2. 기본 개발 환경 설정

IntelliJ, VSCode 기반 개발 설정

1. 공통 준비 사항

1.1 JDK 설치

- 최소: Java 17 (Spring Boot 3.x 이상부터는 Java 17 이상 필수)
- 설치 확인: `java -version`

1.2 Spring Boot 프로젝트 생성

- Spring Initializr(https://start.spring.io)에서 Maven/Gradle + Java 프로젝트 생성
- 생성 시 다음 선택 가능
  - Project: Maven or Gradle
  - Language: Java
  - Spring Boot Version: 최신 안정 버전
  - Dependencies: Spring Web, Spring Data JPA, H2, Lombok 등

---

2. IntelliJ IDEA 설정

2.1 기본 환경 설정

- IntelliJ IDEA Ultimate (Spring Boot 지원 강화) 또는 Community Edition 사용 가능
- File > Project Structure (Ctrl+Alt+Shift+S)
  - Project SDK: Java 17 이상
  - Gradle 설정: Wrapper 사용 (`gradlew`) 권장

2.2 필수 플러그인

- Spring Boot (Ultimate에서는 자동 내장)
- Lombok (Settings > Plugins → lombok 검색 → 설치 후 재시작)
- .properties, .yml 지원은 기본 내장됨

2.3 실행 설정

- `@SpringBootApplication` 클래스 우클릭 → Run 실행
- 또는 Run > Edit Configurations > Add new Application 설정

2.4 기타 설정

- View > Tool Windows > Maven / Gradle 탭 활성화
- Terminal 탭에서 직접 `./gradlew bootRun` 실행 가능
- 코드 어시스트 및 Spring 의존성 인텔리센스 강화

---

3. Visual Studio Code 설정

3.1 필수 확장팩 설치

- **Extension 목록**
  - Java Extension Pack (Microsoft)
  - Spring Boot Extension Pack (Pivotal)
  - Lombok Annotations Support
  - Language Support for Java(TM) by Red Hat
  - Debugger for Java
  - Maven for Java / Gradle for Java

설치 후 VSCode 재시작

3.2 Java 환경 설정

- Java Home 설정 (settings.json 또는 환경 변수)

```
"java.home": "C:\\Program Files\\Java\\jdk-17"
```

3.3 프로젝트 실행

- `src/main/java/.../Application.java` 우클릭 → "Run Java"
- 또는 터미널에서 `./gradlew bootRun` 또는 `mvn spring-boot:run`

3.4 디버깅

- `.vscode/launch.json` 자동 생성됨
- 브레이크포인트 설정 후 F5 (Debug) 실행

3.5 Terminal 명령어 예시

```
./gradlew build
./gradlew bootRun
./gradlew test
```

---

4. 공통 설정 요소

4.1 application.yml or application.properties 생성

```yaml
server:
  port: 8081

spring:
  datasource:
    url: jdbc:h2:mem:testdb
    username: sa
    password:
    driver-class-name: org.h2.Driver
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
```

4.2 Lombok 설정

- IntelliJ: Settings > Build, Execution, Deployment > Compiler > Annotation Processors → Enable
- VSCode: Lombok 플러그인 설치만으로 자동 인식 (설정은 불필요)

---

5. Git 연동

IntelliJ

- VCS → Enable Version Control Integration → Git 선택
- Commit 창 내장
- `.gitignore` 자동 생성 (`.idea`, `target`, `.vscode`, `*.iml` 제외)

VSCode

- Source Control 패널에서 Git 관리
- `.gitignore` 수동 생성 또는 Spring Initializr에서 포함 가능

---

6. 실행 테스트 확인

```java
@RestController
public class HelloController {
    @GetMapping("/")
    public String hello() {
        return "Hello Spring Boot";
    }
}
```

브라우저에서 `http://localhost:8080`으로 접속하여 확인

---

7. 요약 비교

항목	IntelliJ IDEA	Visual Studio Code
추천 대상	Spring 전용 IDE에 익숙한 사용자	경량 IDE, 다양한 언어 병행 개발자
플러그인 지원	강력함 (Spring 지원 내장)	별도 설치 필요
런타임/디버깅	편리하고 직관적	설정 필요하지만 간단
학습 곡선	비교적 낮음	Java에 익숙하면 무난
커뮤니티 활용도	넓음 (Spring 공식 가이드 대부분 IntelliJ 기준)	비교적 적지만 성장 중

Maven vs Gradle 빌드 도구

1. 개요

항목	Maven	Gradle
최초 릴리스	2004년	2012년
설정 언어	XML (`pom.xml`)	Groovy / Kotlin DSL (`build.gradle`)
철학	표준화, 명시적 설정	유연성, 선언적 DSL
빌드 속도	느린 편 (전체 빌드)	빠름 (증분 빌드, 캐시 기반)
사용자 정의	제한적 (XML 기반)	매우 유연 (스크립트 기반 DSL)
멀티모듈 지원	가능하지만 복잡	매우 강력하고 간결
IDE 통합	IntelliJ, Eclipse 등과 우수	대부분 IDE에서 완전 지원
빌드 캐시	없음	있음 (로컬/원격 캐시)
커뮤니티	안정적이고 넓음	점차 확산 중, 최신 트렌드 반영
학습 곡선	쉬움	상대적으로 높음

---

2. 설정 방식 비교

Maven (XML 기반 설정)

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>demo</artifactId>
  <version>1.0.0</version>

  <dependencies>
    <dependency>
```

```
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>
</project>
```

Gradle (Groovy DSL 설정)

```
plugins {
    id 'org.springframework.boot' version '3.2.5'
    id 'io.spring.dependency-management' version '1.1.4'
    id 'java'
}

group = 'com.example'
version = '1.0.0'

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
}
```

---

3. 빌드 속도 비교

빌드 항목	Maven	Gradle
전체 빌드	느림	빠름 (병렬 및 증분 빌드 지원)
의존성 다운로드	네트워크 상태에 따라	캐시 활용 우수
재빌드 시	항상 전체 재실행	변경된 부분만 빌드
빌드 캐시	없음	있음 (로컬/원격 캐시, CI/CD에 유리)

---

4. 커스터마이징 가능성

- Maven은 플러그인 중심의 구조이며, XML 설정만으로는 복잡한 로직 구현이 어려움
- Gradle은 Groovy나 Kotlin DSL 기반으로, 조건문, 반복문, 사용자 함수 등을 자유롭게 작성 가능

```
// Gradle의 커스텀 태스크 예시
task hello {
    doLast {
        println 'Hello, Gradle!'
    }
}
```

---

5. 멀티모듈 프로젝트 비교

항목	Maven	Gradle
설정 복잡도	상대적으로 높음	DSL 덕분에 간단함
모듈 간 의존성 설정	명시적으로 `<modules>` 필요	`settings.gradle`에 include만 추가하면 됨
빌드 속도	전체 재빌드 발생	각 모듈의 변경점만 증분 빌드됨

---

6. 장단점 요약

Maven 장점

- XML 기반 → 명확하고 정형화
- 표준적인 생명주기
- 안정된 플러그인 생태계
- 레거시 시스템에서 폭넓게 사용 중

Maven 단점

- 유연성이 낮음
- 빌드 속도가 느림
- 조건 처리나 동적 설정이 불편함

Gradle 장점

- 유연한 DSL (Groovy, Kotlin)
- 빠른 빌드 (증분, 병렬 빌드, 캐시)
- 커스터마이징 자유도 높음
- 멀티모듈, CI/CD에 최적화

Gradle 단점

- 학습 곡선이 있음
- 설정이 복잡해질 수 있음
- 초기 설정에서 오류가 날 경우 디버깅이 어려움

---

7. Spring Boot와의 관계

Spring Boot는 Maven과 Gradle 모두 지원한다.
 그러나 **Gradle을 점점 더 우선시하는 추세**다.

- Spring Initializr에서 기본 옵션은 Gradle
- Gradle Wrapper를 이용하면 빌드 환경 이식성 향상
- Spring Boot 공식 문서도 Gradle 예제가 더 많아지는 중

8. 선택 기준 정리

상황	권장 빌드 도구
빠르게 시작하고 안정적인 프로젝트가 필요할 때	Maven
빌드 속도와 유연성, 자동화가 중요한 프로젝트일 때	Gradle
CI/CD 환경을 고려한 프로젝트	Gradle
멀티모듈 프로젝트	Gradle
보수적이고 유지보수 중심 조직	Maven
스크립트 기반 설정이 익숙한 경우	Gradle

결론

Maven은 정형화된 빌드를 요구하는 보수적인 프로젝트에 적합하고,
Gradle은 빠르고 유연하며 최신 트렌드에 맞춘 개발을 할 때 유리하다.
Spring Boot에서는 둘 다 공식 지원되지만, Gradle의 활용도가 점점 높아지고 있는 것이 사실이다.

application.properties vs application.yml

1. 공통점

- Spring Boot는 자동으로 `application.properties` 또는 `application.yml`을 로딩한다.
- 둘 중 하나만 사용하거나, 둘 다 혼합해서 사용할 수도 있다. (단, `properties`가 우선 적용됨)
- 설정 내용은 `@Value`, `@ConfigurationProperties`, `Environment` 등으로 접근할 수 있다.

2. 차이점 비교

항목	application.properties	application.yml
표현 형식	평면적 key-value 쌍	계층적 YAML 구조
구조 표현	점(.)으로 구분	들여쓰기로 계층 표현
가독성	단순하지만 중복 많음	계층 표현이 깔끔함
멀티라인 문자열	`\n`, `\t`로 표현	`
배열 표현	`key[0]`, `key[1]`	`-`로 자연스럽게 표현
오류 발생 시	비교적 안전함	들여쓰기 오류가 흔함
Spring 공식 기본값	`application.properties`	`application.yml`도 동등하게 지원됨

3. 예시 비교

예: 데이터베이스 설정

application.properties

```
properties코드 복사spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=1234
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

application.yml

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: root
    password: 1234
    driver-class-name: com.mysql.cj.jdbc.Driver
```

---

4. 배열(List) 표현

application.properties

```
my.values[0]=apple
my.values[1]=banana
my.values[2]=cherry
```

application.yml

```
my:
  values:
    - apple
    - banana
    - cherry
```

---

5. 멀티 프로파일 구성

Spring Boot에서는 환경별 설정을 파일명으로 나눌 수 있다.

파일 구조 예

```
application.yml
application-dev.yml
application-prod.yml
```

application.yml 내용

```yaml
spring:
  profiles:
    active: dev
```

application-dev.yml 내용

```yaml
server:
  port: 8081
```

→ 실행 시 `dev` 프로파일이 활성화되어, 해당 설정이 적용된다.

---

6. 다중 프로파일 표현

application.properties

```properties
spring.profiles.active=dev
```

application.yml

```yaml
spring:
  profiles:
    active: dev
```

또는 하나의 파일 안에서 여러 프로파일을 다음과 같이 구분할 수도 있다.

```yaml
---
spring:
  profiles: dev
server:
  port: 8081
---
spring:
  profiles: prod
server:
  port: 80
```

---

7. 결론 및 추천

기준	추천 형식
간단하고 빠른 설정	`application.properties`
계층형 설정, 구조적 표현	`application.yml`
배열, 복잡한 구조, 블록 표현 필요	`application.yml`

기준	추천 형식
YAML 포맷에 익숙하지 않은 경우	`application.properties`
팀 단위, 협업 환경에서 설정이 많은 경우	`application.yml` 가독성이 높음

결론

- 기능적으로는 동일하지만, **구조화된 설정이 필요한 경우** `application.yml` **이 더 유리**하다.
- **단일 속성만 빠르게 설정하고 싶을 경우** `application.properties` **가 더 간편**하다.
- 실무에서는 대부분 YAML이 표준으로 자리잡고 있으나, 취향이나 프로젝트 복잡도에 따라 선택하면 된다.

환경 프로파일(profile) 설정

1. 프로파일이란?

프로파일은 **Spring 컨텍스트 내의 조건부 설정 블록**이다.
Spring은 애플리케이션 실행 시 **특정 프로파일을 활성화**하고, 그에 해당하는 Bean이나 설정 파일만 적용한다.

---

2. 프로파일 설정 방식

방법 1: `application.yml` 또는 `application.properties`에서 설정

```yaml
spring:
  profiles:
    active: dev
```

또는 `.properties` 방식:

```properties
spring.profiles.active=dev
```

방법 2: 실행 시 인자로 지정

```
java -jar myapp.jar --spring.profiles.active=prod
```

또는 Gradle:

```
./gradlew bootRun --args='--spring.profiles.active=prod'
```

---

3. 프로파일별 설정 파일 분리

프로파일별 설정은 별도의 파일로 구성할 수 있다.

```
src/main/resources/
├── application.yml
├── application-dev.yml
├── application-test.yml
├── application-prod.yml
```

application.yml (공통 설정 또는 프로파일 활성화 지정)

```yaml
spring:
  profiles:
    active: dev
```

application-dev.yml

```yaml
server:
  port: 8081

spring:
  datasource:
    url: jdbc:h2:mem:devdb
```

application-prod.yml

```yaml
server:
  port: 80

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

---

4. 다중 프로파일 구성 (하나의 yml 안에서 분리)

```yaml
---
spring:
  profiles: dev
server:
  port: 8081
---
spring:
  profiles: prod
server:
  port: 80
```

---

5. 프로파일별 Bean 등록

```
@Configuration
public class AppConfig {

    @Bean
    @Profile("dev")
    public DataSource devDataSource() {
        // 개발용 데이터소스
        return new HikariDataSource();
    }

    @Bean
    @Profile("prod")
    public DataSource prodDataSource() {
        // 운영용 데이터소스
        return new HikariDataSource();
    }
}
```

---

6. 다중 프로파일 활성화

여러 개의 프로파일을 동시에 활성화할 수도 있다.

```
spring.profiles.active=dev,local
```

또는

```
spring:
  profiles:
    active:
      - dev
      - local
```

---

7. 프로파일 확인 방법

실행 로그에 다음과 같은 내용이 출력됨:

```
The following 1 profile is active: "dev"
```

또는 코드 내에서 환경 정보 확인:

```java
@Autowired
Environment environment;

public void printProfile() {
    System.out.println(Arrays.toString(environment.getActiveProfiles()));
}
```

---

8. 자주 사용하는 프로파일 이름 예시

환경	프로파일 이름 추천
개발	`dev`
테스트	`test`
로컬 개발	`local`
스테이징	`stage`, `qa`
운영	`prod`

---

9. 자주 하는 실수

- `application.yml`과 `application.properties`를 동시에 사용하는 경우 우선순위에 주의해야 한다.
- `application.yml` 안에서 잘못된 들여쓰기나 `---` 섹션 분리가 누락되면 프로파일 적용이 되지 않을 수 있다.
- 프로파일 이름 철자 실수로 원하는 설정이 적용되지 않을 수 있다.

결론

Spring Boot의 프로파일 시스템은 **환경별로 유연하게 설정을 분리하고 관리**할 수 있도록 한다.
특히 운영 환경, 테스트 환경, 개발 환경을 구분하여 설정할 때 매우 유용하다.

로그 설정 (Logback, Log4j2)

1. Spring Boot의 기본 로깅 구조

요소	설명
기본 로깅 구현체	Logback
로깅 API	SLF4J (Simple Logging Facade for Java)
설정 파일	`logback-spring.xml`, `logback.xml`, `application.properties`
출력 포맷	콘솔 + 로그 파일 설정 가능
외부 로깅 도구	Log4j2, Log4j1, JUL 등으로 교체 가능

---

2. 간단한 로깅 예시

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@RestController
public class SampleController {
    private static final Logger logger =
LoggerFactory.getLogger(SampleController.class);

    @GetMapping("/")
    public String hello() {
        logger.info("Hello 요청 발생");
        return "Hello";
    }
}
```

---

3. Logback 설정 방법

3.1 application.properties 방식 (기본)

```
# 로그 레벨 설정
logging.level.root=INFO
logging.level.org.springframework.web=DEBUG
logging.level.com.example=TRACE

# 로그 파일 지정
logging.file.name=logs/app.log
logging.file.path=logs

# 콘솔 및 파일 출력 형식
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %msg%n
logging.pattern.file=%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger - %msg%n
```

3.2 logback-spring.xml 방식 (권장, 고급 설정 가능)

```
<configuration>
    <property name="LOG_PATH" value="logs" />

    <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${LOG_PATH}/app.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${LOG_PATH}/app.%d{yyyy-MM-dd}.log</fileNamePattern>
        </rollingPolicy>
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
```

```
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="FILE"/>
    </root>
</configuration>
```

※ `logback.xml` 대신 반드시 `logback-spring.xml`을 사용할 것. 그래야 Spring Boot의 profile 처리 (`<springProfile>`)가 가능하다.

---

4. Log4j2로 교체하기

4.1 기존 의존성 제거

Maven:

```
<exclusions>
    <exclusion>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
</exclusions>
```

Gradle:

```
configurations {
    all {
        exclude group: 'org.springframework.boot', module: 'spring-boot-starter-logging'
    }
}
```

4.2 Log4j2 의존성 추가

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

4.3 설정 파일 생성

파일명은 `log4j2-spring.xml` (또는 `log4j2.xml`)

```xml
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss} %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="info">
      <AppenderRef ref="Console"/>
    </Root>
    <Logger name="com.example" level="debug"/>
  </Loggers>
</Configuration>
```

---

5. 로그 레벨 종류

레벨	설명
TRACE	가장 상세한 로그, 주로 개발 중 디버깅용
DEBUG	디버깅 정보
INFO	운영에 필요한 일반 정보
WARN	경고 상황 (정상 동작에 영향은 없으나 위험 가능)
ERROR	예외나 실패 상황
OFF	로그 출력 안 함

---

6. 설정 우선순위

1. `logback-spring.xml` 또는 `log4j2-spring.xml`
2. `logback.xml` 또는 `log4j2.xml`
3. `application.properties` 또는 `application.yml`
4. Spring Boot의 기본 내장 설정

---

7. 로그 출력 위치 지정

- 콘솔
- 파일 (`logging.file.name`, `logging.file.path`)
- 회전 로그 (`rolling policy`)
- 외부 시스템 (예: Logstash, Kafka 등)

---

8. 로그 설정에서 자주 발생하는 실수

- `logback.xml`을 사용할 경우 Spring Profile 적용이 안 됨 (`logback-spring.xml` 사용해야 함)
- XML 태그 누락 또는 오타
- 로그 파일 권한 문제
- 의존성 충돌: Logback과 Log4j2를 동시에 포함

결론

Spring Boot에서는 Logback을 기본으로 제공하지만, Log4j2로도 쉽게 전환 가능하다.
간단한 설정은 `application.properties`로 충분하지만, 복잡한 로그 정책이 필요한 경우 XML 기반 설정(`logback-spring.xml`, `log4j2-spring.xml`)을 사용하는 것이 좋다.

DevTools와 자동 재시작

1. DevTools란 무엇인가?

Spring Boot DevTools는 **개발 생산성을 높이기 위한 기능성 유틸리티**이다.
다음과 같은 기능을 제공한다.

기능	설명
자동 재시작	코드 변경 시 애플리케이션을 자동으로 다시 실행
LiveReload	정적 자원(html, css 등) 변경 시 브라우저 자동 새로고침
빠른 classpath 재시작	전체 JVM을 재시작하지 않고 클래스 로딩만 재시도
캐시 비활성화	Thymeleaf, JSP 등 템플릿 엔진의 캐시 비활성화
H2 Console 자동 연결	브라우저에서 H2 DB 접근 지원
Remote DevTools	원격 애플리케이션 자동 재시작 지원 (보안 주의)

---

2. DevTools 설정 방법

2.1 의존성 추가

Gradle

```
dependencies {
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
}
```

Maven

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <scope>runtime</scope>
</dependency>
```

> `developmentOnly` 또는 `scope=runtime`으로 설정하면 운영 환경에 포함되지 않음.

---

3. 자동 재시작(Auto Restart)

DevTools는 코드 변경이 감지되면 애플리케이션을 자동으로 다시 시작한다.
이는 전체 JVM을 재시작하는 것이 아니라, **classloader를 새로 로드**함으로써 빠른 재시작이 가능하다.

3.1 재시작 트리거 대상

- `.class` 파일이 변경된 경우 (예: `Ctrl+S` 또는 빌드 시)
- `target/classes`, `build/classes` 등 classpath 디렉토리 내부의 변경

3.2 재시작 제외 대상

`/static`, `/public`, `/resources`, `/META-INF` 등 정적 자원은 전체 재시작 없이 변경 가능
(이 경우 LiveReload만 트리거됨)

---

4. LiveReload 기능

LiveReload는 정적 자원이 변경되었을 때 **브라우저를 자동으로 새로고침하는 기능**이다.

4.1 작동 방식

- DevTools가 LiveReload 서버를 구동 (`35729` 포트)
- 브라우저에 LiveReload 플러그인이 설치되어 있어야 함
- 또는 IDE에서 자체 연동됨 (IntelliJ의 경우 자동 동작 가능)

4.2 비활성화 방법

```
spring.devtools.livereload.enabled=false
```

---

5. 캐시 비활성화

템플릿 엔진(Thymeleaf 등)의 캐시가 자동으로 비활성화되어 변경 사항이 즉시 반영된다.

예: Thymeleaf 설정이 자동으로 다음과 같이 변경된다.

```
spring.thymeleaf.cache=false
```

---

6. 설정 파일 변경 감지

`application.properties`, `application.yml` 파일 변경 시에도 자동으로 재시작됨
단, **환경 변수와 VM 옵션은 재시작으로 반영되지 않음** → 이 경우 수동 재시작 필요

---

7. 자동 재시작 비활성화 방법

```
spring.devtools.restart.enabled=false
```

또는 JVM 옵션:

```
-Dspring.devtools.restart.enabled=false
```

---

8. 디렉토리 제외 설정

특정 디렉토리(예: 로그, 이미지 등)는 감시 대상에서 제외 가능

```
spring.devtools.restart.exclude=static/**,public/**
```

---

9. 주의사항

항목	설명
운영 환경에서는 자동으로 DevTools가 비활성화됨	`spring-boot-devtools`는 `classpath`에 있어도 production에서는 작동 안 함
remote-devtools 기능은 보안상 위험	HTTPS 미지원, 기본 포트 노출 위험
전체 애플리케이션 상태가 유지되지 않음	재시작 시 기존 상태가 초기화됨 (세션 등 포함)

---

10. 실전 개발에서 DevTools 활용 패턴

사용 목적	사용 방식
정적 페이지 빠른 확인	HTML/CSS 저장 시 LiveReload
컨트롤러 로직 디버깅	저장 후 자동 재시작을 통해 즉시 확인
프로퍼티 설정 테스트	`application.yml` 저장 → 자동 반영
브라우저 자동 새로고침	LiveReload 플러그인 또는 Intellij 연동 사용

결론

Spring Boot DevTools는 개발 중에만 작동하며,
자동 재시작과 캐시 비활성화를 통해 개발 생산성을 크게 향상시켜 준다.
운영 환경과 개발 환경의 **명확한 구분**이 필요한 만큼, `scope`, `profile`, `활성화 여부` 등을 신중히 설정해야 한다.