

# Spring Framework

# 목차

1. Spring Framework란 ?
2. Spring 개발 환경 구축
3. Spring DI와 IOC
4. 메시지 및 이벤트 처리
5. Annotation 기반 설정
6. Spring AOP
7. Spring MVC 웹 프로그램
8. Spring의 Database 연동

# 1. Spring Framework란 ?

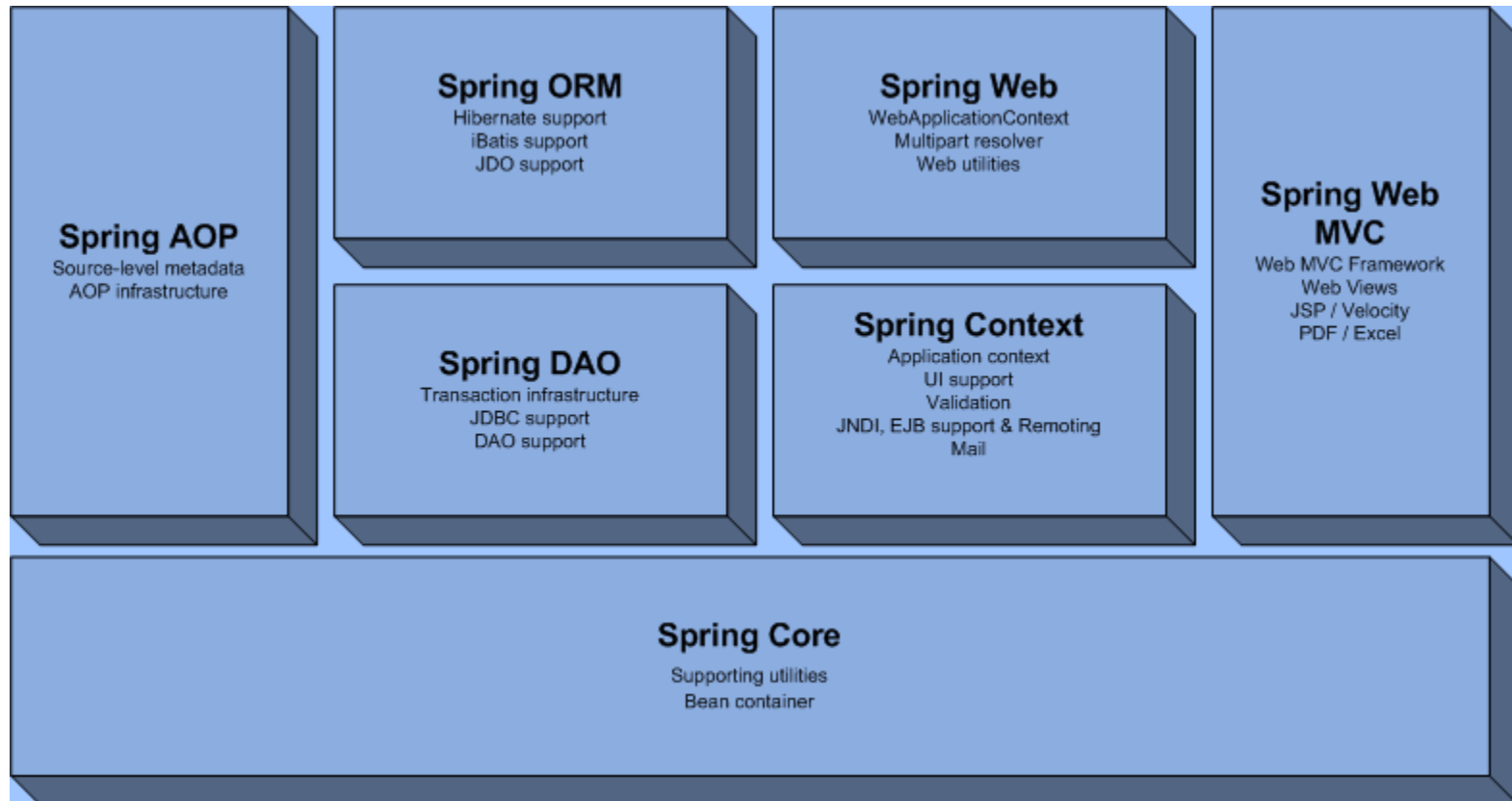
- ✓ Spring Framework의 개요
- ✓ Spring Framework의 모듈
- ✓ Spring Framework의 특징

# Spring Framework의 개요

- 1) JAVA플랫폼을 위한 Open Source 애플리케이션 프레임워크
- 2) 오픈 소스 경량급 엔터프라이즈 애플리케이션 프레임워크
- 3) POJO(Plain Old Java Object) 방식의 프레임워크.
- 4) 대한민국 전자정부 표준 프레임워크의 기반 기술

# Spring Framework의 모듈

7개의 모듈로 잘 조직된 기능과 특성을 포함한다.



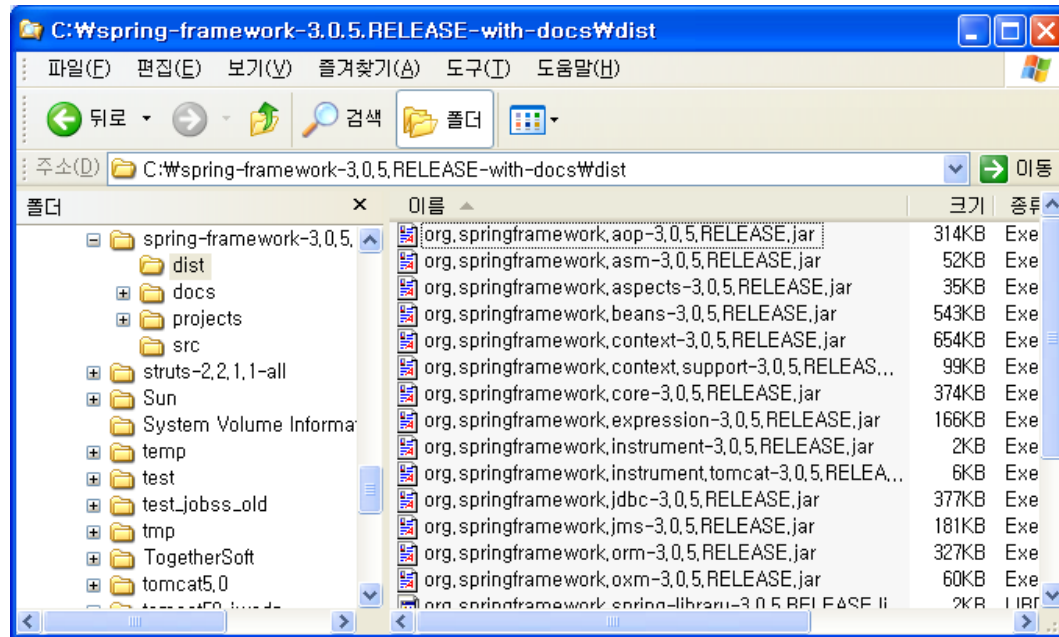
# Spring Framework의 모듈

모듈 명	설 명
Beans	BeanFactory 인터페이스를 통해 구현됨
Core	프레임워크의 가장 기본적인 부분 DI기능 제공
Context	국제화, 이벤트 처리, 리소스 로딩, 서블릿 컨테이너 를 위한 컨텍스트 생성 등의 기능제공
Expression Language	객체에 접근하고 객체를 조작하기 위한 표현 언어 제공
AOP	AOP Alliance에 호환되는 AOP 구현 제공
Aspects	AspectJ와의 통합 제공
Web(MVC/Remoting)	Spring MVC 제공
Data Access /Integration	JDBC를 위한 템플릿 제공. iBatis 및 하이버네이트 등의 ORM api를 위한 통합 레이어 제공. Spring이 제공하는 트랜잭션과의 연동 지원

# Spring Framework의 특징

## 1) 경량(Lightweight)

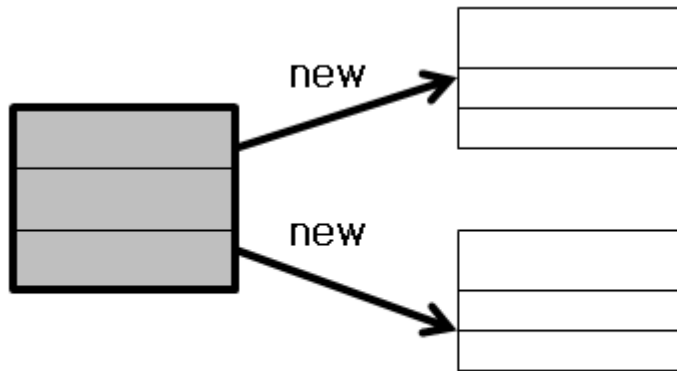
Spring은 그 크기와 부하의 측면에서 경량이며 몇 개의 JAR파일로 구성되므로 설치와 사용이 쉽다.



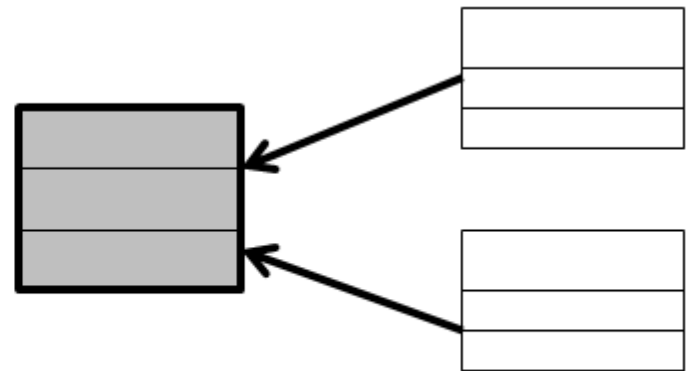
# Spring Framework의 특징

## 2) 제어 역행(IoC - Inversion of Control)

- ① 개발자가 직접 객체를 생성을 하지 않고, 객체의 생성에서 소멸까지를 설정 파일이나 애노테이션을 통해서 컨테이너가 관리
- ② Dependency Injection을 통해 의존성 주입



[ Not IoC ]

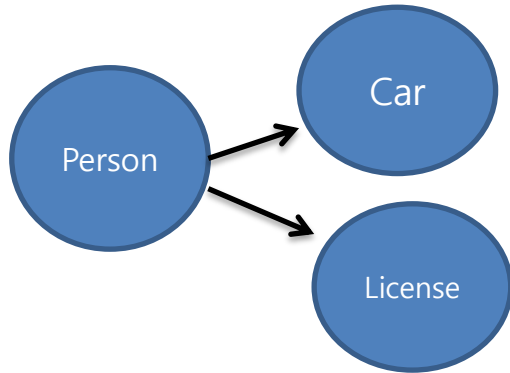


[ IoC ]



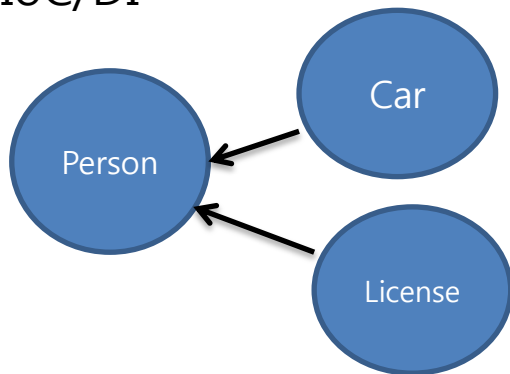
# Spring Framework의 특징

Non-IoC/DI



```
Class Person {  
    public void process(){  
        Car ca = new Car();  
        License lic = new License()  
    }  
}
```

IoC/DI



```
<bean name="person" class="Person">  
    <constructor-arg>  
        <bean class="Car"/>  
    </constructor-arg>  
    <constructor-arg>  
        <bean class="License"/>  
    </constructor-arg>  
</bean>
```

# Spring Framework의 특징

## 3) Aspect Oriented Programming 지원

로깅, 보안, 트랜잭션 등의 공통적인 기능의 활용을 기존의 비즈니스 로직에 영향을 주지 않고 모듈화 처리를 지원하는 프로그래밍 기법

핵심코드

보안처리(보조)

핵심코드

핵심코드

로그처리(보조)

트랜잭션(보조)

핵심코드

[보조 업무가 주 업무 코드에 포함]

- ① 동일한 작업 반복
- ② 보조 업무의 작업 코드변경시, 해당 보조 업무를 사용하는 모든 주 업무 코드의 소스 수정 필요.

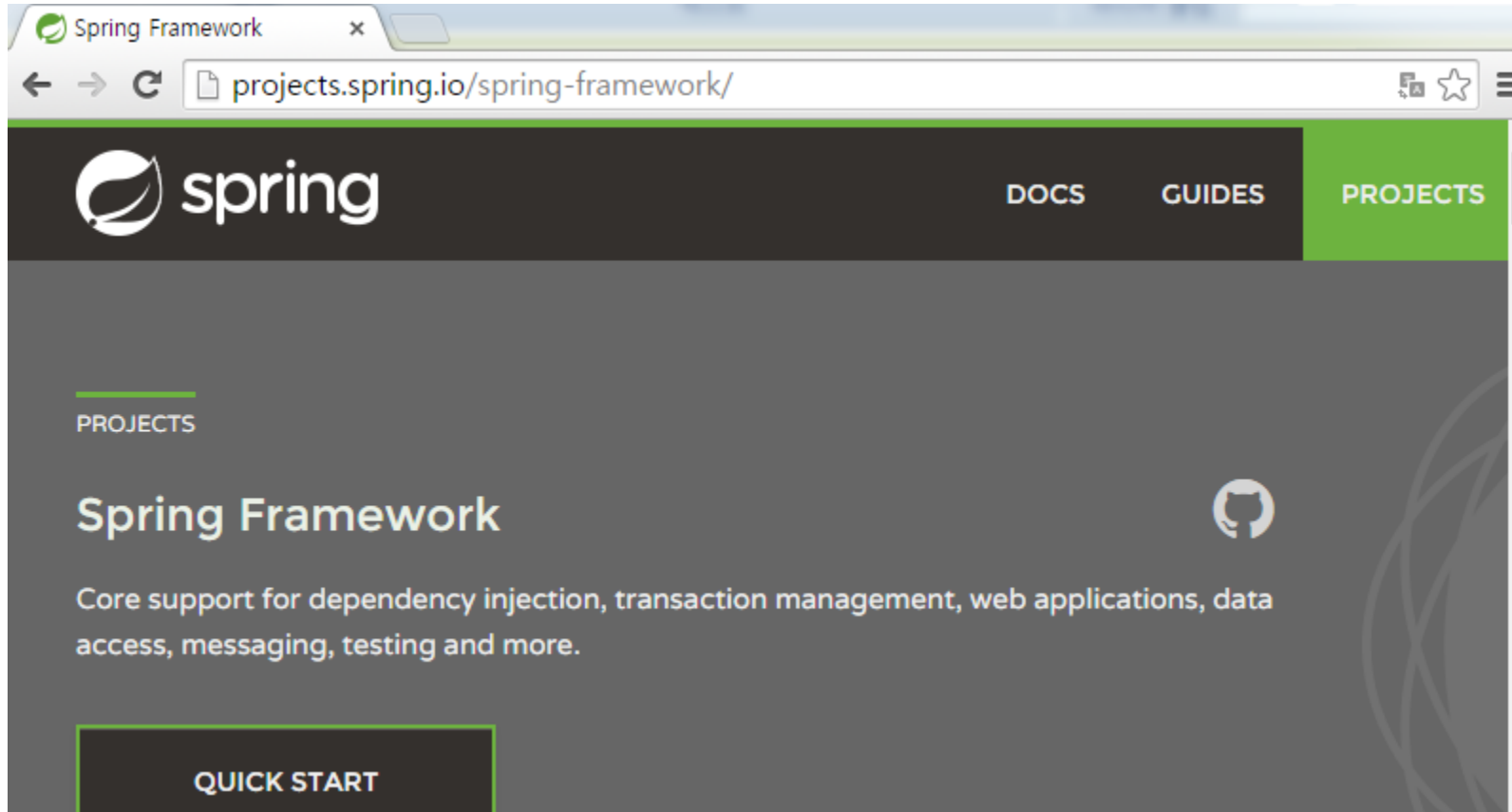
**AOP는 이러한 보조 업무 코드를 주 업무 코드에서 별도로 분리하여 작성하고, 필요할 때에만 도킹(Docking)**

## 2. Spring 개발 환경 구축

- ✓ Spring Framework 다운로드
- ✓ Spring IDE 설치
- ✓ Spring Project 생성

# Spring Framework의 다운로드

## SpringSource 사이트에서 다운

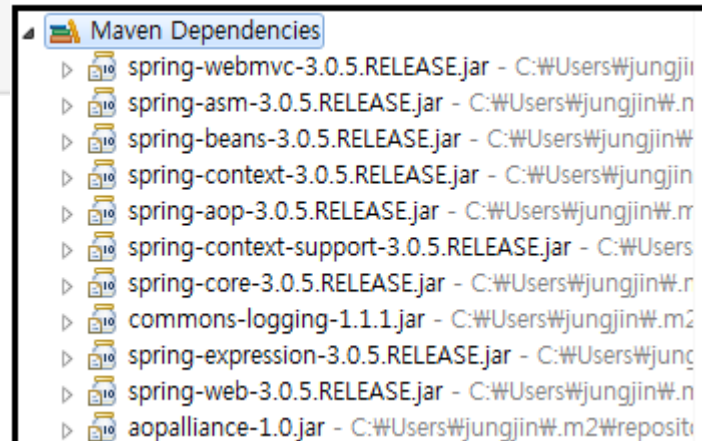


# Spring Framework의 설치

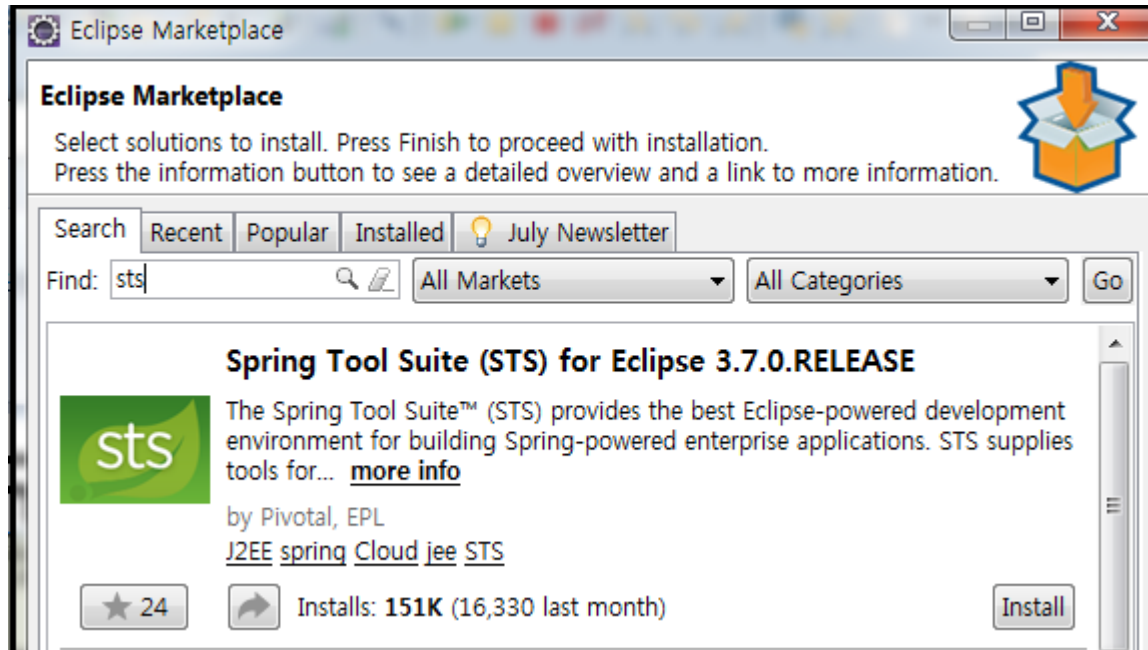
## Maven 이클립스 플러그인

Pom.xml에 중앙저장소에 있는 해당 라이브러리를 다운받고자하는 라이브러리의 dependency 를 추가하면 Maven library에 자동추가됨을 볼 수 있다. 종속라이브러리도 함께 다운된다.

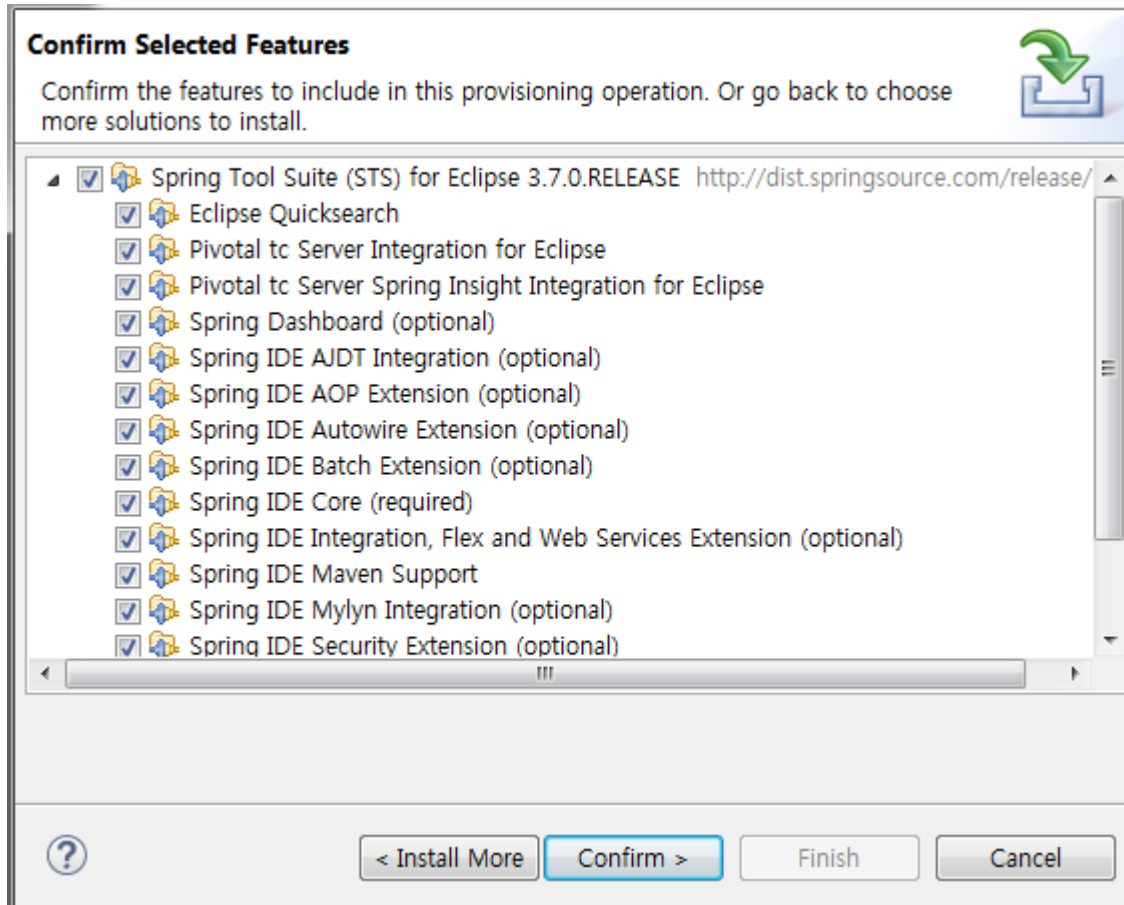
```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.0.0.RELEASE</version>
  </dependency>
</dependencies>
```



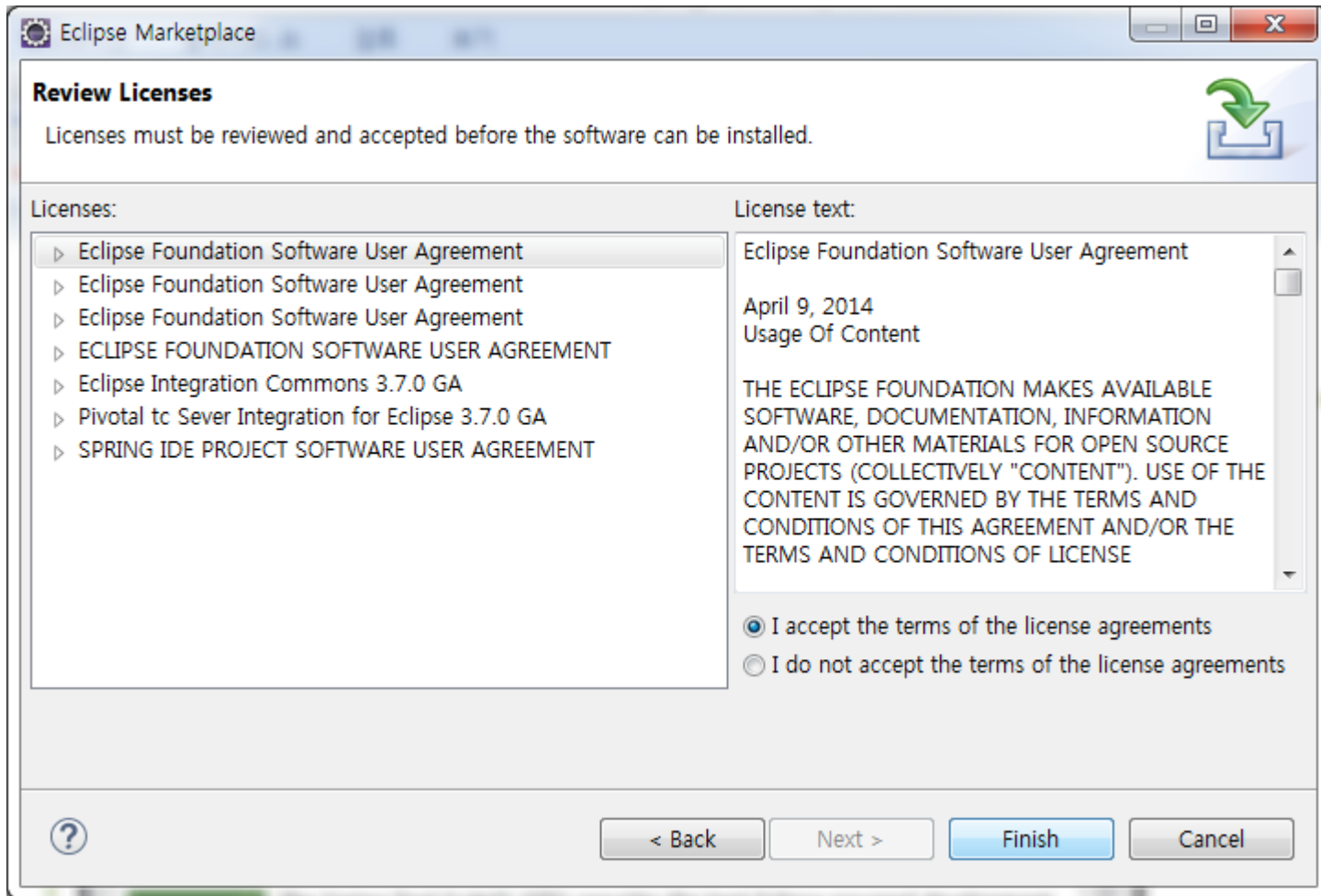
# Spring IDE 설치



# Spring IDE 설치

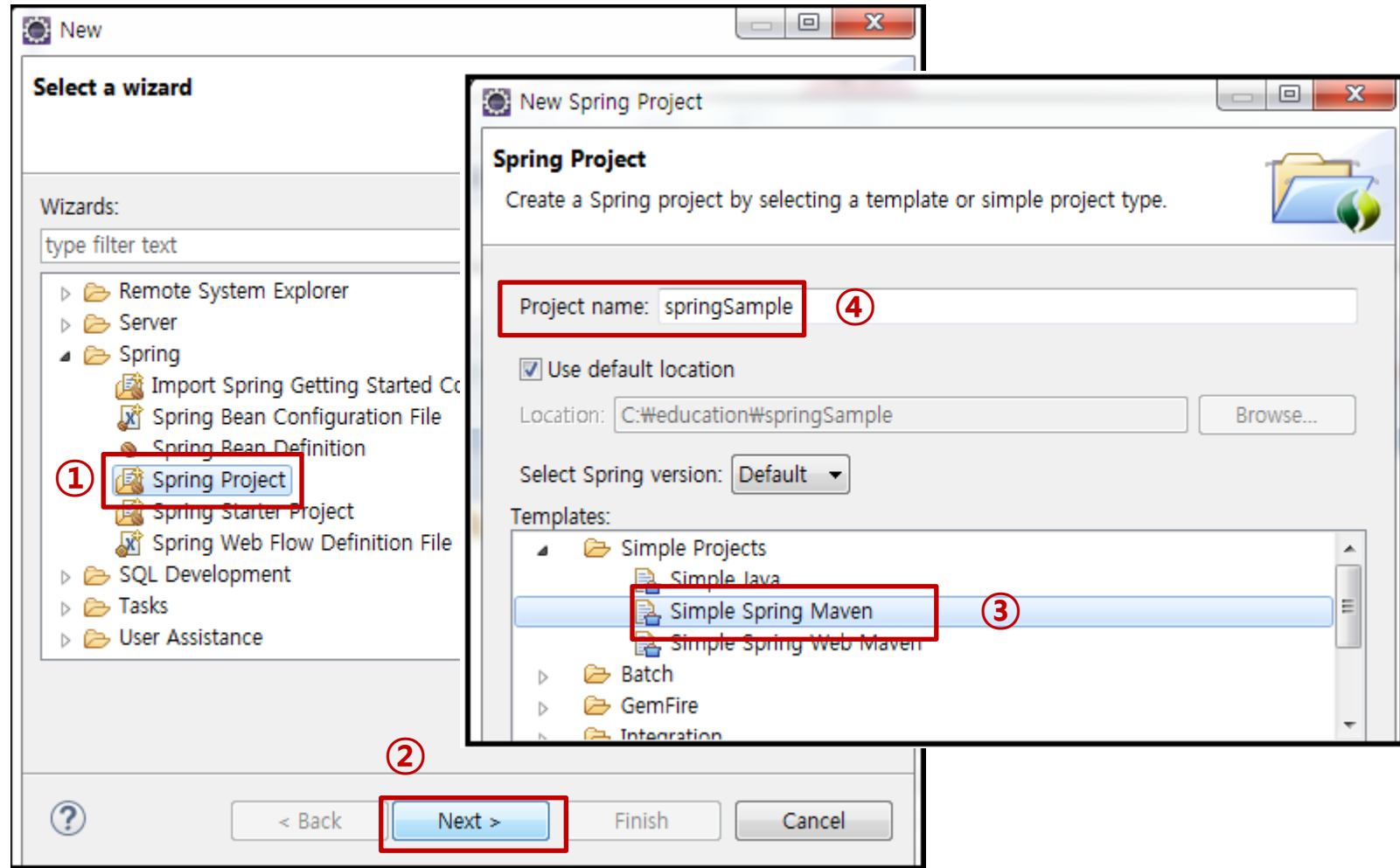


# Spring IDE 설치



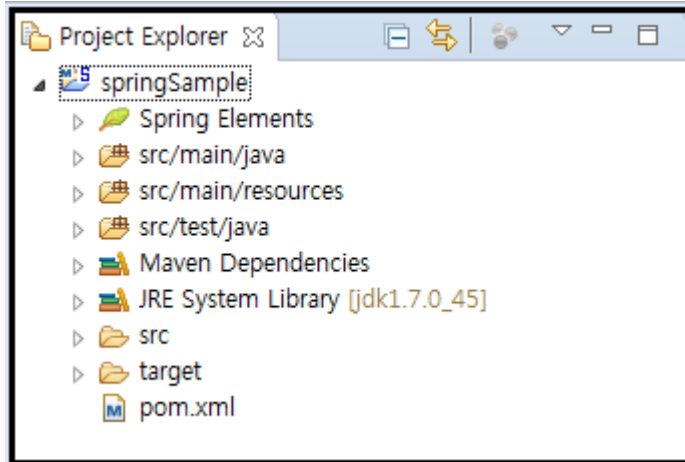


# Spring Project 생성

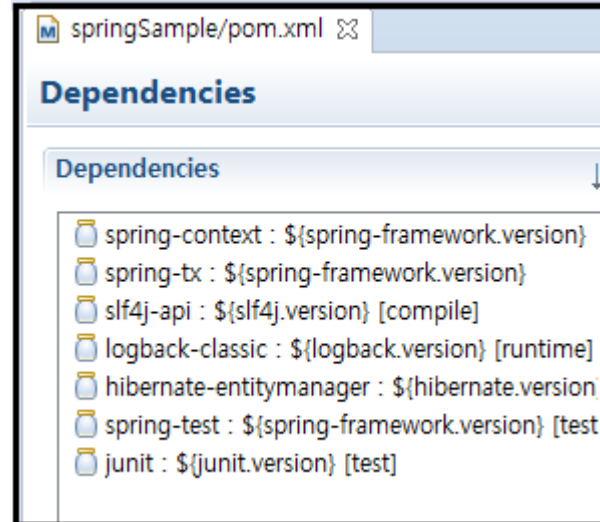


# Spring Project 생성

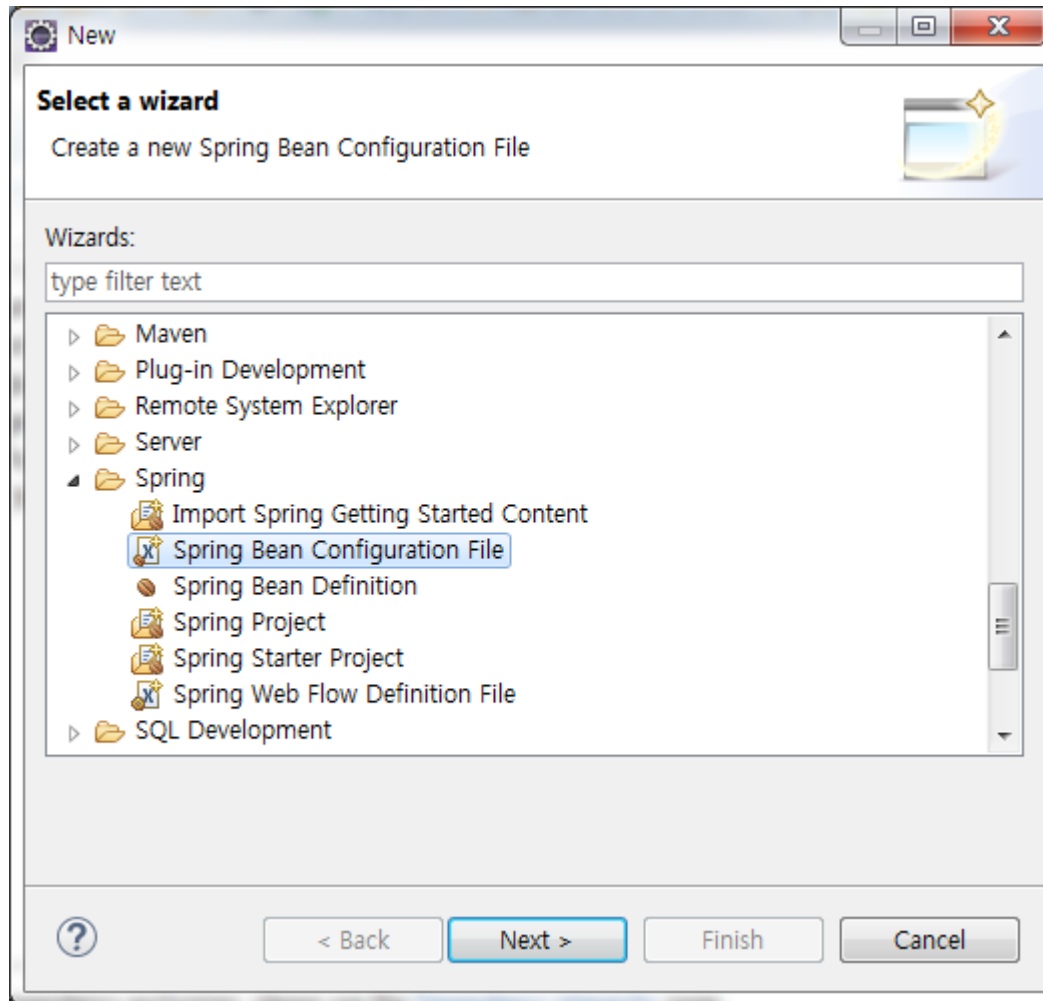
Maven 기반의 project로 완성된 구조



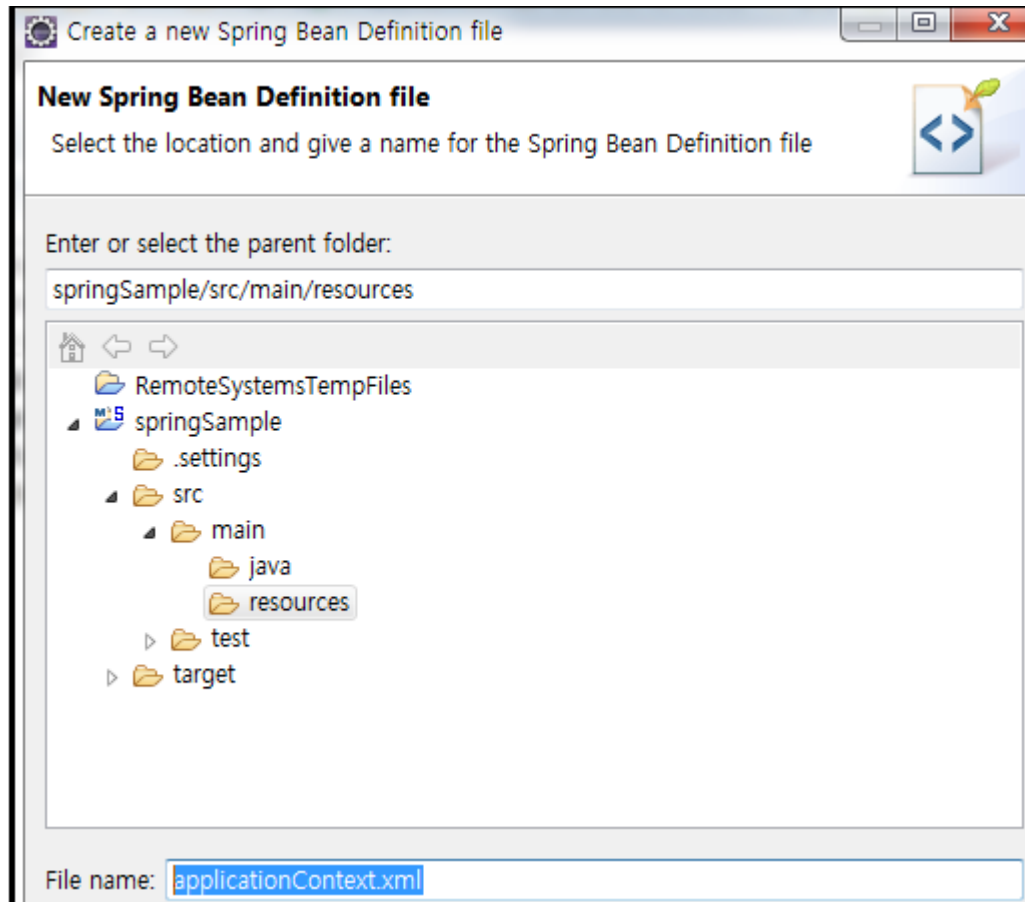
의존 관계의 library들이 자동 설정



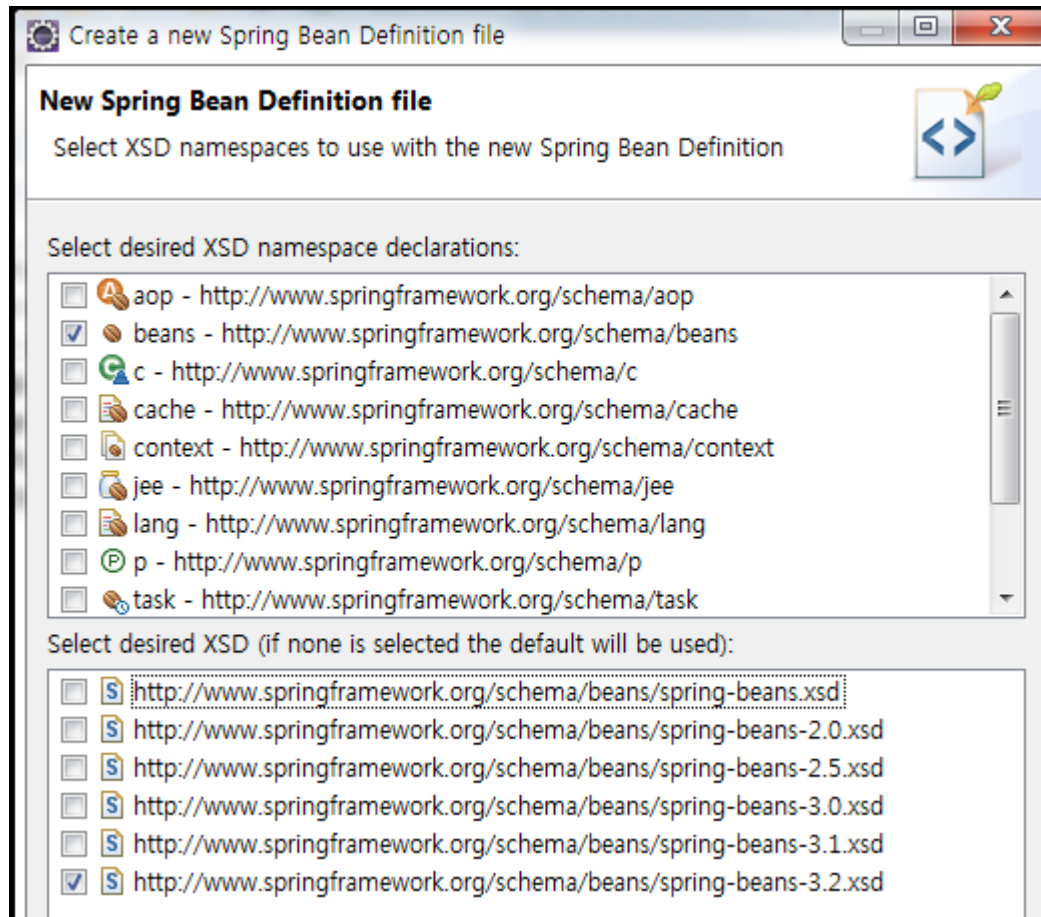
# Spring Project 생성



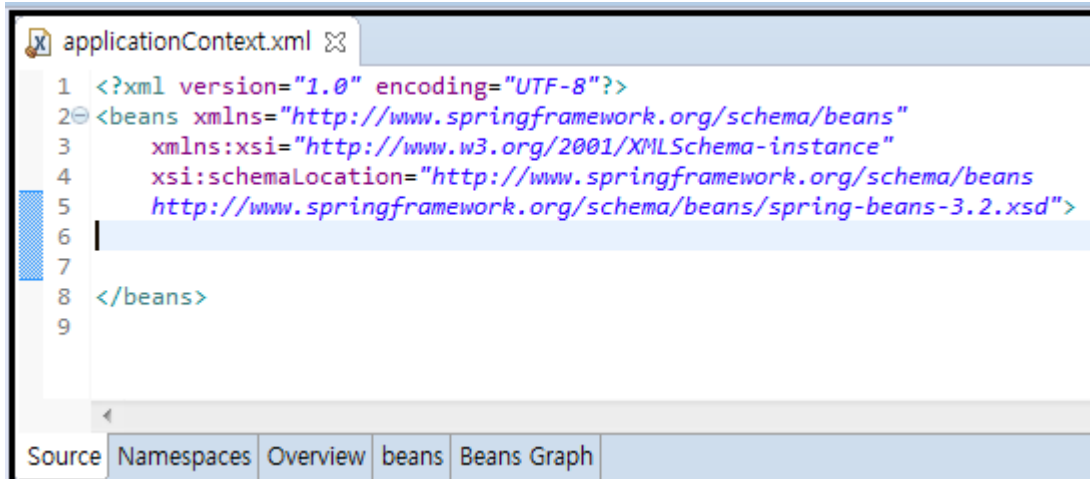
# Spring Project 생성



# Spring Project 생성



# Spring Project 생성



```
applicationContext.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://www.springframework.org/schema/beans
5     http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">
6
7
8 </beans>
9
```

Source Namespaces Overview beans Beans Graph

# Spring Project 생성

## [문제가 발생했다면]

Errors occurred during the build.

Errors running builder 'Maven Project Builder' on project 'springsample'.

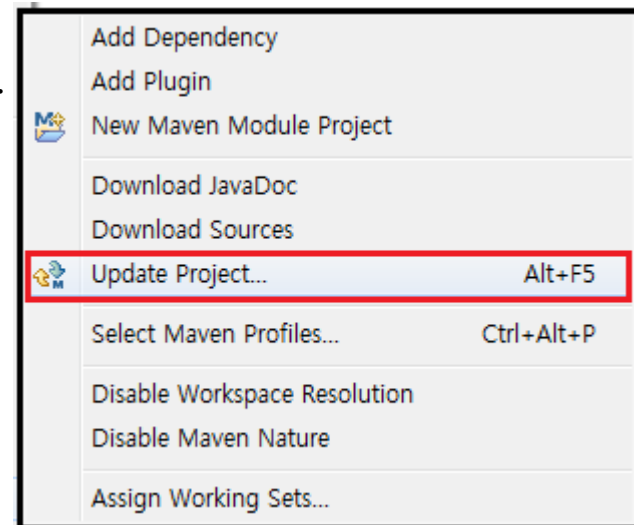
Could not calculate build plan: Plugin org.apache.maven.plugins:maven-resources-plugin:~~~~~

## [이렇게 해결]

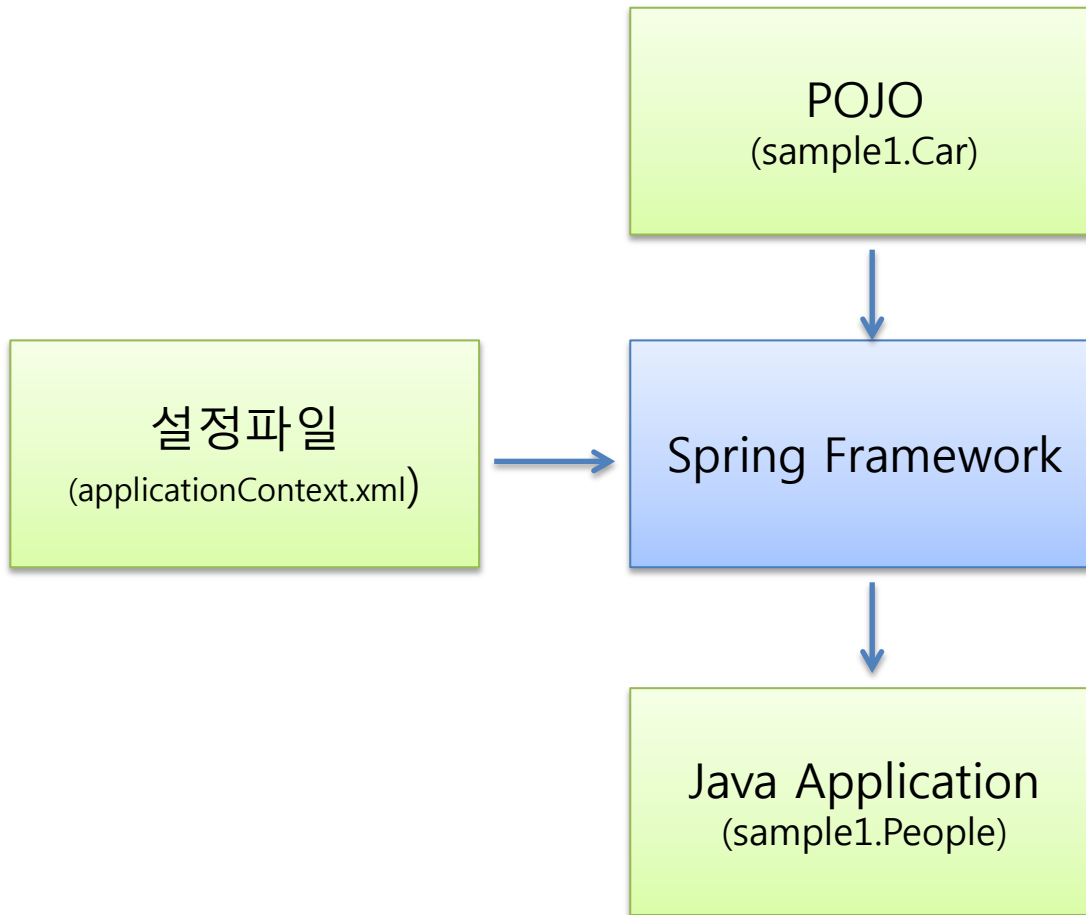
C:\Users\jeoungjin\m2\repository\org\apache\maven\plugins

레파지토리 하위의 메이븐 플러그인을 삭제.

프로젝트의 update를 통해 다시 레파지토리를 다운.



# Spring Project 생성





# Spring Project 생성

```
public class Car {  
    public void carInfo(String info){  
        System.out.println("이 차는 " + info + "입니다.");  
    }  
}
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
        http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">  
    <bean id="car" class="sample1.Car"></bean>  
</beans>
```

```
public class People {  
    public static void main(String[] args) {process();  
        public static void process(){  
            Resource resource = new ClassPathResource("applicationContext.xml");  
            BeanFactory factory = new XmlBeanFactory(resource);  
            Car car =(Car) factory.getBean("car");  
            car.carInfo("새 차");  
        }  
    }  
}
```

# 3. Spring DI와 IOC

- ✓ IOC와 DI 개요
- ✓ Spring 컨테이너
- ✓ Spring Injection방법
- ✓ 컬렉션 타입의 Injection 설정
- ✓ 의존관계 자동 설정
- ✓ 빈 객체의 범위

# IoC와 DI 개요

**Ioc(Inversion of Control)** : 역제어

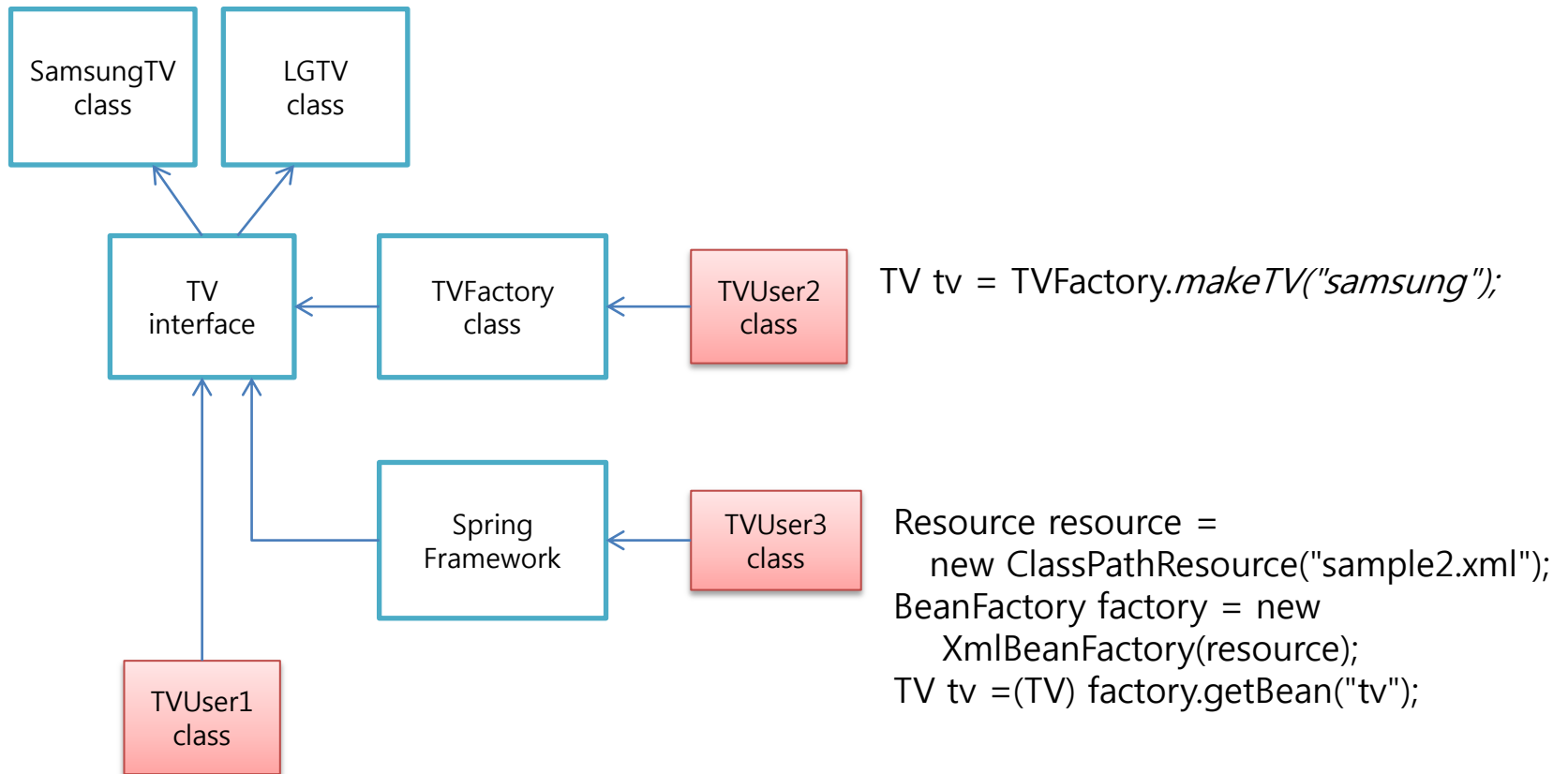
객체에 대한 제어권이 바뀐것을 의미한다.

인스턴스 생성부터 소멸까지의 생명주기를 개발자가 아닌 컨테이너가 대신 해준다.

**DI(Dependency Injection)** :

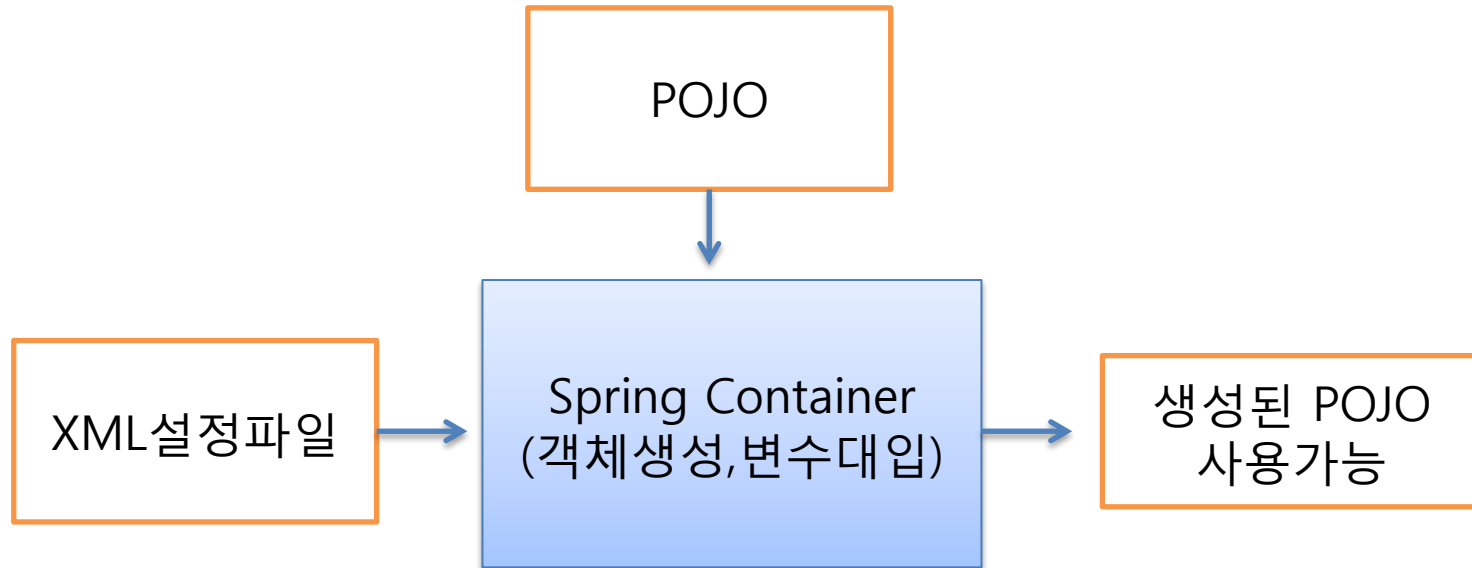
모듈간의 의존성을 Spring framework이 빈 설정 정보를 바탕으로 결정하고 런 타임시 자동으로 부여해준다.

# IoC와 DI 개요



TV tv = new SamsungTV();

# Spring Container



## 빈팩토리 또는 애플리케이션컨텍스트란?

Spring에서 빈의 생성과 관계 설정, 사용, 제거등의 기능을 담당하는 컨테이너이다.

# Spring Container

## 주요 API



### 1) BeanFactory

- 빈이 사용할 때 생성됨
- 메모리를 좀 더 효율적으로 사용할 수 있어서 메모리 사용이 적어야 하는 모바일 관련 개발에 사용됨.

- XML 파일로부터 설정 정보를 활용하는 가장 많이 사용되는 하위 클래스

`org.springframework.beans.factory.xml.XmlBeanFactory`

# Spring Container

## 2) ApplicationContext

- 사용되기 전에 빈을 미리 로딩
- 일반 어플리케이션 개발시 사용됨.
- 메시지의 국제화
- 리소스 관리 및 로딩
- 이벤트 처리 등의 유용한 부가 기능
- 주요 자식 API

ClassPathXmlApplicationContext : 클래스패스에 위치한 XML파일로부터 설정 정보 로딩

FileSystemXmlApplicationContext : 파일 시스템에 위치한 XML파일로부터 설정 정보 로딩

XmlWebApplicationContext : 웹 애플리케이션에 위치한 XML파일로부터 설정 정보 로딩

## 3) WebApplicationContext

- 웹 어플리케이션에서 사용됨.

# Spring Injection방법

## 1) Constructor Injection

의존하는 빈 객체를 클래스를 초기화 할 때 컨테이너로부터 생성자의 파라미터를 통해서 전달받는 방식

- ① Index이용 방법
- ② 타입지정 방법

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://www.springframework.org/schema/beans
5       http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">
6
7   <!-- 방법1 -->
8   <bean id="car" class="sample3.Car">
9       <constructor-arg index="0" value="BMW720"></constructor-arg>
10      <constructor-arg index="1" value="7000"></constructor-arg>
11   </bean>
12   <!-- 방법2 -->
13   <bean id="car" class="sample3.Car">
14       <constructor-arg name="model" value="BMW720"></constructor-arg>
15       <constructor-arg name="price" value="7000"></constructor-arg>
16   </bean>
17
18 </beans>
```



# Spring Injection방법

## 1) Constructor Injection

```
public class People {  
    String name;  
    String phone;  
    Car car;  
    public People(String name, String phone, Car car) {  
        super();  
        this.name = name;  
        this.phone = phone;  
        this.car = car;  
    }  
}
```

```
1 <?xml version="1.0" encoding="UTF-8"?>  
2 <beans xmlns="http://www.springframework.org/schema/beans"  
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
4     xsi:schemaLocation="http://www.springframework.org/schema/beans  
5         http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">  
6  
7 <!-- 예제1 -->  
8 <!-- <bean id="car" class="sample3.Car">  
9     <constructor-arg index="0" value="BMW720"></constructor-arg>  
10    <constructor-arg index="1" value="7000"></constructor-arg>  
11 </bean> -->  
12 <!-- 예제2 -->  
13 <bean id="car" class="sample3.Car">  
14     <constructor-arg name="model" value="BMW720"></constructor-arg>  
15     <constructor-arg name="price" value="7000"></constructor-arg>  
16 </bean>  
17  
18 <bean id="people" class="sample3.People">  
19     <constructor-arg name="name" value="홍길동"></constructor-arg>  
20     <constructor-arg name="phone" value="01055551234"></constructor-arg>  
21     <constructor-arg ref="car"></constructor-arg>  
22 </bean>  
23 </beans>
```

one=" + phone + ", car=" + car

```
public class PeopleApp {  
    public static void main(String[] args) {  
        process();  
    }  
    public static void process(){  
        ApplicationContext factory =  
            new ClassPathXmlApplicationContext("sample3.xml");  
        People p =(People) factory.getBean("people");  
        System.out.println(p);  
    }  
}
```

# Spring Injection방법

## 2) Setter Injection

setter 메소드를 이용하여 의존 관계를 설정한다.

```
public class Car {  
    String model;  
    int price;  
  
    public Car(){}  
    public void setModel(String model) {  
        this.model = model;  
    }  
  
    public void setPrice(int price) {  
        this.price = price;  
    }  
}
```

```
public class People {  
    String name;  
    String phone;  
    Car car;  
    public People(){};  
  
    public void setName(String name) {  
        this.name = name;  
    }  
    public void setPhone(String phone) {  
        this.phone = phone;  
    }  
    public void setCar(Car car) {  
        this.car = car;  
    }  
}
```

```
<bean id="car" class="sample3.Car">  
    <property name="model" value="그랜저"></property>  
    <property name="price" value="4000"></property>  
</bean>  
  
<bean id="people" class="sample3.People">  
    <property name="name" value="김민준"></property>  
    <property name="phone" value="5118"></property>  
    <property name="car" ref="car"></property>  
</bean>
```

# 컬렉션 타입의 Injection 설정

## 1) LIST 또는 배열

```
public class People {  
    String name;  
    String phone;  
    Car car;  
  
    List major;  
    public void setMajor(List major) {  
        this.major = major;  
    }  
}
```

```
<bean id="people" class="sample3.People">  
    <property name="name" value="홍길동"></property>  
    <property name="phone" value="5118"></property>  
    <property name="car" ref="car"></property>  
    <property name="major">  
        <list>  
            <value>컴퓨터공학</value>  
            <value>경영학</value>  
        </list>  
    </property>  
</bean>
```

```
public class License {  
    String title;  
    int year;  
    public String getTitle() {  
        return title;  
    }  
    public void setTitle(String title) {  
        this.title = title;  
    }  
    public int getYear() {  
        return year;  
    }  
    public void setYear(int year) {  
        this.year = year;  
    }  
}
```

```
<property name="licenses">  
    <list>  
        <ref bean="license1"/>  
        <ref bean="license2"/>  
    </list>  
</property>  
</bean>  
<bean id="license1" class="sample3.License">  
    <property name="title" value="정보관리기사"></property>  
    <property name="year" value="1992"></property>  
</bean>  
<bean id="license2" class="sample3.License">  
    <property name="title" value="ITIL"></property>  
    <property name="year" value="2000"></property>  
</bean>
```

# 컬렉션 타입의 Injection 설정

## 2) Map

```
<property name="book">
  <map>
    <entry>
      <key><value>자바의 정석</value></key>
      <ref bean="book1"/>
    </entry>
    <entry>
      <key><value>JSPServlet</value></key>
      <ref bean="book2"/>
    </entry>
  </map>
</property>
</bean>
```

```
<bean id="book1" class="sample3.Book">
  <constructor-arg name="title" value="자바의 정석"></constructor-arg>
  <constructor-arg name="price" value="25000"></constructor-arg>
  <constructor-arg name="kind" value="IT"></constructor-arg>
</bean>
<bean id="book2" class="sample3.Book">
  <constructor-arg name="title" value="JSPServlet"></constructor-arg>
  <constructor-arg name="price" value="35000"></constructor-arg>
  <constructor-arg name="kind" value="IT"></constructor-arg>
</bean>
```

```
public class Book {
    String title;
    String price;
    String kind;

    public Book(){}

    public Book(String title, String price, String kind) {
        super();
        this.title = title;
        this.price = price;
        this.kind = kind;
    }
}
```

```
public class People {
    String name;
    String phone;
    Car car;
    List major;
    List<License> licenses;
    Map<String, Book> book;

    public void setBook(Map book) {
        this.book = book;
    }
}
```

# 컬렉션 타입의 Injection 설정

## 3) Collection 타입 설정 – Set, Properties

```
Set friend;  
  
public void setFriend(Set friend) {  
    this.friend = friend;  
}
```

```
<property name="friend">  
    <set value-type="java.lang.String">  
        <value>친구1</value>  
        <value>친구2</value>  
    </set>  
</property>
```

```
Properties myprofile;  
  
public void setMyprofile(Properties myprofile) {  
    this.myprofile = myprofile;  
}
```

```
<property name="myprofile">  
    <props>  
        <prop key="prop1">propvalue1</prop>  
        <prop key="prop2">propvalue2</prop>  
    </props>  
</property>
```

# 의존관계 자동 설정

의존하는 빈객체의 타입이나 이름을 이용하여 의존객체를 자동으로 설정할 수 있다.

- autowire 속성을 이용
- 자동 설정과 직접 설정의 혼합도 가능

방식	설명
byName	property name과 같은 name을 갖는 bean객체를 설정
byType	Property type과 같은 type을 갖는 bean객체를 설정
constructor	constructor 파라미터 type과 같은 type을 갖는 bean 객체 생성자에 전달

# 의존관계 자동 설정

## 1) byName

```
<bean id="people" class="sample3.People" autowire="byName">
  <property name="name" value="홍길동"></property>
  <property name="phone" value="5118"></property>
  <!-- <property name="car" ref="car"></property> -->
```

```
Car car;

public void setCar(Car car) {
    this.car = car;
}
```

# 의존관계 자동 설정

## 2) byType

```
<bean id="people" class="sample3.People" autowire="byType">
  <property name="name" value="홍길동"></property>
  <property name="phone" value="5118"></property>
  <property name="car" ref="car"></property>

  <property name="major">
    <list>
      <value>컴퓨터공학</value>
      <value>경영학</value>
    </list>
  </property>
  <!-- <property name="licenses">
    <list>
      <ref bean="license1"/>
      <ref bean="license2"/>
    </list>
  </property> -->
```

```
List<License> licenses;

public void setLicenses(List<License> licenses) {
    this.licenses = licenses;
}
```



# Bean 객체의 범위

- ❖ 기본적으로 컨테이너에 한 개의 bean 객체를 생성한다.
- ❖ bean의 범위를 설정할 수 있는 방법을 제공한다.
- ❖ scope 속성을 이용한다.

방식	설명
singleton	컨테이너에 한 개의 bean 객체만 생성한다.(기본값)
prototype	빈을 요청할 때마다 bean 객체를 생성한다
request	HTTP 요청시 마다 bean 객체를 생성한다. (WebApplicationContext에서만 적용된다.)
session	HTTP session마다 bean 객체를 생성한다. (WebApplicationContext에서만 적용된다.)

# Bean 객체의 범위

```
<bean id="people" class="sample3.People" scope="singleton" >
```

```
public class PeopleApp {  
  
    public static void main(String[] args) {  
        process();  
    }  
  
    public static void process(){  
        ApplicationContext factory =  
            new ClassPathXmlApplicationContext("sample3.xml");  
        People p =(People) factory.getBean("people");  
        People p2 =(People) factory.getBean("people");  
        System.out.println(p==p2);  
    }  
}
```

```
<terminated> PeopleApp [Java Application] C:\Program Files\Java\jdk1.7.0_45\bin\javaw.e  
8:18, 2015 3:54:10 *org.springframework.context.support.AbstractApplic  
≡: Refreshing org.springframework.context.support.ClassPathXmlApplicati  
8:18, 2015 3:54:10 *org.springframework.beans.factory.xml.XmlBeanDefini  
≡: Loading XML bean definitions from class path resource [sample3.xml]  
8:18, 2015 3:54:10 *org.springframework.beans.factory.support.DefaultL  
≡: Pre-instantiating singletons in org.springframework.beans.factory.su  
true
```

## 4. 메시지 및 이벤트 처리

✓ 메시지 국제화 처리

# 메시지 국제화 처리

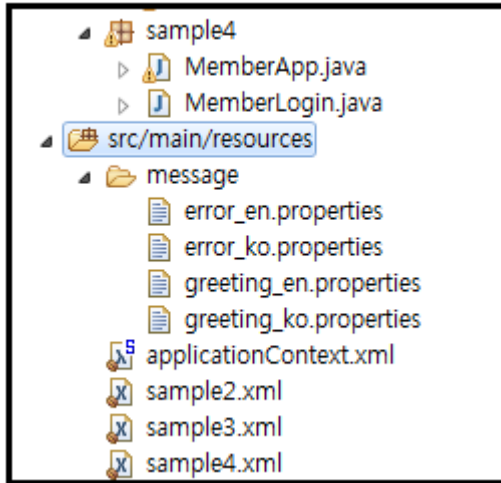
<<interface>>  
org.springframework.context.MessageSource



<<interface>>  
org.springframework.context.ApplicationContext

메시지 국제화를 지원하기 위한 interface 로서  
지역 및 언어에 따라 알맞은 메시지를 구할 수 있는 메소드를 제공한  
다. (ApplicationContext 또는 MessageSource의 getMessage()로  
메시지 획득)

# 메시지 국제화 처리



```
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>message.greeting</value>
      <value>message.error</value>
    </list>
  </property>
</bean>
```

# 메시지 국제화 처리

<<interface>>  
MessageSourceAware



사용자정의 class

```
public class MemberLogin implements MessageSourceAware{  
    MessageSource messageSource;  
  
    public void setMessageSource(MessageSource messageSource) {  
        this.messageSource = messageSource;  
    }  
  
    public String hello(){  
        //Object[] obj = {"안녕하세요", "좋은아침"};  
        Object[] obj = {"Good", "Afternoon~~"};  
        String msg = messageSource.getMessage("afternoon", obj, Locale.ENGLISH);  
        return msg;  
    }  
}
```

# 메시지 국제화 처리

```
public class MemberApp {  
    public static void main(String[] args) {  
        ApplicationContext context = new ClassPathXmlApplicationContext("sample4.xml");  
        MemberLogin member = context.getBean("memberLogin", MemberLogin.class);  
        String msg = member.hello();  
        System.out.println(msg);  
    }  
}
```

# 5. Annotation 기반 설정

- ✓ Annotation 기반 설정의 특징
- ✓ 주요 Annotation
- ✓ Bean 객체 스캔



# Annotation 기반 설정의 특징

- ① XML 설정 파일을 사용하는 대신 자바 annotation을 사용할 수 있다. (자바 5 이상)
- ② annotation의 사용으로 설정파일을 간결화하고, View 페이지와 객체 또는 메소드의 매핑을 명확하게 할 수 있다.
- ③ 소스 코드에 "@annotation"의 형태로 표현하며 클래스, 필드, 메소드의 선언부에 적용 할 수 있다.
- ④ 컴파일러가 실행되기 전에 annotation으로 설정한 내용대로 코드가 작성되었는지 확인하기 위해 실행한다.
- ⑤ 소스 코드에 메타데이터를 보관할 수 있다.
- ⑥ 코드의 가독성을 높일 수도 있다.

# 주요 Annotation

Annotation	설명
@Required	setter주입방식을 이용 javax.annotation.Resource
@Autowired	타입을 기준으로(byType) Bean을 찾아 주입 org.springframework.beans.factory.annotaion
@Resource	이름을 기준으로(byName) Bean을 찾아 주입 org.springframework.beans.factory.annotaion
@Component	자동인식이 되는 일반 컴퍼넌트 org.springframework.stereotype
@Service	비즈니스 서비스를 의미, @Component의 하위
@Repository	일반적으로 DAO에 적용, @Component의 하위
@Controller	MVC component로 사용, @Component의 하위

# 주요 Annotation

## 1) @Required

- ❖ Spring Bean의 멤버 변수인 필수 프로퍼티를 명시할 때 사용한다.
- ❖ 해당 setter 메소드 선언부에 설정한다.
- ❖ 설정 파일에 <context:annotation-config/> 를 추가한다.

```
public class People {  
    String name;  
  
    @Required  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

```
<context:annotation-config />  
  
<bean id="people" class="sample5.People" scope="singleton">  
    <property name="name" value="홍길동"></property>  
    <property name="phone" value="5118"></property>  
    <property name="car" ref="car"></property>  
</bean>
```

# 주요 Annotation

## 2) @Autowired

- ❖ Spring Bean의 의존 관계를 자동 설정할 때 사용한다.
- ❖ 타입을 이용한 프로퍼티 자동 설정 기능을 제공한다.
- ❖ 생성자, 필드, 메소드 선언부에 설정한다.
- ❖ 설정 파일에 <context:annotation-config/> 를 추가한다.

```
<bean id="people" class="sample5.People" scope="singleton">  
  <property name="name" value="홍길동"></property>  
  <property name="phone" value="5118"></property>  
  <!-- <property name="car" ref="car"></property> -->  
</bean>
```

```
@Autowired  
Car car;  
  
public void setCar(Car car) {  
    this.car = car;  
}  
  
List<License> licenses;  
@Autowired  
public void setlicenses(List<License> licenses) {  
    this.licenses = licenses;  
}
```

# 주요 Annotation

## 3) @Resource

- ❖ 애플리케이션이 필요로 하는 자원을 자동 연결해주는 기능을 제공한다.
- ❖ Spring에서 Bean 객체를 전달할때 사용한다.
- ❖ 이름으로(by name) 이용한 프로퍼티 자동 설정 기능을 제공한다.
- ❖ 해당 setter 메소드 선언부에 설정한다.
- ❖ 설정 파일에 <context:annotation-config/> 를 추가한다.

```
@Resource
Car car;
public void setCar(Car car) {
    this.car = car;
}

List<License> licenses;
@Autowired
public void setLicenses(List<License> licenses) {
    this.licenses = licenses;
}
```

# Bean 객체 스캔

- ❖ 특정 Annotation이 붙은 클래스를 자동으로 찾아서 Bean으로 등록해주는 방식이다.
- ❖ Bean Scanner가 지정된 classpath 아래에 있는 모든 패키지의 클래스를 대상으로 자바 코드에 선언된 Annotation 기능에 맞게 자동으로 선별하여 Spring Bean으로 설정한다.
- ❖ "/"를 사용하여 다수의 기본 package를 지정할 수 있다.
- ❖ @Component : 클래스 선언부에 선언한다.
- ❖ <context:component-scan base-package="package명"/> : 설정파일에 기술한다.
- ❖ @Repository, @Service, @Controller는 @Component annotation을 상속받고 있다. name 값으로 Bean의 이름을 지정 가능하며 만약 지정하지 않으면, class 이름의 첫문자를 소문자로 변환하여 Bean 이름을 자동으로 생성한다.

# Bean 객체 스캔

```
<context:component-scan base-package="sample5"></context:component-scan>
<context:annotation-config />

<!-- <bean id="people" class="sample5.People" scope="singleton">
  <property name="name" value="알루튼"></property>
  <property name="phone" value="5118"></property>
  <property name="car" ref="car"></property>
-->
```

```
@Component
public class People {
    String name;

    //@Required
    public void setName(String name) {
        this.name = name;
    }

    @Resource
    Car car;
    public void setCar(Car car) {
        this.car = car;
    }

    List<License> licenses;
    @Autowired
    public void setLicenses(List<License> licenses) {
        this.licenses = licenses;
    }
}
```

# 6. Spring AOP

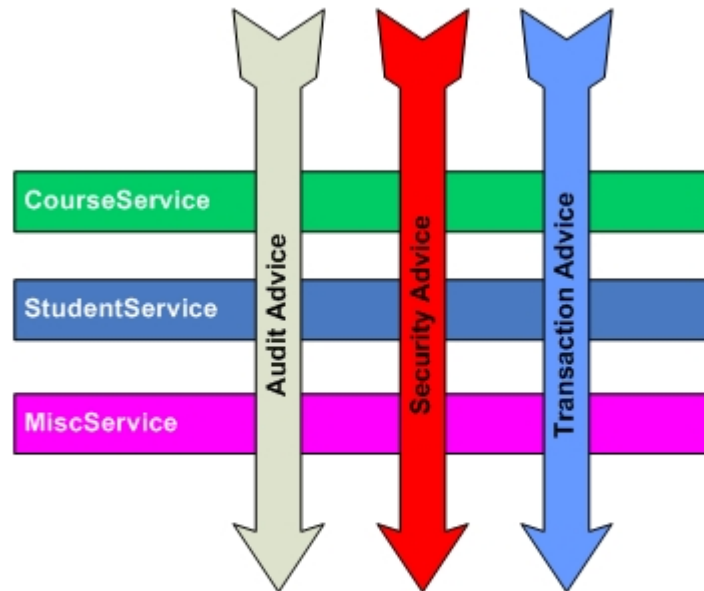
- ✓ AOP 개요
- ✓ AOP 기본 용어들
- ✓ Spring AOP 주요 용어들
- ✓ Spring AOP 구현
- ✓ Advise 종류
- ✓ XML 기반의 POJO 클래스를 이용한 AOP 구현
- ✓ Annotation 기반의 AOP 구현



# AOP 개요

## Aspect Oriented Programming (관점 지향 프로그래밍)

주업무는 크게 변화되지 않지만 주업무를 감싸는 보조업무(로그, 보안, 권한, 인증, Transaction범위 등)는 달라지는 경우가 많다. 달라지는 경우 프로그램에 많은 변경이 필요하다. 이를 줄이기 위해 보조 업무를 주 업무에서 분리하고 외부에서 주입한다.



[참조: 스프링 인 액션]

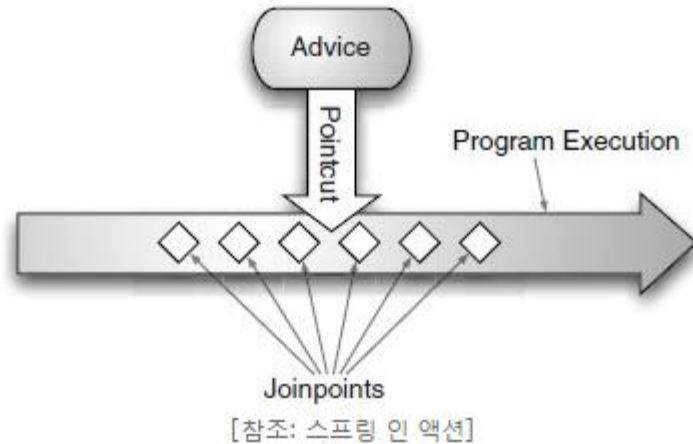
# AOP 기본 용어들

용어	설명
concern	애플리케이션을 개발하기 위하여 관심을 가지고 구현해야 하는 각각의 기능들
core concern	해당 애플리케이션이 제공하는 핵심이 되는 비즈니스 로직
cross-cutting concern	하나의 영역에서만 활용되는 고유한 관심 사항(Concern)이 아니라, 여러 클래스 혹은 여러 계층의 애플리케이션 전반에 걸쳐서 공통적으로 필요로 하는 기능

# Spring AOP 주요 용어들

용어	설명
Target	핵심 로직을 구현하는 클래스 공통 관심 사항을 적용 받게 되는 대상으로 advise가 적용되는 객체
Aspect	여러 객체에 공통으로 적용되는 공통 관심 사항
Advise	조인 포인트에 삽입되어 동작할 수 있는 공통 관심 사항의 코드 *동작시점 : 조인포인트 실행 전, 후로 before, after, after returning, after throwing, around로 구분
Joinpoint	「클래스의 인스턴스 생성 시점」, 「메소드 호출 시점」 및 「예외 발생 시점」과 같이 애플리케이션을 실행할 때 특정 작업이 시작되는 시점으로 Advice를 적용 가능한 지점 즉 어드바이스가 적용될 수 있는 위치
Pointcut	여러 개의 Joinpoint를 하나로 결합한 것
Advisor	Advice와 Pointcut를 하나로 묶어 취급한 것
weaving	공통 관심 사항의 코드인 Advice를 핵심 관심 사항의 로직에 적용 하는 것

# Spring AOP 주요 용어들



# Spring AOP 구현

- ❖ 런타임시에 위빙하는 “프록시 기반의 AOP”를 지원
  - ❖ 프록시 기반의 AOP는 메소드 호출 조인포인트만 지원
  - ❖ 어떤 대상 객체에 대해 AOP를 적용할 지의 여부는 설정 파일에 지정
  - ❖ 어떤 대상 객체에 대해 프록시를 통한 간접 접근
  - ❖ AOP 개발을 위한 추가 library : aspectjweaver.jar
- 
- ❖ 스프링은 AOP 구현을 위해 다음 세가지 방식을 제공한다.
    - ① @AspectJ 어노테이션을 이용한 AOP 구현
    - ② XML Schema를 이용한 AOP 구현
    - ③ 스프링 API를 이용한 AOP 구현

# Spring AOP 구현

❖핵심 로직 구현 방법에 따른 proxy 종류

- ① 핵심 로직의 클래스 구현시 interface가 정의되어 있는 경우  
자바 리플렉션 API가 제공하는 `java.lang.reflect.Proxy` 를 이용하여 프록시 객체를 생성
- ② interface가 정의 되어 있지 않은 핵심 로직의 클래스인 경우  
`cglib.jar`[byte code 생성 library]  
바이트코드 생성 라이브러리를 이용하여 동적으로 각각의 클래스에 대한 프록시 객체를 생성 . 이미 생성한 클래스가 있을 때는 그것을 재사용

# Advise 종류

Advise종류	XML스키마 기반/ @Aspect 기반	설명
Before	<aop:before> @before	target객체의 메소드 호출전 실행
After Returning	<aop:after-returning> @AfterReturning	target객체의 메소드가 예외 없이 실행 된 후 실행
After Throwing	<aop:after-throwing> @AfterThrowing	target객체의 메소드가 실행하는중 예외 발생시 실행
After	<aop:after> @After	target객체의 메소드 실행후 반드시 (try 문의 finally와 유사) 실행
Around	<aop:around> @Around	target객체의 메소드 실행 전, 후 또는 예외 발생 시점에 모두 실행해야 할 로 직을 담아야 할 경우 실행

# XML 기반의 POJO 클래스를 이용한 AOP 구현

## 1) Advice 클래스를 구현

```
import org.aspectj.lang.ProceedingJoinPoint;
public class LogAdvice {
    public Object around(ProceedingJoinPoint point ) throws Throwable{
        String targetName = point.getTarget().getClass().getName();
        String methodName = point.getSignature().getName();
        System.out.println("[Log]대상객체 이름: " + targetName);
        System.out.println("[Log]메서드 이름: " + methodName);
        long sTime = System.nanoTime();
        System.out.println("[Log]" + methodName + "메서드 수행전 ");
        Object obj = point.proceed();
        long eTime = System.nanoTime();
        System.out.println("[Log]" + methodName + "메서드 실행후 소요 시간:" + (eTime - sTime) + "ns");
        System.out.println("[Log]" + methodName + "메서드 실행후 받은 데이터: " + obj);
        return obj;
    }
}
```



# XML 기반의 POJO 클래스를 이용한 AOP 구현

## 1) Advice 클래스를 구현

```
public void beforeMethod(JoinPoint point){
    String targetName = point.getTarget().getClass().getName();
    String methodName = point.getSignature().getName();
    System.out.println("[before Log]대상객체 이름: " + targetName);
    System.out.println("[before Log]메서드 이름: " + methodName);
}
public void afterMethod(JoinPoint point){
    String targetName = point.getTarget().getClass().getName();
    String methodName = point.getSignature().getName();
    System.out.println("[after Log]대상객체 이름: " + targetName);
    System.out.println("[after Log]메서드 이름: " + methodName);
}
public void afterReturningMethod(JoinPoint point){
    String targetName = point.getTarget().getClass().getName();
    String methodName = point.getSignature().getName();
    System.out.println("[afterReturn Log]대상객체 이름: " + targetName);
    System.out.println("[afterReturn Log]메서드 이름: " + methodName);
}
public void afterThrowMethod(JoinPoint point){
    String targetName = point.getTarget().getClass().getName();
    String methodName = point.getSignature().getName();
    System.out.println("[afterThrow Log]대상객체 이름: " + targetName);
    System.out.println("[afterThrow Log]메서드 이름: " + methodName);
}
```

# XML 기반의 POJO 클래스를 이용한 AOP 구현

## 2) Target 클래스를 구현

```
import org.springframework.stereotype.Service;
@Service("user")
public class UserServiceImpl {

    public String writting(){
        System.out.println("user가 writting");
        return "user writting";
    }
}
```

```
import org.springframework.stereotype.Component;
@Component("member")
public class MemberServiceImpl {

    public String meeting(){
        System.out.println("member들이 회의를 합니다.");
        //int i = 10/0;
        return "카페라떼 주문";
    }
}
```

# XML 기반의 POJO 클래스를 이용한 AOP 구현

## 3) Bean 객체 사용

```
public class AopApp {  
  
    public static void main(String[] args) {  
        ApplicationContext context = new ClassPathXmlApplicationContext("sample6.xml");  
  
        MemberServiceImpl member = context.getBean("member", MemberServiceImpl.class);  
        String msg = member.meeting();  
        System.out.println(msg + "받음");  
  
        UserServiceImpl user = context.getBean("user", UserServiceImpl.class);  
        msg = user.writing();  
        System.out.println(msg + "받음");  
    }  
}
```

# XML 기반의 POJO 클래스를 이용한 AOP 구현

## 4) 설정 파일에 AOP관련정보를 설정

```
<context:component-scan base-package="sample6.aop"></context:component-scan>
<!-- <bean id="member" class="sample6.aop.MemberServiceImpl"></bean> -->

<bean id="log" class="sample6.aop.LogAdvise"></bean>
<aop:config>
  <aop:aspect id="commonAdvise" ref="log">
    <aop:pointcut expression="within(sample6.aop.*)" id="aroundLog"/>
    <aop:around method="around" pointcut-ref="aroundLog"/>
    <aop:before method="beforeMethod" pointcut-ref="aroundLog"/>
    <aop:after method="afterMethod" pointcut-ref="aroundLog"/>
    <aop:after-returning method="afterReturningMethod" pointcut-ref="aroundLog"/>
    <aop:after-throwing method="afterThrowMethod" pointcut-ref="aroundLog"/>
  </aop:aspect>
</aop:config>
```

# Annotation 기반의 AOP 구현

- 1) @AspectJ 설정  
<aop:aspectj-autoproxy/>
- 2) Aspect 정의
- 3) Pointcut 정의

```
@Component
@Aspect
public class LogAdvise {
    @Pointcut("within(sample6.aop.*)")
    public void targetMethod(){
        //pointcut annotation 값을 참조하기 위한 dummy method
    }
    @Around("targetMethod()")
    public Object around(ProceedingJoinPoint point ) throws Throwable{
```

```
@Before("targetMethod()")
public void beforeMethod(JoinPoint point){ }
@After("targetMethod()")
public void afterMethod(JoinPoint point){ }
@AfterReturning("targetMethod()")
public void afterReturningMethod(JoinPoint point){ }
@AfterThrowing("targetMethod()")
public void afterThrowMethod(JoinPoint point){ }
```

# Pointcut 표현식

## 1) execution() 표현식

- ① 가장 대표적인 강력한 포인트컷의 지시자
- ② 메소드의 signature를 이루는 접근 제한자, 리턴타입, 메소드명, 파라미터 타입, 예외타입 조건을 조합해서 메소드 단위까지 선택 가능한 가장 정교한 포인트컷 구성 가능

**execution([접근제한자패턴] 리턴타입 [패키지패턴]이름패턴 (파라미터패턴) )**

예제	설명
execution(* print(..))	메서드 이름이 print인 모든 메서드 선정 (리턴 타입, 파라미터 무관)
execution(* drive(int, int))	메소드 이름이 drive이며, 두 개의 int 타입의 파라미터를 가진 모든 메소드 선정(리턴 타입 무관)
execution(* *(..))	리턴 타입, 파라미터의 종류, 개수에 상관없이 모든 메소드 선정

# Pointcut 표현식

## 2) within() 표현식

- ① 특정 타입에 속하는 메소드를 포인트컷으로 설정시 사용
- ② execution()의 여러 조건 중에서 타입 패턴만을 적용
- ③ target 클래스의 타입에만 적용되며 조인포인트는 target 클래스 안에서 선언된 것만 선정됨
- ④ 선택된 타입의 모든 메소드가 AOP 적용대상

예제	설명
<code>within(sample.aop...*)</code>	sample.aop 및 모든 서브패키지가 포함하고 있는 모든 메소드
<code>within(sample.aop.*)</code>	sample.aop 패키지 아래의 인터페이스와 클래스에 있는 모든 메소드
<code>within(sample.aop.Member)</code>	sample.aop 패키지의 Member클래스의 모든 메소드

# Pointcut 정의예제

Pointcut	선택된 Joinpoints
<code>execution(public * *(..))</code>	public 메소드 실행
<code>execution(* set*(..))</code>	이름이 set으로 시작하는 모든 메소드명 실행
<code>execution(* com.xyz.service.AccountService.*(..))</code>	AccountService 인터페이스의 모든 메소드 실행
<code>execution(* com.xyz.service.*.*(..))</code>	service 패키지의 모든 메소드 실행
<code>execution(* com.xyz.service..*.*(..))</code>	service 패키지 및 하위 패키지의 모든 메소드 실행
<code>within(com.xyz.service.*)</code>	service 패키지 내의 모든 결합점
<code>within(com.xyz.service..*)</code>	service 패키지 및 하위 패키지의 모든 결합점
<code>this(com.xyz.service.AccountService)</code>	AccountService 인터페이스를 구현하는 프록시 개체의 모든 결합점
<code>target(com.xyz.service.AccountService)</code>	AccountService 인터페이스를 구현하는 대상 객체의 모든 결합점
<code>args(java.io.Serializable)</code>	하나의 파라미터를 갖고 전달된 인자가 Serializable인 모든 결합점
<code>@target(org.springframework.transaction.annotation.Transactional)</code>	대상 객체가 @Transactional 어노테이션을 갖는 모든 결합점
<code>@within(org.springframework.transaction.annotation.Transactional)</code>	대상 객체의 선언 타입이 @Transactional 어노테이션을 갖는 모든 결합점
<code>@annotation(org.springframework.transaction.annotation.Transactional)</code>	실행 메소드가 @Transactional 어노테이션을 갖는 모든 결합점
<code>@args(com.xyz.security.Classified)</code>	단일 파라미터를 받고, 전달된 인자 타입이 @Classified 어노테이션을 갖는 모든 결합점
<code>bean(accountRepository)</code>	"accountRepository" 빈
<code>!bean(accountRepository)</code>	"accountRepository" 빈을 제외한 모든 빈
<code>bean(*)</code>	모든 빈
<code>bean(account*)</code>	이름이 'account'로 시작되는 모든 빈
<code>bean(*Repository)</code>	이름이 "Repository"로 끝나는 모든 빈
<code>bean(accounting/*)</code>	이름이 "accounting/"로 시작하는 모든 빈
<code>bean(*dataSource)    bean(*DataSource)</code>	이름이 "dataSource" 나 "DataSource" 으로 끝나는 모든 빈

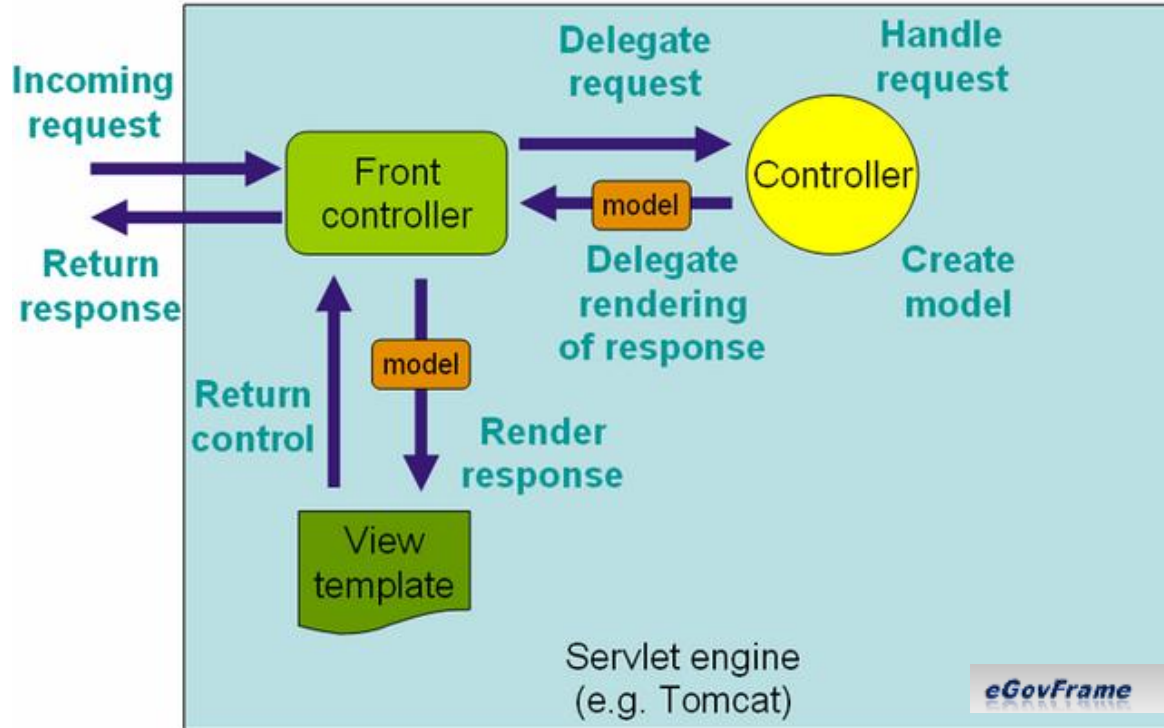


# 7. Spring MVC 웹 프로그램

- ✓ Spring Web MVC framework
- ✓ Spring Web MVC 개발
- ✓ @Controller 관련 Annotation
- ✓ Parameter 전달
- ✓ @Controller 메서드의 리턴
- ✓ View설정
- ✓ 캐릭터 인코딩 처리 필터 설정
- ✓ Exception Handling

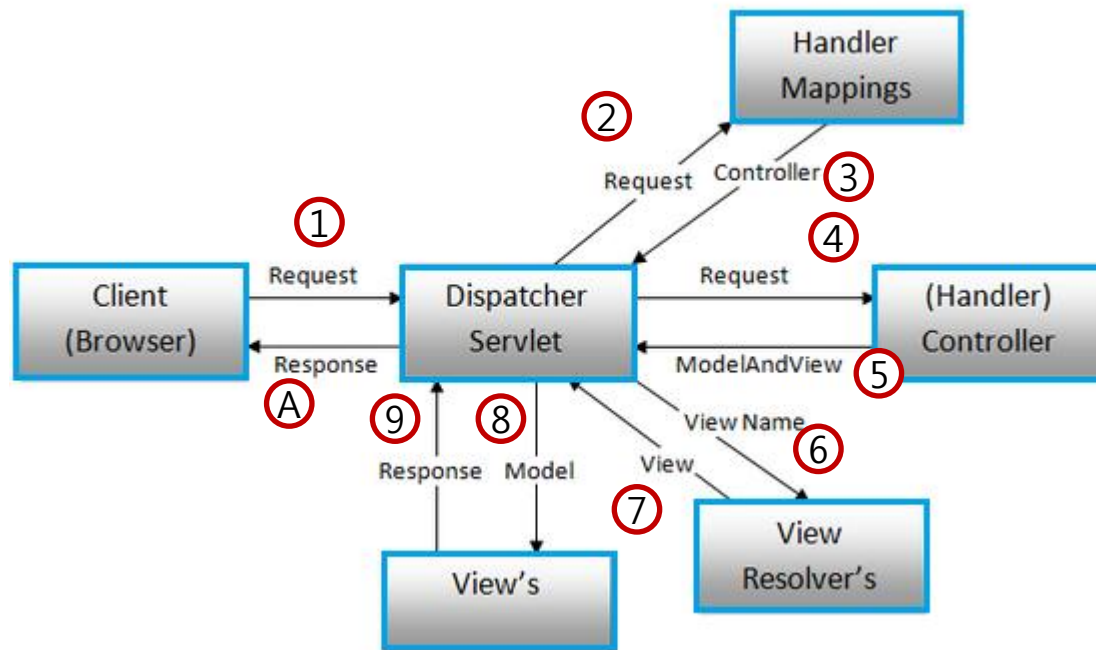
# Spring Web MVC framework

- ❖ 서블릿 기반의 MVC Model2 구조를 제공한다.
- ❖ 전체 애플리케이션을 통합, 관리하기 위해 서버로 들어오는 모든 요청을 먼저 받아서 처리하도록 구성된 Front Controller 패턴을 적용한다.



# Spring Web MVC framework

## ❖ Spring MVC 컴포넌트간의 관계와 흐름



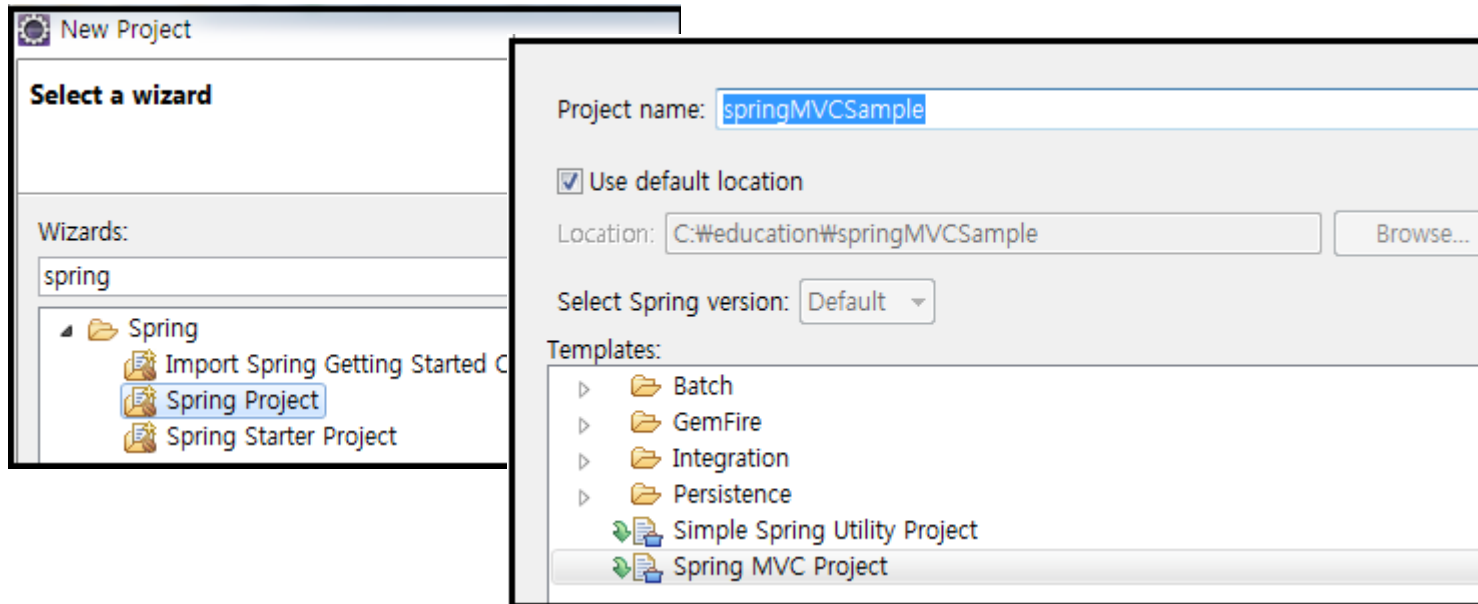
# Spring Web MVC framework

## ❖ Spring MVC 주요 구성요소

구성 요소	설명
DispatcherServlet	클라이언트의 요청을 전달받아 요청에 맞는 컨트롤러를 실행하고 리턴한 결과값을 View에 전달하여 알맞은 응답을 생성한다.
HandlerMapping	클라이언트의 요청 URL을 어떤 컨트롤러가 처리할지 결정한다.
Controller	클라이언트의 요청을 처리하고 결과를 DispatcherServlet에게 리턴한다.
ModelAndView	컨트롤러가 처리한 결과 정보와 뷰 선택에 필요한 정보를 담을 수 있다.
ViewResolver	컨트롤러의 처리 결과를 보여 줄 뷰를 결정한다.
View	컨트롤러의 처리 결과 화면을 생성한다. (JSP)

# Spring Web MVC 개발

## 1) Spring MVC Project



### Project Settings - Spring MVC Project

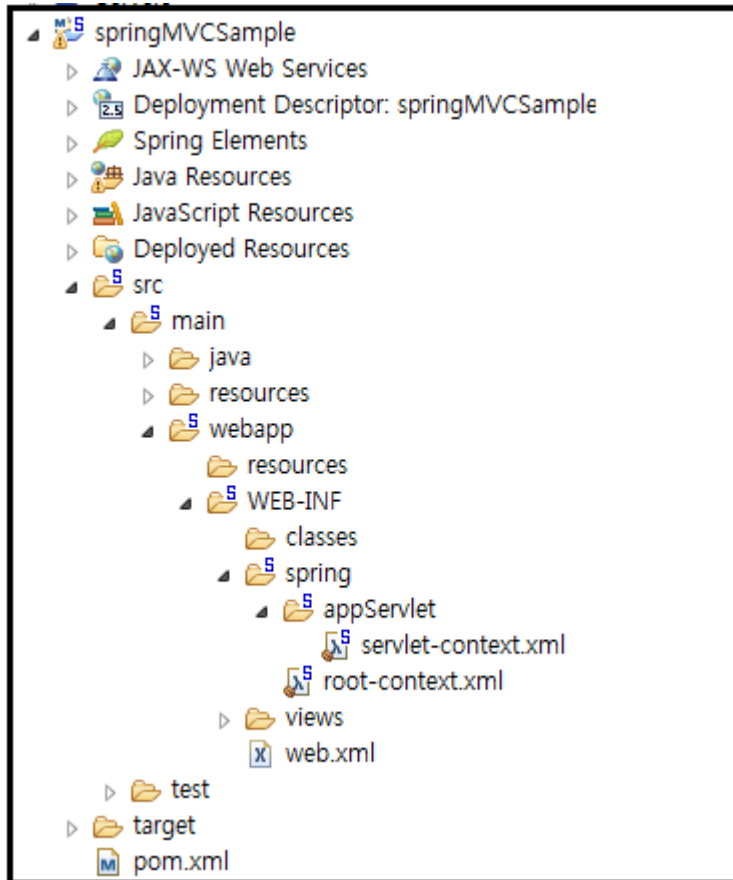
Define project specific settings. Required settings are denoted by "\*".

Please specify the top-level package e.g. com.mycompany.myapp\*

com.bit.education

# Spring Web MVC 개발

## 2) Spring MVC project구성



# Spring Web MVC 개발

## 3) web.xml

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

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

Spring 설정 파일  
(servlet-context.xml)

모든 client 의 요청을  
DispatcherServlet이 처리하도록 요청

# Spring Web MVC 개발

## 4) servlet-context.xml

```
<!-- DispatcherServlet Context: defines this servlet's request-processing infrastructure -->
<!-- Enables the Spring MVC @Controller programming model -->
<annotation-driven />
<!-- Handles HTTP GET requests for
/resources/** by efficiently serving up static resources
in the ${webappRoot}/resources directory -->
<resources mapping="/resources/**" location="/resources/" />
<!-- Resolves views selected for rendering by
@Controllers to .jsp resources in the /WEB-INF/views directory -->
<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <beans:property name="prefix" value="/WEB-INF/views/" />
    <beans:property name="suffix" value=".jsp" />
</beans:bean>
<context:component-scan base-package="com.bit.education" />
```

- ❖ <context:component-scan/> 설정  
@Component, @Service, @Repository, @Controller 가 붙은 클래스들을  
읽어들여 ApplicationContext, WebApplicationContext에 Bean정보를 저장,  
관리한다.
- ❖ ViewResolver 설정



# Spring Web MVC 개발

## 5) Controller와 View구현

```
@Controller
public class HelloController {

    @RequestMapping(value="hello.do")
    public ModelAndView hello(){
        ModelAndView mv = new ModelAndView();
        mv.addObject("title", "안녕하세요");
        mv.setViewName("helloForm");
        return mv;
    }
}
```

```
<body>
    <h1>Spring MVC Project</h1>
    <h2>${title }</h2>
</body>
```

← → 📄 🔄 http://localhost:9090/education/hello.do

Spring MVC Project

안녕하세요

# @Controller 관련 Annotation

Annotation	설명
@Controller	해당 클래스가 Controller임을 의미함.
@RequestMapping	요청에 대해 어떤 Controller, 어떤 메소드가 처리할지를 맵핑하기 위한 것.
@RequestParam	Controller 메소드의 파라미터와 웹요청 파라미터를 맵핑하기 위한 것.
@ModelAttribute	Controller 메소드의 파라미터나 리턴값을 Model 객체와 바인딩하기 위한 것
@SessionAttributes	Model 객체를 세션에 저장하고 사용하기 위한 것. 클래스 레벨(type level)에서 선언

# @Controller 관련 Annotation

## 1) @RequestMapping

- ❖ 요청에 대해 어떤 Controller, 어떤 메소드가 처리할지를 매핑하기 위한 어노테이션이다.
- ❖ 클래스 단위(type level)나 메소드 단위(method level)로 설정할 수 있다.

type level

```
@Controller
@RequestMapping(value="hello.do")
public class HelloController {

    @RequestMapping
    public ModelAndView hello(){
        ModelAndView mv = new ModelAndView();
        mv.addObject("title", "안녕하세요");
        mv.setViewName("helloForm");
        return mv;
    }
}
```

# @Controller 관련 Annotation

## 1) @RequestMapping

method level

```
@Controller
public class HelloController {
    @RequestMapping(value="hello.do", method=RequestMethod.GET)
    public ModelAndView helloGet(){
        ModelAndView mv = new ModelAndView();
        mv.addObject("title", "안녕하세요");
        mv.setViewName("helloForm");
        return mv;
    }
    @RequestMapping(value="hello.do", method=RequestMethod.POST)
    public ModelAndView helloPost(){
        ModelAndView mv = new ModelAndView();
        mv.addObject("title", "안녕하세요");
        mv.setViewName("helloForm");
        return mv;
    }
}
```

# @Controller 관련 Annotation

## 1) @RequestMapping

type level + method level

```
@Controller
@RequestMapping("/hello.do")
public class HelloController {
    @RequestMapping(method=RequestMethod.GET)
    public ModelAndView helloGet(){
        ModelAndView mv = new ModelAndView();
        mv.addObject("title", "안녕하세요");
        mv.setViewName("helloForm");
        return mv;
    }
    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView helloPost(){
        ModelAndView mv = new ModelAndView();
        mv.addObject("title", "안녕하세요");
        mv.setViewName("helloForm");
        return mv;
    }
}
```

# @Controller 관련 Annotation

## 1) @RequestMapping

관련속성	타입	설명
value	String[]	URL값이다. @RequestMapping(value="/insert.do") @RequestMapping(value={"/insert.do", "/memberInsert.do" }) <b>@RequestMapping("/memberInsert.do")</b>
Method	Request Method[]	HTTP Request method값이다. 사용 가능한 메소드 : GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE <b>@RequestMapping(method = RequestMethod.POST)</b>
params	String[]	HTTP Request URL중에 파라미터 이름과 값을 맵핑한다. <b>@RequestMapping(params="Param이름=값" )</b>  @RequestMapping(params={"param1=value1", "param2", "!param3"}) Param1이 value1값을 가지고 있고, param2 파라미터가 있어야 하고, param3라는 파라미터는 없어야 한다.

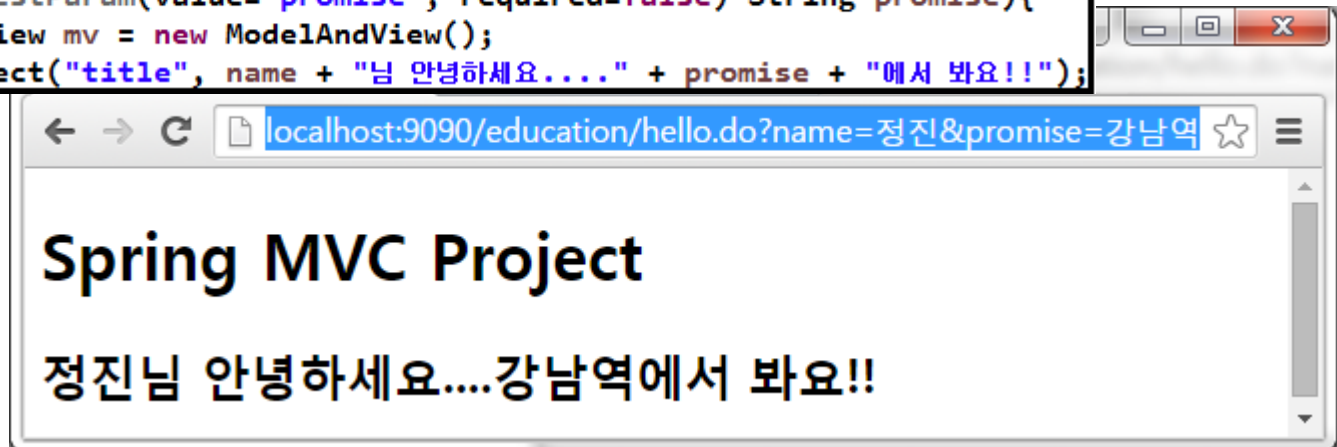
# @Controller 관련 Annotation

## 2) @RequestParam

- ❖ Controller 메소드의 파라미터와 웹요청 파라미터를 맵핑하기 위한 어노테이션이다.

관련속성	타입	설명
value	String	파라미터 이름
required	boolean	해당 파라미터가 반드시 필수인지 여부, 기본값은 true

```
@RequestMapping(method=RequestMethod.GET)
public ModelAndView helloGet(
    @RequestParam(value="name") String name,
    @RequestParam(value="promise", required=false) String promise){
    ModelAndView mv = new ModelAndView();
    mv.addObject("title", name + "님 안녕하세요...." + promise + "에서 봐요!!");
}
```



# Parameter 전달

## 1) HTML 폼과 자바빈 객체

- ❖ 스프링 MVC는 HTML 폼에 입력한 데이터를 자바빈 객체를 이용해서 전달.
- ❖ HTML 폼의 항목 이름과 자바빈 클래스의 프로퍼티 이름이 일치할 경우 폼에 입력한 값을 자바빈 객체의 프로퍼티 값으로 설정해주는 기능.
- ❖ @RequestMapping이 적용된 메서드의 파라미터로 자바빈 타입 추가.

```
public class MemberDTO {  
    String mname;  
    String email;  
    String pwd;  
  
    public String getMname() {  
        return mname;  
    }  
    public void setMname(String mname) {  
        this.mname = mname;  
    }  
    public String getEmail() {  
        return email;  
    }  
    public void setEmail(String email) {  
        this.email = email;  
    }  
}
```

```
<form action="add.do" method="post">  
    <table>  
        <tr>  
            <td>이름</td>  
            <td><input type="text" name="mname" ></td>  
        </tr>  
        <tr>  
            <td>이메일</td>  
            <td><input type="text" name="email" ></td>  
        </tr>  
        <tr>  
            <td>비밀번호</td>  
            <td><input type="text" name="pwd" ></td>  
        </tr>  
    </table>  
</form>
```



# Parameter 전달

## 1) HTML 폼과 자바빈 객체

```
@Controller
public class MemberController {
    @RequestMapping(value="add.do", method=RequestMethod.GET)
    public String memberAddGet(){
        return "memberForm";
    }
    @RequestMapping(value="add.do", method=RequestMethod.POST)
    public ModelAndView memberAddPost(@ModelAttribute("mname") String name,
                                      MemberDTO member){

        System.out.println(member);
        System.out.println(name);
        ModelAndView mv = new ModelAndView();
        mv.addObject("member", member);
        mv.addObject("msg", "입력에 성공하였습니다.");
        mv.setViewName("success");
        return mv;
    }
}
```

← → ↺ 📄 localhost:9090/education/add.do

```
<body>
  <h1>${msg }</h1>
  <h1>${member }</h1>
  <h1>${member.mname }</h1>
  <h1>${mname }</h1>
</body>
```

입력에 성공하였습니다.

MemberDTO [mname=정진, email=wed0406@daum.net, pwd=1234]

정진

정진

# Parameter 전달

## 2) Servlet API 파라미터 전달

컨트롤러 클래스의 @RequestMapping 적용 메서드에는 다섯 가지 타입의 파라미터를 전달 받을 수 있다.

- ❖ javax.servlet.http.HttpServletRequest
- ❖ javax.servlet.ServletRequest
- ❖ javax.servlet.http.HttpServletResponse
- ❖ javax.servlet.ServletResponse
- ❖ javax.servlet.http.HttpSession

```
@RequestMapping(value="list.do")
public String meberList(HttpServletRequest request){
    HttpSession session = request.getSession();
    MemberDTO loginMember = (MemberDTO)session.getAttribute("member");
    System.out.println(loginMember);
    return "memberlist";
}
```

# @Controller 메서드의 리턴

메소드 리턴타입	설명
ModelAndView	리턴 데이터가 Model객체와 View정보가 담긴다.
Model	리턴 데이터가 Model객체에 담긴다.
String	리턴하는 String 값이 View 이름이 된다.
void	메소드가 ServletResponse / HttpServletResponse등을 사용하여 직접 응답을 처리하는 경우이다.

# View 설정

- ❖ 뷰 이름을 명시적으로 지정하려면 ModelAndView나 String을 리턴해야 한다.  
ModelAndView 클래스의 생성자 또는  
setViewName()을 이용해서 뷰 이름을 지정할 수 있다.
- ❖ RequestToViewNameTranslator  
URL로부터 뷰 이름을 결정  
리턴타입이 Model이나 Map인 경우
- ❖ DefaultRequestToViewNameTranslator  
리턴 타입이 void이면서 ServletResponse나 HttpServletResponse 타입의  
파라미터가 없는 경우  
요청 URI로부터 맨 앞의 슬래시와 확장자를 제외한 나머지 부분을 뷰 이름  
으로 사용한다.

# View 설정

방법1)

```
@RequestMapping("/member/list.do")
public ModelAndView memberlist( ){
    ModelAndView mv = new ModelAndView("member/list");
    return mv;
}
```

방법2)

```
@RequestMapping("/member/list.do")
Public ModelAndView memberlist( ){
    ModelAndView mv = new ModelAndView( );
    mv.setViewName("member/list");
}
```

방법3)

```
@RequestMapping("/member/list.do")
public Map<String,Object> memberlist( ){
    HashMap<String, Object> model =
        new HashMap<String,Object> ();
    return model;
}
```

# View 설정

## ❖ Redirect

뷰 이름에 redirect: 접두어를 붙이면, 지정한 페이지로 리다이렉트 된다.

① 현재 서블릿 컨텍스트에 대한 상대적인 경로로 리다이렉트  
redirect:/member/list

② 지정한 절대 URL로 리다이렉트  
redirect:http://localhost:9090/education/list

# 캐릭터 인코딩 처리 필터 설정

스프링은 요청 파라미터의 Character Encoding을 설정할 수 있는 필터 클래스를 제공한다.

```
<filter>
  <filter-name>encodingFilter</filter-name>
  <filter-class>
    org.springframework.web.filter.CharacterEncodingFilter
  </filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>encodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

# Exception Handling

- ❖ 컨트롤러의 @RequestMapping이 선언된 메소드는 모든 타입의 예외를 발생 시킬 수 있다. @RequestMapping메서드가 예외를 발생시킬 경우 웹 브라우저에서 500 응답 코드와 함께 서블릿 컨테이너가 출력한 에러 페이지가 출력된다.

구성요소	설명
HandlerExceptionResolver	HandlerExceptionResolver 인터페이스의 resolveException() 메서드는 발생한 예외 객체를 파라미터로 전달받는다.
@ExceptionHandler	@Controller annotation이 적용된 클래스에 @ExceptionHandler annotation이 적용된 메소드 구현한다.
설정 파일을 이용한 선언적 예외 처리	SimpleMappingExceptionHandler클래스를 이용하기 위해 설정 파일에 기술한다.

```
@ExceptionHandler(Exception.class)
public String processException(Exception e) {
    e.printStackTrace();
    System.out.println(e.getMessage());
    return "/errorPage";
}
```



# Exception Handling

SimpleMappingExceptionHandler는 예외 타입 이름과 특정 뷰 이름 매핑

```
<bean  
class="org.springframework.web.servlet.handler.SimpleMappingExceptionHandler">  
  <property name="exceptionMappings">  
    <props>  
      <prop key="java.lang.ArithmeticException">  
        error/arithmeticException  
      </prop>  
      <prop key="java.lang.Exception">  
        error/exception  
      </prop>  
    </props>  
  </property>  
</bean>
```

# 8. Spring의 Database 연동

- ✓ 범용 데이터 액세스 예외 클래스
- ✓ Data Source 설정
- ✓ Spring Transaction

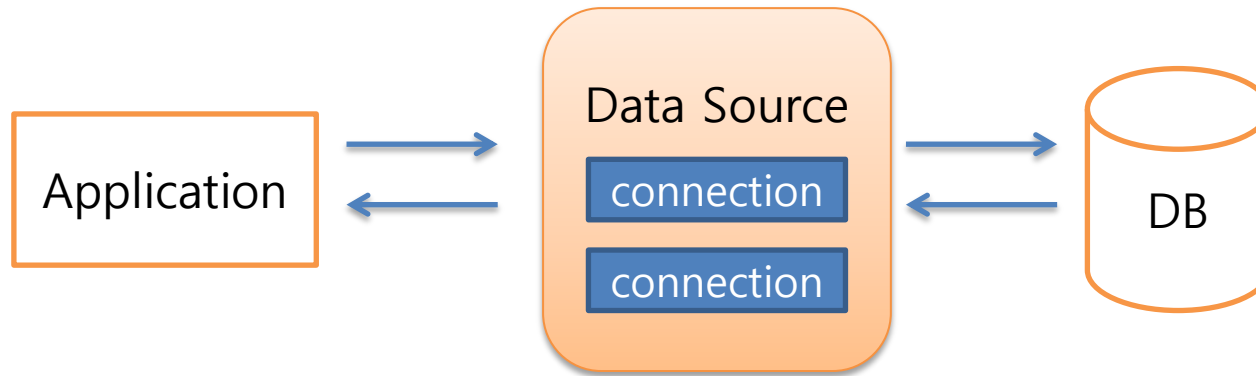
# 범용 데이터 액세스 예외 클래스

- ❖ Spring은 JDBC와 여러 ORM 프레임워크( Hibernate , mybatis 등)와 연동한다.
- ❖ 예외 처리 클래스가 특정 데이터 access 기술에 의존적이지 않도록 예외를 범용예외로 변환해서 처리한다. (설정파일에 등록)

```
<bean  
class="org.springframework.dao.annotation.PersistenceExceptionTr  
anslationPostProcessor" />
```

범용 데이터 액세스 예외 클래스	오류의 원인
DataSourceResourceFailureException	데이터 소스와의 연결에 실패시 발생
EmptyResultDataAccessException	획득하려는 데이터가 미존재시 발생
PermissionDeniedDataAccessException	권한 오류시 발생

# Data Source 설정



- ❖ Connection 객체의 Factory
- ❖ Connection 객체의 생애 주기 관리
- ❖ Connection 객체 생성, 삭제시의 시스템 부하 감소

# Data Source 설정

- 1) Connection Pool을 이용하여 Data Source 설정  
서드파티(apache 제공 : DBCP)가 제공  
jar 파일 : common-dbcnp.jar, commons-pool.jar

```
<dependency>
  <groupId>commons-dbcnp</groupId>
  <artifactId>commons-dbcnp</artifactId>
  <version>1.4</version>
</dependency>

<beans:bean name="dataSource"
  class="org.apache.commons.dbcp.BasicDataSource">
  <beans:property name="driverClassName"
    value="oracle.jdbc.driver.OracleDriver"/>
  <beans:property name="url"
    value="jdbc:oracle:thin:@localhost:1521:xe"/>
  <beans:property name="username" value="hr"/>
  <beans:property name="password" value="hr"/>
</beans:bean>
```

# Data Source

## 2) JNDI을 이용하여 Data Source 설정

서버가 관리하는 데이터 소스 객체를 서버에 내장된 네이밍서비스로 관리

```
<Context>  
  <Resource auth="Container" driverClassName="oracle.jdbc.OracleDriver"  
    maxActive="20" maxIdle="10" maxWait="-1"  
    name="jdbc/myoracle"  
    password="hr" type="javax.sql.DataSource"  
    url="jdbc:oracle:thin:@127.0.0.1:1521:xe" username="hr"/>  
</Context>
```

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/myoracle"  
  resource-ref="true" />
```

# Data Source 설정

## 3) DriverManager을 이용하여 Data Source 설정

SingleConnectionDataSource 및 DriverManagerDataSource 등의 API 활용한다. 단 Connection Pool 기능을 지원하지 않는다.

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

```
<beans:bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <beans:property name="driverClassName"
    value="oracle.jdbc.driver.OracleDriver"/>
  <beans:property name="url"
    value="jdbc:oracle:thin:@localhost:1521:xe"/>
  <beans:property name="username" value="hr"/>
  <beans:property name="password" value="hr"/>
</beans:bean>
```

# Spring Mybatis설정

## [pom.xml]

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.2.2</version>
</dependency>

<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
  <version>1.2.0</version>
</dependency>
```

## [servlet-context.xml]

```
<beans:bean id="sqlSessionFactory"
  class="org.mybatis.spring.SqlSessionFactoryBean">
  <beans:property name="dataSource" ref="dataSource" />
  <beans:property name="mapperLocations" value="classpath:/mapper/*.xml" />
</beans:bean>

<beans:bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
  <beans:constructor-arg index="0" ref="sqlSessionFactory" />
</beans:bean>
```



# Spring Transaction

**트랜잭션이란?** 논리적인 하나의 작업단위를 의미한다.  
여러 단계의 작업이 처리될 경우 모든 단계의 작업이 성공하거나 실패해야 하는 논리적인 작업의 묶음이다.  
모두 성공시 commit 처리하며 하나라도 실패시 rollback 처리한다.

**Spring**은 데이터베이스 연동 기술과 트랜잭션 서비스 사이의 종속성을 제거하고 Spring이 제공하는 트랜잭션 추상 계층을 이용해서 데이터베이스 연동 기술에 상관없이 동일한 방식으로 트랜잭션 기능을 활용하도록 한다.

**DataSourceTransactionManager API** : Connection의 트랜잭션 API를 이용하여 트랜잭션을 관리해주는 트랜잭션 매니저이다.

**<tx:method name= "\*" propagation="REQUIRED"/>** : 메소드 수행시 트랜잭션이 필요하다는 의미이다. 현재 진행중인 트랜잭션이 존재하면 해당 트랜잭션을 사용하며 존재하지 않을 경우엔 새로운 트랜잭션을 생성한다.

# Spring Transaction

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

<beans:bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
p:dataSource-ref="dataSource"/>

<!-- tx 반영 로직 범위 설정, tx 기능의 객체와 매핑 :모든 메서드에 적용-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
<!-- AOP로 biz 로직 범위 지정 + tx매핑 : -->
<aop:config>
    <aop:advisor advice-ref="txAdvice" pointcut="within(member..*)"/>
</aop:config>
```

# Spring file upload

## ❖ 필요 jar 파일

com.springsource.org.apache.commons.fileupload.jar,  
com.springsource.org.apache.commons.io.jar

```
<dependency>  
  <groupId>commons-fileupload</groupId>  
  <artifactId>commons-fileupload</artifactId>  
  <version>1.2.2</version>  
</dependency>  
<dependency>  
  <groupId>commons-io</groupId>  
  <artifactId>commons-io</artifactId>  
  <version>1.3.2</version>  
</dependency>
```

# Spring file upload

- ❖ HTML <form> 태그의 enctype 속성을 multipart/form-data 설정  
`<form action="add.do" method="post" enctype="multipart/form-data">`  
`<input type="file" name="uploadfile" required="required">`
- ❖ 스프링 설정 파일에 MultipartResolver 설정  
`<bean name="multipartResolver"`  
`class="org.springframework.web.multipart.commons.CommonsMultipartRes`  
`olver">`  
`<property name="maxUploadSize" value="10000000"/>`  
`</bean>`
- ❖ DTO에 MultipartFile 파이프의 변수, setter추가  
`public class MemberDTO{`  
`private String email, ename, fileName;`  
`private MultipartFile uploadfile;`  
`.....`  
`}`

# Spring file upload

```
@RequestMapping(value = "/add.do", method = RequestMethod.POST)
public String memberInsert(MemberDTO member, HttpServletRequest request) {
    MultipartFile uploadfile = member.getUploadfile();
    if (uploadfile == null) return "redirect:list.do";
    String path = request.getSession().getServletContext().getRealPath("/upload");
    String fileName = uploadfile.getOriginalFilename();
    String fpath = path + "\\WW" + fileName ;
    member.setFileName(fpath);
    try {
        // 방법1) FileOutputStream 사용
        // byte[] fileData = file.getBytes();
        // FileOutputStream output = new FileOutputStream(fpath);
        // output.write(fileData);
        // 2. File 사용
        File file = new File(fpath);
        uploadfile.transferTo(file);
    } catch (IOException e) {    e.printStackTrace();    }
}
return "redirect:list.do";
}
```

# Spring file download

```
<a href="download.do?p=upload&f=${member.fileName}">
    ${member.fileName}</a>
 <br /> ${member.email}
```

```
@RequestMapping("/download.do" )
public void download(String p, String f, HttpServletRequest request,
    HttpServletResponse response) throws IOException{
    response.setHeader("Content-Disposition", "attachment;filename="+f);
    String fullPath = request.getSession().getServletContext().getRealPath( p + "/" + f );
    FileInputStream fi = new FileInputStream(fullPath);
    ServletOutputStream sout = response.getOutputStream();
    byte[] buf = new byte[1024];
    int size = 0;
    while((size = fi.read(buf, 0, 1024))!=-1){
        sout.write(buf, 0, size);
    }
    fi.close();
    sout.close();
}
```