# Docker Interview Guide - Seal Wallet Containerization


- Docker Interview Guide - Seal Wallet Containerization
  - For Brac Bank Technical Interview
  - Table of Contents
  - 1. Docker Fundamentals
  - 2. Containerization Strategy
  - 3. Docker Components Deep Dive
  - 4. Multi-Container Architecture
  - 5. Production Considerations
  - 6. Common Interview Questions
  - Docker Commands Reference
  - Key Takeaways for Banking Interview


# Docker Interview Guide - Seal Wallet Containerization

## For Brac Bank Technical Interview

---

## Table of Contents


1. Docker Fundamentals
2. Containerization Strategy
3. Docker Components Deep Dive
4. Multi-Container Architecture
5. Production Considerations
6. Common Interview Questions


---

## 1. Docker Fundamentals

### Q1: What is Docker and why did you choose it for your banking application?

**Answer**: Docker is a containerization platform that packages applications with their dependencies into lightweight, portable containers. I chose Docker for the Seal Wallet application because:

1. **Consistency**: "Works on my machine" → "Works everywhere"
2. **Isolation**: Application runs in isolated environment
3. **Portability**: Deploy anywhere (dev, test, prod, cloud)
4. **Scalability**: Easy horizontal scaling for banking workloads
5. **DevOps Integration**: Essential for modern CI/CD pipelines
6. **Resource Efficiency**: Better than VMs for microservices

### Q2: Explain the difference between Image, Container, and Service.

**Answer**:

| Component | Definition | Analogy | Example |
|---|---|---|---|
| **Image** | Blueprint/Template | Class in OOP | `seal-wallet:latest` |
| **Container** | Running instance | Object in OOP | Running Spring Boot app |
| **Service** | Logical grouping | Service in architecture | Database service, App service |

```
# Image: Template
docker build -t seal-wallet .

# Container: Running instance
docker run -p 8080:8080 seal-wallet

# Service: Logical grouping (Docker Compose)
services:
  seal-app:    # Service name
    image: seal-wallet
```

### Q3: What's the difference between Docker and Virtual Machines?

**Answer**:

| Aspect | Virtual Machine | Docker Container |
|---|---|---|
| **OS** | Full guest OS | Shares host OS kernel |
| **Size** | GBs (heavy) | MBs (lightweight) |
| **Startup** | Minutes | Seconds |
| **Resource Usage** | High overhead | Minimal overhead |
| **Isolation** | Hardware-level | Process-level |
| **Use Case** | Different OS needs | Same OS, different apps |

**For Banking**: Containers are perfect for microservices architecture where we need multiple services (auth, payments, notifications) running efficiently.

---

## 2. Containerization Strategy

### Q4: How did you containerize your Spring Boot application?

**Answer**: I used a **multi-stage approach** optimized for production:

```
# Production-ready Dockerfile
FROM eclipse-temurin:21-jre-alpine

# Security: Create non-root user
RUN addgroup -g 1001 -S seal && adduser -u 1001 -S seal -G seal

WORKDIR /app
COPY target/seal-*.jar app.jar
RUN chown seal:seal app.jar

# Run as non-root (Kubernetes requirement)
USER seal

EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

**Key Decisions**: 1. **Base Image**: `eclipse-temurin:21-jre-alpine` (official OpenJDK, lightweight) 2. **Security**: Non-root user (banking security requirement) 3. **Size**: Alpine Linux (5MB vs 100MB+ for Ubuntu) 4. **JRE vs JDK**: Runtime only (smaller, more secure)

### Q5: Why did you choose Eclipse Temurin over other Java images?

**Answer**:

| Image | Pros | Cons | Verdict |
|---|---|---|---|
| **eclipse-temurin** | Official OpenJDK, well-maintained | Newer option | ✅ **Chosen** |
| **openjdk** | Popular, familiar | Deprecated/unmaintained | ❌ Avoid |
| **amazoncorretto** | AWS-optimized | Vendor lock-in | ⚠️ Cloud-specific |

**Banking Consideration**: Eclipse Temurin is backed by the Eclipse Foundation and provides long-term support, crucial for financial applications.

### Q6: How do you handle application configuration in containers?

**Answer**: I use **externalized configuration** following 12-Factor App principles:

```
# Docker Compose environment variables
environment:
  SPRING_DATASOURCE_URL: jdbc:postgresql://postgres:5432/seal_db
  SPRING_DATASOURCE_USERNAME: seal_admin
  SPRING_DATASOURCE_PASSWORD: seal
  SPRING_PROFILES_ACTIVE: docker
  JWT_SECRET: ${JWT_SECRET}  # From external secrets
```

**Benefits**: - **Security**: Sensitive data externalized - **Flexibility**: Different configs for dev/test/prod - **Immutable**: Same image across environments

---

## 3. Docker Components Deep Dive

### Q7: Explain Docker Networks and how containers communicate.

**Answer**: Docker networks enable container-to-container communication:

```
# Custom network creation
networks:
  seal-network:
    driver: bridge
```

**Network Types**: 1. **Bridge** (default): Containers on same host 2. **Host**: Container uses host network 3. **Overlay**: Multi-host communication 4. **None**: No networking

**Container Communication**:

```
# App connects to database by service name
SPRING_DATASOURCE_URL: jdbc:postgresql://postgres:5432/seal_db
#                                        ↑
#                                  Service name (not IP)
```

**How it works**: - Docker provides internal DNS - Service names resolve to container IPs - Automatic load balancing for scaled services

### Q8: What are Docker Volumes and why are they important for banking applications?

**Answer**: Volumes provide **persistent storage** that survives container restarts:

```
volumes:
  # Named volume for database persistence
```

```
  postgres_data:/var/lib/postgresql/data
```

**Volume Types**: 1. **Named Volumes**: Managed by Docker 2. **Bind Mounts**: Host directory mapping 3. **tmpfs**: In-memory storage

**Banking Importance**: - **Data Persistence**: Transaction data survives container restarts - **Backup**: Volumes can be backed up independently - **Performance**: Optimized for database workloads - **Security**: Encrypted volumes for sensitive data

### Q9: How do you manage secrets in Docker containers?

**Answer**: Multiple approaches for different environments:

**Development**:

```yaml
environment:
  POSTGRES_PASSWORD: seal  # Simple for dev
```

**Production**:

```yaml
environment:
  POSTGRES_PASSWORD_FILE: /run/secrets/db_password
secrets:
  db_password:
    external: true
```

**Best Practices**: 1. **Never in Dockerfile**: Secrets not baked into images 2. **Environment Variables**: For non-sensitive config 3. **Docker Secrets**: For production secrets 4. **External Systems**: HashiCorp Vault, AWS Secrets Manager

---

## 4. Multi-Container Architecture

### Q10: Explain your Docker Compose setup and why you structured it this way.

**Answer**: I created **environment-specific** compose files:

```
docker-compose.dev.yml      # Development (app + database)
docker-compose.app-only.yml # CI/CD (app only)
docker-compose.prod.yml     # Production (external database)
```

**Architecture Benefits**: 1. **Separation of Concerns**: Database vs Application lifecycle 2. **Environment Parity**: Consistent across dev/test/prod 3. **Scalability**: Can scale services independently 4. **Maintainability**: Clear service boundaries

### Q11: How do you handle service dependencies in Docker Compose?

**Answer**: Using **health checks** and **dependency management**:

```yaml
postgres:
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U seal_admin -d seal_db"]
    interval: 30s
    timeout: 10s
    retries: 3

seal-app:
  depends_on:
    postgres:
      condition: service_healthy  # Wait for healthy database
```

**Why This Matters**: - **Prevents Race Conditions**: App doesn't start before database - **Graceful Startup**: Proper initialization order - **Reliability**: Automatic retries on failure

### Q12: How would you scale your containerized application?

**Answer**: Multiple scaling strategies:

**Horizontal Scaling**:

```
# Scale app service to 3 instances
docker-compose up --scale seal-app=3
```

**Load Balancing**:

```
nginx:
  image: nginx
  ports:
    - "80:80"
  depends_on:
    - seal-app
```

**Database Scaling**: - **Read Replicas**: For read-heavy workloads - **Connection Pooling**: Already implemented with HikariCP - **Managed Databases**: AWS RDS, Azure Database

---

## 5. Production Considerations

### Q13: How do you ensure security in containerized banking applications?

**Answer**: Multi-layered security approach:

**Container Security**:

```
# Non-root user
USER seal

# Minimal base image
FROM eclipse-temurin:21-jre-alpine

# No unnecessary packages
RUN apk add --no-cache curl
```

**Runtime Security**: - **Read-only containers**: Immutable runtime - **Resource limits**: Prevent resource exhaustion - **Network policies**: Restrict container communication - **Image scanning**: Vulnerability detection

**Banking-Specific**: - **Secrets management**: External secret stores - **Audit logging**: Container activity monitoring - **Compliance**: PCI DSS, SOX requirements

### Q14: How do you monitor containerized applications?

**Answer**: Comprehensive monitoring strategy:

**Health Checks**:

```
HEALTHCHECK --interval=30s --timeout=3s \
  CMD curl -f http://localhost:8080/actuator/health || exit 1
```

**Logging**:

```
# Centralized logging
docker-compose logs -f seal-app
```

**Metrics**: - **Container metrics**: CPU, memory, network - **Application metrics**: Spring Boot Actuator - **Business metrics**: Transaction volume, response times

**Tools**: - **Prometheus + Grafana**: Metrics and dashboards - **ELK Stack**: Centralized logging - **Jaeger**: Distributed tracing

### Q15: How do you handle database migrations in containerized environments?

**Answer**: Automated migration strategy:

**Development**:

```
volumes:
  - ./database/tables.sql:/docker-entrypoint-initdb.d/init.sql
```

**Production**:

```
# Init container pattern
init-db:
  image: flyway/flyway
  command: migrate
  volumes:
    - ./migrations:/flyway/sql
```

**Best Practices**: 1. **Version Control**: All migrations in Git 2. **Rollback Strategy**: Backward-compatible changes 3. **Testing**: Migration testing in staging 4. **Monitoring**: Migration success/failure tracking

---

## 6. Common Interview Questions

### Q16: Walk me through your Docker build process.

**Answer**: Step-by-step build process:

```
# 1. Build application JAR
./mvnw clean package

# 2. Build Docker image
docker build -t seal-wallet:latest ./seal

# 3. Tag for registry
docker tag seal-wallet:latest registry.company.com/seal-wallet:v1.0.0

# 4. Push to registry
docker push registry.company.com/seal-wallet:v1.0.0
```

**Build Optimization**: - **Layer caching**: Optimize Dockerfile layer order - **Multi-stage builds**: Separate build and runtime - **Base image updates**: Regular security updates

### Q17: How do you troubleshoot container issues?

**Answer**: Systematic troubleshooting approach:

**Container Status**:

```
docker ps -a                    # Check container status
```

```
docker logs seal-app-dev        # View application logs
docker inspect seal-app-dev     # Detailed container info
```

**Resource Issues**:

```
docker stats                    # Resource usage
docker system df                # Disk usage
docker system prune             # Cleanup unused resources
```

**Network Issues**:

```
docker network ls               # List networks
docker exec -it seal-app-dev ping postgres  # Test connectivity
```

**Common Issues**: 1. **Port conflicts**: Change host ports 2. **Memory limits**: Increase container memory 3. **Network isolation**: Check network configuration 4. **Volume permissions**: Fix file ownership

## Q18: How does your containerization support CI/CD?

**Answer**: Container-native CI/CD pipeline:

**Build Stage**:

```
# Jenkins pipeline
docker build -t seal-wallet:${BUILD_NUMBER} .
```

**Test Stage**:

```
# Run tests in container
docker run --rm seal-wallet:${BUILD_NUMBER} ./mvnw test
```

**Deploy Stage**:

```
# Deploy to staging/production
docker-compose -f docker-compose.prod.yml up -d
```

**Benefits**: - **Consistency**: Same container across pipeline stages - **Isolation**: Tests run in clean environment - **Speed**: Fast container startup - **Rollback**: Easy version management

## Q19: How would you implement blue-green deployment with containers?

**Answer**: Zero-downtime deployment strategy:

```
# Blue environment (current)
seal-app-blue:
  image: seal-wallet:v1.0.0
  ports:
    - "8080:8080"

# Green environment (new version)
seal-app-green:
  image: seal-wallet:v1.1.0
  ports:
    - "8081:8080"
```

**Deployment Process**: 1. **Deploy Green**: New version on different port 2. **Health Check**: Verify green environment 3. **Switch Traffic**: Load balancer points to green 4. **Monitor**: Watch for issues 5. **Cleanup**: Remove blue environment

## Q20: What are the challenges of running databases in

### containers?

**Answer**: Database containerization considerations:

**Challenges**: 1. **Data Persistence**: Volumes required for data survival 2. **Performance**: I/O overhead compared to bare metal 3. **Backup/Recovery**: Container-aware backup strategies 4. **Clustering**: Complex multi-node setups

**Solutions**:

```yaml
postgres:
  volumes:
    - postgres_data:/var/lib/postgresql/data  # Persistent storage
  deploy:
    resources:
      limits:
        memory: 2G                            # Resource limits
      reservations:
        memory: 1G
```

**Banking Recommendation**: Use **managed databases** (AWS RDS) for production, containers for development/testing.

---

# Docker Commands Reference

## Essential Commands

```bash
# Images
docker build -t seal-wallet .
docker images
docker rmi seal-wallet

# Containers
docker run -p 8080:8080 seal-wallet
docker ps
docker stop <container-id>
docker rm <container-id>

# Docker Compose
docker-compose up
docker-compose down
docker-compose logs seal-app
docker-compose exec seal-app bash

# Cleanup
docker system prune
docker volume prune
docker network prune
```

## Debugging Commands

```bash
# Inspect container
docker inspect seal-app-dev
docker exec -it seal-app-dev /bin/sh
docker logs -f seal-app-dev

# Resource monitoring
docker stats
docker system df
```

---

# Key Takeaways for Banking Interview

1. **Security First**: Non-root users, minimal images, secret management
2. **Production Ready**: Health checks, monitoring, logging
3. **Scalability**: Horizontal scaling, load balancing
4. **Reliability**: Dependency management, graceful shutdowns
5. **Compliance**: Audit trails, immutable infrastructure

**Elevator Pitch**: *"I containerized the Seal Wallet application using Docker with a security-first approach, implementing non-root users, health checks, and multi-environment configurations. The setup supports both development workflows and production CI/CD pipelines, following banking industry best practices for containerized applications."*

---

**Prepared by**: [Your Name]
**Date**: January 15, 2026
**Project**: Seal Wallet Containerization