CHAPTER 1. 시작하기

1. 개발 환경 구축

관리의 편의를 위해 하나의 폴더를 정해서 다운 설치하는 것이 좋다.
프레임워크 사용시 폴더경로에 한글이 있을 경우 문제가 될 수 있는 경우가 있으므로 폴더명에 한글이 포함되지 않도록 한다.
예를 들면, d:₩dev 로 폴더를 지정하고 개발관련 프로그램들을 dev 폴더 안에 다운받고 설치한다.

1) JDK 다운 설치

2) Eclipse 다운 설치

=> 압축 풀고 eclipse 폴더에 있는 eclipse.ini 파일을 연다.

=> 설치한 jdk경로를 추가한다.

```
17 --launcher.appendVmargs
18 --vm
19 C:\Program Files\Java\jdk1.8.0_65\bin\javaw.exe
20 -vmargs
21 -Dosgi.requiredJavaVersion=1.7
22 -Xms256m
23 -Xmx1024m
24
```

3) Tomcat 다운 설치

4) Maven 다운 설치

라이브러리를 다운 설치받을 폴더를 명시적으로 지정해 주는 것이 좋다.
메이븐 설치 폴더 아래의 conf 폴더에 있는 settings.xml 파일을 연다.
주석 처리된 localRepository 란 부분을 찾아서 복사한다. 적당한 위치에 붙여넣기 한 다음 라이브러리들이 저장될 폴더를 지정한다.
예를 들면, d:₩dev₩apache-maven-3.5.0 폴더 밑에 repository 폴더를 만들고

<localRepository> **d:₩dev₩apache-maven-3.5.0₩repository**</localRepository>
라고 추가해 준다.

그 다음 이클립스 내에서 메이븐 설정한다.
=> window 메뉴 > Preference 선택
=> Maven > User Settings 선택
=> Browse 버튼 클릭 > 메이븐의 settings.xml 파일을 선택
=> ok

target 제외
maven을 이용하여 프로젝트를 진행하면 target이라는 폴더가 나온다. 프로젝트를 컴파일 하면 target/classes 디렉토리에 컴파일된 클래스 파일들이 생긴다. 추후 SVN(Subversion), GIT 등을 이용하여 프로젝트의 형상관리를 할 경우, 컴파일된 결과까지 공유 서버에 올라 갈 필요는 없으므로 컴파일된 결과는 제외하고 개발 소스만 올라가도록 설정하는 것이다.
=> 상단의 Window > Preferences를 선택한다.
=> Team > Ignored Resources를 선택한다.
=> Add Pattern을 누른다.
=> */target/* 을 추가한다.
=> ok

5) Spring STS 플러그인 설치
6) Maven Integration 설치
7) 데이터베이스 구축

2. 실습 프로젝트 생성

1) File > New > Other > Spring > Spring Legacy Project
> next > 프로젝트 이름 입력
> Templates 는 'Spring MVC Project' 선택
> Next > 최상위 패키지 지정 : 최소 3개 이상의 레벨로 지정할 것
> Finish

2) 프로젝트 설정 변경

STS 를 이용하여 'Spring MVC Project' 프로젝트를 생성하면 JRE 버전도 맞지 않고 서버 라이브러리도 등록되어 있지 않음 >> 설정을 수정해 주어야 함

> 프로젝트 properties 선택
> 왼쪽 Project Facets 선택 > Java 버전 1.8 또는 설치된 버전으로 수정
> 오른쪽의 Runtimes 탭 선택 > 톰캣 8.0 또는 등록된 서버로 체크함
> Apply > OK

> 왼쪽 Java Build Path > Libraries 탭 클릭 > Tomcat, JRE, Maven 등록 확인함
> Apply > OK

> pom.xml 파일의 Spring 버전을 가장 최신 버전으로 변경함
스프링 버전 확인 : http://projects.spring.io/spring-framework/

```
<properties>
	<java-version>1.8</java-version>
	<org.springframework-version>4.3.10.RELEASE</org.springframework-version>
	<org.aspectj-version>1.6.10</org.aspectj-version>
	<org.slf4j-version>1.6.6</org.slf4j-version>
</properties>
```

pom.xml 을 수정하고 나서 어느 정도 시간이 지나면, Project Explorer 뷰에서 Java Resources / Libreries / Maven Dependencies 에 스프링 버전이 일괄적으로 변경된 것을 확인할 수 있음.

3) 서버에 프로젝트 추가하고 실행 확인함

CHAPTER 2. 스프링 MVC 구조

1. **스프링 MVC 수행 흐름**

1) 클라이언트로부터의 모든 ".do" 요청을 DispatcherServlet 이 받는다.
(프로젝트명/src/main/sebapp/WEB-INF/spring/appServlet/servlet-context.xml)

2) DispatcherServlet 은 내장된 HandlerMapping 기능을 통해 요청을 처리할 Controller 를 검색(component-scan)한다.

3) DispatcherServlet 은 검색된 Controller 를 실행하여 클라이언트의 요청을 처리한다.

4) Controller 는 전달받은 클라이언트 데이터를 추출하고, 서비스의 메소드로 전달한 다음 비즈니스 로직의 수행 결과로 리턴된 정보를 Model 객체에 저장하고 Model을 보여줄 View 정보를 ModelAndView 객체에 저장 또는 뷰파일명을 DispatcherServlet 으로 리턴한다.

5) DispatcherServlet 은 ModelAndView 로부터 View 정보를 추출하거나 리턴된 뷰파일명을 내장된 ViewResolver 를 이용하여 응답할 View를 지정된 뷰 폴더 아래에서 찾는다.

6) DispatcherServlet은 ViewResolver를 통해 찾아낸 View를 실행하여 응답을 전송한다.

7) 응답된 뷰페이지는 jsp, el, jstl, ajax, javascript, html 을 이용해 브라우저에 결과를 출력한다.

2. **DispatcherServlet 등록 및 스프링 컨테이너 구동**

WEB-INF/web.xml 파일에 등록된 정보를 수정한다.
DispatcherServlet 파일명을 설정한 내용에 맞춰 action-servlet.xml 로 수정하거나 있는 그대로 사용해도 된다.
참고로 DispatcherServlet 파일은 여러 개를 작성할 수 있으며, web.xml 에 등록하면 된다.
다음은 폴더명과 파일명을 수정한 경우의 예이다.

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/config/root-context.xml</param-value>
</context-param>
```

```
<servlet>
	<servlet-name>action</servlet-name>
	<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
		<init-param>
			<param-name>contextConfigLocation</param-name>
			<param-value>/WEB-INF/config/*-servlet.xml</param-value>
		</init-param>
		<load-on-startup>1</load-on-startup>
</servlet>

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

CHAPTER 3. 어노테이션 기반 MVC 개발

스프링 설정 파일에 HandlerMapping, Controller, ViewResolver 같은 여러 클래스를 등록해야 하는 데 어노테이션을 활용하면 xml 설정파일 안에서의 <bean> 등록을 줄일 수 있다.

1. 어노테이션 관련 설정

DispatcherServlet 에 <annotation-driven /> 등록 확인함.

그 다음 Controller 클래스가 검색 범위에 포함되도록 하고자 <context:component-scan> 엘리먼트의 base-package 속성에 Controller 클래스들이 있는 가장 상위 패키지를 등록한다.

<context:component-scan base-package="com.kh.hello//controller" />**

또는

<context:component-scan base-package="com.kh.hello" />

2. **@Controller 사용하기**

기존의 스프링 컨테이너가 Controller 클래스를 생성하려면 Controller 클래스들을 스프링 설정 파일(DispatcherServlet)에 <bean> 등록해야 했다. 어노테이션 방식으로 진행할 경우에는 컨트롤러 클래스들을 일일이 <bean> 으로 등록하지 않고 클래스 선언부 위에 "@Controller" 를 붙이면 된다.

스프링 설정 파일에 작성된 <context:component-scan> 이 컨트롤러 객체들을 자동으로 생성해 준다.

Ctrl + Shift + O 키를 눌러 import를 자동 추가한다.

3. **@RequestMapping 사용하기**

클라이언트의 "login.do" 요청에 대해 loginMethod() 가 실행되도록 하려면 @RequestMapping 을 이용하여 HandlerMapping 설정을 대체한다.

HandlerMapping 은 서블릿에서의 @WebServlet("login.do") 즉, 서블릿의 url-pattern 을 의미한다.

@RequestMapping 은 메소드 위에 설정한다. value 속성은 대부분 생략한다.

```
@RequestMapping(value="/login.do", method=RequestMethod.POST)
public ModelAndView loginMethod(){
    return null;
}
```

4. **클라이언트 요청 처리**

대부분 Controller 는 사용자의 입력 정보를 추출하여 VO(Value Object) 에 저장한다. 그리고 비즈니스 컴포넌트의 메소드를 호출할 때 VO 를 인자로 전달한다. 사용자 입력 정보는 HttpSerevletRequest 의 getParameter( ) 메소드를 사용하여 추출한다. 그런데 사용자 입력 정보가 많을 경우와 입력 정보가 변경될 때마다 Controller 의 추출 코드가 수정되어야 할 것이다.

Command 객체를 이용하면 이런 문제를 해결할 수 있다. Command 객체는 Controller 의 메소드 매개변수로 받은 VO 객체라고 보면 된다.

```
@RequestMapping("/login.do")
public ModelAndView loginMethod(MemberVO vo){
        return null;
}
```

스프링 컨테이너가 해당 메소드를 실행할 때 Command 객체를 생성하고 사용자가 입력한 값들을 Command 객체에 셋팅까지 해서 넘겨준다. 즉, 사용자 입력 정보 추출과 VO 객체 생성, 추출값 셋팅 모두를 스프링 컨테이너가 자동으로 처리한다는 것이다.

여기서 중요한 것은 FORM 태그 안의 **INPUT 태그의 NAME 속성의 이름과 VO 객체의 필드명이 반드시 일치**해야 한다. 그리고 각 필드의 Setter 메소드가 반드시 존재해야 한다.

5. 비즈니스 컴포넌트 사용

Spring IoC 에서의 비즈니스 컴포넌트는 VO 클래스, DAO 클래스, Service 인터페이스, Service 구현 클래스 4개의 파일로 구성되어 있다. 그리고 Controller 는 DAO 를 직접 이용해서는 안되며 반드시 비즈니스 컴포넌트를 이용해야 한다.
이유는 비즈니스 컴포넌트에 다형성을 적용하여 클라이언트가 인터페이스를 통해서 비즈니스 컴포넌트를 이용하면 컴포넌트의 구현 클래스(인터페이스 후손 클래스)를 수정하거나, 다른 구현 클래스로 대체해도 이를 사용하는 클라이언트는 수정하지 않아도 되기 때문이다. 즉 비즈니스 컴포넌트가 수정되더라도 이를 사용하는 Controller 는 수정하지 않아도 되게 하려는 것이다.
Service 인터페이스를 작성하고 나서, Service 인터페이스를 상속받은 Service 구현 클래스 이름 위에 @Service("서비스이름") 어노테이션을 표시한다.

서비스 인터페이스 구현은 다음과 같다.

```
package com.kh.hello.member.service;

public interface MemberService {  추상메소드 선언  }
```

서비스 인터페이스 구현체 작성은 다음과 같다.

```
package com.kh.hello.member.service;

import org.springframework.stereotype.Service;

@Service("memberService")
public class MemberServiceImpl implements MemberService {
        //DAO 사용
        @Autowired
        MemberDAO memberDAO;

        @Override
        Public …… 상속받은 서비스 메소드 오버라이딩함.
        DAO 클래스의 사용 메소드로 vo 객체 넘기고 받은 결과 리턴함
}
```

DAO 클래스 구현은 다음과 같다.

```
@Repository("memberDAO")
Public class MemberDAO{

//마이바티스 스프링 연동 모듈이 제공하는 클래스 사용
@Autowired
private SqlSessionTemplate sqlSession;

//각 서비스별 메소드 구현
}
```

컨트롤러는 서비스를 사용한다. @AutoWired 어노테이션을 사용하여 MemberService 타입의 MemberServiceImpl 객체가 의존성 주입된다.

```
@Controller
public class MemberController {

        @Autowired
        private MemberService memberService;
```

비즈니스 컴포넌트 의존성 주입하기

DispatcherServlet 에서 **<context:component-scan base-package="com.kh.hello" />** 로 설정하지 않고

<context:component-scan base-package="com.kh.hello//controller" />**

으로 설정했을 경우에는 실행하면 에러가 발생할 것이다. 이 에러 메시지는 Controller 가 @Autowired 로 의존성 주입을 하려고 하는 데 Service 타입의 객체가 메모리에 없어서 의존성 주입을 할 수 없다는 에러 메시지이다.

@Autowired 어노테이션을 사용하려면 의존성 주입 대상이 되는 객체가 반드시 메모리에 먼저 로딩되어 있어야 한다.

결론은 Controller 보다 의존성 주입이 될 ServiceImpl 객체가 먼저 생성되어 있어야 하는 것이다. 그런데 DispatcherServlet (즉, action-servlet.xml) 파일에는 Controller 객체들만 컴포넌트 스캔하도록 설정했기 때문에 ServiceImpl 객체는 생성되지 않는다.

결국 Controller 보다 의존성 주입 대상이 되는 비즈니스 컴포넌트를 먼저 생성하려면 비즈니스 컴포넌트를 먼저 생성하는 또 다른 스프링 컨테이너가 필요하다. 그리고 이 컨테이너를 서블릿 컨테이너(DispatcherServlet) 보다 먼저 구동하면 된다.

src/main/resources 폴더에 비즈니스 레이어에 해당하는 설정 파일로 root-context.xml(파일명은 임의대로 설정 가능) 파일을 두고, 이 파일을 읽어 비즈니스 컴포넌트들을 메모리에 생성되게 해야 한다. 이 때 사용하는 클래스가 스프링에서 제공하는 ContextLoaderListener 이다.

web.xml 에 <listener> 태그로 등록한다.

중요한 것은 서블릿 컨테이너가 web.xml 파일을 읽어서 구동될 때 ContextLoaderListener 클래스가 자동으로 메모리에 생성된다.

즉 ContextLoaderListener 는 클라이언트의 요청이 없어도 컨테이너가 구동될 때 Pre-Loading 되는 객체이다. 그러므로 roo-context.xml 에 설정된 내용이 먼저 구동된다는 것이다.

web.xml 설정 추가

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

```
-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:root-context.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>
```

root-context.xml 파일 설정

NameSpace 에서 context 체크한 다음, 다음의 내용을 추가한다.

```
<context:component-scan base-package="com.kh.hello.**.service">
    </context:component-scan>
<context:component-scan base-package="com.kh.hello.**.dao">
    </context:component-scan>
```

CHAPTER 4. 스프링과 MyBatis 연동

1. **라이브러리 내려받기**

   DB 연동을 위한 Oracle 드라이버와 MyBatis 를 내려 받기 위해, pom.xml 파일에 <dependency> 엘리먼트 추가한다.

   참조 사이트 : https://mvnrepository.com/

```
<repository>
                <id>oracle</id>
                <name>ORACLE JDBC Repository</name>
                <url>https://maven.atlassian.com/3rdparty/</url>
</repository>
```

추가하고,

```
<!-- MyBatis -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.4.5</version>
        </dependency>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>1.3.1</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/com.oracle/ojdbc6 -->
        <dependency>
            <groupId>com.oracle</groupId>
            <artifactId>ojdbc6</artifactId>
            <version>11.2.0.4.0-atlassian-hosted</version>
        </dependency>

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

        <!-- DBCP -->
        <dependency>
            <groupId>commons-dbcp</groupId>
            <artifactId>commons-dbcp</artifactId>
            <version>1.4</version>
        </dependency>
```

MyBatis는 스프링 쪽에서 연동에 필요한 API 를 제공하지 않으며 MyBatis 쪽에서 스프링 연동에 필요한 API 를 제공한다. 따라서 스프링과 MyBatis 를 연동하려면 MyBatis 에서 제공하는 다음의 클래스들을 이용해서 연동해야 한다.

org.mybatis.spring.SqlSessionFactoryBean

org.mybatis.spring.SqlSessionTemplate

mybatis-3.4.5.jar 파일은 순수 Mybatis 관련 라이브러리이고, mybatis-spring-1.3.1.jar 파일은 Mybatis 와 스프링을 연동하기 위해 사용하는 라이브러리이다.

2. **MyBatis 설정 파일 복사 및 수정**

src/main/resource 폴더 아래에 mappers 폴더를 만들고 XXX-Mapper.xml 파일을 복사해 넣는다.

src/main/resource 폴더에 mybatis-config.xml 파일을 복사해 넣는다.

mybatis-config.xml 파일을 열어서 다음의 항목들을 삭제한다.

```
<!-- Properties 파일 설정 -->
    <properties resource="db.properties"/>

<!-- DataSource 설정 -->
 <environments default="development">
       <environment id="development">
       <transactionManager type="JDBC"/>
       <dataSource type="POOLED">
         <property name="driver" value="${jdbc.driverClassName}"/>
         <property name="url" value="${jdbc.url}" />
         <property name="username" value="${jdbc.username}"/>
         <property name="password" value="${jdbc.password}"/>
       </dataSource>
       </environment>
 </environments>
```

데이터 소스는 DB 연동뿐 아니라 트랜잭션 처럼 다양한 곳에서 사용할 수 있으므로 MyBatis 설정이 아닌 스프링 설정 파일에서 제공하는 것이 맞다. Mybatis-config.xml 파일의 완성된 내용은 다음과 같다.

```
<configuration>

   <!-- Alias 설정 -->
   <typeAliases>
      <typeAlias alias="member" type="com.kh.member.vo.MemberVO"/>
 <typeAlias alias="board" type="com.kh.board.vo.BoardVO"/>
   </typeAliases>
```

```
    <!-- Sql Mapper 설정 -->
    <mappers>
        <mapper resource="mappers/board-mapper.xml"/>
        <mapper resource="mappers/member-mapper.xml"/>
    </mappers>
</configuration>
```

3. **스프링 연동 설정**

우선 스프링 설정 파일(root-context.xml) 파일에 DataSource 와 SqlSessionFactoryBean 클래스와 SqlSessionTemplate 클래스를 Bean 등록해야 한다.

```
<!-- DataSource 등록 -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
        <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
        <property name="username" value="student" />
        <property name="password" value="student" />
    </bean>

    <bean id="sqlSession"
            class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="configLocation"
            value="classpath:mybatis-config.xml">
    </property>
    <property name="dataSource" ref="dataSource"></property>
    </bean>

    <bean class="org.mybatis.spring.SqlSessionTemplate">
            <constructor-arg ref="sqlSession"></constructor-arg>
    </bean>
```

DAO 클래스 구현

dao 클래스를 구현할 때 SqlSessionTemplate 객체를 @Autowired 를 이용하여 의존성 주입하면, SqlSessionTemplate 객체로 DB 연동 로직을 처리할 수 있다.

```
package com.kh.tests.board.dao;

import java.util.List;

import org.mybatis.spring.SqlSessionTemplate;
import  org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import com.kh.tests.board.vo.Board;

@Repository("boardDao")
public class  BoardDao {

	@Autowired
	private SqlSessionTemplate sqlSession;

	public List<Board> selectList(){
		return (List<Board>)  sqlSession.selectList("boardList");
	}
}
```

4. **MyBatis 연동 테스트**

BoardDao 를 의존성 주입할 수 있도록 다음과 같이 구현한다.

```
package com.kh.tests.board.service;

import java.util.List;
import com.kh.tests.board.vo.Board;

public interface BoardService {
	List<Board> selectBoardList();
}
```

```
package com.kh.tests.board.service;

import java.util.List;

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

import com.kh.tests.board.dao.BoardDao;
import com.kh.tests.board.vo.Board;

@Service("boardService")
public class BoardServiceImpl implements BoardService{

	@Autowired
	private BoardDao bDao;

	@Override
	public List<Board> selectBoardList() {
		return bDao.selectList();
	}

}
```

CHAPTER 5. 서버 연결, 구동 확인

CHAPTER 6. 컨트롤러에서 사용할 수 있는 매개변수와 반환타입

1. Controller 의 리턴 타입

컨트롤러에서 메소드를 정의할 때, 리턴타입은 개발자 마음대로 결정할 수 있다. 일반적으로 ModelAndView 와 String 을 리턴타입으로 많이 사용하게 된다. 그렇지만 프로젝트내에서의 코드의 일관성을 고려해서 메소드의 리턴 타입은 매번 다르게 지정하는 것보다는 하나로 통일하여 사용하는 것을 권장하며, 주로 String으로 통일한다.

코드는 아래와 같이 구현한다.

```
public ModelAndView login(UserVO vo, ModelAndView mav) {
        mav.setViewName("boardList");
        return mav;
 }

 public String login(UserVO vo) {
        return "boardList";
}
```

2. 컨트롤러에서 메소드 매개변수로 사용할 수 있는 타입 또는 어노테이션

참고로 스프링에서 컨트롤러의 메소드 매개변수로 선언된 객체는 자동 생성된다. (의존성 주입 : Dependancy Injection)

① HttpServletRequest 와 HttpServletResponse 객체

```
public String login(HttpServletRequest request,
HttpServletResponse response) {
      String userid = request.getParameter("userid");
     ……..
      return "뷰파일명";
     }
```

② 위의 내용과 동일하게 처리할 수 있는 @RequestParam 어노테이션

```
public String login(@RequestParam(value="userid") String userid) {
        ……..
       return "뷰파일명"; }
```

@RequestParam 에 사용되는 속성은 다음과 같다.

@RequestParam(value="뷰로부터 전달된 파라미터 이름",
defaultValue="뷰로부터 전달된 값이 없을 때 사용할 기본값),
required=false/true) ← 파라미터 생략 여부

③ **Command 객체**

스프링에서 뷰페이지의 입력양식의 이름과 VO 클래스의 필드명을 똑같이 만들면, 컨트롤러 메소드 매개변수에서 자동으로 파라미터로 전달된 값을 VO 객체에 대입 처리해 준다.

메소드 매개변수로 선언된 VO 객체를 COMMAND 객체라고 한다.

예를 들어

User.java 에서의 필드 선언이 아래와 같고

```
public class User{
   private String userid;
   private String userpwd;
   private String username;

   public User(){}

   public User(String userid, String userpwd, String username){…….}
}
```

userInsert.html 파일에서의 입력 양식이 아래와 같다면

```
<form action="uinsert" method="post">
   아이디 : <input type="text" name="userid"> <br>
   암 호 : <input type="password" name="userpwd"> <br>
   이 름 : <input type="text" name="username"><br>
   <input type="submit" value="가입하기">
</form>
```

UserController.java 에서 회원 가입 서비스 요청을 전달받을 때, Command 객체로 처리한다면

```
public class UserController{
   @RequestMapping("/uinsert")
   public String userInsert(User user){
```

```
        ……
        return “home”;
    }
}
```

자동으로 다음과 같은 처리가 된 것이다.

```
User user = new User(request.getParameter(“userid”),
                     request.getParameter(“userpwd”),
                     request.getParameter(“username”));
```

주의할 점은 입력양식의 이름과 VO 클래스의 필드명이 반드시 같아야 한다는 것이다.

④ **@ModelAttribute 어노테이션**

이 어노테이션은 Controller 메소드의 매개변수로 선언된 Command 객체의 이름을 변경할 때 사용할 수 있다.

```
public class UserController{
    @RequestMapping(“/uinsert”)
    public String userInsert(@ModelAttribute(“user”) User vo){
        ……
        return “home”;
    }
}
```

또는 뷰로 전달할 객체 정보를 설정(set)할 목적으로도 사용할 수 있다.
다시 말해, request.setAttribute(“user”, user); 의 용도로도 사용할 수도 있다는 것이다.

```
@RequestMapping(“/uinsert”)
@ModelAttribute(“user”)
    public String userInsert(User vo){
        User user = new UserService(vo);
        return “home”;
    }
```

home.jsp파일에서 user 객체의 정보를 ${user.userid} 이렇게 사용할 수 있게 된다.

@ModelAttribute 가 설정된 메소드는 @RequestMapping 어노테이션이 적용된 메소드보다 먼저 호출되며, @ModelAttribute 메소드 실행 결과로 리턴된 객체를 자동으로 Model에 저장하게 된다.

⑤ **@SessionAttribute 어노테이션**

이 어노테이션은 세션 객체에 정보를 저장하거나, 저장된 정보를 컨트롤러 메소드에서 사용하려고 할 때 유용한 어노테이션이다.

```
@Controller
@SessionAttributes(“user”)
public class UserController{
   @RequestMapping(“/uinsert”)
   public String userInsert(@ModelAttribute(“user”) User vo, Model
model){
       model.addAttribute(“user”, userService.getUser(vo));
       return “home”;
   }
}
```

위의 코드는 Model에 “user”라는 이름으로 저장되는 데이터가 있다면, 그 데이터를 세션에도 자동으로 저장하라는 설정이다.

또한 @ModelAttribute(“user”) 는 세션에 user 라는 이름의 객체 정보가 있으면 꺼내와서 vo 객체에 대입하라는 뜻이기도 하다.

CHAPTER 7. 스프링에서의 로깅(logging) 처리

시스템을 작동할 때 시스템의 작동상태의 기록/보존, 이용자의 습성조사 및 시스템 동작의 분석 등을 하기 위해 작동 중의 각종 정보를 기록하여 둘 필요가 있다.

이 기록을 만드는 것을 로깅이라 한다. 또 기록 자체를 로그(log)라고 하며, 로그에는 일반적으로 다음과 같은 것이 있다.

(1) 작동로그

: 시스템의 운전상태를 기록/보존하기 위한 센터 오퍼레이터에게 통지되는 조작문, 지령문, 보고문(이들을 일반 로그라고 한다) 및 하드웨어 장해정보(장해로그) 등의 기록, 작

동로그를 조사함으로써 하드웨어 장해상황, 콘솔 조작상황 등을 파악할 수 있다.

(2) 통계로그

: 이용자의 습성조사, 시스템의 효율화를 목적으로 한 동작의 분석 및 시스템설계의 기초 자료를 얻기 위한 CPU/메모리 사용 효율, 회선 트래픽(traphic), 단말입출력 데이터량, 회선접속/절단시각 등의 기록

<스프링/마이바티스 로깅>

- 스프링/마이바티스는 내부 로그 팩토리를 사용하여 로깅 정보를 제공한다.
  내부 로그 팩토리는 로깅 정보를 다른 로그 구현체 중 하나에 전달한다.

    * SLF4J
    * Apache Commons Logging
    * Log4j 2
    * Log4j
    * JDK logging

- 로깅 솔루션은 내부 스프링/마이바티스 로그 팩토리의 런타임 체크를 통해 선택된다.

- 스프링/마이바티스 로그 팩토리는 가능하면 첫번째 구현체를 사용할 것이다
  (위 로깅 구현체의 나열 순서는 내부적으로 선택하는 우선순위이다).
  만약 스프링/마이바티스가 위 구현체중 하나도 찾지 못한다면, 로깅을 하지 않을 것이다.

- Tomcat 또는 WebSphere 등의 애플리케이션 서버 환경에서는 클래스패스의 일부로 JCL 을 사용한다. 그러면 스프링/마이바티스는 로깅 구현체로 JCL을 사용할 것이다.

- WebSphere 환경에서 Log4J 설정은 무시된다.
  WebSphere 는 자체 JCL 구현체를 제공한다.

이 환경에서의 스프링/마이바티스는 Log4J 설정을 무시하고 JCL을 사용한다.

- 만약 애플리케이션이 클래스패스에 JCL을 포함한 환경에서 다른 로깅 구현체 중 하나를 더 선호한다면, 다음의 메서드 중 하나를 호출하여 다른 로깅 구현체를 선택할 수 있다.

* org.apache.ibatis.logging.LogFactory.useSlf4jLogging();
* org.apache.ibatis.logging.LogFactory.useLog4JLogging();
* org.apache.ibatis.logging.LogFactory.useJdkLogging();
* org.apache.ibatis.logging.LogFactory.useCommonsLogging();
* org.apache.ibatis.logging.LogFactory.useStdOutLogging();

=>> 스프링/마이바티스가 메서드를 호출하기 전에 위 메서드 중 하나를 호출해야 한다. 이 메서드들은 런타임 클래스패스에 구현체가 존재하면 그 로그 구현체를 사용하게 한다. 예를 들어, Log4j 로깅을 선택했지만 런타임에 Log4j 구현체가 클래스패스에 없다면, 마이바티스는 Log4j 구현체의 사용을 무시하고 다른 로깅 구현체를 찾아 다시 사용할 것이다.

- 참조 사이트

* SLF4J
  http://www.slf4j.org/
* Apache Commons Logging
  http://commons.apache.org/proper/commons-logging/
* Apache Log4j
  http://logging.apache.org/log4j/2.x/
* JDK Logging API
  http://www.oracle.com/technetwork/java/index.html

<LOG4j의 Logging 설정(Configuration)>

- LOG4j를 이용해서 스프링/마이바티스의 로깅 구문을 출력하기 위해서는 하나 이상의 설정파일(log4j.properties)과 몇 개의 새로운 jar파일(log4j.jar)이 필요하다.

- 1단계 : log4j.jar 파일 추가하기

아래 URL에서 Log4J를 다운로드한다.

http://logging.apache.org/log4j/2.x/

애플리케이션에 WEB-INF/lib 폴더에 log4j.jar 파일을 추가한다.

- 2단계 : Log4j 프로퍼티 설정하기

일반적으로 log4j.properties 파일은 클래스패스에 저장하여 사용한다.

Log4j는 3개의 element 제공

* logger : 로깅 메시지를 appender에 전달한다.
* appender.* : 로깅 메시지의 출력 방법을 지정한다.
* layout : 출력 형식을 지정한다.

=> appender에 메시지 수준에 대한 레벨을 기술한다.

* FATAL : 가장 치명적인 오류 메시지
* ERROR : 일반적인 오류 메시지
* WARN : 경고성 오류 발생 메시지
* INFO : 일반적인 정보 메시지
* DEBUG : 상세 정보 메시지
* TRACE : 추적 정보 메시지

=> layout에 사용되는 출력 형식요소

* %p : debug, info, error, fatal과 같은 레벨의 순위 출력
* %m : 로그 내용 출력
* %d : 로깅 이벤트가 발생한 시간을 기록

* %t : 로그 이벤트가 발생된 쓰래드의 이름 출력
* %n : 플랫폼 종속적인 개행 문자 출력. ₩r₩n 또는 ₩n
* %c : 카테고리 표시
* %C : 클래스명 표시
* %F : 로깅이 발생한 프로그램 파일명 표시
* %I : 로깅이 발생한 정보 표시
* %L : 로깅이 발생한 라인수 표시
* %M : 로깅이 발생한 메서드명 표시

=> 전역적인 로깅 처리 구문 예

```
#Global logging configuration -- 주석
log4j.rootLogger=ERROR, stdout
-- stdout appender에 에러 메시지 출력
```

=> 마이바티스의 매퍼 로깅 처리 구문 예

: 매퍼 파일 전체의 SQL 구문의 결과 출력

```
#MyBatis logging configuration
log4j.logger.dept.dao.DeptMapper=TRACE
```

: 전체 매퍼 대신에 매퍼 XML에서
"id"의 네임스페이스가 "selectDepartment"의 SQL 구문만 결과 출력

```
#MyBatis logging configuration
log4j.logger.dept.dao.DeptMapper.selectDepartment=TRACE
```

: 쿼리의 결과가 아닌 SQL문을 보고자 할 때는 DEBUG 로 지정

```
log4j.logger.dept.dao=DEBUG
```

<log4j.properties 파일 예>

```
# Global logging configuration
```

```
log4j.rootLogger=ERROR, stdout
# MyBatis logging configuration...
log4j.logger.com.example.BlogMapper=TRACE
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```