<http://thswave.github.io/java/2015/03/02/spring-mvc-test.html>

# How to Spring MVC Unit Test 스프링 MVC 단위 테스트

March 2nd, 2015

# 스프링 MVC 단위 테스트!

계속해서 들려오는 TDD를 해보고 싶었던차에 강제적으로(?) 단위테스트를 만들면서 개발해야되는 상황이 되었습니다.

지금까지 TDD 예제나 소개에서는 간단한 알고리즘 문제 풀이를 다루고 있었고 단순한 JUnit Test 케이스를 만들순 있었습니다.

하지만 Spring MVC의 Controller, Service, Model 성격이 다른 각 계층에 대한 테스트 케이스를 만드려고 하니 막막해졌습니다. MVC를 구성하는 각 레이어 별 어떻게 테스트를 해야 할지 크게 세가지 정도가 감이 오지 않았습니다.

* 1. Controller Method

Controller에 request에 매핑되는 메서드는 일반 객체의 메서드를 호출하듯 마음대로 호출할 수 있는게 아니었기 때문 이를 어떻게 단위테스트에서 호출할 수 있는지 의문이었습니다.

* 2. Controller의 테스트 방식

Controller에서는 Service의 메서드를 호출하기만 하는데 이를 어떻게 단위 테스트 할 수 있는지 의문이었습니다.

* 3. DAO의 테스트 방식

단위테스트는 어느환경에서 실행하여도 성공을 보장해주어야 하는데 데이터베이스를 접근하는 DAO의 경우 데이터베이스 접근이 실패하거나 테스트 할때의 데이터베이스에 있는 테이블 데이터에 따라 성공 혹은 실패하기 때문에 이를 어떻게 처리해야 하는지 의문이었습니다.

이제부터 Spring MVC의 단위테스트를 위한 과정을 하나씩 살펴보겠습니다. Controller, Service, DAO 별로 테스트 하는 방식이 다르기 때문에 하나씩 살펴보겠습니다.

## Controller 단위 테스트

Spring MVC, JUnit, Maven을 사용중이라는 가정하에 maven의 pom.xml에서 스프링 테스트, Mockito, JUnit을 추가해 줍니다.

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-all</artifactId>

<version>1.9.5</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>3.2.3.RELEASE</version>

<scope>test</scope>

</dependency>

그리고 특정 카테고리에 대한 Controller

@RequestMapping(value = "/category")

public class CategoryController {

@Autowired

private CategoryService categoryService;

@RequiredLogin

@RequestMapping(value = "/list", method = RequestMethod.GET)

public ModelAndView allCategoryList() {

return categoryService.method1();

}

}

CategoryController에서는 @Autowired로 CategoryService를 사용합니다.

본격적으로 Test 케이스를 작성해 봅시다.

CategoryControllerTest.java

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(locations = {"file:src/main/webapp/WEB-INF/spring/appServlet/test-servlet-context.xml",

"file:src/main/webapp/WEB-INF/mybatis/test-mybatis-context.xml"})

@WebAppConfiguration

public class CategoryControllerTest {

@Mock

CategoryService categoryService;

@InjectMocks

private CategoryController categoryController;

@Autowired

private WebApplicationContext wac;

private MockMvc mockMvc;

@Before

public void setUp() throws ExceptiMon {

MockitoAnnotations.initMocks(this);

mockMvc = MockMvcBuilders.standaloneSetup(categoryController).build();

}

@Test

public void testCategoryController() throws Exception {

when(categoryService.method1()).thenReturn(10);

mockMvc.perform(get("/category/list")).andExpect(status().isOk());

verify(categoryService).method1();

verifyNoMoreInteractions(categoryService);

}

}

테스트 클래스 상단에 아래와 같이 작성해 줍니다.

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(locations = {"file:src/main/webapp/WEB-INF/spring/appServlet/test-servlet-context.xml",

"file:src/main/webapp/WEB-INF/mybatis/test-mybatis-context.xml"})

@WebAppConfiguration

public class ClassName {

// 생략

}

테스트를 위한 유일한 방법이 아니고 여러가지 다른 방법이 있지만 저는 xml설정 파일을 읽어 테스트하도록 하였습니다.

SpringJUnit4ClassRunner.class 클래스는 spring-test 에서 제공하는 단위테스트를 위한 클래스 러너입니다.

아래 @ContextConfiguration 어노테이션은 스프링 빈 설정, 데이터베이스 설정 등이 있는 servlet-context.xml과 같은 xml 설정을 읽어 스프링 테스트 시작 전에 빈을 초기화 하여 @Autowired 를 통해 빈 팩토리에서 미리 생성한 빈들을 읽어 들일 수 있고 그 밖의 spring 프레임워크 설정들을 읽어 들일 수 있습니다.

위와 같이 설정하고 실행할 경우 스프링을 실행할 때와 마찬가지로 설정 파일을 읽어들이여 Request Mapping 등의 작업을 하는걸 확인할 수 있습니다.

Controller에서 @Autowired로 된걸 단위테스트에서 동일하게 주입시키기 위해 Mockito의 @InjectMocks, @Mock을 설정합니다.

@Mock

CategoryService categoryService;

@InjectMocks

private CategoryController categoryController;

여기서 헷갈리는 부분이 있다면 *@Mock*은 Mockito 단위 테스트 프레임워크라는 것이고 이것은 spring-test와는 무관하다 볼 수 있습니다.

Controller를 테스트 할 것이므로 해당 컨트롤러에 @Autowired 한 CategoryService 빈을 주입해 줍니다.

그리고 반드시 @Before에서 아래와 같이 호출해줍니다.

@Before

public void setUp() throws Exception {

MockitoAnnotations.initMocks(this);

mockMvc = MockMvcBuilders.standaloneSetup(categoryController).build();

}

여기서 MockitoAnnotations.initMocks(this); 이부분이 @Autowired를 해주는 것이라 생각하시면 됩니다. 그리고 Controller 단위테스트의 핵심이라 볼수있는 MockMVC를 설정하게 됩니다.

mockMvc = MockMvcBuilders.standaloneSetup(categoryController).build();

standaloneSetup()말고 webAppContextSetUp 방식도 있으니 참고바랍니다.

* MockMvc : Controller에 request를 수행해줍니다.
* Mockito: 동작이나 값을 대신해주는 목 객체 생성하고 verify를 통해 정상적으로 호출되었는지 검증.

이제 Mockito의 when().thenReturn() 등을 활용해 컨트롤러에 로직을 테스트 해주면 됩니다.

여기서 중요한건 Controller의 수행 흐름을 테스트 한다는 것이지 CategoryService의 동작을 테스트 하는게 아닙니다. (지금은 매우 간단한 로직이지만 controller의 Controlling 해주는 부분이 늘어나면 이를 더 테스트 해야 합니다.)

mockMvc.perform()으로 실제 request를 테스트 해볼 수 있습니다.

mockMvc.perform(get("/category/list")).andExpect(status().isOk());

정상적으로 200 status코드를 받았는지 단순히 테스트하는 정도지만 여기서 다양한 테스트를 해볼 수 있습니다.(예를 들어, json 을 받았다면 해당 데이터가 몇 개의 요소들로 있는지 어떤 데이터로 구성되어 있는지, 또는 예외가 발생한 다면 어떤 예외가 발생할 수 있는지 등 아주 세세하게 테스트 해볼 수 있습니다.)

Controller에서 호출한 메서드의 호출이 되었는지 확인하기 위해서는 mockito의 verify를 사용해야 합니다.

* mockito의 기본적인 사용방법 참조 사이트

<http://www.baeldung.com/mockito-behavior>

<http://www.baeldung.com/mockito-verify>

## 결론

컨트롤러를 테스트 할 수 있는 방법은 여러가지 일 수 있습니다. MockHttpServlet 을 활용하여 테스트 한다거나 혹은, controller를 일반 함수라고 가정하고 각 파라미터들을 주입하고 이를 테스트 해볼 수 있습니다. 단 제 생각은 Controller에는 말 그대로 Request에 대한 Controlling을 하는 부분이 들어가 있어야 하고 Service 레이어에 로직이 들어가 있어야 할것입니다. 그렇다면 로직을 테스트하는 것인지 혹은 어디로 요청을 보내는지 를 테스트 하는지 구분 지어 테스트해야 할 것입니다. spring-test 를 활용하여 컨트롤러를 테스트한 이유는 spring의 AOP라던가 Interceptor 등 스프링이 제공해주는 기능을 포함하여 테트스 할 때 필요할 것입니다. 위 간단한 예제에서는 비록 복잡한 설정이 없었기 때문에 다른 방식으로도 충분히 테스트 할 수 있었지만 스프링 설정을 읽어들여 좀 더 정확한 테스트를 하고자 한다면 spring-test에서 제공하는 MockMvc를 활용하면 더욱 효율적일 것 같습니다.

Service layer 테스트는 다음 포스팅에 계속 하겠습니다.

포스트를 정리 한 후 예제 full source code를 다시 올리겠습니다.

#### ps. request에 attribute를 넣어주는 방법.

MockHttpServletRequestBuilder request = get("category/list");

request.with(new RequestPostProcessor() {

@Override

public MockHttpServletRequest postProcessRequest(MockHttpServletRequest request) {

request.setAttribute("a", "a");

return request;

}

});

return request;

우선 구글링을 통해 단위테스트 작성에 도움되는 사이트들을 소개해 드립니다.

* 참조 사이트 <http://www.petrikainulainen.net/spring-mvc-test-tutorial/>

<http://www.petrikainulainen.net/programming/spring-framework/integration-testing-of-spring-mvc-applications-configuration/>

<http://www.petrikainulainen.net/programming/spring-framework/unit-testing-of-spring-mvc-controllers-configuration/>

<http://thysmichels.com/2014/01/07/junit-test-spring-mvc-web-services/>

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">       <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">  <property name="driverClassName" value="oracle.jdbc.OracleDriver"></property>  <property name="url" value="jdbc:oracle:thin:@2222.47:1521:orcl"></property>  <property name="username" value="444"></property>  <property name="password" value="o\_422"></property>  </bean>  </beans> |
| package com.nhis.ggij.cms.ggijd406;  import com.nhis.ggij.cms.nhis.ggij400.Ggijd406Controller; import com.nhis.ggij.cms.nhis.ggij400.service.Ggijd406Service; import lombok.extern.slf4j.Slf4j; import org.junit.Before; import org.junit.Test; import org.junit.runner.RunWith; import org.mockito.InjectMocks; import org.mockito.Mock; import org.mockito.MockitoAnnotations; 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; import static org.hamcrest.Matchers.\*; import static org.junit.Assert.\*; import static org.mockito.Mockito.\*; import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*get*; import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*post*; import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*; @Slf4j @RunWith(SpringJUnit4ClassRunner.class) //@RunWith(MockitoJUnitRunner.class) @ContextConfiguration(locations =  {  "file:src/test/resources/spring/dispatcher-servlet-test.xml",  "file:src/main/resources/spring/context-common.xml",  "file:src/main/resources/spring/context-config.xml",  "file:src/test/resources/spring/context-datasource-test.xml",  "file:src/main/resources/spring/context-hibernate.xml",  }) @WebAppConfiguration public class Ggijd4062Test {  @Mock  Ggijd406Service ggijd406Service;  @Autowired  private WebApplicationContext wac;  @InjectMocks  Ggijd406Controller ggijd406Controller;  private MockMvc mockMvc;  @Before  public void setUp() throws Exception {  MockitoAnnotations.*initMocks*(this);  mockMvc = MockMvcBuilders.*standaloneSetup*(ggijd406Controller).build();  }   @Test  public void testCategoryController() throws Exception {  *log*.info(ggijd406Controller.toString());  // when(ggijd406Controller.getCm()).thenReturn(10);  mockMvc.perform(*get*("/GGIJD406/555/gg")).andExpect(*status*().isOk()); // mockMvc.perform(get("/category/list")).andExpect(status().isOk()); // verify(categoryService).method1(); // verifyNoMoreInteractions(categoryService);   }  }  http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/testing.html#spring-mvc-test-framework |
| security  <http://docs.spring.io/spring-security/site/docs/current/reference/html/test-method.html>  <https://spring.io/blog/2014/05/23/preview-spring-security-test-web-security>  http://docs.spring.io/spring-security/site/docs/current/reference/html/test-mockmvc.html |
| https://sourceforge.net/p/strutstestcase/discussion/121751/thread/c7e6d86f/ |
| <http://jdm.kr/blog/165> |
|  |
| security적용한것  package com.nhis.ggij.cms.ggijd406;  import com.nhis.ggij.cms.nhis.ggij400.Ggijd406Controller; import com.nhis.ggij.cms.nhis.ggij400.service.Ggijd406Service; import lombok.extern.slf4j.Slf4j; import org.hibernate.Session; import org.hibernate.SessionFactory; import org.junit.Before; import org.junit.Test; import org.junit.runner.RunWith; import org.mockito.InjectMocks; import org.mockito.Mock; import org.mockito.MockitoAnnotations; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.mock.web.MockHttpServletRequest; import org.springframework.mock.web.MockHttpServletResponse; import org.springframework.orm.hibernate5.SessionHolder; import org.springframework.orm.hibernate5.support.OpenSessionInViewFilter; import org.springframework.security.core.context.SecurityContext; import org.springframework.security.core.context.SecurityContextHolder; import org.springframework.security.test.context.support.WithMockUser; import org.springframework.security.web.FilterChainProxy; import org.springframework.security.web.context.HttpSessionSecurityContextRepository; 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.transaction.support.TransactionSynchronizationManager; import org.springframework.web.context.WebApplicationContext;  import javax.servlet.Filter; import javax.servlet.http.HttpSession;  import static org.hamcrest.Matchers.\*; import static org.junit.Assert.\*; import static org.mockito.Mockito.\*; import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*get*; import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*; import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*post*; import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*status*; import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.\*; import static org.springframework.security.test.web.servlet.setup.SecurityMockMvcConfigurers.\*; import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.\*; import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*csrf*; @Slf4j @RunWith(SpringJUnit4ClassRunner.class) //@RunWith(MockitoJUnitRunner.class) @ContextConfiguration(locations =  {  "file:src/test/resources/spring/dispatcher-servlet-test.xml",  "file:src/main/resources/spring/context-common.xml",  "file:src/main/resources/spring/context-config.xml",  "file:src/test/resources/spring/context-datasource-test.xml",  "file:src/main/resources/spring/context-hibernate.xml",  "file:src/main/resources/spring/context-security.xml",  }) @WebAppConfiguration public class Ggijd4062Test {  @Mock  Ggijd406Service ggijd406Service;   @Autowired  private FilterChainProxy springSecurityFilterChain; // @Autowired // private Filter springSecurityFilterChain;   @Autowired  private WebApplicationContext context;  @Autowired  private MockHttpServletRequest request;  @Autowired  private MockHttpServletResponse response;    SessionFactory sessionFactory;  @Autowired  Ggijd406Controller ggijd406Controller;  private MockMvc mockMvc;  @Before  public void setUp() throws Exception { // MockitoAnnotations.initMocks(this);  //mockMvc = MockMvcBuilders.standaloneSetup(ggijd406Controller).build(); // mockMvc = MockMvcBuilders.webAppContextSetup(webappContext) // .addFilters(springSecurityFilterChain) // .build();  OpenSessionInViewFilter hibernateFilter = new OpenSessionInViewFilter();  hibernateFilter.setSessionFactoryBeanName("sessionFactory");  mockMvc = MockMvcBuilders  .*webAppContextSetup*(context) // .addFilter(hibernateFilter, "/\*")  .apply(*springSecurity*())  .build();     sessionFactory = (SessionFactory) context.getBean("sessionFactory");  //open and bind the session for this test thread.  Session s = sessionFactory.openSession();  TransactionSynchronizationManager.*bindResource*(sessionFactory, new SessionHolder(s));  }   //http://jdm.kr/blog/165  @WithMockUser(username="admin99",password = "1234")  @Test  public void testCategoryController() throws Exception {       HttpSession session = mockMvc.perform(*post*("/sign/login").with(*csrf*())  .param("username", "admin99")  .param("password", "1234")) // .andDo(print()) // .andExpect(status().isMovedTemporarily()) // .andExpect(redirectedUrl("/"))  .andReturn()  .getRequest()  .getSession();   request.setSession(session);  SecurityContext securityContext = (SecurityContext) session.getAttribute(HttpSessionSecurityContextRepository.*SPRING\_SECURITY\_CONTEXT\_KEY*);  SecurityContextHolder.*setContext*(securityContext);    ggijd406Controller.detail(request, response,"555","gg");  // log.info(ggijd406Controller.toString());  // when(ggijd406Controller.getCm()).thenReturn(10); // mockMvc.perform(get("/nfcTest")).andDo(print()); // mockMvc.perform(get("/GGIJD406/555/gg")).andDo(print());   // mockMvc.perform(get("/GGIJD406/555/gg")).andExpect(status().isOk()); // mockMvc.perform(get("/category/list")).andExpect(status().isOk()); // verify(categoryService).method1(); // verifyNoMoreInteractions(categoryService);    // sessionFactory.getCurrentSession().getTransaction().commit();   }  } |

# pring MockMvc(spring-test)

Spring│2015-07-03

*Wide* 확대 축소

스프링 프레임워크에서 단위 테스트의 중요성은 다들 아실거라 생각합니다. :D

이번 포스팅에서는 Spring 3.2부터 사용 가능한 MockMvc를 활용한 단위 테스트에 대해 알아보고자 합니다.

**또한 Springboot에서 같은 방법으로 사용가능합니다.**

## 1. Spring MockMvc

spring.io에서 [Spring Framework 3.2 RC1: Spring MVC Test Framework](https://spring.io/blog/2012/11/12/spring-framework-3-2-rc1-spring-mvc-test-framework)라는 이름으로 포스팅이 되었습니다. 시간 나면 읽어보세요. :D

MockMvc는 기존의 MockHttpServletRequest, MockHttpServletResponse을 활용한 단위 테스트에서 발전되었습니다. 기존의 Mock 객체들은 충분한 검증 결과Annotaions, Request, etc. 를 포함한를 보여 줄 수 없었기 때문입니다. 그래서 spring-test 모듈을 스프링 프레임워크에 더했습니다.

## 2. How to use Spring MockMvc

그럼 간단하게 스프링 테스트 모듈을 사용해 봅시다.

### 2.1. Create Project

프로젝트 생성부터 해야겠네요. IDE는 STS이고 다음과 같은 순서로 만듭니다.

File > New > Spring Project > Spring MVC Project

패키지명은 kr.jdm.junit으로 했습니다.

### 2.2. Update pom.xml for Spring

pom.xml 파일을 갱신해야 합니다. 아래와 같은 부분을 추가하거나 수정해 주세요. **Springboot는 이 부분을 무시하셔도 됩니다.**

<properties>

<java-version>1.6</java-version>

<org.springframework-version>4.1.7.RELEASE</org.springframework-version>

<org.aspectj-version>1.6.10</org.aspectj-version>

<org.slf4j-version>1.6.6</org.slf4j-version>

</properties>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>${org.springframework-version}</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.9</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

</dependency>

자바 버전이 낮은것이 흠이긴 하지만, 1.7 또는 1.8 이상으로 업그레이드 하셨다면 의존성 처리를 깨끗하게 하신 뒤에 다음으로 진행합니다.

### 2.3. Controller

자동으로 생성된 컨트롤러의 소스 코드입니다. 이번 포스팅의 목표는 기존에 생성된 프로젝트를 최소한으로 수정해서 사용법을 알아 보는 것입니다. 따라서 기존 생성된 코드는 건들지 않습니다.

View Source Code

### 2.4. Setup

셋업 코드를 작성합니다. 두 가지의 테스트 클래스가 작성됩니다. 해당 클래스들은 src/test/java 디렉토리 하위 kr.jdm.junit 패키지 경로에 만들어집니다.

#### 2.4.1. Setup with StandAlone

먼저 볼 것은 컨트롤러 단독으로 테스트 하기 위한 테스트 클래스입니다.

package kr.jdm.junit;

// 이부분 추가하셔야 합니다.

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;

import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.model;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import org.junit.Before;

import org.junit.Test;

import org.springframework.test.web.servlet.MockMvc;

import org.springframework.test.web.servlet.setup.MockMvcBuilders;

public class StandAloneTest {

private MockMvc mockMvc;

// 테스트 메소드 실행전 셋업 메소드입니다.

@Before

public void setup(){

// 이곳에서 HomeController를 MockMvc 객체로 만듭니다.

this.mockMvc = MockMvcBuilders.standaloneSetup(new HomeController()).build();

}

@Test

public void test() throws Exception{

// HomeController의 "/" 매핑으로 정의합니다.

this.mockMvc.perform(get("/"))

// 처리 내용을 출력합니다.

.andDo(print())

// 상태값은 OK가 나와야 합니다.

.andExpect(status().isOk())

// "serverTime"이라는 attribute가 존재해야 합니다.

.andExpect(model().attributeExists("serverTime"));

}

}

INFO : kr.jdm.junit.HomeController - Welcome home! The client locale is en.

MockHttpServletRequest:

HTTP Method = GET

Request URI = /

Parameters = {}

Headers = {}

Handler:

Type = kr.jdm.junit.HomeController

Method = public java.lang.String kr.jdm.junit.HomeController.home(java.util.Locale,org.springframework.ui.Model)

Async:

Async started = false

Async result = null

Resolved Exception:

Type = null

ModelAndView:

View name = home

View = null

Attribute = serverTime

value = July 3, 2015 9:42:51 PM KST

FlashMap:

MockHttpServletResponse:

Status = 200

Error message = null

Headers = {}

Content type = null

Body =

Forwarded URL = home

Redirected URL = null

Cookies = []

#### 2.4.2. Setup with Application Configuration

정의된 웹 어플리케이션 컨텍스트를 이용한 테스트 클래스를 작성합니다.

package kr.jdm.junit;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;

import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.model;

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.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/appServlet/servlet-context.xml")

public class ContextConfigTest {

@Autowired

private WebApplicationContext context;

private MockMvc mockMvc;

@Before

public void setup(){

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

}

@Test

public void test() throws Exception{

this.mockMvc.perform(get("/"))

.andDo(print())

.andExpect(status().isOk())

.andExpect(model().attributeExists("serverTime"));

}

}

INFO : kr.jdm.junit.HomeController - Welcome home! The client locale is en.

MockHttpServletRequest:

HTTP Method = GET

Request URI = /

Parameters = {}

Headers = {}

Handler:

Type = kr.jdm.junit.HomeController

Method = public java.lang.String kr.jdm.junit.HomeController.home(java.util.Locale,org.springframework.ui.Model)

Async:

Async started = false

Async result = null

Resolved Exception:

Type = null

ModelAndView:

View name = home

View = null

Attribute = serverTime

value = July 3, 2015 9:43:56 PM KST

FlashMap:

MockHttpServletResponse:

Status = 200

Error message = null

Headers = {}

Content type = null

Body =

Forwarded URL = /WEB-INF/views/home.jsp

Redirected URL = null

Cookies = []

### 2.5. Result Console

위의 두가지 테스트 클래스의 Console 결과는 비슷하지만 조금씩 차이도 있습니다. 이부분은 우선 참고만 하시길 바랍니다.

## 3. Analyze MockMvc

간단하게 예제 코드를 통한 단위 테스트 방법을 확인 했습니다. 그럼 하나씩 뜯어봅시다.

### 3.1. MockMvc

해당 클래스는 [MockMvc](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/web/servlet/MockMvc.html) Doc에서 확인 가능합니다. 지원하는 메소드는 .perform()입니다. 이 메소드가 리턴하는 객체는 ResultActions라는 인터페이스입니다.

this.mockMvc.perform(get("/")) // basic

this.mockMvc.perform(post("/")) // send post

this.mockMvc.perform(get("/?foo={var}", "1")) // query string

this.mockMvc.perform(get("/").param("bar", "2")) // using param

this.mockMvc.perform(get("/").accept(MediaType.ALL)) // select media type

post(), put(), delete() 등도 가능합니다.

**TIP**

혹시라도 Filter 처리를 하고 싶다면 MockMvc를 초기화 할때 build() 메소드 호출 전 .addFilter 또는 .addFilters 메소드를 호출해서 필터를 등록해주시면 됩니다.

### 3.2. ResultActions

MockMvc.perform 메소드로 리턴되는 인터페이스입니다. [ResultActions](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/web/servlet/ResultActions.html) Doc에서 확인 가능합니다. 지원 메소드는 andExpect, andDo, andReturn 입니다.

#### 3.2.1. andExpert

예상값을 검증합니다. *assert\** 메소드들과 유사합니다.

// status 값이 정상인 경우를 기대하고 만든 체이닝 메소드 중 일부입니다.

.andExpect(status().isOk())

// contentType을 검증합니다.

.andExpect(content().contentType("application/json;charset=utf-8"))

**WARNING**

혹시라도 mockMvc 객체로 테스트 중 한글이 깨지는 경우 컨트롤러 매핑을 확인해 봅시다.

@RequestMapping(value = "/", method = RequestMethod.POST, produces = "application/json;charset=utf-8")

위처럼 produces 부분에 캐릭터셋도 확실하게 지정을 해줘야 합니다.

#### 3.2.2. andDo

요청에 대한 처리를 합니다. print() 메소드가 일반적입니다.

.andDo(print())

#### 3.2.3. andReturn

테스트 클래스에서 작성은 안했지만 테스트한 결과 객체를 받을 때 사용합니다.

MvcResult result = this.mockMvc.perform(get("/"))

.andDo(print())

.andExpect(status().isOk())

.andExpect(model().attributeExists("serverTime"))

.andReturn();

## 4. Setup with Application Configuration for Springboot

Springboot에서는 아래처럼 테스트 클래스를 만듭니다.

@RunWith(SpringJUnit4ClassRunner.class)

@SpringApplicationConfiguration(classes = DemoApplication.class)

@WebAppConfiguration

public class TestClass {

@Autowired

private WebApplicationContext context;

private MockMvc mvc;

@Before

public void setup(){

this.mvc = MockMvcBuilders.webAppContextSetup(this.context).build();

}

@Test

public void contextLoads() throws Exception {

this.mvc.perform(post("/").param("test", "한글")).andDo(print())

.andExpect(content().contentType("application/json;charset=utf-8"));

}

}

## 5. jsonPath

만약 응답 결과물이 json이라면 [JsonPath](https://github.com/jayway/JsonPath)를 한번 검토해보세요. 다음처럼 사용할 수 있습니다.

/\*

{"message":"val"} 이라는 response를 받았는지 검증하려면 아래처럼 사용하세요.

\*/

.andExpect(jsonPath("$.message").value("val"))

## 6. Closing Remarks

새벽에 오류난 것을 고치는것보다 미리 테스트를 통한 검증으로 발뻗고 잡시다. :D

[Go Top(문서 처음으로)](http://jdm.kr/blog/165)

이 저작물은 [저작자표시-비영리-동일조건변경허락 4.0 국제 라이선스](http://creativecommons.org/licenses/by-nc-sa/4.0/deed.ko)에 따라 이용할 수 있습니다.   
This work is licensed under a [Attribution-NonCommercial-ShareAlike 4.0 International License](http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en).

모든 제품명 및 상표는 해당 소유주의 자산입니다.  
All product names and trademarks are the property of their respective owners.

**저작권 위배** 또는 잘못된 내용이 있으면 부담없이 댓글 또는 메일 부탁드립니다.  
If there are **copyright issues**, please contact webmaster@jdm.kr or write reply.

<http://blog.saltfactory.net/using-resttemplate-in-spring/>

# Spring에서 RestTemplate을 사용하여 REST 기반 서비스 요청과 테스트하기

25 November 2015 on [spring](http://blog.saltfactory.net/tag/spring/), [java](http://blog.saltfactory.net/tag/java/), [rest](http://blog.saltfactory.net/tag/rest/), [springboot](http://blog.saltfactory.net/tag/springboot/), [resttemplate](http://blog.saltfactory.net/tag/resttemplate/)

## 서론

우리는 며칠간에 걸쳐 Spring Boot로 Spring 기반 REST 서비스와 템플릿 뷰를 사용하여 Multipart Form data를 사용하기 위한 컨트롤러를 구현하는 방법을 살펴보았다. 또한 컨트롤러를 테스트하기 위해서 **MockMvc**를 사용하여 짧은 코드로 간단하게 Spring 테스트 프레임워크에서 URL을 요청하여 컨트롤러를 테스트하는 방법도 살펴보았다.

Spring 기반 프로젝트를 진행하면 컴포넌트 내부에서 URL을 요청해야하는 경우가 있다. 이전에는 Apache의 [HttpClient](https://hc.apache.org/httpcomponents-client-ga/) 라이브러리를 포함시켜 Http Request를 컴포넌트 내부에서 사용했지만 최근 Spring에서는 Http Request 요청을 간단하게 사용할 수 있도록 [SpringTemplate](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html)라는 모듈을 포함하고 있다. 이번 포스팅에서는 **RestTemplate**을 사용하여 Spring 안에서 **GET**, **POST**, **PUT**, **PATCH**, **DELETE** REST 요청을 처리하는 방법을 살펴본다.

## 데모를 위한 프로젝트 생성

우리는 앞에서 [spring-boot-rest](https://github.com/saltfactory/saltfactory-java-tutorial/tree/spring-boot-form) 소스 코드에 REST 서비스를 위한 컨트롤러에 대한 예제 코드를 만들었다. 만약 이 글을 처음 보고 있다면 앞의 [Spring에서 REST 서비스를 위한 컨트롤러 생성과 컨트롤러 단위테스트 하기](http://blog.saltfactory.net/java/create-and-test-rest-conroller-in-spring.html) 글을 참조하여 RestTemplate 테스트를 하기 위한 REST 서비스를 위한 컨트롤러를 먼저 만들자.

## @WebIntegrationTest 로 RestTemplate 사용하기

우리는 앞에서 컨트롤러를 만들고 단위 테스트를 할 때 **MockMvc**를 사용하였다. MockMvc는 말 그대로 가짜 웹 서버와 Http request 만들어서 테스트하는 것이다. [@WebIntegrationTest](http://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/WebIntegrationTest.html)는 웹 서비스를 다르게 테스트할 수 있는 방법을 제시하고 있다. **@WebIntegrationTest**는 [@WebAppConfigration](http://docs.spring.io/spring-framework/docs/3.2.0.BUILD-SNAPSHOT/api/org/springframework/test/context/web/WebAppConfiguration.html)과 [@IntegrationTest](http://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/IntegrationTest.html)가 통합되어 만들어진 것으로 Web Application을 설정과 실제 웹 서버를 동작하여 테스트를 하는 것과 같이 테스트를 할 수 있는 방법을 제공하고 있다.

우리는 **RestTemplate**에 관한 테스트를 진행할 것이기 때문에 앞에서 **ArticlesControllerTests** 파일을 참조하여 **src/test/{패키지명}/RestTemplateTests.java** 파일을 다음과 같이 만들자.

**@WebIntegrationTest("server.port=0")** : 테스트를 위해서 동작하는 웹 서버 포트 번호를 지정할 수 있는데 이 값이 **0**이면 랜덤으로 테스트를 할 때 지정하여 동작하게 된다. 이 때 지정된 포트번호는 @Value("{local.server.port}") int port; 형태로 injection으로 값을 가져올 수 있다. WebIntegrationTest 방법으로 테스트를 진행할 때는 실제 테스트를 위한 웹 서버가 동작하는 것이기 때문에 서버에 접근할 수 있는 URL이 필요하다. 우리는 포트번호를 랜덤하게 정하였기 때문에 기본적으로 URL을 만들기 위해서 String baseUrl변수를 만들었고 이것은 테스트가 진행할 때 @before 테스트 시작 전에 포트번호를 가지고 URL의 앞부분을 만들 것이다. 예를 들면 <http://localhost:81268> 와 같은 식으로 만들어지는 것이다. 그리고 우리는 웹 서버에 접근하여 Http Request를 요청하는 것을 **RestTemplate**으로 사용할 것이기 때문에 테스트 전에 객체를 생성하도록 하였다.

package net.saltfactory.tutorial;

import com.fasterxml.jackson.core.JsonProcessingException;

import com.fasterxml.jackson.databind.ObjectMapper;

import org.apache.log4j.Logger;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

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

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

import org.springframework.boot.test.\*;

import org.springframework.http.\*;

import org.springframework.http.client.ClientHttpRequestFactory;

import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;

import org.springframework.http.converter.FormHttpMessageConverter;

import org.springframework.http.converter.json.MappingJackson2HttpMessageConverter;

import org.springframework.mock.web.MockMultipartFile;

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

import org.springframework.util.MultiValueMap;

import org.springframework.web.client.RestTemplate;

import java.net.URI;

import java.nio.charset.Charset;

import java.util.ArrayList;

import java.util.List;

import static org.hamcrest.MatcherAssert.assertThat;

import static org.hamcrest.core.Is.is;

import static org.hamcrest.core.IsEqual.equalTo;

/\*\*

\* filename : RestTemplateTests.java

\* author : saltfactory<saltfactory@gmail.com>

\* created : 11/25/15

\*/

@RunWith(SpringJUnit4ClassRunner.class)

@SpringApplicationConfiguration(classes = SpringBootDemoApplication.class)

@WebIntegrationTest("server.port=0")

public class RestTemplateTests {

Logger logger = Logger.getLogger(this.getClass());

@Value("${local.server.port}")

int port;

@Autowired

ArticlesService articlesService;

private String baseUrl;

RestTemplate restTemplate;

@Before

public void setUp() {

restTemplate = new RestTemplate();

baseUrl = "<http://localhost>:" + String.valueOf(port);

}

private String jsonStringFromObject(Object object) throws JsonProcessingException {

ObjectMapper mapper = new ObjectMapper();

return mapper.writeValueAsString(object);

}

}

이제 **RestTemplate**을 사용하여 우리가 만든 REST 웹 서비스의 컨트롤러에 Http 요청을 해보자.

## GET /api/articles

MockMvc를 사용하여 테스트한 코드를 먼저 살펴보자.

@Test

public void testIndex() throws Exception {

List<Article> articles = articlesService.getArticles();

String jsonString = this.jsonStringFromObject(articles);

MvcResult result = mockMvc.perform(get("/api/articles"))

.andExpect(status().isOk())

.andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION\_JSON))

.andExpect(content().string(equalTo(jsonString)))

.andReturn();

logger.info(result.getResponse().getContentAsString());

}

이제 RestTemplate을 사용하여 실제 웹 서버로 GET 요청을 해보자. RestTemplate으로 Http GET 요청을 하는 방법는 여러가지가 있는데 크게 다음가 같다.

* **restTemplate.getForObject()** : 기본 Http Header를 사용며 결과를 객체로 반환 받는다.
* **restTemplate.getForEntity()** : 기본 Http Header를 사용하며 결과를 Http ResponseEntity로 반환 받는다.
* **restTemplate.exchange()** : Http Header 를 수정할 수 있고 결과를 Http ResponseEntity로 반환 받는다.
* **restTemplate.execute()** : Request/Response 콜백을 수정할 수 있다.

다음 예제는 [http://localhost:{port}/api/articles](http://localhost:%7bport%7d/api/articles) 로 RestTemplate을 사용하여 **HttpMethod.GET** 요청을 하는 테스트이다. 이 때 결과 반환값을 JSON 문자열로 받고 싶어서 결과 반환 값을 **String.class**로 지정하였다. **restTemplate.getForObject(uri,반환될 객체 타입)** 으로 보면 된다. RestTemplate의 HttpMethod.GET의 결과를 확인하기 위해서 로깅을 해보았다. 만약 RestTemplate가 웹 서버에 정상적인 요청을 했다면 Articles의 List 타입이 JSON으로 만들어져 보일것이다. 컨트롤러를 요청한 결과과 맞는지 확인하기 위해서 ArticlesService.getArticles()로 가져오는 결과와 비교했다.

@Test

public void testIndex() throws Exception {

URI uri = URI.create(baseUrl+ "/api/articles");

String responseString = restTemplate.getForObject(uri, String.class);

// 컨트롤러 결과를 로깅

logger.info(responseString);

// 컨트롤러 결과를 확인하기 위한 데이터 가져오기

List<Article> articles = articlesService.getArticles();

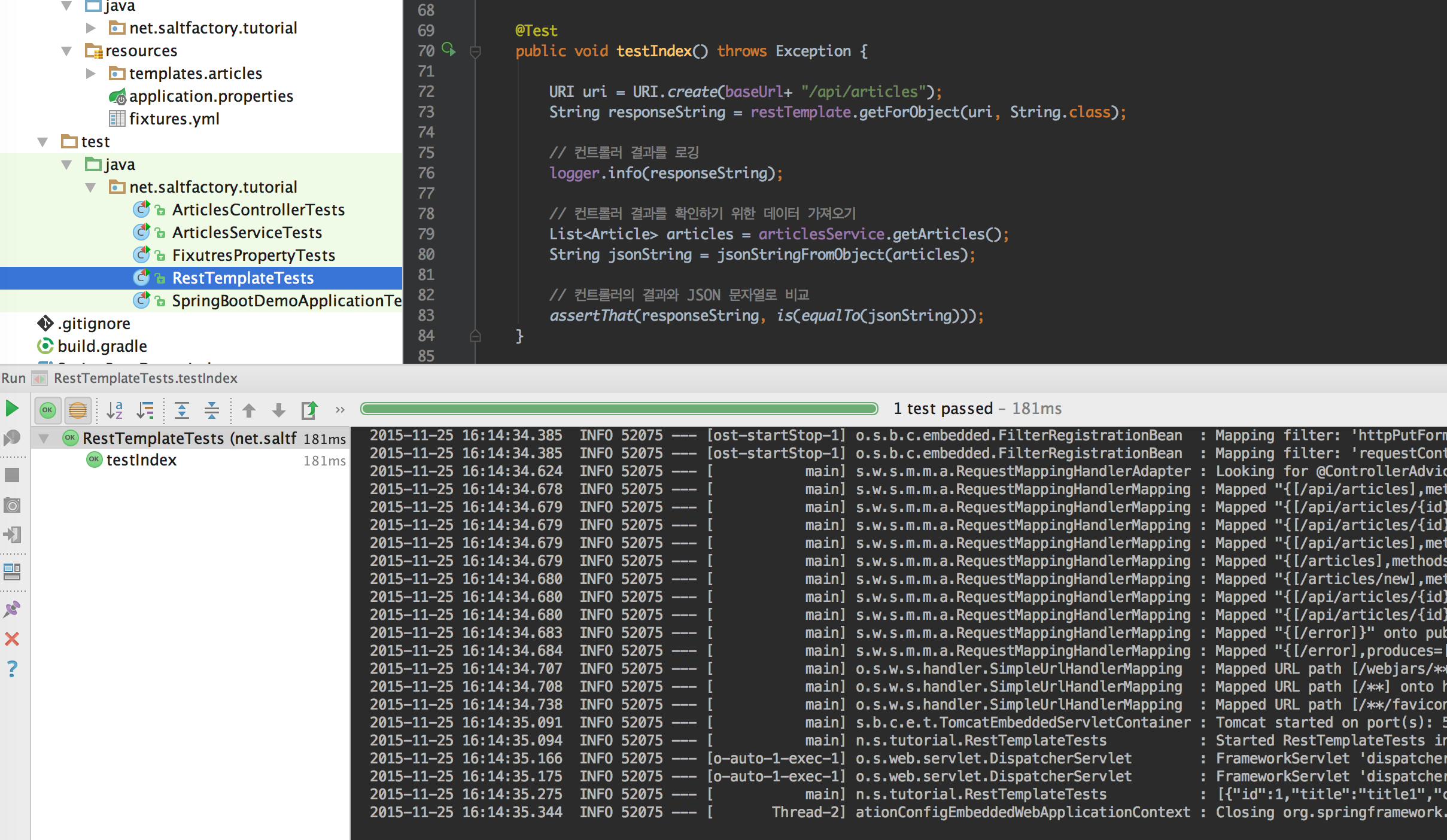
String jsonString = jsonStringFromObject(articles);

// 컨트롤러의 결과와 JSON 문자열로 비교

assertThat(responseString, is(equalTo(jsonString)));

}

RestTemplate.getForObject()로 HttpMethod.GET을 요청한 결과는 정상적이고 컨트롤르에서 반환한 JSON을 로깅을 통해서 확인할 수 있다.



RestTemplate을 사용하여 API 서버에서 JSON 문자열로 반환받게 되면 우리는 이것을 어플리케이션에서 객체 타입으로 다시 JSON 라이브러리를 사용하여 POJO 객체로 변환하는 작업을 할 것이다. 하지만 RestTemplate은 이런 과정을 자동으로 할 수 있다. 우리는 테스트를 위해서 반환 타입을 String.class로 지정하였지만 만약 API를 요청한 결과를 객체에 매핑한 Article 형태로 받고 싶다면 반환 타입에 객체 타입을 지정하면 자동으로 JSON 결과를 객체로 매핑해서 반환해준다.

위 테스트를 다음과 같이 수정해보자.

@Test

public void testIndex() throws Exception {

URI uri = URI.create(baseUrl+ "/api/articles");

// String responseString = restTemplate.getForObject(uri, String.class);

List<Article> resultArticles = Arrays.asList(restTemplate.getForObject(uri, Article[].class));

// 컨트롤러 결과를 로깅

// logger.info(responseString);

// 컨트롤러 결과를 확인하기 위한 데이터 가져오기

List<Article> articles = articlesService.getArticles();

// String jsonString = jsonStringFromObject(articles);

// 컨트롤러의 결과와 JSON 문자열로 비교

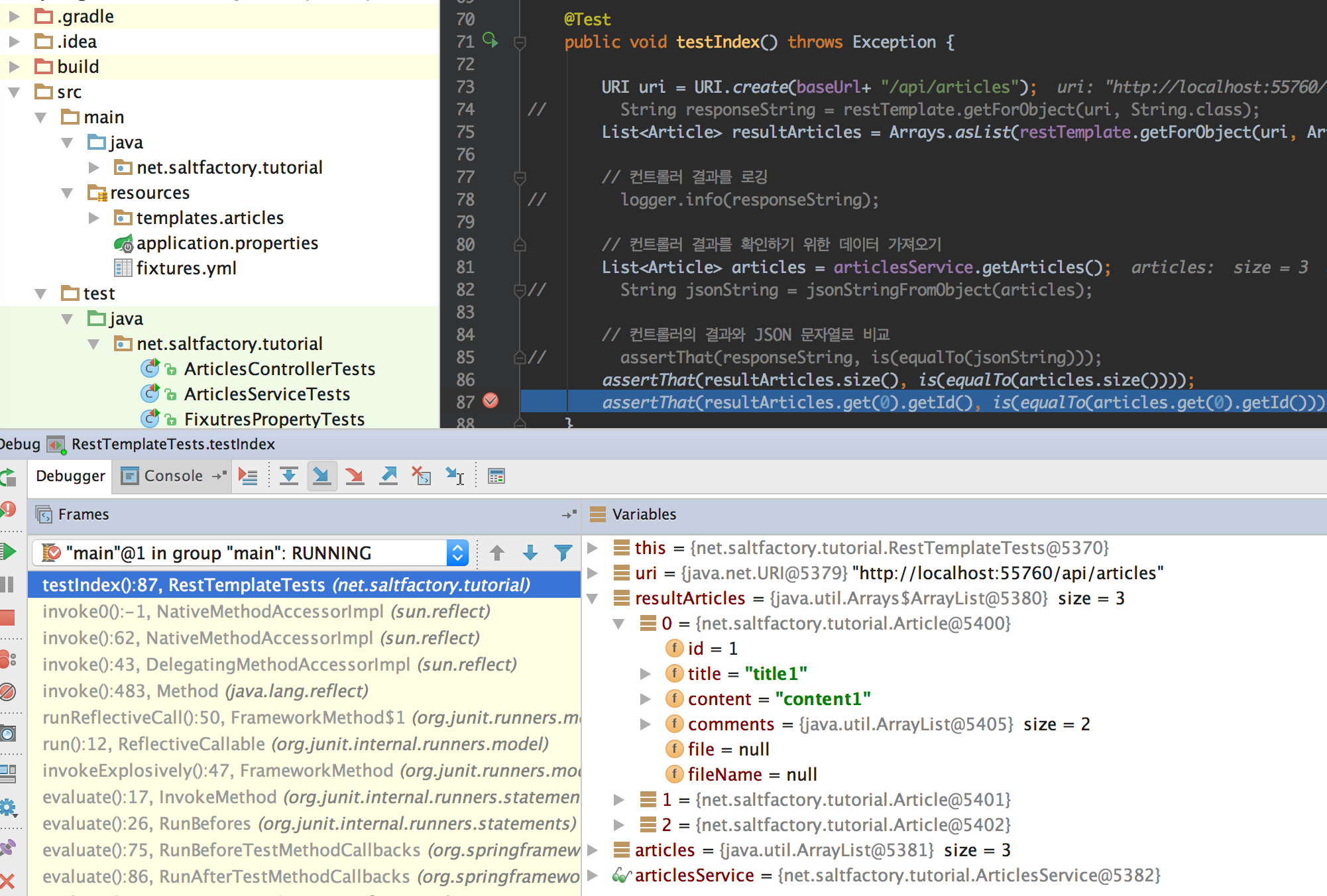
// assertThat(responseString, is(equalTo(jsonString)));

assertThat(resultArticles.size(), is(equalTo(articles.size())));

assertThat(resultArticles.get(0).getId(), is(equalTo(articles.get(0).getId())));

}

브레이크 포인트를 가지고 RestTemplate가 컨트롤러에서 반환한 결과를 살펴보자.



restTemplate.getObjectFor()에 반환되는 객체의 타입을 지정하면 JSON을 자동으로 반한되는 객체로 매핑해주는 것을 확인할 수 있다.

## POST /api/articles

앞에서 우리는 REST 서비스를 위한 컨트롤러에서 @RequestBody를 사용하여 객체를 JSON 타입으로 **HttpMethod.POST**를 보내는 것을 만들고 테스트를 통해 확인하였다. 먼저 MockMvc를 통해 테스트한 코드를 살펴보자. MockMvc 테스트를 통해서 보면 알 수 있듯 post() 요청을 할 때 content() 안에 Article 객체를 JSON 타입으로 변환해서 전송하는 것을 확인할 수 있다.

@Test

public void testCreate() throws Exception {

Article article = new Article();

article.setTitle("testing create article");

article.setContent("test content");

Comment comment = new Comment();

comment.setContent("test comment1");

List<Comment> comments = new ArrayList<>();

comments.add(comment);

article.setComments(comments);

String jsonString = this.jsonStringFromObject(article);

MvcResult result = mockMvc.perform(post("/api/articles")

.contentType(MediaType.APPLICATION\_JSON)

.content(jsonString))

.andExpect(status().isOk())

.andExpect(content().string(equalTo(jsonString))).andReturn();

logger.info(result.getResponse().getContentAsString());

}

이제 RestTemplate으로 실제 웹 서비스 형태로 테스트를 해보자. RestTemplate에서 POST를 요청하는 방법는 위에서 GET을 요청하는 방법과 비슷하다. 다만 getFor 로 시작하는 것을 postFor 로 바꿔주면 된다. 나머지는 동일하다.

* **restTemplate.postForObject()**
* **restTemplate.postForEntity()**
* **restTemplate.exchange()**
* **restTemplate.execute()**

MockMvc에서 가짜로 테스트하는 것과 달리 RestTemplate를 사용하여 실제 서버로 객체를 POST로 보낼 때는 Article의 객체를 그대로 넘겨주면 된다. 아주 간단하다.

@Test

public void testCreate() throws Exception {

URI uri = URI.create(baseUrl + "/api/articles");

Article article = new Article();

article.setTitle("testing create article");

article.setContent("test content");

Comment comment = new Comment();

comment.setContent("test comment1");

List<Comment> comments = new ArrayList<>();

comments.add(comment);

article.setComments(comments);

Article resultArticle = restTemplate.postForObject(uri, article, Article.class);

assertThat(resultArticle.getTitle(), is(equalTo(article.getTitle())));

// String responseString = restTemplate.postForObject(uri, article, String.class);

// String jsonString = jsonStringFromObject(article);

//

// assertThat(responseString, is(equalTo(jsonString)));

// logger.info(responseString);

}

테스트를 진행하면 성공적으로 새로운 Article이 POST로 전송되는 것을 확인할 수 있다. 하지만 한가지 중요한 조건이 있다. 이 때 웹 서버의 컨트롤러에서 이 POST 요청이 매핑되는 곳에서 Article 객체를 매핑하기 위해서는 반드시 **@RequetBody** 요청으로 되어 있어야 한다는 것이다. 다시 한번 서버에서 POST의 컨트롤러 코드를 살펴보자.

@RequestMapping(value = "/api/articles", method = RequestMethod.POST)

@ResponseBody

public Article create(@RequestBody Article article) {

return article;

}

나중에 다시 설명하겠지만 POST로 Article의 새로운 값을 받기 위해서 **@ModelAttribute**를 사용하는 것이 아니라 **@RequestBody**로 POST로 들어오는 객체를 매핑해야한다.

## DELETE /api/articles/{id}

다음은 **HttpMethod.DELETE**의 경우 RestTemplate에서 처리하는 방법을 살펴보자. 위에서 우리는 GET과 POST 요청을 할 때의 네이밍 규칙을 보고 DELETE 요청을 처리하기 위해서 restTemplate.deleteForObject() 라고 코드를 생각할지 모르지만 이것은 잘 못된 생각이다. RestTemplate에서 DELETE와 PUT에 관한 요청은 반환값을 가지지 않을 뿐만 아니라 파라미터 전송도 없다. 하지만 크게 당황하지 않아도 된다. 컨트롤러에서 반환값을 갖기 위해서는 template.exchange()를 사용하면 된다. 만약 반환값에 상관없이 단순하게 DELETE 요청을 할 때는 template.delete()을 사용하여 요청하면 된다.

* **restTemaplate.delete()**
* **restTemplate.exchange()**
* **restTemplate.execute()**

relateTemplate을 사용하여 HttpMethod.DELETE 요청을 처리하는 방법은 다음과 같다. 주석이 되어 있는 부분은 요청 후 반환값이 없을 때 간단하게 사용할 수 있는 방법이다.

만약 DELETE 요청 후 반환값이 필요하면 restTemplate.exchange()로 요청하면 되는데 이 것은 앞에서 restTemplate을 사용하는 방법과 달리 **HttpHeaders**와 **HttpEntity**를 사용하여 요청을 보내는 것을 확인할 수 있다. 그리고 exchange() 메소드에서 **HttpMethod.DELETE**를 보낸다고 method의 타입을 지정하는 것도 알 수 있단. 이유는 exchange()는 말 그대로 사용자가 직접 전달하는 것을 정의하여서 보내는 것이기 때문에 모둔 HttpMethod에서 동일하게 사용할 수 있는 방법이다.

@Test

public void testDelete() throws Exception {

long id = 1;

URI uri = URI.create(baseUrl + "/api/articles/" + id);

// Article article = articlesService.getArticle(id);

// restTemplate.delete(uri);

HttpHeaders headers = new HttpHeaders();

HttpEntity entity = new HttpEntity(headers);

ResponseEntity<String> responseEntity = restTemplate.exchange(uri, HttpMethod.DELETE, entity, String.class);

String jsonString = jsonStringFromObject(articlesService.deleteArticle(id));

assertThat(responseEntity.getStatusCode(), is(HttpStatus.OK));

assertThat(responseEntity.getBody(), is(equalTo(jsonString)));

logger.info(responseEntity.getBody());

}

## PUT /api/articles/{id}

**HttpMethod.PUT**의 요청 또한 위의 **HttpMethod.DELETE**의 방법과 동일하다 단지 PUT은 데이터를 업데이트하기 위한 요청을 하기 때문에 객체를 함께 보내는 것이 다르다. 이 때 주의할 점은 객체를 보낼 때 **HttpEntity**에 header와 함께 보내는 것을 주의한다.

@Test

public void testPut() throws Exception {

long id = 1;

URI uri = URI.create(baseUrl + "/api/articles/" +id);

Article article = articlesService.getArticle(id);

article.setTitle("testing create article");

article.setContent("test content");

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

HttpEntity<Article> entity = new HttpEntity(article, headers);

ResponseEntity<String> responseEntity = restTemplate.exchange(uri, HttpMethod.PUT, entity, String.class);

String jsonString = jsonStringFromObject(article);

assertThat(responseEntity.getStatusCode(), is(HttpStatus.OK));

assertThat(responseEntity.getBody(), is(equalTo(jsonString)));

}

## PATCH /api/articles/{id}

우리는 앞에서 컨트롤러를 만들 때 **PATCH** 는 **PUT**과 유사하다고 말했다. 그래서 RestTemplate으로 PUT을 보내는 방법고 동리하게 하며 exchange()에서 method를 **HttpMethod.PATCH**로 변경하면 될 것이라 생각하기 쉽다. 하지만 안타깝게 기본적으로 RestTemplate은 [POST, GET, PUT, DELETE, OPTIONS](http://www.springframework.net/rest/doc-latest/reference/html/resttemplate.html) 만 제공한다. 그래서 만약 이렇게 코드를 작성하면 다음과 같은 에러를 보게 된다.

@Test

public void testPatch() throws Exception {

long id = 1;

URI uri = URI.create(baseUrl + "/api/articles/" +id);

Article article = articlesService.getArticle(id);

article.setTitle("testing create article");

article.setContent("test content");

HttpHeaders headers = new HttpHeaders();

HttpEntity<Article> entity = new HttpEntity(article, headers);

ResponseEntity<String> responseEntity = restTemplate.exchange(uri, HttpMethod.PATCH, entity, String.class);

String jsonString = jsonStringFromObject(article);

assertThat(responseEntity.getStatusCode(), is(HttpStatus.OK));

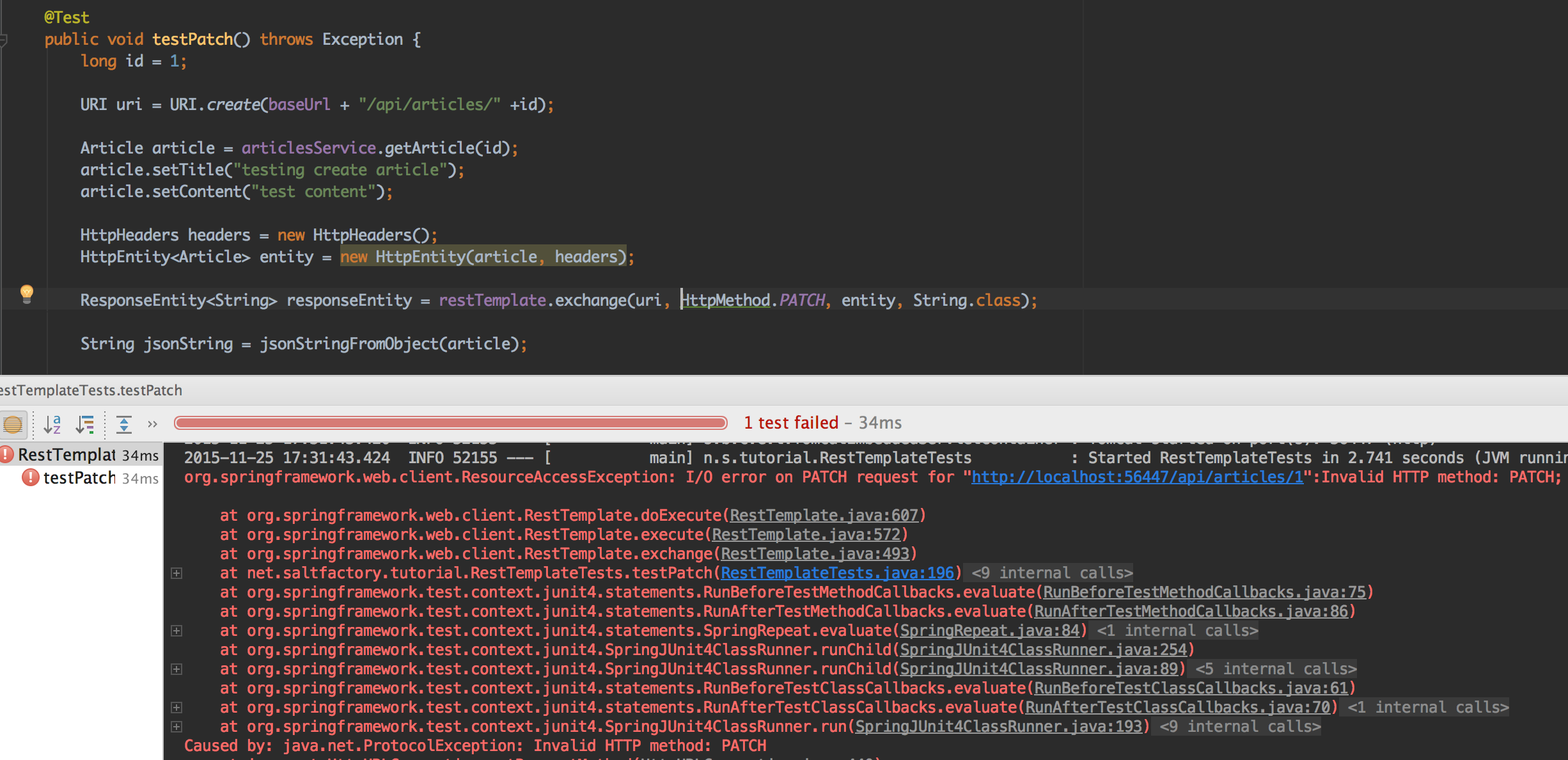
assertThat(responseEntity.getBody(), is(equalTo(jsonString)));

}

위 코드를 실행하면

org.springframework.web.client.ResourceAccessException: I/O error on PATCH request for "<http://localhost:56447/api/articles/1>":Invalid HTTP method: PATCH; nested exception is java.net.ProtocolException: Invalid HTTP method: PATCH

에러를 보게 된다. 이것은 RestTemplate이 기본적으로 **PATCH** 메소드를 지원하고 있지 않기 때문이다.



RestTemplate은 **ClientHttpRequest** 감싸고 있는 모듈인데 기본적으로 [SimpleClientHttpRequestFactory](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/client/SimpleClientHttpRequestFactory.html)로 만들어져 있다. 우리는 [Apache HttpComponents HttpClient](http://hc.apache.org/httpcomponents-client-ga/)를 만들 때 사용하는 [HttpComponentsClientHttpRequestFactory](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/client/HttpComponentsClientHttpRequestFactory.html)으로 ClientHttpRequest를 바꾸어서 사용할 것이다. RestTemplate를 생성할 때 ClientHttpRequestFactory를 변경하여 생성한다. HttpComponentsClientHttpRequestFactory는 Apache HttpClient 라이브러가 필요하다. guild.gradle 파일을 열어서 라이브러리를 추가한다.

buildscript {

ext {

springBootVersion = '1.3.0.RELEASE'

}

repositories {

mavenCentral()

maven { url '<http://repo.spring.io/plugins-release>' }

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")

classpath 'org.springframework.build.gradle:propdeps-plugin:0.0.7'

}

}

apply plugin: 'java'

apply plugin: 'eclipse'

apply plugin: 'idea'

apply plugin: 'spring-boot'

jar {

baseName = 'spring-boot-demo'

version = '0.0.1-SNAPSHOT'

}

sourceCompatibility = 1.8

targetCompatibility = 1.8

repositories {

mavenCentral()

}

dependencies {

compile('org.springframework.boot:spring-boot-starter-thymeleaf')

compile('org.springframework.boot:spring-boot-starter-web')

testCompile('org.springframework.boot:spring-boot-starter-test')

compile('org.springframework.boot:spring-boot-configuration-processor')

compile('org.apache.httpcomponents:httpclient:4.5.1')

}

compileJava.dependsOn(processResources)

configure(allprojects) {

apply plugin: 'propdeps'

apply plugin: 'propdeps-maven'

apply plugin: 'propdeps-idea'

apply plugin: 'propdeps-eclipse'

}

eclipse {

classpath {

containers.remove('org.eclipse.jdt.launching.JRE\_CONTAINER')

containers 'org.eclipse.jdt.launching.JRE\_CONTAINER/org.eclipse.jdt.internal.debug.ui.launcher.StandardVMType/JavaSE-1.8'

}

}

task wrapper(type: Wrapper) {

gradleVersion = '2.9'

}

gradle로 의존성있는 라이브러리를 자동으로 다운로드 받은 후 RestTemplate에서 **PATCH**을 사용하기 위해서 RestTemplate을 **HttpComponentsClientHttpRequestFactory**을 가지고 새롭게 생성한다.

@Test

public void testPatch() throws Exception {

long id = 1;

URI uri = URI.create(baseUrl + "/api/articles/" +id);

Article article = articlesService.getArticle(id);

article.setTitle("testing create article");

article.setContent("test content");

HttpHeaders headers = new HttpHeaders();

HttpEntity<Article> entity = new HttpEntity(article, headers);

ClientHttpRequestFactory httpRequestFactory = new HttpComponentsClientHttpRequestFactory();

restTemplate = new RestTemplate(httpRequestFactory);

ResponseEntity<String> responseEntity = restTemplate.exchange(uri, HttpMethod.PATCH, entity, String.class);

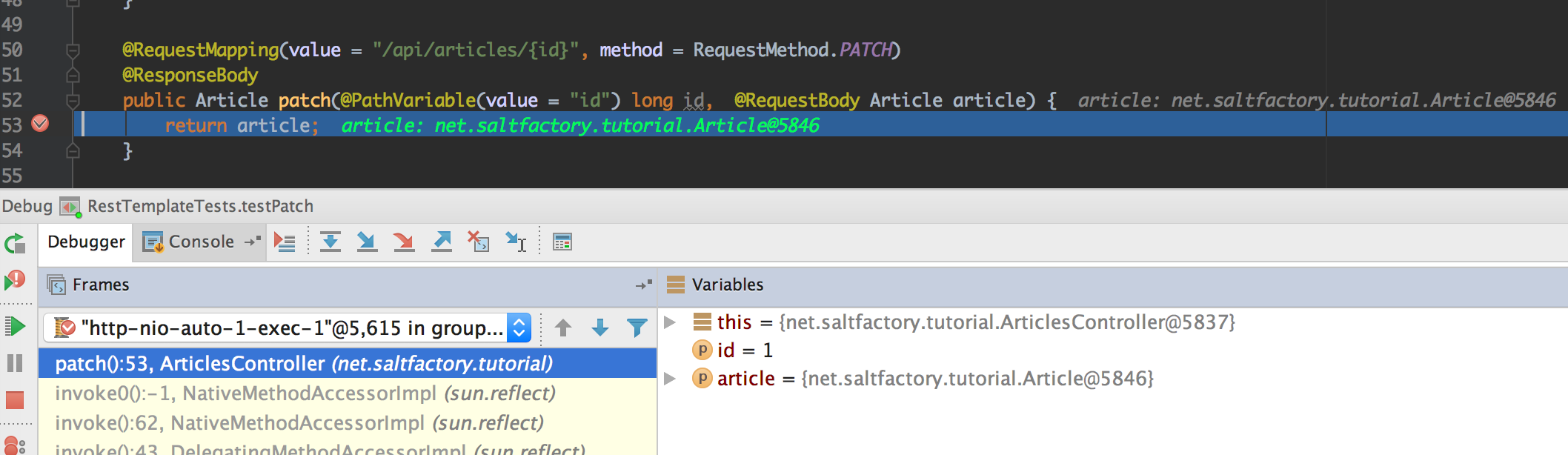
String jsonString = jsonStringFromObject(article);

assertThat(responseEntity.getStatusCode(), is(HttpStatus.OK));

assertThat(responseEntity.getBody(), is(equalTo(jsonString)));

}

브레이크 포인트를 사용하여 컨트롤러에서 확인하면 RestTemplate을 사용하여 요청한 PATCH 요청이 정상적으로 컨트롤러에 요청되는 것을 확인할 수 있다.



## 결론

실제 우리는 Spring을 사용하는 프로젝트에서 컴포넌트 내에서 API 서버로 REST 요청을 한는 작업을 **RestTemplate**을 사용하여 구현하였다. RestTemplate은 이전에 상요하던 Apache의 **HttpClient** 보다 간단하고 짧은 코드로 쉽게 API 서버로 REST 요청을 처리할 수 있다. JSON 형태의 문자열 결과만 받아오는 것 뿐만 아니라 JSON객체나 POJO 객체로 변환하는 작업 없이 컴포넌트 내에서 사용하는 객체로 바로 매핑하여 사용할 수 있다는 점에서 객체 변환 코드 상당 수를 줄일 수 있었다. RestTemplate은 크레 getForObject(), postForObject()와 같이 Object로 매핑할 수 있는 요청과 , getForEntity(), postForEntity()와 같이 Entity로 매핑할 수 있는 요청을 할 수 있다 그리고 delete()나 put()과 같이 반환 없는 요청을 할 수 있다. exchange()나 execute() 같은 함수는 모든 Http method 요청에 사용할 수 있고 사용자가 요청하는 객체를 새롭게 정의하거나 로직을 변경할 수도 있다. RestTemplate은 **PATCH** 메소드를 기본적으로 지원하고 있지 않기 때문에 PATCH 를 지원하기 위해서는 생성할 때 Apache HttpClient 라이브를 사용하여 만든 **HttpComponentsClientHttpRequestFactory**를 사용하여 생성하여 사용하면 된다.

다음에는 RestTemplate을 사용하여 FORM 객체를 컨트롤러의 **@ModelAttribute** 객체로 매핑하는 방법과 **Multipart/Form-data**를 사용하여 파일 업로드를 하는 방법에 대해서 소개할 예정이다.

## 소스코드

* <https://github.com/saltfactory/saltfactory-java-tutorial/tree/spring-boot-resttemplate/SpringBootDemo>

## 참조

1. <http://www.springframework.net/rest/doc-latest/reference/html/resttemplate.html>
2. <https://github.com/spring-projects/rest-shell/issues/21>
3. <https://hc.apache.org/httpcomponents-client-ga/>