

- [Kubernetes Deployment Guide](#)
 - [Overview](#)
 - [Kubernetes Configuration Files](#)
 - [Core Infrastructure](#)
 - [Database Service](#)
 - [Kafka Ecosystem](#)
 - [Application Service](#)
 - [Deployment Automation](#)
 - [Why Kubernetes vs. Docker Compose](#)
 - [Running the Application Locally](#)
 - [Option 1: Standalone Spring Boot with H2 Database](#)
 - [Option 2: Docker Compose \(Recommended for Local Development\)](#)
 - [Option 3: Local Kubernetes with Docker Desktop \(Recommended for Windows\)](#)
 - [Setting Up Docker Desktop Kubernetes](#)
 - [Deploying the Application to Kubernetes](#)
 - [Alternative: Using Minikube](#)
 - [Accessing Kubernetes Deployed Services](#)
 - [Port Forwarding for Local Access](#)
 - [Port Forwarding for the Product Service API](#)
 - [Port Forwarding for Kafka UI](#)
 - [Port Forwarding for PostgreSQL](#)
 - [Multiple Port Forwards](#)
 - [Testing the Application](#)
 - [API Testing](#)
 - [List Products](#)
 - [Create a Product](#)
 - [Update a Product](#)
 - [Delete a Product](#)
 - [GUI Testing](#)
 - [Event Verification](#)
 - [Troubleshooting Kafka](#)
 - [Monitoring Application Health](#)
 - [Testing in CI/CD Pipelines](#)
 - [Stopping and Cleaning Kubernetes Resources](#)
 - [Option 1: Delete the Entire Namespace](#)
 - [Option 2: Scale Down Individual Deployments](#)

- [Option 3: Disable Kubernetes in Docker Desktop](#)
- [Option 4: Shut Down Docker Desktop Completely](#)
- [Kubernetes Deployment for Production](#)
- [Monitoring and Observability](#)
- [Security Considerations](#)
- [High Availability Configuration](#)
- [Troubleshooting](#)

Kubernetes Deployment Guide

Overview

This document provides a comprehensive guide to the Kubernetes configuration for the Product Service microservice. Kubernetes offers a production-grade container orchestration platform that enhances our application's scalability, reliability, and maintainability compared to simple Docker Compose deployments.

Kubernetes Configuration Files

Our Kubernetes configuration consists of several manifest files, each serving a specific purpose in the orchestration of our microservices:

Core Infrastructure

File	Purpose	Key Components
<code>00-namespace.yaml</code>	Creates an isolated Kubernetes namespace	Defines <code>ecommerce</code> namespace with appropriate labels
<code>01-configmap.yaml</code>	Stores non-sensitive configuration data	Contains Spring Boot and Kafka configuration properties
<code>02-secrets.yaml</code>	Securely stores sensitive credentials	Holds base64-encoded database credentials
<code>03-postgres-pvc.yaml</code>	Requests persistent storage for database	Defines storage requirements and access modes

Database Service

File	Purpose	Key Components
04-postgres-deployment.yaml	Deploys PostgreSQL database	Container specs, resource limits, health checks
05-postgres-service.yaml	Provides network access to database	Service definition with appropriate ports

Kafka Ecosystem

File	Purpose	Key Components
06-zookeeper-deployment.yaml	Deploys Zookeeper for Kafka coordination	Container specs, resource limits, health probes
07-zookeeper-service.yaml	Provides network access to Zookeeper	Service definition with all required ports
08-kafka-deployment.yaml	Deploys Kafka message broker	Container configuration, broker settings, health checks
09-kafka-service.yaml	Provides network access to Kafka	Service definition for both internal and external access
10-kafka-ui-deployment.yaml	Deploys Kafka management UI	Container specs, configuration, health checks
11-kafka-ui-service.yaml	Provides network access to Kafka UI	Service definition with port mapping

Application Service

File	Purpose	Key Components
12-product-service-deployment.yaml	Deploys our Spring Boot application	Container specs, rolling update strategy, health probes
13-product-service-service.yaml	Provides network access to our API	Service definition with port mapping

File	Purpose	Key Components
<code>14-ingress.yaml</code>	Manages external HTTP access	Routing rules, TLS configuration, path rewriting

Deployment Automation

File	Purpose	Key Components
<code>kustomization.yaml</code>	Simplifies deployment orchestration	Resource references, common labels and annotations

Why Kubernetes vs. Docker Compose

While Docker Compose is excellent for local development, Kubernetes provides several enterprise-grade capabilities critical for production deployments:

1. **High Availability:** Kubernetes enables running multiple replicas of services with automatic failover
2. **Scaling:** Horizontal pod autoscaling based on CPU/memory metrics
3. **Rolling Updates:** Zero-downtime deployments with gradual instance replacement
4. **Self-healing:** Automatic replacement of failed containers
5. **Service Discovery:** Built-in DNS-based service discovery and load balancing
6. **Resource Management:** Fine-grained control over CPU and memory allocation
7. **Advanced Networking:** Support for network policies, ingress controllers, and service meshes
8. **Secret Management:** Secure handling of sensitive configuration data
9. **Health Monitoring:** Sophisticated health checking with liveness, readiness, and startup probes
10. **Enterprise Ecosystem:** Integration with monitoring, logging, and service mesh platforms

Running the Application Locally

You can run the application locally using three different approaches, each with increasing levels of sophistication:

Option 1: Standalone Spring Boot with H2 Database

The simplest approach for development, using an in-memory database:

```
# Navigate to the product service directory
cd <SOURCE_DIR>\twelve-factor\ecommerce-microservices\product-service

# Build the application
mvn clean package -DskipTests

# Run with default profile (using H2 database)
java -jar target/product-service.jar
```

Option 2: Docker Compose (Recommended for Local Development)

The balanced approach that provides the full stack with minimal setup:

```
# Navigate to the root directory
cd <SOURCE_DIR>\twelve-factor\ecommerce-microservices

# Start all services
docker-compose up

# Access the application at http://localhost:8080/api
# Access Kafka UI at http://localhost:8090
```

Option 3: Local Kubernetes with Docker Desktop (Recommended for Windows)

For testing the full Kubernetes deployment locally, the simplest approach is to use Docker Desktop's built-in Kubernetes:

Setting Up Docker Desktop Kubernetes

1. Enable Kubernetes in Docker Desktop:

- Open Docker Desktop application
- Click on the gear icon (Settings) in the top right
- Select "Kubernetes" from the left sidebar
- Check the box for "Enable Kubernetes"
- Click "Apply & Restart"
- Wait for Docker Desktop to install and configure Kubernetes (this may take several minutes)
- You'll see a green Kubernetes icon in the bottom status bar when it's ready

2. Install kubectl (if not already installed):

```
# With Chocolatey package manager
choco install kubernetes-cli

# Or with winget
winget install kubectl
```

3. Verify Kubernetes is running:

```
kubectl version
kubectl get nodes
```

You should see one node running (your local Docker Desktop).

Deploying the Application to Kubernetes

```
# Navigate to Kubernetes configuration directory
cd <SOURCE_DIR>\twelve-factor\ecommerce-microservices\kubernetes

# Apply all Kubernetes manifests
kubectl apply -k .

# Verify all pods are running (this may take a few minutes for all pods to start)
kubectl get pods -n ecommerce

# Forward ports for local access
kubectl port-forward -n ecommerce svc/product-service-service 8080:80
kubectl port-forward -n ecommerce svc/kafka-ui-service 8090:80
```

Alternative: Using Minikube

If you prefer using Minikube instead of Docker Desktop Kubernetes:

1. Install Minikube:

```
# With Chocolatey
choco install minikube

# Or with winget
winget install minikube
```

2. Start Minikube:

```
minikube start --driver=docker
```

3. Deploy and access as described above with kubectl commands

Accessing Kubernetes Deployed Services

Port Forwarding for Local Access

Kubernetes services run inside the cluster and are not automatically accessible outside it. To access the services from your local machine, you can use port forwarding. This is particularly useful during development and testing.

Port Forwarding for the Product Service API

To access the Spring Boot REST API on your local machine:

```
# In a PowerShell terminal (keep this running)
kubectl port-forward -n ecommerce svc/product-service-service 8080:80
```

This command forwards your local port 8080 to port 80 of the product-service-service in the Kubernetes cluster. You'll see output similar to:

```
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
Handling connection for 8080
```

The service is now accessible at <http://localhost:8080/api/v1/products> in your web browser or API client.

Port Forwarding for Kafka UI

To access the Kafka UI on your local machine:

```
# In another PowerShell terminal (keep this running)
kubectl port-forward -n ecommerce svc/kafka-ui-service 8090:80
```

This forwards your local port 8090 to port 80 of the kafka-ui-service. You can access the Kafka UI at <http://localhost:8090>.

Port Forwarding for PostgreSQL

To connect to PostgreSQL with a database client:

```
kubectl port-forward -n ecommerce svc/postgres-service 5432:5432
```

You can now connect to PostgreSQL at localhost:5432 with username/password: postgres/postgres.

Multiple Port Forwards

You can run multiple port-forward commands simultaneously in different terminals to access all services at once.

Testing the Application

API Testing

With port forwarding set up for the product service (8080:80), you can test the REST API using curl, Postman, or any HTTP client:

List Products

```
curl -X GET http://localhost:8080/api/v1/products
```

Expected response: An array of products (empty if no products are added yet)

Create a Product

```
curl -X POST http://localhost:8080/api/v1/products \
-H "Content-Type: application/json" \
-d '{
  "name": "Test Product",
  "description": "A test product",
  "sku": "TEST-001",
  "price": 19.99,
  "category": "electronics",
  "inventoryLevel": 100
}'
```

Expected response: The created product with an assigned ID

Update a Product

```
curl -X PUT http://localhost:8080/api/v1/products/1 \
-H "Content-Type: application/json" \
-d '{
  "name": "Updated Product",
  "description": "An updated test product",
  "sku": "TEST-001",
  "price": 29.99,
  "category": "electronics",
  "inventoryLevel": 150
}'
```

Expected response: The updated product details

Delete a Product

```
curl -X DELETE http://localhost:8080/api/v1/products/1
```

Expected response: HTTP 204 No Content

GUI Testing

For easier testing without command-line tools:

1. Open your web browser to <http://localhost:8080/api/v1/products>
2. Use a browser extension like "RestMan" for Chrome or "RESTer" for Firefox
3. Or use standalone tools like Postman or Insomnia

Event Verification

To verify that events are properly published to Kafka (when Kafka is fully operational):

1. With port forwarding active for Kafka UI (8090:80), access <http://localhost:8090>
2. Navigate to the Topics section
3. Check that the following topics exist and contain messages after API operations:
 - product-created
 - product-updated
 - product-deleted
 - inventory-updated

Troubleshooting Kafka

If you don't see brokers in the Kafka UI, check if the Kafka and Zookeeper pods are running properly:

```
kubectl get pods -n ecommerce
```

Common issues and solutions:

- CrashLoopBackOff: Check pod logs with `kubectl logs -n ecommerce <pod-name>`
- Connection issues: Ensure services are properly named and discoverable

- For production environments, consider using Strimzi Kafka Operator instead of direct containers

Monitoring Application Health

Check the application's health status using Spring Boot Actuator endpoints:

```
curl http://localhost:8080/actuator/health
```

Expected response: `{"status":"UP"}` if the application is healthy

View detailed health information:

```
curl http://localhost:8080/actuator/health/details
```

Testing in CI/CD Pipelines

For automated testing in CI/CD pipelines, you can use `kubectl` in scripts:

```
# Wait for deployment to be ready
kubectl wait --for=condition=available --timeout=300s deployment/product-service -n
ecommerce

# Get the service endpoint
SERVICE_IP=$(kubectl get svc product-service-service -n ecommerce -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')

# Run tests against the endpoint
curl -X GET http://$SERVICE_IP/api/v1/products
```

Stopping and Cleaning Kubernetes Resources

When you're done working with your Kubernetes environment, you can clean up resources and stop components using several methods.

Option 1: Delete the Entire Namespace

The quickest way to remove all resources from your deployment is to delete the namespace:

```
# This removes all resources in the ecommerce namespace
kubectl delete namespace ecommerce
```

You'll see confirmation: `namespace "ecommerce" deleted`

Option 2: Scale Down Individual Deployments


To temporarily stop specific components without removing their configurations:

```
# Scale deployments to zero replicas
kubectl scale deployment product-service -n ecommerce --replicas=0
kubectl scale deployment postgres -n ecommerce --replicas=0
kubectl scale deployment kafka -n ecommerce --replicas=0
kubectl scale deployment zookeeper -n ecommerce --replicas=0
kubectl scale deployment kafka-ui -n ecommerce --replicas=0
```

This approach keeps all configurations intact, making it easy to scale back up later.

Option 3: Disable Kubernetes in Docker Desktop

To completely stop Kubernetes while keeping Docker running:

1. Open Docker Desktop
2. Click the gear icon () in the top right to open Settings
3. In the left sidebar, select "Kubernetes"
4. Uncheck the option "Enable Kubernetes"
5. Click "Apply & Restart"

This will free up system resources while keeping Docker available.

Option 4: Shut Down Docker Desktop Completely

For maximum resource recovery:

1. Right-click the Docker icon in your system tray
2. Select "Quit Docker Desktop"

This will stop both Docker and Kubernetes completely.

Kubernetes Deployment for Production

For production deployment to a managed Kubernetes service like AKS, EKS, or GKE:

1. Build and push Docker images to a container registry:

```
docker build -t your-registry/product-service:1.0.0 .  
docker push your-registry/product-service:1.0.0
```

2. Update the image reference in `12-product-service-deployment.yaml`:

```
image: your-registry/product-service:1.0.0
```

3. Configure domain name and TLS in the ingress configuration
4. Apply the configuration to your cluster:

```
kubect1 apply -k .
```

Monitoring and Observability

Our Kubernetes configuration supports integration with monitoring tools:

1. **Prometheus**: Scrapes metrics from the `/actuator/prometheus` endpoint
2. **Grafana**: Visualizes metrics with pre-configured dashboards

3. **ELK Stack:** Collects and analyzes application logs
4. **Jaeger/Zipkin:** Captures distributed traces for request tracking

Security Considerations

The Kubernetes configuration implements several security best practices:

1. **Secret Management:** Sensitive data stored in Kubernetes Secrets
2. **Non-root User:** Application container runs as non-root user
3. **Resource Limits:** Prevent resource exhaustion attacks
4. **Network Policies:** Can be added to restrict pod-to-pod communication
5. **TLS Termination:** Configured at the ingress level

High Availability Configuration

For production deployments, consider these additional configurations:

1. **Multi-replica Deployments:** Critical services run multiple replicas
2. **Pod Disruption Budgets:** Ensure minimum available instances during maintenance
3. **Pod Anti-affinity Rules:** Distribute replicas across nodes
4. **Topology Spread Constraints:** Ensure pods are distributed across failure domains

Troubleshooting

Common troubleshooting commands:

```
# Check pod status
kubectl get pods -n ecommerce

# View pod logs
kubectl logs -n ecommerce pod/pod-name

# Describe a pod for events and status
kubectl describe pod -n ecommerce pod-name

# Execute commands in a container
kubectl exec -it -n ecommerce pod/pod-name -- /bin/sh
```

```
# View configmaps and secrets
kubectl get configmap -n ecommerce
kubectl get secret -n ecommerce
```