

01.SpringFramework.....1
 02.IoC.....12
 03.Mvc.....30
 04.Aop.....52
 05.Log.....62



Spring Framework

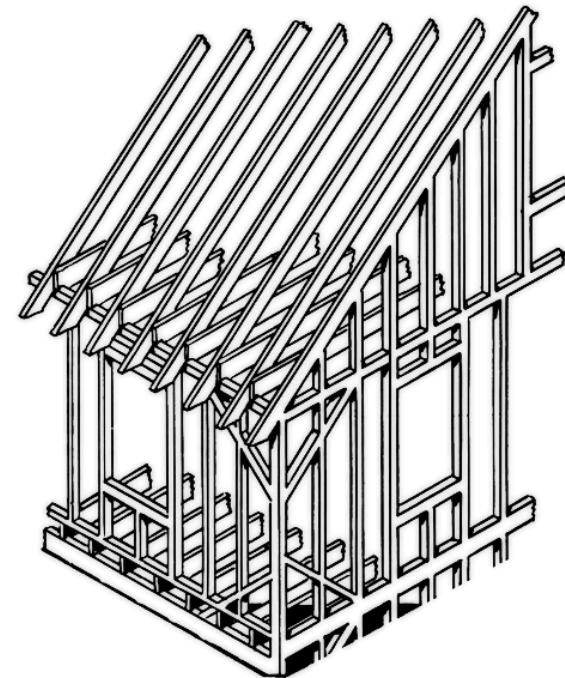
# Framework

- Framework

- ✓ Framework를 한글로 번역하면 '뼈대'라고 함
- ✓ 애플리케이션의 기본 구조를 제공하는 것이 Framework임
- ✓ Framework는 애플리케이션을 개발할 때 사용할 수 있도록 미리 만들어 놓은 클래스나 인터페이스 등을 제공함
- ✓ 스트러츠, 스프링과 같은 Framework들이 있음

- Framework 장점

1. 처음부터 다 만들 필요가 없다.
  - 이미 만들어진 기능을 가져다 사용하므로 효율성이 높고 품질이 보장됨
  - 개발 시간이 짧아짐
2. 개발 방법이 정해져 있다.
  - 개발의 표준화로 인해서 생산성이 높아짐
  - 개발 후 유지보수 및 기능의 확장이 용이함
  - 신입 개발자도 경력 개발자처럼 세련된 코드를 작성할 수 있음

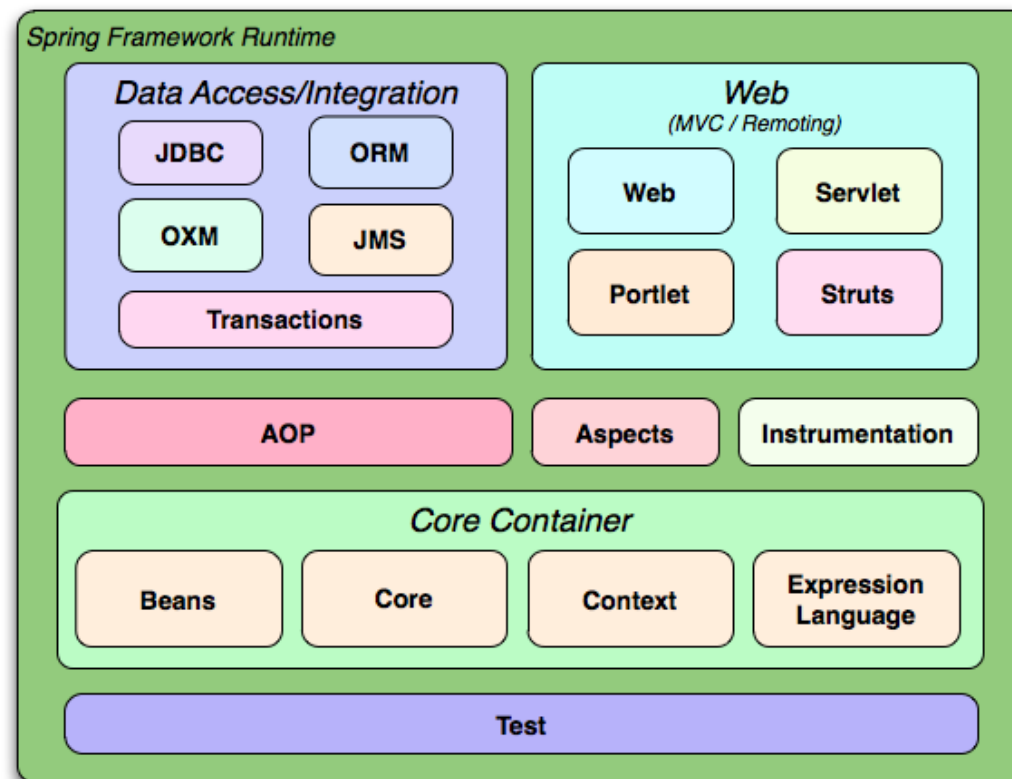


기본 틀을 제공하는 Framework

# Spring Framework

- Spring Framework
  - ✓ 2004년 로드 존슨(Rod Johnson)이 만든 오픈소스 프레임워크
  - ✓ 가장 널리 사용하는 프레임워크 중 하나
  - ✓ 자바 기반의 엔터프라이즈 애플리케이션 개발을 담당하던 EJB(Enterprise JavaBeans)를 대체함

- Spring Framework 구조 (출처 : <https://docs.spring.io/>)



# Spring Framework 특징

- Spring Framework 특징

1. 경량(Lightweight) 프레임워크

- 1) EJB에 비해 가벼운 경량 프레임워크(Lightweight Framework)
- 2) 몇 개의 모듈과 JAR 파일로 구성

2. 컨테이너(Container) 지원

- 1) 특정 객체의 생성과 관리를 담당하며 객체 운용에 필요한 다양한 기능을 제공
- 2) Singleton, Prototype 등 여러 형태의 객체 운용 가능

3. 제어의 역행(IOC)

- 1) 객체 생성을 개발자가 대신 컨테이너가 담당
- 2) 애플리케이션을 구성하는 객체 간 결합을 느슨한 결합(낮은 결합도)으로 유지

4. 관점지향 프로그래밍(AOP)

- 1) 비즈니스 메소드를 구성하는 공통 모듈과 핵심 모듈을 분리하여 작성
- 2) 모든 비즈니스 메소드가 반복해서 사용하는 공통 로직을 핵심 비즈니스 로직과 구분하여 작성

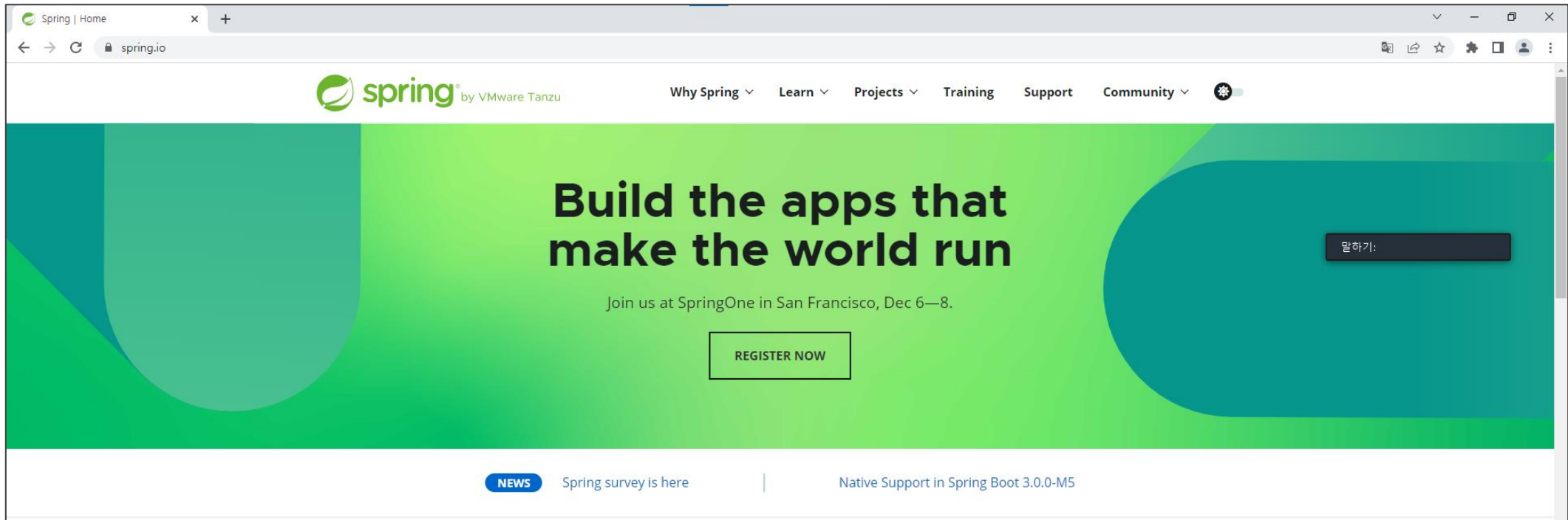
# Spring Framework 모듈

- Spring Framework 모듈

모듈	설명
spring-beans	스프링 컨테이너를 이용해서 객체를 생성하는 기본 기능 제공
spring-context	객체 생성, 라이프 사이클 처리, 스키마 확장과 같은 기능 제공
spring-aop	AOP 기능 제공
spring-web	REST 클라이언트, 데이터 변환 처리, 서블릿 필터, 파일 업로드 지원 등 웹 개발에 필요한 기반 기능을 제공
spring-webmvc	스프링 기반의 MVC 프레임워크 제공, 웹 애플리케이션을 개발하는데 필요한 컨트롤러, 뷰 구현을 제공
spring-websocket	스프링 MVC에서 웹 소켓 연동 처리
spring-tx	트랜잭션 처리를 위한 추상 레이어 제공
spring-jdbc	JDBC 프로그래밍을 보다 쉽게 할 수 있는 템플릿 제공
spring-orm	하이버네이트, JPA, MyBatis 등과의 연동 지원
spring-jms	JMS 서버와 메시지를 쉽게 주고 받을 수 있도록 하기 위한 템플릿, 애노테이션 제공
spring-context-support	스케줄링, 메일 발송, 캐시 연동, 벨로시티 등 부가 기능 제공

# Spring Framework

- Spring Framework 공식 홈페이지  
✓ <https://spring.io>



# Spring Framework 준비

- Spring Framework 준비
  - ✓ JDK ([www.oracle.com](http://www.oracle.com)) 설치 필요
  - ✓ Apache Tomcat Server ([tomcat.apache.org](http://tomcat.apache.org)) 설치 필요
  - ✓ STS ([spring.io](http://spring.io)) 또는 Eclipse ([www.eclipse.org](http://www.eclipse.org)) 설치 필요
- STS
  - ✓ STS, Spring Tool Suite
  - ✓ Spring Framework 사용을 위한 각종 라이브러리를 추가해 둔 IDE(Integrated Development Environment, 통합 개발 환경)
  - ✓ Eclipse와 거의 동일한 개발 환경을 제공함
  - ✓ STS3와 STS4 버전이 제공
    - ✓ STS3 : Spring Legacy Project 지원. 현재 더 이상 업데이트 되지 않음.
    - ✓ STS4 : Spring Boot Project 지원
  - ✓ 다운로드 링크
    - STS3 : <https://github.com/spring-attic/toolsuite-distribution/wiki/Spring-Tool-Suite-3>
    - STS4 : <https://spring.io/tools>



# 참고1. Eclipse에서 STS3 플러그인 설치하기

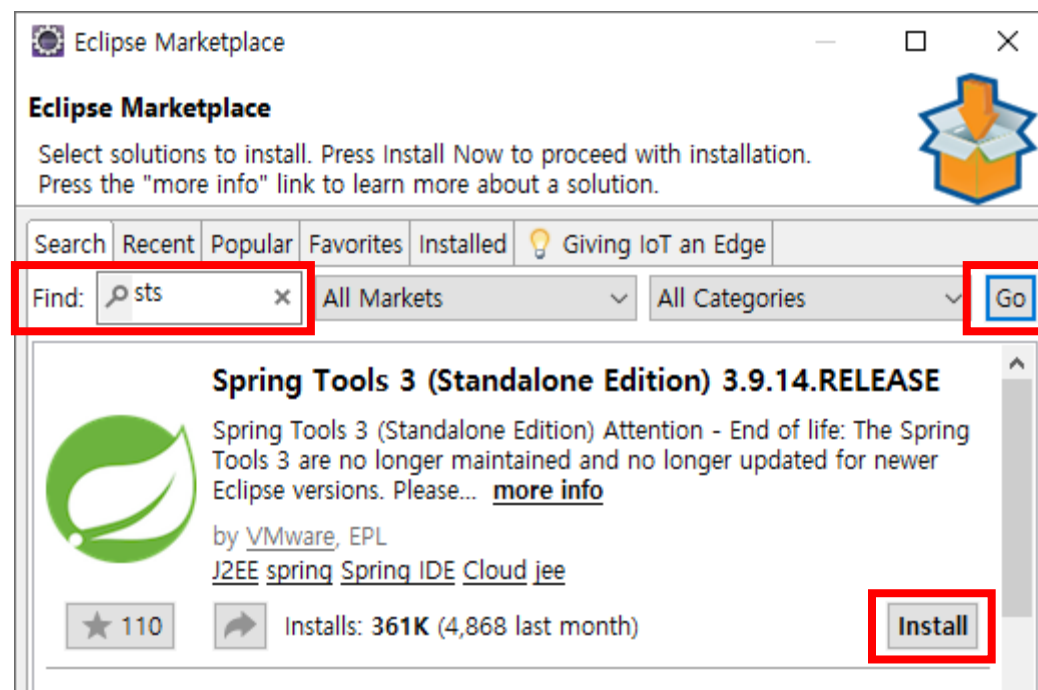
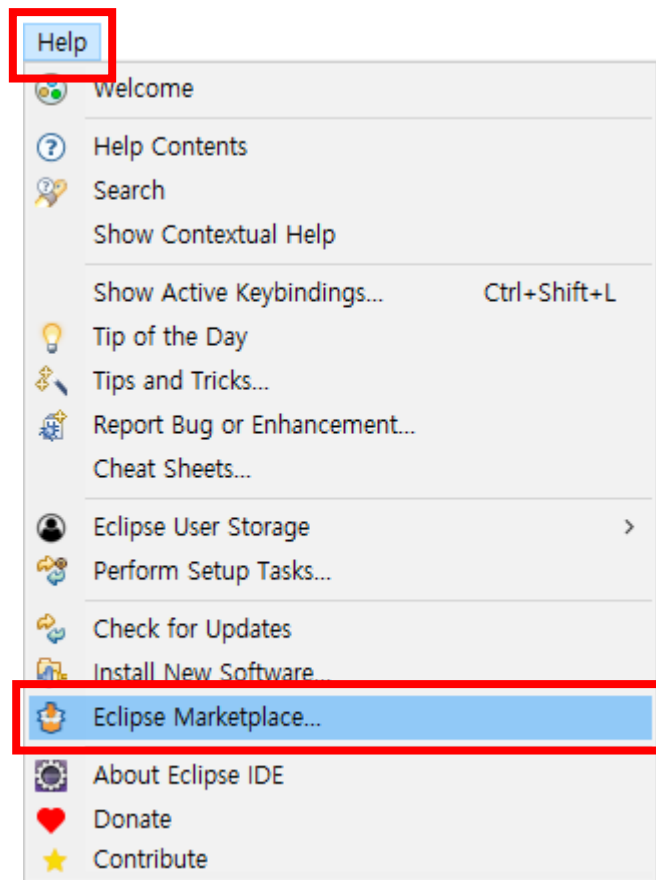
- 이클립스에서 플러그인 설치하기

- ✓ STS를 다운로드 받지 않고, 기존에 사용하던 이클립스에 플러그인을 설치할 수 있음

[Help] - [Eclipse Marketplace...]

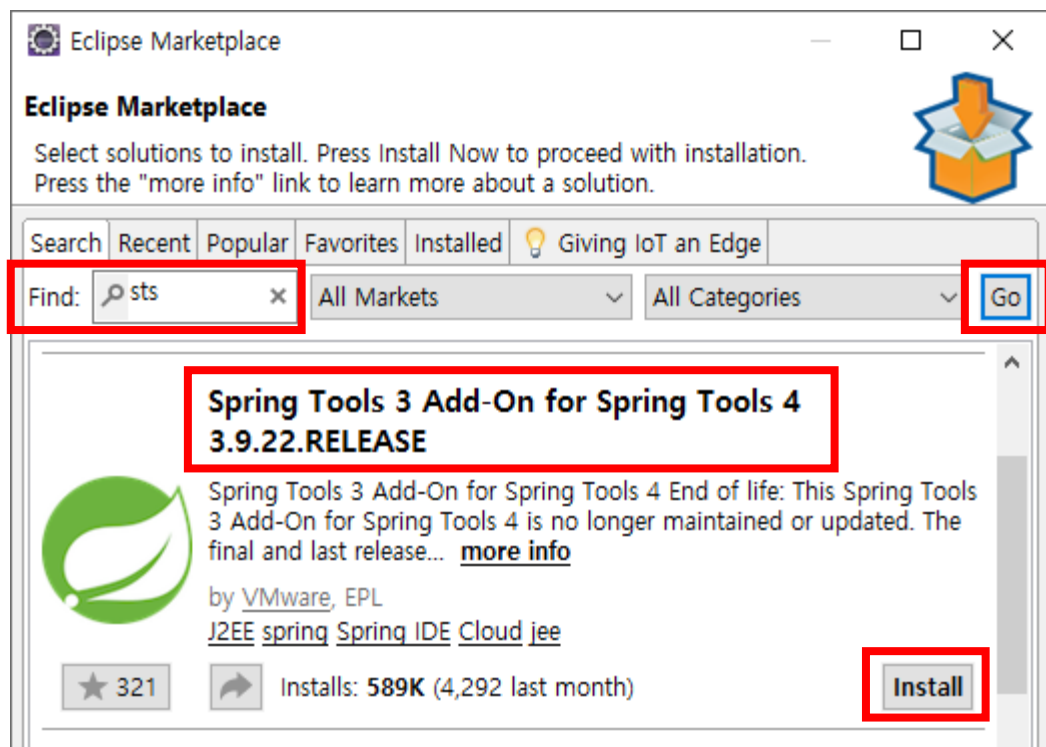
--->

sts 검색 후 Spring Tools 3 버전을 설치

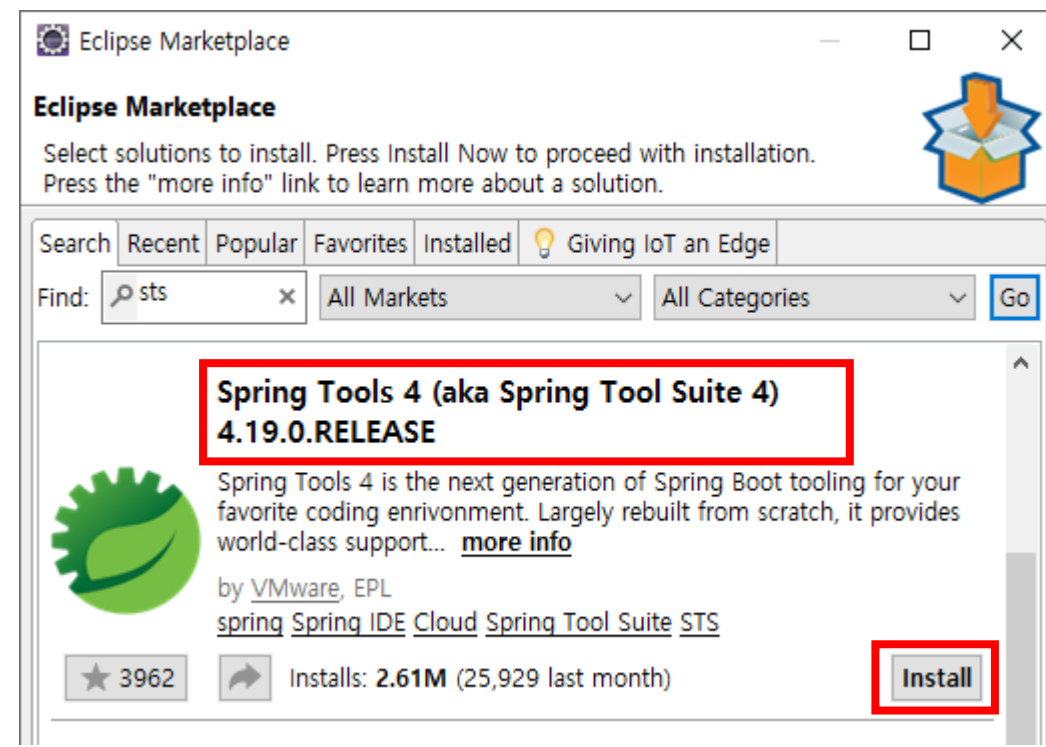


# 참고2. Eclipse에서 STS4 플러그인 설치하기

sts 검색 후 Spring Tools 3 Add-On 먼저 설치



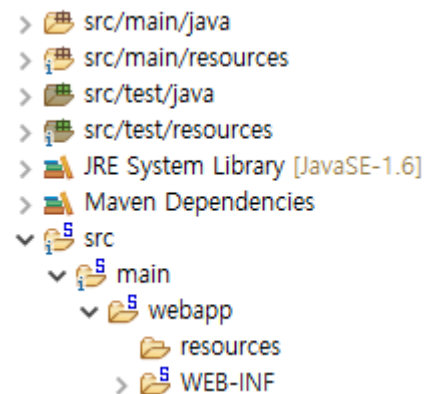
---> Spring Tools 4 버전을 설치



# Maven Project

- 메이븐 프로젝트 구조

1. src/main/java
2. src/main/resources
3. src/test/java
4. src/test/resources
5. src/main/webapp



- 메이븐 프로젝트 파일 배치

- |                       |  |
|-----------------------|--|
| 1. src/main/java      | ➤ 자바 소스 파일   |
| 2. src/main/resources | ➤ 리소스 파일(프로퍼티, XML 등)  |
| 3. src/test/java      | ➤ 자바 테스트 코드(단위 테스트, 통합 테스트)  |
| 4. src/test/resources | ➤ 자바 테스트 과정에 필요한 리소스 파일  |
| 5. src/main/webapp    | ➤ 웹 애플리케이션 관련 파일(resources 디렉터리(HTML/CSS/JS 등), WEB-INF 디렉터리(JSP 등)) |

# Maven Project

- pom.xml

1. 메이븐 프로젝트 정보 : 프로젝트 이름, 개발자 목록, 라이선스 정보 등
2. 빌드 설정 : 빌드 툴, 소스/리소스, 플러그인 등
3. 빌드 환경 : 사용자의 환경마다 달라질 수 있는 프로파일 정보
4. POM 연관 정보 : 의존 프로젝트, 상위 프로젝트, 하위 모듈 등

- pom.xml 주요 태그

1. <groupId>            ➤ 프로젝트 그룹 Id
2. <artifactId>        ➤ 프로젝트 아티팩트 Id (컨텍스트)
3. <name>                ➤ 프로젝트 이름
4. <packaging>        ➤ 패키징 타입 (war, jar)
5. <properties>        ➤ 버전 및 속성 관리
6. <dependencies>      ➤ 프로젝트가 참조하는 의존 프로젝트들
7. <dependency>        ➤ 개별 의존 프로젝트
8. <build>               ➤ 빌드에서 사용할 플러그인



IoC

# IoC

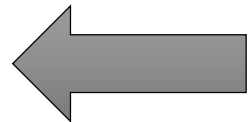
- IoC

- ✓ IoC, Inversion of Control
- ✓ 제어의 역전
- ✓ 개발자가 프로그램을 제어하지 않고, Framework가 프로그램을 제어하는 것을 의미함
- ✓ 객체 생성, 의존관계 설정(Dependency), 생명주기(Lifecycle) 등을 Framework가 직접 관리하는 것을 말함

- IoC 컨테이너

- ✓ 컨테이너(Container) : 객체의 생명주기를 관리하고 생성된 인스턴스를 관리함
- ✓ Spring Framework에서 객체를 생성과 소멸을 담당하고 의존성을 관리하는 컨테이너를 IoC 컨테이너라고 함
- ✓ IoC 컨테이너 = 스프링 컨테이너

컨테이너에 있는  
객체를 받아서 사용

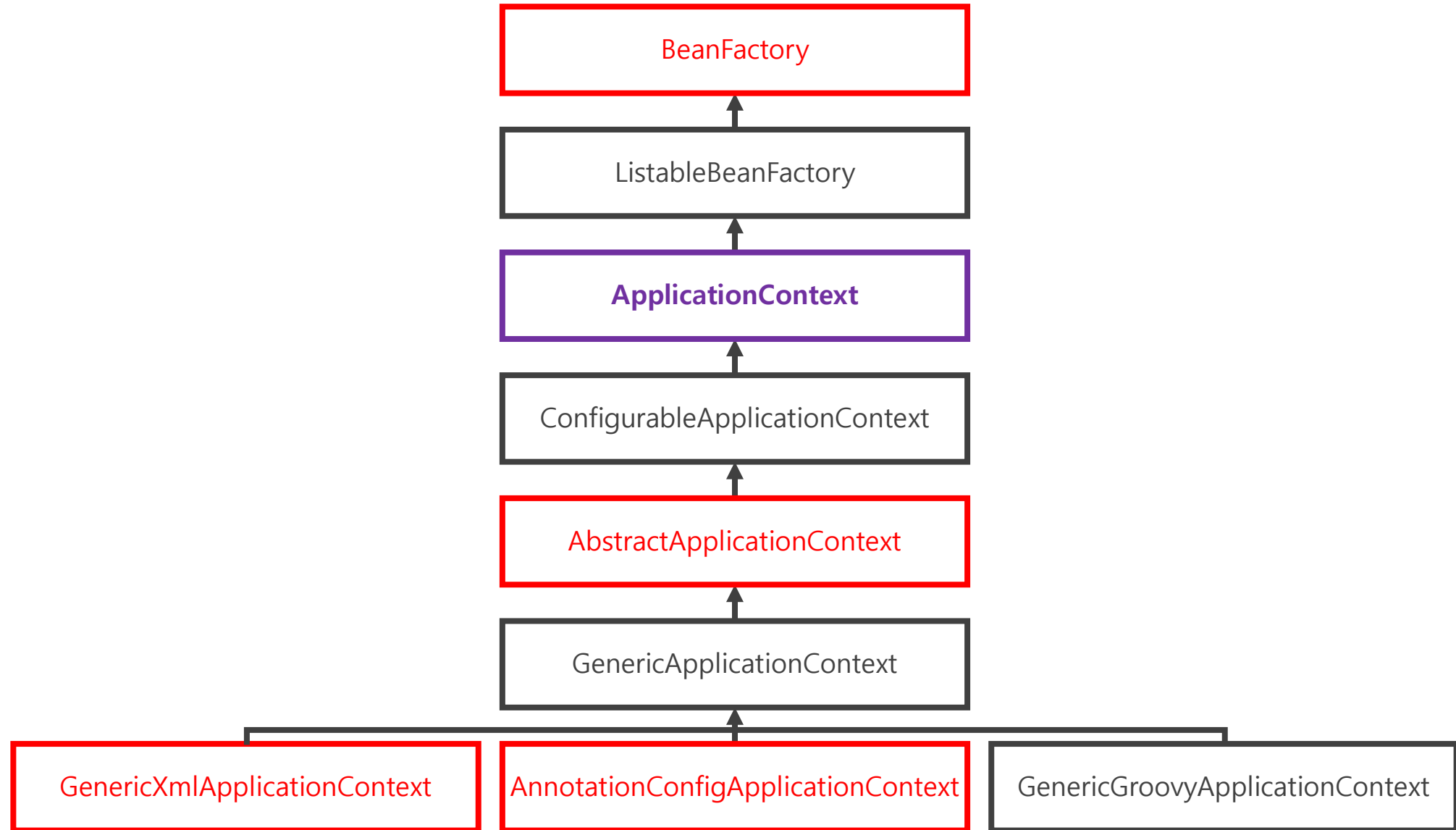


- ✓ getBean( )
- ✓ @Inject
- ✓ @Autowired
- ✓ ...



**IoC Container** (ApplicationContext)

# IoC 컨테이너 구조



# IoC 컨테이너 종류

- IoC 컨테이너 주요 종류

컨테이너	의미
BeanFactory	<ul style="list-style-type: none"><li>스프링 빈 설정 파일(Spring Bean Configuration File)에 등록된 bean을 생성하고 관리하는 가장 기본적인 컨테이너</li><li>클라이언트 요청에 의해서 bean을 생성함</li></ul>
ApplicationContext	<ul style="list-style-type: none"><li>트랜잭션 관리, 메시지 기반 다국어 처리 등 추가 기능을 제공</li><li>bean으로 등록된 클래스들을 객체 생성 즉시 로딩시키는 방식으로 동작</li></ul>
GenericXmlApplicationContext	<ul style="list-style-type: none"><li>파일 시스템 또는 클래스 경로에 있는 XML 설정 파일을 로딩하여 &lt;bean&gt; 태그로 등록된 bean을 생성하는 컨테이너</li></ul>
AnnotationConfigApplicationContext	<ul style="list-style-type: none"><li>자바 애너테이션(Java Annotation)에 의해서 bean으로 등록된 bean을 생성하는 컨테이너</li><li>@Configuration, @Bean 애너테이션 등이 필요함</li></ul>

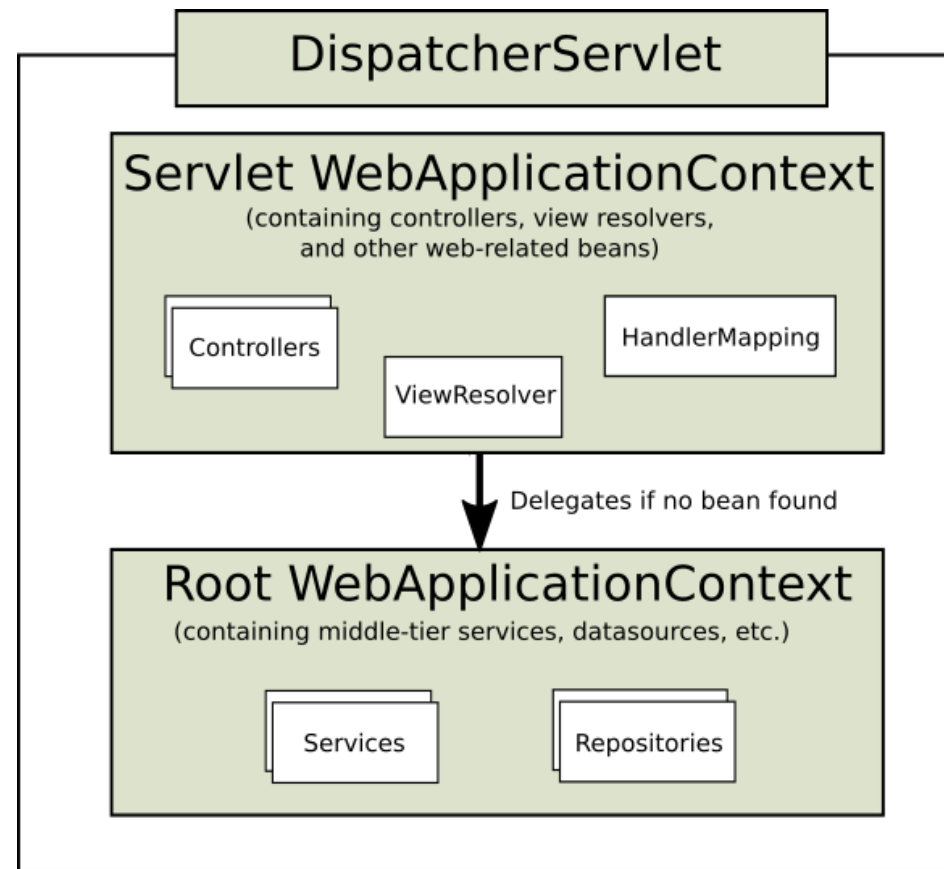


# XML 파일을 이용한 Bean 생성

- root-context.xml
  - ✓ View와 관련이 없는 객체를 정의
  - ✓ Service, Repository(DAO), DB 등과 관련된 Bean을 등록함
  - ✓ <bean> 태그를 이용해서 Bean을 등록
- servlet-context.xml
  - ✓ 요청에 관련된 객체를 정의
  - ✓ Controller, ViewResolver, Interceptor 등과 관련된 설정을 처리함
  - ✓ <beans:bean> 태그를 이용해서 Bean을 등록함

## Bean???

Spring Container에 의해서 관리되는  
POJO(Plain Old Java Object)를  
의미한다. 쉽게 말하면  
자바 객체가 곧 Bean이다.



(출처 : <https://docs.spring.io/>)

# <property> 태그

- <property> 태그
  - ✓ Setter를 이용해서 값을 전달하고 저장함
  - ✓ 작성방법-1

```
<property name="필드">
  <value>값</value>
</property>
```
  - ✓ 작성방법-2

```
<property name="필드" value="값" />
```

```
<bean id="address" class="com.group.project.Address">
  <property name="zipCode" value="12345" />
  <property name="jibunAddr" value="서울시 강남구 논현동" />
  <property name="roadAddr" value="서울시 강남구 강남대로" />
</bean>
```

<property> 태그는  
Setter를 이용해서 value를 전달한다.

*injection*

```
public class Address {
    private String zipCode;
    private String jibunAddr;
    private String roadAddr;
    public String getZipCode() {
        return zipCode;
    }
    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }
    public String getJibunAddr() {
        return jibunAddr;
    }
    public void setJibunAddr(String jibunAddr) {
        this.jibunAddr = jibunAddr;
    }
    public String getRoadAddr() {
        return roadAddr;
    }
    public void setRoadAddr(String roadAddr) {
        this.roadAddr = roadAddr;
    }
}
```

# <property> 태그

✓ 주의!

참조 타입의 값을 저장할 때는 value가 아닌 ref 속성을 사용해야 함

```
<bean id="contact" class="com.group.project.Contact">
  <property name="tel" value="010-1111-1111" />
  <property name="addr" ref="address" />
</bean>

<bean id="address" class="com.group.project.Address">
  <property name="zipCode" value="12345" />
  <property name="jibunAddr" value="서울시 강남구 논현동" />
  <property name="roadAddr" value="서울시 강남구 강남대로" />
</bean>
```

*injection*

```
public class Contact {
    private String tel;
    private Address addr;
    public String getTel() {
        return tel;
    }
    public void setTel(String tel) {
        this.tel = tel;
    }
    public Address getAddr() {
        return addr;
    }
    public void setAddr(Address addr) {
        this.addr = addr;
    }
}
```

특정 Bean을 값으로 전달할 때는  
ref 속성을 사용한다.

# <constructor-arg> 태그

- <constructor-arg> 태그

- ✓ Constructor를 이용해서 값을 전달하고 저장함
- ✓ 작성방법-1

```
<constructor-arg>
  <value>값</value>
</constructor-arg>
```
- ✓ 작성방법-2

```
<constructor-arg value="값" />
```

<constructor-arg> 태그는  
Constructor를 이용해서 value를 전달한다.

```
<bean id="address" class="com.group.project.Address">
  <constructor-arg value="12345" />
  <constructor-arg value="서울시 강남구 논현동" />
  <constructor-arg value="서울시 강남구 강남대로" />
</bean>
```

생성자의 매개변수와  
순서가 반드시 일치해야 한다.

```
public class Address {
    private String zipCode;
    private String jibunAddr;
    private String roadAddr;
    public Address(String zipCode, String jibunAddr, String roadAddr) {
        this.zipCode = zipCode;
        this.jibunAddr = jibunAddr;
        this.roadAddr = roadAddr;
    }
}
```

*injection*

# XML to Annotation

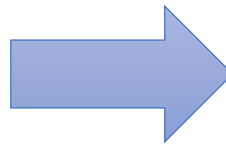
- XML을 이용한 Bean 생성의 한계
  - ✓ 프로젝트의 규모가 커지면서 XML의 관리가 어려워짐
  - ✓ XML에서는 자바 데이터를 사용하기가 불편함
  - ✓ 애노테이션을 이용하는 새로운 빈 생성 방식이 개발되었음
  - ✓ 기존 XML을 이용하는 방식은 애노테이션을 이용한 방식으로 대체되는 중임
  - ✓ Spring Boot 프로젝트는 빈 생성을 위한 root-context.xml이나 servlet-context.xml 파일을 기본적으로 지원하지 않고 있음

<xml>

<bean>

<beans:bean>

</xml>



@Annotation

@Configuration

@Bean

@Component

# Annotation을 이용한 Bean 생성

- @Configuration
  - ✓ 클래스 레벨 애노테이션
  - ✓ @Configuration이 명시된 클래스는 Bean으로 등록됨
  - ✓ @Bean을 등록하기 위해서는 반드시 @Configuration을 등록
- @Bean
  - ✓ 메소드 레벨 애노테이션
  - ✓ @Bean이 명시된 메소드가 반환하는 값이 Bean으로 등록됨
  - ✓ 개발자가 직접 Bean을 만들어서 반환하는 방식
  - ✓ 스프링은 메소드 이름을 Bean의 이름으로 등록함
  - ✓ @Bean을 사용하는 메소드가 포함된 클래스는 반드시 @Configuration을 사용해야 함
- @Component
  - ✓ 클래스 레벨 애노테이션
  - ✓ @Component가 명시된 클래스는 자동으로 Bean으로 등록됨
  - ✓ @Component가 명시된 클래스를 찾기 위해서는 반드시 미리 컴포넌트를 찾을 위치를 등록해야 하는데, 이를 Component Scan이라고 함
  - ✓ Spring Legacy Project는 servlet-context.xml에 Component Scan이 등록되어 있어 별도의 설정이 필요 없고, Spring Boot Project는 @SpringBootApplication에 Component Scan이 포함되어 있어 별도의 설정이 필요 없음
  - ✓ 스프링에서 가장 권장하는 Bean 생성 방식임

# @Component

- @Component

- ✓ @Component가 명시된 클래스는 스프링에 의해서 자동으로 Bean이 등록됨
- ✓ 스프링은 Component Scan이 등록되어 있어야만 @Component가 명시된 클래스를 찾을 수 있음

- Component Scan

- XML에서 Component Scan 등록

```
<context:component-scan base-package="spring.component.scan"/>
```

- Java에서 Component Scan 등록

```
@ComponentScan(basePackages = {"spring.component.scan"})
```

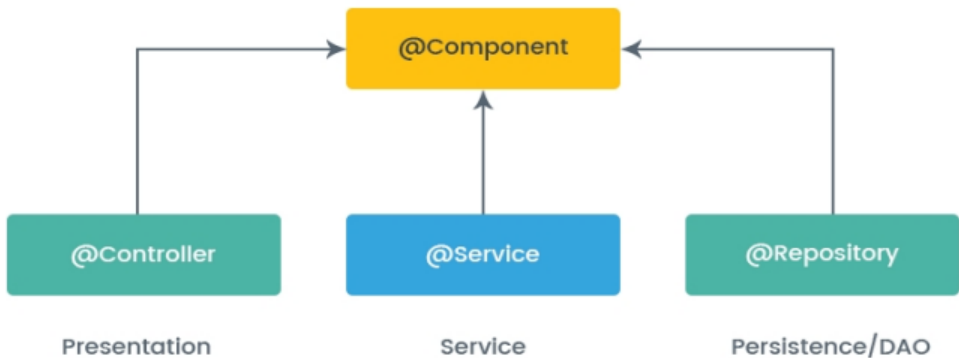
```
src/main/java  
└─ spring.component.scan  
    └─ Member.java
```

- 자동 등록 대상

```
@Component  
public class Member {  
  
    private String id;  
    private String pw;  
}
```

# @Component 계층 구조

- @Component의 계층 구조



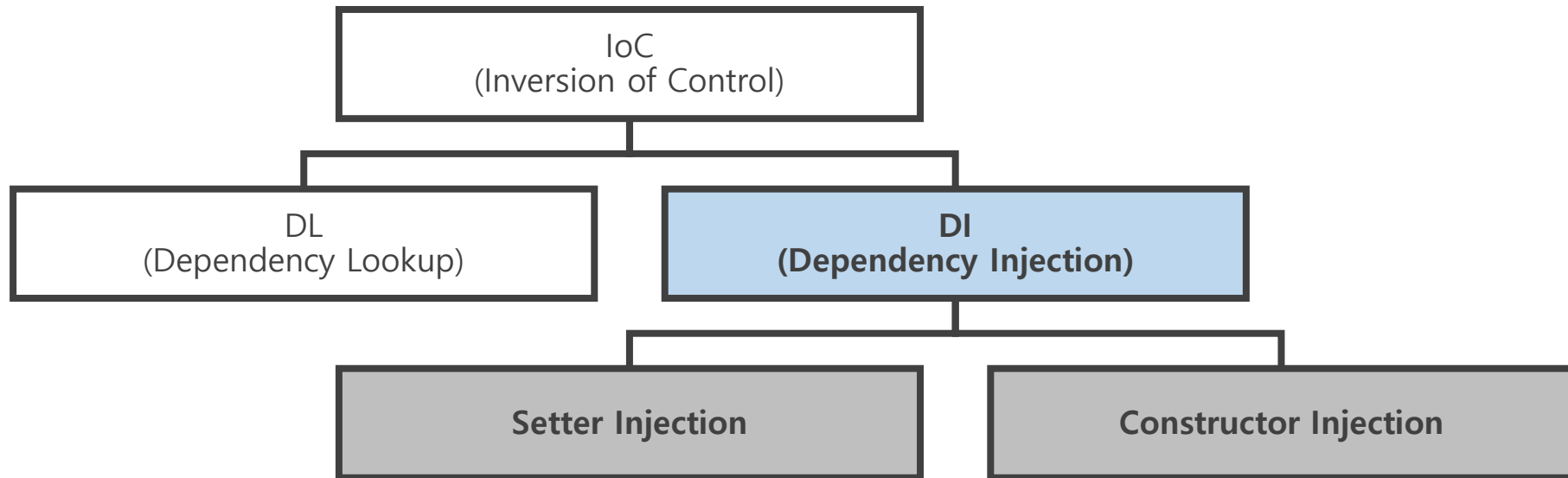
- 주요 애노테이션

애노테이션	의미
@Component	<ul style="list-style-type: none"><li>• 스테레오 타입의 최상의 객체</li></ul>
@Controller	<ul style="list-style-type: none"><li>• 요청과 응답을 처리하는 Controller 클래스에서 사용</li><li>• Spring MVC 아키텍처에서 자동으로 Controller로 인식됨</li></ul>
@Service	<ul style="list-style-type: none"><li>• 비즈니스 로직을 처리하는 Service 클래스에서 사용</li><li>• Service Interface를 구현하는 ServiceImpl 클래스에서 사용</li></ul>
@Repository	<ul style="list-style-type: none"><li>• 데이터베이스 접근 객체(DAO)에서 사용</li><li>• 데이터베이스 처리 과정에서 발생하는 예외를 변환해주는 기능을 포함함</li></ul>



# IoC 구현 방식

- IoC 구현 방식



- Dependency Lookup

- ✓ 컨테이너가 애플리케이션 운용에 필요한 객체를 생성하면 클라이언트는 컨테이너가 생성한 객체를 검색(Lookup)해서 사용하는 방식
- ✓ 실제 애플리케이션 개발에서 사용하지 않음

- Dependency Injection

- ✓ 객체 사이의 의존 관계를 컨테이너가 직접 설정하는 방식
- ✓ 스프링 빈 설정 파일에 등록된 정보를 바탕으로 컨테이너가 객체를 처리하는 방식으로 Spring Framework에서 주로 사용하는 방식

# DI

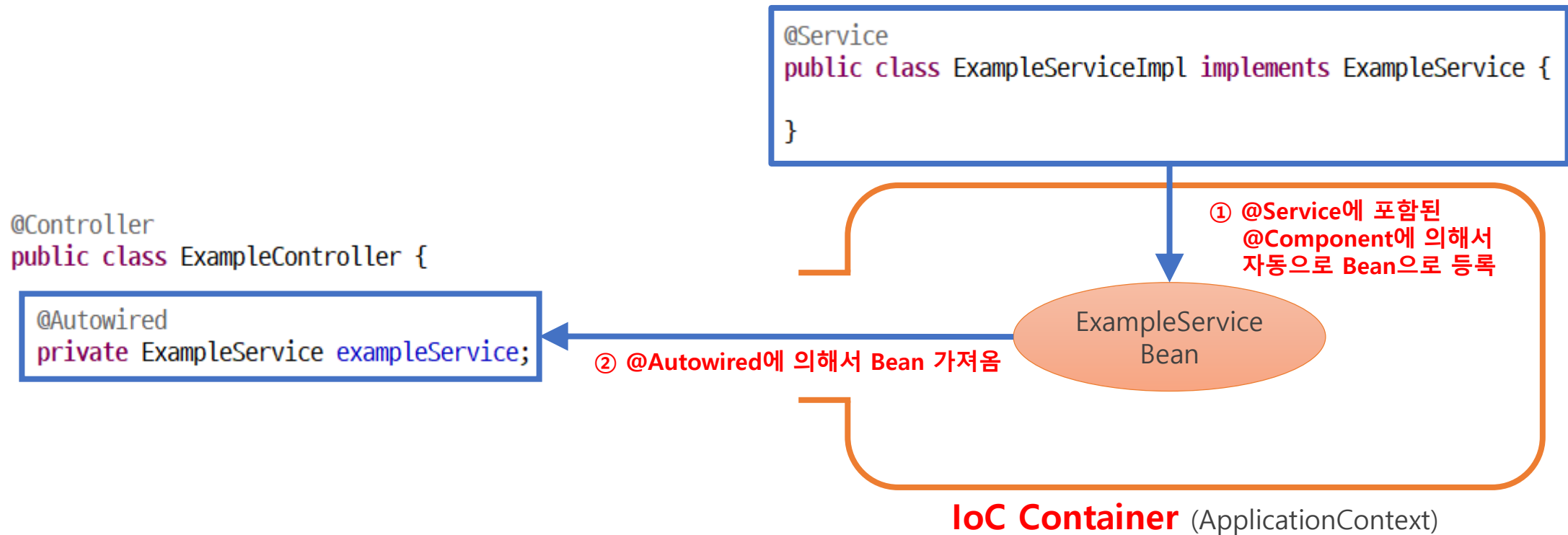
- DI
  - ✓ DI, Dependency Injection
  - ✓ 컨테이너에 등록된 Bean을 가져오는 방식
  - ✓ Field Injection, Setter Injection, Constructor Injection 방식이 있음
  - ✓ @Inject, @Autowired 등의 애노테이션을 이용해서 처리할 수 있음
- DI 애노테이션

애노테이션	의미
@Autowired	<ul style="list-style-type: none"><li>• Bean의 타입(class 속성)이 일치하면 가져옴</li><li>• 동일한 타입이 여러 개 있는 경우 Bean의 이름이 일치하면 가져옴</li><li>• 생성자에서는 @Autowired 생략 가능</li><li>• org.springframework.beans.factory.annotation.Autowired</li></ul>
@Qualifier	<ul style="list-style-type: none"><li>• Bean의 이름(id 속성)이 일치하면 가져옴</li><li>• 동일한 타입의 Bean이 여러 개 있는 경우 @Qualifier를 추가하여 Bean을 구분할 수 있음</li><li>• org.springframework.beans.factory.annotation.Qualifier</li></ul>
@Inject	<ul style="list-style-type: none"><li>• Bean의 타입(class 속성)이 일치하면 가져옴</li><li>• 동일한 타입이 여러 개 있는 경우 Bean의 이름이 일치하면 가져옴</li><li>• javax.inject.Inject</li></ul>

# Field Injection

- Field Injection

- ✓ 필드 주입
- ✓ @Autowired가 명시된 필드로 Bean을 가져옴
- ✓ 모든 필드마다 @Autowired를 명시해야 함
- ✓ IoC Container에 생성된 Bean이 없어도 코드 작성 시 NULL 여부를 체크하지 않아서 위험할 수 있음
- ✓ 코드 작성 시 순환 참조 여부를 미리 확인할 수 없음



# Setter Injection

- Setter Injection

- ✓ 세터 주입
- ✓ @Autowired가 명시된 세터의 매개변수로 Bean을 가져옴
- ✓ 세터에 @Autowired를 한 번만 명시하면 모든 매개변수로 Bean을 가져오기 때문에 편리함
- ✓ Field Injection처럼 IoC Container에 생성된 Bean이 없어도 코드 작성 시 NULL 여부를 체크하지 않아서 위험할 수 있음
- ✓ 코드 작성 시 순환 참조 여부를 미리 확인할 수 없음

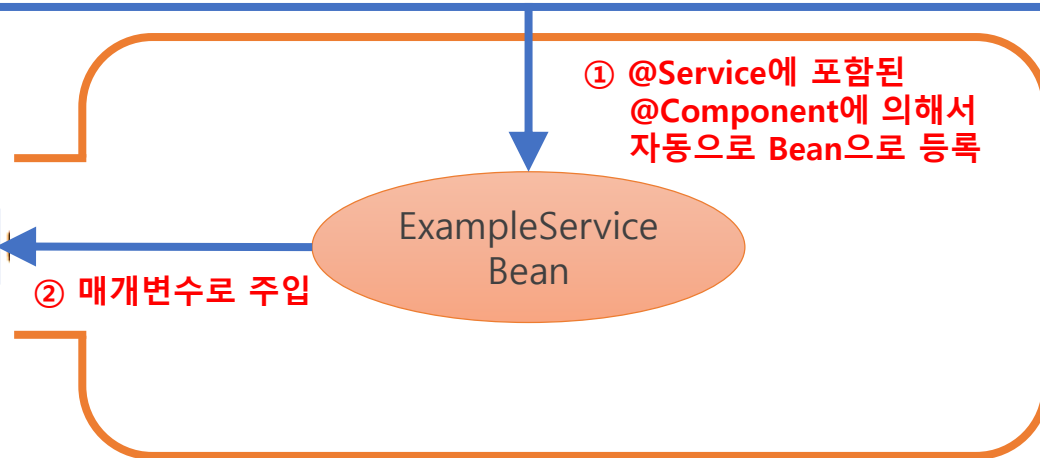
```
@Controller  
public class ExampleController {
```

```
    private ExampleService exampleService;
```

```
    @Autowired
```

```
    public void setExampleService(ExampleService exampleService) {  
        this.exampleService = exampleService;  
    }
```

```
@Service  
public class ExampleServiceImpl implements ExampleService {  
  
}
```



**IoC Container** (ApplicationContext)

# Constructor Injection

- Constructor Injection
  - ✓ 생성자 주입
  - ✓ 생성자(Generate constructor using field)를 만들면 생성자의 매개변수로 Bean을 가져옴
  - ✓ @Autowired를 명시할 필요가 없음

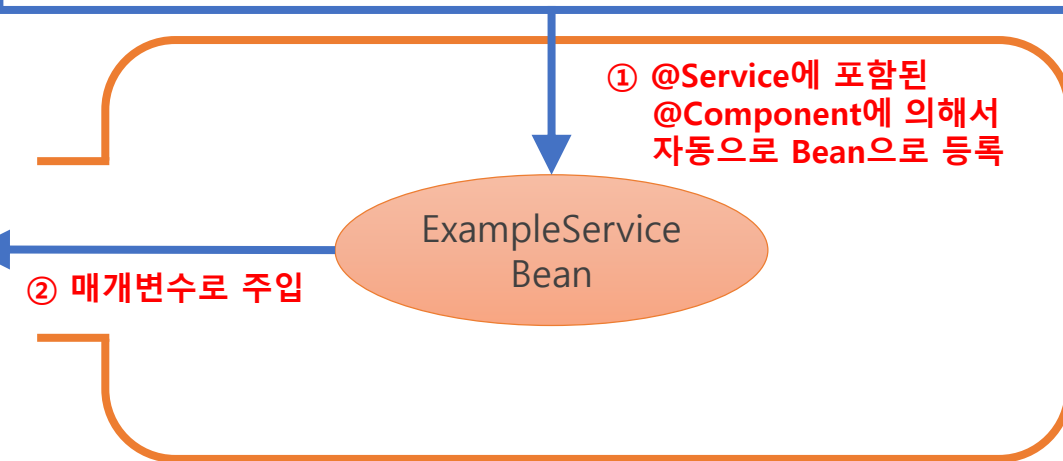
```
@Controller
public class ExampleController {

    private ExampleService exampleService;

    public ExampleController(ExampleService exampleService) {
        this.exampleService = exampleService;
    }
}
```

```
@Service
public class ExampleServiceImpl implements ExampleService {

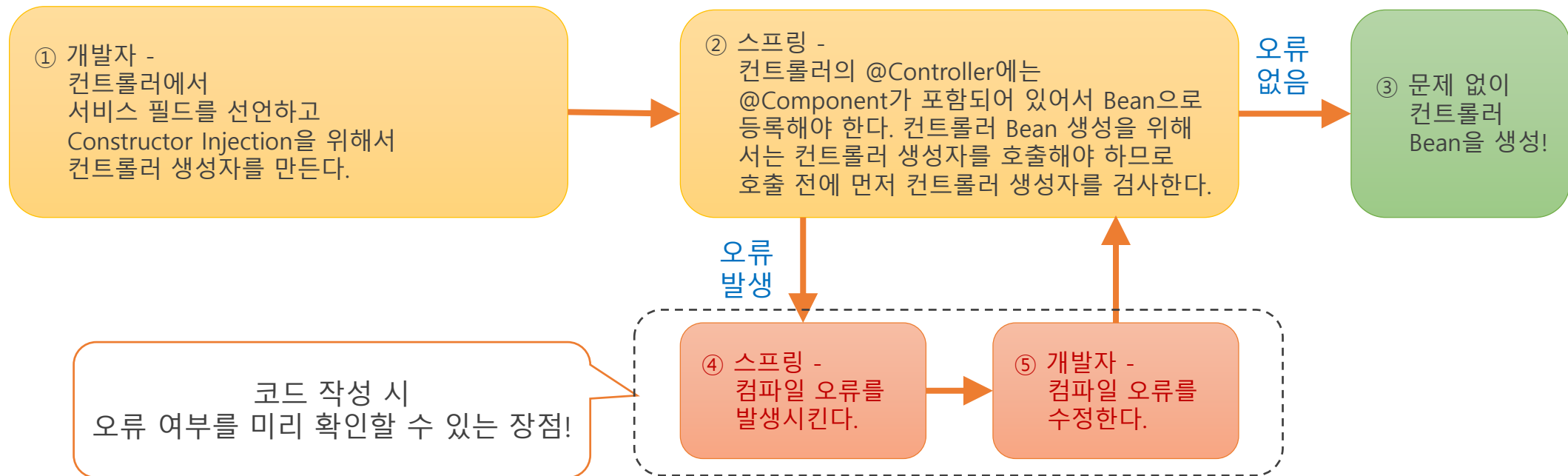
}
```



**IoC Container** (ApplicationContext)

# Constructor Injection 사용 권장

- Constructor Injection 사용 권장
  - ✓ Constructor Injection에 잘못된 부분이 있는 경우 컴파일 오류가 발생함
  - ✓ 3가지 DI 방식 중 가장 안전한 방식
- Controller의 Constructor Injection 예시

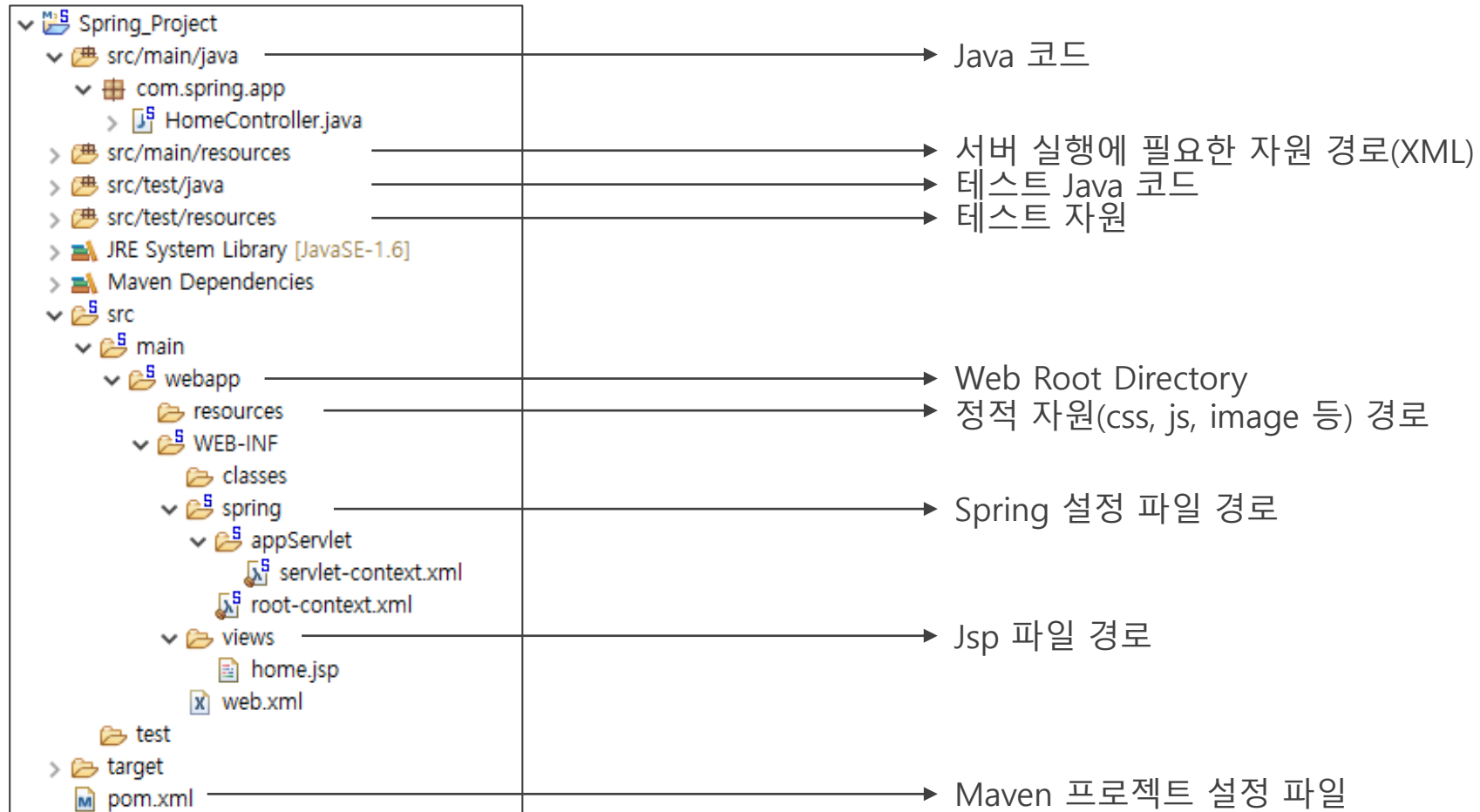




Spring MVC

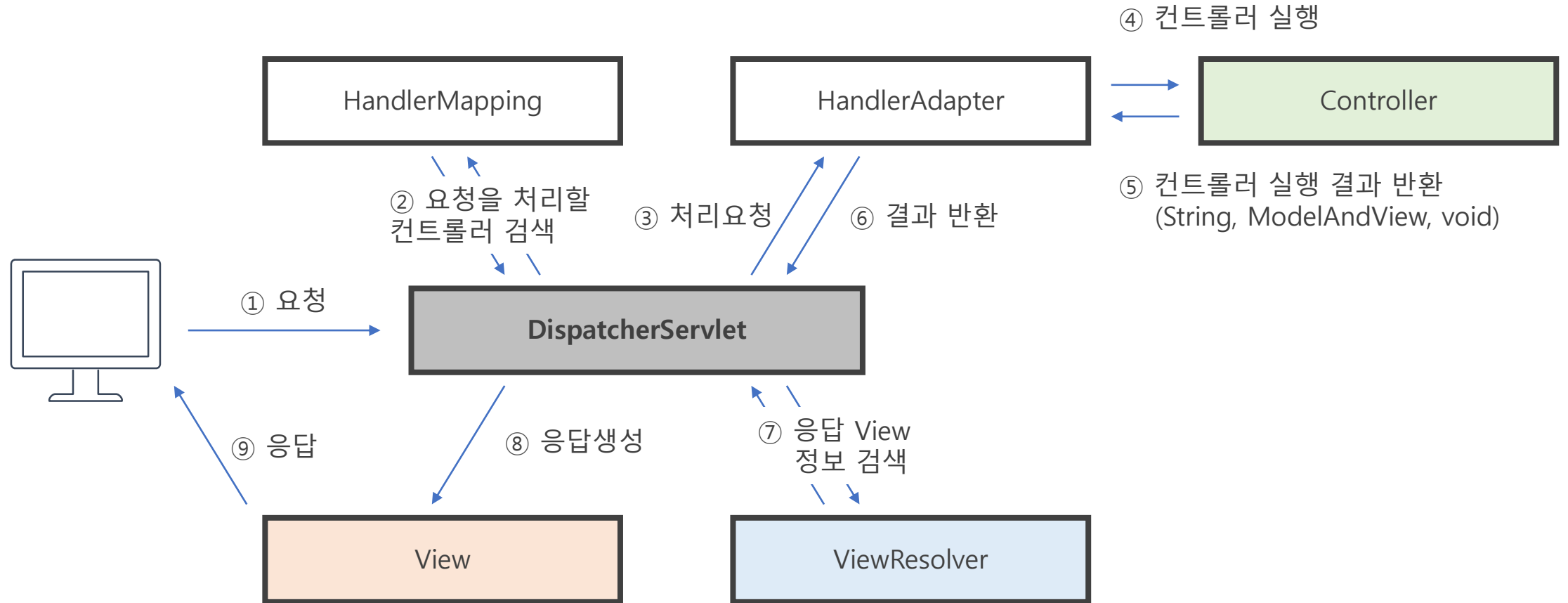
# Spring MVC

- Spring MVC Template





# Spring MVC 동작 원리



# DispatcherServlet

- DispatcherServlet
  - ✓ 스프링 프레임워크의 핵심 서블릿
  - ✓ web.xml에 관련 정보 작성
  - ✓ 기본적으로 servlet-context.xml 설정 파일을 읽어서 스프링 프레임워크를 구동함

```
<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>
```

DispatcherServlet

DispatcherServlet은  
servlet-context.xml의  
내용을 이용해서 동작함

DispatcherServlet이 동작하는 경로는 "/"이므로 동일한  
ContextPath를 가진 모든 경로에서 동작함

# servlet-context.xml

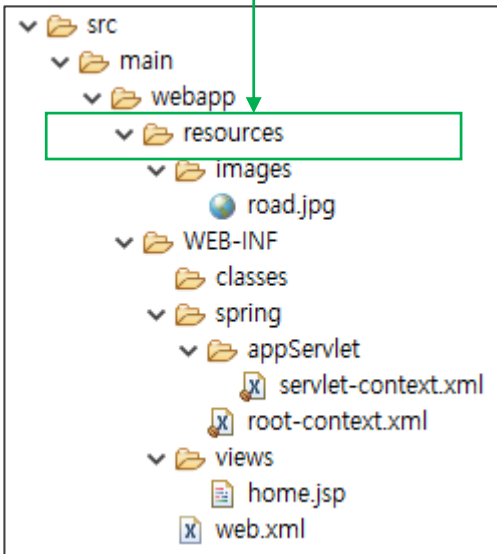
- <annotation-driven />
  - ✓ @Controller Annotation을 활성화
  - ✓ Spring MVC에서 Controller에게 요청하기 위해 필요한 HandlerMapping과 HandlerAdapter를 자동으로 bean으로 등록
- HandlerMapping
  - ✓ @Controller가 적용된 객체를 컨트롤러하고 함
  - ✓ HandlerMapping은 요청을 처리할 컨트롤러를 @RequestMapping값을 이용해서 검색함
- HandlerAdapter
  - ✓ 요청을 처리할 컨트롤러의 메소드를 실행함
  - ✓ 메소드 실행 결과를 ModelAndView 객체로 변환한 뒤 DispatcherServlet에게 반환함

```
<!-- Enables the Spring MVC @Controller programming model -->  
<annotation-driven />
```

# servlet-context.xml

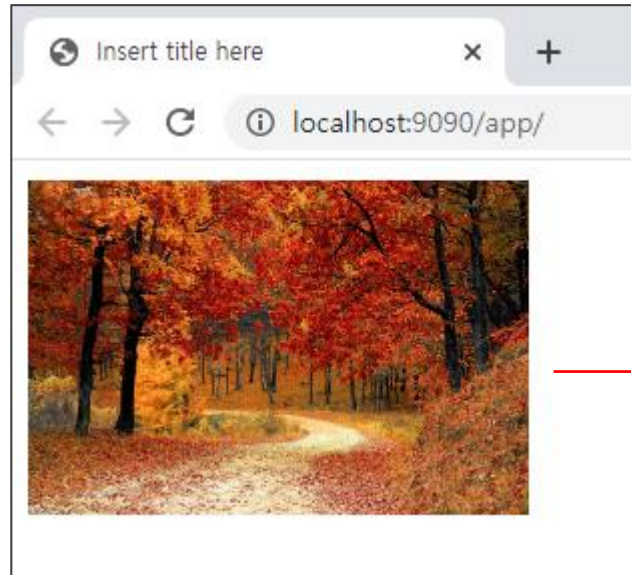
- `<resources mapping="/resources/**" location="/resources/">`
  - ✓ 웹 구성 요소 중에서 정적 자원들의 경로와 호출 방법을 기술
  - ✓ 정적 자원 : 멀티미디어 데이터(이미지, 오디오, 비디오 등), CSS, JS 등

```
<!-- Handles HTTP GET requests for /resources/** by efficiently serving up static resources in the ${webappRoot}/resources directory -->  
<resources mapping="/resources/**" location="/resources/" />
```



```

```



이미지 경로

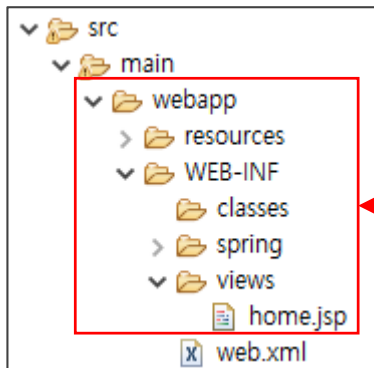
`http://localhost:9090/app/resources/images/road.jpg`

**app**는 `<%=request.getContextPath()%>`  
이므로 **ContextPath**를 의미함

# servlet-context.xml

- <beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  - ✓ 뷰 리졸버
  - ✓ HandlerAdapter에 의해서 반환된 ModelAndView 객체에 저장된 뷰 정보를 처리함

```
<!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->
<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>
```



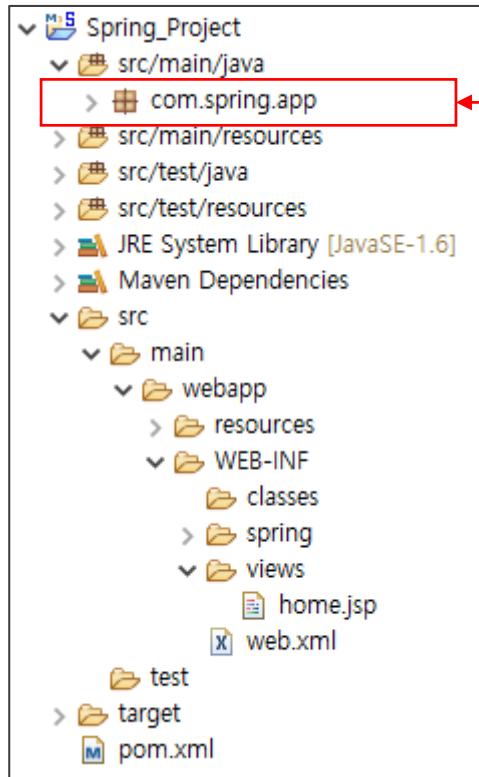
```
@RequestMapping(value="/", method=RequestMethod.GET)
public String home() {
    return "home";
}
```

실제 응답할 뷰는 뷰 리졸버에 의해서 다음과 같이 처리됨  
"home" 앞에 "/WEB-INF/views/" 추가  
"home" 뒤에 ".jsp" 추가

return "/WEB-INF/views/home.jsp"

# servlet-context.xml

- `<context:component-scan base-package="com.spring.app" />`
  - ✓ base-package에 지정된 패키지에 저장된 클래스들을 스캔하고 자동으로 bean을 생성함
  - ✓ @Component, @Controller, @Service, @Repository 등의 Annotation이 추가된 클래스를 bean으로 등록함
  - ✓ Spring MVC Project의 top-level package와 동일해야 함



```
<context:component-scan base-package="com.spring.app" />
```

동일해야 함

# Spring MVC 주요 Annotation

- 주요 Annotation

Annotation	의미	사용
@Controller	스프링 MVC의 컨트롤러 객체임을 명시	클래스
@RequestMapping	특정 RequestURI에 매핑되는 컨트롤러(클래스)나 메소드임을 명시	클래스, 메소드
@RequestParam	요청에서 특정 파라미터의 값을 가져올 때 사용	파라미터
@RequestHeader	요청에서 특정 HTTP헤더 정보를 가져올 때 사용	파라미터
@ModelAttribute	파라미터를 처리한 객체를 뷰까지 전달	메소드, 파라미터
@Component	Bean으로 만들어 줘야 할 객체임을 명시	클래스
@Service	서비스 객체에 추가하는 @Component	클래스
@Repository	DAO 객체에 추가하는 @Component	클래스
@PathVariable	RequestURI에 포함된 값을 가져올 때 사용	파라미터
@RequestBody	요청 본문에 포함된 데이터가 파라미터로 전달	파라미터
@ResponseBody	반환 값이 HTTP 응답 메시지로 전송	메소드, 리턴타입
@CookieValue	쿠키가 존재하는 경우 쿠키 이름을 이용해서 쿠키 값을 가져올 때 사용	파라미터
@SessionAttribute	Model의 정보를 세션에서 유지할 때 사용	클래스

# Controller

- Controller

- ✓ @Controller가 적용된 클래스
- ✓ @RequestMapping을 이용해 요청 URL 및 요청 메소드 파악
- ✓ 메소드 단위로 요청을 처리
- ✓ 메소드는 결과를 출력할 뷰 이름을 반환

클래스를  
컨트롤러로  
인식

```
@Controller
```

```
public class HomeController {
```

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

```
    @RequestMapping(value = "/", method = RequestMethod.GET)
```

RequestURI가 ContextPath("/")인 GET 방식의 요청을 처리

```
    public String home(Locale locale, Model model) {
```

```
        logger.info("Welcome home! The client locale is {}.", locale);
```

```
        Date date = new Date();
```

```
        DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG, DateFormat.LONG, locale);
```

```
        String formattedDate = dateFormat.format(date);
```

```
        model.addAttribute("serverTime", formattedDate );
```

model을 이용해 뷰로 전달(forward)할 속성(Attribute)을 저장

```
        return "home";
```

응답 결과가 나타날 뷰이름은 "home.jsp"

```
    }
```

```
}
```



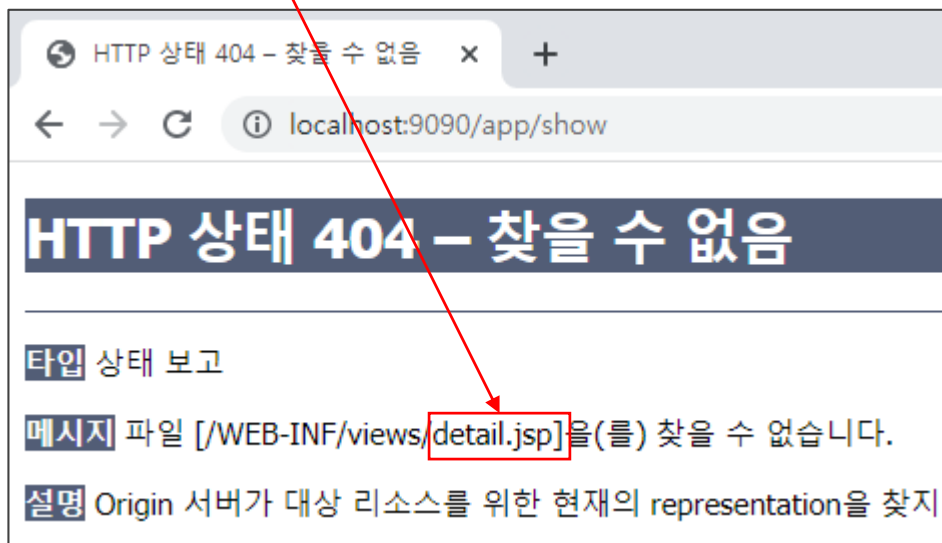
# Controller Method 반환 타입

- Controller Method

- ✓ 컨트롤러는 하나의 요청을 하나의 메소드로 처리함
- ✓ Spring MVC Pattern에서는 메소드의 반환 타입을 String 또는 void로 설정할 수 있음

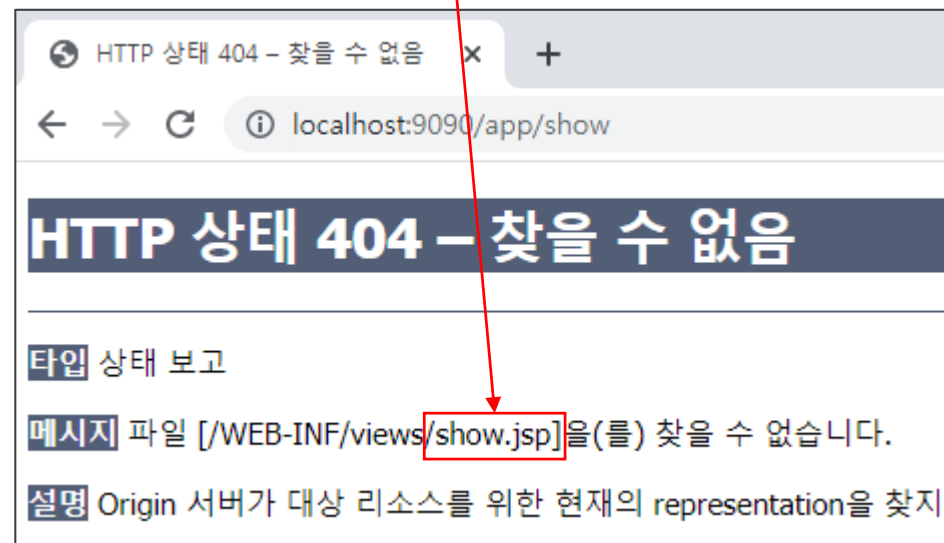
- 반환 타입이 String인 경우  
반환값을 뷰(Jsp)의 이름으로 인식

```
@RequestMapping(value="show")  
public String method1() {  
    return "detail";  
}
```



- 반환 타입이 void인 경우  
매핑값을 뷰(Jsp)의 이름으로 인식

```
@RequestMapping(value="show")  
public void method2() {  
}
```



# @RequestMapping

- @RequestMapping
  - ✓ 요청 URL과 요청 메소드를 인식할 수 있는 Annotation
  - ✓ 요청 메소드에 따라서 @GetMapping, @PostMapping 등으로 변경할 수 있음
- @RequestMapping 주요 기능

구분	예시	의미
value	@RequestMapping(value="/")	"/" 요청
	@RequestMapping(value={"/", "index"})	"/"와 "index" 요청
	@RequestMapping(value="/member/*.do")	"/member" 로 시작하고 ".do"로 끝나는 요청
method	@RequestMapping(method=RequestMethod.GET)	GET 방식(조회)
	@RequestMapping(method=RequestMethod.POST)	POST 방식(삽입)
	@RequestMapping(method=RequestMethod.PUT)	PUT 방식(수정)
	@RequestMapping(method=RequestMethod.DELETE)	DELETE 방식(삭제)
content type	@RequestMapping(consumes="application/json")	요청 콘텐츠가 JSON임
	@RequestMapping(produces="application/json")	응답 콘텐츠가 JSON임

# 기본 URL Pattern

- web.xml의 기본 URL Pattern
  - ✓ 기본 Pattern은 컨텍스트 패스(Context Path)로 되어 있음
  - ✓ 다른 Pattern으로 수정하면 전체 URL이 다르게 설정됨

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

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

- @RequestMapping(value="members") 처리 방식

컨텍스트 패스	<url-pattern>	전체 URL
app	/	http://localhost:8080/app/members
	/home/*	http://localhost:8080/app/home/members

# Encoding Filter

- Encoding Filter
  - ✓ web.xml에 CharacterEncodingFilter를 추가
  - ✓ request.setCharacterEncoding("UTF-8")을 대체할 수 있는 필터

```
<filter>
  <filter-name>encodingFilter</filter-name>
  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
  <init-param>
    <param-name>forceEncoding</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>encodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

문자셋 인코딩

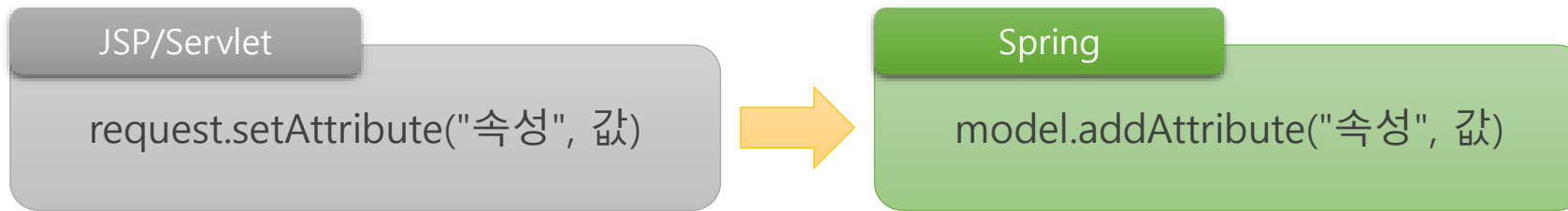
UTF-8

동일한 ContextPath를 가지는 모든 경로에 적용

# Model

- Model

- ✓ 뷰가 응답 화면을 구성할 때 필요로하는 데이터를 전달하는 인터페이스
- ✓ JSP/Servlet에서는 request를 이용해서 데이터를 전달하였으나, 스프링에서는 model을 이용하여 데이터를 전달



```
@RequestMapping(value = "/", method = RequestMethod.GET)
public String home(Locale locale, Model model) {
    logger.info("Welcome home! The client locale is {}.", locale);

    Date date = new Date();
    DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG, DateFormat.LONG, locale);

    String formattedDate = dateFormat.format(date);

    model.addAttribute("serverTime", formattedDate );

    return "home";
}
```

컨트롤러의 메소드 매개변수로 Model model 선언

model을 이용해 뷰로 전달(forward)할 속성(Attribute)을 저장

home.jsp에서 \${serverTime}으로 전달된 값을 확인

# 요청 파라미터

- 스프링의 요청 파라미터 처리 방식
  - ① HttpServletRequest의 getParameter() 메소드
  - ② @RequestParam 애너테이션
  - ③ 커맨드 객체



요청 파라미터



## Controller

JSP/Servlet

한 가지 방법만 지원

HttpServletRequest

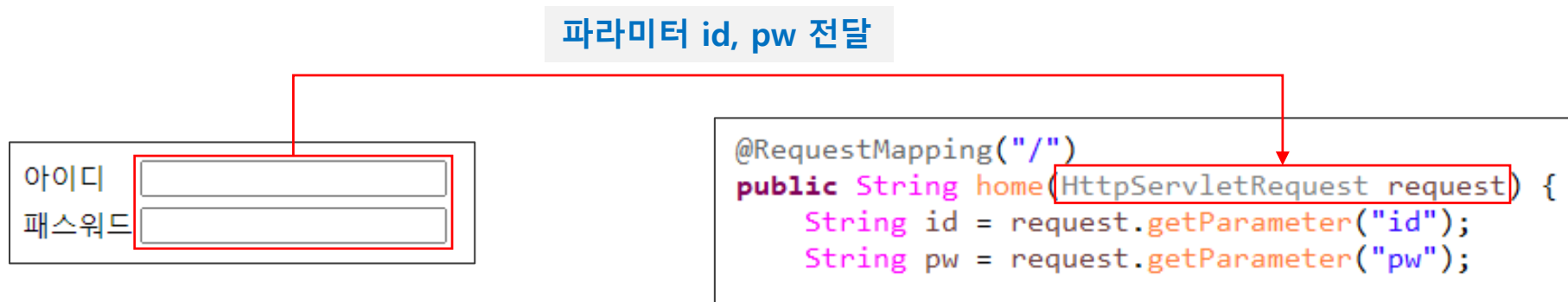
Spring

다양한 방법을 지원

- HttpServletRequest
- @RequestParam
- MemberVO member

# HttpServletRequest

- HttpServletRequest
  - ✓ 요청을 처리하는 HttpServletRequest 인터페이스
  - ✓ 파라미터로 전달된 값을 `getParameter()` 메소드를 이용해서 처리



# @RequestParam

- @RequestParam
  - ✓ 파라미터를 인식하고 변수에 저장하는 애너테이션
  - ✓ @RequestParam 애너테이션을 생략하고 변수명만 작성할 수 있음
  - ✓ 속성
    - value : 파라미터 이름 작성
    - required : 필수 여부 지정(디폴트 true)
    - defaultValue : 파라미터가 없는 경우 사용할 문자열 기본값

```
<a href="${contextPath}/read?pid=admin&week=sat">
```

파라미터 pid, week 전달

```
@RequestMapping(value="read", method=RequestMethod.GET)  
public String read(  
    @RequestParam(value="pid") String pid,  
    @RequestParam(value="week", required=false, defaultValue="mon") String week) {
```

파라미터 pid는 필수이므로 전달되지 않는다면 Exception 발생  
파라미터 week는 필수가 아니며 만약 전달되지 않는다면 week=mon으로 처리



# 커맨드 객체

- 커맨드 객체

- ✓ 여러 개의 요청 파라미터를 객체로 전달 받는 방식
- ✓ 여러 개의 요청 파라미터를 일일이 처리하는 것보다 편리하게 개선된 방식
- ✓ 커맨드 객체의 setter를 이용해서 파라미터를 전달 받음

아이디	<input type="text"/>
이름	<input type="text"/>
연락처	<input type="text"/>

```
@RequestMapping(value="register", method=RequestMethod.POST)
public String register(
    @RequestParam("id") String id,
    @RequestParam("name") String name,
    @RequestParam("tel") String tel) {
```

커맨드 객체 방식으로 개선

```
@RequestMapping(value="register", method=RequestMethod.POST)
public String register(MemberVO vo) {
```

# 커맨드 객체

- 커맨드 객체 특징

- ✓ 커맨드 객체를 자동으로 뷰까지 전달함
- ✓ 별도로 model.addAttribute()를 처리할 필요가 없음

아이디	<input type="text" value="admin"/>
이름	<input type="text" value="관리자"/>
연락처	<input type="text" value="02-500-1000"/>
<input type="button" value="전송"/>	



```
@RequestMapping(value="register", method=RequestMethod.POST)
public String register(MemberVO vo) {
    return "home";
}
```

전달될 때 클래스명(MemberVO)의 첫 글자를 소문자로 변환한 이름을 사용함  
(객체명(vo)을 사용하는 것이 아님!)

```
<div>아이디 ${memberVO.id}</div>
<div>이름 ${memberVO.name}</div>
<div>연락처 ${memberVO.tel}</div>
```

home.jsp



아이디 admin  
이름 관리자  
연락처 02-500-1000

# @ModelAttribute

- @ModelAttribute

- ✓ 커맨드 객체를 model로 전달할 때 이름을 변경하고자 하는 경우에 사용
- ✓ @ModelAttribute에서 지정한 이름으로 뷰까지 전달함

아이디	<input type="text" value="admin"/>
이름	<input type="text" value="관리자"/>
연락처	<input type="text" value="02-500-1000"/>
<input type="button" value="전송"/>	

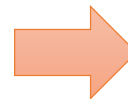


```
@RequestMapping(value="register", method=RequestMethod.POST)
public String register(@ModelAttribute("member") MemberVO vo) {
    return "home";
}
```

클래스명 MemberVO 대신 member를 사용!

```
<div>아이디 ${member.id}</div>
<div>이름 ${member.name}</div>
<div>연락처 ${member.tel}</div>
```

home.jsp



아이디 admin  
이름 관리자  
연락처 02-500-1000

# Redirect

- Redirect

- ✓ 컨트롤러의 반환값이 "redirect:"으로 시작하면 리다이렉트로 이동함
- ✓ response.sendRedirect()를 대체하는 스프링의 방식
- ✓ "redirect:" 뒤에는 새로운 요청 URL이 오기 때문에 특정 URLMapping값을 작성해야 함  
(뷰이름을 작성하는 것이 아님!)

```
@RequestMapping(value="register", method=RequestMethod.POST)
public String register(@ModelAttribute("member") MemberVO vo) {

    return "redirect:/view";
}

@RequestMapping(value="view", method=RequestMethod.GET)
public String view() {

    return "detail";
}
```

URLMapping "view"로 리다이렉트

리다이렉트는 기존 request를 유지하지 않음

detail.jsp

```
<div>아이디 ${member.id}</div>
<div>이름 ${member.name}</div>
<div>연락처 ${member.tel}</div>
```



아이디  
이름  
연락처



AOP

# AOP

- AOP

- ✓ AOP, Aspect Oriented Programming
- ✓ 관점 지향 프로그래밍
- ✓ 문제를 바라보는 관점을 기준으로 프로그래밍하는 패러다임
- ✓ 각 문제를 해결하는 핵심로직과 모든 문제에 적용되는 공통로직을 기준으로 프로그래밍을 분리하는 것

- AOP 도입 이유

- ✓ 애플리케이션 개발에 필요한 다양한 공통 모듈(로그, 보안, 트랜잭션 처리 등)이 사용됨
- ✓ 대부분의 비즈니스로직에서 필요하기 때문에 반복적으로 유사한 코드를 매번 작성해야 함
- ✓ 공통 모듈을 비즈니스로직마다 작성해야 하는 문제를 개선하기 위해서 AOP가 도입

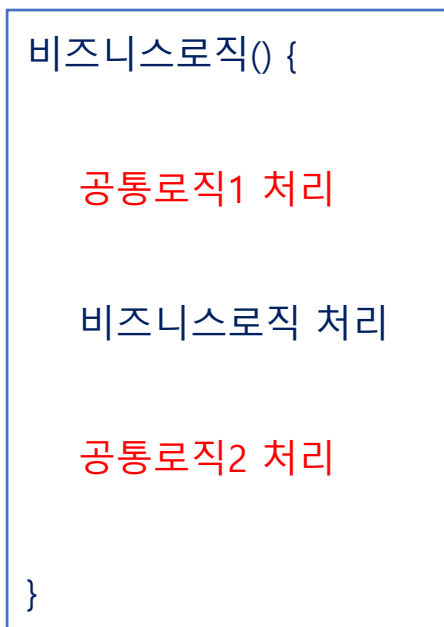


# AOP 처리 방식

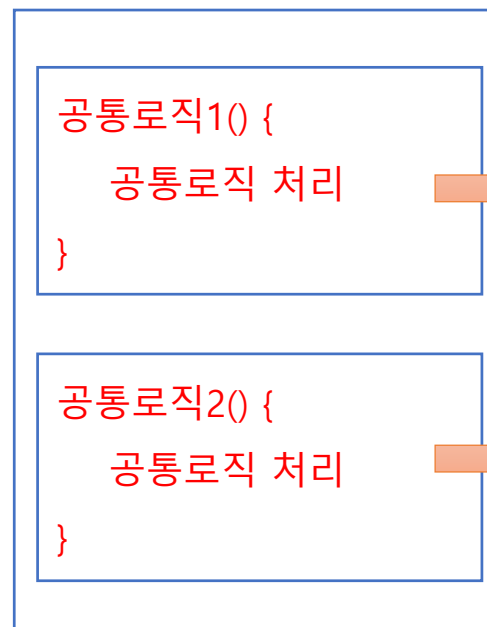
- AOP 처리 방식

- 비즈니스로직 작성 시 공통 모듈을 직접 호출하지 않음
- 공통 모듈을 만들어 두고 AOP 기능을 적용하면 자동으로 비즈니스 로직에 공통로직이 삽입되는 방식으로 동작함

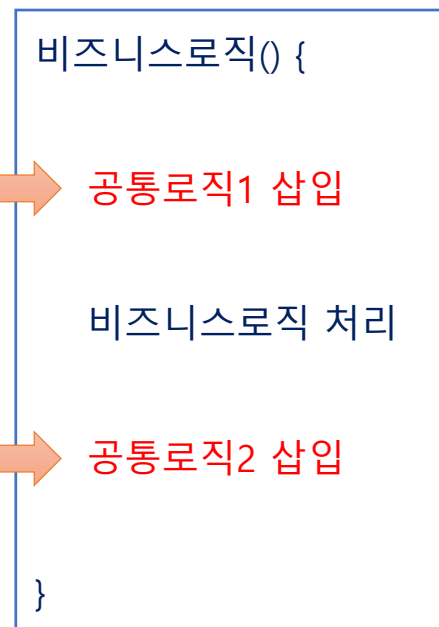
공통 모듈 + 비즈니스로직



공통 모듈



비즈니스로직



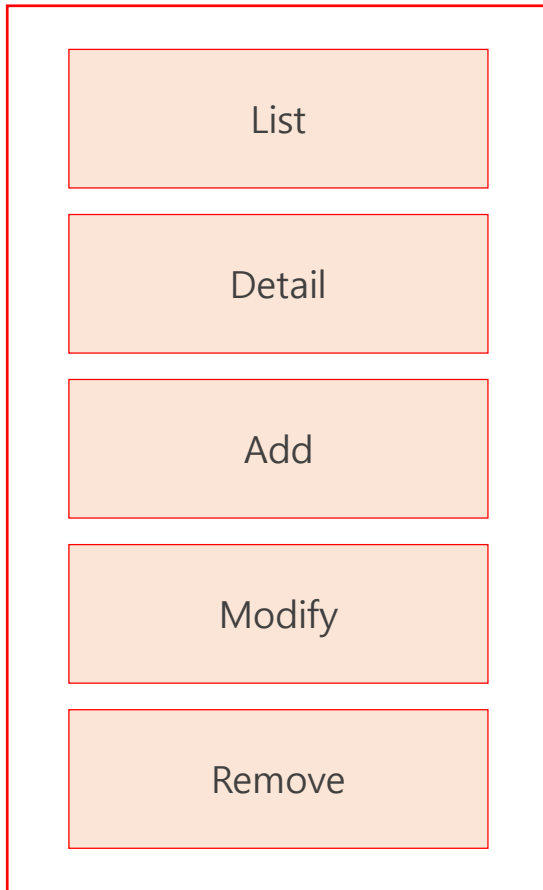
# AOP 용어

- JoinPoint(조인포인트)
  - ✓ 클라이언트가 호출하는 모든 비즈니스 메소드를 의미함
  - ✓ JoinPoint 중 일부가 PointCut이 되기 때문에 PointCut 대상 또는 PointCut 후보라고도 함
- PointCut(포인트컷)
  - ✓ JoinPoint의 부분집합
  - ✓ 실제 Advice가 적용되는 JoinPoint를 의미함
  - ✓ 정규 표현식(Regular Expression) 또는 AspectJ 문법으로 PointCut을 정의함
- Advice(어드바이스)
  - ✓ 언제 공통 모듈을 비즈니스 메소드에 적용할 것인지 정의하는 것을 의미함
  - ✓ 비즈니스 메소드 실행 전, 후, 주변, 예러 발생시 넣을 수 있음



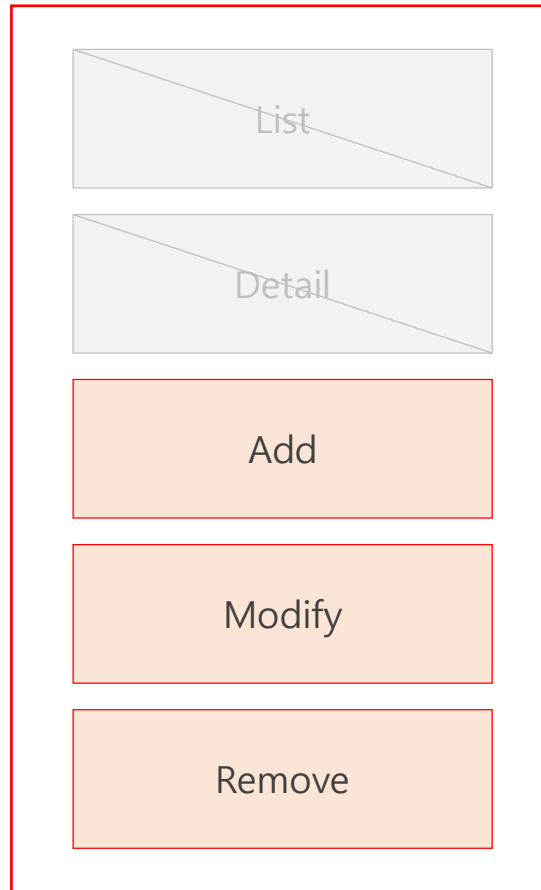
# AOP 용어

전체 5개의 비즈니스 메소드가 있다.



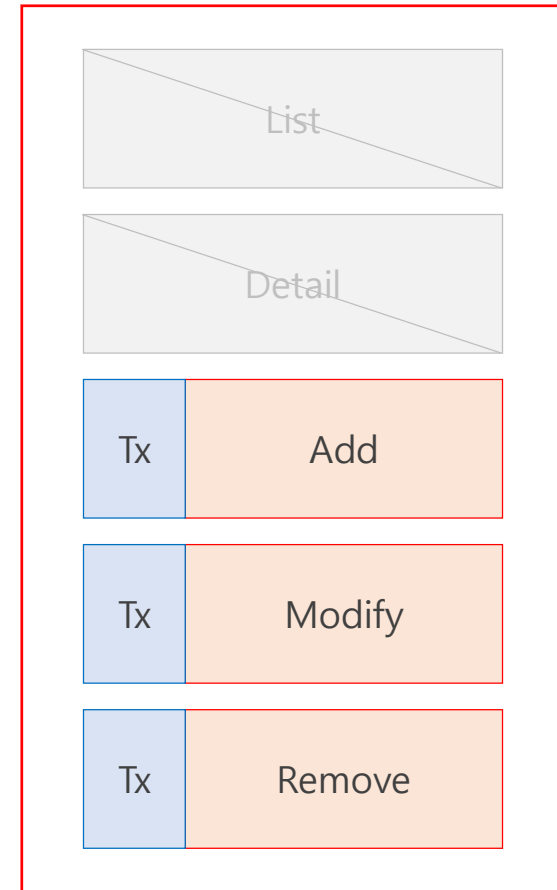
JoinPoint

이 중 3개의 비즈니스 메소드에  
공통 모듈을 적용하고자 한다.



PointCut

적용할 공통 모듈은  
트랜잭션처리이다.



Advice

# JoinPoint

- JoinPoint 인터페이스

- ✓ JoinPoint는 Spring AOP 또는 AspectJ에서 AOP가 적용되는 지점을 의미함
- ✓ AspectJ에서는 이 지점을 JoinPoint라는 인터페이스로 나타낼 수 있음
- ✓ 모든 어드바이스(Advice)는 JoinPoint 인터페이스 타입의 파라미터를 첫 번째 매개변수로 선언하고 사용할 수 있음 (org.aspectj.lang.JoinPoint)
- ✓ Around 어드바이스는 반드시 ProceedingJoinPoint 타입의 파라미터를 필수로 선언해야 함

```
@Around("execution(* com.spring.app.controller.*.*(..))")
public Object logging(ProceedingJoinPoint joinPoint) throws Throwable {

    return null;
}
```

- JoinPoint 인터페이스 주요 메소드

반환타입	이름	의미
Object[]	getArgs()	메소드로 전달되는 argument 반환
Signature	getSignature()	어드바이스 되는 메소드의 시그니처 반환
Object	getTarget()	대상 객체 반환
String	toString()	어드바이스 되는 메소드의 문자열 설명 반환

# PointCut 표현식

- PointCut 표현식

- ✓ AspectJ 포인트컷 표현식은 execution() 포인트컷 지시자를 이용하여 작성

- ✓ 문법 구조

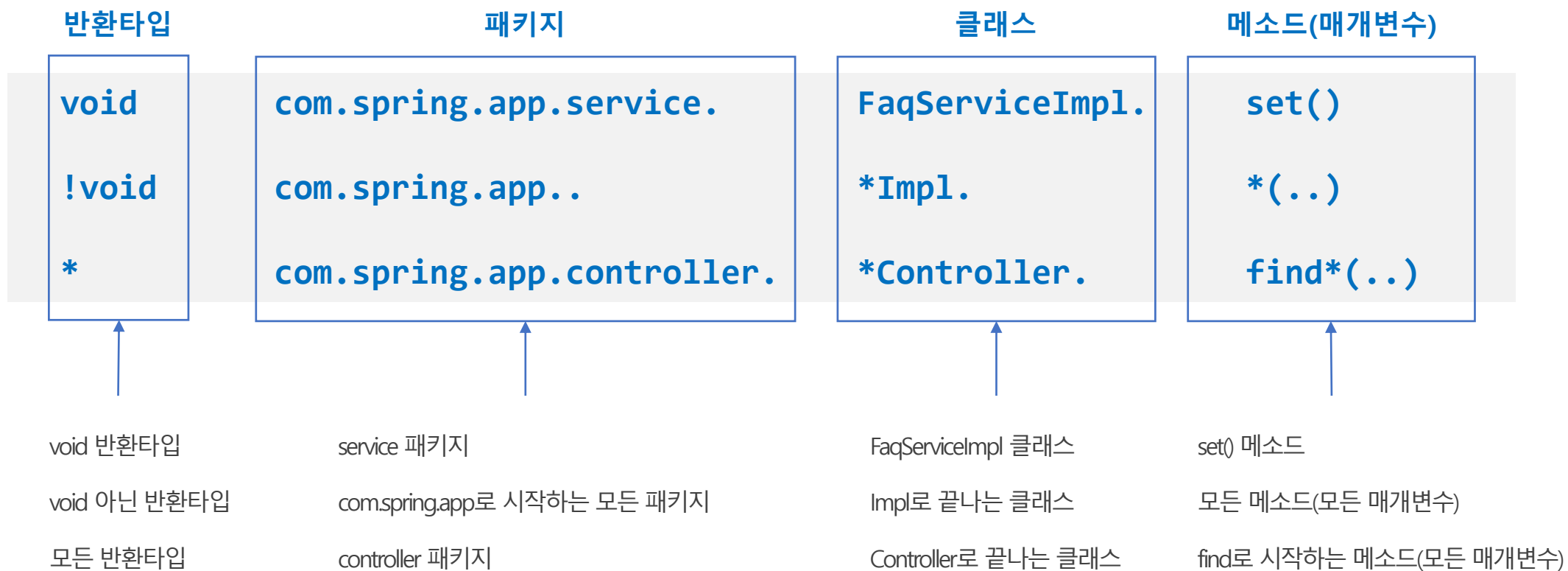
execution( [접근제한자] [반환타입] 패키지.클래스.메소드(매개변수) )

- execution() 지시자

구분	종류	의미
접근제한자	private, public	메소드의 접근제한자를 의미함. 생략가능
반환타입	*, void, !void	메소드의 반환타입을 의미함
메소드	메소드명, *	PointCut되는 메소드를 의미함
매개변수	*, ..	..는 개수 상관없이 모든 매개변수, *는 1개의 모든 매개변수

# PointCut 표현식

- PointCut 표현식 작성 예시



# Advice

- Advice
  - ✓ 어드바이스
  - ✓ 언제 공통 관심 기능(공통 모듈)을 핵심 로직(비즈니스 로직)에 적용할 것인지를 정의함
  - ✓ 언제 적용할 것인지에 따라 5가지 종류로 나눌 수 있음
- Advice 종류

종류	동작시점	반환타입	매개변수타입
@Before	메소드 실행 이전	void	JointPoint
@After	메소드 실행 이후	void	JointPoint
@Around	메소드 실행 이전/이후	Object	ProceedingJoinPoint
@AfterReturning	예외 없는 메소드 실행 이후	void	JointPoint
@AfterThrowing	예외 발생한 메소드 실행 이후	void	JointPoint

# Weaving

- Weaving
  - ✓ 위빙
  - ✓ Advice를 비즈니스 메소드에 적용하는 것을 의미함
- Weaving 방식
  - ① 컴파일 타임에 Weaving 하기
    - : 컴파일할 때 공통 코드가 삽입되고 AOP가 적용된 클래스 파일(.class)을 생성함
  - ② 클래스 로딩타임에 Weaving 하기
    - : 원본 클래스를 수정하지 않고, 클래스를 메모리에 로딩할 때 JVM이 AOP를 적용한 바이트 코드를 사용함
  - ③ 런타임에 Weaving 하기
    - : 스프링이 지원하는 방식으로 Proxy를 사용함
- pom.xml

```
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>${org.aspectj-version}</version>
  <scope>runtime</scope>
</dependency>
```

`<org.aspectj-version>1.9.6</org.aspectj-version>`



LOG

# Log

- Log
  - ✓ 로그
  - ✓ 애플리케이션이 동작하는 동안의 각종 이벤트를 기록으로 남기는 것
  - ✓ 개발 기간에는 로그 패키지를 이용하여 디버깅에 활용
  - ✓ 로그 출력을 특정 파일이나 DB에 저장해 둘 수 있음
- Java Logging Framework
  - ✓ 자바 애플리케이션을 개발할 때 사용할 수 있는 로그 처리 프레임워크
  - ✓ `java.util.logging` 패키지, Log4j, Logback등이 존재



# Log4j

- Log4j

- ✓ 자바 기반의 Logging 유틸리티
- ✓ 가장 오래된 Java Logging Framework
- ✓ 프로그램 개발 도중 디버깅을 위해 정해진 양식에 맞춰 화면이나 파일로 기록을 남길 수 있음

기술지원 중

Log4j 2



## 1. Apache Log4j Core

8,738 usages

[org.apache.logging.log4j](https://org.apache.logging.log4j) » [log4j-core](#)

Apache

Implementation for Apache Log4J, a highly configurable logging tool that focuses on performance and low garbage generation. It has a plugin architecture that makes it extensible and supports asynchronous logging based on LMAX Disruptor.

Last Release on Sep 17, 2022

기술지원 종료

Log4j 1



## 2. Apache Log4j

17,964 usages

[log4j](#) » [log4j](#)

Apache

Legacy version of Log4J logging framework. Log4J 1 has reached its end of life and is no longer officially supported. It is recommended to migrate to Log4J 2.

Last Release on May 26, 2012

# Log4j 로그 레벨

- 로그 레벨

- ✓ 무분별한 로그로 인해 중요한 로그를 찾는 것이 어려워지는 것을 막기 위해 어떤 내용까지를 로그로 남길 것인지 표준으로 정해 놓은 레벨
- ✓ FATAL > ERROR > WARN > INFO > DEBUG > TRACE 순으로 레벨이 정해짐  
(INFO 레벨 적용 시 FATAL, ERROR, WARN, INFO 레벨의 로그가 남겨짐)

- 로그 레벨의 종류

구분	의미
FATAL	응용프로그램이 중단될 가능성이 있는 매우 심각한 오류 이벤트를 지정
ERROR	응용프로그램 실행을 계속 허용할 수 있는 오류 이벤트를 지정
WARN	잠재적으로 유해한 상황을 지정
INFO	대략적인 수준에서 응용프로그램의 진행률을 강조하는 정보 메시지를 지정
DEBUG	응용프로그램을 디버깅하는데 가장 유용한 세부 정보 이벤트를 지정 (개발 단계에서 주로 사용하는 레벨, 운영 단계에서는 로그가 너무 많아 감당이 안 될 수 있음)
TRACE	DEBUG보다 세분화된 정보 이벤트를 지정

# Commons Logging과 SLF4J

- Commons Logging

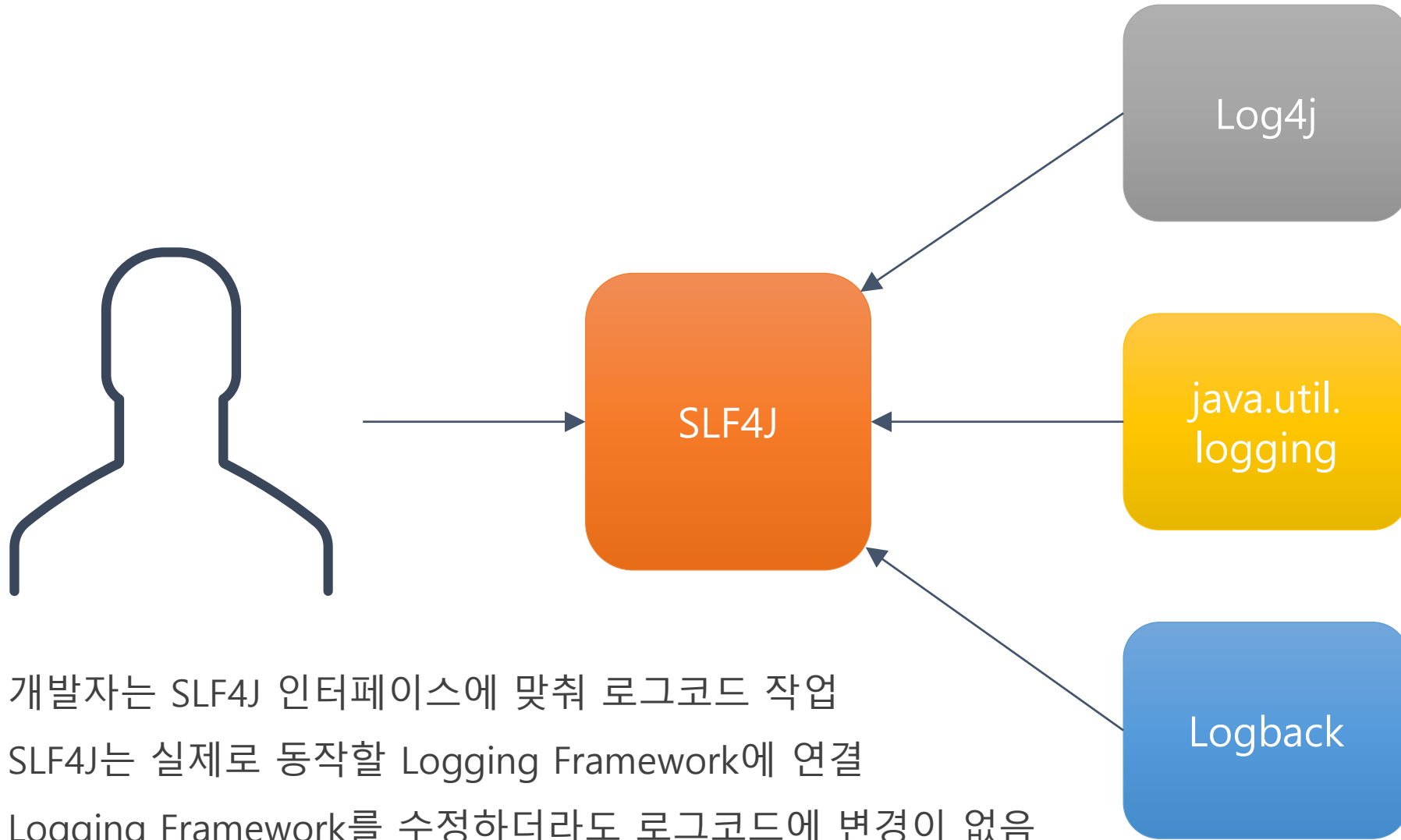
- ✓ 정식 명칭은 Jakarta Commons Logging(JCL)
- ✓ 아파치 재단에서 제공하는 공통 로깅 API로 facade 역할을 수행함  
(\* facade : 퍼사드, 어떤 소프트웨어의 간략한 인터페이스를 제공하는 디자인 패턴)
- ✓ 런타임 시점에 다른 Logging Framework를 찾기 때문에 비효율적임

가장 많이 사용되는  
Logging Interface

- SLF4J

- ✓ Simple Logging Facade for Java
- ✓ Commons Logging과 마찬가지로 공통 로깅 API 역할을 수행
- ✓ Log4j, Logback 등과 같이 다른 Logging Framework에 대한 인터페이스 역할을 수행
- ✓ 컴파일 시점에 다른 Logging Framework를 바인딩하기 때문에 효율적임(기존의 Commons Logging의 단점을 보완)

# SLF4J



# Spring 기본 Logging

- Spring Legacy Project에 포함된 Logging Library

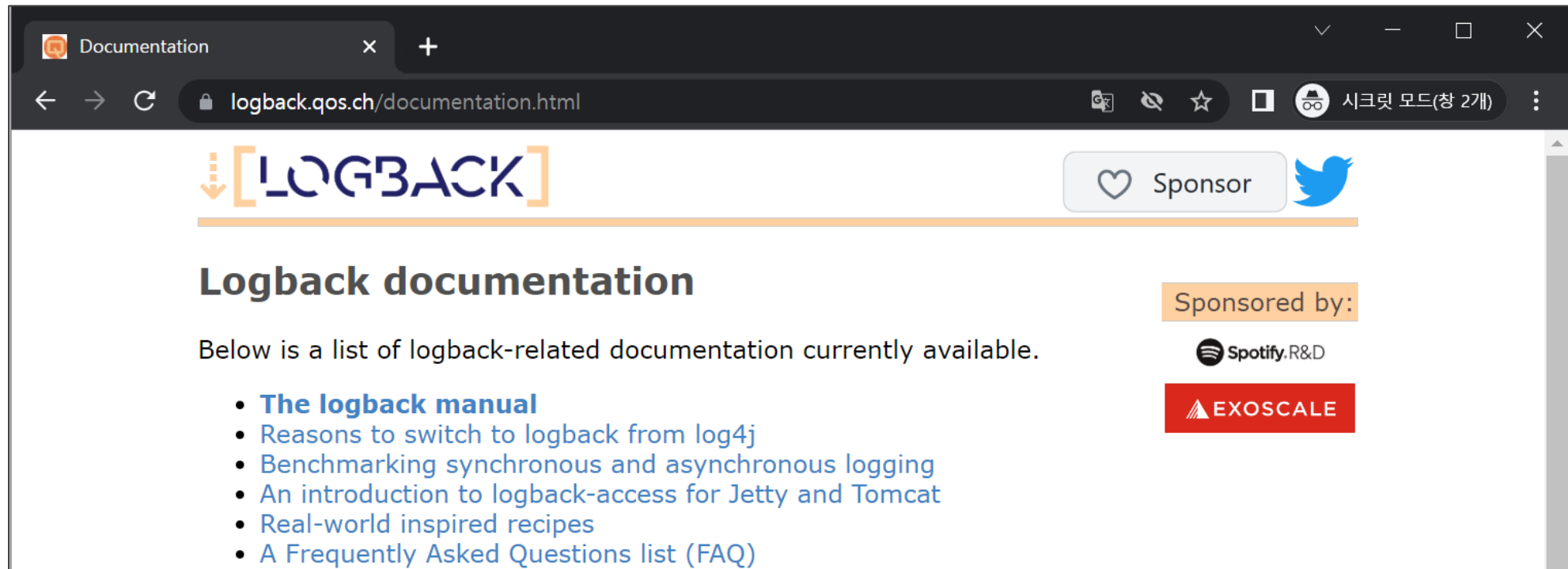
라이브러리	기능
log4j	Log4j 라이브러리
slf4j-api	SLF4J 라이브러리
jcl-over-slf4j	Commons Logging 대신 SLF4J 패키지에 있는 JCL 대체 클래스 및 메소드를 사용할 수 있는 브릿지
slf4j-log4j12	Log4j 라이브러리와 SLF4J 라이브러리를 연동시켜주는 브릿지



SLF4J 인터페이스를 이용해 Log4j를 사용하는 예시

# Logback

- Logback
  - ✓ Log4j의 후속 버전 Java Logging Framework
  - ✓ Log4j보다 10배의 성능 개선이 이루어지고 메모리 점유도 줄어듦
  - ✓ 현재 가장 널리 사용되고 있음
  - ✓ SLF4J의 구현체
  - ✓ 6단계의 로그 레벨을 가짐  
(OFF > ERROR > WARN > INFO > DEBUG > TRACE)



# Logback 설정

- 디펜던시 설정

```
<properties>
  <java-version>11</java-version>
  <org.springframework-version>5.3.3</org.springframework-version>
  <org.aspectj-version>1.9.6</org.aspectj-version>
  <org.slf4j-version>1.7.30</org.slf4j-version>
</properties>
```

SLF4J 버전 수정

```
<!-- SLF4G -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>${org.slf4j-version}</version>
</dependency>
```


```
<!-- Logback -->
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.4.4</version>
  <scope>test</scope>
</dependency>
```

Log4j 디펜던시를  
Logback으로 변경


<scope>test</scope>는 제거

# Logback 설정

- src/main/resources

▼  src/main/resources

📁 META-INF

 logback.xml

src/main/resources/logback.xml 추가

- logback.xml 예시

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>
                %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
            </pattern>
        </encoder>
    </appender>

    <logger name="com.spring.app" level="INFO" />

    <root level="OFF">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```



# logback.xml

- logback.xml 구성

- ① appender : 로그 메시지를 출력할 대상(콘솔, 파일, 원격 소켓 서버, 메일, DB 등)
  - encoder : 출력 형식(Pattern)을 지정
- ② logger : 특정 패키지의 로그 처리, 로그 메시지를 appender에게 전달하는 로그의 주체
- ③ root : 전체 로그 처리, logger로 선택한 log 이외의 정보를 처리

- Appender 종류

종류	기능
ch.qos.logback.core.ConsoleAppender	콘솔에 로그 메시지 출력
ch.qos.logback.core.FileAppender	하나의 파일에 로그 메시지 출력
ch.qos.logback.core.rolling.RollingFileAppender	파일을 바꿔가며 로그 메시지 출력
ch.qos.logback.classic.net.SocketAppender	원격 서버에 로그 메시지 출력
ch.qos.logback.classic.net.SMTPAppender	로그 메시지를 이메일로 전송
ch.qos.logback.classic.db.DBAppender	DB에 로그 메시지를 출력