

## **All About Kubernetes**

Kubernetes (often abbreviated as K8s) is an open-source container orchestration platform designed to automate deploying, scaling, and managing containerized applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF).

# **Key Features of Kubernetes**

## **1. Container Orchestration**

- Manages and deploys containers efficiently.
- Automates container lifecycle management.

## **2. Scalability**

- Allows applications to scale up or down automatically based on demand.
- Supports horizontal and vertical scaling of containers.

## **3. Self-Healing**

- Restarts failed containers automatically.
- Replaces and reschedules containers when a node dies.

## **4. Load Balancing & Service Discovery**

- Distributes network traffic across multiple containers.
- Provides built-in DNS service discovery.

## **5. Automated Rollouts & Rollbacks**

- Manages application updates with zero downtime.
- Ensures rollbacks if an update fails.

## **6. Storage Orchestration**

- Supports various storage backends (local storage, cloud storage, NFS, etc.).
- Allows persistent storage for stateful applications.

## **7. Multi-Cloud & Hybrid Cloud Support**

- Works across public, private, and hybrid cloud environments.
  - Can run on AWS, Azure, GCP, and on-premise infrastructure.
-

# Kubernetes Architecture

## 1. Master Node (Control Plane)

Responsible for managing the cluster and consists of:

- **API Server** – The main entry point for interacting with Kubernetes.
- **Scheduler** – Assigns workloads to nodes.
- **Controller Manager** – Maintains cluster state (e.g., replicating pods).
- **etcd** – A key-value store for cluster data.

## 2. Worker Nodes

Run application workloads and consist of:

- **Kubelet** – Communicates with the master node and manages container execution.
  - **Kube Proxy** – Handles networking and load balancing.
  - **Container Runtime** – Runs the actual containers (e.g., Docker, containerd).
- 

# Kubernetes Objects

## 1. Pods

- The smallest deployable unit in Kubernetes.
- A pod can contain one or more containers.

## 2. Deployments

- Ensures a specified number of pods are running.
- Handles rolling updates and rollbacks.

## 3. Services

- Provides network access to pods.
- Types: ClusterIP, NodePort, LoadBalancer, and ExternalName.

## 4. ConfigMaps & Secrets

- ConfigMaps store non-sensitive configuration data.
- Secrets store sensitive information (e.g., passwords, API keys).

## 5. Persistent Volumes (PV) & Persistent Volume Claims (PVC)

- Provides storage for stateful applications.
  - Decouples storage from pods.
- 

## Kubernetes Networking

- **Cluster Networking** – Each pod gets a unique IP address.
  - **Service Discovery** – Kubernetes provides DNS resolution for services.
  - **Ingress** – Manages external access to services using an HTTP/HTTPS proxy.
- 

## Kubernetes Tools & Ecosystem

- **kubectl** – CLI tool for managing Kubernetes clusters.
  - **Helm** – Package manager for Kubernetes applications.
  - **Prometheus & Grafana** – Monitoring and visualization tools.
  - **Kubernetes Dashboard** – Web UI for cluster management.
- 

## Kubernetes vs. Docker

- **Docker** is a container runtime, while **Kubernetes** orchestrates multiple containers.
  - **Docker Compose** is for managing multiple containers on a single host, while **Kubernetes** is designed for multi-node clusters.
- 

## Getting Started with Kubernetes

1. Install **Minikube** for local Kubernetes testing.
2. Use **kubectl** to interact with the cluster.
3. Deploy applications using YAML configurations.
4. Monitor and scale applications using Kubernetes features.

Would you like help setting up Kubernetes or deploying an application? 🚀

You said:

yes

*ChatGPT said:*

ChatGPT

Great! Let's start by setting up a **local Kubernetes cluster** using **Minikube** and deploying a simple application.

---

## Step 1: Install Kubernetes and Minikube

### Prerequisites:

1. **Install Docker** (or another container runtime like Podman/Containerd).
  - Download from: <https://www.docker.com/get-started>
2. **Install kubectl** (Kubernetes CLI tool).
  - On Linux/macOS:

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
chmod +x kubectl
sudo mv kubectl /usr/local/bin/
```

- On Windows: Download from Kubernetes official site.
3. **Install Minikube** (Local Kubernetes cluster).
    - On Linux/macOS:

```
curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube-
linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

- On Windows: Download from Minikube official site.
- 

## Step 2: Start Minikube

Once everything is installed, start your Kubernetes cluster:

```
minikube start
```

To check if it's running:

```
kubectl get nodes
```

Expected output:

NAME	STATUS	ROLES	AGE	VERSION
minikube	Ready	control-plane,master	5m	v1.XX.0

---

## Step 3: Deploy a Simple Application

We'll deploy an **Nginx web server** using Kubernetes.

### 1. Create a Deployment

```
kubectl create deployment nginx --image=nginx
```

### 2. Expose the Deployment as a Service

```
kubectl expose deployment nginx --type=NodePort --port=80
```

### 3. Get the Service Details

```
kubectl get services
```

Look for `nginx` service and note the `PORT`.

### 4. Access the Application

```
minikube service nginx --url
```

Open the given URL in your browser. 🎉

---

## Step 4: Deploying with YAML (More Advanced Approach)

Instead of using CLI commands, we can use a YAML configuration file.

### 1. Create a file `nginx-deployment.yaml`:

```
yaml
```

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30007
  type: NodePort

```

## 2. Apply the YAML file:

```
kubectl apply -f nginx-deployment.yaml
```

## 3. Get the Minikube Service URL:

```
minikube service nginx-service --url
```

# Next Steps

- Scale your app:

```
kubectl scale deployment nginx-deployment --replicas=4
```

- Monitor Pods:

```
kubectl get pods
```

- View Logs:

```
kubectl logs -f <pod-name>
```

Would you like help deploying a more complex app, like a database-backed application? 🚀

## Overview of the Architecture

We will deploy:

1. **MySQL Database** (for storing data)
  2. **Spring Boot Microservices** (REST API for business logic)
  3. **Kubernetes Services** (for inter-service communication)
- 

## Step 1: Create a MySQL Deployment in Kubernetes

### 1. Create a `mysql-deployment.yaml` file:

yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
```

```

      labels:
        app: mysql
    spec:
      containers:
        - name: mysql
          image: mysql:8.0
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: "rootpass"
            - name: MYSQL_DATABASE
              value: "microservices_db"
          ports:
            - containerPort: 3306
          volumeMounts:
            - mountPath: /var/lib/mysql
              name: mysql-storage
      volumes:
        - name: mysql-storage
          persistentVolumeClaim:
            claimName: mysql-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
spec:
  selector:
    app: mysql
  ports:
    - protocol: TCP
      port: 3306
      targetPort: 3306
  clusterIP: None

```

## 2. Apply the MySQL configuration:

```
kubectl apply -f mysql-deployment.yaml
```

## 3. Verify MySQL is Running:

```
kubectl get pods
```

# Step 2: Create a Spring Boot Microservice

## 1. Create a New Spring Boot Project

Use [Spring Initializr](#) and select:

- **Spring Boot version:** 3.x
- **Dependencies:** Spring Web, Spring Data JPA, MySQL Driver



Extract the generated project and open it in **IntelliJ** or **VS Code**.

---

## 2. Update `application.properties` to Connect to MySQL

Edit `src/main/resources/application.properties`:

```
properties
CopyEdit
spring.datasource.url=jdbc:mysql://mysql-service:3306/microservices_db
spring.datasource.username=root
spring.datasource.password=rootpass
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.application.name=user-service
server.port=8080
```

---

## 3. Create an Entity and Repository

### *Create `User.java`*

```
package com.example.demo.model;

import jakarta.persistence.*;

@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;

    // Getters and Setters
}
```

### *Create `UserRepository.java`*

```
package com.example.demo.repository;

import com.example.demo.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
}
```

---

## 4. Create a REST Controller

### Create UserController.java

```
java

package com.example.demo.controller;

import com.example.demo.model.User;
import com.example.demo.repository.UserRepository;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {

    private final UserRepository userRepository;

    public UserController(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @GetMapping
    public List<User> getUsers() {
        return userRepository.findAll();
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userRepository.save(user);
    }
}
```

---

## Step 3: Containerize the Spring Boot Application

### 1. Create a Dockerfile

```
dockerfile

FROM openjdk:17
WORKDIR /app
COPY target/*.jar app.jar
CMD ["java", "-jar", "app.jar"]
EXPOSE 8080
```

### 2. Build and Push the Image to Docker Hub

```
mvn clean package -DskipTests
docker build -t yourdockerhubusername/user-service:v1 .
docker push yourdockerhubusername/user-service:v1
```

If using **Minikube**, build locally:

```
eval $(minikube docker-env)
docker build -t user-service:v1 .
```

---

## Step 4: Deploy the Spring Boot Microservice to Kubernetes

### 1. Create `user-service-deployment.yaml`

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
        - name: user-service
          image: yourdockerhubusername/user-service:v1
          ports:
            - containerPort: 8080
          env:
            - name: SPRING_DATASOURCE_URL
              value: "jdbc:mysql://mysql-service:3306/microservices_db"
            - name: SPRING_DATASOURCE_USERNAME
              value: "root"
            - name: SPRING_DATASOURCE_PASSWORD
              value: "rootpass"
---
apiVersion: v1
kind: Service
metadata:
  name: user-service
spec:
  selector:
    app: user-service
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: NodePort
```

### 2. Apply the Deployment

```
kubectl apply -f user-service-deployment.yaml
```

### 3. Get the Service URL

```
minikube service user-service --url
```

Now, open the URL in Postman or a browser:

- **GET** /users → Retrieve users
- **POST** /users (with JSON body) → Create a user

---

## Step 5: Verify and Monitor

- Check Pods:

```
kubectl get pods
```

- View Logs:

```
kubectl logs -f <user-service-pod-name>
```

- Scale the Service:

```
kubectl scale deployment user-service --replicas=3
```

---

## Next Steps

- Implement **Eureka Service Discovery** for multiple microservices.
- Add **API Gateway (Spring Cloud Gateway or Nginx)** for better routing.
- Use **Kubernetes Ingress** for cleaner URLs.
- Set up **Grafana + Prometheus** for monitoring.

Would you like help with scaling, monitoring, or adding another microservice? 🚀

40

You said:

yes

*ChatGPT said:*

ChatGPT

Great! Let's take our **Spring Boot microservices on Kubernetes** to the next level by adding **service discovery, API gateway, and monitoring**.

---

## Step 1: Service Discovery with Eureka

In a **microservices architecture**, each service may have multiple instances running. **Eureka** helps services discover each other dynamically.

### 1. Create an Eureka Server

Use **Spring Initializr** and add:

- **Spring Boot version:** 3.x
  - **Dependencies:** Eureka Server
- 

### 2. Configure Eureka Server (`application.properties`)

```
properties

server.port=8761
spring.application.name=eureka-server

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

---

### 3. Enable Eureka Server in `EurekaServerApplication.java`

Java

```
package com.example.eurekaserver;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

```
}
```

---

## 4. Containerize the Eureka Server

Create a Dockerfile:

```
dockerfile

FROM openjdk:17
WORKDIR /app
COPY target/*.jar app.jar
CMD ["java", "-jar", "app.jar"]
EXPOSE 8761
```

## 5. Deploy Eureka Server on Kubernetes

*Create eureka-deployment.yaml*

```
yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: eureka-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: eureka-server
  template:
    metadata:
      labels:
        app: eureka-server
    spec:
      containers:
        - name: eureka-server
          image: yourdockerhubusername/eureka-server:v1
          ports:
            - containerPort: 8761
---
apiVersion: v1
kind: Service
metadata:
  name: eureka-server
spec:
  selector:
    app: eureka-server
  ports:
    - protocol: TCP
      port: 8761
      targetPort: 8761
  type: NodePort

kubectl apply -f eureka-deployment.yaml
```

Now, access `http://<minikube-ip>:<eureka-port>` in your browser.

---

## Step 2: Register Microservices with Eureka

Modify `application.properties` of your microservice:

```
properties

server.port=8080
spring.application.name=user-service

eureka.client.serviceUrl.defaultZone=http://eureka-server:8761/eureka/
eureka.instance.prefer-ip-address=true
```

Now, when the microservice starts, it **registers itself with Eureka**.

---

## Step 3: Implement an API Gateway

### 1. Create an API Gateway Using Spring Cloud Gateway

Use **Spring Initializr** and add:

- **Spring Cloud Gateway**
  - **Eureka Discovery Client**
- 

### 2. Configure `application.properties`

```
Properties

server.port=8082
spring.application.name=api-gateway

eureka.client.serviceUrl.defaultZone=http://eureka-server:8761/eureka/

spring.cloud.gateway.routes[0].id=user-service
spring.cloud.gateway.routes[0].uri=lb://user-service
spring.cloud.gateway.routes[0].predicates=Path=/users/**
```

Now, when you call `http://api-gateway:8082/users`, it automatically routes to `user-service`.

---

### 3. Deploy API Gateway on Kubernetes

*Create api-gateway-deployment.yaml*

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-gateway
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api-gateway
  template:
    metadata:
      labels:
        app: api-gateway
    spec:
      containers:
        - name: api-gateway
          image: yourdockerhubusername/api-gateway:v1
          ports:
            - containerPort: 8082
---
apiVersion: v1
kind: Service
metadata:
  name: api-gateway
spec:
  selector:
    app: api-gateway
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8082
  type: NodePort
```

```
kubectl apply -f api-gateway-deployment.yaml
```

Now, calling `http://<minikube-ip>:<api-gateway-port>/users` routes to the **User Service**



---

## Step 4: Monitoring with Prometheus & Grafana

### 1. Install Prometheus & Grafana

```
kubectl create namespace monitoring
```



```
kubectl apply -f https://raw.githubusercontent.com/prometheus-operator/prometheus-operator/main/bundle.yaml
```

## 2. Configure Prometheus to Monitor Services

Create `prometheus-config.yaml`:

yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: monitoring
data:
  prometheus.yml: |
    global:
      scrape_interval: 5s
    scrape_configs:
      - job_name: 'spring-boot'
        metrics_path: '/actuator/prometheus'
        static_configs:
          - targets: ['user-service:8080']
```

Apply it:

```
kubectl apply -f prometheus-config.yaml
```

---

## 3. Deploy Grafana

```
kubectl apply -f
https://raw.githubusercontent.com/grafana/grafana/main/deploy/kubernetes/grafana-deployment.yaml
```

Access Grafana UI and **add Prometheus as a data source**.

---

## Step 5: Test and Verify

```
kubectl get pods
kubectl get services
```

**Verify Each Component:**

- ✅ **Eureka** → `http://eureka-server:8761/`
  - ✅ **API Gateway** → `http://api-gateway:8082/users`
  - ✅ **Prometheus** → `http://prometheus-service:9090`
  - ✅ **Grafana** → `http://grafana-service:3000`
- 

## Next Steps

- **Add Zipkin/Jaeger** for distributed tracing.
- **Deploy another microservice** (e.g., Order Service).
- **Set up Kubernetes Ingress** for a cleaner domain name.

Would you like help with any of these next steps? 🚀

Awesome! Let's break it down step by step. We'll cover everything:

- ✅ **Eureka for Service Discovery**
  - ✅ **Spring Cloud Gateway for API Routing**
  - ✅ **Prometheus & Grafana for Monitoring**
  - ✅ **Zipkin/Jaeger for Distributed Tracing**
  - ✅ **Adding an Order Service**
  - ✅ **Kubernetes Ingress for Domain Management**
- 

## Step 1: Distributed Tracing with Zipkin/Jaeger

Tracing helps track **requests across multiple microservices** to debug latency issues.

### 1. Deploy Zipkin to Kubernetes

*Create `zipkin-deployment.yaml`*

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: zipkin
spec:
  replicas: 1
  selector:
    matchLabels:
      app: zipkin
  template:
    metadata:
```

```
    labels:
      app: zipkin
    spec:
      containers:
        - name: zipkin
          image: openzipkin/zipkin
          ports:
            - containerPort: 9411
---
apiVersion: v1
kind: Service
metadata:
  name: zipkin
spec:
  selector:
    app: zipkin
  ports:
    - protocol: TCP
      port: 9411
      targetPort: 9411
  type: NodePort
```

Apply it:

```
kubectl apply -f zipkin-deployment.yaml
```

Access **Zipkin UI** at <http://<minikube-ip>:9411>.

---

## 2. Configure Microservices for Tracing

In `application.properties` of each service:

```
properties

spring.zipkin.base-url=http://zipkin:9411
management.tracing.sampling.probability=1.0
```

Now, all requests will be **tracked by Zipkin**.

---

## Step 2: Add an Order Microservice

### 1. Create `Order.java` Entity

```
java

package com.example.order.model;
```

```

import jakarta.persistence.*;

@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String product;
    private Integer quantity;
    private Double price;

    // Getters and Setters
}

```

## 2. Create OrderRepository.java

```

java

package com.example.order.repository;

import com.example.order.model.Order;
import org.springframework.data.jpa.repository.JpaRepository;

public interface OrderRepository extends JpaRepository<Order, Long> {
}

```

## 3. Create OrderController.java

```

java

package com.example.order.controller;

import com.example.order.model.Order;
import com.example.order.repository.OrderRepository;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/orders")
public class OrderController {

    private final OrderRepository orderRepository;

    public OrderController(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }

    @GetMapping
    public List<Order> getOrders() {
        return orderRepository.findAll();
    }
}

```

```
    @PostMapping
    public Order createOrder(@RequestBody Order order) {
        return orderRepository.save(order);
    }
}
```

---

## 4. Register `order-service` with Eureka

properties

```
spring.application.name=order-service
eureka.client.serviceUrl.defaultZone=http://eureka-server:8761/eureka/
```

---

## 5. Deploy `order-service` on Kubernetes

*Create `order-service-deployment.yaml`*

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: order-service
  template:
    metadata:
      labels:
        app: order-service
    spec:
      containers:
        - name: order-service
          image: yourdockerhubusername/order-service:v1
          ports:
            - containerPort: 8081
          env:
            - name: SPRING_DATASOURCE_URL
              value: "jdbc:mysql://mysql-service:3306/microservices_db"
            - name: SPRING_DATASOURCE_USERNAME
              value: "root"
            - name: SPRING_DATASOURCE_PASSWORD
              value: "rootpass"
---
apiVersion: v1
kind: Service
metadata:
  name: order-service
spec:
  selector:
    app: order-service
```

```
ports:
  - protocol: TCP
    port: 80
    targetPort: 8081
  type: NodePort
```

Deploy it:

```
kubectl apply -f order-service-deployment.yaml
```

---

## Step 3: Update API Gateway for Order Service

Modify `application.properties`:

```
properties

spring.cloud.gateway.routes[1].id=order-service
spring.cloud.gateway.routes[1].uri=lb://order-service
spring.cloud.gateway.routes[1].predicates=Path=/orders/**
```

---

## Step 4: Set Up Kubernetes Ingress

Ingress provides **a single entry point** for microservices.

### 1. Enable Ingress in Minikube

```
minikube addons enable ingress
```

### 2. Create an Ingress Controller

*Create `ingress.yaml`*

```
yaml

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: microservices-ingress
spec:
  rules:
    - host: user-service.local
      http:
        paths:
          - path: /users
            pathType: Prefix
            backend:
              service:
```

```
        name: user-service
        port:
          number: 80
- host: order-service.local
  http:
    paths:
      - path: /orders
        pathType: Prefix
        backend:
          service:
            name: order-service
            port:
              number: 80
```

Apply it:

```
kubect1 apply -f ingress.yaml
```

Now, access:

- **User Service:** <http://user-service.local/users>
  - **Order Service:** <http://order-service.local/orders>
- 

## Step 5: Test Everything

### 1. Check if all pods are running

```
kubect1 get pods
```

### 2. Test API Gateway

```
curl http://<api-gateway-ip>/users
curl http://<api-gateway-ip>/orders
```

### 3. Check Eureka Service Registry

Visit <http://eureka-server:8761/>

### 4. Monitor in Prometheus

Visit <http://<prometheus-ip>:9090>

### 5. Check Traces in Zipkin

Visit <http://zipkin:9411/>

---

## Final Architecture Overview

- ✓ **Eureka** → Service Discovery
- ✓ **API Gateway** → Load Balancing & Routing
- ✓ **MySQL** → Database
- ✓ **User & Order Services** → Independent microservices
- ✓ **Zipkin** → Distributed Tracing
- ✓ **Prometheus & Grafana** → Monitoring
- ✓ **Ingress** → Clean URLs

Would you like to **add security (JWT, OAuth2)** or **CI/CD (GitHub Actions, ArgoCD)** next?



You said:

yes

*ChatGPT said:*

ChatGPT

Memory updated

Great! Let's now implement **security (JWT, OAuth2)** and **CI/CD (GitHub Actions, ArgoCD)** for your microservices.

---

## Step 1: Implement Security (JWT, OAuth2)

We'll use **Keycloak** for authentication and **Spring Security** for protecting endpoints.

### 1. Deploy Keycloak on Kubernetes

Create `keycloak-deployment.yaml`:

yaml

```
apiVersion: apps/v1
kind: Deployment
```



```

metadata:
  name: keycloak
spec:
  replicas: 1
  selector:
    matchLabels:
      app: keycloak
  template:
    metadata:
      labels:
        app: keycloak
    spec:
      containers:
        - name: keycloak
          image: quay.io/keycloak/keycloak:latest
          args: ["start-dev"]
          env:
            - name: KEYCLOAK_ADMIN
              value: "admin"
            - name: KEYCLOAK_ADMIN_PASSWORD
              value: "admin"
            - name: KC_DB
              value: "mysql"
            - name: KC_DB_URL
              value: "jdbc:mysql://mysql-service:3306/keycloak"
            - name: KC_DB_USERNAME
              value: "root"
            - name: KC_DB_PASSWORD
              value: "rootpass"
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: keycloak
spec:
  selector:
    app: keycloak
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
  type: NodePort

```

Apply it:

```
kubectl apply -f keycloak-deployment.yaml
```

Access Keycloak at `http://<minikube-ip>:<keycloak-port>`, log in with **admin/admin**, and create a realm, client, and users.

---

## 2. Secure Microservices with JWT

### Add Dependencies in `pom.xml`

```
xml

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

### Configure JWT in `application.properties`

```
properties

spring.security.oauth2.resourceserver.jwt.issuer-
uri=http://keycloak:8080/realms/microservices
```

### Secure Endpoints in `SecurityConfig.java`

```
java

package com.example.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.oauth2.server.resource.authentication.JwtAuthent
icationConverter;
import
org.springframework.security.oauth2.server.resource.authentication.JwtGranted
AuthoritiesConverter;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/users/**").hasAuthority("SCOPE_user")
                .requestMatchers("/orders/**").hasAuthority("SCOPE_order")
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2.jwt(jwt ->
jwt.jwtAuthenticationConverter(jwtAuthConverter())));
        return http.build();
    }
}
```

```

    private JwtAuthenticationConverter jwtAuthConverter() {
        JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new
JwtGrantedAuthoritiesConverter();
        grantedAuthoritiesConverter.setAuthorityPrefix("ROLE_");
        grantedAuthoritiesConverter.setAuthoritiesClaimName("roles");

        JwtAuthenticationConverter authenticationConverter = new
JwtAuthenticationConverter();

        authenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesC
onverter);
        return authenticationConverter;
    }
}

```

Now, microservices will **validate JWT tokens** and restrict access based on roles.

---

## Step 2: Implement CI/CD with GitHub Actions & ArgoCD

### 1. Setup GitHub Actions for CI

*Create .github/workflows/build-deploy.yml*

yml

```
name: Build and Deploy Microservices
```

```
on:
```

```
  push:
    branches:
      - main
```

```
jobs:
```

```
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
        uses: actions/checkout@v3

      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
          distribution: 'temurin'
          java-version: '17'

      - name: Build with Maven
        run: mvn clean package -DskipTests

      - name: Build Docker Image
        run: docker build -t yourdockerhubusername/microservice:v1 .

      - name: Push Image to DockerHub
```

```

    run: |
        echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{
secrets.DOCKER_USERNAME }}" --password-stdin
        docker push yourdockerhubusername/microservice:v1

deploy:
  needs: build
  runs-on: ubuntu-latest
  steps:
    - name: Deploy to Kubernetes
      run: kubectl apply -f kubernetes/

```

### ◆ What This Does:

1. Builds the project using **Maven**
2. Creates a **Docker image**
3. Pushes the image to **DockerHub**
4. Deploys the new image to **Kubernetes**

---

## 2. Install ArgoCD for Continuous Deployment

ArgoCD will automatically **sync Git changes to Kubernetes**.

```

kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-
cd/stable/manifests/install.yaml

```

Access ArgoCD:

```

kubectl port-forward svc/argocd-server -n argocd 8080:443

```

Go to **https://localhost:8080** and log in with:

```

kubectl get secrets -n argocd argocd-initial-admin-secret -o
jsonpath="{.data.password}" | base64 --decode

```

---

## 3. Deploy Microservices with ArgoCD

Create `argocd-app.yaml`:

```

yaml

apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:

```

```
name: microservices-app
namespace: argocd
spec:
  destination:
    namespace: default
    server: https://kubernetes.default.svc
  source:
    repoURL: https://github.com/yourgithubusername/microservices-k8s.git
    path: kubernetes
    targetRevision: main
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

Apply it:

```
kubectl apply -f argocd-app.yaml
```

Now, **ArgoCD** will watch your **GitHub repo** and deploy changes automatically! 🚀

---

## Final Architecture

- ✅ **OAuth2 + JWT Security** (Keycloak)
- ✅ **CI/CD Pipeline** (GitHub Actions)
- ✅ **Continuous Deployment** (ArgoCD)

- 💡 Now, users must **log in via Keycloak** and obtain a JWT token.
  - 💡 **GitHub Actions** builds & pushes images.
  - 💡 **ArgoCD** deploys updates automatically.
- 

## Next Steps

- **Istio** for advanced traffic management
- **Kafka/RabbitMQ** for event-driven microservices
- **Horizontal Pod Autoscaling** for better performance

Which one would you like to explore next? 🚀

You said:

all

*ChatGPT said:*

ChatGPT

Memory updated

Awesome! Let's break down the next steps:

1. **Istio for Advanced Traffic Management** 🚦
  2. **Kafka/RabbitMQ for Event-Driven Microservices** 🔁
  3. **Horizontal Pod Autoscaling (HPA) for Performance Optimization** ✅
- 

## Step 1: Implement Istio for Traffic Management

Istio provides **service mesh capabilities** like load balancing, traffic control, and security.

### 1. Install Istio

```
curl -L https://istio.io/downloadIstio | sh -  
cd istio-*/  
export PATH=$PWD/bin:$PATH  
istioctl install --set profile=demo -y  
kubectl label namespace default istio-injection=enabled
```

✅ **This enables automatic sidecar injection** for all microservices.

---

### 2. Deploy Microservices with Istio

Modify `order-service-deployment.yaml` to include Istio:

```
yaml  
  
apiVersion: networking.istio.io/v1beta1  
kind: VirtualService  
metadata:  
  name: order-service  
spec:  
  hosts:  
    - order-service  
  http:  
    - route:  
      - destination:  
        host: order-service
```

```
subset: v1
```

Apply it:

```
kubectl apply -f order-service-deployment.yaml
```

✅ Now, Istio **controls traffic routing**.

---

### 3. Enable Istio Ingress Gateway

Create `istio-gateway.yaml`:

```
yaml

apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: microservices-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - "*"

---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: gateway-routes
spec:
  hosts:
    - "*"
  gateways:
    - microservices-gateway
  http:
    - match:
        - uri:
            prefix: /orders
      route:
        - destination:
            host: order-service
            port:
              number: 80
```

Apply it:

```
sh
```

```
kubectl apply -f istio-gateway.yaml
```

✅ Now, requests are routed via **Istio Gateway**.

---

## Step 2: Implement Kafka/RabbitMQ for Event-Driven Microservices

### 1. Deploy Kafka on Kubernetes

Create `kafka-deployment.yaml`

```
yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kafka
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kafka
  template:
    metadata:
      labels:
        app: kafka
    spec:
      containers:
        - name: kafka
          image: bitnami/kafka:latest
          env:
            - name: KAFKA_CFG_ZOOKEEPER_CONNECT
              value: "zookeeper:2181"
            - name: ALLOW_PLAINTEXT_LISTENER
              value: "yes"
          ports:
            - containerPort: 9092
---
apiVersion: v1
kind: Service
metadata:
  name: kafka
spec:
  selector:
    app: kafka
  ports:
    - protocol: TCP
      port: 9092
      targetPort: 9092
```



Apply it:

```
sh
```

```
kubectl apply -f kafka-deployment.yaml
```

 Now, Kafka is ready to handle events.

---

## 2. Publish and Consume Kafka Messages in Spring Boot

### Add Dependencies (`pom.xml`)

```
xml
```

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

### Configure Kafka in `application.properties`

```
properties
```

```
spring.kafka.bootstrap-servers=kafka:9092
spring.kafka.consumer.group-id=microservices-group
```

### Produce Events in Order Service

```
java
```

```
@Autowired
private KafkaTemplate<String, String> kafkaTemplate;

@PostMapping
public Order createOrder(@RequestBody Order order) {
    Order savedOrder = orderRepository.save(order);
    kafkaTemplate.send("order-topic", "Order created: " +
        savedOrder.getId());
    return savedOrder;
}
```

### Consume Events in Payment Service

```
java
```

```
@KafkaListener(topics = "order-topic", groupId = "microservices-group")
public void processOrder(String message) {
    System.out.println("Received Order Event: " + message);
}
```

✅ Now, events are published and consumed asynchronously.

---

## Step 3: Implement Horizontal Pod Autoscaling (HPA)

HPA scales microservices automatically based on CPU or memory usage.

### 1. Enable Metrics Server

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

✅ This allows Kubernetes to **measure CPU & memory usage**.

---

### 2. Deploy HPA for Order Service

```
kubectl autoscale deployment order-service --cpu-percent=50 --min=1 --max=5
```

✅ This will **scale pods** dynamically.

---

## Final Architecture

- ✅ **Istio** → Traffic control & security
- ✅ **Kafka** → Asynchronous communication
- ✅ **HPA** → Automatic scaling

Would you like to **add more observability (Jaeger, OpenTelemetry)** or **optimize database performance** next? 🚀

4o

You said:

all

*ChatGPT said:*

ChatGPT

Memory updated

Great! Now, let's implement:

1. **Jaeger/OpenTelemetry for Observability & Distributed Tracing** 🔍
  2. **Database Performance Optimization** ⚡
- 

## Step 1: Implement Jaeger & OpenTelemetry for Observability

Jaeger helps track requests across microservices for debugging and performance monitoring.

### 1. Deploy Jaeger on Kubernetes

```
kubectl create namespace observability
kubectl apply -n observability -f
https://raw.githubusercontent.com/jaegertracing/jaeger-
kubernetes/master/jaeger-production-template.yml
```

✅ This deploys **Jaeger** with **Elasticsearch** as storage.

#### Access Jaeger UI

```
kubectl port-forward svc/jaeger-query -n observability 16686:16686
```

Go to **http://localhost:16686** to view traces.

---

## 2. Integrate OpenTelemetry in Spring Boot Microservices

### Add Dependencies (pom.xml)

```
xml

<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-api</artifactId>
  <version>1.16.0</version>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-exporter-jaeger</artifactId>
  <version>1.16.0</version>
```

```
</dependency>
```

## Configure OpenTelemetry in `application.properties`

```
properties
```

```
otel.exporter.jaeger.endpoint=http://jaeger-collector.observability:14250
otel.traces.exporter=jaeger
otel.metrics.exporter=none
otel.logs.exporter=none
```

## Instrument Spring Boot Services

```
java
```

```
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.Tracer;

@RestController
@RequestMapping("/orders")
public class OrderController {

    private final Tracer tracer;

    public OrderController(Tracer tracer) {
        this.tracer = tracer;
    }

    @GetMapping("/{id}")
    public Order getOrder(@PathVariable Long id) {
        Span span = tracer.spanBuilder("Fetching Order").startSpan();
        try {
            return orderService.getOrder(id);
        } finally {
            span.end();
        }
    }
}
```

 Now, all microservices **send trace data to Jaeger** for monitoring.

---

## Step 2: Optimize Database Performance

### 1. Enable Connection Pooling with HikariCP

In `application.properties`:

```
properties
```

```
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.idle-timeout=30000
spring.datasource.hikari.connection-timeout=20000
spring.datasource.hikari.max-lifetime=1800000
```

✅ This ensures **efficient database connections**.

---

## 2. Implement Caching with Redis

### Step 1: Deploy Redis on Kubernetes

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: redis
          image: redis:latest
          ports:
            - containerPort: 6379
---
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  selector:
    app: redis
  ports:
    - protocol: TCP
      port: 6379
      targetPort: 6379
```

Apply it:

```
kubectl apply -f redis-deployment.yaml
```

✅ **Redis is now running.**

## Step 2: Add Redis Cache in Spring Boot

xml

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

## Step 3: Configure Redis in `application.properties`

properties

```
spring.redis.host=redis
spring.redis.port=6379
```

## Step 4: Cache Order Data in `OrderService.java`

java

```
@Service
public class OrderService {

    @Autowired
    private OrderRepository orderRepository;

    @Cacheable(value = "orders", key = "#id")
    public Order getOrder(Long id) {
        return orderRepository.findById(id).orElseThrow();
    }
}
```

✅ **Frequently accessed orders are cached in Redis, reducing DB load.**

---

## Final Architecture

✅ **Jaeger/OpenTelemetry** → Observability & Tracing

✅ **Database Optimization** → Faster queries with **HikariCP** & **Redis** caching

What do you want to improve next? **Resilience (Circuit Breaker with Resilience4j)** or **API Rate Limiting (Spring Cloud Gateway)**? 🚀

