Part 05 Test 지원

Chapter 16 Test 지원

: 소프트웨어를 개발하는데 있어서 Test는 필수 요소이다. Test를 개발하는 것은 개발 효율을 높이는 데 도움을 주며, 자동화 된 Test를 통해 생산성 향상은 배가 된다.

SPRING은 JUnit4를 위한 지원 Class를 통해서 SPRING 기반의 통합 Test Code를 작성할 수 있도록 하고 있으며, Mock Test를 위한 지원 Class도 제공하고 있다.

본 절에서는 SPRING의 JUnit4 지원과 SPRING이 제공하는 Mock 생성 지원, 그리고 @Controller Class의 단위 Test 방법을 살펴보도록 하겠다.

1. Junit4의 SPRING 통합 Test

: SPRING Container로 부터 빈 객체를 가져와 올바르게 동작하는지의 여부를 Test 해야 한다고 하자. 이 때, JUnit의 @Test Annotation을 사용할 경우 SPRING 컨텍스트를

생성해 주어야 할 것이다.

SPRING 통합 Test를 수행하기 위한 모든 Test에 위와 같은 초기화 Code를 입력하는 것은 성가신 일이다.

JUnit4는 Annotation을 이용하여 Test를 수행할 수 있는 기능을 제공하고 있으며, @RunWith Annotation을 이용해서 Test를 실행할 org.junit.runner.Runner

구현 Class를 지정할 수 있다. SPRING은 SPRING 컨텍스트 설정 및 DI, Transaction 처리 등을 지원해주는 Runner 구현 Class를 제공하는데, 본 절에서는 SPRING이

제공하는 Runner Class와 Annotation을 이용해서 JUnit4 기반의 Test를 구현하는 방법을 살펴보도록 하자.

public class GuestMessageServiceTest {

@Autowired

private ApplicationContext applicationContext;

@Autowired

private GuestMessageService guestMessageService;

@Test

@Transactional

@Rollback(false)

public void write() {

GuestMessage message = new GuestMessage();

message.setGuestName("최범균");

message.setMessage("축하합니다.");

GuestMessage written = guestMessageService.write(message);

assertNotNull(written.getId());

}

@Test

@Transactional

public void list() {

GuestMessageList messageList = guestMessageService.getMessageList(1);

assertNotNull(messageList);

}

@Test

public void assertDao() {

GuestMessageDao dao = applicationContext.getBean(GuestMessageDao.class);

assertNotNull(dao);

}

}

1.1 SpringJUnit4ClassRunner를 사용한 JUnit4 기반 Test

: JUnit4 기반의 Test에서 SPRING 컨텍스트에 설정된 빈 객체를 사용하고 싶다면 아래와 같이 SpringJUnit4ClassRunner Class를 @RunWith Annotation 값으로

설정하면 된다.

package madvirus.spring.chap16.service;

import static org.junit.Assert.assertNotNull;

import madvirus.spring.chap16.model.GuestMessage;

import madvirus.spring.chap16.model.GuestMessageList;

import org.junit.Test;

import org.junit.runner.RunWith;

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

import org.springframework.test.context.ContextConfiguration;

import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:applicationContext.xml")

public class GuestMessageServiceTest {

@Autowired

private GuestMessageService guestMessageService;

@Test

public void write() {

GuestMessage message = new GuestMessage();

message.setGuestName("최범균");

message.setMessage("축하합니다.");

GuestMessage written = guestMessageService.write(message);

assertNotNull(written.getId());

}

@Test

public void list() {

GuestMessageList messageList = guestMessageService.getMessageList(1);

assertNotNull(messageList);

}

}

SPRING 컨텍스트를 생성할 때 사용될 설정 파일의 경로는 @ContextConfiguration Annotation의 locations 앨리먼트를 이용해서 설정한다. 만약 두 개 이상의

설정 파일을 사용해야 한다면 다음과 같이 배열을 이용해서 설정해 주면 된다.

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:dao.xml", "classpath:service.xml")

public class GuestMessageServiceTest {

...

}

Test Code에서 필요로 하는 SPRING 빈 객체는 @Autowired Annotation이나 @Resource Annotation을 이용하여 설정한다. 위 Code의 경우 SPRING 컨텍스트에서

타입이 GuestMessageService인 빈 객체를 라이18의 guestMessageService 필드에 할당한다.

1.2 Transaction 처리를 위한 설정

: Test 메서드를 Transaction 범위 내에서 실행하고 싶다면 다음의 Annotation을 사용하면 된다.

- @TransactionConfiguration : Test Class에 적용되며, Transaction 관리자와 기본 롤백 규칙을 설정한다.

- @Rollback : Test 메서드에 적용되며, 메서드 단위로 롤백 규칙을 설정한다.

- @Transactional : Test Class 또는 메서드에 적용

아래 Code는 위 Annotation을 적용하여 Transaction 범위내에서 Test를 수행하는 예제 Code이다.

package madvirus.spring.chap16.service;

import static org.junit.Assert.assertNotNull;

import madvirus.spring.chap16.model.GuestMessage;

import madvirus.spring.chap16.model.GuestMessageList;

import org.junit.Test;

import org.junit.runner.RunWith;

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

import org.springframework.test.annotation.Rollback;

import org.springframework.test.context.ContextConfiguration;

import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import org.springframework.test.context.transaction.TransactionConfiguration;

import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:applicationContext.xml")

@TransactionConfiguration(transactionManager = "txMgr", defaultRollback = true)

@Transactional

public class GuestMessageServiceTest {

@Autowired

private ApplicationContext applicationContext;

@Test

@Rollback(false)

public void write() {

GuestMessage message = new GuestMessage();

message.setGuestName("최범균");

message.setMessage("축하합니다.");

GuestMessage written = guestMessageService.write(message);

assertNotNull(written.getId());

}

@Test

public void list() {

GuestMessageList messageList = guestMessageService.getMessageList(1);

assertNotNull(messageList);

}

}

@TransactionConfiguration Annotation은 Test Class를 위한 Transaction 관리 정보를 설정한다. transactionManager 엘리먼트를 사용해서 PlatformTransactionManager

빈을 설정하며, 기본 값은 "transactionManager" 이다. defaultRollback 속성을 이용해서 Test 메서드 실행 후 Transaction을 롤백할 지의 여부를 설정하며, 기본 값은 true이다.

@Transactional Annotation을 Class에 적용하면, 각 Test 메서드가 @TransactionConfiguration Annotation이 지정한 Transaction 범위내에서 실행되며, Class가 아닌

메서드에 @Transactional Annotation을 적용하면 지정한 메서드만 Transaction 범위 내에서 실행된다.

만약 Test 메서드의 롤백 여부를 명시적으로 설정하고 싶다면 @Rollback Annotation을 사용한다. @Rollback Annotation의 값을 true로 지정하면 @TransactionConfiguration

Annotation의 설정에 상관 없이 Test 메서드 실행 후 Transaction이 롤백되며, @Rollback Annotation의 값이 false이면 (위 Code의 라인26) Test 메서드

실행 후 Transaction이 커밋된다.

Test 메서드에 대한 Transaction이 시작되기 전에 어떤 초기화 작업을 해주어야만 한다면, @BeforeTransaction Annotation이 적용된 메서드를 작성하면 된다. 반대로

Transaction이 종료된 이후에 정리 작업을 해 주어야 할 경우에는 @AfterTransaction Annotation이 적용된 메서드를 작성하면 된다.

1.3 SPRING 컨텍스트 참조 및 컨텍스트 리로딩 처리

: Test를 수행하는 데 사용되는 SPRING 컨텍스트 객체를 사용해야 한다면, @Autowired Annotation을 이용해서 ApplicationContext를 전달받으면 된다.

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:applicationContext.xml")

public class GuestMessageServiceTest {

@Autowired

private GuestMessageService guestMessageService;

@Test

public void assertDao() {

GuestMessageDao dao = applicationContext.getBean(GuestMessageDao.class);

assertNotNull(dao);

}

}

SPRING이 제공하는 SpringJUnit4ClassRunner를 이용해서 Test를 수행할 경우, ApplicationContext 객체를 최초에 로딩하면 이후 실행되는 Test 메서드는 모두

동일한 ApplicationContext 객체를 사용하게 된다. 하지만, Test 메서드를 실행하는 과정에서 ApplicationContext의 내용이 변경될 수도 있을 것이다. 이런 경우

다음에 실행되는 Test 메서드는 영향을 받을 수 있다.

이전 Test 메서드의 실행 결과로 ApplicationContext가 변경되고 그로 인해 현재 Test가 올바르게 동작되지 않을 경우, @DirtiesContext Annotation을 Test

메서드에 적용해서 문제를 해결할 수 있다.

@Test

@DirtiesContext

public void makeContextDirty(){

}

@Test

public void needNotDirtiesContext(){

}

@DirtiesContext Annotation이 적용된 Test 메서드를 실행할 경우, 다음 Test 메서드를 실행하기 전에 ApplicationContext를 리로딩 하게 된다. 따라서, 다음

Test는 초기화 상태의 ApplicationContext를 사용해서 Test를 진행하게 된다.

2. @Controller Class 단위 Test

2.1 SPRING Mock 객체를 이용한 WEB 관련 Class 단위 Test

: @Controller를 이용해서 구현한 컨트롤러 Class에서 HttpServletRequest 객체를 메서드로 전달받아 사용해야 하는 경우가 존재한다. 이 경우, 컨트롤러 Class를

단위 Test 하려면 모의 HttpServletRequest 객체를 전달해 주어야 하는데, 이럴때 유용하게 사용되는 Class가 MockHttpServletRequest Class이다.

(1) MockHttpServletRequest 기본 사용

: 아래와 같이 HttpServletRequest를 필요로 하는 컨트롤러 Class가 있다고 하자.

@Controller

public class HelloController {

@RequestMapping("/hello.do")

public ModelAndView hello(HttpServletRequest request) {

ModelAndView mav = new ModelAndView("hello");

mav.addObject("greeting", "안녕하세요");

mav.addObject("name", request.getParameter("name"));

return mav;

}

}

이 경우 HelloController Class의 단위 Test를 구현하려면 HttpServletRequest에 대한 Mock 객체가 필요한데, SPRING Test Module이 제공하는 MockHttpServletRequest

Class를 사용하면 손쉽게 HttpServletRequest에 대한 Mock 객체를 제공 할 수 있다.

package madvirus.spring.chap16.controller;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import org.springframework.mock.web.MockHttpServletRequest;

import org.springframework.web.servlet.ModelAndView;

public class HelloControllerUnitTest {

@Test

public void hello() {

MockHttpServletRequest mockRequest = new MockHttpServletRequest();

mockRequest.setMethod("POST");

mockRequest.setCharacterEncoding("euc-kr");

mockRequest.addParameter("name", "최범균");

HelloController controller = new HelloController();

ModelAndView mav = controller.hello(mockRequest);

assertEquals("hello", mav.getViewName());

assertEquals("최범균", mav.getModelMap().get("name"));

}

}

MockHttpServletRequest를 사용하려면, 먼저 MockHttpServletRequest 객체를 생성한 뒤, setMethod()를 이용해서 HTTP 메서드를 설정한다. HTTP 메서드를

설정했다면 addParameter() 메서드를 이용해서 컨트롤러를 실행하는데 필요로 하는 파라미터를 추가한다. addParameter() 메서드 중 주로 사용되는 것은

다음의 두 가지 이다.

- addParameter(String name, String value) : 이름이 name이고 값이 value인 파라미터를 추가한다.

- addParameter(String name, String[] values) : 이름이 name이고 값이 여러개인 파라미터를 추가한다. checkbox 타입의 <input> 태그와 같이 여러개의 값을

입력 받는 경우를 Test할 때 주로 사용된다.

MockHttpServletRequest의 HTTP 메서드 및 파라미터 정보를 설정했다면, 컨트롤러를 Test 하기 위해 MockHttpServletRequest 객체와 MockHttpServletResponse

객체를 컨트롤러의 요청 처리 메서드에 전달한다.

Test Code에서는 컨트롤러가 리턴한 ModelAndView 객체의 값을 이용해서 컨트롤러가 올바르게 동작했는지의 여부를 확인하면 된다.

MockHttpServletRequest request = new MockHttpServletRequest;

request.setMethod("GET");

...

LoginController loginController = new LoginController();

ModelAndView mav = loginController.handleRequest(request, response);

assertEquals(validViewName, mav.getViewName());

(2) 쿠키 설정

: 컨트롤러에서 쿠키가 필요로 한다면 MockHttpServletRequest.setCookies() 메서드를 사용해서 쿠키 정보를 설정할 수 있다. setCookies() 메서드는 Cookie[]을

값으로 전달받으므로 아래 Code와 같이 javax.servlet.http.Cookie 객체를 직접 생성한 뒤에 메서드에 전달하면 컨트롤러에서 해당 쿠키를 사용할 수 있게 된다.

MockHttpServletRequest request = new MockHttpServletRequest();

Cookie[] cookies = {new Cookie("runas", "madvirus"), new Cookie("id", "nals")};

request.setCookies(cookies);

컨트롤러가 HttpServletRequest.getCookies() 메서드를 호출하면, setCookies() 메서드를 통해서 설정한 Cookie 배열을 리턴하므로, 컨트롤러를 Test할 때에

필요한 쿠키가 있다면, setCookies() 메서드를 이용해서 알맞게 설정하면 된다.

(3) HttpSession을 위한 MockHttpSession Class

: 컨트롤러를 Test 하는 데 HttpSession이 필요하다면 다음과 같이 MockHttpSession 객체를 생성해서 MockHttpServletRequest.setSession() 메서드에 전달한다.

MockHttpServletRequest request = new MockHttpServletRequest();

MockHttpSession session = new MockHttpSession();

session.setAttribute("userInfo", userInfo);

session.setNew(true);

request.setSession(session);

컨트롤러가 request.getSession() 메서드를 호출하면 setSession() 메서드에 설정한 MockHttpSession 객체가 리턴된다. 따라서, MockHttpSession의 setAttribute()

메서드를 사용해서 컨트롤러를 Test 하는 데 HttpSession에 저장되어 있어야 할 속성 값을 설정하면 된다. HttpSession.isNew()가 리턴할 값은 setNew()

메서드를 사용해서 할당할 수 있다.

(4) 파일 업로드 Test

: 파일을 업로드 받는 컨트롤러를 Test 하려면 MockMultipartHttpServletRequest Class를 사용해서 Mock request 객체를 생성한다.

MockMultipartHttpServletRequest multipartRequest = new MockMultipartHttpServletRequest();

multipartRequest.setMethod("POST");

InputStream fileInputStream = ...; //파일로 사용될 데이터를 InputStream으로 구함.

multipartRequest.addFile(new MockMultipartFile("reportFile", "temp/SpringReport.doc",null,"fileInputStream);

byte[] fileBytes = ...; //파일로 사용할 데이터를 byte[]로 구함.

multipartRequest.addFile(new MockMultipartFile("sourceCode", "temp/SubmitReportControllerTest.java", null, fileBytes);

multipartRequest.addParameter("subject", "주제");

ModelAndView mav = new reportController.handleRequest(multipartRequest, response);

MockMultipartHttpServletRequest는 MockHttpServletRequest를 상속 받았으며, 첨부 파일 정보를 추가할 수 있는 addFile() 메서드를 추가적으로 제공하고 있다.

addFile() 메서드는 MultipartFile Interface를 구현한 객체를 전달받는데, SPRING이 제공하는 MockMultipartFile Class를 사용하면 파일 정보를 간단하게 추가할 수

있다.

MockMultipartFile Class는 다음과 같은 생성자를 제공하고 있다.

- MockMultipartFile(Strin name, byte[] content)

- MockMultipartFile(Strin name, InputStream contentStream)

- MockMultipartFile(Strin name, String originalFileName, String contentType, byte[] content)

- MockMultipartFile(Strin name, String originalFileName, String contentType, InputStream contentStream)

MockMultipartFile 생성자의 name은 파라미터의 이름을 의미하며, Byte[] 이나 InputStream은 파일 데이터를 의미한다. originalFileName 파라미터는 원본 파일 이름을

지정할 때 사용되며, contentType은 파일의 컨텐츠 타입을 지정할 때 사용된다.