Hands On Enterprise Project

Seifeddine Jemaa

February 11, 2025

🚀You'are aiming to build a **JEE Enterprise Application** in **Docker**, using **Continuous Delivery(CD)**, a **well-structured deployment pipeline**, and following **Test-Driven Development(TDD)**. Here's a detailed *step-by-step roadmap* to achieve this.

1 📁 Step 1: Setup the Project Repository

We'll use **Maven** for dependency management, **JUnit/Arquillian** for testing, **Docker** for containerization, and **Jenkins** for Continuous Delivery.

📌 Repository Structure

This structure (figure 1) keeps **code**, **infrastructure**, and **automation separate** while being modular.

bash Copy Edit

```bash
jee-enterprise-app/
|── app/                                # Main JEE application
|   ├── src/main/java/com/example/
|   |   ├── config/                     # Configuration classes
|   |   ├── controller/                 # REST controllers
|   |   ├── service/                    # Business logic
|   |   ├── repository/                 # Database access
|   |   ├── model/                      # JPA entities
|   ├── src/test/java/com/example/      # TDD tests (JUnit, Arquillian)
|   ├── pom.xml                         # Build configuration
|── infra/                              # Deployment scripts
|   ├── docker/                         # Docker images & compose
|   ├── kubernetes/                     # Kubernetes deployment files
|   ├── terraform/                      # IaC for cloud provisioning
|── cicd/                               # CI/CD pipelines
|   ├── Jenkinsfile                     # Jenkins pipeline
|   ├── github-actions.yml              # GitHub Actions pipeline
|   ├── gitlab-ci.yml                   # GitLab CI pipeline
|── README.md                           # Documentation
|── .gitignore                          # Git ignore file
```

Figure 1: Repository Structure

2 </> Step 2: Implement the Application (TDD Approach)

1. **Define Business Requirements**
   - **User Management** (Login, Registration, Roles)
   - **Product Management** (CRUD APIs)
   - **Order Processing** (Place Order, Payments)
   - **Admin Dashboard** (Monitoring & Analytics)
2. **Write Failing Unit Tests (TDD)** Example: Test user registration service.
   - Use **JUnit5 + Mockito** for testing.
   - Use **Arquillian** for integration testing.

```java
@Test
void testUserRegistration() {
    User user = new User("john_doe", "password");
    when(userRepository.save(any(User.class))).thenReturn(user);

    User registeredUser = userService.register(user);

    assertNotNull(registeredUser);
    assertEquals("john_doe", registeredUser.getUsername());
}
```

Figure 2: Test user registration service.

3. **Implement the Feature**
   - Create **JPA entities** for database interaction.
   - Implement **RESTful APIs** with **JAX-RS** or **Spring Boot**.
   - Secure API with **JWT/OAuth2**.
4. **Refactor Code Make Tests Pass**
   - Run *mvn test* after each change.
   - Ensure **100% test coverage** (use JaCoCo).

3 Step 3: Dockerize the Application

1. **Create Dockerfile**
2. **Create *docker-compose.yml***
   - **Ensures reproducibility** across environments.
   - **Run everything locally** using **docker-compose up**

```dockerfile
FROM openjdk:17-jdk
WORKDIR /app
COPY target/jee-app.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Figure 3: Dockerfile.

```yaml
version: "3.8"
services:
  app:
    build: .
    ports:
      - "8080:8080"
    depends_on:
      - db
  db:
    image: postgres:15
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: password
```

Figure 4: docker-compose.yml

4 🚀 Step 4: Continuous Delivery Pipeline

1. **CI/CD Pipeline Using Jenkins**
   - Triggers on every commit.
   - Runs **unit tests** before deployment.
   - Deploys to **Kubernetes cluster**.

   Jenkinsfile

5 🔧 Step 5: Kubernetes Deployment

1. **Create *deployment.yaml***
2. **Deploy to Kubernetes**

```groovy
pipeline {
    agent any
    stages {
        stage('Checkout') {
            steps { git 'https://github.com/your-repo.git' }
        }
        stage('Build') {
            steps { sh 'mvn clean package' }
        }
        stage('Test') {
            steps { sh 'mvn test' }
        }
        stage('Docker Build') {
            steps { sh 'docker build -t myapp:latest .' }
        }
        stage('Deploy to Kubernetes') {
            steps { sh 'kubectl apply -f kubernetes/deployment.yaml' }
        }
    }
}
```

Figure 5: Jenkinsfile.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jee-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: jee-app
  template:
    metadata:
      labels:
        app: jee-app
    spec:
      containers:
        - name: jee-app
          image: myapp:latest
          ports:
            - containerPort: 8080
```

Figure 6: deployment.yaml

```sh
kubectl apply -f kubernetes/deployment.yaml
kubectl get pods
```

Figure 7: Deployment command lines.

6 🔍 Step 6: Automated Testing & Monitoring

1. **Automated Testing**
   - Run **integration tests** in a Dockerized environment.

```sh
mvn test
```

Figure 8: Run integration tests.

   - Run **API tests** using **Postman/Newman**.

```sh
newman run api-tests.json
```

Figure 9: Run API Tests using Newman.

2. **Monitoring**
   - Use **Prometheus + Grafana** for application monitoring.
   - Use **ELK Stack (Elasticsearch, Logstash, Kibana)** for log aggregation.