

Spring Framework Core

1. Spring Core의 핵심 개념
2. IoC, DI, Bean, 어노테이션 분석

Y-A, Dominica KIM

Spring Framework란?

- **오픈소스 자바 엔터프라이즈 프레임워크** - 자바 기반의 오픈소스 프레임워크로 엔터프라이즈급 애플리케이션 개발을 위한 다양한 기능 제공
- **경량 컨테이너** - 객체 생성과 소멸을 관리하며 싱글톤 패턴으로 메모리 효율성 보장
- **확장성 및 유지보수** - 모듈식 아키텍처로 대규모 애플리케이션의 확장성과 유지보수 용이

스프링의 역사와 발전

1 2002년

Rod Johnson이 'Expert One-on-One J2EE Design and Development'에서 처음 소개

2 2003년

오픈소스 프로젝트로 공개되어 전 세계 개발자 커뮤니티에 확산

3 2014년~현재

Spring Boot, Spring Cloud 등으로 생태계 확장 및 마이크로서비스 지원 강화

Spring과 Spring Boot의 차이점

Spring Framework와 Spring Boot의 주요 차이점 비교

구분	Spring Framework	Spring Boot
설정 방식	XML 또는 Java 기반 설정 파일 필요	자동 설정(Auto Configuration) 지원
서버 구동	외부 서버 설정 필요(Tomcat, Jetty 등)	내장 서버(Embedded Server) 제공
의존성 관리	각 라이브러리별 버전 관리 필요	스타터(Starter)를 통한 의존성 자동 관리
배포 방식	WAR 파일 생성 후 웹 서버에 배포	JAR 파일로 독립 실행 가능(Standalone)
개발 생산성	초기 설정 및 환경 구성에 시간 소요	빠른 개발 시작과 마이크로서비스 지원
프로젝트 구조	개발자가 직접 구조 설계 필요	프로젝트 구조와 규칙 자동 생성
모니터링	별도 모니터링 도구 연동 필요	Actuator를 통한 모니터링 기능 기본 제공

Spring과 Spring Boot 선택 가이드

프로젝트 요구사항에 맞는 프레임워크 선택 기준

Spring Framework 선택 시기

- 기존 레거시 시스템과의 통합이 필요할 때
- 애플리케이션 구성을 세밀하게 제어해야 할 때
- 특정 서버 환경에 최적화가 필요한 경우
- 맞춤형 설정과 구성이 중요한 대규모 엔터프라이즈 애플리케이션

Spring Boot 선택 시기

- 빠른 개발 및 배포가 필요한 프로젝트
- 마이크로서비스 아키텍처 구현 시
- 개발 생산성과 빠른 시작이 중요한 경우
- 클라우드 네이티브 애플리케이션 개발 시
- 최소한의 설정으로 빠르게 시작하려는 신규 프로젝트

프로젝트의 규모, 개발 속도, 유연성 요구사항에 따라 적절한 프레임워크를 선택하는 것이 중요하다.

Spring 온라인 리소스

리소스 이름	유형	설명
Spring 공식 문서	문서	Spring Framework에 대한 공식 가이드 및 레퍼런스 문서
Spring Boot 문서	문서	Spring Boot에 특화된 문서 및 가이드
Baeldung	튜토리얼	Spring 관련 깊이 있는 튜토리얼과 예제 코드 제공
Spring 블로그	블로그	Spring 팀의 공식 블로그로 최신 업데이트 및 기술 정보 제공
Spring 프로젝트 GitHub	소스 코드	Spring 프로젝트의 소스 코드 저장소
StackOverflow	커뮤니티	개발자들의 질문과 답변으로 이루어진 지식 공유 플랫폼
InfoQ Spring 섹션	뉴스/아티클	Spring 관련 최신 트렌드와 심층 분석 아티클
Spring Academy	교육	VMware에서 제공하는 Spring 공식 교육 과정

스프링 개발 환경 및 리소스

스프링 프레임워크 개발에 활용할 수 있는 주요 IDE 및 개발 도구

개발 환경/사이트	유형	특징	공식 웹사이트
Eclipse (STS)	IDE	Spring Tool Suite를 통한 스프링 전용 기능 제공	spring.io/tools
IntelliJ IDEA	IDE	스프링 프로젝트 지원이 뛰어난 통합 개발 환경	jetbrains.com/idea
VSCode	경량 에디터	확장 프로그램을 통한 스프링 개발 지원	code.visualstudio.com
Spring Initializr	웹 서비스	스프링 프로젝트 생성 및 의존성 설정 도구	start.spring.io
JDOODLE	온라인 컴파일러	간단한 스프링 코드 테스트 가능한 온라인 도구	jdoodle.com
GitPod	클라우드 IDE	브라우저에서 스프링 개발 환경 제공	gitpod.io

Visual Studio Code 확장 프로그램

Spring 개발을 위한 필수 VSCode 확장 프로그램

확장 프로그램	설명
Spring Boot Extension Pack	Spring Boot 애플리케이션 개발을 위한 통합 확장 팩으로 여러 Spring 관련 도구를 포함
Extension Pack for Java	Java 개발에 필요한 디버깅, 테스트, 리팩토링 및 Maven/Gradle 지원 확장 팩
Spring Boot Dashboard	스프링 부트 애플리케이션 관리 및 실행을 위한 대시보드 인터페이스 제공
XML	XML 파일 편집, 유효성 검사 및 자동 완성 기능을 제공하여 Spring XML 구성 작업 지원
Spring Boot Tools	application.properties 및 application.yml 파일을 위한 고급 지원 및 자동완성 기능

* 모든 확장 프로그램은 VSCode 마켓플레이스에서 무료로 설치 가능

스프링에서 디자인 패턴을 알아야 하는 이유

스프링 프레임워크는 다양한 디자인 패턴을 기반으로 설계되었으며, 이러한 패턴을 이해하면 스프링의 내부 동작 원리를 더 깊이 파악할 수 있다.

프레임워크 이해도 향상

스프링의 핵심 기능인 IoC, DI, AOP 등은 모두 디자인 패턴을 기반으로 구현되어 있어, 패턴을 이해하면 스프링을 더 효과적으로 활용할 수 있다.

코드 품질 개선

디자인 패턴을 적용하면 유지보수성, 확장성이 높은 코드를 작성할 수 있으며, 스프링과의 통합이 더욱 자연스러워진다.

문제 해결 능력 강화

스프링에서 발생하는 문제들을 디자인 패턴 관점에서 분석하면 근본 원인을 파악하고 해결책을 도출하는 데 도움이 된다.

결과적으로, 디자인 패턴에 대한 지식은 단순히 스프링 API를 사용하는 차원을 넘어 프레임워크를 마스터하고 고급 개발자로 성장하기 위한 필수 요소이다.

객체 지향 디자인 패턴의 종류와 개념

디자인 패턴은 소프트웨어 개발 과정에서 발생하는 문제를 해결하기 위한 검증된 솔루션으로, 크게 3가지로 분류되어 있다.

- **생성 패턴(Creational Patterns)**: 객체 생성 메커니즘을 다루는 패턴으로, 객체가 생성되는 방식을 유연하게 제어하여 상황에 적합한 객체를 생성하고 있다.
- **구조 패턴(Structural Patterns)**: 클래스와 객체를 더 큰 구조로 조합하는 방법을 다루는 패턴으로, 유연하고 효율적인 구조를 설계하고 있다.
- **행위 패턴(Behavioral Patterns)**: 객체 간의 상호작용과 책임 분배를 다루는 패턴으로, 객체 간 커뮤니케이션을 개선하고 있다.

GoF(Gang of Four) 디자인 패턴은 에릭 감마(Erich Gamma), 리차드 헬름(Richard Helm), 랄프 존슨(Ralph Johnson), 존 블리시디스(John Vlissides)가 1994년 '디자인 패턴: 재사용 가능한 객체지향 소프트웨어의 요소'라는 책에서 처음 소개한 23가지 디자인 패턴을 의미하고 있다.

이 패턴들은 객체 지향 설계에서 자주 발생하는 문제들에 대한 표준화된 해결책으로, 소프트웨어 개발자들 사이에서 공통 언어를 제공하며 코드의 품질, 유지보수성, 확장성을 크게 향상시키고 있다.

객체 지향 디자인 패턴의 종류와 개념

GoF(Gang of Four)에서 정의한 생성 패턴은 객체의 생성 과정에 관한 패턴입니다

패턴명	개념	실무 사용 예
싱글톤(Singleton)	클래스의 인스턴스가 하나만 생성되도록 보장하는 패턴	스프링 빈, 데이터베이스 연결 객체, 로깅 인스턴스
팩토리 메소드(Factory Method)	객체 생성을 서브클래스로 위임하는 패턴	프레임워크 API, UI 컴포넌트 생성, JDBC 드라이버 매니저
추상 팩토리(Abstract Factory)	관련된 객체들의 집합을 생성하는 인터페이스 제공	UI 테마 시스템, 데이터베이스 구현체 교체, 다양한 플랫폼 호환 컴포넌트
빌더(Builder)	복잡한 객체의 생성 과정과 표현을 분리하는 패턴	복잡한 DTO 객체 생성, 문서 변환기, 롬복 @Builder
프로토타입(Prototype)	기존 객체를 복제하여 새로운 객체를 생성하는 패턴	복잡한 객체 캐싱, 설정 복제, 자바의 Cloneable 인터페이스

이러한 생성 패턴들은 객체 생성 메커니즘을 다루어 코드의 유연성을 높이고 재사용성을 증가시킵니다.

객체 지향 디자인 패턴의 종류와 개념

GoF(Gang of Four)에서 정의한 구조 패턴은 클래스와 객체의 구성에 관한 패턴입니다

패턴명	개념	실무 사용 예
어댑터(Adapter)	호환되지 않는 인터페이스를 함께 동작하도록 변환	레거시 시스템 통합, 외부 라이브러리 연동
브릿지(Bridge)	구현부와 추상층을 분리하여 독립적 변형 가능	디바이스 드라이버, 다양한 플랫폼 지원 GUI
컴포지트(Composite)	객체들의 관계를 트리 구조로 구성하는 패턴	파일 시스템, UI 컴포넌트 계층 구조
데코레이터(Decorator)	객체에 동적으로 책임을 추가하는 패턴	자바 I/O 스트림, 웹 요청 필터
퍼사드(Facade)	복잡한 서브시스템에 단순화된 인터페이스 제공	API 게이트웨이, 서비스 레이어
플라이웨이트(Flyweight)	공유를 통해 많은 객체를 효율적으로 지원	문자열 풀, 캐시 시스템
프록시(Proxy)	다른 객체에 대한 접근을 제어하는 대리자 제공	스프링 AOP, 지연 로딩, 캐싱

이러한 구조 패턴들은 클래스와 객체를 조합하여 더 큰 구조를 형성하면서 시스템의 유연성과 효율성을 높입니다.

객체 지향 디자인 패턴: 행위 패턴 - 1

패턴	설명	사용 예시
책임 연쇄(Chain of Responsibility)	요청을 처리할 수 있는 객체가 여러 개일 때 그 객체들을 연결하여 책임을 떠넘기는 패턴	예외 처리 체인, 이벤트 버블링, 서블릿 필터
커맨드(Command)	요청을 객체로 캡슐화하여 클라이언트와 수신자를 분리하는 패턴	트랜잭션 처리, 작업 큐, 메뉴 시스템, 명령어 실행 취소
인터프리터(Interpreter)	언어의 문법 표현을 정의하고 해당 언어로 된 문장을 해석하는 패턴	SQL 파서, 정규식 엔진, 수식 계산기
이터레이터(Iterator)	컬렉션 내부 구조를 노출하지 않고 요소에 순차적으로 접근하는 방법을 제공하는 패턴	자바 컬렉션 프레임워크, JDBC ResultSet, DOM NodeList
중재자(Mediator)	객체 간의 상호작용을 캡슐화하는 객체를 정의하는 패턴	채팅 시스템, 항공 교통 제어, UI 컨트롤러
메멘토(Memento)	객체의 내부 상태를 저장하고 이전 상태로 복원할 수 있게 하는 패턴	트랜잭션 롤백, 실행 취소 기능, 게임 상태 저장

이러한 행위 패턴들은 객체 간의 책임 할당과 알고리즘을 다루며, **효과적인 커뮤니케이션과 유연한 동작을 가능하게 합니다.**

객체 지향 디자인 패턴: 행위 패턴 - 2

패턴	설명	사용 예시
옵저버(Observer)	객체 상태 변경 시 의존 객체들에게 자동으로 통지하는 패턴	이벤트 처리 시스템, MVC 아키텍처, 데이터 바인딩
상태(State)	객체의 내부 상태에 따라 행위를 변경할 수 있게 하는 패턴	게임 캐릭터 상태, 주문 처리 흐름, 문서 승인 프로세스
전략(Strategy)	알고리즘을 정의하고 각각을 캡슐화하여 교체 가능하게 만드는 패턴	정렬 알고리즘, 결제 시스템, 압축 방식
템플릿 메소드(Template Method)	알고리즘의 구조를 정의하고 일부 단계를 하위 클래스에서 구현하게 하는 패턴	프레임워크 툴 메소드, JUnit 테스트, 데이터 처리 파이프라인
방문자(Visitor)	객체 구조의 요소들을 변경하지 않고 새로운 연산을 추가할 수 있게 하는 패턴	문서 트리 처리, AST 순회, 보고서 생성

Spring 프레임워크의 디자인 패턴

Spring 프레임워크는 다양한 디자인 패턴을 활용하여 유연하고 확장 가능한 애플리케이션 개발을 지원합니다.

디자인 패턴	설명
의존성 주입 (Dependency Injection)	객체 간의 의존 관계를 외부에서 주입하여 결합도를 낮추고 테스트 용이성을 높이는 패턴
팩토리 패턴 (Factory Pattern)	BeanFactory와 ApplicationContext를 통해 객체 생성 로직을 캡슐화하는 패턴
싱글톤 패턴 (Singleton Pattern)	기본적으로 모든 빈을 싱글톤으로 관리하여 리소스 효율성을 높이는 패턴
템플릿 메소드 패턴 (Template Method)	JdbcTemplate, RestTemplate 등에서 알고리즘의 골격을 정의하고 세부 구현을 하위 클래스에 위임하는 패턴
프록시 패턴 (Proxy Pattern)	AOP 구현에 활용되며, 원본 객체에 대한 접근을 제어하고 부가 기능을 추가하는 패턴
옵저버 패턴 (Observer Pattern)	이벤트 처리와 리스너 메커니즘에 사용되는 패턴으로 ApplicationEvent와 ApplicationListener 활용

스프링의 5대 핵심 모듈

모듈명	주요 기능	핵심 컴포넌트
Spring Core	IoC 컨테이너, DI 지원	BeanFactory, ApplicationContext
Spring AOP	관점 지향 프로그래밍 지원	Aspect, Advice, Pointcut
Spring MVC	웹 애플리케이션 개발 프레임워크	DispatcherServlet, Controller
Spring Data	데이터 액세스 기술 통합	Repository, JPA, JDBC
Spring Security	인증 및 권한 부여 프레임워크	Authentication, Authorization

각 모듈은 필요에 따라 독립적으로 사용하거나 조합하여 사용할 수 있다.

Spring Core란?

Spring Core는 프레임워크의 근간이 되는 핵심 모듈입니다. Spring Container를 통해 Bean의 생성, 소멸 등 전체 생명주기를 관리합니다.

Spring Core는 다음과 같은 핵심 가치를 제공한다.

핵심 가치	설명
IoC(제어의 역전)	객체의 생성과 생명주기 관리를 개발자가 아닌 프레임워크가 담당
DI(의존성 주입)	객체 간의 결합도를 낮추고 코드의 재사용성과 테스트 용이성 향상
컨테이너 기반	BeanFactory를 기반으로 하며, ApplicationContext가 이를 확장하여 더 풍부한 기능 제공

Spring Core 구성 계층

Spring Core는 다음과 같은 계층적 구조로 이루어져 있습니다.



이러한 계층적 구조를 통해 Spring Core는 애플리케이션의 유연성과 확장성을 제공합니다.

엔터프라이즈 환경에서의 Spring

대규모 트랜잭션 지원

수많은 동시 사용자와 복잡한 트랜잭션
을 안정적으로 처리

실무 활용 예: 금융권 결제 시스템, 대형
쇼핑몰 주문 처리, 항공권 예약 시스템 등
에서 다중 트랜잭션 관리에 활용

표준화된 확장 구조

엔터프라이즈 애플리케이션의 모든 계층
에 일관된 구조 적용 가능

실무 활용 예: 대기업 내부 시스템 통합,
마이크로서비스 아키텍처 구현, 레거시
시스템 점진적 현대화에 활용

모듈화와 통합

IoC를 통한 효과적인 모듈화로 서비스
간 통합 용이

실무 활용 예: ERP 시스템 구축, 서비스
간 API 연동, 다양한 솔루션(CRM, SCM
등) 통합에 활용

IoC: 제어의 역전이란?

기존 방식

```
public class OrderService {  
    private final PaymentProcessor paymentProcessor;  
  
    public OrderService() {  
        this.paymentProcessor = new  
CardPaymentProcessor();  
    }  
    public void processOrder(Order order) {  
        paymentProcessor.processPayment(order.getAmount());  
    }  
}
```

IoC 적용 후

```
@Service  
public class OrderService {  
    private final PaymentProcessor paymentProcessor;  
    @Autowired  
    public OrderService(PaymentProcessor  
paymentProcessor) {  
        this.paymentProcessor = paymentProcessor;  
    }  
    public void processOrder(Order order) {  
        paymentProcessor.processPayment(order.getAmount());  
    }  
}
```

IoC의 3가지 유형

인터페이스 기반

인터페이스를 통한 의존성 정의 및 구현체 교체가 용이한 구조

```
public interface PaymentService {  
    void processPayment(double amount);  
}  
  
@Component  
public class OrderService {  
    @Autowired  
    private PaymentService paymentService;  
}
```

컨스트럭터 기반

객체 생성 시점에 의존성 주입으로 필수 의존성 보장

```
@Service  
public class ProductService {  
    private final ProductRepository repo;  
  
    @Autowired  
    public  
    ProductService(ProductRepository repo) {  
        this.repo = repo;  
    }  
}
```

서비스 로케이터/팩토리

중앙화된 서비스 검색 메커니즘과 팩토리 패턴 활용

```
public class ServiceLocator {  
    private static Map services;  
    public static T getService(Class clazz) {  
        return (T)  
    services.get(clazz.getName());  
    }  
}
```

IoC 컨테이너란?

Bean 관리

객체 생성, 연결, 관리를 담당하는 핵심 엔진

의존성 처리

수많은 객체 간 의존관계를 자동으로 연결

생명주기 단계

생성, 초기화, 사용, 소멸의 전체 과정 제어

Bean의 정의와 역할

빈(Bean)이란?

스프링 컨테이너에 의해 생성되고 관리되는 자바 객체

POJO(Plain Old Java Object) 기반의 재사용 가능한 컴포넌트

빈 스코프

싱글톤: 컨테이너당 하나의 인스턴스 (기본값)

프로토타입: 요청마다 새 인스턴스

기타: request, session, application
스코프

주요 빈 종류

DTO, DAO, Service, Controller 등 다양한 계층의 객체가 빈으로 등록

Bean의 주요 클래스와 역할

클래스	설명
BeanFactory	스프링 컨테이너의 최상위 인터페이스로 빈 객체를 생성하고 관리하는 기본 기능 제공
ApplicationContext	BeanFactory를 확장한 인터페이스로 엔터프라이즈 애플리케이션에 필요한 추가 기능 제공
BeanDefinition	빈 객체에 대한 메타데이터를 저장하고 컨테이너에 빈 정의 등록을 위한 클래스
BeanPostProcessor	빈 초기화 전후에 사용자 정의 로직을 추가할 수 있는 인터페이스
FactoryBean	복잡한 초기화 로직이 필요한 객체를 생성하기 위한 패턴을 구현한 인터페이스
BeanWrapper	개별 빈 객체에 대한 접근과 프로퍼티 설정을 담당하는 인터페이스

Bean 등록 방식: XML vs 어노테이션

XML 기반 등록

spring-beans.xml 등의 파일에 선언적으로 정의

```
<bean id="userService"
      class="com.example.UserService">
    <property name="userDao"
              ref="userDao"/>
</bean>
```

어노테이션 기반 등록

클래스에 직접 어노테이션을 추가하여 자동 등록

```
@Service
public class UserService {
    @Autowired
    private UserDao userDao;
}
```

Bean Scopes(스코프 종류)

Singleton (기본)

컨테이너당 하나의 인스턴스만 생성
모든 요청이 동일한 객체 참조

Prototype

요청마다 새로운 인스턴스 생성
상태 보존이 필요한 경우 사용

웹 스코프

Request, Session, Application
웹 애플리케이션에서 요청/세션별 관리

Bean 생명주기 관리

생성

컨테이너가 Bean 인스턴스를 생성하고 프로퍼티 설정

초기화

설정 완료 후 @PostConstruct
메소드 호출

사용

애플리케이션에서 Bean 활용

소멸

컨테이너 종료 전
@PreDestroy 메소드 실행 후
Bean 제거

ApplicationContext 역할

역할	설명	주요 클래스/메소드
빈 관리	Bean 생성, 구성, 조회 및 생명주기 관리	getBean(), getBeanDefinitionNames(), getType()
자원 관리	메시지, 국제화(i18n) 지원, 리소스 로딩	MessageSource, ResourceLoader, getResource()
이벤트 처리	Bean 간 이벤트 발행/수신 기능 제공	ApplicationEventPublisher, publishEvent()
다양한 구현체	ClassPathXmlApplicationContext, AnnotationConfigApplicationContext 등	refresh(), close(), getEnvironment()

BeanFactory vs ApplicationContext

구분	BeanFactory	ApplicationContext
기능	기본적인 DI 기능만 제공	BeanFactory 기능 + 부가 기능
빈 로딩	지연 로딩(요청 시 로드)	즉시 로딩(시작 시 모두 로드)
부가 기능	없음	국제화, 이벤트 발행, 리소스 로딩 등
사용 사례	메모리 제약 환경	대부분의 스프링 애플리케이션

DI: 의존성 주입이란?

의존성 주입은 느슨한 결합을 통해 유연한 시스템 구조를 제공한다. 객체 간 연결을 자동화하는 자동 의존성 관리 기능을 지원하며, 컨테이너가 객체를 생성하고 주입하는 객체 조립 방식을 사용한다.

의존성 주입(Dependency Injection)은 객체가 필요로 하는 의존 객체를 직접 생성하지 않고 외부에서 제공받는 패턴으로 코드의 결합도가 낮아지고 테스트와 유지보수가 용이해진다.

```
public class UserService {  
    private UserRepository userRepository;  
    public UserService(UserRepository userRepository) {  
        this.userRepository = userRepository;  
    }  
    public User findUser(String id) {  
        return userRepository.findById(id);  
    }  
}
```

DI가 필요한 이유

결합도 최소화

컴포넌트 간 직접적인 의존성을 제거
해 유연성 증가

테스트 용이성

실제 객체 대신 Mock 객체로 단위 테
스트 간소화

재사용성 확보

다양한 환경에서 동일 컴포넌트 재사
용 가능

DI의 3가지 주요 방식

1. 생성자 주입

객체 생성 시점에 의존성 주입

불변성 보장 및 순환 참조 방지

2. 세터 주입

setter 메소드를 통한 의존성 주입

선택적인 의존성 처리에 유리

3. 필드 주입

필드에 직접 @Autowired 사용

코드는 간결하나 테스트가 어려움

생성자 주입의 장단점

장점

생성자 주입은 객체의 불변성을 보장하고 필수 의존성을 명확히 하며, 순환 참조를 컴파일 시점에 감지할 수 있다. 또한 의존성 주입이 명확하여 단위 테스트를 작성하기 용이하다.

```
public class UserService {  
    private final UserDao userDao;  
  
    public UserService(UserDao userDao) {  
        this.userDao = userDao;  
    }  
}
```

단점

필요한 의존성이 많을 경우 생성자의 매개변수가 늘어나 코드가 복잡해질 수 있으며, 선택적 의존성 주입을 위해 여러 생성자를 오버로딩해야 할 수도 있다.

주의사항

순환 참조 발생 시 애플리케이션 실행 불가 하며 설계 재검토 또는 다른 주입 방식 고려

Setter 주입의 특징

특징	설명
선택적 의존성	필수가 아닌 의존성 처리에 적합
런타임 변경 가능	의존성 재구성 가능
가독성	각 세터 메소드의 목적 명확

```
public class UserService {  
    private UserDao userDao;  
  
    @Autowired  
    public void setUserDao(UserDao userDao) {  
        this.userDao = userDao;  
    }  
}
```

필드 주입의 장점과 한계

특징	설명	코드 예시
장점	<ul style="list-style-type: none">- 코드 간결성- 빠른 개발 속도- 적은 코드량- 직관적인 코드 구조	<pre>public class UserService { @Autowired private UserDao userDao; // 별도의 생성자나 세터 불필요 }</pre>
한계	<ul style="list-style-type: none">- 단위 테스트 어려움- 의존성 숨김- 불변성 보장 불가- 순환 참조 감지 어려움	<pre>public class UserServiceTest { // 필드 주입된 의존성을 직접 설정할 방법이 없음 @Test public void testSaveUser() { UserService service = new UserService(); // userDao를 설정할 수 없음 } }</pre>

DI와 Reflection

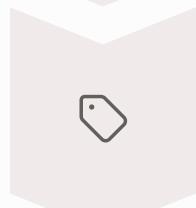
단계	과정	설명
1	메타데이터 분석	스프링은 클래스의 메타정보를 런타임에 분석
2	동적 객체 생성	리플렉션으로 클래스 인스턴스 동적 생성
3	동적 의존성 주입	필드 접근 제한자와 무관하게 의존성 주입 수행
4	성능 고려	리플렉션은 직접 호출보다 느리지만 초기화 시점에만 사용

Bean 설정: XML 방식



XML 파일 생성

spring-context.xml 또는 applicationContext.xml 작성



<bean> 태그 정의

id, class 속성으로 빈 식별 및 클래스 지정



의존성 지정

<property>나 <constructor-arg>로 의존성 주입



컨테이너 생성

ClassPathXmlApplicationContext로 컨텍스트 로드

XML 설정 예시 코드

XML 설정 파일

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ....>

    <bean id="userDao" class="com.example.UserDao"
/>

    <bean id="userService"
class="com.example.UserService">
        <property name="userDao" ref="userDao" />
    </bean>
</beans>
```

자바 코드에서 로드

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(
        "applicationContext.xml");

UserService userService =
    context.getBean("userService",
        UserService.class);
```

XML 방식은 설정과 코드의 물리적 분리가 명확합니다. 설정 변경 시
코드 재컴파일이 필요 없습니다.

Bean 설정: 어노테이션 방식

컴포넌트 스캔

@ComponentScan으로 자동 빈 등록 활성화
지정된 패키지 내 어노테이션 클래스 스캔

스테레오타입 어노테이션

@Component 계열 어노테이션으로 빈 등록
@Service, @Repository, @Controller 등

자동 연결

@Autowired로 의존 빈 자동 주입
@Qualifier로 동일 타입 빈 구분

어노테이션 기반 DI 이해

구분	설명	목적	예시
타입 기반 검색	@Autowired는 기본적으로 타입으로 빈 검색	적절한 타입의 빈을 자동으로 연결	@Autowired private UserService userService;
이름 기반 한정	@Qualifier로 동일 타입 중 특정 빈 지정	동일 타입의 여러 빈 중 선택	@Qualifier("mainUserDao") private UserDao userDao;
필수성 제어	required 속성으로 의존성 필수 여부 결정	선택적 의존성 주입 허용	@Autowired(required=false) private Logger logger;
생명주기 관리	Bean 생성부터 소멸까지 자동 관리	자원의 효율적 사용과 정리	@PostConstruct public void init() {...}

@Component의 세부 분류

어노테이션	설명
@Component	일반적인 스프링 관리 컴포넌트
@Controller	웹 요청을 처리하는 프레젠테이션 계층
@Service	비즈니스 로직을 포함하는 서비스 계층
@Repository	데이터 접근 계층 (DAO)

의미적으로 구분된 어노테이션은 아키텍처 계층별 역할을 명확히 하고, 특화된 기능을 제공합니다. 예를 들어 @Repository는 데이터 액세스 예외를 스프링 예외로 변환합니다.

@Configuration와 @Bean

@Configuration

빈 정의를 포함하는 클래스 지정

XML 설정 파일의 대안

```
@Configuration  
public class AppConfig {  
    // 빈 정의 메소드들...  
}
```

@Bean

메소드 레벨 어노테이션

메소드의 반환 객체를 빈으로 등록

```
@Bean  
public UserService userService() {  
    return new UserServiceImpl(userDao());  
}
```

```
@Bean  
public UserDao userDao() {  
    return new UserDaoImpl();  
}
```

@ComponentScan 동작 원리

스캔 시작점 지정	basePackages 속성으로 검색 패키지 지정
어노테이션 탐색	지정된 패키지 내 모든 @Component 계열 클래스 검색
필터링 적용	includeFilters/excludeFilters로 포함/제외 대상 정의
빈 등록 및 관리	찾은 모든 대상 클래스를 빈으로 등록하고 라이프사이클 관리

실전 코드: 어노테이션 Bean 등록

애플리케이션 설정

```
@Configuration  
@ComponentScan(basePackages = "com.example")  
public class AppConfig {  
    // 수동 빈 등록이 필요한 경우  
    @Bean  
    public DataSource dataSource() {  
        return new BasicDataSource();  
    }  
}
```

서비스 클래스

```
@Service  
public class ProductService {  
    private final ProductRepository repository;  
  
    @Autowired  
    public ProductService(  
        ProductRepository repository) {  
        this.repository = repository;  
    }  
  
    public List<Product> findAll() {  
        return repository.findAll();  
    }  
}
```

실전 코드: **@Autowired** 주입 예시

생성자 주입 (권장)

```
@Service
public class UserService {
    private final UserRepository userRepository;
    private final EmailService emailService;

    @Autowired // 생성자가 하나면 생략 가능
    public UserService(
        UserRepository userRepository,
        EmailService emailService) {
        this.userRepository = userRepository;
        this.emailService = emailService;
    }
}
```

세터 주입

```
@Service
public class OrderService {
    private PaymentService paymentService;

    @Autowired
    public void setPaymentService(
        PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

필드 주입

```
@Controller
public class ProductController {
    @Autowired
    private ProductService productService;

    // 필드 직접 주입 (테스트 어려움)
}
```

@Qualifier로 Bean 구분 주입

문제 상황

여러 구현체가 있는 경우 어떤 빈을 주입해야 할지 모호함

```
public interface PaymentService {  
    void processPayment(Order order);  
}
```

```
@Service  
public class CreditCardPaymentService  
    implements PaymentService { /*...*/ }
```

```
@Service  
public class PayPalPaymentService  
    implements PaymentService { /*...*/ }
```

@Qualifier 해결책

특정 빈의 식별자 지정

```
@Service("creditCardPayment")  
public class CreditCardPaymentService  
    implements PaymentService { /*...*/ }
```

```
@Service  
public class OrderService {  
    private final PaymentService paymentService;
```

```
@Autowired  
public OrderService(  
    @Qualifier("creditCardPayment")  
    PaymentService paymentService) {  
    this.paymentService = paymentService;  
}  
}
```

Bean Validation 어노테이션

JSR-380 표준

Bean Validation API는 객체 검증을 위한 자바 표준

주요 어노테이션

@Valid, @NotNull, @Size, @Min, @Max, @Email 등

사용 방법

엔티티 클래스의 필드에 어노테이션 적용 후 @Valid로 검증

```
public class User {  
    @NotNull(message = "ID는 필수입니다")  
    private Long id;  
  
    @NotBlank(message = "이름은 필수입니다")  
    @Size(min = 2, max = 30, message = "이름은 2-30자 사이여야 합니다")  
    private String name;  
  
    @Email(message = "유효한 이메일이 아닙니다")  
    private String email;  
}
```

Bean 스코프 실전 예제

싱글톤 vs 프로토타입

```
@Component  
@Scope("singleton") // 기본값이라 생략 가능  
public class SingletonBean {  
    private int counter = 0;  
  
    public int increment() {  
        return ++counter;  
    }  
}  
  
@Component  
@Scope("prototype")  
public class PrototypeBean {  
    private int counter = 0;  
  
    public int increment() {  
        return ++counter;  
    }  
}
```

테스트 코드

```
@Service  
public class ScopeTestService {  
    @Autowired  
    private ApplicationContext ctx;  
  
    public void testScopes() {  
        // 싱글톤: 항상 같은 인스턴스  
        SingletonBean s1 =  
            ctx.getBean(SingletonBean.class);  
        SingletonBean s2 =  
            ctx.getBean(SingletonBean.class);  
        System.out.println(s1 == s2); // true  
  
        // 프로토타입: 매번 새 인스턴스  
        PrototypeBean p1 =  
            ctx.getBean(PrototypeBean.class);  
        PrototypeBean p2 =  
            ctx.getBean(PrototypeBean.class);  
        System.out.println(p1 == p2); // false  
    }  
}
```

Spring 컨테이너 동작 흐름



라이프사이클 콜백: `@PostConstruct`

초기화 콜백

의존성 주입 완료 후 자동 호출

데이터베이스 연결, 캐시 로딩 등에 활용

JSR-250 표준 어노테이션

주의사항

- static 메소드에 사용 불가
- final 메소드에 사용 불가
- 반환값은 void여야 함
- 매개변수 없어야 함
- 예외 처리 필요

```
@Component
public class DataInitializer {
    @Autowired
    private UserRepository userRepository;

    @PostConstruct
    public void init() {
        // 의존성 주입 완료 후 실행
        System.out.println("초기화 로직 실행 중...");
        if (userRepository.count() == 0) {
            userRepository.save(new User("admin"));
        }
    }
}
```

스프링 4.3부터는 생성자 주입과 `@PostConstruct`를 함께 사용하는 것이 권장됩니다.

라이프사이클 콜백: `@PreDestroy`

소멸 콜백

빈 소멸 직전에 호출

리소스 해제, 연결 종료 등에 활용

활용 사례

- 데이터베이스 연결 해제
- 소켓 연결 종료
- 스레드 풀 종료
- 임시 파일 삭제
- 메시지 브로커 연결 해제
- 캐시 저장

싱글톤 빈에서는 중요하나, 프로토타입 빈에서는 호출되지 않습니다.

```
@Component
public class DatabaseConnection {
    private Connection connection;

    @PostConstruct
    public void connect() {
        // DB 연결 설정
        System.out.println("데이터베이스 연결");
    }

    @PreDestroy
    public void disconnect() {
        // 리소스 정리
        System.out.println("데이터베이스 연결 종료");
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                // 예외 처리
            }
        }
    }
}
```

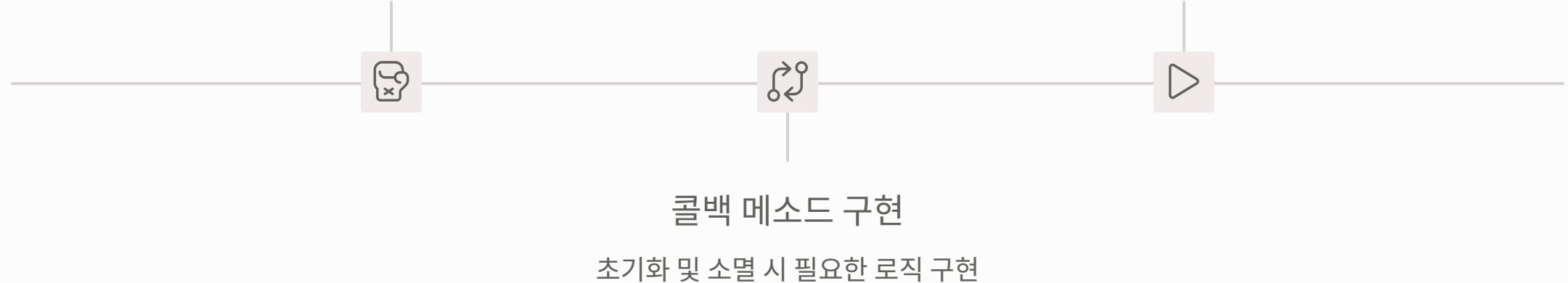
빈 생명주기 실습 예제

서비스 빈 생성

생명주기 콜백을 포함한 서비스 클래스 구현

애플리케이션 실행

콜백 호출 순서 확인 및 로그 관찰



```
@Service
public class BookService {
    private final BookRepository bookRepository;
    private Map<Long, Book> bookCache;

    @Autowired
    public BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
        System.out.println("생성자 호출: 의존성 주입");
    }

    @PostConstruct
    public void initialize() {
        System.out.println("@PostConstruct: 캐시 초기화");
        bookCache = new HashMap<>();
        bookRepository.findAll().forEach(book ->
            bookCache.put(book.getId(), book));
    }

    @PreDestroy
    public void shutdown() {
        System.out.println("@PreDestroy: 리소스 정리");
        bookCache.clear();
    }
}
```

프로퍼티 파일 관리와 DI

1

프로퍼티 파일 생성

application.properties 또는 application.yml
작성

2

프로퍼티 소스 설정

@PropertySource 어노테이션으로 파일 등록

3

@Value 주입

필드나 생성자 파라미터에 @Value로 값 주입

```
# application.properties
app.name=My Spring Application
app.version=1.0.0
db.url=jdbc:mysql://localhost:3306/mydb
db.username=user
db.password=secret
```

```
// Java 코드
@Service
public class ConfigurableService {
    @Value("${app.name}")
    private String appName;

    @Value("${db.url}")
    private String dbUrl;
}
```

환경별 프로필(@Profile)

프로필 정의

```
@Configuration  
@Profile("dev")  
public class DevConfig {  
    @Bean  
    public DataSource dataSource() {  
        return new EmbeddedDatabaseBuilder()  
            .setType(EmbeddedDatabaseType.H2)  
            .build();  
    }  
}
```

```
@Configuration  
@Profile("prod")  
public class ProdConfig {  
    @Bean  
    public DataSource dataSource() {  
        BasicDataSource dataSource = new  
        BasicDataSource();  
        dataSource.setUrl("jdbc:mysql://prod-server/db");  
        return dataSource;  
    }  
}
```

컴포넌트 레벨 적용

```
@Service  
@Profile("dev")  
public class DevService implements MyService {  
    // 개발 환경용 구현  
}  
  
@Service  
@Profile("prod")  
public class ProdService implements MyService {  
    // 운영 환경용 구현  
}
```

프로필 활성화

spring.profiles.active 속성으로 활성화
-Dspring.profiles.active=dev,test

FactoryBean 개념 및 적용

FactoryBean이란?

- 복잡한 객체 생성 로직을 캡슐화하는 패턴
- 빈 생성 과정을 커스터마이징하는 인터페이스
- 프록시, 연결 풀 등 복잡한 객체 생성에 활용

주요 메소드

- getObjectType(): 생성할 객체 타입 반환
- getObjectType(): 생성 객체 타입 반환
- isSingleton(): 싱글톤 여부 결정

사용 사례

- JNDI 객체 조회
- 프록시 객체 생성
- 동적 빈 구성
- 조건부 빈 생성

```
public class ConnectionFactoryBean implements  
FactoryBean<Connection> {  
    private String url;  
    private String username;  
    private String password;  
  
    @Override  
    public Connection getObject() throws Exception {  
        return DriverManager.getConnection(url,  
username, password);  
    }  
  
    @Override  
    public Class<?> getObjectType() {  
        return Connection.class;  
    }  
  
    @Override  
    public boolean isSingleton() {  
        return false; // 매번 새로운 연결 생성  
    }  
}
```

메시지 소스와 국제화 빈 연동

메시지 소스는 다국어 지원을 위한 Spring의 핵심 기능으로, 애플리케이션의 국제화(i18n)를 쉽게 구현할 수 있게 해줍니다.

MessageSource 빈 설정

```
@Bean  
public MessageSource messageSource() {  
    ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();  
    messageSource.setBasename("messages");  
    messageSource.setDefaultEncoding("UTF-8");  
    return messageSource;  
}
```

메시지 프로퍼티 파일 구성

messages_ko.properties

```
greeting=안녕하세요  
welcome.message=환영합니다, {0}님!
```

messages_en.properties

```
greeting=Hello  
welcome.message=Welcome, {0}!
```

메시지 소스 사용 예제

```
@Component
public class MessageComponent {
    private final MessageSource messageSource;

    @Autowired
    public MessageComponent(MessageSource messageSource) {
        this.messageSource = messageSource;
    }

    public void displayMessages(Locale locale) {
        String greeting = messageSource.getMessage("greeting", null, locale);
        String welcome = messageSource.getMessage("welcome.message",
            new Object[]{"Spring"}, locale);
        System.out.println(greeting);
        System.out.println(welcome);
    }
}
```

국제화 메시지를 JSP나 Thymeleaf에서 사용

JSP

```
<spring:message code="greeting"/>
<spring:message code="welcome.message" arguments="${userName}"/>
```

Thymeleaf

```
<p th:text="#{greeting}"></p>
<p th:text="#{welcome.message(${userName})}"></p>
```

DI와 테스트 코드 전략

단위 테스트

```
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {
    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Test
    public void findByUsername() {
        // given
        User mockUser = new User("testUser");
        when(userRepository.findByUsername("testUser"))
            .thenReturn(Optional.of(mockUser));

        // when
        User found = userService.findByUsername("testUser");

        // then
        assertNotNull(found);
        assertEquals("testUser", found.getUsername());
    }
}
```

스프링에서의 통합 테스트 전략

통합 테스트는 애플리케이션의 여러 컴포넌트가 함께 올바르게 동작하는지 검증합니다.

```
@SpringBootTest
public class UserServiceIntegrationTest {
    @Autowired
    private UserService userService;

    @Autowired
    private UserRepository userRepository;

    @BeforeEach
    public void setup() {
        userRepository.deleteAll();
        userRepository.save(new User("testUser"));
    }

    @Test
    public void findByUsername() {
        User found = userService.findByUsername("testUser");
        assertNotNull(found);
        assertEquals("testUser", found.getUsername());
    }
}
```

위 코드는 실제 스프링 컨테이너를 구동하여 의존성 주입이 제대로 이루어지는지 확인하고, 컴포넌트 간의 상호작용을 테스트합니다.

@SpringBootTest 어노테이션은 전체 애플리케이션 컨텍스트를 로드하여 실제 환경과 유사한 테스트를 가능하게 합니다.

스프링에서의 싱글톤 보장원리

- **싱글톤 레지스트리:** 빈 이름과 인스턴스를 매핑하는 내부 저장소
- **빈 조회 과정:** 빈 요청 시 레지스트리 확인 후 존재하면 재사용
- **동시성 제어:** ConcurrentHashMap 활용한 스레드 안전성 보장
- **성능 최적화:** 메모리 효율성 및 객체 생성 비용 절감

스프링은 기본적으로 모든 빈을 싱글톤으로 관리합니다. 이는 전통적인 싱글톤 패턴과 달리 스프링 컨테이너가 직접 인스턴스 생성과 관리를 담당하므로 테스트하기 쉽고 전역 상태 오염 위험이 적습니다.

Custom BeanPostProcessor

1

인터페이스 구현

BeanPostProcessor 인터페이스 구현

2

사전 처리

postProcessBeforeInitialization 구현

3

사후 처리

postProcessAfterInitialization 구현

4

빈 등록

프로세서를 빈으로 등록하여 활성화

```
@Component  
public class LoggingBeanPostProcessor implements BeanPostProcessor {  
    @Override  
    public Object postProcessBeforeInitialization(Object bean, String beanName)  
        throws BeansException {  
        if (bean instanceof Service) {  
            System.out.println("Before initialization: " + beanName);  
        }  
        return bean;  
    }  
}
```

```
@Override  
public Object postProcessAfterInitialization(Object bean, String beanName)  
    throws BeansException {  
    if (bean instanceof Repository) {  
        System.out.println("After initialization: " + beanName);  
    }  
    return bean;  
}
```

스프링 애플리케이션 초기화 과정



스프링 Core 고급 활용 팁

ObjectProvider 활용: 자연 로딩 및 선택적 의존성 처리

```
@Autowired  
ObjectProvider<UserService> provider;  
  
void doSomething() {  
    provider.getIfAvailable()?.process();  
}
```

순환 참조 해결: 세터 주입 또는 @Lazy 어노테이션 활용

```
@Service  
public class ServiceA {  
    private ServiceB serviceB;  
  
    @Autowired  
    public ServiceA(@Lazy ServiceB serviceB) {  
        this.serviceB = serviceB;  
    }  
}
```

커스텀 스코프: Scope 인터페이스 구현으로 특수 빈 생명주기 관리

```
@Component  
@Scope(value = "custom",  
proxyMode = ScopedProxyMode.TARGET_CLASS)  
public class CustomScopedBean {}
```

실무 프로젝트 예시: Bean 설정 비교

XML 설정 프로젝트

구조:

- src/main/resources/applicationContext.xml
- src/main/resources/infrastructure.xml
- src/main/resources/security.xml

장점:

- 중앙집중식 설정 관리
- 코드 변경 없이 설정 변경 가능
- 레거시 시스템 통합 용이

어노테이션 설정 프로젝트

구조:

- config/AppConfig.java
- config/DatabaseConfig.java
- service/UserService.java (@Service)
- repository/UserRepository.java (@Repository)

장점:

- 타입 안전성
- 리팩토링 용이
- 코드와 설정 통합으로 명확성
- 개발 생산성 향상

결합도 최소화와 재사용성 확보



결합도 감소

IoC/DI 적용 후 코드 결합도 감소율

코드 재사용

컴포넌트 재사용 증가율

개발 시간 단축

신규 기능 개발 시간 감소율

스프링의 IoC/DI를 적용하면 객체 간 결합도가 크게 줄어들어 코드의 유연성과 재사용성이 향상됩니다. 인터페이스 기반 설계와 컨테이너의 객체 관리를 통해 대규모 프로젝트에서도 모듈성이 유지되어 개발과 유지보수 효율이 증가합니다.

참고 자료



핵심 개념 요약

IoC, DI, Bean 생명주기, 스코프, 어노테이션 기반 설정의 중
요성 이해



추천 학습 자료

Spring 공식 문서, Baeldung 튜토리얼, 토비의 스프링 도서



샘플 코드

GitHub 예제 프로젝트, Spring Guides, Spring Boot 스타
터



커뮤니티

Stack Overflow, Spring 포럼, 한국 스프링 사용자 모임

주요 온라인 리소스

리소스 명	주소	특징
Spring 공식 웹사이트	spring.io	공식 문서, 가이드, 튜토리얼
Baeldung	baeldung.com	고품질 Spring 튜토리얼
Spring 블로그	spring.io/blog	최신 업데이트 및 릴리스 정보
GitHub 저장소	github.com/spring-projects	소스 코드 및 이슈 트래킹
Stack Overflow	stackoverflow.com/questions/tagg ed/spring	질문 및 답변 커뮤니티