

# Spring Framework

S p r i n g   F r a m e w o r k



# Index

## CONTENTS



01 Spring &  
개발환경



03 의존성주입  
(DI)



05 MyBatis



07 Spring 기타



02 IoC &  
Contatiner



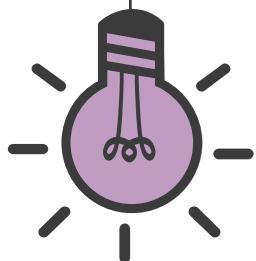
04 관점지향  
프로그램(AOP)



06 Spring MVC

01

## Spring & 개발환경





# 1. Spring

## 1-1. SpringFramework 등장배경.

EJB를 사용하면 애플리케이션 작성을 쉽게 할 수 있다.

저수준의 트랜잭션이나 상태관리, 멀티 쓰레딩, 리소스 풀링과 같은 복잡한 저수준의 API 따위를 이해하지 못하더라도 아무 문제 없이 애플리케이션을 개발할 수 있다.

– Enterprise JavaBean 1.0 Specification, Chapter 2 Goals

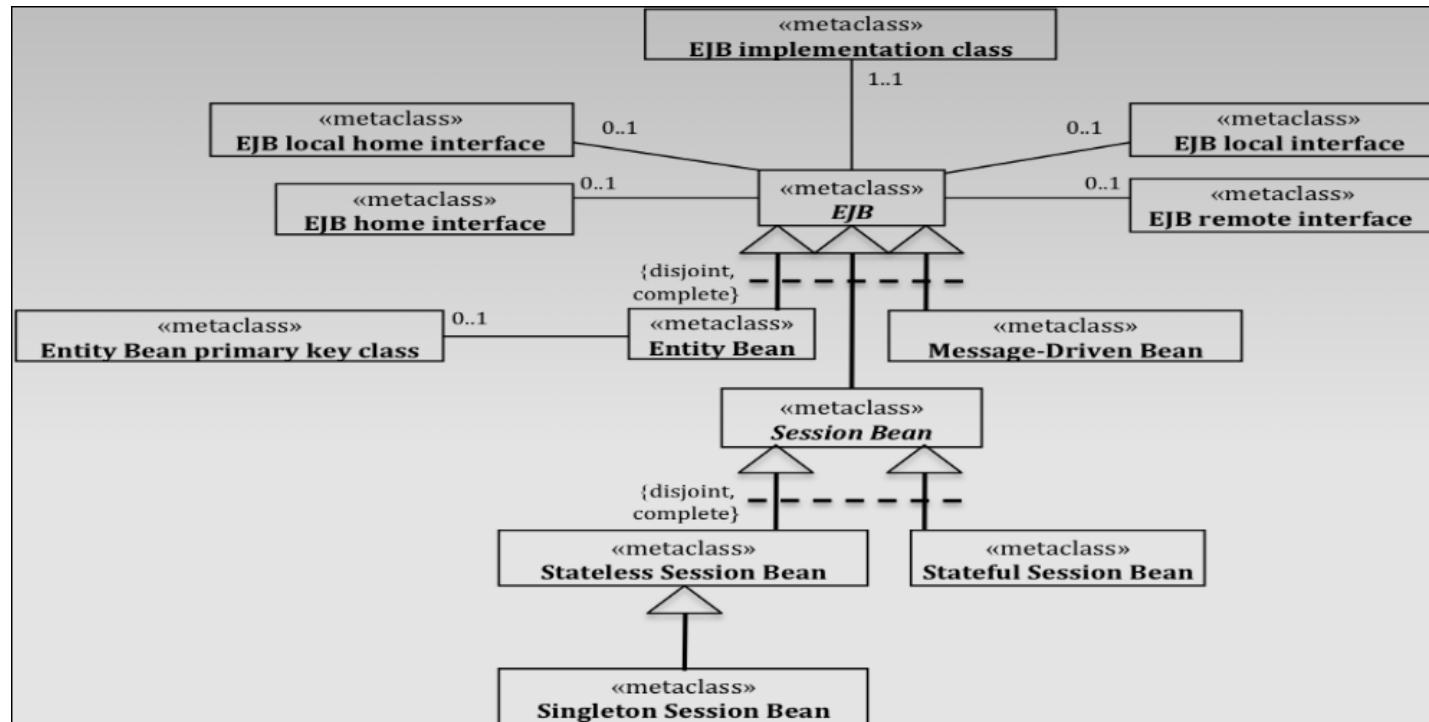


# 1. Spring

## 1-1. SpringFramework 등장배경.

### ✓ EJB... 현실에서의 반영은 어렵다.

- 코드 수정 후 반영하는 과정 자체가 거창해 기능은 좋지만
- 복잡한 스펙으로 인한 개발의 효율성이 떨어짐
- 어플리케이션을 테스트하기 위해서는 반드시 EJB서버가 필요하다.





# 1. Spring

## 1-1. SpringFramework 등장배경.

- ✓ 웹사이트가 점점 커지면서 엔터프라이즈급의 서비스가 필요하게 됨
  - 세션бин에서 Transaction 관리가 용이함
  - 로긴, 분산처리, 보안등
- ✓ 자바진영에서는 EJB가 엔터프라이즈급 서비스로 각광을 받게 됨
  - EJB스펙에 정의된 인터페이스에 따라 코드를 작성하므로 기존에 작성된 POJO를 변경해야 함
  - 컨테이너에 배포를 해야함 테스트가 가능해 개발속도가 저하됨
  - 배우기 어렵고, 설정해야 할 부분이 많음
- ✓ Rod Johnson이 'Expert One-on-One J2EE Development without EJB'라는 저서에서 EJB를 사용하지 않고 엔터프라이즈 어플리케이션을 개발하는 방법을 소개함(스프링의 모태)
  - AOP나 DI같은 새로운 프로그래밍 방법론으로 가능
  - POJO로 전언적인 프로그래밍 모델이 가능해짐



# 1. Spring

## 1-1. SpringFramework 등장배경.

- ✓ 점차 POJO + 경량 프레임워크를 사용하기 시작
- ✓ POJO (Plain Old Java Object)
  - 특정 프레임워크나 기술에 의존적이지 않은 자바 객체
  - 특정 기술에 종속적이지 않기 때문에 생산성, 이식성 향상
  - Plain : component interface를 상속받지 않는 특징  
(특정 framework에 종속되지 않는)
  - Old : EJB 이전의 java class를 의미
- ✓ 경량 프레임워크
  - EJB가 제공하는 서비스를 지원해 줄 수 있는 프레임워크 등장
  - Hibernate, JDO, iBatis(myBatis), Spring



# 1. Spring

## 1-1. SpringFramework 등장배경.

### ✓ POJO + Framework

- EJB서버와 같은 거창한 컨테이너가 필요 없다.
- 오픈소스 프레임워크라 사용이 무료.
- 각종 기업용 어플리케이션 개발에 필요한 상당히 많은 라이브러리가 지원.
- 스프링프레임워크는 모든 플랫폼에서 사용이 가능하다.
- 스프링은 웹분야 뿐만이 아니라 어플리케이션등 모든 분야에 적용이 가능한 다양한 라이브러리를 가지고 있다.



# 1. Spring

## 1-2. http://spring.io

The screenshot shows the official Spring website at <http://spring.io>. The page features a dark blue header with the Spring logo and navigation links for PROJECTS, GUIDES, and BLOG. Below the header, a large banner announces the release of Spring Cloud Edgware. The banner text reads: "Spring Cloud Edgware Released" and "Spring Cloud Edgware debuts a JDBC-backed config server, Contract, Sleuth and Vault improvements. [Learn More](#)". A horizontal ellipsis indicates more content follows. The main content area below the banner has a white background with the text "Spring: the source for modern java". To the left, there's a diagram showing a "Your App" icon connected to three green hexagonal nodes labeled "Build Anything" (with a Spring Boot icon), "Coordinate Anything" (with a Spring Cloud icon), and "Connect Everything" (with a Spring Cloud Data Flow icon). A vertical bar on the right side of the diagram has a slider handle.

Spring

Spring by Pivotal

PROJECTS GUIDES BLOG

Spring Cloud Edgware Released

Spring Cloud Edgware debuts a JDBC-backed config server, Contract, Sleuth and Vault improvements. [Learn More](#)

Spring: the source for modern java

Your App

Build Anything

Coordinate Anything

Connect Everything



# 1. Spring

## 1-2. <http://spring.io/projects>



### SPRING BOOT

Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.



### SPRING FRAMEWORK

Provides core support for dependency injection, transaction management, web apps, data access, messaging and more.



### SPRING CLOUD DATA FLOW

An orchestration service for composable data microservice applications on modern runtimes.



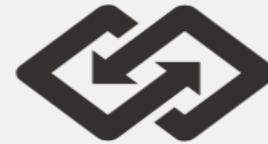
### SPRING CLOUD

Provides a set of tools for common patterns in distributed systems. Useful for building and deploying microservices.



### SPRING DATA

Provides a consistent approach to data access – relational, non-relational, map-reduce, and beyond.



### SPRING INTEGRATION

Supports the well-known *Enterprise Integration Patterns* via lightweight messaging and declarative adapters.



# 1. Spring

## 1-3. SpringFramework?

- ✓ 엔터프라이즈급 애플리케이션을 만들기 위한 모든 기능을 종합적으로 제공하는 경량화 된 솔루션이다.
- ✓ JEE(Java Enterprise Edition)가 제공하는 다수의 기능을 지원하고 있기 때문에, JEE를 대체하는 Framework로 자리잡고 있다.
- ✓ SpringFramework는 JEE가 제공하는 다양한 기능을 제공하는 것 뿐만 아니라, DI(Dependency Injection)나 AOP (Aspect Oriented Programming)와 같은 기능도 지원 한다.



# 1. Spring

## 1-3. SpringFramework?

- ✓ Spring Framework는 자바로 Enterprise Application을 만들 때 포괄적으로 사용하는 Programming 및 Configuration Model을 제공해 주는 Framework로 Application 수준의 인프라 스트럭쳐를 제공.
- ✓ 즉, 개발자가 복잡하고 실수하기 쉬운 Low Level에 신경 쓰지 않고 Business Logic개발에 전념할 수 있도록 해준다.



Enterprise System이란 서버에서 동작하며  
기업의 업무를 처리해주는 System

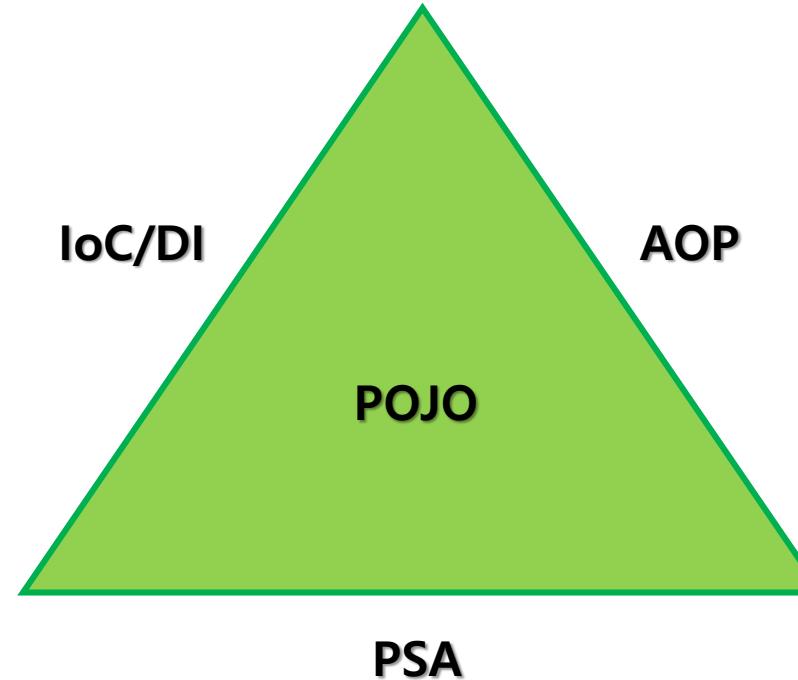


# 1. Spring

## 1-4. SpringFramework의 구조.

### ✓ Spring 삼각형.

- Enterprise Application 개발 시 복잡함을 해결하는 Spring의 핵심.
  - POJO
  - PSA
  - IoC/DI
  - AOP





# 1. Spring

## 1-4. SpringFramework의 구조.

### ✓ Spring 삼각형.

- Enterprise Application 개발 시 복잡함을 해결하는 Spring의 핵심.

- POJO

- PSA

- IoC/DI

- AOP



### ✓ POJO (Plain Old Java Object)

- 특정 환경이나 기술에 종속적이지 않은 객체지향 원리에 충실한 자바객체.
- 테스트하기 용이하며, 객체지향 설계를 자유롭게 적용할 수 있다.



# 1. Spring

## 1-4. SpringFramework의 구조.

### ✓ Spring 삼각형.

- Enterprise Application 개발 시 복잡함을 해결하는 Spring의 핵심.
  - POJO
  - **PSA**
  - IoC/DI
  - AOP



### ✓ **PSA (Portable Service Abstraction)**

- 환경과 세부기술의 변경과 관계없이 일관된 방식으로 기술에 접근할 수 있게 해주는 설계 원칙.
- 트랜잭션 추상화, OXM 추상화, 데이터 액세스의 Exception 변환기능..등 기술적인 복잡함은 추상화를 통해 Low Level의 기술 구현 부분과 기술을 사용하는 인터페이스로 분리.
- 예를 들어 데이터베이스에 관계없이 동일하게 적용 할 수 있는 트랜잭션 처리방식.



# 1. Spring

## 1-4. SpringFramework의 구조.

### ✓ Spring 삼각형.

- Enterprise Application 개발 시 복잡함을 해결하는 Spring의 핵심.
  - POJO
  - PSA
  - **IoC/DI**
  - AOP



### ✓ **IoC/DI (Dependency Injection)**

- DI는 유연하게 확장 가능한 객체를 만들어 두고 객체 간의 의존관계는 외부에서 다이나믹하게 설정.



# 1. Spring

## 1-4. SpringFramework의 구조.

### ✓ Spring 삼각형.

- Enterprise Application 개발 시 복잡함을 해결하는 Spring의 핵심.
  - POJO
  - PSA
  - IoC/DI
  - **AOP**



### ✓ AOP (Aspect Oriented Programming)

- 관심사의 분리를 통해서 소프트웨어의 모듈성을 향상.
- 공통 모듈을 여러 코드에 쉽게 적용 가능.



# 1. Spring

## 1-5. SpringFramework의 특징.

### ✓ 경량컨테이너

- 스프링은 자바객체를 담고 있는 컨테이너이다.
- 스프링 컨테이너는 이들 자바 객체의 생성과 소멸과 같은 라이프사이클을 관리
- 언제든지 스프링 컨테이너로부터 필요한 객체를 가져와 사용할 수 있다.

### ✓ DI(Dependency Injection – 의존성 지원) 패턴 지원

- 스프링은 설정파일이나, 어노테이션을 통해서 객체 간의 의존 관계를 설정할 수 있다.
- 따라서, 객체는 의존하고 있는 객체를 직접 생성하거나 검색할 필요가 없다.



# 1. Spring

## 1-5. SpringFramework의 특징.

### ✓ AOP(Aspect Oriented Programming – 측면 지향 프로그래밍) 지원

- AOP는 문제를 바라보는 관점을 기준으로 프로그래밍하는 기법이다.
- 이는 문제를 해결하기 위한 **핵심관심 사항**과 전체에 적용되는 **공통관심 사항**을 기준으로 프로그래밍 함으로서 공통 모듈을 여러 코드에 쉽게 적용할 수 있도록 한다.
- 스프링은 자체적으로 프록시 기반의 AOP를 지원하므로 트랜잭션이나, 로깅, 보안과 같이 여러 모듈에서 공통으로 필요로 하지만 실제 모듈의 핵심이 아닌 기능들을 분리하여 각 모듈에 적용이 가능하다.

### ✓ POJO(Plain Old Java Object ) 지원

- 특정 인터페이스를 구현하거나 또는 클래스를 상속하지 않는 일반 자바 객체 지원.
- 스프링 컨테이너에 저장되는 자바객체는 특정한 인터페이스를 구현하거나, 클래스 상속 없이도 사용이 가능하다.
- 일반적인 자바 객체를 칭하기 위한 별칭 개념이다.



# 1. Spring

## 1-5. SpringFramework의 특징.

### ✓ IoC(Inversion of Control – 제어의 반전)

- IoC는 스프링이 갖고 있는 핵심적인 기능이다.
- 자바의 객체 생성 및 의존관계에 있어 모든 제어권은 개발자에게 있었다.
- Servlet과 EJB가 나타나면서 기존의 제어권이 Servlet Container 및 EJB Container에게 넘어가게 됐다
- 단, 모든 객체의 제어권이 넘어간 것은 아니고 Servlet, EJB에 대한 제어권을 제외한 나머지 객체 제어권은 개발자들이 담당하고 있다.
- 스프링에서도 객체에 대한 생성과 생명주기를 관리할 수 있는 기능을 제공하고 있는데 이런 이유로 [Spring Container] 또는 [IoC Container] 라고 부르기도 한다.

### ✓ 스프링은 트랜잭션 처리를 위한 일관된 방법을 제공한다

- JDBC, JTA 또는 컨테이너가 제공하는 트랜잭션을 사용하든, 설정파일을 통해 트랜잭션 관련정보를 입력하기 때문에 트랜잭션 구현에 상관 없이 동일한 코드를 여러 환경에서 사용이 가능하다.



# 1. Spring

## 1-5. SpringFramework의 특징.

- ✓ **스프링은 영속성과 관련된 다양한 API를 지원한다.**

- 스프링은 JDBC를 비롯하여 iBatis, Mybatis, Hibernate, JPA등 DB처리를 위해 널리 사용되는 라이브러리와 연동을 지원하고 있다.

- ✓ **스프링은 다양한 API에 대한 연동을 지원한다.**

- 스프링은 JMS, 메일, 스케줄링등 엔터프라이즈 어플리케이션 개발에 필요한 다양한 API를 설정파일과 어노테이션을 통해서 손쉽게 사용할 수 있도록 지원하고 있다.



# 1. Spring

## 1-6. SpringFramework Module. (1/2)

**Spring AOP**  
Source-level  
Metadata

**Spring ORM**  
Hibemate, iBATIS and  
JDO Support

**Spring Web**  
WebApplicationContext  
Multipart Resolver Web  
Utilities

**Spring MVC**  
Web Framework  
Web Views  
JSP, Velocity,  
Freemarker, PDF,  
Excel, XML/XSL

**Spring DAO**  
Transaction  
infrastructure JDBC  
and DAO support

**Spring Context**  
ApplicationContext  
UI Support  
Validation  
JNDI, EJB & Remoting  
Support mail

**Spring Core**  
Supporting Utilities  
Bean Factory / Container



# 1. Spring

## 1-6. SpringFramework Module. (2/2)

1. Spring Core	Spring Framework의 핵심 기능을 제공 하며, Core 컨테이너의 주요 컴포넌트는 Bean Factory이다.
2. Spring Context	- Spring을 컨테이너로 만든 것이 Spring core의 BeanFactory라면, Spring을 Framework로 만든 것은 Context module이며, 이 module은 국제화된 메시지, Application 생명주기 이벤트, 유효성 검증 등을 지원함으로써 BeanFactory의 개념을 확장을 한다.
3. Spring AOP	설정 관리 기능을 통해 AOP 기능을 Spring Framework와 직접 통합 시킨다.
4. Spring DAO	Spring JDBC DAO 추상 레이어는 다른 데이터베이스 벤더들의 예외 핸들링과 오류 메시지를 관리하는 중요한 예외계층을 제공합니다.
5. Spring ORM	Spring Framework는 여러 ORM(Object Relational Mapping) Framework에 플러그인 되어, Object Relational 툴(JDO, Hibernate, iBatis)을 제공한다.
6. Spring Web	Web Context module은 Application Context module 상위에 구현되어, Web 기반 Application에 context를 제공한다.
7. Spring Web MVC	Spring Framework는 자체적으로 MVC 프레임워크를 제공하고 있으며, 스프링만 사용해도 MVC기반의 웹 어플리케이션을 어렵지 않게 개발이 가능하다.



# 1. Spring

## 1-7. 개발환경(1/3)

### ✓ JDK 설치

- <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- 환경변수 설정
  - JAVA\_HOME, CLASSPATH, Path

### ✓ Tomcat 설치

- <http://tomcat.apache.org/>
- apache-tomcat-x.x.x version download 후 설치

### ✓ Oracle 설치

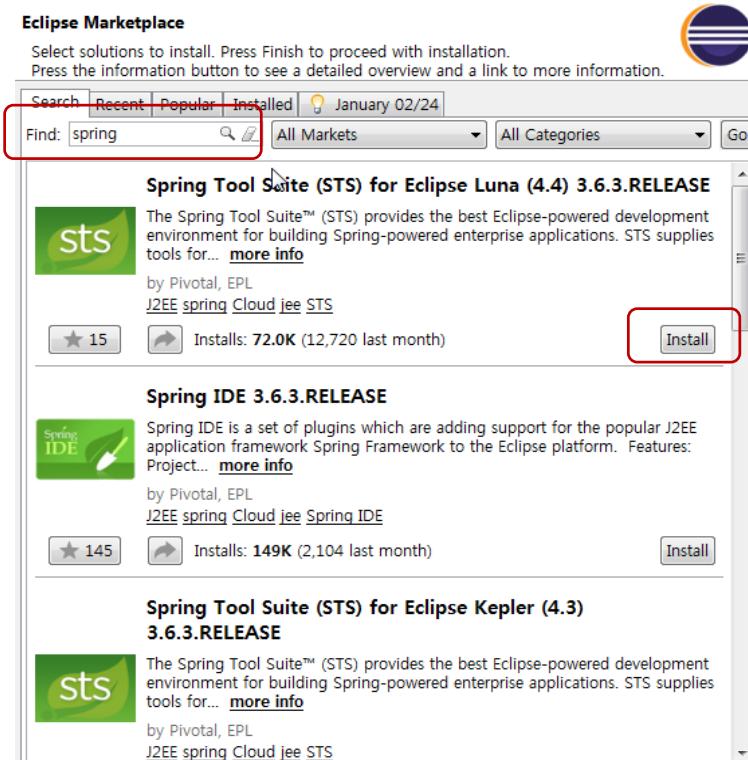
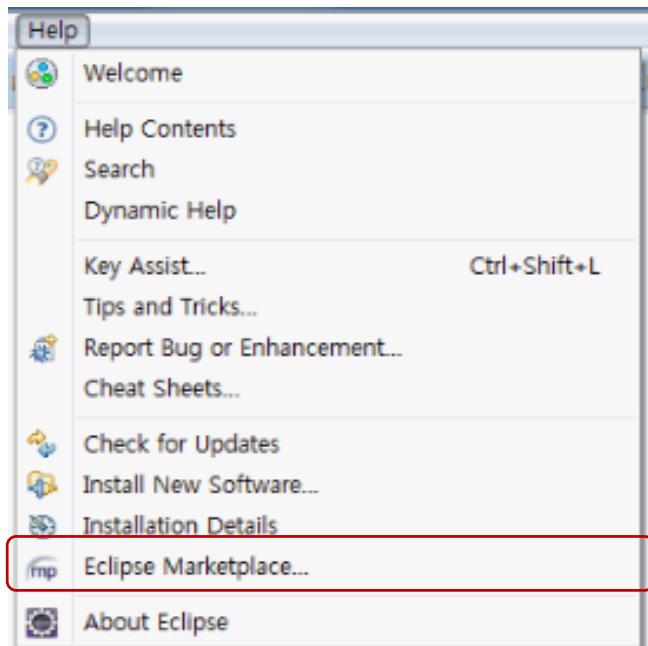
- <http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html>

# 1. Spring

## 1-7. 개발환경(2/3)

### ✓ Eclipse 설치

- <http://www.eclipse.org/downloads>
- Eclipse IDE for Java EE Developer download
- Help > Eclipse Marketplace > spring 검색
- Plug-in 설치



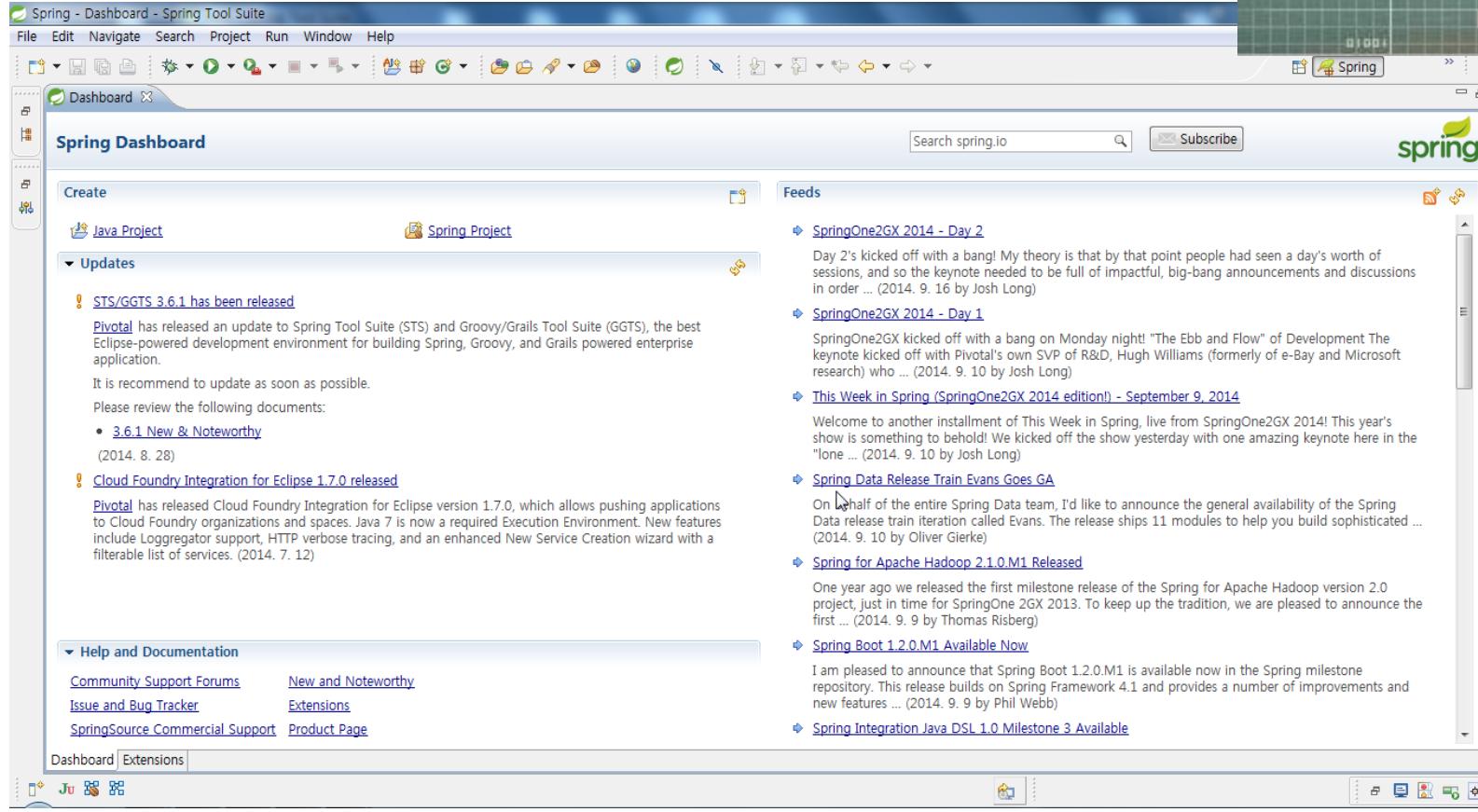


# 1. Spring

## 1-7. 개발환경(3/3)

✓ 실습 : STS(Eclipse + Spring, Spring Tool Suite), Maven

- <http://spring.io/tools/sts/>

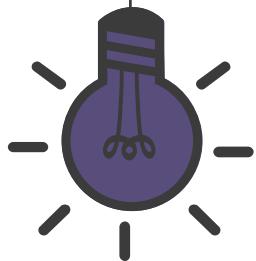


The screenshot shows the Spring Dashboard interface within the Spring Tool Suite (STS). The dashboard has a "Create" section with links for "Java Project" and "Spring Project". A "Updates" section highlights the release of "STS/GGTS 3.6.1". It includes a message from Pivotal about the update and a note to update as soon as possible. Below this, there's a link to "Cloud Foundry Integration for Eclipse 1.7.0 released". The "Help and Documentation" section provides links to "Community Support Forums", "New and Noteworthy", "Issue and Bug Tracker", "Extensions", "SpringSource Commercial Support", and "Product Page". On the right side, there's a "Feeds" section displaying a list of news items from the SpringOne2GX 2014 conference, including "Day 2", "Day 1", "This Week in Spring (SpringOne2GX 2014 edition) - September 9, 2014", "Spring Data Release Train Evans Goes GA", "Spring for Apache Hadoop 2.1.0.M1 Released", and "Spring Boot 1.2.0.M1 Available Now". There's also a link to "Spring Integration Java DSL 1.0 Milestone 3 Available".



02

## IoC & Container





## 2. IoC & Container

### 2-1. IoC (Inversion of Control)

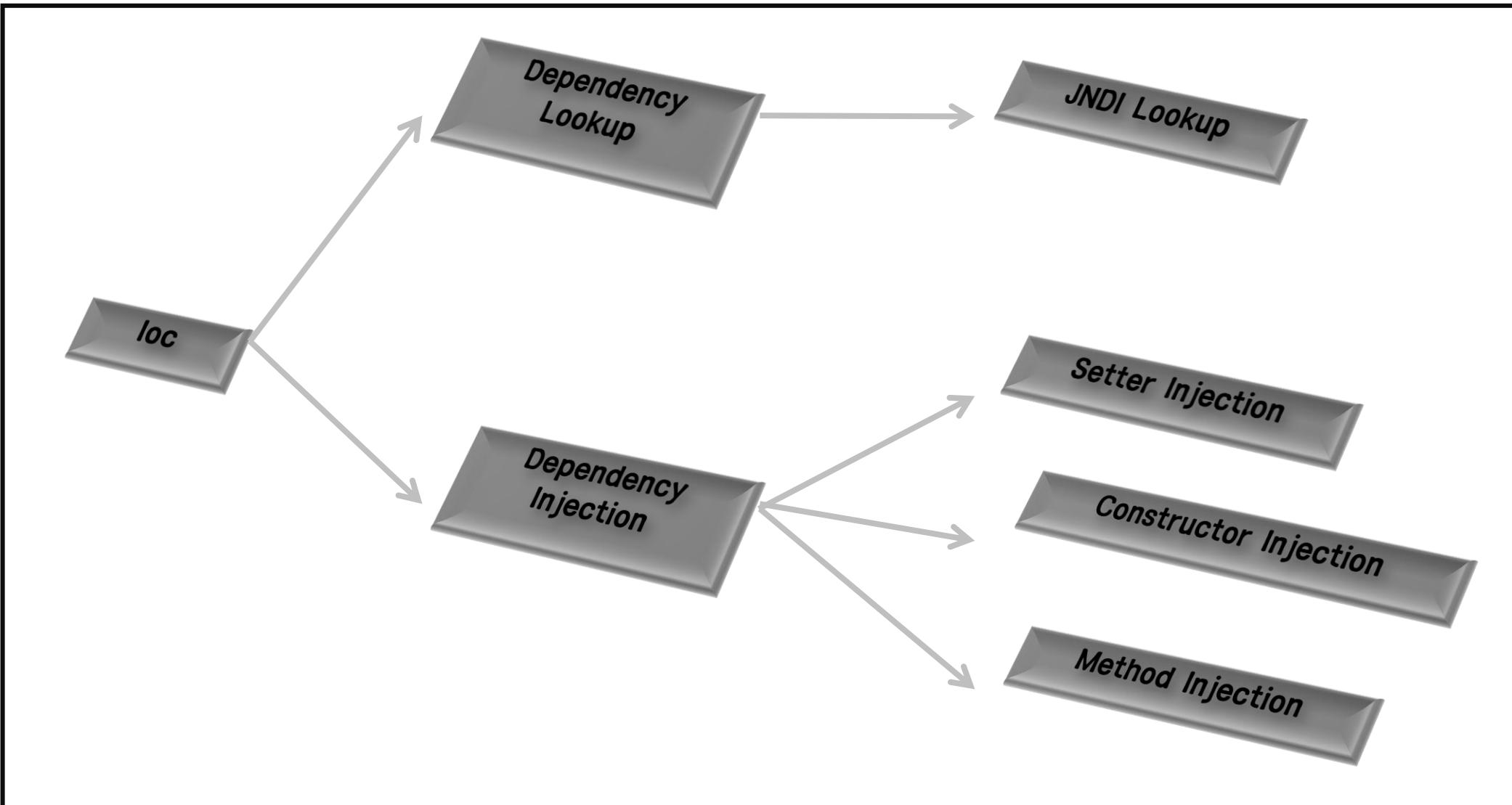
- ✓ IoC (Inversion of Control, 제어의 역행)

- IoC/DI.
- 객체지향 언어에서 Object간의 연결 관계를 런타임에 결정.
- 객체 간의 관계가 느슨하게 연결됨(loose coupling).
- IoC의 구현 방법 중 하나가 DI(Dependency Injection).



## 2. IoC & Container

### 2-2. IoC 유형(1/3)





## 2. IoC & Container

### 2-2. IoC 유형(2/3)

#### ✓ Dependency Lookup

- 컨테이너가 lookup context를 통해서 필요한 Resource나 Object를 얻는 방식.
- JNDI 이외의 방법을 사용한다면 JNDI관련 코드를 오브젝트 내에서 일일이 변경해 주어야 함.
- Lookup 한 Object를 필요한 타입으로 Casting 해 주어야 함.
- Naming Exception을 처리하기 위한 로직이 필요.



## 2. IoC & Container

### 2-2. IoC 유형(3/3)

#### ✓ Dependency Injection

- Object에 lookup 코드를 사용하지 않고 컨테이너가 직접 의존 구조를 Object에 설정 할 수 있도록 지정해 주는 방식.
- Object가 컨테이너의 존재 여부를 알 필요가 없음.
- Lookup 관련된 코드들이 Object 내에서 사라짐.
- Setter Injection과 Constructor Inject.



## 2. IoC & Container

### 2-3. Container

#### ✓ Container 란?

- 객체의 생성, 사용, 소멸에 해당하는 라이프사이클을 담당
- 라이프사이클을 기본으로 애플리케이션 사용에 필요한 주요 기능을 제공

#### ✓ Container 기능

- 라이프사이클 관리
- Dependency 객체 제공
- Thread 관리
- 기타 애플리케이션 실행에 필요한 환경

#### ✓ Container 필요성

- 비즈니스 로직 외에 부가적인 기능들에 대해서는 독립적으로 관리되도록 하기 위함
- 서비스 look up이나 Configuration에 대한 일관성을 갖기 위함
- 서비스 객체를 사용하기 위해 각각 Factory 또는 Singleton 패턴을 직접 구현하지 않아도 됨



## 2. IoC & Container

### 2-3. Container

#### ✓ IoC Container

- 오브젝트의 생성과 관계설정, 사용, 제거 등의 작업을 애플리케이션 코드 대신 독립된 컨테이너가 담당.
- 컨테이너가 코드 대신 오브젝트에 대한 제어권을 갖고 있어 IoC라고 부름.
- 이런 이유로, 스프링 컨테이너를 IoC 컨테이너라고 부르기도 함.
- 스프링에서 IoC를 담당하는 컨테이너에는 BeanFactory, ApplicationContext가 있음.

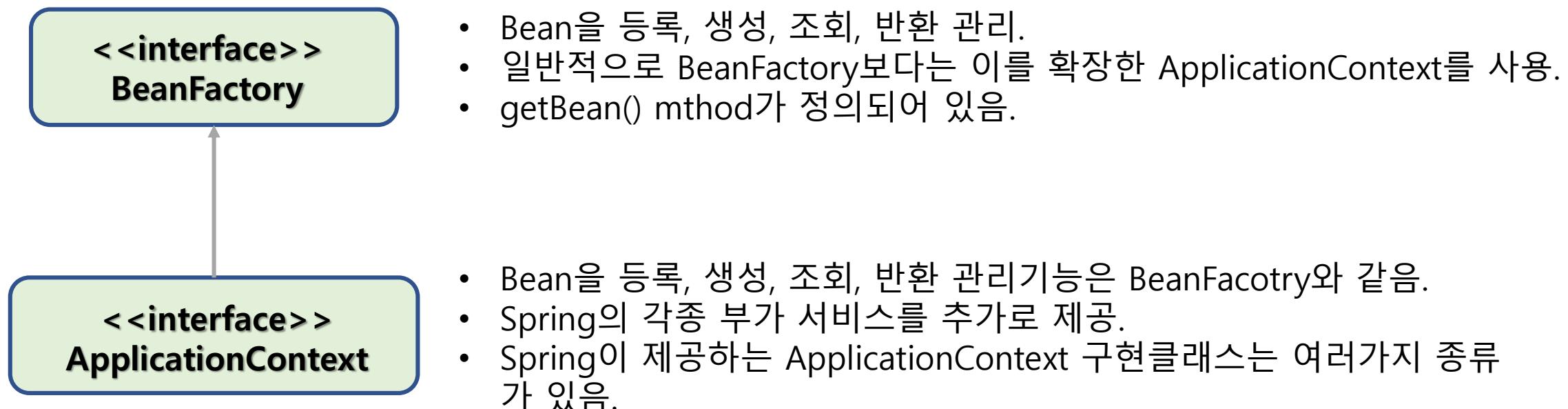


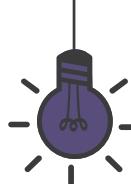
## 2. IoC & Container

### 2-3. Container

#### ✓ Spring DI Container

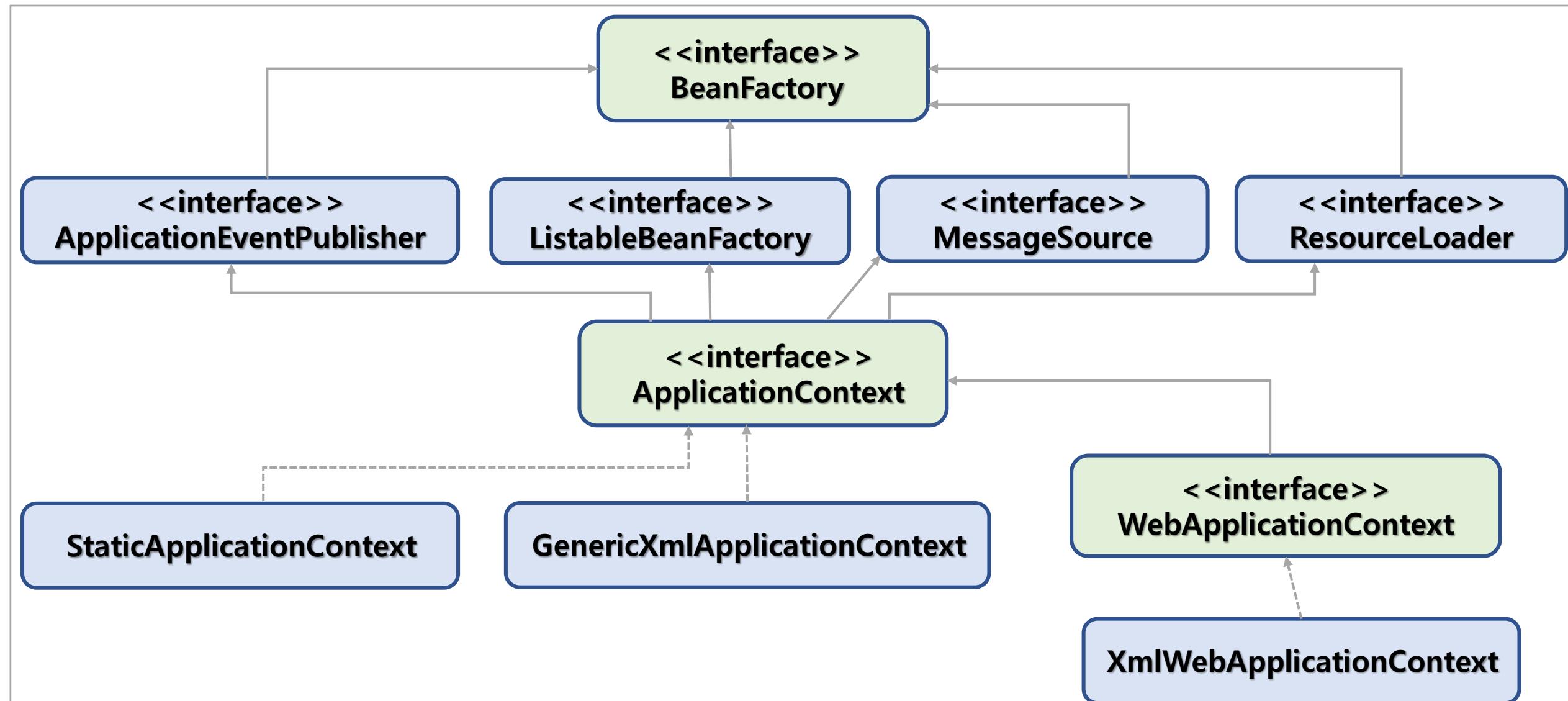
- Spring DI Container가 관리하는 객체를 빈(Bean)이라 하고, 이 빈들의 생명주기(Life-Cycle)를 관리하는 의미로 빈팩토리(BeanFactory)라 한다.
- Bean Factory에 여라 가지 컨테이너 기능을 추가하여 ApplicationContext라 한다.





## 2. IoC & Container

### 2-4. BeanFactory & ApplicationContext





## 2. IoC & Container

### 2-5. IoC 개념 (1/5)

#### ✓ 객체 제어 방식

- 기존 : 필요한 위치에서 개발자가 필요한 객체 생성 로직 구현
- IoC : 객체 생성을 Container에게 위임하여 처리

#### ✓ IoC 사용에 따른 장점

- 객체간의 결합도를 떨어뜨릴 수 있음 (loose coupling)

#### ✓ 객체간 결합도가 높으면?

- 해당 클래스가 유지보수 될 때 그 클래스와 결합된 다른 클래스도 같이 유지보수 되어야 할 가능성이 높음

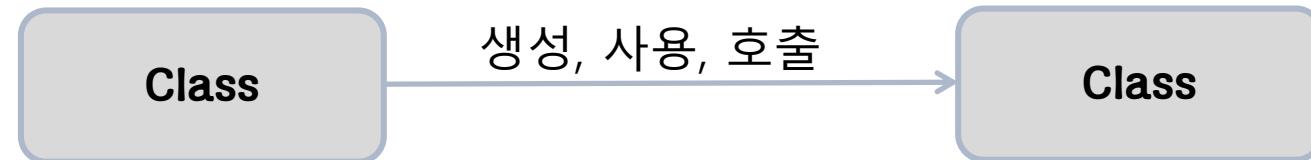


## 2. IoC & Container

### 2-5. IoC 개념 (2/5)

#### ✓ 객체간 강한 결합

- 클래스 호출 방식.
- 클래스내에 선언과 구현이 모두 되어 있기 때문에 다양한 형태로 변화가 불가능.





## 2. IoC & Container

### 2-5. IoC 개념 (2/5)

#### ✓ 객체간 강한 결합

- MemberService 구현체와 AdminService 구현체를 HomeController에서 직접 생성하여 사용.
- MemberService 또는 AdminService가 교체되거나 내부 코드가 변경되면 HomeController까지 수정해야 할 가능성이 있음.

```
public class HomeController {  
  
    private MemberServiceImpl memberService = new MemberServiceImpl();  
    private AdminServiceImpl adminService = new AdminServiceImpl();  
  
    public void addUser(MemberDto memberDto) {  
        memberService.joinMember(memberDto);  
        adminService.registerMember(memberDto);  
    }  
}
```

HomeController에서 MemberService의 구현체와 AdminService의 구현체를 직접 생성

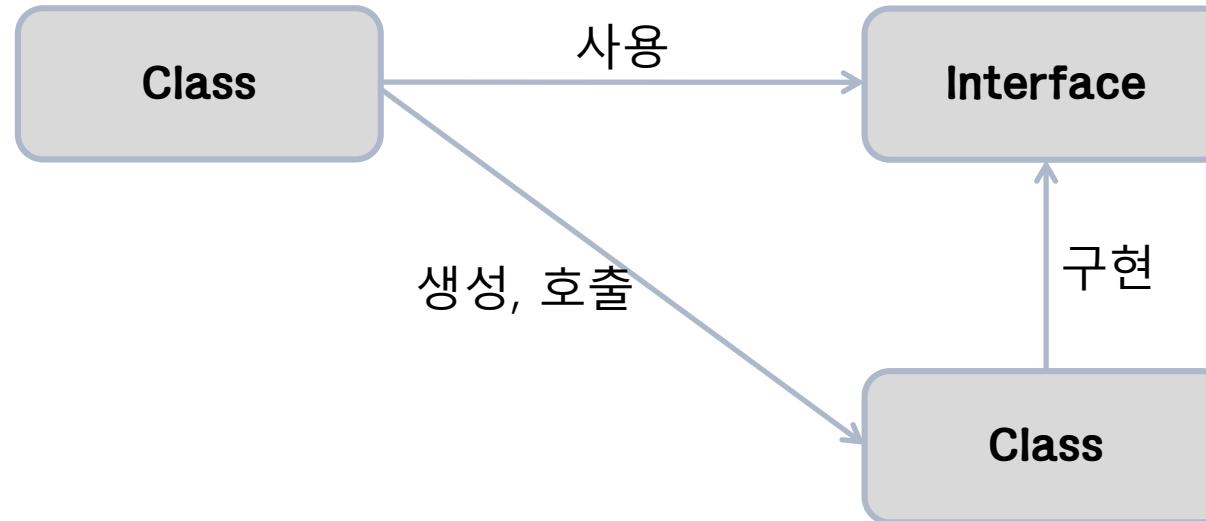


## 2. IoC & Container

### 2-5. IoC 개념 (3/5)

✓ 객체간의 강한 결합을 다형성을 통해 결합도를 낮춤.

- 인터페이스 호출 방식.
- 구현클래스 교체가 용이하여 다양한 형태로 변화가능.
- 하지만 인터페이스 교체 시 호출클래스도 수정해야 함.





## 2. IoC & Container

### 2-5. IoC 개념 (3/5)

#### ✓ 객체간의 강한 결합을 다형성을 통해 결합도를 낮춤.

- MemberService와 AdminService는 CommonService를 상속.
- HomeController에서 각 서비스를 이용 시 MemberService와 AdminService는 CommonService Type으로 사용 가능.

```
public interface MemberService extends CommonService {  
}
```

```
public interface AdminService extends CommonService {  
    public List<MemberDto> listMember();  
}
```

```
public class HomeController {  
  
    private CommonService memberService = new MemberServiceImpl();  
    private CommonService adminService = new AdminServiceImpl();  
  
    public void addUser(MemberDto memberDto) {  
        memberService.registerMember(memberDto);  
        adminService.registerMember(memberDto);  
    }  
  
}
```

MemberService와 AdminService 모두  
CommonService Type으로 이용 가능.

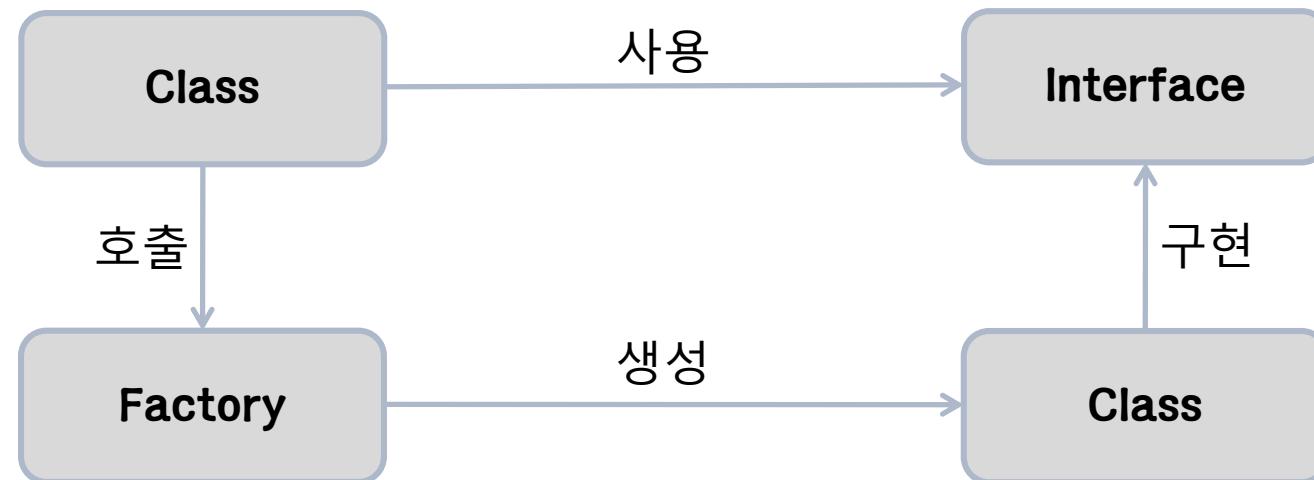


## 2. IoC & Container

### 2-5. IoC 개념 (4/5)

✓ 객체간의 강한 결합을 Factory를 통해 결합도를 낮춤.

- 팩토리 호출 방식.
- 팩토리방식은 팩토리가 구현클래스를 생성하므로 클래스는 팩토리를 호출.
- 인터페이스 변경 시 팩토리만 수정하면 됨. 호출클래스에는 영향을 미치지 않음.
- 하지만 클래스에 팩토리를 호출하는 소스가 들어가야함. 그것 자체가 팩토리에 의존함을 의미한다.





## 2. IoC & Container

### 2-5. IoC 개념 (4/5)

#### ✓ 객체간의 강한 결합을 Factory를 통해 결합도를 낮춤.

- 각 Service를 생성하여 반환하는 Factory 사용.
- Service를 이용하는 쪽에서는 Interface만 알고 있으면 어떤 구현체가 어떻게 생성되는지에 대해서는 알 필요 없음.
- 이 Factory 패턴이 적용된 것이 Container의 기능이며 이 Container의 기능을 제공해 주고자 하는 것이 IoC 모듈.

```
public class HomeController {  
  
    private CommonService memberService = ServiceFactory.getMemberService();  
    private CommonService adminService = ServiceFactory.getAdminService();  
  
    public void addUser(MemberDto memberDto) {  
        memberService.registerMember(memberDto);  
        adminService.registerMember(memberDto);  
    }  
}
```

```
public class ServiceFactory {  
  
    public static CommonService getMemberService() {  
        return new MemberServiceImpl();  
    }  
  
    public static CommonService getAdminService() {  
        return new AdminServiceImpl();  
    }  
}
```

각 서비스를 생성하여 반환해 주는 Factory.



## 2. IoC & Container

### 2-5. IoC 개념 (5/5)

✓ 객체간의 강한 결합을 Assembler를 통해 결합도를 낮춤.

- IoC 호출 방식.
- 팩토리 패턴의 장점을 더하여 어떠한 것에도 의존하지 않는 형태가 됨.
- 실행시점(Runtime)에 클래스 간의 관계가 형성이 됨.





## 2. IoC & Container

### 2-5. IoC 개념 (5/5)

#### ✓ 객체간의 강한 결합을 Assembler를 통해 결합도를 낮춤.

- 각 Service의 LifeCycle을 관리하는 Assembler를 사용.
- Spring Container가 외부조립기(Assembler) 역할을 함.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">

    <bean id="memberService" class="com.test.hello.service2.MemberServiceImpl"/>
    <bean id="adminService" class="com.test.hello.service2.AdminServiceImpl"/>

</beans>
```

Service 구현체 객체의 생성 및 관리를 Spring Container가 함.

```
//Resource resource = new ClassPathResource("com/test/hello/controller4/applicationContext.xml");
//BeanFactory factory = new XmlBeanFactory(resource);
//CommonService memberService = (MemberService) factory.getBean("memberService");
//CommonService adminService = (AdminService) factory.getBean("adminService");

ApplicationContext context =
    new ClassPathXmlApplicationContext("com/test/hello/controller4/applicationContext.xml");
CommonService memberService = context.getBean("memberService", MemberService.class);
CommonService adminService = context.getBean("adminService", AdminService.class);
```



## 2. IoC & Container

### 2-6. Spring DI 용어 (1/2)

#### ✓ 빈(Bean)

- 스프링이 IoC 방식으로 관리하는 오브젝트를 말한다.
- 스프링이 직접 그 생성과 제어를 담당하는 오브젝트만을 Bean이라고 부른다.

#### ✓ 빈 팩토리(Bean Factory)

- 스프링이 IoC를 담당하는 핵심 컨테이너.
- Bean을 등록, 생성, 조회, 반환하는 기능을 담당.
- 일반적으로 BeanFactory를 바로 사용하지 않고 이를 확장한 ApplicationContext를 이용한다.

#### ✓ 애플리케이션 컨텍스트(Application Context)

- BeanFactory를 확장한 IoC 컨테이너이다.
- Bean을 등록하고 관리하는 기본적인 기능은 BeanFactory와 동일하다.
- 스프링이 제공하는 각종 부가 서비스를 추가로 제공한다.
- BeanFactory라고 부를 때는 주로 빈의 생성과 제어의 관점에서 이야기하는 것이고, 애플리케이션 컨텍스트라고 할 때는 스프링이 제공하는 애플리케이션 지원 기능을 모두 포함해서 이야기하는 것이라고 보면 된다.



## 2. IoC & Container

### 2-6. Spring DI 용어 (2/2)

#### ✓ 설정정보/설정 메타정보(configuration metadata)

- 스프링의 설정정보란 ApplicationContext 또는 BeanFactory가 IoC를 적용하기 위해 사용하는 메타정보를 말한다. 이는 구성정보 내지는 형상정보라는 의미이다.
- 설정정보는 IoC 컨테이너에 의해 관리되는 Bean 객체를 생성하고 구성할 때 사용됨.

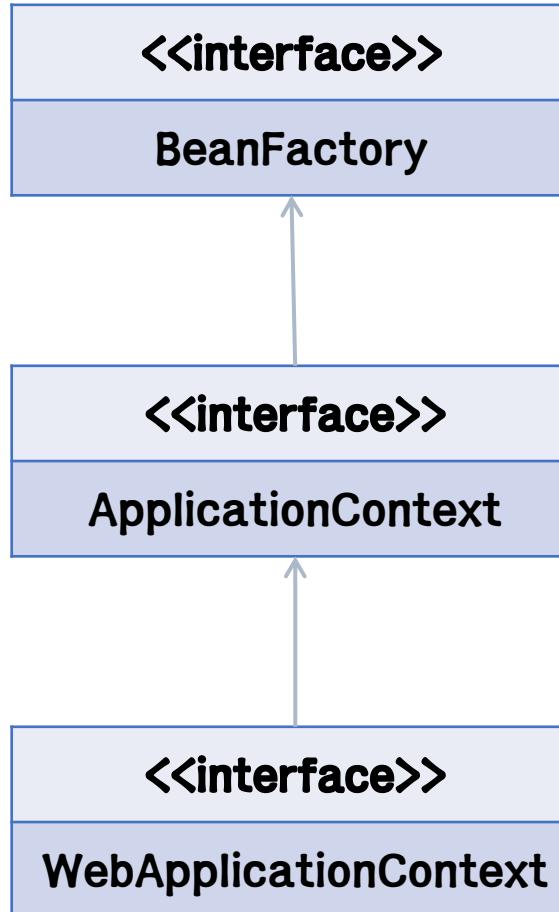
#### ✓ 스프링 프레임워크

- 스프링 프레임워크는 IoC 컨테이너, ApplicationContext를 포함해서 스프링이 제공하는 모든 기능을 통틀어 말할 때 주로 사용한다.



## 2. IoC & Container

### 2-7. Spring Container



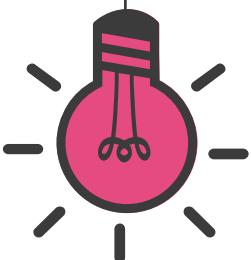
- 빈(Bean) 객체에 대한 생성과 제공을 담당.
- 단일 유형의 객체를 생성하는 것이 아니라, 여러 유형의 빈을 생성, 제공.
- 객체간의 연관 관계를 설정, 클라이언트의 요청 시 빈을 생성.
- 빈의 라이프 사이클을 관리.

- BeanFactory가 제공하는 모든 기능 제공.
- 엔터프라이즈 애플리케이션을 개발하는데 필요한 여러 기능을 추가함.
- I18N, 리소스 로딩, 이벤트 발생 및 통지.
- 컨테이너 생성시 모든 빈 정보를 메모리에 로딩함.

- 웹 환경에서 사용할 때 필요한 기능이 추가된 애플리케이션 컨텍스트.
- 가장 많이 사용하며, 특히 XmlWebApplicationContext를 가장 많이 사용

03

의존성 주입  
(DI, Dependency Injection)





# 3. Dependency Injection

## 3-1. 빈 생성 범위

### ✓ 싱글톤 빈(Singleton Bean)

- 스프링 빈은 기본적으로 싱글톤으로 만들어짐.
- 그러므로, 컨테이너가 제공하는 모든 빈의 인스턴스는 항상 동일함.
- 컨테이너가 항상 새로운 인스턴스를 반환하게 만들고 싶을 경우 scope를 prototype으로 설정해야 함.

```
@Component("memberService")
@Scope("singleton")
public class MemberServiceImpl implements MemberService {

    @Override
    public int registerMember(MemberDto memberDto) {
        return 0;
    }
}
```

```
<bean id="memberService" class="com.test.hello.service2.MemberServiceImpl" scope="prototype"/>
<bean id="adminService" class="com.test.hello.service2.AdminServiceImpl"/>
```



### 3. Dependency Injection

#### 3-2. 빈 범위 지정

##### 범위

##### 설명

**singleton**

스프링 컨테이너당 하나의 인스턴스 빌만 생성(default).

**prototype**

컨테이너에 빈을 요청할 때마다 새로운 인스턴스 생성.

**request**

HTTP Request별로 새로운 인스턴스 생성.

**session**

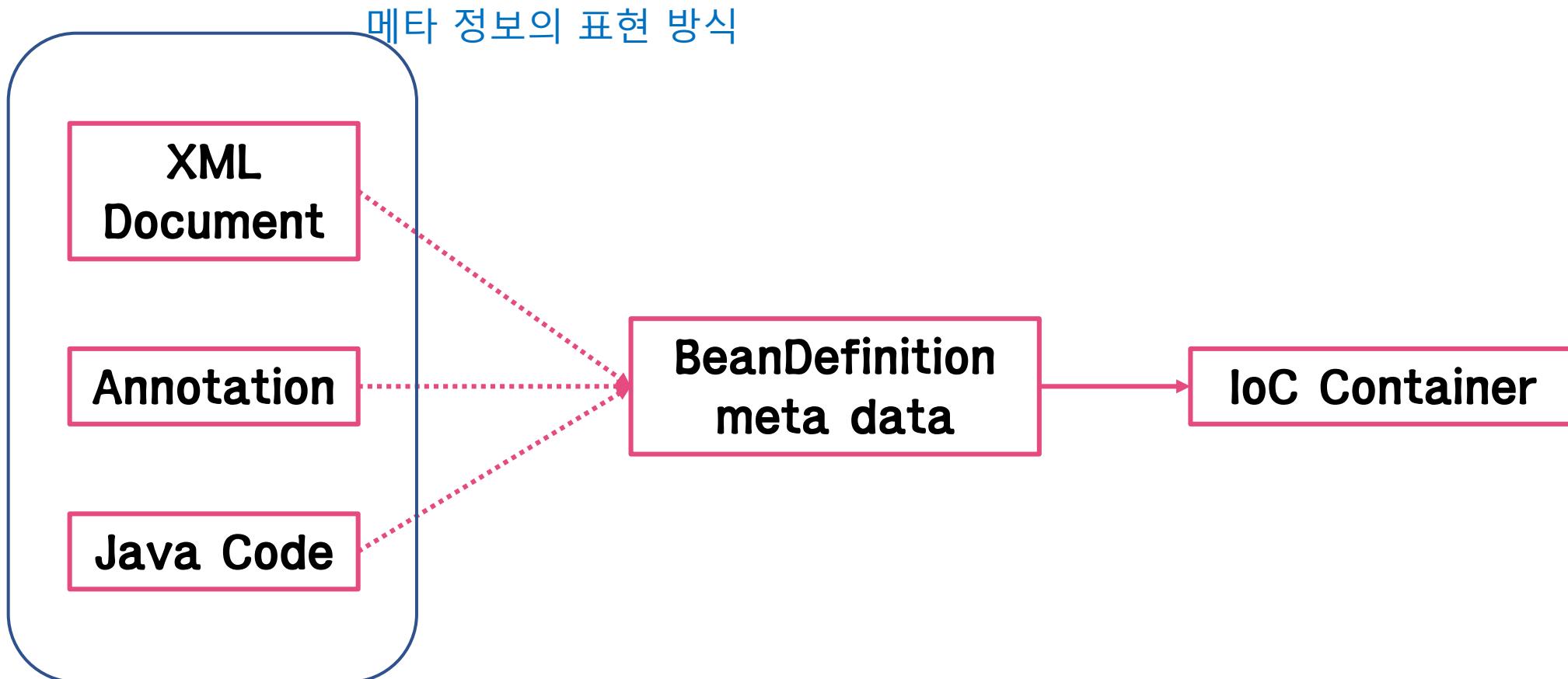
HTTP Session별로 새로운 인스턴스 생성.



### 3. Dependency Injection

#### 3-3. 스프링 빈 설정

- ✓ 스프링 빈 설정 메타정보





# 3. Dependency Injection

## 3-3. 스프링 빈 설정 - XML

### ✓ XML 문서

- XML문서 형태로 빈의 설정 메타 정보를 기술.
- 단순하며 사용하기 쉬움, 가장 많이 사용하는 방식.
- <bean> 태그를 통해 세밀한 제어 가능.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">

    <bean id="memberDao" class="com.test.hello.dao.MemberDaoImpl"/>

    <bean id="memberService" class="com.test.hello.service2.MemberServiceImpl" scope="prototype">
        <property name="memberDao" ref="memberDao"/>
    </bean>

    <bean id="adminService" class="com.test.hello.service2.AdminServiceImpl"/>

</beans>
```



# 3. Dependency Injection

## 3-3. 스프링 빈 설정 - annotation

### ✓ Annotation

- 어플리케이션의 규모가 커지고 빈의 갯수가 많아질 경우 XML 파일을 관리하는 것이 번거로움.
- 빈으로 사용될 클래스에 특별한 annotation을 부여해 주면 자동으로 빈 등록 가능.
- “오브젝트 빈 스캐너”로 “빈 스캐닝”을 통해 자동 등록.
  - 빈 스캐너는 기본적으로 클래스 이름을 빈의 아이디로 사용.
  - 정확히는 클래스 이름의 첫 글자만 소문자로 바꾼 것을 사용.

```
@Component
public class MemberServiceImpl implements MemberService {

    @Autowired
    private MemberDao memberDao;

    @Override
    public int registerMember(MemberDto memberDto) {
        return memberDao.registerMember(memberDto);
    }

}
```



# 3. Dependency Injection

## 3-3. 스프링 빈 설정 - annotation

- ✓ Annotation으로 빈을 설정 할 경우 반드시 component-scan을 설정 해야 한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <context:component-scan base-package="com.test.hello.*"/>

</beans>
```



# 3. Dependency Injection

## 3-3. 스프링 빈 설정 - annotation

### ✓ **Stereotype annotation** 종류

- 빈 자동등록에 사용할 수 있는 annotation
- 빈 자동인식을 위한 annotation이 여러가지인 이유
  - 계층별로 빈의 특성이나 종류를 구분.
  - AOP Pointcut 표현식을 사용하면 특정 annotation이 달린 클래스만 설정 가능.
  - 특정 계층의 빈에 부가기능을 부여.

Stereotype	적용 대상
<b>@Repository</b>	Data Access Layer의 DAO 또는 Repository 클래스에 사용. DataAccessException 자동변환과 같은 AOP의 적용 대상을 선정하기 위해 사용.
<b>@Service</b>	Service Layer의 클래스에 사용.
<b>@Controller</b>	Presentation Layer의 MVC Controller에 사용. 스프링 웹 서블릿에 의해 웹 요청을 처리하는 컨트롤러 빈으로 선정.
<b>@Component</b>	위의 Layer 구분을 적용하기 어려운 일반적인 경우에 설정.

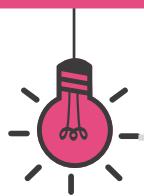


### 3. Dependency Injection

#### 3-4. 의존관계 주입(DI)

##### ✓ **Dependency Injection**

- 객체간의 의존관계를 자신이 아닌 외부의 조립기가 수행.
- 제어의 역행(inversion of Control, IoC) 이라는 의미로 사용.
- DI를 통해 시스템에 있는 각 객체를 조정하는 외부 개체가 객체들에게 생성시에 의존관계를 주어짐
- 느슨한 결합(loose coupling)의 주요강점
  - 객체는 인터페이스에 의한 의존관계만을 알고 있으며, 이 의존관계는 구현 클래스에 대한 차이를 모르는 채 서로 다른 구현으로 대체가 가능.



### 3. Dependency Injection

---

#### 3-5. 스프링의 DI 지원

- ✓ **Spring Container**가 DI 조립기를 제공.
  - 스프링 설정파일을 통하여 객체간의 의존관계를 설정.
  - Spring Container가 제공하는 api를 이용해 객체를 사용.



# 3. Dependency Injection

## 3-6. 스프링 설정

### ✓ XML 문서이용

- Application에서 사용할 Spring 자원들을 설정하는 파일
- Spring Container는 설정파일에 설정된 내용을 읽어 Application에서 필요한 기능들을 제공.
- Root tag는 <beans>
- 파일명은 상관없다.

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-4.3.xsd">

</beans>
```



# 3. Dependency Injection

## 3-6. 스프링 설정

### ✓ 기본설정 – 빈 객체 생성 및 주입

- 주입 할 객체를 설정파일에 설정
  - <bean> : 스프링 컨테이너가 관리할 Bean 객체를 설정
- 기본 속성
  - name : 주입 받을 곳에서 호출 할 이름 설정.
  - id : 주입 받을 곳에서 호출 할 이름 설정(유일 값).
  - class : 주입 할 객체의 클래스.
  - factory-method : Singleton 패턴으로 작성된 객체의 factory 메소드 호출

```
<bean id="memberDao" class="com.test.hello.dao.MemberDaoImpl"/>

<bean id="memberService" class="com.test.hello.service2.MemberServiceImpl" scope="prototype">
    <property name="memberDao" ref="memberDao"/>
</bean>

<bean id="adminService" class="com.test.hello.service2.AdminServiceImpl"/>
```



# 3. Dependency Injection

## 3-6. 스프링 설정

### ✓ 기본설정 – 빈 객체 얻기

- 설정 파일에 설정한 bean을 Container가 제공하는 주입기 역할의 api를 통해 주입 받는다.

```
//Resource resource = new ClassPathResource("com/test/hello/controller4/applicationContext.xml");
//BeanFactory factory = new XmlBeanFactory(resource);
//CommonService memberService = (MemberService) factory.getBean("memberService");
//CommonService adminService = (AdminService) factory.getBean("adminService");

ApplicationContext context =
    new ClassPathXmlApplicationContext("com/test/hello/controller4/applicationContext.xml");
CommonService memberService = context.getBean("memberService", MemberService.class);
CommonService adminService = context.getBean("adminService", AdminService.class);
```



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Constructor 이용 (1/8)

- 객체 또는 값을 생성자를 통해 주입 받는다.
- <constructor-arg> : <bean>의 하위태그로 설정한 bean 객체 또는 값을 생성자를 통해 주입하도록 설정.
  - 설정 방법 : <ref>, <value>와 같은 **하위태그**를 이용하여 설정하거나 또는 **속성**을 이용하여 설정.

#### 1. 하위태그 이용

- a. 객체 주입 시 : <ref bean="bean name"/>
- b. 문자열(String), primitive data 주입 시 : <value>값</value>
  - \* type 속성 : 값은 기본적으로 String으로 처리

값의 타입을 명시해야 하는 경우 사용

ex) <value type="int">10</value>

#### 2. 속성 이용

- a. 객체 주입 시 : <constructor-arg ref="bean name"/>
- b. 문자열(String), primitive data 주입 시 : <constructor-arg value="값"/>



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Constructor 이용 (2/8)

```
public Player(int num, String name) {//1번 생성자
    this.num = num;
    this.name = name;
}

public Player(int num, String name, int position) {//2번 생성자
    this.num = num;
    this.name = name;
    this.position = position;
}

public Player(int num, String name, double record) {//3번 생성자
    super();
    this.num = num;
    this.name = name;
    this.record = record;
}

public Player(int num, String name, int position, double record) {//4번 생성자
    this.num = num;
    this.name = name;
    this.position = position;
    this.record = record;
}
```

주입 받을 객체 생성자



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Constructor 이용 (3/8)

- 1번 생성자 주입 예

```
public Player(int num, String name) {//1번 생성자  
    this.num = num;  
    this.name = name;  
}
```

```
<bean id="player1" name="p1,py1" class="com.test.di.Player">  
    <constructor-arg>  
        <value>31</value>  
    </constructor-arg>  
    <constructor-arg>  
        <value>정수빈</value>  
    </constructor-arg>  
</bean>
```

OR

```
<bean id="player1" name="p1,py1" class="com.test.di.Player">  
    <constructor-arg value="31"/>  
    <constructor-arg value="정수빈"/>  
</bean>
```



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Constructor 이용 (4/8)

- 2번 생성자 주입 예

```
public Player(int num, String name, int position) { // 2번 생성자  
    this.num = num;  
    this.name = name;  
    this.position = position;  
}
```

```
<bean id="player2" class="com.test.di.Player">  
    <constructor-arg>  
        <value>31</value>  
    </constructor-arg>  
    <constructor-arg>  
        <value>정수빈</value>  
    </constructor-arg>  
    <constructor-arg name="position">  
        <value>8</value>  
    </constructor-arg>  
</bean>
```

name이 아닌 index나 type 시 문제점 확인

OR

```
<bean id="player2" class="com.test.di.Player">  
    <constructor-arg value="31"/>  
    <constructor-arg value="정수빈"/>  
    <constructor-arg name="position" value="8"/>  
</bean>
```



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Constructor 이용 (5/8)

- 3번 생성자 주입 예

```
public Player(int num, String name, double record) { //3번 생성자  
    super();  
    this.num = num;  
    this.name = name;  
    this.record = record;  
}
```

```
<bean id="player3" class="com.test.di.Player">  
    <constructor-arg>  
        <value>31</value>  
    </constructor-arg>  
    <constructor-arg>  
        <value>정수빈</value>  
    </constructor-arg>  
    <constructor-arg>  
        <value>0.312</value>  
    </constructor-arg>  
</bean>
```

OR

```
<bean id="player3" class="com.test.di.Player">  
    <constructor-arg value="31"/>  
    <constructor-arg value="정수빈"/>  
    <constructor-arg value="0.312"/>  
</bean>
```



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Constructor 이용 (6/8)

- 4번 생성자 주입 예

```
public Player(int num, String name, int position, double record) {//4번 생성자
    this.num = num;
    this.name = name;
    this.position = position;
    this.record = record;
}
```

```
<bean id="player4" class="com.test.di.Player">
    <constructor-arg>
        <value>31</value>
    </constructor-arg>
    <constructor-arg>
        <value>정수빈</value>
    </constructor-arg>
    <constructor-arg>
        <value>8</value>
    </constructor-arg>
    <constructor-arg>
        <value>0.312</value>
    </constructor-arg>
</bean>
```

OR

```
<bean id="player4" class="com.test.di.Player">
    <constructor-arg value="31"/>
    <constructor-arg value="정수빈"/>
    <constructor-arg value="8"/>
    <constructor-arg value="0.312"/>
</bean>
```



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Constructor 이용 (7/8)

- 생성자의 argument의 순서를 지키지 않을 경우.(4번생성자)

```
<bean id="player5" class="com.test.di.Player">
    <constructor-arg type="int">
        <value>8</value>
    </constructor-arg>
    <constructor-arg index="0">
        <value>31</value>
    </constructor-arg>
    <constructor-arg type="double">
        <value>0.312</value>
    </constructor-arg>
    <constructor-arg name="name">
        <value>정수빈</value>
    </constructor-arg>
</bean>
```

속성(type,index, name)을  
이용하여 match시켜준다.



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Constructor 이용 (8/8)

- 주입 받는 argument가 reference인 경우.

```
public class PlayerService {  
    private PlayerDao playerDao;  
  
    public PlayerService(PlayerDao playerDao) {  
        super();  
        this.playerDao = playerDao;  
    }  
}
```

```
<bean id="dao" class="com.test.di.PlayerDao"/>  
<bean id="service1" class="com.test.di.PlayerService">  
    <constructor-arg>  
        <ref bean="dao"/>  
    </constructor-arg>  
</bean>  
  
<bean id="service2" class="com.test.di.PlayerService">  
    <constructor-arg ref="dao"/>  
</bean>
```



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Property 이용 (1/6)

- property를 통해 객체 또는 값을 주입 받는다. – setter method
  - 주의 : setter를 통해서는 하나의 값만 받을 수 있다.
- <property> : <bean>의 하위태그로 설정한 bean 객체 또는 값을 property를 통해 주입하도록 설정.
  - 설정 방법 : <ref>, <value>와 같은 **하위태그**를 이용하여 설정하거나 또는 **속성**을 이용하여 설정.
    1. 하위태그 이용
      - a. 객체 주입 시 : <ref bean="bean name"/>
      - b. 문자열(String), primitive data 주입 시 : <value>값</value>
    2. 속성 이용 : name – 값을 주입할 property 이름(setter의 이름)
      - a. 객체 주입 시 : <property name="propertname" ref="bean name"/>
      - b. 문자열(String), primitive data 주입 시 : < property name="propertname" value="값" />
    3. xml namespace를 이용하여 설정



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Property 이용 (2/6)

- 하위태그 이용
  - <ref bean="bean name"/> – 객체 주입 시
  - <value>값</value> – 문자열(String), primitive data 주입 시
- 속성 이용
  - <property name="property name" ref="bean name"/>
  - <property name="property name" value="값"/>
- XML Namespace를 이용
  - <bean> 태그의 스키마 설정에 namespace 등록 `xmlns:p="http://www.springframework.org/schema/p"`
  - Bean 설정 시 <bean> 태그이 속성으로 설정
    - 기본데이터 주입 > p:propertynname="value"
    - bean 주입 > p:propertynname-ref="bean\_id"

```
<bean id="adminService" class="com.test.hello.service2.AdminServiceImpl"  
      p:name="안효인"  
      p:memberDao-ref="memberDao"  
/>
```



### 3. Dependency Injection

#### 3-7. 스프링 빈 의존 관계설정 - xml

##### ✓ Property 이용 (3/6)

```
public void setNum(int num) {  
    this.num = num;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
public void setPosition(int position) {  
    this.position = position;  
}  
  
public void setRecord(double record) {  
    this.record = record;  
}
```

주입 받을 클래스의 property



# 3. Dependency Injection

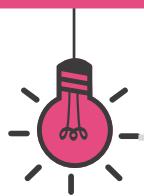
## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Property 이용 (4/6)

- Primitive data 주입

```
<bean id="player1" class="com.test.di.Player">
    <property name="num">
        <value>31</value>
    </property>
    <property name="name">
        <value>정수빈</value>
    </property>
    <property name="position" value="8"/>
    <property name="record" value="0.312"/>
</bean>
```

값 주입



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Property 이용 (5/6)

- Reference data 주입

주입 받을 객체

```
public void setPlayerDao(PlayerDao playerDao) {  
    this.playerDao = playerDao;  
}
```

reference 주입

```
<bean id="dao" class="com.test.di.PlayerDao"/>  
<bean id="service1" class="com.test.di.PlayerService">  
    <property name="playerDao">  
        <ref bean="dao"/>  
    </property>  
</bean>  
<bean id="service2" class="com.test.di.PlayerService">  
    <property name="playerDao" ref="dao"/>  
</bean>
```



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Property 이용 (6/6)

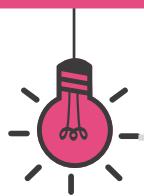
- Xml namespace

주입 받을 객체

```
public void setPlayerDao(PlayerDao playerDao) {  
    this.playerDao = playerDao;  
}  
  
public void setAge(int age) {  
    this.age = age;  
}
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xmlns:p="http://www.springframework.org/schema/p"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans ht  
  
       <bean id="dao" class="com.test.di.PlayerDao"/>  
       <bean id="service1" class="com.test.di.PlayerService"  
             p:age="30"  
             p:playerDao-ref="dao"  
       />  
   </beans>
```

XML namespace를 이용하여 주입



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Collection 계열 주입 (1/5)

- <constructor-arg> 또는 <property>의 하위 태그로 Collection값을 설정하는 태그를 이용하여 값 주입 설정
- 설정 태그

태그	Collection 종류	설명
<list>	java.util.List	List 계열 컬렉션 값 목록 전달
<set>	java.util.Set	Set계열 컬렉션 값 목록 전달
<map>	java.util.Map	Map 계열 컬렉션에 key-value의 값 목록 전달
<props>	java.util.Properties	Properties에 key(String)-value(String)의 값 목록 전달



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Collection 계열 주입 (2/5)

- <list>
  - List계열 컬렉션이나 배열에 값들을 넣기.
  - <ref>, <value> 태그를 이용해 값 설정.
  - <ref bean="bean\_id"/> : bean 객체를 list에 추가.
  - <value [type="type"]>값</value> : 문자열(String), Primitive값 list에 추가

주입 받는 객체

```
private List myList;

public void setMyList(List myList) {
    this.myList = myList;
}
```

```
<bean id="player" class="com.test.di.Player"/>
<bean id="listdi" class="com.test.di.ListDi">
    <property name="myList">
        <list>
            <value>20</value> String으로 저장됨
            <value type="java.lang.Integer">20</value> Integer로 저장됨
            <ref bean="player"/>
        </list>
    </property>
</bean>
```



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Collection 계열 주입 (3/5)

- <set>
  - Set계열에 객체 넣기.
    - 속성 : value-type > value 타입 설정.
  - <value>, <ref>를 이용해 값을 넣는다.

주입 받는 객체

```
private Set mySet;

public void setMySet(Set mySet) {
    this.mySet = mySet;
}
```

```
<bean id="player" class="com.test.di.Player"/>
<bean id="setdi" class="com.test.di.SetDi">
    <property name="mySet">
        <set>
            <value>20</value>
            <value type="java.lang.Integer">20</value>
            <ref bean="player"/>
        </set>
    </property>
</bean>
```



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Collection 계열 주입 (4/5)

- <map>
  - Map계열 컬렉션에 객체 넣기.
    - 속성 : key-type, value-type > key와 value의 타입을 고정시킬경우 사용
  - <entry>를 이용해 key-value를 map에 등록
    - 속성
      - key, key-ref : key 설정
      - value, value-ref : 값 설정

주입 받는 객체

```
private Map myMap;  
  
public void setMyMap(Map myMap) {  
    this.myMap = myMap;  
}
```

```
<bean id="player" class="com.test.di.Player"/>  
<bean id="mapdi" class="com.test.di.MapDi">  
    <property name="myMap">  
        <map>  
            <entry key="username" value="한효민"/>  
            <entry key="py" value-ref="player"/>  
        </map>  
    </property>  
</bean>
```



# 3. Dependency Injection

## 3-7. 스프링 빈 의존 관계설정 - xml

### ✓ Collection 계열 주입 (5/5)

- <props>
  - java.util.Properties에 값(문자열) 넣기.
  - <prop>를 이용해 key-value를 properties에 등록
    - 속성
      - key : key 설정
      - 같은 태그 사이에 넣는다. : <prop key="username">안효인</prop>

```
private Properties dbInfo;

public void setDbInfo(Properties dbInfo) {
    this.dbInfo = dbInfo;
}
```

주입 받는 객체

```
<bean id="propdi" class="com.test.di.PropertiesDi">
    <property name="dbInfo">
        <props>
            <prop key="driver">oracle.jdbc.driver.OracleDriver</prop>
            <prop key="url">jdbc:oracle:thin:@localhost:1521:orcl</prop>
            <prop key="dbid">test</prop>
            <prop key="dbpass">test</prop>
        </props>
    </property>
</bean>
```



### 3. Dependency Injection

#### 3-8. 스프링 빈 의존 관계설정 - annotation

##### ✓ Annotation

- 멤버변수에 직접 정의 하는 경우 setter method를 만들지 않아도 됨.

###### annotation

###### 설명

@Resource

Spring 2.5부터 지원.  
멤버변수, setter method에 사용 가능.  
**타입**에 맞춰서 연결.

@Autowired

Spring 2.5부터 지원.  
**Spring에서만 사용 가능.**  
Required 속성을 통해 DI여부 조정.  
멤버변수, setter, constructor, 일반 method 사용 가능.  
**타입**에 맞춰서 연결.

@Inject

Spring 3.0부터 지원.  
Framework에 종속적이지 않음  
Javax.inject-x.x.x.jar 필요

멤버변수, setter, constructor, 일반 method 사용 가능.  
**이름**으로 연결.



# 3. Dependency Injection

## 3-8. 스프링 빈 의존 관계설정 - annotation

- ✓ 특정 Bean의 기능 수행을 위해 다른 Bean을 참조해야 하는 경우 사용한다.
- ✓ **@Autowired**
  - Spring Framework에서 지원하는 Dependency 정의 용도의 Annotation으로, Spring Framework에 종속적이긴 하지만 정밀한 Dependency Injection이 필요한 경우에 유용하다.
- ✓ **@Resource**
  - JSR-250 표준 Annotation으로 Spring Framework 2.5.\* 부터 지원하는 Annotation이다. @Resource는 JNDI 리소스(datasource, java messaging service destination or environment entry)와 연관 지어 생각할 수 있으며, 특정 Bean이 JNDI 리소스에 대한 Injection을 필요로 하는 경우에는 @Resource를 사용할 것을 권장한다.
- ✓ **@Inject**
  - JSR-330 표준 Annotation으로 Spring 3부터 지원하는 Annotation이다. 특정 Framework에 종속되지 않은 애플리케이션을 구성하기 위해서는 @Inject를 사용할 것을 권장한다. @Inject를 사용하기 위해서는 클래스 패스 내에 JSR-330 라이브러리인 javax.inject-x.x.x.jar 파일이 추가되어야 함에 유의해야 한다.



### 3. Dependency Injection

#### 3-8. 스프링 빈 의존 관계설정 - annotation

- ✓ 멤버변수에 @Resource

```
@Component
public class MemberServiceImpl implements MemberService {

    @Resource(name="mDao") //동일한 타입의 빈이 여러 개일 경우 name을 통해서 구분.
    MemberDao memberDao;

    @Override
    public int registerMember(MemberDto memberDto) {
        return memberDao.registerMember(memberDto);
    }

}
```



### 3. Dependency Injection

#### 3-8. 스프링 빈 의존 관계설정 - annotation

- ✓ setter method에 @Resource

```
@Component
public class MemberServiceImpl implements MemberService {

    MemberDao memberDao;

    @Resource(name="mdao")
    public void setMemberDao(MemberDao memberDao) {
        this.memberDao = memberDao;
    }

    @Override
    public int registerMember(MemberDto memberDto) {
        return memberDao.registerMember(memberDto);
    }
}
```



# 3. Dependency Injection

## 3-8. 스프링 빈 의존 관계설정 - annotation

### ✓ 생성자에 @Autowired

- 동일한 타입의 bean이 여러 개일 경우에는 @Qualifier("name")으로 식별한다.

```
@Component
public class MemberServiceImpl implements MemberService {

    MemberDao memberDao;
    AdminDao adminDao;

    public MemberServiceImpl() {}

    @Autowired // 생성자가 여러 개 일 경우에는 하나만 사용 가능.
    public MemberServiceImpl(@Qualifier("mdao") MemberDao memberDao,
                           @Qualifier("adao") AdminDao adminDao) {
        super();
        this.memberDao = memberDao;
        this.adminDao = adminDao;
    }

    @Override
    public int registerMember(MemberDto memberDto) {
        return memberDao.registerMember(memberDto);
    }

}
```



# 3. Dependency Injection

## 3-8. 스프링 빈 의존 관계설정 - annotation

### ✓ constructor에 @Autowired

- 동일한 타입의 bean이 여러 개일 경우에는 @Qualifier("name")으로 식별한다.

```
@Component
public class MemberServiceImpl implements MemberService {

    @Autowired
    @Qualifier("mdao") //동일 타입이 여러 개 일 경우.
    MemberDao memberDao;

    @Autowired
    @Qualifier("adao")
    AdminDao adminDao;

    public MemberServiceImpl() {}

    @Override
    public int registerMember(MemberDto memberDto) {
        return memberDao.registerMember(memberDto);
    }

}
```



# 3. Dependency Injection

## 3-8. 스프링 빈 의존 관계설정 - annotation

### ✓ 일반 method에 @Autowired

- 동일한 타입의 bean이 여러 개일 경우에는 @Qualifier("name")으로 식별한다.

```
@Component
public class MemberServiceImpl implements MemberService {

    MemberDao memberDao;
    AdminDao adminDao;

    public MemberServiceImpl() {}

    @Autowired
    public void initService(@Qualifier("mdao") MemberDao memberDao,
                           @Qualifier("adao") AdminDao adminDao) {
        this.memberDao = memberDao;
        this.adminDao = adminDao;
    }

    @Override
    public int registerMember(MemberDto memberDto) {
        return memberDao.registerMember(memberDto);
    }
}
```



# 3. Dependency Injection

## 3-9. 기타 설정

### ✓ 빈 객체의 생성단위

- BeanFactory를 통해 Bean을 요청 시 객체생성의 범위(단위)를 설정
- <bean>의 scope 속성을 이용해 설정
  - scope의 값

범위	설명
<b>singleton</b>	스프링 컨테이너당 하나의 인스턴스 빈만 생성(default).
<b>prototype</b>	컨테이너에 빈을 요청할 때마다 새로운 인스턴스 생성.
<b>request</b>	HTTP Request별로 새로운 인스턴스 생성.
<b>session</b>	HTTP Session별로 새로운 인스턴스 생성.

- request, session은 WebApplicationContext에서만 적용 가능.



# 3. Dependency Injection

## 3-9. 기타 설정

### ✓ singleton & prototype

```
<bean id="memberService" class="com.test.hello.service2.MemberServiceImpl" scope="singleton"/>  
<bean id="memberService" class="com.test.hello.service2.MemberServiceImpl" scope="prototype"/>
```

- Singleton은 spring application context에서 getBean()을 이용하여 bean 객체를 반환 받을 때 **동일한** instance를 반환 함.
- prototype 은 spring application context에서 getBean()을 이용하여 bean 객체를 반환 받을 때 **새로운** instance를 반환 함.



# 3. Dependency Injection

## 3-9. 기타 설정

### ✓ Factory Method로부터 빈(bean) 생성

- getBean()으로 호출 시 private 생성자도 호출 하여 객체를 생성한다.

그러므로 위의 상황에서 factory method만 호출 해야 객체를 얻을 수 있는 것은 아니다.

```
public class DBConnection {  
  
    private static DBConnection instance;  
  
    private DBConnection() {}  
  
    public static DBConnection getInstance() {  
        if(instance == null)  
            instance = new DBConnection();  
        return instance;  
    }  
}
```

Singleton class는 static factory method를 통해서 instance 생성 가능.  
단, 하나의 instance만 생성함

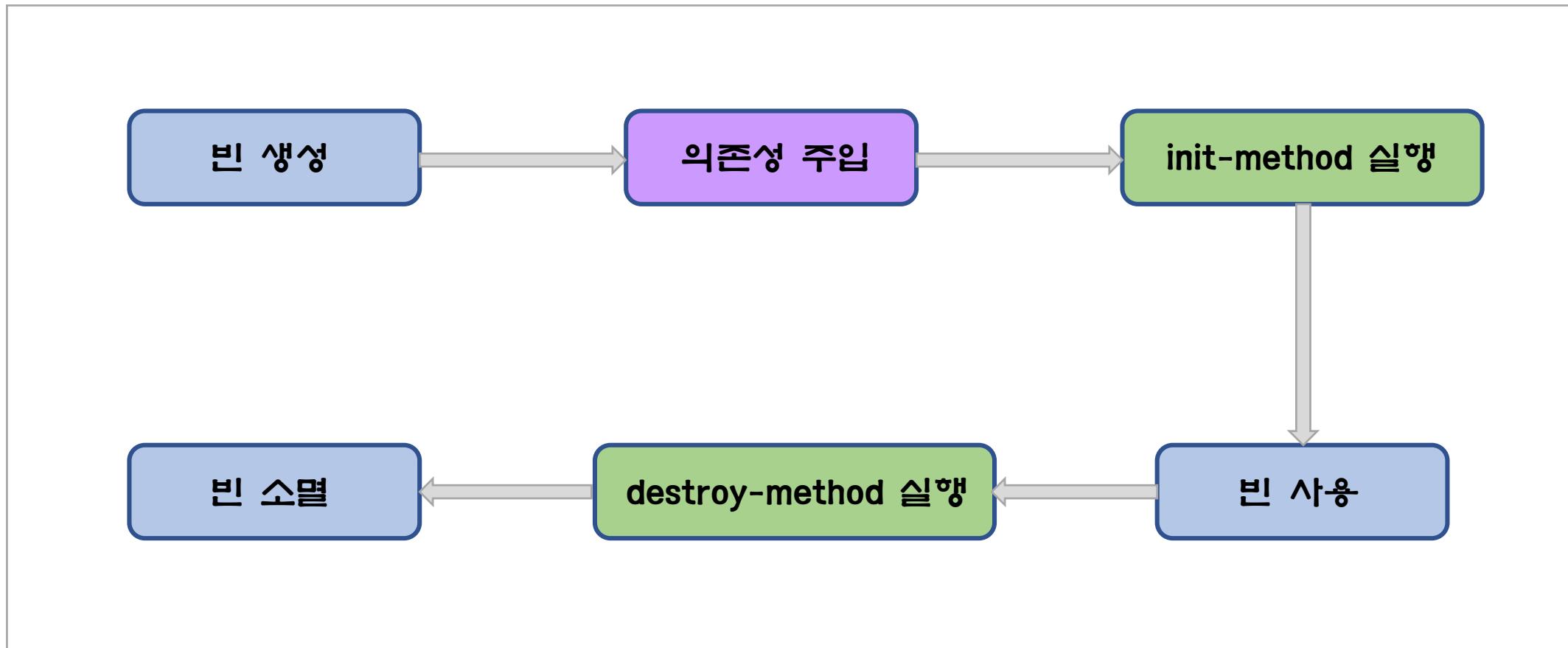
```
<bean id="dbc" class="com.test.di.DBConnection" factory-method="getInstance"/>
```



### 3. Dependency Injection

#### 3-10. 스프링 빈의 생명 주기(LifeCycle)

##### ✓ LifeCycle

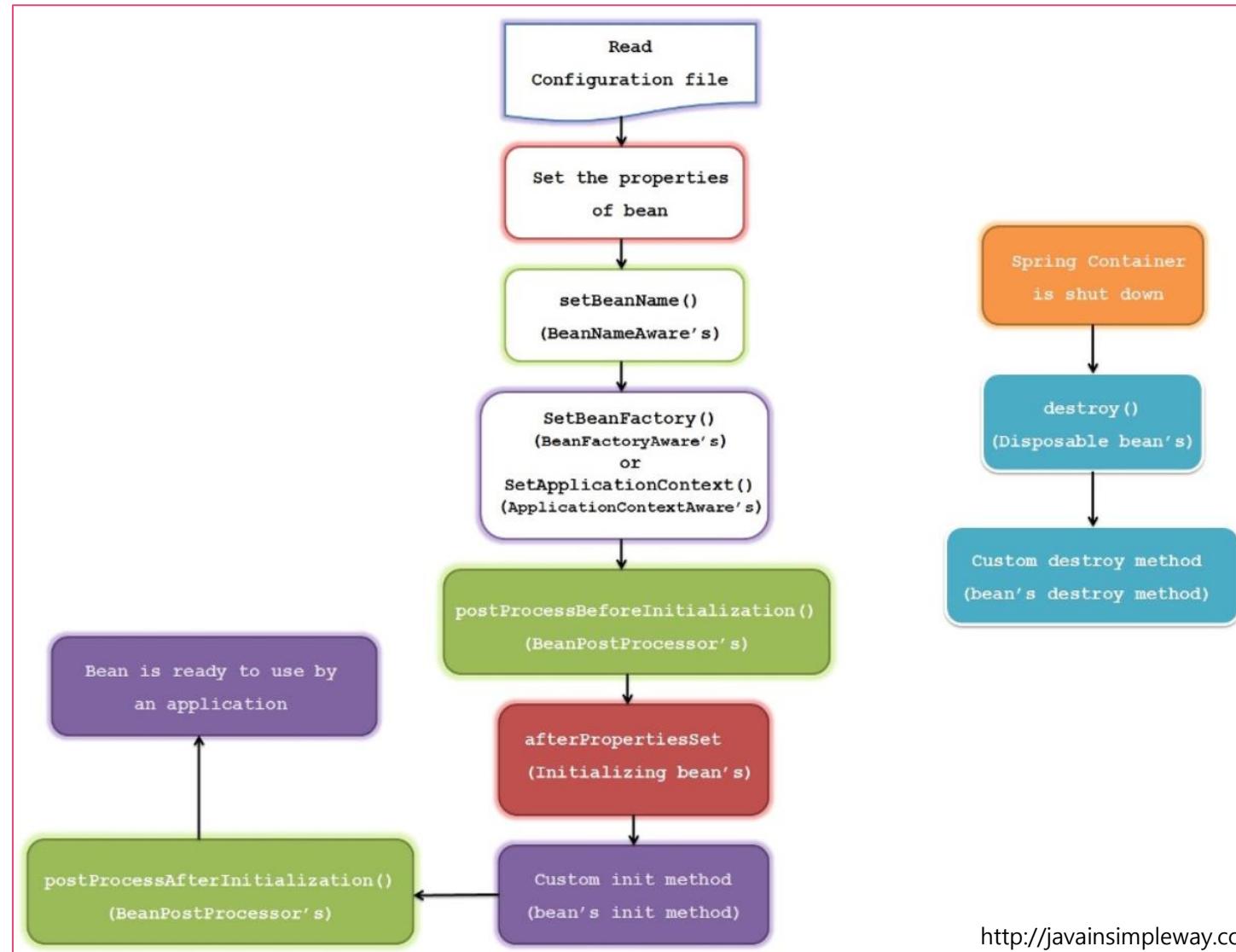




# 3. Dependency Injection

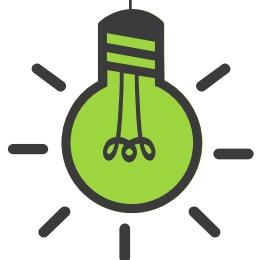
## 3-10. 스프링 빈의 생명 주기(LifeCycle)

### ✓ LifeCycle



04

관점 지향 프로그래밍  
(AOP,  
Aspect Oriented Programming)





## 4. Aspect Oriented Programming

### 4-1. AOP개요와 용어

#### ✓ 핵심 관심 사항과 공통(부가) 관심 사항.

- 핵심 관심 사항(core concern)과 공통 관심 사항(cross-cutting concern)
- 기존 OOP에서는 공통관심사항을 여러 모듈에서 적용하는데 있어 중복된 코드를 양상 하는 한계가 존재함.
- 이를 해결하기 위해 AOP가 등장.
- Aspect Oriented Programming은 문제를 해결하기 위한 핵심 관심 사항과 전체에 적용되는 공통 관심 사항을 기준으로 프로그래밍함으로써 공통 모듈을 손쉽게 적용할 수 있게 함.



## 4. Aspect Oriented Programming

### 4-1. AOP개요와 용어

#### ✓ **AOP(Aspect Oriented Programming) 개요.**

- AOP는 application에서의 관심사의 분리(기능의 분리) 즉, 핵심적인 기능에서 부가적인 기능을 분리한다.
- 분리한 부가기능을 어스펙트(Aspect)라는 독특한 모듈형태로 만들어서 설계하고 개발하는 방법.
- OOP를 적용하여도 핵심기능에서 부가기능을 쉽게 분리된 모듈로 작성하기 어려운 문제점을 AOP가 해결.
- AOP는 부가기능을 어스펙트(Aspect)로 정의하여, 핵심기능에서 부가기능을 분리함으로써 핵심기능을 설계하고 구현할 때 객체지향적인 가치를 지킬 수 있도록 도와주는 개념.



# 4. Aspect Oriented Programming

## 4-1. AOP개요와 용어

### ✓ AOP 적용 예 (1/2)

- 간단한 메소드의 성능 검사
  - 개발 도중 특히 DB에 다향의 데이터를 넣고 빼는 등의 배치 작업에 대하여 시간을 측정해 보고 쿼리를 개선하는 작업은 매우 의미가 있다. 이 경우 매번 해당 메소드의 처음과 끝에 System.currentTimeMillis();를 사용하거나, 스프링이 제공하는 StopWatch코드를 사용하기는 매우 번거롭다.
  - 이런 경우 해당 작업을 하는 코드를 밖에서 설정하고 해당 부분을 사용하는 것이 편리하다.
- 트랜잭션 처리
  - 트랜잭션의 경우 비즈니스 로직의 전후에 설정된다.
  - 하지만 매번 사용하는 트랜잭션(try{ ~~~ }catch{}부분)의 코드는 번거롭고, 소스를 더욱 복잡하게 보여준다.



# 4. Aspect Oriented Programming

## 4-1. AOP개요와 용어

### ✓ AOP 적용 예 (2/2)

- 예외반환
  - 스프링에는 `DataAccessException`이라는 매우 잘 정의되어 있는 예외 계층 구조가 있다.  
예전 하이버네이트 예외들은 몇 개 없었고 그나마도 `UncatchedException`이 아니었다.  
이렇게 구조가 별로 안 좋은 예외들이 발생했을 때, 그걸 잡아서 잘 정의되어있는 예외 계층 구조로 변환해서 다시  
던지는 Aspect는 제 3의 프레임워크를 사용할 때, 본인의 프레임워크나 애플리케이션에서 별도의 예외 계층 구조로  
변환하고 싶을 때 유용하다.
- 아키택처 검증
- 기타
  - 하이버네이트와 JDBC를 같이 사용할 경우, DB 동기화 문제 해결
  - 멀티쓰레드 Safety 관련하여 작업해야 하는 경우, 메소드들에 일괄적으로 락을 설정하는 Aspect
  - 데드락등으로 인한 `PessimisticLockingFailureException`등의 예외를 만났을 때 재시도하는 Aspect
  - 로깅, 인증, 권한등..

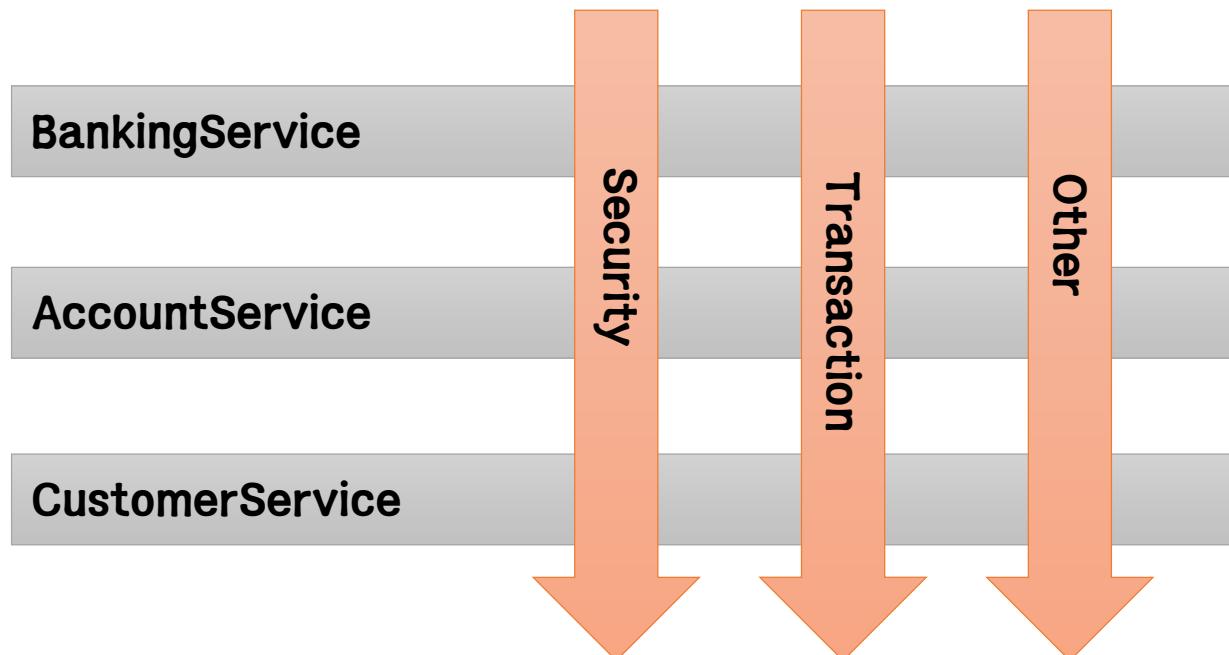


# 4. Aspect Oriented Programming

## 4-1. AOP개요와 용어

### ✓ AOP 구조

- 핵심 관심 사항 : BankingService, AccountService, CustomerService
- 공통 관심 사항 : Security, Transaction, Other



- 핵심 관심 사항에 공통 관심 사항을 어떻게 적용시킬 것인가...



# 4. Aspect Oriented Programming

## 4-1. AOP개요와 용어

### ✓ Spring AOP 용어 (1/3)

- Target
  - 핵심기능을 담고 있는 모듈로, 타겟은 부가기능을 부여할 대상이 됨.
- Advice
  - 어느 시점(ex : method 수행 전/후, 예외발생 후등.)에 어떤 공통 관심 기능(Aspect)을 적용할 지 정의 한 것. 타겟에 제공할 부가기능을 담고 있는 모듈.
- JoinPoint
  - Aspect가 적용 될 수 있는 지점 (ex : method, field)
  - 즉 타겟 객체가 구현한 인터페이스의 모든 method는 JoinPoint가 됨.



# 4. Aspect Oriented Programming

## 4-1. AOP개요와 용어

### ✓ Spring AOP 용어 (2/3)

- Pointcut
  - 공통 관심 사항이 적용될 JoinPoint.
  - Advice를 적용할 타겟의 method를 선별하는 정규표현식.
  - Pointcut 표현식은 execution으로 시작하고, method의 Signature를 비교하는 방법을 주로 이용.
- Aspect
  - 여러 객체에서 공통으로 적용되는 공통 관심 사항 (transaction, logging, security..)
  - AOP의 기본 모듈.
  - **Aspect = Advice + Pointcut**
  - Aspect는 Singleton 형태의 객체로 존재.



# 4. Aspect Oriented Programming

## 4-1. AOP개요와 용어

### ✓ Spring AOP 용어 (3/3)

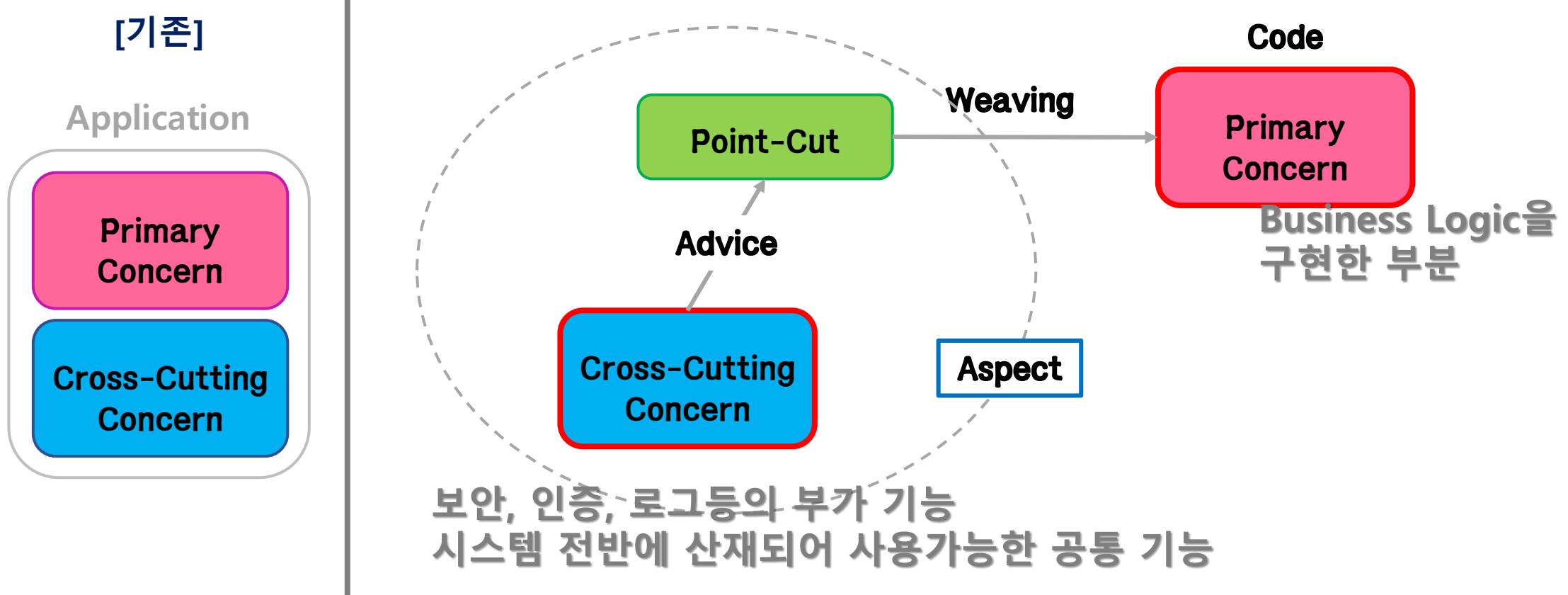
- Advisor
  - Advisor = Advice + Pointcut
  - Advisor는 Spring AOP에서만 사용되는 특별한 용어.
- Weaving
  - 어떤 Advice를 어떤 Pointcut(핵심사항)에 적용시킬 것인지에 대한 설정(Advisor)
  - 즉 Pointcut에 의해서 결정된 타겟의 Joinpoint에 부가기능(Advice)을 삽입하는 과정을 뜻함.
  - Weaving은 AOP의 핵심기능(Target)의 코드에 영향을 주지 않으면서 필요한 부가기능(Advice)을 추가 할 수 있도록 해주는 핵심적인 처리과정.



# 4. Aspect Oriented Programming

## 4-1. AOP개요와 용어

### ✓ AOP 비교



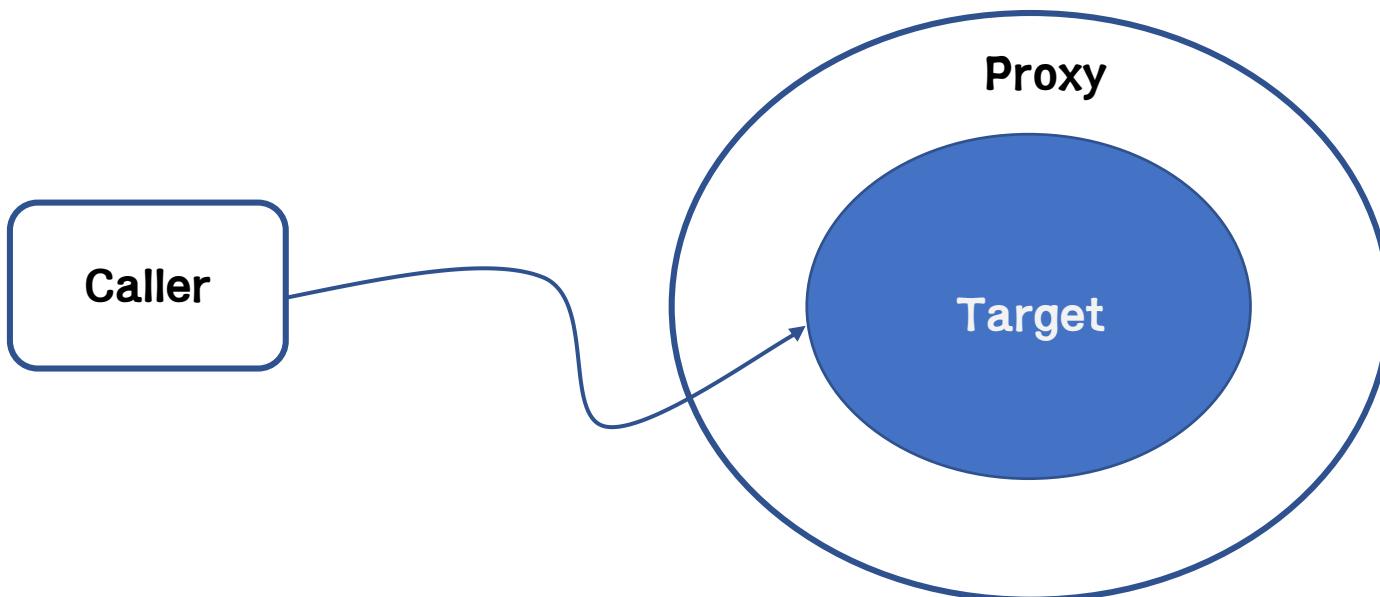


## 4. Aspect Oriented Programming

### 4-2. Spring AOP 특징 (1/3)

- ✓ **Spring은 프록시(Proxy) 기반 AOP를 지원.**

- Spring은 Target 객체에 대한 Proxy를 만들어 제공.
- Target을 감싸는 Proxy는 실행시간(Runtime)에 생성.
- Proxy는 Advice를 Target 객체에 적용하면서 생성되는 객체.



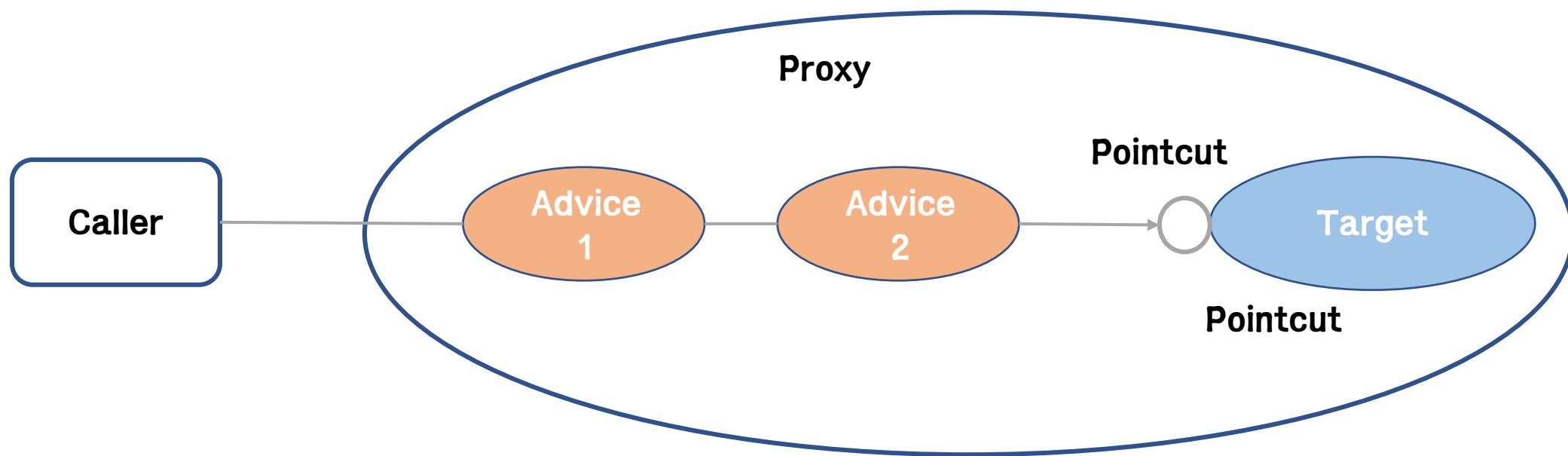


# 4. Aspect Oriented Programming

## 4-2. Spring AOP 특징 (2/3)

### ✓ 프록시(Proxy)가 호출을 가로챈다(Intercept).

- Proxy는 Target 객체에 대한 호출을 가로챈 다음 Advice의 부가기능 로직을 수행하고 난 후에 Target의 핵심 기능 로직을 호출한다.(전처리 Advice)
- 또는 Target의 핵심 기능 로직 method를 호출한 후에 부가기능(Advice)을 수행하는 경우도 있다.(후처리 Advice)





## 4. Aspect Oriented Programming

### 4-2. Spring AOP 특징 (3/3)

- ✓ **Spring AOP는 method JoinPoint만 지원.**

- Spring은 동적 Proxy를 기반으로 AOP를 구현하므로 method JoinPoint만 지원한다.
- 즉, 핵심기능(Target)의 method가 호출되는 런타임 시점에만 부가기능(Advice)을 적용할 수 있다.
- 반면 AspectJ 같은 고급 AOP framework를 사용하면 객체의 생성, 필드값의 조회와 조작, static method 호출 및 초기화 등의 다양한 작업에 부가기능을 적용할 수 있다.



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ AOP 구현방법

- POJO Class를 이용한 AOP 구현
- Spring API를 이용한 AOP 구현
- 어노테이션(Annotation)을 이용한 AOP 구현



## 4. Aspect Oriented Programming

### 4-3. Spring AOP 구현

#### ✓ POJO 기반 AOP 구현

- XML Schema 확장기법을 통해 설정파일을 작성.
- POJO 기반 Advice Class 작성.



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – 설정 파일 (1/5)

- XML Schema를 이용한 AOP 설정.
  - aop namespace와 XML schema 추가.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

</beans>
```



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – 설정 파일 (2/5)

- XML Schema를 이용한 AOP 설정.
  - aop namespace를 이용한 설정.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

    <bean id="Logging" class="com.test.aop.LoggingTest"/>

    <aop:config>
        <aop:pointcut id="Logmethod" expression="execution(public * com.test.aop..*(..))"/>
        <aop:aspect id="logAspect" ref="Logging">
            <aop:around pointcut-ref="Logmethod" method="pringLog"/>
        </aop:aspect>
    </aop:config>

</beans>
```



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – 설정 파일 (3/5)

- AOP 설정 태그.

Tag	설명
<code>&lt;aop:config&gt;</code>	aop 설정의 root 태그 (weaving들의 묶음.)
<code>&lt;aop:aspect&gt;</code>	Aspect 설정 (하나의 weaving에 대한 설정.)
<code>&lt;aop:pointcut&gt;</code>	Pointcut 설정.

- Advice 설정태그

Tag	설명
<code>&lt;aop:before&gt;</code>	method 실행 전 실행 될 Advice.
<code>&lt;aop:after-returning&gt;</code>	method가 정상 실행 후 실행 될 Advice.
<code>&lt;aop:after-throwing&gt;</code>	method에서 예외 발생시 실행 될 Advice. (catch block)
<code>&lt;aop:after&gt;</code>	method가 정상 또는 예외 발생에 상관없이 실행 될 Advice (finally block)
<code>&lt;aop:around&gt;</code>	모든 시점(실행 전,후)에서 적용시킬 수 있는 Advice



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – 설정 파일 <aop:aspect> (4/5)

- 한 개의 Aspect(공통 관심 기능)을 설정.
- ref 속성을 통해 공통기능을 가지고 있는 bean을 연결.
- id는 이 태그의 식별자 설정.
- 자식 태그로 <aop:pointcut> advice관련 태그가 올 수 있다.

```
<bean id="logging" class="com.test.aop.LoggingTest"/>

<aop:config>
    <aop:aspect id="LogAspect" ref="logging">
        <aop:pointcut id="Logmethod" expression="execution(public * com.test.aop..*(..))"/>
        <aop:around pointcut-ref="Logmethod" method="pringlog"/>
    </aop:aspect>
</aop:config>
```



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – 설정 파일 <aop:pointcut> (5/5)

- Pointcut(공통 기능이 적용될 곳)을 지정하는 태그.
- <aop:config>나 <aop:aspect>의 자식 태그.
  - <aop:config> 전역적으로 사용
  - <aop:aspect> 내부에서 사용
- AspectJ 표현식을 통해 pointcut 지정
- 속성 :
  - id : 식별자로 advice 태그에서 사용됨
  - expression : pointcut 지정

```
<aop:pointcut id="logmethod" expression="execution(public * com.test.aop..*(..))"/>
```

```
<bean id="Logging" class="com.test.aop.LoggingTest"/>

<aop:config>
    <aop:aspect id="LogAspect" ref="Logging">
        <aop:pointcut id="logmethod" expression="execution(public * com.test.aop..*(..))"/>
        <aop:around pointcut-ref="logmethod" method="pringlog"/>
    </aop:aspect>
</aop:config>
```



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – AspectJ 표현식 (1/4)

- AspectJ에서 지원하는 패턴 표현식
- Spring은 method 호출 관련 명시자만 지원

**명시자(제한자패턴? 리턴타입패턴? 이름패턴(파라미터패턴))**

? : 생략가능

- 명시자
  - execution : 실행시킬 method 패턴을 직접 입력하는 경우.
  - within : method가 아닌 특정 타입에 속하는 method들을 설정할 경우.
  - bean : 2.5버전에서 추가됨. 설정파일에 지정된 빈의 이름(name 속성)을 이용해 pointcut 설정



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – AspectJ 표현식 (2/4)

- 표현 : 예) execution(public \* a.b..\*Service.get\*(..))

**명시자(제한자패턴? 리턴타입패턴? 이름패턴(파라미터패턴))**

- 제한자 패턴에는 public, protected 또는 생략

- ＊ : 1개의 모든 값을 표현

- argument에서 쓰인 경우 : 1개의 argument

- package에서 쓰인 경우 : 1개의 하위 package

- .. : 0개 이상의 모든 값을 표현

- argument에서 쓰인 경우 : 0개 이상의 argument

- package에서 쓰인 경우 : 0개 이상의 하위 package

- 위 예 설명

적용 하려는 method들의 패턴은 public 제한자를 가지며 리턴 타입에는 모든 타입이 올 수 있다. 이름은 a.b로 시작하는 package와 그 하위 패키지에 있는 모든 클래스 중 Service로 끝나는 class중에서 get으로 시작하는 method이며 argument는 0개 이상 오며 반환 타입은 상관없다.



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – AspectJ 표현식 (3/4)

- Example (1/2)

Pointcut	선택된 Joinpoints
execution(public * *(..))	public 메소드 실행
execution(* set*(..))	이름이 set으로 시작하는 모든 메소드명 실행
execution(* set*(..))	이름이 set으로 시작하는 모든 메소드명 실행
execution(* com.test.service.AccountService.*(..))	AccountService 인터페이스의 모든 메소드 실행
execution(* com.test.service.*.*(..))	service 패키지의 모든 메소드 실행
execution(* com.test.service..*.*(..))	service 패키지와 하위 패키지의 모든 메소드 실행
within(com.test.service.*)	service 패키지 내의 모든 결합점
within(com.test.service..*)	service 패키지 및 하위 패키지의 모든 결합점
this(com.test.service.AccountService)	AccountService 인터페이스를 구현하는 프록시 개체의 모든 결합점
target(com.test.service.AccountService)	AccountService 인터페이스를 구현하는 대상 객체의 모든 결합점
args(java.io.Serializable)	하나의 파라미터를 갖고 전달된 인자가 Serializable인 모든 결합점



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – AspectJ 표현식 (4/4)

- Example (2/2)

Pointcut	선택된 Joinpoints
@target(org.springframework.transaction.annotation.Transactional)	대상 객체가 @Transactional 어노테이션을 갖는 모든 결합점
@within(org.springframework.transaction.annotation.Transactional)	대상 객체의 선언 타입이 @Transactional 어노테이션을 갖는 모든 결합점
@annotation(org.springframework.transaction.annotation.Transactional)	실행 메소드가 @Transactional 어노테이션을 갖는 모든 결합점
@args(com.test.security.Classified)	단일 파라미터를 받고, 전달된 인자 타입이 @Classified 어노테이션을 갖는 모든 결합점
bean(accountRepository)	"accountRepository" 빈
!bean(accountRepository)	"accountRepository" 빈을 제외한 모든 빈
bean(*)	모든 빈
bean(account*)	이름이 'account'로 시작되는 모든 빈
bean(*Repository)	이름이 "Repository"로 끝나는 모든 빈
bean(accounting/*)	이름이 "accounting/"로 시작하는 모든 빈
bean(*dataSource)    bean(*DataSource)	이름이 "dataSource" 나 "DataSource" 으로 끝나는 모든 빈



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현

- XML Schema 확장기법을 통해 설정파일을 작성.
- POJO 기반 Advice Class 작성.



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice 작성

- POJO 기반 Aspect Class 작성
  - 설정 파일의 advice 관련 태그에 맞게 작성한다.
  - <bean>으로 등록 하며 <aop:aspect>의 ref 속성으로 참조한다.
  - 공통 기능 메소드 : advice 관련 태그들의 method 속성의 값이 method의 이름이 된다.



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice 정의 관련 태그

- 속성
  - pointcut : 직접 pointcut을 설정. 호출 할 method의 패전 지정.
  - pointcut-ref : <aop:pointcut> 태그의 id명을 넣어 pointcut 지정
  - method : Aspect bean에서 호출할 method명 지정

```
<bean id="logging" class="com.test.aop.LoggingTest"/>

<aop:config>
    <aop:aspect id="LogAspect" ref="Logging">
        <aop:pointcut id="Logmethod" expression="execution(public * com.test.aop..*(..))"/>
        <aop:around pointcut-ref="Logmethod" method="pringlog"/>
    </aop:aspect>
</aop:config>
```





# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice Class.

- POJO 기반의 Class로 작성.
  - class명이나 method명에 대한 제한은 없다.
  - method 구문은 호출되는 시점에 따라 달라 질 수 있다.
  - method의 이름은 advice 태그(<aop:before/>)에서 method 속성의 값이 method명이 된다.



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice 종류 (1/5)

- **Before Advice**

- 대상 객체의 메소드가 실행되기 전에 실행됨.
- return type : 리턴 값을 갖더라도 실제 Advice의 적용과정에서 사용되지 않기 때문에 void을 쓴다.
- argument : 없거나 대상객체 및 호출되는 메소드에 대한 정보 또는 파라미터에 대한 정보가 필요하다면 JoinPoint 객체를 전달.

```
<aop:config>
    <aop:aspect id="beforeAspect" ref="userCheckAdvice">
        <aop:pointcut id="publicMethod" expression="execution(public * com.test.spring.aop..*Controller.*(..))"/>
        <aop:before method="before" pointcut-ref="publicMethod"/>
    </aop:aspect>
</aop:config>
```

```
public void before(JoinPoint joinPoint) {
    String name = joinPoint.getSignature().toShortString();
    System.out.println("Advice : " + name);
}
```

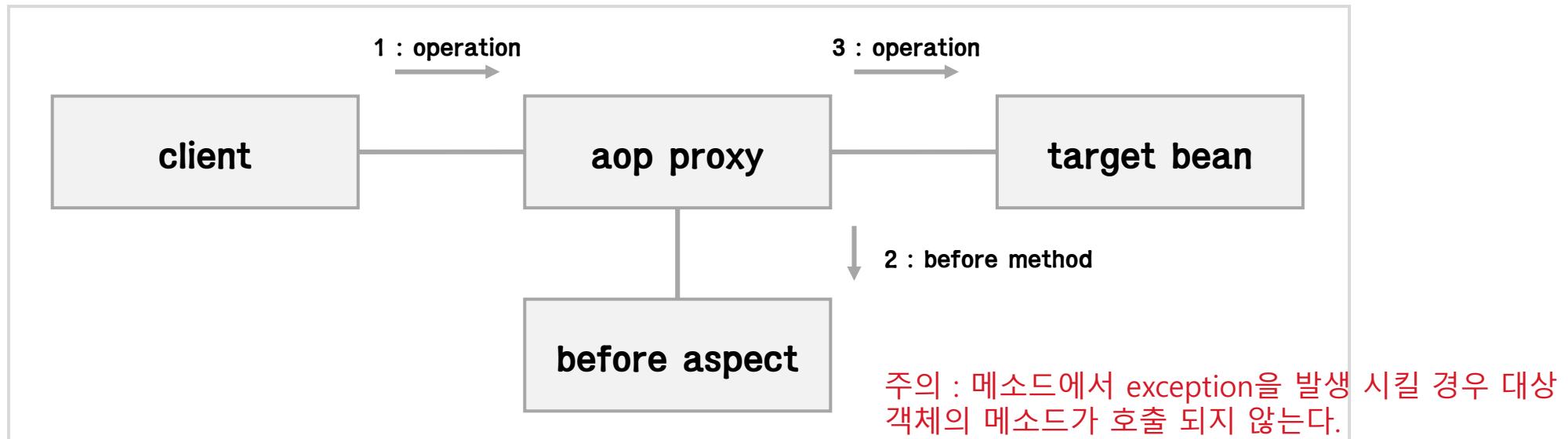


# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice 종류 (1/5)

- Before Advice 실행순서.



1. 빈 객체를 사용하는 코드에서 스프링이 생성한 AOP 프록시의 메소드를 호출.
2. AOP 프록시는 <aop:before>에서 지정한 메소드를 호출.
3. AOP 프록시는 Aspect 기능 실행 후 실제 빈 객체의 메소드를 호출.



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice 종류 (2/5)

- **After Returning Advice**

- 대상 객체의 method 실행이 정상적으로 끝난 뒤 실행됨.
- return type : void
- argument :
  - 없거나 org.aspectj.lang.JoinPoint 객체를 받는다. JoinPoint는 항상 첫 argument로 사용.
  - 대상 method에서 반환되는 특정 객체 타입의 값을 argument로 받을 수 있다.

```
<aop:config>
    <aop:aspect id="afterAspect" ref="historyAdvice">
        <aop:pointcut id="publicMethod" expression="execution(public * com.test.spring.aop..*Controller.*(..))"/>
        <!-- 메소드가 정상적으로 결과값을 리턴했을 경우. -->
        <aop:after-returning method="history" pointcut-ref="publicMethod" returning="ret"/>
    </aop:aspect>
</aop:config>
```

```
public void history(Object ret) throws Throwable{
    System.out.println("HistoryAdvice : " + ret);
}
```

```
public void history(Joinpoint joinPoint, Object ret) throws Throwable{
    System.out.println("HistoryAdvice : " + ret);
}
```

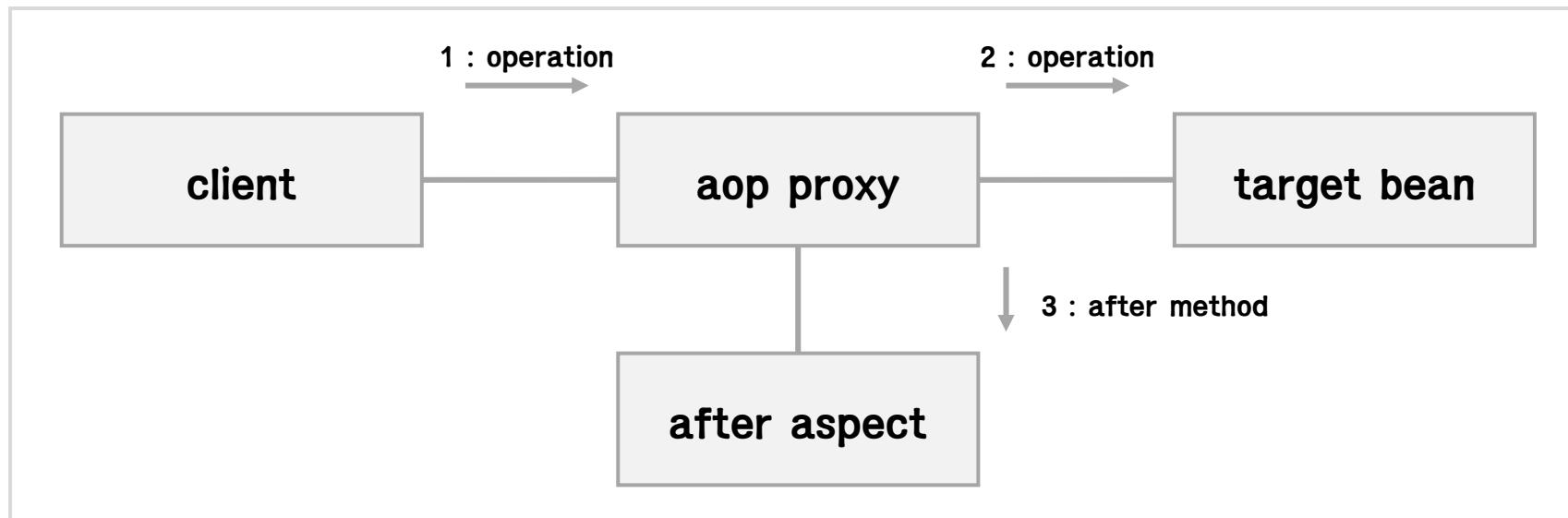


# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice 종류 (2/5)

- After Returning Advice 실행순서.



1. 빈 객체를 사용하는 코드에서 스프링이 생성한 AOP 프록시의 메소드를 호출.
2. AOP 프록시는 실제 빈 객체의 메소드를 호출(정상 실행).
3. AOP 프록시는 .<aop:after-returning>에서 지정한 메소드를 호출.



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice 종류 (3/5)

- **After Throwing Advice**

- 대상 객체의 method 실행 중 예외가 발생한 경우 실행됨.
- return type : void
- argument :
  - 없거나 JoinPoint 객체를 받는다. JoinPoint는 항상 첫 argument로 사용.
  - 대상 method에서 전달되는 예외객체를 argument로 받을 수 있다.

```
<aop:config>
    <aop:aspect id="exceptionAspect" ref="exceptionAdvice">
        <aop:pointcut id="publicMethod" expression="execution(public * com.test.spring.aop..*Controller.*(..))"/>
        <!-- exception이 발생했을경우 호출 -->
        <aop:after-throwing method="exceptionAfter" pointcut-ref="publicMethod"/>
    </aop:aspect>
</aop:config>
```

```
public void exceptionAfter(Exception ex){
    System.out.println("ExceptionAdvice");
}
```

```
public void exceptionAfter(Joinpoint joinPoint, Exception ex){
    System.out.println("ExceptionAdvice : " + ex);
}
```

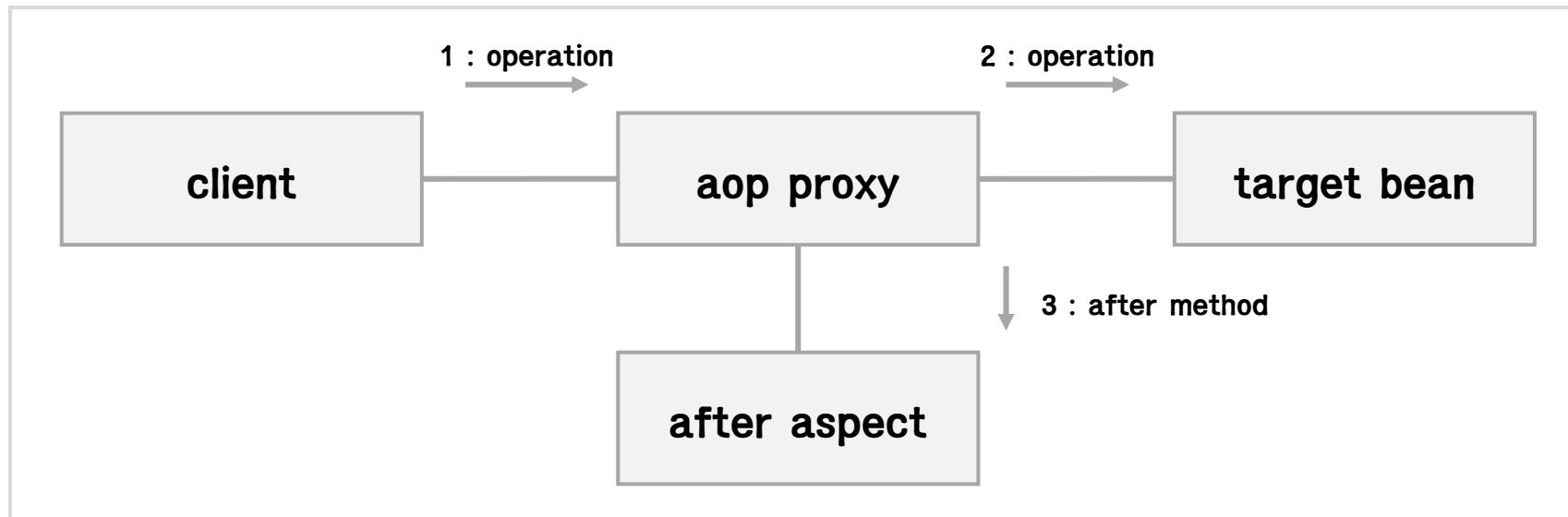


# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice 종류 (3/5)

- After Throwing Advice 실행순서.



1. 빈 객체를 사용하는 코드에서 스프링이 생성한 AOP 프록시의 메소드를 호출.
2. AOP 프록시는 실제 빈 객체의 메소드를 호출(exception 발생).
3. AOP 프록시는 .<aop:after-throwing>에서 지정한 메소드를 호출.



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice 종류 (4/5)

- **After Advice**

- 대상 객체의 method가 정상적으로 실행 되었는지 아니면 exception을 발생 시켰는 지의 여부와 상관 없이 메소드 실행 종료 후 공통 기능 적용.
- return type : void
- argument :
  - 없거나 JoinPoint 객체를 받는다. JoinPoint는 항상 첫 argument로 사용.

```
<aop:config>
    <aop:aspect id="afterAspect" ref="countAdvice">
        <aop:pointcut id="publicMethod" expression="execution(public * com.test.spring.aop..*Controller.*(..))"/>
        <!-- 정상 실행 또는 exception 발생 모두 호출 finally와 비슷 -->
        <aop:after method="after" pointcut-ref="publicMethod"/>
    </aop:aspect>
</aop:config>
```

```
public void after(JoinPoint joinPoint) {
    String name = joinPoint.toShortString();
    System.out.println("CountAdvice : " + name);
}
```

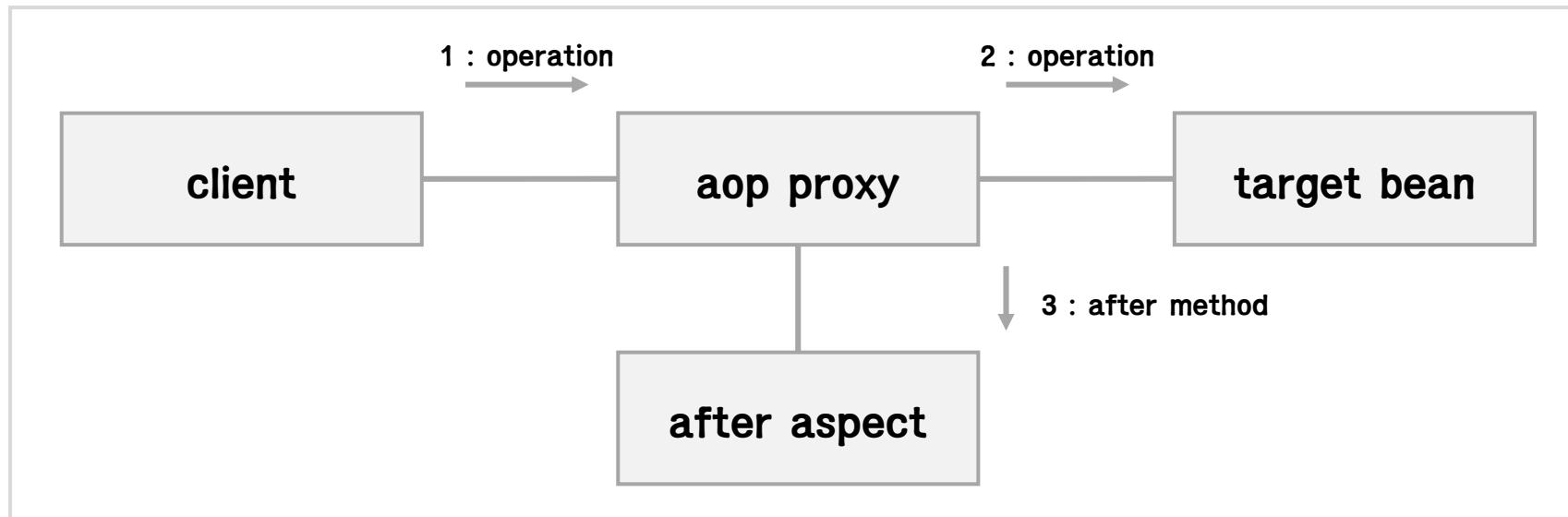


# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice 종류 (4/5)

- After Advice 실행순서.



1. 빈 객체를 사용하는 코드에서 스프링이 생성한 AOP 프록시의 메소드를 호출.
2. AOP 프록시는 실제 빈 객체의 메소드를 호출(정상 실행, exception 발생 : java의 finally와 같음).
3. AOP 프록시는 .<aop:after>에서 지정한 메소드를 호출.



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice 종류 (5/5)

- **Around Advice**

- 위의 네가지 Advice를 다 구현 할 수 있는 Advice.
- return type : Object
- argument :
  - org.aspectj.lang.ProceedingJoinPoint를 반드시 첫 argument로 지정.

```
<aop:config>
    <aop:aspect id="traceAspect" ref="ptAdvice">
        <aop:pointcut id="publicMethod" expression="execution(public * com.test.spring.aop..*Controller.*(..))"/>
        <aop:around method="trace" pointcut-ref="publicMethod"/>
    </aop:aspect>
</aop:config>
```

```
public Object trace(ProceedingJoinPoint joinPoint) throws Throwable {
    String signature = joinPoint.getSignature().toShortString();
    System.out.println("PerformanceTraceAdvice : " + signature);
    long start = System.currentTimeMillis();
    try {
        Object result = joinPoint.proceed();
        return result;
    } finally {
        long finish = System.currentTimeMillis();
        System.out.println("PerformanceTraceAdvice : " + signature
            + " 실행 시간 - " + (finish - start) + "ms");
    }
}
```

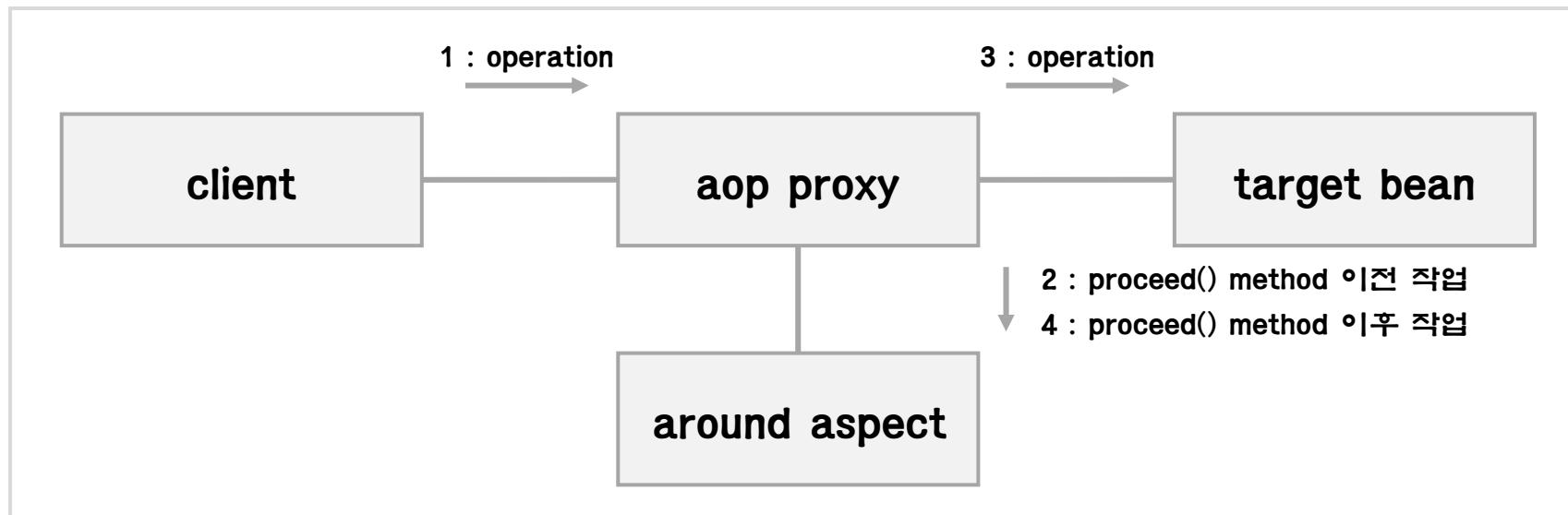


# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ POJO 기반 AOP 구현 – Advice 종류 (5/5)

- Around Advice 실행순서.



1. 빈 객체를 사용하는 코드에서 스프링이 생성한 AOP 프록시의 메소드를 호출.
2. AOP 프록시는 <aop:around>에서 지정한 메소드를 호출.
3. AOP 프록시는 실제 빈 객체의 메소드를 호출.
4. AOP 프록시는 <aop:around>에서 지정한 메소드를 호출.



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ JoinPoint Object 구성요소

- 대상 객체에 대한 정보를 가지고 있는 객체로 Spring Container로 부터 받는다.
- org.aspectj.lang 패키지에 있다.
- 반드시 Aspect method의 첫 argument로 와야 한다.
- 주요 method

Method	설명
Object getTarget()	대상 객체를 리턴
Object[ ] getArgs()	파라미터로 넘겨진 값을 배열로 리턴. 넘어온 값이 없으면 빈 배열이 return 됨
Signature getSignature()	호출되는 method의 정보 Signature : 호출되는 method에 대한 정보를 가진 객체
String getName()	Method 이름
String toLongString()	Method 전체 syntax를 return
String toShortString()	Method를 축약해서 return – 기본은 method 이름



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

### ✓ @Aspect Annotation을 이용한 AOP

- @Aspect Annotation을 이용하여 Aspect Class에 직접 Advice 및 Pointcut 등을 설정.
- 설정 파일에 `<aop:aspectj-autoproxy/>` 를 반드시 추가.
- Aspect Class를 <bean>으로 등록.
- 어노테이션(Annotation)
  - @Aspect : Aspect Class 선언
  - @Before("pointcut")
  - @AfterReturning(pointcut="", returning "")
  - @AfterThrowing(pointcut="", throwing "")
  - @After("pointcut")
  - @Around("pointcut")
- Around를 제외한 나머지 method들은 첫 argument로 JoinPoint를 가질 수 있다.
- Around method는 argument로 ProceedingJoinPoint를 가질 수 있다.



# 4. Aspect Oriented Programming

## 4-3. Spring AOP 구현

- ✓ @Aspect Annotation을 이용한 AOP

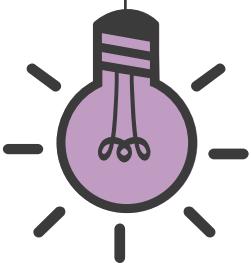
```
@Aspect
public class PerformanceTraceAdvice {

    @Pointcut("execution(public * com.test.spring.aop..*Controller.*(..))")
    public void profileTarget(){}

    @Around("profileTarget()")
    public Object trace(ProceedingJoinPoint joinPoint) throws Throwable{
        String signature = joinPoint.getSignature().toShortString();
        System.out.println("PerformanceTraceAdvice : "+signature);
        long start = System.currentTimeMillis();
        try {
            Object result = joinPoint.proceed();
            return result;
        } finally {
            long finish = System.currentTimeMillis();
            System.out.println("PerformanceTraceAdvice : "
                +signature+ " 실행시간 - "+(finish - start)+"ms");
        }
    }
}
```

05

## MyBatis





## 5. MyBatis

### 5-1. MyBatis 개요와 특징

✓ **MyBatis는 Java Object와 SQL문 사이의 자동 Mapping 기능을 지원하는 ORM Framework.**

- <http://blog.mybatis.org/>
- MyBatis는 SQL을 별도의 파일로 분리해서 관리.
- Object – SQL 사이의 parameter mapping 작업을 자동으로 해줌.
- MyBatis는 Hibernate나 JPA(Java Persistence API)처럼 새로운 DB 프로그래밍 패러다임을 익혀야 하는 부담이 없이, 개발자가 익숙한 SQL을 그대로 이용하면서 JDBC 코드 작성의 불편함을 제거해 주고, 도메인 객체나 VO 객체를 중심으로 개발이 가능.



# 5. MyBatis

## 5-1. MyBatis 개요와 특징

### ✓ **MyBatis 특징.**

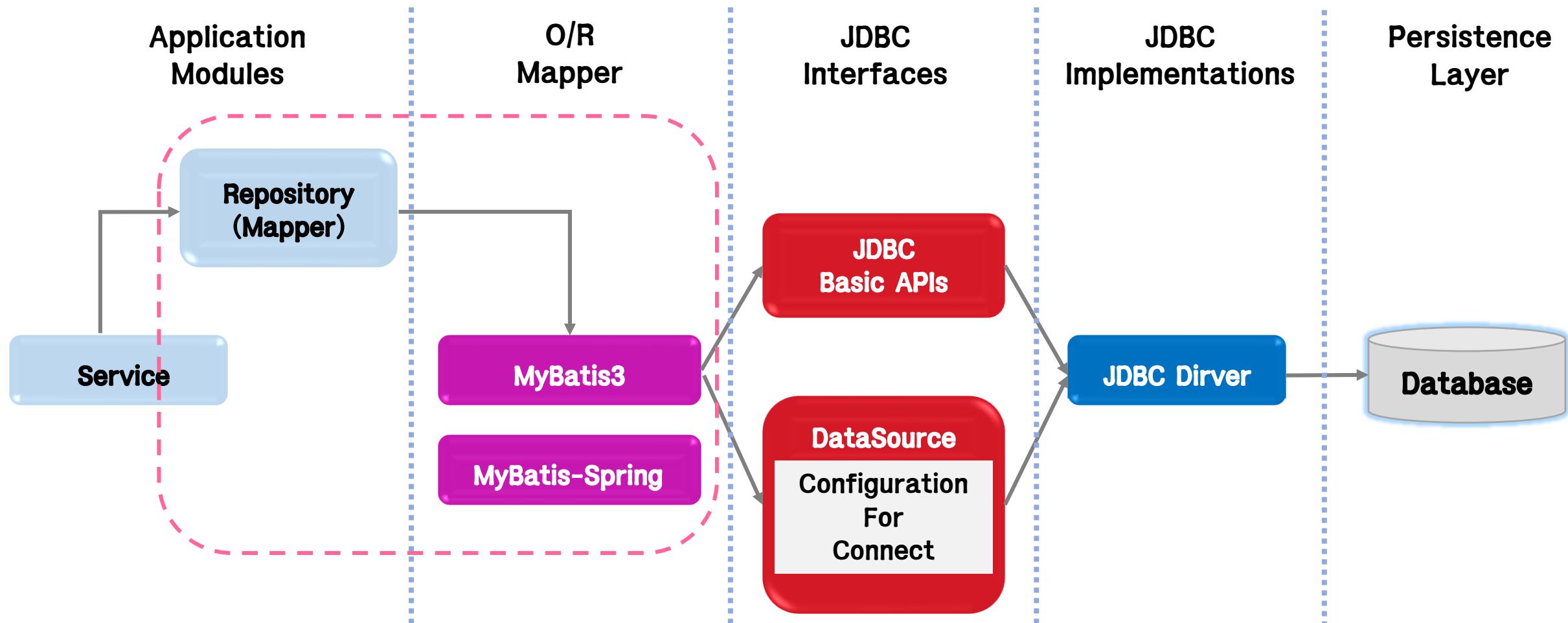
- 쉬운 접근성과 코드의 간결함.
  - 가장 간단한 persistence framework.
  - XML형태로 서술된 JDBC 코드라 생각해도 될 만큼 JDBC의 모든 기능을 MyBatis가 대부분 제공.
  - 복잡한 JDBC 코드를 걷어내며 깔끔한 소스코드를 유지.
  - 수동적인 parameter 설정과 Query 결고에 대한 mapping 구문을 제거.
- SQL문과 프로그래밍 코드의 분리.
  - SQL에 변경이 있을 때마다 자바 코드를 수정하거나 컴파일 하지 않아도 됨.
  - SQL 작성과 관리 또는 검토를 DBA와 같은 개발자가 아닌 다른 사람에게 맡길 수 있음.
- 다양한 프로그래밍 언어로 구현 가능.
  - JAVA, C#, .NET, Ruby, ...



# 5. MyBatis

## 5-2. MyBatis와 MyBatis-Spring의 주요 Component

- ✓ MyBatis와 MyBatis-Spring을 사용한 DB Access Architecture.

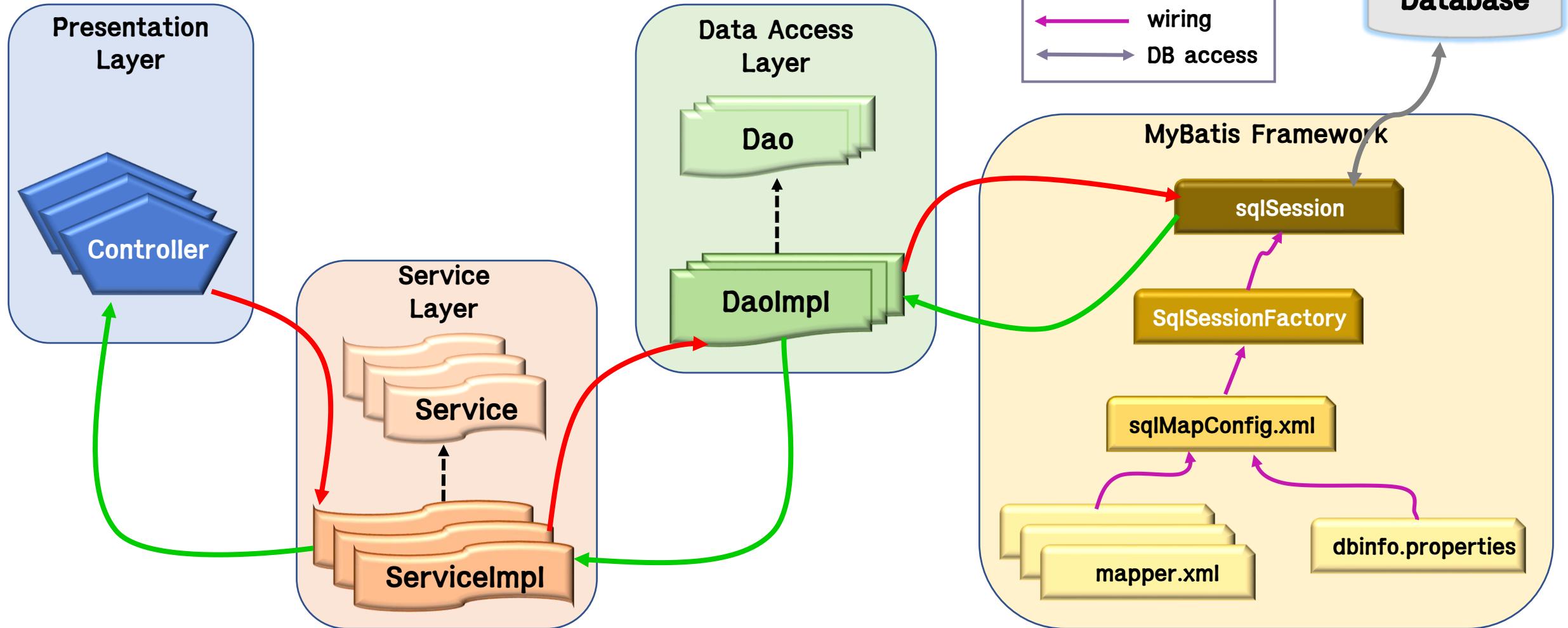




# 5. MyBatis

## 5-2. MyBatis와 MyBatis-Spring의 주요 Component

- ✓ MyBatis를 사용하는 Data Access Layer.

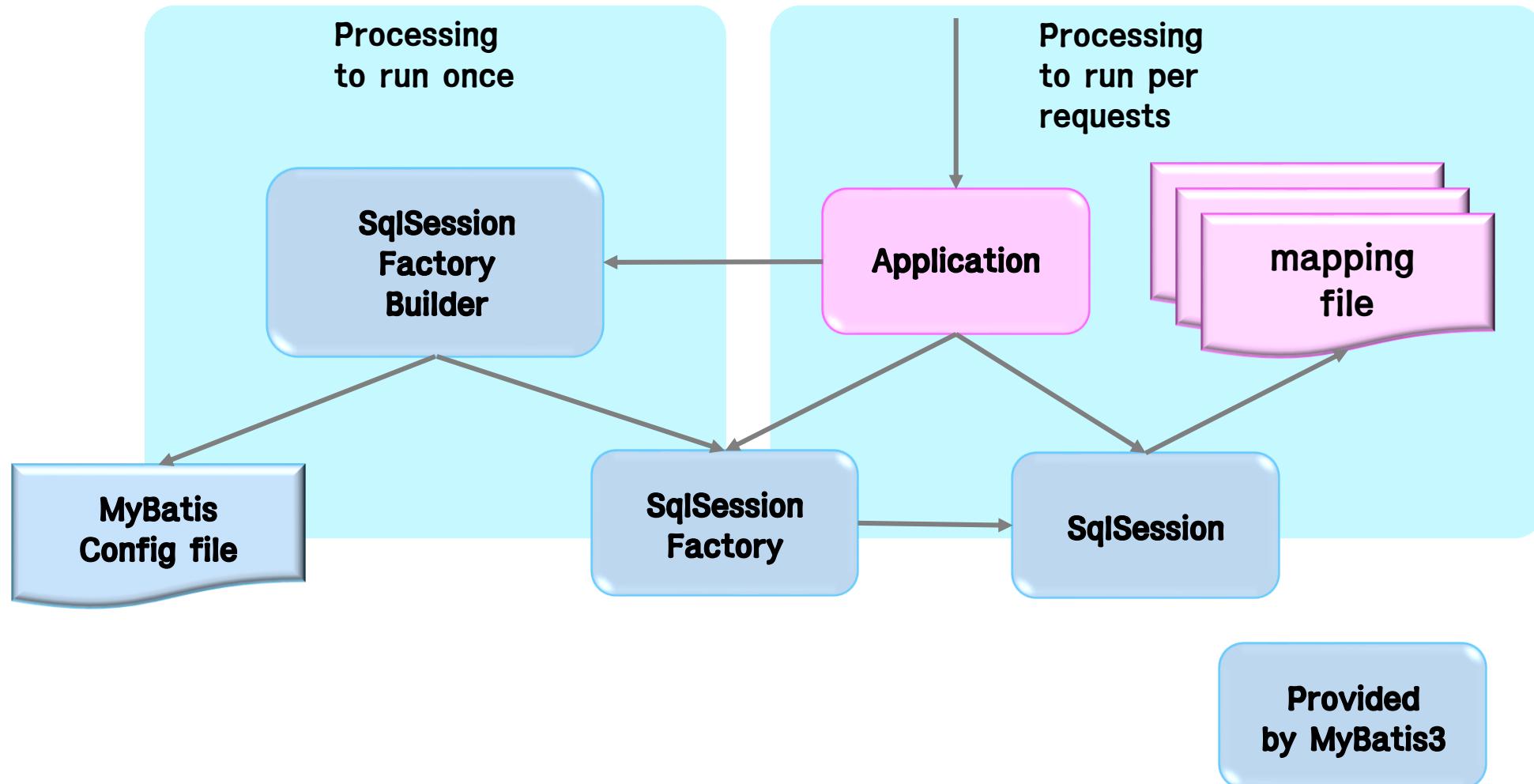




## 5. MyBatis

### 5-2. MyBatis와 MyBatis-Spring의 주요 Component

- ✓ MyBatis 3의 주요 Componet.





## 5. MyBatis

### 5-2. MyBatis와 MyBatis-Spring의 주요 Component

- ✓ **MyBatis 3의 주요 Component의 역할.**

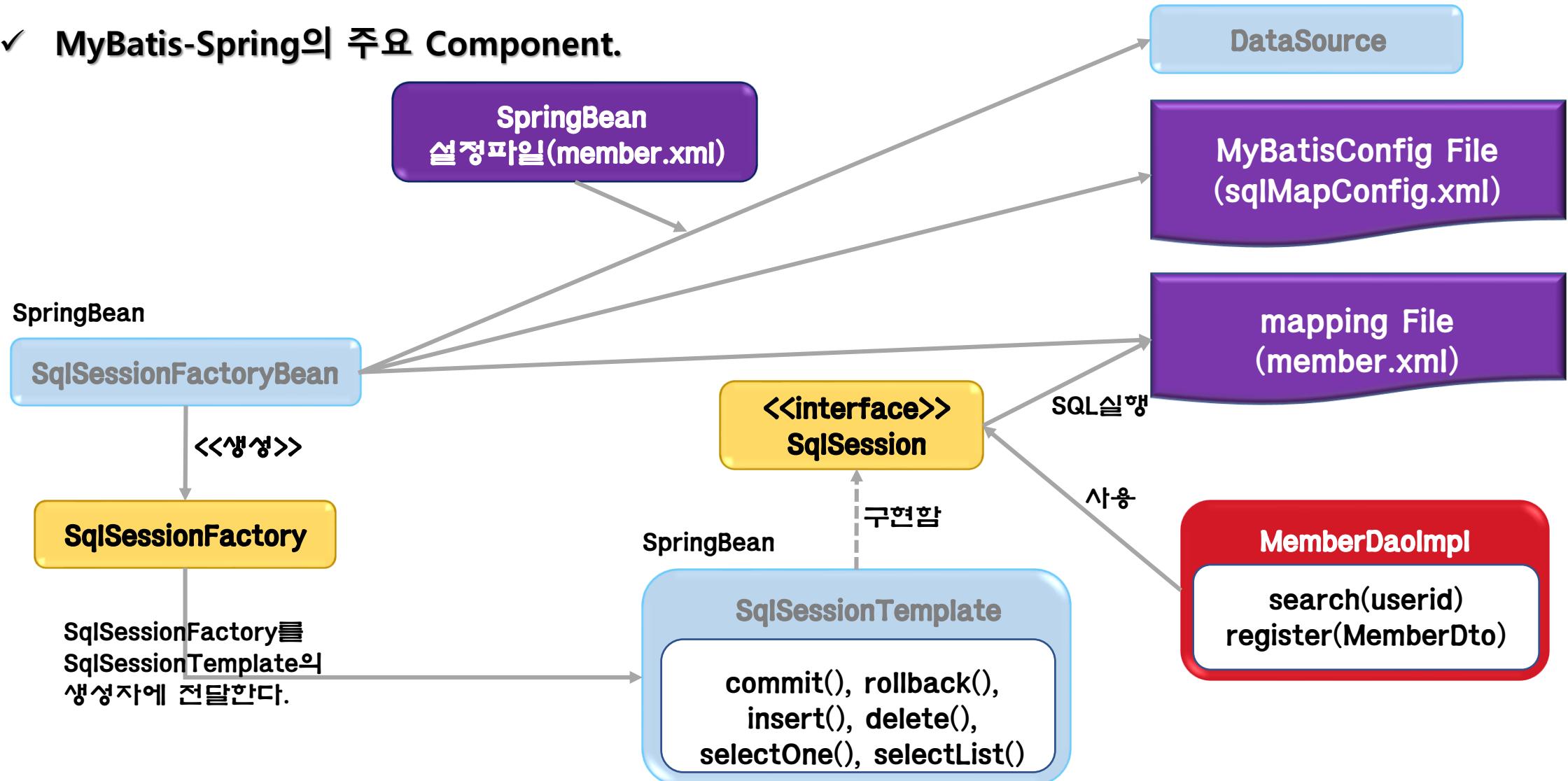
파일	설명
MyBatis 설정파일 (sqlMapConfig.xml)	데이터베이스의 접속 주소 정보나 객체의 alias, Mapping 파일의 경로 등의 고정된 환경 정보를 설정.
SqlSessionFactoryBuilder	MyBatis 설정 파일을 바탕으로 SqlSessionFactory를 생성.
SqlSessionFactory	SqlSession을 생성.
SqlSession	핵심적인 역할을 하는 Class로 SQL 실행이나 Transaction 관리를 실행. SqlSession 오브젝트는 Thread-Safe하지 않으므로 thread마다 필요에 따라 생성.
mapping 파일 (member.xml)	SQL 문과 ORMapping을 설정.



# 5. MyBatis

## 5-2. MyBatis와 MyBatis-Spring의 주요 Component

- ✓ MyBatis-Spring의 주요 Component.





## 5. MyBatis

### 5-2. MyBatis와 MyBatis-Spring의 주요 Component

- ✓ **MyBatis-Spring의 주요 Component의 역할.**

파일	설명
MyBatis 설정파일 (sqlMapConfig.xml)	Dto 객체의 정보를 설정한다.(Alias)
SqlSessionFactoryBean	MyBatis 설정 파일을 바탕으로 SqlSessionFactory를 생성. Spring Bean으로 등록해야 함.
SqlSessionTemplate	핵심적인 역할을 하는 클래스로서 SQL 실행이나 Transaction 관리를 실행. SqlSession interface를 구현하며, Thread-safe하다. Spring Bean으로 등록해야 함.
mapping 파일 (member.xml)	SQL문과 ORMapping을 설정.
Spring Bean 설정파일 (beans.xml)	SqlSessionFactoryBean을 Bean에 등록할 때 DataSource 정보와 MyBatis Config 파일 정보, Mapping 파일의 정보를 함께 설정함. SqlSessionTemplate을 Bean으로 등록.



## 5. MyBatis

### 5-3. MyBatis 3의 Mapper Interface

#### ✓ **Mapper Interface.**

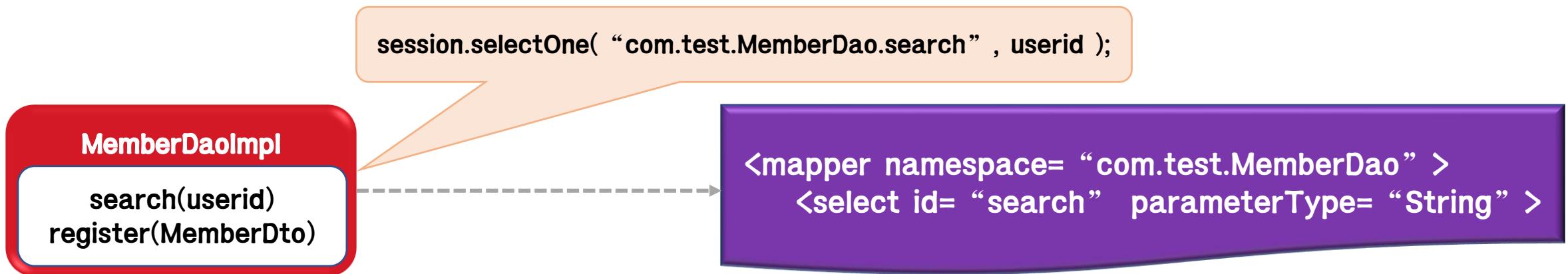
- Mapper Interface는 mapping 파일에 기재된 SQL을 호출하기 위한 Interface.
  - Mapper Interface는 SQL을 호출하는 프로그램을 Type Safe하게 기술하기 위해 MyBatis 3.x부터 등장.
  - Mapping 파일에 있는 SQL을 java interface를 통해 호출할 수 있도록 해줌.



## 5. MyBatis

### 5-3. MyBatis 3의 Mapper Interface

- ✓ **Mapper Interface를 사용하지 않았을 경우.**



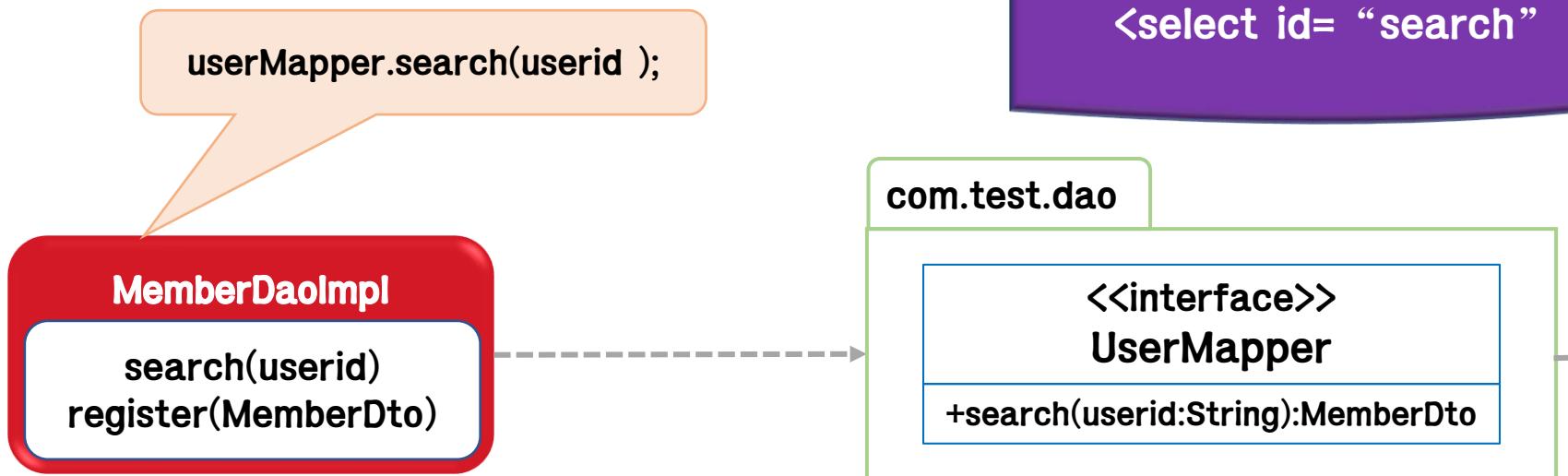
- Mapper Interface를 사용하지 않으면,
  - SQL을 호출하는 프로그램은 SqlSession의 method의 argument에 문자열로 **namespace + “.” + SQL ID**로 지정.
  - 문자열로 지정하기 때문에 오타에 의한 버그가 생기거나, IDE에서 제공하는 code assist를 사용할 수 없음.



## 5. MyBatis

### 5-3. MyBatis 3의 Mapper Interface

- ✓ **Mapper Interface**를 사용했을 경우.



```
<mapper namespace=“com.test.dao.UserMapper”>
  <select id=“search” parameterType=“String”>
```

#### • MemberDaoImpl

```
search(userid)  
register(MemberDto)
```

- UserMapper Interface는 개발자가 작성.

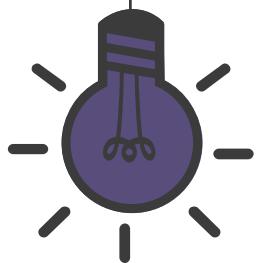
- **packagename + “.” + InterfaceName + “.” + methodName** | namespace + “.” + SQL ID가 되도록 Namespace와 SQL ID를 설정해야 함.

- Namespace 속성에는 package를 포함한 Mapper Interface의 이름을 작성.

- SQL ID에는 mapping하는 method의 이름을 지정.

06

## Spring MVC





# 6. MVC(Model-View-Controller)

## 6-1. MVC Pattern 개념

### ✓ MVC(Model-View-Controller) Pattern

- Model
  - 어플리케이션 상태의 캡슐화.
  - 상태 쿼리에 대한 응답.
  - 어플리케이션의 기능 표현.
  - 변경을 view에 통지.
- View
  - 모델을 화면에 시각적으로 표현.
  - 모델에게 업데이트 요청.
  - 사용자의 입력을 컨트롤러에 전달.
  - 컨트롤러가 view를 선택하도록 허용.
- Controller
  - 어플리케이션의 행위 정의.
  - 사용자 액션을 모델 업데이트와 mapping.
  - 응답에 대한 view 선택 .



# 6. MVC(Model-View-Controller)

## 6-1. MVC Pattern 개념

### ✓ MVC(Model-View-Controller) Pattern

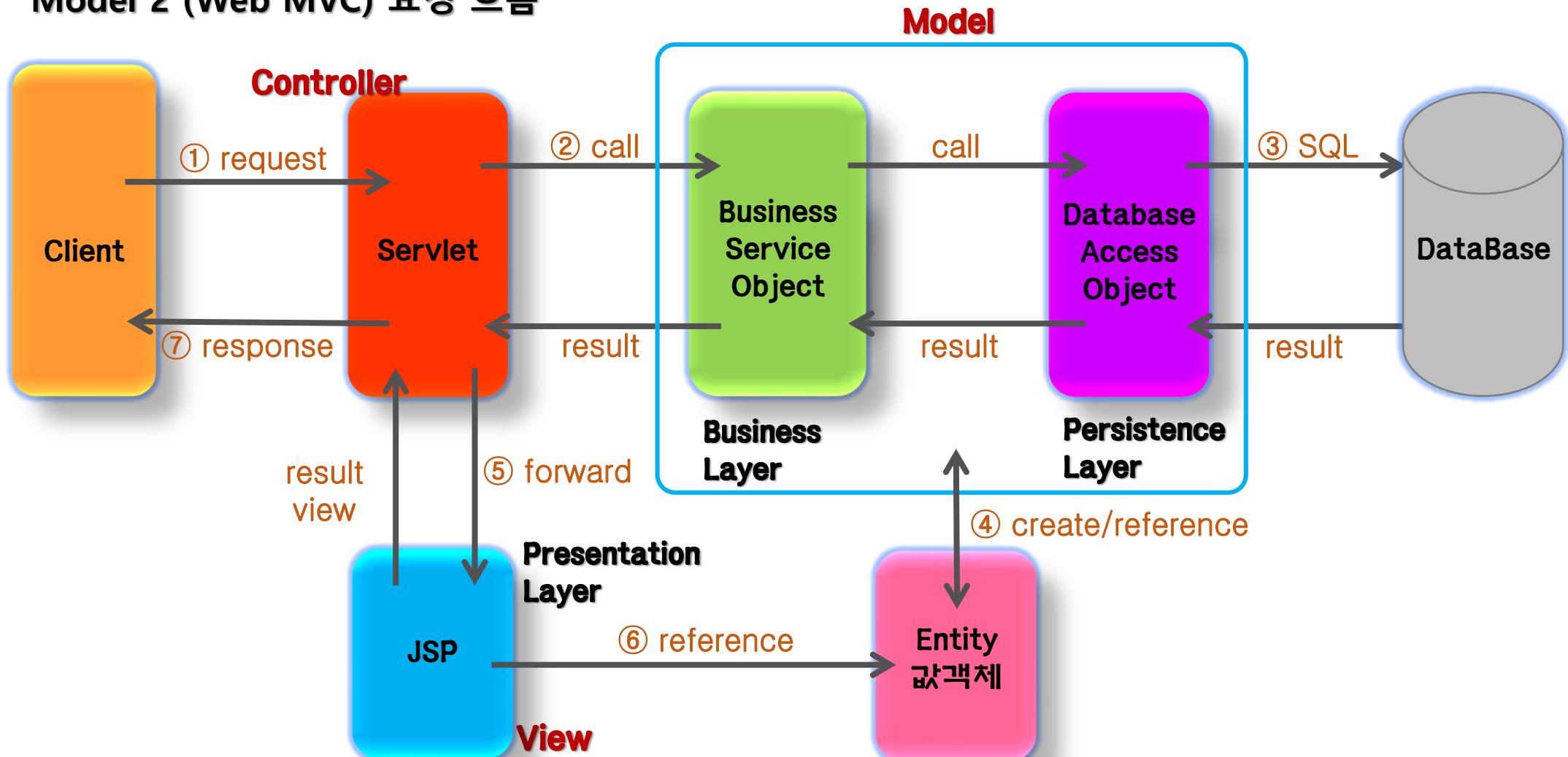
- 어플리케이션의 확장을 위해 Model, View, Controller 세가지 영역으로 분리.
- 컴포넌트의 변경이 다른 영역 컴포넌트에 영향을 미치지 않음(유지보수 용이).
- 컴포넌트 간의 결합성이 낮아 프로그램 수정이 용이(확장성이 뛰어남).
- 장점
  - 화면과 비즈니스 로직을 분리해서 작업 가능.
  - 영역별 개발로 인하여 확장성이 뛰어남.
  - 표준화된 코드를 사용하므로 공동작업이 용이하고 유지보수성이 좋음.
- 단점
  - 개발과정이 복잡해 초기 개발속도가 늦음.
  - 초보자가 이해하고 개발하기에 다소 어려움.



# 6. MVC(Model-View-Controller)

## 6-1. MVC Pattern 개념

- ✓ Model 2 (Web MVC) 요청 흐름





## 6. MVC(Model-View-Controller)

### 6-2. Spring MVC

#### ✓ Spring MVC 특징.

- Spring은 DI나 AOP같은 기능뿐만 아니라, Servlet 기반의 WEB 개발을 위한 MVC Framework를 제공.
- Spring MVC는 Model2 Architecture와 Front Controller Pattern을 Framework 차원에서 제공.
- Spring MVC Framework는 Spring을 기반으로 하고 있기 때문에 Spring이 제공하는 Transaction 처리나 DI 및 AOP등을 손쉽게 사용.



## 6. MVC(Model-View-Controller)

### 6-2. Spring MVC

#### ✓ Spring MVC와 Front Controller Pattern.

- 대부분의 MVC Framework들은 Front Controller Pattern을 적용해서 구현.
- Spring MVC도 Front Controller 역할을 하는 DispatcherServlet이라는 Class를 계층의 맨 앞단에 놓고, 서버로 들어오는 모든 요청을 받아서 처리하도록 구성.
- 예외가 발생했을 때 일관된 방식으로 처리하는 것도 Front Controller의 역할.



# 6. MVC(Model-View-Controller)

## 6-2. Spring MVC

### ✓ Spring MVC 구성요소. (1/2)

- DispatcherServlet (Front Controller)
  - 모든 클라이언트의 요청을 전달받음.
  - Controller에게 클라이언트의 요청을 전달하고, Controller가 리턴한 결과값을 View에게 전달하여 알맞은 응답을 생성.
- HandlerMapping
  - 클라이언트의 요청 URL을 어떤 Controller가 처리할지를 결정.
  - URL과 요청 정보를 기준으로 어떤 핸들러 객체를 사용할지 결정하는 객체이며, DispatcherServlet은 하나 이상의 핸들러 매핑을 가질 수 있음.
- Controller
  - 클라이언트의 요청을 처리한 뒤, Model을 호출하고 그 결과를 DispatcherServlet에 알려준다.



## 6. MVC(Model-View-Controller)

### 6-2. Spring MVC

#### ✓ Spring MVC 구성요소. (2/2)

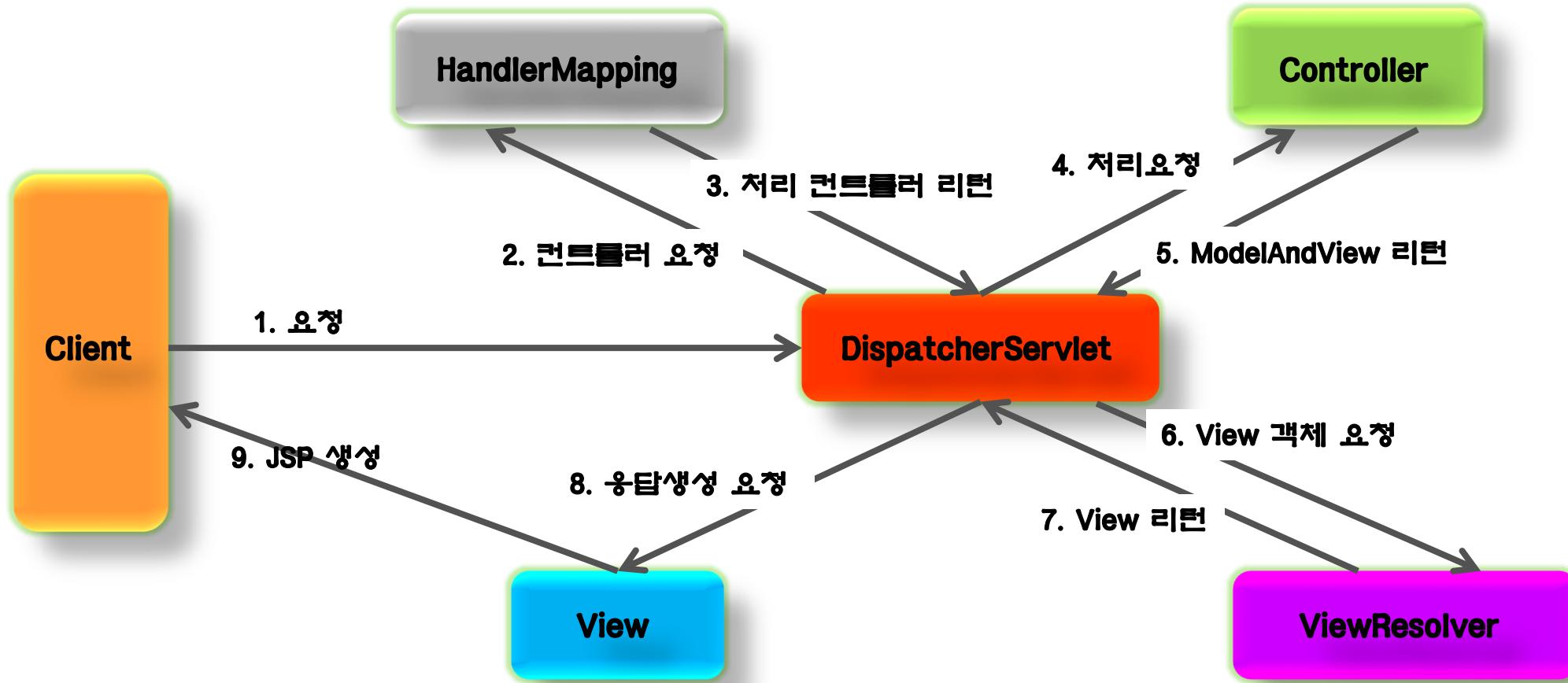
- ModelAndView
  - Controller가 처리한 데이터 및 화면에 대한 정보를 보유한 객체.
- ViewResolver
  - Controller가 리턴한 뷰 이름을 기반으로 Controller의 처리 결과를 보여줄 View를 결정.
- View
  - Controller의 처리결과를 보여줄 응답화면을 생성.



# 6. MVC(Model-View-Controller)

## 6-2. Spring MVC

- ✓ Spring MVC 요청 흐름

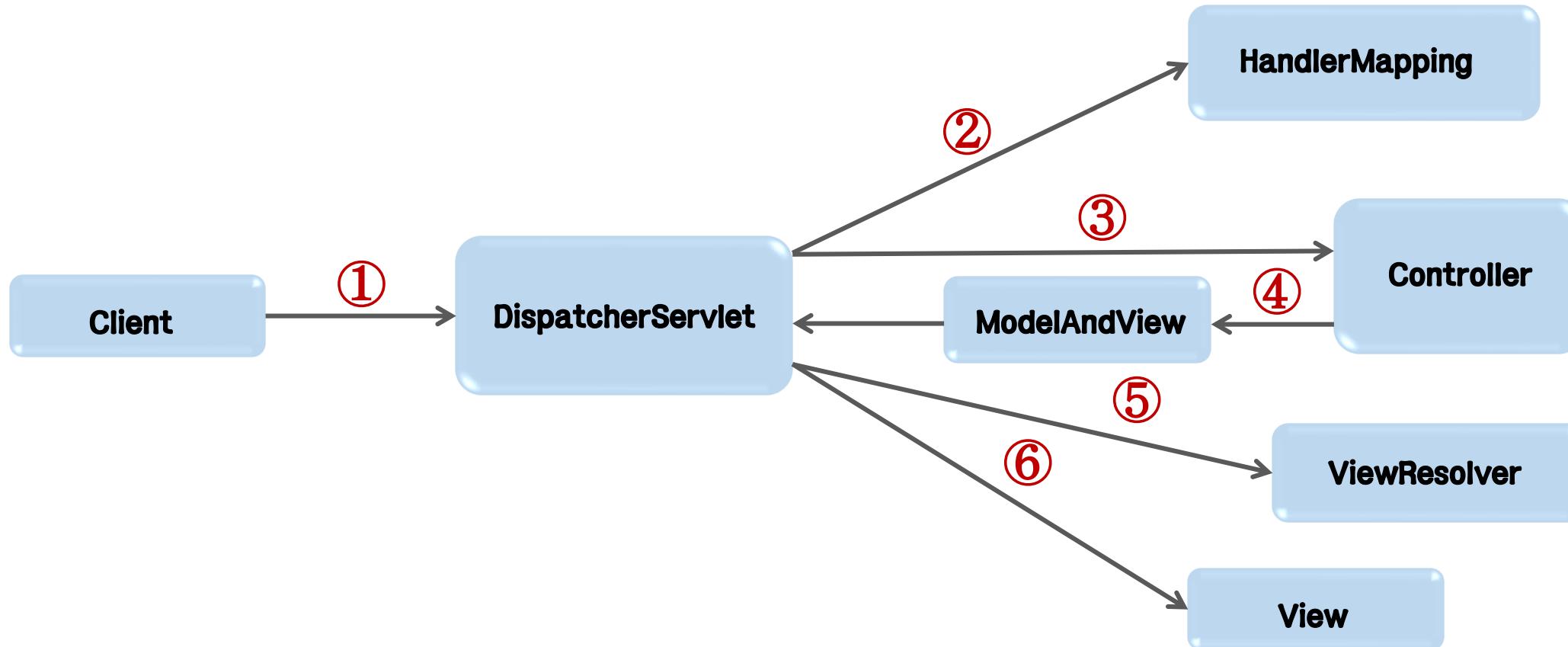




# 6. MVC(Model-View-Controller)

## 6-2. Spring MVC

- ✓ Spring MVC 실행 순서 (1/2)





# 6. MVC(Model-View-Controller)

## 6-2. Spring MVC

### ✓ Spring MVC 실행 순서 (2/2)

① DispatcherServlet이 요청을 수신.

- 단일 Front Controller Servlet.
- 요청을 수신하여 처리를 다른 컴포넌트에 위임.
- 어느 Controller에 요청을 전송할지 결정 .

② DispatcherServlet은 Handler Mapping에 어느 Controller를 사용할 것인지 문의.

- URL과 Mapping.

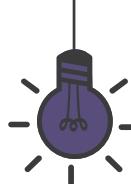
③ DispatcherServlet은 요청을 Controller에게 전송하고 Controller는 요청을 처리한 후 결과 리턴.

- Business Logic 수행 후 결과 정보(Model)가 생성되어 JSP와 같은 view에서 사용됨.

④ ModelAndView Object에 수행결과가 포함되어 DispatcherServlet에 리턴.

⑤ ModelAndView는 실제 JSP정보를 갖고 있지 않으며, ViewResolver가 논리적 이름을 실제 JSP이름으로 변환.

⑥ View는 결과정보를 사용하여 화면을 표현함 .



# 6. MVC(Model-View-Controller)

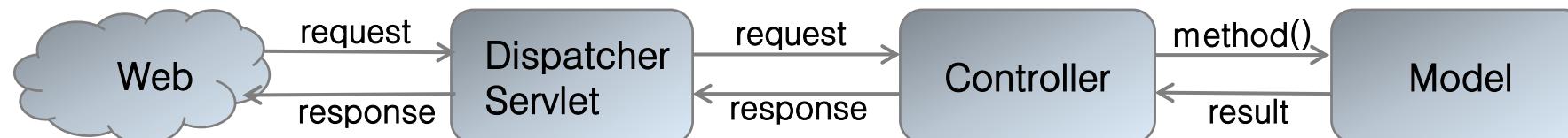
## 6-3. Spring MVC 구현

### ✓ Spring MVC를 이용한 Application 구현 Step

- web.xml에 DispatcherServlet 등록 및 Spring 설정파일 등록.
- 설정 파일에 HandlerMapping 설정.
- Controller 구현 및 Context 설정 파일(servlet-context.xml)에 등록.
- Controller와 JSP의 연결을 위해 View Resolver 설정.
- JSP 코드 작성.

### ✓ Controller 작성

- 좋은 디자인은 Controller가 많은 일을 하지 않고 Service에 처리를 위임.





# 6. MVC(Model-View-Controller)

## 6-3. Spring MVC 구현

### ✓ web.xml – DispatcherServlet 설정

- <init-param>을 설정 하지 않으면 “<servlet-name>-servlet.xml” file에서 applicationContext의 정보를 load.
- Spring Container는 설정파일의 내용을 읽고 ApplicationContext 객체를 생성.
- <url-pattern>은 DispatcherServlet이 처리하는 URL Mapping pattern을 정의.
- Servlet이므로 1개 이상의 DispatcherServlet 설정 가능.
- <load-on-startup>1</load-on-startup>설정 시 WAS startup시 초기화 작업진행.



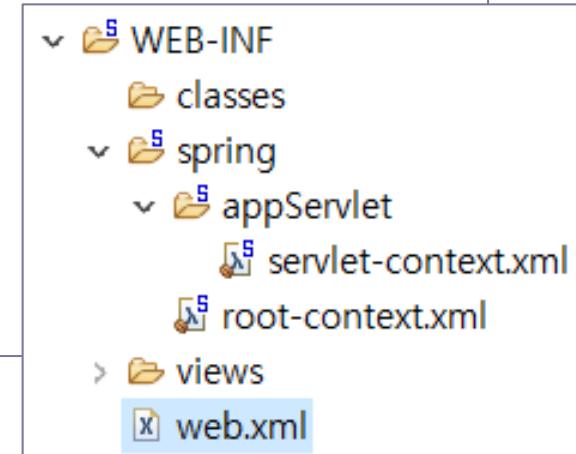
# 6. MVC(Model-View-Controller)

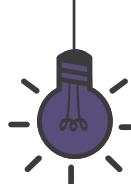
## 6-3. Spring MVC 구현

- ✓ web.xml – DispatcherServlet 설정

```
<!-- Processes application requests -->
<servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```





# 6. MVC(Model-View-Controller)

## 6-3. Spring MVC 구현

### ✓ web.xml – DispatcherServlet 설정

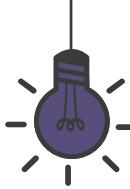
- DispatcherServlet을 여러 개 설정 가능.
- 각 DispatcherServlet마다  
각각의 ApplicationContext 생성.

```
<!-- Processes application requests -->
<servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet>
    <servlet-name>appServlet2</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/appServlet/servlet-context2.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>appServlet2</servlet-name>
    <url-pattern>*.action</url-pattern>
</servlet-mapping>
```



# 6. MVC(Model-View-Controller)

## 6-3. Spring MVC 구현

### ✓ web.xml – 최상위 Root ContextLoader 설정

- Context 설정 파일들을 로드하기 위해 web.xml 파일에 리스너 설정.(ContextLoaderListener)

```
<!-- Creates the Spring Container shared by all Servlets and Filters -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

- 리스너 설정이 되면 /WEB-INF/spring/root-context.xml 파일을 읽어서 공통적으로 사용되는 최상위 Context를 생성.
- 그 외의 다른 컨텍스트 파일들을 최상위 어플리케이션 컨텍스트에 로드하기 위해서는...

```
<!-- The definition of the Root Spring Container shared by all Servlets and Filters -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spring/root-context.xml
        classpath:com/test/web/application.xml
    </param-value>
</context-param>
```

클래스 패스에 위치한 설정파일로부터 로드

```
<param-value>
    /WEB-INF/spring/root-*.xml
</param-value>
```



# 6. MVC(Model-View-Controller)

## 6-3. Spring MVC 구현

### ✓ Application Context 분리

- 어플리케이션 레이어에 따라 어플리케이션 컨텍스트 분리

Layer	설정파일
Security Layer	board-security.xml
Web Layer	board-servlet.xml
Service Layer	board-service.xml
Persistence Layer	board-dao.xml



# 6. MVC(Model-View-Controller)

## 6-3. Spring MVC 구현

- ✓ Controller Class 작성. (HomeController.java)

```
@Controller  
public class HomeController {  
  
    private static final Logger logger = LoggerFactory.getLogger(HomeController.class);  
  
    @RequestMapping(value = "/", method = RequestMethod.GET)  
    public String home(Locale locale, Model model) {  
        logger.info("Welcome home! The client locale is {}.", locale);  
  
        model.addAttribute("message", "안녕하세요 스프링!!!!");  
  
        return "index";  
    }  
}
```

- ✓ Context 설정파일에 Controller 등록. (servlet-context.xml)

```
<beans:bean class="com.test.web.HomeController"/>
```



# 6. MVC(Model-View-Controller)

## 6-3. Spring MVC 구현

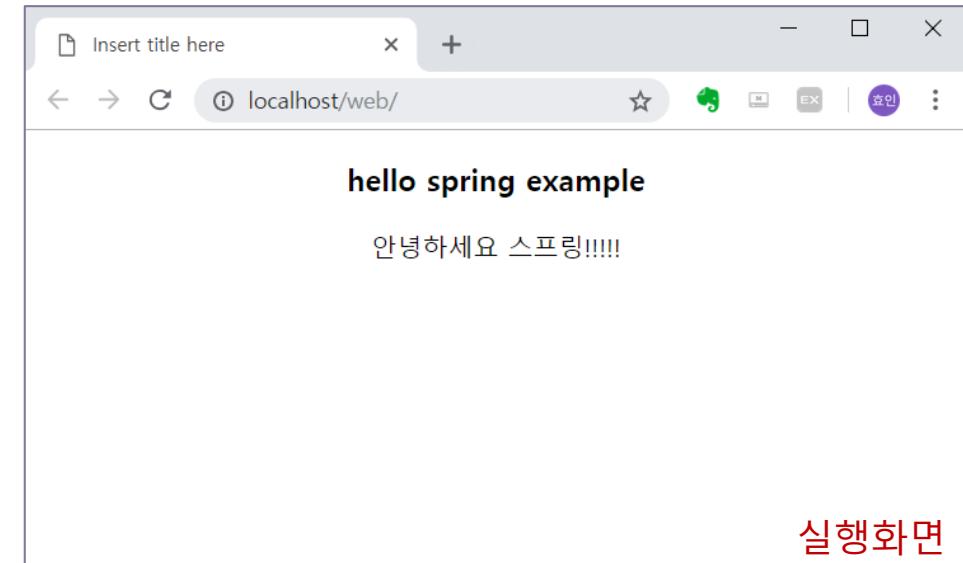
- ✓ Controller와 response page 연결을 위한 ViewResolver 설정. (servlet-context.xml)

```
<beans:bean class="com.test.web.HomeController"/>

<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <beans:property name="prefix" value="/WEB-INF/views/" />
    <beans:property name="suffix" value=".jsp" />
</beans:bean>
```

- ✓ JSP (index.jsp)

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<div align="center">
<h3>hello spring example</h3>
${message}
</div>
</body>
</html>
```



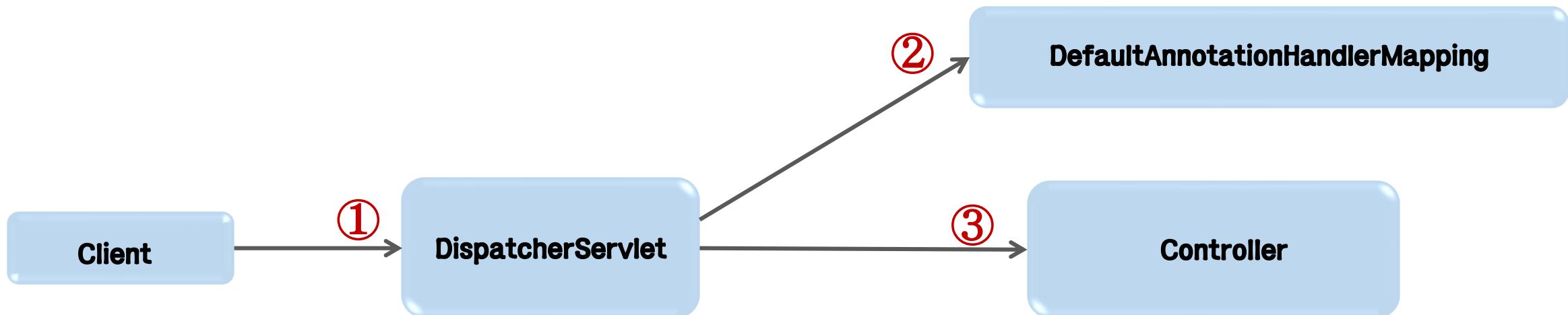


## 6. MVC(Model-View-Controller)

### 6-4. @Controller

#### ✓ @Controller와 @RequestMapping 선언.

- method 단위의 mapping이 가능.
- DefaultAnnotationHandlerMapping과 AnnotationHandlerAdapter를 사용함.
  - Spring 3.0부터는 기본설정이므로 별도의 추가 없이 사용 가능.





# 6. MVC(Model-View-Controller)

## 6-4. @Controller

- ✓ Controller Class는 Client의 요청을 처리.
- ✓ **@Controller 선언.**
  - Class 타입에 적용.
  - Spring 3.0부터는 @Controller 사용을 권장.

```
@Controller  
@RequestMapping("/reboard")  
public class ReboardController {  
  
    @Autowired  
    private ReboardService reboardService;  
  
    @Autowired  
    private CommonService commonService;  
  
    @RequestMapping("/list.kitri")  
    public ModelAndView list(@RequestParam Map<String, String> map) {  
        ModelAndView mav = new ModelAndView();  
  
        List<ReboardDto> list = reboardService.listArticle(map);  
        PageNavigation navigation = commonService.getPageNavigation(map);  
        navigation.setRoot("/board");  
        navigation.makeNavigator();  
        mav.addObject("list", list);  
        mav.addObject("navigator", navigation);  
        mav.setViewName("reboard/list");  
    }  
}
```

AbstractController class, AbstractCommonController class, Controller interface 등을 확장해서 Controller 작성은 지양함.



## 6. MVC(Model-View-Controller)

### 6-4. @Controller

- ✓ Controller Class를 <bean>에 등록.

```
<beans:bean id="reboardController" class="com.test.board.controller.ReboardController">
    <beans:property name="reboardService" ref="reboardService"/>
</beans:bean>
```

- ✓ Controller Class 자동 스캔.

- context:component-scan 선언
- base-package에 설정된 package내의 class중 @Controller annotation이 적용된 클래스는 자동 스캔대상.

```
<context:component-scan base-package="com.test.board.controller"/>
```



# 6. MVC(Model-View-Controller)

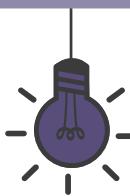
## 6-4. @Controller

### ✓ @RequestMapping 선언.

- 요청 URL mapping 정보를 설정.
- 클래스타입과 메소드에 설정 가능.

```
@Controller  
public class ReboardController {  
  
    private ReboardService reboardService;  
  
    public void setReboardService(ReboardService reboardService) {  
        this.reboardService = reboardService;  
    }  
  
    @RequestMapping("/reboard/write.do")  
    public String write() {  
        return "reboard/write";  
    }  
  
    @RequestMapping("/reboard/list.do")  
    public ModelAndView list(@RequestParam Map<String, String> map) {  
        List<ReboardDto> article = reboardService.listArticle(map);  
        return new ModelAndView("reboard/list", "list", article);  
    }  
}
```

```
@Controller  
@RequestMapping("/reboard")  
public class ReboardController {  
  
    private ReboardService reboardService;  
  
    public void setReboardService(ReboardService reboardService) {  
        this.reboardService = reboardService;  
    }  
  
    @RequestMapping("/write.do")  
    public String write() {  
        return "reboard/write";  
    }  
  
    @RequestMapping("/list.do")  
    public ModelAndView list(@RequestParam Map<String, String> map) {  
        List<ReboardDto> article = reboardService.listArticle(map);  
        return new ModelAndView("reboard/list", "list", article);  
    }  
}
```



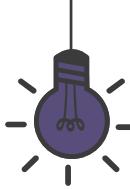
# 6. MVC(Model-View-Controller)

## 6-4. @Controller

### ✓ Controller method의 HTTP method에 한정.

- 같은 URL 요청에 대하여 HTTP method(GET, POST...)에 따라 서로 다른 메소드를 mapping 할 수 있음.

```
@Controller  
public class HomeController {  
  
    private static final Logger logger = LoggerFactory.getLogger(HomeController.class);  
  
    @RequestMapping(value = "/index.do", method = RequestMethod.GET)  
    public String home(Model model) {  
        model.addAttribute("message", "안녕하세요 스프링(GET)!!!!");  
  
        return "index";  
    }  
  
    @RequestMapping(value = "/index.do", method = RequestMethod.POST)  
    public String home2(Model model) {  
        model.addAttribute("message", "안녕하세요 스프링(POST)!!!!");  
  
        return "index";  
    }  
}
```



# 6. MVC(Model-View-Controller)

## 6-4. @Controller

- ✓ 아래의 Controller에서 @RequestMapping annotation을 설정하지 않으면? HTTP ERROR 404
- ✓ @RequestMapping에서 method를 설정하지 않으면? HTTP ERROR 500

```
@Controller
@RequestMapping(value = "/index.do")
public class HomeController {

    private static final Logger Logger = LoggerFactory.getLogger(HomeController.class);

    @RequestMapping(method = RequestMethod.GET)
    public String home(Model model) {
        model.addAttribute("message", "안녕하세요 스프링(GET)!!!!!");
        return "index";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String home2(Model model) {
        model.addAttribute("message", "안녕하세요 스프링(POST)!!!!!");
        return "index";
    }
}
```



# 6. MVC(Model-View-Controller)

## 6-4. @Controller

### ✓ Controller method의 parameter type. (1/2)

- Controller method의 parameter로 다양한 Object를 받을 수 있음.

Parameter Type	설명
HttpServletRequest	
HttpServletResponse	필요시 Servlet API를 사용할 수 있음
HttpSession	
Java.util.Locale	현재 요청에 대한 Locale
InputStream, Reader	요청 컨텐츠에 직접 접근할 때 사용
OutputStream, Writer	응답 컨텐츠를 생성할 때 사용
@PathVariable annotation 적용 파라미터	URI 템플릿 변수에 접근할 때 사용
@RequestParam annotation 적용 파라미터	HTTP 요청 파라미터를 매핑
@RequestHeader annotation 적용 파라미터	HTTP 요청 헤더를 매핑
@CookieValue annotation 적용 파라미터	HTTP 쿠키 매핑



# 6. MVC(Model-View-Controller)

## 6-4. @Controller

### ✓ Controller method의 parameter type. (2/2)

- Controller method의 parameter로 다양한 Object를 받을 수 있음.

Parameter Type	설명
@RequestBody annotation 적용 파라미터	HTTP 요청의 몸체 내용에 접근할 때 사용
Map, Model, ModelMap	view에 전달할 model data를 설정할 때 사용
커맨드 객체	HTTP 요청 parameter를 저장 한 객체, 기본적으로 클래스 이름을 모델명으로 사용. @ModelAttribute annotation 설정으로 모델명을 설정할 수 있음.
Errors, BindingResult	HTTP 요청 파라미터를 커맨드 객체에 저장한 결과, 커맨드 객체를 위한 파라미터 바로 다음에 위치 폼 처리를 완료 했음을 처리하기 위해 사용.
SessionStatus	@SessionAttributes annotation을 명시한 session속성을 제거하도록 이벤트를 발생 시킨다.



# 6. MVC(Model-View-Controller)

## 6-4. @Controller

- ✓ **@RequestParam annotation을 이용한 parameter mapping.**

<http://localhost/web/index.do?name=안효인&age=30>

```
@Controller  
public class HomeController {  
  
    @RequestMapping(value = "/index.do", method = RequestMethod.GET)  
    public String home(@RequestParam("name") String name,  
                      @RequestParam("age") int age, Model model) {  
        model.addAttribute("message", name + "(" + age + ")님 안녕하세요!!!!");  
        return "index";  
    }  
}
```

```
@Controller  
public class HomeController {  
  
    @RequestMapping(value = "/index.do", method = RequestMethod.GET)  
    public String home(@RequestParam(value="name", required=false) String name,  
                      @RequestParam(value="age", defaultValue="25") int age, Model model) {  
        model.addAttribute("message", name + "(" + age + ")님 안녕하세요!!!!");  
        return "index";  
    }  
}
```

필수여부

기본값



# 6. MVC(Model-View-Controller)

## 6-4. @Controller

### ✓ HTML form과 Command Object(DTO, VO...).

- SpringMVC는 form에 입력한 data를 JavaBean 객체를 이용해서 전송 할 수 있음.

```
<form method="POST" action="${root}/board/write.do">  
제목 : <input type="text" name="subject"><br>  
내용 : <textarea name="content"></textarea><br>  
<input type="submit" value="글쓰기"/>
```

```
@Controller  
@RequestMapping("/board")  
public class BoardController {  
  
    @RequestMapping(value="/write.do", method=RequestMethod.GET)  
    public String write() {  
        return "board/write";  
    }  
  
    @RequestMapping(value="/write.do", method=RequestMethod.POST)  
    public String write(BoardDto boardDto) {  
  
        return "board/writeok";  
    }  
}
```

```
public class BoardDto {  
  
    private String subject;  
    private String content;  
  
    public void setSubject(String subject) {  
        this.subject = subject;  
    }  
  
    public void setContent(String content) {  
        this.content = content;  
    }  
}
```



# 6. MVC(Model-View-Controller)

## 6-4. @Controller

- ✓ Command 객체를 List로 받기.

```
<form method="POST" action="${root}/shop/basket.do">
    <input type="text" name="productList[0].pnum">
    <input type="text" name="productList[0].name">
    <input type="text" name="productList[0].price">
    <br>
    <input type="text" name="productList[1].pnum">
    <input type="text" name="productList[1].name">
    <input type="text" name="productList[1].price">
    <input type="submit" value="장바구니저장">
</form>
```

```
public class Bascket {
    private List<Product> productList;
    public List<Product> getProductList() {
        return productList;
    }
    public void setProductList(List<Product> productList) {
        this.productList = productList;
    }
}
```

```
@Controller
@RequestMapping("/shop")
public class ShopController {

    @RequestMapping(value="/basket.do", method=RequestMethod.POST)
    public String basket(Basket bascket, Model model) {
        model.addAttribute("basketList", bascket);
        return "shop/basketlist";
    }
}
```

```
public class Product {
    private int pnum;
    private String name;
    private int price;
    public int getPnum() {
        return pnum;
    }
    public void setPnum(int pnum) {
        this.pnum = pnum;
    }
}
```

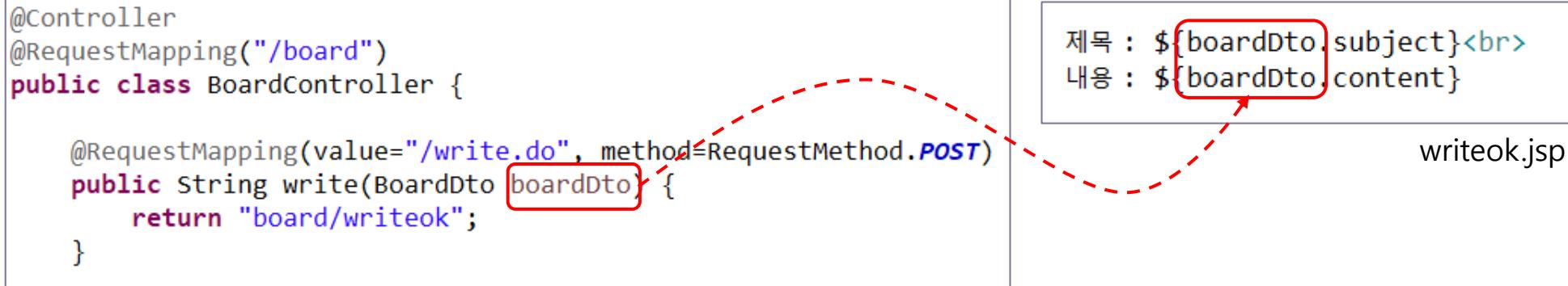


# 6. MVC(Model-View-Controller)

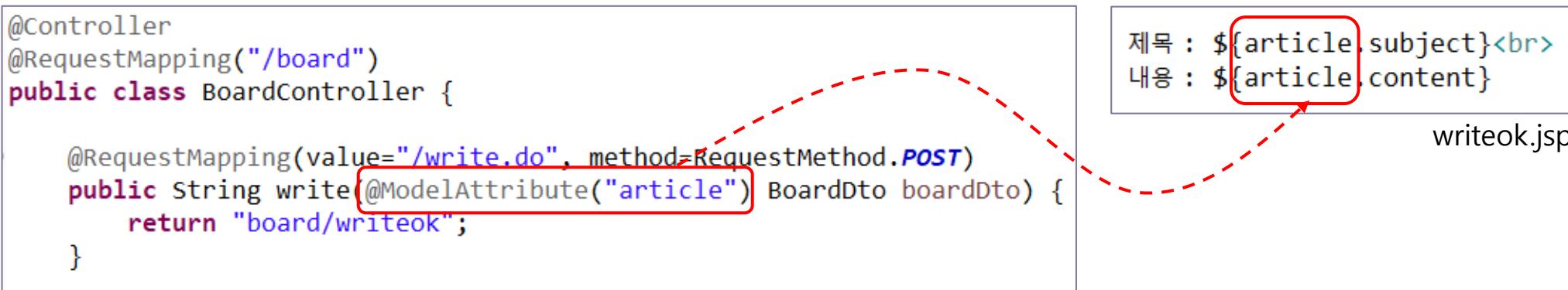
## 6-4. @Controller

### ✓ View에서 Command 객체에 접근.

- Command 객체는 자동으로 반환되는 Model에 추가됨.
- Controller의 @RequestMapping annotation method에서 전달받은 Command 객체에 접근.



- `@ModelAttribute`를 사용하여 View에서 사용할 Command 객체의 이름을 변경.





## 6. MVC(Model-View-Controller)

### 6-4. @Controller

- ✓ `@CookieValue annotation을 이용한 Cookie mapping.`

```
@Controller
public class HomeController {

    public String hello(@CookieValue("author") String authorValue) {
        return "ok";
    }
    public String hello(@CookieValue(value="author", required=false, defaultValue="user") String authorValue) {
        return "ok";
    }
}
```

- ✓ `@RequestHeader annotation을 이용한 header mapping.`

```
@Controller
public class HomeController {

    public String hello(@RequestHeader("Accept-Language") String headerLanguage) {
        return "ok";
    }
}
```



## 6. MVC(Model-View-Controller)

### 6-4. @Controller

#### ✓ **@RequestBody parameter type.**

- HTTP 요청 Body가 그대로 객체에 전달됨.
- AnnotationMethodHandlerAdapter에는 HttpMessageConverter 타입의 메시지 변환기가 기본으로 여러 개 등록되어 있음.
- @RequestBody가 붙은 parameter가 있으면 해당 미디어 타입을 확인 후 처리 가능한 변환기(Converter)가 자동으로 객체로 변환시켜 줌.
- 주로 @ResponseBody와 함께 사용됨.



# 6. MVC(Model-View-Controller)

## 6-4. @Controller

### ✓ Servlet API 사용.

- HttpSession의 생성을 직접 제어해야 하는 경우.
- Controller가 Cookie를 생성해야 하는 경우.
- Servlet API를 선호하는 경우.
  - javax.servlet.ServletRequest / javax.servlet.http.HttpServletRequest
  - javax.servlet.ServletResponse / javax.servlet.http.HttpServletResponse
  - javax.servlet.http.HttpSession

```
@Controller  
public class HomeController {  
  
    public String hello(HttpServletRequest request, HttpServletResponse respons) {  
        return "ok";  
    }  
}
```

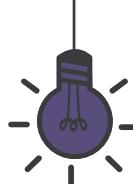


# 6. MVC(Model-View-Controller)

## 6-4. @Controller

- ✓ Controller Class에서 method의 return type 종류.

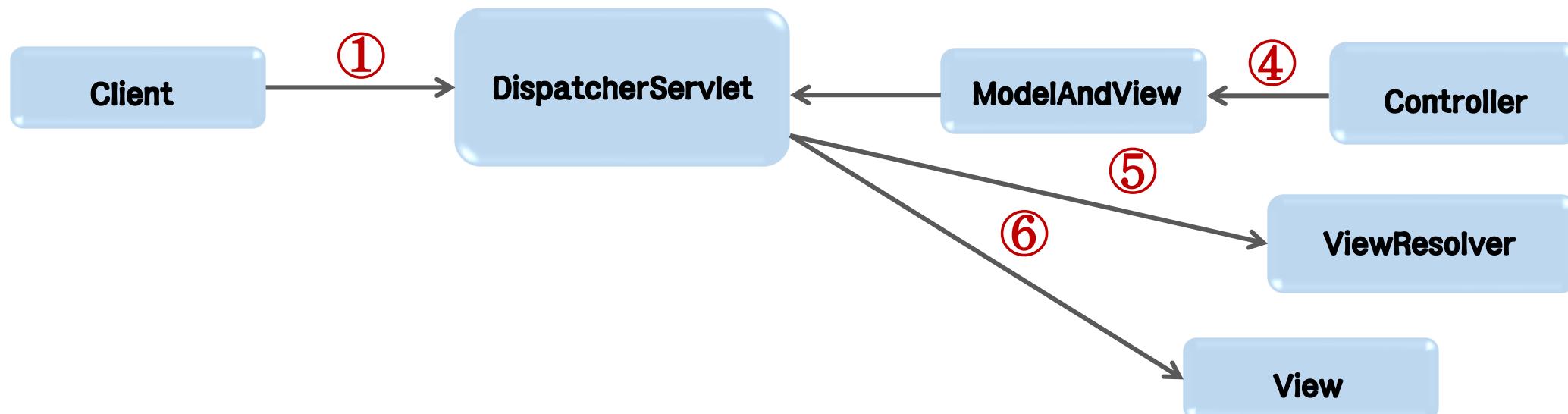
Return Type	설명
ModelAndView	model 정보 및 view 정보를 담고있는 ModelAndView 객체.
Model	view에 전달할 객체 정보를 담고있는 Model을 반환한다. 이때 view 이름은 요청 URL로부터 결정된다. (RequestToViewNameTranslator)
Map	view에 전달 할 객체 정보를 담고 있는 Map을 반환한다. 이때 view 이름은 요청 URL로부터 결정된다. (RequestToViewNameTranslator)
String	view 이름을 반환한다.
View	view 객체를 직접 리턴, 해당 View 객체를 이용해서 view를 생성한다. method가 ServletResponse나 HttpServletResponse 타입의 parameter를 갖는 경우 method가 직접 응답을 처리
void	한다고 가정한다. 그렇지 않을 경우 요청 URL로부터 결정된 View를 보여준다. (RequestToViewNameTranslator)
@ResponseBody	method에서 @ResponseBody annotation이 적용된 경우, 리턴 객체를 HTTP 응답으로 전송한다.
Annotation 적용	HttpMessageConverter를 이용해서 객체를 HTTP 응답 스트림으로 변환한다.



## 6. MVC(Model-View-Controller)

### 6-5. View 지정

- ✓ Controller에서는 처리 결과를 보여줄 View 이름이나 객체를 리턴하고, DispatcherServlet은 View이름이나 View 객체를 이용하여 view를 생성.
  - 명시적 지정.
  - 자동 지정.





## 6. MVC(Model-View-Controller)

### 6-5. View 지정

- ✓ ViewResolver : 논리적 view와 실제 JSP파일 mapping.
  - servlet-context.xml

```
<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <beans:property name="prefix" value="/WEB-INF/views/" />
    <beans:property name="suffix" value=".jsp" />
</beans:bean>
```

InternalResourceViewResolver는  
prefix + 논리뷰 + suffix로 설정  
ex)/WEB-INF/views/board/list.jsp



# 6. MVC(Model-View-Controller)

## 6-5. View 지정

### ✓ View 이름 명시적 지정.

- ModelAndView와 String 리턴타입.

```
@Controller  
public class HomeController {  
  
    @RequestMapping("/hello.do")  
    public ModelAndView hello() {  
        ModelAndView mav = new ModelAndView("hello");  
        return mav;  
    }  
}
```

```
@Controller  
public class HomeController {  
  
    @RequestMapping("/hello.do")  
    public ModelAndView hello() {  
        ModelAndView mav = new ModelAndView();  
        mav.setViewName("hello");  
        return mav;  
    }  
}
```

```
@Controller  
public class HomeController {  
  
    @RequestMapping("/hello.do")  
    public String hello() {  
        return "hello";  
    }  
}
```



# 6. MVC(Model-View-Controller)

## 6-5. View 지정

### ✓ View 자동 지정.

- RequestToViewNameTranslator를 이용하여 URL로 부터 view 이름을 결정.
- 자동 지정 유형.
  - return type이 Model이나 Map인 경우.
  - return type이 void이면서 ServletResponse나 HttpServletResponse 타입의 parameter가 없는 경우.

```
@Controller  
public class HomeController {  
  
    @RequestMapping("/hello.do")  
    public Map<String, Object> hello() {  
        Map<String, Object> model = new HashMap<String, Object>();  
        return model;  
    }  
}
```

hello가 view 이름이 됨.



# 6. MVC(Model-View-Controller)

## 6-5. View 지정

### ✓ redirect view.

- View 이름에 “**redirect:**” 접두어를 붙이면, 지정한 페이지로 redirect 됨.
- redirect:/board/list.html?pg=1
- redirect:http://localhost/board/list.html?pg=1

```
@Controller
public class BoardRegisterController {
    @Autowired
    private BoardService boardService;

    // 글등록후 1페이지 글리스트로 이동.
    @RequestMapping(value="board/register.html", method=RequestMethod.POST)
    public String register(@ModelAttribute("article") BoardDto boardDto) {
        boardService.registerArticle(boardDto);
        return "redirect:board/list.html?pg=1";
    }
}
```



## 6. MVC(Model-View-Controller)

### 6-6. Model 생성

#### ✓ View에 전달하는 데이터.

- @RequestMapping annotation이 적용된 method의 Map, Model, ModelMap.
- @RequestMapping method가 return하는 ModelAndView.
- @ModelAttribute annotation이 적용된 method가 return 한 객체.



# 6. MVC(Model-View-Controller)

## 6-6. Model 생성

- ✓ Map, Model, ModelMap을 통한 설정.
  - parameter로 받는 방식.

```
@Controller  
public class HomeController {  
  
    @RequestMapping("/hello.do")  
    public String hello(Map model) {  
        model.put("message", "안녕하세요");  
        return "hello";  
    }  
}
```

```
@Controller  
public class HomeController {  
  
    @RequestMapping("/hello.do")  
    public String hello(Model model) {  
        model.addAttribute("message", "안녕하세요");  
        return "hello";  
    }  
}
```



## 6. MVC(Model-View-Controller)

### 6-6. Model 생성

#### ✓ Model Interface.

- Model addAttribute(String name, Object value);
- Model addAttribute(Object value);
- Model addAllAttributes(Collection<?> values);
- Model addAllAttributes(Map<String, ?> attributes);
- Model mergeAttributes(Map<String, ?> attributes);
- boolean containsAttribute(String name);



# 6. MVC(Model-View-Controller)

## 6-6. Model 생성

### ✓ ModelAndView를 통한 Model 설정.

- Controller에서 처리결과를 보여줄 view와 view에 전달할 값(model)을 저장하는 용도로 사용.
- setViewName(String viewname);
- addObject(String name, Object value);

```
@Controller  
public class HomeController {  
  
    @RequestMapping("/hello.do")  
    public ModelAndView hello() {  
        ModelAndView mav = new ModelAndView();  
        mav.setViewName("hello");  
        mav.addObject("message", "안녕하세요");  
        return mav;  
    }  
}
```



# 6. MVC(Model-View-Controller)

## 6-6. Model 생성

### ✓ @ModelAttribute annotation을 이용한 mode data 처리.

- @RequestMapping annotation이 적용되지 않은 별도 method로 모델이 추가될 객체를 생성.

```
@Controller  
public class HomeController {  
  
    @ModelAttribute("modelAttrMessage")  
    public String maMessage() {  
        return "bye bye~~";  
    }  
  
    @RequestMapping("/hello.do")  
    public String hello(Model model) {  
        model.addAttribute("message", "안녕하세요");  
        return "index";  
    }  
}
```

```
<div align="center">  
    <h3>hello spring example</h3>  
    ${message}<br>  
    ${modelAttrMessage}<br>  
    <a href="${root}/board/write.do">글쓰기</a>  
</div>
```



# 6. MVC(Model-View-Controller)

## 6-7. 요청 URI 매칭

### ✓ 전체 경로와 Servlet기반 경로 매칭.

- DispatcherServlet은 DefaultAnnotationHandlerMapping Class를 기본으로 HandlerMapping 구현체로 사용.
- Default로 Context 내의 경로가 아닌 Servlet 경로를 제외한 나머지 경로에 대해 mapping.

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>*.html</url-pattern>
    <url-pattern>/product/*</url-pattern>
</servlet-mapping>
```

```
@RequestMapping("/board/list.html")
@RequestMapping("/product/list") → 매칭 안됨(다음 페이지 해결)
```



## 6. MVC(Model-View-Controller)

### 6-7. 요청 URI 매칭

#### ✓ Servlet기반 경로 매칭.

- Servlet 경로를 포함한 전체 경로를 이용해서 매칭하려는 경우.
- @RequestMapping("/product/list")

```
<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">
    <property name="alwaysUseFullPath" value="true"/>
</bean>

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
    <property name="alwaysUseFullPath" value="true"/>
</bean>
```



# 6. MVC(Model-View-Controller)

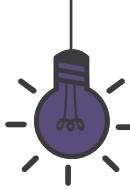
## 6-7. 요청 URI 매칭

- ✓ @PathVariable 어노테이션을 이용한 URI 템플릿.

- RESTful 방식
  - <http://localhost/users/troment>
  - <http://localhost/product/>
  - <http://localhost/forum/board1/10>
- @RequestMapping Annotation 값으로 {템플릿변수}를 사용.
- @ PathVariable Annotation을 이용해서 {템플릿변수}와 동일한 이름을 갖는 parameter를 추가.

```
@Controller
public class BoardViewController {
    @Autowired
    private BoardService boardService;

    @RequestMapping("/blog/{userId}/board1/{articleSeq}")
    public String viewArticle(@PathVariable String userId, @PathVariable int articleSeq, Model model) {
        BoardDto boardDto = boardService.getBlogArticle(userId, articleSeq);
        model.addAttribute("article", boardDto);
        return "view";
    }
}
```



## 6. MVC(Model-View-Controller)

### 6-7. 요청 URI 매칭

- ✓ **@RequestMapping Annotation의 추가설정 방법(1/2).**
  - @RequestMapping Annotation을 class와 method에 함께 적용하는 경우.

```
@Controller
@RequestMapping("/blog/{userId}")
public class BoardViewController {
    @Autowired
    private BoardService boardService;

    @RequestMapping("/board1/{articleSeq}")
    public String viewArticle(@PathVariable String userId, @PathVariable int articleSeq, Model model) {
        BoardDto boardDto = boardService.getBlogArticle(userId, articleSeq);
        model.addAttribute("article", boardDto);
        return "view";
    }
}
```



## 6. MVC(Model-View-Controller)

### 6-7. 요청 URI 매칭

#### ✓ @RequestMapping Annotation의 추가설정 방법(2/2).

- Ant 스타일의 URI패턴 지원.
  - ? : 하나의 문자열과 대치.
  - \* : 하나 이상의 문자열과 대치.
  - \*\* : 하나 이상의 디렉토리와 대치.

```
@RequestMapping("/members/*.do")
@RequestMapping("/blog/*/board/{articleSeq}")
```



# 6. MVC(Model-View-Controller)

## 6-8. 유효성검사(Validation)

### ✓ 폼 입력 값 검증.

- @ModelAttribute로 binding된 Object를 검증하는데 사용.
- Spring은 유효성 검사를 위한 Validator Interface와 검사 결과를 저장 할 Errors Interface를 제공.
- Validator interface를 이용한 폼 입력 값 검증.

```
import org.springframework.validation.Errors;

public interface Validator {

    // 해당 클래스 validation 지원 여부
    boolean supports(Class<?> claszz);

    // 검증 결과 문제가 있는 경우 error 객체에 정보를 저장
    void validate(Object target, Errors errors);

}
```



# 6. MVC(Model-View-Controller)

## 6-8. 유효성검사(Validation)

### ✓ 폼 입력 값 검증.

- Validator 구현.
  - 게시글 입력 값 검사 후 통과하지 못한 경우 페이지에 오류 메시지를 보여주도록 구현.

```
public class BoardArticleValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return BoardDto.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        BoardDto boardDto = (BoardDto) target;

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "subject", "required", "제목은 반드시 입력!!!");

        if(boardDto.getContent() == null || boardDto.getContent().length() < 10 )
            errors.rejectValue("content", "lengthsize", new Object[]{10}, "내용은 10자 이상!!!");
    }
}
```



# 6. MVC(Model-View-Controller)

## 6-8. 유효성검사(Validation)

### ✓ **@Valid annotation과 @InitBinder annotation을 이용한 검증 실행.**

- **@Valid** : 스프링 프레임워크가 validation을 호출하도록 annotation을 이용하여 설정함.
- **@InitBinder** : Validator를 바인딩.

```
@Controller
@RequestMapping("/board")
public class BoardController {

    @RequestMapping(value = "/write.do", method = RequestMethod.GET)
    public String write() {
        return "board/write";
    }

    @RequestMapping(value = "/write.do", method = RequestMethod.POST)
    public String write(@ModelAttribute("article") @Valid BoardDto boardDto, BindingResult result) {
        if(result.hasErrors())
            return "board/write";
        return "board/writeok";
    }

    @InitBinder
    protected void initBinder(WebDataBinder binder){
        binder.setValidator(new BoardArticleValidator());
    }
}
```

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
```

pom.xml에 추가



# 6. MVC(Model-View-Controller)

## 6-8. 유효성검사(Validation)

### ✓ Validation 결과 출력.

- Spring 커스텀 태그를 사용.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<c:set var="root" value="${pageContext.request.contextPath}" />
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<spring:hasBindErrors name="article"/>
<form method="POST" action="${root}/board/write.do">
제목 : <input type="text" name="subject"><form:errors path="article.subject"/><br>
내용 : <textarea name="content"></textarea><form:errors path="article.content"/><br>
<input type="submit" value="글쓰기"/>
</form>
</body>
</html>
```

커스텀 태그를 이용하여 여러 정보를 설정



## 6. MVC(Model-View-Controller)

### 6-9. HandlerInterceptor를 통한 요청 가로채기

- ✓ Controller가 요청을 처리하기 전/후 처리.
- ✓ 로깅, 모니터링 정보 수집, 접근제어 처리 등의 실제 BusinessLogic과는 분리되어 처리해야 하는 기능들을 넣고 싶을 때 유용함.
- ✓ interceptor를 여러 개 설정 할 수 있음(순서주의!!).



## 6. MVC(Model-View-Controller)

### 6-9. HandlerInterceptor를 통한 요청 가로채기

- ✓ HandlerInterceptor 제공 method.

#### HandlerInterceptor method

```
boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
```

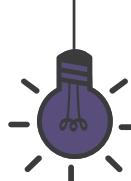
- false를 반환하면 request를 바로 종료.

```
void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)
```

- Controller 수행 후 호출.

```
void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)
```

- view를 통해 클라이언트에 응답을 전송한 뒤 실행.
- 예외가 발생하여도 실행.



## 6. MVC(Model-View-Controller)

### 6-9. HandlerInterceptor를 통한 요청 가로채기

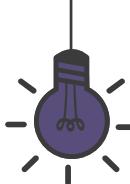
- ✓ **HandlerInterceptor** 인터페이스 구현.
- ✓ **HandlerInterceptorAdaptor** 클래스 제공.

```
public class LoggingInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        System.out.println("intercept!! preHandle");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        System.out.println("intercept!! postHandle");
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
        System.out.println("intercept!! afterCompletion");
    }
}
```



## 6. MVC(Model-View-Controller)

### 6-9. HandlerInterceptor를 통한 요청 가로채기

- ✓ **Interceptor 설정하기. : servlet-context.xml**

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="*.html"/>
    <bean class="com.test.hello.LoggingInterceptor"/>
  </mvc:interceptor>
</mvc:interceptors>
```

- ✓ 위의 경우 요청 처리시 interceptor의 preHandle, postHandle, afterCompletion 함수의 호출 순서는?
  - Controller method 전/후/응답 완료 후 호출됨을 확인.

```
정보: Server startup in 1754 ms
intercept!! preHandle
intercept!! postHandle
intercept!! afterCompletion
```



## 6. MVC(Model-View-Controller)

### 6-9. HandlerInterceptor를 통한 요청 가로채기

#### ✓ 여러 개의 interceptor 등록.

- EtcInterceptor.java interceptor 추가.

```
public class EtcInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        System.out.println("EtcInterceptor!! preHandle");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        System.out.println("EtcInterceptor!! postHandle");
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
        System.out.println("EtcInterceptor!! afterCompletion");
    }
}
```



# 6. MVC(Model-View-Controller)

## 6-9. HandlerInterceptor를 통한 요청 가로채기

- ## ✓ Interceptor 설정하기. : servlet-context.xml

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="*.html"/>
        <bean class="com.test.hello.LoggingInterceptor"/>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="*.html"/>
        <bean class="com.test.hello.EtcInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

- ✓ 위의 경우 요청 처리시 interceptor의 preHandle, postHandle, afterCompletion 함수의 호출 순서는?

- Interceptor 2개 등록 수행 결과(순서 확인!!!).

```
정보: Server startup in 1788 ms
LoggingInterceptor!! preHandle
EtcInterceptor!! preHandle
    >>>>>>>>>>>>> hello method call! <<<<<<<<<<<<<<<
EtcInterceptor!! postHandle
LoggingInterceptor!! postHandle
EtcInterceptor!! afterCompletion
LoggingInterceptor!! afterCompletion
```



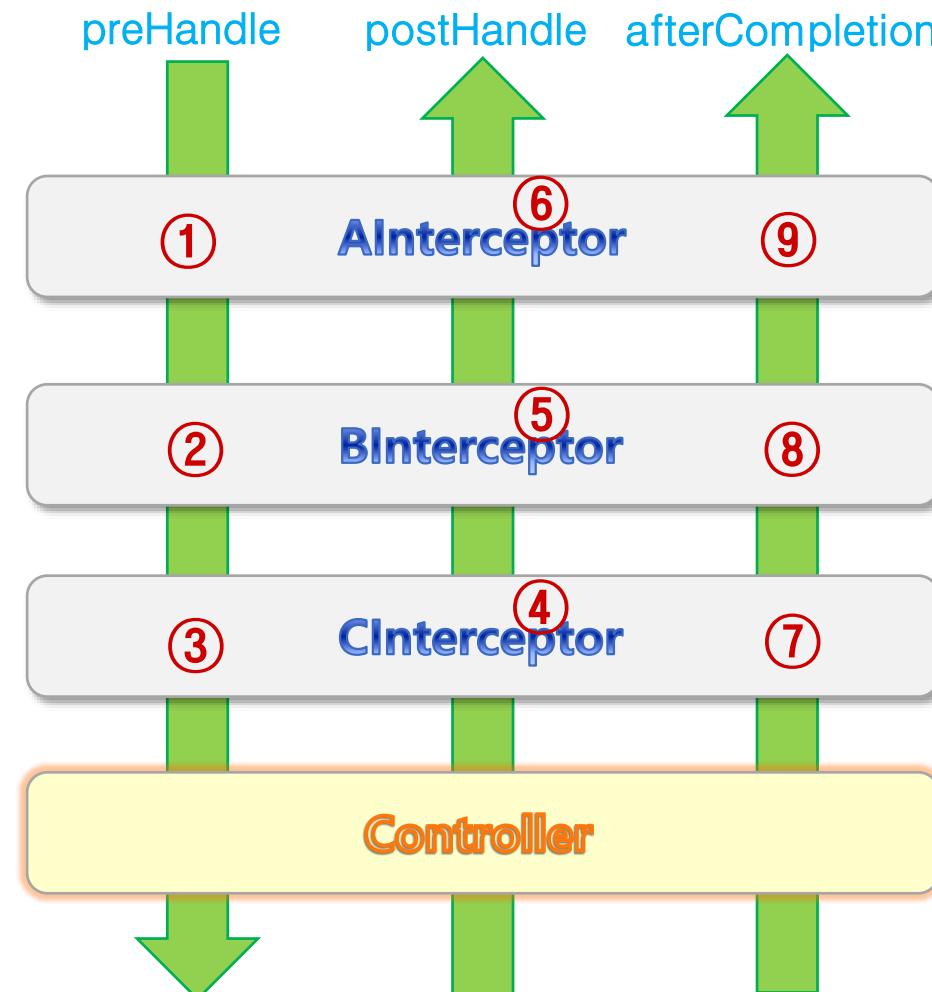
# 6. MVC(Model-View-Controller)

## 6-9. HandlerInterceptor를 통한 요청 가로채기

- ✓ **Interceptor 호출 순서.**

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="*.html"/>
        <bean class="com.test.hello.AInterceptor"/>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="*.html"/>
        <bean class="com.test.hello.BInterceptor"/>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="*.html"/>
        <bean class="com.test.hello.CInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

servlet-context.xml





## 6. MVC(Model-View-Controller)

### 6-10. Exception Handling

#### ✓ **@RequestMapping** 메서드는 모든 타입의 예외를 발생시킬 수 있음.

- WebBrowser에는 500 응답코드와 Servlet Container가 출력한 에러 페이지 출력.
- SpringMVC를 이용해서 원하는 에러 페이지를 보여주고 싶다면 SpringMVC가 제공하는 HandlerExceptionResolver 인터페이스를 사용한다.
- SpringMVC가 제공하는 HandlerExceptionResolver 인터페이스의 구현체.
  - AnnotationMethodHandlerExceptionResolver
    - @ExceptionHandler 어노테이션이 적용된 메서드를 이용.
  - DefaultHandlerExceptionResolver
    - 스프링 관련 예외 타입 처리.
  - SimpleMappingExceptionResolver
    - 예외 타입 별로 뷰 이름을 지정.
  - ResponseStatusExceptionResolver
    - 예외를 특정 HTTP 응답 상태코드로 전환하여 단순한 500에러가 아닌 의미 있는 HTTP 응답상태를 반환하는 방법.



## 6. MVC(Model-View-Controller)

### 6-10. Exception Handling

#### ✓ **@ExceptionHandler** 어노테이션을 이용한 예외처리.

- parameter로 받을 수 있는 type.
  - HttpServletRequest, HttpServletResponse, HttpSession
  - Locale, InputStream / Reader, OutputStream / Writer
  - 예외타입
- return type.
  - ModelAndView, Model, Map, View, String, void

```
@Controller
public class HomeController {

    @RequestMapping("/user/{userId}")
    public ModelAndView getUser(@PathVariable String userId) {
        ModelAndView mav = new ModelAndView();

        return mav;
    }

    @ExceptionHandler(NullPointerException.class)
    public String handleNullPointerException(NullPointerException e) {
        return "error/nullException";
    }
}
```



## 6. MVC(Model-View-Controller)

### 6-10. Exception Handling

- ✓ **SimpleMappingExceptionResolver** 클래스를 이용한 에러 페이지 지정.

- 예외 타입별로 에러 페이지를 지정할 수 있음.

```
<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <prop key="java.lang.NullPointerException">nullerror</prop>
            <prop key="java.lang.Exception">error</prop>
        </props>
    </property>
</bean>
```



## 6. MVC(Model-View-Controller)

### 6-10. Exception Handling

#### ✓ Error page.

- nullerror.jsp

```
<%@ page language="java" contentType="text/html; charset=EUC-KR"
    pageEncoding="EUC-KR"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">
<title>Insert title here</title>
</head>
<body>
<center>
<h1>Error 발생!!!!</h1>
<font color="red">
${errorMessage}
</font>
</center>
</body>
</html>
```



## 6. MVC(Model-View-Controller)

### 6-11. @RequestBody, @ResponseBody

- ✓ **HttpMessageConverter를 이용한 변환 처리.**

- 주요 HttpMessageConverter 구현 Class.

AnnotationMethodHandlerAdapter는  
(\*) 표시된 클래스를 기본적으로 사용

구현클래스	설명
ByteArrayHttpMessageConverter (*)	HTTP 메시지와 byte 배열 사이의 변환을 처리. 컨텐츠 타입은 application/octet-stream
StringHttpMessageConverter (*)	HTTP 메시지와 String 사이의 변환을 처리. 컨텐츠 타입은 text/plain;charset=ISO-8859-1
FormHttpMessageConverter (*)	HTML 폼 데이터를 MultiValueMap으로 전달받을 때 사용. 컨텐츠 타입은 application/x-www-form-urlencoded
SourceHttpMessageConverter (*)	HTTP 메시지와 javax.xml.transform.Source 사이의 변환을 처리. 컨텐츠 타입은 application/xml 또는 text/xml
MarshallingHttpMessageConverter	스프링의 Marshaller와 Unmarshaller를 이용해서 XML HTTP 메시지와 객체 사이의 변환을 처리. 컨텐츠 타입은 application/xml 또는 text/xml
MappingJacksonHttpMessageConverter	Jackson 라이브러리를 이용해서 JSON HTTP 메시지와 객체 사이의 변환 처리. 컨텐츠 타입은 application/json



## 6. MVC(Model-View-Controller)

### 6-11. @RequestBody, @ResponseBody

- ✓ Content-Type과 Accept헤더 기반의 변환처리.
  - @AnnotationMethodHandlerAdapter가 HttpMessageConverter를 이용해서 요청 몸체 데이터를 @RequestBody Annotation이 적용된 자바 객체로 변환 할 때에는 HTTP요청헤더의 Content-Type헤더에 명시된 미디어 타입(MIME)을 지원하는 HttpMessageConverter를 구현체로 사용.
  - @ResponseBody Annotation을 이용해서 리턴하는 객체를 HTTP 메시지의 body로 변환할 때에는 HTTP요청 헤더의 Accept 헤더에 명시된 미디어타입을 지원하는 HttpMessageConverter 구현체를 선택.



## 6. MVC(Model-View-Controller)

### 6-11. @RequestBody, @ResponseBody

- ✓ Content-Type과 Accept header 기반의 변환 처리.

```
<script type="text/javascript">

xmlhttp.open("GET", "json.html", true);
xmlhttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xmlhttp.setRequestHeader("Accept", "application/json");

xmlhttp.send();

</script>
```



json타입의 response를 accept함

```
@RequestMapping(value="boardmemo/json.html", method=RequestMethod.GET, headers="accept=application/json")
@ResponseBody
public BoardMemoList listJson() {
    [
        return null;
}
```



## 6. MVC(Model-View-Controller)

### 6-11. @RequestBody, @ResponseBody

✓ WebSystem 간의 XML, JSON 형식의 Data를 주고받는 경우.

- XML, JSON → Java Object (unmarshalling)
- Java Object → XML, JSON (marshalling)

✓ @RequestBody, @ResponseBody Annotation은 HTTP 메시지 Body에 Java 객체를 XML이나 JSON등의 타입으로 변환하여 담고, 역으로 변환하는데 사용.

#### • marshalling

한 객체의 메모리에서의 표현방식을 저장 또는 전송에 적합한 다른 데이터 형식으로 변환하는 과정이다.

또한 이는 데이터를 컴퓨터 프로그램의 서로 다른 부분 간에 혹은 한 프로그램에서 다른 프로그램으로 이동해야 할 때도 사용된다.

이는 대체로 어떤 한 언어로 작성된 프로그램의 출력 매개변수들을, 다른 언어로 작성된 프로그램의 입력으로 전달해야 하는 경우에 필요하다.

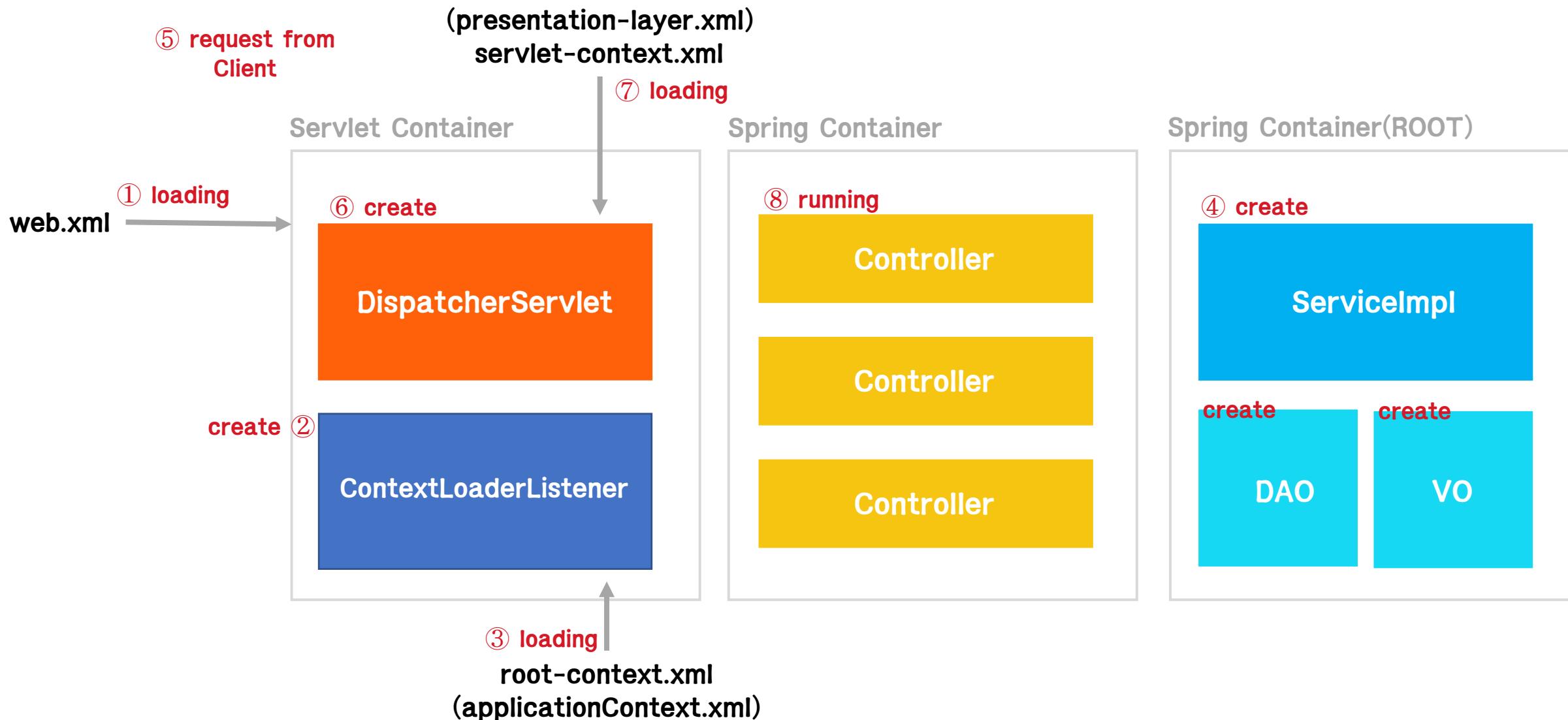
마샬링은 직렬화(serialization)와 유사하며 동일하게 간주되기도 하지만 매개변수를 바이트 스트림으로 변환하는 직렬화와는 차이가 있다.

참조 : 위키백과, 템즈



# 6. MVC(Model-View-Controller)

## 6-12. WebApplication 동작원리





# 6. MVC(Model-View-Controller)

## 6-12. WebApplication 동작원리

### ✓ 실행 순서

1. 웹 어플리케이션이 실행되면 Tomcat(WAS)에 의해 web.xml이 loading.
2. web.xml에 등록되어 있는 ContextLoaderListener (Java Class)가 생성.

ContextLoaderListener class는 ServletContextListener interface를 구현하고 있으며, ApplicationContext를 생성하는 역할을 수행.

3. 생성된 ContextLoaderListener는 root-context.xml을 loading.
4. root-context.xml에 등록되어 있는 Spring Container가 구동. 이 때 개발자가 작성한 Business Logic(Service)에 대한 부분과 Database Logic(DAO), VO 객체들이 생성.
5. Client로 부터 요청(request)가 들어옴.
6. DispatcherServlet(Servlet)이 생성. DispatcherServlet은 FrontController의 역할을 수행.

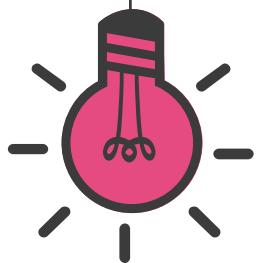
Client로부터 요청 온 메시지를 분석하여 알맞은 PageController에게 전달하고 응답을 받아 요청에 따른 응답을 어떻게 할 지 결정. 실질적인 작업은 PageController에서 이루어 진다.

이러한 클래스들을 HandlerMapping, ViewResolver Class라고 함.

7. DispatcherServlet은 servlet-context.xml을 loading.
8. 두 번째 Spring Container가 구동되며 응답에 맞는 PageController들이 동작. 이 때 첫 번째 Spring Container가 구동되면서 생성된 DAO, VO, ServiceImpl 클래스들과 협업하여 알맞은 작업을 처리.

07

Spring 기타





# 7. RESTful API

## 7-1. OPEN API

### ✓ OPEN API?? (Application Programming Interface)

- OPEN API는 프로그래밍에서 사용할 수 있는 개방되어 있는 상태의 Interface.
- Naver, Daum등 포털 서비스 사이트나 통계청, 기상청, 우체국 등과 같은 관공서, 공공 데이터 포털 (<https://www.data.go.kr/>)이 가지고 있는 데이터를 외부 응용 프로그램에서 사용할 수 있도록 OPEN API를 제공하고 있다.
- OPEN API와 함께 거론되는 기술이 REST이며, 대부분의 OPEN API는 REST방식으로 지원.



# 7. RESTful API

## 7-2. REST

### ✓ REST(Representational State Transfer).

- 2000년도 로이 필딩(Roy Fielding)의 박사학위 논문에 최초로 소개.
- 웹의 장점을 최대한 활용할 수 있는 아키텍처(설계구조)로써 REST를 발표.
- HTTP URI를 통해 제어할 자원(Resource)을 명시하고, HTTP Method(GET, POST, PUT, DELETE)을 통해 해당 자원(Resource)을 제어하는 명령을 내리는 방식의 아키텍처.

### ✓ REST 구성.

- 자원 (Resource). – URI
- 행위 (Verb). – HTTP Method
- 표현 (Representations).



## 7. RESTful API

### 7-1. 별첨 참조

✓ 간의.

- 자바



## 7. RESTful API

### 7-1. 별첨 참조

✓ 간의.

- 자바

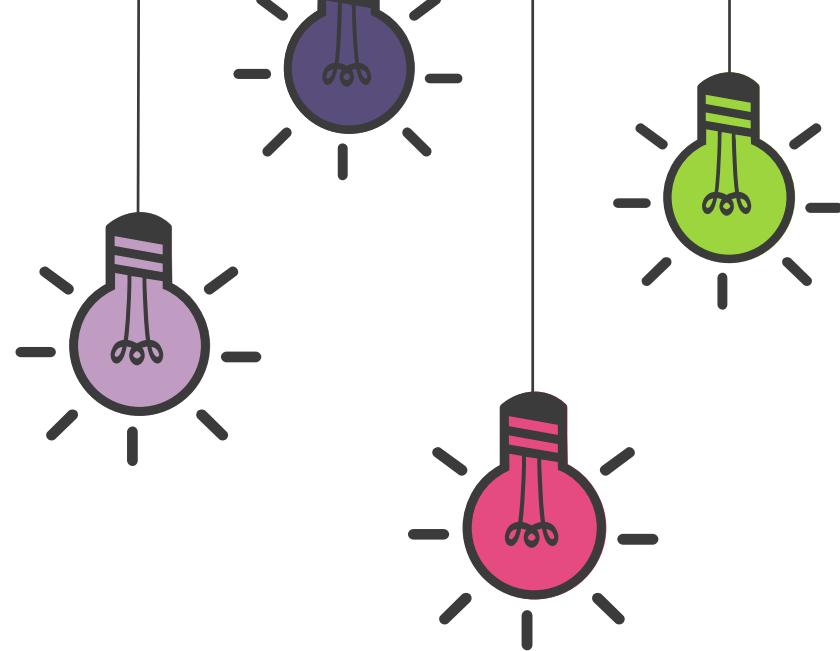


## 7. RESTful API

### 7-1. 별첨 참조

✓ 간의.

- 자바



THANK YOU FOR WATCHING

감사합니다



안효인  
troment@nate.com