1- JUnit을 이용한 테스트 진행 방법

1. pom.xml 수정

- Spring Framework의 버전을 사용하고자 하는 버전으로 변경

```
<properties>
	<java-version>1.8</java-version>
	<org.springframework-version>5.2.8.RELEASE</org.springframework-version>
```

- JUnit <dependency>를 찾아 버전을 4.7 -> 4.12로 변경

**** Junit 은 4.12 버전 또는 그 이상이어야 사용할 수 있음.**

2. mvnrepository 에서 spring-test을 검색하여 사용 중인 Spring Framework 버전과 같은 버전을 pom.xml에 추가

```
<dependency>
	<groupId>junit</groupId>
	<artifactId>junit</artifactId>
	<version>4.12</version>
	<scope>test</scope>
</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-test -->
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-test</artifactId>
	<version>${org.springframework-version} </version>
	<scope>test</scope>
</dependency>
```

*** maven plugin 의 자바 버전도 수정**

```
<plugin>
		<groupId>org.apache.maven.plugins</groupId>
		<artifactId>maven-compiler-plugin</artifactId>
		<version>2.5.1</version>
		<configuration>
```

```
                <source>1.8</source>
                <target>1.8</target>
                <compilerArgument>-Xlint:all</compilerArgument>
                <showWarnings>true</showWarnings>
                <showDeprecation>true</showDeprecation>
            </configuration>
        </plugin>
```

3. test 진행 시 was를 이용한 배포 없이 진행을 해야 함.

was를 이용하지 않으면 web.xml과 내부에서 호출되는 설정관련 xml 파일이 로딩이 되지 않음

➔ **이를 해결하기 위하여 @WebAppConfigration 어노테이션을 사용함**

*** @WebAppConfigration**

- Controller및 web환경에 사용되는 bean들을 자동으로 생성하여 등록하게 됨.
- 해당 어노테이션을 사용하기 위해서는 **servlet의 버전이 3.1.0 이여만 가능**

  -> spring mvc project 생성 시 servlet의 버전은 2.5로 설정되어 있음

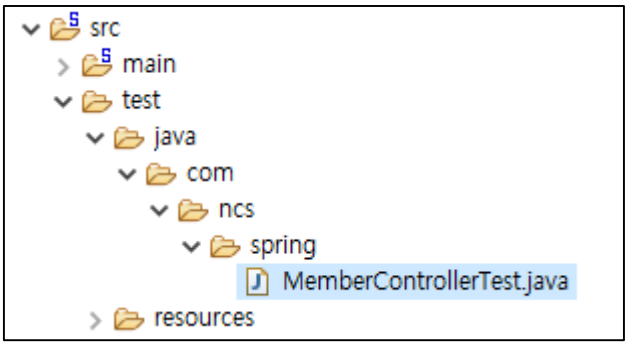
- pom.xml -> artifactId가 servlet-api 인 dependency를 주석처리
- mvnrepository 에서 **javax.servlet-api** 검색 -> 3.1버전 추가
-

```
<!-- Servlet 버전을 3.1 로 변경 -->

<!-- <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.5</version>
        <scope>provided</scope>
</dependency> -->

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scop>
</dependency>

<dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.1</version>
        <scope>provided</scope>
</dependency>
```

4. 구현된 기능의 테스트를 위한 테스트용 Class 생성

- **src/test/java** 패키지 하위에 테스트를 위한 Class 생성

ex) MemberContollerTest.java



5. 생성한 테스트용 Class에 다음 어노테이션을 추가

@RunWith(SpringJUnit4ClassRunner.class)

// JUnit에 내장된 Runner대신 그 클래스를 실행.

// SpringJUnit4ClassRunner는 스프링 테스트를 위한 Runner

@WebAppConfiguration

// Controller및 web환경에 사용되는 bean들을 자동으로 생성하여 등록

@ContextConfiguration(locations={"file:추가할 설정 파일의 경로"}),

// 자동으로 만들어줄 애플리케이션 컨텍스트의 설정 파일 경로를 지정

예> @ContextConfiguration(locations = "/root-context.xml")

//테스트에서의 루트는 "src/test/resources 임.

ex) MemberContollerTest.java에 어노테이션 적용

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(locations={"file:src/main/webapp/WEB-INF/spring/root-context.xml", "file:src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml"})
```

- root-context.xml 파일은 Spring 프로젝트 배포 시 프로젝트 관련 설정(DB연결, 파일 업로드 등) 내용을 작성해 주는 파일로 반드시 필요함!!!
- servlet-context.xml 파일은 Dispatcher Servlet 역할을 하는 파일로 Handler Mapping, View Resolver 역할 수행을 위한 파일로 반드시 필요함!!!

6. 테스트용 Class 에서 사용될 필드 선언

@Autowired

private WebApplicationContext wac;

// 현재 실행중인 애플리케이션의 구성을 제공하는 인터페이스

private MockMvc mockMvc;

// client 요청 내용을 controller에서 받아 처리하는 것과 같은 테스트를 진행할 수 있는 클래스.

7. 테스트 수행 전 테스트 진행에 사용될 MockMvc 객체 생성 메소드 setup() 작성

@Before // JUnit 테스트 진행 전 먼저 실행하는 것을 지정하는 어노테이션

public void setup() {

this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();

}

8. 테스트 내용을 수행할 메소드 작성

```
@Test // 테스트용 메소드임을 명시하는 어노테이션
public void testMemberLogin() throws Exception {
        try{
        } catch(Exception e){
        }
}
```

9. 테스트 메소드 내부에 mockMvc를 이용하여 매핑될 url과 필요한 데이터가 담긴 가상의 요청을 작성. (perfrom() 메소드)

```
mockMvc.perform( post("/login.do").param("id","admin").param("pwd","1234"))
```

10. 처리되어진 내용을 출력 (.andDo(print()))

11. 응답 상태값이 에러가 없는 정상적인 상태(status 가 200)가 되도록 검증 (.andExpect(status().isOk())

12. 테스트용 Controller를 JUnit Test를 이용하여 실행

```java
@Test
public void testMemberLogin() throws Exception {

    try {
        // 9, 10, 11번 메소드 체이닝 이용
        mockMvc.perform( post("/login.do").param("id","admin").param("pwd","1234") )
                .andDo(print())
                .andExpect(status().isOk());

    } catch (Exception e) {

    }

}
```

테스트용 Class 코드

```
package com.ncs.spring;

import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilder
s.post;
import static
org.springframework.test.web.servlet.result.MockMvcResultHandlers.
print;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.
status;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import
org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;


@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(locations={"file:src/main/webapp/WEB-
INF/spring/root-context.xml",
"file:src/main/webapp/WEB-INF/spring/appServlet/servlet-
context.xml"})
public class MemberControllerTest {

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

	@Autowired
	private WebApplicationContext wac;
	private MockMvc mockMvc;

	@Before
	public void setup() {
		this.mockMvc =
MockMvcBuilders.webAppContextSetup(this.wac).build();
```

```
	}

	@Test
	public void testMemberLogin() throws Exception {

		try {

	mockMvc.perform( post("/login.do").param("id","admin").param
("pwd","1234") )
				.andDo(print())
				.andExpect(status().isOk());

		} catch (Exception e) {

		}

	}
```

2 - jUnit 의 테스트 지원 어노테이션

@Test

테스트를 수행하는 메소드를 지정합니다. jUnit 에서는 각각의 테스트가 서로 영향을 주지 않고 독립적으로 실행되는 것을 지향합니다. 따라서 @Test 단위 마다 필요한 객체를 생성해 지원해줍니다.

@Ignore

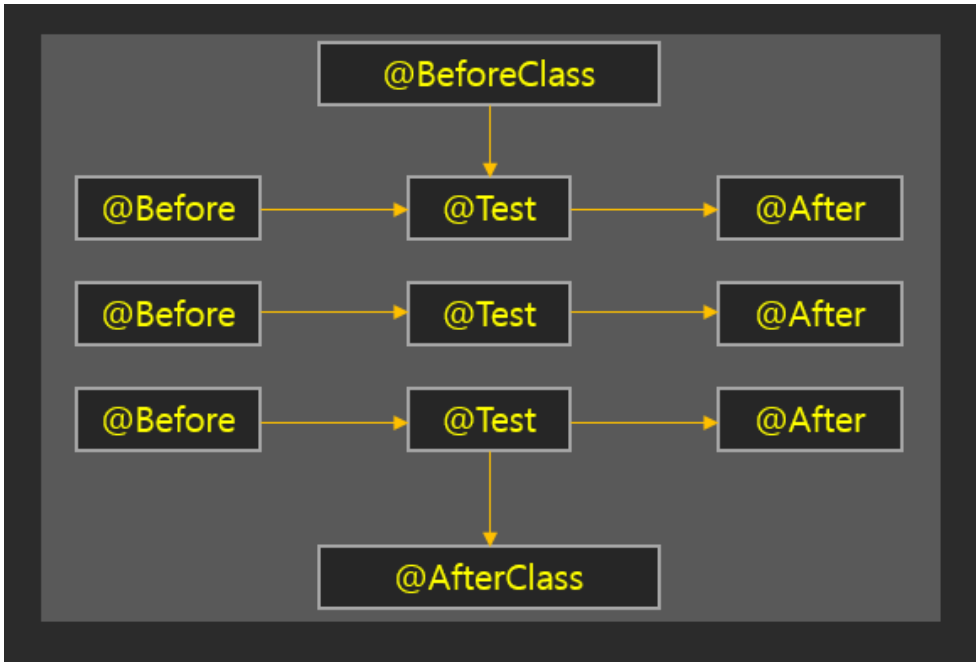
테스트를 실행하지 않도록 해줍니다. 메소드는 남겨두되 테스트에 포함되지 않도록 하려면 이 어노테이션을 붙여두면 됩니다.

@Before / @After

테스트 메소드가 실행되기 전, 후로 항상 실행되는 메소드를 지정합니다. 공통적으로 실행되어야 하는 메소드가 있다면 어노테이션을 붙여주면 됩니다. 각각의 테스트 메소드에 적용됩니다.

@BeforeClass / @AfterClass

각각의 메소드가 아닌 해당 클래스에서 딱 한번만 수행되는 메소드입니다. 테스트 메소드의 갯수와 상관없이 딱 한번만 실행됩니다.



3 - Spring-Test 의 어노테이션

@RunWith(SpringJUnit4ClassRunner.class)

ApplicationContext 를 만들고 관리하는 작업을 할 수 있도록 jUnit 의 기능을 확장해줍니다. 스프링의 핵심 기능인 컨테이너 객체를 생성해 테스트에 사용할 수 있도록 해준다고 보면 됩니다. 원래 jUnit 에서는 테스트 메소드별로 객체를 따로 생성해 관리하는 반면, Spring-Test 라이브러리로 확장된 jUnit 에서는 컨테이너 기술을 써서 싱글톤으로 관리되는 객체를 사용해 모든 테스트에 사용하게 됩니다.

@ContextConfiguration(locations = "classpath:xml 파일위치")

스프링 빈(Bean) 설정 파일의 위치를 지정할 수 있습니다. 굳이 별도로 컨테이너를 추가하지 않고 Bean 을 등록해둔 xml 파일을 지정해 컨테이너에서 사용할 수 있도록 해줍니다. 위의 @RunWith 어노테이션은 컨테이너를 생성하겠다는 의미인데, 어떤 파일을 참조할지 모르는 상태이기 때문에 이 어노테이션을 함께 써줘야 합니다.

파일 위치의 루트는 "src/test/resources" 폴더입니다. 필요한 설정 파일은 이곳에 복사해놓고 사용해도 됩니다. 하지만 매번 파일을 복사하면 힘들기 때문에 "file:Full path" 형식으로 써주면 운영 개발에서 사용하는 파일을 불러올 수 있습니다. 대괄호 { } 를 붙이면 여러개도 모두 가져올 수 있습니다. 아래 예시에서 추가 설명하겠습니다.

@Autowired

스프링에서 사용하는 것과 같습니다. 자동으로 의존성 주입을 해줍니다.