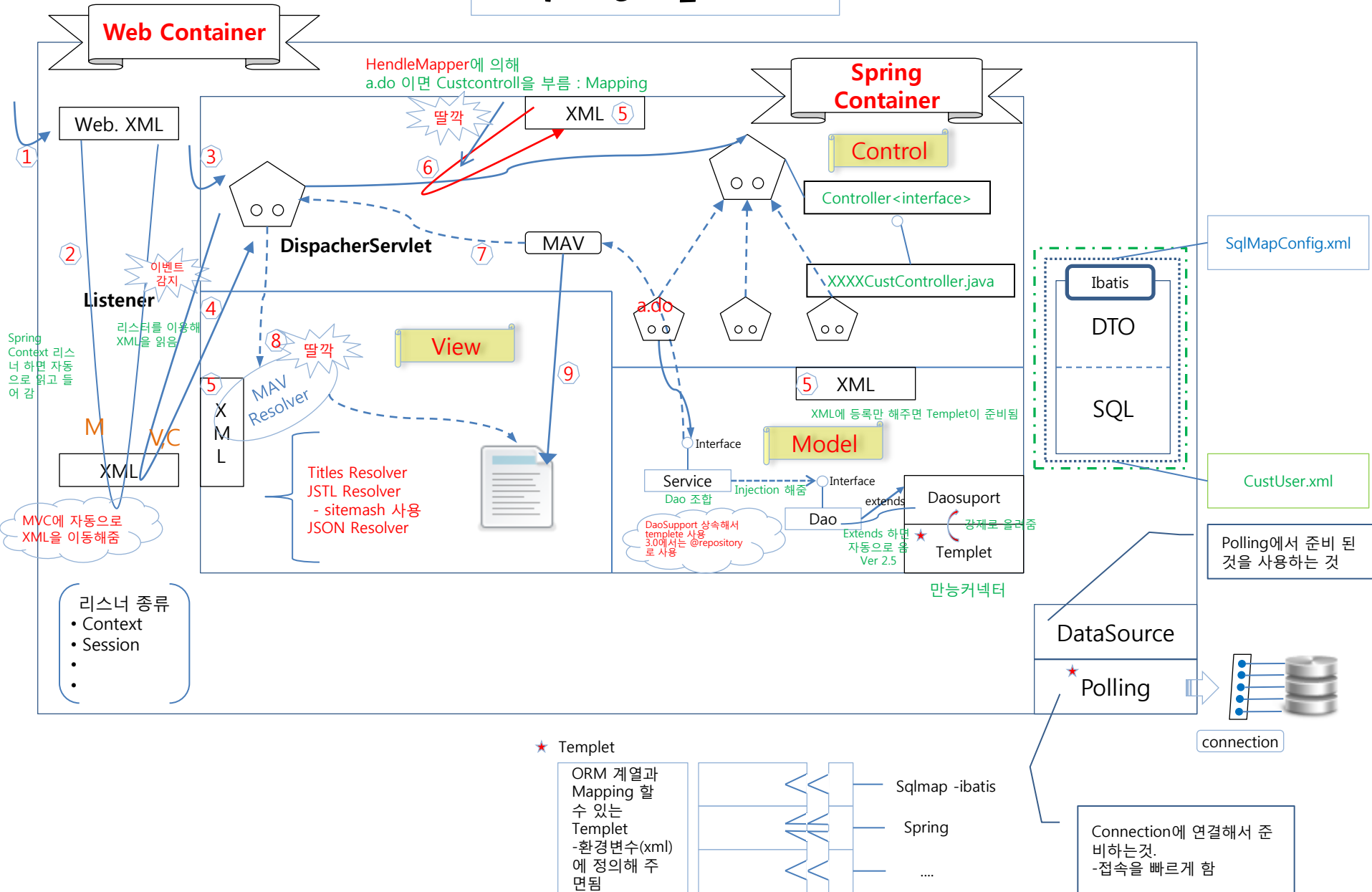
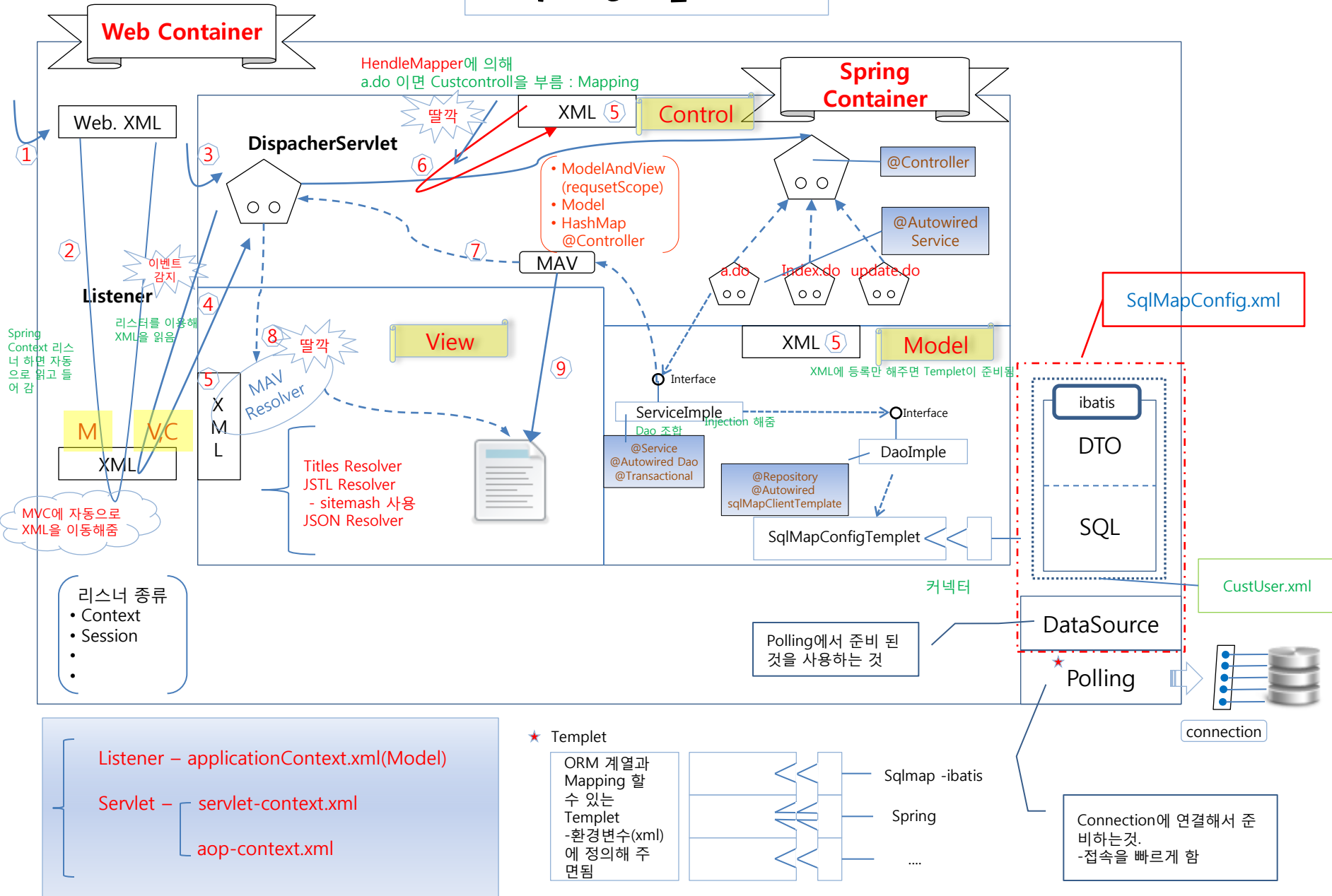


SPRING 30

Spring 흐름 v2.5



Spring 흐름 v3.0



Spring 흐름

- ① Spring container에 접근하기 전에 web.xml을 읽는다.
listener와 dispatcher가 읽힌다.
- ② Listener가 mvc에 관련된 환경변수를 읽어 들이고, 변화가 없는 정적인 정보들, model에 관련된 정보를 주로 읽는것이 좋다.
- ③ dispatcher servlet으로 이동
- ④ Dispatcher servlet이 v,c에 관련된 정보를 읽어 들인다.
- ⑤ mvc에 대한 정보를 모두 읽어 들였다는 표시
- ⑥ list.do 라고 부르면, list.do라는 이름을 가진 controller로 가게 되는데
hendleMapper가 중간에서 가로채 위치를 알려준다.
- ⑦ Controller에서 dao를 읽어 들여 필요한 db를 가져오며, 객체를 담아 지정된
위치로 갈 수 있도록 ModelAndView가 처리해준다.
- ⑧ ModelAndView에서 넘어온 정보를 resolver가 가로채 저장된 위치를 알려준다.
- ⑨ ModelAndView가 객체를 가지고 지정된 뷰로 이동한다.

Spring 3대 용어

D.I (Dependency Injection) / I.O.C (Inversion of Control)

A.O.P (Aspect Oriented Programming)

O.C.P(Open Closed Principle)

D.I

- 밖에서 만들어서 안에다 넣어줌

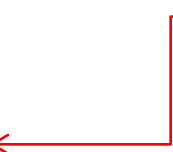
셋터 주입(setter), 생성자 주입(constructor), 메소드 주입(method)

```
@Controller
public class TestController{
    private ITestService testServiceImpl;
    private ITestDao testDaoImpl;

    //Setter Injection
    public setTestService(ITestServiceImpl testServiceImpl){
        this.testServiceImpl = testServiceImpl;
    }
}
```

Make(IMagic magic){
 아규먼트의 타입은 IMagic(부모타입으로 설정해서 모든 타입을 받을 수 있게 해준다.
}

New magic

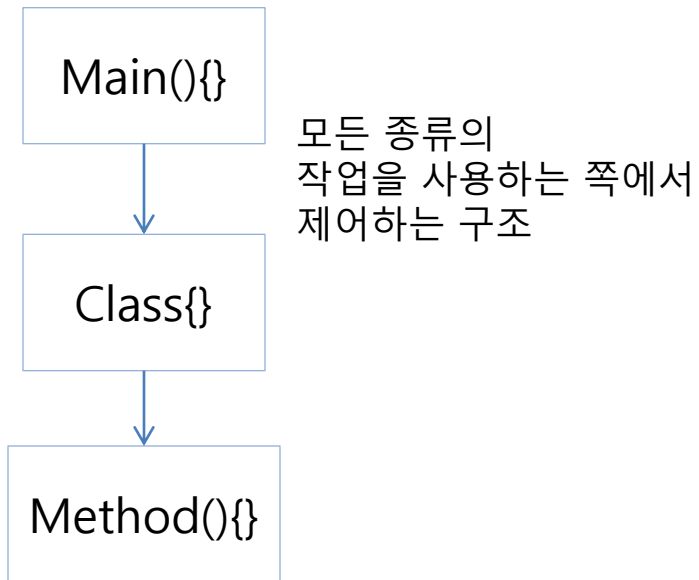


I.O.C

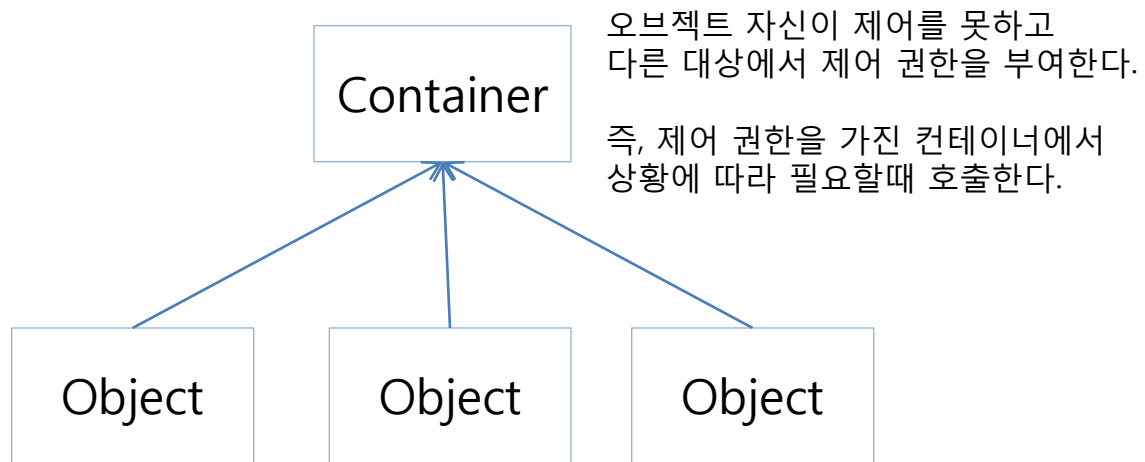
제어 관계의 역전

-프로그램의 제어 흐름 구조가 뒤바뀌는 것

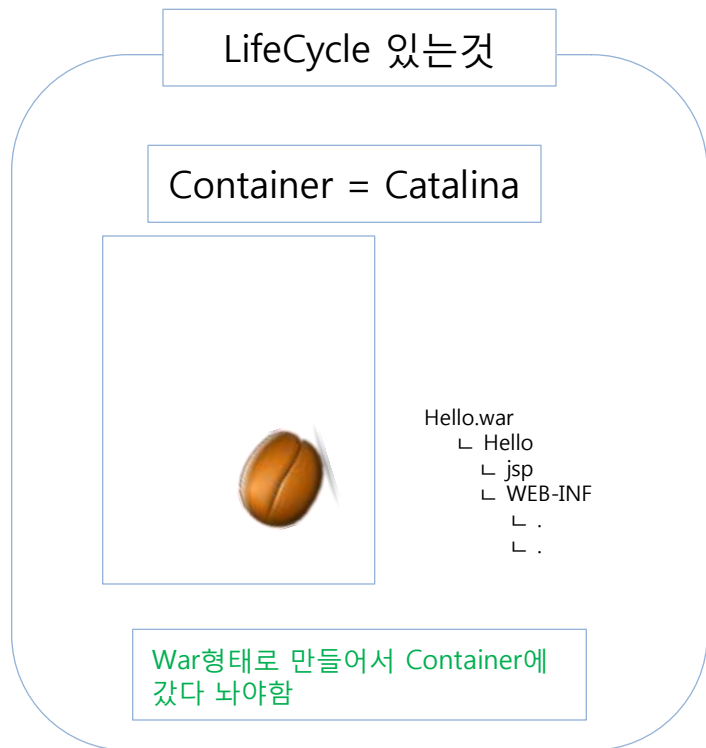
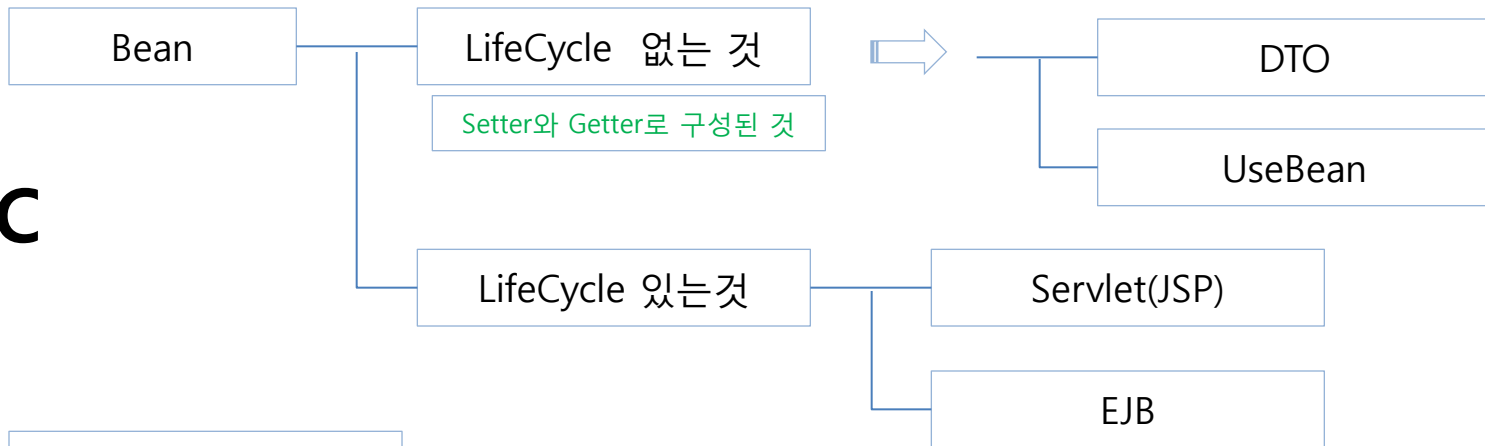
기존의 흐름



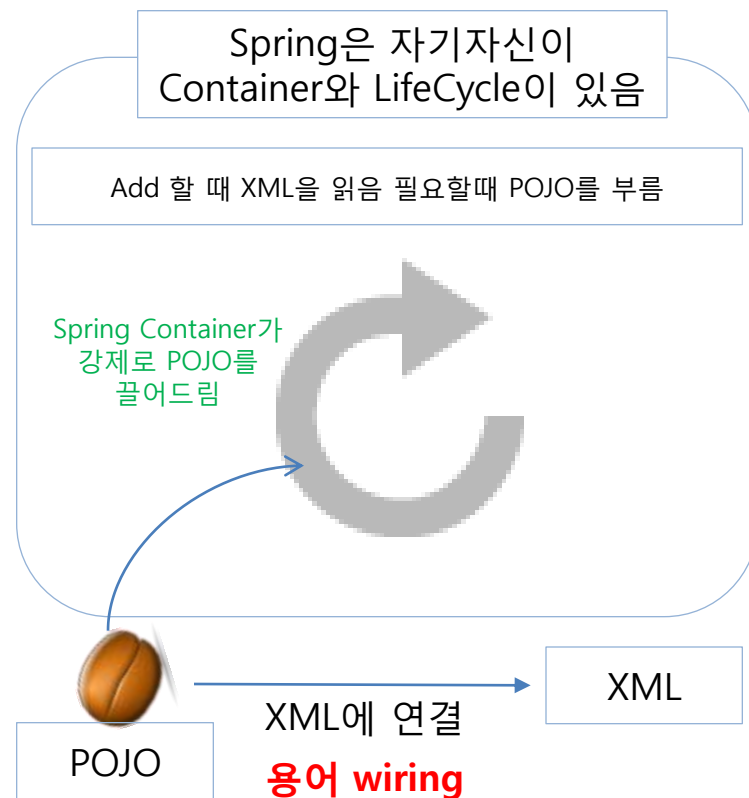
제어 역전



I.O.C



대응되는 것
IOC
(Inversion Of Control)



AOP

business만 구현
하면 됨

EJB

Enterprise = BL + 기술

AOP = CC (core Concern) + CCC (Cross Cutting Concern)
ex) JDBC 단계 ex) log, 예외처리등



wiring

O

☆

△

이미 구현해 놓음

O

◇

☆

△

Point cut(ccc가
실행 되는 지점)

OCP

인터페이스를 통해 제공되는 확장 포인트는 확장을 위해 개방되어 있고, 인터페이스를 이용하는 클래스는 자신의 변화가 불필요하게 일어나지 않도록 굳게 폐쇄되어 있다.
즉, A가 B를 의존 한다면, B를 가져다 쓰는 A는 변화가 없게 폐쇄 시키고, 공개 되어 있는 B는 언제든지 다른 기능으로 변화를 줄 수 있는 확장성이 좋게 개방시켜 두는 게 좋다.

A가 B를 의존한다

<<use>>

A
closed
Sixmagic

B
open
Oddmagic

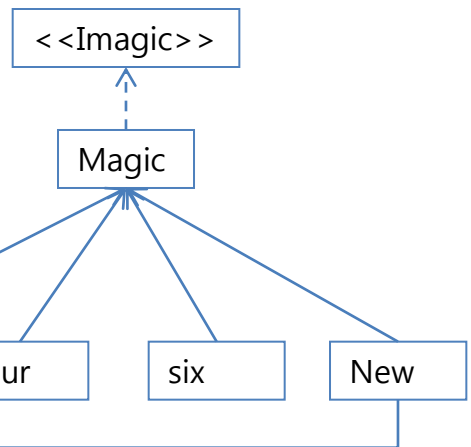
마방진 예로 들기

문제점

만약 odd마방진이 아닌 new마방진으로 바꾸게 된다면 Make()안에 로직도 다 바꿔야 한다.

```
Make(){  
    OddMagic odd=new OddMagic(n/2);  
    int [][]m=odd.make();  
}
```

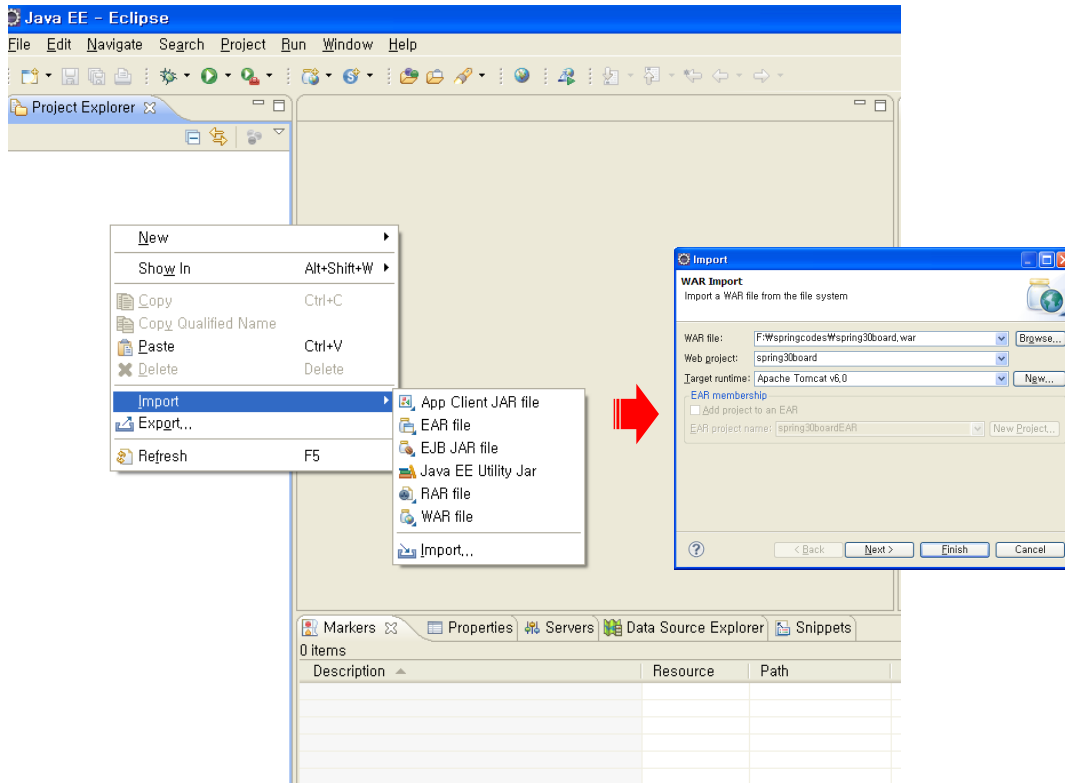
```
Make(Imagic ma){  
    아규먼트 타입은 Imagic(인터페이스로 설정해서 Imagic을 구현한 모든 타입을 받게 만든다)으로 선언하고, 안에 로직은 어떤 내용이 들어와도 쓰일 수 있게 만들어 놓는다.  
}
```



DI개념

Spring 실습

Spring30board.war → import



Spring 환경설정

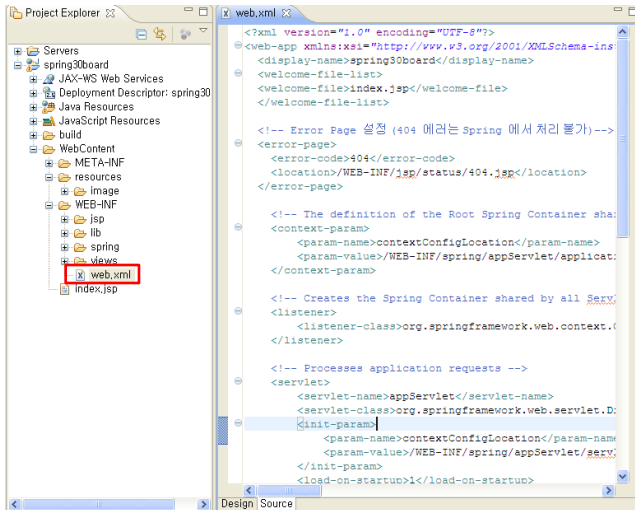
web.xml

aop-context.xml

applicationContext.xml

servlet-context.xml

Web.xml



1. <display-name> 확인

<display-name>spring30board</display-name>

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

- import 했을 경우 처음에 <display-name>을 확인하여 project 이름과 같은지 확인한다

2. <listener> 및 <context-param>

<!-- The definition of the Root Spring Container shared by all Servlets and Filters -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/appServlet/applicationContext.xml</param-value>

</context-param>

<!-- Creates the Spring Container shared by all Servlets and Filters -->

<listener>

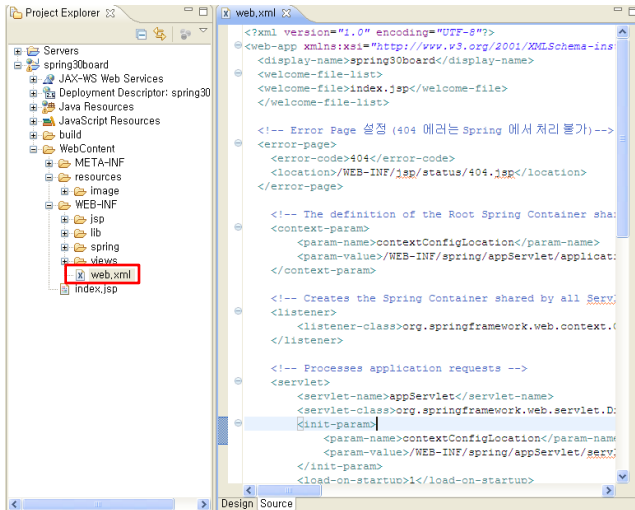
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

- ~~~. ContextLoaderListener 가 자동으로 <context-param>을 읽음

Web.xml

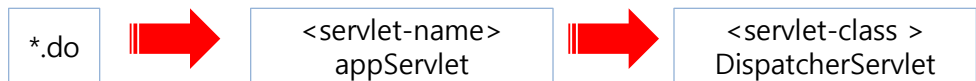
3. <servlet> & <servlet-mapping>



```
<!-- 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>/*.do</url-pattern>
</servlet-mapping>
```

- *.do 를 부르면 <servlet-name>이 appServlet을 찾는다
- <servlet-class>의 appServlet은 <servlet-class> DispatcherServlet을 실행



4. <filter> - 가로채기

<!-- [Filter Chain] Encoding Filter (EUC) -->

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

  <init-param>
    <param-name>forceEncoding</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>

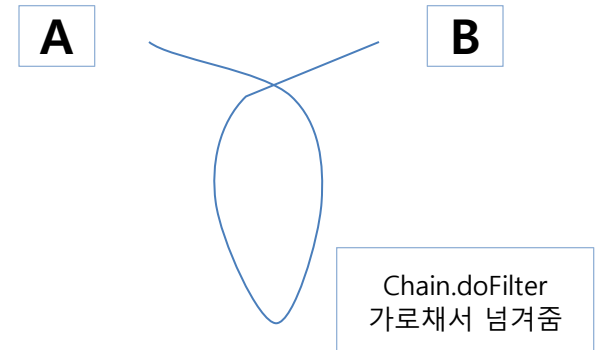
<filter-mapping>
  <filter-name>encodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

-<url-pattern>이 /* 이기 때문에 모든 url을 filter인 가로채기를 이용해 encoding을 UTF-8로 변경해 준다

Filter는 필터를 implements 했음

@overring
doFilter를 구현해야함.

doFilter(request, response);



applicationContext.xml
Model 단을 읽어 드릴 때 listener 사용

Servlet-context.xml

<annotation-driven /> // @annotation을 사용하겠다

<default-servlet-handler/>

```
<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>
```

```
<bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
<property name="definitions">
<list>
<value>/WEB-INF/views/layouts.xml</value>
</list>
</property>
</bean>
```

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolver">
<property name="requestContextAttribute" value="requestContext"/>
<property name="viewClass"
value="org.springframework.web.servlet.view.tiles2.TilesView"/>
</bean>
```

<context:component-scan base-package="com.hankyung.boards" />

페이지로 보낼 때
경로지정을
/web-inf/views/페이지명.jsp
으로 앞과 뒤에
항상 붙여서 불러준다
Tiles로 부르지 않고 이름으로만
부른다면

Tiles로 부를 때 설정한다.
layouts.xml 경로 지정하고
Tiles라이브러리 등록

com.hankyung.boards 패키지
의 경로에 @controller를 찾겠다.

applicationContext.xml

```
<!-- Properties Files -->
<bean id= "propertyConfigurer"

class= "org.springframework.beans.factory.config.PropertyPlaceholderConfigur
er">
<property name= "locations">
<list>
<value> classpath:properties/jdbc.properties </value>
</list>
</property>
</bean>

<!-- DataSource Configuration (NSG)-->
<bean id= "dataSource" class= "org.apache.commons.dbcp.BasicDataSource"
destroy-method= "close">
<property name= "driverClassName" value= "${jdbc.driverClassName}"/>
<property name= "url" value= "${jdbc.url}"/>
<property name= "username" value= "${jdbc.username}"/>
<property name= "password" value= "${jdbc.password}"/>
<property name= "initialSize" value= "${jdbc.initialSize}"/>
<property name= "maxActive" value= "${jdbc.maxActive}"/>
<property name= "minIdle" value= "${jdbc.initialSize}"/>
<property name= "maxWait" value= "3000"/>
<property name= "poolPreparedStatements" value= "true"> </property>
<property name= "maxOpenPreparedStatements" value= "50"> </property>
</bean>
```

DB연결하기
-db정보를 입력한
Properties파일을
만들어 xml에서 읽어
들이기 위한 경로 설정

Jdbc.properties

```
# Local DB
jdbc.driverClassName
=oracle.jdbc.driver.Or
acleDriver
jdbc.url=jdbc:oracle:th
in:@localhost:1521:xe
jdbc.username=hr
jdbc.password=hr
jdbc.initialSize=5
jdbc.maxActive=20
```

DB정보 읽어 들여
DataSource라고 객체를
생성하고 DataSource에
정보를 담는다

applicationContext.xml

<!-- SqlMap을 POJO로 생성한다. -->

```
<bean id="sqlMapClient"
class="org.springframework.orm.ibatis.SqlMapClientFactoryBean"
p:dataSource-ref="dataSource"
p:configLocation="/WEB-INF/sqlMap/sqlMapConfig.xml"/>
```

sqlMapConfig 객체 생성

```
<bean id="sqlMapClientTemplate"
class="org.springframework.orm.ibatis.SqlMapClientTemplate">
<property name="sqlMapClient" ref="sqlMapClient"/>
</bean>
```

Template 객체 생성-db를 연결
하기 위해 ibatis용 template 생성

<!-- TransactionManager를 POJO로 생성한다. -->

```
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
<property name="dataSource" ref="dataSource"/>
</bean>
```

@Transaction을 사용하기
위한 설정

<!-- AOP Transaction 설정 (Annotation) -->

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

aop-context.xml

```
<bean name="logAop" class="com.hankyung.boards.aop.LogExcute"/>
```

```
<!-- 로그 -->
```

```
<aop:config >
```

```
<aop:pointcut id="servicelogPoint"
```

```
expression="execution(public * com.hankyung.boards.model.*Dao*.*(..))" />
```

```
<aop:aspect id="logAop" ref="logAop" >
```

```
<aop:before method="before" pointcut-ref="servicelogPoint" />
```

```
<aop:after-returning method="afterReturning" pointcut-ref="servicelogPoint"/>
```

```
<aop:after-throwing method="daoError" pointcut-ref="servicelogPoint"/>
```

```
</aop:aspect>
```

```
</aop:config>
```

```
<!-- AOP Transaction 설정 (Annotation) -->
```

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

LogExcute.java bean 생성

패키지 안에 *Dao*.* 를 포함하는 클래스명을 가진 클래스가 실행이 된다면

Dao가 실행하기 전 before메서드를 실행시킨다.

Dao가 실행된 후 after메서드를 실행시킨다.

Error가 발생하면 daoError메서드를 실행

Index.jsp

인덱스 페이지에서 index.do를 호출할 때

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt"
prefix="fmt" %>
<%@ page session="false" %>
<fmt:requestEncoding value="UTF-8"/>
<html>
<head>
<title>Home</title>
</head>
<body>
안녕하세요
<%-- <c:redirect url='./index.do'/> --%>

<%
response.sendRedirect("./index.do");
%>
</body>
</html>
```

Index.do를 부르면 homecontroller.java의 index.do를 부른다.

HomeController.java

```
@Controller
public class HomeController {
    private static final Logger logger = LoggerFactory.getLogger(HomeController.class);

    @RequestMapping(value = "/index.do", method = RequestMethod.GET)
    public String index(Locale locale, Model model) {

        return "home";
    }
    //
```

Servlet-context.xml에서 에서 설정한
context:component-scan에 의해 @controller를 찾는다

ModelAndView-2.0
Model-3.0

페이지에서 index.do를 불렀다면
Controller에 "/index.do를 찾을 것이다

Home이라고 리턴을 했다면
Servlet-context.xml에서 설정한
*InternalResourceViewResolver*가
/web-inf/view/home.jsp
형식으로 만들어준다

HomeController.java 2

화면에 데이터 가져오기

```
@Controller
public class JHAnswerController {
    @Autowired
    private IJAnswerBoardService janswerBoardService;

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

Dao와 연결하기 위해 service와 wire해줌

Model(ModelAndView)
에 객체를 담는다

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

```
public String boardlist(Model model) {
    List<JAnswerBoard> lists=new ArrayList<JAnswerBoard>();
    lists=janswerBoardService.getAllBoards();
    model.addAttribute("lists", lists );
    return "boardlist.tiles";
} //
```

Service의 getAllBoards()를
부른다. 그러면 @service를 부르고
Service를 구현한 dao의
getAllBoards()를
호출한다.

```
@ExceptionHandler
public @ResponseBody String handle(DataAccessException e) {
    return "DataAccessException handled! "+e;
}
}
```

Tiles를 사용 할경우 *.tiles로
부른다.
만약 그냥 컨트롤로 보낼 경우
객체를 담지 않고
Redirect:list.do 와같이 작성한다.

ICustUserService.java

```
public interface ICustUserService {  
    List<CustUserDto> getAllCustUsers();  
    CustUserDto getCustUser(String id);  
    int custAdd(CustUserDto custdto);  
    int deleteCustUser(String id);  
    int deleteCustUsers(String[] ids);  
    int updateCustUser(CustUserDto  
    custdto);  
}
```

implements

CustUserService.java

```
@Service  
public class CustUserService  
implements ICustUserService {  
    @Autowired  
    private ICustUserDao custUserDao;  
  
    @Override  
    @Transactional(readOnly=true)  
    public List<CustUserDto>  
    getAllCustUsers() {  
  
        return custUserDao.getAllCustUsers();  
    }  
}
```

Wire하기

Dependency
Injection

ICustUserDao.java

```
public interface ICustUserDao {  
    List<CustUserDto> getAllCustUsers();  
    CustUserDto getCustUser(String id);  
    int custAdd(CustUserDto custdto);  
    int deleteCustUser(String id);  
    int deleteCustUsers(String[] ids);  
    int updateCustUser(CustUserDto  
    custdto);  
}
```

implements

CustUserDaoImple.java

```
@Repository  
public class CustUserDaoImple  
implements ICustUserDao {  
  
    @Autowired  
    private SqlMapClientTemplate  
    sqlMapClientTemplate;  
  
    public SqlMapClientTemplate  
    getSqlMapClientTemplate() {  
        return sqlMapClientTemplate;  
    }  
}
```

Ibtis 연결을 위한
Wire하기

Transaction 구현하기

```
@Service
public class JAnswerBoardServiceImpl
implements IJAnswerBoardService{

    @Autowired
    private IJAnswerBoard jAnswerBoard;

    @Override
    @Transactional(readOnly=true)
    public List<JAnswerBoard> getAllBoards() {
        return jAnswerBoard.getAllBoards();
    }

    @Override
    @Transactional
    public JAnswerBoard getBoard(int seq) {
        return jAnswerBoard.getBoard(seq);
    }

    @Override
    @Transactional
    public int answerBoard(JAnswerBoard dto) {
        jAnswerBoard.ansupdate(dto);
        jAnswerBoard.ansinsert(dto);
        return 1;
    }
}
```

@transactional
이라고 써주면 끝

Service에서
Dao의 메서드를
부르면 됨

```
@Repository
public class JAnswerBoardDaoImpl
implements IJAnswerBoard{

    @Autowired
    private SqlMapClientTemplate
    sqlMapClientTemplate;

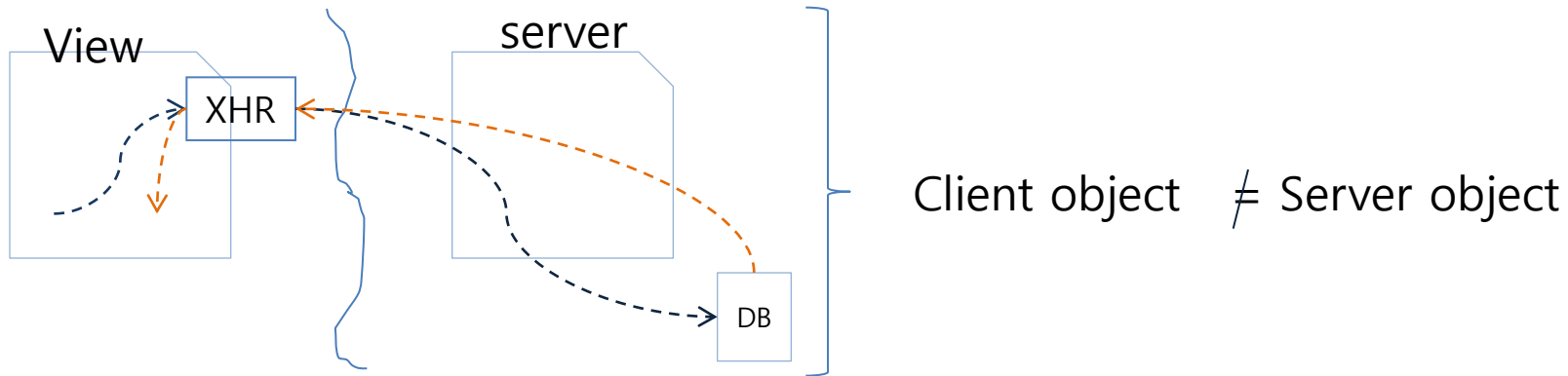
    public SqlMapClientTemplate
    getSqlMapClientTemplate(){
        return sqlMapClientTemplate;
    }

    @Override
    public int ansinsert(JAnswerBoard dto) throws
    DataAccessException{
        getSqlMapClientTemplate().insert("jAnswerMap
        .ansinsert",dto);

        return 1;
    }

    @Override
    public int ansupdate(JAnswerBoard dto)throws
    DataAccessException{
        getSqlMapClientTemplate().update("jAnswerM
        ap.ansupdate",dto);
        return 1;
    }
}
```


Ajax 개념



Round trip

Sync : 서버에서 모든 내용을 push 해준다.

- 1.open (post)
- 2.send(data)
- 3.callback

Async : 필요한 부분만 push 해준다. -> 속도가 빠르다.



Ajax 구현

Custlist.jsp

```
function showallcust(){
$.ajax({
  type: "POST",
  url: "<%=application.getContextPath()%>/cust/custlistjson.do",
  //data: "id="+id,
  async: true,
  success: function(msg){
    outputList2(msg);
  }
});
}
```

Server로 요청을 한다

Id의 이름으로 id값을 넘긴다

서버에서 값을 받아 오는데 성공하면 function을 실행한다. Msg는 넘어 온 값

responseBody는
Xml,json 형식으로 전달할때 사용된다.

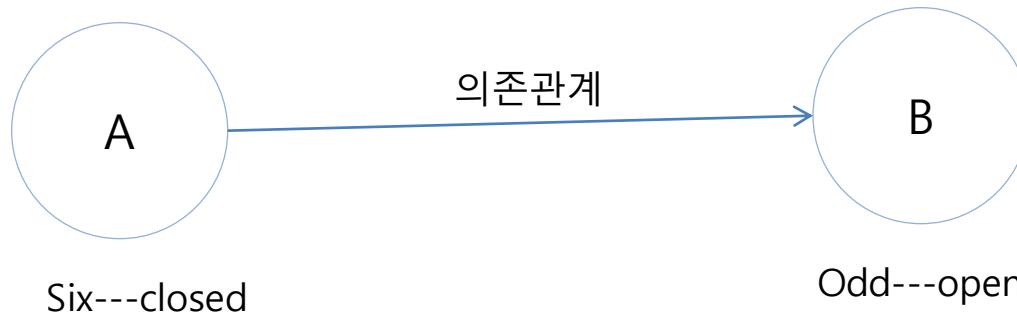
Controller.java

```
@RequestMapping(value = "/{group}/custlistjson.do", method = RequestMethod.POST)
@ResponseBody
public Map<String , List<CustUserDto>> custlistjson(Model model) {
  List<CustUserDto> lists=new ArrayList<CustUserDto>();
  lists=custUserService.getAllCustUsers();
  Map<String , List<CustUserDto>> maps=new HashMap<String, List<CustUserDto>>();
  maps.put("custs", lists);
  logger.info("Welcome home! the client custlistjson is "+ lists.toString());
  return maps;
}
```

Map에 담아서 보내면
자동으로 json형태로 보내진다.

디자인 패턴 원리

- SRP(simple Responsibility)
- OCP(open-closed)-→ DI
- LSP(Liskov substitution)
- DIP(Depend Inversion)
- ISP(Interface Segregation)



디자인 패턴 원리

-SRP(single Responsibility)

응집력을 높인다: 자신의 클래스는 자신의 목적에 필요한 기능만 구현한다.

-OCP: 의존 관계에 있는 두개의 클래스에서 의존하는 클래스는 open
필요로 하는 클래스는 closed한다.

-LSP:치환하기 (부모를 바꿔치기 할수 있게 만들어준다는 개념)

-DIP:

-ISP:

강의내용

Bean - life cycle(x)- pojo
- life cycle(o)- jsp/servlet, EJB

A.P.I.E- 추상,다형성,상속,은닉

I.O.C

Spring container 자체가 life cycle이 있어서
미리 만들어 놓은 여러 pojo 들을 와이어링 해놓고 환경변수에 따라
필요할 때 끌고 들어가 자동으로 life cycle이 실행된다.
결국 life cycle을 고려 하지 않고 코딩을 하게 되므로 기존에 고려하던 것과는
다른 제어의 역전 현상이 벌어지는데 이것이 I.O.C개념이며,
이때 강제로 pojo를 넣어 줘야 하는데 이것을 D.I 개념이라고 한다.