

# App Server 기본

- 강경미 ([carami@nate.com](mailto:carami@nate.com))

# 자바 개발 환경 만들기

# Git 설치

# Git 설치

- Git 설치 후 다음의 내용을 설정한다.

```
git config --global user.name "이름"  
git config --global user.email 이메일  
git config --global core.autocrlf true
```

# JDK 설치

# JDK 11 설치

- 초보자가 공부할 때는 JDK 8도 충분. 서비스 업체들의 경우 11이상을 사용하는 경우가 많다.
- JDK 17을 고려하는 경우
  - Java 17이 2021년 9월 출시.
  - 10부터는 6개월마다 출시. LTS(Long Term Support)버전은 3년마다 출시.
  - JDK8 LTS가 JDK11 LTS보다 유지보수 기간이 더 길다. 그래서, 직접 17로 업데이트하는 것을 고려
- M1 Mac용 JDK는 17버전에서 지원.
- <https://adoptium.net/releases.html?variant=openjdk11&jvmVariant=hotspot>
- M1 Mac용 11JDK - Azul Zulu JDK
- <https://www.azul.com/downloads/?version=java-11-its&os=macos&architecture=arm-64-bit&package=jdk>

# JDK 11 설치

- 윈도우 사용자는 zip파일을, 맥 사용자는 tar.gz을 다운로드한다. (M1 맥 사용자는 뒤에 따로 설치 방법을 설명합니다.)
- 압축을 해제한다. jdk-XXX 폴더가 생성된다. XXX는 버전에 따라 다른 부분을 표현하였다.
- 디렉토리 구조 ( `tree -d -L 3 jdk-XXX` 명령으로 확인)

## JDK 11 설치

- `sudo mv jdk-XXX /Library/Java/JavaVirtualMachines`
- XXX는 버전에 따라 달라지는 부분을 표현하였다.
- sudo 명령은 root관리자 권한으로 실행하라는 명령이다. 본인의 암호를 입력한다.
- 윈도우 사용자는 `c:\\Program Files\\` 폴더에 복사한다.

# JDK 11 설치 - Mac사용자

- 터미널에서 `echo $0` 이라고 명령한다.

- zsh라고 나올 경우

`code ~/.zshrc` 명령을 수행한다.

- bash라고 나올 경우

`code ~/.bashrc` 명령을 수행한다.

- 파일을 열었으면 마지막 줄에 다음을 추가한다.

```
export JAVA_HOME=$(/usr/libexec/java_home -v 11)
export PATH=$PATH:$JAVA_HOME/bin
```

## JDK 11 설치 - Mac사용자

- 터미널을 종료하고 재시작한다. 다음과 같이 명령한다.

```
java --version  
javac -version
```

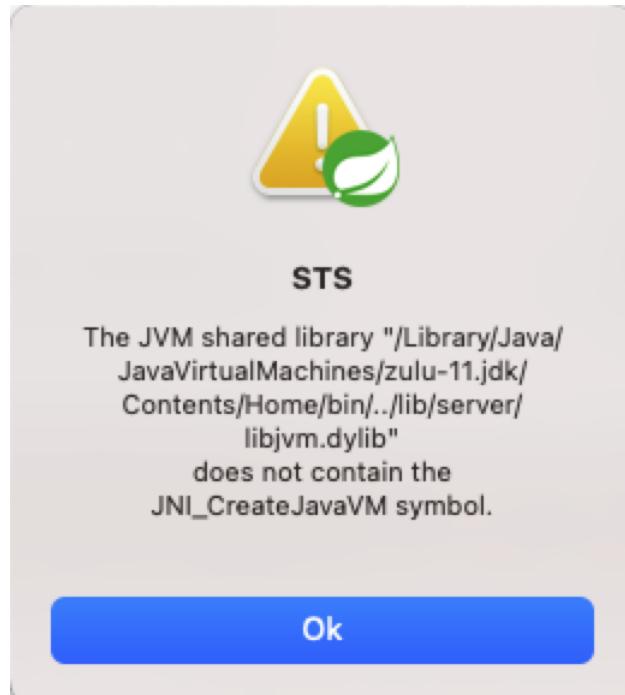
## JDK 11 설치 - 윈도우 사용자

구글에서 JAVA\_HOME PATH 환경변수라고 검색해보자.

<https://vmpo.tistory.com/6>

# STS3는 M1 맥을 지원하지 않기 때문에 인텔 JDK, 인텔 STS3를 설치한다.

- STS 3를 다운로드 한 후 설치한다.
- [https://download.springsource.com/release/STS/3.9.18.RELEASE/dist/e4.21/spring-tool-suite-3.9.18.RELEASE-e4.21.0-linux-gtk-x86\\_64.tar.gz](https://download.springsource.com/release/STS/3.9.18.RELEASE/dist/e4.21/spring-tool-suite-3.9.18.RELEASE-e4.21.0-linux-gtk-x86_64.tar.gz)
- STS를 실행하면 다음과 같은 실행 오류가 발생한다.



<https://lynmp.com/ko/article/rm867dd8a6e26fbbae>

- 손상되었기 때문에 열 수 없습니다. 라고 나온다면 Gatekeeper를 비활성화 해야 실행될 수도 있다.
- sudo spctl --master-disable 명령을 터미널에서 실행한다.
- sudo spctl --master-enable 명령을 실행하면 다시 활성화된다.

- x86용으로 빌드된 STS3가 m1 JAVA로는 실행할 수 없다는 오류 메시지이다.
- IBM x86 JDK를 다운로드 받는다.

<https://developer.ibm.com/languages/java/semeru-runtimes/downloads/>

[https://github.com/ibmruntimes/semeru11-binaries/releases/download/jdk-11.0.15+10\\_openj9-0.32.0/ibm-semeru-open-jdk\\_x64\\_mac\\_11.0.15\\_10\\_openj9-0.32.0.tar.gz](https://github.com/ibmruntimes/semeru11-binaries/releases/download/jdk-11.0.15+10_openj9-0.32.0/ibm-semeru-open-jdk_x64_mac_11.0.15_10_openj9-0.32.0.tar.gz)

- 압축을 해제한후 원하는 경로에 복사한다. 필자의 경우 다음과 같은 폴더에 압축을 해제한 폴더를 복사하였다.

```
/Users/carami/tools/ibmx86_jdk-11.0.15+10
```

- /Applications/STS.app/Contents/Info.plist의 -vm 옵션 부분이 `<!--` `-->` 으로 주석문 처리되어 있다. 주석문을 제거하고 알맞게 경로를 수정한다.

```
<string>-vm</string><string>/Users/carami/tools(ibmx86_jdk-11.0.15+10/Contents/Home/bin/java</string>
```

## 자바 프로그램 작성과 실행

- 터미널에서 특정 디렉토리로 이동한다. (소스가 저장될 디렉토리)
- code Hello.java 을 실행한 후 아래와 같이 저장한다.

```
public class Hello{  
    public static void main(String[] args){  
        System.out.println("Hello");  
    }  
}
```

## 자바 프로그램 작성과 실행

- java 컴파일러 javac 명령으로 hello.java를 컴파일 한다.

```
javac Hello.java
```

- 컴파일이 성공하면 오류메시지가 없이 Hello.class 파일이 생성된다.

- ls -la 명령으로 파일이 생성되었는지 확인한다.

- JVM(자바 가상 머신)으로 Hello.class 를 실행한다. java 명령이 JVM을 의미한다. 이 때 확장자는 입력하지 않는다.

```
java Hello
```

# **STS 3 설치**

<https://spring.io/tools>에서 다운로드

본인의 운영체제에 맞는 STS 3를 다운로드 받아 설치한다.

# Tomcat 8.5 다운/설치

<https://tomcat.apache.org/>

좌측의 Tomcat 8을 클릭하여 다운로드 한다. 10부터는 Java EE가 아니라 Jakarta EE이다.

The Apache Tomcat® software is an open source implementation of the [Jakarta Servlet](#), [Jakarta WebSocket](#), [Jakarta Annotations](#) and [Jakarta Authentication](#) specifications. These specifications are part of the Jakarta EE platform.

The Jakarta EE platform is the evolution of the Java EE platform. Tomcat 10 and later implement the Jakarta EE specifications developed as part of Java EE.

The Apache Tomcat software is developed in an open and participatory environment and released under the Apache Software License. The Tomcat project is intended to be a collaboration of the best-of-breed developers from around the world. To learn more about getting involved, [click here](#).

Apache Tomcat software powers numerous large-scale, mission-critical web applications across the globe. These users and their stories are listed on the [PoweredBy](#) wiki page.

Apache Tomcat, Tomcat, Apache, the Apache feather, and the Apache Tomcat project logo are trademarks of the Apache Software Foundation.

### Tomcat 10.1.1 Released

The Apache Tomcat Project is proud to announce the release of version 10.1.1 of Apache Tomcat. This release is the first release of the Jakarta EE 10 platform.

Applications that run on Tomcat 9 and earlier will not run on Tomcat 10 without changes. Java classes and resources must be placed in the `$CATALINA_BASE/webapps-javaee` directory and Tomcat will automatically convert them to the `webapps` directory. This conversion is performed using the [Apache Tomcat migration tool for off-line use](#).

The notable changes in this release are:

# 본인의 운영체제에 맞는 Tomcat을 다운로드 받아 설치한다.

- 맥, 리눅스 사용자는 tar.gz
- 윈도우 사용자는 zip

## 8.5.83

Please see the [README](#) file for packaging information. It explains what every distribution contains.

### Binary Distributions

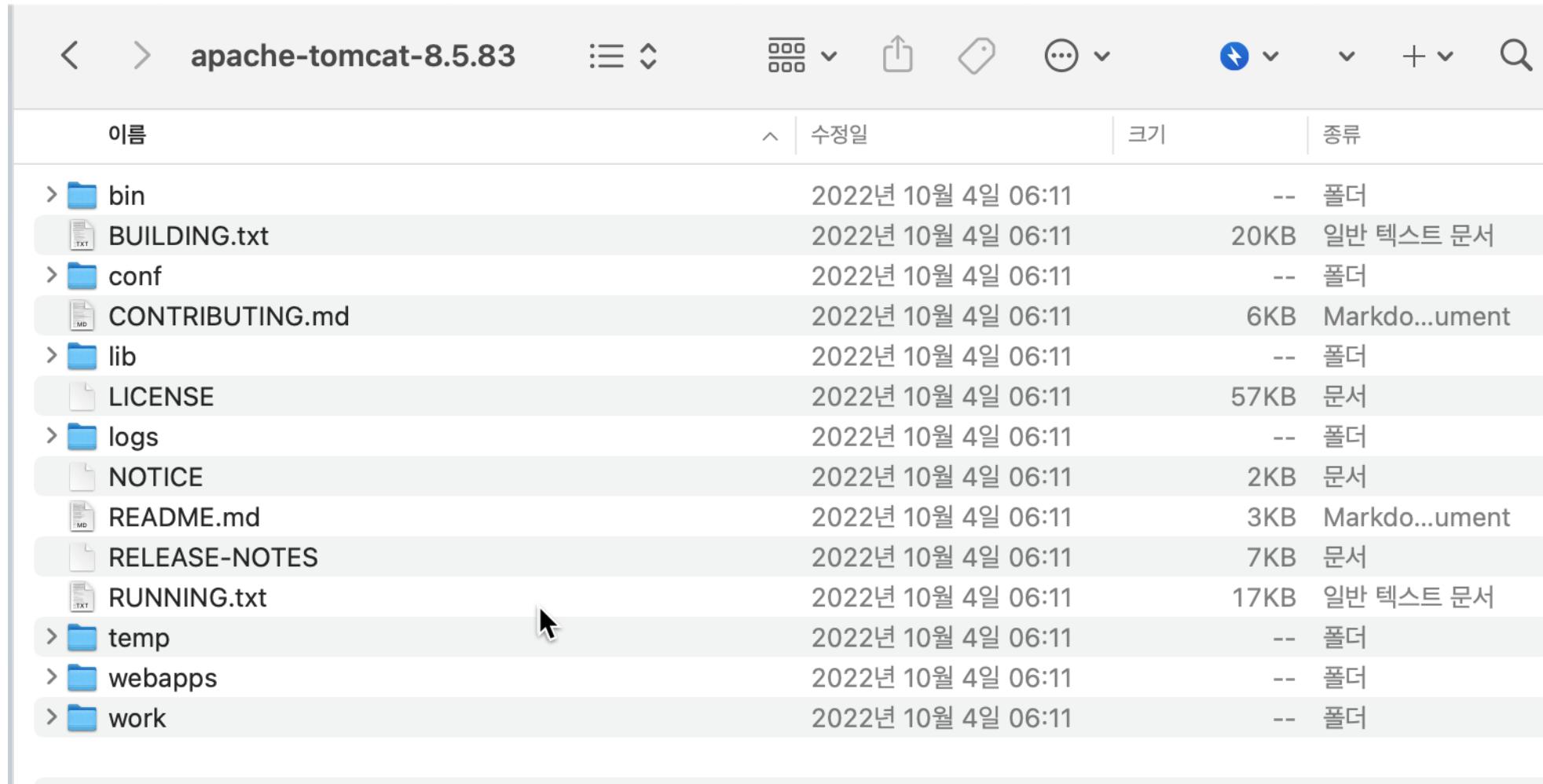
- Core:
  - [zip \(pgp, sha512\)](#)
  - [tar.gz \(pgp, sha512\)](#)
  - [32-bit Windows zip \(pgp, sha512\)](#)
  - [64-bit Windows zip \(pgp, sha512\)](#)
  - [32-bit/64-bit Windows Service Installer \(pgp, sha512\)](#)
- Full documentation:
  - [tar.gz \(pgp, sha512\)](#)
- Deployer:
  - [zip \(pgp, sha512\)](#)
  - [tar.gz \(pgp, sha512\)](#)
- Extras:
  - [Web services jar \(pgp, sha512\)](#)
- Embedded:
  - [tar.gz \(pgp, sha512\)](#)
  - [zip \(pgp, sha512\)](#)



### Source Code Distributions

- [tar.gz \(pgp, sha512\)](#)
- [zip \(pgp, sha512\)](#)

# 알맞은 폴더에 압축을 해제한다.



The screenshot shows a file explorer window with the title bar "apache-tomcat-8.5.83". The toolbar includes icons for back, forward, search, and file operations. The main area displays a list of files and folders:

이름	수정일	크기	종류
> bin	2022년 10월 4일 06:11	--	폴더
BUILDING.txt	2022년 10월 4일 06:11	20KB	일반 텍스트 문서
> conf	2022년 10월 4일 06:11	--	폴더
CONTRIBUTING.md	2022년 10월 4일 06:11	6KB	Markdo...ument
> lib	2022년 10월 4일 06:11	--	폴더
LICENSE	2022년 10월 4일 06:11	57KB	문서
> logs	2022년 10월 4일 06:11	--	폴더
NOTICE	2022년 10월 4일 06:11	2KB	문서
README.md	2022년 10월 4일 06:11	3KB	Markdo...ument
RELEASE-NOTES	2022년 10월 4일 06:11	7KB	문서
RUNNING.txt	2022년 10월 4일 06:11	17KB	일반 텍스트 문서
> temp	2022년 10월 4일 06:11	--	폴더
> webapps	2022년 10월 4일 06:11	--	폴더
> work	2022년 10월 4일 06:11	--	폴더

**bin 폴더에 Tomcat 실행파일이 존재한다.**

- 맥, 리눅스
  - [startup.sh](#), [shutdown.sh](#)
- 윈도우
  - [startup.bat](#) , [shutdown.bat](#)

실행파일을 수정해서 실행되는 내용이 포그라운드로 보여지게 할 수 있다.

```
exec "$PRGDIR"/"$EXECUTABLE" start "$@"
```

위와 같은 마지막 줄을

```
exec "$PRGDIR"/"$EXECUTABLE" run "$@"
```

로 수정한다.

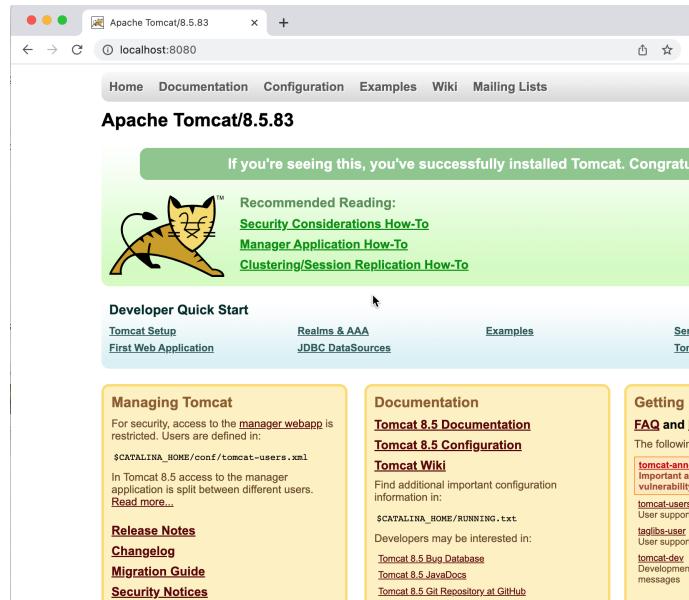
# Tomcat 실행

- 8080 포트로 실행된다.

```
13-Oct-2022 12:28:54.830 정보 [main] org.apache.coyote.AbstractProtocol.init 프로토콜 핸들러 ["http-nio-8080"]을 (를 )
초기화합니다.
13-Oct-2022 12:28:54.873 정보 [main] org.apache.catalina.startup.Catalina.load Initialization processed in 354 ms
13-Oct-2022 12:28:54.899 정보 [main] org.apache.catalina.core.StandardService.startInternal 서비스 [Catalina]을 (를 ) 시작합니다.
13-Oct-2022 12:28:54.899 정보 [main] org.apache.catalina.core.StandardEngine.startInternal 서버 엔진을 시작합니다 : [Apache Tomcat/8.5.83]
13-Oct-2022 12:28:54.906 정보 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 웹 애플리케이션 디렉토리 [/Users/urstory/devel/tools/apache-tomcat-8.5.83/webapps/docs]을 (를 ) 배치합니다.
13-Oct-2022 12:28:55.072 정보 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 웹 애플리케이션 디렉토리 [/Users/urstory/devel/tools/apache-tomcat-8.5.83/webapps/docs]에 대한 배치가 [165] 밀리초에 완료되었습니다.
13-Oct-2022 12:28:55.073 정보 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 웹 애플리케이션 디렉토리 [/Users/urstory/devel/tools/apache-tomcat-8.5.83/webapps/manager]을 (를 ) 배치합니다.
13-Oct-2022 12:28:55.088 정보 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 웹 애플리케이션 디렉토리 [/Users/urstory/devel/tools/apache-tomcat-8.5.83/webapps/manager]에 대한 배치가 [15] 밀리초에 완료되었습니다.
13-Oct-2022 12:28:55.088 정보 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 웹 애플리케이션 디렉토리 [/Users/urstory/devel/tools/apache-tomcat-8.5.83/webapps/examples]을 (를 ) 배치합니다.
13-Oct-2022 12:28:55.187 정보 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 웹 애플리케이션 디렉토리 [/Users/urstory/devel/tools/apache-tomcat-8.5.83/webapps/examples]에 대한 배치가 [99] 밀리초에 완료되었습니다.
13-Oct-2022 12:28:55.187 정보 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 웹 애플리케이션 디렉토리 [/Users/urstory/devel/tools/apache-tomcat-8.5.83/webapps/ROOT]을 (를 ) 배치합니다.
13-Oct-2022 12:28:55.193 정보 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 웹 애플리케이션 디렉토리 [/Users/urstory/devel/tools/apache-tomcat-8.5.83/webapps/ROOT]에 대한 배치가 [6] 밀리초에 완료되었습니다.
13-Oct-2022 12:28:55.193 정보 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 웹 애플리케이션 디렉토리 [/Users/urstory/devel/tools/apache-tomcat-8.5.83/webapps/host-manager]을 (를 ) 배치합니다.
13-Oct-2022 12:28:55.200 정보 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 웹 애플리케이션 디렉토리 [/Users/urstory/devel/tools/apache-tomcat-8.5.83/webapps/host-manager]에 대한 배치가 [7] 밀리초에 완료되었습니다.
13-Oct-2022 12:28:55.204 정보 [main] org.apache.coyote.AbstractProtocol.start 프로토콜 핸들러 ["http-nio-8080"]을 (를 )
시작합니다.
13-Oct-2022 12:28:55.211 정보 [main] org.apache.catalina.startup.Catalina.start Server startup in 537 ms
```

# Tomcat 실행

- <http://localhost:8080>
  - Tomcat설치폴더/webapps/ROOT 애플리케이션이 실행된 결과이다.
  - <http://localhost:8080/examples> 는 Tomcat설치폴더/webapps/examples 의 결과가 보여진다.
  - ROOT 웹 애플리케이션의 application context path는 / 이다.



## **Tomcat을 종료한다.**

- [shutdown.sh](#) or shutdown.bat 를 실행한다.

# H2 DBMS 실행하기

## H2 DBMS 다운로드

<https://www.h2database.com/html/main.html>

- zip 파일을 다운로드 받는다.  
version-2.1.214/h2-2022-06-13.zip

## H2 DBMS 실행

- 압축을 해제한다.

mac 사용자의 경우 압축해제폴더/bin으로 이동하여 다음과 같이 실행한다.

```
chmod 755 h2.sh
```

- 실행한다.

- 윈도우 사용자 - 압축해제폴더/bin 폴더에서 h2.bat 를 실행한다.

- 맥 사용자 - 압축해제폴더/bin 폴더에서 h2.sh를 실행한다.

# H2 DBMS 실행

- 자동으로 브라우저가 열린다. "연결"버튼을 클릭한다.

← → ⌂ ⓘ localhost:8082/login.jsp?jsessionid=9dbc78b5be4b10387972d58024995312

---

English ▾ 설정 도구 도움말

로그인

저장한 설정: Generic H2 (Embedded) ▾

설정 이름: Generic H2 (Embedded)

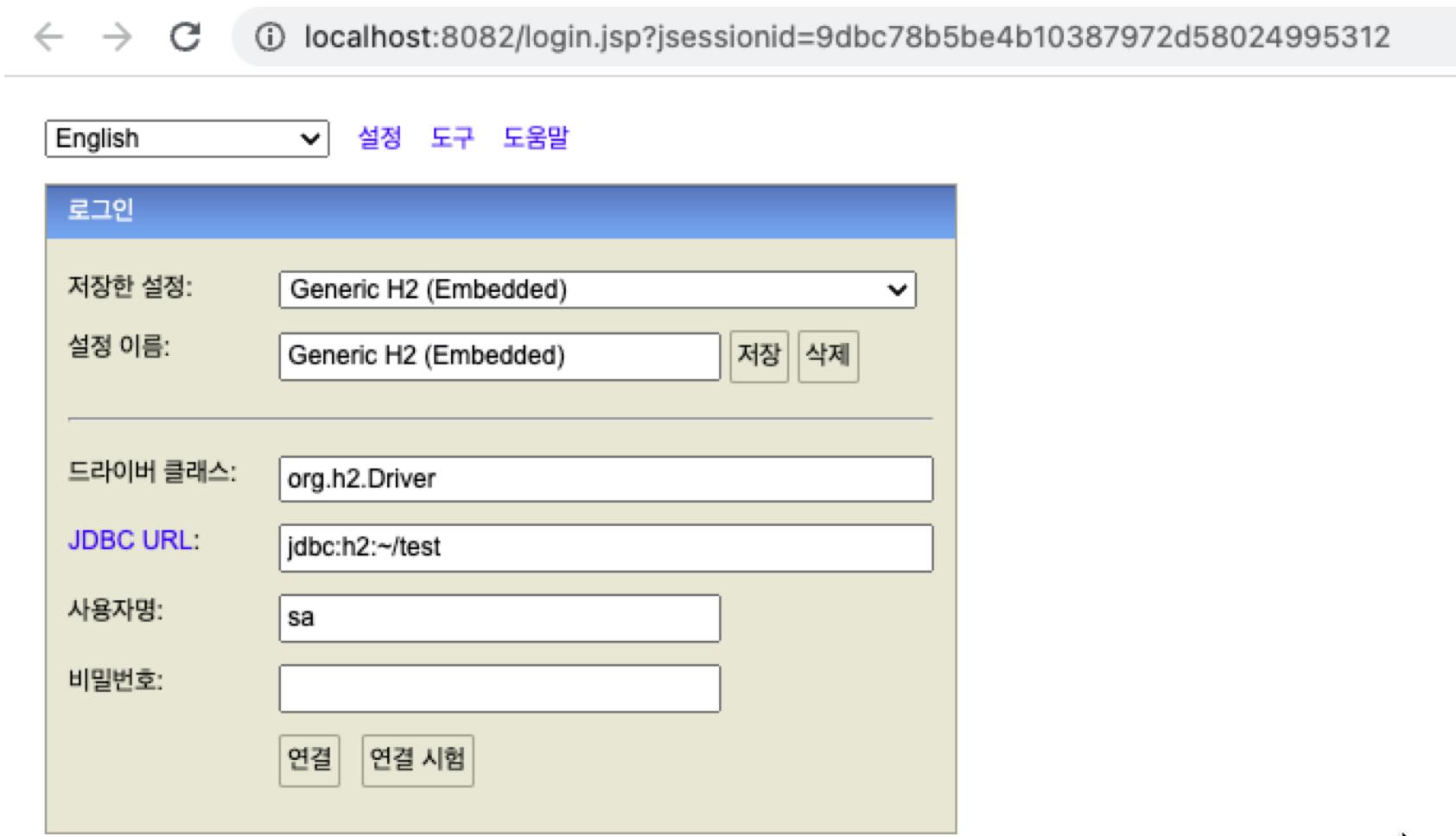
---

드라이버 클래스: org.h2.Driver

JDBC URL: jdbc:h2:~/test

사용자명: sa

비밀번호:



## H2 JDBC 연결

pom.xml

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.1.214</version>
</dependency>
```

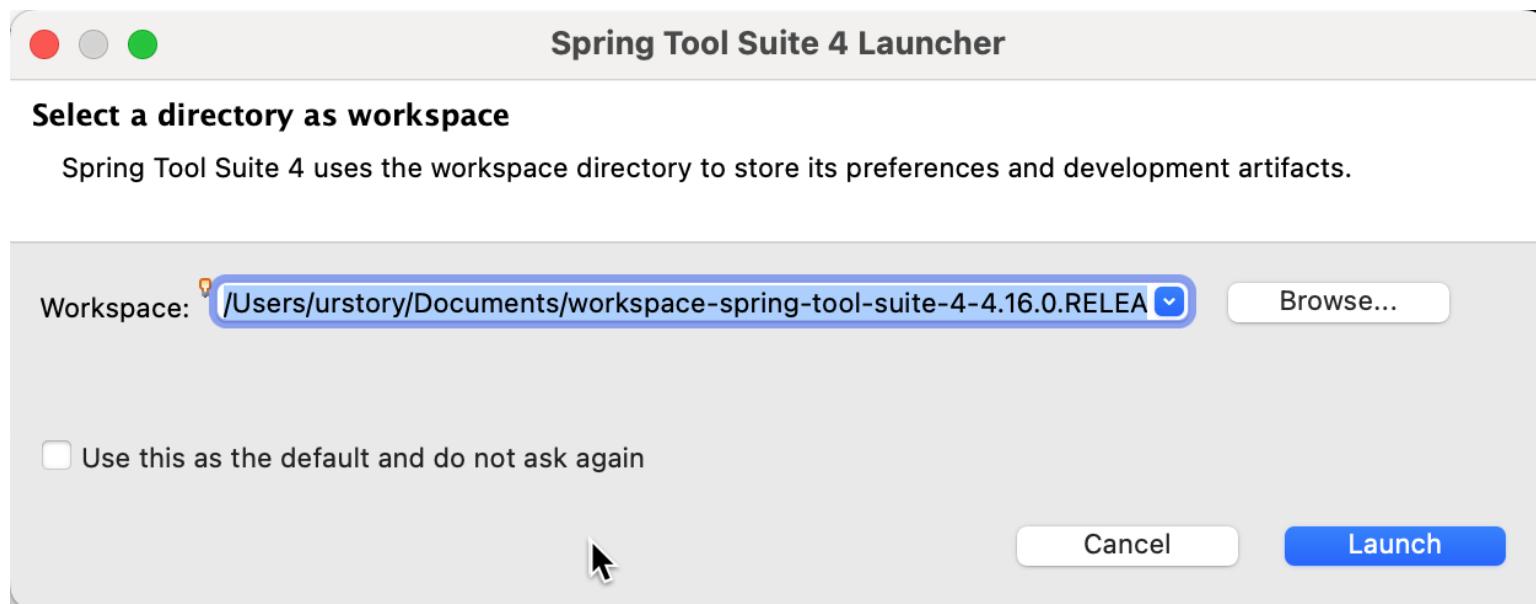
- H2 DBMS에서 SQL을 최대한 MySQL과 호환하도록 하려면 JDBC URL의 뒤에 ;MODE=MySQL 을 붙여준다.

```
private String driverClassName = "org.h2.Driver";
private String url = "jdbc:h2:tcp://localhost/~/test;MODE=MySQL";
private String username = "sa";
private String password = "";
```

## **STS 3 실행하기**

## 워크 스페이스 지정하기

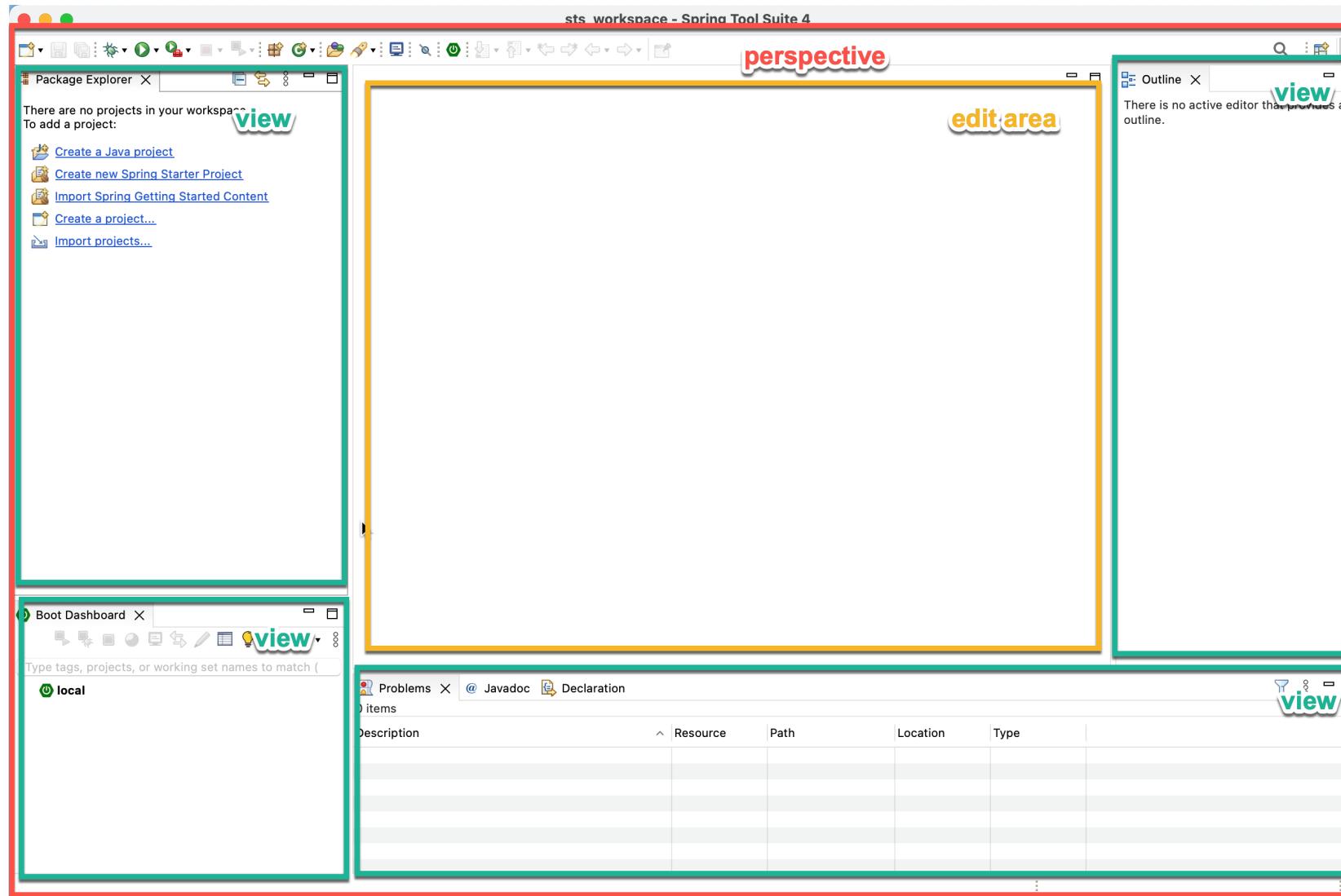
- STS 3를 실행하면 워크 스페이스를 물어본다. 워크 스페이스란 프로젝트가 저장되는 경로를 말한다. 본인이 관리하기 쉬운 폴더를 생성한 후 워크 스페이스를 지정한다. 경로를 지정하면 자동으로 폴더가 생성된다.



## 생성된 워크 스페이스 경로 살펴보기

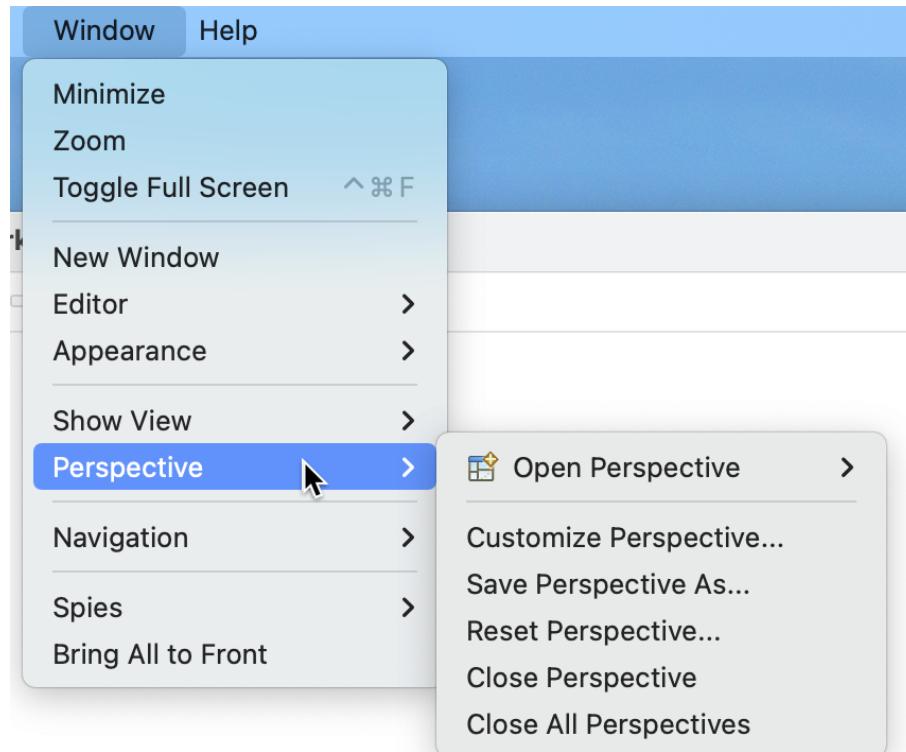
- 자동으로 생성된 워크스페이스 경로를 가보면, .metadata라는 hidden 폴더가 생성된 것을 알 수 있다. 이 폴더는 STS 3가 관리하는 폴더이다. 삭제하거나, 수동으로 편집하면 안된다.

```
> cd sts_workspace
> ls -la
total 0
drwxr-xr-x    3 ursstory  staff   96 10 13 12:34 .
drwxr-xr-x   17 ursstory  staff  544 10 13 12:34 ..
drwxr-xr-x    6 ursstory  staff  192 10 13 12:34 .metadata
```



# Perspective or View 메뉴

- Perspective를 리셋하거나, View를 추가로 보는 등의 작업을 수행할 수 있다.



# Spring FW 탄생 배경 , History

# Spring FW 등장한 이유

- EJB(Enterprise JavaBeans)
  - EJB는 애플리케이션 작성을 쉽게 해준다.
  - EJB는 선언적 프로그래밍 모델
  - 트랜잭션, 보안, 분산컴퓨팅 이런것들을 굉장히 쉽게 할 수 있다.
  - EJB를 구동시킬 수 있는 Web Application Server가 등장
  - EJB 너무 싫다. (개발도 힘들고, 배포도 힘들고.....)

# Expert One-on-One: J2EE Design and Development

- 책 제목
- 로드 존슨(Rod Johnson)
  - Spring을 소개한다.
  - EJB에서 했던일들을 더욱 간단한 Java Bean이 처리할 수 있게된다.

# 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

2022년 9월 기준 최신 버전 - 5.3.23

- 6.0은 Java 17이상에서 동작할 예정

## 스프링 버전별 새로운 기능 : 1.x

- IoC(Inverstion of Control)
- 컨테이너(DI 컨테이너)
- AOP(Aspect Orientied Programming)
- XML기반의 빈 정의
- 프레임워크 모듈간의 결합도 약화
- 트랜잭션 관리
- 데이터 엑세스 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 인티그레이션(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

- 향상된 시작경험
- Deprecated된 패키지 및 메서드 제거
- Java 8 지원
- Java EE 6, 7 지원
- Groovy 빈 정의 DSL
- 코어 컨테이너 개선
- 전반적인 웹 개선
- 웹소켓, SockJS, STOMP 메시징
- 테스트 개선

## 스프링 버전별 새로운 기능 : 4.1

- JMS 개선
- 캐싱 개선
- 웹 개선
- 웹소켓 메시징 개선
- 테스트 개선

## **스프링 버전별 새로운 기능 : 4.2**

- 코어 컨테이너 개선
- 데이터 액세스 개선
- JMS 개선
- 웹 개선
- 웹소켓 메시징 개선
- 테스트 개선

## 스프링 버전별 새로운 기능 : 4.3

- 코어 컨테이너 개선
- 데이터 엑세스 개선
- 캐싱 개선
- JMS 개선
- 웹 개선
- 웹소켓 메시징 개선
- 테스트 개선
- 새로운 라이브러리 및 서버 세대 지원

## **스프링 버전별 새로운 기능 : 5.0**

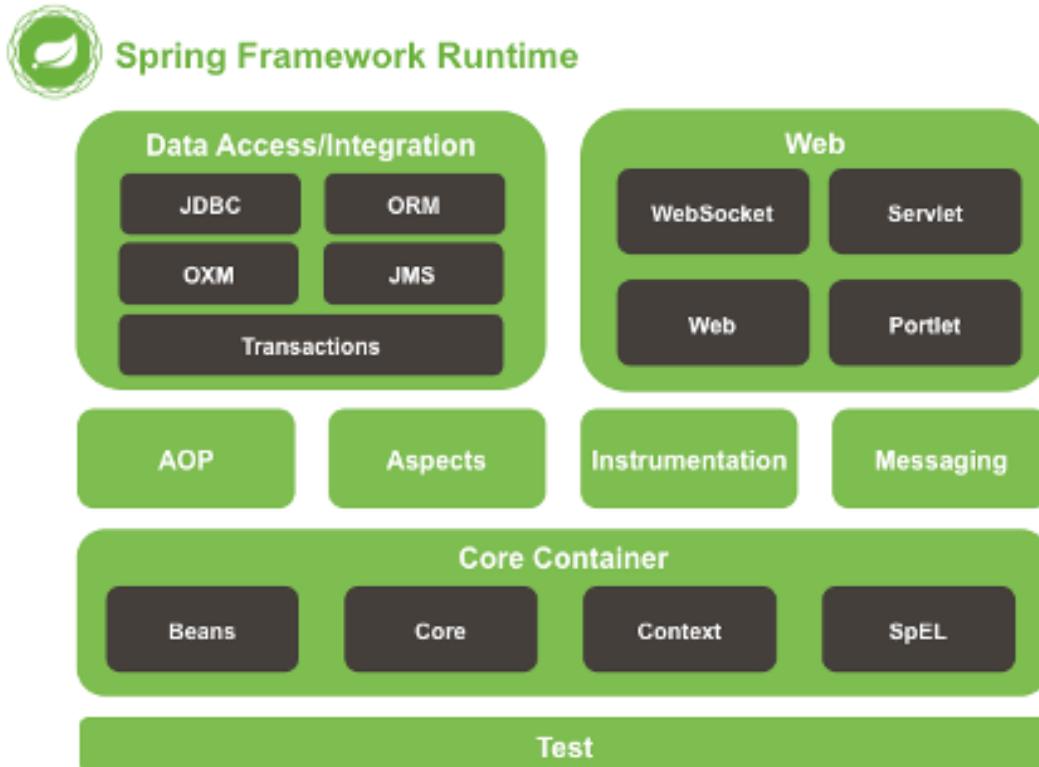
- JDK 8+9와 Java EE 7 베이스라인
- 패키지, 클래스, 메서드 제거
- 코어 컨테이너 개선
- 일반적인 웹 개선
- 리액티브(Reactive) 프로그래밍 모델
- 테스트 개선

# Spring FW 개요 & 구성요소

# Spring Framework라?

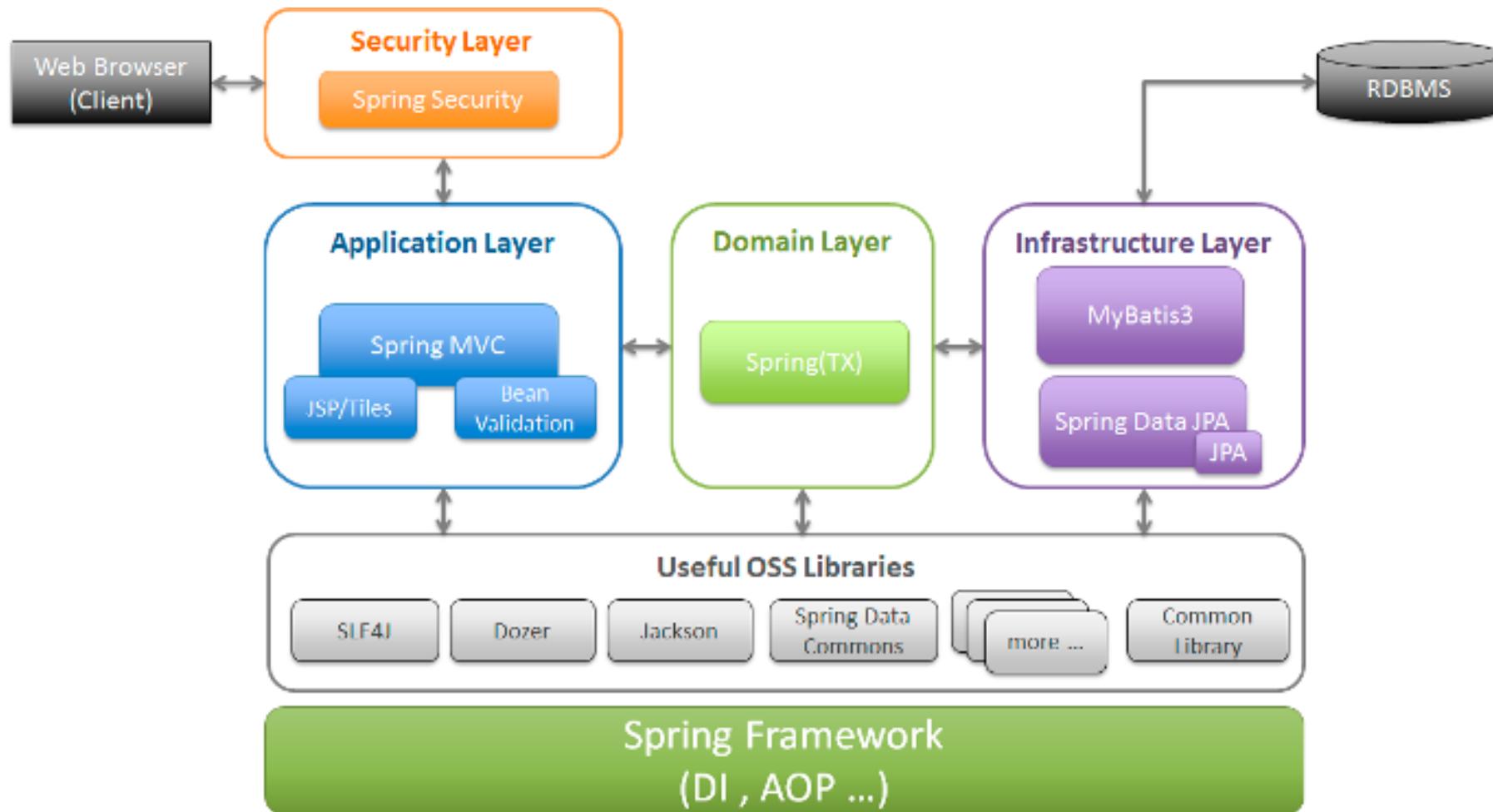
- 엔터프라이즈급 어플리케이션을 구축할 수 있는 가벼운 솔루션이자, 원스-스탑-숍(One-Stop-Shop) -원-스탑-숍 (One-Stop-Shop) : 모든 과정을 한꺼번에 해결하는 상점.
- 원하는 부분만 가져다 사용할 수 있도록 모듈화가 잘 되어 있다.
- POJO를 이용한 가볍고 non-invasive(비 침투적) 개발
- IoC 컨테이너이다.
- DI와 인터페이스 지향을 통한 느슨한 결합도
- 선언적으로 트랜잭션을 관리할 수 있다.
- 완전한 기능을 갖춘 MVC Framework를 제공한다.
- AOP와 공통 규약을 통한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

# Spring Framework 모듈



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

# Spring Framework 중요 구성 요소



## AOP 와 인스트루멘테이션(�strumentation)

- spring-AOP : AOP 얼라이언스(Aliance)와 호환되는 방법으로 AOP를 지원한다.
- spring-aspects : AspectJ와의 통합을 제공한다.
- spring-instrument : 인스트루멘테이션을 지원하는 클래스와 특정 WAS에서 사용하는 클래스로 더 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 엑세스(Data Access) / 통합(Integration)

- 데이터 엑세스/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-messaging모듈과의 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. HTTP 클라이언트와 Spring의 원격 지원을 위한 웹 관련 부분을 제공한다.
- spring-webmvc : Web-Servlet 모듈이라고도 불리며, Spring MVC 및 REST 웹 서비스 구현을 포함한다.
- spring-websocket : 웹 소켓을 지원한다.
- spring-webmvc-portlet : 포틀릿 환경에서 사용할 MVC 구현을 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
- 현재는 Spring개발의 표준으로 자리매김하고 있다.

## 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.
- JAVA EE 스펙 8 버전 이후엔 이클립스 제단으로 이관되면서 Jakarta EE로 이름이 변경되었다.  
package명도 javax로 시작되던 것이 jakarta로 시작되도록 변경되었다. 최신 WAS의 경우  
J2EE스펙을 지키는 것이 아니라 Jakarta EE스펙을 지키도록 만들어질 것이기 때문에 기존에 만  
들었던 코드가 리팩토링 될 필요가 있다.

## 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.

# 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

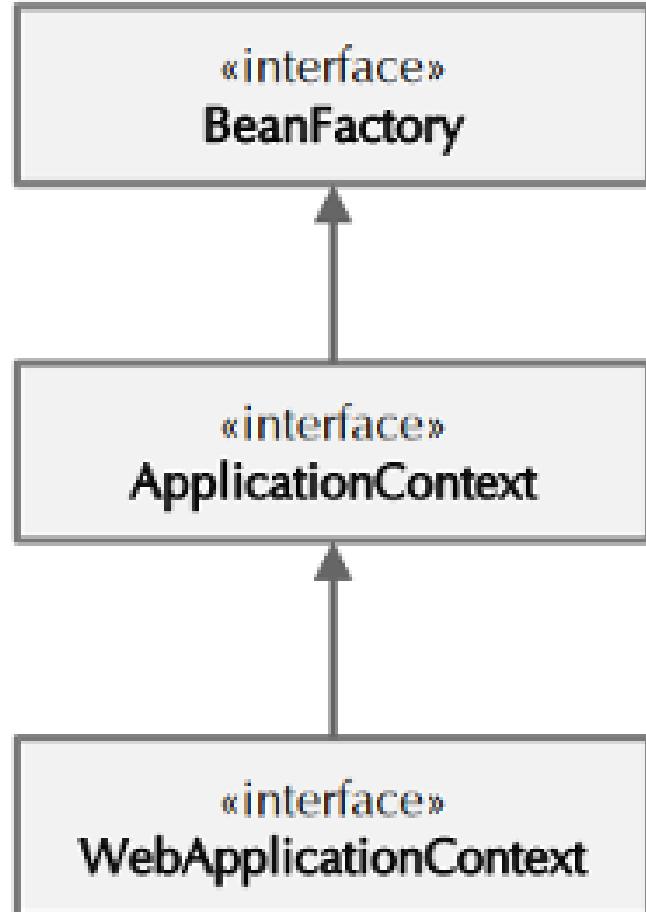
# Bean이란?

- Java에서 인스턴스 생성
  - 프로그래머가 직접 인스턴스를 만든다.

```
Book book = new Book();
```

- Bean은 컨테이너가 관리하는 객체
  - 객체의 생명주기를 컨테이너를 관리한다.
  - 객체를 싱글턴으로 만들것인지, 프로토타입으로 만들것인지

# 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 등지
  - 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

.....

## 스프링의 핵심 기능

- 관점지향 컨테이너
  - 빈을 생성, 관리한다.
  - 관점지향(AOP, aspect-oriented programming)

# Book.java

```
public class Book { // Book 클래스
    private String title; // title 인스턴스 field(속성)
    private int price; // price 인스턴스 field

    // Book생성자
    public Book(String title, int price) {
        this.title = title;
        this.price = price;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }
}
```

## Book클래스의 인스턴스 생성

```
Book book1 = new Book("즐거운 자바", 20000);
```

1. new Book("즐거운 자바", 20000)
  - 생성자가 호출되면 Heap메모리에 인스턴스가 생성된다.
2. book1 은 1)에서 생성한 인스턴스를 참조한다.
  - book1 참조변수

## 프로그래머가 직접 인스턴스를 생성 사용

```
public class BookExam01 {  
    public static void main(String[] args) {  
        Book book = new Book("Java", 10000);  
        System.out.println(book.getTitle());  
        System.out.println(book.getPrice());  
    }  
}
```

## **Bean을 만들 때 규칙**

- 기본 생성자가 있어야 한다.

# 객체를 싱글턴으로 생성해주고, 자기자신도 싱글턴인 ApplicationContext

- 컨테이너 역할을 수행한다.
- Spring 이 제공하는 컨테이너는 이것보다 훨씬 복잡한 일을 한다.

```
package exam;

import java.io.*;
import java.lang.reflect.Method;
import java.util.*;

public class ApplicationContext {
    // 1. 싱글턴 패턴 적용 - 자기 자신을 참조하는 static 필드를 선언한다. 바로 초기화
    private static ApplicationContext instance = new ApplicationContext();
    // 3. 1.에서만든 인스턴스를 반환하는 static메소드를 만든다.
    public static ApplicationContext getInstance(){
        return instance;
    }
}
```

```
private Properties props;
private Map objectMap;

// 2. 싱글턴 패턴 적용 - 생성자를 private으로 바꾼다.
private ApplicationContext(){
    props = new Properties();
    objectMap = new HashMap<String, Object>();
    try {
        props.load(new FileInputStream("src/main/resources/Beans.properties"));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
public Object getBean(String id) throws Exception{  
    Object o1 = objectMap.get(id);  
    if(o1 == null) {  
        String className = props.getProperty(id);  
        // class name에 해당하는 문자열을 가지고 인스턴스를 생성할 수 있다.  
        // Class.forName(className)은 CLASSPATH부터 className에 해당하는 클래스를 찾은 후  
        // 그 클래스 정보를 반환한다.  
        Class clazz = Class.forName(className);  
  
        // ClassLoader를 이용한 인스턴스 생성. 기본생성자가 있어야 한다.  
        Object o = clazz.newInstance(); // clazz정보를 이용해 인스턴스를 생성한다.  
        objectMap.put(id, o);  
        o1 = objectMap.get(id);  
    }  
    return o1;  
}
```

```
package exam;

public class ApplicationContextMain {
    public static void main(String[] args) throws Exception{
        ApplicationContext context = ApplicationContext.getInstance();

        Book book = (Book)context.getBean("book1"); // id에 해당하는 인스턴스를 달라.
        book.setPrice(5000);
        book.setTitle("즐거운 자바");

        System.out.println(book.getPrice());
        System.out.println(book.getTitle());

        System.out.println("-----");
        Book book2 = (Book)context.getBean("book1");
        System.out.println(book2.getTitle());

        if(book == book2){
            System.out.println("같은 인스턴스");
        }else{
            System.out.println("다른 인스턴스");
        }
    }
}
```

# ApplicationContext

- ApplicationContext는 다양한 인터페이스를 상속받고 있다.
- 스프링 컨테이너의 핵심 인터페이스!

```
org.springframework.context  
Interface ApplicationContext
```

- 그중에서도 BeanFactory도 ApplicationContext는 상속받는다.

```
org.springframework.beans.factory  
Interface BeanFactory
```

## ApplicationContext를 구현하고 있는 대표적인 클래스

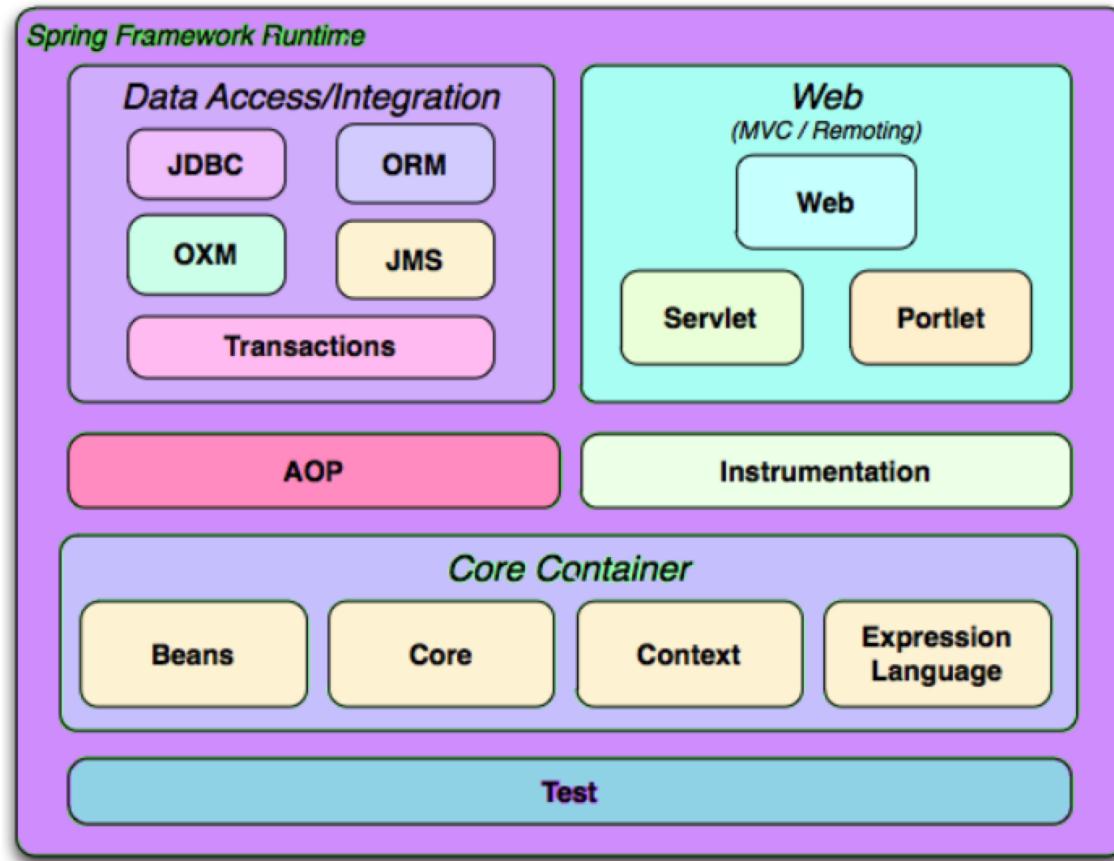
- CLASSPATH에서 XML설정파일을 읽어들여 동작한다.

```
org.springframework.context.support  
Class ClassPathXmlApplicationContext
```

# 스프링 프레임워크의 핵심 모듈

- Core Container 부분이 가장 핵심
- Gradle에서 아래의 라이브러리를 추가한다.

implementation group: 'org.springframework', name: 'spring-context', version: '5.3.23'



# Spring ApplicationContext를 사용해보자

```
package com.example.spring02;

import exam.Book;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringApplicationContextExam {
    public static void main(String[] args) {
        // 인스턴스를 인터페이스 타입으로 참조할 수 있다.
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        Book book1 = (Book)context.getBean("book1");
        Book book1 = context.getBean("book1", Book.class);
        book1.setTitle("즐거운 Spring Boot");
        book1.setPrice(5000);

        System.out.println(book1.getTitle());
        System.out.println(book1.getPrice());

        Book book2 = (Book)context.getBean("book1");
        System.out.println(book2.getTitle());
    }
}
```

## 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 활용가능
- 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;
```

## 타입을 사용한 오토와이어링 방식 2/2

- 인젝션을 할 수 있는 여러개의 빈을 발견하게 될 경우에는 NoUniqueBeanDefinitionException이 발생한다. 여러개일경우 @Qualifier애노테이션으로 빈 이름을 지정한 후 선택해서 사용해야한다.
- @Primary 애노테이션을 사용하면 @Qualifier를 사용하지 않았을 때 우선적으로 선택할 빈을 지정할 수 있다.
- @Qualifier역할을 하는 사용자 정의 애노테이션을 사용해서 표현할 수도 있다. 사용자 정의 애노테이션에 @Qualifier를 설정한다.

## 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 필드 이름과 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 스캔 할 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.

## 사용법

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

# 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
  - XML 파일을 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.

## 스코프 설정

- 자바 기반의 설정에서는 @Bean 애노테이션이 붙은 메소드에 @Scope 애노테이션을 추가해서 스코프를 명시한다. Bean은 기본적으로 싱글턴이지만 prototype으로 할 경우 매번 새로운 인스턴스가 생성된다.

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

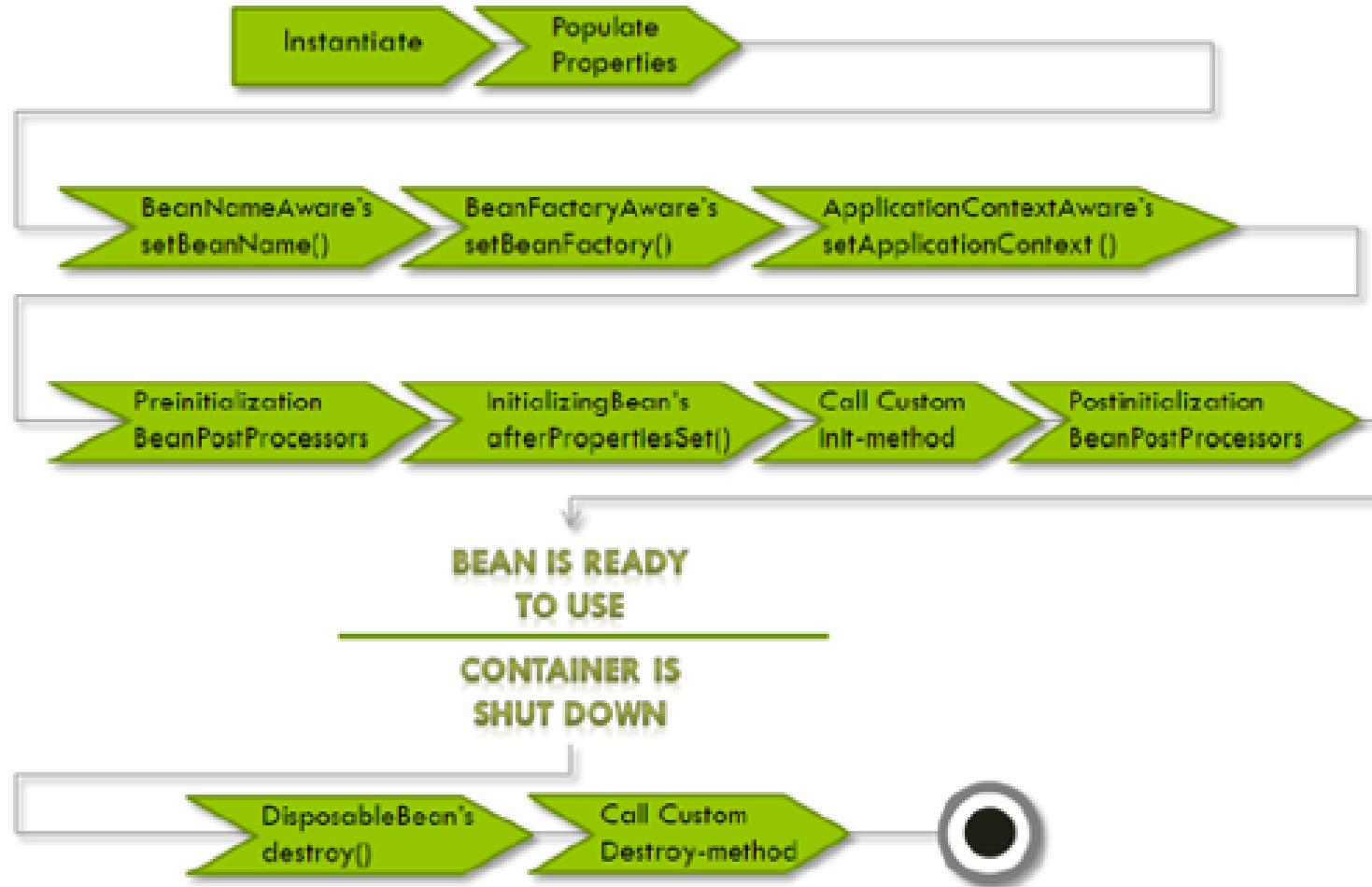
## 다른 스코프의 빈 주입

- 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 세가지 단계로 구분할 수 있다.
  - i. 빈 초기화 단계(initialization)
  - ii. 빈 사용 단계(activation)
  - iii. 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 BeanPostProcessor인터페이스를 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 destory-method와 함께 선언됐으면 지정된 메소드가 호출된다.

## 생명주기 관련 애노테이션

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

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

- `@PreDestroy` - JSR-250스펙에서 정의되어 있으며 종료될 때 사용할 메소드 위에 사용하면 된다.
- `@Bean(destroyMethod)` - `@Bean(destroyMethod="destroy")`와 같은 형태로 사용함으로써 종료될때 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 분할하는 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

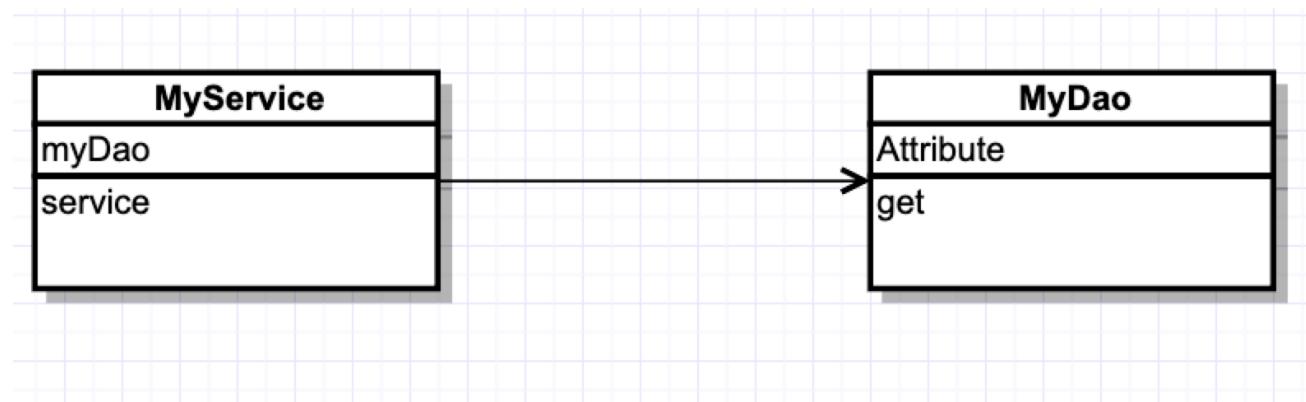
- 자바 명령행 옵션으로 프로파일을 지정하는 방법  
-Dspring.profiles.active=real
- 환경 변수로 프로파일을 지정하는 방법  
export SPRING\_PROFILES\_ACTIVE=real
- web.xml 파일에 프로파일을 지정하는 방법

```
<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.

# MyService & MyDao 클래스다이어그램

- 연관관계
  - MyService가 MyDao를 가진다.



# MyService & MyDao

- 프로그래머가 직접 인스턴스를 생성하고 주입하는 방법
- setter주입

```
MyService myService = new MyService();
MyDao myDao = new MyDao();
myService.setMyDao(myDao);
```

- 생성자에 주입

```
MyService myService = new MyService(new MyDao());
```

## Spring 설정으로 주입

```
MyService myService = new MyService();
MyDao myDao = new MyDao();
myService.setMyDao(myDao);
```

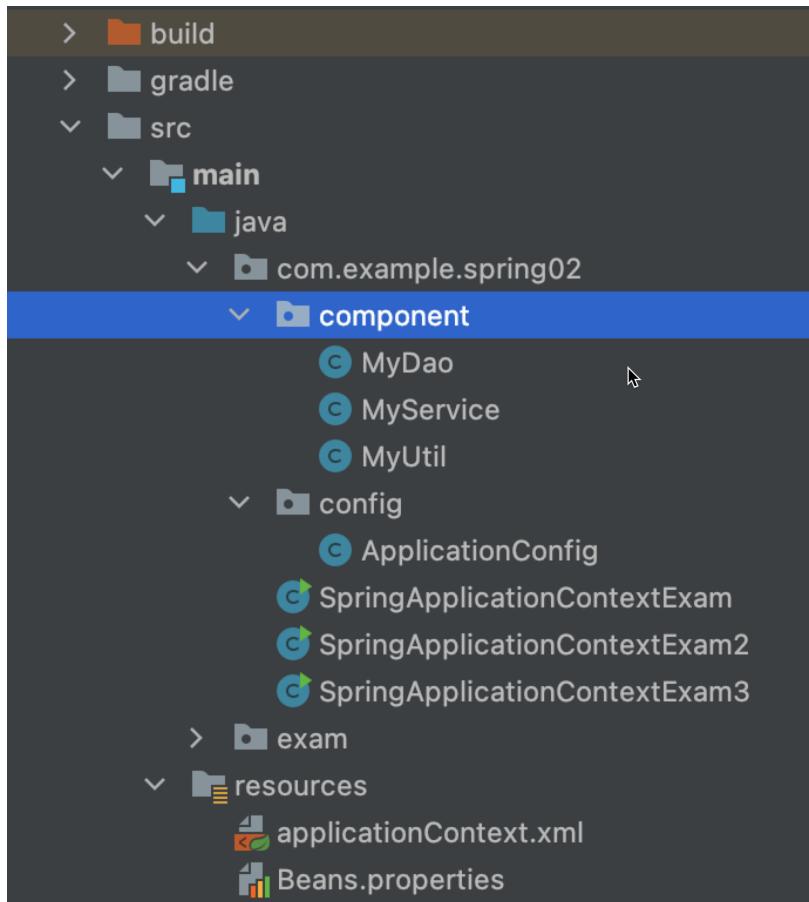
```
<bean id="myService" class="com.example.spring02.component.MyService">
    <!-- setMyDao -->
    <property name="myDao" ref="myDao"></property>
</bean>
<bean id="myDao" class="com.example.spring02.component.MyDao"></bean>
```

# AnnotationConfigApplicationContext

- Spring 3.0부터 등장
- Annotation기반 (Java Config, Component Scan)

```
org.springframework.context.annotation  
Class AnnotationConfigApplicationContext
```

# 실습 예제 프로젝트 구조



# ApplicationConfig.java

```
package com.example.spring02.config;

import com.example.spring02.component.MyDao;
import com.example.spring02.component.MyService;
import com.example.spring02.component.MyUtil;
import exam.Book;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

// Java Config 설정을 AnnotationConfigApplicationContext 읽어들인다.
// ApplicationConfig에 대한 인스턴스를 만든다.
@Configuration
public class ApplicationConfig {
    public ApplicationConfig(){
        System.out.println("ApplicationConfig()");
    }

    // <bean id="book1" class="exam.Book"></bean>
    // 메소드 이름 : id
    @Bean
    public Book book1(){
        return new Book(); // exam.Book
    }
}
```

```
/*
    <bean id="book2" class="exam.Book">
        <property name="title" value="즐거운 자바"></property>
        <property name="price" value="5000"></property>
    </bean>
*/
@Bean
public Book book2(){
    Book book = new Book();
    book.setTitle("즐거운 자바.");
    book.setPrice(9000);
    return book;
}
```

```
/*
    <bean id="myService" class="com.example.spring02.component.MyService">
        <!-- setMyDao -->
        <property name="myDao" ref="myDao"></property>
    </bean>
*/
@Bean(name = "myService2") // bean id가 myService2
public MyService myService(MyDao myDao){
    MyService myService = new MyService();
    myService.setMyDao(myDao);
    return myService;
}

// <bean id="myDao" class="com.example.spring02.component.MyDao"></bean>
@Bean
public MyDao myDao(){
    return new MyDao();
}

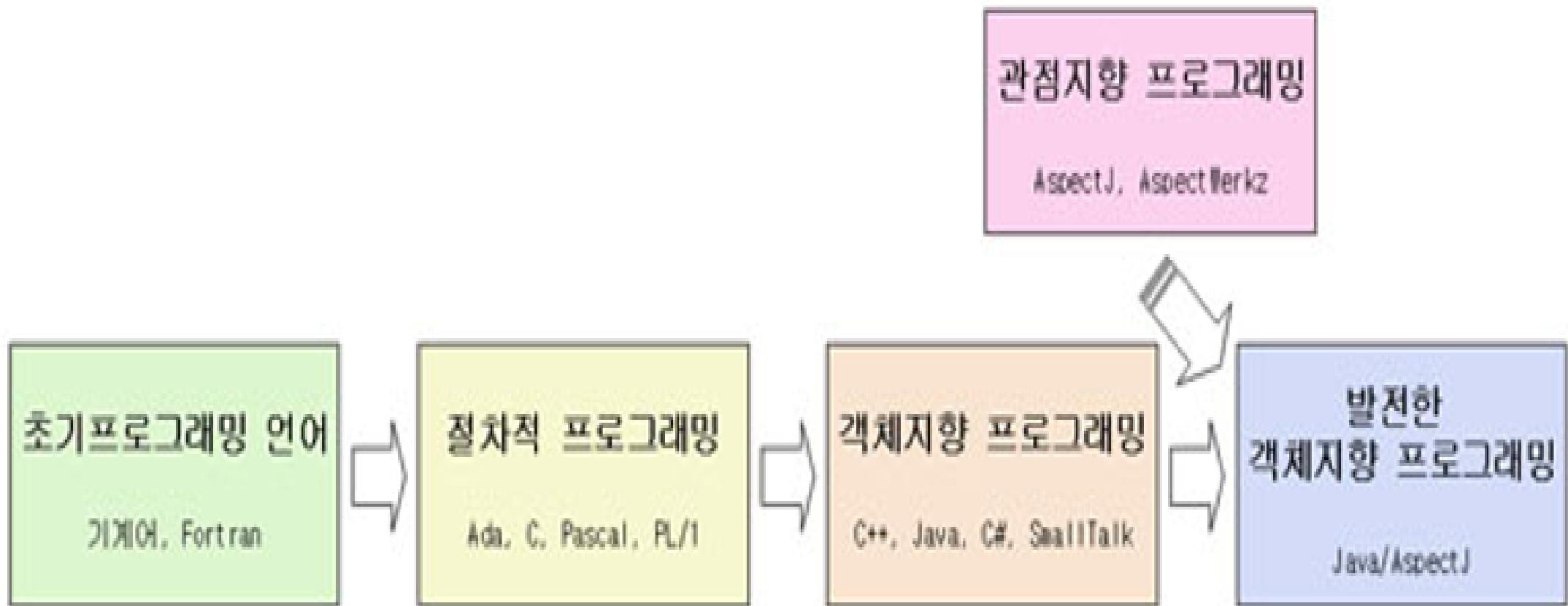
// @Bean
// public MyUtil myUtil(){
//     return new MyUtil();
// }
```

# AOP

## AOP 란?

- 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.

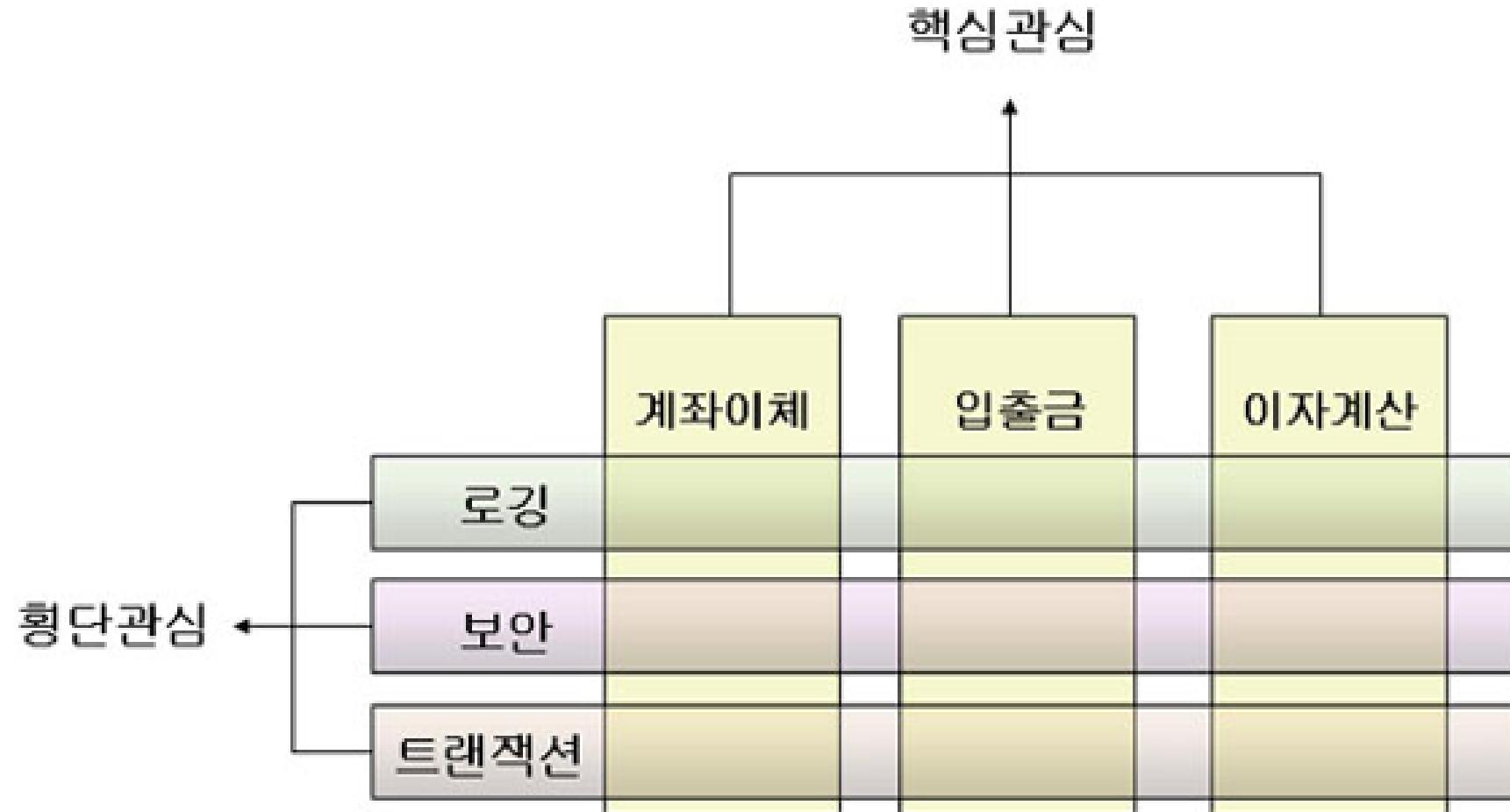
# AOP 란?



## 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 메소드 등을 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.

# 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

- @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

## 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.

# Spring JDBC를 이용한 DAO개발

## DTO란?

- DTO란 Data Transfer Object의 약자이다.
- 계층간 데이터 교환을 위한 자바빈즈이다.
- 여기서의 계층이란 컨트롤러 뷰, 비지니스 계층, 퍼시스턴스 계층을 의미한다.
- 일반적으로 DTO는 로직을 가지고 있지 않고, 순수한 데이터 객체이다.

## DTO의 예

- 필드와 getter, setter를 가진다. 추가적으로 `toString()`, `equals()`, `hashCode()`등의 `Object` 메소드를 오버라이딩 할 수 있다.

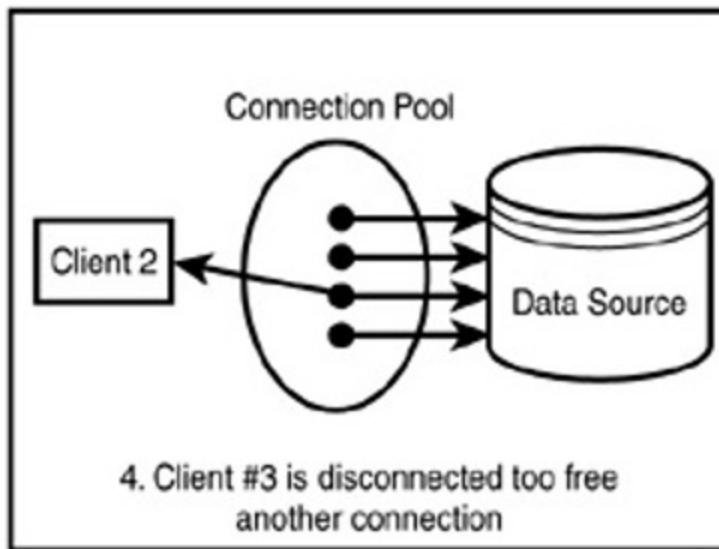
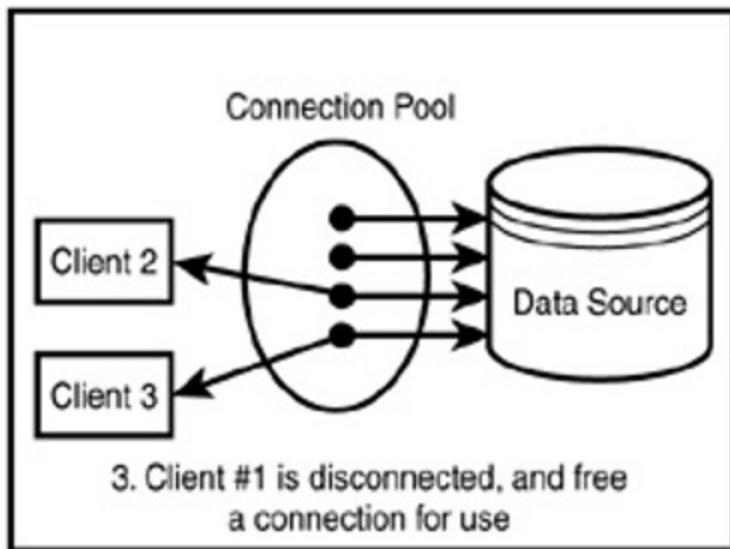
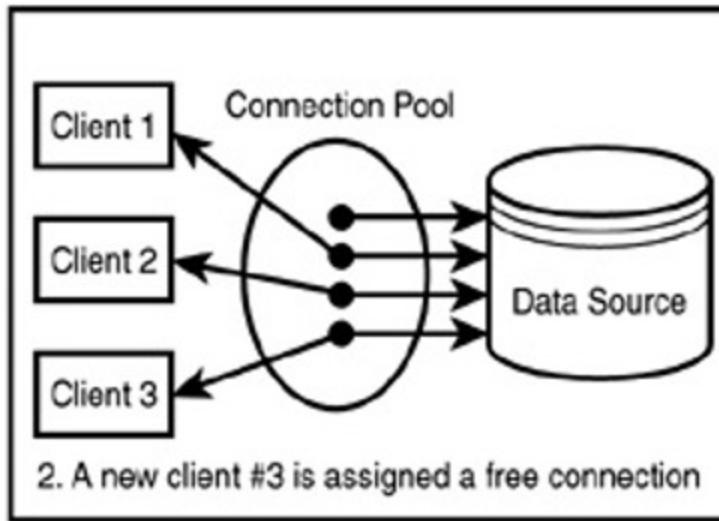
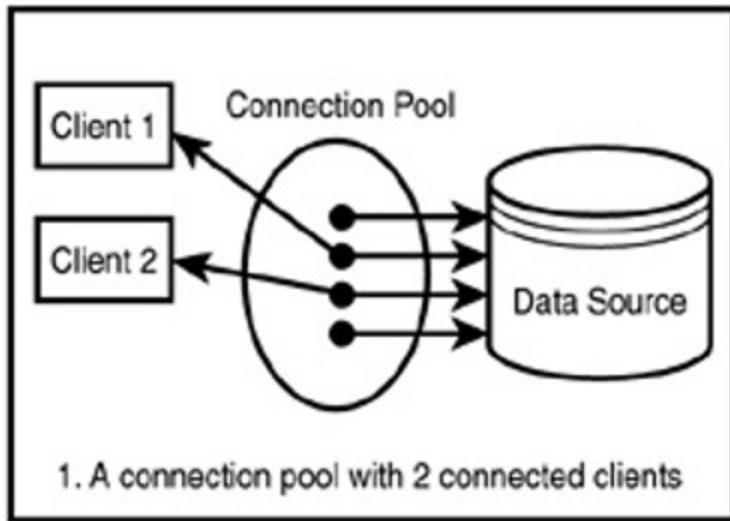
```
public class ActorDTO {  
    private Long id;  
    private String firstName;  
    private String lastName;  
    public String getFirstName() {  
        return this.firstName;  
    }  
    public String getLastName() {  
        return this.lastName;  
    }  
    public Long getId() {  
        return this.id;  
    }  
    // .....
```

## DAO란?

- DAO란 Data Access Object의 약자로 데이터를 조회하거나 조작하는 기능을 전담하도록 만든 객체이다.
- 보통 데이터베이스를 조작하는 기능을 전담하는 목적으로 만들어진다.

## ConnectionPool 이란?

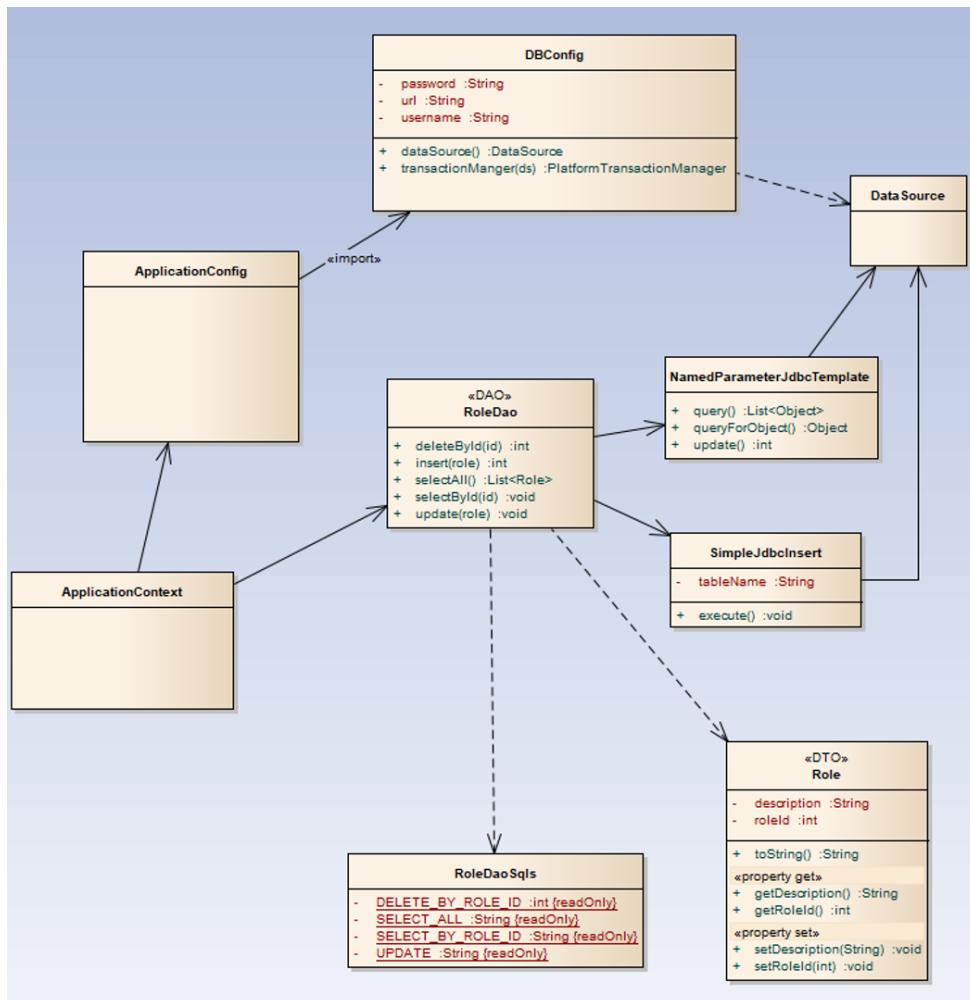
- DB연결은 비용이 많이 듈다.
- 커넥션 풀은 미리 커넥션을 여러 개 맺어 둔다.
- 커넥션이 필요하면 커넥션 풀에게 빌려서 사용한 후 반납한다.
- 커넥션을 반납하지 않으면 어떻게 될까?



## DataSource란?

- DataSource는 커넥션 풀을 관리하는 목적으로 사용되는 객체이다.
- DataSource를 이용해 커넥션을 얻어오고 반납하는 등의 작업을 수행한다.

# Spring JDBC를 이용한 DAO작성 실습

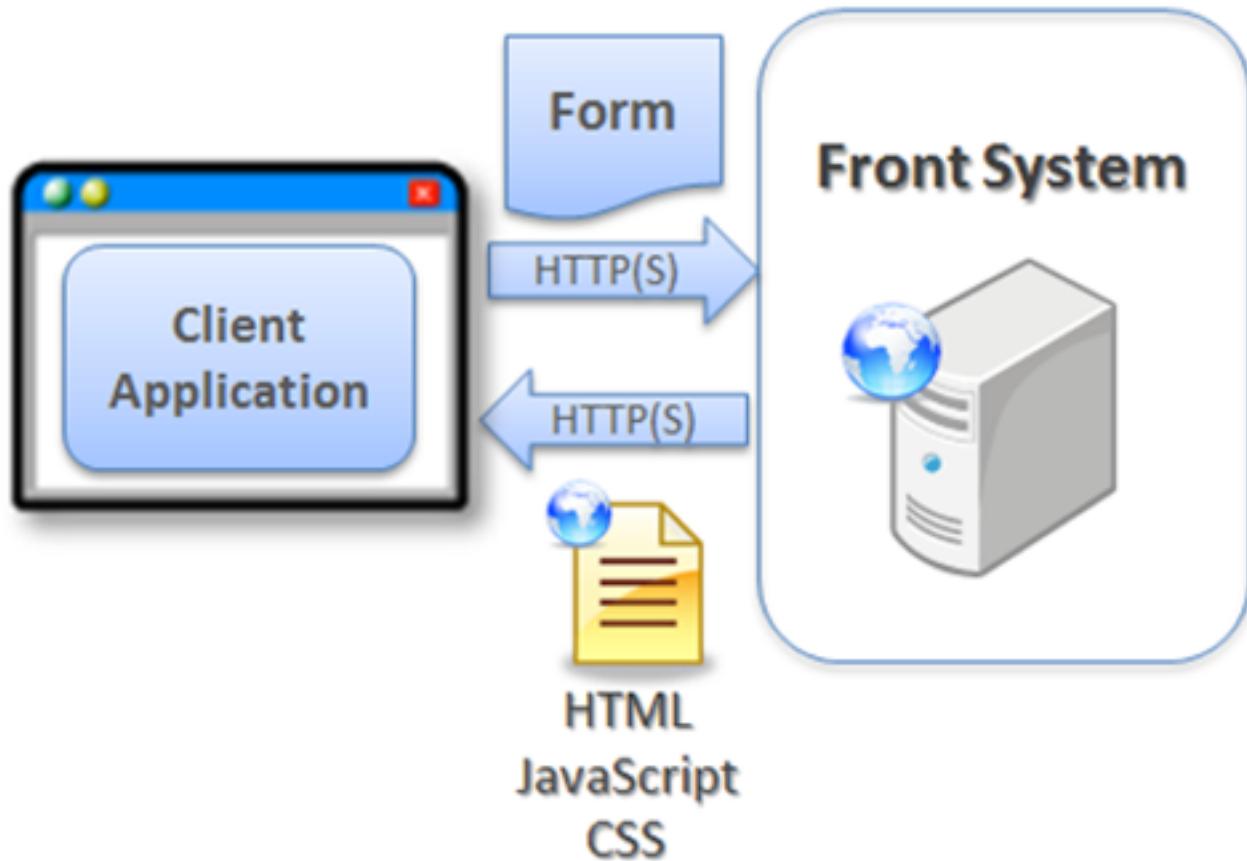


# Spring MVC

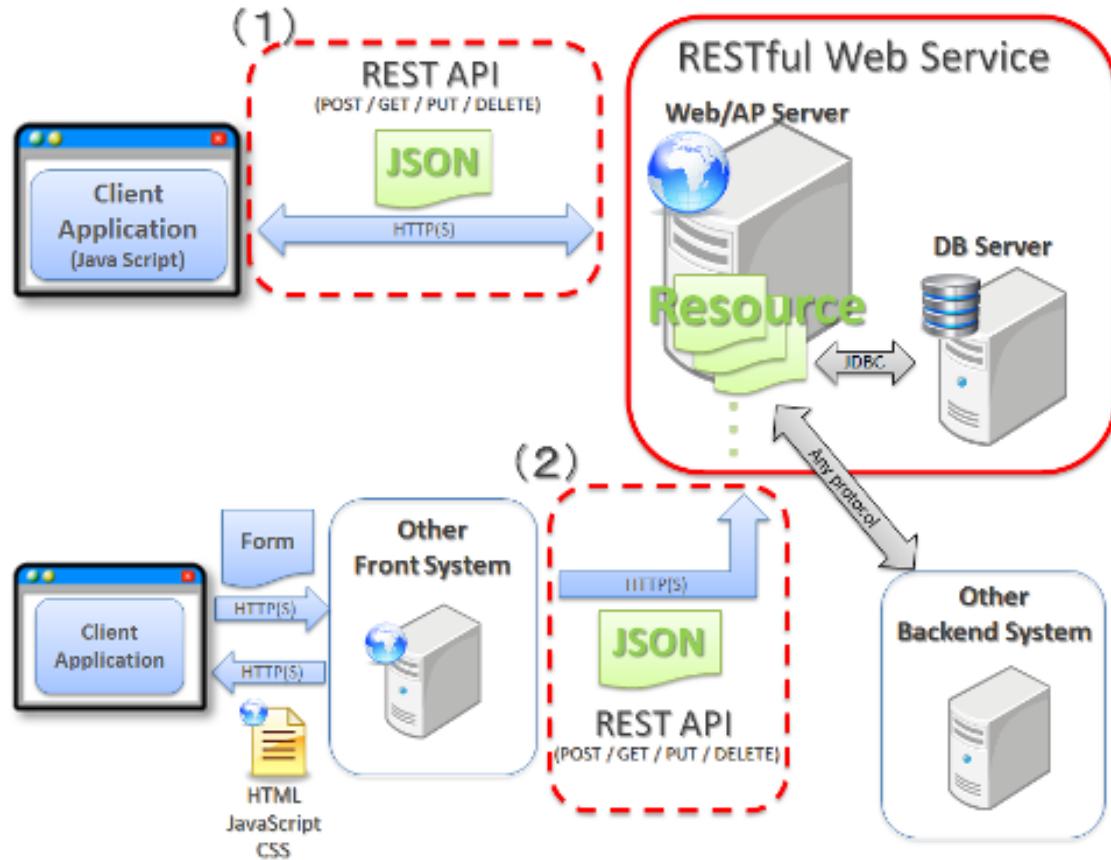
## 웹 애플리케이션의 종류

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

## 화면으로 응답하는 애플리케이션

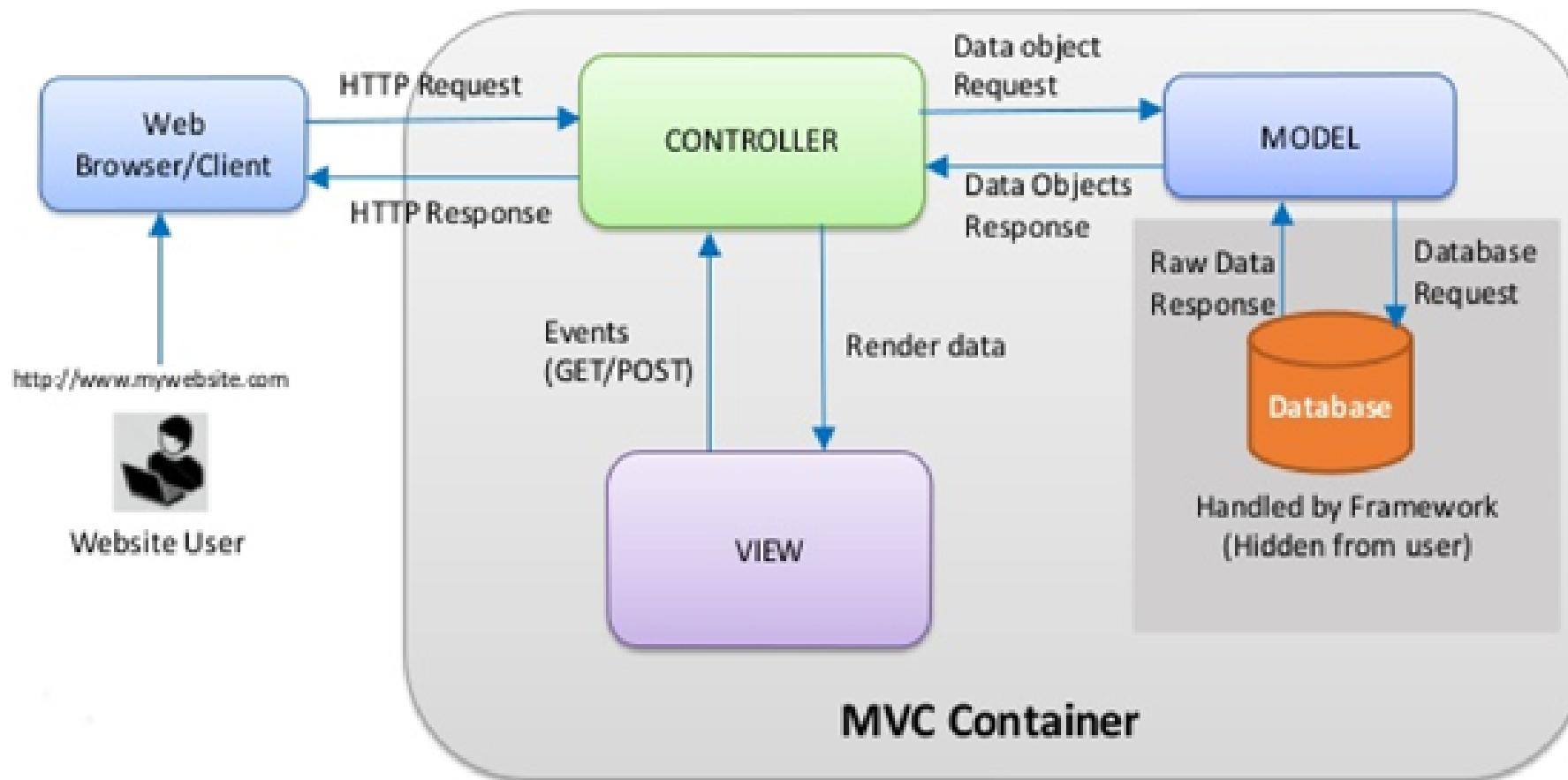


# 데이터로 응답하는 애플리케이션



# MVC

- MVC는 Model-View-Controller의 약자이다.

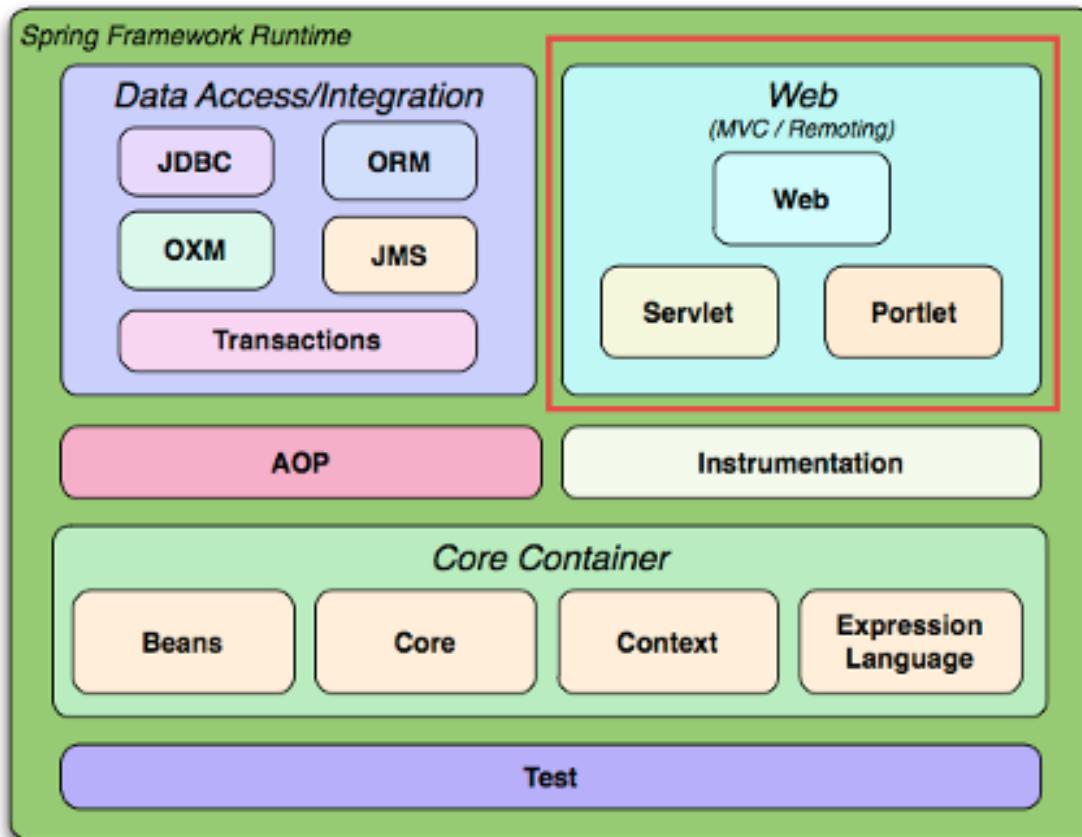


# MVC

- 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

- MVC 패턴을 지원하는 Spring Module



## Web Application

- 웹 애플리케이션(web application) 또는 웹 앱은 소프트웨어 공학적 관점에서 인터넷이나 인트라넷을 통해 웹 브라우저에서 이용할 수 있는 응용 소프트웨어를 말한다.

# 웹 애플리케이션 개발의 특징 1/2

- POJO 구현
  - 컨트롤러나 모델 등의 클래스는 POJO 형태로 구현한다. 특정 프레임워크 등에 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.

## 웹 애플리케이션 개발의 특징 2/2

- 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.

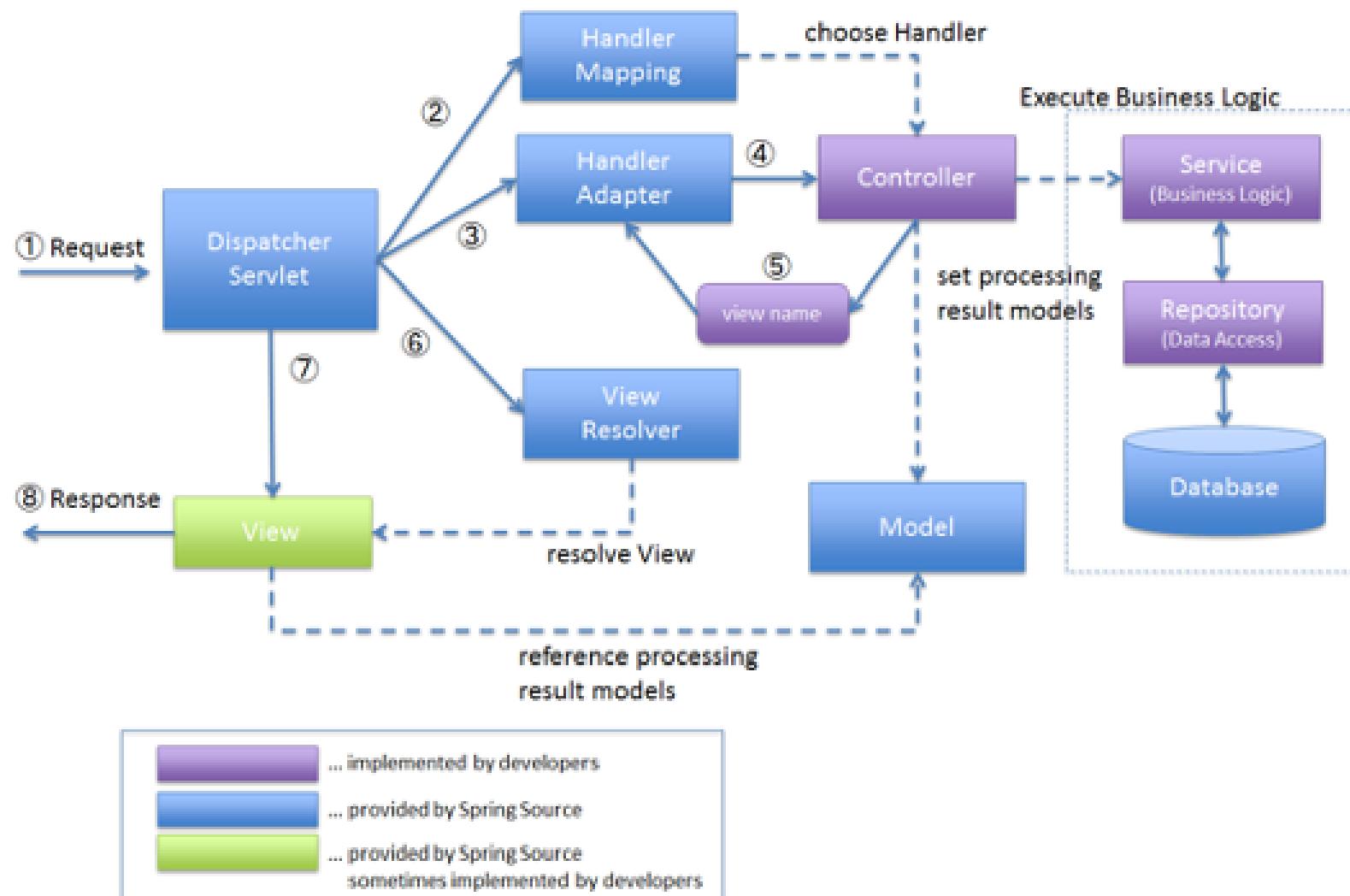
## MVC 프레임워크로서의 특징 1/2

- 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.

## MVC 프레임워크로서의 특징 2/2

- 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 기본 동작 흐름



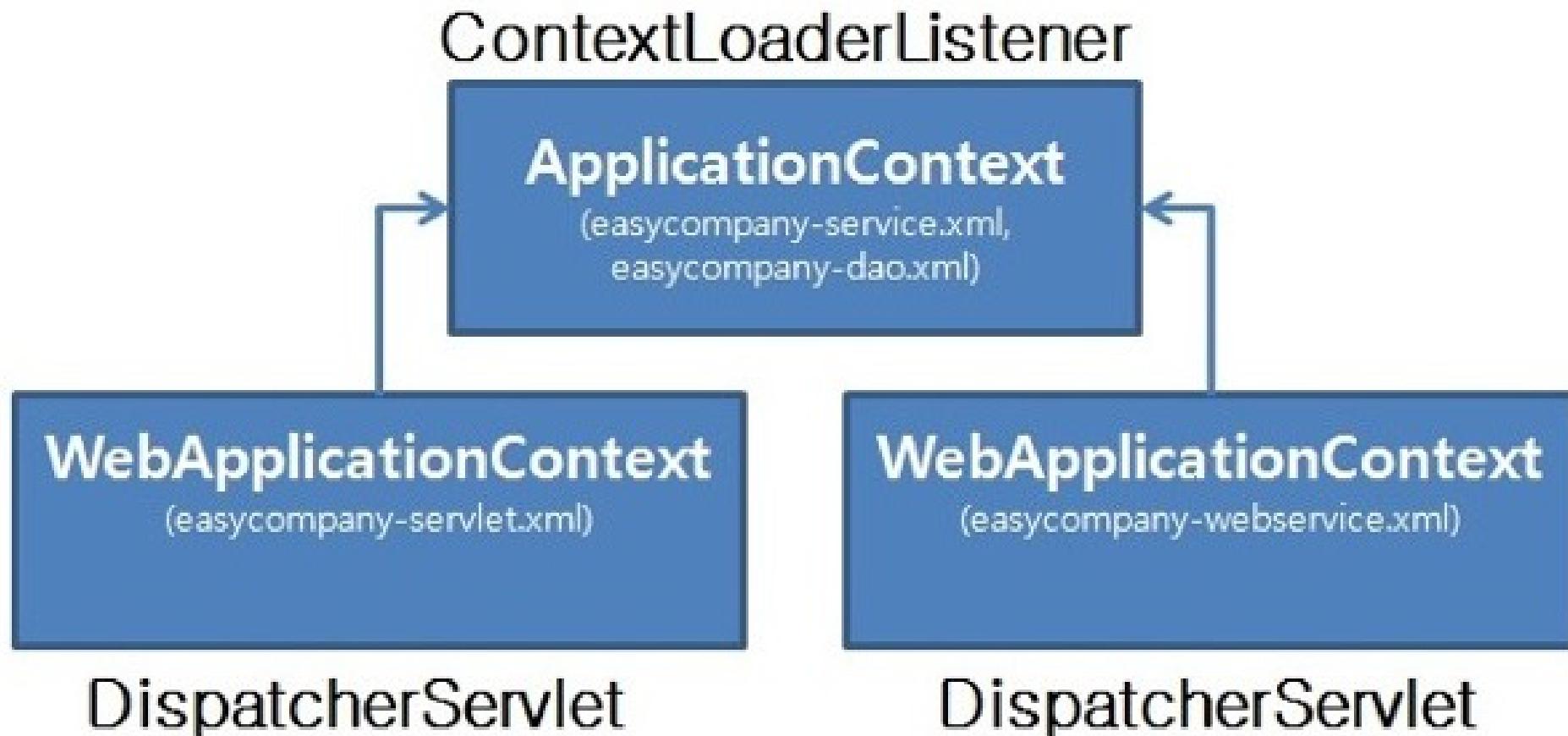
## 요청 처리를 위해 사용되는 컴포넌트

- 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 경우 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.xml 파일에서 DispatcherServlet 설정하기 | 1/2

- xml spring 설정 읽어들이도록 DispatcherServlet 설정

```
<?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 2/2

```
<?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

- com.example.webmvc.config.WebMvcContextConfiguration

```
@Configuration  
@EnableWebMvc  
@ComponentScan(basePackage = { "com.example.webmvc.controller"})  
public class WebMvcContextConfiguration extends WebMvcConfigurerAdapterP{  
    .....  
}
```

## @Configuration

- org.springframework.context.annotation 의 Configuration 애노테이션과 Bean 애노테이션  
코드를 이용하여 스프링 컨테이너에 새 로운 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 핸드러 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 수행 한다.
- 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 오버라이딩하여 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 Get 사이에서 모델을 공유하기 위한 Map 객체다. 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.

## **Controller(Handler) 클래스 작성하기**

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

# @RequestMapping

- Http 요청과 이를 다루기 위한 Controller의 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 경우에 는 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 매핑 조건에 지정하는 경우에는 params속성을 사용한다. params속성에서 지원하는 지정 형식은 다음과 같다.
- name : 지정한 파라미터가 존재하는 경우에 매핑 대상이 된다.
- !name : 지정한 파라미터가 존재하지 않는 경우에 매핑 대상이 된다.
- name=value : 파라미터 값이 지정한 값에 해당하는 경우에 매핑 대상이 된다.
- name!=value : 파라미터 값이 지정한 값에 해당하지 않는 경우 매핑 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 경우에는 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 대상이 된다.

## @RequestMapping – Accept 헤더 사용

- 요청 Accept 헤더 값을 매팅 조건에 지정하는 경우 produces속성을 사용한다. produces속성에서 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 1/3

- @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.

## 인수에 지정 가능한 애노테이션 2/3

- `@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 3/3

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

## 관례에 따른 묵시적인 인수 값 결정

- 다음 규칙에 따라 요청 정보의 값을 인수의 값으로 채워 넣을 수 있다.
  - 인수의 타입이 String이나 Integer같은 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.

## 컨트롤러 핸들러 메소드 리턴 값 1/2

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

## 컨트롤러 핸들러 메소드 리턴 값 2/2

- java.lang.String
  - 이동 대상의 뷰 이름을 반환한다.
- void
  - HttpServletResonse에 직접 응답 데이터를 쓰거나 RequestToViewNameTranslator의 매커니즘을 이용해 뷰 이름을 결정할 때 void를 사용한다.
- org.springframework.http.HttpEntity<?>
- org.springframework.http.ResponseEntity<?>
  - 응답 헤더와 응답 본문에 직렬화된 객체를 반환한다. 반환한 객체는 HttpMessageConverter매커니즘을 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, StreamResponseBody형을 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.Valid`를 지정한다. `@Validated`를 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 공백 문자를 설정한다. '미입력은 허용하지만 만약 입력이 되었다면 최소한 6자 이상일 것' 이라는 요구사항을 Bean Validation 표준 애노테이션으로는 충족시킬수가 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 `null`로 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 검사할 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.

# 레이어드 아키텍처 - Controller, Service, Repository 그리고 Transaction

# 웹 페이지는 중복 개발되는 요소가 존재한다.

The screenshot shows the NAVER Zikin homepage. At the top, there's a search bar and a navigation menu with links like '질문하기', 'Q&A', '오픈마켓', etc. Below the header, there are several promotional banners: one for '질문하기' with a question mark icon, another for '자식의 힘' with a smiling face, and a third for '자식의 힘 vs 즐거운 힘'. A large green button labeled '질문하기' is prominent. On the left, there's a sidebar with '질문마을' and other categories. The main content area features a section titled '질문마을' with a red border around it, containing three questions: 1. '강수지 첫번째남편' (with 10 votes), 2. '나 자신에게 상을 하나 줄 수 있다면?' (with 10 votes), and 3. '노후 경유차 운행제한 질문' (with 2 votes). To the right of this is a '2018 GLOBAL V LIVE TOP10 MONSTA X' banner.

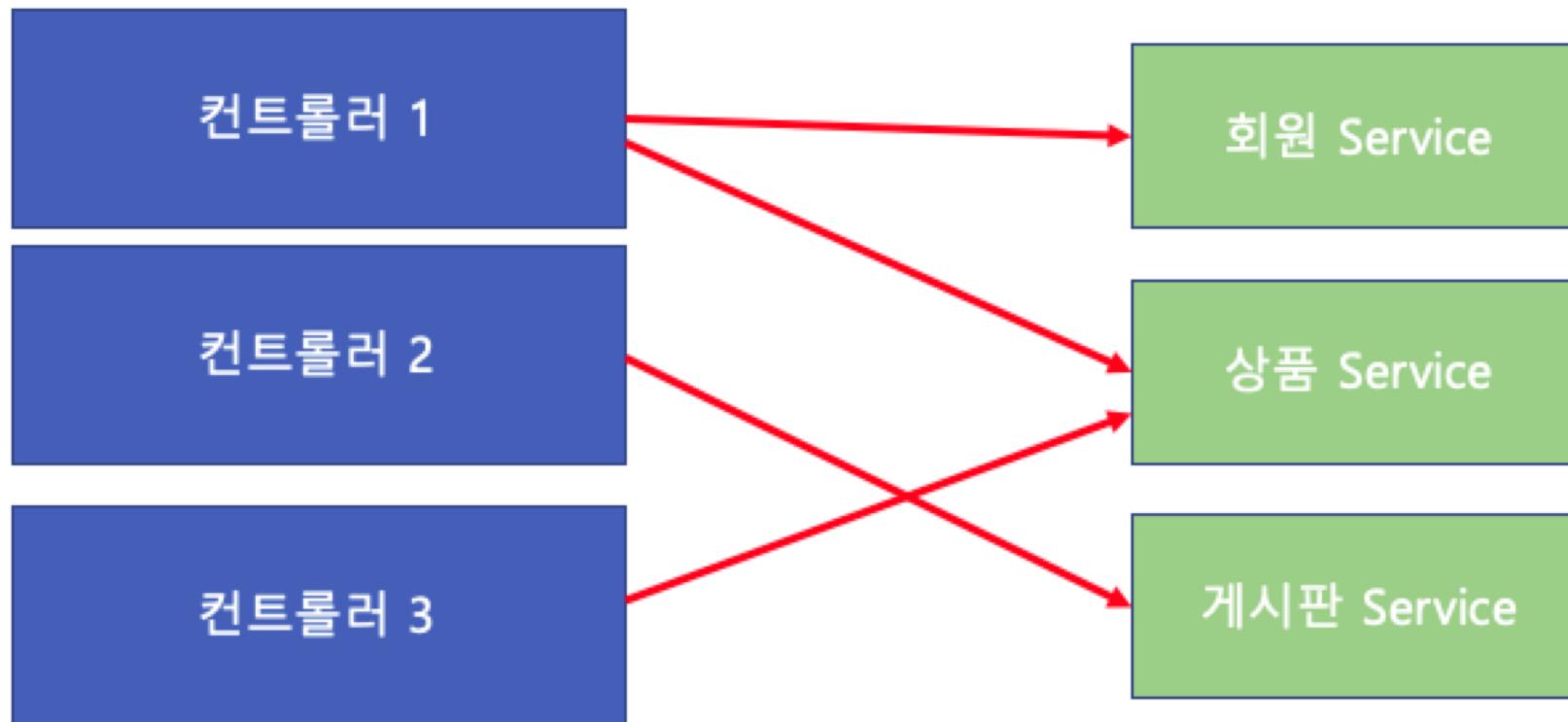
This screenshot shows a detailed view of a question from the '질문하기' section. The question is '노후 경유차 운행제한 질문' (Is there a restriction on driving electric vehicles in old age?). It includes a profile picture of the asker, their location (Seoul, South Korea), and a timestamp (2018-08-28 20:59). A blue circular badge says '경원도가 물어보았다'. Below the question, there's a response from the asker: '제 자녀는 2005년 3월에 구매한 소형트럭(2005) 경유차입니다. 차량등록증을 놓니 경16kg이라고 되어있네요. 노후 경유차 기사를 접해보면 2005년 이전 2.5톤 경유자는 2017~2020년 서울, 인천, 경기에서 할 수 없다고 하던데요 그동안 차의 경우 2021년부터 운행을 못한다는 얘기인가요?'. There are also sections for '답변을 기다리는 질문' and '질문자 자격된 경우, 추가 답변 등록이 불가합니다.'

## 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.
  - 잔액이 얼마인지 조회한다.
  - 출금하려는 금액이 잔액보다 작은지 검사한다.
  - 출금하려는 금액이 잔액보다 작다면 (잔액 - 출금액)으로 수정한다.
  - 언제, 어디서 출금했는지 정보를 기록한다.
  - 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 완료될때까지, 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

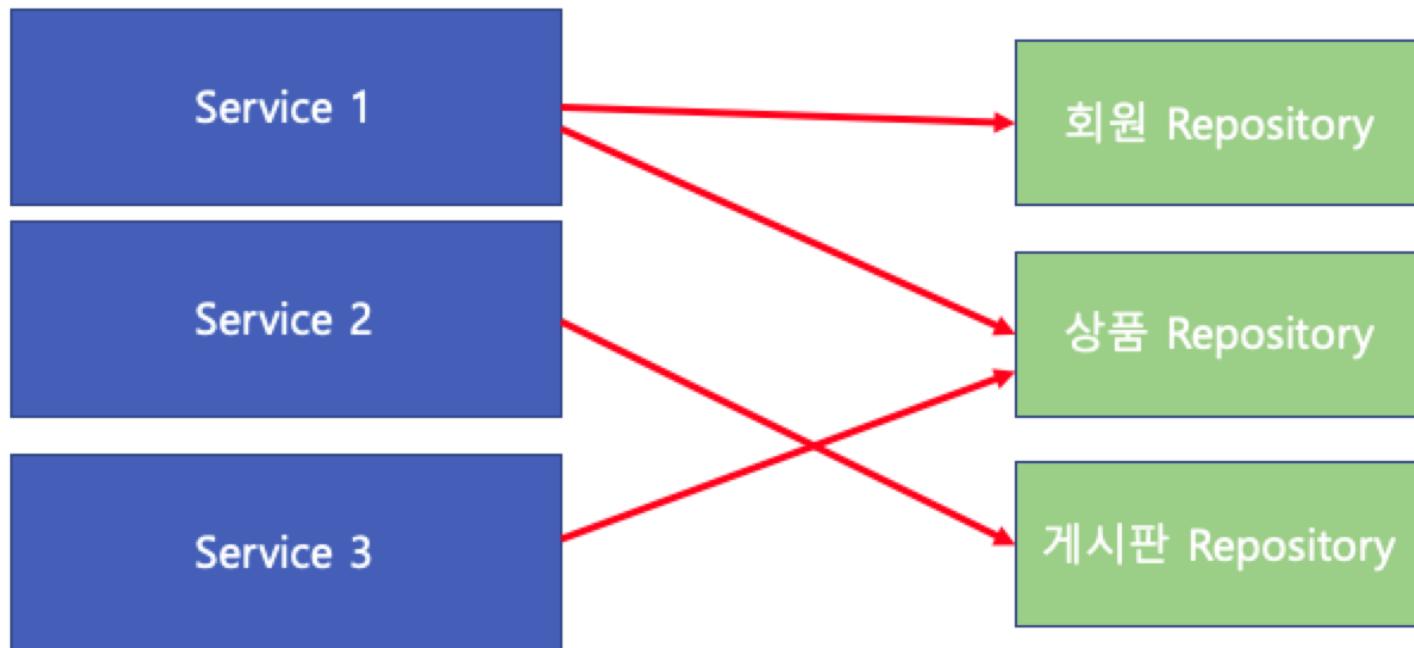
- DB에 연결된 후 Connection객체의 setAutoCommit메소드에 false를 파라미터로 지정한다.
- 입력,수정,삭제 SQL이 실행을 한 후 모두 성공했을 경우 Connection이 가지고 있는 commit() 메소드를 호출한다.

## @EnableTransactionManagement

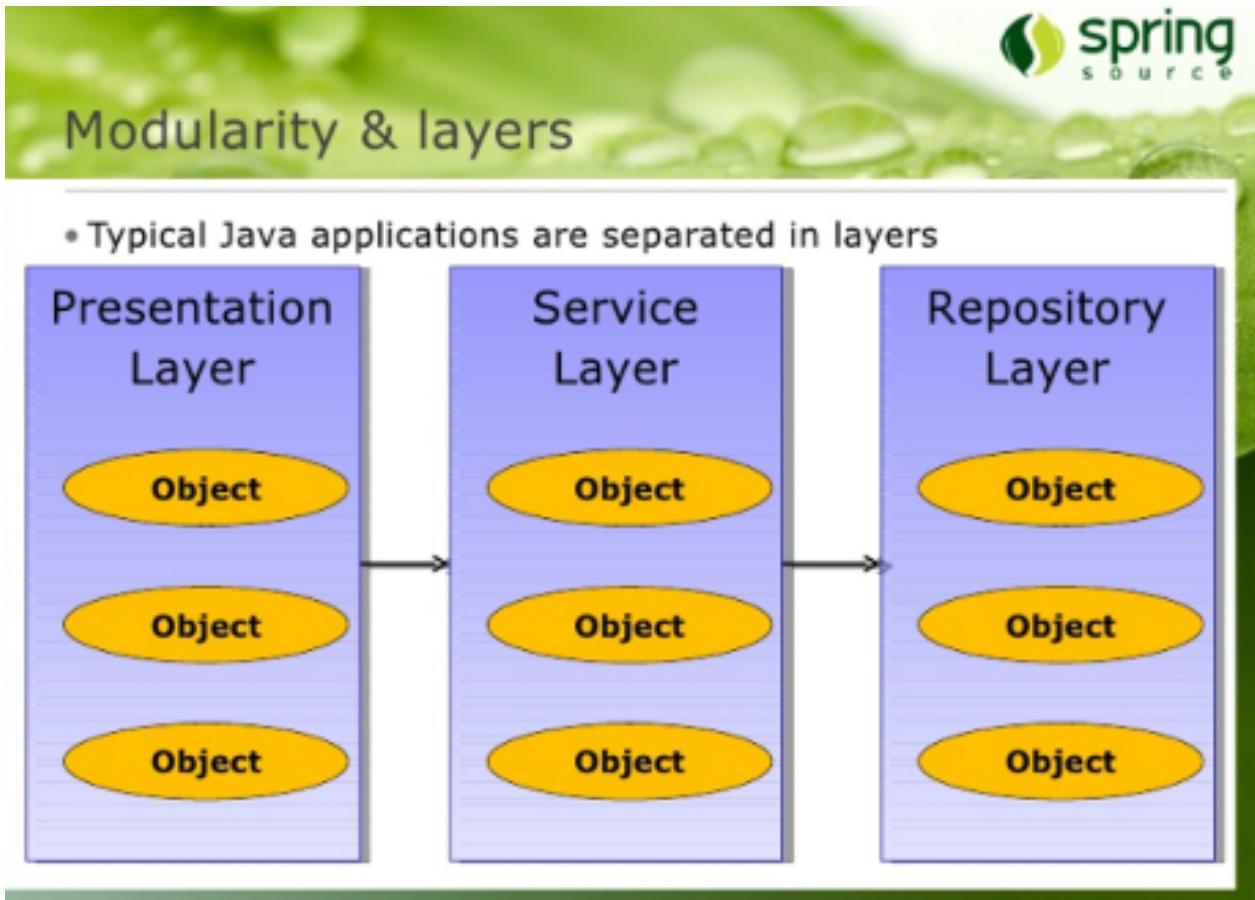
- Spring Java Config파일에서 트랜잭션을 활성화 할 때 사용하는 애노테이션
- Java Config를 사용하게 되면 PlatformTransactionManager 구현체를 모두 찾아서 그 중에 하나를 매핑해 사용한다.
- 특정 트랜잭션 메니저를 사용하고자 한다면 TransactionManagementConfigurer를 Java Config파일에서 구현하고 원하는 트랜잭션 메니저를 리턴하도록 한다.
- 아니면, 특정 트랜잭션 메니저 객체를 생성시 @Primary 애노테이션을 지정한다.

## 서비스 객체에서 중복으로 호출되는 코드의 처리

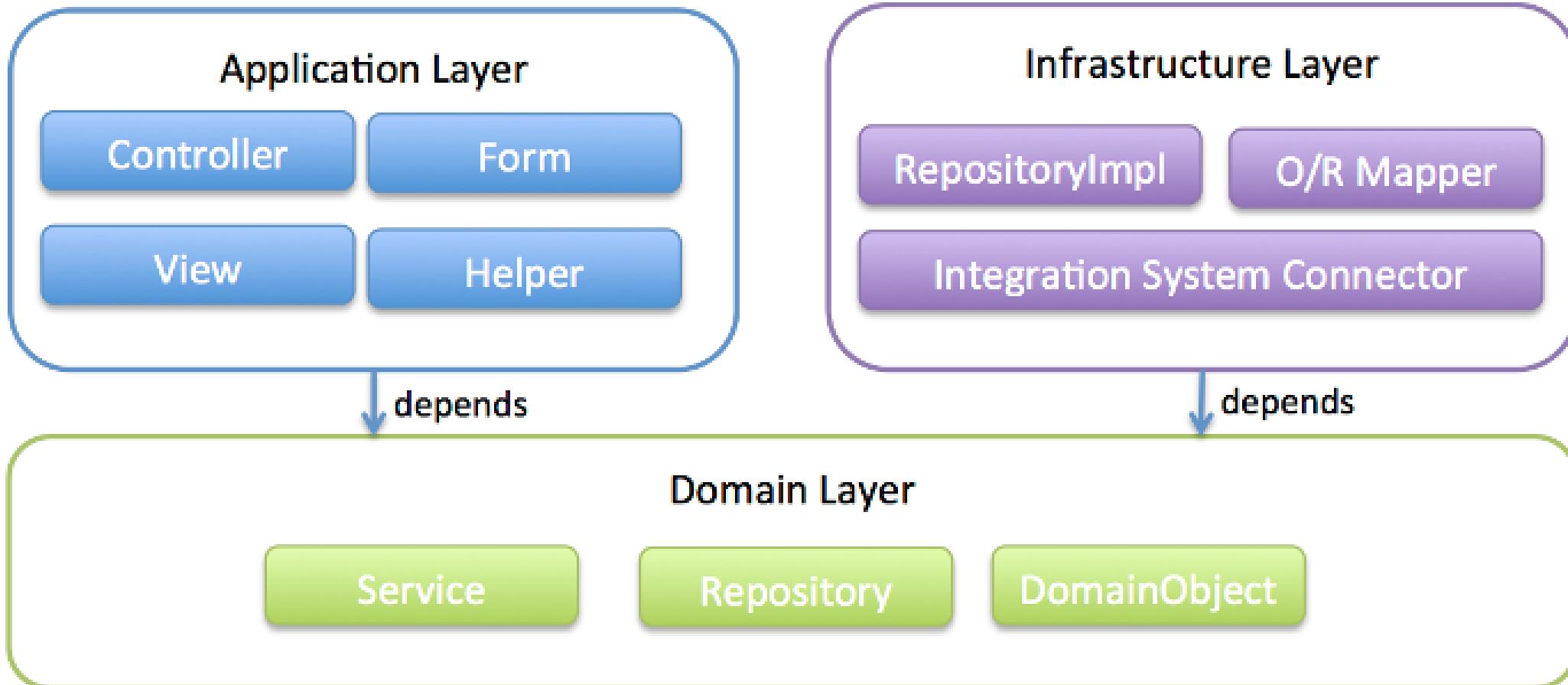
- 데이터 액세스 메소드를 별도의 Repository(Dao) 객체에서 구현하도록 하고 서비스는 Repository 객체를 사용하도록 한다.



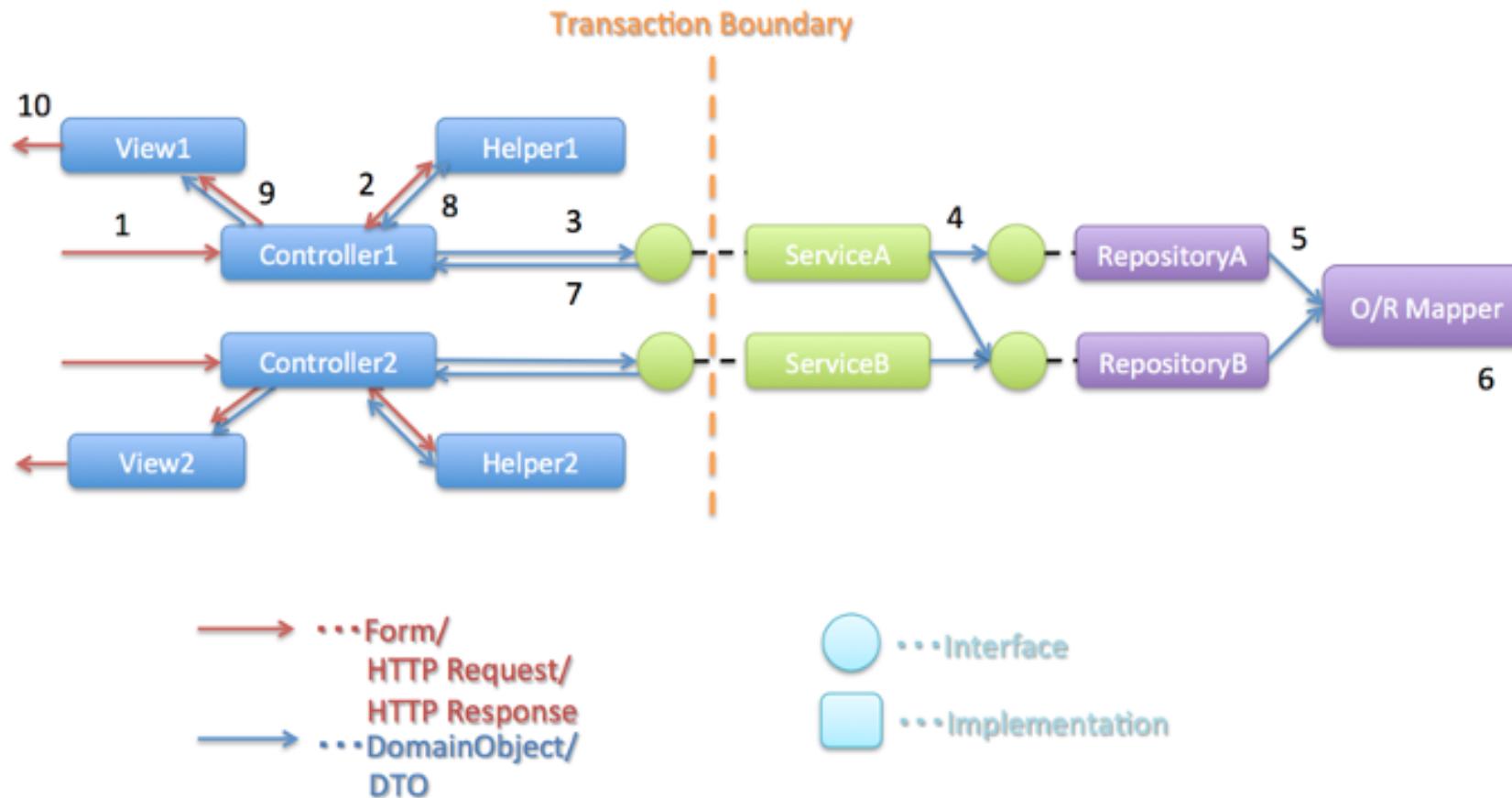
# 레이어드 아키텍처 1/3



## 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.

**Web API (Rest API에 대한 규칙은 너무 지키기 어려워)**

## @RestController

- Spring 4에서 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
- @EnableWebMvc로 인한 기본 설정.
  - WebMvcConfigurationSupport를 사용하여 Spring MVC 구현을 하고 있음.
  - Default MessageConvertor를 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.

# Cookie & Session

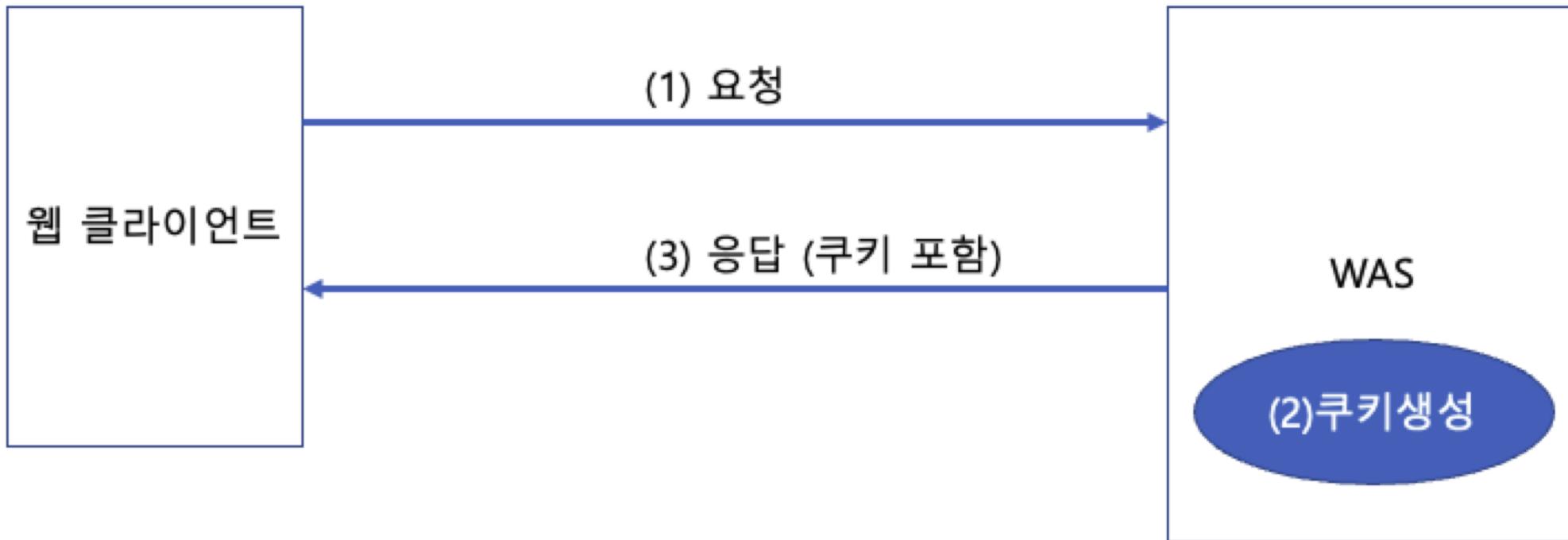
## 웹에서의 상태 유지 기술

- 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.

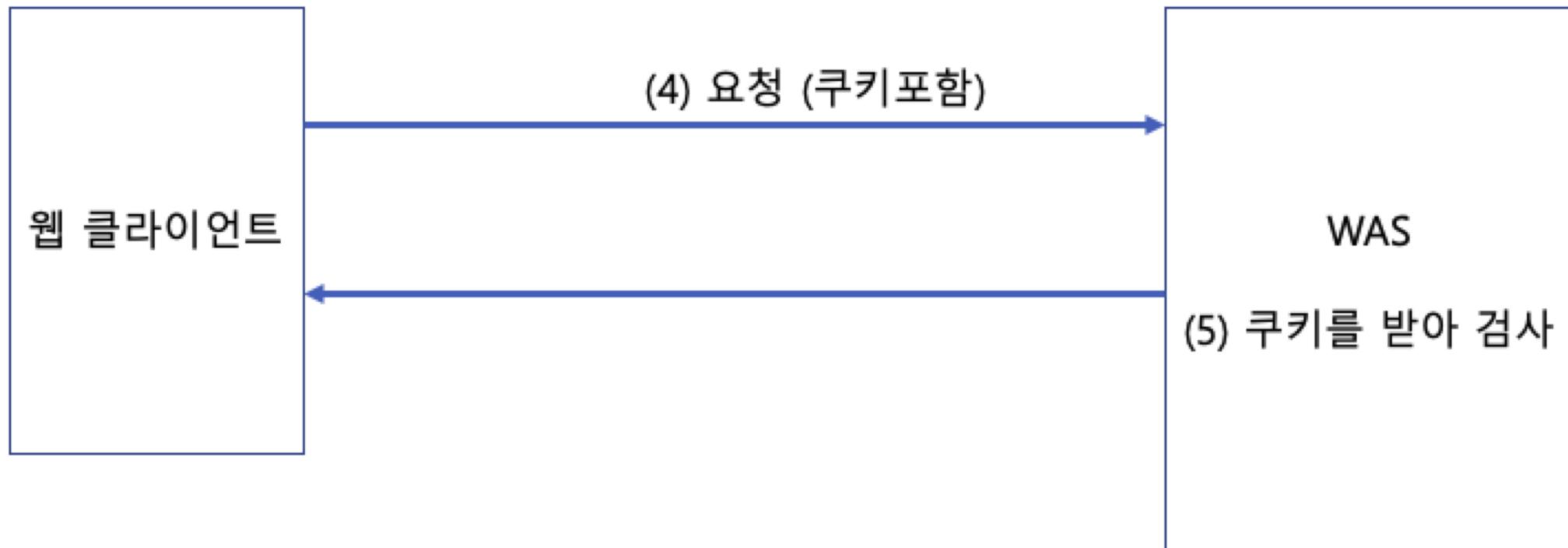
# 쿠키(Cookie)와 세션(Session)

- 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

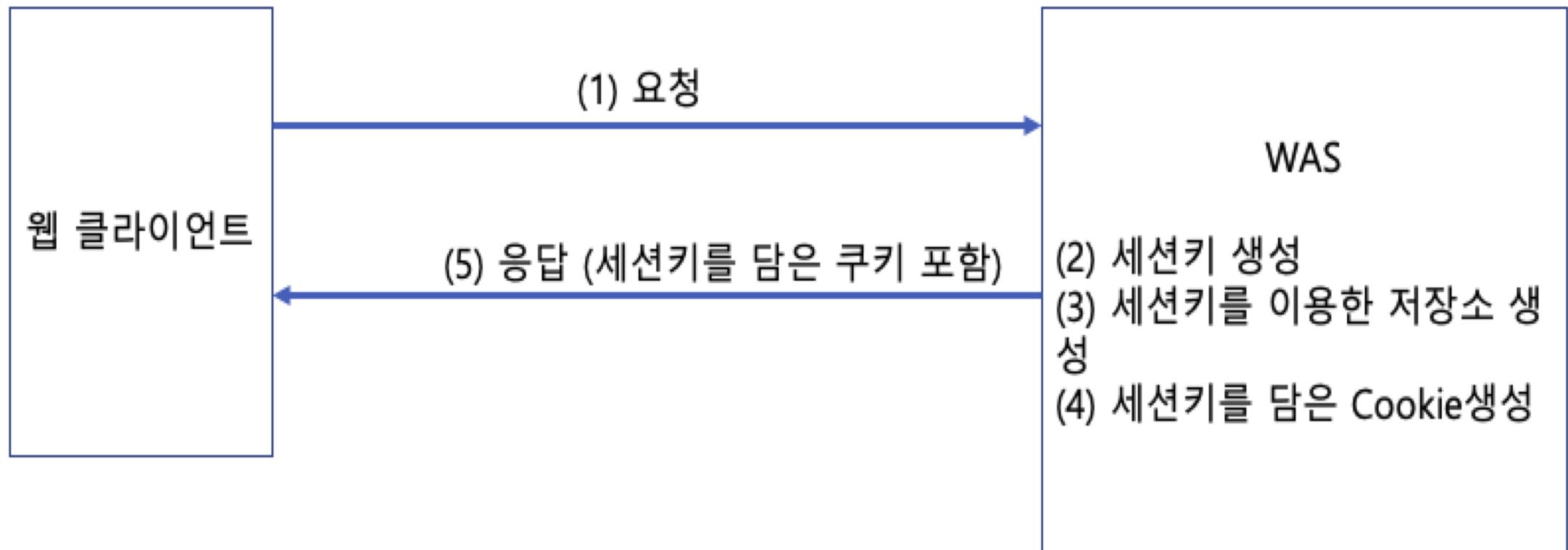
## 쿠키(Cookie) 동작 이해 1/2



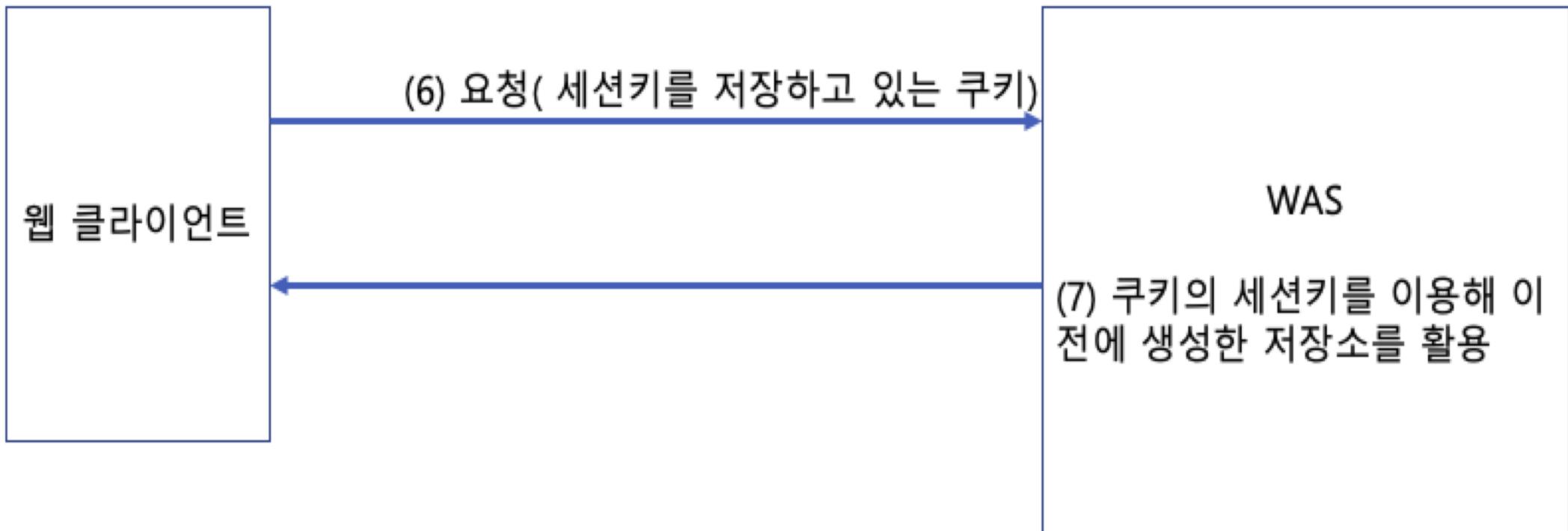
## 쿠키(Cookie) 동작 이해 2/2



## 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 이용해서 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  $(72460*60)$ 로 설정한다.

# javax.servlet.http.Cookie

반환형	메소드 이름	메소드 기능
int	<u>getMaxAge()</u>	쿠키의 최대지속 시간을 초단위로 지정 -1 일 경우 브라우저가 종료되면 쿠키를 만료
String	<u>getName()</u>	쿠키의 이름을 스트링으로 반환
String	<u>getValue()</u>	쿠키의 값을 스트링으로 반환
void	<u>setValue(String newValue)</u>	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)`
  - `name`과 `value`의 쌍으로 객체 `Object`를 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	<code>getCreationTime()</code>	를 세션이 생성된 시간까지 지난 시간을 계산하여 밀리세컨드로 반환
String	<code>getId()</code>	세션에 할당된 유일한 식별자(ID)를 String 타입으로 반환
int	<code>getMaxInactiveInterval()</code>	현재 생성된 세션을 유지하기 위해 설정된 최대 시간을 초의 정수 형으로 반환, 지정하지 않으면 기본 값은 1800초, 즉 30분이며, 기본 값도 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	<code>getAttribute(String name)</code>	name이란 이름에 해당되는 속성값을 Object 타입으로 반환, 해당되는 이름이 없을 경우에는 null을 반환
Enumeration	<code>getAttributeNames()</code>	속성의 이름들을 Enumeration 타입으로 반환
void	<code>invalidate()</code>	현재 생성된 세션을 무효화 시킴
void	<code>removeAttribute(String name)</code>	name으로 지정한 속성의 값을 제거
void	<code>setAttribute(String name, Object value)</code>	name으로 지정한 z이름에 value 값을 할당
void	<code>setMaxInactiveInterval(int interval)</code>	세션의 최대 유지시간을 초 단위로 설정
boolean	<code>isNew()</code>	세션이 새로이 만들어졌으면 true, 이미 만들어진 세션이면 false를 반환

## **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 `form` 태그로 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>
```

# Swagger

- swagger정리함. swagger관련 팁은 이 문서에 계속 기록해주세요. [@김준형@오세진](#)

## Swagger란?

Swagger란 서버로 요청되는 URL 리스트를 HTML화면으로 문서화 및 테스트 할 수 있는 라이브러리이다. 간단하게 설명하면 Swagger는 API Spec 문서이다. API를 엑셀이나 가이드 문서를 통해 관리하는 방법은 주기적인 업데이트가 필요하기 때문에 관리가 쉽지 않고 시간이 오래 걸린다. 그래서 Swagger를 사용해 API Spec 문서를 자동화해주어 간편하게 API문서를 관리하면서 테스트할 수 있다.

## Swagger 설정하기

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

# Swagger 설정하기

```
@Configuration  
@EnableSwagger2  
@SuppressWarnings("unchecked") // warning 밑줄 제거를 위한 태그  
public class SwaggerConfig {  
  
    @Bean  
    public Docket todosApi() {  
        return getDocket("todos", Predicates.or(  
            PathSelectors.regex("/api/todos/.*")));  
    }  
}
```

```
@Bean
public Docket allApi() {
    return getDocket("전체", Predicates.or(
        PathSelectors.any()));
}

//swagger 설정.
public Docket getDocket(String groupName, Predicate<String> predicate) {
    return new Docket(DocumentationType.SWAGGER_2)
        .groupName(groupName)
        .select()
        .apis(RequestHandlerSelectors.basePackage("examples.springmvc"))
        .paths(predicate)
        .apis(RequestHandlerSelectors.any())
        .build();
}
```

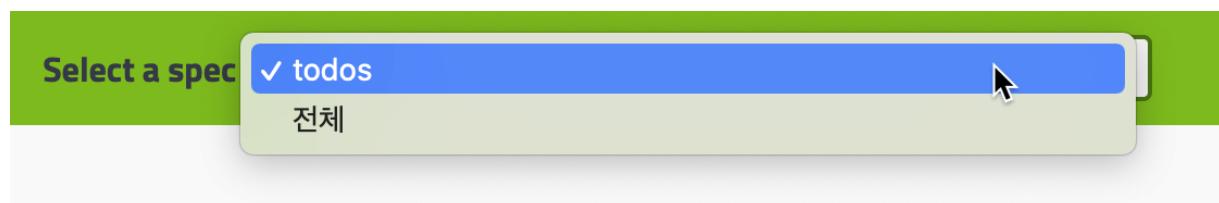
# Swagger Config 설명

- .consume()과 .produces()는 각각 Request Content-Type, Response Content-Type에 대한 설정. (선택)
- .apiInfo()는 Swagger API 문서에 대한 설명을 표기하는 메소드. (선택)
- .apis()는 Swagger API 문서로 만들기 원하는 BasePackage 경로. (필수)
- .path()는 URL 경로를 지정하여 해당 URL에 해당하는 요청만 Swagger API 문서로 만든다. (필수)
  - /api/todos/.\*: 정규표현식 . 은 문자하나를 말한다..\*은 어떤 문자든 올 수 있다는 것을 의미한다.
    - /api/todos/100 , api/todos/list 등

# Swagger 확인하기

<http://localhost:8080/webtodoserver/swagger-ui.html#/todo-controller>

The screenshot shows the Swagger UI interface for the 'todos' API. At the top, there is a navigation bar with the 'swagger' logo and a dropdown menu labeled 'Select a spec' with 'todos' selected. Below the navigation bar, the title 'Api Documentation 1.0' is displayed, along with the base URL information: [ Base URL: localhost:8080/webtodoserver ] and the API documentation URL: <http://localhost:8080/webtodoserver/v2/api-docs?group=todos>. Under the title, there are links for 'Api Documentation', 'Terms of service', and 'Apache 2.0'. The main content area is titled 'todos Todo Controller'. It lists four API endpoints: 'GET /api/todos getTodos' (blue background), 'POST /api/todos addTodo' (green background), 'PUT /api/todos updateTodo' (orange background), and 'DELETE /api/todos deleteTodo' (red background). At the bottom of the main content area, there is a 'Models' section with a right-pointing arrow. The entire screenshot is framed by a red border.



- 실행시키고 싶은 API를 선택하고 "Try it out" 버튼을 클릭한다.
- Execute 버튼을 클릭한다.

**todos** Todo Controller

GET /api/todos getTodos

Parameters

No parameters

Responses

Response content type \*/\*

Code	Description
200	OK
	Example Value   Model
	[ { "done": true, "id": 0, "todo": "string" } ]
401	Unauthorized
403	Forbidden
404	Not Found

Try it out

- 결과가 출력된다.

The screenshot shows a Swagger UI interface for a REST API. The top bar indicates a **GET** request to the endpoint **/api/todos** with the operation ID **getTodos**. Below the header, there are sections for **Parameters** (none), **Responses**, and a **Curl** command. The **Responses** section shows a successful response (200 OK) with an empty JSON body and detailed headers. It also lists other responses for 401, 403, and 404 errors. The bottom right corner of the interface has a red **Cancel** button.

## Swagger 핵심 Annotation

- RestController에 Annotation을 설정하여 Swagger-UI에서 원하는 값들이 출력하도록 할 수 있다.

```
@Api(tags = "todos")
@RestController
@RequestMapping(path = "/api/todos")
public class TodoController {
```

- @Api 안붙였을 때

todo-controller Todo Controller >

---

Models >

- @Api 붙였을 때

todos Todo Controller >

---

Models >

## @ApiOperation = Method 설명

- @ApiOperation으로 해당 Controller 안의 method의 설명을 추가할 수 있다.

```
@ApiOperation(  
    value = "Todo 목록을 읽어온다."  
    , notes = "모든 Todo 목록을 읽어온다.")  
@GetMapping  
public List<Todo> getTodos(){  
    return todoService.getTodos();  
}
```

- API에 설명이 표시된다.

**todos** Todo Controller ▾

<b>GET</b>	/api/todos	Todo 목록을 읽어온다.	🔒
<b>POST</b>	/api/todos	addTodo	🔒
<b>PUT</b>	/api/todos	updateTodo	🔒
<b>DELETE</b>	/api/todos	deleteTodo	🔒

## @ApilmplicitParam = Request Parameter 설명

- @ApilmplicitParam Annotation으로 해당 API Method 호출에 필요한 Parameter들의 설명을 추가할 수 있다.
- dataType, paramType, required의 경우 해당 name의 정보가 자동으로 채워지므로 생략 할 수 있다. paramType의 경우 @RequestParam은 query를, @PathVariable은 path를 명시해주면 된다. 만약 해당 Method의 Parameter가 복수일 경우, @ApilmplicitParams로 @ApilmplicitParam을 복수개 사용할 수 있다.

```
@ApiOperation(  
    value = "id에 해당하는 Todo를 읽어온다."  
    , notes = "id에 해당하는 Todo를 읽어온다.")  
@ApiImplicitParam(  
    name = "id"  
    , value = "Todo 아이디"  
    , required = true  
    , dataType = "Long"  
    , paramType = "path"  
    , defaultValue = "None")  
@GetMapping("/{id}")  
public Todo getTodo(@PathVariable(name = "id")Long id){  
    return todoService.getTodo(id);  
}
```

**GET** /api/todos Todo 목록을 읽어온다.  

모든 Todo 목록을 읽어온다.

**Parameters** 

No parameters

## **@ApiResponse = Response 설명**

- @ApiResponse Annotation으로 해당 method의 Response에 대한 설명을 작성할 수 있다.
- 복수개의 Response에 대한 설명을 추가하고 싶다면, @ApiResponses를 사용하면 된다.

```
@ApiOperation(  
    value = "Todo 목록을 읽어온다."  
    , notes = "모든 Todo 목록을 읽어온다.")  
@ApiResponse(  
    code = 200  
    , message = "성공입니다."  
)  
@GetMapping  
public List<Todo> getTodos(){  
    return todoService.getTodos();  
}
```

- 복수개를 설정하고 싶을 경우

```
@ApiResponses({  
    @ApiResponse(code = 200, message = "성공입니다.")  
    , @ApiResponse(code = 400, message = "접근이 올바르지 않습니다.")  
})
```

- Default Response Message들을 삭제하고 싶다면, Swagger Config에 Docket에 useDefaultResponseMessages(false)를 설정해주면 된다. @ApiResponse로 설명하지 않은 401, 403, 404 응답들이 사라진다.

```
public Docket getDocket(String groupName, Predicate<String> predicate) {  
    return new Docket(DocumentationType.SWAGGER_2)  
        .useDefaultResponseMessages(false) // 지정한 응답 코드만 보여지도록 한다.  
        .securityContexts(Arrays.asList(securityContext())) // bearer인증 설정  
        .securitySchemes(Arrays.asList(apiKey())) // bearer 인증 설정  
            .groupName(groupName)  
        .select()  
        .apis(RequestHandlerSelectors.basePackage("examples.springmvc"))  
        .paths(predicate)  
        .apis(RequestHandlerSelectors.any())  
        .build();  
}
```

## **@ApiParam = DTO field 설명**

- @ApiParam Annotation으로 DTO의 field에 대한 설명을 추가할 수 있다.
- @ModelAttribute와 함께 사용될 때 설명에 표시가 된다.

**POST** /api/todos/test addTodo2 

**Parameters** 

Name	Description
done boolean (query)	
<b>id * required</b> integer(\$int64) (query)	ToDo ID
todo string (query)	

## **@ApiModelProperty = DTO 예제 설명**

- @ApiModelProperty Annotation을 사용하는 DTO Class Field에 추가하면, 해당 DTO Field의 예제 Data를 추가할 수 있다.

```
package examples.springmvc.domain;

import java.util.Objects;

import io.swagger.annotations.ApiModelProperty;
import io.swagger.annotations.ApiParam;

public class Todo {
    private Long id;
    private String todo;
    private boolean done;

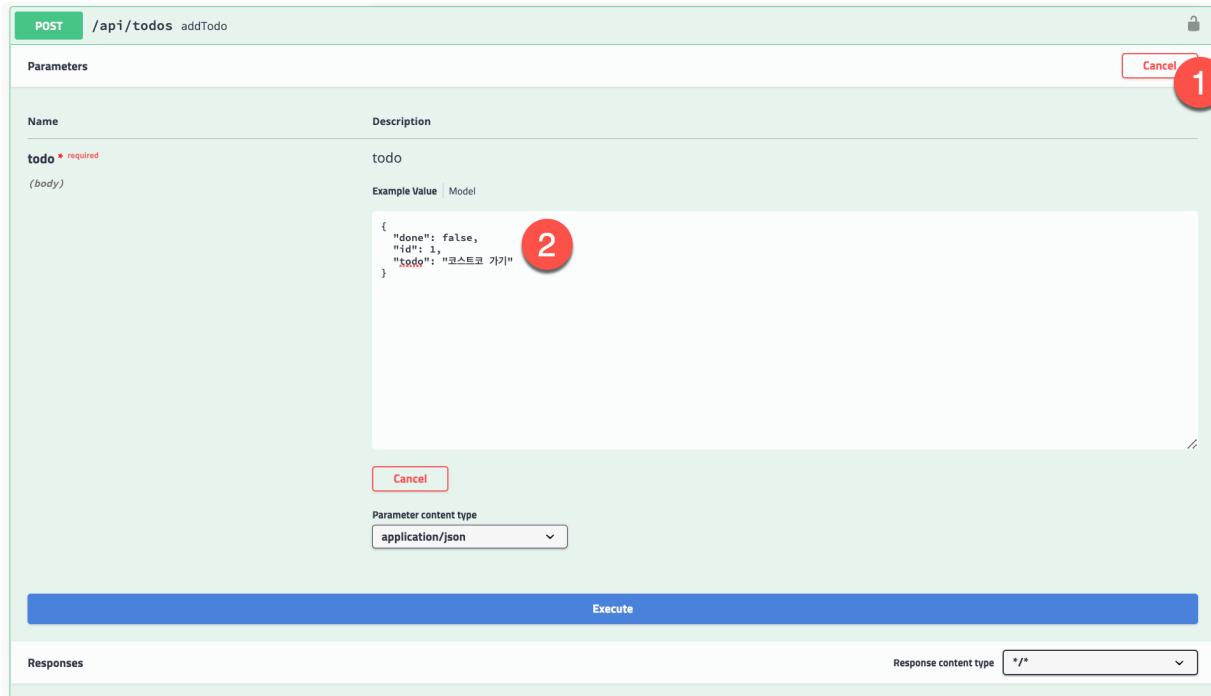
    public Todo(){
        this.done = false;
    }

    @ApiModelProperty(
        name = "id"
        , example = "1"
    )
    @ApiParam(value = "ToDo ID", required = true)
    public Long getId() {
        return id;
    }
}
```

```
public void setId(Long id) {  
    this.id = id;  
}  
  
public String getTodo() {  
    return todo;  
}  
  
@ApiModelProperty(  
    name = "todo"  
    , example = "코스트코 가기"  
)  
public void setTodo(String todo) {  
    this.todo = todo;  
}  
  
public boolean isDone() {  
    return done;  
}
```

```
@ApiModelProperty(  
    name = "done"  
    , example = "false"  
)  
public void setDone(boolean done) {  
    this.done = done;  
}  
  
@Override  
public boolean equals(Object o) {  
    if (this == o) return true;  
    if (o == null || getClass() != o.getClass()) return false;  
    Todo todo = (Todo) o;  
    return Objects.equals(id, todo.id);  
}  
  
@Override  
public int hashCode() {  
    return Objects.hash(id);  
}  
}
```

- Todo를 @RequestBody로 입력받는데 어떤 경우는 example에 하나의 값만 보여지고 어떤 경우엔 3개보여지고 하는 방법은 없다. (있으면 알려주세요.)



## @Apilgnore = Swagger UI 상 무시

- 위 같은 코드에서 해당 method getNotice에서는 UserDTO의 id 만 필요한 상황인데, Parameter에 DTO를 삽입하여 모든 Parameter 정보가 노출되었다. @Apilgnore을 추가함으로써, @ApilmplicitParam으로 선언하지 않은 parameter 정보들을 무시할 수 있다.
- 또한 method의 return type 앞에 명시해 해당 method를 아예 Swagger UI 에서 노출되지 않게 바꿀 수도 있다.

```
@PostMapping("/test")
public Todo addTodo2(@ModelAttribute Todo todo) {

    return todo;
}
```

**POST** /api/todos/test addTodo2

**Parameters**

**Name** **Description**

**id** \* required  
string  
(path)

사용자 아이디  
None

**Execute**

**Responses**

**Code** **Description**

200 OK

**Example Value** | Model

```
{  
    "done": false,  
    "id": 1,  
    "todo": "코스트코 가기"  
}
```

- `@ApiIgnore` 를 이용해 `ModelAttribute`로 입력받는 파라미터들을 무시하고, `id`만 입력받도록 설정한다.

```
@ApiImplicitParams (
    @ApiImplicitParam(
        name = "id"
        , value = "사용자 아이디"
        , required = true
        , dataType = "string"
        , paramType = "path"
        , defaultValue = "None")
)
@PostMapping("/test")
public Todo addTodo2(@ApiIgnore @ModelAttribute Todo todo) {

    return todo;
}
```

**POST** /api/todos/test addTodo2

 [Cancel](#)

**Parameters**

Name	Description
<b>id</b> * required string (path)	사용자 아이디 None

**Execute**

**Responses** Response content type `*/*`

Code	Description
200	<b>OK</b>  <b>Example Value</b>   Model <pre>{     "done": false,     "id": 1,     "todo": "코스트코 가기" }</pre>

## Swagger UI API 화면 커스텀

- Swagger Config Class에서 .apilnfo(ApiInfo Type)을 작성하면, Swagger UI 기본 화면의 API 설명 문구등을 커스텀할 수 있다.

```
private static final String API_NAME = "ToDo API";
private static final String API_VERSION = "0.0.1";
private static final String API_DESCRIPTION = "ToDo API 명세서";

.....
//swagger 설정.
public Docket getDocket(String groupName, Predicate<String> predicate) {
    return new Docket(DocumentationType.SWAGGER_2)
        .useDefaultResponseMessages(false) // 지정한 응답 코드만 보여지도록 한다.
        .securityContexts(Arrays.asList(securityContext())) // bearer인증 설정
        .securitySchemes(Arrays.asList(apiKey())) // bearer 인증 설정
        .groupName(groupName)
        .apiInfo(apiInfo()) // <--- apiInfo를 추가한다.
.....
}

public ApiInfo apiInfo() {
    return new ApiInfoBuilder()
        .title(API_NAME)
        .version(API_VERSION)
        .description(API_DESCRIPTION)
        .build();
}
```

# ToDo API 0.0.1

[ Base URL: localhost:8080/webtodoserver ]

<http://localhost:8080/webtodoserver/v2/api-docs?group=todos>

ToDo API 명세서

## Bearer 인증 관련 설정

- Bearer 인증이란?
  - Bearer 토큰은 토큰을 소유한 사람에게 액세스 권한을 부여하는 일반적인 토큰 클래스입니다. 액세스 토큰, ID 토큰, 자체 서명 JWT는 모두 Bearer 토큰입니다. 인증에 Bearer 토큰을 사용하려면 HTTPS 와 같은 암호화된 프로토콜로 제공되는 보안이 필요합니다.

## Bearer 인증 관련 설정

```
//swagger 설정.  
public Docket getDocket(String groupName, Predicate<String> predicate) {  
    return new Docket(DocumentationType.SWAGGER_2)  
        .securityContexts(Arrays.asList(securityContext())) // bearer인증 설정  
        .securitySchemes(Arrays.asList(apiKey())) // bearer 인증 설정  
            .groupName(groupName)  
        .select()  
        .apis(RequestHandlerSelectors.basePackage("examples.springmvc"))  
        .paths(predicate)  
        .apis(RequestHandlerSelectors.any())  
        .build();  
}
```

```
private SecurityContext securityContext() {
    return SecurityContext.builder()
        .securityReferences(defaultAuth())
        .build();
}

private List<SecurityReference> defaultAuth() {
    AuthorizationScope authorizationScope = new AuthorizationScope("global", "accessEverything");
    AuthorizationScope[] authorizationScopes = new AuthorizationScope[1];
    authorizationScopes[0] = authorizationScope;
    return Arrays.asList(new SecurityReference("Authorization", authorizationScopes));
}

private ApiKey apiKey() {
    return new ApiKey("Authorization", "Authorization", "header");
}
```

# Api Documentation<sup>1.0</sup>

[ Base URL: localhost:8080/webtodoserver ]

<http://localhost:8080/webtodoserver/v2/api-docs?group=todos>

Api Documentation

[Terms of service](#)

[Apache 2.0](#)

Authorize 

## todos Todo Controller

GET

/api/todos getTodos



POST

/api/todos addTodo



PUT

/api/todos updateTodo



DELETE

/api/todos deleteTodo



## Available authorizations

X

### Authorization (apiKey)

Name: Authorization

In: header

Value:

Bearer 키값

 Authorize

Close

## Available authorizations

X

### Authorization (apiKey)

**Authorized**

Name: Authorization

In: header

**Value:** \*\*\*\*\*

Logout

Close



- Berarer 인증이란 Http 헤더이름은 Authorization, 헤더 값은 "Bearer 키값"으로 오는 형태이다. 이 값을 필터 등으로 공통으로 처리하도록 보통 프로그래밍 한다.

```
@GetMapping("/bearerTest")
public String bearerTest(@ApiIgnore @RequestHeader("Authorization") String authorization) {
    return authorization;
}
```

# Spring Data JPA

# JPA

- Java Persistence API
- 자바 진영의 ORM 기술 표준

# JPA소개

하이버네이트(오픈 소스)

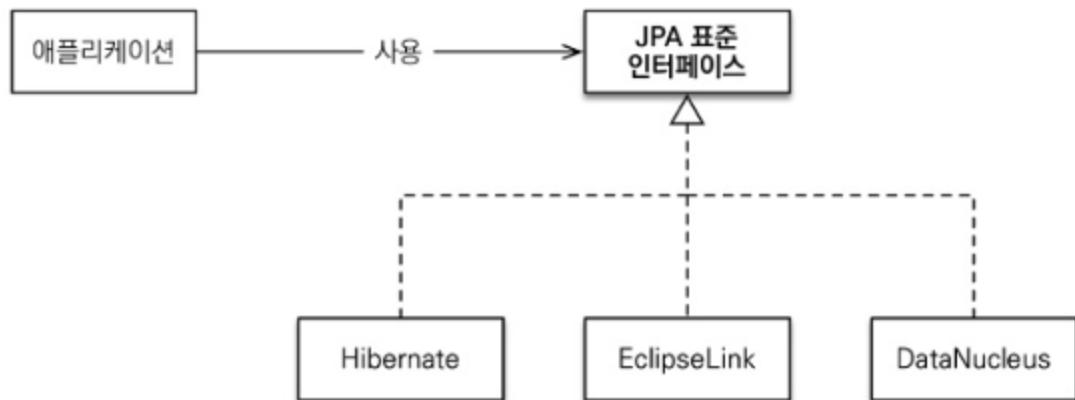


EJB - 엔티티 빙(자바 표준)

JPA(자바 표준)

# JPA는 표준 명세

- JPA는 인터페이스의 모음
- JPA 2.1 표준 명세를 구현한 3가지 구현체
- 하이버네이트, EclipseLink, DataNucleus



## 데이터 베이스 프로그래밍 시 문제점

- 데이터베이스에 표현된 테이블 간의 관계를 자바 객체 간의 참조 관계로 표현하는 것이 어렵다.
- SQL을 사용할 때 입출력한 값을 담을 객체(Data Transfer Object라는 클래스)를 너무 많이 만든다.
- 테이블 이름만 다른 비슷한 SQL을 너무 만든다.
- SQL에서 읽고 쓰는 테이블 칼럼과 자바 객체의 프로퍼티를 매팅하는 것은 기계적이고 소모적인 작업이다.

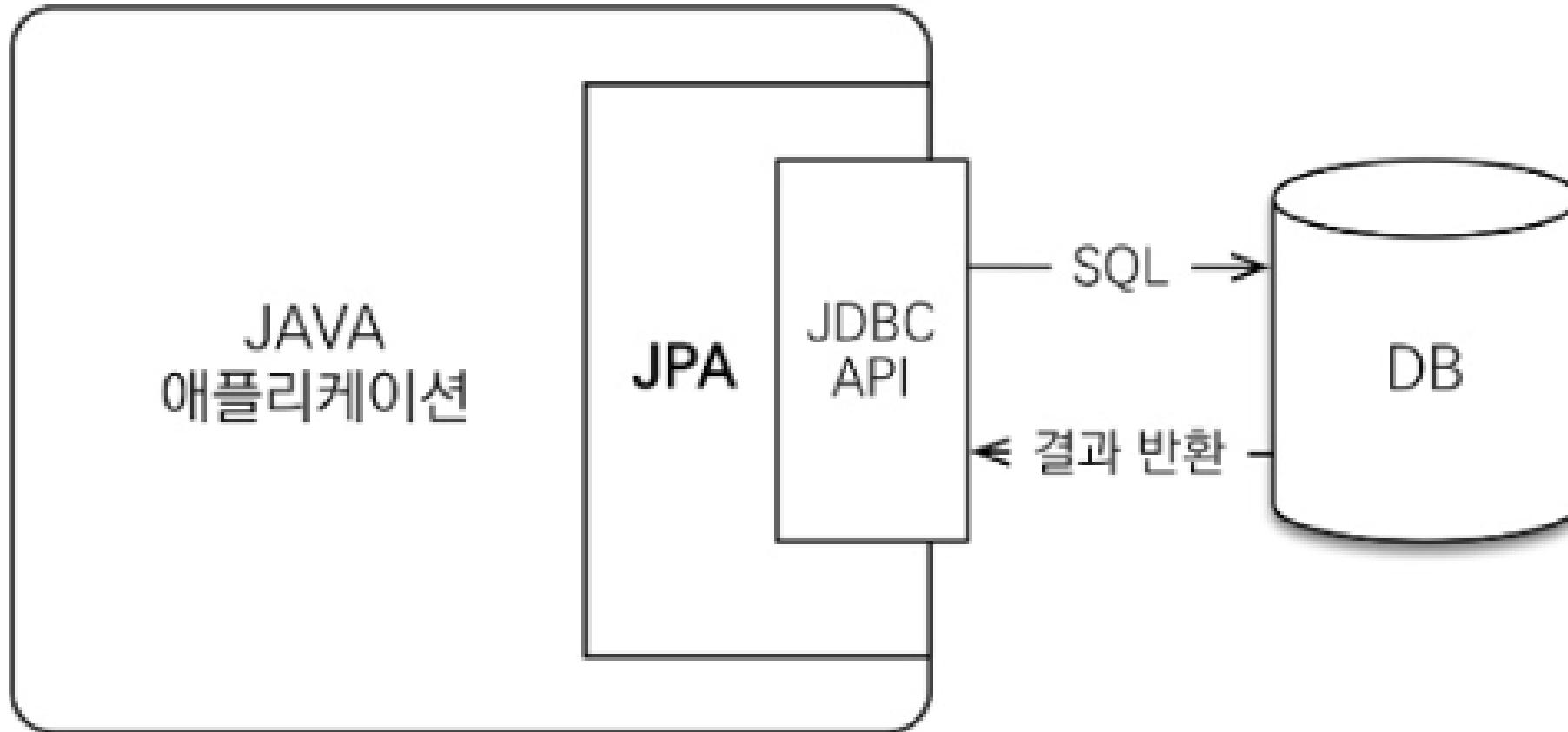
## JPA를 사용하는 이유

- SQL 중심적인 개발에서 객체 중심으로 개발
- 생산성
- 유지보수
- 패러다임의 불일치 해결
- 성능
- 데이터 접근 추상화와 벤더 독립성
- 표준

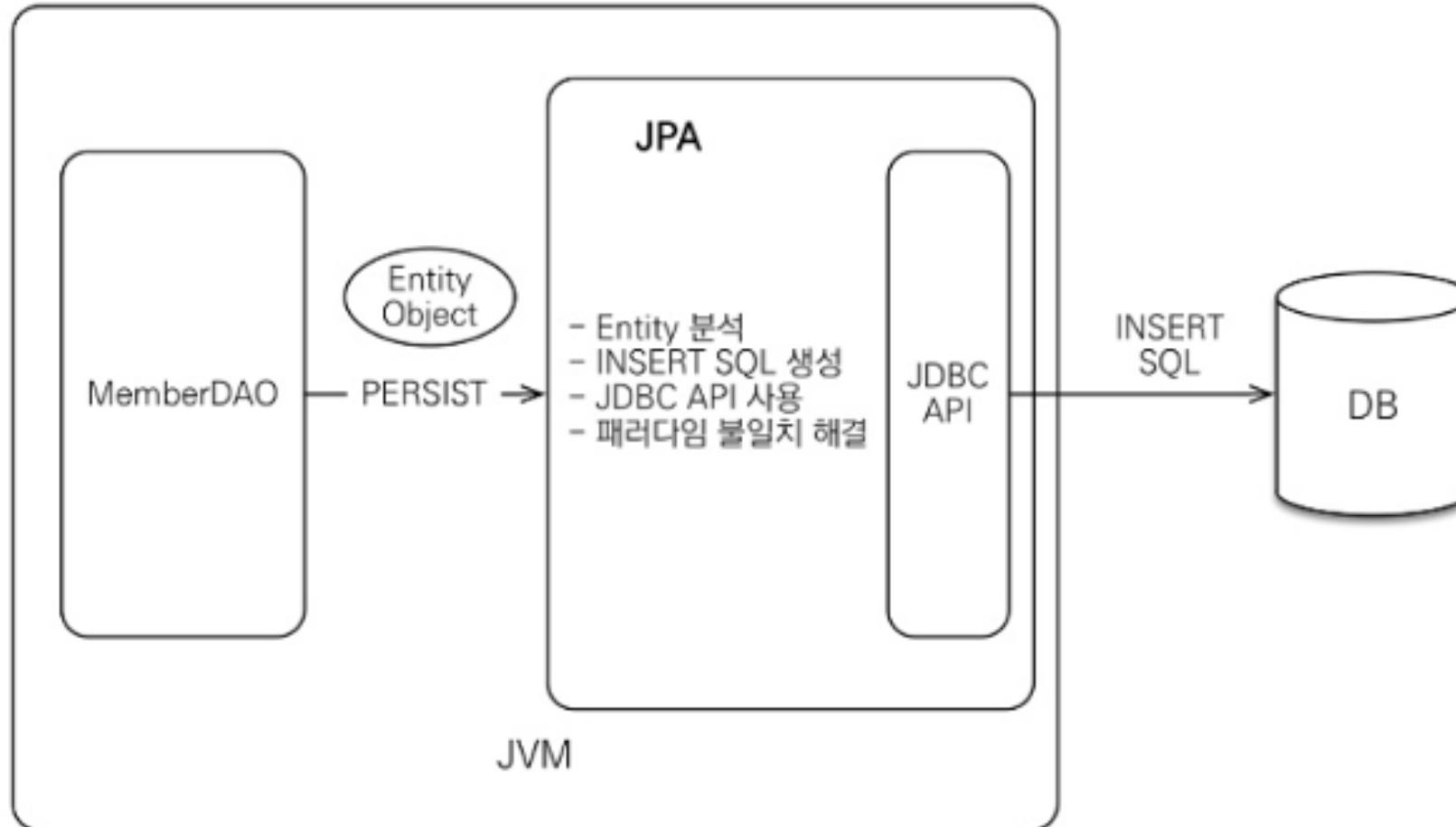
# ORM?

- Object-relational mapping
- 객체에 데이터를 읽고 쓰는 방식으로 구현하는 기술
- 객체 관계 매핑
- 객체는 객체대로 설계
- 관계형 데이터베이스는 관계형 데이터베이스대로 설계
- ORM 프레임워크가 중간에서 매핑
- 대중적인 언어에는 대부분 ORM 기술이 존재

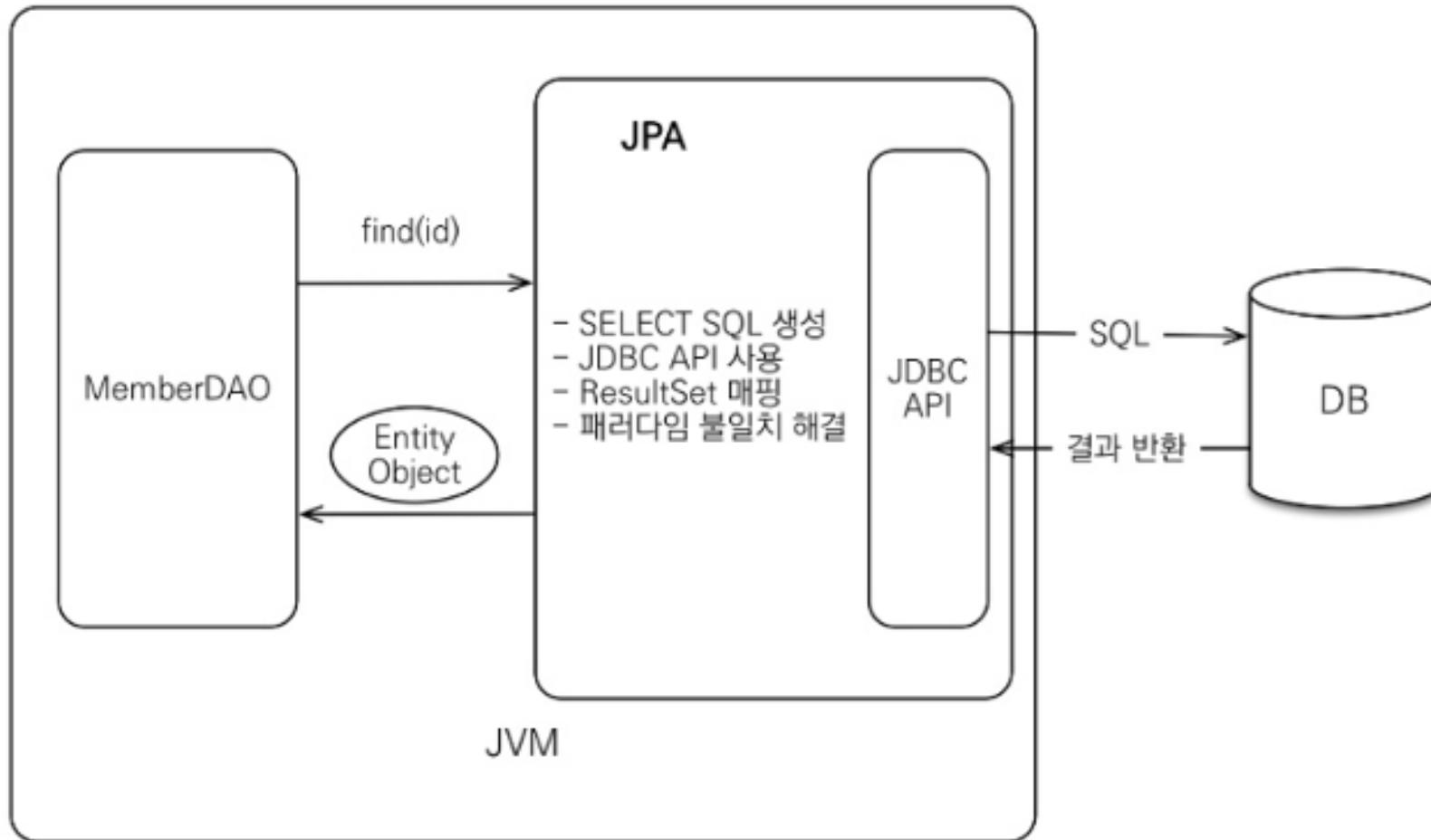
## JPA는 애플리케이션과 JDBC사이에서 동작



# JPA 동작 - 저장



## JPA동작 - 조회



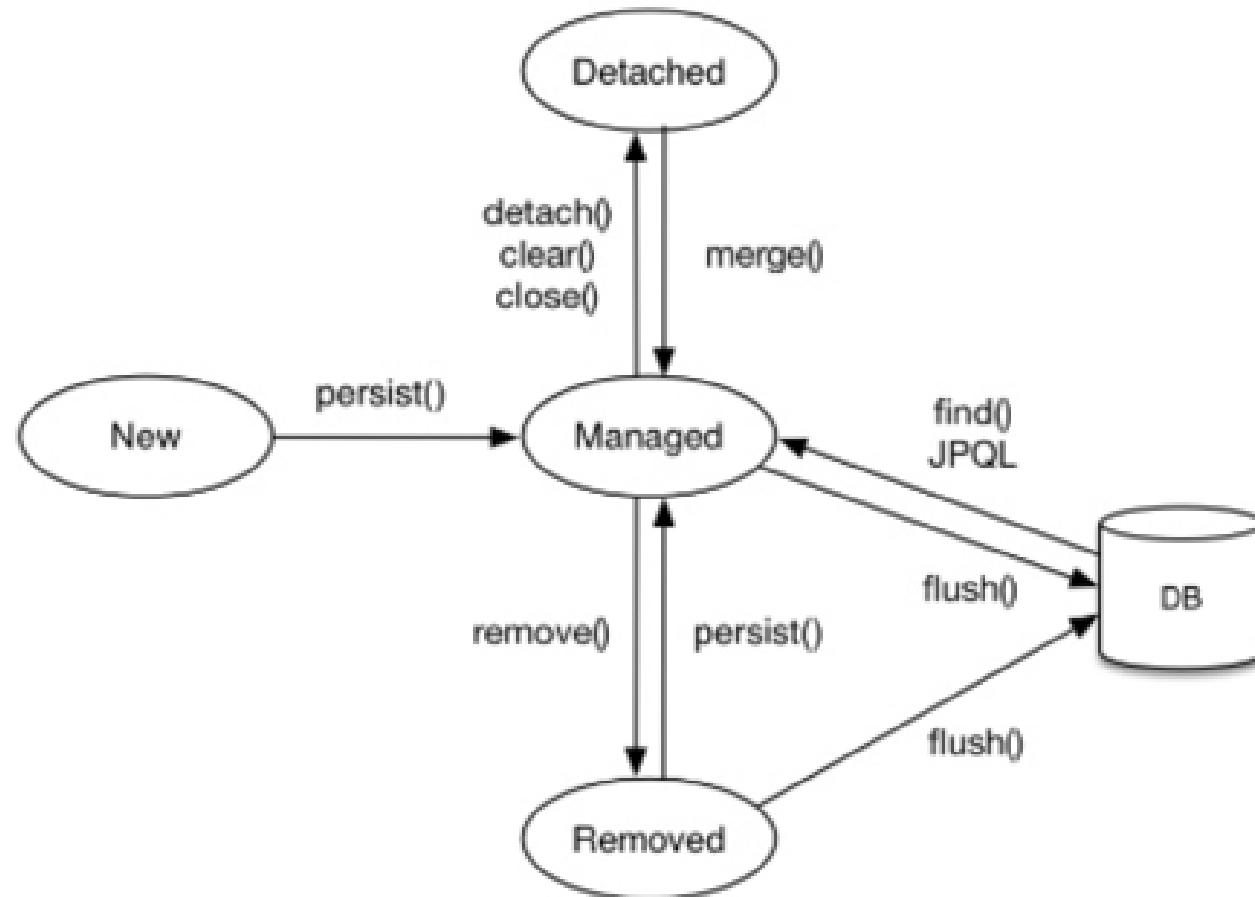
## 영속성 컨텍스트(Persistence context)

- 엔티티를 영구 저장하는 환경
- 논리적인 개념에 가깝다. 영속성 컨텍스트는 엔티티 메니저를 생성할 때 하나 만들어진다. 그리고 엔티티 메니저는 영속성 컨텍스트에 접근할 수 있고, 영속성 컨텍스트를 관리할 수 있다.

## 엔티티 생명주기

- 비영속(new/transient) : 영속성 컨텍스트와 전혀 관계가 없는 상태
- 영속(managed) : 영속성 컨텍스트에 저장된 상태
- 준영속(detached) : 영속성 컨텍스트에 저장되었다가 분리된 상태
- 삭제(removed) : 삭제된 상태

# 엔티티의 생명주기



## 영속성 컨텍스트 특징

- 영속성 컨텍스트와 식별자 값. : 영속성 컨텍스트는 엔티티를 식별자 값으로 구분한다. 영속상태는 식별자 값이 반드시 있어야 한다.
- 영속성 컨텍스트와 데이터 베이스 저장 : JPA는 보통 트랜잭션이 커밋하는 순간 영속성 컨텍스트에 새로 저장된 엔티티를 데이터베이스에 반영한다. 이를 flush라고 한다.
- 영속성 컨텍스트가 엔티티를 관리할 때의 장점 : 1차 캐시, 동일성 보장, 트랜잭션을 지원하는 쓰기 지연, 변경감지, 지연 로딩

## 엔티티 조회

- 영속성 컨텍스트는 내부에 캐시를 가지고 있는데, 이를 1차 캐시라 한다. 영속상태의 엔티티는 모두 이곳에 저장된다.
- 1차 캐시의 키는 식별자 값이다. 이미 1차캐시에 있을 경우 db에서 조회하지 않고 캐시에서 조회하게 된다.

## 영속 엔티티의 동일성 보장

- 같은 id의 값을 가진 엔티티를 조회하면 `==` 를 이용했을 경우 참이 나온다. 이를 동일성이라한다.  
이는 실제 인스턴스가 같다는 것을 의미한다.

# Entity

- 데이터베이스에서 영속적으로 저장된 데이터를 자바 객체로 매팅한 것을 Entity라고 한다.
- Entity는 메모리 상에 자바 객체의 인스턴스 형태로 존재하며, 이후에 설명할 EntityManager에 의해 데이터베이스의 데이터와 동기화된다.
- Entity는 POJO를 이용한 클래스로 기술할 수 있지만 Entity임을 JPA에 구현에 인식시키거나 매팅에 필요한 정보를 추가하기 위해 JPA가 제공하는 애노테이션을 사용해야 한다.
- JPA에서는 객체를 고유하게 식별하기 위한 기본키를 Entity 프로퍼티와 테이블에 갖게 할 필요가 있다. 이 기본키에 의해 Entity와 데이터베이스에 영속화된 데이터가 연결된다.

# Entity 애노테이션 정리

- **@Entity**

이 어노테이션은 해당 클래스가 엔티티임을 알리기 위해 사용합니다. 애플리케이션이 실행이 될 때 엔티티 자동검색을 통하여 이 어노테이션이 선언 된 클래스들은 엔티티 빈으로 등록합니다.

- **@Table**

데이터의 저장소, 테이블을 의미합니다. name 값은 실제 데이터베이스의 테이블명을 의미하며, 이 어노테이션은 생략이 가능합니다. 어노테이션을 생략하면 클래스의 이름을 테이블의 이름으로 자동 인식하게 됩니다.

# Entity 애노테이션 정리

- **@Id**  
엔티티빈의 기본키를 의미합니다. 이 어노테이션은 하나의 엔티티에는 반드시 하나가 존재해야 합니다. 복수키도 설정할 수 있습니다
- **@GeneratedValue**  
데이터베이스에 의해 자동으로 생성된 값이라는 의미입니다. 즉, 프로그램 상에서 조작된 데이터가 아닌, 실제 데이터베이스에 데이터가 영속(저장)될 때 생성되는 값입니다. 몇가지 생성전략이 존재합니다.

# Entity 애노테이션 정리

- **@Column**

필드와 테이블의 컬럼을 매핑시켜줍니다. 이 어노테이션은 생략이 가능하며, 생략시 필드의 이름이 테이블의 컬럼으로 자동으로 매핑이됩니다.

- name속성(String)

- 필드와 매핑 될 컬럼의 이름을 명시합니다.

- nullable속성(boolean)

- 해당 컬럼이 null값을 허용하는가 하지않는가의 여부입니다.

- length속성(int)

- 컬럼의 길이값을 의미합니다.

- @Column
  - unique속성(boolean)
    - 컬럼이 유일한 값을 가져야 하는가 아닌가의 여부입니다.
  - insertable속성(boolean)
    - 엔티티가 영속될 때 insert에 참여할지 말지를 결정합니다. 기본값은 true
  - updatable속성(boolean)
    - 변경된 필드의 값을 테이블에도 반영할지를 결정합니다. 기본값은 true
    - name속성을 제외한 나머지 속성은 잘 사용되지 않을것이라고 생각됩니다. nullable, length, unique는 DDL과 관련된 속성이고, insertable, updatable은 원래 잘 사용되지 않는 속성이기 때문입니다.

## 복합키 설정 방법 - @IdClass 이용

```
create table order_product (
    order_id integer not null,
    product_id integer not null,
    amount integer not null,
    primary key (order_id, product_id)
)
```

```
@Data
class OrderProductPK implements Serializable {
    private int orderId;
    private int productId;
}
```

```
@Data  
@Entity  
@IdClass(OrderProductPK.class)  
class OrderProduct {  
  
    @Id  
    @Column(name = "ORDER_ID")  
    private int orderId;  
  
    @Id  
    @Column(name = "PRODUCT_ID")  
    private int productId;  
  
    private int amount;  
}
```

# Entity 예

```
@Entity ---- (1) Entity애노테이션을 부여하여 Entity클래스임을 나타낸다.  
@Table(name = "file") ---- (2) Table애노테이션을 이용하여 매핑할 테이블을 지정한다.  
                           생략할 경우 클래스명을 대문자로 한 이름의 테이블과 매핑된다.  
public class File implements Serializable { ---(3) 반드시 Serializable할 필요는 없지만,  
                           확장성을 고려해서 Serializable하도록 작성한다.  
@Id --- (4) Id애노테이션을 부여하고 기본키임을 나타낸다.  
                           기본키가 복합키로 된 경우에는 @javax.persistence.EmbeddedId를 사용해 대응할 수 있다.  
@GeneratedValue ---- (5) GeneratedValue 애노테이션을 부여해 기본키 생성을 JPA에 맡길 수 있다.  
                           strategy속성에 GenerationType을 지정해 생성방법을 지정할 수 있지만 기본적으로는 GenerationType.AUTO, 즉  
                           사용하는 데이터베이스의 최적의 키 생성 방법이 자동으로 선택된다.  
@Column(name = "file_id") --- (6) Column애노테이션을 부여하고 매핑되는 칼럼명을 지정한다. 생략한 경우 프로퍼티명을 대문자로 한 이름의 컬럼에 매핑한다.  
private Integer fileId;  
  
@Column(name = "name")  
private String name;  
.....  
}
```

# EntityManager

- Entity를 필요에 따라 데이터베이스와 동기화하는 역할을 담당하는 것이 EntityManager다.
- EntityManager에는 영속성 컨텍스트(Persistence Context)라는 Entity를 관리하기 위한 영역이 있다.
- 애플리케이션이 데이터베이스의 데이터에 접근하는 경우에는 반드시 EntityManager를 통해 영속성 컨텍스트의 Entity를 취득하거나 새로 생성한 Entity를 영속성 컨텍스트에 등록해야 한다. 이를 통해 EntityManager가 Entity변경을 추적할 수 있어 적절한 타이밍에 데이터베이스와 동기화한다.
- EntityManager는 Entity의 상태를 변경하거나 데이터베이스의 동기화를 위한 API를 제공한다.

# EntityManager가 제공하는 대표적인 API 1/3

- <T> T find(java.lang.Class<T> entityClass, java.lang.Object primaryKey)
  - 기본키를 지정해서 Entity를 검색하고 반환한다. 영속성 컨텍스트에 해당하는 Entity가 존재하지 않는 경우 데이터베이스에서 SQL(Select문)을 발생해 해당 데이터를 취득하고 Entity를 생성해서 반환한다.
- void persist(java.lang.Object entity)
  - 애플리케이션에서 생성한 인스턴스를 Entity로 영속성 컨텍스트에서 관리한다. SQL의 Insert문에 해당하지만 persist메소드를 실행한 시점에는 데이터베이스의 SQL은 실행되지 않고 영속성 컨텍스트에 축적된다.
- <T> T merge(T entity)
  - 영속성 컨텍스트에서 관리되고 있지만 분리 상태가 된 Entity를 영속성 컨텍스트에서 다시 관리한다. 관리 상태의 경우에는 차이점을 추적할 수 없기 때문에 데이터베이스에 변경을 반영하기 위해 SQL(Insert문)이 영속성 컨텍스트에 축적된다.

# EntityManager가 제공하는 대표적인 API 2/3

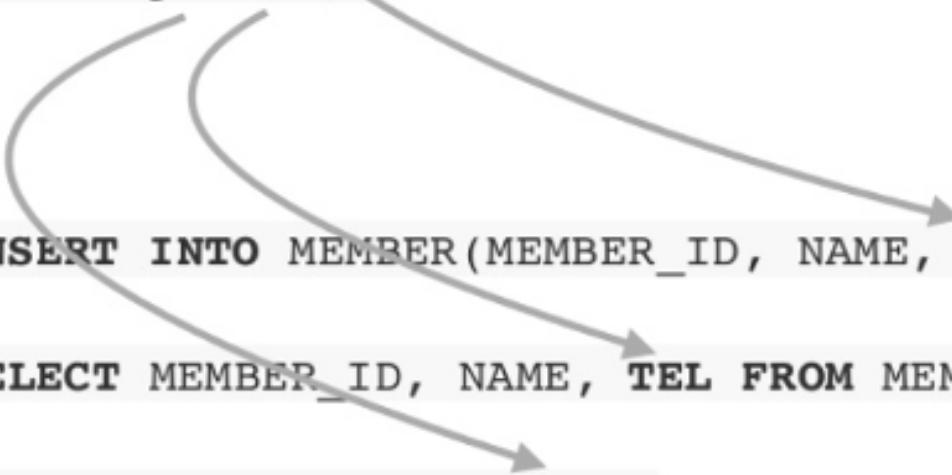
- void remove(java.lang.Object entity)
  - Entity를 영속성 컨텍스트 및 데이터베이스에서 삭제한다. persist메소드나 merge메소드와 마찬가지로 데이터베이스에 SQL(delete문)이 즉시 실행되지 않고 영속성 컨텍스트에 축적된다.
- void flush()
  - 영속성 컨텍스트에 축적된 모든 Entity의 변경 정보를 데이터베이스에 강제적으로 동기화한다. 일반적으로 데이터베이스에 반영하는 작업은 트랜잭션을 커밋할 때 하지만 커밋 이전에 반영할 필요가 있는 경우에 사용한다.
- void refresh(java.lang.Object entity)
  - Entity의 상태를 데이터베이스의 데이터로 강제로 변경한다. 데이터베이스에 반영되지 않는 Entity에 대해 변경된 사항은 덮어쓰게 된다.

# EntityManager가 제공하는 대표적인 API 3/3

- <T> TypeQuery<T> createQuery(java.lang.String qlString, java.lang.Class<T> resultClass)
  - 기본키 이외의 것으로 데이터베이스에 접근하는 경우에는 JPA용 쿼리를 실행해 Entity를 취득하거나 변경할 수 있다. 이 API는 쿼리를 작성하기 위한 API중 하나로서 비슷한 API가 여러 개 제공된다.
- void detach(java.lang.Object entity)
  - Entity를 영속성 컨텍스트에서 삭제하고 분리 상태로 만든다. 이 Entity에 대해 변경된 모든 사항은 merge 메소드를 실행하지 않는 한 데이터베이스에 반영되지 않는다.
- void clear()
  - 영속성 컨텍스트에서 관리되는 모든 Entity를 분리 상태로 만든다.
- Boolean contains(java.lang.Object entity)
  - Entity가 영속성 컨텍스트에서 관리되는지를 반환한다.

## 유지보수 - 필드 변경시 모든 SQL 수정

```
public class Member {  
  
    private String memberId;  
    private String name;  
    private String tel;  
    ...  
}
```

INSERT INTO MEMBER(MEMBER\_ID, NAME, TEL) VALUES ...

SELECT MEMBER\_ID, NAME, TEL FROM MEMBER M

UPDATE MEMBER SET ... TEL=?

## 유지보수 - 필드만 추가하면 됨, SQL은 JPA가 알아서 처리

```
public class Member {  
  
    private String memberId;  
    private String name;  
    private String tel;  
    ...  
}  
  
INSERT INTO MEMBER(MEMBER_ID, NAME, TEL) VALUES ...  
  
SELECT MEMBER_ID, NAME, TEL FROM MEMBER M  
  
UPDATE MEMBER SET ... TEL=?
```

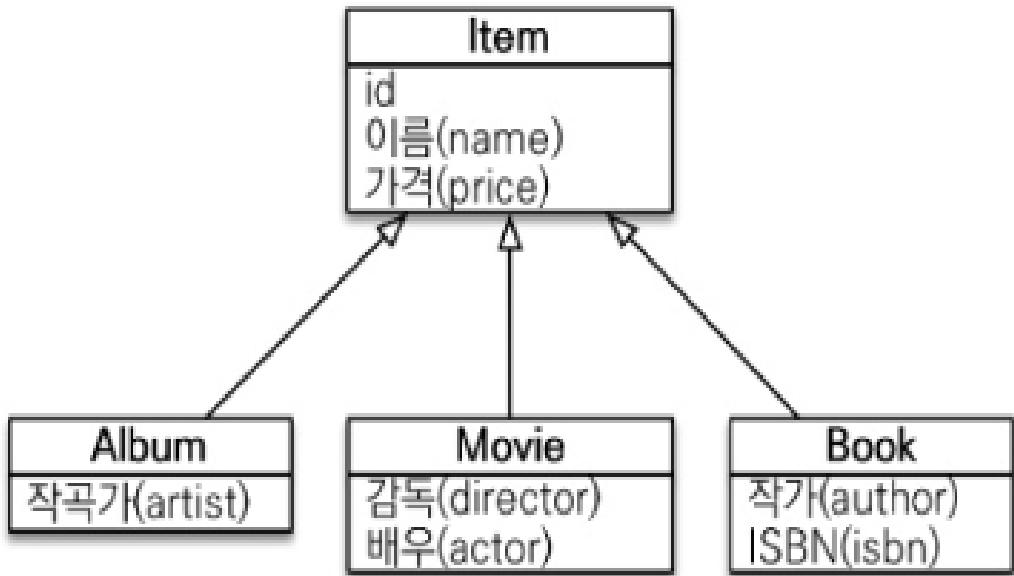
## 끝없이 결과가 나오는 json메시지

- @JsonManagedReference
- @JsonBackReference
- @JsonIgnore

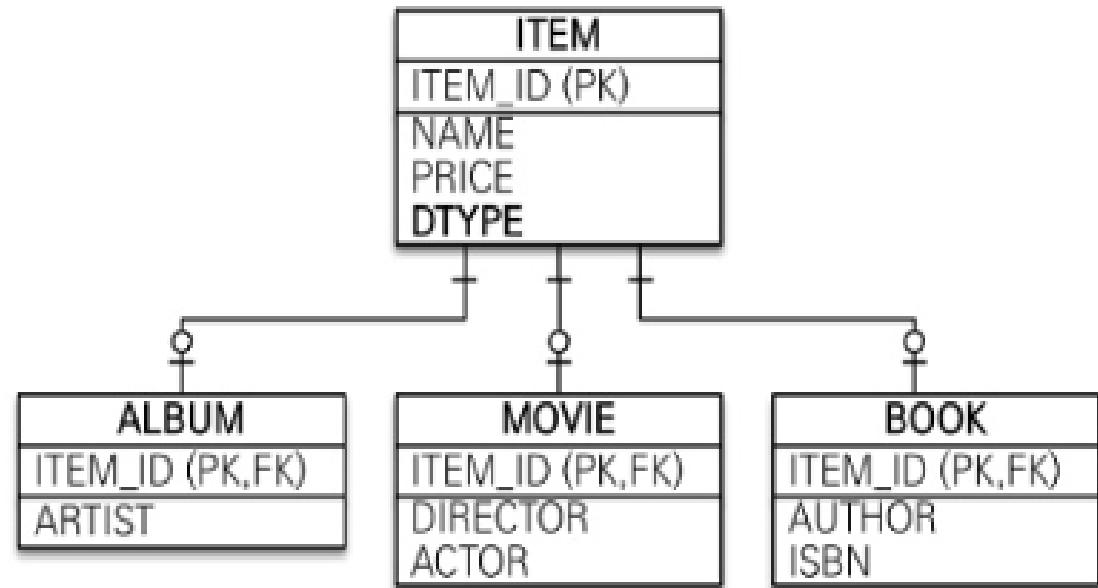
## JPA와 패러다임의 불일치 해결

- JPA와 상속
- JPA와 연관관계
- JPA와 객체 그래프 탐색
- JPA와 비교하기

# JPA와 상속



[객체 상속 관계]



[Table 슈퍼타입 서브타입 관계]

## JPA와 상속

```
jpa.persist(album);
```

개발자가 할일

```
INSERT INTO ITEM ...  
INSERT INTO ALBUM ...
```

나머진 JPA가 처리

## JPA와 상속

```
Album album = jpa.find(Album.class, albumId);
```

개발자가 할일

```
SELECT I.* , A.*  
FROM ITEM I  
JOIN ALBUM A ON I.ITEM_ID = A.ITEM_ID
```

나머진 JPA가 처리

## JPA와 연관관계, 객체 그래프 탐색

```
member.setTeam(team);  
jpa.persist(member);
```

연관관계 저장

```
Member member = jpa.find(Member.class, memberId);  
Team team = member.getTeam();
```

객체 그래프 탐색

## 신뢰할 수 있는 엔티티 계층

```
class MemberService {  
    ...  
    public void process() {  
        Member member = memberDAO.find(memberId);  
        member.getTeam(); //자유로운 객체 그래프 탐색  
        member.getOrder().getDelivery();  
    }  
}
```

## JPA와 비교하기

```
String memberId = "100";
Member member1 = jpa.find(Member.class, memberId);
Member member2 = jpa.find(Member.class, memberId);

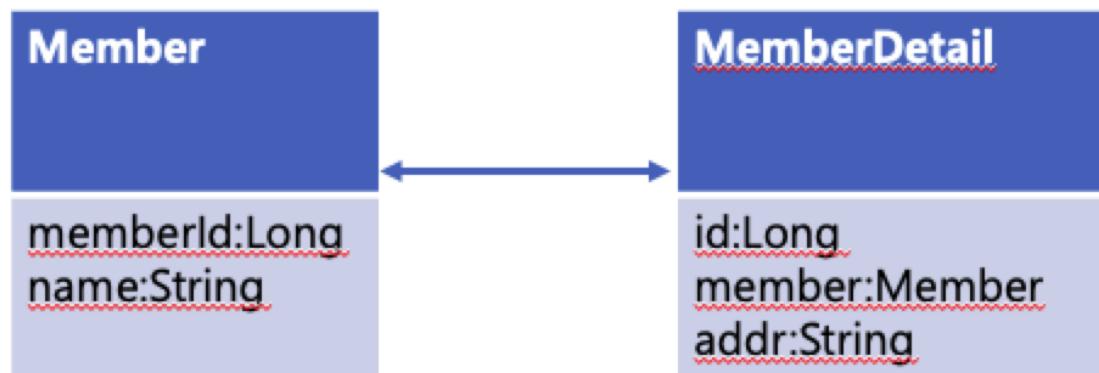
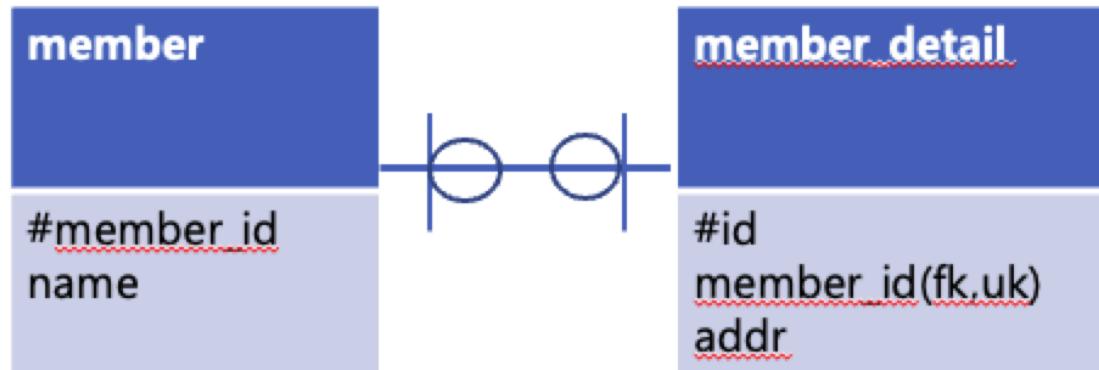
member1 == member2; //같다.
```

동일한 트랜잭션에서 조회한 엔티티는 같음을 보장

## 연관관계

- JPA에서는 데이터베이스의 연관관계를 Entity간의 참조 관계로 매팅한다. 두 객체 간의 카디널리티(일대일, 일대다, 다대다)와 방향에 따라 다음과 같은 유형으로 분류할 수 있다. 양방향은 되도록 지양한다.
  - 단방향 일대일
  - 양방향 일대일
  - 단방향 일대다
  - 단방향 다대일
  - 양방향 일대다/다대일
  - 단방향 다대다
  - 양방향 다대다

## 엔티티 매핑(Entity Mapping) – 1:1, 단방향, 대상테이블에 외래키



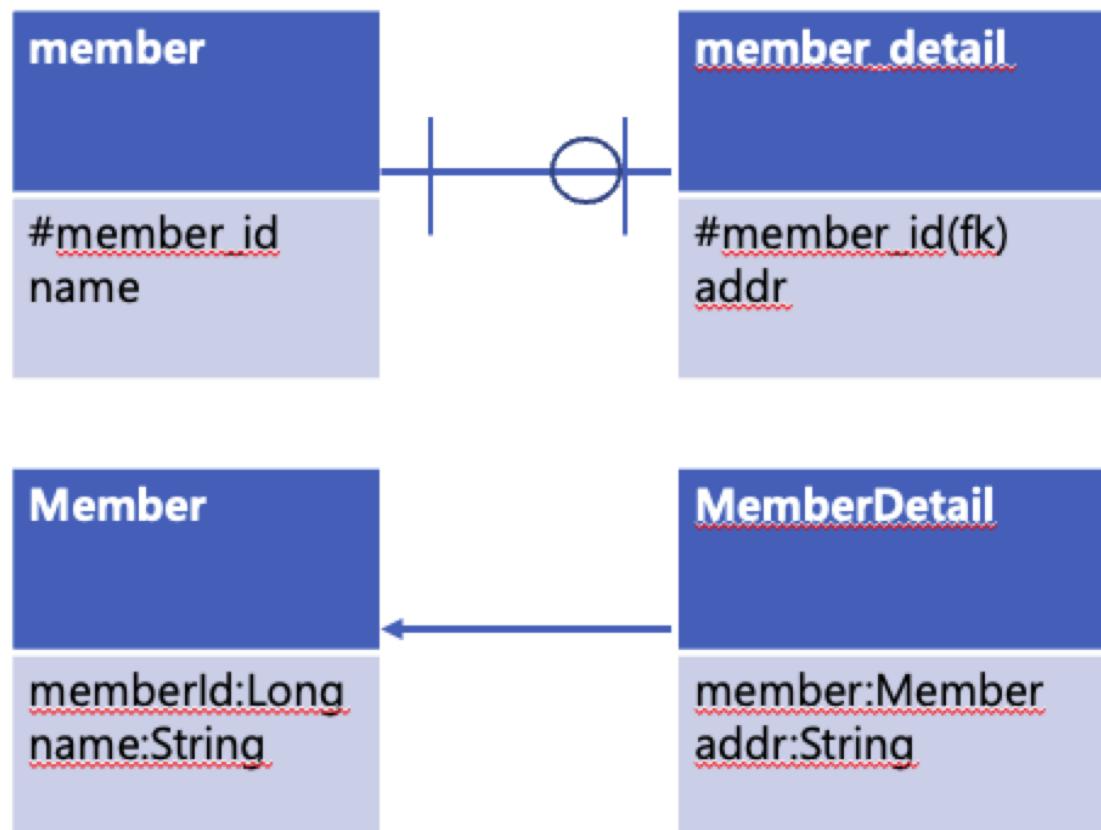
# 엔티티 매핑(Entity Mapping) – 1:1, 단방향, 대상테이블에 외래키

```
@Entity  
class Member{  
    @Id @GeneratedValue  
    private Long memberId;  
  
    private String name;  
  
    @OneToOne(mappedBy="member")  
    private MemberDetail memberDetail ;  
}  
  
@Entity  
class MemberDetail{  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    private String addr;  
  
    @OneToOne  
    @JoinColumn(name="member_id")  
    private Member member;  
}
```

## 엔티티 매핑(Entity Mapping) – 1 : 1, 단방향, 대상테이블에 외래키

- MemberDetail 테이블쪽에 Member테이블의 주키가 내려오는 경우로 JPA에서는 1:1관계에서 대상테이블에 외래키가 있는 경우는 지원하지 않는다. 1:N 단방향 매핑은 JPA2.0 부터 가능하다.
- 1:1, 단방향 매핑에서 대상 테이블에 외래키가 있는 것을 지원하지 않으므로 양방향으로 만들고 Member쪽에 mappedBy를 사용하여 MemberDetail을 Owner로 만들어야 한다.

# 엔티티 매핑(Entity Mapping) – 1 : 1 식별관계, 단방향 , 대상테이블에 외래 키



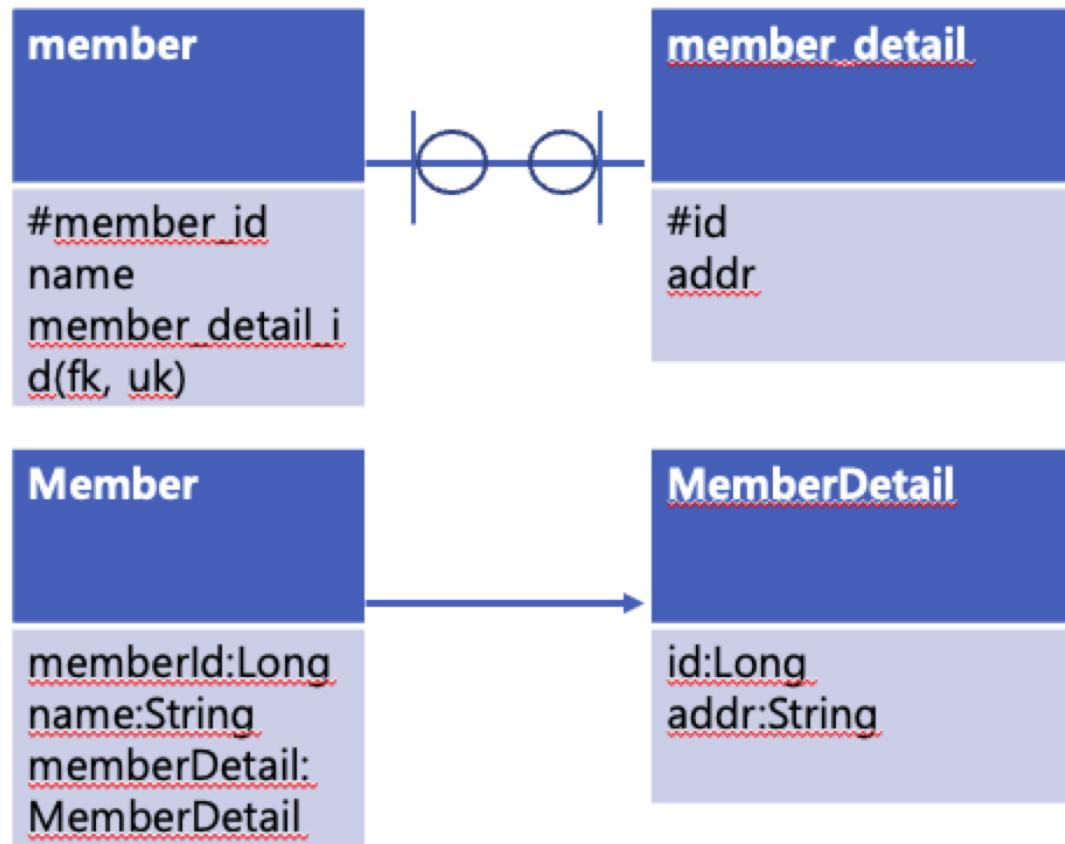
# 엔티티 매핑(Entity Mapping) – 1:1 식별관계, 단방향, 대상테이블에 외래 키

```
@Entity  
class Member{  
    @Id  
    @GeneratedValue  
    private Long memberId;  
  
    private String name;  
}  
  
@Entity  
class MemberDetail{  
    @Id  
    private Long id;  
  
    private String addr;  
  
    @OneToOne  
    @MapsId  
    @JoinColumn(name="member_id")  
    private Member member;  
}
```

## 엔티티 매핑(Entity Mapping) – 1:1, 단방향, 대상테이블에 외래키

- 주 테이블의 PK가 자식테이블에 식별자(PK) 형태로 외래키로 내려가는 경우이며, 부모테이블의 기본키를 자식 테이블에서도 기본키로 사용하는 경우이며 자식테이블의 기본키로는 부모테이블에서 내려오는 외래키만을 사용하고, 식별자가 하나인 경우에는 @MapsId를 사용한다.

## 엔티티 매핑(Entity Mapping) – 1:1, 단방향, 주 테이블에 외래키



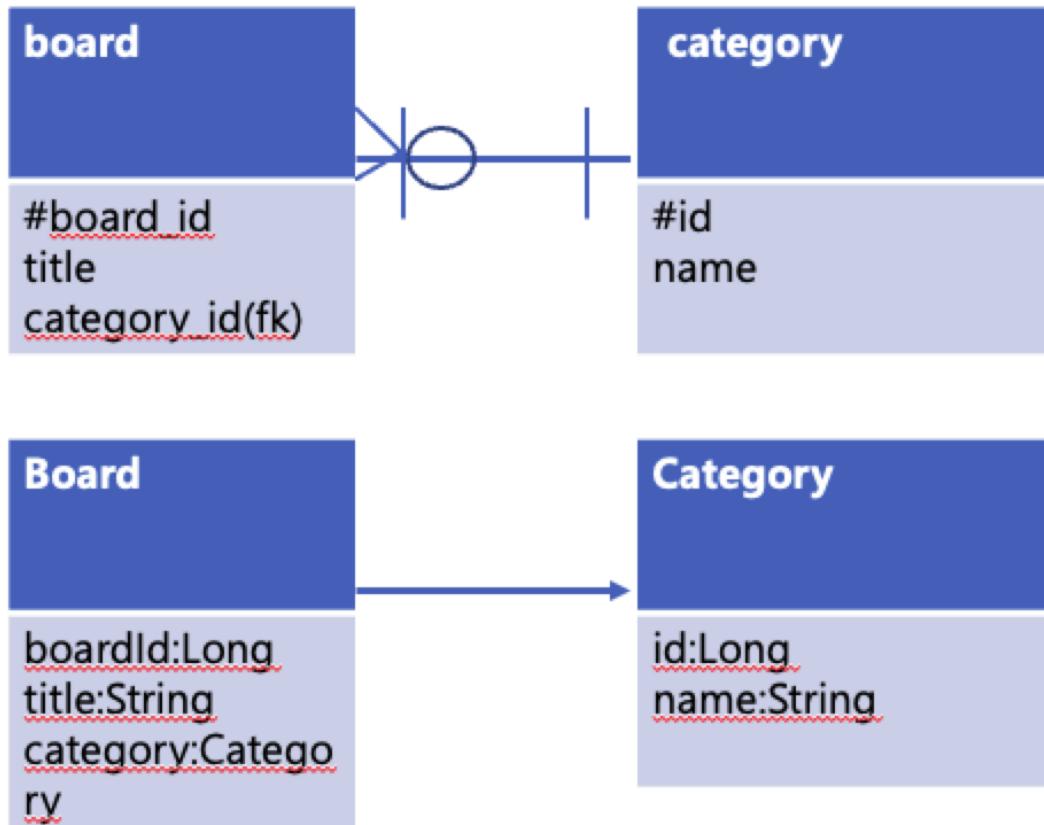
# 엔티티 매핑(Entity Mapping) – 1:1, 단방향, 주 테이블에 외래키

```
@Entity  
class Member{  
    @Id  
    @GeneratedValue  
    private Long memberId;  
  
    private String name;  
  
    @OneToOne(cascade = CascadeType.ALL)  
    @JoinColumn(name = "member_detail_id")  
    private MemberDetail memberDetail ;  
}  
  
@Entity  
class MemberDetail{  
    @Id  
    private Long id;  
  
    private String addr;  
}
```

## 엔티티 매핑(Entity Mapping) – 1 : 1, 단방향 , 주 테이블에 외래키

- 1:1 관계에서는 주 테이블이나 대상 테이블 모두 외래키를 가질 수 있다아래는 주테이블인 Member 테이블에 외래키가 있고 주테이블에서 대상 테이블을 참조하는 단방향 관계이다.

## 엔티티 매핑(Entity Mapping) – N:1 단방향



# 엔티티 매핑(Entity Mapping) – N:1 단방향

```
@Entity  
class Board{  
    @Id  
    @GeneratedValue  
    private Long boardId;  
  
    private String title;  
  
    @ManyToOne  
    @JoinColumn(name = "category_id")  
    private Category category ;  
}  
  
@Entity  
class Category{  
    @Id  
    private Long id;  
  
    private String name;  
}
```

## @OnDelete

```
@ManyToOne  
@OnDelete(action=onDeleteAction.CASCADE)  
private Dept dept;
```

- @OnDelete를 사용하여 ON DELETE CASCADE 제약조건을 생성 할 수 있다. JPA에서 생성하는 테이블 생성 코드(CREATE TABLE EMP...)에서 deptno 칼럼 정의하는 부분에 on delete cascade 옵션을 추가해 준다.

## @ForeignKey

```
@ManyToOne  
 @JoinColumn(name="deptno")  
 @ForeignKey(name="fk_dept_emp")  
 private Dept dept;
```

- @ForeignKey를 사용하여 외래키를 지정할 수 있다.

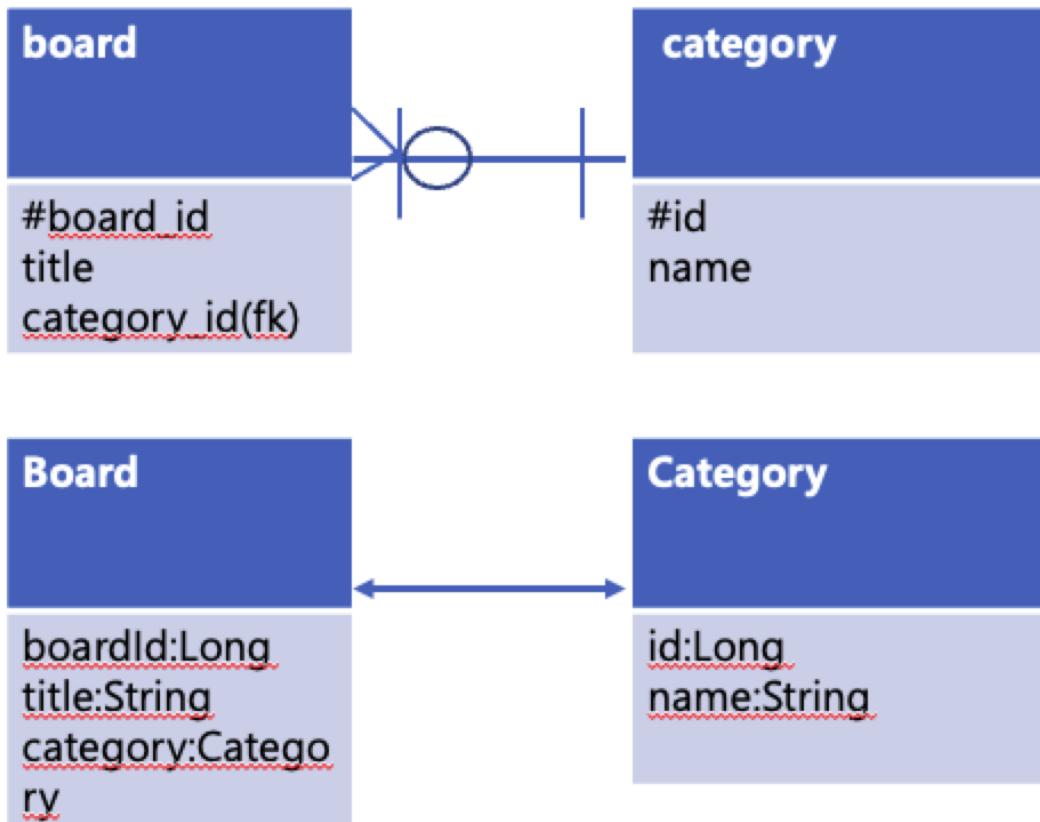
## @ManyToOne 속성 1/2

- optional : false인 경우 연관되는 데이터가 항상 있어야 한다.
- fetch : 데이터 패치 전략, 즉시로딩(FetchType.EAGER), 자연로딩(FetchType.LAZY) 값이 있으 며 추천 방법은 모든 연관관계에 자연 로딩을 사용하고 상황에 따라 꼭 필요한 곳에만 즉시 로 딩을 사용하는 것이 좋다. 자연로딩은 어떤 객체의 필요한 속성정보가 있을 경우 사용하기 직전에 로딩한다는 의미이고 즉시로딩은 객체가 참조될 때 그 즉시 로딩한다는 의미이다.
  - 기본값 :
  - @OneToMany : FetchType.LAZY
  - @ManyToOne : FetchType.EAGER
  - @ManyToMany : FetchType.LAZY
  - @OneToOne : FetchType.EAGER

## @ManyToOne 속성 2/2

- cascade : 영속성전이를 뜻하는데 어떤 객체를 영속성 객체로 만들 때 연관된 객체도 같이 영속성 객체로 만든다. CascadeType은 여러 종류가 있다.
  - ALL : 부모의 영속성 변화가 자식에게 모두 전이 시킨다. (부모가 영속화되면 자식도 영속화되고, 부모가 저장되면 자식도 저장되고, 부모가 삭제되면 자식도 삭제된다.)
  - PERSIST: 부모가 영속화 될 때 자식도 영속화 된다.
  - MERGE : 트랜잭션이 종료되고 detach 상태에서 연관 엔티티를 추가하거나 변경된 이후에 부모 엔티티가 merge()를 수행하게 되면 변경사항이 적용된다. 연관 엔티티의 추가 및 수정 모두 반영된다.
  - REMOVE : 부모를 삭제할 때 연관된 자식 엔티티도 삭제된다.
  - REFRESH : 부모 엔티티가 REFRESH(DB에서 값을 다시 읽음)되면 연관도 자식 엔티티도 REFRESH 된다.
  - DETACH : 부모 엔티티가 detach를 수행하면 연관된 엔티티도 detach 상태가 되어 영속성 컨텍스트를 빠져나와 변경 사항이 반영되지 않는다.

# 엔티티 매핑(Entity Mapping) – N:1 양방향



# 엔티티 매핑(Entity Mapping) – N:1 양방향

```
@Entity  
class Board{  
    @Id  
    @GeneratedValue  
    private Long boardId;  
  
    private String title;  
  
    @ManyToOne  
    @JoinColumn(name = "category_id")  
    private Category category ;  
}  
  
@Entity  
class Category{  
    @Id  
    private Long id;  
  
    private String name;  
  
    @OneToMany(mappedBy="category")  
    //Set이외 List, Map등 사용가능 하다.  
    private Set<Board> boards;  
}
```

## 엔티티 매핑(Entity Mapping) – N:1 양방향

- 양방향 관계에서는 @OneToMany쪽에 mappedBy를 표시해서 연관관계의 주인인 아니라고 반드시 표시해야 하는데 이렇게 선언된 필드는 DB 테이블에 칼럼으로는 생성되지 않고 단지 ~~에 의해 매핑된다라고 해석하면 된다. Category 엔티티의 boards 필드는 DB에 칼럼으로 생성되지 않고 단지 Board쪽의 category 필드에 의해 매핑된다는 의미이다.
- @OneToMany쪽에 mappedBy 속성으로 주인이 아니라고 표시해야 하며 자신의 객체참조를 가리키는 Owner 테이블(외래키 테이블, Board)의 필드명을 지정하면 된다. (Owner쪽은 @ManyToOne이 된다.) 양방향 관계에서 관계 연결처리를 하는 쪽, 외래키가 있는쪽(多)을 Owner라 한다. 다(多) 쪽인 @ManyToOne은 항상 연관관계의 주인(Owner)이 되므로 mappedBy를 설정할 필요 없다.

## 엔티티 매핑(Entity Mapping) – N:1 양방향

- 양방향 관계에서 @ManyToOne은 Owning Side이며, @OneToMany는 Inverse Side이다.
- 양방향 관계에서 Inverse Side의 mappedBy는 @OneToOne, @OneToMany 또는 @ManyToMany에 사용되고 Owning Side의 InverseBy는 @OneToOne, @ManyToOne 또는 @ManyToMany에서 사용하는 속성이다.
- boards 필드는 자연, 즉시 읽기에 따라 로딩되는 시점은 다르지만 Category 엔티티가 DB에서 검색될 때 자동으로 활성화 된다.
- Board 엔티티는 주인(OWNER)이며 주인 엔티티에서 일어난 변화만 DB에 반영되며 반대쪽 엔티티인 Category 엔티티의 변화는 DB에 저장되지 않으며 검색시에만 참조된다.
- 대개 Owner쪽(Board)의 category 필드는 외래키로 인덱스로 이용되어 검색속도가 향상 될 수 있다.

## @OneToMany

- N:1 양방향관계에서 one 쪽에서 many 쪽으로의 방향성 또는 1:N 단방향에서 many쪽의 데 이터를 참조하므로 Set, List, Map을 사용할 수 있다.
- List는 순서가 있고 중복을 허락하는 자료구조 이므로 자료들의 순서를 알 수 있고, 정렬이 가능하다. 자료들의 순서(인덱스)를 알기 위해서는 별도 인덱스 칼럼(@OrderColumn)이 정의되어야 한다.
- Map은 Key, Value 쌍으로 자료를 저장하는데 key설정을 위해 @MapKey를 사용하며, 실제 테이블의 칼럼을 지정할 때는 @MapKeyColumn을 사용한다.

## @OneToMany 속성

- targetEntity : 연결을 맺는 상대 엔티티
- cascade : 연관 엔티티의 영속성전이 전략을 설정.
- mappedBy : 양방향 관계에서 주체가 되는 쪽(Many쪽, 외래키가 있는 쪽)의 자신을 가리키는 필드명을 기술하여 주인이 아님을 표시.
- orphanRemoval : 엔티티에서 삭제가 일어난 경우 연관관계에 있는 엔티티도 같이 삭제할지의 여부를 결정한다. Cascade는 JPA 레이어의 정의이고 이 속성은 DB레이어에서 직접 처리한다. 기본은 false.

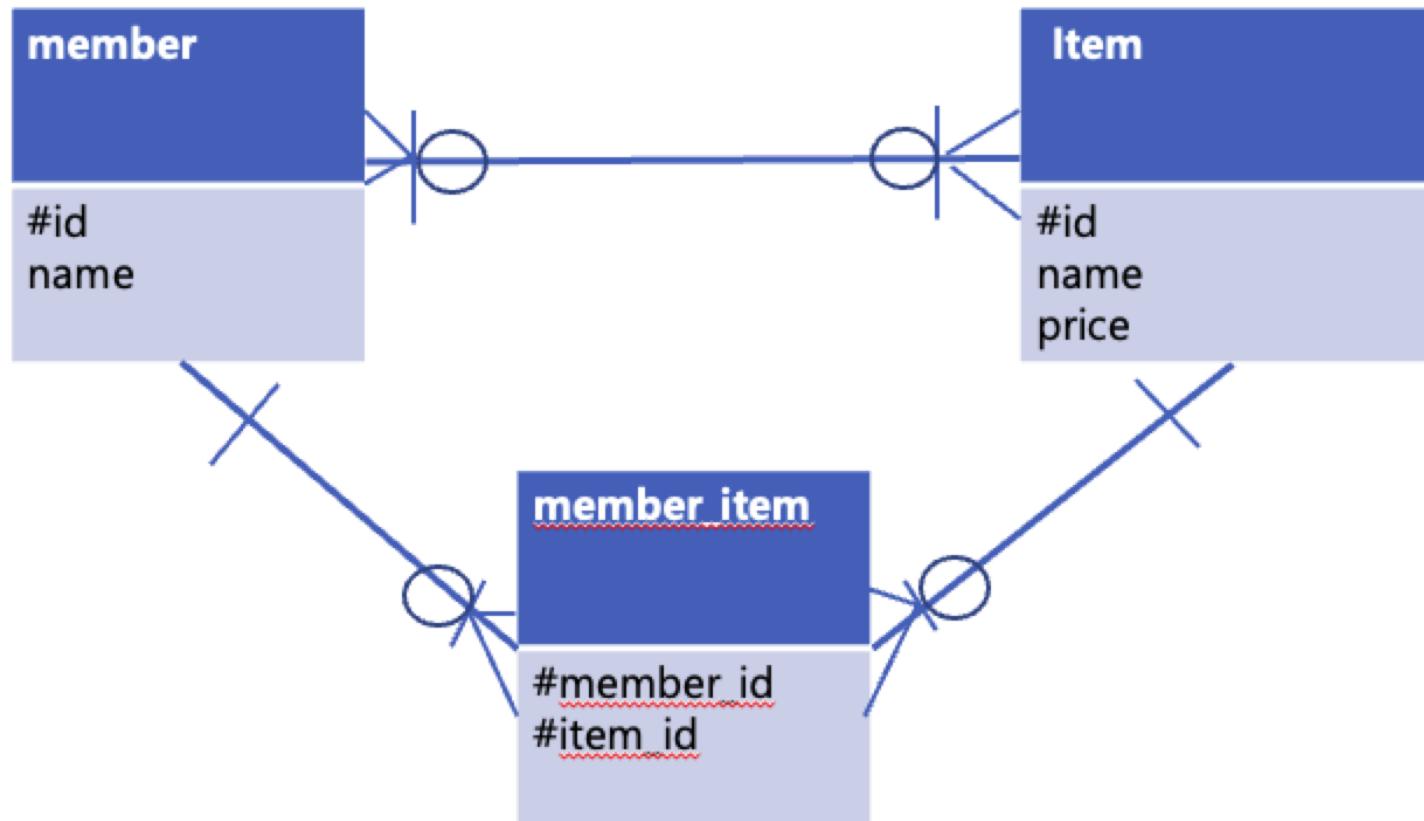
## mappedBy 1/2

- 테이블은 외래키 하나로 두 테이블의 연관관계를 관리하는데 엔티티를 단방향으로 매핑하면 참조를 하나만 사용하고, 양방향 관계로 설정하면 객체의 참조는 양쪽에서 하나씩 둘인데, 외래키는 하나이므로 두 엔티티 중 하나(OWNER)를 정해서 테이블의 외래키를 관리해야 한다. MANY쪽(Board)이 OWNER이며 mappedBy는 @OneToMany쪽(Category)의 컬렉션 칼럼(boards)에 기술하여 OWNER가 아님을 정의한다. (테이블의 칼럼으로 만들어지지 않는다.)
- mappedBy는 @OneToOne, @OneToMany, @ManyToMany 어노테이션에서 사용할 수 있으며 mappedBy가 없으면 JPA에서 양방향 관계라는 것을 모르고 두 엔티티의 매핑 테이블을 생성한다.

## mappedBy 2/2

- mappedBy의 값은 OWNER쪽 엔티티(Board)의 필드(category)에 대응된다. 즉 Category 엔티티의 boards 칼럼은 DB에 생성되는 칼럼이 아니고 EMP쪽의 dept 필드와 단지 매핑되는 필드임을 정의하는 것이다.
- ManyToOne 양방향 관계에서 Many측에는 mappedBy요소를 사용할 수 없다.(MANY 쪽이 OWNER), OneToOne 양방향 관계 OWNER는 반대쪽(VERSE SIDE)에 대한 FK를 가지는 쪽이다.
- ManyToMany 양방향 관계는 양쪽 중 아무나 OWNER가 될 수 있다.

## M:N 연관관계



## M:N 연관관계



## M:N 연관관계 1/2

- 회원(Member), 아이템(Item)은 다 : 다 관계이다. 회원은 여러 아이템을 가질 수 있고, 하나의 아이템 역시 여러 회원에게 할당 될 수 있기 때문이다. 보통 관계형 DB에서는 다 : 다 관계는 1 : 다, 다 : 1로 나누어서 풀게 된다. 그러나 객체에서는 보통 2개의 엔티티 클래스로 다 : 다 관계를 만드는데, 회원에서 아이템을 컬렉션에 넣어 접근 가능하고, 반대로 아이템에서도 회원들을 컬렉션에 넣어 접근하면 양쪽에서 접근이 가능하다.
- 만약 조인 테이블이 추가적인 별도의 속성(필드)를 가져야 한다면 조인 테이블에 대응되는 조인 클래스(엔티티 클래스)도 반드시 만들어야 한다. 아래 예제의 경우 조인 테이블의 추가적인 별도의 속성을 가지지 않으므로 엔티티 클래스는 2개로 작성 되었다.

## M:N 연관관계 2/2

- 다 : 다 매핑을 위해 @ManyToMany 어노테이션을 사용하며 RDB에 테이블로 생성될 때는 다 : 다인 테이블을 연결시켜주기 위한 조인 테이블이 있어야 한다. 회원(member), 아이템 (item) 관계는 다:다 관계인데 이 둘을 연결시켜주기 위한 조인 테이블이 있어야 한다는 것이다. 엔티티 클래스는 두개로 표현하지만 DB에 생성될 때 조인테이블이 생성되며 조인 테이블을 지정할 때는 @JoinTable 어노테이션을 사용한다.
  - 다 : 다 양방향 관계에서는 한쪽을 Owning Side, 다른쪽을 Inverse Side라고 하는데 @ManyToMany 어노테이션에 mappedBy 속성을 사용하여 Inverse Side 라고 표시를 한다. 즉 연관관계의 주인이 아님을 지정한다. (mappedBy가 없는 곳이 Owning Side)

# Member Entity

```
private List<Item> items;  
.....  
@ManyToMany(cascade = {CascadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH})  
@JoinTable(name = "member_item",  
          joinColumns = @JoinColumn(name = "member_id", referencedColumnName = "id") ,  
          inverseJoinColumns = @JoinColumn(name = "item_id", referencedColumnName = "id") )  
@OrderColumn(name="item_order") //member_item에 item_order칼럼 생성  
public List<Hobby> getItems() {  
    return items;  
}
```

## Item Entity

```
private Set<Member> members;  
  
@ManyToMany(mappedBy = "items")  
public Set<Member> getMembers() {  
    return members;  
}
```

## 엔티티 설명

- 만약 @JoinTable 어노테이션으로 조인테이블을 지정하지 않으면 기본적인 이름은 주인테이블 이름\_반대쪽 테이블을 가리키는 주인테이블의 필드명(mappedby에서 기술한 값)이 된다. 또한 조인테이블에 생성되는 Primary Key의 순서도 주인 테이블의 Primary Key에 대한 외래키가 먼저 위치한다.
- 소스 오브젝트의 Primary Key에 대한 외래키는 JoinColumns 속성으로, 타겟 오브젝트의 Primary Key에 대한 외래키는 InverseJoinColumns 속성으로 지정한다.

## 식별자 자동생성 (@GeneratedValue) 1/3

- 식별자 자동생성은 @GeneratedValue 어노테이션으로 지정한다.
- 복합키 보다는 대행키(인공키, Artifitial Key) 사용을 권장한다.  
@GeneratedValue의 strategy 속성에 값을 지정해 여러 가지 식별자 자동 생성 전략을 선택할 수 있는데 AUTO, TABLE, SEQUENCE, IDENTITY 값으로 지정한다. 이 값들은 열거형인 GenerationType에 정의되어 있다.
- GenerationType.AUTO : 데이터베이스에 관계없이 식별자를 자동 생성 하라는 의미, DB가 변경되더라도 수정할 필요 없다.

## 식별자 자동생성 (@GeneratedValue) 2/3

- JPA 공급자에 따라 기본설정이 다름
  - Oracle을 사용할 경우 SEQUENCE가 기본
  - MS SQL Server의 경우 IDENTITY가 기본
- 오라클이라면 하이버네이트 글로벌 시퀀스(hibernate\_sequence)를 사용하므로 엔티티에서 두 군데 이상 AUTO로 사용한다면 시퀀스를 별도로 만들고 명시적으로 SequenceGenerator를 기술하여 사용하는 것이 좋다.
- GenerationType.TABLE : 가장 유연하고 이식성이 좋은 방법으로 키를 위한 테이블이 생성되어 있어야 하며, ID 생성 테이블은 두 개의 칼럼을 반드시 가져야 한다. 처음 칼럼은 특정 시퀀스를 식별하는데 사용되는 문자열 타입이고 모든 Generator를 위한 위한 주키다. 두 번째 칼럼은 생성되고 있는 실제 ID 값인 시퀀스를 저장하는 숫자 타입으로 이 칼럼에 저장되는 값은 시퀀스에 할당되었던 마지막 값이 된다. 하나의 테이블에 여러 식별자 값을 저장할 수 있다.

## 식별자 자동생성 (@GeneratedValue) 3/3

- GenerationType.SEQUENCE : DB에서 생성해 놓은 시퀀스를 이용하는 방법으로 오라클, DB2, PostgreSQL, H2 데이터베이스 등에서 사용한다.
- GenerationType.IDENTITY : MySQL, MS-SQL 처럼 DB 자체적으로 식별자 칼럼에 자동증분 속성이 있는 경우에 사용한다. 엔티티의 주키 칼럼을 위한 정의는 DB 스키마 정의의 일부분으로 정의되어야 한다.

# Spring Data JPA

- Spring + JPA

## Query Mehtod

- find...By : findBoardByTitle
- read...By : readBoardByTitle
- query...By : queryBoardByTitle
- get...By : getBoardByTitle
- count...By : countBoardByTitle

## **Query Method - find by 특정 칼럼 처리**

- Collection findBy + 속성이름(속성타입)

## Query Method - like 구문처리

- 단순 like : Like
- 키워드 + '%' : StartingWith
- '%' + 키워드 " : EndingWith
- '%' + 키워드 + '%' : Cotaining

## Query Method - and 혹은 or 조건 처

- 2개 이상 속성 검색할 경우 And or Or를 사용  
ex> findByTitleContainingOrContentContaining

## Query Method - 부등호 처리

- 와 < 부등호 처리는 GreaterThan 과 LessThan을 이용

```
public Collection<Board> findByTitleContainingAndBnoGreaterThanOrLessThan(String keyword, Long num);
```

## Query Method - order by 처리

- OrderBy + 속성 + Asc or Desc를 이용

```
public Collection<Board> findByBnoGreaterThanOrOrderBnoDesc(Long bno);
```

## Query Method - 페이징 처리와 정렬

- 모든 쿼리 메소드의 마지막 파라미터로 페이지 처리를 할 수 있는 Pageable 인터페이스와 정렬을 처리하는 Sort 인터페이스를 사용할 수 있다.
- org.springframework.data.domain.Pageable 인터페이스 구현한 PageRequest 클래스.
- boot 2.0에서 new PageRequest()는 deprecated 되어서 PageRequest.of()를 사용.
- bno > 0 order by bno desc 조건  

```
public Page<Board> findByBnoGreaterThanOrOrderByBnoDesc(Long bno, Pageable paging);
```
- 파라미터에 Pageable 적용되고, 리턴타입이 Collection<> 대신 Slice<>, Page<>, List<>로 이용.

## Query Method - 페이지 생성자 설정

- final Pageable paging = PageRequest.of(0, 10); // 페이지 번호(0부터 시작), 페이지당 데이터수

## Query Method - 페이징 생성자에 정렬 객체 전달 가능

- `PageRequest(int page, int size, SortDirection direction, String... props);` // 페이지 번호, 페이지당 데이터 수, 정렬 방향, 속성(칼럼) 들
- `new PageRequest(0, 10, Sort.Direction.ASC, "bno");`

## JPA JPQL

- JPQL(Java Persistence Query Language)은 지속적 엔티티를 저장하는 데 사용되는 메커니즘과 독립적으로 지속적 엔티티에서 검색을 정의하는 데 사용되는데 데이터베이스와 관련된 SQL을 사용하지 않고 오브젝트를 검색하는 언어이다.

# JPQL 쿼리를 기술하는 방법

- JPQL(Java Persistence QueryLanguage)
  - SQL처럼 JPA독자적인 쿼리 언어를 써서 Entity를 가져오거나 값을 변경할 때 사용한다.  
SQL이 데이터를 입출력할 때 테이블이나 테이블의 행과 열로 표현하는 반면, JPQL은 Entity나 Entity의 컬렉션, 그리고 Entity의 프로퍼티명으로 표현하는 점이 다르다.
- Criteria Query
  - JPQL과 개념은 비슷하나 좀 더 객체지향적으로 기술하는 방법으로 JPA 2.0에서 추가되었다. JPQL은 문자열로 기술되기 때문에 타입 검사를 컴파일 시점에 할 수 없어 타입 불일치 같은 오류가 잠재적으로 발생할 수 있다. 한편 Criteria Query는 문자열이 아니라 Builder 패턴의 CriteriaQuery객체를 이용해 자바 코드처럼 쿼리를 기술하기 때문에 컴파일 시점에 타입 검사를 할 수 있어 쿼리 작성 과정에서 발생하는 실수를 미연에 방지할 수 있다.
- Native Query
  - SQL을 직접 기술해서 Entity를 취득하거나 갱신하는 방법으로, 성능 등의 다양한 이유로 데이터베이스 제품에 의존적인 최적화된 기능이 필요할 때 이 방법을 사용한다.

## JPQL 기본 문법

```
select  
from  
[where]  
[group by]  
[having]  
[orderby]
```

- jpql을 쉽게 도와주는 도구들 : Criteria 쿼리, QueryDSL

# Spring Security

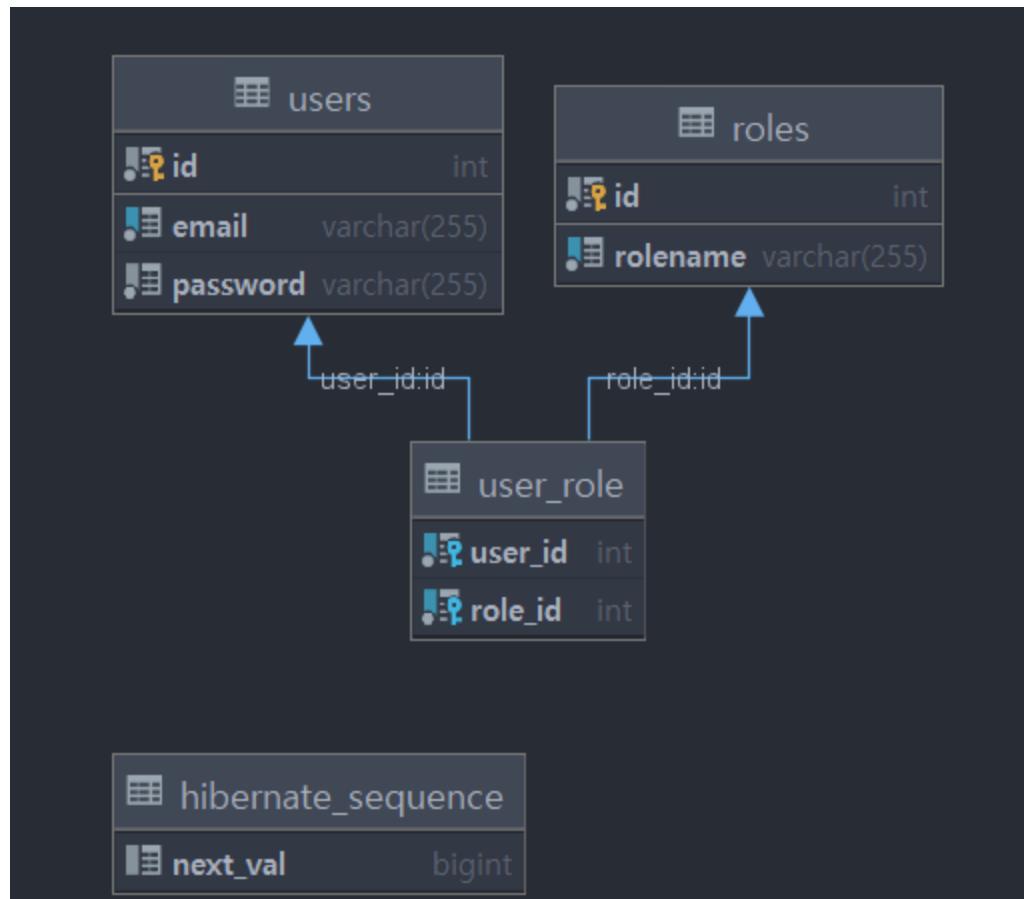
# pom.xml 파일에 라이브러리를 추가한다.

```
<!-- spring security -->
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>${spring-security.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
    <version>${spring-security.version}</version>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>${spring-security.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-taglibs</artifactId>
    <version>${spring-security.version}</version>
</dependency>
```

# 회원 정보와 권한 정보를 ManyToMany로 하여 생성한다.

- email, password로 로그인한다.
- 권한 이름은 반드시 ROLE\_ 로 시작 해야 한다. (Spring Security가 내부적으로 그렇게 읽어들임)



# Role.java

```
package com.exam.todojpa.domain;

import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.Table;

@Entity
@Table(name="roles")
public class Role
{
    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    private Integer id;

    @Column(nullable=false, unique=true)
    private String rolename;

    @ManyToMany(mappedBy="roles")
    private List<User> users;

    public Role(){} // 기본 생성자

    public Role(String rolename) {
        this.rolename = rolename;
    }
}
```

```
public Integer getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

public String getRolename() {
    return rolename;
}

public void setRolename(String rolename) {
    this.rolename = rolename;
}

public List<User> getUsers() {
    return users;
}

public void setUsers(List<User> users) {
    this.users = users;
}

}
```

# User

```
package com.exam.todojpa.domain;

import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.Table;

@Entity
@Table(name="users")
public class User
{
```

```
@Id  
@GeneratedValue(strategy= GenerationType.AUTO)  
private Integer id;  
  
@Column(nullable=false) // Null이 되면 안됨.  
private String password;  
  
@Column(nullable=false, unique=true) //unique=true 해야함.  
private String email;  
  
@ManyToMany(cascade=CascadeType.MERGE)  
@JoinTable(  
    name="user_role",  
    joinColumns={@JoinColumn(name="USER_ID", referencedColumnName="ID")},  
    inverseJoinColumns={@JoinColumn(name="ROLE_ID", referencedColumnName="ID")})  
private List<Role> roles;  
  
public User() {} // 기본 생성자
```

```
public Integer getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}
```

```
public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public List<Role> getRoles() {
    return roles;
}

public void setRoles(List<Role> roles) {
    this.roles = roles;
}

@Override
public String toString() {
    return "User [id=" + id + ", email=" + email + "]";
}

}
```

# RoleRepository

- 회원 가입시에 필요하다. 로그인시에는 필요없음.

```
package com.exam.todojpa.repository;  
  
import java.util.Optional;  
  
import org.springframework.data.jpa.repository.JpaRepository;  
  
import com.exam.todojpa.domain.Role;  
  
public interface RoleRepository extends JpaRepository<Role, Integer> {  
    Optional<Role> findByRolename(String rolename);  
}
```

# UserRepository

- 로그인할 때 필요한 findByEmail메소드를 추가한다.

```
package com.exam.todojpa.repository;  
  
import java.util.Optional;  
  
import org.springframework.data.jpa.repository.JpaRepository;  
  
import com.exam.todojpa.domain.User;  
  
public interface UserRepository extends JpaRepository<User, Integer>{  
    Optional<User> findByEmail(String email);  
}
```

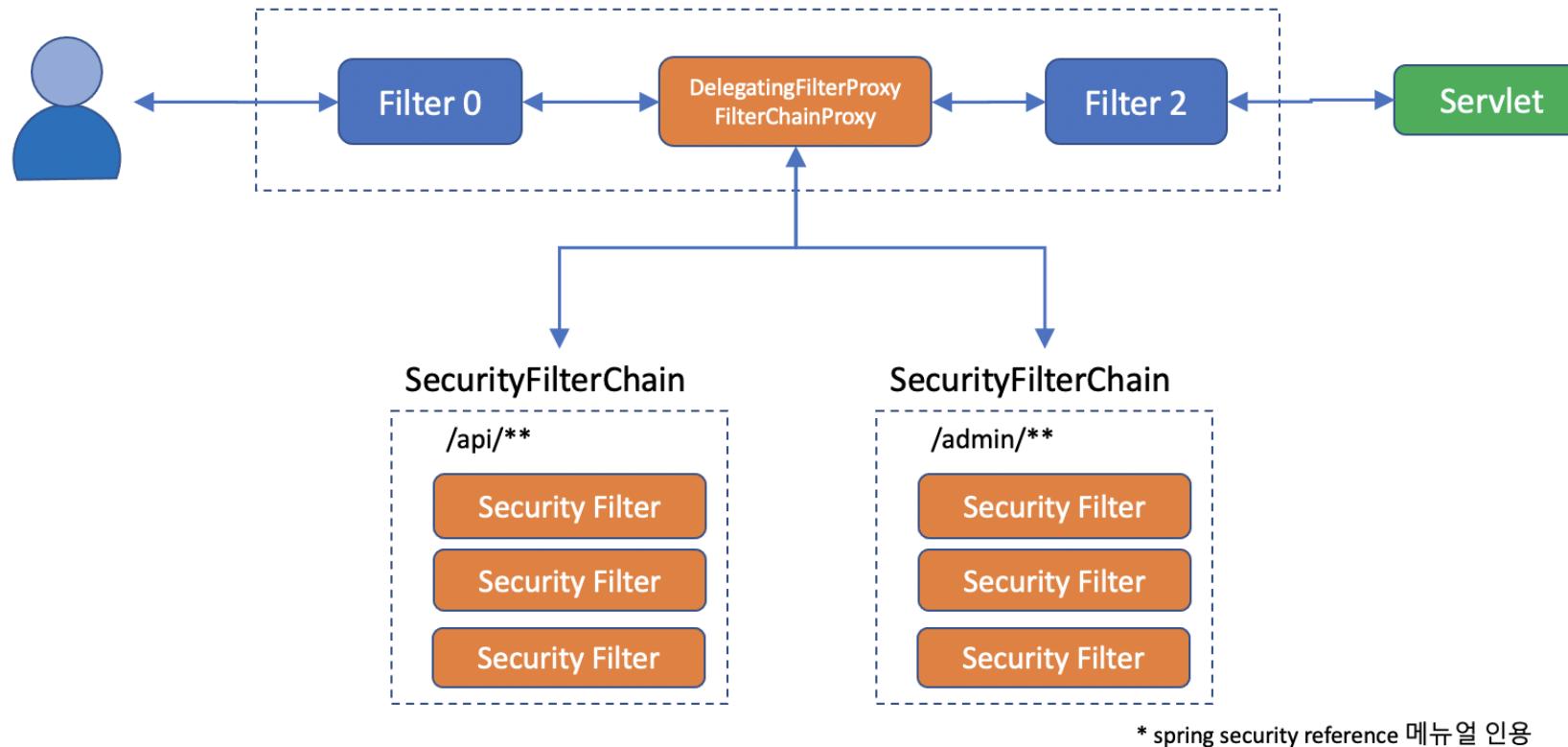
# Spring Security 용어

- 인증 Authentication : 해당 사용자가 본인이 맞는지를 확인하는 절차
- 인가 Authorization : 인증된 사용자가 요청한 자원에 접근 가능한지를 결정하는 절차
- 접근 주체 Principal : 보호받는 Resource에 접근하는 대상
- 비밀번호 Credential : Resource에 접근하는 대상의 비밀번호
- 권한 : 인증된 주체가 어플리케이션의 동작을 수행할 수 있도록 허락되어 있는지를 결정한다. - 인증 과정을 통해 주체가 증명된 이후 권한을 부여할 수 있다. - 권한 부여에도 두 가지 영역이 존재하는데 웹 요청 권한과 메서드 호출 및 도메인 인스턴스에 대한 접근 권한 부여가 있다.

## 인증 (Authentication) -> 인증 성공 후 -> 인가 (Authorization)

- Spring Security는 기본적으로 인증 절차를 거친 후에 인가 절차를 진행하게 되며, 인가 과정에서 해당 리소스에 대한 접근 권한이 있는지를 확인하게 된다. 이러한 인증과 인가를 위해 Principal을 아이디로, Credential을 비밀번호로 사용하는 인증 방식을 사용한다.

# Spring Security 필터



- Spring Security는 여러개의 Servlet Filter를 가지고 있음. 이것을 스프링 컨테이너가 Bean으로 관리하기 위해 DelegatingFilterProxy에 등록하여 사용

## Spring Security 필터

- Spring Security는 DelegatingFilterProxy 라는 필터를 만들어 메인 Filter Chain에 끼워넣고, 그 아래 다시 SecurityFilterChain 그룹을 등록한다.
- 그렇게 하며 URL에 따라 적용되는 Filter Chain을 다르게 하는 방법을 사용한다.
- 어떠한 경우에는 해당 Filter를 무시하고 통과하게 할 수도 있다.

# Security의 Filter의 종류

- HeaderWriterFilter : Request의 Http 해더를 검사하여 header를 추가하거나 빼주는 역할을 한다.
- CorsFilter : 허가된 사이트나 클라이언트의 요청인지 검사하는 역할을 한다.
- CsrfFilter : Post나 Put과 같이 리소스를 변경하는 요청의 경우 내가 내보냈던 리소스에서 올라온 요청인지 확인한다.
- LogoutFilter : Request가 로그아웃하겠다고 하는것인지 체크한다.
- UsernamePasswordAuthenticationFilter : username / password로 로그인을 하려고 하는지 체크하여 승인이 되면 Authentication을 부여하고 이동 할 페이지로 이동한다.
- ConcurrentSessionFilter : 동시 접속을 허용할지 체크한다.
- BearerTokenAuthenticationFilter : Authorization 해더에 Bearer 토큰을 인증해주는 역할을 한다.

## Security의 Filter의 종류

- BasicAuthenticationFilter : Authorization 헤더에 Basic 토큰을 인증해주는 역할을 한다.
- RequestCacheAwareFilter : request한 내용을 다음에 필요할 수 있어서 Cache에 담아주는 역할을 한다. 다음 Request가 오면 이전의 Cache값을 줄 수 있다.
- SecurityContextHolderAwareRequestFilter : 보안 관련 Servlet 3 스펙을 지원하기 위한 필터라고 한다.
- RememberMeAuthenticationFilter : 아직 Authentication 인증이 안된 경우라면 RememberMe 쿠키를 검사해서 인증 처리해준다.
- AnonymousAuthenticationFilter : 앞선 필터를 통해 인증이 아직도 안되었으면 해당 유저는 익명 사용자라고 Authentication을 정해주는 역할을 한다. (Authentication이 Null인 것을 방지!!)

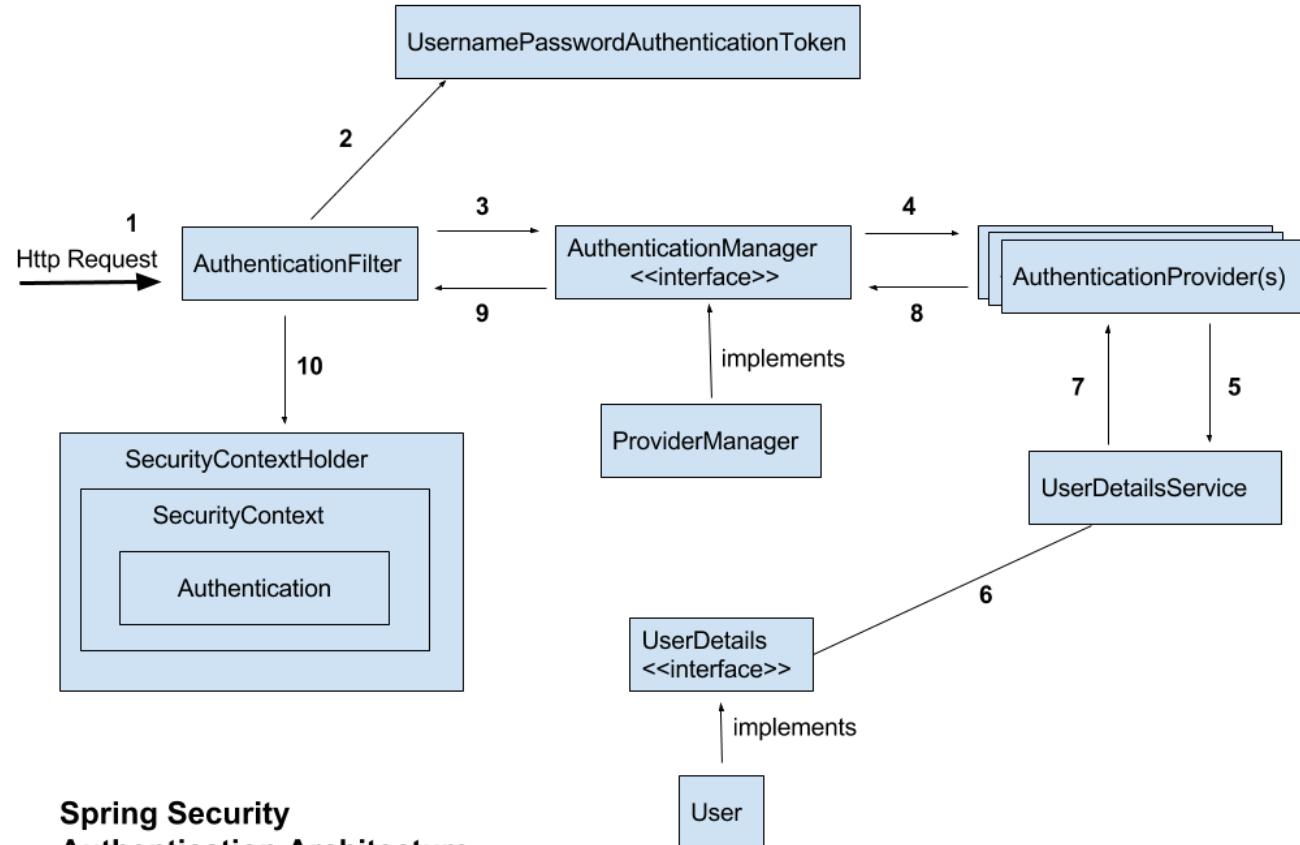
## Security의 Filter의 종류

- SessionManagementFilter : 서버에서 지정한 세션정책에 맞게 사용자가 사용하고 있는지 검사하는 역할을 한다.
- ExceptionTranslationFilter : 해당 필터 이후에 인증이나 권한 예외가 발생하면 해당 필터가 처리를 해준다.
- FilterSecurityInterceptor : 사용자가 요청한 request에 들어가고 결과를 리턴해도 되는 권한 (Authorization)이 있는지를 체크한다. 해당 필터에서 권한이 없다는 결과가 나온다면 위의 ExceptionTranslationFilter필터에서 Exception을 처리해준다.

## Filter Chain 확인하는 방법

- WebSecurityConfigurerAdapter을 상속받아 Filter Chain을 만드는 Class위에 @EnableWebSecurity(debug = true) 어노테이션을 붙여주면 현재 실행되는 Security Filter들을 확인할 수 있다.

# 일반적인 Form Login 절차



Chathuranga Tennakoon  
[www.springbootdev.com](http://www.springbootdev.com)

# 일반적인 Form Login 절차

## 1. 요청 수신

- 사용자가 form을 통해 로그인 정보가 담긴 Request를 보낸다.

## 2. 토큰 생성

- AuthenticationFilter가 요청을 받아서 UsernamePasswordAuthenticationToken 토큰(인증용 객체)을 생성
- UsernamePasswordAuthenticationToken은 해당 요청을 처리할 수 있는 Provider을 찾는데 사용

## 3. AuthenticationFilter로 부터 인증용 객체를 전달 받는다.

- Authentication Manager에게 처리 위임
- Authentication Manager는 List 형태로 Provider들을 갖고 있다.

## 일반적인 Form Login 절차

### 4. Token을 처리할 수 있는 Authentication Provider 선택

- 실제 인증을 할 AuthenticationProvider에게 인증용 객체를 다시 전달한다.

### 5. 인증 절차

- 인증 절차가 시작되면 AuthenticationProvider 인터페이스가 실행되고 DB에 있는 사용자의 정보와 화면에서 입력한 로그인 정보를 비교

### 6. UserDetailsService의 loadUserByUsername 메소드 수행

- AuthenticationProvider 인터페이스에서는 authenticate() 메소드를 오버라이딩 하게 되는데 이 메소드의 파라미터인 인증용 객체로 화면에서 입력한 로그인 정보를 가져올 수 있다.

## 일반적인 Form Login 절차

7. AuthenticationProvider 인터페이스에서 DB에 있는 사용자의 정보를 가져오려면, UserDetailsService 인터페이스를 사용한다.
8. UserDetailsService 인터페이스는 화면에서 입력한 사용자의 username으로 loadUserByUsername() 메소드를 호출하여 DB에 있는 사용자의 정보를 UserDetails 형으로 가져온다. 만약 사용자가 존재하지 않으면 예외를 던진다. 이렇게 DB에서 가져온 이용자의 정보와 화면에서 입력한 로그인 정보를 비교하게 되고, 일치하면 Authentication 참조를 리턴하고, 일치 하지 않으면 예외를 던진다.
9. 인증이 완료되면 사용자 정보를 가진 Authentication 객체를 SecurityContextHolder에 담은 이후 AuthenticationSuccessHandle를 실행한다.(실패시 AuthenticationFailureHandler를 실행한다.)

## Form 로그인시 사용되는 객체

### 1. (UsernamePassword)AuthenticationFilter

- 아이디와 비밀번호를 사용하는 form 기반 인증
- 설정된 로그인 URL로 오는 요청을 감시하며, 유저 인증 처리인 AuthenticationManager를 통한 인증 실행
- 인증이 성공한다면 인증용 객체를 SecurityContext에 저장 후 AuthenticationSuccessHandler 실행
- 실패한다면 AuthenticationFailureHandler 실행

## Form 로그인시 사용되는 객체

### 2. AuthenticationProvider

- 화면에서 입력한 로그인 정보와 DB정보를 비교
- Spring Security의 AuthenticationProvider을 구현한 클래스로 security-context에 provider로 등록 후 인증절차를 구현한다.
- login view에서 login-processing-url로의 form action 진행 시 해당 클래스의 supports() > authenticate() 순으로 인증 절차 진행

# Form 로그인시 사용되는 객체

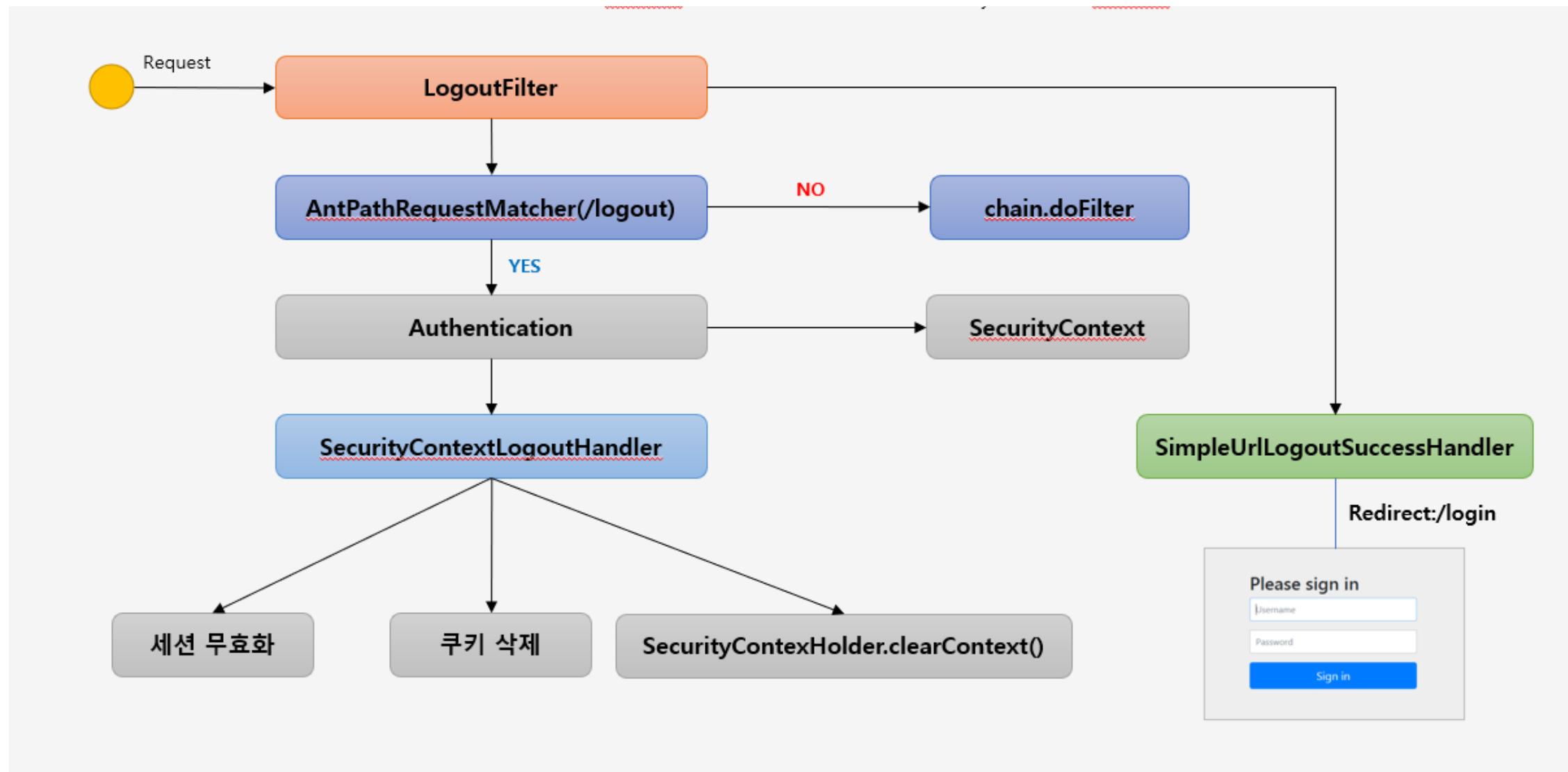
## 3. UserDetailsService

- UserDetailsService 인터페이스는 DB에서 유저 정보를 가져오는 역할
- public UserDetails loadUserByUsername(String username) 메소드를 사용자는 오버라이딩 한다.
  - username은 로그인 id를 의미한다. email로 로그인하게 했다면 username은 email을 말한다.
  - 사용자는 Database에서 정보를 읽어오도록 구현한다.
  - UserDetails인터페이스를 구현하는 객체에 로그인한 사용자 정보와 권한 정보를 담아서 리턴해야 한다.

## 4. UserDetails

- UserDetails는 username(로그인 id), 권한정보에 대한 메소드를 가진다.
- 원할 경우 UserDetails인터페이스를 구현하고 있는 객체에 username, 권한 정보 외에도 추가적인 정보를 가지게 할 수 있다. 로그인시 이 정보들을 읽어들일 수 있다.
- 해당 멤버변수들은 getter, setter를 만들어서 사용하면 된다.
- UserDetails를 구현하는 org.springframework.security.core.userdetails.User를 예제에서 사용하였다.

# logout 절차 이해하기



## logout 절차 이해하기

- Logout의 버튼을 누르는 순간부터 위의 로직이 동작한다. (/logout URL 접근하는 것만으로는 아무 일도 일어나지 않음)
- 현재 요청이 /logout이 맞는지 AntPathRequestMatcher를 이용해서 확인한다. 아니라면, 다른 요청이기 때문에 다음 필터로 넘겨준다.
- 로그아웃 요청이 왔다고 하면, Security Context로부터 인증객체를 꺼낸다.
- 꺼낸 인증객체를 SecurityContextLogoutHandler에게 전달해준다.
- LogoutHandler는 세션 무효화, 쿠키 삭제, SecurityContextHolder.clearContext()를 이용해 로그아웃 처리를 해준다.
- 처리가 완료되면, Logout Filter는 리다이렉트 URL을 따라 리다이렉팅 해준다.

# CustomUserDetailsService (UserDetailsService 구현체)

```
package com.exam.todojpa.service;

import java.util.Collection;

import javax.transaction.Transactional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.exam.todojpa.domain.User;
import com.exam.todojpa.repository.UserRepository;

@Service
@Transactional
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;
```

```
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    User user = userRepository.findByEmail(username)
        .orElseThrow(() -> new UsernameNotFoundException("Email: " + username + " not found"));
    return new org.springframework.security.core.userdetails.User(user.getEmail(),
        user.getPassword(), getAuthorities(user));
}

private static Collection<? extends GrantedAuthority> getAuthorities(User user){
    String[] userRoles = user.getRoles()
        .stream()
        .map((role) -> role.getRolename())
        .toArray(String[]::new);

    Collection<GrantedAuthority> authorities = AuthorityUtils.createAuthorityList(userRoles);
    return authorities;
}
```

## roles 테이블에 미리 저장해야 할 값

- 반드시 값은 ROLE\_ 로 시작 해야 한다.
- 스프링 설정에선 "USER", "ADMIN"으로 사용한다. (ROLE\_는 생략하고 사용)

```
insert into ROLES values (1, 'ROLE_USER');
insert into ROLES values (2, 'ROLE_ADMIN');
```

# SecurityWebApplicationInitializer

- `AbstractSecurityWebApplicationInitializer`는 `WebApplicationInitializer`의 구현이므로 스프링에 의해 찾아지고 웹 컨테이너와 `DelegatingFilterProxy`를 등록하는데 사용된다.

```
package com.exam.todojpa.config;

import org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;

public class SecurityWebApplicationInitializer
    extends AbstractSecurityWebApplicationInitializer {
}
```

## Filter Chain이 적용되는 URL 설정하는 방법

- 해당 filter를 동작시킬 URL을 설정하려면 http.antMatcher를 통해 설정해야한다.
- 모든 request에 대해 동작하려면 다음과 같이 http.antMatcher("/\*\*")로 하며 /api로 시작하는 path에 대해서 적용을 하고 싶다면 http.antMatcher("/api/\*\*") 와 같이 해줘야한다.
- 여러 종류의 URL에 대해 여러 Filter를 만들고 싶다면 SecurityConfig 클래스를 여러개 만들어 줘야하며 - 여러개의 sequrityConfig은 순서가 중요하여 class위에 @Order annotation을 붙여줘야한다.

# SecurityConfig

```
package com.exam.todojpa.config;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.builders.WebSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.security.web.authentication.AuthenticationSuccessHandler;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    // 5.7.4 버전 이후부터는 설정이 완전히 바뀌었다. ( 일부분은 레퍼런스를 찾기 어려움)
```

```
@Autowired  
private UserDetailsService customUserDetailsService;  
  
@Bean // 암호화, 암호 일치 확인을 위해 필요. 복호화는 안되는 알고리즘이다.  
public PasswordEncoder passwordEncoder(){  
    return new BCryptPasswordEncoder(); // BCryptPasswordEncoder 생성  
}  
  
@Autowired  
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
    //어떤 UserDetailsService를 사용하고, 어떤 PasswordEncoder를 사용하는지  
    auth.userDetailsService(customUserDetailsService)  
        .passwordEncoder(passwordEncoder());  
}  
  
// 정적 리소스 경로는 Spring Security 필터 채인이 동작하지 않도록 설정한다.  
@Override  
public void configure(WebSecurity web) throws Exception {  
    web.ignoring().antMatchers("/css/**", "/js/**", "/images/**",  
        "/lib/**", "/*/*.*.html", "/*/*.*.css", "/*/*.*.js");  
}
```

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/addUser").permitAll()  
        .antMatchers("/api/**").hasRole("USER")  
        .antMatchers("/todo.html").hasRole("USER")  
        // ignoring되어 있어서 권한이 없어도 접근된다. 이경운 html에서특정  
        // api를 호출한 후 403 오류가 발생하면redirect하도록 한다.  
        .antMatchers("/admin/api/**").hasRole("ADMIN")  
        .anyRequest().authenticated()  
        .and().formLogin().loginPage("/loginform.html")  
        .passwordParameter("password")  
        .usernameParameter("email")  
        .loginProcessingUrl("/login")  
        .failureUrl("/userregform.html")  
        .successHandler(new CustomneAuthenticationSuccessHandler())  
        .permitAll()  
        .and().logout().logoutUrl("/logout").logoutSuccessUrl("/").permitAll()  
        .and().exceptionHandling()  
        .authenticationEntryPoint(new AlwaysSendUnauthorized401AuthenticationEntryPoint())  
        .and().csrf().disable();  
}
```

- 내부클래스

```
// api의 경우 인증 오류가 발생할 경우 redirect되는 것이 아니라 SC_UNAUTHORIZED
// http 상태 코드를 반환하도록 한다. 401
class AlwaysSendUnauthorized401AuthenticationEntryPoint implements AuthenticationEntryPoint {
    @Override
    public final void commence(HttpServletRequest request, HttpServletResponse response,
        AuthenticationException authException) throws IOException {
        System.out.println("error");
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED);
    }
}

class CustomneAuthenticationSuccessHandler implements AuthenticationSuccessHandler {

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,
        Authentication authentication) throws IOException, ServletException {
        response.sendRedirect("/todojpa/");
    }
}
```

## WebAppInitializer 를 수정한다.

```
public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        // TODO Auto-generated method stub
        return new Class<?>[] {ApplicationConfig.class, SecurityConfig.class};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        // TODO Auto-generated method stub
        return new Class<?>[] {MvcConfig.class};
    }
}
```

# loginform.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>loginform</title>
</head>
<body>

<h1> 로그인 폼 <h1>

<hr>

    <form method="post" action="login">
        email : <input type="text" name="email"><br>
        암호 : <input type="password" name="password"><br>
        <input type="submit"/>
    </form>
</body>
</html>
```

# HomeController

```
package com.exam.todojpa.controller;

import java.security.Principal;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.exam.todojpa.domain.User;
import com.exam.todojpa.service.UserService;

@Controller
public class HomeController {
```

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

@Autowired
UserService userService;

@RequestMapping(value = "/", method = RequestMethod.GET)
public String home(Principal principal, Model model) {
    logger.info("principal : {}", principal);

    if(principal != null) {
        User user = userService.getUser(principal.getName());

        logger.info("user : {}", user);
    }
    return "home";
}
```

# home.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<%@ page session="false" %>
<html>
<head>
    <title>Home</title>
</head>
<body>
<sec:authorize access="!isAuthenticated()">
    <a href="loginform.html">login</a>
</sec:authorize>
<sec:authorize access="isAuthenticated()">
    Welcome Back, <sec:authentication property="name"/> <a href="logout">logout</a>
    <br>
    <hr>
    <a href="todo.html">todo</a>
</sec:authorize>

</body>
</html>
```

# userregform.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>회원가입 폼</title>
</head>
<body>
<h1> 회원 가입 폼 <h1>

<hr>

    <form method="post" action="addUser">
        email : <input type="text" name="email"><br>
        암호 : <input type="password" name="password"><br>
        <input type="submit"/>
    </form>
</body>
</html>
```

# UserController

```
package com.exam.todojpa.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;

import com.exam.todojpa.service.UserService;

@Controller
public class UserController {

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

```
@Autowired  
PasswordEncoder passwordEncoder;  
  
@Autowired  
UserService userService;  
  
@PostMapping(value = "/addUser")  
public String addUser(@RequestParam("email") String email,  
                      @RequestParam("password") String password) {  
  
    String encodePassword = passwordEncoder.encode(password);  
  
    userService.addUser(email, encodePassword);  
  
    return "redirect:/";  
}  
}
```

# UserService

- 회원 가입, 회원정보가 필요할때 사용

```
package com.exam.todojpa.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.exam.todojpa.domain.Role;
import com.exam.todojpa.domain.User;
import com.exam.todojpa.repository.RoleRepository;
import com.exam.todojpa.repository.UserRepository;

@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private RoleRepository roleRepository;
```

```
@Transactional
public User addUser(String email, String encodePassword) {
    User user = new User();
    user.setEmail(email);
    user.setPassword(encodePassword);

    Role role = roleRepository.findByRolename("ROLE_USER").orElseThrow();
    user.setRoles(List.of(role));

    return userRepository.save(user);
}

@Transactional(readOnly = true)
public User getUser(String email) {
    return userRepository.findByEmail(email).get();
}
```

# todo.js

- todo.html에서 javascript를 이용해 처음에 목록을 가지고 오는데 status가 200이 아닐때 (여기선 401)이 발생했을 때 '/todojpa/'로 리다이렉트 처리시킨다.

```
function getTodos(){
    const xhr = new XMLHttpRequest();
    xhr.open("get","http://localhost:8080/todojpa/api/todos");
    xhr.onreadystatechange = function(){
        if(xhr.readyState === 4 ){
            if(xhr.status === 200){
                // json 문자열을 json 객체로 변환시킨다.
                var todos = JSON.parse(this.responseText);
                console.log(todos);
                for(let i = 0; i < todos.length; i++){
                    todoItemAdd(todos[i]);
                }
            }else{
                window.location.href = '/todojpa/';
            }
        }
    }
    xhr.send();
}
```



# Spring 5, JUnit 5를 이용한 테스트

# Spring Test란

Spring Test는 스프링 프레임워크에서 제공하는 통합 테스트 및 스프링 MVC 통합 테스트를 지원합니다. Java 코드에 대해서 스프링 컨테이너 생성, 트랜잭션, DB와 연동 관리를 지원하며 웹 관련 Mock테스트를 위한 객체를 지원합니다.

## 실행 환경

- Junit 5, Spring Test , Mockito

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.3.23</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
</dependency>
```

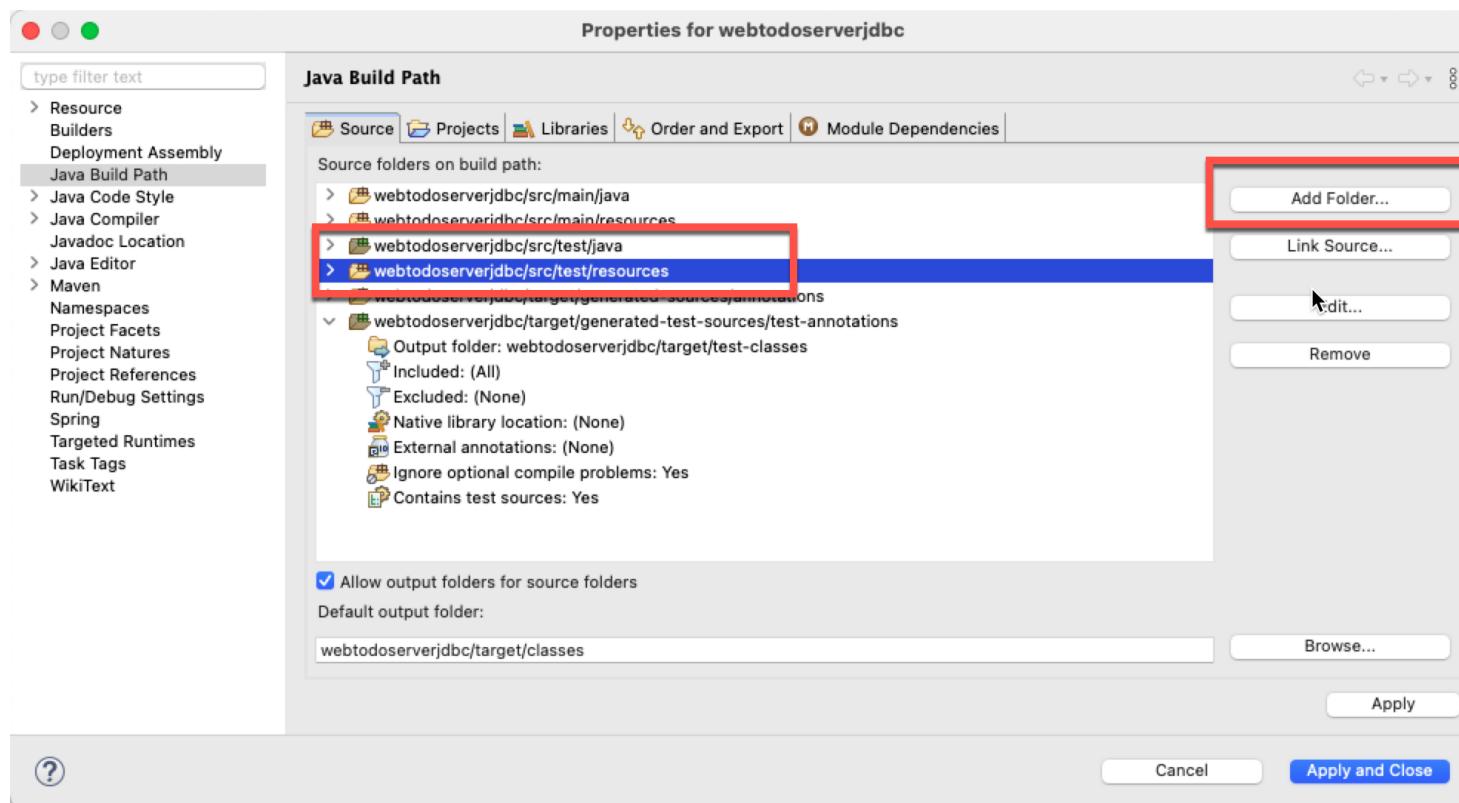
```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>4.8.1</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest</artifactId>
    <version>2.2</version>
    <scope>test</scope>
</dependency>
```

## 로그를 위한 라이브러리 추가

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.3</version>
</dependency>
```

- 다음과 같은 폴더를 작성하고 테스트 코드와 리소스 파일을 작성한다.
  - src/test/java
  - src/test/resources



Test시 테스트용 테이블, 데이터를 관리하기 위해 준비한다.



## test-data.sql

```
drop table todo if exists;

create TABLE todo (
    id bigint generated by default as identity,
    todo VARCHAR(255),
    done boolean,
    primary key(id)
);

insert into todo (id, todo, done) values(1, 'hello', true);
insert into todo (id, todo, done) values(2, 'hi', true);
insert into todo (id, todo, done) values(3, 'smile', true);
```

## clean-up.sql

```
drop table todo;
```

# 로그 설정 파일 작성

- test/resources 폴더에 logback.xml 파일로 작성

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

    <contextListener class="ch.qos.logback.classic.jul.LevelChangePropagator">
        <resetJUL>true</resetJUL>
    </contextListener>

    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{5} - %msg%n</pattern>
        </encoder>
    </appender>

    <logger name="examples.springmvc" level="info"/>
    <logger name="org.springframework" level="off"/>

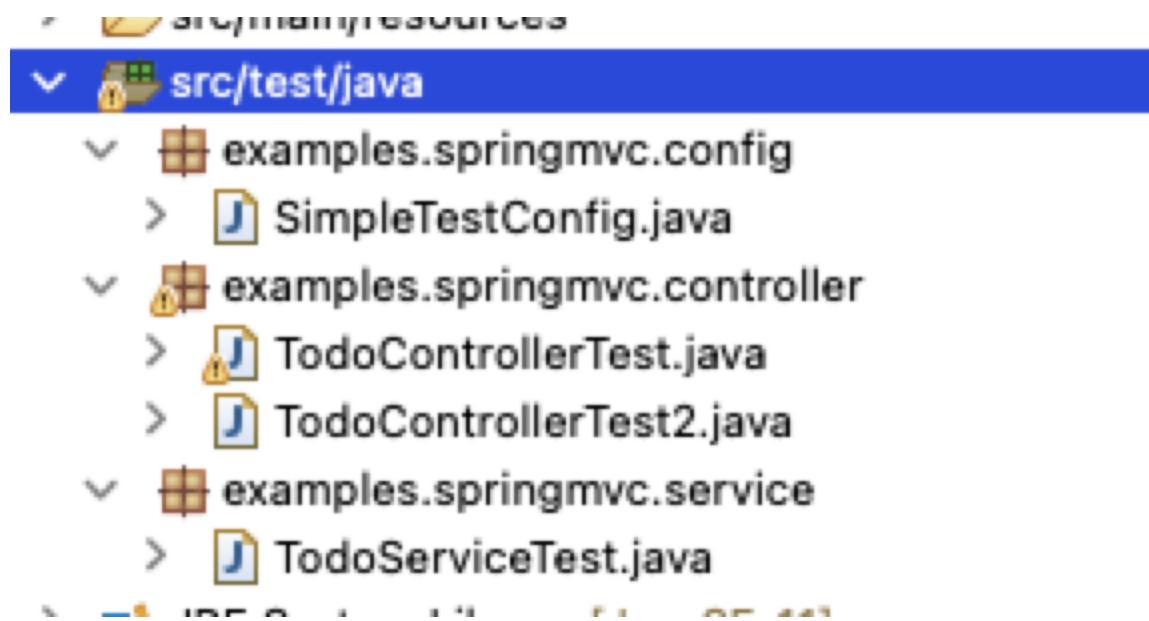
    <root level="info">
        <appender-ref ref="console" />
    </root>
</configuration>
```

## 로그 설정 파일 작성

- 로그는 debug, info, warn, error 순으로 심각하다. (로그레벨)
- 보통 아키텍트가 아키텍처를 정의할 때 어떤 경우에 어떤 로그레벨을 사용할지 결정한다.
- 위의 설정은 examples.springmvc로 시작하는 패키지의 info 이상의 로그를 남긴다는 의미다.
- info이상의 로그는 모두 console로 기록한다.
- console은 위의 appender로 정의 되었다. ConsoleAppender는 모니터에 출력을 하는 객체로 정해진 포맷형태로 출력을 한다.

# Test를 위한 DataSource를 Java Config에서 작성한다.

- SimpleTestConfig.java



# Test를 위한 DataSource를 Java Config에서 작성한다.

- SimpleTestConfig.java
  - 애플리케이션과 함께 실행되는 임베디드 H2를 실행하고 이를 이용해 DataSource를 생성 한다.

```
package examples.springmvc.config;

import javax.sql.DataSource;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
```

```
@Configuration
public class SimpleTestConfig {

    private static Logger logger = LoggerFactory.getLogger(SimpleTestConfig.class);

    @Bean(name = "dataSource")
    public DataSource dataSource() {
        try {
            EmbeddedDatabaseBuilder dbBuilder = new EmbeddedDatabaseBuilder();
            return dbBuilder.setType(EmbeddedDatabaseType.H2).build();
        } catch (Exception e) {
            logger.error("임베디드 DataSource 빈을 생성할 수 없습니다!", e);
            return null;
        }
    }
}
```

- ```
private static Logger logger = LoggerFactory.getLogger(SimpleTestConfig.class);
```

  - 로그 객체를 선언하였다. getLogger()메소드 안에는 로그를 남길 클래스 정보를 넣어준다.
- 내장 H2데이터베이스를 실행하고, 접속할 수 있는 DataSource를 반환한다.
  - 주의할점은 H2가 실행되고 있으면 안된다. port충돌남.

```
EmbeddedDatabaseBuilder dbBuilder = new EmbeddedDatabaseBuilder();
return dbBuilder.setType(EmbeddedDatabaseType.H2).build();
```

# JUnit 5 애노테이션

## @Test

- 본 어노테이션을 붙이면 Test 메서드로 인식하고 테스트 한다. JUnit5 기준으로 접근제한자가 Default 여도 된다. (JUnit4 까지는 public이어야 했었다.)

## @BeforeAll

- 본 어노테이션을 붙인 메서드는 해당 테스트 클래스를 초기화할 때 딱 한번 수행되는 메서드다. 메서드 시그니처는 static 으로 선언해야한다.

## @BeforeEach

- 본 어노테이션을 붙인 메서드는 테스트 메서드 실행 이전에 수행된다.

# JUnit 5 애노테이션

## @AfterAll

- 본 어노테이션을 붙인 메서드는 해당 테스트 클래스 내 테스트 메서드를 모두 실행시킨 후 딱 한 번 수행되는 메서드다. 메서드 시그니처는 static으로 선언해야한다.

## @AfterEach

- 본 어노테이션을 붙인 메서드는 테스트 메서드 실행 이후에 수행된다.

## @Disabled

- 본 어노테이션을 붙인 테스트 메서드는 무시된다.

## **org.springframework.test.context.jdbc.\***

```
import org.springframework.test.context.jdbc.Sql;  
import org.springframework.test.context.jdbc.SqlConfig;  
import org.springframework.test.context.jdbc.SqlGroup;
```

### **@Sql annotation 이란?**

- SQL 스크립트 혹은 쿼리를 실행시킨다. 주로 테스트 클래스, 메소드에 사용된다. 쉽게 말해 테스트 환경에서 데이터를 CRUD 할 수 있는 방법을 제공한다.

### **@SqlGroup 이란?**

- @Sql 을 여러개 그룹화 해서 사용할 수 있다.

# TodoServiceTest

```
package examples.springmvc.service;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

import java.util.List;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.jdbc.Sql;
import org.springframework.test.context.jdbc.SqlConfig;
import org.springframework.test.context.jdbc.SqlGroup;
import org.springframework.test.context.junit.SpringJUnitConfig;

import examples.springmvc.config.ApplicationConfig;
import examples.springmvc.config.SimpleTestConfig;
import examples.springmvc.domain.Todo;
```

```
// 임베디드 DB를 사용하기 위한 Config클래스와 Service, Repository, 트랜잭션 설정을 한 ApplicationConfig.class를 읽어들니다.
@SpringJUnitConfig(classes = {SimpleTestConfig.class, ApplicationConfig.class})
@DisplayName("Integration SingerService Test")
@ActiveProfiles("test")
public class TodoServiceTest {

    private static Logger logger = LoggerFactory.getLogger(TodoServiceTest.class);

    // test할 객체를 주입받는다.
    @Autowired
    TodoService todoService;

    // JUNIT5 라이프싸이클과 관련된 애노테이션
    @BeforeAll
    static void setUp() {
        logger.info("--> @BeforeAll - 이 클래스의 모든 테스트 메소드를 실행하기 전에 실행합니다.");
    }

    @AfterAll
    static void tearDown(){
        logger.info("--> @AfterAll - 이 클래스의 모든 테스트 메소드를 실행한 다음에 실행합니다.");
    }

    @BeforeEach
    void init() {
        logger.info("--> @BeforeEach - 이 클래스의 각 테스트 메소드 실행하기 전에 실행합니다.");
    }

    @AfterEach
    void dispose() {
        logger.info("--> @AfterEach - 이 클래스의 각 테스트 메소드 실행한 다음에 실행합니다.");
    }
}
```

```
@Test
public void test1() {
    logger.info("test1");
}

// 테스트가 실행될때 test-data.sql이 실행되고 테스트가 종료되면 clean-up.sql이 실행된다.
@Test
@DisplayName("todo 목록 구하기 테스트")
@SqlGroup({
    @Sql(value = "classpath:db/test-data.sql",
        config = @SqlConfig(encoding = "utf-8", separator = ";", commentPrefix = "--"),
        executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD),
    @Sql(value = "classpath:db/clean-up.sql",
        config = @SqlConfig(encoding = "utf-8", separator = ";", commentPrefix = "--"),
        executionPhase = Sql.ExecutionPhase.AFTER_TEST_METHOD),
})
public void findAll() {
    logger.info("findAll");
    List<Todo> result = todoService.getTotos();
    assertNotNull(result); // result가 Null이 아니면 성공
    assertEquals(3, result.size()); // result결과가 3건이면 성공
}
```

```
@Test
@DisplayName("id가 1인 Todo 조회하기")
@SqlGroup({
    @Sql(value = "classpath:db/test-data.sql",
        config = @SqlConfig(encoding = "utf-8", separator = ";", commentPrefix = "--"),
        executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD),
    @Sql(value = "classpath:db/clean-up.sql",
        config = @SqlConfig(encoding = "utf-8", separator = ";", commentPrefix = "--"),
        executionPhase = Sql.ExecutionPhase.AFTER_TEST_METHOD),
})
public void testFindByFirstNameAndLastNameOne() throws Exception {
    logger.info("testFindByFirstNameAndLastNameOne");
    Todo todo = todoService.getToto(1L);
    assertNotNull(todo);
}
}
```

## 실행 결과

```
22:38:26.711 [main] INFO e.s.s.TodoServiceTest - --> @BeforeAll - 이 클래스의 모든 테스트 메소드를 실행하기 전에 실행합니다.  
TodoDao Dao  
22:38:26.994 [main] INFO e.s.s.TodoServiceTest - --> @BeforeEach - 이 클래스의 각 테스트 메소드 실행하기 전에 실행합니다.  
22:38:26.995 [main] INFO e.s.s.TodoServiceTest - findAll  
22:38:27.030 [main] INFO e.s.s.TodoServiceTest - --> @AfterEach - 이 클래스의 각 테스트 메소드 실행한 다음에 실행합니다.  
22:38:27.044 [main] INFO e.s.s.TodoServiceTest - --> @BeforeEach - 이 클래스의 각 테스트 메소드 실행하기 전에 실행합니다.  
22:38:27.044 [main] INFO e.s.s.TodoServiceTest - test1  
22:38:27.044 [main] INFO e.s.s.TodoServiceTest - --> @AfterEach - 이 클래스의 각 테스트 메소드 실행한 다음에 실행합니다.  
22:38:27.068 [main] INFO e.s.s.TodoServiceTest - --> @BeforeEach - 이 클래스의 각 테스트 메소드 실행하기 전에 실행합니다.  
22:38:27.068 [main] INFO e.s.s.TodoServiceTest - testFindByFirstNameAndLastNameOne  
22:38:27.077 [main] INFO e.s.s.TodoServiceTest - --> @AfterEach - 이 클래스의 각 테스트 메소드 실행한 다음에 실행합니다.  
22:38:27.090 [main] INFO e.s.s.TodoServiceTest - --> @AfterAll - 이 클래스의 모든 테스트 메소드를 실행한 다음에 실행합니다.
```

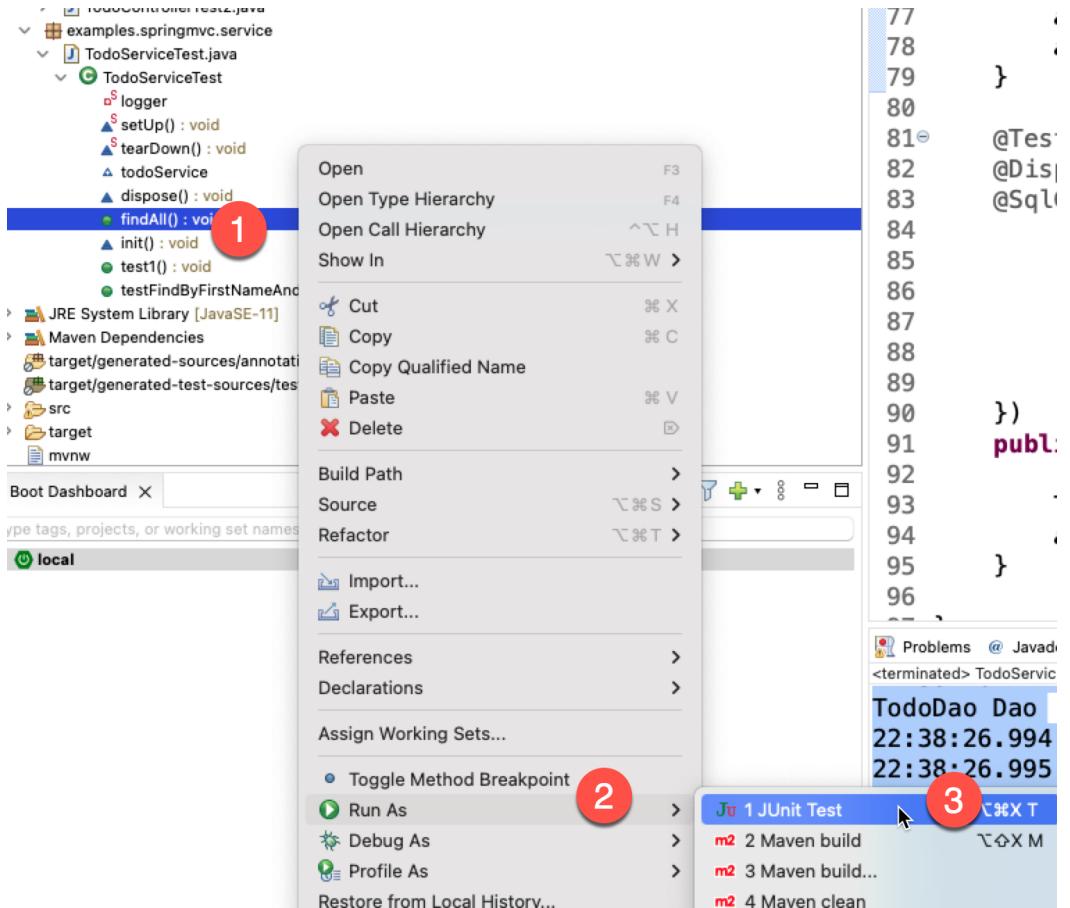
## 실행결과

Runs: 3/3      ✘ Errors: 0      ✘ Failures: 0

Integration SingerService Test [Runner: JUnit 5] (0.166 s)

- todo 목록 구하기 테스트 (0.120 s)
- test1() (0.001 s)
- id가 1인 Todo 조회하기 (0.044 s)

# 특정 메소드만 테스트하기



The screenshot shows the Eclipse IDE interface with the code editor displaying Java code. The code includes annotations like `@Test`, `@Disabled`, and `@Sql`. Below the code editor is the `Problems` view, which shows two errors related to the `TodoService` class. At the bottom of the screen, the `Run Configuration` bar is visible, showing the following items:

- JUnit Test (selected)
- Maven build (disabled)
- Maven build...
- Maven clean (disabled)

# Controller 테스트하기

## MockMvc

- MockMvc는 웹 어플리케이션을 서버에 배포하지 않고도 스프링 MVC의 동작을 재현할 수 있는 클래스이다. 테스트는 MockMvc를 이용해 테스트용으로 확장된 DispatcherServlet으로 요청을 보낸다. 그리고 DispatcheServlet은 요청을 받아 맵핑 정보를 보고 이에 맞는 핸들러 메소드를 호출한다. 최종적으로 테스트 메소드는 MockMvc가 반환하는 실행 결과를 받아 실행 결과가 맞는지 검증한다.
- MockMvcBuilders.standaloneSetup()
  - MockMvc 인스턴스를 생성해주는 메소드이다. standaloneSetup은 테스트에서 사용할 컨트롤러 하나만 지정해서 생성한다.

- `.perform()`
  - 괄호 내에 있는 문자열을 URL로 요청을 보내는 메소드이다.
- `.andExpect()`
  - 괄호 내에 있는 메소드들을 이용해 어떤 결과를 예측하는지 작성하는 메소드이다. `Status()`, `content()` 등 내부에서 사용할 수 있는 다양한 메소드가 있다.

## Mockito

- Controller를 테스트 할때 Controller가 사용하는 Service는 잘 돌아간다고 가정하고 테스트하고 싶은 경우가 있다. 오픈 소스인 Mockito는 내가 원하는 값을 반환하는 가짜 객체를 생성할 수 있도록 해준다.

## Mockito

- 이병헌의 유명한 대사. 모히또가서 몰디브 한잔 할때의 그....
- @Mock
  - mock 객체를 생성한다. 말그대로 가짜 객체가 바이트 코드 조작으로 만들어진다. 이 객체에 녹음기 녹음하듯이 원하는 형태로 동작하도록 할 수 있다.
- @InjectMocks
  - @InjectMocks라는 어노테이션이 존재하는데, @Mock이 붙은 목객체를 @InjectMocks 이 붙은 객체에 주입시킬 수 있다. 실무에서는 @InjectMocks(Service) @Mock(DAO) 이런식으로 Service테스트 목객체에 DAO 목객체를 주입시켜 사용한다.

# Controller 테스트하기

```
package examples.springmvc.controller;

// static 메소드를 static import 하고 있다.
import static org.hamcrest.CoreMatchers.containsString;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import java.util.ArrayList;
import java.util.List;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import com.fasterxml.jackson.databind.ObjectMapper;

import examples.springmvc.config.ApplicationConfig;
import examples.springmvc.config.MvcConfig;
import examples.springmvc.domain.Todo;
import examples.springmvc.service.TodoService;
```

```
// MvcMock을 사용하려면 아래와 같은 애노테이션이 설정되어 한다.  
// 그렇지 않으면 웹 환경이 구성 안된다.  
@ExtendWith(SpringExtension.class)  
@WebAppConfiguration  
@ContextConfiguration(classes = {MvcConfig.class, ApplicationConfig.class})  
public class TodoControllerTest2 {  
  
    // TodoService가 반환할 값  
    private final List<Todo> todos = new ArrayList<>();  
  
    // 가짜 객체가 주입될 Controller를 선언한다.  
    @InjectMocks  
    private TodoController todoController;  
  
    // 가짜 객체  
    @Mock  
    private TodoService todoService;  
  
    private MockMvc mockMvc;
```

```
// 테스트 코드가 실행되기 전에 Mockito환경을 초기화하고
// MockMvc를 초기화 한다.
@BeforeEach
public void initTodos() {
    Todo todo = new Todo();
    todo.setId(1l);
    todo.setTodo("haha");
    todo.setDone(false);
    todos.add(todo);

    // 해당 코드를 호출하지 않으면 @Mock, @InjectMock이 동작하지 않는다.
    MockitoAnnotations.openMocks(this);
    // todoController를 테스트 하기 위한 mockMvc를 선언한다.
    this.mockMvc = MockMvcBuilders.standaloneSetup(todoController).build();
}
```

```
@Test
public void testList() throws Exception {
    ObjectMapper objectMapper = new ObjectMapper();

    // todoService목객체의 getTodos()메소드를 호출하면
    // todos가 리턴되도록 설정한다.
    when(todoService.getTodos()).thenReturn(todos);

    // mockMvc에게 GET방식으로 api/todos를 호출한다.
    // 200 OK가 나오고, todos를 JSON으로 변환한 값과 api의 응답값이 같을 경우 성공하도록 한다.
    // get, status, content 등은 static import되어 있다. 초보자들은 이 부분이 힘들수 있다.
    this.mockMvc.perform(get("/api/todos"))
        .andExpect(status().isOk())
        .andExpect(content().string(containsString(objectMapper.writeValueAsString(todos))))
        .andDo(print());

    // Controller에서 목 객체인 todoService의 getTodos()를 한번 호출했는지 확인한다.
    // 1번 호출되지 않았다면 Controller는 이상하게(?)동작했다는 의미이다.
    verify(todoService, times(1)).getTodos();
}

}
```

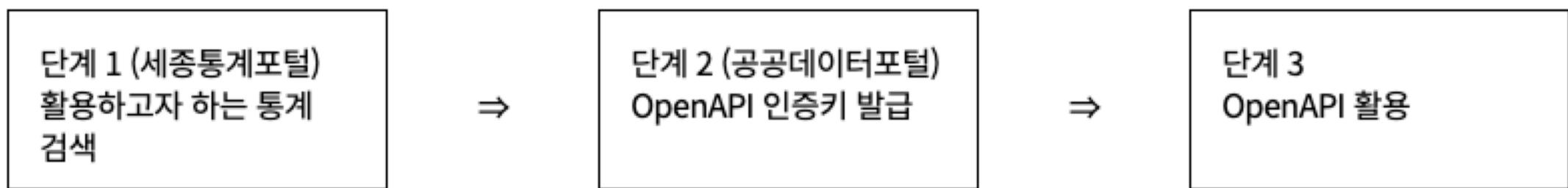
# Open API 활용

## API 인증방식

- API Key방식
- API Token방식
- 3자 인증 방식
- .....
- <https://bcho.tistory.com/955>

# Open API 사용 방법

예) 세종 특별 자치시 세종 통계 Open API



## Open API URL 호출 예시

http://apis.data.go.kr/5690000/sejongStatDataView/getList?serviceKey=[ 인증키 ]&tblId=DT\_20802N\_006&pageNo=1&numOfRows=10

Open API URL

인증키

테이블명

기본인자

## Open API를 만들기 위해서는 다음을 필요

- 인증 키 발급 시스템 필요
  - 기간, 어떤 API를 사용, 사용하는 사용자 정보 ....
- 발급받은 인증키를 Header, Parameter 등으로 전달하여 API 실행
  - API는 내부적으로 인증키가 맞는지 확인하여 맞을 경우 정보를 제공한다. 틀리면 403 값을 HTTP 상태코드로 반환한다.

## HTTP 상태 401(Unauthorized) 이란?

- HTTP 상태 중 401(Unauthorized)는 클라이언트가 인증되지 않았거나, 유효한 인증 정보가 부족하여 요청이 거부되었음을 의미하는 상태값이다. 즉, 클라이언트가 인증되지 않았기 때문에 요청을 정상적으로 처리할 수 없다고 알려주는 것이다.
- 401(Unauthorized) 응답을 받는 대표적인 경우는 로그인이 되어 있지 않은 상태에서 무언가 요청을 하는 경우이다. 예를 들어 어떤 쇼핑몰 사이트에 로그인을 하지 않았는데 나의 결제 내역과 같은 정보를 달라고 하면 401(Unauthorized)를 반환받게 될 것이다.  
이와 많이 혼동되는 HTTP 상태로 403(FORBIDDEN)이 있다.

## HTTP 상태 403(FORBIDDEN) 이란?

- HTTP 상태 중 403(FORBIDDEN)는 서버가 해당 요청을 이해했지만, 권한이 없어 요청이 거부되었음을 의미하는 상태값이다. 즉, 클라이언트가 해당 요청에 대한 권한이 없다고 알려주는 것이다.

403(FORBIDDEN) 응답을 받는 대표적인 경우는 로그인하여 인증되었지만 접근 권한이 없는 무언가를 요청하는 경우이다. 예를 들어 어떤 쇼핑몰에 접속하여 로그인까지 하였지만, 다른 사용자의 결제 내역을 달라고 하면 403(FORBIDDEN)을 반환받게 될 것이다.

## 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>

## 참고문서

- 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

## 참고 문서

참고 -

<http://projects.spring.io/spring-data/>

[http://www.querydsl.com/static/querydsl/4.0.1/reference/ko-KR/html\\_single/](http://www.querydsl.com/static/querydsl/4.0.1/reference/ko-KR/html_single/)

감사합니다.