3. 핵심 아키텍처 이해

IoC (제어의 역전) / DI (의존성 주입)

1. IoC (Inversion of Control) – 제어의 역전

정의

전통적인 프로그래밍에서는 **객체 생성, 의존성 관리, 실행 흐름**을 개발자가 직접 통제한다.
하지만 IoC는 이 제어권을 **프레임워크에게 넘기는 것**을 말한다.

예시 (비교)

전통적인 방식

```
UserService userService = new UserService(new UserRepository());
```

→ 개발자가 객체를 **직접 생성하고 연결**

IoC 방식 (Spring)

```
@Component
public class UserService {
    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

→ **Spring이 객체를 생성하고 주입**
→ 개발자는 로직만 신경쓰면 됨

---

2. DI (Dependency Injection) – 의존성 주입

정의

DI는 **객체가 의존하는 객체(Dependency)를 외부로부터 주입받는 구조**를 말한다.
즉, 클래스가 다른 클래스에 의존할 때, 이를 직접 생성하지 않고 **주입받도록 한다.**

종류

DI 유형	설명	예시
생성자 주입	생성자 파라미터로 의존 객체를 전달	권장 방식
필드 주입	`@Autowired`로 필드에 직접 주입	테스트 어려움, 비권장

DI 유형	설명	예시
Setter 주입	세터 메서드를 통해 주입	선택적 의존성에 적합

생성자 주입 예시 (권장 방식)

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

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

Spring은 `PaymentService` 객체를 찾고 자동으로 생성자에 주입한다.

---

3. Spring의 IoC Container

정의

IoC Container는 Spring에서 Bean 객체를 생성하고, 관리하고, 필요한 곳에 주입해주는 **중앙 관리자**다.
대표적인 컨테이너는 다음과 같다.

- `ApplicationContext`: 기본 컨테이너
- `AnnotationConfigApplicationContext`, `WebApplicationContext`, `GenericApplicationContext`

Bean 등록 예시

```
@Configuration
public class AppConfig {
    @Bean
    public MemberService memberService() {
        return new MemberService(memberRepository());
    }

    @Bean
    public MemberRepository memberRepository() {
        return new MemoryMemberRepository();
    }
}
```

---

4. Spring Boot에서는 어떻게 자동화되는가?

Spring Boot는 아래 애노테이션을 기반으로 자동으로 IoC/DI 처리를 해준다.

애노테이션	설명
`@Component`	자동 등록 대상 클래스
`@Service`, `@Repository`, `@Controller`	역할별 컴포넌트의 명확한 구분
`@Autowired`	의존성 주입 (생성자, 필드, 세터에 적용 가능)
`@Configuration`	Bean 정의를 포함하는 설정 클래스
`@Bean`	수동 Bean 등록 메서드

---

5. DI와 IoC의 관계

항목	설명
IoC	객체의 생성 및 관리 책임을 프레임워크에 넘김
DI	그 객체 간의 의존성을 주입받는 방식

> IoC는 **전체 개념**이고, DI는 IoC를 **구현하는 기법 중 하나**이다.
> 즉, DI는 IoC의 실현 방식이다.

---

6. IoC/DI의 장점

- 객체 간 결합도 감소 (Low Coupling)
- 유연한 구조 (OCP 원칙 적용 가능)
- 테스트 용이 (Mock 주입이 쉬움)
- 재사용성 향상
- 유지보수 및 확장성 개선

---

7. 오해 방지

- DI는 **new 연산자를 쓰지 말라는 뜻이 아님**
  → 다만 Spring 관리 범위 내에서는 객체 생성을 프레임워크에 위임하는 것이 좋음
- `@Autowired`는 항상 자동 주입되는 것은 아님
  → 주입 대상이 명확히 하나일 경우에만 자동 주입된다 (아닐 경우 오류 발생 또는 `@Qualifier` 필요)

결론

Spring Boot는 IoC와 DI를 기반으로 동작하며, 이로 인해 객체 생성과 관리, 연결을 자동화할 수 있다.
덕분에 애플리케이션은 **구성보다 비즈니스 로직에 집중된 구조**로 개발할 수 있으며, **확장과 유지보수가 쉬운 구조**로 발전할 수 있다.

Bean 생성과 생명주기

Spring Framework와 Spring Boot에서 **Bean**은 Spring IoC 컨테이너가 관리하는 **객체(인스턴스)**를 의미한다.
모든 의존성 주입(DI), 제어의 역전(IoC), 설정 자동화 등의 핵심은 **Bean의 생성과 생명주기 관리**에 기반을 두고 있다.

이 개념을 제대로 이해하면 **Spring 내부 동작 원리**, **AOP**, **초기화/종료 작업 처리**, **성능 최적화**, **테스트 주입** 등의 기반 지식이 완성된다.

---

1. Bean이란?

Spring Container(ApplicationContext)에 의해 생성되고 관리되는 **객체(인스턴스)**를 의미한다.

다음과 같은 방법으로 등록된다.

- `@Component`, `@Service`, `@Repository`, `@Controller` → 자동 등록
- `@Bean` → 수동 등록
- XML 설정 (`<bean>`) → Spring Boot에서는 거의 사용하지 않음

---

2. Bean 등록 방식

등록 방법	설명	예시
자동 등록	`@Component` 및 유사 애노테이션 기반	`@Component`, `@Service` 등
수동 등록	`@Configuration` 클래스 + `@Bean` 메서드	`@Bean public Object a() {}`
XML 등록	`<bean>` 태그 사용	Spring Boot에서는 권장되지 않음

---

3. Bean의 생명주기 개요

Spring Bean은 다음과 같은 순서로 생명주기를 가진다:

```
1. 객체 생성 (new)
2. 의존성 주입 (DI)
3. 초기화 콜백 (InitializingBean, @PostConstruct)
4. 애플리케이션에서 사용됨
5. 종료 콜백 (DisposableBean, @PreDestroy)
```

---

4. 각 단계 상세

1. 생성

- Spring이 `new`를 호출하거나 `@Configuration` 클래스의 `@Bean` 메서드를 실행하여 객체 생성

2. 의존성 주입 (Dependency Injection)

- `@Autowired`, 생성자 주입 등을 통해 필요한 의존 객체를 주입

3. 초기화

초기화 시점에 특정 작업(로그 출력, 연결 등)을 수행하고 싶다면 다음 방법 사용

방법	설명
`@PostConstruct`	초기화 직전 실행
`InitializingBean.afterPropertiesSet()`	인터페이스 방식
`@Bean(initMethod = "init")`	수동 Bean 등록 시 지정 가능

예:

```
@PostConstruct
public void init() {
    System.out.println("초기화 작업 실행");
}
```

---

4. 사용

- 컨테이너에서 주입된 Bean은 애플리케이션 전역에서 사용된다.

---

5. 종료

Spring 애플리케이션이 종료될 때 자원 해제, 로그 저장, DB 연결 종료 등 정리 작업 필요 시:

방법	설명
`@PreDestroy`	종료 직전에 자동 호출
`DisposableBean.destroy()`	인터페이스 방식
`@Bean(destroyMethod = "cleanup")`	수동 Bean 설정 방식

예:

```java
@PreDestroy
public void cleanup() {
    System.out.println("종료 작업 실행");
}
```

5. Bean 생명주기 콜백 정리

콜백	시점	방식
`@PostConstruct`	생성 및 의존성 주입 후	애노테이션
`InitializingBean.afterPropertiesSet()`	생성 직후	인터페이스
`@PreDestroy`	종료 전	애노테이션
`DisposableBean.destroy()`	종료 전	인터페이스
`@Bean(initMethod, destroyMethod)`	수동 Bean 설정 시	메서드명 명시

6. 스코프에 따른 생명주기 차이

스코프	특징	생명주기
`singleton`	기본값. 컨테이너당 하나만 생성됨	컨테이너 시작 ~ 종료까지
`prototype`	요청할 때마다 새로 생성	요청 시 생성, 종료는 사용자가 관리
`request`, `session`	웹 환경 전용	요청 또는 세션 범위 동안 유지

예:

```java
@Component
@Scope("prototype")
public class PrototypeBean {
    @PostConstruct
    public void init() {
        System.out.println("Prototype 생성됨");
    }
}
```

7. Bean 생명주기 예시

```java
@Component
public class MyBean implements InitializingBean, DisposableBean {

    @PostConstruct
    public void postConstruct() {
```

```
        System.out.println("PostConstruct 호출됨");
    }

    @Override
    public void afterPropertiesSet() {
        System.out.println("InitializingBean.afterPropertiesSet 호출됨");
    }

    @PreDestroy
    public void preDestroy() {
        System.out.println("PreDestroy 호출됨");
    }

    @Override
    public void destroy() {
        System.out.println("DisposableBean.destroy 호출됨");
    }
}
```

→ 애플리케이션 실행 및 종료 시, 각 메서드가 호출되는 것을 로그에서 확인 가능

결론

Spring의 Bean 생명주기는 프레임워크가 객체를 어떻게 **생성, 주입, 초기화, 종료**하는지를 설명하는 핵심 메커니즘이다.
이를 잘 이해하면 **자원 관리**, **초기 설정**, **정리 로직**, **스코프 분리**, **테스트 주입** 등을 더욱 효과적으로 설계할 수 있다.

Component Scanning

Component Scanning은 Spring Framework와 Spring Boot에서 클래스 경로(Classpath)를 자동으로 검색하여,
특정 애노테이션이 붙은 클래스를 **자동으로 Bean으로 등록**해주는 기능이다.
이 덕분에 `@Bean`을 매번 수동으로 작성하지 않아도 되고, **애플리케이션 구조 설계가 더 모듈화되고 효율적으로** 된다.

---

1. 개요

Spring은 다음 애노테이션이 붙은 클래스들을 스캔하여 IoC 컨테이너에 자동 등록한다.

애노테이션	역할	설명
`@Component`	일반적인 Bean	모든 종류의 Bean 등록 가능
`@Service`	서비스 계층	내부 로직 처리 클래스
`@Repository`	DAO 계층	DB 연동, 예외 변환 자동 처리
`@Controller`	웹 계층	요청 처리용 Controller
`@RestController`	`@Controller + @ResponseBody`	REST API 전용 Controller

※ 위 모든 애노테이션은 사실상 `@Component`의 특수화된 형태다.

---

2. Component Scanning의 작동 방식

Spring Boot 애플리케이션의 진입점에 위치한 `@SpringBootApplication`은 내부적으로 다음과 같이 구성되어 있다.

```
@SpringBootApplication
// 내부적으로 아래 애노테이션 포함
@ComponentScan(basePackages = "com.example")
```

즉, `@SpringBootApplication`이 선언된 패키지를 기준으로 **하위 모든 패키지의 `@Component` 계열 클래스를 자동 스캔**한다.

---

3. 스캔 대상 범위

- 기본적으로는 `@SpringBootApplication`이 선언된 클래스의 **패키지 기준으로 하위 패키지를 스캔**
- 다른 경로도 포함시키고 싶다면 `@ComponentScan`을 명시적으로 사용

```
@ComponentScan(basePackages = {
    "com.example.service",
    "com.example.repository"
})
```

---

4. 자동 등록 vs 수동 등록

등록 방식	방법	장점	단점
자동 등록	`@Component`, `@Service` 등	반복 작업 감소, 유지보수 쉬움	관리 범위 추적 어려움
수동 등록	`@Configuration + @Bean`	제어력, 이름 지정 명확	반복적이고 장황함

Spring Boot에서는 **자동 등록**을 기본으로 사용하고,
특수한 경우에만 수동 등록을 사용한다.

---

5. Bean 이름 규칙

자동 등록된 Bean은 기본적으로 **클래스 이름의 첫 글자만 소문자로 변환**한 이름을 사용한다.

```
@Component
public class MyService {
}
```

→ Bean 이름: `"myService"`

명시적으로 이름 지정도 가능:

```java
@Component("customService")
public class MyService {
}
```

---

6. 중복된 Bean 이름 충돌 방지

다음 상황에서는 에러 발생:

```java
@Component
public class MyComponent {}

@Component
public class MyComponent {} // 같은 이름, 에러 발생
```

해결 방법:

- Bean 이름을 명시적으로 다르게 설정
- `@Primary` 또는 `@Qualifier`를 사용하여 어떤 Bean을 주입할지 지정

---

7. 예시: 구조 예시

```
com.example
 ㄴ DemoApplication.java (@SpringBootApplication)
 ㄴ controller/
     ㄴ HomeController.java (@RestController)
 ㄴ service/
     ㄴ MemberService.java (@Service)
 ㄴ repository/
     ㄴ MemberRepository.java (@Repository)
```

→ `DemoApplication.java`가 `com.example`에 위치하므로, 모든 하위 패키지가 자동 스캔 대상이 된다.

---

8. 조건부 컴포넌트 등록

조건에 따라 Bean을 등록하려면 아래와 같은 애노테이션을 함께 사용한다.

애노테이션	역할
`@ConditionalOnProperty`	특정 속성이 true일 때만 등록
`@ConditionalOnClass`	특정 클래스가 존재할 때만 등록
`@Profile`	특정 프로파일 활성 시 등록

---

9. 주의할 점

- 자동 등록을 사용하는 경우 **패키지 구조가 명확**해야 한다.
- 스캔 대상 외부에 있는 컴포넌트는 **등록되지 않음** → 직접 `@ComponentScan`으로 지정 필요
- 빈 이름 충돌 시 오류 발생

결론

Component Scanning은 Spring이 DI와 IoC를 자동으로 구현하게 해주는 핵심 기법이다.
패키지 구조를 체계적으로 유지하고, 역할별 애노테이션을 잘 사용하면 코드의 **구조화, 유지보수성, 확장성**이 크게 향상된다.

AOP (관점 지향 프로그래밍)

AOP (Aspect-Oriented Programming, 관점 지향 프로그래밍)은
Spring 프레임워크의 핵심 개념 중 하나로, **공통 기능(횡단 관심사, cross-cutting concern)**을
핵심 비즈니스 로직과 **분리하여 재사용성, 모듈화, 유지보수성을 향상**시키는 프로그래밍 기법이다.

---

1. AOP란 무엇인가?

AOP는 프로그램을 핵심 비즈니스 로직(예: 주문 처리)과 **공통 처리 로직(예: 로깅, 보안, 트랜잭션, 인증)**으로 나누어
공통 로직을 **별도의 모듈(Aspect)**로 추출하고, **필요한 시점에 끼워넣는 방식**이다.

---

2. 용어 정리 (핵심 개념)

용어	설명
Aspect	공통 관심사를 담은 모듈 (예: 로깅, 보안)
Join Point	Aspect가 적용될 수 있는 지점 (예: 메서드 호출 시점)
Advice	실제 수행될 코드 (before, after, around 등)
Pointcut	Advice가 적용될 Join Point의 조건 (표현식)
Weaving	Aspect와 실제 코드(핵심 비즈니스)를 연결하는 과정
Proxy	스프링이 AOP를 적용하기 위해 생성하는 대리 객체 (JDK 동적 프록시 or CGLIB 기반)

---

3. 대표적인 AOP 적용 대상

- 로깅 (메서드 진입/종료, 실행 시간 등)
- 트랜잭션 처리
- 보안 검사 (권한 확인)
- 입력 검증
- 예외 처리 / 공통 에러 로깅

- 캐싱 / 리포트 생성

---

4. Spring AOP 기반 구조

Spring은 기본적으로 **프록시 기반의 AOP**를 사용한다.
→ 내부적으로 **JDK 동적 프록시** 또는 **CGLIB 프록시**를 생성하여
지정된 메서드에 Aspect를 삽입한다.

---

5. Spring Boot에서 AOP 사용 준비

5.1 의존성 추가

Gradle:

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

Maven:

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

---

6. 실전 예시

6.1 Aspect 클래스 정의

```
@Aspect
@Component
public class LoggingAspect {

    @Pointcut("execution(* com.example.service..*(..))")
    public void serviceMethods() {}

    @Before("serviceMethods()")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("[Before] 실행 메서드: " +
joinPoint.getSignature().toShortString());
    }

    @AfterReturning(pointcut = "serviceMethods()", returning = "result")
    public void logAfter(Object result) {
        System.out.println("[AfterReturning] 반환값: " + result);
    }

    @AfterThrowing(pointcut = "serviceMethods()", throwing = "e")
```

```
    public void logException(Exception e) {
        System.out.println("[AfterThrowing] 예외 발생: " + e.getMessage());
    }
}
```

6.2 적용 대상 클래스

```
@Service
public class MemberService {
    public String join(String name) {
        System.out.println("회원 가입 처리 중: " + name);
        return "Welcome, " + name;
    }
}
```

→ `MemberService.join()` 호출 시 `@Before`, `@AfterReturning` 등이 자동 실행됨

---

7. Pointcut 표현식 예시

표현식	설명
`execution(* *..*Service.*(..))`	모든 Service 클래스의 모든 메서드
`execution(public * *(..))`	public 접근자의 모든 메서드
`execution(* com.example..*(String))`	인자가 String인 메서드
`within(com.example.repository..*)`	특정 패키지 내부 모든 클래스
`@annotation(com.example.Loggable)`	특정 애노테이션이 붙은 메서드

---

8. Advice 유형

Advice	설명
`@Before`	메서드 실행 전
`@AfterReturning`	정상 반환 후
`@AfterThrowing`	예외 발생 시
`@After`	성공/실패 무관하게 실행 후
`@Around`	실행 전후 전체 제어 (가장 강력함)

```java
@Around("execution(* com.example..*(..))")
public Object measureTime(ProceedingJoinPoint joinPoint) throws Throwable {
    long start = System.currentTimeMillis();
    Object result = joinPoint.proceed();  // 원래 메서드 실행
    long end = System.currentTimeMillis();
    System.out.println("실행 시간: " + (end - start) + "ms");
    return result;
}
```

---

9. 주의사항

- **Spring AOP는 메서드 단위의 프록시 기반 AOP**이다. 클래스 내부 메서드 호출은 적용되지 않음 (self-invocation 문제)
- `@Transactional`과 `@Async`도 내부적으로 AOP로 동작한다
- JDK 프록시는 인터페이스 기반, CGLIB 프록시는 클래스 상속 기반

결론

Spring의 AOP는 비즈니스 코드에 **공통 로직을 삽입하지 않고도 동작을 확장**할 수 있게 한다.
서비스가 커질수록 핵심 로직과 부가 로직을 분리하는 것이 중요하며, AOP는 이 역할에 최적화된 기술이다.

Proxy 기반 동작 원리

Spring의 **AOP (Aspect-Oriented Programming)**와 **DI (Dependency Injection)**는 모두 **프록시(Proxy)** 기반으로 동작한다.
Spring은 우리가 등록한 객체 대신에 **프록시 객체를 생성하여 빈으로 주입**하고,
그 프록시 객체가 **핵심 로직 전후로 부가기능을 삽입**하는 방식으로 동작한다.

---

1. 프록시란 무엇인가?

프록시(Proxy)란, **실제 객체에 대한 접근을 제어하는 대리 객체**다.
클라이언트는 진짜 객체가 아닌 프록시를 통해 요청을 보내며, 프록시는 다음을 수행할 수 있다:

- 메서드 호출 전/후 로직 삽입 (로깅, 트랜잭션 등)
- 요청 차단, 인증 처리
- 결과 조작
- 진짜 객체에게 위임

---

2. Spring에서의 Proxy 용도

Spring은 다음과 같은 경우에 프록시를 사용한다:

기능	설명
AOP	`@Transactional`, `@Async`, 커스텀 `@Around` 등

기능	설명
DI	의존성 주입 대상이 인터페이스일 경우 동적 프록시 생성
Lazy 초기화	`@Lazy`, `ObjectProvider` 등의 객체 지연 생성
WebClient 등 비동기 호출	내부적으로 프록시 구조 사용

---

3. Spring Proxy 종류

프록시 방식	조건	생성 방식	특징
JDK 동적 프록시	대상이 인터페이스 구현체일 때	`java.lang.reflect.Proxy`	인터페이스 기반
CGLIB 프록시	대상이 클래스일 때	Bytecode 조작하여 서브클래스 생성	클래스 기반, final 클래스 불가
Javassist	Spring에서 거의 사용 안 함	Bytecode 변환 기반	고급 Bytecode 수준

Spring Boot는 기본적으로 **JDK 동적 프록시**를 사용하며, 필요 시 자동으로 CGLIB을 사용한다.

---

4. 프록시 생성 흐름 (Spring AOP 예)

```
클래스: MemberService

  ↓ AOP 적용

프록시 객체: MemberService$$EnhancerBySpringCGLIB

  ↓ 메서드 호출

[1] @Before → 로깅
[2] 실제 메서드 실행
[3] @AfterReturning → 로깅
```

---

5. 코드 예시: 프록시 직접 구현 (JDK 기반)

```
interface MemberService {
    void join(String name);
}

class MemberServiceImpl implements MemberService {
    public void join(String name) {
        System.out.println("회원 가입 처리: " + name);
    }
}
```

```
class LoggingHandler implements InvocationHandler {
    private final Object target;

    public LoggingHandler(Object target) {
        this.target = target;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("Before: " + method.getName());
        Object result = method.invoke(target, args);
        System.out.println("After: " + method.getName());
        return result;
    }
}
```

```
MemberService target = new MemberServiceImpl();
MemberService proxy = (MemberService) Proxy.newProxyInstance(
    MemberService.class.getClassLoader(),
    new Class[]{MemberService.class},
    new LoggingHandler(target)
);
proxy.join("kim");
```

실행 결과:

```
Before: join
회원 가입 처리: kim
After: join
```

---

6. Spring의 프록시 적용 예시

```
@Service
@Transactional
public class OrderService {
    public void createOrder() {
        // 실제 비즈니스 로직
    }
}
```

- Spring은 `OrderService`에 프록시를 생성함
- `createOrder()`가 호출될 때 프록시가 트랜잭션을 시작하고, 실제 메서드를 실행한 뒤 커밋/롤백

---

7. 주의: 자기 자신 호출 (self-invocation)

프록시 기반의 AOP에서는 **자기 자신의 내부 메서드 호출에는 AOP가 적용되지 않는다**.

예:

```
public class MyService {
    @Transactional
    public void methodA() {
        methodB(); // 이 호출은 프록시를 거치지 않음 → 트랜잭션 미적용
    }

    @Transactional
    public void methodB() { }
}
```

해결 방법:

- 내부 호출 분리: 메서드B를 다른 Bean으로 분리
- AOP 설정 시 AspectJ 사용 (비권장)

---

8. 프록시 Bean 확인

다음과 같은 방법으로 현재 객체가 프록시인지 확인할 수 있다.

```
System.out.println("클래스: " + target.getClass());
AopProxyUtils.ultimateTargetClass(target); // 실제 클래스 확인
AopUtils.isAopProxy(target); // 프록시 여부
```

---

9. 결론

Spring은 AOP나 DI 같은 기능을 위해 **대상 객체를 프록시로 감싸서 Bean으로 등록**한다.
이 프록시는 메서드 실행 전후에 **부가기능을 삽입하거나 실행을 제어**하는 역할을 한다.
이 동작 원리를 이해하면 `@Transactional`, `@Async`, `@Cacheable`, `@Retryable` 등이 어떻게 작동하는지도 명확해진다.