Deployment, Testing, and Real-World Practices in Spring Boot

---

1. Software Testing Approaches in Spring Boot

Testing is a **core part of real-world software development**. Spring Boot provides built-in support for **unit testing, integration testing, and slice testing** with JUnit and Mockito.

1.1 Unit Testing

- Focuses on testing **individual classes or methods** in isolation.
- Common tools: **JUnit 5** and **Mockito**.

Example: Testing Service Layer

```
@SpringBootTest
public class UserServiceTest {

    @Autowired
    private UserService userService;

    @MockBean
    private UserRepository userRepository;

    @Test
    void testFindUserByEmail() {
        User mockUser = new User(1L, "Alice", "alice@gmail.com");

Mockito.when(userRepository.findByEmail("alice@gmail.com")).thenReturn
(mockUser);

        User user = userService.findByEmail("alice@gmail.com");
        assertEquals("Alice", user.getName());
    }
```

```
}
```

1.2 Integration Testing

- Tests how multiple components (Controller + Service + Repository) work together.
- Spring Boot annotations:

Annotation	Purpose
`@SpringBootTest`	Full context loading, end-to-end test.
`@WebMvcTest`	Tests only the **Controller layer** with MockMvc.
`@DataJpaTest`	Tests **Repository layer** with in-memory database.

Example: @WebMvcTest

```
@WebMvcTest(UserController.class)
public class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    void testGetUser() throws Exception {
        Mockito.when(userService.findById(1L))
               .thenReturn(new User(1L, "Bob", "bob@gmail.com"));

        mockMvc.perform(get("/users/1"))
               .andExpect(status().isOk())
               .andExpect(jsonPath("$.name").value("Bob"));
    }
```

```
}
```

Example: @DataJpaTest

```
@DataJpaTest
public class UserRepositoryTest {

    @Autowired
    private UserRepository userRepository;

    @Test
    void testSaveUser() {
        User user = new User(null, "Charlie", "charlie@gmail.com");
        User saved = userRepository.save(user);
        assertNotNull(saved.getId());
    }
}
```

---

2. Containerization using Docker

Containerization allows packaging an application with all its dependencies into a **Docker image**.

Workflow (Text Diagram)

`Spring Boot App → Build JAR → Create Dockerfile → Build Image → Run Container`

Steps to Containerize Spring Boot App

1. Build application JAR:

```
mvn clean package -DskipTests
```

2. Create a `Dockerfile`:

```
FROM openjdk:17-jdk-slim
VOLUME /tmp
COPY target/myapp.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

3. Build Docker image:

```
docker build -t my-springboot-app .
```

4. Run container:

```
docker run -p 8080:8080 my-springboot-app
```

---

3. Basic CI/CD Practices

CI/CD (Continuous Integration / Continuous Deployment) automates build, test, and deployment.

Workflow Diagram (Text Form)

```
Developer Commit → GitHub/GitLab → CI/CD Pipeline (Jenkins/GitHub Actions)
→ Build & Test → Docker Image → Deploy (Cloud / Server)
```

Example: GitHub Actions CI/CD for Spring Boot

```
name: Java CI with Maven

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
```

```
    - name: Set up JDK 17
      uses: actions/setup-java@v2
      with:
        java-version: '17'
        distribution: 'adopt'
    - name: Build with Maven
      run: mvn clean install
```

---

4. Cloud Deployment Methods

Spring Boot apps can be deployed to **Heroku, AWS, Azure, or Google Cloud**.

4.1 Deployment on Heroku

1. Install Heroku CLI.
2. Create `Procfile`:

```
web: java -jar target/myapp.jar
```

3. Deploy:

```
git init
heroku create
git push heroku master
heroku open
```

4.2 Deployment on AWS Elastic Beanstalk

1. Package application:

```
mvn clean package
```

2. Install AWS CLI & Elastic Beanstalk CLI.

3. Deploy:

```
eb init
eb create my-springboot-env
eb deploy
```

---

5. Real-World Practices

5.1 Logging

- Use **SLF4J with Logback** (default in Spring Boot).
- Store logs in **files** and integrate with monitoring tools (ELK stack: Elasticsearch, Logstash, Kibana).

Example: application.properties

```
logging.level.org.springframework=INFO
logging.file.name=logs/app.log
```

5.2 Configuration Management

- Use **application.properties / application.yml** for environment configs.
- Externalize sensitive configs using **Spring Cloud Config Server** or **environment variables**.

Profiles Example:

```
spring:
  profiles: dev
  datasource:
    url: jdbc:h2:mem:testdb
---
```

```
spring:
  profiles: prod
  datasource:
    url: jdbc:mysql://prod-db:3306/mydb
```

---

6. Putting It All Together – Workflow

```
Code → Unit Tests (@SpringBootTest) → Integration Tests (@WebMvcTest, @DataJpaTest)
   → Build JAR (Maven/Gradle) → Dockerize → Push Image to Registry
   → CI/CD Pipeline (GitHub Actions/Jenkins) → Deploy on Cloud (Heroku/AWS)
   → Monitor Logs + Manage Configs
```

---

7. Conclusion

- **Testing** ensures quality (unit + integration).
- **Docker** enables portability.
- **CI/CD** automates delivery.
- **Cloud Deployment** makes apps scalable.
- **Logging + Config Management** are essential for real-world maintainability.

Together, these practices ensure **Spring Boot applications are production-ready, reliable, and maintainable**.