**Testing improvements in Spring Boot 1.4**

<https://spring.io/blog/2016/04/15/testing-improvements-in-spring-boot-1-4>

- 완전히 분리된 구조로 각 독립 파트가 단위 테스트 되도록 해라.

- 생성자 주입(constructor injection)을 사용하여 오브젝트를 인스턴스화해라.

필드 주입(field injection)을 사용하지 마라. 테스트 코드를 작성하기 어렵다.

스프링 4.3 에서 생성자 주입을 사용하는 컴포넌트는 작성하기 매우 쉬우며 @Autowired 가 더이상 필요없다.

하나의 생성자가 있는 경우 스프링은 묵시적으로 그걸 autowire 대상으로 간주하게 된다.

● 아래와 같은 컴포넌트가 있는 경우 (생성자가 1개임)

[?](http://home.zany.kr:9003/board/bView.asp?bCode=52299899&aCode=13734)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Component  public class MyComponent {        private final SomeService service;        public MyComponent(SomeService service) {          this.service = service;      }  } |

● MyComponent 테스트를 위해 아래와 같이 간단히 직접 생성하여 메서드를 호출하면 된다.

[?](http://home.zany.kr:9003/board/bView.asp?bCode=52299899&aCode=13734)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Test  public void testSomeMethod() {      SomeService service = mock(SomeService.class);      MyComponent component = new MyComponent(service);      // setup mock and class component methods  } |

**● Spring Boot 1.3 에서는...**  
● @ContextConfiguration 어노테이션과 SpringApplicationContextLoader 조합으로 아래와 같이 사용했을 것이다.

[?](http://home.zany.kr:9003/board/bView.asp?bCode=52299899&aCode=13734)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @RunWith(SpringJUnit4ClassRunner.class)  @ContextConfiguration(classes=MyApp.class, loader=SpringApplicationContextLoader.class)  public class MyTest {        // ...    } |

● 아래와 같이 @SpringApplicationConfiguration 어노테이션을 사용했을수도 있고,

[?](http://home.zany.kr:9003/board/bView.asp?bCode=52299899&aCode=13734)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @RunWith(SpringJUnit4ClassRunner.class)  @SpringApplicationConfiguration(MyApp.class)  public class MyTest {        // ...    } |

● @IntegrationTest 어노테이션을 조합했을 수도 있다.

[?](http://home.zany.kr:9003/board/bView.asp?bCode=52299899&aCode=13734)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @RunWith(SpringJUnit4ClassRunner.class)  @SpringApplicationConfiguration(MyApp.class)  @IntegrationTest  public class MyTest {        // ...    } |

● @WebIntegrationTest 어노테이션을 사용했을수도 있을 것이다. (또는 @IntegrationTest + @WebAppConfiguration 조합도 가능하다)

[?](http://home.zany.kr:9003/board/bView.asp?bCode=52299899&aCode=13734)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @RunWith(SpringJUnit4ClassRunner.class)  @SpringApplicationConfiguration(MyApp.class)  @WebIntegrationTest  public class MyTest {        // ...    } |

@WebIntegrationTest(randomPort=true) 로 랜덤 포트로 실행하거나,

@IntegrationTest("myprop=myvalue") 나 @TestPropertySource(properties="myprop=myvalue") 와 같이 프로퍼티값을 사용했을수도 있을것이다.

**● Spring Boot 1.4 에서는...**

Spring Boot 1.4 에서는 좀 더 단순해져서, @SpringBootTest 어노테이션 단 하나만 사용하여

일반적인 테스트를 수행할 수 있다. 몇가지 인자들은 뒤에서 언급한다.

**● Spring Boot 1.4 통합 테스트는 아래와 같은 형태이다.**

[?](http://home.zany.kr:9003/board/bView.asp?bCode=52299899&aCode=13734)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @RunWith(SpringRunner.class)  @SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT)  public class MyTest {        // ...    } |

@RunWith(SpringRunner.class) 는 JUnit 에게 스프링 테스트 지원(Spring's testing support)을 사용하겠다고 알려주는 구문이다.

SpringRunner 는 SpringJUnit4ClassRunner 의 새로운 이름이다. 좀 더 눈에 잘 들어온다.

@SpringBootTest 어노테이션은 "스프링 테스트 지원의 부트스트랩 입니다 (bootstrap with Spring Boot's support)" 라는 걸 의미한다.

(e.g. load application.properties and give me all the Spring Boot goodness)

webEnvironment 속성은 테스트를 위해 지정한 웹 환경 설정을 허용하겠다는 뜻이다.

MOCK 서블릿 환경 또는 랜덤(RANDOM\_PORT)이나 지정(DEFINED\_PORT)포트로 실행되는

실제 HTTP 서버로 테스트를 시작할 수 있다.

지정한 환경을 불러오기 위해 @SpringBootTest 의 속성을 사용할 수 있다.

테스트는 먼저 @Configuration 이 붙은 클래스들을 로드하려고 시도할 것이고,

실패한다면 @SpringBootApplication 가 붙은 클래스를 찾을 것이다.

@SpringBootTest 어노테이션은 환경으로 정의될 추가 속성을 지정할 수 있는 "properties" 속성을 제공한다.

프로퍼티들은 @TestPropertySource 어노테이션과 완전이 동일한 방법으로 로드된다.

**● 아래는 실제 REST 엔드포인트를 호출하는 예제이다.**

[?](http://home.zany.kr:9003/board/bView.asp?bCode=52299899&aCode=13734)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @RunWith(SpringRunner.class)  @SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT)  public class MyTest {        @Autowired      private TestRestTemplate restTemplate;        @Test      public void test() {          this.restTemplate.getForEntity(              "/{username}/vehicle", String.class, "Phil");      }  } |

@SpringBootTest 가 사용되었기 때문에 TestRestTemplate 이 bean 으로써 사용될 수 있음에 주목하자.

http://localhost:${local.server.port} 의 상대 경로(relative paths)로 처리된다.

테스트시 사용되는 서버에 실제 포트를 주입하기 위해 @LocalServerPort 어노테이션을 사용할 수도 있다.

실제 시스템에서 테스트할 때, 지정한 빈(bean)들을 목아웃(mock out)하는 것이 도움이 된다는 걸 알 수 있다.

서비스 시뮬레이션을 포함하는 목킹(mocking)의 일반적인 시나리오는

테스트가 실행 중일 때는 사용할 수 없다는 것 또는 라이브 시스템에서 발생시키기 어려운 테스트 실패 시나리오이다.

- Common scenarios for mocking include simulating services that you can’t use when running tests,

- or testing failure scenarios that are difficult to trigger in a live system.

Spring Boot 1.4 에서는 이미 존재하는 빈(bean)을 대체하거나 새로 생성하는 Mockito 를 손쉽게 만들수 있다.

- With Spring Boot 1.4 you can easily create a Mockito mocks that can replace an existing bean,

- or create a new one

[?](http://home.zany.kr:9003/board/bView.asp?bCode=52299899&aCode=13734)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | @RunWith(SpringRunner.class)  @SpringBootTest(webEnvironment = WebEnvironment.RANDOM\_PORT)  public class SampleTestApplicationWebIntegrationTests {        @Autowired      private TestRestTemplate restTemplate;        @MockBean      private VehicleDetailsService vehicleDetailsService;        @Before      public void setup() {          given(this.vehicleDetailsService.              getVehicleDetails("123")          ).willReturn(              new VehicleDetails("Honda", "Civic"));      }        @Test      public void test() {          this.restTemplate.getForEntity("/{username}/vehicle",              String.class, "sframework");      }  } |

- VehicleDetailsService 의 모형(mock)으로 Mockito 를 생성한다.

- 빈(bean)으로써 ApplicationContext 에 주입한다.

- 테스트의 필드로 주입한다.

- setup 메서드에 행위자(behavior)를 기술한다.

- Trigger something that will ultimately call the mock.

모형(mocks)은 테스트를 통과하면서 자동으로 리셋될 것이다.

SpringBoot 1.4.0 Test 적용하기 (1)  
출처: <http://jojoldu.tistory.com/33> [기억보단 기록을]

최근에 SpringBoot가 1.4.0 버전으로 릴리즈 되었다.



이번 버전에 대한 공부도 할겸 그동안 계속 미루고 있었던 TDD 공부도 시작할겸해서

SpringBoot 1.4.0 버전을 기준으로 한 TDD 내용을 정리하려고 한다.

사실 페이스북이나 슬랙에서 TDD에 대한 찬양론은 많이 보고 들었지만, 개인적으로 사내에서 한번도 써보질 못해서 내가 이런 내용을 정리한다는것에 걱정이 먼저 앞선다.

그래도 이렇게 공개를 하는건,

틀린 내용을 들키는게 무서워서 감추고 있다가 **잘못된 지식을 계속 가지고 가는것 보다는 얼른 틀리고 정확한 내용을 인지하는게 낫다고 생각**하기 때문이다.



(틀린건 가감없이 사랑의 매를!!)

여튼 지금부터 TDD에 대한 내용을 정리하겠다.

순서는

@DataJpaTest -> @WebMvcTest -> SecurityTest -> @SpringbootTest 로 진행될 예정이며, 시간이 되면 javascript까지 해보려고 한다.

모든 코드는 [Github](https://github.com/jojoldu/blog-code/tree/master/springboot-test)에서 확인할수 있다.

그럼 이제 진짜로 시작!

# SpringBoot 1.4 Test 사용하기

[공식문서](http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html)를 참고하며 기록하는 SpringBoot Test 적용하기

TDD를 기반으로 프로젝트를 시작하는 예제   
1. 테스트 코드를 통해 Entity와 Dao 구현   
2. 테스트 코드를 통해 Controller 구현   
3. 테스트 코드를 통해 Oauth 인증 구현   
TDD로 실전 프로젝트를 해본적이 없어 개인적으로 만들 서비스의 예행연습으로 보고 진행함을 먼저 얘기한다.

## 1. @DataJpaTest

* SpringBoot에서 **JPA만 테스트**할 수 있도록 제공하는 어노테이션
* 개발의 첫 단계인 Entity 설계 단계에서 불필요한 코드 작성 없이, Entity간의 관계 설정 및 기능 테스트가 가능해졌다.
  + 예를 들자면 View를 만들거나, Controller를 작성하는 것 등등 **Entity 설계 확인을 위한 코드** 작성이 필요없어졌다.
* 사용법은 간단하다.

@RunWith(SpringRunner.class) //Junit 테스트 선언

@DataJpaTest // DataJpaTest 선언

public class DataJpaTest {

/\*

Repository == Dao

본인이 테스트하려는 Dao를 선언하고 기능을 테스트 하면 된다.

아래는 여기 프로젝트에서 사용한 코드의 일부이다.

\*/

@Autowired

private MemberRepository memberRepository;

@Autowired

private PostRepository postRepository;

@Autowired

private CommentRepository commentRepository;

}

* H2 + JPA + @DataJpaTest = AWESOME!!
* 단위 테스트가 끝날때 마다 자동으로 DB를 롤백시켜 준다 (우앙!)
* 여기에서 사용할 예제 Entity는 아래와 같다
  + 사용자 : Member
  + 글 : Post
  + 댓글 : Comment
* JPA 참고 자료
  + 아라한사님이 [번역하신 공식 문서](http://arahansa.github.io/docs_spring/jpa.html)
  + 김영한님의 [자바 ORM 표준 JPA 프로그래밍](http://www.yes24.com/24/goods/19040233)

### 1.1 상황1

* Post와 Comment간의 관계 설정
* 하나의 글은 여러개의 댓글을 가질 수도 있고, 없을 수도 있다.
* 하나의 글을 조회하면 해당하는 댓글이 같이 와야 한다.

ManyToOne(다대일) 양방향으로 해결

* Code (자세한 코드는 생략)

// Post 클래스

@Entity

public class Post {

@OneToMany(mappedBy="post", cascade = CascadeType.ALL)

private List<Comment> comments;

}

// Comment 클래스

@Entity

public class Comment {

@ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)

@JoinColumn(foreignKey = @ForeignKey(name = "fk\_comment\_post"))

private Post post;

}

@Test

public void test\_Post와Comment관계정의() throws Exception {

Post savedPost = postRepository.save(post);

savedPost.addComment(comment); // 글에 댓글 추가

comment.setPost(savedPost);

commentRepository.save(comment); // 댓글에 글 추가

Post firstPost = postRepository.findOne(1L);

Comment firstComment = commentRepository.findOne(1L);

assertThat(savedPost.getContent(), is("content"));

assertThat(savedPost.getComments().get(0).getContent(), is(firstComment.getContent()));

}

* OneToMany(일대다) 를 왜 쓰지 않은걸까? 예를 들어 Comment를 수정해야하는일이 생길 경우

// OneToMany(일대다) 단방향

Post post = postRepository.findOne(1L);

List<Comment> comments = post.getComments();

Comment comment = comments.get(0);

comment.setXXX(); // update

// ManyToOne(다대일) 양방향

Comment comment = commentRepository.findOne(1L);

comment.setXXX(); // update

* 즉, 일대다 단방향일 경우 '다'에 속해있는 객체 하나를 수정하기 위해선 '일'을 조회하고 '일' 내부에 있는 '다'에서 원하는 객체를 다시 뽑아내야하는 과정이 필요하다.
* 반변에 다대일 양방향일 경우 수정을 원하는 하위 객체를 바로 수정할 수가 있기 때문에 **다대일 양방향으로 해결하는것을 권장**한다.

### 1.2 상황2

* Member와 Comment간의 관계 설정
* 사용자가 직접 댓글 작성 기능 구현
* 글이 올라오면, 사용자는 해당 글에 댓글을 남길수 있다.
* 한명의 사용자는 여러개의 글에 여러개의 댓글을 작성할 수 있다.
* 사용자 정보 조회시 해당 사용자가 작성한 댓글을 모두 조회할 수 있어야 한다.

상황 1.1과 동일한 ManyToOne(다대일) 양방향으로 해결   
객체간 연간관계는 **양방향이란게 없기 때문**에, 이를 해결하기 위해 단방향 2개(Comment -> Member와 Member -> Comment)를 사용한것이라고 보면 된다.

* Code (자세한 코드는 생략)

// Member 클래스

@Entity

public class Member {

@OneToMany(mappedBy="member", cascade = CascadeType.ALL)

@OrderBy("idx DESC")

private List<Comment> comments;

}

// Comment 클래스

@Entity

public class Comment {

@ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)

@JoinColumn(foreignKey = @ForeignKey(name = "fk\_comment\_member"))

private Member member;

}

// Test 코드

@Test

public void test\_Member와Comment관계정의() throws Exception {

Post savedPost = postRepository.save(post);

Member savedMember = memberRepository.save(member);

savedPost.addComment(comment);

savedMember.addComment(comment);

comment.setPostAndMember(savedPost, savedMember);

commentRepository.save(comment);

Post afterPost = postRepository.findOne(1L);

Member afterMember = memberRepository.findOne(1L);

assertThat(afterPost.getComments().get(0).getContent(), is("댓글"));

assertThat(afterMember.getComments().get(0).getContent(), is("댓글"));

assertThat(commentRepository.findAll().size(), is(1)); // savedPost와 savedMember에 각각 addComment를 했지만 결국 comment는 1개가 들어간것을 확인

}

### 1.3 상황3

* Member와 Post간의 관계 설정
* 사용자별 즐겨찾기 기능 구현
* 사용자는 0개 혹은 다수의 글를 가질수 있다.
* 글은 꼭 사용자가 있어야 하는것은 아니다.
* 하나의 글은 여러 사용자에 참조 될 수도 있다.

JoinTable을 통해 해결   
사용자와 글은 선택적(Optional) 참조이므로 joinColumn 으로는 해결할 수가 없다.

* Code (자세한 코드는 생략)

// Member 클래스

@Entity

public class Member {

@OneToMany()

@JoinTable(name="MEMBER\_POST",

joinColumns=@JoinColumn(name="MEMBER\_IDX"),

inverseJoinColumns=@JoinColumn(name="POST\_IDX"))

@OrderBy("idx DESC")

private List<Post> favorites;

}

// Post 클래스는 변경 없음

// Test 코드

@Test

public void test\_Post와Member관계정의() throws Exception {

Member member2 = new Member("test@gmail.com", new ArrayList<>(), new LinkedHashSet<>());

Post savedPost = postRepository.save(post);

member.addPost(savedPost);

member2.addPost(savedPost);

Member savedMember = memberRepository.save(member);

Member savedMember2 = memberRepository.save(member2);

assertThat(savedMember.getFavorites().stream().findFirst().orElse(new Post()).getContent(), is("content")); // 1번 사용자의 1번 글이 post인지 확인

assertThat(savedMember2.getFavorites().stream().findFirst().orElse(new Post()).getContent(), is("content")); // 2번 사용자의 1번 글이 post인지 확인

}

### 1.4 상황4

* ORM에서 컬렉션 사용법
* 사용자는 중복된 글을 가질수 없다. 여러개의 글을 가질순 없지만 고유하게 하나씩 있어야만 한다.
* 이럴 경우 Member.favorites가 List타입일 경우 중복 제거를 위한 비지니스 로직이 추가되어야 한다.
* 중복제거를 로직으로 해결하지말고 **자료구조로 해결**하자

Member.favorites를 List에서 Set으로 변경하여 해결

* Code (자세한 코드는 생략)

// Member 클래스

@Entity

public class Member {

@OneToMany()

@JoinTable(name="MEMBER\_POST",

joinColumns=@JoinColumn(name="MEMBER\_IDX"),

inverseJoinColumns=@JoinColumn(name="POST\_IDX"))

@OrderBy("idx DESC")

private Set<Post> favorites;

}

// Post 클래스는 변경 없음

// Test 코드

@Test

public void test\_oneToMany에서Set과List차이() throws Exception {

Post savedPost = postRepository.save(post);

member.addPost(savedPost);

member.addPost(savedPost);

Member savedMember = memberRepository.save(member);

assertThat(savedMember.getFavorites().size(), is(1)); // 2개의 Post를 넣었지만 결국 중복된게 제거되서 1개만 등록된것을 확인할수 있다.

}

### 1.5 상황5

* 상속관계 매핑
* Post가 Job, Tech, Essay 라는 3가지 타입으로 분류되도록 해야한다.
* 3타입 모두 가지고 있는 컬럼은 같다. (idx, content, updateDate, comments)
* 객체지향적 코드 작성을 위해 각 클래스는 분리하길 원한다.

JPA의 상속관계중 단일테이블전략을 사용한다.   
조인전략의 경우 3타입이 서로 다른 컬럼을 1개이상 가지고 있으며, 차후 별도로 컬럼이 추가/삭제 될 가능성이 높은 경우에 고려해볼만 하다.   
하지만 일반적으로 동일한 속성들을 가지고 있는 경우엔 단일 테이블 전략이 더 좋다   
조회 속도 역시 불필요한 조인이 없어 일반적으로 더 빠르다.

* Code

// Post 클래스

@Entity

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE) // 상속관계(단일테이블) 선언

@DiscriminatorColumn(name="DTYPE") // 하위 엔티티들을 구분하는 컬럼명 (default가 DTYPE라서 생략가능, 정보전달을 위해 명시함)

public abstract class Post { }

// Job 클래스 (Tech, Essay 역시 동일함)

@Entity

@DiscriminatorValue("JOB") // DTYPE에 저장될 값

public class Job extends Post { }

// PostRepository 인터페이스

// 제네릭을 사용하여 하위 인터페이스 타입 보장

public interface PostRepository<T extends Post> extends JpaRepository<T, Long>{}

// JobRepository 인터페이스

public interface JobRepository extends PostRepository<Job>{}

// Test 코드

@Test

public void test\_상속관계() throws Exception {

jobRepository.save(new Job("잡플래닛", LocalDateTime.now(), new ArrayList<>()));

techRepository.save(new Tech("OKKY", LocalDateTime.now(), new ArrayList<>()));

essayRepository.save(new Essay("임백준", LocalDateTime.now(), new ArrayList<>()));

Job savedJob = jobRepository.findAll().get(0);

Tech savedTech = techRepository.findAll().get(0);

Essay savedEssay = essayRepository.findAll().get(0);

assertThat(savedJob.getContent(), is("잡플래닛"));

assertThat(savedTech.getContent(), is("OKKY"));

assertThat(savedEssay.getContent(), is("임백준"));

}

1번 스탭을 통해 Repository (Dao) 의 기능테스트가 끝이났으니 Controller 구현 & 테스트를 진행해보자

[SpringBoot 1.4.0 Test 적용하기(2)](http://jojoldu.tistory.com/34)  
  
출처: <http://jojoldu.tistory.com/33> [기억보단 기록을]

<http://jojoldu.tistory.com/34>

### [SpringBoot 1.4.0 Test 적용하기 (2)](http://jojoldu.tistory.com/34)

창천향로 창천향로 2016.09.08 21:22

[전편](http://jojoldu.tistory.com/33)에 이어 2번째 테스트를 진행하겠다.

전편과 동일하게 이번 내용 역시 모든 코드는 [Github](https://github.com/jojoldu/blog-code/tree/master/springboot-test) 에서 확인할 수 있다.

# SpringBoot 1.4 Test 사용하기 (2)

## 2. @WebMvcTest

* DataJpa 어노테이션이 Repository (Dao) 에 대한 테스트라면 WebMvcTest는 Controller을 위한 테스트 어노테이션이다.
* Scan 대상은 아래와 같다.
  + @Controller
  + @ControllerAdvice
  + @JsonComponent
  + Filter
  + WebMvcConfigurer and HandlerMethodArgumentResolver
* MockMvc를 자동으로 지원하고 있어 별도의 HTTP 서버 없이 Controller 테스트를 진행할 수 있다.
* 사용법 역시 간단하다.

@RunWith(SpringRunner.class)

@WebMvcTest(HomeController.class)

public class WebMvcTest {

@Autowired

private MockMvc mvc;

@Test

public void test\_샘플() throws Exception {

this.mvc.perform(get("/hello").accept(MediaType.TEXT\_PLAIN)) // /hello 라는 url로 text/plain 타입을 요청

.andExpect(status().isOk()) // 위 요청에 따라 결과가 status는 200이며

.andExpect(content().string("Hello World")); // response body에 "Hello World" 가 있는지 검증

}

}

### 2.1 상황1

* 호출한 URL의 View를 검증한다.
* 시작페이지 구성을 위해 "/" 를 요청하면 home.ftl을 전달하는지 테스트한다.
* code

@Test

public void test\_View검증() throws Exception {

this.mvc.perform(get("/")) // /로 url 호출

.andExpect(status().isOk()) // 위 요청에 따라 결과가 status는 200이며

.andExpect(view().name("home")); // 호출한 view의 이름이 home인지 확인 (확장자는 생략)

}

### 2.2 상황2

* 호출한 URL의 View와 Model 데이터를 검증한다.
* 아직 Service Layer의 상세스펙은 나온 상태가 아니다.
* 시작페이지 구성을 위해 "/" 를 요청하면 home.ftl과 Job, Tect, Essay List를 전달하는지 테스트한다.
* code

// service의 상세스펙이 나오지 않았으니 인터페이스로 service를 먼저 선언한다.

public interface PostService {

List<Job> getJobList();

List<Tech> getTechList();

List<Essay> getEssayList();

}

// 테스트 코드

@MockBean // postService에 가짜 Bean을 등록

private PostService postService;

@Test

public void test\_Model검증및ServiceMocking() throws Exception {

Job[] jobs = {new Job("잡플래닛", LocalDateTime.now(), new ArrayList<>())};

Tech[] techs = {new Tech("OKKY", LocalDateTime.now(), new ArrayList<>())};

given(this.postService.getJobList()) // this.postService.getJobList 메소드를 실행하면

.willReturn(Arrays.asList(jobs)); // Arrays.asList(jobs) 를 리턴해줘라.

given(this.postService.getTechList())

.willReturn(Arrays.asList(techs));

given(this.postService.getEssayList())

.willReturn(new ArrayList<>());

mvc.perform(get("/"))

.andExpect(status().isOk())

.andExpect(view().name("home"))

.andExpect(model().attributeExists("jobs")) // model에 "jobs" 라는 key가 존재하는지 확인

.andExpect(model().attribute("jobs", IsCollectionWithSize.hasSize(1))) // jobs model의 size가 1인지 확인

.andExpect(model().attribute("techs", contains(techs[0]))) // techs model이 "OKKY" 라는 객체를 가지고 있는지 확인

.andExpect(model().attribute("essays", is(empty()))); // 빈 Collection인지 확인

}

### 2.3 상황3

* Job 데이터를 입력받고 "/" 로 redirect 시킨다.
* 아직 Service Layer의 상세스펙은 나온 상태가 아니므로 reqeust, response 결과가 정상적인지만 확인한다.
* Job 스펙에 맞춰 content가 parameter에 포함된다.
* code

// controller 코드

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

public String addJob(Job job) {

postService.addJob(job);

return "redirect:/";

}

// test 코드

@Test

public void test\_Job데이터입력하기() throws Exception {

mvc.perform(post("/job")

.contentType(MediaType.APPLICATION\_FORM\_URLENCODED)

.param("content", "많이 와주세요! http://jojoldu.tistory.com"))

.andExpect(status().is3xxRedirection()) // 302 redirection이 발생했는지 확인

.andExpect(header().string("Location", "/")) // location이 "/" 인지 확인

.andDo(MockMvcResultHandlers.print()); // test 응답 결과에 대한 모든 내용 출력

}

* .param() 외에도 **.header(), cookie(), sessionAttr()** 등의 메소드들이 지원되니 꼭 사용해보길 바란다!

### 2.4 상황4

* @ResponseBody를 통해 Json 데이터를 리턴시킨다.
* Ajax 요청에 대응하기 위해 getJob 메소드는 Job 객체의 Json 형태를 리턴시킨다.
* 정상적으로 저장되었는지 확인할 수 있어야 한다.
* code

// Controller 코드

@RequestMapping(value="/job/{idx}")

@ResponseBody

public Job getJob(@PathVariable long idx) {

return this.postService.getJob(idx);

}

// test 코드

@Test

public void test\_Json결과비교하기() throws Exception {

Job job = new Job("많이 와주세요! http://jojoldu.tistory.com", LocalDateTime.now(), new ArrayList<>());

given(this.postService.getJob(1)) // getJob 메소드에 인자값 1이 입력될 경우

.willReturn(job); // job 객체를 리턴한다. 이럴경우 service 구현체가 없어도 테스트가 가능하다.

mvc.perform(get("/job/1").accept(MediaType.APPLICATION\_JSON))

.andExpect(status().isOk())

.andExpect(jsonPath("$.content").value("많이 와주세요! http://jojoldu.tistory.com")) // json 데이터중 content 속성의 값을 비교한다.

.andDo(MockMvcResultHandlers.print());

}

* 일반적으로 Json 데이터를 테스트 코드상에서 비교하려면 Json 전체 데이터가 필요한데, update date같이 날짜가 있는 경우엔 전체 데이터 비교가 힘들다. 때문에 Json 데이터 중 일부의 데이터만 비교하는것이 좋다.
* $.content 코드는 일반적으로 **jsonPath** 라는 문법으로 불린다. jsonPath를 사용하여 content 값만 비교하였다.
* 자세한 사용법은 [링크](http://goessner.net/articles/JsonPath/)를 참고한다.

### 2.5 상황5

* Job 데이터 조회시 데이터가 없을 경우 NotFound Exception을 발생시킨다.
* NotFoundExcption은 Job/Tech/Essay 만을 나타낼수 있도록, PostNotFoundException 이라는 새로운 Exception으로 처리한다.
* code

// PostNotFound Exception

@ResponseStatus(HttpStatus.NOT\_FOUND) // 404 NOT\_FOUND status

public class PostNotFoundException extends RuntimeException{ // 직접 생성한 Exception

public PostNotFoundException(long idx) {

super("could not find post '" + idx + "'.");

}

}

// Controller의 메소드

@RequestMapping(value="/job/{idx}")

public Job getJob(@PathVariable long idx) {

return Optional.ofNullable(this.postService.getJob(idx))

.orElseThrow(() -> new PostNotFoundException(idx)); // this.postService.getJob(idx)가 null일 경우 PostNotFoundException 발생

}

//테스트 코드

@Test

public void test\_Exception체크() throws Exception {

given(this.postService.getJob(1)) // getJob 메소드에 인자값 1이 입력될 경우

.willReturn(null); // exception 발생을 위해 null 리턴

mvc.perform(get("/job/1")) // /job/1 을 호출할 경우

.andExpect(status().isNotFound()); // not found exception이 나오는지 아닌지 테스트

}

출처: <http://jojoldu.tistory.com/34> [기억보단 기록을]

<http://stackoverflow.com/questions/38168163/setting-up-mockmvc-with-webmvctest-in-spring-boot-1-4-mvc-testing>

|  |
| --- |
| @RunWith(SpringRunner.class)  @WebMvcTest(controllers = CategoryRestService.class)  @ContextConfiguration(classes={MjApplication.class, WebSecurityConfig.class})  public class CategoryRestServiceTest {  @MockBean  private CategoryRepository repository;  @MockBean  CurrentUserDetailsService currentUserDetailsService;  @MockBean  TokenAuthProvider tokenAuthProvider;  @Autowired  MockMvc mockMvc;  private MediaType contentType = new MediaType(MediaType.APPLICATION\_JSON.getType(),  MediaType.APPLICATION\_JSON.getSubtype(), Charset.forName("utf8"));  @Test  public void getCategories() throws Exception {  Category category1 = new Category();  category1.setName("Test Category 1");  category1.setId(1L);  Category category2 = new Category();  category2.setName("Test Category 2");  category2.setId(2L);  List<Category> categoryList = new ArrayList<Category>();  categoryList.add(category1);  categoryList.add(category2);  given(this.repository.findAll())  .willReturn(categoryList);  mockMvc.perform(get("/public/rest/category"))  .andExpect(status().isOk())  .andExpect(content().contentType(contentType))  .andExpect(jsonPath("$[0].id", is(1)))  .andExpect(jsonPath("$[0].name", is("Test Category 1")))  .andExpect(jsonPath("$[1].id", is(2)))  .andExpect(jsonPath("$[1].name", is("Test Category 2")));  }  } |

<https://spring.io/blog/2016/04/15/testing-improvements-in-spring-boot-1-4#testing-the-spring-mvc-slice>

<https://spring.io/guides/gs/spring-boot/>