13. Spring Boot 심화

Spring Boot 자동 설정 원리

🧩 1. 정의

> **Spring Boot는 애플리케이션에서 사용 중인 라이브러리를 분석해서,**
> 개발자가 직접 설정하지 않아도 필요한 Bean, 설정, 컴포넌트를 자동으로 등록해줌.

즉,

- `application.properties`나 `@EnableXXX` 없이도
- **라이브러리 의존성만 추가하면 동작하는 이유**가 바로 자동 설정 덕분이다.

🔍 2. 핵심 동작 흐름

Spring Boot는 다음 순서로 자동 설정을 처리한다:

```
1. @SpringBootApplication → @EnableAutoConfiguration 포함
2. SpringFactoriesLoader → META-INF/spring.factories 읽기
3. AutoConfiguration 클래스 로딩 (조건부 @Configuration)
4. @Conditional로 실제 환경을 검사한 뒤 Bean 등록 여부 결정
```

🎯 3. 핵심 구성 요소

요소	설명
`@EnableAutoConfiguration`	자동 설정 활성화
`spring.factories`	어떤 자동 설정 클래스들을 불러올지 정의
`@AutoConfiguration` / `@Configuration`	실제 자동 설정 클래스
`@ConditionalXXX`	클래스/빈/프로퍼티의 존재 여부에 따라 조건부 설정

✅ 3-1. 예: `@SpringBootApplication` 내부 구조

```
@SpringBootApplication
// ↓ 이것이 자동 설정의 시작점
@EnableAutoConfiguration
```

✅ 3-2. `spring.factories` 예

```
# spring-boot-autoconfigure-3.0.x.jar
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
...
```

➡️ 이 설정이 자동으로 읽혀서 관련 설정들이 활성화됨

🔧 4. 조건 기반 설정: `@ConditionalXXX`

자동 설정은 무조건 등록하지 않고, 다음 조건이 맞는 경우에만 설정을 적용해.

조건 어노테이션	설명
`@ConditionalOnClass`	클래스가 classpath에 있으면 적용
`@ConditionalOnMissingBean`	특정 Bean이 등록되어 있지 않으면 적용
`@ConditionalOnProperty`	특정 프로퍼티가 설정되어 있으면 적용
`@ConditionalOnBean`	특정 Bean이 이미 있을 때만 적용
`@ConditionalOnWebApplication`	웹 애플리케이션 환경일 경우만 적용

📘 예제: JPA 자동 설정 일부

```
@Configuration
@ConditionalOnClass({EntityManager.class})
@ConditionalOnMissingBean(LocalContainerEntityManagerFactoryBean.class)
@EnableConfigurationProperties(JpaProperties.class)
public class HibernateJpaAutoConfiguration {
    // DataSource, EntityManager 설정 자동 구성
}
```

🧱 5. 실전 예제

✅ 5-1. `spring-boot-starter-data-jpa` 의존성 추가 시

자동으로 다음이 설정됨:

- `DataSourceAutoConfiguration`
- `JpaRepositoriesAutoConfiguration`
- `HibernateJpaAutoConfiguration`

이 설정 덕분에 `@EnableJpaRepositories`, `EntityManagerFactory`, `DataSource` 없이도 작동함.

✅ 5-2. 커스텀 AutoConfiguration 만들기

```
@Configuration
@ConditionalOnProperty(name = "feature.enabled", havingValue = "true")
public class CustomFeatureAutoConfiguration {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

그리고:

```
# resources/META-INF/spring.factories
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.example.autoconfig.CustomFeatureAutoConfiguration
```

🧠 6. 실무 설계 팁

팁	설명
기본 설정은 자동 구성 사용	복잡하지 않다면 수동 설정 없이 시작해도 충분
불필요한 설정 끄기	`spring.autoconfigure.exclude` 사용
직접 Bean 등록 시 자동 설정 무시됨	`@ConditionalOnMissingBean` 조건 때문
`@ConditionalOnProperty`로 제어	특정 기능의 on/off를 쉽게 구현 가능
설정 구조 파악은 Actuator 확인	`/actuator/beans`, `/actuator/conditions`로 현재 설정 확인 가능

✅ 마무리 요약

항목	설명
자동 설정의 시작	`@EnableAutoConfiguration`
설정 로딩 방식	`spring.factories`의 자동 설정 클래스 목록
핵심 제어 조건	`@ConditionalOnClass`, `@ConditionalOnMissingBean`, `@ConditionalOnProperty`
장점	설정 생략, 개발 생산성 극대화
단점	내부 동작을 모르면 디버깅 어려움, 튜닝 어려움

`@SpringBootApplication`

🧩 1. 정의

> Spring Boot 애플리케이션의 **시작 클래스**에 붙이는 **핵심 메타 어노테이션**

즉, 이 어노테이션 하나만 붙이면

- 자동 설정
- 컴포넌트 스캔
- 설정 클래스 등록
  까지 **모두 한 번에 처리됨**

🔍 2. 내부 구성 (3개의 어노테이션 조합)

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@SpringBootConfiguration                 // @Configuration 포함
@EnableAutoConfiguration                // 자동 설정 기능 활성화
@ComponentScan                           // 패키지 하위 Bean 자동 등록
public @interface SpringBootApplication {
}
```

내부 어노테이션	역할
`@SpringBootConfiguration`	설정 클래스임을 명시 (`@Configuration` 포함)
`@EnableAutoConfiguration`	`spring.factories` 기반 자동 설정
`@ComponentScan`	현재 클래스의 패키지 하위에서 Bean을 자동 탐색

📦 3. 사용 예시

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

🎯 4. 주요 역할 정리

기능	설명
설정 클래스 지정	`@Configuration` 대체

기능	설명
컴포넌트 스캔	현재 위치를 기준으로 Bean 자동 등록
자동 설정	의존성 기반 자동 Bean 구성 (`DataSource`, `WebMvc`, `JPA` 등)
애플리케이션 부트스트랩	`SpringApplication.run()`과 연결되어 전체 앱 실행 시작

🧠 5. 컴포넌트 스캔 기준

- `@ComponentScan`은 **현재 클래스가 위치한 패키지를 기준으로 하위 패키지를 스캔**
- 하위가 아닌 **상위 패키지에 Bean이 있으면 등록되지 않음**
  ➡️ **항상 최상위 패키지에 놓는 것이 권장**

예:

```
com.example
├── MyApplication.java   ← 여기 있어야 함
└── user
    └── UserService.java (@Service)
```

⚙️ 6. 확장 방법 (커스터마이징)

✅ ① 컴포넌트 스캔 범위 변경

```
@SpringBootApplication(scanBasePackages = "com.example.other")
```

✅ ② 자동 설정 제외

```
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
```

⚠️ 7. 주의사항

항목	설명
다른 설정 클래스에 중복 선언 ❌	`@SpringBootApplication`은 한 곳에만
Bean 중복 등록 시 예외	자동 설정이 예상치 못한 Bean을 등록할 수 있음
컴포넌트 스캔 범위 반드시 확인	클래스 위치에 따라 Bean 미등록 이슈 발생 가능
`main()` 함수 필수	Spring Boot는 `main()`에서 실행됨 (`SpringApplication.run`)

✅ 마무리 요약

항목	설명
구성	`@Configuration` + `@ComponentScan` + `@EnableAutoConfiguration`
위치	프로젝트 **최상단 패키지**에 위치해야 함
핵심 기능	자동 설정 + Bean 등록 + 앱 부트스트랩
실무 팁	자동 설정을 제외하거나 확장할 땐 `exclude`, `scanBasePackages` 활용
상호작용	`application.properties`, 의존성, Bean 구성에 따라 동작 결정됨

`@EnableAutoConfiguration`, `@Conditional*` 계열 어노테이션

🧩 1. `@EnableAutoConfiguration`이란?

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration
```

✔️ 핵심 기능 요약

- **Spring Boot 자동 설정의 시작점**
- `spring.factories` 또는 `AutoConfiguration.imports`에 명시된 **자동 설정 클래스들을 읽어와서 등록함**
- 내부적으로 `@Import(AutoConfigurationImportSelector.class)`를 통해 자동 설정 후보들을 스캔

🔍 2. 어떻게 자동 설정되는가?

```
@SpringBootApplication
   ↳ @EnableAutoConfiguration
       ↳ @Import(AutoConfigurationImportSelector.class)
           ↳ 자동 설정 클래스 목록 로딩 (spring.factories 또는 AutoConfiguration.imports)
               ↳ @Conditional* 검사
                   ↳ 조건 만족 시 @Configuration → Bean 등록
```

⚙️ 3. `@Conditional*` 계열 어노테이션: 개념과 역할

> 자동 설정 클래스는 **무조건 적용되지 않고**,
> `@Conditional*` 계열 어노테이션의 조건에 **모두 만족해야만** 등록된다.

이 어노테이션들은 **"조건부 Bean 등록"**, **"상황에 따라 적용 제어"**라는 핵심 설계를 가능하게 함.

🧩 4. 주요 `@Conditional*` 어노테이션 종류

어노테이션	동작 조건	주 용도
`@ConditionalOnClass`	클래스가 classpath에 있으면 적용	의존성 감지
`@ConditionalOnMissingClass`	클래스가 없으면 적용	경량 설정
`@ConditionalOnBean`	특정 타입의 Bean이 이미 있을 때만	Bean 조립 순서 제어
`@ConditionalOnMissingBean`	특정 Bean이 없을 때만	기본 Bean 제공
`@ConditionalOnProperty`	특정 프로퍼티가 설정됐을 때만	설정 값 기반 ON/OFF
`@ConditionalOnExpression`	SpEL 조건이 참일 때	동적 제어
`@ConditionalOnResource`	특정 리소스 존재 시	config 파일 등 감지
`@ConditionalOnWebApplication`	웹 애플리케이션일 경우만	웹 MVC 설정
`@ConditionalOnNotWebApplication`	웹 환경이 아닐 경우만	CLI, batch
`@ConditionalOnJava`	특정 Java 버전 이상일 때	JVM 환경 제어

📘 5. 실전 예제 (Spring Boot 내부 구조 일부)

```
@Configuration
@ConditionalOnClass(DataSource.class)
@ConditionalOnProperty(name = "spring.datasource.url")
@ConditionalOnMissingBean(DataSource.class)
public class DataSourceAutoConfiguration {

    @Bean
    public DataSource dataSource() {
        return new HikariDataSource(); // 기본 DataSource 설정
    }
}
```

조건	설명
`DataSource.class`가 classpath에 있고	JDBC 관련 의존성이 추가되어 있을 때만
`spring.datasource.url` 프로퍼티가 존재하고	application.yml에 DB URL이 있을 때

조건	설명
`DataSource` 타입의 Bean이 미리 등록되어 있지 않다면	직접 정의하지 않은 경우

→ 자동으로 HikariCP가 등록됨.

🧠 6. 실무에서 쓰는 패턴

✅ ① 설정 키 기반으로 기능 on/off 제어

```
@Configuration
@ConditionalOnProperty(
  name = "feature.mail.enabled",
  havingValue = "true",
  matchIfMissing = false
)
public class MailServiceAutoConfiguration {
    @Bean
    public MailService mailService() {
        return new MailService();
    }
}
```

```
feature:
  mail:
    enabled: true
```

✅ ② 커스터마이징을 허용하는 방식

```
@Bean
@ConditionalOnMissingBean
public JwtEncoder jwtEncoder() {
    return new DefaultJwtEncoder();
}
```

➡️ 개발자가 `JwtEncoder`를 수동으로 등록하면 **자동 설정은 비활성화됨**

✅ 마무리 요약

항목	설명
`@EnableAutoConfiguration`	자동 설정의 진입점. 설정 클래스를 가져오게 함
`@Conditional*`	자동 설정 클래스가 **실제로 적용될지 말지를 조건별로 결정**
실무 사용법	Bean 중복 방지, 조건부 기능 활성화, 유연한 설정

항목	설명
추천 패턴	`@ConditionalOnClass` + `@ConditionalOnProperty` + `@ConditionalOnMissingBean` 3종 조합
설계 이점	개발자는 설정 안 해도 대부분 기본 환경 구성 완료됨. 필요 시 확장 가능

외부 설정 바인딩

- `@ConfigurationProperties`, `@Value`

🧩 1. 개요

어노테이션	역할
`@Value`	단일 값(문자열, 숫자 등)을 **직접 주입**
`@ConfigurationProperties`	외부 설정을 **객체 단위로 매핑**해서 주입 (묶음 주입)

✅ 2. `@Value` — 단일 값 주입

📘 예제

```
@Component
public class MyComponent {

    @Value("${myapp.title}")
    private String title;

    @Value("${server.port:8080}")
    private int port; // 기본값 8080
}
```

✅ 특징

- 빠르고 간단함
- 단일 필드에만 사용
- `SpEL` 지원 (`#{}` 표현 가능)
- 기본값 지정 가능 (`:defaultValue`)

✅ 3. `@ConfigurationProperties` — 묶음 객체 주입

📘 예제

```yaml
myapp:
  title: "My App"
  version: "1.0"
  security:
    enabled: true
```

```java
@Configuration
@ConfigurationProperties(prefix = "myapp")
public class MyAppProperties {
    private String title;
    private String version;
    private Security security;

    public static class Security {
        private boolean enabled;
        // getter/setter
    }

    // getter/setter
}
```

- 사용 시 빈으로 등록 필요 → `@Component` 또는 `@Configuration + @EnableConfigurationProperties` 조합

✅ 특징

항목	설명
객체 단위 바인딩	여러 값을 한 번에 주입 가능 (계층 구조도 가능)
타입 안정성	Integer, Boolean 등 타입 자동 변환 및 검증 가능
YAML/Properties 지원	prefix 지정 방식으로 계층 구조 쉽게 매핑
IDE 자동완성 지원	Spring Boot 환경에서 `.yml` 편집기 지원됨

⚠️ 4. `@Value` vs `@ConfigurationProperties` 비교표

항목	`@Value`	`@ConfigurationProperties`
주입 방식	단일 값	객체 단위 (묶음 바인딩)
바인딩 대상	필드에 직접	클래스 전체
타입 변환	단순 문자열 → 기본형	복합 객체까지 자동 변환

항목	`@Value`	`@ConfigurationProperties`
계층형 구조 지원	❌	✅
기본값 지정	✅ (`:default`)	❌ (기본은 오류 발생)
SpEL 사용	✅ (`#{}`)	❌
검증 지원	❌	✅ (`@Validated`, `@NotNull`)
설정 자동완성	❌	✅ (Spring Boot Config Metadata 지원)
실무 적합성	소량/즉시/간단 설정	✅ 대규모 설정, 유지보수 중심 설계에 강함

💡 5. 실무 권장 방식

상황	권장 방식
단일 값 빠르게 읽기	`@Value`
다수의 관련 설정을 그룹으로 주입	✅ `@ConfigurationProperties` (주력 방식)
설정값 검증이 필요한 경우	`@ConfigurationProperties` + `@Validated`
외부 환경을 강력히 추상화할 때	`@ConfigurationProperties` + 클래스 기반 접근

🔐 6. 설정 클래스 등록 방식

✅ 자동 등록

```
@Component
@ConfigurationProperties(prefix = "myapp")
```

✅ 수동 등록 (더 명시적)

```
@ConfigurationProperties(prefix = "myapp")
public class MyAppProperties {
    ...
}

@Configuration
@EnableConfigurationProperties(MyAppProperties.class)
public class MyConfig { }
```

🎯 7. 유효성 검증 예제

```
@ConfigurationProperties("myapp")
@Validated
public class MyAppProperties {

    @NotBlank
    private String title;

    @Min(1)
    private int threadCount;

    // getter/setter
}
```

Spring Boot 실행 시 설정이 틀리면 **애플리케이션 부팅 실패**로 방지 가능

✅ 마무리 요약

항목	설명
`@Value`	빠르고 간단, 단일 필드에 사용
`@ConfigurationProperties`	묶음 매핑, 계층 구조, 유효성 검증, 실무 최적화
추천	대부분의 실무 설정은 `@ConfigurationProperties`로 구성하는 것이 유지보수에 유리
스프링 부트 철학과 궁합	`application.yml` + 객체 바인딩 기반이 스프링 부트의 기본 설계 방향

다중 설정 파일 관리

🧩 1. 목적

- **운영 환경(dev/prod/test/staging)**에 따라 서로 다른 설정 값을 사용하고 싶을 때
- 예: `개발용 DB`, `운영용 DB`, `로컬용 설정`, `로깅 레벨`, `외부 API 키`, `서버 포트` 등

➡️ Spring Boot는 **다중** `application-{profile}.yml` **또는** `.properties` 파일을 통해
자동으로 환경을 분리하고 로딩할 수 있어.

📦 2. 설정 파일의 계층 구조

```
src/
└── main/
    └── resources/
        ├── application.yml              ← 공통 (기본 설정)
        ├── application-dev.yml          ← 개발 환경용
        ├── application-prod.yml         ← 운영 환경용
        ├── application-test.yml         ← 테스트 환경용
```

⚙️ 3. 활성화 방법 (`spring.profiles.active`)

✅ 방법 1: `application.yml` 내부에서 활성화

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

> 📌 `application.yml`이 먼저 로딩되고,
> `active: dev`에 따라 → `application-dev.yml`이 **추가로 병합됨**

✅ 방법 2: 실행 파라미터로 지정 (실무 추천)

```
# IntelliJ 또는 shell 실행 시
--spring.profiles.active=prod
```

또는 `VM options`:

```
-Dspring.profiles.active=prod
```

> 실무에서는 배포 자동화 도구, CI/CD에서 프로파일을 동적으로 지정함

🎯 4. 설정 병합 원리 (덮어쓰기 아님)

- Spring Boot는 **기본 설정(`application.yml`)을 먼저 로딩**하고,
- 이후 `application-{profile}.yml`을 **병합(override)**하는 방식

```yaml
# application.yml
server:
  port: 8080
  compression:
    enabled: true
```

```yaml
# application-prod.yml
server:
  port: 80
```

➡️ 최종 설정:

```yaml
server:
  port: 80
  compression:
    enabled: true
```

📘 5. 예제: 환경별 DB 설정 분리

```yaml
# application.yml
spring:
  datasource:
    username: default
    password: default
```

```yaml
# application-dev.yml
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/devdb
    username: dev
    password: dev

```

```yaml
# application-prod.yml
spring:
  datasource:
    url: jdbc:mysql://prod-db:3306/proddb
    username: prod
    password: prod
```

🔄 6. `@Profile` 어노테이션과 함께 사용하기

```java
@Service
@Profile("dev")
public class LocalMailService implements MailService { ... }

@Service
@Profile("prod")
public class SmtpMailService implements MailService { ... }
```

➡️ `spring.profiles.active`가 `dev`면 `LocalMailService`만 등록됨

📚 7. 고급: 다중 프로파일 지정

```yaml
spring:
  profiles:
    active: "dev,local"
```

- `application-dev.yml`, `application-local.yml`이 순서대로 적용됨
- 뒤에 나올수록 우선순위가 높음 (override)

🔐 8. 실무 운영 팁

팁	설명
`application.yml`은 공통 설정	공통 로깅, 타임존, JSON 직렬화 등
민감 정보는 별도 환경변수 관리	`application-prod.yml`에 절대 키 직접 쓰지 말기
CI/CD에서 `--spring.profiles.active` 전달	운영/스테이징 배포 시 자동화
`application-local.yml`은 `.gitignore`로 제외	개인 개발 설정은 공유 금지
설정 충돌 시 actuator로 확인	`/actuator/env`에서 실제 값 확인 가능

✅ 마무리 요약

항목	설명
기본 파일	`application.yml` (공통 설정)
환경별 분기	`application-{profile}.yml`
활성화 방법	`spring.profiles.active=prod`
병합 방식	위에 덮는 게 아니라 key 단위로 병합
실무 기준	프로파일별 파일 분리 + 자동화 설정 제어 + 민감정보 외부화

프로파일 기반 설정

- `@Profile`

🧩 1. `@Profile`이란?

> Spring에서 **특정 환경(Profile)**이 활성화되어 있을 때만
> **해당 Bean 또는 설정 클래스가 등록되도록 제어하는 어노테이션**

🎯 2. 왜 사용하는가?

- 개발(dev) 환경에서는 `로컬 DB`, `Mock API`, `Console Logger`
- 운영(prod) 환경에서는 `실제 DB`, `실제 SMTP`, `File Logger`

➡️ 환경에 따라 Bean 구성을 바꿔야 할 때
➡️ **조건부로 Bean을 등록/제외**하고 싶을 때 사용

⚙️ 3. 설정 방식 (활성 프로파일 지정)

✅ `application.yml`에서 지정

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

✅ 실행 시 파라미터로 지정

```
--spring.profiles.active=prod
```

📦 4. 사용 위치

대상	예시	설명
Bean 클래스	`@Component`, `@Service`, `@Repository`	해당 Bean 전체 등록 제어
설정 클래스	`@Configuration`	특정 설정 클래스만 환경에 따라 적용
Bean 메서드	`@Bean`	`@Configuration` 안에서 조건부 Bean 등록

📘 5. 실전 예제

✅ ① 컴포넌트에 직접

```
@Service
@Profile("dev")
public class LocalMailService implements MailService {
    public void send() { System.out.println("로컬 메일 발송"); }
}
java코드 복사@Service
@Profile("prod")
public class SmtpMailService implements MailService {
    public void send() { /* SMTP 발송 */ }
}
```

✅ ② 설정 클래스에 조건부 Bean 등록

```
@Configuration
public class AppConfig {

    @Bean
    @Profile("dev")
    public DataSource h2DataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .build();
    }

    @Bean
    @Profile("prod")
    public DataSource mysqlDataSource() {
        return DataSourceBuilder.create()
            .url("jdbc:mysql://prod-db:3306/app")
            .username("prod")
            .password("prod")
            .build();
    }
}
```

➡️ `dev`일 때는 H2, `prod`일 때는 MySQL 사용

🔁 6. 여러 프로파일 동시 지정

```
@Profile({"local", "dev"})
public class LocalService implements SomeService {}
```

> `local` 또는 `dev` 중 하나라도 활성화되면 등록됨

❌ 7. 비활성화 또는 특정 환경 제외

Spring 자체적으로는 `@Profile("!prod")`도 지원함 (Spring 4.0+)

```
@Profile("!prod")
public class NonProdLogger {}
```

> 단, 이 방식은 실수 방지를 위해 복잡한 환경에서는 **Bean 조건문이나** `@ConditionalOnProperty`로 대체하는 게 좋음

🧠 8. 실무 설계 패턴

목적	설계 방식
DB 커넥션 변경	`@Profile`에 따라 다른 DataSource 빈 등록
로컬용 구현체 vs 운영용 구현체	`@Service`, `@Component`에 `@Profile`
테스트 전용 구성	`@Profile("test")`로 Mockito Bean 등록
복잡한 조건이 필요할 때	`@Conditional` 또는 `@ConditionalOnProperty` 사용

✅ 마무리 요약

항목	설명
정의	특정 Profile이 활성화될 때만 해당 Bean을 등록
위치	클래스, 메서드, 설정 클래스
활성화 방법	`spring.profiles.active`로 지정
실무 용도	환경별 구현 분리, 설정 분기, 테스트 전용 Bean 구성
대안	복잡 조건 시 `@ConditionalOnProperty`, `@Conditional` 추천

커스텀 Starter 생성

🧩 1. 커스텀 Starter란?

> Spring Boot에서 **반복적인 설정, Bean 구성, 의존성 등을 패키지로 모듈화**해서
> 다른 프로젝트에서 `spring-boot-starter-xxx`처럼 **의존성만 추가하면 자동 적용되도록 만든 라이브러리**

🎯 2. 언제 쓰는가?

- 여러 프로젝트에서 공통으로 사용하는 기능(로그, 보안, 공통 API, 메일 등)을 모듈화하고 싶을 때
- 설정, Bean 등록, 프로퍼티 바인딩을 자동으로 처리하고 싶을 때
- 회사 전용 공통 Infra, 공통 보안 설정, 공통 메시징, 공통 컨트롤러 등 구성

⚙️ 3. 구성 요소 요약

구성 요소	역할
`starter` 모듈	다른 프로젝트에서 의존성 추가 시 사용

구성 요소	역할
`autoconfigure` 모듈	자동 설정 로직을 담은 모듈 (빈 등록, 조건 제어 등)
`spring.factories` 또는 `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`	자동 설정 클래스를 Spring Boot에 등록

🧱 4. 구조 예시

```
my-custom-starter/
├── my-starter-autoconfigure/     ← 자동 설정 모듈
|   └── ... (설정 클래스, properties, factories 등)
├── my-starter/                   ← 외부에 노출할 starter
|   └── pom.xml (자동 설정 모듈 포함)
```

> `my-starter`는 실제로 Bean 등록 X. 단지 `autoconfigure`를 transitively 포함하는 wrapper임.

📦 5. 실전 예제: "hello-starter"

✅ 1단계: `hello-spring-boot-autoconfigure` 생성

📄 `HelloProperties.java`

```java
@ConfigurationProperties(prefix = "hello")
public class HelloProperties {
    private String name = "World";
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

📄 `HelloService.java`

```java
public class HelloService {
    private final HelloProperties props;

    public HelloService(HelloProperties props) {
        this.props = props;
    }

    public String sayHello() {
        return "Hello, " + props.getName();
    }
}
```

`HelloAutoConfiguration.java`

```
@Configuration
@ConditionalOnClass(HelloService.class)
@EnableConfigurationProperties(HelloProperties.class)
public class HelloAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public HelloService helloService(HelloProperties props) {
        return new HelloService(props);
    }
}
```

`resources/META-INF/spring.factories` (Spring Boot 2.x용)

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.example.hello.HelloAutoConfiguration
```

✅ **또는 Spring Boot 3.x** 이상에서는:

```
resources/META-
INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports
```

```
com.example.hello.HelloAutoConfiguration
```

✅ 2단계: `hello-spring-boot-starter` 생성

```
<!-- hello-starter의 pom.xml -->
<dependencies>
  <dependency>
    <groupId>com.example</groupId>
    <artifactId>hello-spring-boot-autoconfigure</artifactId>
    <version>1.0.0</version>
  </dependency>
</dependencies>
```

🧪 6. 사용 예제 (다른 프로젝트에서)

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>hello-spring-boot-starter</artifactId>
  <version>1.0.0</version>
</dependency>
```

```yaml
hello:
  name: Jeongseok
```

```java
@RestController
public class HelloController {
    private final HelloService helloService;
    public HelloController(HelloService helloService) {
        this.helloService = helloService;
    }

    @GetMapping("/hello")
    public String hello() {
        return helloService.sayHello();  // → Hello, Jeongseok
    }
}
```

🔐 7. 실무 설계 팁

목적	설계 패턴
공통 설정 모듈화	`starter-autoconfigure`에 설정 클래스, properties 분리
외부 노출용은 starter만 제공	실제 설정은 내부에서 함
기능 활성화 제어	`@ConditionalOnProperty`로 ON/OFF 가능하게 설계
복잡한 기능 선택	`@ConditionalOnBean`, `@ConditionalOnClass` 조합 사용
설정 값 자동완성 지원	`spring-boot-configuration-processor` 추가

✅ 마무리 요약

항목	설명
`@Configuration` + `@Conditional*` + `@EnableConfigurationProperties`	자동 설정의 핵심 조합
`spring.factories` or `AutoConfiguration.imports`	Spring Boot가 자동으로 읽어들이는 경로
`starter` 모듈	외부에 제공할 wrapper
실무 유용성	공통 기능 재사용, 멀티 모듈 환경에서 대폭 생산성 향상

DevTools, Hot Reload

🧩 1. Spring Boot DevTools란?

> Spring Boot에서 제공하는 **개발 전용 도구 모듈**로,
> **자동 재시작**, **LiveReload**, **캐시 비활성화**, **속성 변경 감지**, **템플릿 핫 리로딩** 등을 지원함.

> ✔ 운영 환경에서는 자동 비활성화됨 (classpath 확인)

✅ 2. 주요 기능 요약

기능	설명
자동 재시작	클래스 파일이 변경되면 앱을 자동으로 재시작
LiveReload 지원	브라우저가 자동으로 새로고침됨
템플릿 캐시 제거	Thymeleaf, JSP 등의 캐시를 끔 (변경 즉시 반영)
설정값 변경 감지	`application.yml` 변경 시 자동 적용
H2 Console 자동 등록	기본 설정으로 편하게 확인 가능

⚙️ 3. 설치 방법

✅ Maven

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

✅ Gradle

```
runtimeOnly("org.springframework.boot:spring-boot-devtools")
```

> 🧠 `runtimeOnly` 또는 `optional`로 추가해야 **빌드 결과물에는 포함되지 않음**

📦 4. 작동 구조 요약

Spring DevTools는 다음과 같은 방식으로 작동해:

```
클래스 파일 변경 감지 (target/classes)
 → 내장 재시작 클래스로 전체 Context 재생성
 → 빠르게 앱 재시작
```

✅ **전체 JVM을 종료하지 않고, Spring Context만 재시작**하기 때문에 빠름.

🔁 5. 자동 재시작 vs LiveReload 차이

항목	자동 재시작	LiveReload
대상	Java 클래스, 설정 파일 변경	HTML, CSS, JS 등 정적 리소스 변경
방식	Spring Context 재시작	브라우저 자동 새로고침
실행 방식	Java Watcher (파일 감지)	LiveReload 서버 + 브라우저 플러그인
브라우저 개입	❌	✅ (확장 프로그램 또는 iframe 삽입 필요)

💡 6. IntelliJ 연동 팁

✅ "Make project automatically"

- `Ctrl + Shift + A` → "Registry" 검색 → `compiler.automake.allow.when.app.running` **체크**
- Preferences → Build, Execution, Deployment → Compiler → **Build project automatically ON**

> ☝️ 이 설정 없으면 **자동 재시작이 안 됨**

📘 7. 설정 예시

✅ `application.yml` 개발 전용 설정

```yaml
spring:
  thymeleaf:
    cache: false
  freemarker:
    cache: false
  devtools:
    restart:
      enabled: true
    livereload:
      enabled: true
```

🔥 8. 실무에서 자주 묻는 Q&A

질문	답변
운영 배포 시 포함되나요?	❌ classpath에 `spring-devtools`가 있으면 자동 비활성화됨
static/js, static/css 수정은 재시작하나요?	❌ LiveReload로 브라우저만 새로고침
Java 수정 → 빌드하면?	✅ 자동 재시작됨

질문	답변
속도 느린데요?	너무 많은 파일 변경, dependency scan 크기 문제 → `restart.exclude` 사용 권장
변경 반영이 안 되는데요?	IntelliJ 자동 빌드 설정 체크, 외부 빌드도 확인 (Gradle build task X)

🧠 9. 고급 설정 (`META-INF/spring-devtools.properties`)

```
# 특정 패키지나 파일은 재시작 제외
restart.exclude=static/**,public/**,templates/**,resources/**

# classloader 재활용 (속도 향상)
restart.include.custom=com.example.sharedlib
```

✅ 마무리 요약

항목	설명
`spring-boot-devtools`	개발 편의용 모듈 (운영 환경에서는 자동 제외됨)
자동 재시작	클래스 파일 변경 시 Spring Context만 재시작
LiveReload	HTML/CSS/JS 등 정적 리소스 변경 시 브라우저 자동 새로고침
IntelliJ 설정 필수	자동 빌드 + registry 설정 체크해야 실시간 적용됨
실무 팁	빠른 피드백 루프 + 템플릿 캐시 OFF + 안전하게 운영 분리됨