# Kubernetes Troubleshooting Journey - Real-World Experience

This document captures our journey troubleshooting and fixing a Spring Boot microservice deployment in Kubernetes. The experience demonstrates common challenges with containerized applications and provides solutions that can be applied to similar situations.

## 🔍 Initial Problem Assessment

Our ecommerce microservices application was experiencing several critical issues:

1. **Spring Boot Application Failures**: The product service pods were entering `CrashLoopBackOff` state
2. **Database Connectivity Issues**: PostgreSQL connection failures with authentication errors
3. **Kafka Configuration Problems**: Bean dependency issues in the message broker setup
4. **LoadBalancer Misconfiguration**: External access to the API was not working properly

## 🛠 Step 1: Diagnosing Kafka Configuration Issues

### Problem

The initial error logs showed:

```
org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating
bean with name 'productEventPublisher'...
Caused by: java.lang.IllegalArgumentException: Could not resolve placeholder
'spring.kafka.producer.client-id' in value "${spring.kafka.producer.client-id}"
```

### Solution

We created `fix_kafka_config.ps1` to address the missing Kafka configuration properties:

```
# Add all required Kafka producer properties to the deployment
kubectl set env deployment/product-service-final -n ecommerce \
  SPRING_KAFKA_PRODUCER_CLIENT_ID=product-service-client \
  SPRING_KAFKA_PRODUCER_BATCH_SIZE=16384 \
  SPRING_KAFKA_PRODUCER_ACKS=all \
  SPRING_KAFKA_PRODUCER_RETRIES=3 \
  SPRING_KAFKA_PRODUCER_BUFFER_MEMORY=33554432 \
  SPRING_KAFKA_PRODUCER_COMPRESSION_TYPE=snappy
```

## Result

While this addressed some configuration issues, we still had problems with the Kafka setup. Instead of trying to fix all Kafka properties, we decided to disable Kafka entirely for our deployment since it wasn't immediately required for the core application functionality.

# 🛠 Step 2: Addressing PostgreSQL Connection Issues

## Problem

After disabling Kafka, we encountered PostgreSQL connection errors:

```
java.net.UnknownHostException: postgres-fixed
```

## Analysis

We discovered:

1. The service was looking for a hostname `postgres-fixed` that didn't match any available PostgreSQL service
2. We needed to point to the correct PostgreSQL service

## Solution

We created a script `fix_db_connection.ps1` to target the correct PostgreSQL service:

```
# Add all required PostgreSQL connection properties
kubectl set env deployment/product-service-fixed -n ecommerce \
  SPRING_DATASOURCE_URL=jdbc:postgresql://postgres-service:5432/product_db \
  SPRING_DATASOURCE_USERNAME=postgres \
  SPRING_DATASOURCE_PASSWORD=YOUR_DB_PASSWORD
```

### Result

The service could resolve the PostgreSQL hostname, but we still encountered authentication issues.

## 🛠 Step 3: PostgreSQL Authentication Failures

### Problem

Even after fixing the hostname, we encountered password authentication failures:

```
FATAL: password authentication failed for user "postgres"
```

### Analysis

We found a mismatch between:

- PostgreSQL ConfigMap password: `YOUR_DB_PASSWORD`
- Application configuration: `YOUR_DB_PASSWORD`
- Actual working password: `YOUR_DB_PASSWORD` (verified from working deployments)

### Solution

We created `fix_postgres_password.ps1` to update the deployment with the correct credentials:

```
# Update with correct database name AND correct password
kubectl set env deployment/product-service-fixed -n ecommerce \
  SPRING_DATASOURCE_URL=jdbc:postgresql://postgres-service:5432/product_db \
  SPRING_DATASOURCE_USERNAME=postgres \
  SPRING_DATASOURCE_PASSWORD=YOUR_DB_PASSWORD
```

## Result

Our product service successfully connected to PostgreSQL and started running properly.

# 🛠 Step 4: Environment Cleanup

## Problem

Multiple deployments and pods cluttered the environment, making troubleshooting difficult:

- Multiple PostgreSQL instances
- Redundant product service deployments
- Diagnostic pods that were no longer needed

## Analysis

We identified which resources were critical to maintain and which could be safely removed.

## Solution

We created cleanup scripts:

1. `cleanup_deployments.ps1` to remove failed deployments:

```
# Delete the failing product-service-final deployment
kubectl delete deployment product-service-final -n ecommerce
```

2. `cleanup_minimal.ps1` to standardize on our fixed deployment:

```
# Delete the product-service-minimal deployment
kubectl delete deployment product-service-minimal -n ecommerce
```

3. `cleanup_postgres.ps1` to remove redundant database instances:

```
# Delete redundant PostgreSQL deployments, but preserve postgres-fixed
kubectl delete deployment postgres-clean -n ecommerce --ignore-not-found
kubectl delete deployment postgres-test -n ecommerce --ignore-not-found
kubectl delete deployment postgres-nopvc -n ecommerce --ignore-not-found
```

## Result

We successfully simplified our environment to include only the essential and working components.

# 🛠 Step 5: LoadBalancer Configuration

## Problem

After our cleanup, the external API endpoint stopped working:

```
http://aee62280f41e04181bf13ba432fd2092-1458733407.us-west-
2.elb.amazonaws.com/api/products
```

## Analysis

We discovered two issues with the LoadBalancer:

1. The selector was pointing to `app=product-service` but our working deployment used `app=product-service-fixed`
2. The LoadBalancer was targeting port 8080, but our application ran on 8081

## Solution

We created `fix_loadbalancer.ps1` to update the LoadBalancer configuration:

```
# Patch the LoadBalancer service to point to our fixed product service
kubectl patch service product-service-loadbalancer -n ecommerce --type='json' -p='[
  {"op": "replace", "path": "/spec/selector", "value": {"app": "product-service-
fixed"}},
  {"op": "replace", "path": "/spec/ports/0/targetPort", "value": 8081}
]'
```

## Result

The LoadBalancer correctly routed external traffic to our working product service, making the API accessible again.

## 📋 Verification Scripts

We also created verification scripts to monitor and test our application:

1. `verify_service.ps1` - To verify the service and database connectivity:

```
# Test the health endpoint from inside the pod
kubectl exec -it $podName -n ecommerce -- curl -s
http://localhost:8081/actuator/health
```

2. `test_and_monitor.ps1` - For comprehensive monitoring and testing:

```
# Set up port forwarding to access the product service
kubectl port-forward $productPod 8081:8081 -n ecommerce
# Test API endpoints
Invoke-RestMethod -Uri "http://localhost:8081/actuator/health"
```

## 🧠 Key Learnings and Best Practices

1. **Configuration Management**

    - Always ensure environment variables match your infrastructure configuration
    - Use ConfigMaps and Secrets properly to manage sensitive information
    - Document all configuration changes for future reference

2. **Troubleshooting Methodology**
    - Start with focused diagnostic scripts
    - Look at logs carefully for root causes
    - Verify each fix incrementally

3. **Kubernetes Best Practices**
    - Clean up unused resources to avoid confusion
    - Use proper selectors and labels consistently
    - Implement readiness and liveness probes for automatic healing

4. **Database Connectivity**
    - Verify service names match what applications expect
    - Ensure credentials are consistent across configurations
    - Check database name matches application expectations

5. **External Access Configuration**
    - Verify LoadBalancer selectors match deployment labels
    - Ensure port configurations match application settings
    - Allow time for LoadBalancer changes to propagate

## 📊 Final Environment State

Our final environment consists of:

- **Single Product Service**: `product-service-fixed` deployment
- **Single PostgreSQL Instance**: `postgres-fixed` deployment
- **Properly Configured LoadBalancer**: Correctly routing to our product service
- **API Endpoint**: Functional and accessible externally

## 📝 Scripts Overview

| Script | Purpose | Key Actions |
|---|---|---|
| `fix_kafka_config.ps1` | Fix Kafka configuration | Set Kafka environment variables |
| `disable_kafka.ps1` | Disable Kafka completely | Set `SPRING_PROFILES_ACTIVE=dev` and `SPRING_KAFKA_ENABLED=false` |
| `fix_db_connection.ps1` | Fix PostgreSQL connection | Update database connection parameters |
| `fix_postgres_password.ps1` | Fix PostgreSQL authentication | Set correct database credentials |
| `fix_product_service.ps1` | Deploy fixed product service | Create properly configured deployment |
| `cleanup_deployments.ps1` | Remove redundant deployments | Delete failed product service deployment |
| `cleanup_minimal.ps1` | Standardize on fixed deployment | Remove working but redundant deployment |
| `cleanup_postgres.ps1` | Clean up PostgreSQL deployments | Remove multiple database instances |
| `fix_loadbalancer.ps1` | Fix external API access | Update LoadBalancer configuration |
| `verify_service.ps1` | Verify service functionality | Test API endpoints and database connectivity |
| `test_and_monitor.ps1` | Comprehensive monitoring | Test and monitor all components |

This troubleshooting journey demonstrates the complexity of working with containerized applications and the importance of systematic problem-solving in Kubernetes environments. The scripts and lessons learned here can be applied to similar issues in other deployments.