

# Spring Framework

## 웹 어플리케이션 개발

강경미(carami@nate.com)

# Spring Release History

- 1.0 : 2004년 4월
  - 2.0 : 2006년 6월
  - 2.5 : 2007년 11월
  - 3.0 : 2011년 12월
  - 3.1.4 : 2013년 1월
  - 4.0.1 : 2014년 1월
  - 5.0 : 2017년 9월
- 
- 2003년 6월 아파치 2.0 라이센스로 공개

# 스프링 버전별 새로운 기능 : 1.x

- IoC(Inverson of Control)
- 컨테이너(DI 컨테이너)
- AOP(Aspect Orientied Programming)
- XML기반의 빈 정의
- 프레임워크 모듈간의 결합도 약화
- 트랜잭션 관리
- 데이터 액세스 등
- 스트럿츠, 하이버네이트등과 조합하는 방식이 유행

# 스프링 버전별 새로운 기능 : 2.0

- 스프링 시큐리티(Spring Security), 스프링 웹 플로우(Spring Web Flow) 모듈 프로젝트 시작됨.
- 2.5(2007년) – DI(Dependency Injection)와 MVC(Model View Controller)를 애노테이션 방식으로 설정할 수 있게 됨.
- 시스템 연계 및 배치 처리를 위한 스프링 인테그레이션(Spring Integration)과 스프링 배치(Spring Batch)등의 스프링 프로젝트가 시작됨
- 로드존슨의 인터페이스21 -> 스프링소스(Spring Source)로 사명 변경 및 유럽에서 미국으로 이사(?)

# 스프링 버전별 새로운 기능 : 3.0

- 자바 기반의 설정(Java-based configuration)과 DI의 자바 사양(Spec)인 JSR 330을 지원
- JPA(Java Persistence API) 2.0과 빈 검증 기능(Bean Validator) 등 Java EE 6의 사양에 대해서 지원
- RESTful 프레임워크로 사용할 수 있도록 스프링 MVC가 개선

# 스프링 버전별 새로운 기능 : 4.0

스프링 프레임워크 4.0의 새로운 기능 및 향상된 기능

1. 향상된 시작경험
2. Deprecated된 패키지 및 메서드 제거
3. Java 8 지원
4. Java EE 6, 7 지원
5. Groovy 빈 정의 DSL
6. 코어 컨테이너 개선
7. 전반적인 웹 개선
8. 웹소켓, SockJS, STOMP 메시징
9. 테스트 개선

# 스프링 버전별 새로운 기능 : 4.1

스프링 프레임워크 4.1의 새로운 기능 및 향상된 기능

1. JMS 개선
2. 캐싱 개선
3. 웹 개선
4. 웹소켓 메시징 개선
5. 테스트 개선

# 스프링 버전별 새로운 기능 : 4.2

스프링 프레임워크 4.2의 새로운 기능 및 향상된 기능

1. 코어 컨테이너 개선
2. 데이터 액세스 개선
3. JMS 개선
4. 웹 개선
5. 웹소켓 메시징 개선
6. 테스트 개선

# 스프링 버전별 새로운 기능 : 4.3

스프링 프레임워크 4.3의 새로운 기능 및 향상된 기능

1. 코어 컨테이너 개선
2. 데이터 액세스 개선
3. 캐싱 개선
4. JMS 개선
5. 웹 개선
6. 웹소켓 메시징 개선
7. 테스트 개선
8. 새로운 라이브러리 및 서버 세대 지원

# 스프링 버전별 새로운 기능 : 5.0

1. JDK 8+9와 Java EE 7 베이스라인
2. 패키지, 클래스, 메서드 제거
3. 코어 컨테이너 개선
4. 일반적인 웹 개선
5. 리액티브(Reactive) 프로그래밍 모델
6. 테스트 개선

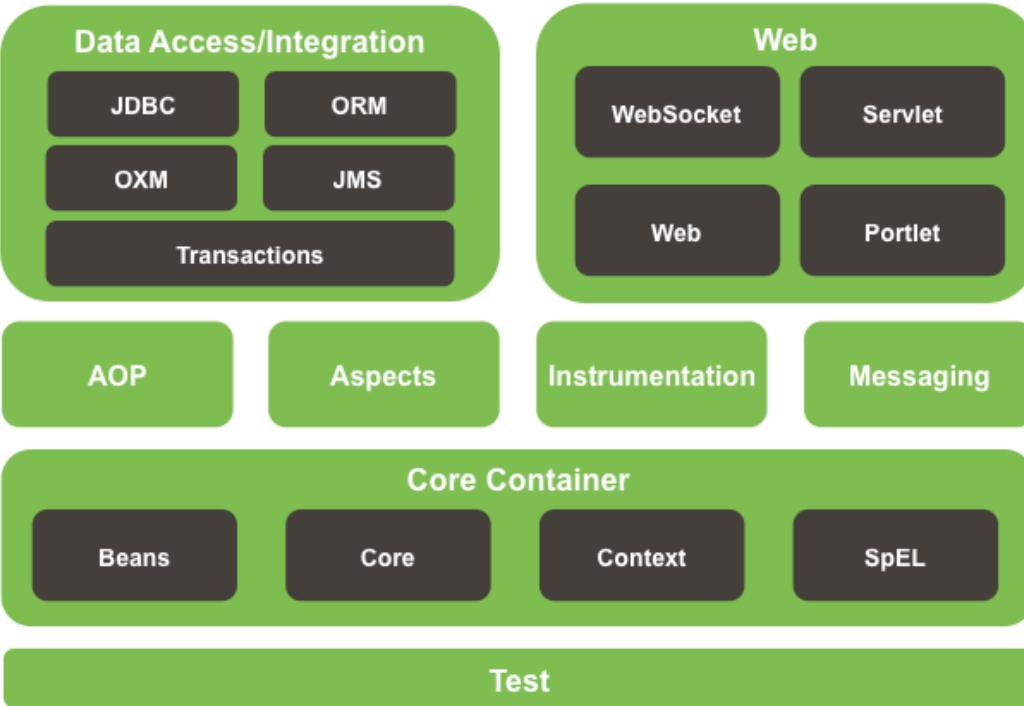
# Spring 프로젝트

# Spring Framework란?

- 엔터프라이즈급 어플리케이션을 구축할 수 있는 가벼운 솔루션이자, 원스-스탑-숍(One-Stop-Shop)
- 원하는 부분만 가져다 사용할 수 있도록 모듈화가 잘 되어 있다.
- POJO를 이용한 가볍고 non-invasive(비 침투적) 개발
- IoC 컨테이너이다.
- DI와 인터페이스 지향을 통한 느슨한 결합도
- 선언적으로 트랜잭션을 관리할 수 있다.
- 완전한 기능을 갖춘 MVC Framework를 제공한다.
- AOP와 공통 규약을 통한 선언적 프로그래밍
- 템플릿을 통한 상투적인 코드 축소
- 스프링은 도메인 논리 코드와 쉽게 분리될 수 있는 구조를 가지고 있다.

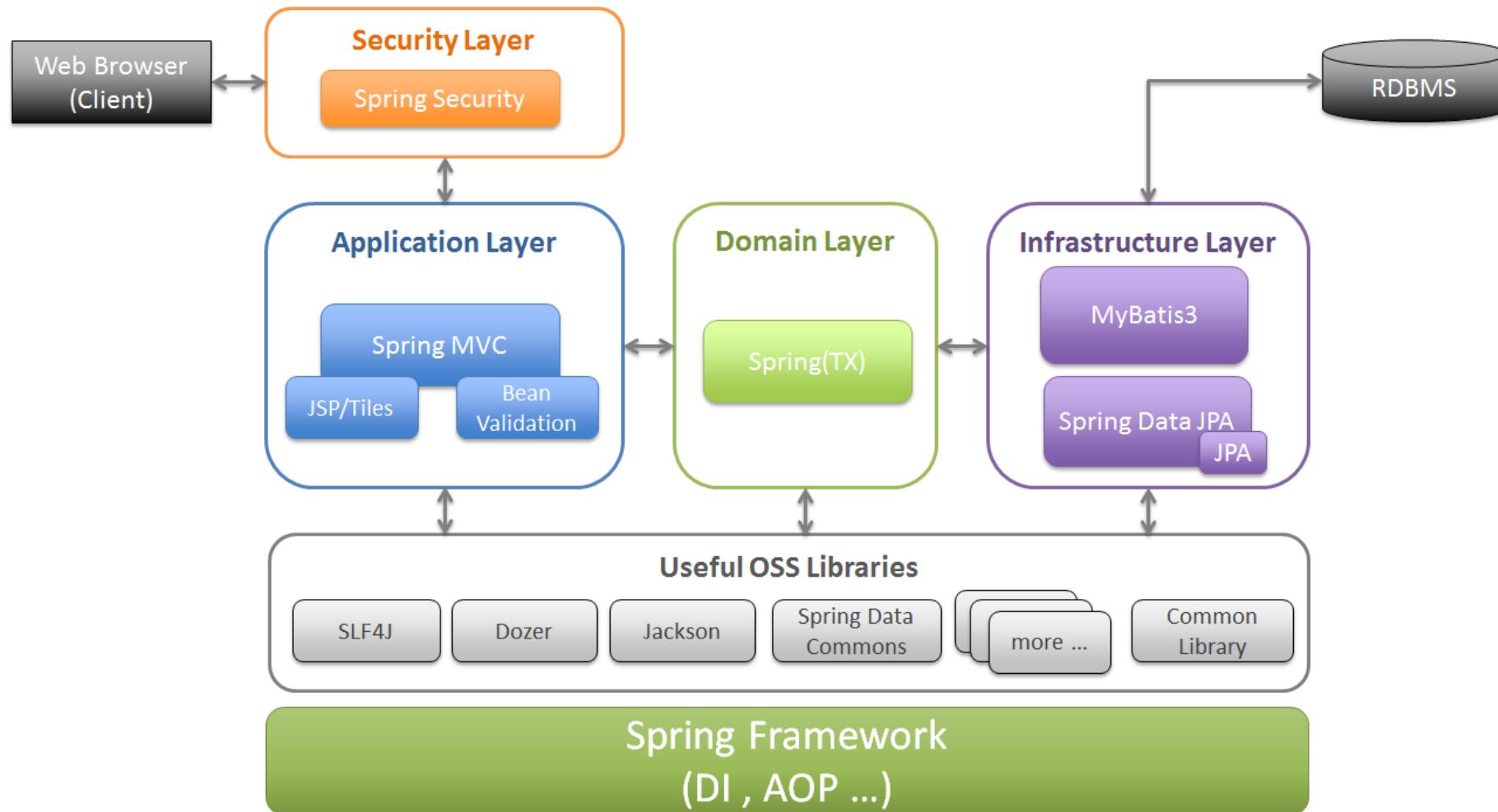
\* 원-스탑-숍 (One-Stop-Shop) : 모든 과정을 한꺼번에 해결하는 상점.

# 프레임 워크 모듈



- 스프링 프레임워크는 약 20 개의 모듈로 구성되어 있다.
- 필요한 모듈만 가져다 사용 할 수 있다.

# Spring Framework의 중요 구성 요소



# AOP 와 인스트루멘테이션 (Instrumentation)

- **spring-AOP** : AOP 얼라이언스(Aliance)와 호환되는 방법으로 AOP를 지원한다.
- **spring-aspects** : AspectJ와의 통합을 제공한다.
- **spring-instrument** : 인스트루멘테이션을 지원하는 클래스와 특정 WAS에서 사용하는 클래스로더 구현체를 제공한다. 참고로 BCI(Byte Code Instrumentations)은 런타임이나 로드(Load) 때 클래스의 바이트 코드에 변경을 가하는 방법을 말합니다.

# 메시징(Messaging)

- spring-messaging : 스프링 프레임워크 4는 메시지 기반 어플리케이션을 작성할 수 있는 Message, MessageChannel, MessageHandler 등을 제공한다. 또한, 해당 모듈에는 메소드에 메시지를 맵핑하기 위한 어노테이션도 포함되어 있으며, Spring MVC 어노테이션과 유사하다.

# 데이터 액세스(Data Access) / 통합 (Integration)

- 데이터 액세스/통합 계층은 JDBC, ORM, OXM, JMS 및 트랜잭션 모듈로 구성되어 있다.
- **spring-jdbc** : 자바 JDBC프로그래밍을 쉽게 할 수 있도록 기능을 제공한다.
- **spring-tx** : 선언적 트랜잭션 관리를 할 수 있는 기능을 제공한다.
- **spring-orm** : JPA, JDO및 Hibernate를 포함한 ORM API를 위한 통합 레이어를 제공한다.
- **spring-oxm** : JAXB, Castor, XMLBeans, JiBX 및 XStream과 같은 Object/XML 맵핑을 지원한다.
- **spring-jms** : 메시지 생성(producing) 및 사용(consuming)을 위한 기능을 제공. Spring Framework 4.1부터 **spring-messaging**모듈과의 통합을 제공한다.

# 웹(Web)

- 웹 계층은 spring-web, spring-webmvc, spring-websocket, spring-webmvc-portlet 모듈로 구성된다.
- **spring-web** : 멀티 파트 파일 업로드, 서블릿 리스너 등 웹 지향 통합 기능을 제공한다.  
HTTP클라이언트와 Spring의 원격 지원을 위한 웹 관련 부분을 제공한다.
- **spring-webmvc** : Web-Servlet모듈이라고도 불리며, Spring MVC및 REST 웹 서비스 구현을 포함한다.
- **spring-websocket** : 웹 소켓을 지원한다.
- **spring-webmvc-portlet** : 포틀릿 환경에서 사용할 MVC구현을 제공한다.

# Spring Framework sub-projects 1/2

- Spring IO Platform - 스프링 프레임워크 의존성 관리
- Spring Boot
- Spring Framework
- Spring Cloud Data Flow - Big Data
- Spring Cloud - 클라우드 기반
- Spring Data - JPA, MongoDB, Redis, Elasticsearch ...

# Spring Framework sub-projects 2/2

- Spring Integration - Enterprise Integration Pattern
- Spring Batch
- Spring Security
- Spring HATEOAS
- Spring Social
- Spring AMQP
- Spring Mobile

# Spring Boot

- 최소한의 설정만으로 프로덕션 레벨의 스프링 기반 애플리케이션을 쉽게 개발할 수 있게 도와주는 스프링 프레임워크.
- 2014년 1.0 발표
- 쉽게 개발할 수 있다는 장점으로 인해 점점 지지를 받고 성장중

# JAVA EE와의 관계

- JAVA EE의 안티로 스프링이 개발되기 시작했지만 J2EE를 부정하는 것은 아니다.
- 스프링 프레임워크도 J2EE기반의 자바 애플리케이션 프레임워크이다.
- JAVA EE도 스프링의 장점을 받아들여 스프링 프레임워크와 비슷한 스타일로 개발할 수 있다.
- 스프링과 Java EE의 차이가 줄어들고 있다.
- 스프링은 상대적으로 신기술을 도입하는 속도가 상대적으로 빠르다. 새로운 기능과 아키텍처가 필요할 경우라면 스프링을 사용하는 것이 최선의 선택이다.

# 스프링 코어(DI, AOP)

# 컨테이너(Container)란?

- 컨테이너는 인스턴스의 생명주기를 관리한다.
- 생성된 인스턴스들에게 추가적인 기능을 제공한다.

# IoC란?

- IoC란 Inversion of Control의 약어이다. inversion은 사전적 의미로는 '도치, 역전'이다. 보통 IoC를 제어의 역전이라고 번역한다.
- 개발자는 프로그램의 흐름을 제어하는 코드를 작성한다. 그런데, 이 흐름의 제어를 개발자가 하는 것이 아니라 다른 프로그램이 그 흐름을 제어하는 것을 IoC라고 말한다.

# POJO

- POJO = Plain Old Java Object
- <https://www.martinfowler.com/bliki/POJO.html>

```
public class HelloWorld{  
}
```

웹개발시 servlet

```
public class HelloServlet extends HttpServlet{  
}
```

# DI란?

- DI는 Dependency Injection의 약자로, 의존성 주입이란 뜻을 가지고 있다.
- DI는 클래스 사이의 의존 관계를 빈(Bean)설정 정보를 바탕으로 컨테이너가 자동으로 연결해주는 것을 말한다.

# DI 컨테이너에서 인스턴스를 관리할 때의 장점

- 인스턴스의 스코프를 제어할 수 있다.
- 인스턴스의 생명 주기를 제어할 수 있다.
- AOP방식으로 공통 기능을 집어넣을 수 있다.
- 의존하는 컴포넌트 간의 결합도를 낮춰서 단위 테스트를 쉽게 만든다.

# DI 컨테이너

- 스프링 공식문서에서는 IoC컨테이너라고 기재하고 있음
- 개발자들은 보통 DI컨테이너라고 말함
- 스프링 프레임워크 외의 DI컨테이너
  - CDI(Contexts & Dependency Injection)
  - Google Guice
  - Dagger

# DI 예

- DI가 적용 안 된 예.
- 개발자가 직접 인스턴스를 생성한다.
- Spring에서 DI가 적용된 예.
- 엔진 type의 v5변수에 아직 인스턴스가 할당되지 않았다.
- 컨테이너가 v5변수에 인스턴스를 할당해주게 된다.

```
class 엔진{
```

```
}
```

```
class 자동차{
```

```
    엔진 v5 = new 엔진()
```

```
}
```

```
@Component
```

```
class 엔진{
```

```
}
```

```
@Component
```

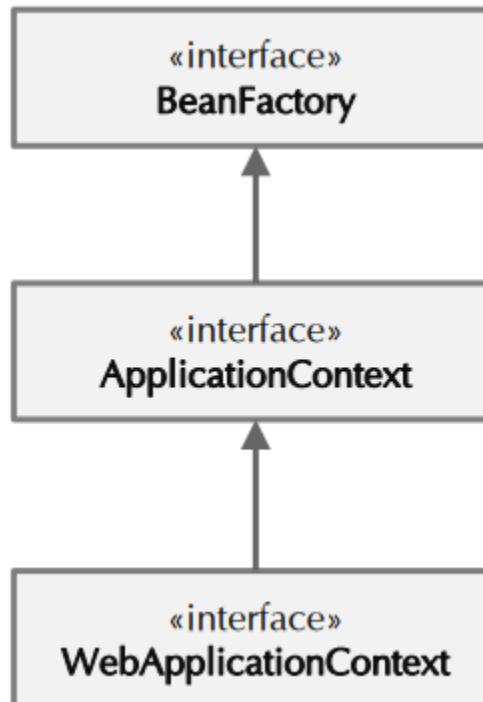
```
class 자동차{
```

```
    @Autowired
```

```
    엔진 v5;
```

```
}
```

# IoC 컨테이너



- 빈(bean) 객체에 대한 생성과 제공을 담당
  - 단일 유형의 객체를 생성하는 것이 아니라, 여러 유형의 빈(bean)을 생성, 제공
  - 객체 간의 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 생성 시 모든 빈 정보를 메모리에 로딩함
- 
- 웹 환경에서 사용할 때 필요한 기능이 추가된 애플리케이션 컨텍스트
  - 가장 많이 사용하며, 특히 XmlWebApplicationContext를 가장 많이 사용

# Spring에서 제공하는 IoC/DI 컨테이너

- BeanFactory : IoC/DI에 대한 기본 기능을 가지고 있다.
- ApplicationContext : BeanFactory의 모든 기능을 포함하며, 일반적으로 BeanFactory보다 추천된다. 트랜잭션처리, AOP등에 대한 처리를 할 수 있다. BeanPostProcessor, BeanFactoryPostProcessor등을 자동으로 등록하고, 국제화 처리, 어플리케이션 이벤트 등을 처리할 수 있다.

# ApplicationContext

- AnnotationConfigApplicationContext - 하나 이상의 Java Config 클래스에서 스프링 애플리케이션 컨텍스트를 로딩
- AnnotationConfigWebApplicationContext - 하나 이상의 Java Config 클래스에서 웹 애플리케이션 컨텍스트를 로딩
- ClassPathXmlApplicationContext - 클래스패스에 위치한 xml파일에서 컨텍스트를 로딩
- FileSystemXmlApplicationContext - 파일 시스템에서 지정된 xml파일에서 컨텍스트를 로딩
- XmlWebApplicationContext - 웹 애플리케이션에 포함된 xml파일에서 컨텍스트를 로딩

# 사용법

```
ApplicationContext context = new  
FileSystemXmlApplicationContext("/spring/context.xml");
```

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

```
ApplicationContext context = new  
AnnotationConfigApplicationContext(Config.class);
```

# 빈 설정 방법의 차이

- 자바 기반 설정 방식
  - 자바 클래스에 @Configuration 애노테이션을 메소드에 @Bean 애노테이션을 사용해서 빈을 정의하는 방법으로 스프링 3.0부터 사용할 수 있다. 최근에는 스프링 기반 애플리케이션 개발에 자주 사용되고 특히 스프링 부트에서 이 방식을 많이 사용하고 있다.
- XML 기반 설정 방식
  - XML파일을 사용하는 방법으로 <bean>요소의 속성에 FQCN(Fully-Qualified Class Name)을 기술하면 빈이 정의된다. <constructor-arg>나 <property>요소를 사용해 의존성을 주입한다. 스프링 프레임워크 1.0부터 사용할 수 있다.
- 애노테이션 기반 설정 방식
  - @Component 같은 마커(Marker) 애노테이션이 부여된 클래스를 탐색해서(Component scan) DI컨테이너에 빈을 자동으로록하는 방법이다. 스프링 프레임워크 2.5부터 사용할 수 있다.

# 의존성 주입 방법

- 설정자 기반 의존성 주입 방식(setter-based dependency injection)
- 생성자 기반 의존성 주입 방식(constructor-based dependency injection)
- 필드 기반 의존성 주입 방식(field-based dependency)

# 오토와이어링(autowiring)

- 자바 기반 설정 방식에서 @Bean메소드를 사용하거나 XML기반 설정 방식에서 <bean>요소를 사용하는 것처럼 명시적으로 빈을 정의하지 않고도 DI컨테이너에 빈을 자동으로 주입하는 방식이다.
- 오토와이어링 방식
  - 타입을 사용한 방식(autowiring by type)
  - 이름을 사용한 방식(autowiring by name)

# 타입을 사용한 오토와이어링 방식 1/2

- @Autowired 애노테이션은 타입으로 오토와이어링을 하는 방식
- Setter 인젝션, 생성자 인젝션, Field 인젝션에서 모두 활용 가능
- 타입으로 오토와이어링을 할 때는 기본적으로 의존성 주입이 반드시 성공한다고 가정. 주입할 타입에 해당하는 빈을 찾지 못하면 NoSuchBeanDefinitionException이 발생한다. 필수 조건을 사용하고 싶지 않을 경우 @Autowired(required = false)로 설정한다.
- 스프링 4부터는 required=false를 사용하는 대신 JDK 8부터 추가된 java.util.Optional을 사용할 수 있다.

```
@Autowired  
Optional<UserService> UserService;
```

- 인젝션을 할 수 있는 여러개의 빈을 발견하게 될 경우에는 NoUniqueBeanDefinitionException이 발생한다. 여러개일 경우 @Qualifier 애노테이션으로 빈 이름을 지정한 후 선택해서 사용해야 한다.
- @Primary 애노테이션을 사용하면 @Qualifier를 사용하지 않았을 때 우선적으로 선택할 빈을 지정할 수 있다.

# 타입을 사용한 오토와이어링 방식 2/2

- @Qualifier역할을 하는 사용자 정의 애노테이션을 사용해서 표현할 수도 있다. 사용자 정의 애노테이션에 @Qualifier를 설정 한다.

# 이름으로 오토와이어링 하기

- 빈의 이름이 필드명이나 프로퍼티명과 일치할 경우에 빈 이름으로 필드 인젝션을 할 수 있다.
- JSR-250사양을 지원하는 @Resource애노테이션을 활용한다.
- @Resource애노테이션에는 name속성을 생략할 수 있는데, 필드 인젝션을 하는 경우에는 필드이름과 같은 이름의 빈이 선택되고, Settter인젝션을 하는 경우에는 프로퍼티 이름과 같은 이름의 빈이 선택된다.
- 생성자 인젝션에서는 @Resource애노테이션을 사용할 수 없다.

# 컴포넌트 스캔(Component Scan) 1/2

- 클래스 로더(Class Loader)를 스캔하면서 특정 클래스를 찾은 다음 DI 컨테이너에 등록하는 방법을 말한다.
- 별도의 설정의 없는 기본 설정에서는 다음과 같은 애노테이션이 붙은 클래스가 탐색 대상이 되고 탐색된 컴포넌트는 DI컨테이너에 등록된다.
  - @Component, @Controller, @Service, @Repository, @RestController
  - @Configuration
  - @ControllerAdvice
  - @ManagedBean(java.annotation.ManagedBean)
  - @Named(javax.inject.Named)

# 컴포넌트 스캔(Component Scan) 2/2

- 다음과 같은 방법으로 특정 패키지 이하를 스캔한다.

```
@ComponentScan(basePackages = "examples.di")
```

- 기본 스캔 대상 외에도 추가로 다른 컴포넌트를 포함하고 싶을 경우 필터를 적용한 컴포넌트 스캔할 수 있다. 스프링 프레임워크는 다음과 같은 필터를 제공한다.

- 애노테이션을 활용할 필터(ANNOTATION)
- 할당 가능한 타입을 활용한 필터(ASSIGNABLE\_TYPE)
- 정규 표현식 패턴을 활용한 필터(REGEX)
- AspectJ패턴을 활용한 필터(ASPECTJ)

```
@ComponentScan(basePackages = "examples.di" includeFilters = {  
    @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE,  
        classes = { MyService.class })  
})
```

- 기본 스캔 대상에 필터를 적용해 특정 컴포넌트를 추가하는 것과 반대로 excludeFilters 속성을 이용해서 걸러낼 수도 있다.

# Bean Scope 1/2

- DI컨테이너는 빈의 생존 기간도 관리한다. 빈의 생존 기간을 빈 스코프(Bean Scope)라고 한다.
- 스프링 프레임워크에서 사용 가능한 스코프
  - singleton
    - DI컨테이너를 기동할 때 빈 인스턴스 하나가 만들어지고, 이후 부터는 그 인스턴스를 공유하는 방식이다. 기본 스코프다.
  - prototype
    - DI컨테이너에 빈을 요청할 때마다 새로운 빈 인스턴스가 만들어진다. 멀티 스레드 환경에서 오동작이 발생하지 않아야 하는 빈일 경우 사용한다.
  - request
    - HTTP 요청이 들어올 때마다 새로운 빈 인스턴스가 만들어진다. 웹 애플리케이션을 만들 때만 사용할 수 있다.
  - session
    - HTTP 세션이 만들어질 때마다 새로운 빈 인스턴스가 만들어진다. 웹 애플리케이션을 만들 때만 사용할 수 있다.

# Bean Scope 2/2

- 스프링 프레임워크에서 사용 가능한 스코프
  - global session
    - 포틀렛 환경에서 글로벌 HTTP 세션이 만들어질 때마다 새로운 빈 인스턴스가 만들어 진다. 포틀릿을 사용한 웹 애플리케이션을 만들 때만 사용할 수 있다.
  - application
    - 서블릿 컨텍스트(Servlet Context)가 만들어질 때마다 빈 인스턴스가 만들어진다. 웹 애플리케이션을 만들 때만 사용할 수 있다.
  - custom
    - 스코프 이름을 직접 정할 수 있고 정의한 규칙에 따라 빈 인스턴스를 만들 수 있다.

# 스코프 설정

- 자바 기반의 설정에서는 @Bean 애노테이션이 붙은 메소드에 @Scope 애노테이션을 추가해서 스코프를 명시한다.

```
@Bean  
@Scope("prototype")  
MyService myService(){  
    return new MyService();  
}
```

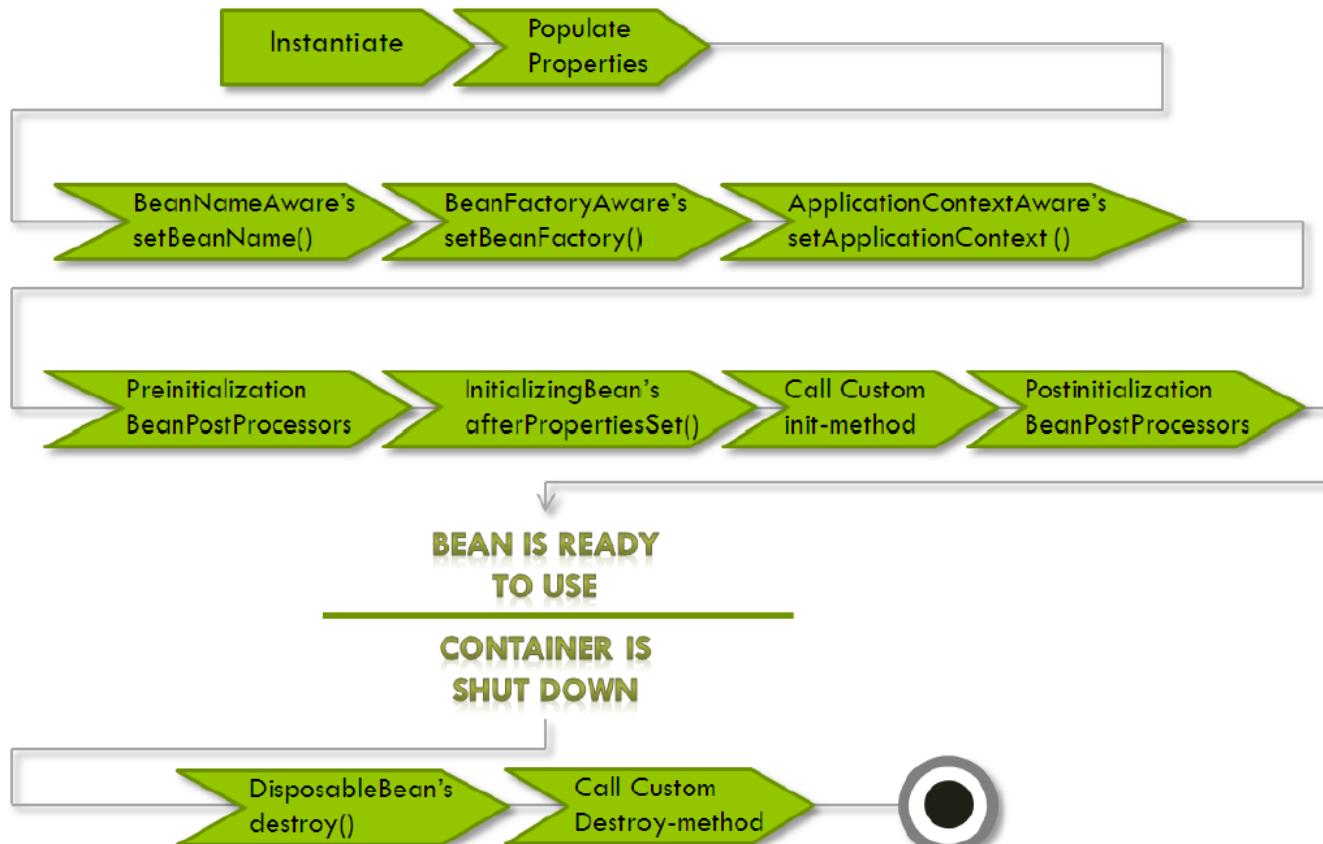
# 다른 스코프의 빈 주입

- 스코프 별 오래사는 순서  
singleton > session > request
- DI 컨테이너에 의해 주입된 빈은 자신의 스코프와 상관없이 주입받는 빈의 스코프를 따르게 된다.
  - prototype 스코프 빈을 singleton 스코프 빈에 주입할 경우 prototype 스코프 빈은 singleton 빈이 살아있는 한 다시 만들 필요가 없기 때문에 결과적으로 singleton과 같은 수명을 살게 된다.

# Bean의 생명주기

- DI 컨테이너에서 관리되는 빈의 생명주기는 크게 다음의 세 가지 단계로 구분할 수 있다.
  1. 빈 초기화 단계(initialization)
  2. 빈 사용 단계(activation)
  3. 빈 종료 단계(destruction)

# Bean의 생명주기



1. 스프링이 빈을 인스턴스화 한다.
2. 스프링이 값과 빈의 레퍼런스를 빈의 프로퍼티로 주입한다.
3. 빈이 BeanNameAware를 구현하면 스프링이 빈의 ID를 setBeanName()메소드에 넘긴다.
4. 빈이 BeanFactoryAware를 구현하면 setBeanFactory()메소드를 호출하여 빈 팩토리 전체를 넘긴다.
5. 빈이 ApplicationContextAware를 구현하면 스프링이 setApplicationContext()메소드를 호출하고 둘러싼 애플리케이션 컨텍스트에 대한 참조를 넘긴다.
6. 빈이 Bean PostProcessor인터페이스를 구현하면 스프링은 postProcessBeforeInitialization()메소드를 호출한다.
7. 빈이 InitializingBean인터페이스를 구현하면 스프링은 afterPropertiesSet()메소드를 호출한다. 마찬가지로 빈이 init-method와 함께 선언됐으면 지정한 초기화 메소드가 호출된다.
8. 빈이 BeanPostProcessor를 구현하면 스프링은 postProcessAfterInitialization()메소드를 호출한다.
9. 빈은 애플리케이션에서 사용할 준비가 된 것이며, 애플리케이션 컨텍스트가 소멸될 때까지 애플리케이션 컨텍스트에 남아있게 된다.
10. 빈이 DisposableBean인터페이스를 구현하면 스프링은 destroy()메소드를 호출한다. 마찬가지로 빈이 destroy-method와 함께 선언됐으면 지정된 메소드가 호출된다.

# 생명주기 관련 애노테이션

- `@PostConstruct` - JSR-250 스펙으로 JSR-250을 구현하고 있는 다른 프레임워크에서도 사용 가능하다. 인스턴스 생성 후에 호출된다.
- `@Bean(initMethod)` - `@Bean(initMethod="init")` 과 같은 형태로 사용함으로써 초기화 메소드를 사용할 수 있다. 아래는 Java Config에서의 사용 예이다.

```
@Bean(initMethod="init")  
  
public MyBean mybean(){  
  
    return new MyBean();  
  
}
```

- `@PreDestroy` - JSR-250 스펙에서 정의되어 있으며 종료될 때 사용할 메소드 위에 사용하면 된다.
- `@Bean(destroyMethod)` - `@Bean(destroyMethod="destroy")`과 같은 형태로 사용함으로써 종료될 때 호출되도록 할 수 있다.

# 빈 설정 분할

- DI 컨테이너에서 관리하는 빈이 많아지면 많아질수록 설정 내용도 많아져서 관리하기가 어려워진다. 이럴 때는 빈 설정 범위를 명확히 하고 가독성도 높이기 위해 목적에 맞게 분활하는 것이 좋다.
- 자바 기반 설정의 분할
  - @Import 애노테이션을 사용한다.
- XML 기반 설정의 분할
  - <import>요소를 사용한다.

# Profile별 설정 구성

- 스프링 프레임워크에서는 설정 파일을 특정 환경이나 목적에 맞게 선택적으로 사용할 수 있도록 그룹화할 수 있으며, 이 기능을 Profile이라한다.
- 자바 기반 설정 방식에서 프로파일을 지정할 때는 @Profile 애노테이션을 사용한다.  
`@Profile("dev")`  
`@Profile("dev", "real")`
- XML기반 설정에서는 Mbeans>요소의 profile속성을 활용한다.

# Profile 선택

- 자바 명령행 옵션으로 프로파일을 지정하는 방법  
-Dspring.profiles.active=real
- 환경 변수로 프로파일을 지정하는 방법  
export SPRING\_PROFILES\_ACTIVE=real
- web.xml 파일에 프로파일을 지정하는 방법  

```
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>real</param-value>
</context-param>
```
- spring.profiles.active를 지정하지 않으면 기본값으로 spring.profiles.default에 지정된 프로파일을 사용한다.

# JSR 330

- 자바 표준 사양 중에는 DI와 관련해서 JSR 330이 있다.
- DI 기능을 사용하기 위한 API가 정의돼 있다. 스프링이 아닌 다른 컨테이너들도 해당 표준을 구현하고 있다.

@Autowired : @Inject (JSR 330)

@Component : @Named (JSR 330)

@Qualifier : @Named (JSR 330)

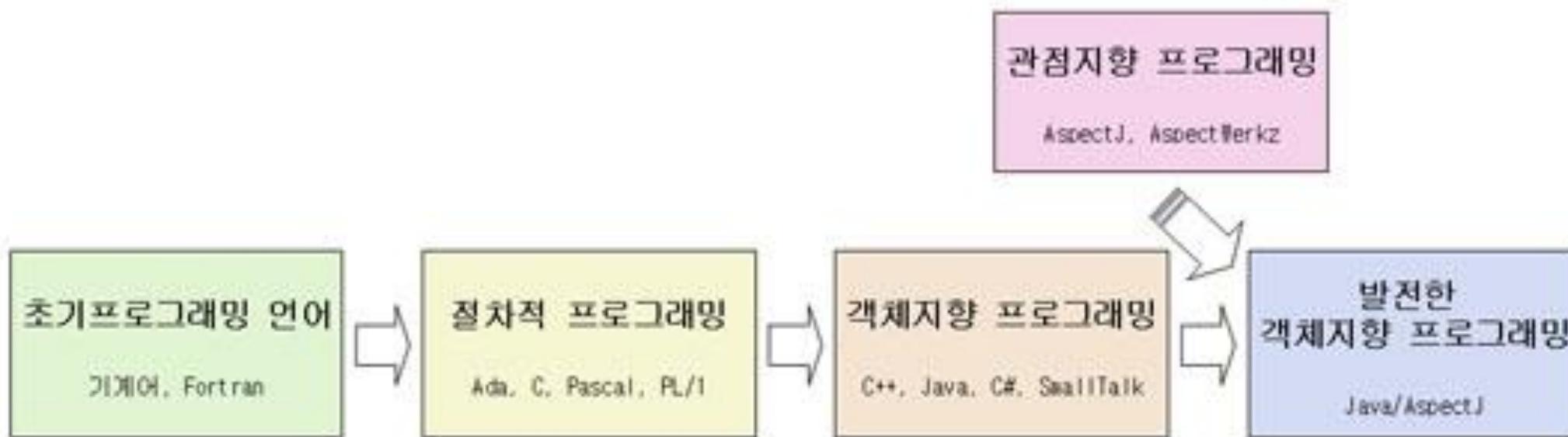
@Scope : @Scope(JSR 330)

# AOP

# AOP란?

- 관점지향 프로그래밍(Aspect Oriented Programming, 이하 AOP)
- 자체적인 언어라기보다는 기존의 OOP언어를 보완하는 확장 형태로 사용되고 있다.
- 자바진영에서 사용되는 AOP도구 중 대표적인 것으로 AspectJ, JBossAOP, SpringAOP가 존재 한다.

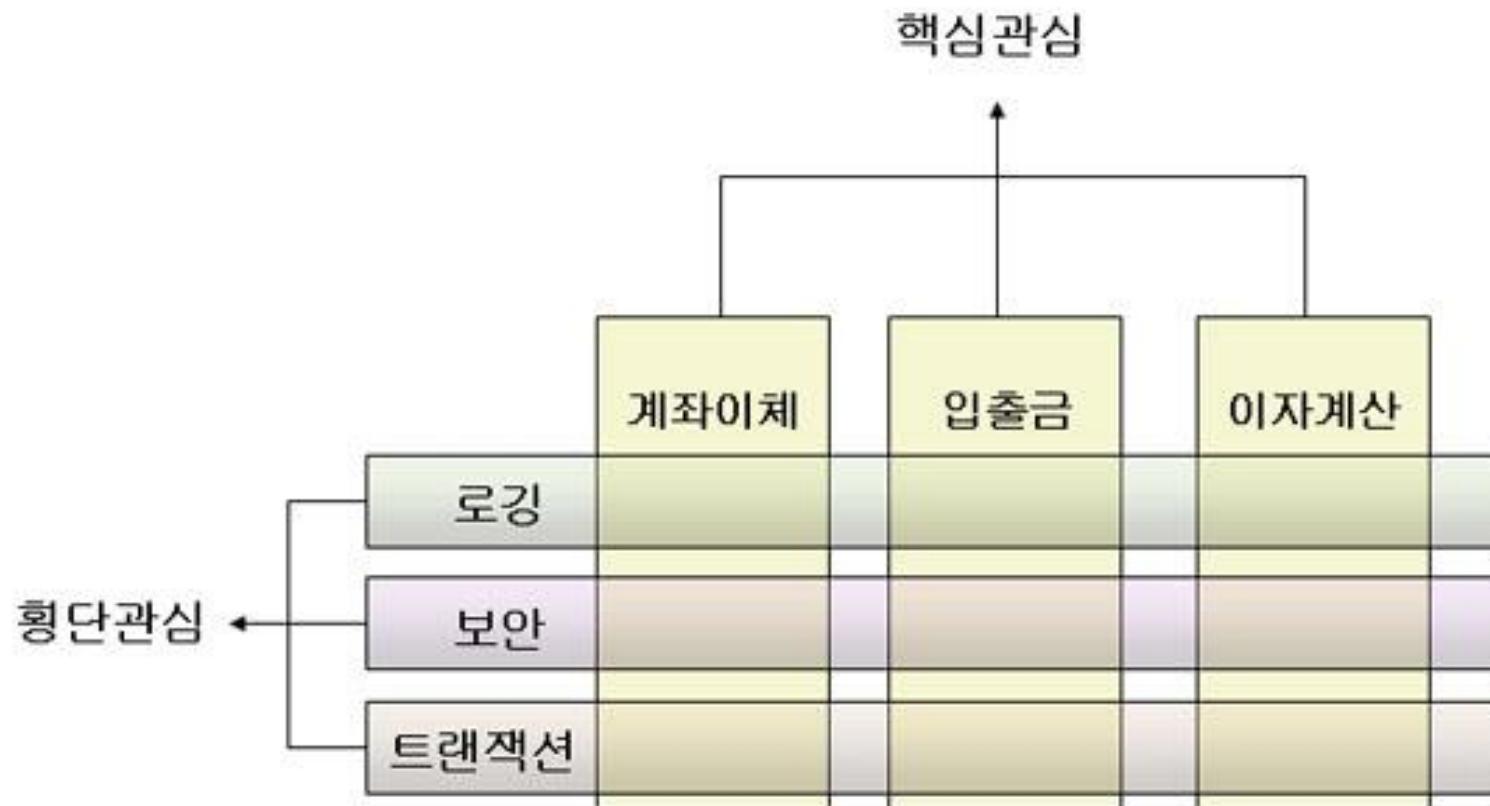
# AOP란?



# AOP의 목적 및 장점

- "중복을 줄여서 적은 코드 수정으로 전체 변경을 할 수 있게하자"라는 목적에서 출발
- AOP의 필요성을 이해하는 가장 기초가 되는 개념은 '관심의 분리(Separation of Concerns)'이다
- 핵심관점(업무로직) + 횡단관점(트랜잭션/로그/보안/인증 처리 등)으로 관심의 분리를 실현
- 중복되는 코드 제거, 효율적인 유지보수, 높은 생산성, 재활용성 극대화, 변화 수용이 용이 등의 장점이 있다.

# AOP의 목적 및 장점



# AOP 용어

- **Joinpoint**

- 메소드를 호출하는 '시점', 예외가 발생하는 '시점'과 같이 애플리케이션을 실행할 때 특정 작업이 실행되는 '시점'을 의미한다.

- **Advice**

- Joinpoint에서 실행되어야 하는 코드
- 횡단관점에 해당함 (트랜잭션/로그/보안/인증등..)

- **Target**

- 실질적인 비지니스 로직을 구현하고 있는 코드
- 핵심관점에 해당함 (업무로직)

# AOP 용어

- **Pointcut**

- Target 클래스와 Advice가 결합(Weaving)될 때 둘 사이의 결합규칙을 정의하는 것이다
- 예로 Advice가 실행된 Target의 특정 메소드등을 지정

- **Aspect**

- Advice와 Pointcut을 합쳐서 하나의 Aspect라고 한다.
- 즉 일정한 패턴을 가지는 클래스에 Advice를 적용하도록 지원할 수 있는 것을 Aspect라고 한다.

- **Weaving**

- AOP에서 Joinpoint들을 Advice로 감싸는 과정을 Weaving이라고 한다.
- Weaving 하는 작업을 도와주는 것이 AOP 툴이 하는 역할이다

# Spring AOP의 특징

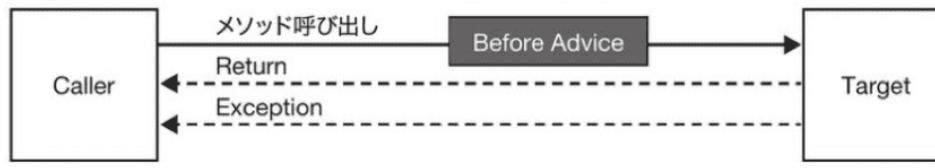
- 스프링은 Aspect의 적용 대상(target)이 되는 객체에 대한 Proxy를 만들어 제공.
- 대상객체(Target)를 사용하는 코드는 대상객체(Target)를 Proxy를 통해서 간접적으로 접근.
- Proxy는 공통기능(Advice)을 실행한 뒤 대상객체(Target)의 실제 메서드를 호출하거나 또는 대상객체(Target)의 실제 메소드가 호출된 뒤 공통기능(Advice)을 실행

# 스프링 프레임워크에서 지원하는 어드바이스 유형

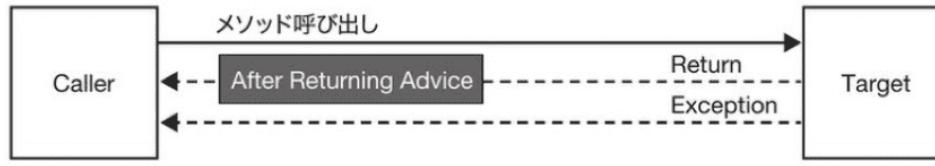
- Before
  - 조인 포인트 전에 실행된다. 예외가 발생하는 경우만 제외하고 항상 실행된다.
- After Returning
  - 조인 포인트가 정상적으로 종료한 후에 실행된다. 예외가 발생하면 실행되지 않는다.
- After Throwing
  - 조인 포인트에서 예외가 발생했을 때 실행된다. 예외가 발생하지 않고 정상적으로 종료하면 실행되지 않는다.
- After
  - 조인 포인트에서 처리가 완료된 후 실행된다. 예외 발생이나 정상 종료 여부와 상관없이 항상 실행된다.
- Around
  - 조인 포인트 전후에 실행된다.

# 어드바이스의 유형별 동작방식

## ■ Before Advice



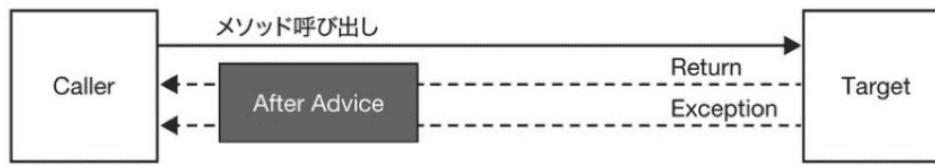
## ■ After Returning Advice



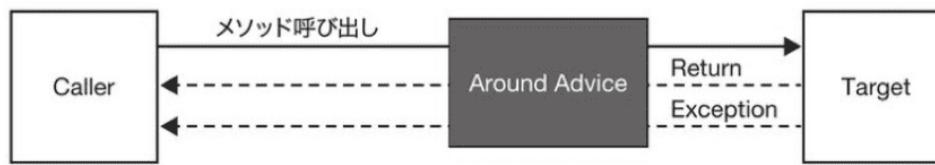
## ■ After Throwing Advice



## ■ After Advice



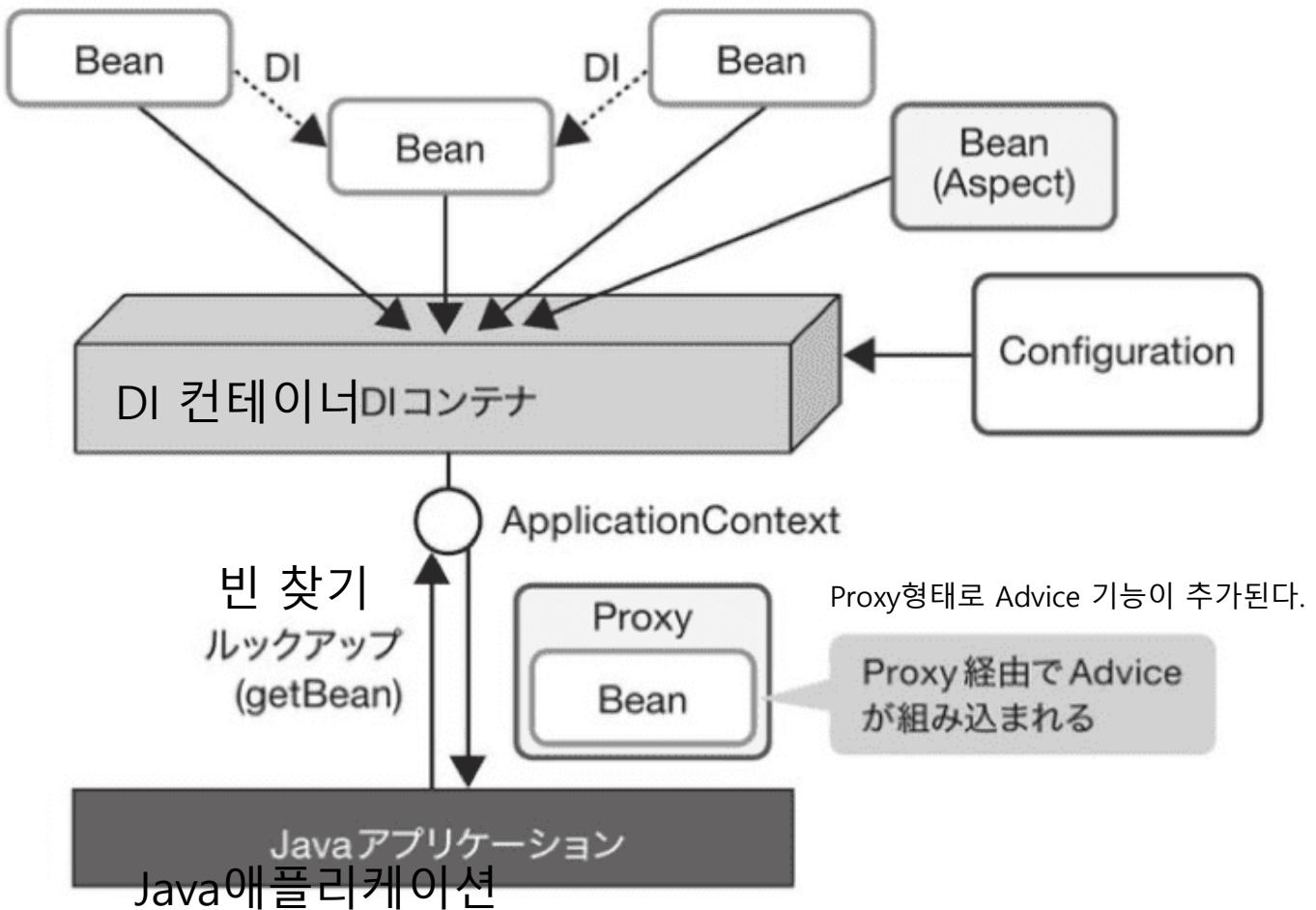
## ■ Around Advice



- 일본어 부분은 "메소드 호출 "

# Spring AOP

- 스프링 프레임워크 안에는 AOP를 지원하는 모듈로 스프링 AOP가 포함돼 있다.
- 스프링 AOP에는 DI컨테이너에서 관리하는 빈들을 타깃으로 어드바이스를 적용하는 기능이 있는데, 조인 포인트에 어드바이스를 적용하는 방법은 프락시 객체를 만들어서 대체하는 방법을 쓴다.
- 스프링 AOP에는 실제 개별 현장에서 폭넓게 사용돼온 AspectJ라는 AOP프레임워크가 포함돼 있다.
- AspectJ는 에스펙트와 어드바이스를 정의하기 위한 애노테이션이나 포인트컷 표현언어, 위빙 메커니즘 등을 제공하는 역할을 한다.



# 자바 기반 설정 방식에서의 어드바이스 정의 1/3

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class ServiceMonitor {
    @AfterReturning("execution(* examples..*Service.*(..))")
    public void logServiceAccess(JoinPoint joinPoint) {
        System.out.println("Completed: " + joinPoint);
    }
}
```

# 자바 기반 설정 방식에서의 어드바이스 정의 2/3

- Spring Boot에서 의존성을 추가하려면 다음의 의존성을 추가한다.

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

- @Aspect

- 스프링 빈에 @Aspect를 명시하면 해당 빈이 Aspect로 작동한다.
- 클래스 레벨에 @Order를 명시하여 @Aspect 빈 간의 작동 순서를 정할 수 있다. int 타입의 정수로 순서를 정할 수 있는데 값이 낮을수록 우선순위가 높다. 기본값은 가장 낮은 우선순위를 가지는 Ordered.LOWEST\_PRECEDENCE이다.
- @Aspect가 명시된 빈에는 어드바이스(Advice)라 불리는 메소드를 작성할 수 있다. 대상 스프링 빈의 메소드의 호출에 끼어드는 시점과 방법에 따라 @Before, @After, @AfterReturning, @AfterThrowing, @Around 등을 명시할 수 있다.

- @Before

- 대상 객체의 메서드 호출 전에 공통 기능을 실행

# 자바 기반 설정 방식에서의 어드바이스 정의 3/3

- @After
  - 어드바이스를 명시하면 대상 메소드의 실행 후에 끼어 들어 원하는 작업을 할 수 있다. 역시 끼어들기만 할 뿐 대상 메소드의 제어나 가공은 불가능하다.
- @AfterReturning
  - @Aspect 클래스의 메소드 레벨에 @AfterReturning을 명시하면 해당 메소드의 실행이 종료되어 값을 리턴할 때 끼어 들 수 있다. 리턴 값을 확인할 수 있을 뿐 대상 메소드의 제어나 가공은 불가능하다.
- @Around
  - @Around 어드바이스는 앞서 설명한 어드바이스의 기능을 모두 포괄하는 종합선물세트와도 같다. 대상 메소드를 감싸는 느낌으로 실행 전후 시점에 원하는 작업을 할 수 있다. 대상 메소드의 실행 제어 및 리턴 값 가공도 가능하다.
- JoinPoint
  - 대상 객체 및 호출되는 메서드에 대한 정보나 전달되는 파라미터에 대한 정보가 필요한 경우 org.aspectj.lang.JoinPoint를 파라미터로 추가

# 포인트컷 표현식

- 포인트 컷을 사용하기 위해 "execution(\* examples..\*Service.\*(..))" 표현을 사용했다. 이처럼 표현식을 이용한 조인 포인트 선택 기능은 AspectJ가 제공하며, Spring AOP는 Aspect가 제공하는 포인트 컷 표현식을 상당수 지원한다.
- 메소드 명으로 조인 포인트 선택
  - execution(\* examples.user.UserService.\*(..))
    - examples.user.UserService클래스에서 임의의 메소드를 대상으로 한다.
  - execution(\* examples.user.UserService.find\*(..))
    - examples.user.UserService클래스에서 find로 시작하는 메소드를 대상으로 한다.
  - execution(String examples.user.UserService.\*(..))
    - examples.user.UserService클래스에서 반환값의 타입이 String인 메소드로 대상으로 한다.
  - execution(\* examples.user.UserService.\*(String, ..))
    - examples.user.UserService클래스에서 첫 번째 매개변수의 타입이 String인 메소드를 대상으로 한다.

# 포인트컷 표현식에 사용되는 와일드 카드

- \* : 기본적으로 임의의 문자열을 의미한다. 패키지를 표현할 때는 임의의 패키지 1개 계층을 의미한다. 메소드의 매개변수를 표현할 때는 임의의 인수 1개를 의미한다.
- .. : 패키지를 표현할 때 임의의 패키지 0개 이상 계층을 의미한다. 메소드의 매개변수를 표현할 때는 임의의 인수 0개 이상을 의미한다.
- + : 클래스명 뒤에 붙여 쓰며, 해당 클래스와 해당 클래스의 서브 클래스 혹은 구현 클래스 모두를 의미한다.

# 타입으로 조인 포인트 생성

- 타입 정보를 활용해 조인 포인트를 선택할 수 있는데 이때는 `within`지시자를 사용한다. `within`지시자는 클래스명의 패턴만 사용하기 때문에 `execution`지시자에 비해 상대적으로 표현식이 간결하다.
- `within(examples.service..*)`
  - 임의의 클래스에 속한 임의의 메소드를 대상으로 한다. 단 임의의 클래스는 service패키지나 이 패키지의 서브 패키지에 속한다.
- `within(examples.service.UserServiceImpl)`
  - `UserServiceImpl`클래스의 메소드를 대상으로 한다. 단 `UserServiceImpl`클래스는 examples.service패키지에 속한다.
- `within(examples.service.UserService)`
  - `UserService`인터페이스를 구현한 클래스의 메소드를 대상으로 한다. 단 `UserService`인터페이스는 examples.service패키지에 속한다.

# 그 밖의 기타 방법으로 조인 포인트 선택

- bean(\*Service)
  - DI 컨테이너에 관리되는 빈 가운데 이름이 'Service'로 끝나는 빈의 메소드를 대상으로 한다.
- @annotation(examples.annotation.TraceLog)
  - @TraceLog 애노테이션이 붙은 메소드를 대상으로 한다.
- @within(examples.annotation.TraceLog)
  - @TraceLog 애노테이션이 붙은 클래스의 메소드를 대상으로 한다.

# Named Pointcut 활용

- 포인트 컷에 이름을 붙여두면 나중에 그 이름으로 포인트 컷을 재사용할 수 있다. 이렇게 이름이 붙여진 포인트 컷을 네임드 포인트 컷이라한다.
- 네임드 포인트컷은 @Pointcut애노테이션으로 정의할 수 있는데, 이 애노테이션이 붙은 메소드 이름이 포인트 컷의 이름이 된다. 참고로 이때 메소드의 반환값은 void로 한다.

# 어드바이스 대상 객체와 인수 정보 가져오기

- JoinPoint 타입의 인수를 활용하면 어드바이스 대상 객체나 메소드를 호출할 때 전달된 인수의 정보를 가져올 수 있다.
- getTarget메소드를 이용하면 프락시가 입혀지기 전의 원본 대상 객체 정보를, getThis메소드를 이용하면 프락시를, getArgs메소드를 이용하면 인수 정보를 가져올 수 있다.
- 단 JoinPoint 인터페이스의 메소드는 반환값이 Object타입이기 때문에 실제로 사용하기 전에는 형변환해야한다. 이 과정에서 타입이 호환되지 않으면 ClassCastException이 발생 할 수 있다. 이럴 때는 포인트컷 지시자인 target이나 this, args 등을 활용해 대상 객체나 인수를 어드바이스 메소드에 파라미터로 바로 바인딩하면 된다. 이 방법을 사용하면 getTarget메소드나 getArgs메소드의 결과를 형변환할 필요가 없어지고 타입이 맞지 않는 경우는 자연스럽게 어드바이스에서 제외된다. 결국 타입 불일치로 인한 오동작을 미연에 방지할 수 있는데 이런 상황을 '타입 안전(type safe)하다'라고 말한다.

# 스프링 프로젝트에서 활용되는 AOP 기능

- 트랜잭션 관리
- Spring Security에서 제공하는 인가 기능
- 캐싱 – 캐싱을 활성화하고 @Cacheable애노테이션을 지정하면 메소드의 매개변수 등을 키로 사용해 메소드의 실행결과를 캐시로 관리할 수 있다.
- 비동기 처리 – 비동기 처리를 하고 싶은 메소드에 @Async애노테이션을 붙여주고, 반환값으로 CompletableFuture나 DeferredResult타입의 값을 반환하게 만들면 해당 메소드는 AOP방식으로 별도의 스레드에서 실행될 수 있다.
- 재처리 – 스프링 Retry라는 프로젝트를 활용해 재처리를 AOP로 구현할 수 있다.

# 데이터 바인딩과 형 변환

- 자바 객체의 프로퍼티에 외부에서 입력된 값을 설정하는 과정을 데이터 바인딩이라한다.
- 데이터 바인딩이나 프로퍼티 관점에서 다뤄지는 객체를 자바빈즈(JavaBeans)라고 좀 더 구체적으로 언급한다.
- 프로퍼티의 입력 값으로 사용되는 값은 웹 애플리케이션의 요청 파라미터, 프로퍼티 파일의 설정 값, XML기반 설정에서 빈을 정의할 때 지정한 프로퍼티 값과 같은 다양한 형태가 있다. 그 중에서 가장 대표적인 것은 웹 애플리케이션에서 사용하는 요청 파라미터이다.

# String 값에 대한 데이터 바인딩

- 데이터를 바인딩하는 기능은 DataBinder클래스가 제공하는데, HTTP를 사용하는 환경에서는 서블릿 API에 맞게 커스터마이징 된 ServletRequestDataBinder 클래스를 사용하면 된다.

```
MemberForm form = new MemberForm();
ServletRequestDataBinder dataBinder = new
    ServletRequestDataBinder(form);
dataBinder.bind(request);
```

- 데이터 바인딩 기능을 활용하면 자바빈즈의 프로퍼티 개수가 100개가 넘어도 단 3줄로 표현할 수 있다. 심지어 스프링 MVC의 데이터 바인딩 기능을 활용하면 3줄마저도 필요없어진다.

# 스프링의 형 변환

- 데이터 바인딩을 할 때는 자바빈즈의 프로퍼티를 타입에 맞게 입력된 문자열 값을 형 변환해야 한다. 형 변환을 위해 다음과 같은 3가지 방식을 제공한다.
  - 빈을 조작하거나 래핑
    - 스프링 프레임워크 초창기부터 제공된 기법으로 빈의 프로퍼티를 읽거나 쓰기 위해 원래 있던 빈을 조작하고 래핑하는 방법을 제공한다. PropertyEditor인터페이스의 구현클래스를 활용해 형변환을 한다.
  - 형 변환
    - 스프링 프레임워크 3.0부터 지원하는 기법으로 Converter인터페이스등의 구현 클래스를 활용해 형변환을 한다. PropertyEditor는 변환하기 전의 값이 String으로 제한되는데 반해 Converter는 String외의 타입에 대해서도 형 변환할 수 있다.
  - 필드 값의 형식 포맷팅
    - 스프링 프레임워크 3.0부터 지원되는 기법으로 Formatter인터페이스등의 구현 클래스를 활용해 형 변환한다. Formatter는 String과 임의의 클래스를 상호 변환하기 위한 인터페이스로서, 형 변환할 때 로캘(Local)을 구려한 국제화 기능도 제공한다. 주로 숫자나 날짜와 같이 로캘에 따라 다른 포맷 형태를 가진 클래스와 형 변환이 필요할 때 사용한다.

# 프로퍼티 관리

- 애플리케이션에서 사용하는 각종 설정 값을 읽어오기 위해서는 프로퍼티 관리가 필요하다. 스프링 프레임워크는 프로퍼티를 관리할 수 있는 효과적으로 관리할 수 있는 관리 메커니즘을 제공한다.
- @Value 애노테이션을 활용하면 별도로 분리된 프로퍼티 값을 소스 코드에 주입해서 사용할 수 있다.  
`@Value("SpEL표기법")`
- 자바 기반 설정 방식을 사용한다면 @PropertySource 애노테이션으로 위치를 지정할 수 있다.
- 스프링 프레임워크는 자바 프로퍼티 파일뿐만 아니라 JVM시스템 프로퍼티, 환경 변수 등을 같은 방식으로 사용할 수 있다.
- 만약 같은 이름의 프로퍼티가 JVM 시스템 프로퍼티, 프로퍼티 파일, 환경 변수에 있다면 우선 순위는 JVM 시스템 프로퍼티, 환경변수, 프로퍼티 파일 순서이다.

# SpEL (스프링 표현언어 : Spring Expression Language)

- SpEL은 스프링 프레임워크가 제공하는 표현언어다.
- SpEL은 스프링 프레임워크 뿐만 아니라 다양한 스프링 프로젝트에서도 사용되고 있다. Spring Security, Spring Data JPA, Spring Boot 등
- SpEL을 사용하려면 pom.xml 파일에 의존 라이브러리를 추가해야 한다. org.springframework:spring-expression

# SpEL에서 쓸 수 있는 표현식 유형 1/2

- 리터럴 값
  - SpEL은 문자열, 수치 값(지수 표기, Hex표기, 소수점, 음의 부호 등), 부울 값, 날짜나 시간과 같은 리터럴 표현을 지원한다. 문자열의 경우 작은 따옴표를 사용해 표현한다.
- 객체 생성
  - SpEL은 List나 Map을 생성하는 표현이나 new 연산자를 사용해 배열이나 임의의 객체를 생성하는 표현도 지원한다. 참고로 이때 사용되는 타입은 프리미티브 타입을 제외하면 FQCN을 사용한다.
- 프로퍼티 참조
  - SpEL은 자바 빈지의 프로퍼티에 접근하기 위한 표현을 지원한다. 기본적으로 프로퍼티의 이름을 명시하면 되는데, 중첩된 객체의 프로퍼티, 컬렉션, 배열안에 포함된 요소 혹은 Map안에 담긴 요소 등에 대해서도 접근할 수 있다.
- 메소드 호출
  - SpEL은 자바 객체의 메소드를 호출하기 위한 표현을 지원한다.
- 타입
  - SpEL은 타입을 의미하는 표현인 'T(타입의 FQCN)'을 지원한다.

# SpEL에서 쓸 수 있는 표현식 유형 2/2

- 변수 참조
  - SpEL은 변수 개념이 있어서 변수에 접근하기 위한 표현인 '#변수명' 을 지원한다.
- 빈 참조
  - SpEL은 DI컨테이너에서 빈을 참조하기 위한 표현으로 '@빈이름' 을 지원한다.
- 연산자
  - SpEL은 표준 관계 연산자인 <, >, >=, <=, !=, !은 물론 인스턴스를 비교하는 instanceof, 정규표현식으로 일치 여부를 확인하는 matches를 지원한다.
- 템플릿
  - SpEL은 텍스트 본문 안에 표현식을 집어 넣어 텍스트에 표현식의 결과를 표시하게 만드는 템플릿(Template)기능을 지원한다. 표현식 부분은 '#{표현식}'과 같은 방법으로 기재하면 된다.
- 컬렉션
  - SpEL은 컬렉션 안에서 조건과 일치하는 요소를 추출하기 위한 표현(Collection Selection)이나 컬렉션 안에 있는 요소가 가진 특정 프로퍼티의 값을 추출하기 위한 표현(Collection Projection)을 지원한다.

# 리소스 추상화

- 애플리케이션을 개발할 때는 설정 파일과 같은 다양한 리소스를 필요로 한다. 이러한 리소스의 위치는 다양할 수 있다. 디렉토리에 있거나 war파일 혹은 jar파일 심지어 또 다른 웹서버등에 있을 수 있다. 이러한 리소스의 위치를 모두 알고 접근해야 하지만 스프링 프레임워크가 제공하는 리소스 추상화 기능을 활용하면 구체적인 위치 정보를 직접 다루지 않아도 리소스를 접근 할 수 있다.
- 스프링 프레임워크는 리소스를 추상화하기 위해 org.springframework.core.io.Resource 인터페이스와 쓰기 가능한 리소스라는 의미로 WritableResource 인터페이스를 제공한다.
- AWS지원 프로젝트에 관리되는 데이터에 대한 업로드와 다운로드 기능도 WritableResource 인터페이스 구현 클래스를 제공한다. 이를 통해 소스코드를 변경하지 않고 데이터의 저장 위치를 Amazon S3나 파일시스템으로 바꿔 쓸 수 있다.

# Resource 인터페이스의 중요 구현 클래스

- ClassPathResource
  - 클래스 패스 상에 있는 리소스를 다루기 위한 클래스
- FileSystemResource
  - java.io패키지의 클래스를 사용해 파일 시스템 상의 리소스를 다루기 위한 클래스이다. WritableResource도 구현하고 있다.
- PathResource
  - Java SE 7에 추가된 java.nio.file패키지의 클래스를 사용해 파일시스템상의 리소스를 다루기 위한 클래스 WritableResource도 구현하고 있다.
- UrlResource
  - URL 상의 웹 리소스를 다루기 위한 클래스 경로에 <http://>를 쓰고 HTTP프로토콜을 사용하는 것이 일반적이다. 단 경로에 file:// 를 쓰면 파일 시스템 상의 리소스도 다룰 수 있다.
- SErvletContextResource
  - 웹 애플리케이션 상의 리소스를 다루기 위한 클래스

# 메시지 관리

- 애플리케이션을 개발 하다보면 문자열 형태의 메시지를 자주 사용해야한다. 이러한 메시지를 소스코드에서 분리하는 가장 대표적인 예가 다국어를 지원하는 국제화 기능이다.
- 스프링 프레임워크는 외부에서 메시지 정보를 가져오는 기능을 제공하는데, 그 핵심이 MessageSource인터페이스다. MessageSource는 메시지 정보의 출처를 추상화하기 위한 것으로, 어딘가에 있을 메시지 정보를 가져오기 위해 getMessage메소드를 제공한다.

# MessageSource의 중요 구현 클래스

- ResourceBundleMessageSource
  - Java SE 표준의 `java.util.ResourceBundle`을 사용 프로퍼티 파일에서 메시지를 가져온다.
- ReloadableResourceBundleMessageSource
  - 스프링이 제공하는 `org.springframework.core.io.Resource`를 사용한다. 프로퍼티 파일에서 가져온다. `java.util.ResourceBundle` 기능을 확장한다.

# MyBatis2 와 스프링

# MyBatis2.x와 스프링의 연계

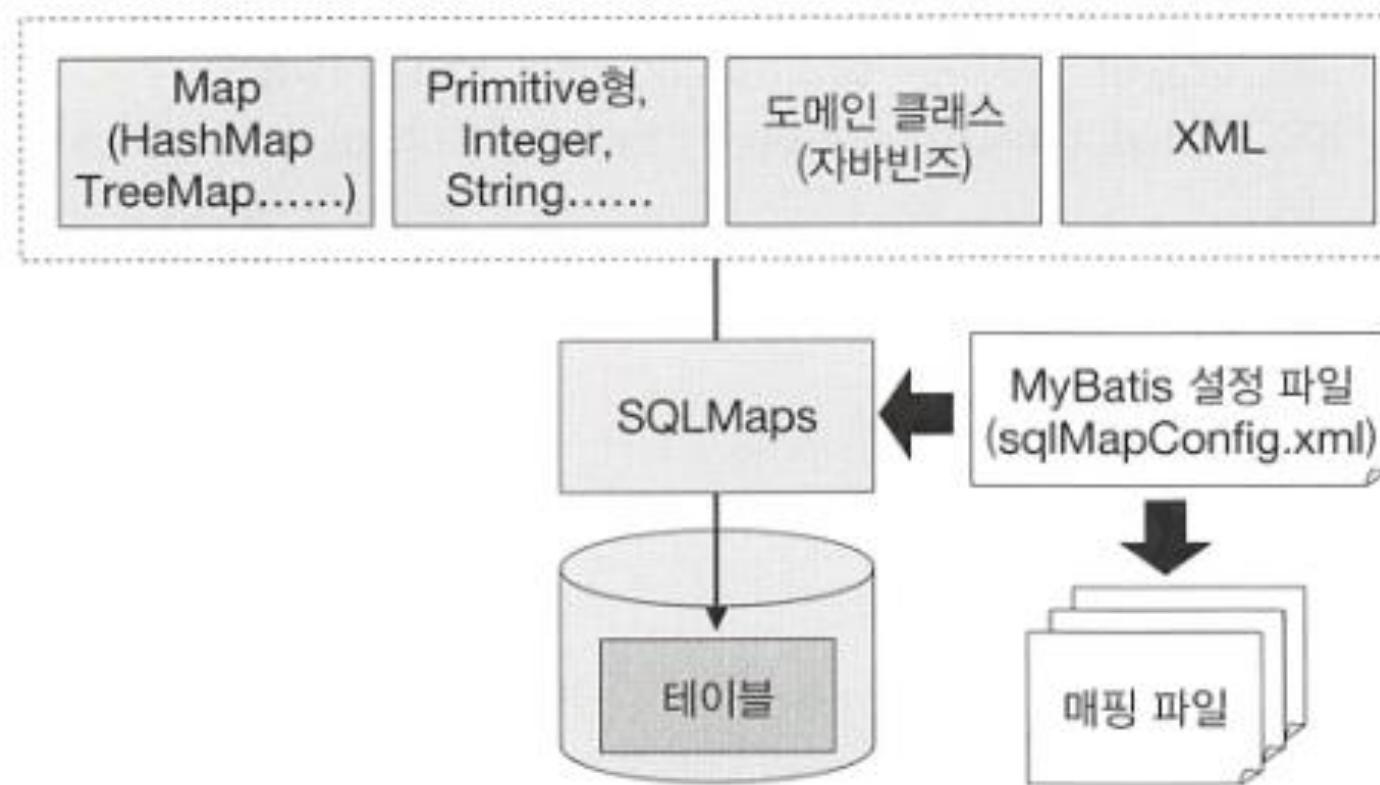
- SQL문을 직접 기술하여 DB프로그래밍을 하고 싶다?
  - MyBatis
  - Spring JDBC
- MyBatis 2.x란 데이터에 엑세스하기 위한 프레임워크다.
- OR매핑을 가능하게 하면서 SQL은 개발자가 직접 기술하는 JDBC와 JPA중간 정도의 위치라고 보면 된다.
- iBatis라는 이름의 제품이었지만 3.x발표와 함께 MyBatis로 이름이 변경되었다.
- 아파치 라이센스 2.0을 따른다.

# MyBatis 2.x의 구조

- MyBatis 2.x 는 SQLMaps와 DaoFramework라는 두 개의 프레임워크로 구성된다.
- MyBatis 2.x 는 XML파일로 정의된 SQL문과 OR매핑의 정의를 이용해 데이터베이스를 엑세스하는 기능을 제공하는 프레임워크이다.



# MyBatis 2.x 프레임워크 구조



# MyBatis 2.x를 이용해 데이터에 엑세스하는 기본 흐름

1. 사용할 데이터베이스의 접속 주소와 사용자 이름 등의 설정. SQL문과 OR매핑을 정의한 XML파일의 위치를 지정한 XML파일을 준비한다.(주로 sqlMapConfig.xml이라고 이름 붙인다.)
2. 애플리케이션에서 이용할 SQL문과 그 SQL문의 실행 결과를 JavaBeans에 매핑하는 OR매핑 정의를 XML파일로 준비한다.

# sqlMapConfig.xml

- DataSource 설정
- SqlMap파일 설정

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig
PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
    <transactionManager type="JDBC" >
        <dataSource type="SIMPLE">
            <property name="JDBC.Driver" value="org.hsqldb.jdbc.JDBCDriver"/>
            <property name="JDBC.ConnectionURL" value="jdbc:hsqldb:hsq://localhost/
sample"/>
            <property name="JDBC.Username" value="sa"/>
            <property name="JDBC.Password" value="" />
        </dataSource>
    </transactionManager>
    <sqlMap resource="sample/Friendly.xml" /> ①
</sqlMapConfig>
```

# Friendly.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
 "http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap namespace="Friendly" >
  <resultMap id="ownerResult" class="sample.biz.domain.Owner">
    <result property="ownerId" column="OWNER_ID" />
    <result property="ownerName" column="OWNER_NAME"/>
  </resultMap>

  <!-- Owner SQL -->
  <select id="findOwner" parameterClass="java.lang.String" resultMap=
    "ownerResult">
    select OWNER_ID, OWNER_NAME from OWNER
      where OWNER.OWNER_ID = #value#
  </select>
</sqlMap>
```

# MyBatis2.x의 매팅 파일구조 1/2

태그	속성	설명
resultMap	id	OR 매팅 이름
	class	릴레이셔널 데이터를 매팅하는 JavaBeans의 클래스 이름
statement	id	SQL 이름(INSERT, UPDATE, DELETE, SELECT 문의 이용 가능)
	parameterClass	SQL 문에서 사용할 바인드 변수를 설정하는 클래스 이름. 덧붙여 java.lang.String이나 수치형으로 지정된 값을 #value#에 바인드할 수 있다.

# MyBatis2.x의 매팅 파일구조 2/2

태그	속성	설명
	parameterMap	SQL 문에서 사용할 바인드 변수를 설정하는 맵 이름
	resultClass	SELECT 문으로 얻은 결과가 매팅되는 클래스 이름
	resultMap	SELECT 문으로 얻은 결과가 매팅되는 맵 이름
insert (update, delete)	id	SQL 이름
	parameterClass	statement 태그의 같은 이름의 속성과 같다.
	parameterMap	SQL 문에서 사용할 바인드 변수를 설정하는 맵 이름
select	id	SELECT 이름
	parameterClass	statement 태그의 같은 이름의 속성과 같다.
	parameterMap	SELECT 문에서 사용할 바인드 변수를 설정하는 맵
	resultClass	statement 태그의 같은 이름의 속성과 같다.
	resultMap	statement 태그의 같은 이름의 속성과 같다.

# 실행소스

```
public static void main(String[] args) {
    try {
        Reader reader = Resources.getResourceAsReader("sample/sqlMapConfig.xml"); ①
        SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
        reader.close();

        Owner owner = (Owner)sqlMap.queryForObject("findOwner","1"); ②
        System.out.println("OwnerId = " + owner.getOwnerId() + " :OwnerName = " + owner.
            getOwnerName());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

# 명명된 바인드 변수

```
select OWNER_ID, OWNER_NAME from OWNER where OWNER.OWNER_ID = #ownerId#
```

```
<select id="findOwner" parameterClass="java.util.Map" resultMap="ownerResult">
    select OWNER_ID, OWNER_NAME from OWNER
    where OWNER.OWNER_ID = #ownerId#
</select>
```

```
Map param = new HashMap();
param.put("ownerId","1");
Owner owner = (Owner)sqlMap.queryForObject("findOwner",param);
```

# 명명된 바인드 변수

```
<select id="findOwner" parameterClass="sample.biz.domain.Owner"  
    resultMap="ownerResult">  
    select OWNER_ID, OWNER_NAME from OWNER  
    where OWNER.OWNER_ID = #ownerId#  
</select>
```

```
Owner param = new Owner();  
param.setOwnerId("1");  
Owner owner = (Owner)sqlMap.queryForObject("findOwner",param);
```

# 명명된 바인드 변수

```
<select id="findOwnerByNames" parameterClass="java.util.Map" resultMap="ownerResult">
    select O.OWNER_ID, O.OWNER_NAME from
        OWNER O inner join PET P
        ON O.OWNER_NAME = P.OWNER_NAME
    where O.OWNER_NAME = #owner.ownerName# and P.PET_NAME = #pet.petName#
</select>
```

```
Map<String, Object> map = new HashMap<String, Object>(); Owner owner = new Owner();
owner.setOwnerName("홍길동");
map.put("owner", owner);
Pet pet = new Pet();
pet.setPetName("바둑이");
map.put("pet", pet);
Owner result = (Owner)sqlMap.queryForObject("findOwnerByNames", map);
```

# 명명된 바인드 변수

```
<select id="findOwner" parameterClass="java.lang.String" resultMap="ownerResult">
    select OWNER_ID, OWNER_NAME from OWNER
    where OWNER.OWNER_ID = #value#
</select>
```

```
String ownerId = "1";
Owner owner = (Owner)sqlMap.queryForObject("findOwner",ownerId);
```

# 동적 SQL

- MySQL 2.x는 WHERE구문을 동적으로 조립할 수 있다. 이 기능에 의해 WHERE구문의 지정 조건이 달라도 그만큼 SQL문을 준비할 필요가 없어졌다.
- 이 편리한 동적 SQL문 조립 기능을 사용하려면 dynamic태그로 에워싼 XML태그로 지정한다. dynamic태그의 prepend속성은 동적으로 조립할(추가할)SQL문 앞에 부가하는 문자열이다.

# 동적 SQL

```
<select id="findOwner" parameterClass="sample.biz.domain.Owner" resultMap="ownerResult">
    select OWNER_ID, OWNER_NAME from OWNER
    <dynamic prepend="where">
        <isNotNull prepend="and" property="ownerId">
            OWNER.OWNER_ID = #ownerId#
        </isNotNull>
        <isNotNull prepend="and" property="ownerName">
            OWNER.OWNER_NAME = #ownerName#
        </isNotNull>
    </dynamic>
</select>

Owner owner = (Owner)sqlMap.queryForObject("findOwner",null);

Owner param = new Owner();
param.setOwnerName("홍길동");
Owner owner = (Owner)sqlMap.queryForObject("findOwner",param);
```

# 동적 SQL - 공통 속성

속성	설명
prepend	지정한 값을 필요에 따라 SQL 문 앞에 부가한다. [리스트 12–4]의 경우에는 OWNER.OWNER_ID = #ownerId#가 AND OWNER OWNER_ID = #ownerId#가 된다.
property	비교할 프로퍼티 이름
compareProperty	비교되는 다른 쪽 프로퍼티 이름. 파라미터 값과 property 값을 비교하는 것이 아니라 프로퍼티로 지정된 값을 다른 프로퍼티와 비교하고 싶을 때 등에 이용한다.
compareValue	비교되는 값. property="name" compareValue="foo"는 name="foo"와 같은 의미가 된다.

# 동적 SQL

isEqual, (isnotEqual)

property로 지정한 프로퍼티 값과 compareValue의 값이 같을 때 태그로 에워싼 SQL 문을 유효하게 한다. 반대로 같지 않을 때를 검사하고 싶다면 isNotEqual을 이용한다.

```
<isEqual property="firstName" compareValue="Clinton" >  
    where ACC_FIRST_NAME = 'Clinton'  
</isEqual>
```

isGreaterThan,  
(isGreaterEqaul,  
isLessThan,  
isLessEqual)

property로 지정한 프로퍼티 값이 compareValue 값보다 클 때 유효하게 된다. 같을 때도 유효하게 하려면 isGreaterEqual을 이용한다. 작을 때 유효하게 하려면 isLessThan, isLessEqual을 이용한다.

```
<isGreaterThan property="age" compareValue="20" >  
    where ACC_ID = 1  
</isGreaterThan>
```

isPropertyAvailable,  
(isNotPropertyAvailable)

파라미터로 지정된 JavaBeans에 property로 지정한 프로퍼티가 있을 때 유효하게 된다. 반대로 property로 지정한 프로퍼티가 없을 때 유효하게 하려면 isNotPropertyAvailable을 이용한다.

```
<isPropertyAvailable property="firstName" >  
    where ACC_FIRST_NAME = #firstName#  
</isPropertyAvailable>
```

# 동적 SQL

isNull, (isNotNull)	property로 지정한 프로퍼티 값이 null일 때 유효하게 된다. 반대로 null이 아닐 때 유효하게 하려면 isNotNull을 이용한다.
	<pre>&lt;isNotNull property="firstName" &gt;     where ACC_FIRST_NAME = #firstName# &lt;/ isNotNull &gt;</pre>
isEmpty, (isNotEmpty)	property로 지정한 프로퍼티의 값이 null 혹은 Bean 문자("")인 경우, 또는 프로퍼티의 값이 컬렉션일 때, null 혹은 크기가 0일 때 유효하게 된다. 반대로 Bean 문자가 아니거나 컬렉션의 크기가 0이 아닐 때 유효하게 하려면 isNotEmpty를 이용한다.
Iterator	파라미터로 지정된 java.util.List의 각 요소를 SQL 문에 부가할 수 있다.
	<pre>select     ACC_ID as id</pre>

# 동적 SQL

```
from ACCOUNT
<dynamic prepend="WHERE" >
    <iterate open="ACC_FIRST_NAME IN (" close=")"
        conjunction="," >
        #value[]#
    </iterate>
</dynamic>
```

---

이 예에서 작성되는 SQL은 아래처럼 된다(…는 java.util.List의 요소 수만큼 계  
속된다).

---

```
select ACC_ID as id from ACCOUNT WHERE ACC_FIRST_NAME
IN(?, ?,.....)
```

# 동적 SQL

동적으로 SQL 문을 추가할 수 있다. 방법은 우선 파라미터에 java.util.Map을 지정한다. 그 Map의 키를 \$로 에워싼 문자열로 지정한다. 이 키와 일치하는 값에 추가할 SQL 문을 저장한다.

- 자바 프로그램

```
Map param = new HashMap();
param.put("statement","ACC_ID,ACC_FIRST_NAME");
List list = sqlMap.queryForList("test2",param);
```

- SQL의 정의

\$임의의 문자열\$

```
<select id="test2" parameterClass="java.util.Map"
      resultClass="testdomain.Account">
    select
      ACC_ID as id,
      ACC_FIRST_NAME as firstName
    from ACCOUNT
    order by $statement$
</select>
```

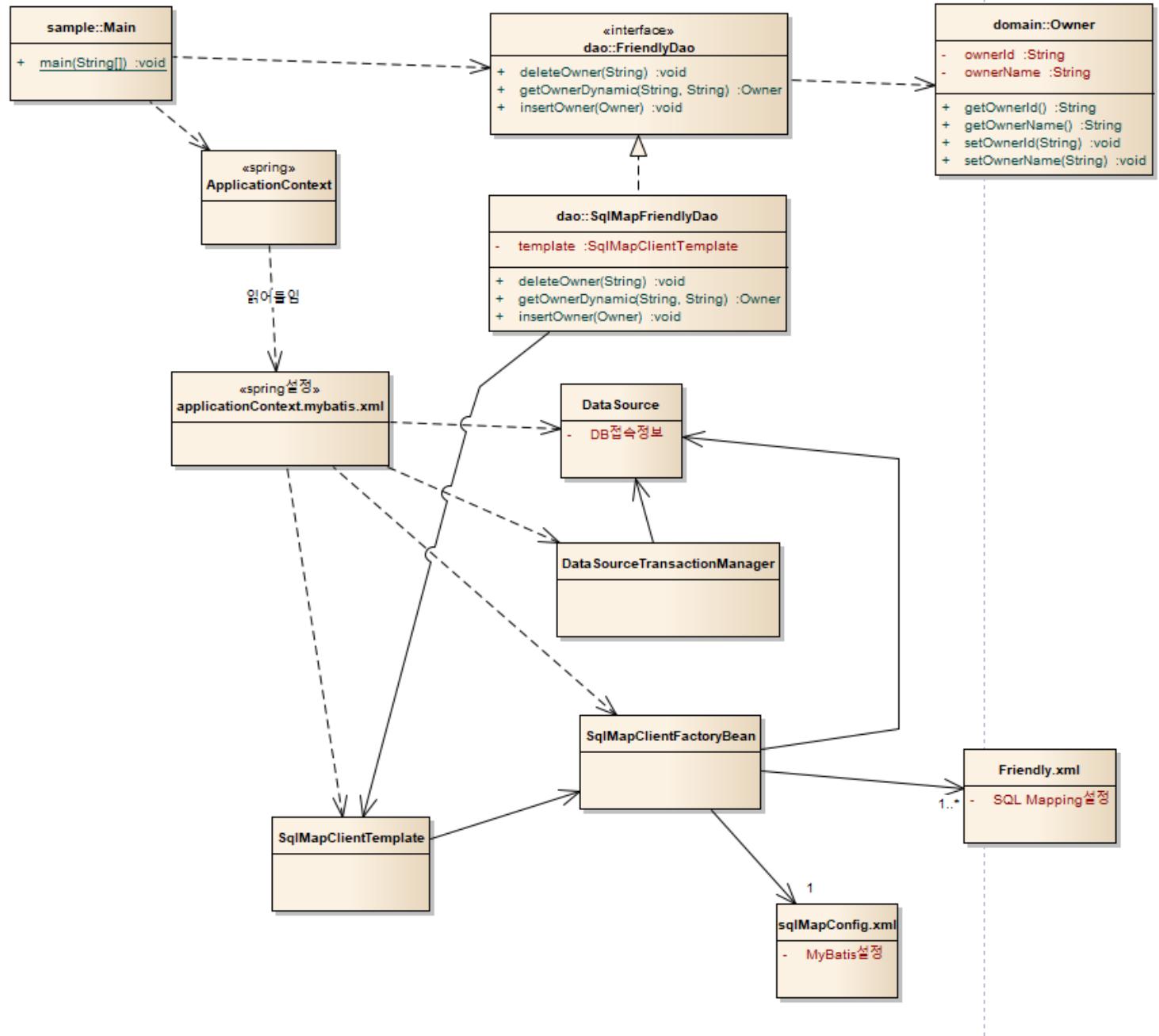
이 예에서 생성된 SQL은 아래와 같다.

```
select ACC_ID as id , ACC_FIRST_NAME as firstName from
ACCOUNT order by ACC_ID, ACC_FIRST_NAME
```

# 프라이머리 키 생성

```
<!--Oracle SEQUENCE Example -->
<insert id="insertProduct-ORACLE" parameterClass="sample.biz.domain.Product"> ...
    <selectKey resultClass="int" keyProperty="id">
        SELECT STOCKIDSEQUENCE.NEXTVAL AS ID FROM DUAL
    </selectKey>
    insert into PRODUCT (PRD_ID,PRD_DESCRIPTION)
        values (#id#, #description#)
</insert> ...
```

# MyBatis2.x와 스프링의 연계



# MyBatis3 와 스프링

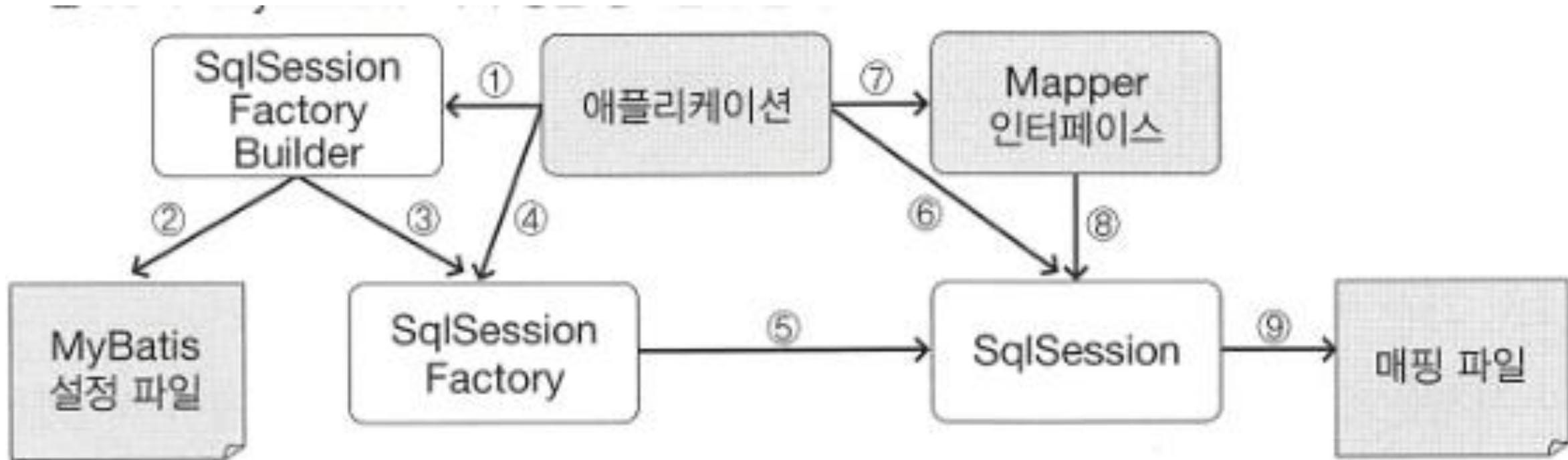
# MyBatis3.x란 무엇인가?

- MyBatis3.x는 앞의 MyBatis2.x의 후속으로 등장하였다.
- 2010년 5월에 버전 3.0이 공개되었다.
- MyBatis 3.x는 MyBatis2.x의 특징어있던 XML에 의한 SQL과 OR 매핑의 유연한 작성법을 답습하면서 새로운 기능을 추가했다.
- MyBatis 2.x와 호환성이 없다.
- MyBatis 3.x의 새로운 특징은 Mapper라는 개념이 제공된다. Mapper를 이용함으로써 MyBatis2.x일때보다 애플리케이션의 소스코드를 타입에 안전하게 기술할 수 있게 되었다.

# MyBatis 3.의 주요 구성물

구성물	역할
MyBatis 설정 파일	데이터베이스의 접속 주소 정보나 매팅 파일의 경로 등의 고정된 환경 정보를 설정한다. XML 파일로 기술한다.
SqlSessionFactoryBuilder	MyBatis 설정 파일을 바탕으로 SqlSessionFactory를 생성한다. 애플리케이션을 시작할 때 사용해 SqlSessionFactory를 생성하면 SqlSessionFactoryBuilder 오브젝트는 필요 없어지므로 참조를 유지할 필요는 없다. 애플리케이션의 시작 시에 쓰고 버리는 이미지다.
SqlSessionFactory	SqlSession을 생성한다. SqlSessionFactory 오브젝트는 스레드 세이프하며, 애플리케이션 안의 프로그램은 하나의 오브젝트를 싱글톤 패턴 등으로 공유해야만 한다. 스프링과 연계할 때는 DI 컨테이너에 관리시킨다.
SqlSession	SQL 발행이나 트랜잭션 관리를 실행한다. SqlSession 오브젝트는 스레드 세이프하지 않으므로 스레드마다 필요에 따라 생성하고 폐기한다.
Mapper 인터페이스	매팅 파일에 기재된 SQL을 호출하기 위한 인터페이스. 애노테이션으로 SQL이 나 OR 매팅을 기술할 수도 있다. 오브젝트는 MyBatis3.x가 자동으로 생성한다. Mapper 오브젝트는 SqlSession 오브젝트와 관련해 생성되므로 SqlSession 오브젝트와 함께 생성하고 폐기한다.
매팅 파일	SQL과 OR 매팅을 설정한다. XML 파일로 기술한다.

# MyBatis 3.x의 구성물 상호간의 관계



\* 음영은 애플리케이션 개발자가 작성할 것

# MyBatis 설정 파일 예제 sample/mybatis.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC" />
            <dataSource type="POOLED">
                <property name="driver" value=" org.hsqldb.jdbc.JDBCDataSource" />
                <property name="url"
                    value=" jdbc:hsqldb:hsq://localhost/sample" />
                <property name="username" value="sa" />
                <property name="password" value=" " />
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="sample/dao/PetMapper.xml" />
    </mappers>
</configuration>
```

①

②

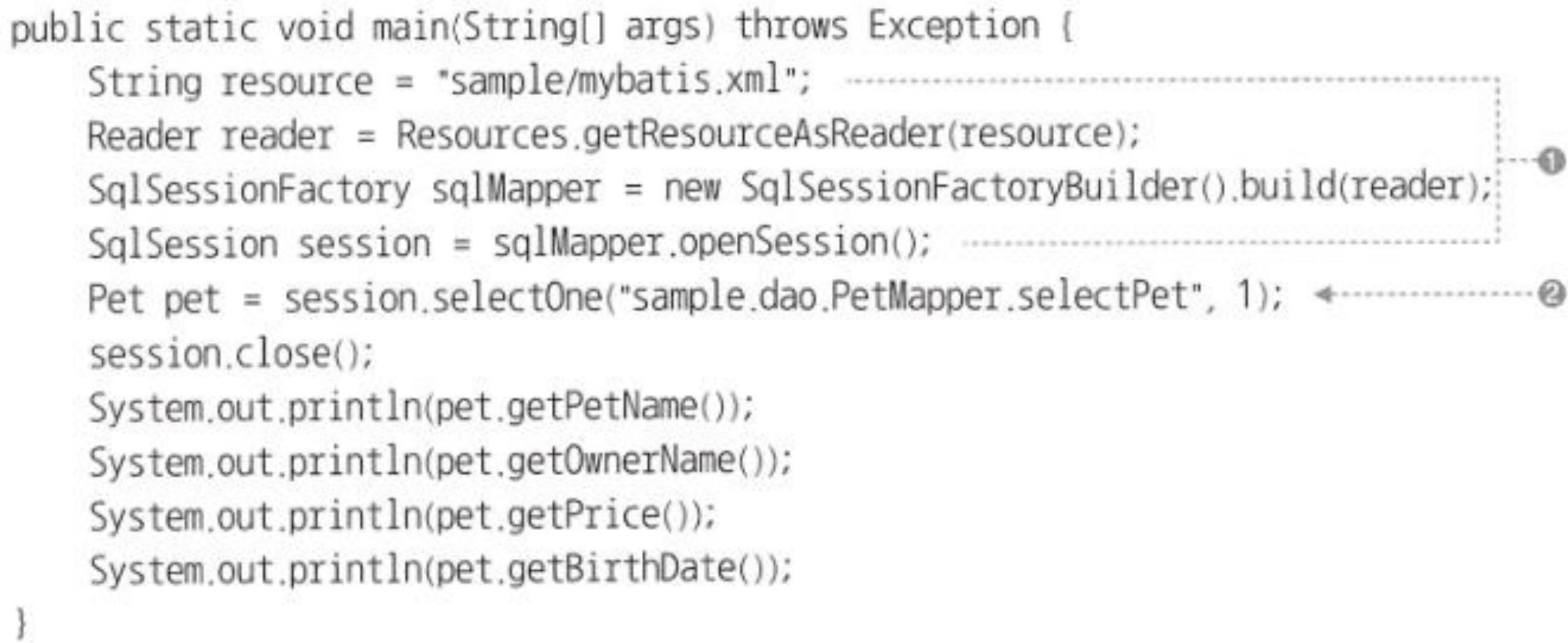
# 매핑 파일 예제(sample/dao/PetMapper.xml)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd"> ①
<mapper namespace="sample.dao.PetMapper">

  <select id="selectPet" parameterType="int" resultMap="petResultMap"> ②
    SELECT * FROM PET WHERE PET_ID = #{petId}
  </select>
  <resultMap id="petResultMap" type="sample.biz.domain.Pet"> ③
    <id property="petId" column="PET_ID" />
    <result property="petName" column="PET_NAME" />
    <result property="ownerName" column="OWNER_NAME" />
    <result property="price" column="PRICE" />
    <result property="birthDate" column="BIRTH_DATE" />
  </resultMap>
</mapper>
```

# SQL을 호출하는 프로그램 예제

```
public static void main(String[] args) throws Exception {  
    String resource = "sample/mybatis.xml"; .....  
    Reader reader = Resources.getResourceAsReader(resource);  
    SqlSessionFactory sqlMapper = new SqlSessionFactoryBuilder().build(reader); .....  
    SqlSession session = sqlMapper.openSession(); .....  
    Pet pet = session.selectOne("sample.dao.PetMapper.selectPet", 1); .....  
    session.close(); .....  
    System.out.println(pet.getPetName()); .....  
    System.out.println(pet.getOwnerName()); .....  
    System.out.println(pet.getPrice()); .....  
    System.out.println(pet.getBirthDate()); .....  
}
```



# MyBatis2.x와 MyBatis3.x의 구성물 대응

	MyBatis2.x	MyBatis3. x
설정 파일	SQL Map 설정 파일(sqlMapConfig.xml) 매핑 파일(Friendly.xml)	MyBatis 설정 파일(mybatis.xml) 매핑 파일(PetMapper.xml)
클래스	SqlMapClientBuilder SqlMapClient	SqlSessionFactoryBuilder SqlSession

# MyBatis 설정 파일

- MyBatis설정파일은 SQL의 발행과 OR매핑의 전제가 되는 정보를 설정하는 파일이다.

# 매핑 파일 - 네임스페이스

- 매핑 파일의 루트요소 mapper태그의 속성으로서 지정하는 네임스페이스는 매핑파일에 기술된 SQL을 그룹으로 묶는 상자 같은 것이다.



# 매핑 파일 - SQL의 기술

- SQL을 기술할 때는 select 태그, insert 태그, update태그, delete 태그를 사용한다.

```
<select id="selectPet" parameterType="int" resultMap="petResultMap">
    SELECT * FROM PET WHERE PET_ID = #{petId}
</select>
```

# 매핑 파일 - SQL의 파라미터 바인딩

- 파라미터가 하나인 바인딩. 파라미터가 하나일 때는 아무거나  
지정해도 상관없다. MyBatis2.x에서는 #value#를 사용

---

```
<select id="selectPet" parameterType="int" resultMap="petResultMap">
    SELECT * FROM PET WHERE PET_ID = #{petId}
</select>
```

# 매핑 파일 - SQL의 파라미터 바인딩

- DTO를 활용한 파라미터 바인딩
- Pet의 getPetId(), getPetName(), getOwnerName(), getPrice(),  
getBirthDate() 메소드의 결과로 바인딩된다.

```
<insert id="insertPet" parameterType="Pet">
    INSERT INTO PET (PET_ID, PET_NAME, OWNER_NAME, PRICE, BIRTH_DATE)
    VALUES (#{petId}, #{petName}, #{ownerName}, #{price}, #{birthDate})
</insert>
```

# 매핑 파일 - SQL호출 (DTO사용)

```
Pet pet = new Pet();
pet.setPetId(10);
pet.setPetName("나비");
pet.setOwnerName("홍길동");
pet.setPrice(10000);
pet.setBirthDate(new Date());
session.insert("sample.dao.PetMapper.insertPet", pet);
```

# 매핑 파일 - SQL의 파라미터 바인딩(Map)

```
<insert id="insertPet" parameterType="hashmap">
    INSERT INTO PET (PET_ID, PET_NAME, OWNER_NAME, PRICE, BIRTH_DATE)
    VALUES (#{petId}, #{petName}, #{ownerName}, #{price}, #{birthDate})
</insert>
```

```
Map<String, Object> map = new HashMap<String, Object>();
map.put("petId", 10);
map.put("petName", "나비");
map.put("ownerName", "홍길동");
map.put("price", 10000);
map.put("birthDate", new Date());
session.insert("sample.dao.PetMapper.insertPet", pet);
```

# 매핑 파일 - 복수의 JavaBeans의 바인딩

```
<select id="selectPetByNames" parameterType="hashmap" resultMap="petResultMap">
    SELECT * FROM
        PET P INNER JOIN OWNER O
        ON P.OWNER_NAME = O.OWNER_NAME
    WHERE
        P.PET_NAME = #{pet.petName} <----- ①
        AND O.OWNER_NAME = #{owner.ownerName} <----- ②
</select>

Map<String, Object> map = new HashMap<String, Object>();
Pet pet = new Pet();
pet.setPetName("바둑이");
map.put("pet", pet);
Owner owner = new Owner();
owner.setOwnerName("홍길동");
map.put("owner", owner);
Pet result = session.selectOne("sample.dao.PetMapper.selectPetByNames", map);
```

# 매핑 파일 - select문의 결과 매핑

```
<select id="selectPet" parameterType="int" resultType="Pet">
    SELECT
        PET_ID as petId, PET_NAME as petName,
        OWNER_NAME as ownerName, price, BIRTH_DATE as birthDate
    FROM PET WHERE PET_ID = #{petId}
</select>
```

```
Pet pet = (Pet) session.selectOne("sample.dao.PetMapper.selectPet", id);
```

```
List<Pet> list = session.selectList("sample.dao.PetMapper.selectPet", id);
```

# 매핑 파일 - resultMap을 사용한 간단한 매핑

```
<select id="selectPet" parameterType="int" resultMap="petResultMap">
    SELECT * FROM PET WHERE PET_ID = #{petId}
</select>
<resultMap id="petResultMap" type="Pet">
    <id property="petId" column="PET_ID" />
    <result property="petName" column="PET_NAME" />
    <result property="ownerName" column="OWNER_NAME" />
    <result property="price" column="PRICE" />
    <result property="birthDate" column="BIRTH_DATE" />
</resultMap>
```

# 매핑 파일 - resultMap을 사용한 복잡한 매핑

```
<select id="selectOwner" parameterType="int" resultMap="ownerResultMap">
    SELECT * FROM OWNER o INNER JOIN PET p
    on o.OWNER_NAME=p.OWNER_NAME
    WHERE o.OWNER_NAME = #{ownerName}
</select>
<resultMap id="ownerResultMap" type="Owner">
    <id property="ownerName" column="OWNER_NAME" />
    <collection property="petList" ofType="Pet">
        <id property="petId" column="PET_ID" />
        <result property="petName" column="PET_NAME" />
        <result property="ownerName" column="OWNER_NAME" />
        <result property="price" column="PRICE" />
        <result property="birthDate" column="BIRTH_DATE" />
    </collection>
</resultMap>
```

## 매핑 파일 - resultMap태그의 자식 요소로 지정할 수 있는 태그 목록

태그	용도
constructor	JavaBeans의 생성자의 인수에 칼럼 값이나 매핑한 다른 JavaBeans의 오브젝트를 설정한다.
id	JavaBeans의 프로퍼티에 프라이머리 키의 칼럼 값을 설정한다.
result	JavaBeans의 프로퍼티에 칼럼 값을 설정한다.
association	JavaBeans의 오브젝트에 1:1로 연결하는 다른 JavaBeans의 매핑을 설정한다.
collection	JavaBeans의 오브젝트에 1:多로 연결하는 다른 JavaBeans의 매핑을 설정한다.
discriminator	칼럼 값에 의해 사용할 매핑 설정을 전환한다.

# 매핑 파일 - 동적 SQL

```
<select id="findPet" parameterType="Pet" resultMap="petResultMap">
    SELECT * FROM PET
    <where>
        <if test="petName != null ">
            PET_NAME = #{petName}
        </if>
        <if test="ownerName != null ">
            AND OWNER_NAME = #{ownerName}
        </if>
        <if test="price != null ">
            AND PRICE = #{price}
        </if>
    </where>
</select>
```

SELECT \* FROM PET WHERE PET\_NAME= #{petName} AND PRICE= #{price}

# 매핑 파일 - 동적인 SQL을 위한 태그

태그	용도
if	파라미터의 조건에 따라 문자열을 추가한다. 필수 속성인 test 속성에 조건식을 기술한다.
choose	자바의 Switch 문과 같은 요령으로 복수의 조건에서 하나를 선택해 문자열을 추가 한다.
	조건식과 조건에 적합한 때에 기술할 문자열을 설정한다. 필수 속성인 test 속성에 조건식을 기술한다. choose 태그 안에 여러 개 기술할 수 있다.
otherwise	when 태그 중 어느 조건에도 적합하지 않을 때 기술할 문자열을 설정한다.
trim	SQL 문의 문자열을 부분적으로 치환한다.
where	WHERE 구문을 동적으로 생성한다.
set	UPDATE 문의 SET 구문을 동적으로 생성한다.
foreach	반복해서 문자열을 추가한다.

# 매핑 파일 - INSERT시의 키 생성

- 데이터베이스의 자동 증가를 사용하는 예제

```
<insert id="insertPet" parameterType="Pet" useGeneratedKeys="true"
    keyProperty="petId">
    INSERT INTO PET (PET_NAME, OWNER_NAME, PRICE, BIRTH_DATE)
    VALUES (#{petName},#{ownerName},#{price},#{birthDate})
</insert>
Pet pet = new Pet();
pet.setPetName("나비");
pet.setOwnerName("홍길동");
pet.setPrice(10000);
pet.setBirthDate(new Date());
session.insert("sample.dao.PetMapper.insertPet", pet);
System.out.println(pet.getPetId());
```

# 매핑 파일 - INSERT시의 키 생성(시퀀스 사용)

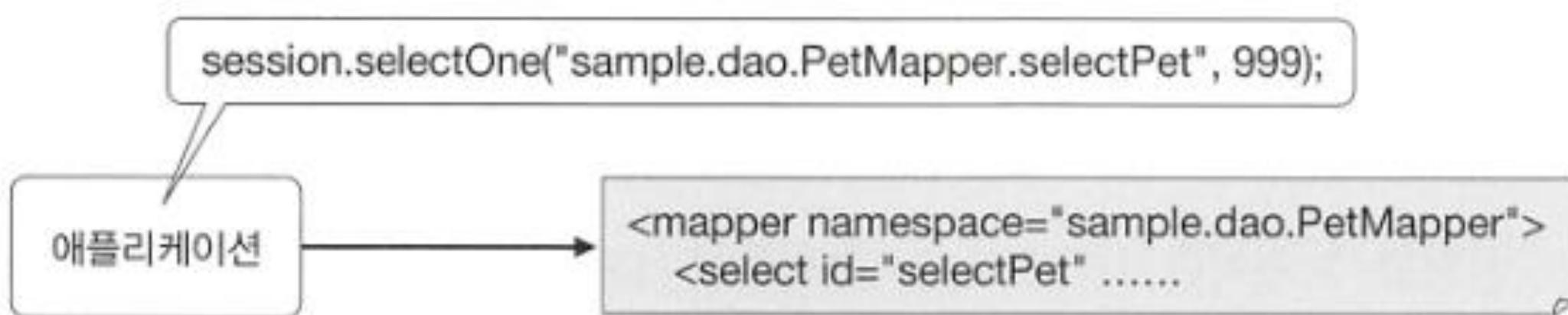
```
<insert id="insertPet" parameterType="sample.biz.domain.Pet">
    <selectKey keyProperty="petId" resultType="int" order="BEFORE">
        CALL NEXT VALUE FOR PET_ID_SEQ
    </selectKey>
    INSERT INTO PET (PET_ID, PET_NAME, OWNER_NAME, PRICE, BIRTH_DATE)
    VALUES (# {petId}, # {petName}, # {ownerName}, # {price}, # {birthDate})
</insert>
```

# Mapper인터페이스

- Mapper인터페이스는 SQL을 호출하는 프로그램을 타입 세이프하게 기술하고자 MyBatis 3.x부터 등장한 것이다.
- Mapper를 사용하지 않으면 SQL을 호출하는 프로그램은 SqlSession의 메소드의 인수에 문자열로 네임스페이스 + "." + SQL의 ID를 지정할 필요가 있었다.

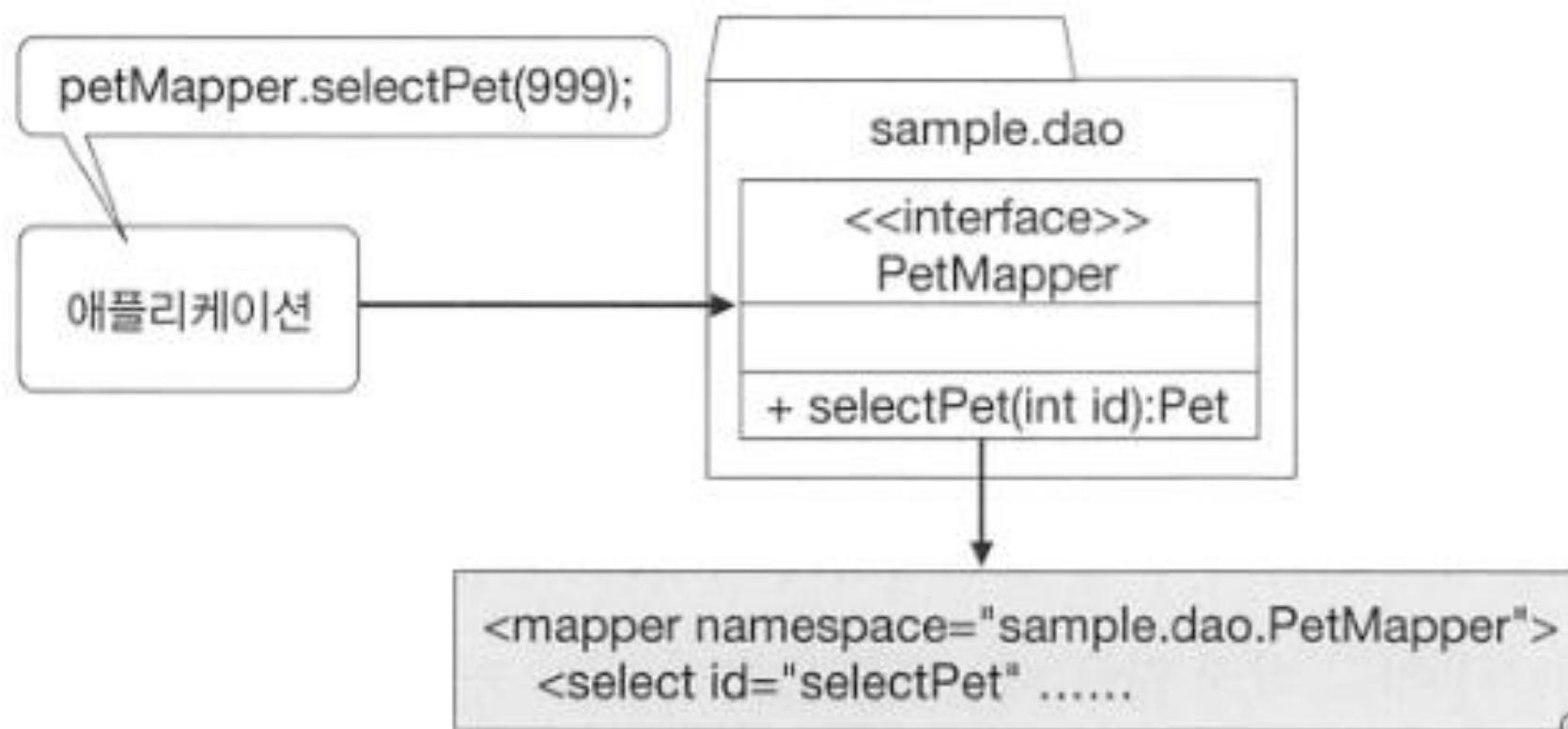
# Mapper인터페이스

- Mapper인터페이스를 사용하지 않을 때



# Mapper인터페이스

- Mapper인터페이스를 사용할 때



# Mapper 인터페이스

- PetMapper인터페이스

```
package sample.dao;  
public interface PetMapper {  
    Pet selectPet(int id);  
}
```

# Mapper 인터페이스

- Mapper인터페이스의 메소드 호출

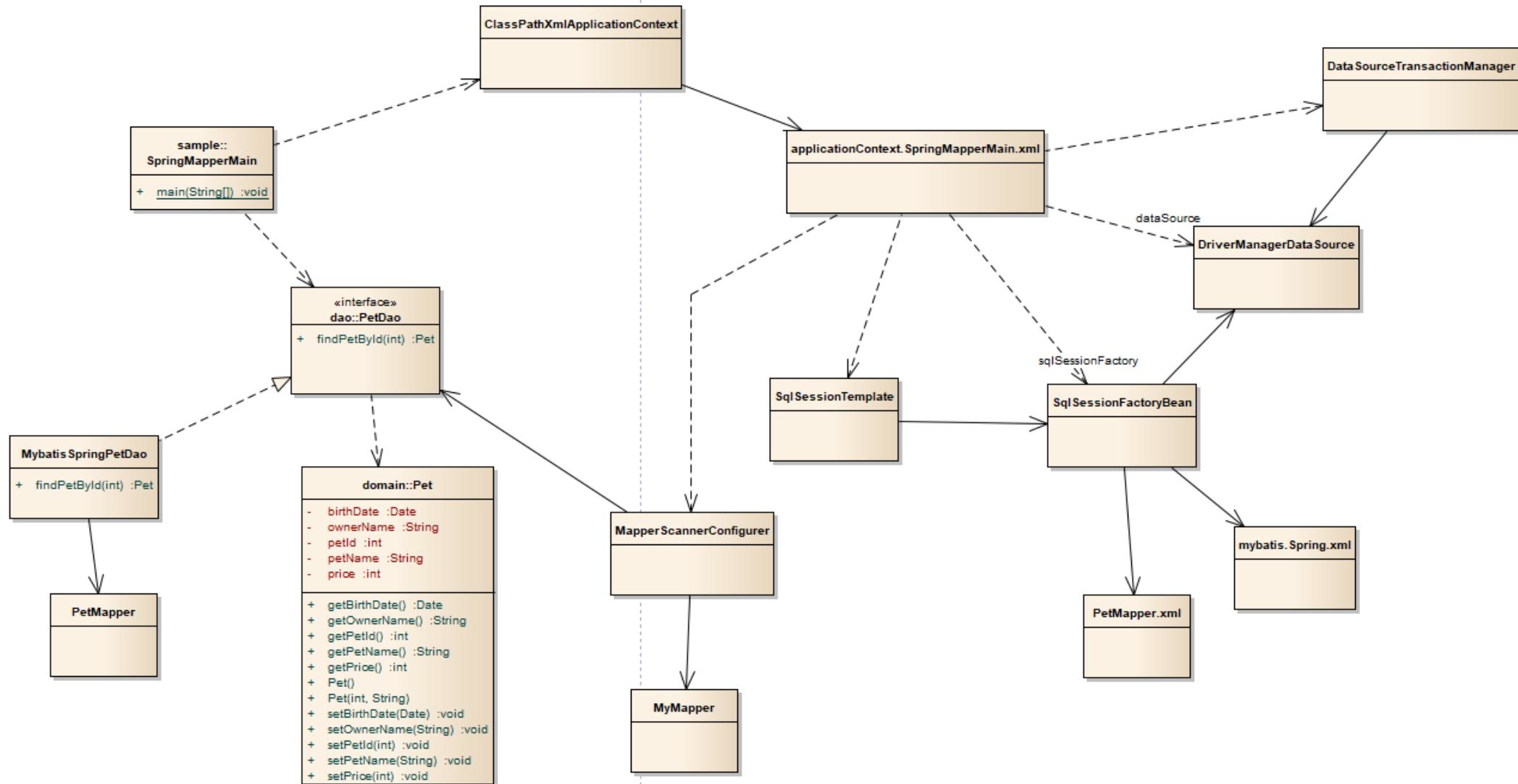
```
PetMapper mapper = session.getMapper(PetMapper.class);  
Pet pet = mapper.selectPet(1);
```

# 애노테이션

- Mapper인터페이스에는 그 밖에도 애노테이션을 이용할 수 있는 특징이 있다. 애노테이션을 이용하면 매핑 파일에서 기술했던 SQL이나 OR매핑을 애노테이션으로 기술할 수 있다.

```
@Select("SELECT * FROM PET WHERE PET_ID=#{petId}")  
Pet selectPet(int id);
```

# MyBatis3.x와 스프링의 연계



# 트랜잭션

# 트랜잭션 관리

- 스프링은 트랜잭션 관리를 위한 코드를 비지니스 로직에서 분리하기 위한 구조나 다른 트랜잭션을 투명하게 처리할 수 있는 API를 제공한다.
- 스프링 트랜잭션 처리의 중심이 되는 인터페이스는 PlatformTransactionManager다. 해당 인터페이스는 트랜잭션 처리에 필요한 API를 제공하며 개발자가 API를 호출하는 것으로 트랜잭션 조작을 수행할 수 있다.

# PlatformTransactionManager의 구현 클래스

- DataSourceTransactionManager
  - JDBC 및 마이바티스 등의 JDBC 기반 라이브러리로 데이터베이스에 접근하는 경우에 이용한다.
- HibernateTransactionManager
  - 하이버네이트를 이용해 데이터베이스에 접근하는 경우에 이용한다.
- JpaTransactionManager
  - JPA로 데이터베이스에 접근하는 경우에 이용한다.
- JtaTransactionManager
  - JTA에서 트랜잭션을 관리하는 경우에 이용한다.
- WebLogicJtaTransactionManager
  - 애플리케이션 서버인 웹로직(Weblogic)의 JTA에서 트랜잭션을 관리하는 경우에 이용한다.
- WebSpereUowTransactionManager
  - 애플리케이션 서버인 웹 스피어(WebSpere)의 JTA에서 트랜잭션을 관리하는 경우에 이용한다.

# 트랜잭션 관리자 정의

- 스프링 프레임워크의 트랜잭션 관리자를 사용할 때는 다음의 두가지 작업을 해야 한다.
  - PlatformTransactionManager의 빈을 정의한다.
  - 트랜잭션을 관리해야 하는 메소드를 정의한다.

# 선언적 트랜잭션

- 선언적 트랜잭션은 미리 선언된 룰에 따라 트랜잭션을 제어하는 방법이다.
- 선언적 트랜잭션의 장점은 정해진 룰을 준수함으로써 트랜잭션의 시작과 커밋, 롤백 등의 일반적인 처리를 비즈니스 로직 안에 기술할 필요가 없다.
- 스프링 프레임워크는 선언적 트랜잭션을 이용하는 방법으로 다음의 2가지 방법을 제공한다.
  - Transactional을 이용한 선언적 트랜잭션
  - XML설정을 이용한 선언적 트랜잭션

# @Transactional 애노테이션 속성 1/2

- value
  - 여러 트랜잭션 관리자를 이용하는 경우 이용하는 트랜잭션 관리자의 qualifier를 지정한다. 기존 트랜잭션 관리자를 이용하는 경우에는 생략할 수 있다.
- transactionManager
  - value의 별칭(스프링 프레임워크 4.2 버전부터 추가)
- propagation
  - 트랜잭션의 전파 방식을 지정한다.
- isolation
  - 트랜잭션의 격리 수준을 지정한다.
- timeout
  - 트랜잭션 제한 시간(초)를 지정한다. 기본값은 -1(사용하는 데이터베이스의 사양이나 설정에 따라 다름)이다.

# @Transactional 애노테이션 속성 2/2

- **readonly**
  - 트랜잭션의 읽기 전용 플래그를 지정한다. 기본값은 false(읽기 전용이 아니다).
- **rollbackFor**
  - 여기에 지정한 예외가 발생하면 트랜잭션을 롤백시킨다. 예외 클래스명을 여러 개 나열할 수 있으며 ','로 구분한다. 따로 지정해주지 않으면 RuntimeException과 같은 비검사 예외가 발생할 때 트랜잭션이 롤백한다.
- **rollbackForClassName**
  - 여기에 지정한 예외가 발생하면 트랜잭션을 롤백시킨다. 예외 이름을 여러 개 나열할 수 있으며 ','로 구분한다.
- **noRollbackFor**
  - 여기에 지정한 예외가 발생하더라도 트랜잭션을 롤백시키지 않는다. 예외 클래스명을 여러개 나열할 수 있으며 ','로 구분한다.
- **noRollbackForClassName**
  - 여기에 지정한 예외가 발생하더라도 트랜잭션을 롤백시키지 않는다. 예외 이름을 여러 개 나열 할 수 있으며 ','로 구분한다.

# 명시적 트랜잭션

- 명시적 트랜잭션은 커밋이나 롤백과 같은 트랜잭션 처리를 소스코드에 직접 명시적으로 기술하는 방법이다. 메소드 단위보다도 더 작은 단위로 트랜잭션을 제어하고 싶거나 선언적 트랜잭션으로 표현하기 어려운 섬세한 트랜잭션 제어가 필요할 때 이 방법을 사용한다.
- PlatformTransactionManager를 이용한 명시적인 트랜잭션 제어
  - TransactionDefinition 및 TransactionStatus를 이용해 트랜잭션의 시작과 커밋, 그리고 롤백을 명시적으로 처리한다.
- TransactionTemplate을 활용한 명시적 트랜잭션 제어
  - PaltformTransactionManager보다도 구조적으로 트랜잭션 제어를 기술 할 수 있다.
  - 트랜잭션 제어 작업을 TransactionCallback 인터페이스가 제공하는 메소드에 구현하고 TransactionTemplate의 execute메소드에 인수로 전달한다. TransactionTemplate은 JdbcTemplate과 같은 방식을 사용하고 있기 때문에 애플리케이션 개발자는 필수 처리 로직만 구현하면 된다.

# 트랜잭션 격리 수준

- 트랜잭션 격리 수준은 참조하는 데이터나 변경한 데이터를 다른 트랜잭션으로부터 어떻게 격리할 것인지를 결정한다.
- 격리 수준은 여러 트랜잭션의 동시 실행과 데이터의 일관성과 깊이 관련 돼 있다.
- 트랜잭션 격리 수준은 스프링의 기본값인 DEFAULT에서 다른 수준으로 변경하고 싶은 경우에는 `@Transacational`의 `isolation` 속성, `TransactionDefinition`과 `TransactionTemplate`의 `setIsolationLevel` 메소드에서 지정할 수 있다.
- 스프링 프레임워크에서는 데이터베이스의 기본 설정과 4개의 트랜잭션 격리 수준을 이용할 수 있다. 지원하는 모든 격리 수준이 실제로 사용할 수 있는지는 사용하는 데이터베이스가 어떻게 구현했느냐에 따라 달라질 수 있다.

# 트랜잭션 격리 수준

- DEFAULT
  - 사용하는 데이터베이스의 기본 격리 수준을 이용한다.
- READ\_UNCOMMITTED
  - 더티 리드(Dirty Read). 반복되지 않는 읽기(Unrepeatable Read), 팬텀 읽기(Phantom Read)가 발생한다. 이 격리 수준은 커밋되지 않은 변경 데이터를 다른 트랜잭션에서 참조하는 것을 허용한다. 만약 변경 데이터가 롤백된 경우 다음 트랜잭션에서 무효한 데이터를 조회하게 된다.
- READ\_COMMITTED
  - 더티 리드를 방지하지만 반복되지 않는 읽기, 팬텀 읽기는 발생한다. 이 격리 수준은 커밋되지 않은 변경 데이터를 다른 트랜잭션에서 참조하는 것을 금지한다.
- REPEATABLE\_READ
  - 더티리드, 반복되지 않은 읽기를 방지하지만 팬텀 읽기는 발생한다.
- SERIALIZABLE
  - 더티리드, 반복되지 않는 읽기, 팬텀 읽기를 방지한다.

# DBMS 격리 수준 1/3

- 트랜잭션 수준으로 읽기 일관성을 강화 하려면 트랜잭션 고립화 수준을 높여 주어야 한다.
- 트랜잭션 고립화 수준을 조절하는 방법은 네가지 수준이 있다.
- ANSI / ISO SQL standard(SQL92)에서 정의함

# DBMS 격리 수준 2/3

- 레벨 0 ( = Read Uncommitted )
  - 트랜잭션에서 처리중인 / 아직 커밋되지 않은 데이터를 다른 트랜잭션이 읽는 것을 허용
  - Dirty Read, Non-repeatable Read, Phantom Read 현상 발생
  - 오라클은 이 레벨을 지원하지 않음
- 레벨 1 ( = Read Committed )
  - Dirty Read 방지 : 트랜잭션이 커밋되어 확정된 데이터만 읽는 것을 허용
  - 대부분의 DBMS가 기본모드로 채택하고 있는 일관성 모드
  - Non-Repeatable Read, Phantom Read 현상 발생
  - DB2, SQL Server, Sybase의 경우 읽기 공유 Lock을 이용해서 구현, 하나의 레코드를 읽을 때 Lock을 설정하고 해당 레코드에서 빠지는 순간 Lock해제
  - Oracle은 Lock을 사용하지 않고 쿼리시작 시점의 Undo 데이터를 제공하는 방식으로 구현

# DBMS 격리 수준 3/3

- 레벨 2 ( = Repeatable Read )

- 선행 트랜잭션이 읽은 데이터는 트랜잭션이 종료될 때까지 후행 트랜잭션이 갱신하거나 삭제하는 것을 불허함으로써 같은 데이터를 두 번 퀼리했을 때 일관성 있는 결과를 리턴
- Phantom Read 현상 발생
- DB2, SQL Server의 경우 트랜잭션 고립화 수준을 Repeatable Read로 변경하면 읽은 데이터에 걸린 공유 Lock을 커밋할 때까지 유지하는 방식으로 구현
- Oracle은 이 레벨을 명시적으로 지원하지 않지만 for update 절을 이용해 구현 가능,  
SQL Server 등에서도 for update 절을 사용할 수 있지만 커서를 명시적으로 선언할 때만 사용 가능함
- MySQL에서 기본적으로 사용하는 격리수준

- 레벨 3 ( = Serializable Read )

- 선행 트랜잭션이 읽은 데이터를 후행 트랜잭션이 갱신하거나 삭제하지 못할 뿐만 아니라 중간에 새로운 레코드를 삽입하는 것도 막아줌
- 완벽한 읽기 일관성 모드를 제공

# **Dirty Read, Non-Repeatable Read, Phantom Read**

## **Dirty Read ( = Uncommitted Dependency)**

- 커밋되지 않은 수정 중인 데이터를 다른 트랜잭션에서 읽을 수 있도록 허용할 때 발생
- 대부분 DBMS가 기본 트랜잭션 고립화 수준을 레벨1로 설정하고 있어 Dirty Read는 발생하지 않음  
= 커밋된 데이터만 읽을 수 있도록 허용
- SQL Server, Sybase 등은 데이터를 읽을 때 공유 Lock을 사용,
- 갱신중인 레코드에는 배타적 Lock이 걸림, 이는 공유 Lock과는 호환되지 않아 갱신중인 레코드는 읽지 못함(Lock에 의한 동시성 저하 발생)

## **Non-Repeatable Read**

- 한 트랜잭션 내에서 같은 쿼리를 두 번 수행 할 때 그 사이에 다른 트랜잭션이 값을 수정 또는 삭제함으로써
- 두 쿼리의 결과가 상이하게 나타나는 비일관성이 발생하는 것을 말함

## **Phantom Read**

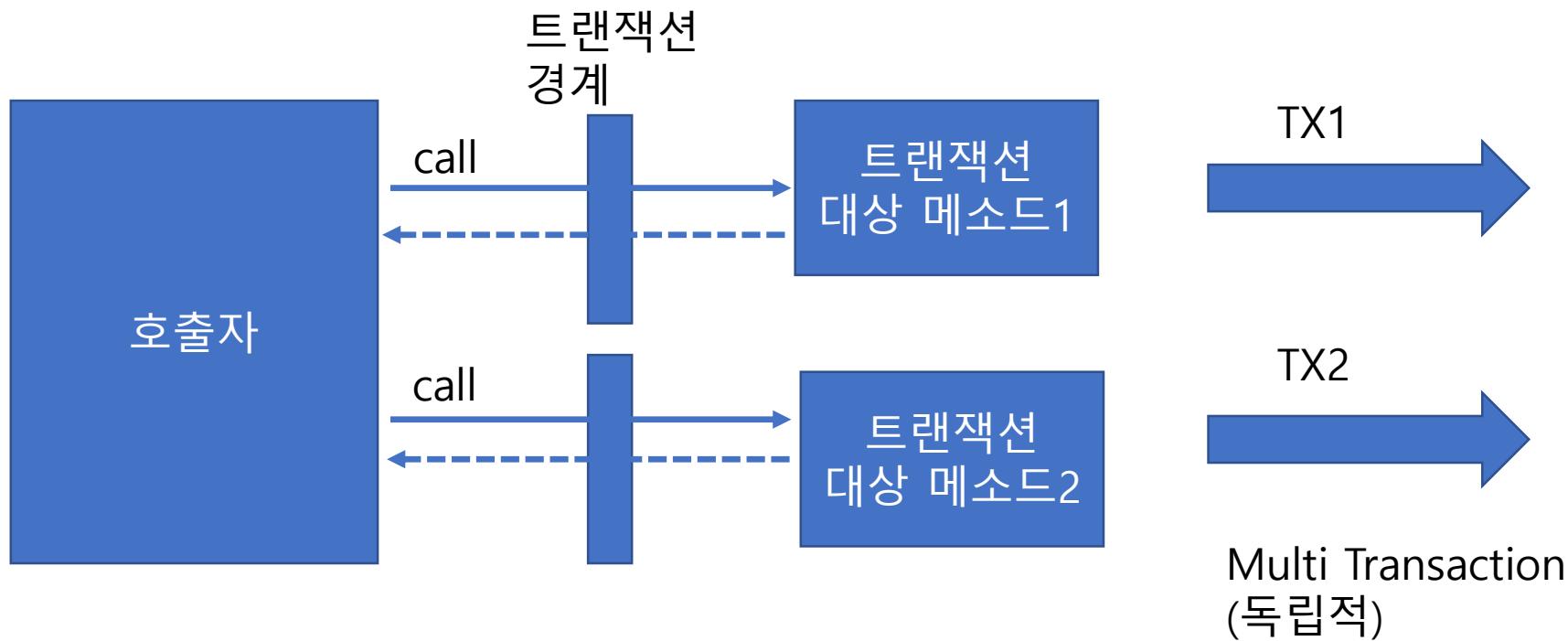
- 한 트랜잭션 안에서 일정 범위의 레코드를 두 번 이상 읽을 때, 첫번째 쿼리에서 없던 레코드가 두번째 쿼리에서 나타나는 현상
- 이는 트랜잭션 도중 새로운 레코드가 삽입되는 것을 허용하기 때문에 나타나는 현상

# 트랜잭션 전파 방식

- 트랜잭션 전파 방식은 트랜잭션 경계에서 트랜잭션에 참여하는 방법을 결정한다.
- 지원하는 방식에 따라 '새로운 트랜잭션을 시작하는 것', '이미 시작된 트랜잭션에 참여하는 것'과 같이 몇 가지 선택지가 준비 돼 있다.

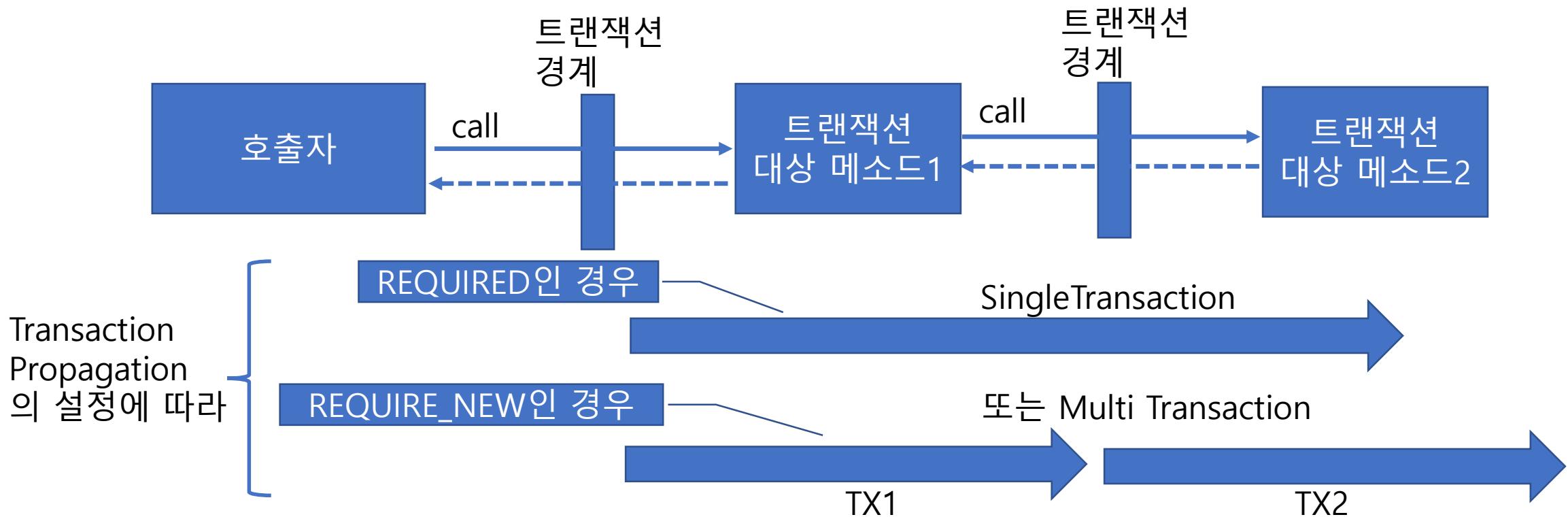
# 트랜잭션 경계와 전파 방식 1/2

- 트랜잭션 전파 방식을 의식해야 하는 경우는 트랜잭션 경계가 중첩될 때다. 여러 개의 트랜잭션 경계가 중첩되지 않았다면 트랜잭션을 TX1시작 -> TX1커밋 -> TX2 시작 -> TX2 커밋과 같이 순차적으로 제어만 하면 되기 때문에 굳이 전파 방식을 의식할 필요는 없다.



# 트랜잭션 경계와 전파 방식 2/2

- 트랜잭션 관리 대상이 되는 메소드 안에서 또 다른 트랜잭션 관리 대상이 되는 메소드를 호출한 경우에는 트랜잭션의 전파 방식을 고려해야 한다. 예를 들어, 두 메소드가 각각 독립적인 트랜잭션으로 관리될지, 혹은 같은 트랜잭션의 관리 범위에 들어가는지는 트랜잭션의 전파 방식에 따라 좌우된다.



## 스프링 프레임워크에서 이용 가능한 트랜잭션 전파방식 1/2

- 스프링 프레임워크에서는 다음과 같은 7가지 트랜잭션 전파 방식을 이용할 수 있다.
- 트랜잭션 전파 방식을 기본값인 REQUIRED에서 다른 방식으로 변경하고 싶은 경우에는 @Transactional의 propagation속성이나 TransactionDefinition과 TransactionTemplate의 setPropagationBehavior메소드에서 지정할 수 있다.

# 스프링 프레임워크에서 이용 가능한 트랜잭션 전파방식 2/2

- REQUIRED
  - 이미 만들어진 트랜잭션이 존재한다면 해당 트랜잭션 관리 범위 안에 함께 들어간다. 만약 이미 만들어진 트랜잭션이 존재하지 않는다면 새로운 트랜잭션을 만든다.
- REQUIRES\_NEW
  - 이미 만들어진 트랜잭션 범위안에 들어가지 않고 반드시 새로운 트랜잭션을 만든다. 만약 이미 만들어진 트랜잭션이 아직 종료되지 않았다면 새로운 트랜잭션은 보류 상태가 되어 이전 트랜잭션이 끝나는 것을 기다려야 한다.
- MANDATORY
  - 이미 만들어진 트랜잭션 범위 안에 들어가야 한다. 만약 기존에 만들어진 트랜잭션이 없다면 예외가 발생한다.
- NEVER
  - 트랜잭션 관리를 하지 않는다. 이미 만들어진 트랜잭션이 있다면 예외가 발생한다.
- NOT\_SUPPORTED
  - 트랜잭션을 관리하지 않는다. 만약 이미 만들어진 트랜잭션이 있다면 이전 트랜잭션이 끝나는 것을 기다려야 한다.
- SUPPORTS
  - 이미 만들어진 트랜잭션이 있다면 그 범위 안에 들어가고, 만약 트랜잭션이 없다면 트랜잭션 관리를 하지 않는다.
- NESTED
  - REQUIRED와 마찬가지로 현재 트랜잭션이 존재하지 않으면 새로운 트랜잭션을 만들고 이미 존재하는 경우에는 이미 만들어진 것을 계속 이용하지만, NESTED가 적용된 구간 중첩된 트랜잭션처럼 취급한다.NESTED구간 안에서 롤백이 발생할 경우 NESTED구간 안의 처리 내용은 모두 롤백되지만 NESTED구간 밖에서 실행된 처리 내용은 롤백되지 않는다. 단 부모 트랜잭션에서 롤백되면 NESTED구간의 트랜잭션은 모두 롤백된다.

# 트랜잭션 전파 방식을 의식할 필요가 있는 예

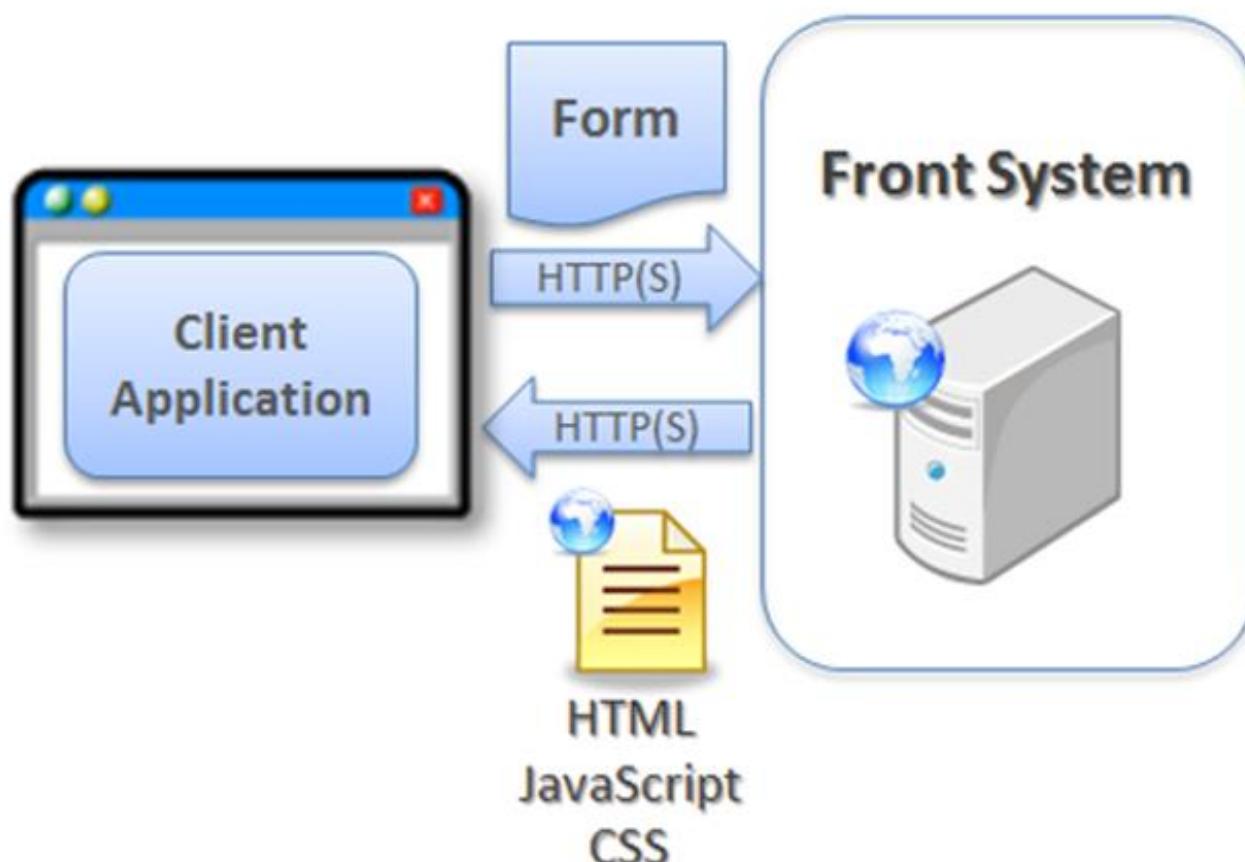
- 애플리케이션에서 처리 내용을 추적하는 로그를 데이터베이스에 저장해야 한다고 가정하자. 만약 비지니스 로직이 실패했을 때 업무 데이터는 롤백하고 싶지만 로그 데이터는 커밋하고 싶을 때 트랜잭션 경계가 겹치게 된다.
- 이미 비지니스 로직과 로그 처리를 서로 다른 트랜잭션 경계로 정의했다고 스프링의 기본 전파 방식인 REQUIRED를 사용하고 있다면 로그 데이터가 업무 데이터와 함께 롤백할 수 있다. 이러한 경우에는 로그 출력용 트랜잭션의 전파 방식을 REQUIRES\_NEW로 변경하면 된다. 즉 업무 트랜잭션과는 다른 새로운 트랜잭션을 만들어서 업무 트랜잭션이 롤백되더라도 로그 출력 트랜잭션은 롤백되지 않도록 만들어야 한다.

# Spring MVC

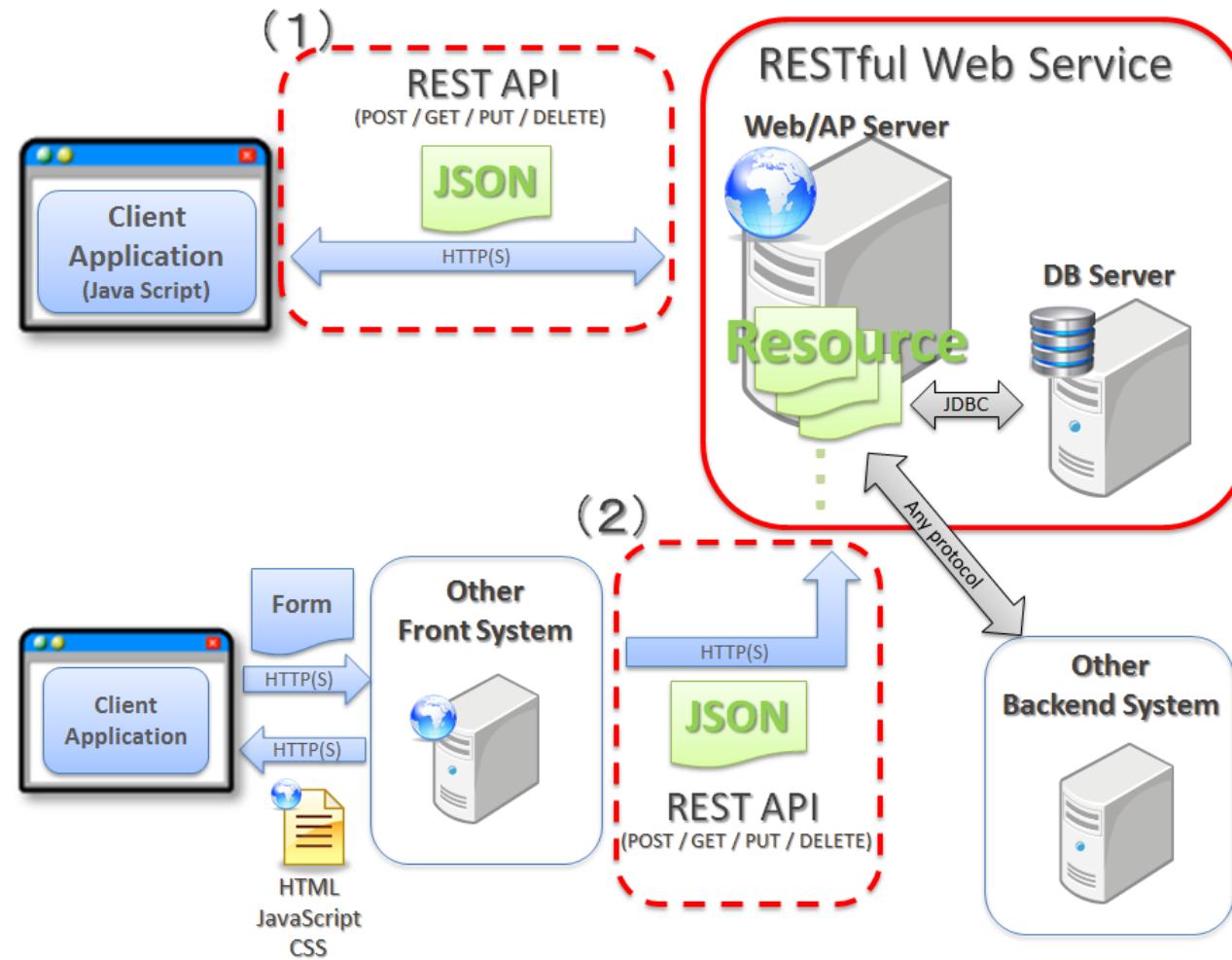
# 웹 애플리케이션의 종류

- 화면으로 응답하는 애플리케이션
  - @Controller 애노테이션 사용
- 데이터로 응답하는 애플리케이션
  - @RestController 애노테이션 사용

# 화면으로 응답하는 애플리케이션

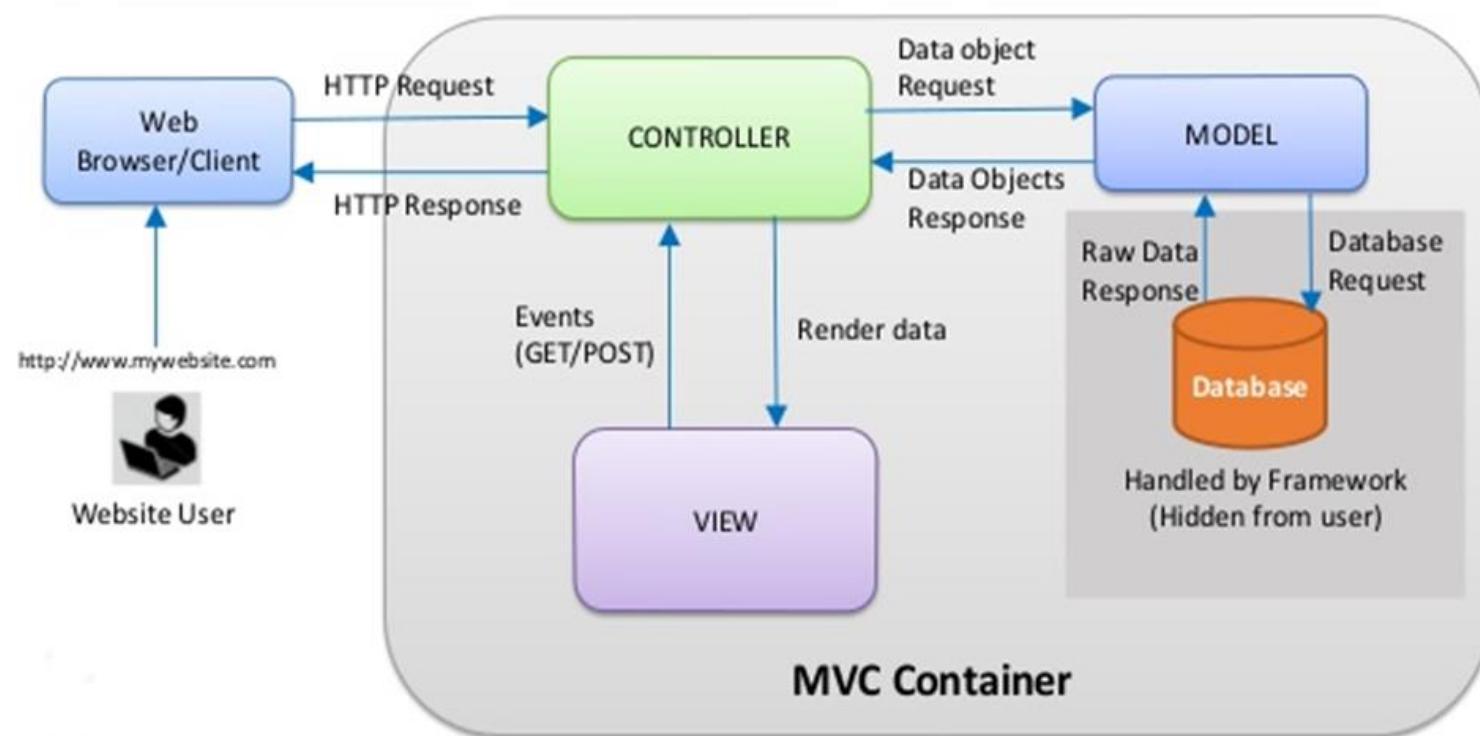


# 데이터로 응답하는 애플리케이션



# MVC?

- MVC는 Model-View-Controller의 약자이다.

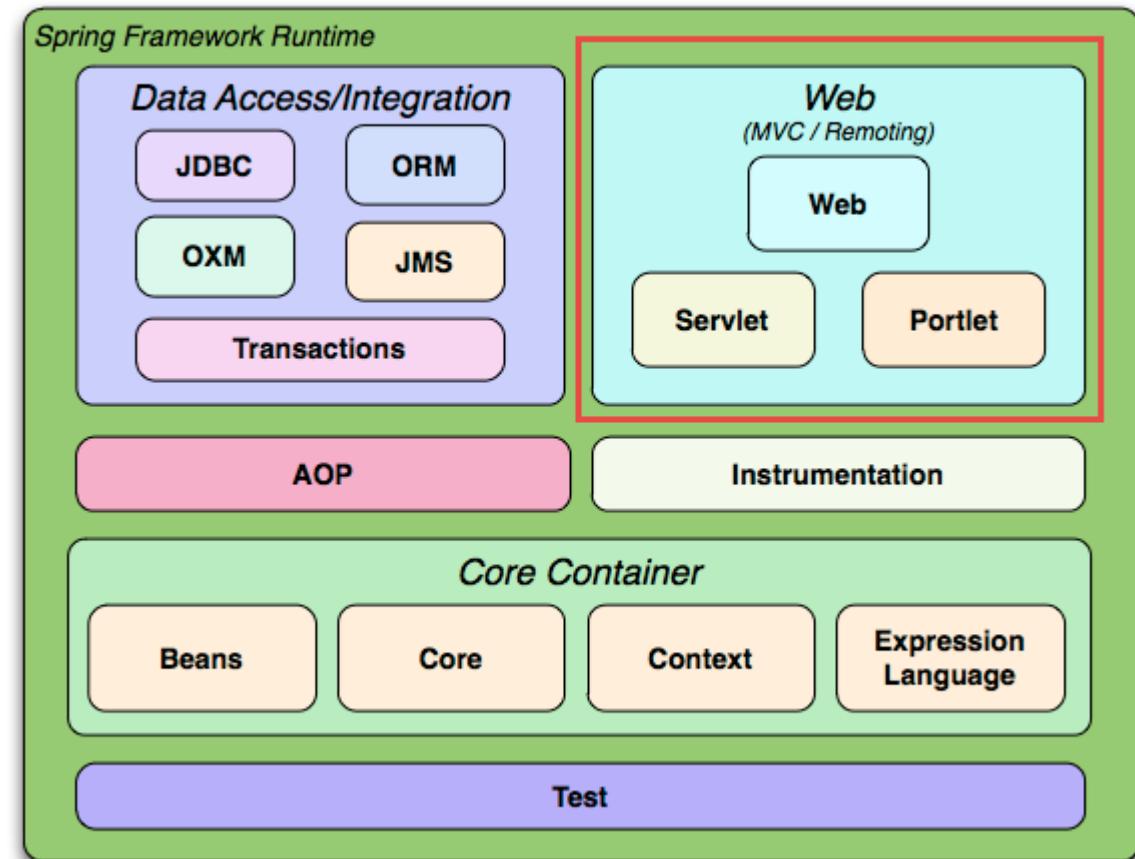


# MVC?

- Model : 애플리케이션 상태(데이터)나 비즈니스 로직을 제공하는 컴포넌트.
- View : 모델이 보유한 애플리케이션 상태(데이터)를 참조하고 클라이언트에 반환할 응답 데이터를 생성하는 컴포넌트.
- Controller : 요청을 받아 모델과 뷰의 호출을 제어하는 컴포넌트로 컨트롤러라는 이름처럼 요청과 응답의 처리흐름을 제어한다.

# Spring Web Module

- Model2 MVC 패턴을 지원하는 Spring Module



# 웹 애플리케이션 개발의 특징

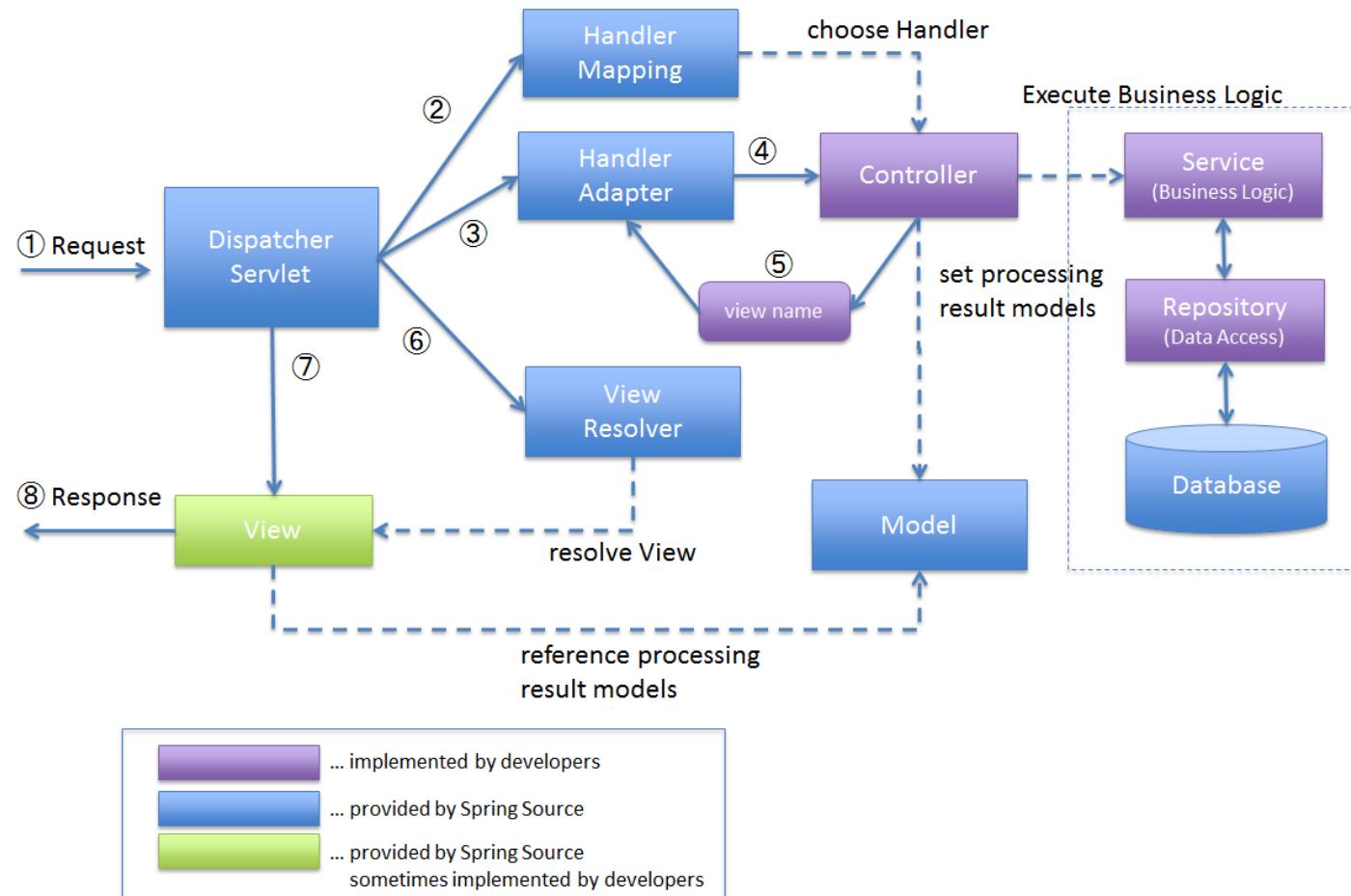
- POJO 구현
  - 컨트롤러나 모델 등의 클래스는 POJO형태로 구현한다. 특정 프레임워크등에 종속적이지 않기 때문에 테스트 등이 용이하다.
- 애노테이션을 이용한 정의 정보 설정
  - 요청 매팅과 같은 각종 정의 정보를 설정 파일이 아닌 애노테이션을 이용해 설정할 수 있다. 비즈니스 로직과 그 로직을 수행하기 위한 각종 정의 정보가 자바 클래스에 함께 기술되기 때문에 효율적으로 개발할 수 있다.
- 유연한 메소드 시그니처 정의
  - 컨트롤러 클래스의 메소드 매개변수에는 처리에 필요한 것만 골라서 정의할 수 있다. 인수에 지정할 수 있는 타입도 다양한 타입이 지원되며, 프레임워크가 인수에 전달하는 값을 자동으로 담아주거나 변환하기 때문에 사양변경이나 리펙토링에 강한 아키텍처를 가진다.
- Servlet API 추상화
  - 스프링 MVC는 서블릿 API를 추상화하는 기능을 가진다. 서블릿 API를 직접 이용하는 것보다 쉽게 개발할 수 있다.
- 뷰 구현 기술의 추상화
  - 컨트롤러는 뷰 이름(뷰의 논리적인 이름)을 반환하고 스프링 MVC는 뷰 이름에 해당하는 화면이 표시하게 한다. 컨트롤러는 뷰 이름만 알면 되기 때문에 그 뷰가 어떤 구현기술(JSP, Thymeleaf, Freemarker등)로 만들어졌는지 자세히 몰라도 된다.
- 스프링의 DI 컨테이너와 연계
  - 스프링 MVC는 스프링의 DI 컨테이너 상에서 동작하는 프레임워크다. 스프링의 DI컨테이너가 제공하는 DI나 AOP와 같은 구조를 그대로 활용할 수 있다는 장점이 있다.

# MVC 프레임워크로서의 특징

- 풍부한 확장 포인트 제공
  - 스프링 MVC에서는 컨트롤러나 뷰와 같이 각 역할별로 필요한 인터페이스를 제공한다. 기본 동작을 확장하고자 한다면 이러한 인터페이스를 재 구현하면 된다. 커스터마이징이 강력하다.
- 엔터프라이즈 애플리케이션에 필요한 기능 제공
  - 스프링 MVC는 단순히 MVC패턴의 프레임워크 구현만 제공하는 것이 아니라 메시지 관리, 세션 관리, 국제화 등 다양한 기능을 제공한다.
- 서드파티 라이브러리와의 연계지원
  - 스프링 MVC는 서드파티 라이브러리를 이용할 때 필요한 각종 아답터를 제공하며 다음과 같은 라이브러리를 스프링 MVC와 연계해서 사용할 수 있다.
    - Jackson(JSON/XML처리), Google Gson(JSON처리), Google Protocol Buffers(Protocol Buffers로 불리는 직렬화 형식 처리), Apache Tiles(레이아웃 엔진), Freemarker(템플릿 엔진), Rome(RSS/Feed처리), JasperReports(보고서 출력), ApachePO(엑셀 처리), Hibernate Validator(빈 유효성 검증), Joda-Time(날짜/시간 처리)
- 서드파티 라이브러리 자체가 스프링 MVC와의 연계를 지원하는 경우도 있다.
  - Thymeleaf(템플릿 엔진), HDIV(보안 강화)

# Spring MVC 구성요소와 동작 이해

# Spring MVC 기본 동작 흐름



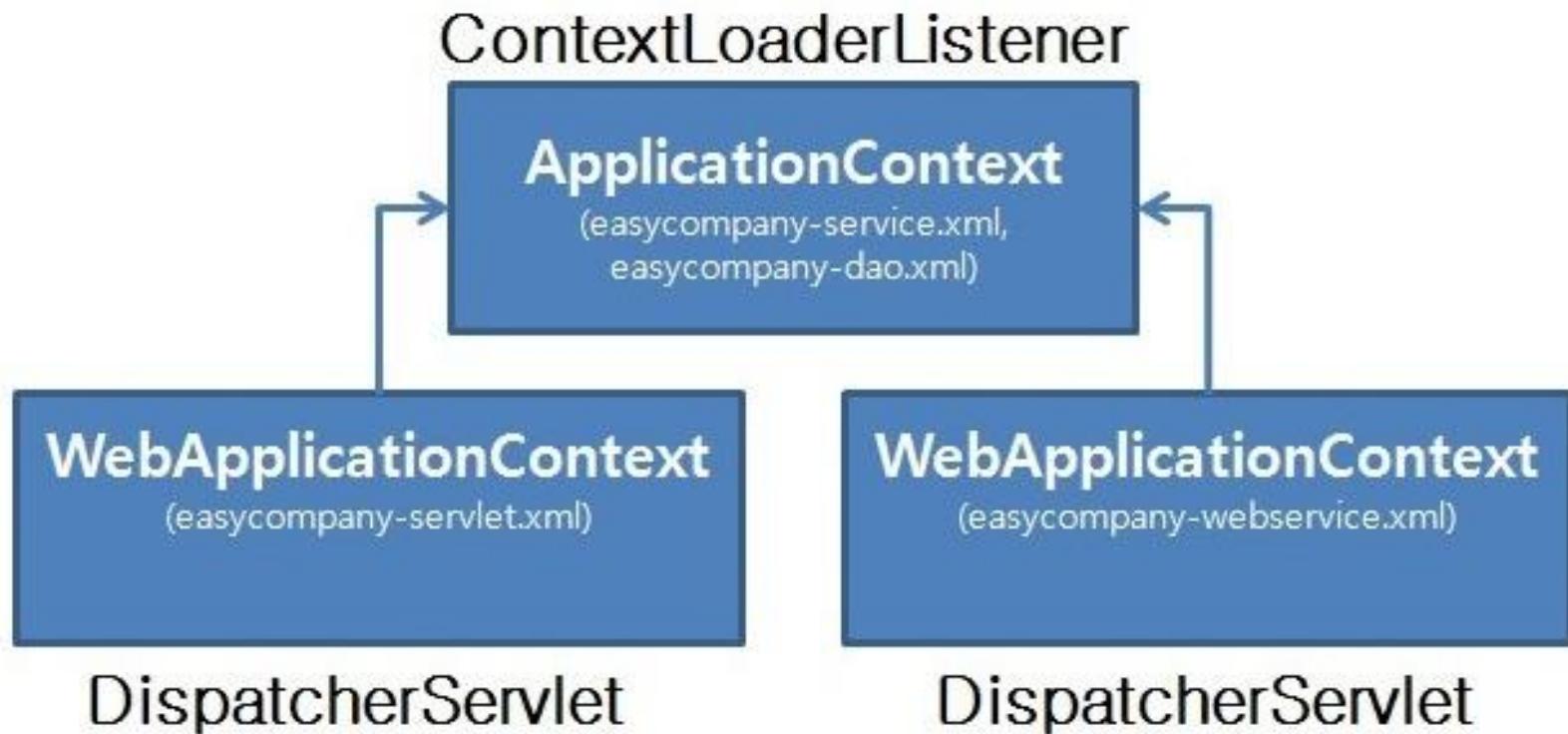
# 요청 처리를 위해 사용되는 컴포넌트

- DispatcherServlet
  - HandlerMapping
  - HandlerAdapter
  - MultipartResolver
  - LocaleResolver
  - ThemeResolver
  - HandlerExceptionResolver
  - RequestToViewNameTranslator
  - ViewResolver
  - FlashMapManager

# Spring MVC 프로젝트 설정

- ContextLoaderListener 설정
  - 웹 애플리케이션에서 사용할 애플리케이션 컨텍스트를 만들려면 서블릿 컨테이너에 ContextLoaderListener를 설정한다.
- DispatcherServlet 설정
  - 프론트 컨트롤러 역할을 수행하는 DispatcherServlet을 등록한다.
- CharacterEncodingFilter 설정
  - 입력 값의 한국어가 깨지지 않도록 CharacterEncodingFilter를 서블릿 컨테이너에 등록한다.
- ViewResolver 설정
  - 스프링 MVC에서는 논리적인 뷰 이름을 보고 실제로 표시할 물리적인 뷰가 무엇인지 판단할 때 ViewResolver 컴포넌트를 사용한다. JSP를 사용하기 위해서는 JSP용 ViewResolver를 설정한다.

# DispatcherServlet을 여러개 정의할 경우의 애플리케이션 컨텍스트 구성



# DispatcherServlet을 FrontController로 설정하기

- web.xml 파일에 설정
- javax.servlet.ServletContainerInitializer 사용
  - 서블릿 3.0 스펙 이상에서 web.xml파일을 대신해서 사용할 수 있다.
- org.springframework.web.WebApplicationInitializer 인터페이스를 구현해서 사용

# web.xml파일에서 DispatcherServlet 설정하기 1/2

- xml spring 설정 읽어들이도록 DispatcherServlet설정

```
<?xml version="1.0" encoding = "UTF-8"?>
<web-app>

    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:WebMVCConfig.xml</param-value>
        </init-param>
    </servlet>
</web-app>
```

# web.xml파일에서 DispatcherServlet 설정하기

- Java config spring 설정 읽어들이도록 DispatcherServlet설정

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="2.5" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
metadata-complete="true">

    <display-name>Spring JavaConfig Sample</display-name>

    <servlet>
        <servlet-name>mvc</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>
        </init-param>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>kr.or.connect.webmvc.config.WebMvcContextConfiguration</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>mvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

# WebApplicationInitializer를 구현해서 설정하기 1/2

- Spring MVC는 ServletContainerInitializer를 구현하고 있는 SpringServletContainerInitializer를 제공한다.
- SpringServletContainerInitializer는 WebApplicationInitializer 구현체를 찾아 인스턴스를 만들고 해당 인스턴스의 onStartup메소드를 호출하여 초기화 한다.

# WebApplicationInitializer를 구현해서 설정하기 2/2

```
public class WebApplicationInitializer implements WebApplicationInitializer {

    private static final String DISPATCHER_SERVLET_NAME = "dispatcher";

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        registerDispatcherServlet(servletContext);
    }

    private void registerDispatcherServlet(ServletContext servletContext) {
        AnnotationConfigWebApplicationContext dispatcherContext =
            createContext(WebMvcContextConfiguration.class);
        ServletRegistration.Dynamic dispatcher;
        dispatcher = servletContext.addServlet(DISPATCHER_SERVLET_NAME,
            new DispatcherServlet(dispatcherContext));
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");
    }

    private AnnotationConfigWebApplicationContext createContext(final Class<?>... annotatedClasses) {
        AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();
        context.register(annotatedClasses);
        return context;
    }
}
```

# Spring MVC 설정

- kr.or.connect.webmvc.config.WebMvcContextConfiguration

```
@Configuration  
@EnableWebMvc  
@ComponentScan(basePackages = { "kr.or.connect.webmvc.controller" })  
public class WebMvcContextConfiguration extends WebMvcConfigurerAdapter {  
  
    ....  
  
}
```

# @Configuration

- org.springframework.context.annotation 의 Configuration 애노테이션과 Bean 애노테이션 코드를 이용하여 스프링 컨테이너에 새로운 빈 객체를 제공할 수 있다.

# @EnableWebMvc

- DispatcherServlet의 RequestMappingHandlerMapping, RequestMappingHandlerAdapter, ExceptionHandlerExceptionResolver, MessageConverter 등 Web에 필요한 빈들을 대부분 자동으로 설정 해준다.
- xml로 설정의 <mvc:annotation-driven/> 와 동일하다.
- 기본 설정 이외의 설정이 필요하다면 WebMvcConfigurerAdapter 를 상속받도록 Java config class를 작성한 후, 필요한 메소드를 오버라이딩 하도록 한다.

# @EnableWebMvc

```
@Configuration
public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {
    ...
    @Autowired(required = false)
    public void setConfigurers(List<WebMvcConfigurer> configurers) {
        if (!CollectionUtils.isEmpty(configurers)) {
            this.configurers.addWebMvcConfigurers(configurers);
        }
    }
}
```

```
.....
@Import(DelegatingWebMvcConfiguration.class)
public @interface EnableWebMvc {
}
```



# WebMvcConfigurationSupport

- <https://github.com/spring-projects/spring-framework/blob/master/spring-webmvc/src/main/java/org/springframework/web/servlet/config/annotation/WebMvcConfigurationSupport.java>

# @ComponentScan

- ComponentScan 애노테이션을 이용하면 Controller, Service, Repository, Component 애노테이션이 붙은 클래스를 찾아 스프링 컨테이너가 관리하게 된다.
- DefaultAnnotationHandlerMapping과 RequestMappingHandlerMapping 구현체는 다른 핸들러 맵핑보다 훨씬 더 정교한 작업을 수행한다. 이 두 개의 구현체는 애노테이션을 사용해 맵핑 관계를 찾는 매우 강력한 기능을 가지고 있다. 이들 구현체는 스프링 컨테이너 즉 애플리케이션 컨텍스트에 있는 요청 처리 빈에서 RequestMapping 애노테이션을 클래스나 메소드에서 찾아 HandlerMapping 객체를 생성하게 된다.
  - HandlerMapping은 서버로 들어온 요청을 어느 핸들러로 전달할지 결정하는 역할을 수행한다.
- DefaultAnnotationHandlerMapping은 DispatcherServlet이 기본으로 등록하는 기본 핸들러 맵핑 객체이고, RequestMappingHandlerMapping은 더 강력하고 유연하지만 사용하려면 명시적으로 설정해야 한다.

# WebMvcConfigurerAdapter

- org.springframework.web.servlet.config.annotation.

## WebMvcConfigurerAdapter

- @EnableWebMvc 를 이용하면 기본적인 설정이 모두 자동으로 되지만, 기본 설정 이외의 설정이 필요할 경우 해당 클래스를 상속 받은 후, 메소드를 오버라이딩 하여 구현한다.

# DispatcherServlet

- 프론트 컨트롤러 (Front Controller)
- 클라이언트의 모든 요청을 받은 후 이를 처리할 핸들러에게 넘기고 핸들러가 처리한 결과를 받아 사용자에게 응답 결과를 보여준다.
- DispatcherServlet은 여러 컴포넌트를 이용해 작업을 처리한다.

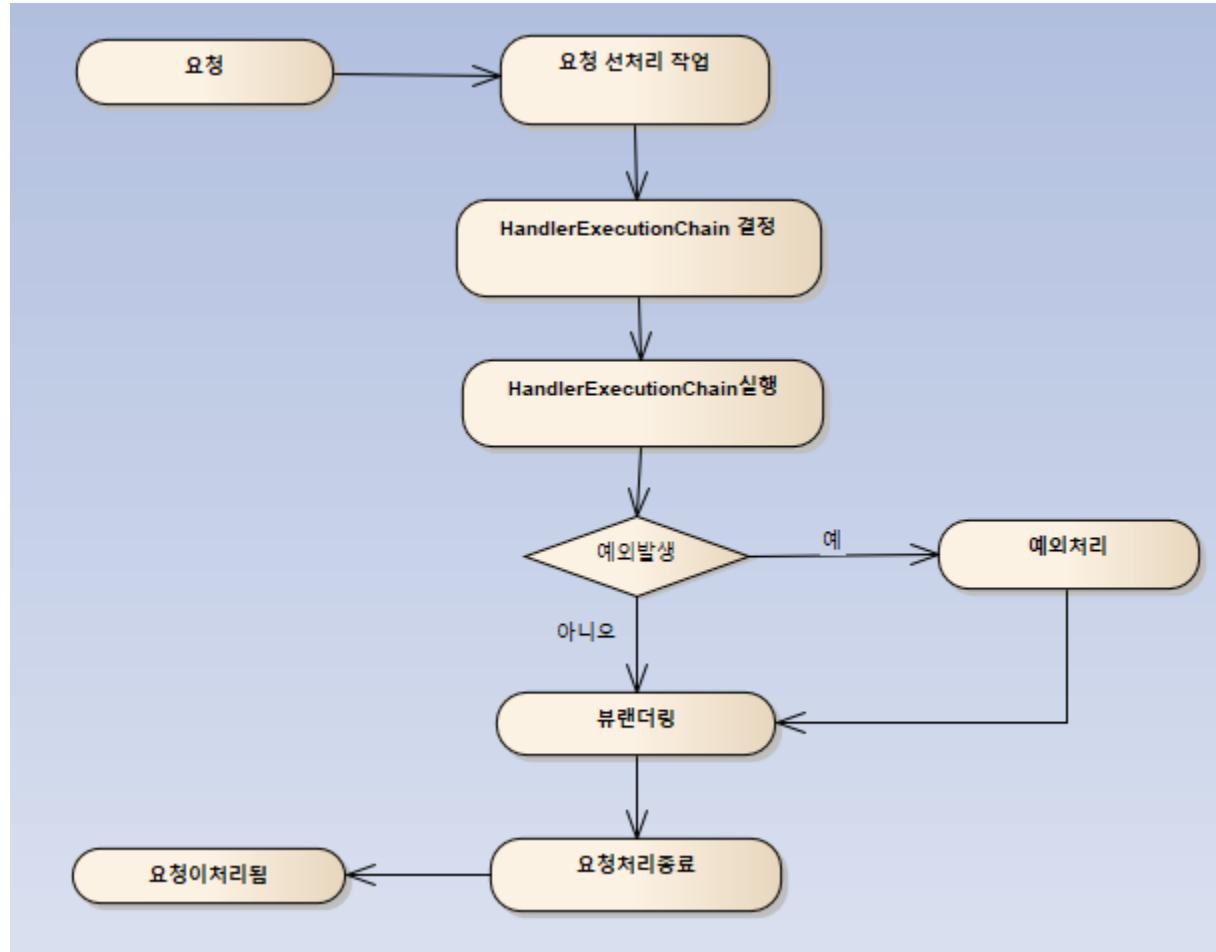
# 프레임워크 내부 동작에 필요한 인터페이스 1/2

- HandlerExceptionResolver
  - 예외 처리를 하기 위한 인터페이스. 스프링 MVC가 제공하는 기본 구현 클래스가 적용돼 있다.
- LocaleResolver, LocaleContextResolver
  - 클라이언트의 로캘 정보를 확인하기 위한 인터페이스. 스프링 MVC가 제공하는 기본 구현 클래스가 적용돼 있다.
- ThemeResolver
  - 클라이언트의 테마(UI 스타일)을 결정하기 위한 인터페이스. 스프링 MVC가 제공하는 기본 구현 클래스가 적용돼 있다.
- FlashMapManager
  - FlashMap이라는 객체를 관리하기 위한 인터페이스. FlashMap은 PRG(Post Redirect Get)패턴의 Redirect와 Get사이에서 모델을 공유하기 위한 Map객체다. 스프링 MVC에서 제공하는 기본 구현 클래스가 적용돼 있다.

# 프레임워크 내부 동작에 필요한 인터페이스 2/2

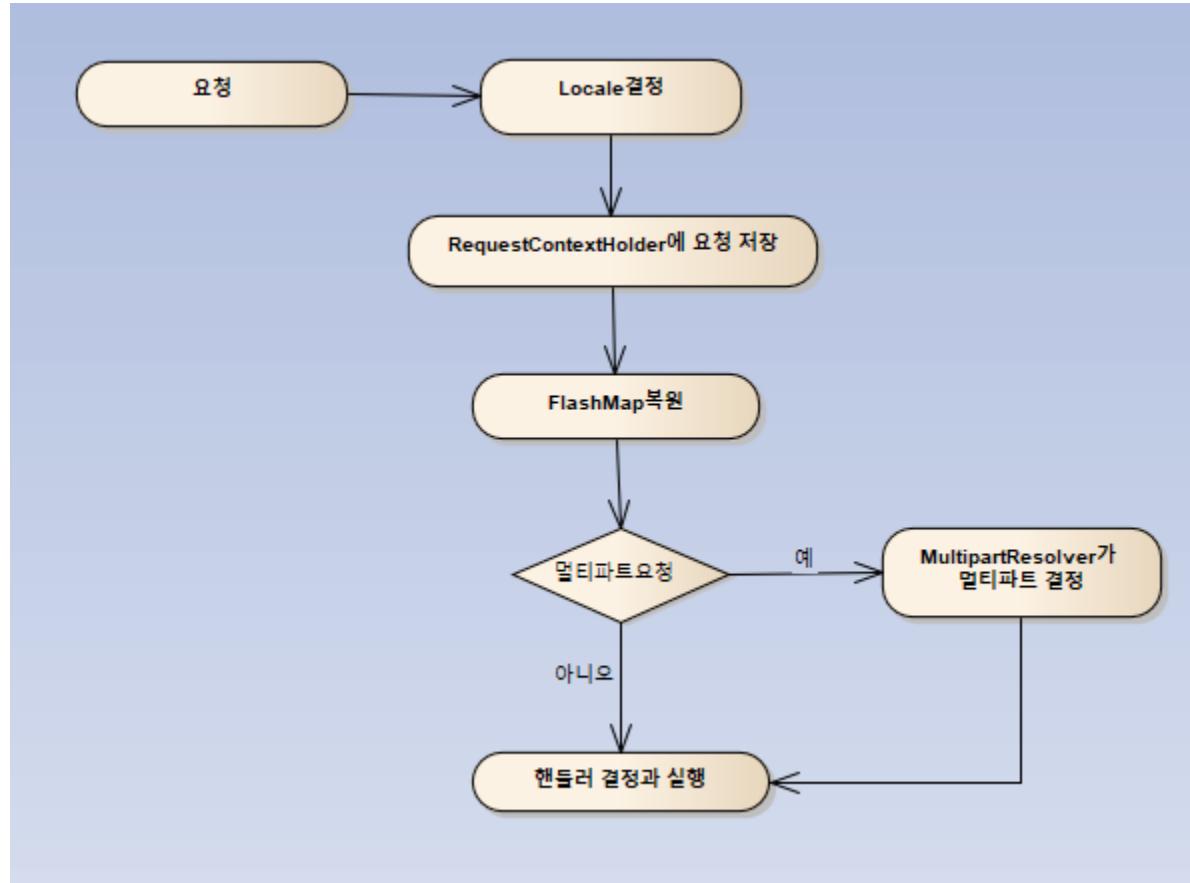
- RequestToViewNameTranslator
  - 핸들러가 뷰 이름과 뷰를 반환하지 않은 경우에 적용되는 뷰 이름을 해결하기 위한 인터페이스. 스프링 MVC에서 제공하는 기본 구현 클래스가 적용돼 있다.
- HandlerInterceptor
  - 핸들러 실행 전후에 하는 공통 처리를 구현하기 위한 인터페이스. 이 인터페이스는 애플리케이션 개발자가 구현하고 스프링 MVC에 등록해서 사용할 수 있다.
- MultipartResolver
  - 멀티파트 요청을 처리하기 위한 인터페이스. 스프링 MVC에서 몇 가지 구현 클래스가 제공되고 있지만 기본적으로 적용되지 않는다.

# DispatcherServlet 내부 동작흐름



# DispatcherServlet 내부 동작흐름 상세 - 요청 선처리 작업

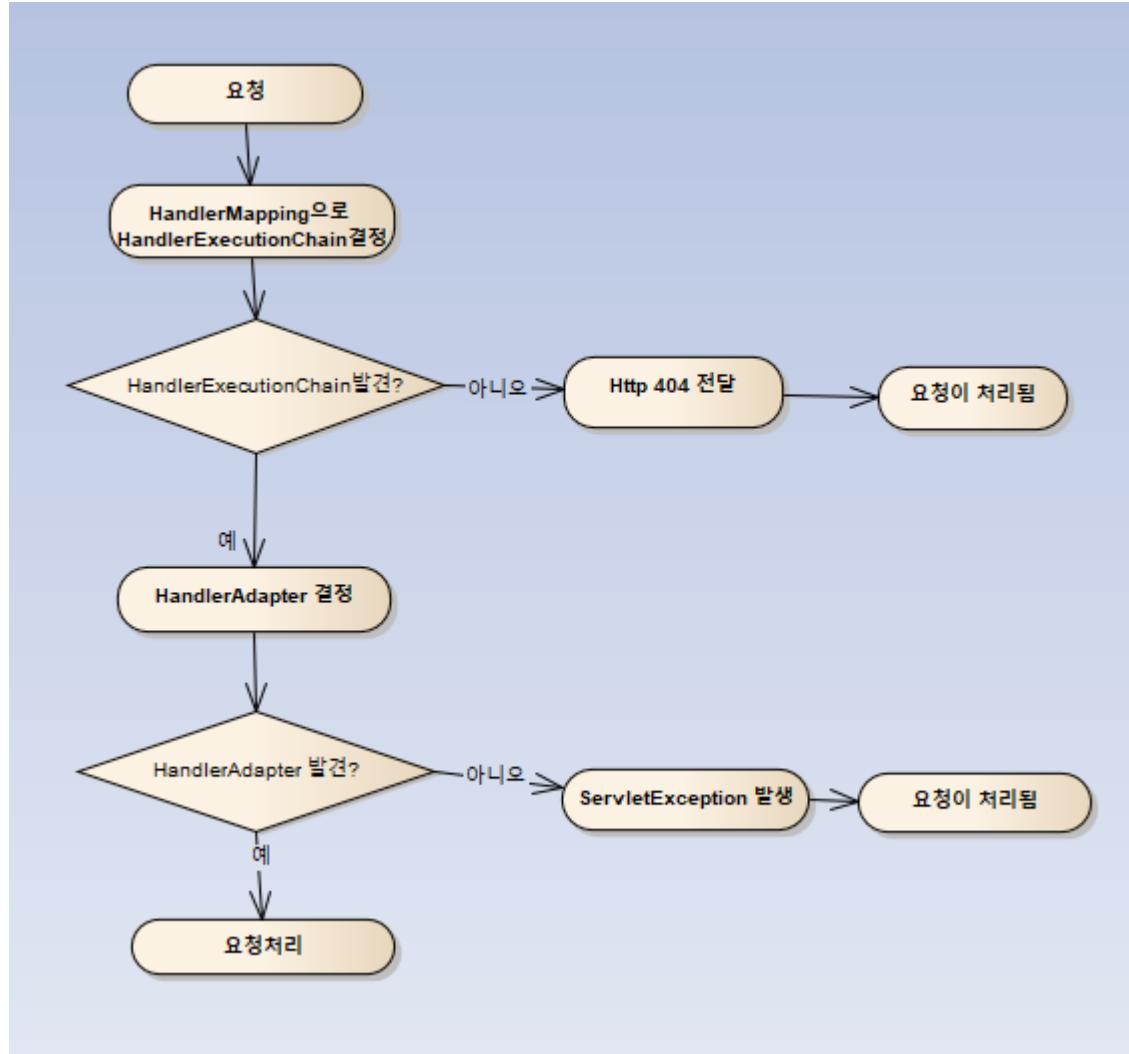
ThreadLocal



# 요청 선처리 작업시 사용된 컴포넌트

- org.springframework.web.servlet.LocaleResolver
  - 지역 정보를 결정해주는 전략 오브젝트이다. 디폴트인 AcceptHeaderLocaleResolver는 HTTP 헤더의 정보를 보고 지역정보를 설정해준다.
- org.springframework.web.servlet.FlashMapManager
  - FlashMap 객체를 조회(retrieve) & 저장을 위한 인터페이스
  - RedirectAttributes의 addFlashAttribute 메소드를 이용해서 저장한다.
  - 리다이렉트 후 조회를 하면 바로 정보는 삭제된다.
- org.springframework.web.context.request.RequestContextHolder
  - 일반 빈에서 HttpServletRequest, HttpServletResponse, HttpSession 등을 사용할 수 있도록 한다.
  - 해당 객체를 일반 빈에서 사용하게 되면, Web에 종속적이 될 수 있다.
- org.springframework.web.multipart.MultipartResolver
  - 멀티파트 파일 업로드를 처리하는 전략

# DispatcherServlet 내부 동작흐름 상세 - 요청 전달



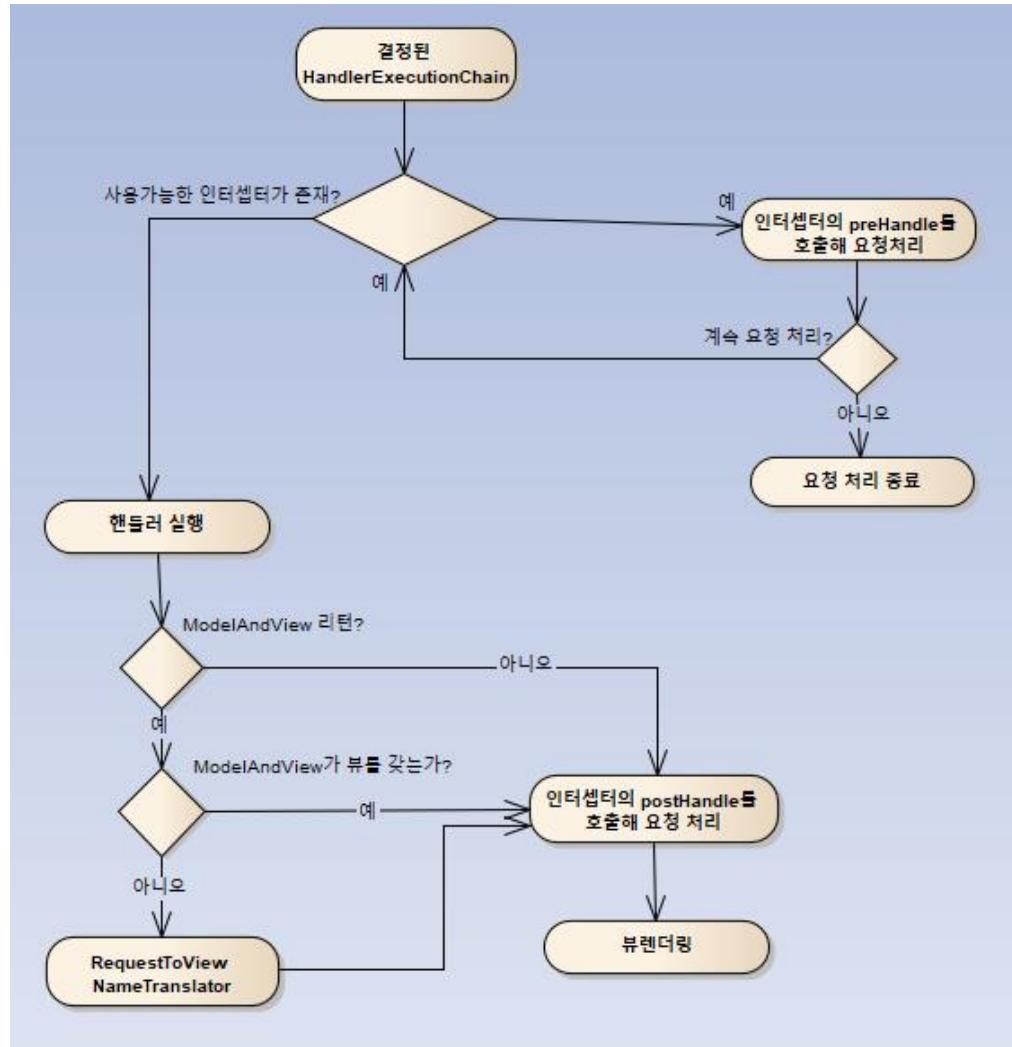
# 요청 전달시 사용된 컴포넌트 1/2

- org.springframework.web.servlet.HandlerMapping
  - HandlerMapping 구현체는 어떤 핸들러가 요청을 처리할지에 대한 정보를 알고 있다. (프레임워크 관점으로는 핸들러라고 말하지만 Controller 클래스를 의미한다.)
  - HandlerMapping을 구현하는 다양한 구현 클래스 중에서 가장 현대적인 방법으로 구현된 클래스는 RequestMappingHandlerMapping이다. 해당 클래스는 @RequestMapping에 정의된 설정 정보를 바탕으로 실행 할 핸들러를 선택한다.
- org.springframework.web.servlet.HandlerExecutionChain
  - HandlerExecutionChain 구현체는 실제로 호출된 핸들러에 대한 참조를 가지고 있다. 즉, 무엇이 실행되어야 될지 알고 있는 객체라고 말할 수 있으며, 핸들러 실행전과 실행후에 수행될 HandlerInterceptor도 참조하고 있다.

# 요청 전달시 사용된 컴포넌트 2/2

- org.springframework.web.servlet.HandlerAdapter
  - 실제 핸들러를 실행하는 역할을 담당한다.
  - 스프링 MVC는 비슷한 기능을 하는 다양한 구현 클래스를 제공하지만 `RequestMappingHandlerMapping`클래스에 의해 선택된 핸들러 메소드를 호출할때는 `RequestMappingHandlerAdapter`클래스를 사용한다.
  - `RequestMappingHandlerAdapter`클래스에는 핸들러 메소드에 매개변수를 전달하고 메소드의 처리결과를 반환값으로 되돌려 보내는 것과 같은 스프링 MVC에서 상당히 중요한 기능을 수행한다. 핸들러 메소드에 매개변수를 전달할 때는 요청받은 데이터를 자바 객체로 변환하고, 입력값이 올바른지 검사하는 것까지 한꺼번에 이뤄진다.

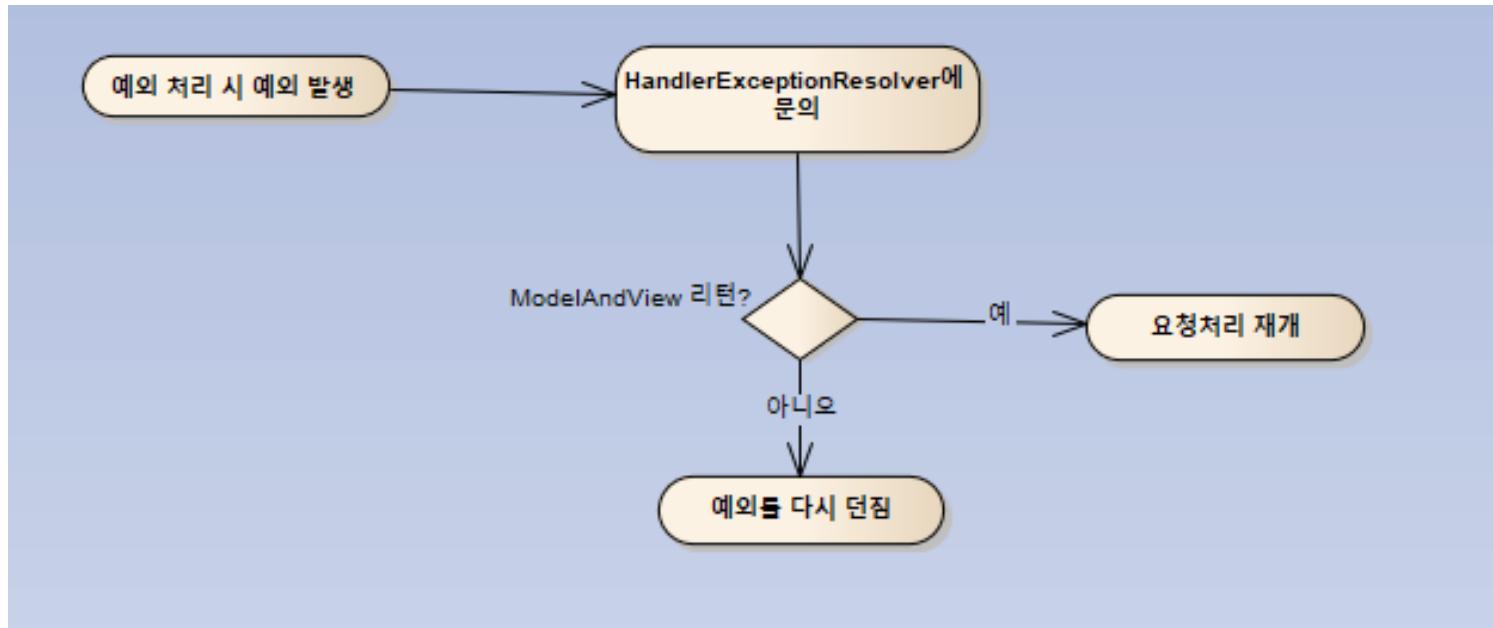
# DispatcherServlet 내부 동작흐름 상세 - 요청 처리



# 요청 처리시 사용된 컴포넌트

- **org.springframework.web.servlet.ModelAndView**
  - ModelAndView는 Controller의 처리 결과를 보여줄 view와 view에서 사용할 값을 전달하는 클래스이다.
- **org.springframework.web.servlet.RequestToViewNameTranslato  
r**
  - 컨트롤러에서 뷰 이름이나 뷰 오브젝트를 제공해주지 않았을 경우 URL과 같은 요청정보를 참고해서 자동으로 뷰 이름을 생성해주는 전략 오브젝트이다. 디폴트는 DefaultRequestToViewNameTranslator이다.

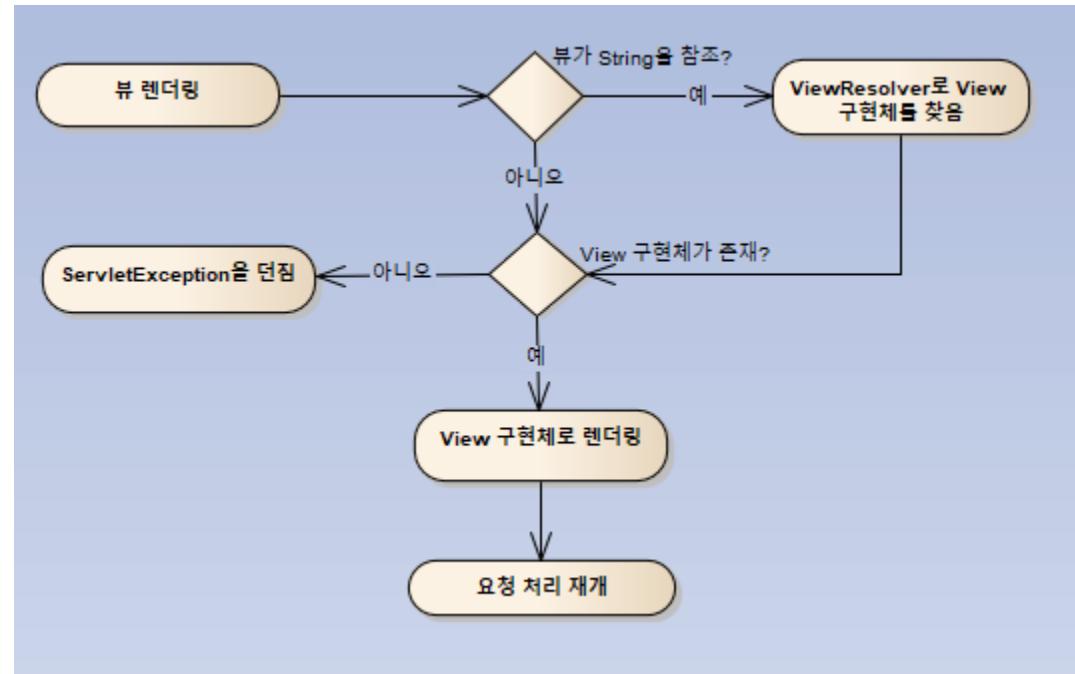
# DispatcherServlet 내부 동작흐름 상세 - 예외처리



# 예외 처리시 사용된 컴포넌트

- **org.springframework.web.servlet.HandlerExceptionResolver**
  - 기본적으로 DispatcherServlet이 DefaultHandlerExceptionResolver를 등록한다.
  - HandlerExceptionResolver는 예외가 던져졌을 때 어떤 핸들러를 실행 할 것인지에 대한 정보를 제공한다.

# DispatcherServlet 내부 동작흐름 상세 - 뷰 렌더링 과정

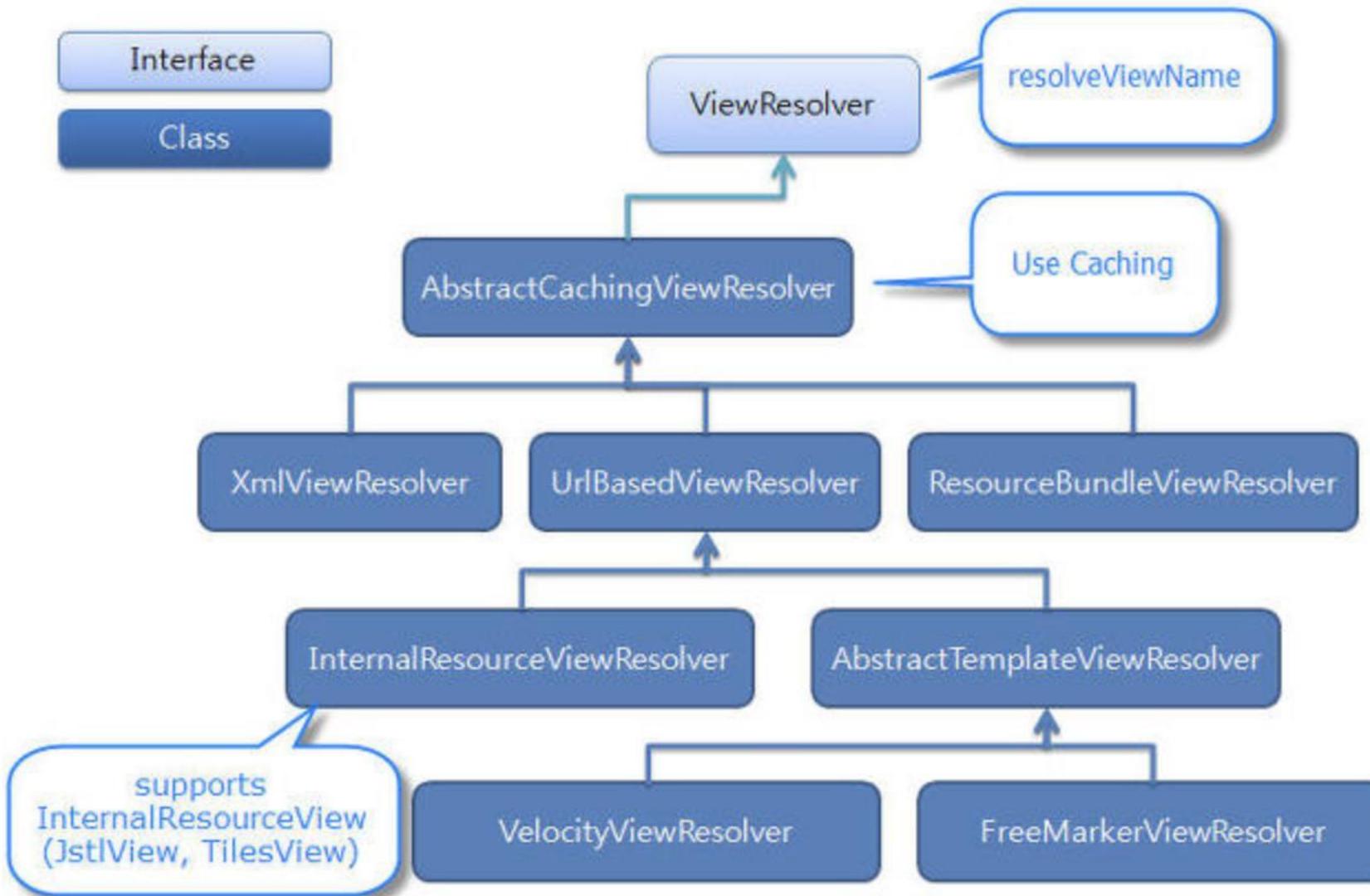


# 뷰 렌더링 과정시 사용된 컴포넌트 1/2

- org.springframework.web.servlet.**ViewResolver**
  - 컨트롤러가 리턴한 뷰 이름을 참고해서 적절한 뷰 오브젝트를 찾아주는 로직을 가진 전략 오프젝트이다. 뷰의 종류에 따라 적절한 뷰 리졸버를 추가로 설정해줄 수 있다.

# ViewResolver

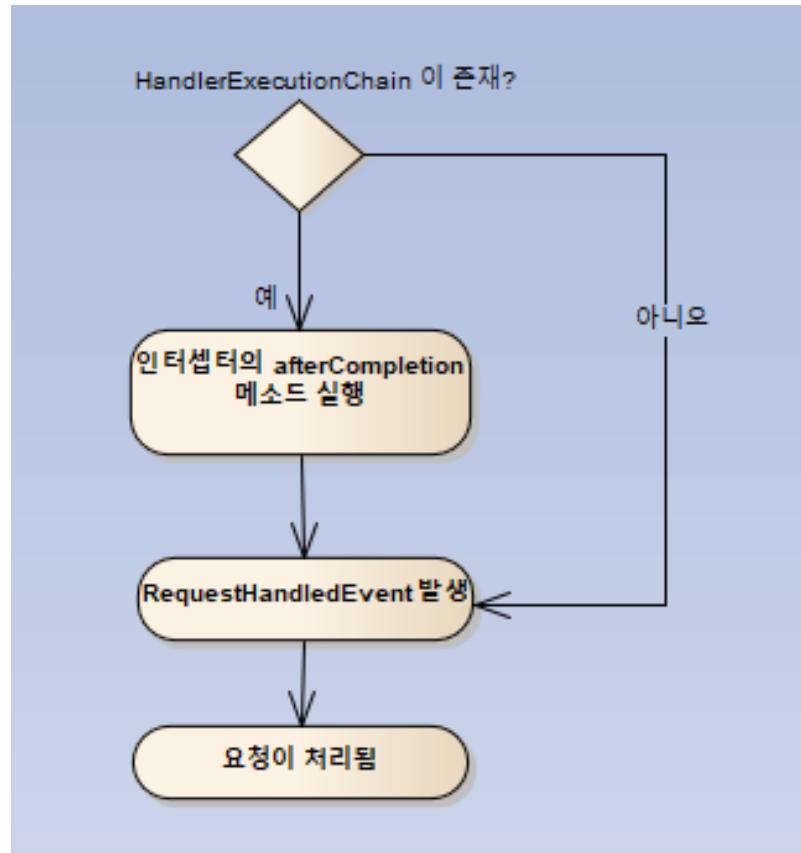
뷰 리졸버	설명
<b>BeannameViewResolver</b>	논리적 뷰 이름과 동일한 ID를 갖는 <bean>으로 등록된 View의 구현체를 찾는다.
<b>ContentNegotiatingViewResolver</b>	요청되는 콘텐츠 형식에 기반을 두어 선택한 하나 이상의 다른 뷰 리졸버에 위임한다.
<b>FreeMarkerViewResolver</b>	FreeMarker 기반의 템플릿을 찾는다. 경로는 논리적 뷰 이름에 접두어와 접미어를 붙여 구성
<b>InternalResourceViewResolver</b>	웹 어플리케이션의 WAR 파일 내에 포함된 뷰 템플릿을 찾는다. 뷰 템플릿의 경로는 논리적 뷰 이름에 접두어와 접미어를 붙여 구성
<b>JasperReportsViewResolver</b>	Jasper Reports 리포트 파일로 정의된 뷰를 찾는다. 경로는 논리적 뷰 이름에 접두어와 접미어를 붙여 구성
<b>TilesViewResolver</b>	Tiles 템플릿으로 정의된 뷰를 찾는다. 템플릿 이름은 논리적 뷰 이름에 접두어와 접미어를 붙여 구성
<b>UrlBasedViewResolver</b>	ViewResolver의 구현체로 특별한 맵핑 정보 없이 view 이름을 URL로 사용 View 이름과 실제 view 자원과의 이름이 같을 때 사용할 수 있다.
<b> ResourceBundleViewResolver</b>	ViewResolver의 구현체로 리소스 파일을 사용합니다. views.properties를 기본 리소스 파일로 사용합니다.
<b>VelocityLayoutViewResolver</b>	VelocityViewResolver의 서브클래스로, 스프링의 VelocityLayoutView를 통해 페이지 구성을 지원
<b>VelocityViewResolver</b>	Velocity 기반의 뷰를 찾는다. 경로는 논리적 뷰 이름에 접두어와 접미어를 붙여 구성
<b>XmlViewResolver</b>	ViewResolver의 구현체로 XML파일을 사용합니다. WEB-INF/views.xml을 기본 설정파일로 사용합니다.
<b>XsltViewResolver</b>	XSLT기반의 뷰를 찾는다. XSLT스타일시트의 경로는 논리적 뷰 이름에 접두어와 접미어를 붙여 구성



# 뷰 렌더링 과정시 사용된 컴포넌트 2/2

- org.springframework.web.servlet.View
- View 인터페이스는 클라이언트에 반환하는 응답 데이터를 생성하는 역할을 한다. 스프링 MVC는 다양한 구현 클래스를 제공한다. 주요 구현 클래스는 다음과 같다.
  - InternalResourceView : 템플릿 엔진으로 JSP를 이용할 때 사용하는 클래스
  - JstlView : 템플릿 엔진으로 JSP + JSTL을 이용할 때 사용하는 클래스

# DispatcherServlet 내부 동작흐름 상세 - 요청 처리 종료



# Controller(Handler) 클래스 작성하기

- @Controller 애노테이션을 클래스 위에 붙인다.
- 맵핑을 위해 @RequestMapping 애노테이션을 클래스나 메소드에서 사용한다.

# @RequestMapping

- Http 요청과 이를 다루기 위한 Controller 의 메소드를 연결하는 어노테이션
- Http Method 와 연결하는 방법
  - @RequestMapping("/users", method=RequestMethod.POST)
  - From Spring 4.3 version
    - @GetMapping
    - @PostMapping
    - @PutMapping
    - @DeleteMapping
    - @PatchMapping
- 클래스 레벨과 메소드 레벨 모두에 지정할 수 있다. 두 곳에서 동시에 같은 속성을 지정할 경우에는 다음과 같이 동작한다.
  - value(path), method, params, headers, name의 각 속성은 병합된 값이 적용된다.
  - consumes, produces의 각 속성은 메소드 레벨에 지정된 값으로 오버라이딩 된다.

# @RequestMapping에 지정 가능한 속성

- value : 요청 경로(또는 경로 패턴)을 지정한다.
- path : value속성의 별명을 지정한다.
- method : HTTP메소드 값(GET, POST, PUT등)을 지정한다.
- params : 요청 파라미터 유무나 파라미터 값을 지정한다.
- headers : 헤더 유무나 헤더 값을 지정한다.
- consumes : Content-Type 헤더 값(미디어 타입)을 지정한다.
- produces : Accept 헤더 값(미디어 타입)을 지정한다.
- name : 매팅 정보에 임의의 이름을 지정한다. 이 속성에 지정하는 값에 따라 매팅 룰이 바뀌는 것은 없다.

# @RequestMapping - 경로 패턴 사용

- 요청 경로에는 정적으로 표현된 구체적인 경로만이 아니라 동적으로 표현된 경로 패턴도 지정할 수 있다.
  - URI 템플릿 형식의 경로 패턴  
`/accounts/{accountId}`
  - URI 템플릿 형식의 경로 패턴 + 정규 표현식  
`/accounts/{accountId:[a-f0-9-]{36}}`
  - 앤트 스타일 경로 패턴  
`/**/accounts/me/email`

# @RequestMapping - 요청 파라미터 사용

- 요청 파라미터를 매팅 조건에 지정하는 경우에는 params속성을 사용한다. params속성에서 지원하는 지정 형식은 다음과 같다.
- name : 지정한 파라미터가 존재하는 경우에 매팅 대상이 된다.
- !name : 지정한 파라미터가 존재하지 않는 경우에 매팅 대상이 된다.
- name=value : 파라미터 값이 지정한 값에 해당하는 경우에 매팅 대상이 된다.
- name!=value : 파라미터 값이 지정한 값에 해당하지 않는 경우 매팅 대상이 된다.

# @RequestMapping – 요청 헤더 사용

- 요청 헤더를 매팅 조건으로 지정할 경우 headers속성을 사용한다. headers속성에서 지원하는 지정 형식은 params속성과 같다.

# @RequestMapping – Content-Type 헤더 사용

- 요청의 Content-Type 헤더 값을 매핑 조건으로 지정하는 경우에  
는 consume 속성을 사용한다. consume 속성에서 지원하는 지정  
형식은 다음과 같다.
- mediaType : 미디어 타입이 지정한 값인 경우 매핑 대상이 된다.
- !mediaType : 미디어 타입이 지정한 값이 아닌 경우 매핑 대상  
이 된다.

# @RequestMapping – Accept 헤더 사용

- 요청 Accept 헤더 값을 매팅 조건에 지정하는 경우 produces 속성을 사용한다. produces 속성에서 지원하는 지정 형식은 consumes와 같다.

# 컨트롤러 핸들러 메소드 매개변수 타입 1/4

- javax.servlet.ServletRequest
- **javax.servlet.http.HttpServletRequest**
- org.springframework.web.multipart.MultipartRequest
- org.springframework.web.multipart.MultipartHttpServletRequest
- javax.servlet.ServletResponse
- **javax.servlet.http.HttpServletResponse**
- **javax.servlet.http.HttpSession**
- org.springframework.web.context.request.WebRequest

# 컨트롤러 핸들러 메소드 매개변수 타입 2/4

- org.springframework.web.context.request.NativeWebRequest
- java.util.Locale
- java.io.InputStream
- java.io.Reader
- java.io.OutputStream
- java.io.Writer
- javax.security.Principal
- java.util.Map
- org.springframework.ui.Model
- org.springframework.ui.ModelMap

# 컨트롤러 핸들러 메소드 매개변수 타입 3/4

- org.springframework.web.multipart.MultipartFile
- javax.servlet.http.Part
- org.springframework.web.servlet.mvc.support.RedirectAttributes
- org.springframework.validation.Errors
- org.springframework.validation.BindingResult
- org.springframework.web.bind.support.SessionStatus
- org.springframework.web.util.UriComponentsBuilder
- org.springframework.http.HttpEntity<?>
- Command 또는 Form 객체

## 컨트롤러 핸들러 메소드 매개변수 타입 4/4

- @RequestParam
- @RequestHeader
- @RequestBody
- @RequestPart
- @ModelAttribute
- @PathVariable
- @CookieValue

# 인수에 지정 가능한 애노테이션

- **@PathVariable**
  - @RequestMapping 의 path 에 변수명을 입력받기 위한 place holder 가 필요함
  - place holder 의 이름과 PathVariable 의 name 값과 같으면 mapping 됨.
  - required 속성은 default true 임.
- **@MatrixVariable**
  - URL에서 매트릭스 변수 값을 가져오기 위한 애노테이션(기본 설정에서 는 사용할 수 없다.)

# 인수에 지정 가능한 애노테이션

- `@RequestParam`
  - Mapping 된 메소드의 Argument 에 붙일 수 있는 어노테이션
  - `@RequestParam`의 name에는 http parameter 의 name 과 맵핑
  - `@RequestParam`의 required는 필수인지 아닌지 판단.
- `@RequestHeader`
  - 요청정보의 헤더 정보를 읽어들 일 때 사용
  - `@RequestHeader(name="헤더명") String` 변수명

# 인수에 지정 가능한 애노테이션

- `@RequestBody`
  - 요청 본문 내용을 가져오기 위한 애노테이션. 요청 본문은 `HttpMessageConverter` 구조를 사용해 지정한 타입으로 변환한다.
- `@CookieValue`
  - 쿠키 값을 가져오기 위한 애노테이션

# 요청 데이터 이외의 정보를 인수로 지정할 수 있는 애노테이션

- **@ModelAttribute**
  - 모델에 저장된 객체를 인수로 받을 수 있다. 인수가 자바빈즈 형태라면 생략할 수 있다.
- **@Value**
  - '\${...}'와 같은 플레이스 홀더로 대체된 값이나 '#{...}'과 같은 SpEL식의 실행 결과를 인수로 받을 수 있다.
- 스프링 4.3부터 **@SessionAttribute**와 **@RequestAttribute**가 추가되어 HttpSession과 HttpServletRequest에 저장된 객체를 받아올 수 있다.

# 관례에 따른 목시적인 인수 값 결정

- 다음 규칙에 따라 요청 정보의 값을 인수의 값으로 채워 넣을 수 있다.
  - 인수의 타입이 String이나 Integer 같은 간단한 타입인 경우에는 인수의 이름과 일치하는 요청 파라미터에서 값을 가져올 수 있다.
  - 인수의 타입이 자바빈즈의 기본 속성명과 일치하는 객체를 Model에서 가져올 수 있다. 빈의 해당하는 객체가 Model에 없다면 기본 생성자를 호출해서 새로운 객체를 생성한다.

# 타입을 선택할 때 주의할 점

- 서블릿 API(HttpServletRequest, HttpServletResponse, HttpSession, Part)나 저수준 API(InputStream, OutputStream, Reader, Writer, Map)의 타입을 사용할 수도 있지만 이 API를 사용하면 애플리케이션에서 유지보수성을 떨어트릴 수 있다.

# 핸들러 메소드 리턴 값 1/2

- org.springframework.web.servlet.ModelAndView
  - 이동 대상의 뷰 이름과 이동 대상에 전달할 데이터를 반환한다.
- org.springframework.ui.Model
  - 이동 대상에 전달할 데이터를 반환한다.
- java.util.Map
- org.springframework.ui.ModelMap
- org.springframework.web.servlet.View

# 핸들러 메소드 리턴 값 2/2

- `java.lang.String`
  - 이동 대상의 뷰 이름을 반환한다.
- `void`
  - `HttpServletResponse`에 직접 응답 데이터를 쓰거나 `RequestToViewNameTranslator`의 매커니즘을 이용해 뷰 이름을 결정할 때 `void`을 사용한다.
- `org.springframework.http.HttpEntity<?>`
- `org.springframework.http.ResponseEntity<?>`
  - 응답 헤더와 응답 본문에 직렬화된 객체를 반환한다. 반환한 객체는 `HttpMessageConverter`매커니즘을 이용해 임의의 형식으로 직렬화된다.
- **기타 리턴 타입**
- 스프링 MVC는 서버 측에서 비동기 처리도 지원하고 있어 `Callable<?>`, `CompletableFuture<?>`(Java 8이상 사용가능), `DeferredResult<?>`, `WebAsyncTask<?>`, `ListenableFuture<?>`를 반환할 수 있다. 또한 HTTP 스트리밍을 지원하고 있어 `ResponseBodyEmitter`, `SseEmitter`, `StreamResponseBody`형을 반환할 수 있다.

# 핸들러 메소드 반환값을 위한 주요 애노테이션

- 메소드에 애노테이션을 지정하면 임의의 객체를 Model에 저장하거나 응답 본문에 직렬화 할 수 있다.
- @ModelAttribute : Model에 저장하는 객체를 반환한다.(반환값의 형이 자바빈즈의 경우에는 생략 가능)
- @ResponseBody : 응답 본문에 직렬화하는 객체를 반환한다. 객체는 HttpMessageConverter의 메커니즘을 이용해 임의의 형식으로 직렬화한다.

# 입력값 검사

- 스프링 MVC는 Bean Validation 기능을 이용해 요청 파라미터 값이 바인딩된 폼 클래스(또는 커맨드 클래스)의 입력값 검사를 한다.
- 스프링 MVC의 기본 동작에서는 폼 클래스에 대한 입력값 검사를 하지 않는다. 입력값 검사를 하기 위해서는 메소드 매개변수에 폼 클래스를 정의하고  
@org.springframework.validation.annotation.Validated 또는  
@javax.validation.Valid를 지정한다. @Validated를 사용하면 Bean Validation의 유효성 검증 그룹 메커니즘을 이용할 수 있다.

# 입력값 검사 결과의 판정

- 입력값 검사와 검사 결과(BindingResult)를 만드는 것은 프레임워크에서 해주지만 입력값 검사 결과에 대한 오류를 판단하고 그에 맞는 처리를 하는 것은 애플리케이션 측에서 구현해야 한다. 입력값 검사 후, 오류 정보를 확인하려면 BindingResult의 메소드를 사용하면 된다.

## 입력값 검사 - BindingResult에서 제공하는 오류 판단 메소드

- `hasErrors()` : 오류가 발생할 경우 `true`를 반환한다.
- `hasGlobalErrors()` : 객체 레벨의 오류가 발생한 경우 `true`로 발생한다.
- `hasFieldErrors()` : 필드 레벨의 오류가 발생한 경우 `true`를 반환한다.
- `hasFieldErrors(String)` : 인수에 지정한 필드에서 오류가 발생한 경우 `true`를 반환한다.

## 입력값 검사 - 미입력 처리

- 텍스트 필드에 값을 입력하지 않은 상태로 HTML폼을 전송하면 스프링 MVC는 폼 객체에 공백문자를 설정한다. '미입력은 허용하지만 만약 입력이 되었다면 최소한 6자 이상일 것'이라는 요구사항을 Bean Validation 표준 애노테이션으로는 충족시킬 수가 없다. 이럴 경우에는 스프링에서 제공하는 `org.springframework.beans.propertyeditors.StringTrimmerEditor`를 사용하는 것을 고려하는 것이 좋다. `StringTrimmerEditor`는 요청 파라미터 값을 trim하고 그 결과가 공백 문자인 경우에는 null로 변환시킨다.

# 입력값 검사 규칙 지정

- 입력값 검사 규칙은 Bean Validation이 제공하는 제약 애노테이션으로 설정한다. 검사 규칙은 크게 다음 세 가지로 분류할 수 있다.
  - Bean Validation 표준 제약 애노테이션
  - 서드파티에서 구현한 제약 애노테이션(보통 Hibernate Validator를 사용한다.)
  - 직접 구현한 제약 애노테이션

## 입력값 검사 – Bean Validation 표준 제약 애노테이션 1/2

- `@NotNull` : 필수 항목 검사
- `@Size` : 자리수 검사
  - `min` : 허용 최솟값을 지정한다.(기본값은 0)
  - `max` : 허용 최댓값을 지정한다.(기본값은 Integer.MAX\_VALUE)
- `@Pattern` : 문자 유형 검사
  - `regexp` : 정규 표현식의 패턴 문자열을 지정한다.
  - `flags` : 플래그(옵션)을 지정한다.

## 입력값 검사 – Bean Validation 표준 제약 애노테이션 2/2

- @Min, @Max : 수치의 범위를 검사할 때 사용한다.
- @DecimalMin, @DecimalMax : BigDecimal타입의 범위를 검사 할 때 사용한다.
- PropertyEditor와 @DateTimeFormat : 날짜/시간의 유효성 검사
- @AssertTrue와 @AssertFalse : 지정한 불린값(true or false)인지 검사
- 사용자 정의 유효성 검사기를 직접 구현해서 만드는 방법

# 화면이동 1/3

- 이동 대상을 지정하는 방법
  - 이동 대상을 지정할 때는 핸들러의 메소드가 뷰 이름(이동 대상에 할당된 논리적인 이름)을 반환하도록 만들면 된다. 일단 뷰 이름을 반환하면 스프링 MVC가 ViewResolver를 통해 논리적인 뷰 이름과 연결된 물리적인 뷰(예:jsp)가 어떤 것인지 판단하게 된다.
- 요청 경로로 리다이렉트
  - 다음 이동 대상이 리다이렉트해야 할 요청 경로라면 뷰 이름에 'redirect: + 리다이렉트할 요청 경로'를 지정한다.

# 화면이동 2/3

- 요청 파라미터 지정
  - 리다이렉트를 사용할 때 다음 이동 대상에 요청 파라미터를 전달해야 한다면 `org.springframework.web.servlet.mvc.support.RedirectAttribute`에 파라미터로 저장하면 된다.
- 경로 변수 지정
  - 리다이렉트할 URL을 동적으로 만들어야 할 때는 URL에 경로 변수를 추가하고, 경로 변수에 들어갈 값은 `RedirectAttributes`에 저장하면 된다.

# 화면이동 3/3

- 요청 경로로 포워드
  - 다음 이동 대상이 리다이렉트해야 할 요청 경로라면 뷰 이름에 'forward: + 전송 대상의 요청 경로'를 지정한다.
- 뷰와의 데이터 연계
  - 뷰 처리에 필요한 데이터(자바 객체)는 Model에 저장해야 연계가 된다. Model에 자바 객체를 저장하면 스프링 MVC가 뷰에서 접근할 수 있는 영역 (JSP라면 HttpServletRequest)에 자바 객체를 익스포트해준다.
  - 자바 객체를 Model에 저장하는 방법은 다음의 두 가지가 있다.
    - Model API를 직접 호출한다.
    - ModelAttribute 애노테이션이 붙은 메소드를 준비한다.

# 웹 페이지는 중복 개발되는 요소가 존재한다.

NAVER 지식iN

이용안내 보기 | 로그인

홈 답변하기 Q&A 오픈사전 사람들 베스트 프로필 지식iN전당 오늘의질문 교육기부 장학기부 파트너센터 Q 질문하기

오늘의질문

혼자이고 싶다는 생각이 드는 순간은?

지식iN 공식블로그 소개  
지식iN 공식 블로그에서  
새로운 소식을 만나보세요!  
자식남자 이웃추가도  
꼭 해주실거죠?

봉어빵 vs 잉어빵

전문가들에게 질문해 보세요!

의사 한의사 변호사 노무사 수의사 약사 세무사

지식iN 통계 19시 기준

2월 7일 컴퓨터통신 분야에서는 4천명의 지식iN들이 답변하셨습니다.

1 컴퓨터통신 2 게임 3 엔터테인먼트, 예술

19% 81%

10대 20대 30대 40대 50대 남자 여자

많이 본 Q&A 07월 18시 기준

1 강수지 첫번째남편  
강수지 첫번째남편은 누구?  
조회수 3658 | 답변수 1

2 나 자신에게 상을 하나 줄 수 있다면?  
출석률이 원복한 사람에게 주는 개근상, 성적이 우수...  
조회수 1055 | 답변수 1871

3 노후 경유차 운행제한 질문  
제 차는 2005년 3월에 구매한 쏘렌토(2005) 경유차입니다.  
차량등록증을 보니 2560kg이라고 되어있네요  
노후 경유자 기사를 접해보면 2005년 이전 2.5톤 경유자는 2017~2020년 서울, 인천, 경기에서 탈수 없다고 하던데요

그럼 제 차의 경우 2021년부터 운행을 못한다는 얘기인가요?  
조심히 타서 앞으로 10년은 더 탈수 있을거 같은데....  
혹시 매연저감장치 설치시 계속탈수는 있는지 알고싶습니다.

활동소식이 궁금하다면  
NAVER 로그인

아이디 찾기 | 비밀번호 찾기 회원가입

2018 GLOBAL V LIVE TOP10 MONSTA X 2/8 오후 10시 V LIVE

NAVER 지식iN

이용안내 보기 | 로그인

홈 답변하기 Q&A 오픈사전 사람들 베스트 프로필 지식iN전당 오늘의질문 교육기부 장학기부 파트너센터 Q 질문하기

Q&A > 생활 > 자동차 > 자동차 구조, 역사

2월 강원도가 둘씩인다

노후 경유차 운행제한 질문

비공개 | 질문 25건 | 질문마감률 100% | 질문채택률 95.4% | 2016.08.20. 20:50 | 조회수 36,718

제 차는 2005년 3월에 구매한 쏘렌토(2005) 경유차입니다  
차량등록증을 보니 2560kg이라고 되어있네요  
노후 경유자 기사를 접해보면 2005년 이전 2.5톤 경유자는 2017~2020년 서울, 인천, 경기에서 탈수 없다고 하던데요

그럼 제 차의 경우 2021년부터 운행을 못한다는 얘기인가요?  
조심히 타서 앞으로 10년은 더 탈수 있을거 같은데....  
혹시 매연저감장치 설치시 계속탈수는 있는지 알고싶습니다.

답변을 기다리는 질문

1. 질문자 채택된 경우, 추가 답변 등록이 불가합니다.

많이 본 Q&A 07월 18시 기준

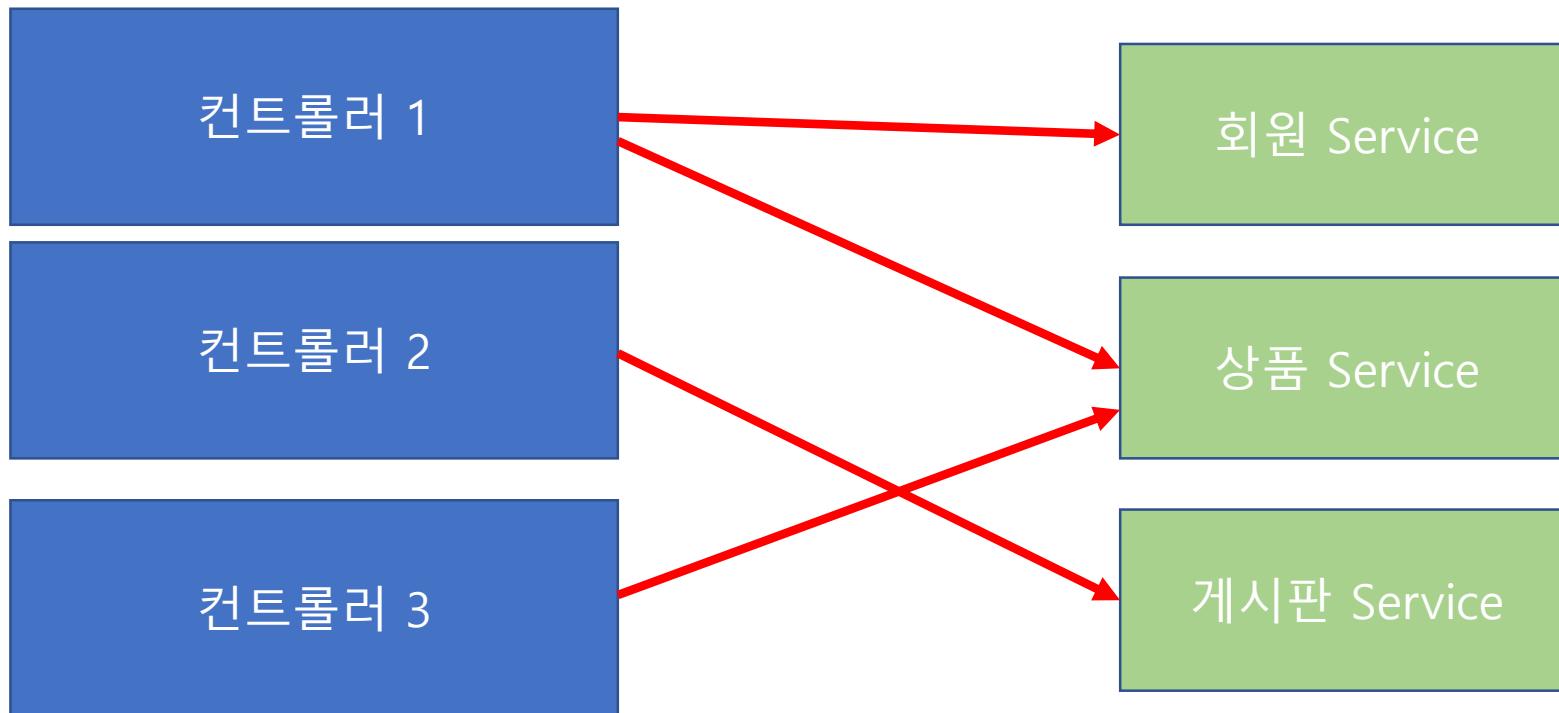
1 강수지 첫번째남편  
조회 2650 | 답변 1

# Controller에서 중복되는 부분을 처리하려면?

- 별도의 객체로 분리한다.
- 별도의 메소드로 분리한다.
- 예를 들어 쇼핑몰에서 게시판에서도 회원 정보를 보여주고, 상품 목록 보기에서도 회원 정보를 보여줘야 한다면 회원 정보를 읽어오는 코드는 어떻게 해야할까?

# 컨트롤러와 서비스

- 비지니스 메소드를 별도의 Service 객체에서 구현하도록 하고 컨트롤러는 Service 객체를 사용하도록 한다.



# 서비스(Service) 객체란?

- 비지니스 로직(Business logic)을 수행하는 메소드를 가지고 있는 객체를 서비스 객체라고 한다.
- 보통 하나의 비지니스 로직은 하나의 트랜잭션으로 동작한다.

# 트랜잭션(Transaction)이란?

- 트랜잭션은 하나의 논리적인 작업을 의미한다.
- 트랜잭션의 특징은 크게 4가지로 구분된다.
  - 원자성 (Atomicity)
  - 일관성 (Consistency)
  - 독립성 (Isolation)
  - 지속성 (Durability)

# 원자성 (Atomicity)

- 전체가 성공하거나 전체가 실패하는 것을 의미한다.
- 예를 들어 "출금"이라는 기능의 흐름이 다음과 같다고 생각해 보자.
  1. 잔액이 얼마인지 조회한다.
  2. 출금하려는 금액이 잔액보다 작은지 검사한다.
  3. 출금하려는 금액이 잔액보다 작다면 (잔액 - 출금액)으로 수정한다.
  4. 언제, 어디서 출금했는지 정보를 기록한다.
  5. 사용자에게 출금한다.
- 위의 작업이 4번에서 오류가 발생했다면 어떻게 될까? 4번에서 오류가 발생했다면, 앞의 작업들을 모두 원래대로 복원을 시켜야 한다. 이를 rollback이라고 한다. 5번까지 모두 성공했을 때만 정보를 모두 반영해야 한다. 이를 commit한다고 한다. 이렇게 rollback하거나 commit을 하게 되면 하나의 트랜잭션 처리가 완료된다.

# 일관성 (Consistency)

- 일관성은 트랜잭션의 작업 처리 결과가 항상 일관성이 있어야 한다는 것이다. 트랜잭션이 진행되는 동안에 데이터가 변경 되더라도 업데이트된 데이터로 트랜잭션이 진행되는것이 아니라, 처음에 트랜잭션을 진행 하기 위해 참조한 데이터로 진행된다. 이렇게 함으로써 각 사용자는 일관성 있는 데이터를 볼 수 있는 것이다.

# 독립성 (Isolation)

- 독립성은 둘 이상의 트랜잭션이 동시에 병행 실행되고 있을 경우에 어느 하나의 트랜잭션이라도 다른 트랜잭션의 연산을 끼어들 수 없다. 하나의 특정 트랜잭션이 완료될 때까지, 다른 트랜잭션이 특정 트랜잭션의 결과를 참조할 수 없다.

# 지속성 (Durability)

- 지속성은 트랜잭션이 성공적으로 완료되었을 경우, 결과는 영구적으로 반영되어야 한다는 점이다.

# JDBC 프로그래밍에서 트랜잭션 처리 방법

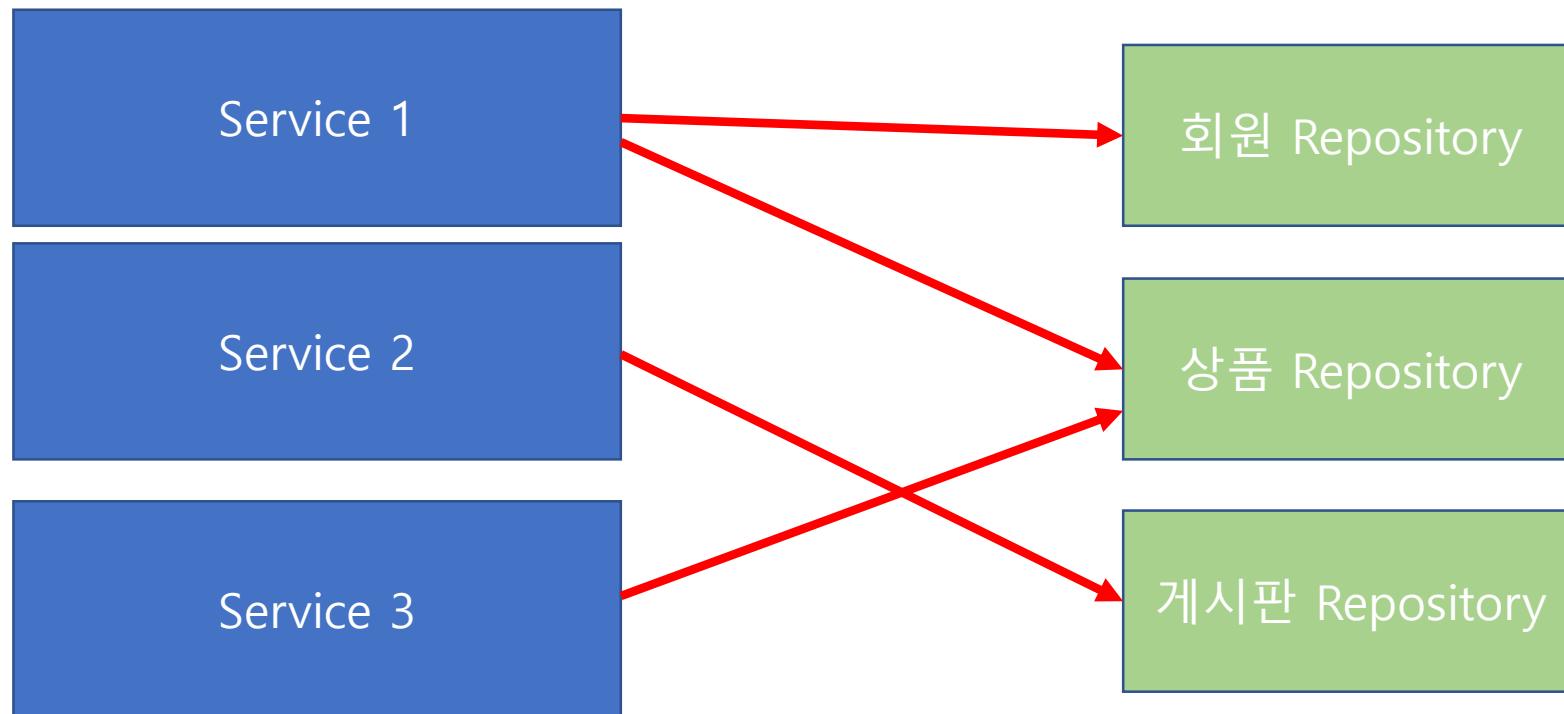
- DB에 연결된 후 Connection객체의 setAutoCommit메소드에 false를 파라미터로 지정한다.
- 입력,수정,삭제 SQL이 실행을 한 후 모두 성공했을 경우 Connection이 가지고 있는 commit()메소드를 호출한다.

# @EnableTransactionManagement

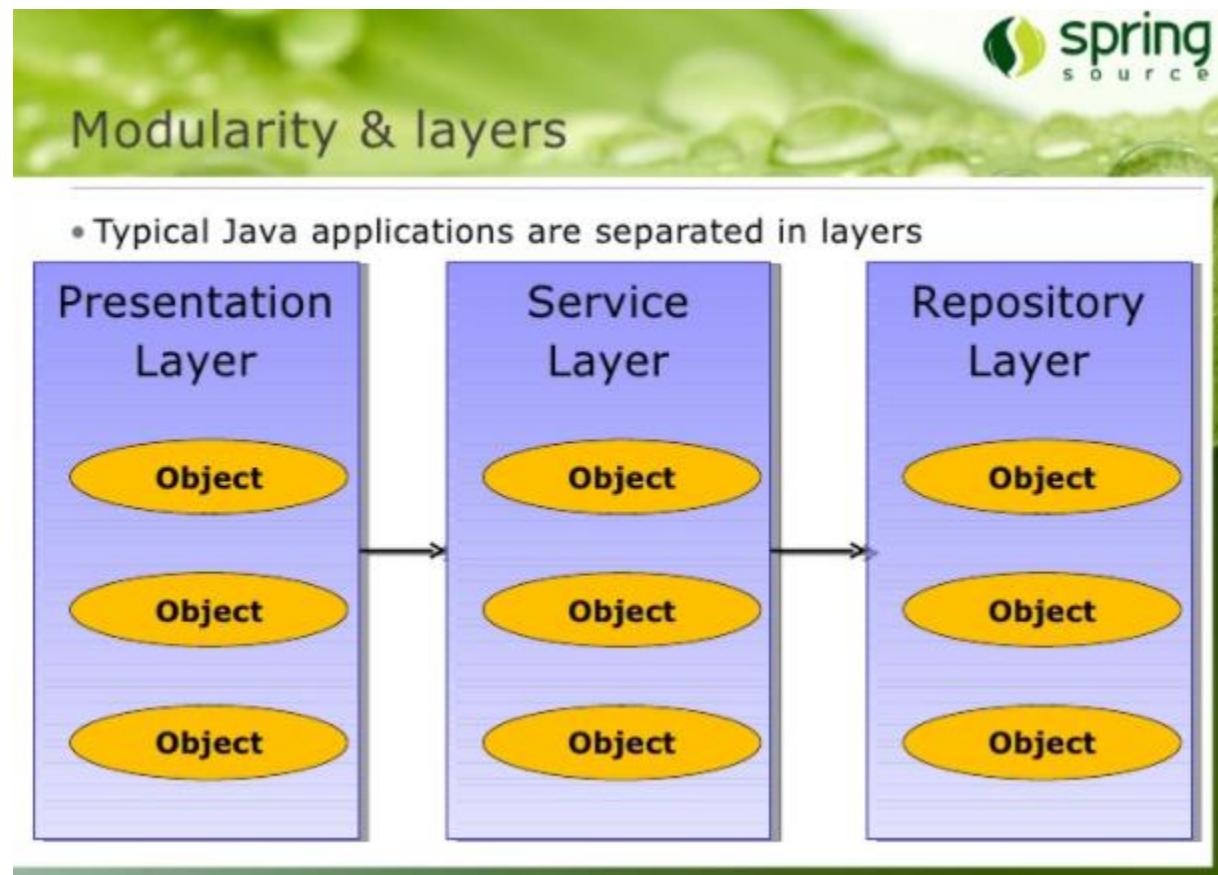
- Spring Java Config파일에서 트랜잭션을 활성화 할 때 사용하는 애노테이션
- Java Config를 사용하게 되면 PlatformTransactionManager 구현체를 모두 찾아서 그 중에 하나를 매핑해 사용한다.
- 특정 트랜잭션 메니저를 사용하고자 한다면 TransactionManagementConfigurer 를 Java Config파일에서 구현하고 원하는 트랜잭션 메니저를 리턴하도록 한다.
- 아니면, 특정 트랜잭션 메니저 객체를 생성시 @Primary 애노테이션을 지정한다.

# 서비스 객체에서 중복으로 호출되는 코드의 처리

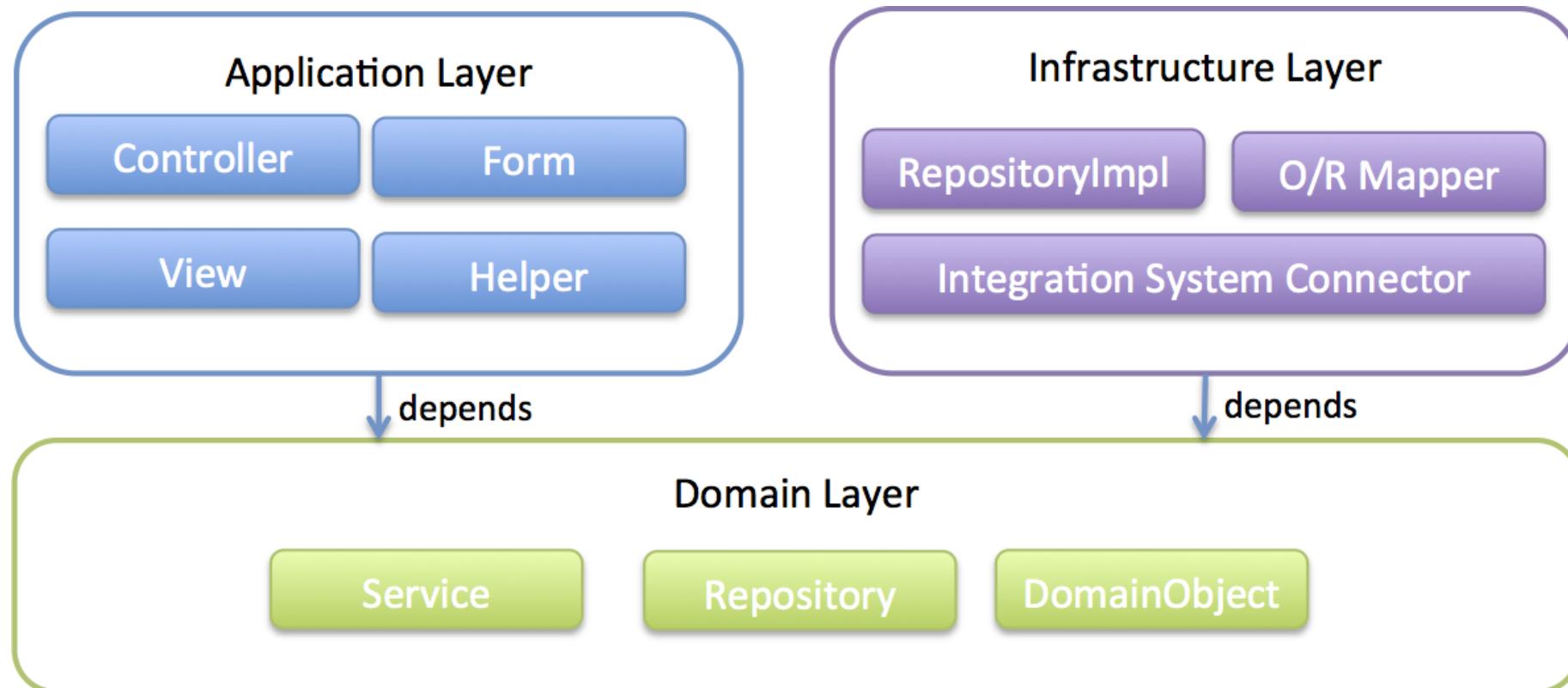
- 데이터 액세스 메소드를 별도의 Repository(Dao) 객체에서 구현하도록 하고 서비스는 Repository 객체를 사용하도록 한다.



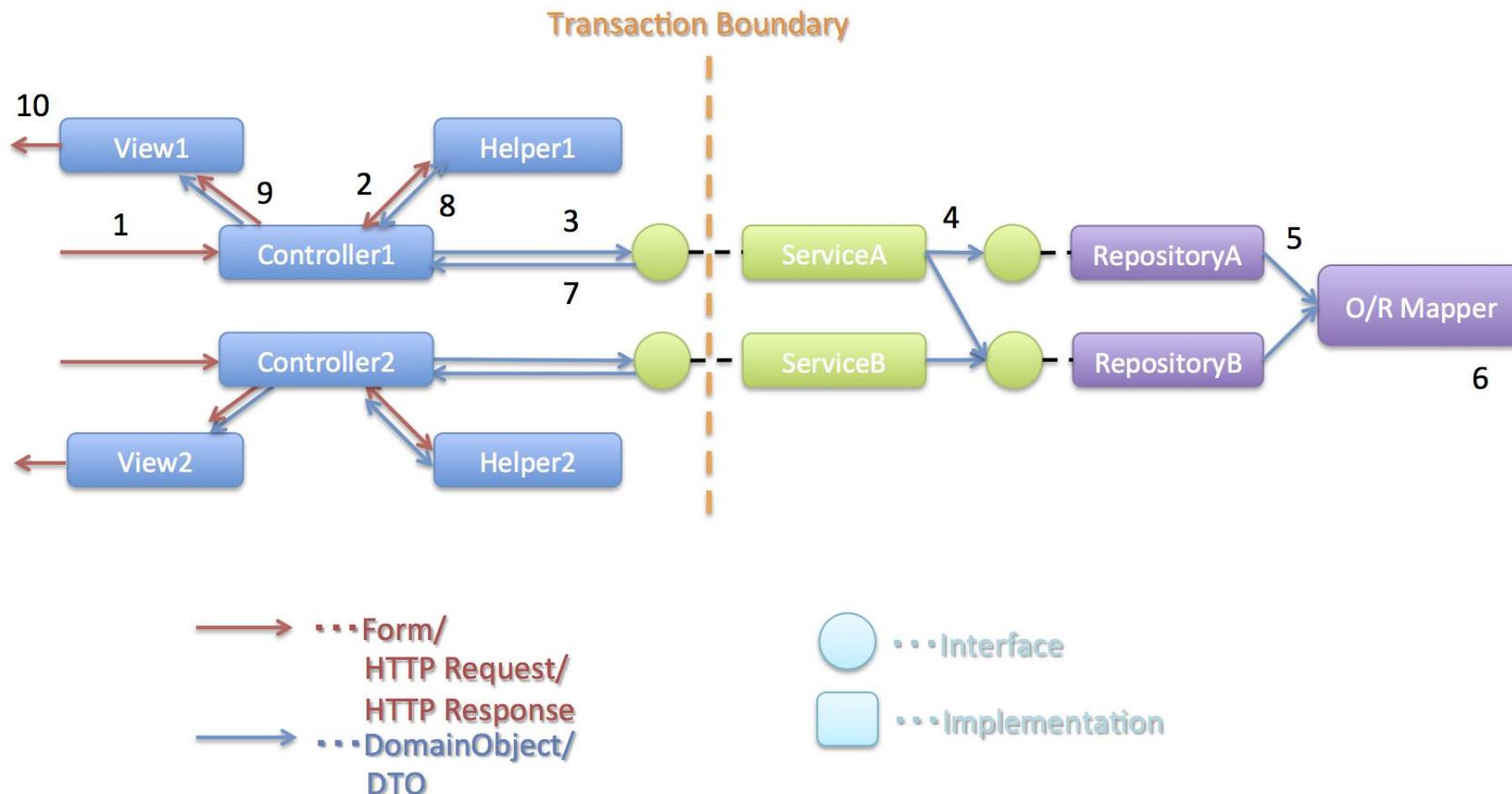
# 레이어드 아키텍처 1/3



# 레이어드 아키텍처 2/3



# 레이어드 아키텍처 3/3



# 설정의 분리

- Spring 설정 파일을 프리젠테이션 레이어쪽과 나머지를 분리할 수 있다.
- web.xml 파일에서 프리젠테이션 레이어에 대한 스프링 설정은 DispatcherServlet이 읽도록 하고, 그 외의 설정은 ContextLoaderListener를 통해서 읽도록 한다.
- DispatcherServlet을 경우에 따라서 2개 이상 설정 할 수 있는데 이 경우에는 각각의 DispatcherServlet의 ApplicationContext 가 각각 독립적이기 때문에 각각의 설정 파일에서 생성한 빈을 서로 사용할 수 없다.
- 위의 경우와 같이 동시에 필요한 빈은 ContextLoaderListener를 사용함으로써 공통으로 사용하게 할 수 있다.
- ContextLoaderListener와 DispatcherServlet은 각각 ApplicationContext 를 생성하는데, ContextLoaderListener가 생성하는 ApplicationContext가 root컨텍스트가 되고 DispatcherServlet이 생성한 인스턴스는 root컨텍스트를 부모로 하는 자식 컨텍스트가 된다. 참고로, 자식 컨텍스트들은 root컨텍스트의 설정 빈을 사용할 수 있다.

# @RestController

- Spring 4 에서 Rest API 또는 Web API를 개발하기 위해 등장한 애노테이션
- 이전 버전의 @Controller와 @ResponseBody를 포함한다.

# MessageConvertor

- 자바 객체와 HTTP 요청 / 응답 바디를 변환하는 역할
- @ResponseBody, @RequestBody
- @EnableWebMvc 로 인한 기본 설정.
  - WebMvcConfigurationSupport 를 사용하여 Spring MVC 구현을 하고 있음.
  - Default MessageConvertor 를 제공하고 있음.
  - <https://github.com/spring-projects/spring-framework/blob/master/spring-webmvc/src/main/java/org/springframework/web/servlet/config/annotation/WebMvcConfigurationSupport.java>의 addDefaultHttpMessageConverters메소드 항목 참조

# MessageConverter 종류

MessageConverter 종류	기능
ByteArrayHttpMessageConverter	converts byte arrays
StringHttpMessageConverter	converts Strings
ResourceHttpMessageConverter	converts org.springframework.core.io.Resource for any type of octet stream
SourceHttpMessageConverter	converts javax.xml.transform.Source
FormHttpMessageConverter	converts form data to/from a MultiValueMap<String, String>.
Jaxb2RootElementHttpMessageConverter	converts Java objects to/from XML (added only if JAXB2 is present on the classpath)
MappingJackson2HttpMessageConverter	converts JSON (added only if Jackson 2 is present on the classpath)
MappingJacksonHttpMessageConverter	converts JSON (added only if Jackson is present on the classpath)
AtomFeedHttpMessageConverter	converts Atom feeds (added only if Rome is present on the classpath)
RssChannelHttpMessageConverter	converts RSS feeds (added only if Rome is present on the classpath)

# json 응답하기

- 컨트롤러의 메소드에서는 json으로 변환될 객체를 반환한다.
- jackson라이브러리를 추가할 경우 객체를 json으로 변환하는 메시지 컨버터가 사용되도록 @EnableWebMvc에서 기본으로 설정되어 있다.
- jackson라이브러리를 추가하지 않으면 json메시지로 변환할 수 없어 500오류가 발생한다.
- 사용자가 임의의 메시지 컨버터(MessageConverter)를 사용하도록 하려면 WebMvcConfigurerAdapter의 configureMessageConverters메소드를 오버라이딩하도록 한다.

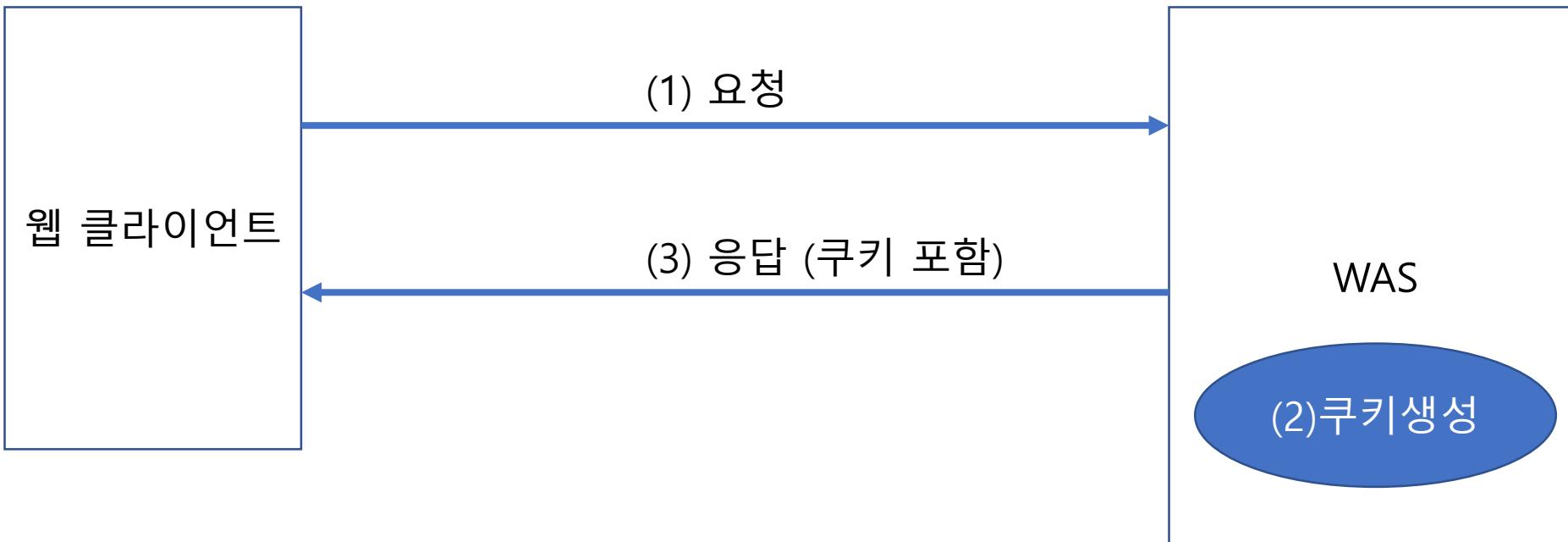
# 웹에서의 상태 유지 기술

- http프로토콜은 상태 유지가 안되는 프로토콜이다.
  - 이전에 무엇을 했고, 지금 무엇을 했는지에 대한 정보를 갖고 있지 않음
  - 웹 브라우저(클라이언트)의 요청에 대한 응답을 하고 나면 해당 클라이언트와의 연결을 지속하지 않음.
- 상태 유지를 위해 Cookie와 Session기술이 등장함.

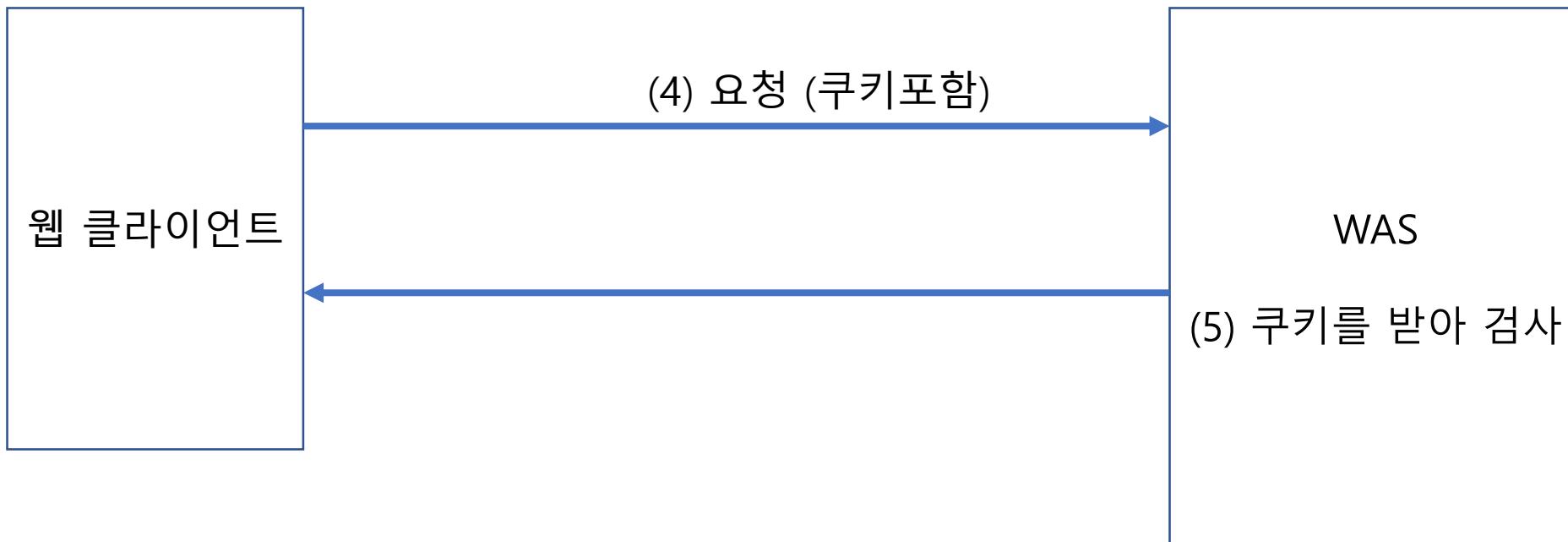
# 쿠키(Cookie)와 세션(Session)

- 쿠키
  - 사용자 컴퓨터에 저장
  - 저장된 정보를 다른 사람 또는 시스템이 볼 수 있는 단점
  - 유효시간이 지나면 사라짐
- 세션
  - 서버에 저장
  - 서버가 종료되거나 유효시간이 지나면 사라짐

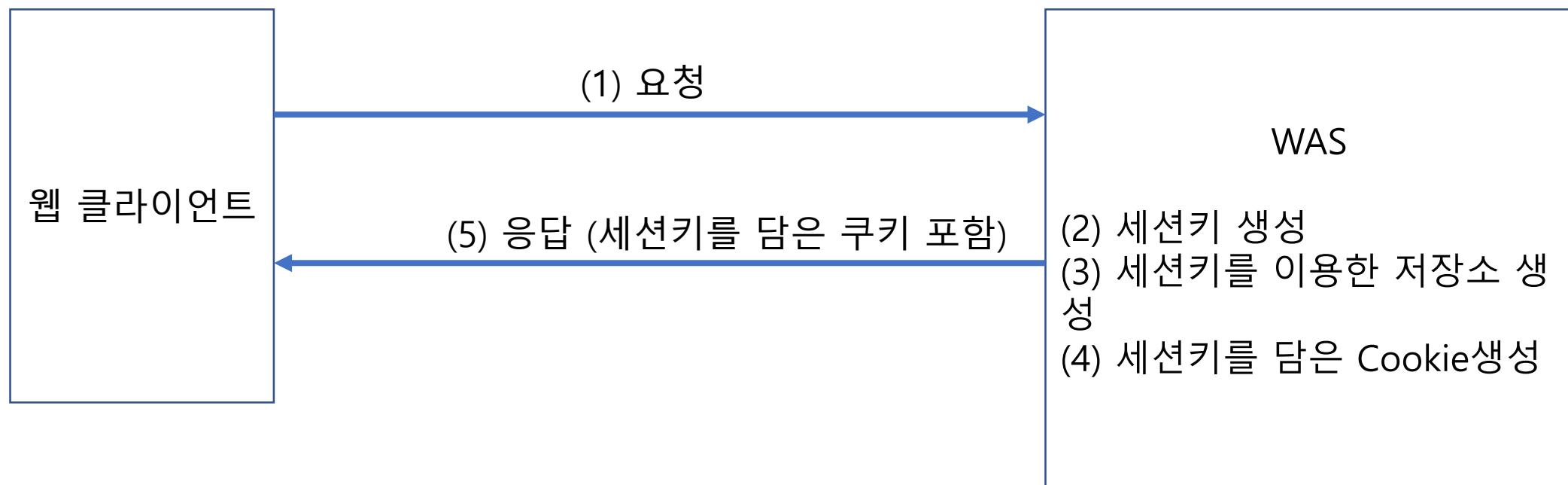
# 쿠키(Cookie) 동작 이해 1/2



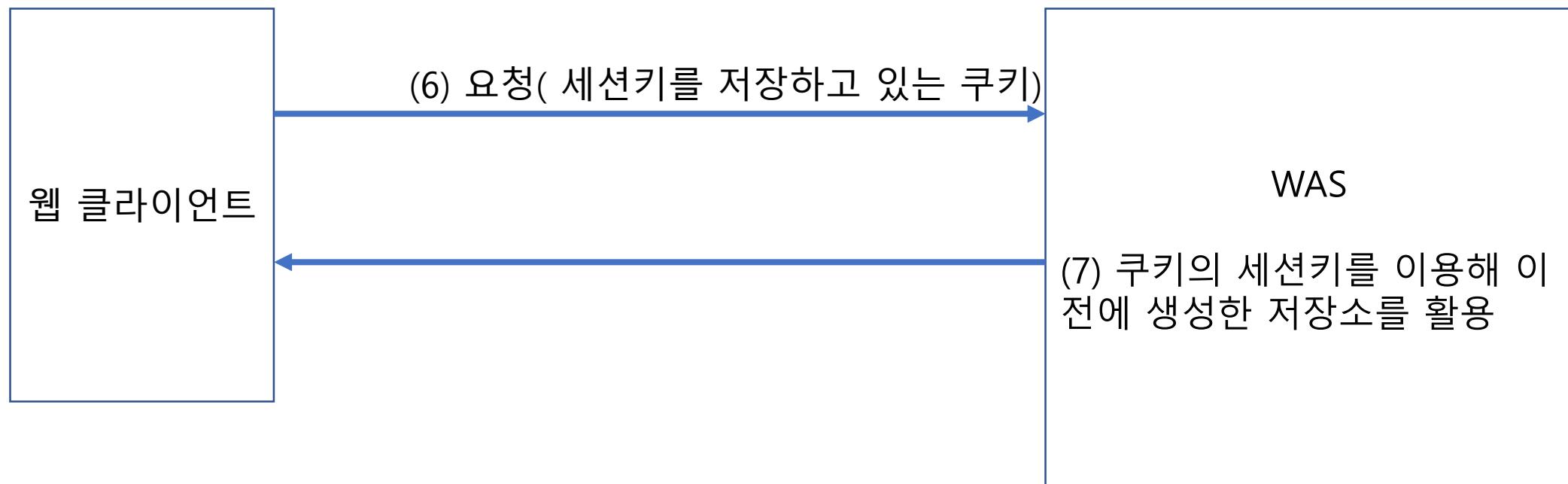
# 쿠키(Cookie) 동작 이해 2/2



# 세션의 동작 이해 1/2



# 세션의 동작 이해 2/2



# 쿠키 정의

- 정의
  - 클라이언트 단에 저장되는 작은 정보의 단위.
  - 클라이언트에서 생성하고 저장될 수 있고, 서버단에서 전송한 쿠키가 클라이언트에 저장될 수 있다.
- 이용 방법
  - 서버에서 클라이언트의 브라우저로 전송되어 사용자의 컴퓨터에 저장
  - 저장된 쿠키는 다시 해당하는 웹 페이지에 접속 할 때, 브라우저에서 서버로 쿠키를 전송
  - 쿠키는 이름(name)과 값(value)으로 구성된 자료를 저장
    - 이름과 값 외에도 주석(comment), 경로(path), 유효기간(maxage, expiry), 버전(version), 도메인(domain)과 같은 추가적인 정보를 저장

# 쿠키 정의

- 쿠키는 그 수와 크기에 제한
  - 하나의 쿠키는 4K Byte 크기로 제한
  - 브라우저는 각각의 웹사이트 당 20개의 쿠키를 허용
  - 모든 웹 사이트를 합쳐 최대 300개를 허용
  - 그러므로 클라이언트 당 쿠키의 최대 용량은 1.2M Byte

# javax.servlet.http.Cookie

- 서버에서 쿠키 생성, Response의 addCookie메소드를 이용해 클라이언트에게 전송

```
Cookie cookie = new Cookie(이름, 값);
```

```
response.addCookie(cookie);
```

- 쿠키는 (이름, 값)의 쌍 정보를 입력하여 생성
- 쿠키의 이름은 알파벳과 숫자로만 구성되고, 쿠키 값은 공백, 괄호, 등호, 콤마, 콜론, 세미콜론 등은 포함 불가능

# javax.servlet.http.Cookie

- 클라이언트가 보낸 쿠키 정보 읽기

```
Cookie[] cookies = request.getCookies();
```

- 쿠키 값이 없으면 null이 반환된다.
- Cookie가 가지고 있는 getName()과 getValue()메소드를 이용해 서 원하는 쿠키정보를 찾아 사용한다.

# javax.servlet.http.Cookie

- 클라이언트에게 쿠키 삭제 요청
  - 쿠키를 삭제하는 명령은 없고, maxAge가 0인 같은 이름의 쿠키를 전송한다.

```
Cookie cookie = new Cookie("이름", null);
cookie.setMaxAge(0);
response.addCookie(cookie);
```

# javax.servlet.http.Cookie

- 쿠키의 유효기간 설정
  - 메소드 `setMaxAge()`
    - 인자는 유효기간을 나타내는 초 단위의 정수형
    - 만일 유효기간을 0으로 지정하면 쿠키의 삭제
    - 음수를 지정하면 브라우저가 종료될 때 쿠키가 삭제
  - 유효기간을 10분으로 지정하려면
    - `cookie.setMaxAge(10 * 60); //초 단위 : 10분`
    - 1주일로 지정하려면  $(7*24*60*60)$ 로 설정한다.

# javax.servlet.http.Cookie

반환형	메소드 이름	메소드 기능
int	getMaxAge()	쿠키의 최대지속 시간을 초단위로 지정 -1 일 경우 브라우저가 종료되면 쿠키를 만료
String	getName()	쿠키의 이름을 스트링으로 반환
String	getValue()	쿠키의 값을 스트링으로 반환
void	setValue(String newValue)	쿠키에 새로운 값을 설정할 때 사용

# Spring MVC에서의 Cookie 사용

- @CookieValue 애노테이션 사용
  - 컨트롤러 메소드의 파라미터에서 CookieValue애노테이션을 사용함으로써 원하는 쿠키정보를 파라미터 변수에 담아 사용할 수 있다.

컨트롤러메소드(@CookieValue(value="쿠키이름", required=false, defaultValue="기본값") String 변수명)

# 세션 정의

- 정의
  - 클라이언트 별로 서버에 저장되는 정보
- 이용 방법
  - 웹 클라이언트가 서버측에 요청을 보내게 되면 서버는 클라이언트를 식별하는 session id를 생성
  - 서버는 session id를 이용해서 key와 value를 이용한 저장소인 HttpSession을 생성
  - 서버는 session id를 저장하고 있는 쿠키를 생성하여 클라이언트에 전송
  - 클라이언트는 서버측에 요청을 보낼때 session id를 가지고 있는 쿠키를 전송
  - 서버는 쿠키에 있는 session id를 이용해서 그 전 요청에서 생성한 HttpSession을 찾고 사용한다

# 세션 생성 및 얻기

```
HttpSession session = request.getSession();
```

```
HttpSession session = request.getSession(true);
```

- request의 getSession()메소드는 서버에 생성된 세션이 있다면 세션을 반환하고 없다면 새롭게 세션을 생성하여 반환한다. 새롭게 생성된 세션인지는 HttpSession이 가지고 있는 isNew()메소드를 통해 알 수 있다.

```
HttpSession session = request.getSession(false);
```

- request의 getSession()메소드에 파라미터로 false를 전달하면, 이미 생성된 세션이 있다면 반환하고 없으면 null을 반환한다.

# 세션에 값 저장

- `setAttribute(String name, Object value)`
  - name과 value의 쌍으로 객체 Object를 저장하는 메소드
  - 세션이 유지되는 동안 저장할 자료를 저장

`session.setAttribute(이름, 값)`

# 세션에 값 조회

- `getAttribute(String name)` 메소드
  - 세션에 저장된 자료는 다시 `getAttribute(String name)` 메소드를 이용해 조회
  - 반환 값은 Object 유형이므로 저장된 객체로 자료유형 변환이 필요
  - 메소드 `setAttribute()`에 이용한 name인 "id"를 알고 있다면 바로 다음과 같이 바로 조회

```
String value = (String) session.getAttribute("id");
```

# javax.servlet.http.HttpSession

반환형	메소드 이름	메소드 기능
long	getCreationTime()	를 세션이 생성된 시간까지 지난 시간을 계산하여 밀리세컨드로 반환
String	getId()	세션에 할당된 유일한 식별자(ID)를 String 타입으로 반환
int	getMaxInactiveInterval()	현재 생성된 세션을 유지하기 위해 설정된 최대 시간을 초의 정수형으로 반환, 지정하지 않으면 기본 값은 1800초, 즉 30분이며, 기본값도 서버에서 설정 가능

# 세션에 값 삭제

- `removeAttribute(String name)` 메소드
  - `name`값에 해당하는 세션 정보를 삭제한다.
- `invalidate()` 메소드
  - 모든 세션 정보를 삭제한다.

# javax.servlet.http.HttpSession

반환형	메소드 이름	메소드 기능
Object	getAttribute(String name)	name이란 이름에 해당되는 속성값을 Object 타입으로 반환, 해당되는 이름이 없을 경우에는 null을 반환
Enumeration	getAttributeNames()	속성의 이름들을 Enumeration 타입으로 반환
void	invalidate()	현재 생성된 세션을 무효화 시킴
void	removeAttribute(String name)	name으로 지정한 속성의 값을 제거
void	setAttribute(String name, Object value)	name으로 지정한 이름에 value 값을 할당
void	setMaxInactiveInterval(int interval)	세션의 최대 유지시간을 초 단위로 설정
boolean	isNew()	세션이 새로이 만들어졌으면 true, 이미 만들어진 세션이면 false를 반환

# 아규먼트 리졸버 작성방법 1/2

- org.springframework.web.method.support.HandlerMethodArgumentResolver 를 구현한 클래스를 작성한다.
- supportsParameter메소드를 오버라이딩 한 후, 원하는 타입의 인자가 있는지 검사한 후 있을 경우 true가 리턴되도록 한다.
- resolveArgument메소드를 오버라이딩 한 후, 메소드의 인자로 전달할 값을 리턴한다.

# javax.servlet.http.HttpSession

- 세션은 클라이언트가 서버에 접속하는 순간 생성
  - 특별히 지정하지 않으면 세션의 유지 시간은 기본 값으로 30분 설정
  - 세션의 유지 시간이란 서버에 접속한 후 서버에 요청을 하지 않는 최대 시간
  - 30분 이상 서버에 전혀 반응을 보이지 않으면 세션이 자동으로 끊어짐.
  - 이 세션 유지 시간은 web.xml파일에서 설정 가능

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

# @SessionAttributes & @ModelAttribute

- @SessionAttributes파라미터로 지정된 이름과 같은 이름이 @ModelAttribute에 지정되어 있을 경우 메소드가 반환되는 값은 세션에 저장된다.
- 아래의 예제는 세션에 값을 초기화하는 목적으로 사용되었다.

```
@SessionAttributes("user")  
  
public class LoginController {  
  
    @ModelAttribute("user")  
  
    public User setUpUserForm() {  
  
        return new User();  
  
    }  
  
}
```

# @SessionAttributes & @ModelAttribute

- @SessionAttributes의 파라미터와 같은 이름이 @ModelAttribute에 있을 경우 세션에 있 객체를 가져온 후, 클라이언트로 전송받은 값을 설정한다.

```
@Controller
```

```
@SessionAttributes("user")
```

```
public class LoginController {
```

```
.....
```

```
    @PostMapping("/dologin")
```

```
    public String doLogin(@ModelAttribute("user") User user, Model model) {
```

```
.....
```

```
}
```

```
}
```

# @SessionAttribute

- 메소드에 @SessionAttribute가 있을 경우 파라미터로 지정된 이름으로 등록된 세션 정보를 읽어와서 변수에 할당한다.

```
@GetMapping("/info")  
  
public String userInfo(@SessionAttribute("user") User user) {  
    //...  
    //...  
    return "user";  
}
```

# SessionStatus

- SessionStatus 는 컨트롤러 메소드의 파라미터로 사용할 수 있는 스프링 내장 타입이다. 이 오브젝트를 이용하면 현재 컨트롤러의 @SessionAttributes에 의해 저장된 오브젝트를 제거할 수 있다.

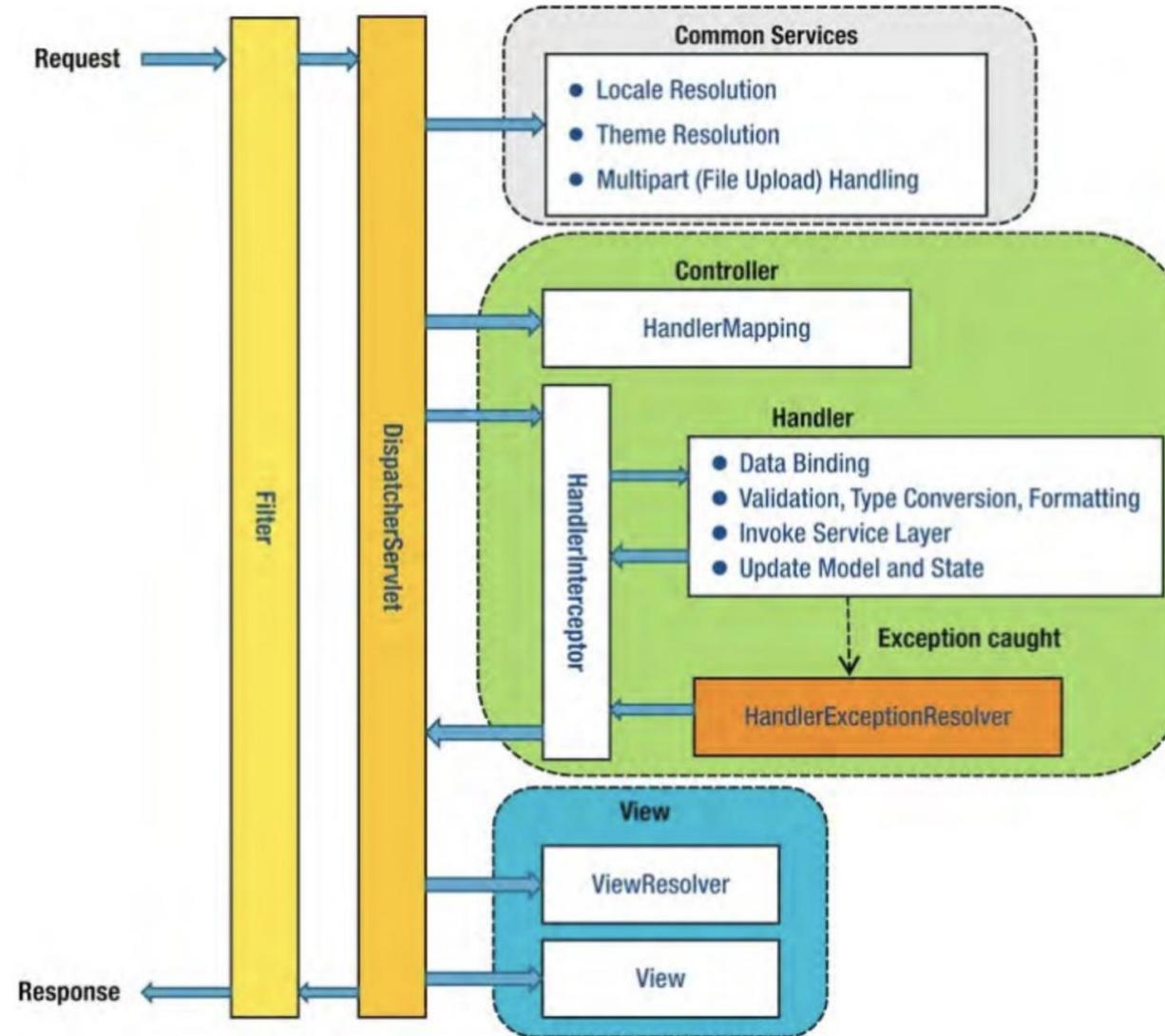
```
@Controller  
@SessionAttributes("user")  
public class UserController {  
    ....  
    @RequestMapping(value = "/user/add", method = RequestMethod.POST)  
    public String submit(@ModelAttribute("user") User user, SessionStatus sessionStatus) {  
        ....  
        sessionStatus.setComplete();  
        ....  
    }  
}
```

# Spring MVC - form tag 라이브러리

- `modelAttribute` 속성으로 지정된 이름의 객체를 세션에서 읽어와서 form태그로 설정된 태그에 값을 설정한다.

```
<form:form action="login" method="post" modelAttribute="user">  
Email : <form:input path="email" /><br>  
Password : <form:password path="password" /><br>  
<button type="submit">Login</button>  
</form:form>
```

# 인터셉터(Interceptor)?



# 인터셉터(Interceptor)?

- Interceptor는 Dispatcher servlet에서 Handler(Controller)로 요청을 보낼 때, Handler에서 Dispatcher servlet으로 응답을 보낼 때 동작한다.

# 인터셉터 작성법

- org.springframework.web.servlet.HandlerInterceptor 인터페이스를 구현한다.
- org.springframework.web.servlet.handler.HandlerInterceptorAdapter 클래스를 상속받는다.
- Java Config를 사용한다면, WebMvcConfigurerAdapter가 가지고 있는 addInterceptors메소드를 오버라이딩하고 등록하는 과정을 거친다.
- xml 설정을 사용한다면, <mvc:interceptors>요소에 인터셉터를 등록한다.

# 아규먼트 리졸버란?

- 컨트롤러의 메소드의 인자로 사용자가 임의의 값을 전달하는 방법을 제공하고자 할 때 사용된다.
- 예를 들어, 세션에 저장돼 있는 값 중 특정 이름의 값을 메소드 인자로 전달한다.

# 아규먼트 리졸버 작성방법 2/2

- Java Config에 설정하는 방법
  - WebMvcConfigurerAdapter를 상속받은 Java Config파일에서 addArgumentResolvers 메소드를 오버라이딩 한 후 원하는 아규먼트 리졸버 클래스 객체를 등록한다.
- xml 파일에 설정하는 방법

```
<mvc:annotation-driven>  
  
    <mvc:argument-resolvers>  
        <bean class="아규먼트리졸버클래스"></bean>  
    </mvc:argument-resolvers>  
  
</mvc:annotation-driven>
```

# Spring MVC의 기본 ArgumentResolver들

```
private List<HandlerMethodArgumentResolver> getDefaultArgumentResolvers() {
    List<HandlerMethodArgumentResolver> resolvers = new ArrayList<>();

    // Annotation-based argument resolution
    resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(), false));
    resolvers.add(new RequestParamMapMethodArgumentResolver());
    resolvers.add(new PathVariableMethodArgumentResolver());
    resolvers.add(new PathVariableMapMethodArgumentResolver());
    resolvers.add(new MatrixVariableMethodArgumentResolver());
    resolvers.add(new MatrixVariableMapMethodArgumentResolver());
    resolvers.add(new ServletModelAttributeMethodProcessor(false));
    resolvers.add(new RequestResponseBodyMethodProcessor(getMessageConverters(), this.requestResponseBodyAdvice));
    resolvers.add(new RequestPartMethodArgumentResolver(getMessageConverters(), this.requestResponseBodyAdvice));
    resolvers.add(new RequestHeaderMethodArgumentResolver(getBeanFactory()));
    resolvers.add(new RequestHeaderMapMethodArgumentResolver());
    resolvers.add(new ServletCookieValueMethodArgumentResolver(getBeanFactory()));
    resolvers.add(new ExpressionValueMethodArgumentResolver(getBeanFactory()));
    resolvers.add(new SessionAttributeMethodArgumentResolver());
    resolvers.add(new RequestAttributeMethodArgumentResolver());

    // Type-based argument resolution
    resolvers.add(new ServletRequestMethodArgumentResolver());
    resolvers.add(new ServletResponseMethodArgumentResolver());
    resolvers.add(new HttpEntityMethodProcessor(getMessageConverters(), this.requestResponseBodyAdvice));
    resolvers.add(new RedirectAttributesMethodArgumentResolver());
    resolvers.add(new ModelMethodProcessor());
    resolvers.add(new MapMethodProcessor());
    resolvers.add(new ErrorsMethodArgumentResolver());
    resolvers.add(new SessionStatusMethodArgumentResolver());
    resolvers.add(new UriComponentsBuilderMethodArgumentResolver());

    // Custom arguments
    if (getCustomArgumentResolvers() != null) {
        resolvers.addAll(getCustomArgumentResolvers());
    }

    // Catch-all
    resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(), true));
    resolvers.add(new ServletModelAttributeMethodProcessor(true));

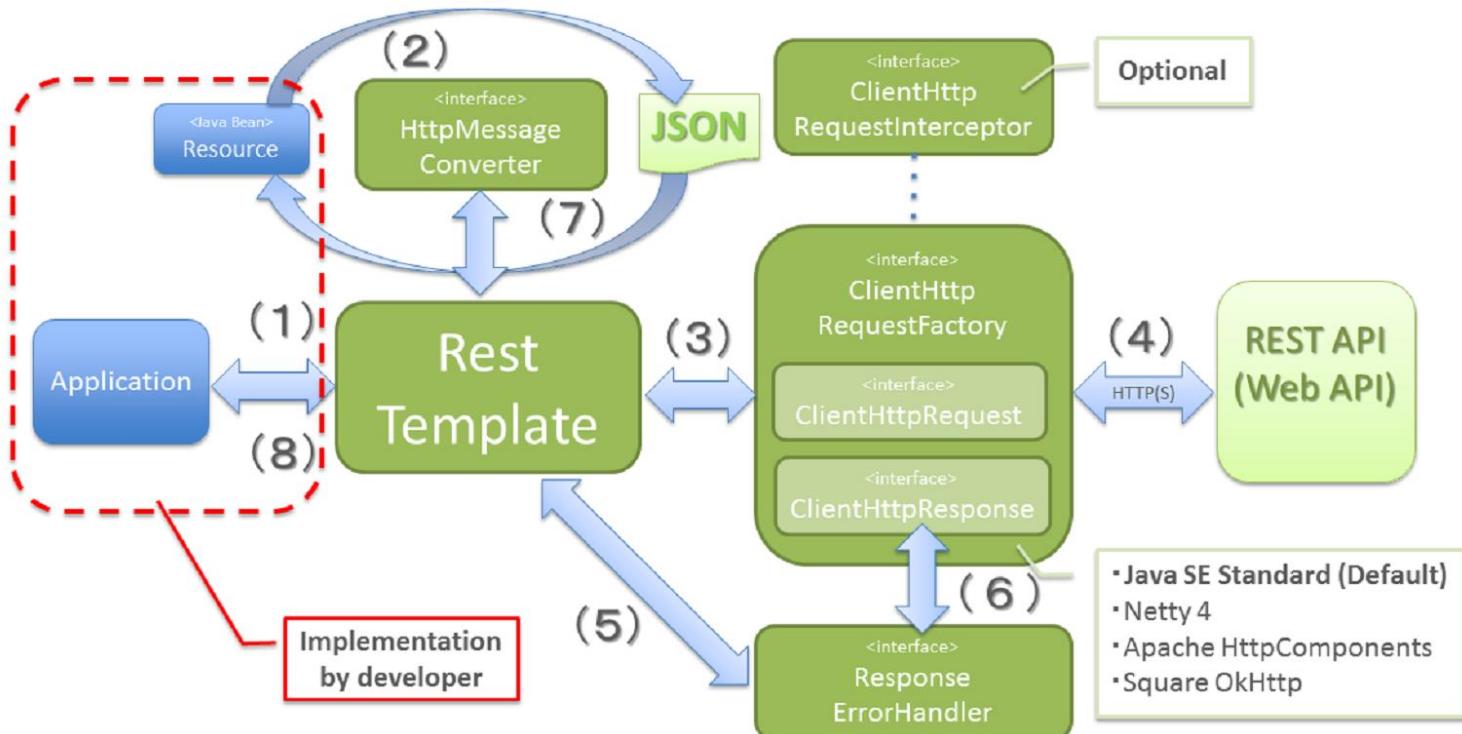
    return resolvers;
}
```

# Web API Client

# RestTemplate이란?

- RestTemplate은 REST API를 호출할 때 사용할 메소드를 제공하는 클래스다.
- 클래스명에 Rest가 있어서 오해할 수 있지만, 엄밀히 말하자면 RestTemplate은 REST API를 위한 전용 클래스가 아니다. RestTemplate이 스프링 프레임워크가 제공하는 HTTP 클라이언트 기능의 진입점이 되는 클래스로서 REST API와 궁합이 잘맞는 메소드도 많이 갖추고 있다.
- RestTemplate은 인스턴스를 생성하는 부분을 제외하면 한줄의 코드로 다음과 같은 것을 행할 수 있다.
  - 요청 URL 조립
  - HTTP 요청 전송
  - HTTP 응답 수신
  - 응답 본문을 자바 객체로 변환

# RestTemplate 동작 구조



(1) 애플리케이션은 RestTemplate메소드를 호출해서 REST API를 의뢰한다.

(2) RestTemplate은 HttpMessageConverter를 사용해 자바 객체를 메시지(JSON 등)로 변환하고 ClientHttpRequestFactory에서 가져온 ClientHttpRequest에게 메시지 전송을 의뢰한다.

(3) ClientHttpRequest는 Java SE의 표준 클래스나 서드파티 라이브러리 클래스를 이용해 HTTP 프로토콜로 메시지를 전송한다.

(4) RestTemplate은 REST API로부터 응답받은 메시지를 HttpMesageConverter를 사용해 자바 객체로 변환한 후 애플리케이션에게 반환한다.

# RestTemplate을 구성하는 컴포넌트

- org.springframework.http.converter.HttpMessageConverter
  - HTTP본문 메시지와 자바빈즈를 서로 변환하기 위한 인터페이스다.
- org.springframework.http.client.ClientHttpRequestFactory
  - 요청을 전송할 객체(ClientHttpRequest인터페이스를 구현)를 만들기 위한 인터페이스다. 응답은 ClientHttpResponse인터페이스를 구현한 클래스가 수신한다. 스프링 프레임워크는 Java SE표준, Netty, OkHttp, Apache HttpComponents를 활용한 HTTP통신 구현 클래스를 제공한다.
- org.springframework.http.client.ClientHttpRequestInterceptor
  - HTTP통신 전후의 공통 처리를 통합하기 위한 인터페이스다. 이것을 활용하면 요청 헤더(인증 헤더, 요청 추적 헤더 등)의 추가, 통신 과정의 로그 기록, 통신 처리의 재시도와 같은 공통적인 처리를 RestTemplate에 통합할 수 있다.
- org.springframework.web.client.ResponseErrorHandler
  - 애플리케이션에서 에러가 발생하면 그 오류가 어떤 유형인지를 파악하고, 그 오류 상황에 맞는 적절한 조치를 해야한다. 이때 이 인터페이스를 구현하면 되는데 스프링 프레임워크는 HTTP 상태 코드의 400번대 오류를 처리할 수 있는 구현 클래스 (DefaultResponseErrorHandler)를 제공해야 한다.

# RestTemplate의 Bean 등록과 Autowired

```
@Bean
```

```
RestTemplate restTemplate(){  
    return new RestTemplate();  
}
```

-----

```
@Autowired
```

```
RestTemplate restTemplate;
```

# REST API 호출 1/2

- `getForObject`
  - GET 메소드를 사용해 리소르를 가져오기 위한 메소드, 응답 본문을 임의의 자바 객체로 변환해서 가져올 때 사용한다.
- `getForEntity`
  - GET 메소드를 사용해 리소스를 가져오기 위한 메소드, 응답을  `ResponseEntity`로 가져올 때 사용한다.
- `headForHeaders`
  - HEAD 메소드를 사용해 리소스 헤더 정보를 가져오기 위한 메소드
- `postForLocation`
  - POST 메소드를 사용해 리소스를 만들기 위한 메소드, 만들어진 리소스에 접근하기 위한 URI만 가져올 때 사용한다.
- `postForObject`
  - POST 메소드를 사용해 리소스를 만들기 위한 메소드. 응답 본문을 임의의 자바 객체로 변환해서 가져올 때 사용한다.

# REST API 호출 2/2

- postForEntity
  - POST 메소드를 사용해 리소스를 만들기 위한 메소드. 응답을 ResponseEntity로 가져올 때 사용한다.
- put
  - PUT메소드를 사용해 리소스를 만들거나 갱신하기 위한 메소드
- delete
  - DELETE메소드를 사용해 리소스를 삭제하기 위한 메소드
- optionsForAllow
  - OPTIONS메소드를 사용해 호출 가능한 HTTP메소드(REST API)목록을 가져오기 위한 메소드
- exchange
  - 임의의 HTTP메소드를 사용해 리소스에 접근하기 위한 메소드. 요청 헤더에 임의의 헤더 값을 설정할 때 사용한다.
- execute
  - 임의의 HTTP메소드를 사용해 리소스에 접근하기 위한 메소드. HttpMessageConverter를 사용하지 않고 본문을 읽거나 쓸 때 사용한다.

# 오류 응답 처리

- RestTemplate의 기본 구현에는 HTTP상태 코드가 400이상인 오류 응답이 나올때 다음과 같은 예외가 발생한다.
- 사용자 정의 상태 코드 관련 예외 클래스
  - org.springframework.web.client.UnknownHttpStatusCodeException
- 클라이언트 오류 계열의 상태 코드(4xx)
  - org.springframework.web.client.HttpClientErrorException
- 서버 오류 계열의 상태 코드(5xx)
  - org.springframework.web.client.HttpServerErrorException

# 타임아웃 지정

- 서버와의 통신에서 타임아웃을 지정하고 싶다면 다음과 같이 빈을 정의한다.

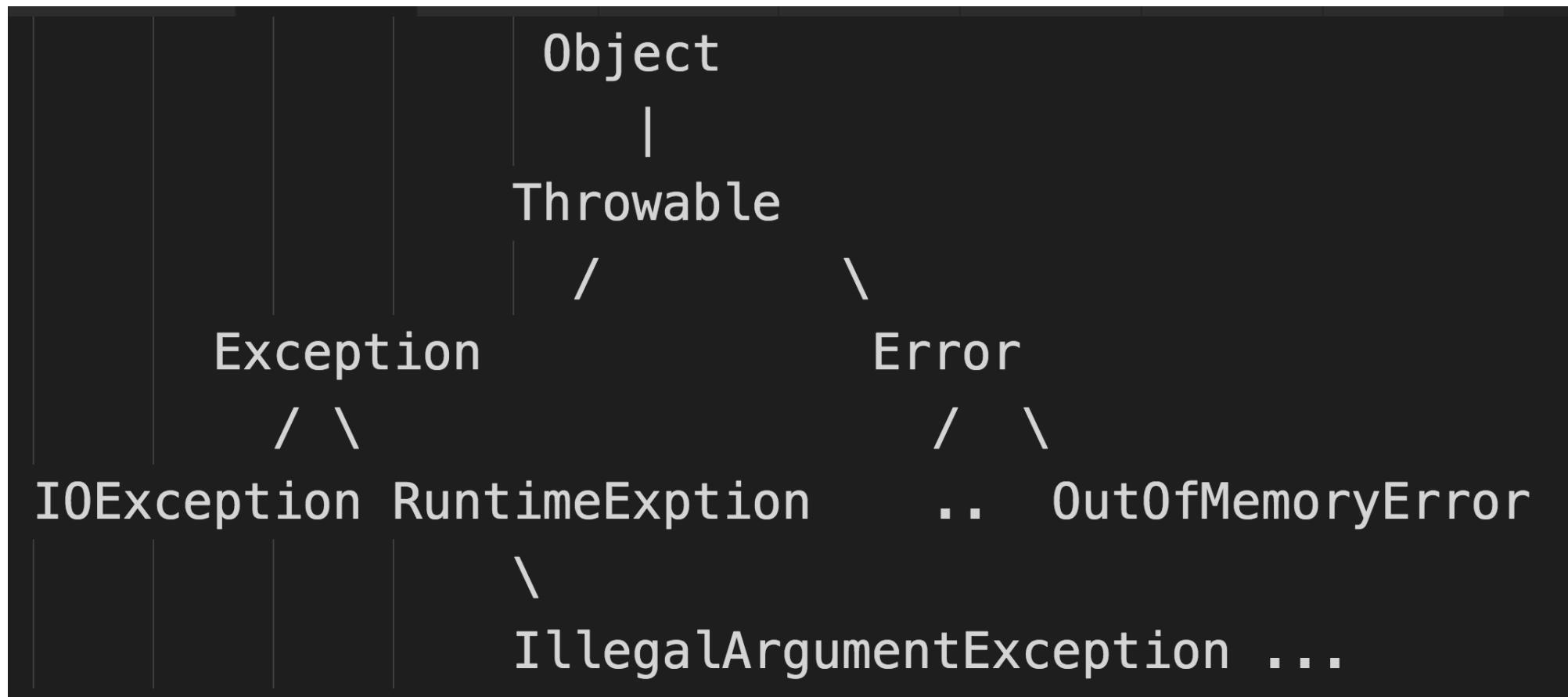
@Bean

```
RestTemplate restTemplate(){  
    SimpleClientHttpRequestFactory clientHttpRequestFactory = new SimpleClientHttpRequestFactory();  
    clientHttpRequestFactory.setConnectTimeout(2000); // (1)  
    clientHttpRequestFactory.setReadTimeout(2000); // (2)  
    RestTemplate restTemplate = new RestTemplate(clientHttpRequestFactory);  
    return restTemplate;  
}
```

- (1) connectTimeout 프로퍼티에 서버와의 타임아웃 시간(밀리초)를 설정한다. 타임아웃 시간을 초과하면 org.springframework.web.client.ResourceAccessException이 발생한다.
- (2) readTimeout프로퍼티에 응답 데이터의 읽기 타임아웃 시간(밀리초)를 설정한다. 타임아웃 시간을 초과하면 ResourceAccessException이 발생한다.

# Exception 처리

# 예외 클래스의 계층 구조



# try, catch, throw, throws

1. Exception이 발생하면, catch로 잡는다.
2. catch로 잡은 Exception을 해당 Layer에 맞는 RuntimeException 객체로 변환하여 throw한다.
3. Exception이 발생한 메소드에는 해당 Layer에 맞는 RuntimeException을 throws하도록 정의한다.
4. 발생한 Exception을 프리젠테이션 레이어까지 계속 throws 한다.
5. 프리젠테이션 레이어에서는 알맞은 오류 메시지를 화면에 출력한다.

# ControllerAdvice 1/3

```
@ControllerAdvice(annotations = Controller.class)
@Order(2)
public class GlobalControllerExceptionHandler {

    @ExceptionHandler(value = CookieTheftException.class)
    public String handleCookieTheftException(CookieTheftException e){
        SecurityContextHolder.getContext().setAuthentication(null);
        return "redirect:/members/login";
    }

    @ExceptionHandler(value = Exception.class)
    public String handleException(Exception e) {
        return "exceptions/exception";
    }
}
```

# ControllerAdvice 2/3

```
└@Setter  
└@Getter  
public class ApiError {  
    private HttpStatus status;  
    private String message;  
  
    └💡 public ApiError(HttpStatus status, String message) {  
        super();  
        this.status = status;  
        this.message = message;  
    }  
}
```

# ControllerAdvice 3/3

- Web API 오류 메시지는 개발자 도구에서 확인해야한다.

```
@ControllerAdvice(annotations = RestController.class)
@Order(1)
public class GlobalRestControllerExceptionHandler {

    @ExceptionHandler(value = { Exception.class })
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    @ResponseBody
    protected ApiError handleConflict(RuntimeException e, WebRequest request) {
        ApiError apiError =
            new ApiError(HttpStatus.INTERNAL_SERVER_ERROR, e.getLocalizedMessage());
        return apiError;
    }
}
```

# 참고 문서

- <https://spring.io/guides/gs/securing-web/>
- <http://www.baeldung.com/spring-boot-security-autoconfiguration>
- <https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/>
- <https://terasolunaorg.github.io/guideline/5.2.1.RELEASE/en/index.html>

# 참고 문서

- 자바 ORM 표준 JPA 프로그래밍 – 에이콘출판사 (김영한)
- 스프링 프레임워크의 실제 – ITC
- 스프링 입문을 위한 자바 객체 지향의 원리와 이해 – 위키북스 (김종민)
- 스프링 철저 입문 – 위키북스(NTT 데이터 저)
- 스프링 인 액션 – 제이펍
- 스프링 MVC 프로그래밍 - 에이콘출판사