

20. 고급 내부 구조 및 커스터마이징

Spring 컨테이너 동작 방식

1. 스프링 컨테이너란?

Spring 컨테이너란?

→ 객체(Bean)의 생성, 관리, 의존성 주입, 생명주기 등을 담당하는 중앙 관리소이다.

핵심 역할

역할	설명
Bean 생성	Java 객체를 생성 (new)
Bean 조립	의존성 주입 (DI) 수행
Bean 생명주기 관리	초기화, 소멸 처리
AOP 적용	프록시로 기능 확장
환경 기반 설정	프로파일, 외부 설정 연동
이벤트 처리	이벤트 퍼블리싱 및 구독 처리

2. 컨테이너의 종류

컨테이너 구현체	설명
<code>BeanFactory</code>	가장 단순한 DI 컨테이너 (지연 로딩)
<code>ApplicationContext</code>	<code>BeanFactory</code> 확장, 실전에서 사용됨
<code>AnnotationConfigApplicationContext</code>	자바 기반 설정 전용
<code>GenericWebApplicationContext</code>	Spring MVC 환경에서 사용됨
<code>WebApplicationContext</code>	<code>DispatcherServlet</code> 과 연동된 웹 전용 컨텍스트

→ 실무에서는 거의 항상 `ApplicationContext` 또는 그 하위 구현체를 사용한다.

3. ApplicationContext 생성 과정

실행 흐름 요약

```
1 @SpringBootApplication
2 public class MyApp {
3     public static void main(String[] args) {
4         SpringApplication.run(MyApp.class, args);
5     }
6 }
```

```
5     }
6 }
```

실행 시 내부적으로:

```
1 SpringApplication → createApplicationContext()
2   → AnnotationConfigServletWebServerApplicationContext
3     → BeanDefinition 생성
4     → BeanFactoryPostProcessor 적용
5     → Bean 생성 (Constructor, DI)
6     → 초기화 콜백 (InitializingBean, @PostConstruct)
7     → ApplicationEventListener 등록
```

4. Bean 등록 단계 (Spring 내부 순서)

◆ 1단계: BeanDefinition 등록

```
1 @Configuration
2 public class AppConfig {
3     @Bean
4     public UserService userService() {
5         return new UserService();
6     }
7 }
```

→ 이 코드는 `UserService`에 대한 **설계도(BeanDefinition)**만 등록됨
아직 객체는 생성되지 않음

◆ 2단계: BeanFactoryPostProcessor 실행

```
1 @Component
2 public class MyProcessor implements BeanFactoryPostProcessor {
3     public void postProcessBeanFactory(ConfigurableListableBeanFactory bf) {
4         // 빈 정의 수정 가능
5     }
6 }
```

→ **Bean 생성 전** 설정 변경 가능

◆ 3단계: Bean 생성 및 의존성 주입

```
1 @Component
2 public class OrderService {
3     private final UserService userService;
4
5     public OrderService(UserService userService) { // 생성자 주입
6         this.userService = userService;
7     }
8 }
```

→ 의존하는 Bean이 모두 존재하면, 실제 객체를 생성하고 주입

◆ 4단계: 초기화 콜백

방식	설명
<code>@PostConstruct</code>	초기화 메서드 지정
<code>InitializingBean</code>	<code>afterPropertiesSet()</code>
<code>initMethod</code>	XML 또는 Java 설정에서 명시적으로 지정 가능

◆ 5단계: AOP 프록시 적용

- `@Transactional`, `@Async`, `@Cacheable` 등이 프록시 대상이면 → AOP 적용
- 실제 Bean 대신 **Proxy** 객체를 등록

◆ 6단계: ApplicationListener 등록 및 이벤트 퍼블리싱

```
1 @Component
2 public class MyListener implements ApplicationListener<ContextRefreshedEvent> {
3     public void onApplicationEvent(ContextRefreshedEvent event) {
4         System.out.println("컨텍스트 초기화 완료!");
5     }
6 }
```

5. 전체 생명주기 순서

1	1. 설정 클래스 탐색
2	2. BeanDefinition 스캔 및 등록
3	3. BeanFactoryPostProcessor 실행
4	4. Bean 생성 (Constructor)
5	5. 의존성 주입 (Setter, Constructor, Field)
6	6. BeanPostProcessor 실행
7	7. 초기화 콜백 (@PostConstruct 등)
8	8. AOP 프록시 적용
9	9. ApplicationEvent 등록
10	10. ApplicationRunner, CommandLineRunner 실행

6. BeanScope와 컨테이너

스코프	의미
singleton	한 컨텍스트에 하나의 인스턴스 (기본값)
prototype	요청 시마다 새로운 인스턴스
request	HTTP 요청 단위 (웹)
session	HTTP 세션 단위 (웹)
application	서블릿 컨텍스트 단위

→ prototype 스코프는 생성 후 컨테이너가 생명주기 관리하지 않음

7. 정리: 핵심 클래스 구조

클래스	역할
ApplicationContext	Spring 전체 컨테이너
DefaultListableBeanFactory	Bean 저장소, 생성 책임
AnnotationConfigApplicationContext	자바 기반 설정 전용 컨텍스트
BeanDefinition	Bean의 메타정보 (타입, scope, 생성자 등)
BeanPostProcessor	빈 생성 직후 개입 (ex: 프록시 생성)
ApplicationEventPublisher	이벤트 발행기
Environment	외부 설정 (@value, @ConfigurationProperties) 읽기

✅ 마무리 요약

개념	요약
컨테이너	스프링 객체(Bean)을 등록, 관리하는 핵심
동작 순서	등록 → 조립(DI) → 초기화 → 프록시 → 실행
관리 대상	@Component, @Bean, @Service 등으로 등록한 객체
구성 요소	BeanFactory, ApplicationContext, BeanPostProcessor 등
확장성	AOP, 이벤트, 프로파일, 외부 설정 등과 통합 가능

빈 등록 과정

📌 전체 흐름 요약

Spring 컨테이너가 실행되면, 아래 단계로 빈이 등록돼:

1. 설정 정보 분석
2. Component Scan or @Bean 분석
3. BeanDefinition 생성 및 등록
4. Bean 생성 전 처리 (Post Processor)
5. Bean 실제 생성 (생성자)
6. 의존성 주입
7. 초기화 콜백
8. 프록시 처리 (AOP)

🔍 1. 설정 클래스 분석

```
1 @SpringBootApplication // @ComponentScan + @Configuration + @EnableAutoConfiguration
2 public class App {
3     public static void main(String[] args) {
4         SpringApplication.run(App.class, args);
5     }
6 }
```

- 내부적으로 `AnnotationConfigApplicationContext` 또는 웹에선 `AnnotationConfigServletWebServerApplicationContext` 생성
- 이 클래스가 핵심 설정 클래스 (`@Configuration`)를 기준으로 모든 Bean 등록을 시작



2. Component Scan / Bean 등록

2.1 컴포넌트 스캔

```
1 @Component
2 public class UserService {}
```

→ 스프링은 `@ComponentScan` 으로 지정한 패키지에서 `@Component`, `@Service`, `@Repository`, `@Controller` 등을 찾아 `BeanDefinition`을 생성

2.2 @Bean 직접 등록

```
1 @Configuration
2 public class AppConfig {
3     @Bean
4     public OrderService orderService() {
5         return new OrderService();
6     }
7 }
```

→ 이 경우도 마찬가지로 `BeanDefinition` 객체가 생성됨



3. BeanDefinition 생성 및 등록

BeanDefinition이란?

Spring이 "이 클래스는 어떤 Bean으로 등록될 수 있는지에 대한 메타 정보"를 담고 있는 객체이다.

정보	예시
class	<code>com.example.UserService</code>
scope	singleton
lazy-init	false
beanName	<code>userService</code>
dependencies	<code>orderService</code> , <code>userRepository</code> 등

이 시점까지는 아직 **Bean** 객체가 생성되지 않음



4. BeanFactoryPostProcessor 처리

`ApplicationContext`가 초기화되기 전,

`BeanFactory`에 등록된 모든 `BeanDefinition`을 가공할 수 있는 **후처리**기들이 실행된다.

```

1 @Component
2 public class CustomProcessor implements BeanFactoryPostProcessor {
3     public void postProcessBeanFactory(ConfigurableListableBeanFactory factory) {
4         // 등록된 BeanDefinition 수정 가능
5     }
6 }

```

5. Bean 생성

드디어 실제 객체가 생성되었다.

생성 시점은?

- 기본적으로 ApplicationContext 초기화 시점에 모든 singleton Bean을 생성
- prototype은 요청할 때마다 생성

```

1 public class UserService {
2     public UserService(OrderService orderService) {
3         // 생성자 주입
4     }
5 }

```

→ 생성자, setter, field 등으로 의존성이 자동으로 주입됨

6. 의존성 주입 (DI)

DI는 다음 방식 중 하나로 수행된다:

방식	예시
생성자 주입	<code>@Autowired</code> 생성자
필드 주입	<code>@Autowired private UserRepository repo;</code>
세터 주입	<code>setXxx()</code> 에 <code>@Autowired</code>

Spring은 이를 위해 의존성 탐색 → 인스턴스 생성 → 주입 → 캐싱 과정을 거침

7. 초기화 콜백

생성 및 DI가 완료된 후, Bean은 초기화 단계로 넘어간다.

방법	설명
<code>@PostConstruct</code>	초기화 메서드 지정
<code>InitializingBean</code>	<code>afterPropertiesSet()</code> 메서드 호출

방법	설명
<code>@Bean(initMethod = "...")</code>	명시적 지정

8. BeanPostProcessor 처리

Spring은 생성된 Bean에 대해 **후처리기(BeanPostProcessor)**를 실행해서,

- AOP 프록시 적용
- `@Transactional`, `@Async`, `@Validated` 등 기능 부착
- 커스텀 동작 삽입

```

1 public class MyProcessor implements BeanPostProcessor {
2     public Object postProcessAfterInitialization(Object bean, String name) {
3         // 반환값이 바뀌면 프록시로 대체됨
4         return bean;
5     }
6 }
```

9. 실제 ApplicationContext에 Bean 등록 완료

이후부터는 `getBean("userService")`, `@Autowired UserService userService` 등이 전부 **ApplicationContext 내부의 싱글톤 빈 저장소에서 꺼내오는 것**

동작 순서 요약

```

1  @Bean 등록
2  ↓
3  BeanDefinition 생성
4  ↓
5  BeanFactoryPostProcessor 적용
6  ↓
7  Bean 생성 (Constructor)
8  ↓
9  DI 처리
10 ↓
11 BeanPostProcessor 적용 (AOP 등)
12 ↓
13 초기화 메서드 호출
14 ↓
15 ApplicationContext에 등록 완료
```


✅ 마무리 요약

단계	역할
BeanDefinition	클래스 → 메타데이터로 등록
BeanFactoryPostProcessor	빈 생성 전에 정의 수정
BeanPostProcessor	빈 생성 직후 후처리 (프록시 등)
실제 생성 시점	singleton: 컨테이너 초기화 시 / prototype: 요청 시
최종 저장 위치	ApplicationContext 내부의 싱글톤 레지스트리

BeanDefinition, BeanFactoryPostProcessor, BeanPostProcessor

1. BeanDefinition

개념

- `BeanDefinition`은 **Bean**을 정의하기 위한 메타정보를 담고 있는 객체이다.
- 즉, Spring이 클래스를 실제 **Bean**으로 등록하기 전의 설계도라고 생각하면 된다.

```
1 class BeanDefinition {
2     String beanClassName;    // 어떤 클래스인가?
3     String scope;            // singleton, prototype 등
4     boolean lazyInit;
5     List<PropertyValue> properties;
6     ConstructorArg[] constructorArgs;
7 }
```

생성 시점

- 컴포넌트 스캔, XML, `@Bean`, `@Import` 등으로부터 추출됨
- 이때까지는 객체는 생성되지 않음

```
1 @Bean
2 public UserService userService() {
3     return new UserService();
4 }
```

→ 위 코드가 있으면 내부적으로 `BeanDefinition` 객체가 먼저 만들어짐.

🔴 BeanDefinition의 주요 속성들

속성	설명
<code>beanClassName</code>	클래스 이름 (com.example.UserService)
<code>scope</code>	<code>singleton</code> , <code>prototype</code> , <code>request</code> , <code>session</code> 등
<code>lazyInit</code>	true면 초기화 지연
<code>initMethodName</code>	초기화 콜백 메서드
<code>propertyValues</code>	DI할 필드 값들 (setter 주입용)
<code>constructorArgs</code>	생성자 파라미터 값들

2. 📌 BeanFactoryPostProcessor

🔴 개념

- **BeanDefinition**을 수정할 수 있는 후처리기.
- 즉, Bean이 실제로 생성되기 전 단계에서, **정의 자체를 가공**할 수 있다.

🔴 사용 목적

사용 예	설명
<code>PropertySourcesPlaceholderConfigurer</code>	<code>@Value("\${...}")</code> 치환 처리
<code>ConfigurationClassPostProcessor</code>	<code>@Configuration</code> , <code>@Bean</code> 등록
Custom Bean 등록기	코드로 빈을 수동 등록

💡 실전 예제

```
1  @Component
2  public class MyBFPP implements BeanFactoryPostProcessor {
3      @Override
4      public void postProcessBeanFactory(ConfigurableListableBeanFactory factory) {
5          for (String name : factory.getBeanDefinitionNames()) {
6              BeanDefinition def = factory.getBeanDefinition(name);
7              if (name.equals("targetBean")) {
8                  def.setScope("prototype");
9                  def.setLazyInit(true);
10             }
11         }
12     }
13 }
```

→ `targetBean`의 scope과 lazy 속성을 코드에서 변경할 수 있음

📌 실행 시점

- Spring이 ApplicationContext를 생성할 때, **Bean이 생성되기 전에 단 한 번 실행됨**
 - 단, `@Component`로 등록되거나 `@Import`로 포함돼야 함
-

3. 📌 BeanPostProcessor

📌 개념

- **Bean 객체가 생성된 이후, 초기화 직전에 개입하는 확장 포인트**
 - 가장 많이 쓰이는 Spring의 확장기술(AOP, `@Transactional`, `@Async` 등)의 기반
-

📌 동작 순서

1. 생성자 호출
 2. 의존성 주입 완료
 3. 초기화 콜백 이전
 4. **BeanPostProcessor의 `postProcessBeforeInitialization()` 실행**
 5. `@PostConstruct`, `InitializingBean.afterPropertiesSet()` 실행
 6. **`postProcessAfterInitialization()` 실행**
 7. ApplicationContext에 등록 완료
-

💡 실전 예제

```
1  @Component
2  public class MyBPP implements BeanPostProcessor {
3
4      @Override
5      public Object postProcessBeforeInitialization(Object bean, String name) {
6          if (bean instanceof UserService) {
7              System.out.println("BeforeInit: " + name);
8          }
9          return bean;
10     }
11
12     @Override
13     public Object postProcessAfterInitialization(Object bean, String name) {
14         if (bean instanceof UserService) {
15             System.out.println("AfterInit: " + name);
16         }
17         return bean;
18     }
19 }
```

📌 주요 활용 예

활용	설명
AOP	프록시 객체로 교체
<code>@Transactional</code>	트랜잭션 프록시 적용
<code>@Async</code> , <code>@Scheduled</code>	쓰레드 비동기화
<code>@Validated</code>	JSR-303 유효성 검사
로깅/트래킹	자동 주입된 로거 삽입 등

4. Bean 생명주기와 이 3가지의 위치 관계

1. BeanDefinition 생성 ← [BeanDefinition]
2. BeanFactoryPostProcessor 실행 ← [BeanFactoryPostProcessor]
3. Bean 생성 (new) → 의존성 주입
4. BeanPostProcessor.beforeInit() ← [BeanPostProcessor]
5. 초기화 콜백
6. BeanPostProcessor.afterInit() ← [BeanPostProcessor]
7. ApplicationContext에 등록 완료

✅ 마무리 요약 비교

항목	시점	대상	역할
<code>BeanDefinition</code>	컨테이너 초기화 초반	메타 정보	Bean의 설계도
<code>BeanFactoryPostProcessor</code>	빈 생성 전	BeanDefinition	정의 자체 수정
<code>BeanPostProcessor</code>	빈 생성 후	실제 객체	초기화, 프록시

✳️ 고급 확장: 그 외 관련 인터페이스들

인터페이스	설명
<code>SmartInitializingSingleton</code>	모든 Bean 생성 완료 후 실행
<code>InstantiationAwareBeanPostProcessor</code>	생성자 직전/직후에 개입 가능
<code>DestructionAwareBeanPostProcessor</code>	소멸 직전에 후처리
<code>MergedBeanDefinitionPostProcessor</code>	병합된 BeanDefinition 처리
<code>ImportBeanDefinitionRegistrar</code>	외부에서 Bean 정의 직접 등록

클래스패스 스캔 원리

📌 개념 정리

클래스패스 스캔이란?

Spring이 지정된 패키지 내의 `.class` 파일을 리플렉션 기반으로 탐색하여

`@Component` 계열 어노테이션이 붙은 클래스를 자동으로 **BeanDefinition**으로 등록하는 동작

🔄 작동 흐름 요약

```
1 @Configuration
2 @ComponentScan(basePackages = "com.example")
3 public class AppConfig {
4 }
```

⬇ 실행 시

```
1 1. basePackages 경로 스캔
2 2. .class 파일 메타데이터 읽기 (ASM)
3 3. @Component, @Service 등 있는 클래스 식별
4 4. BeanDefinition 생성
5 5. ApplicationContext에 등록
```

1. 📦 기본 출발점: `@ComponentScan`

```
1 @SpringBootApplication // 내부에 @ComponentScan 포함
2 public class MainApp {
3     public static void main(String[] args) {
4         SpringApplication.run(MainApp.class, args);
5     }
6 }
```

→ 기본적으로 `@SpringBootApplication`은 현재 클래스의 패키지 기준으로 하위 전체를 스캔

```
1 @Target(ElementType.TYPE)
2 @Retention(RetentionPolicy.RUNTIME)
3 @ComponentScan(
4     basePackages = ...
5 )
```

2. 🔍 어떤 어노테이션이 대상인가?

어노테이션	설명
<code>@Component</code>	기본 빈 등록 어노테이션
<code>@Service</code>	<code>@Component</code> 의 특수화
<code>@Repository</code>	<code>@Component</code> 의 특수화 (예외 변환 기능 내장)
<code>@Controller</code>	MVC 전용
<code>@RestController</code>	<code>@Controller</code> + <code>@ResponseBody</code>

이들은 전부 내부적으로 `@Component`가 붙어 있음.

3. ⚙️ 실제 스캔 로직의 핵심 클래스

클래스	역할
<code>ClassPathBeanDefinitionScanner</code>	<code>.class</code> 파일 스캔 수행
<code>MetadataReader</code>	<code>.class</code> 메타정보 리딩
<code>AnnotationMetadata</code>	어노테이션 정보 읽기
<code>BeanDefinitionRegistry</code>	스캔 결과를 등록할 대상 컨테이너

4. 📁 내부 동작 상세 순서

① `basePackage` 경로 탐색

- `com.example` → `classpath:/com/example/`로 변환
- 내부의 `.class` 파일들 탐색

② `.class` 파일 메타정보 읽기 (ASM 라이브러리)

Spring은 클래스 전체를 로드하지 않고도 클래스의 어노테이션/슈퍼클래스/메서드 시그니처 등을 추출할 수 있음.

```
1 MetadataReaderFactory metadataReaderFactory = new CachingMetadataReaderFactory();
2 MetadataReader reader = metadataReaderFactory.getMetadataReader(classResource);
3 AnnotationMetadata metadata = reader.getAnnotationMetadata();
```

- 빠르고 효율적인 이유: 클래스를 메모리에 로딩하지 않고 `bytecode` 레벨에서 분석

③ 필터 적용

```
1 includeFilters = {
2     @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = MyMarker.class)
3 }
4 excludeFilters = {
5     @ComponentScan.Filter(type = FilterType.REGEX, pattern = ".*Internal.*")
6 }
```

→ @Component 외에도 커스텀 필터 조건으로 포함/제외 가능

④ BeanDefinition 생성 및 등록

- 탐색된 클래스의 메타정보로 `ScannedGenericBeanDefinition` 생성
- `beanNameGenerator` 로 이름 결정 (보통 `userService`, `orderService` 등)
- 최종적으로 `DefaultListableBeanFactory` 에 등록

```
1 registry.registerBeanDefinition(beanName, beanDefinition);
```

5. 💡 커스텀 Component 스캔 조건 (고급)

```
1 @Target(ElementType.TYPE)
2 @Retention(RetentionPolicy.RUNTIME)
3 @Component
4 public @interface MyComponent {
5 }
```

```
1 @ComponentScan(includeFilters = {
2     @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = MyComponent.class)
3 })
```

→ 직접 만든 어노테이션도 스캔 대상으로 설정 가능

6. ⚠ 주의할 점

실수	설명
@ComponentScan의 basePackages 누락	기본 위치가 현재 클래스 패키지 기준이라 상위 패키지의 Bean이 누락될 수 있음
@Component 빠짐	어노테이션이 없으면 스캔 대상이 아님
빈 이름 충돌	같은 이름의 Bean이 여러 개 등록되면 에러
클래스를 직접 로딩하지 않음	리플렉션 기반이 아니라 bytecode 탐색 (성능 최적화)

7. 실전 예: 스캔 대상 조건과 필터 조합

```
1 @ComponentScan(  
2     basePackages = "com.example.app",  
3     includeFilters = {  
4         @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = Service.class)  
5     },  
6     excludeFilters = {  
7         @ComponentScan.Filter(type = FilterType.REGEX, pattern = ".*Internal.*")  
8     }  
9 )
```

✅ 마무리 요약

항목	설명
목적	지정된 패키지에서 Bean 어노테이션이 붙은 클래스를 탐색
핵심 클래스	<code>ClassPathBeanDefinitionScanner</code>
탐색 방법	클래스 전체 로딩 없이 ASM으로 <code>.class</code> bytecode 읽음
대상 어노테이션	<code>@Component</code> , <code>@Service</code> , <code>@Repository</code> , <code>@Controller</code> , 커스텀
결과물	<code>ScannedGenericBeanDefinition</code> → BeanFactory에 등록

커스텀 어노테이션과 메타 어노테이션

◆ 1. 개념 정의

용어	설명
커스텀 어노테이션	사용자가 직접 정의한 어노테이션
메타 어노테이션	어노테이션 위에 붙는 어노테이션 (즉, 어노테이션을 설명하는 어노테이션)

📌 2. 메타 어노테이션 종류 (Java + Spring)

어노테이션	설명
<code>@Target</code>	어노테이션을 붙일 수 있는 위치 (클래스, 메서드 등)
<code>@Retention</code>	어노테이션이 유지되는 범위 (소스, 클래스, 런타임)
<code>@Documented</code>	Javadoc에 포함 여부
<code>@Inherited</code>	상속 여부
<code>@Component</code>	Spring Bean 등록 메타 어노테이션

어노테이션	설명
<code>@Service</code> , <code>@Controller</code>	전부 <code>@Component</code> 의 메타 어노테이션
<code>@Qualifier</code> , <code>@Transactional</code> , <code>@RequestMapping</code>	Spring에서 메타 정보를 가진 기능성 어노테이션

3. 커스텀 어노테이션 만드는 기본 구조

예: 커스텀 `@MyService` 만들기

```

1  @Target(ElementType.TYPE)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Component // 중요: 이걸 붙여야 ComponentScan 대상이 됨
5  public @interface MyService {
6      String value() default "";
7  }
```

`@Component`가 메타 어노테이션으로 들어가 있어야 Spring이 이걸 Bean으로 인식함

사용 예:

```

1  @MyService("orderService")
2  public class OrderService {
3      ...
4  }
```

→ 이 클래스는 자동으로 **Spring Bean**으로 등록되고, 이름은 `"orderService"`가 됨

4. 커스텀 어노테이션 + 메타 어노테이션 실전 패턴

(1) 커스텀 컨트롤러 + API 버전

```

1  @Target(ElementType.TYPE)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @RestController
5  @RequestMapping("/api/v1")
6  public @interface ApiV1Controller {
7  }
```

→ 사용 시:

```

1 @ApiV1Controller
2 public class UserController {
3     @GetMapping("/users")
4     public List<User> list() { ... }
5 }

```

(2) 커스텀 트랜잭션 롤백 어노테이션

```

1 @Target({ElementType.METHOD, ElementType.TYPE})
2 @Retention(RetentionPolicy.RUNTIME)
3 @Transactional(rollbackFor = Exception.class)
4 public @interface RollbackOnException {
5 }

```

→ `@RollbackOnException` 만 붙이면 자동으로 `@Transactional(rollbackFor = Exception.class)` 효과

(3) 커스텀 유효성 검증

```

1 @Target(ElementType.FIELD)
2 @Retention(RetentionPolicy.RUNTIME)
3 @Constraint(validatedBy = MyValidator.class)
4 public @interface Myvalidation {
5     String message() default "유효하지 않음";
6     Class<?>[] groups() default {};
7     Class<? extends Payload>[] payload() default {};
8 }

```

5. 커스텀 어노테이션 읽기 (리플렉션 기반)

```

1 MyService annotation = clazz.getAnnotation(MyService.class);
2 String name = annotation.value();

```

→ Bean 등록 여부를 확인하거나 동작을 조건부로 제어할 수 있음

6. 실무에서 자주 쓰는 커스텀 어노테이션 예시

목적	어노테이션 예
API 버전 구분	<code>@ApiV1Controller</code> , <code>@ApiV2Controller</code>
로깅 공통 적용	<code>@Loggable</code> , <code>@Trace</code>
보안 권한	<code>@AdminOnly</code> , <code>@LoginRequired</code>
트랜잭션 커스터마이징	<code>@ReadOnlyTransaction</code> , <code>@WriteTransaction</code>

목적	어노테이션 예
테스트 분류	@UnitTest, @IntegrationTest
공통 유효성 검사	@ValidEmail, @PasswordPolicy

⚠ 7. 주의사항

항목	주의
@Component 계열 메타 어노테이션 없으면 스캔 대상 ❌	
@Target 정확히 명시하지 않으면 사용 위치 제한됨	
@Retention(RUNTIME) 이 아니면 리플렉션에서 인식 ❌	
이름 중복 발생 시 충돌 가능 (value, name 등의 속성)	
스프링에서 메타 어노테이션을 인식하는 조건도 있음 (예: AOP는 RUNTIME 유지 필수)	

✅ 마무리 요약

항목	설명
커스텀 어노테이션	사용자가 직접 정의한 기능성 태그
메타 어노테이션	어노테이션 자체를 설명하는 어노테이션 (@Target, @Retention, @Component 등)
실전 예	@MyService, @ApiController, @TransactionalService
Bean 등록 조건	@Component, @Service 같은 메타 어노테이션 필요
리플렉션 기반 동작	AOP, Validator, 인터셉터 등에서 동작 감지 가능

SPI 구조

📌 1. SPI란 무엇인가?

SPI (Service Provider Interface) 는
외부 라이브러리나 사용자 정의 모듈이 특정 인터페이스 구현체를 제공하고,
애플리케이션 런타임에 이를 자동으로 탐지하여 로드할 수 있도록 하는 구조이다.

즉, “구현체는 나중에 주고, 시스템은 알아서 찾아서 쓰게 해줘”라는 설계 방식이다.

2. Java 표준 SPI 구조

기본 구성

1. 인터페이스를 정의한다
2. 그 인터페이스의 구현체를 제공한다
3. 구현체를 `META-INF/services/` 폴더에 등록한다
4. `ServiceLoader` 로 로딩한다

예시

① 인터페이스 정의

```
1 public interface NotificationService {
2     void send(String message);
3 }
```

② 구현체 작성

```
1 public class EmailNotificationService implements NotificationService {
2     public void send(String message) {
3         System.out.println("Email: " + message);
4     }
5 }
```

③ SPI 등록 파일 생성

 `src/main/resources/META-INF/services/com.example.NotificationService`

```
1 com.example.EmailNotificationService
```

이 파일은 라인마다 한 개의 구현 클래스 이름을 적는 포맷

④ ServiceLoader로 로딩

```
1 ServiceLoader<NotificationService> loader =
2     ServiceLoader.load(NotificationService.class);
3
4 for (NotificationService service : loader) {
5     service.send("Hello SPI!");
6 }
```

3. Spring이 SPI를 쓰는 방식

Spring 자체도 많은 부분에서 **Java SPI + Spring식 SPI**를 혼용해서 확장 구조를 만듦.

대표적인 Spring SPI 기반 확장 포인트

SPI 인터페이스	설명
<code>org.springframework.context.ApplicationContextInitializer</code>	Spring Context 초기 확장
<code>org.springframework.boot.SpringApplicationRunListener</code>	Spring Boot 앱 초기화 감시자
<code>org.springframework.boot.env.EnvironmentPostProcessor</code>	환경 설정 조작
<code>org.springframework.beans.factory.support.BeanDefinitionRegistryPostProcessor</code>	빈 정의 동적 등록
<code>org.springframework.core.io.support.SpringFactoriesLoader</code>	Spring의 SPI 구현체 로더

META-INF/spring.factories

Spring Boot에서는 SPI 등록을 `spring.factories` 파일을 통해 수행한다.

 `src/main/resources/META-INF/spring.factories`

```
1 org.springframework.boot.SpringApplicationRunListener=\
2 com.example.MyCustomRunListener
3
4 org.springframework.context.ApplicationContextInitializer=\
5 com.example.MyContextInitializer
```


4. 핵심 클래스: SpringFactoriesLoader

```
1 List<MySPI> implementations = SpringFactoriesLoader.loadFactories(MySPI.class,
  classLoader);
```

- Spring은 Java의 `ServiceLoader`를 확장해서 `spring.factories` 파일에서 SPI 구현체들을 로드함
- 여러 개가 자동으로 로딩되며, 순서를 제어할 수 있음 (`@Order`, `Ordered` 인터페이스)

5. 실전 예제: ApplicationContextInitializer

```
1 public class MyInitializer implements
  ApplicationContextInitializer<ConfigurableApplicationContext> {
2     @Override
3     public void initialize(ConfigurableApplicationContext context) {
4         System.out.println("🔥 MyInitializer 실행됨!");
5     }
6 }
```

 META-INF/spring.factories

```
1 org.springframework.context.ApplicationContextInitializer=\n2 com.example.MyInitializer
```

Spring Boot가 실행되면 자동으로 초기화에 개입됨.

6. Java SPI vs Spring SPI 비교

항목	Java SPI (ServiceLoader)	Spring SPI (SpringFactoriesLoader)
등록 위치	META-INF/services/	META-INF/spring.factories
다중 지원	가능	가능
순서 제어	불가	가능 (@Order , Ordered)
리플렉션 사용	예	예
성능 최적화	기본	캐싱 및 다중 로딩 최적화
사용 사례	JDBC, SLF4J 등	Spring Boot 자동 설정, 컨텍스트 초기화 등

7. 마무리 요약

개념	설명
SPI	인터페이스를 외부에서 구현하고 런타임에 로딩하는 패턴
Java SPI	ServiceLoader + META-INF/services/
Spring SPI	SpringFactoriesLoader + META-INF/spring.factories
사용 목적	모듈 확장성, 느슨한 결합, 동적 로딩
적용 예	AutoConfiguration, AOP, Context 초기화기, Bean 등록기 등

ClassLoader 및 Reflection 활용

1. ClassLoader란?

JVM에서 클래스 파일(.class)을 읽어와 메모리(메타 영역)에 적재하는 동적 로딩 시스템이다.

동작 원리

```
1 .class 파일 ↓\n2 → ClassLoader → Class 객체 생성 → 메타영역 등록
```

◆ ClassLoader 계층 구조

ClassLoader	설명	예
BootstrapClassLoader	JVM 내장	<code>java.lang.*</code> , <code>java.util.*</code>
ExtensionClassLoader	<code>ext</code> 디렉터리 클래스	JDK 확장 API
ApplicationClassLoader	<code>classpath</code> 의 모든 클래스	우리가 작성한 코드
(Spring) URLClassLoader	동적으로 Jar 로드	플러그인 로딩

✅ 주요 메서드

```
1 ClassLoader cl = Thread.currentThread().getContextClassLoader();
2 Class<?> clazz = cl.loadClass("com.example.MyClass");
```

✨ 2. Reflection (리플렉션)이란?

Java에서 클래스, 메서드, 필드, 생성자 등 메타정보를 런타임에 동적으로 접근하고 조작할 수 있는 기능.

🔧 주요 API

메서드	설명
<code>Class.forName("...")</code>	클래스 로딩
<code>getDeclaredMethods()</code>	메서드 목록
<code>getConstructor()</code>	생성자 얻기
<code>newInstance()</code>	객체 동적 생성
<code>setAccessible(true)</code>	private 필드/메서드 접근 가능

📌 예제

```
1 Class<?> clazz = Class.forName("com.example.UserService");
2
3 Object instance = clazz.getDeclaredConstructor().newInstance();
4
5 Method method = clazz.getMethod("sayHello");
6 method.invoke(instance);
```

3. Spring에서의 ClassLoader + Reflection 활용

(1) 클래스패스 스캔

- Spring은 컴포넌트 스캔 시 `.class` 파일을 모두 읽고, **ASM (bytecode parser)** 를 사용해 클래스 이름/어노테이션만 읽고 실제 클래스를 **로드하지 않음** (`Class.forName` ❌)

```
1 ClassLoader cl = getClass().getClassLoader();
2 MetadataReader reader = metadataReaderFactory.getMetadataReader(resource);
3 AnnotationMetadata metadata = reader.getAnnotationMetadata();
```

→ 성능 개선과 프록시 적용을 위해 **클래스 로딩 최소화**

(2) DI (의존성 주입)

```
1 Constructor<?> constructor = clazz.getConstructor(Dependency.class);
2 Object bean = constructor.newInstance(dependency);
```

→ Spring은 객체 생성 시 생성자/필드/메서드를 Reflection으로 분석하여 자동 주입

(3) AOP 프록시 생성

```
1 ProxyFactory factory = new ProxyFactory(target);
2 factory.addAdvice(new MyAdvice());
3 Object proxy = factory.getProxy();
```

→ `java.lang.reflect.Proxy` 또는 CGLIB을 이용해 런타임에 프록시 클래스 생성

(4) @Autowired, @Value, @Transactional 등 처리

- 리플렉션을 사용해서 `@Autowired` 가 붙은 필드를 찾아 빈을 주입
- `@Transactional` 이 붙은 메서드를 찾아 트랜잭션 경계를 프록시로 감쌘
- `@Value("${...}")` → Environment에서 읽어서 필드에 주입

(5) 커스텀 어노테이션 처리

```
1 Field[] fields = clazz.getDeclaredFields();
2 for (Field f : fields) {
3     if (f.isAnnotationPresent(MyAnnotation.class)) {
4         ...
5     }
6 }
```


→ 런타임에 어노테이션 정보 읽고 동작 결정 가능

🚧 4. 성능 및 보안 이슈

항목	설명
성능 이슈	리플렉션은 일반 호출보다 느림 (JIT 최적화 어려움)
보안 제한	<code>setAccessible(true)</code> 는 Java 9 이후 모듈 보안 이슈 발생 가능
클래스 로딩 충돌	커스텀 ClassLoader에서 같은 클래스 중복 로딩 문제 발생 가능

✅ 5. 마무리 요약

기술	역할	예시
ClassLoader	<code>.class</code> 를 JVM에 적재	<code>ClassLoader.loadClass(...)</code>
Reflection	클래스 구조에 런타임 접근	<code>clazz.getMethod(...)</code> , <code>method.invoke(...)</code>
Spring 활용	Bean 생성, DI, AOP, AutoConfiguration	<code>@Autowired</code> , <code>@ComponentScan</code> , <code>@Transactional</code> 등

📌 추가: Spring Boot AutoConfiguration에서의 활용

- Spring Boot는 `META-INF/spring.factories` 를 기반으로 클래스 이름을 읽은 뒤, 실제 클래스를 **ClassLoader로 로딩하고**, 조건에 따라(`@ConditionalOnClass`) Bean을 생성

```
1 | if (ClassUtils.isPresent("com.fasterxml.jackson.databind.ObjectMapper", classLoader)) {
2 |     // ObjectMapper가 classpath에 있으면 JSON 자동 구성
3 | }
```