example of exposing a service as a LoadBalancer:

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: LoadBalancer
```

3.2 Setting Up Ingress Controllers and Ingress Rules

- **Ingress Controllers**: Ingress controllers manage external HTTP and HTTPS traffic to services within a Kubernetes cluster. They provide routing, load balancing, SSL termination, and more.
    - Example of an Ingress resource:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - host: my-app.example.com
    http:
      paths:
      - path: /
```

```
pathType: Prefix
backend:
  service:
    name: my-service
    port:
      number: 80
```

3.3 Service Discovery and Networking in Kubernetes

- **Service Discovery**: Kubernetes automatically provides DNS names to services, so Pods can access services by their names, such as my-service:80.
    - Pods in the same namespace can easily communicate with each other without any configuration changes.
- **Networking**: Kubernetes networking provides a flat network model where all Pods can reach each other, regardless of the node they are running on.

3.4 Best Practices for Exposing Applications

- **Use Ingress Controllers**: Ingress controllers provide fine-grained control over traffic routing and allow SSL termination.
- **Secure Communication**: Always use HTTPS to secure traffic between clients and your application.
- **Service Load Balancing**: Use a LoadBalancer or Ingress for high availability and efficient traffic management.