17. 운영 및 배포

JAR 배포 / WAR 배포

Spring Boot는 유연한 **배포 옵션**을 제공한다.
주요 방식으로는 **Executable JAR(독립 실행형 JAR)** 와 **WAR(Web Application Archive)** 두 가지가 있다.

각 방식은 **운영 환경**과 **기존 인프라**에 따라 선택하게 되며, 각각의 특성과 장단점이 존재한다.

1️⃣ 개요

배포 방식	특징
JAR 배포	자체 내장 톰캣(Tomcat)/Jetty/Undertow 포함, 독립 실행 가능
WAR 배포	외부 WAS(Web Application Server)에 배포 → Tomcat, WebLogic 등

2️⃣ JAR 배포 📦

특징

- **Spring Boot의 기본 배포 방식** → Embedded Servlet Container(내장 톰캣 등)를 포함한 **독립 실행형 애플리케이션**.
- 별도의 외부 WAS 설치/구성이 필요 없음.
- `java -jar` 명령어로 쉽게 실행 가능 → Cloud Native, Container 기반 환경에 적합 🚀.

Build 설정 예시 (Gradle)

```
plugins {
    id 'org.springframework.boot' version '3.2.4'
    id 'io.spring.dependency-management' version '1.1.4'
    id 'java'
}

bootJar {
    enabled = true
}

bootWar {
    enabled = false
}
```

실행 명령

```
java -jar myapp.jar
```

주요 장점

✅ 간단한 배포/실행 → 운영 환경 단순화
✅ DevOps, Kubernetes, Docker 기반 환경에 최적화
✅ 모든 의존성 포함 → 배포 아티팩트 관리 용이

주요 단점

❌ 외부 WAS 환경에 통합 필요 시 적합하지 않음 (기업 레거시 WAS 요구 시 한계 존재)

3️⃣ WAR 배포 🏛️

특징

- 기존 **WAS(Tomcat, WebLogic, JBoss 등)** 에 배포 가능한 표준 Web Application Archive.
- `ServletContainerInitializer` 기반으로 Spring Boot 초기화 가능.
- **내장 톰캣 사용 안함** → 외부 WAS의 Lifecycle에 따라 구동.

Build 설정 예시 (Gradle)

```
plugins {
    id 'org.springframework.boot' version '3.2.4'
    id 'io.spring.dependency-management' version '1.1.4'
    id 'war'
    id 'java'
}

bootJar {
    enabled = false
}

bootWar {
    enabled = true
}
```

Application 클래스 구성

```
@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application)
{
        return application.sources(MyApplication.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

WAR 배포 시 사용 예시

- Tomcat 9+ `/webapps` 디렉토리에 `myapp.war` 배포
- WebLogic, WebSphere 환경에 통합 배포

주요 장점

✅ 기존 기업용 **전통적 WAS 인프라에 호환성** 우수
✅ **레거시 시스템과 통합 배포** 필요 시 활용 가능

주요 단점

❌ WAS 설치/운영 관리 필요 → DevOps 흐름에는 적합성 떨어짐
❌ 외부 WAS 환경 종속성 존재 → 완전한 Cloud Native 아키텍처에 부적합

4️⃣ 비교표 🗂️

항목	JAR 배포	WAR 배포
배포 방식	`java -jar` 실행형 JAR	외부 WAS 배포용 WAR
내장 서버	Tomcat, Jetty, Undertow 포함	없음 (외부 WAS 필요)
실행 독립성	완전 독립적 실행 가능	외부 WAS 종속
운영 환경	Cloud, Kubernetes, Docker에 최적화	기업 레거시 인프라 환경
DevOps 친화성	매우 우수	낮음
배포 복잡도	매우 단순	상대적으로 복잡
업그레이드 용이성	자체 관리 가능	외부 WAS 버전 제약 고려 필요

5️⃣ 선택 가이드 🎯

상황	권장 방식
신규 서비스 개발, 클라우드 운영	JAR 배포 권장
Kubernetes 기반 배포	JAR 배포 권장
기존 기업용 WAS (WebLogic, WebSphere 등) 사용	WAR 배포 필요
레거시 시스템과 통합 필요	WAR 배포 고려
DevOps 기반 자동화 배포 원할 경우	JAR 배포 최적

6️⃣ 결론 🚀

✅ Spring Boot에서는 기본적으로 **JAR 배포가 권장**된다.
✅ JAR 배포는 Cloud Native/DevOps에 적합한 방식으로 최신 애플리케이션에 최적화되어 있다.
✅ WAR 배포는 기존 레거시 인프라(WAS)와의 호환성을 위한 옵션으로 유지되고 있으며, 일부 대기업 환경에서는 여전히 필요하다.
✅ 설계 초기 단계에서 **운영 환경/배포 전략**을 명확히 정의하고 적합한 방식을 선택해야 한다.

Docker 이미지 빌드 및 실행

Spring Boot 애플리케이션은 **Docker 컨테이너 기반**으로 배포하면 **환경 일관성 확보**, **자동화된 배포**, **확장성 확보** 등의 이점을 누릴 수 있다.
Docker 이미지를 빌드하는 방법은 크게 두 가지가 있다:

- Dockerfile 작성 → 명시적 이미지 구성
- Spring Boot Buildpack 사용 → 자동 이미지 빌드

이번 절에서는 두 가지 방법 모두 설명한다.

1️⃣ 사전 준비

- **Docker Desktop / Docker CLI 설치** → `docker --version` 확인
- Spring Boot 애플리케이션 JAR 빌드 가능 상태 확보 → `./gradlew build` 또는 `./mvnw package`

2️⃣ Dockerfile 작성법 📝

기본 Dockerfile 예시

```
# 1️⃣ 베이스 이미지 설정 (Java 17 예시)
FROM eclipse-temurin:17-jdk-alpine

# 2️⃣ 환경 변수 설정
ENV TZ=Asia/Seoul
ENV JAVA_OPTS=""

# 3️⃣ JAR 파일 복사
COPY build/libs/myapp.jar app.jar

# 4️⃣ 실행 명령
ENTRYPOINT ["sh", "-c", "java $JAVA_OPTS -jar /app.jar"]
```

구성 설명

단계	설명
FROM	경량화된 JDK 기반 이미지 사용
ENV	타임존, JVM 옵션 등 설정 가능

단계	설명
COPY	빌드된 JAR 파일 컨테이너 내부에 복사
ENTRYPOINT	애플리케이션 실행 명령

3️⃣ Docker 이미지 빌드

```
docker build -t myapp:1.0 .
```

- `-t myapp:1.0` → 이미지 이름과 태그 지정
- `.` → 현재 디렉토리 기준으로 Dockerfile 읽기

이미지 빌드 확인:

```
docker images
```

4️⃣ Docker 컨테이너 실행 🚀

```
docker run -d -p 8080:8080 --name myapp-container myapp:1.0
```

옵션	설명
-d	백그라운드(detached) 모드로 실행
-p	호스트 포트:컨테이너 포트 매핑
--name	컨테이너 이름 지정
myapp:1.0	사용할 이미지

컨테이너 상태 확인:

```
docker ps
```

로그 확인:

```
docker logs myapp-container
```

정지:

```
docker stop myapp-container
```

삭제:

```
docker rm myapp-container
```

5️⃣ Buildpack 이용 자동 이미지 빌드 ⏳

Spring Boot 2.3+ 부터는 **Buildpack 기반 Docker 이미지 빌드**를 지원한다 (Gradle 사용 시 예시).

```
./gradlew bootBuildImage --imageName=myapp:1.0
```

장점

✅ Dockerfile 없이 바로 이미지 생성 가능
✅ Layer 최적화 자동 적용
✅ Spring 공식 지원 → 안정성 ↑

생성된 이미지 확인:

```
docker images
```

실행 동일:

```
docker run -d -p 8080:8080 --name myapp-container myapp:1.0
```

build.gradle 설정 예시

```
bootJar {
    enabled = true
}

bootBuildImage {
    imageName = 'myapp:1.0'
}
```

👉 `bootBuildImage`는 **추가 설정 없이** Spring Boot + Buildpack 기반 최적화된 이미지를 자동으로 만들어준다 🚀.

6️⃣ 비교표 🗂️

방법	특징	추천 사용 상황
Dockerfile 직접 작성	자유로운 커스터마이징 가능	최적화된 커스텀 이미지 필요 시
Buildpack 사용 (`bootBuildImage`)	매우 간편, Spring 공식 지원	표준 Spring Boot 서비스 컨테이너화 시

7️⃣ 최적화 팁 💡

- **멀티 스테이지 빌드 활용** → 빌드 단계와 실행 단계 분리로 이미지 경량화 가능.
- **Layer cache 활용** → 의존성 변경 없는 경우 빠른 빌드 가능.
- **최소 base image 사용** → `eclipse-temurin` 또는 `distroless/java` 사용 권장.

멀티 스테이지 예시

```
# Build Stage
FROM gradle:7.5-jdk17 AS builder
COPY --chown=gradle:gradle . /home/gradle/project
WORKDIR /home/gradle/project
RUN gradle build --no-daemon

# Run Stage
FROM eclipse-temurin:17-jre-alpine
COPY --from=builder /home/gradle/project/build/libs/myapp.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

👉 Build 단계는 Gradle 이미지 사용, Run 단계는 경량 JRE 사용 → 이미지 용량 최소화 🚀.

결론 🗂️

✅ Spring Boot는 Docker 이미지 빌드/배포를 위한 다양한 방법을 제공한다.
✅ Buildpack 사용 시 빠르게 표준 이미지 생성 가능 → DevOps 환경에 적합.
✅ Dockerfile 직접 작성 시 고도화된 최적화 및 커스터마이징 가능.
✅ Container 기반 배포 환경(Kubernetes 등)에서는 반드시 `docker build → docker push → deploy` 흐름을 표준화하는 것이 좋다.

CI/CD 파이프라인 (GitHub Actions, Jenkins)

(GitHub Actions & Jenkins 기반)

CI/CD(Continuous Integration / Continuous Delivery or Deployment) 는 소프트웨어 개발에서 **자동화된 빌드, 테스트, 배포 파이프라인**을 구성하여 **개발 속도**와 **서비스 품질**을 동시에 향상시키는 필수적인 전략이다.

이번 절에서는 **GitHub Actions**와 **Jenkins** 기반으로 대표적인 Spring Boot 애플리케이션 **CI/CD 구성법**을 설명한다.

1️⃣ 전체 구성 흐름

```
개발자 Push / Pull Request
    ↓
CI 단계 (Build, Test, Lint, SonarQube 등)
    ↓
CD 단계 (Docker Build → Registry Push → Deploy)
    ↓
운영 환경 자동 반영 or 승인 기반 반영
```

2️⃣ GitHub Actions 구성 🚀

기본 개념

- GitHub 제공 **Workflow 기반 CI/CD 서비스**
- `.github/workflows/*.yml` 에 파이프라인 정의
- Github-hosted Runner 사용 가능 (Ubuntu, Windows, macOS)

기본 Workflow 예시

```yaml
# .github/workflows/ci-cd.yml
name: CI/CD Pipeline

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout Code
      uses: actions/checkout@v3

    - name: Set up JDK 17
      uses: actions/setup-java@v3
      with:
        distribution: 'temurin'
        java-version: '17'

    - name: Build with Gradle
      run: ./gradlew build

    - name: Docker Build
      run: docker build -t myapp:${{ github.sha }} .

    - name: Docker Push
      env:
        DOCKER_USER: ${{ secrets.DOCKER_USER }}
        DOCKER_PASSWORD: ${{ secrets.DOCKER_PASSWORD }}
      run: |
        echo $DOCKER_PASSWORD | docker login -u $DOCKER_USER --password-stdin
        docker tag myapp:${{ github.sha }} myrepo/myapp:latest
        docker push myrepo/myapp:latest
```

구성 설명

✅ **CI 단계**:

- 코드 Checkout → Build → Test → Docker Build
  ✅ **CD 단계**:
- Docker Registry(예: Docker Hub, AWS ECR 등) Push → 이후 배포 단계 구성 가능

GitHub Actions 장점

✅ GitHub 통합 → Pull Request 기반 자동 Trigger 용이
✅ YAML 기반 선언적 구성
✅ Serverless 형태 → 운영 인프라 불필요
✅ 무료 tier 제공 (용량 제한 있음)

GitHub Actions 한계

❌ 고정 IP 사용 어려움
❌ 복잡한 배포 인프라 구성 시 한계 존재 → 별도 배포 서버 필요

3️⃣ Jenkins 구성 🏛️

기본 개념

- **자체 서버에서 구동되는 CI/CD 자동화 서버**
- Job 기반으로 파이프라인 구성
- Groovy 기반 **Declarative Pipeline**, **Scripted Pipeline** 지원

Jenkinsfile 예시 (Declarative Pipeline)

```
pipeline {
    agent any

    environment {
        DOCKER_CREDENTIAL_ID = 'docker-hub-credentials'
    }

    stages {
        stage('Checkout') {
            steps {
                checkout scm
            }
        }

        stage('Build') {
            steps {
                sh './gradlew clean build'
            }
        }

        stage('Docker Build & Push') {
```

```
            steps {
                script {
                    docker.withRegistry('', DOCKER_CREDENTIAL_ID) {
                        def appImage = docker.build("myrepo/myapp:${env.BUILD_NUMBER}")
                        appImage.push('latest')
                    }
                }
            }
        }

        stage('Deploy') {
            steps {
                sh './scripts/deploy.sh'
            }
        }
    }

    post {
        success {
            echo 'Build, Docker Push & Deploy Success 🎉'
        }
        failure {
            echo 'Build Failed ❌'
        }
    }
}
```

구성 설명

✅ Checkout → Build → Docker Build & Push → Deploy 단계 구성
✅ Deploy 단계는 보통 **SSH** 또는 **Kubernetes CLI(kubectl)** 를 통해 배포 진행.

4️⃣ GitHub Actions vs Jenkins 비교

항목	GitHub Actions	Jenkins
설치 필요 여부	불필요 (GitHub 제공)	필요 (Jenkins 서버 운영 필요)
구성 방법	Workflow(YAML)	Jenkinsfile(Groovy)
확장성	GitHub Marketplace 제공	Plugin 기반으로 매우 유연
Trigger	GitHub 이벤트 기반	SCM Hook / Timer / Manual 등 유연
비용	GitHub 무료 tier 제공	자체 서버 운영 필요 → 인프라 비용 발생
사용 적합성	GitHub 기반 DevOps 구축 시 유리	복잡한 배포/인프라 관리 필요 시 유리

5️⃣ CI/CD 구성 Best Practice 🎁

- **Build 단계**:
  - Lint 검사 (Checkstyle, SpotBugs 등)
  - Unit Test
  - Integration Test
  - SonarQube Code Quality 검사 (Optional)
- **CD 단계**:
  - Docker Build
  - Image Registry Push (Docker Hub, AWS ECR 등)
  - Deployment (Kubernetes, VM, AWS ECS 등)
- **보안 고려**:
  - Docker Push 시 **Secret 관리 필수** (GitHub Secrets or Jenkins Credentials)
  - Build Pipeline 내 **Credential 노출 방지**
  - Deploy 단계는 Production 환경 보호 정책 적용 (Manual Approval 등)

결론 🗂️

✅ GitHub Actions는 **간편한 CI/CD 파이프라인 구축**에 매우 적합 → GitHub 기반 DevOps 흐름과 자연스러운 통합 제공.
✅ Jenkins는 **복잡한 Workflow, 외부 시스템과의 통합**이 필요한 경우 유리 → 대규모 엔터프라이즈 환경에서 여전히 널리 사용됨.
✅ 상황에 따라 **두 가지 방법을 병행**하는 것도 가능 → GitHub Actions로 Build/Test → Jenkins로 Stage별 Deployment 구성.

무중단 배포 전략

무중단 배포는 서비스가 운영 중인 상태에서 **사용자 경험에 영향을 주지 않고 애플리케이션을 배포(업데이트)하는 기법**이다.
고가용성(High Availability) 시스템에서는 필수적인 요구사항으로 간주된다.

잘못된 배포로 인한 **서비스 중단(502/503 오류)** 을 방지하고, 안정적인 배포 체계를 구축하는 것이 목표다.

1️⃣ 무중단 배포의 기본 원리

```
현재 트래픽 처리 중인 프로세스 유지
    ↓
새 버전 인스턴스 준비 (warm-up)
    ↓
트래픽을 새 인스턴스로 점진적 전환
    ↓
기존 버전 인스턴스 종료
```

👉 핵심 포인트 → **트래픽을 단계적으로 이동시키는 전략**이 필요 🚀.

2️⃣ 주요 무중단 배포 전략 유형 🗂️

전략	특징
Rolling Update	기존 인스턴스를 순차적으로 교체
Blue/Green Deployment	별도 새로운 인프라 준비 → 전환 시점 제어
Canary Deployment	소량 트래픽을 새 버전으로 먼저 전환 후 점진 확대
Shadow Deployment	새 버전에 트래픽 복제 후 테스트 (실제 사용자 영향 없음)

3️⃣ Rolling Update 전략 🔄

개념

- 기존 인스턴스를 한 번에 전부 내리는 것이 아니라 **1개씩 또는 일정 비율만 교체**.
- Kubernetes, AWS ECS, 많은 플랫폼에서 기본 제공.

동작 흐름

```
서버 그룹 N개 중 1개 업데이트 → 정상 확인 → 다음 1개 업데이트 ...
```

장점

✅ 인프라 추가 필요 없음
✅ 적용이 간단하고 빠름

단점

❌ 새 버전과 기존 버전 간 호환성 이슈 발생 시 대응 복잡
❌ DB 마이그레이션에는 주의 필요 (Backward Compatibility 중요)

4️⃣ Blue/Green Deployment 🔵🟢

개념

- **현재 프로덕션(Blue)와 새 버전(Green)을 동시에 준비**.
- 전환 시점에 **로드 밸런서 라우팅만 교체**.

동작 흐름

```
Blue 프로덕션 동작 중
    ↓
Green 새 버전 배포 및 준비
    ↓
테스트 후 Load Balancer → Green 전환
    ↓
Blue는 Rollback 용도로 잠시 유지
```

장점

✅ 배포 실패 시 빠른 롤백 가능
✅ DB 마이그레이션 시 별도 테스트 후 전환 가능
✅ A/B 테스트 용도로도 활용 가능

단점

❌ 인프라 이중 유지 필요 → 비용 증가
❌ Pipeline 구성 복잡도 증가

5️⃣ Canary Deployment 🐤

개념

- 새 버전을 **일부 사용자에게 먼저 적용**하여 문제 여부 확인 후 점진적 전체 전환.

동작 흐름

```
10% 트래픽 → New Version
    ↓
문제 없음 확인
    ↓
50% 트래픽 → New Version
    ↓
100% 전환
```

장점

✅ 리스크 최소화 → 사용자 체감 문제 발생 시 빠른 대응 가능
✅ 기능 단위 릴리즈 전략(A/B Test)과도 궁합 우수

단점

❌ Routing Layer에서 세밀한 트래픽 제어 필요 → 인프라 복잡성 ↑

6️⃣ Shadow Deployment 👥

개념

- 실시간 Production 트래픽을 **복제(Shadow)하여 새 버전으로 전달**, 실제 응답은 사용되지 않음.
- 새로운 시스템의 안정성/성능 테스트에 활용.

동작 흐름

```
실사용 트래픽 → 기존 시스템 정상 응답
    + 동일 트래픽 Shadow → New Version에서 비동기 처리 후 결과 모니터링
```

장점

✅ Production 환경에서 새 버전 테스트 가능
✅ 사용자 영향 없음

단점

❌ 구현 난이도 높음
❌ 트래픽 복제 구성 필요

7️⃣ 일반적인 플랫폼별 구성 방법 🚀

플랫폼	권장 전략
Kubernetes	Rolling Update 기본 제공, Blue/Green 가능 (Ingress/Service 활용), Canary 가능 (Argo Rollouts 등)
AWS ECS	Rolling Update 기본 제공, Blue/Green → CodeDeploy + ALB 사용 가능
AWS Elastic Beanstalk	Rolling Update, Blue/Green 지원
On-Premise (Tomcat + Nginx 등)	Blue/Green 구성 권장, Reverse Proxy에서 단계적 전환 구현
Service Mesh (Istio, Linkerd 등)	Canary, Blue/Green 고급 트래픽 제어 가능

8️⃣ 무중단 배포 시 고려해야 할 사항 ⚠️

DB 마이그레이션

- 반드시 **Backward Compatible** 설계 필요
- Rolling Update, Blue/Green 모두 적용 시 주의
- **비파괴적 변경 → 데이터 Backfill → 코드 배포 → Schema 변경** 단계 구성 권장.

세션 관리

- StateLess 서비스 구성 권장 (JWT, External Session Store 등 사용)
- Stateful 서비스는 세션 Migration 전략 필요.

트래픽 분산 제어

- Kubernetes Ingress, Istio VirtualService, Nginx 등으로 세밀한 라우팅 구성.

9️⃣ 결론 🗂️

✅ 무중단 배포는 **고가용성 서비스**에서 필수 전략이다.
✅ Rolling Update는 **가장 기본적이고 쉬운 방법** → Kubernetes 등에서 많이 사용.
✅ Blue/Green은 **강력한 Rollback 보장** → 안정성이 중요한 서비스에 적합.
✅ Canary는 **점진적 전환 전략**으로 리스크를 최소화할 수 있다.
✅ Shadow Deployment는 고급 전략으로, **새 시스템 안정성 검증**에 매우 유효하다.

프로세스 관리: systemd, supervisor, pm2

(systemd, supervisor, pm2)

서버에서 애플리케이션을 **백그라운드 서비스로 안정적으로 실행**하고,
자동 재시작, **모니터링**, **로그 관리** 등을 수행하려면 별도의 **프로세스 관리 도구**가 필요하다.

이번 절에서는 대표적인 3가지 프로세스 관리 도구:

- **systemd** (Linux 기본 서비스 관리자)
- **supervisor** (범용 프로세스 관리자)
- **pm2** (Node.js 기반 프로세스 관리자 → 범용 사용 가능)

를 비교하고 구성법을 설명한다.

1️⃣ systemd 🏛️

개요

- **Linux 표준 서비스 관리 프레임워크** → 대부분의 최신 Linux 배포판에서 기본 제공 (systemctl 사용).
- 서비스 단위로 관리 (`.service` 파일 구성).
- 부팅 시 자동 시작(Enable), 서비스 상태 관리(Active/Inactive) 가능.

기본 사용법

서비스 파일 작성

```
# /etc/systemd/system/myapp.service
[Unit]
Description=My Spring Boot App
After=network.target

[Service]
User=ubuntu
WorkingDirectory=/home/ubuntu/myapp
ExecStart=/usr/bin/java -jar myapp.jar
SuccessExitStatus=143
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
```

명령어

```
# 서비스 등록 후 리로드
sudo systemctl daemon-reload

# 서비스 시작
sudo systemctl start myapp.service

# 서비스 중지
sudo systemctl stop myapp.service

# 서비스 상태 확인
sudo systemctl status myapp.service

# 부팅 시 자동 시작 설정
sudo systemctl enable myapp.service
```

장점

✅ Linux 표준 → 설치 필요 없음
✅ 자동 시작, 재시작 지원
✅ 강력한 로그 관리 (`journalctl`) 지원

단점

❌ 설정 시 root 권한 필요
❌ 빠른 수정/재시작에는 불편 (config reload 필요)

2️⃣ Supervisor 🧰

개요

- **경량화된 범용 프로세스 관리자** → 여러 플랫폼에서 사용 가능.
- Python 기반 → 설치 필요.
- 간단한 구성 파일로 여러 프로세스를 관리 가능.

설치

```
sudo apt-get install supervisor
```

구성 예시

```
# /etc/supervisor/conf.d/myapp.conf
[program:myapp]
directory=/home/ubuntu/myapp
command=/usr/bin/java -jar myapp.jar
autostart=true
autorestart=true
stderr_logfile=/var/log/myapp.err.log
stdout_logfile=/var/log/myapp.out.log
user=ubuntu
```

명령어

```
# 설정 반영
sudo supervisorctl reread
sudo supervisorctl update

# 서비스 제어
sudo supervisorctl start myapp
sudo supervisorctl stop myapp
sudo supervisorctl status myapp
```

장점

✅ 설치와 설정이 매우 간단
✅ 여러 프로세스 동시 관리 가능
✅ 상세한 로그 설정 가능

단점

❌ systemd와 비교 시 시스템 표준은 아님
❌ 별도 설치 필요

3️⃣ pm2 🚰

개요

- Node.js 기반으로 개발된 프로세스 관리자 → Node.js 뿐 아니라 **Java, Python, 기타 실행 파일**도 관리 가능.
- 개발자 친화적인 CLI 제공 → 빠른 관리 가능.
- 상태 및 성능 모니터링 대시보드 제공.

설치

```
npm install pm2 -g
```

실행 예시

```
# 애플리케이션 실행
pm2 start myapp.jar --interpreter=java -- --jar

# 또는 (Spring Boot JAR 실행 시)
pm2 start "java -jar myapp.jar" --name myapp
```

서비스 등록 (Startup Script 생성 → systemd 연동 가능)

```
pm2 startup systemd
pm2 save
```

프로세스 제어

```
pm2 list
pm2 restart myapp
pm2 stop myapp
pm2 delete myapp
pm2 logs myapp
```

장점

✅ CLI가 매우 강력하고 직관적
✅ 실행 프로세스 상태 실시간 확인 가능 (`pm2 monit`)
✅ 플랫폼 독립적 → 개발, 테스트 서버에서도 동일 구성 가능
✅ Log rotation, Cluster mode 지원

단점

❌ Node.js runtime 필요
❌ production Linux 환경에서는 systemd 보다 낮은 우선순위로 사용됨 (다만 dev/stage에서는 유용)

4️⃣ 비교표 🗂️

항목	systemd	supervisor	pm2
설치 필요 여부	X (기본 제공)	O	O (npm install)
주요 대상	OS 레벨 서비스	범용 프로세스	범용 프로세스 (Node.js 친화적)
부팅 자동 시작	O	O	O (pm2 startup)
구성 난이도	보통	쉬움	매우 쉬움

항목	systemd	supervisor	pm2
CLI 관리 도구	systemctl	supervisorctl	pm2 CLI
Log 관리	journalctl	별도 파일 설정	내장 log 관리 + rotation 지원
사용 용도	운영 환경, 고정 서비스 관리	dev/stage/운영 서비스 다 가능	dev/stage 서비스에 적합, 운영에서는 systemd 보완용으로 사용 가능

5️⃣ 결론 🚀

✅ **systemd** → 리눅스 서버 운영 환경에서 **운영 서비스 등록 및 관리에 표준적으로 사용**.
✅ **supervisor** → 빠른 구성, 다중 프로세스 간편 관리 시 유용 (운영/개발 모두 활용 가능).
✅ **pm2** → 개발/테스트 환경에서 편리하며, Node.js 애플리케이션 외에도 범용으로 사용 가능.
→ **pm2 + systemd** 연계도 가능 (pm2 startup systemd).

운영 환경에서는 보통:

```
개발 단계 → pm2 사용
Stage 환경 → supervisor 또는 pm2
운영 환경 → systemd 중심 구성, 필요한 경우 supervisor 연계
```