MongoDB JUnit & Mockito

소프트웨어공학과 / 201332001 강인성 / hogu9401@gmail.com

0. JUnit 정의와 소개

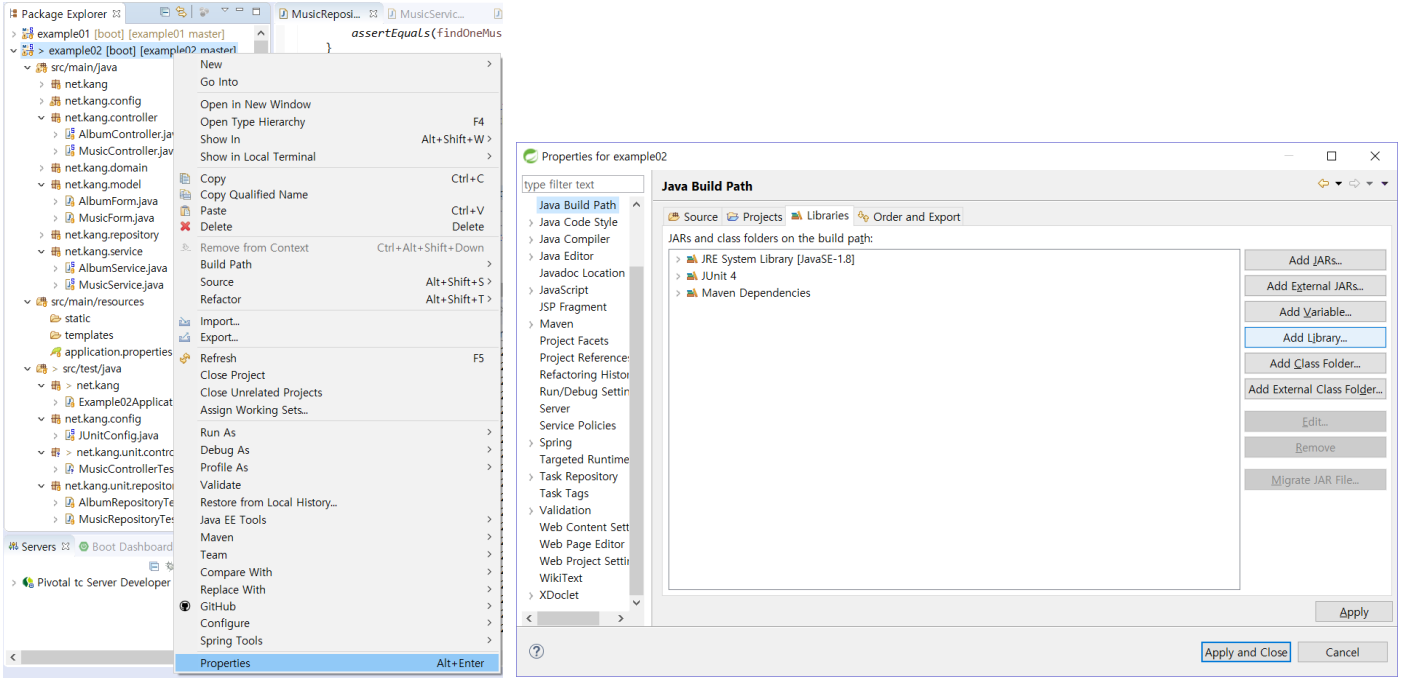
A. JUnit 사용하기

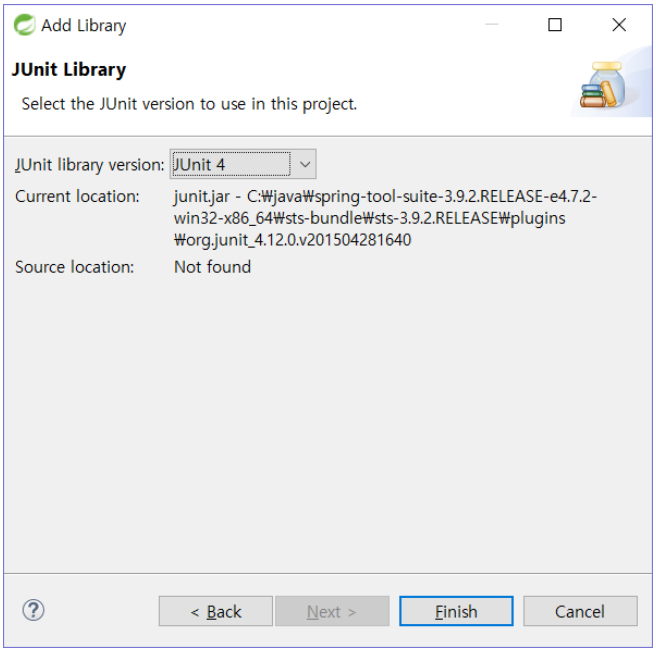
Unit은 말 그대로 단위라는 뜻이다. JUnit은 패키지 단위 별 테스트 도구로 보면 되는데 외부 테스트 프로그램을 작성하여 System.out 함수를 이용하지 않고 오로지 프로그램 테스트를 진행할 때 걸리는 시간도 관리해줘서 오픈소스로 제공을 하여 STS에서도 무난하게 이용할 수 있다. 우리가 원래는 Repository, Service, Controller 클래스들에 대해서 오로지 작성만 해 주고 넘어갔지만, 기본 패키지(main) 뿐만이 아니라 테스트 전용 패키지(test)도 존재하였다는 사실을 알 수 있다. 이를 이용해서 JUnit를 각 단위 별로 나뉘어서 사용하는 방안에 대해 인지를 해보도록 하겠다.

JUnit의 특성으로는 다음과 같다.

- Unit Test Framework(단위 테스트 Framework) 중에 하나이다.
- 문자나 GUI를 기반으로 실행을 한다.
- 단정문(assert로 시작하는 함수 이름들이 대다수.)으로 테스트 케이스의 수행 결과 판별.
- 결과는 성공, 실패 중 하나로 표기를 하고 @Before, @Test, @After 등 말 뜻대로 알 수 있는 간략한 어노테이션을 이용해서 구성하면 된다.

JUnit 4 설정 방법은 JUnit을 사용하고 싶은 프로젝트 이름에 오른쪽 버튼의 Properties에 들어가서 Java Build Path를 클릭해 Library를 새로 추가하면 된다.

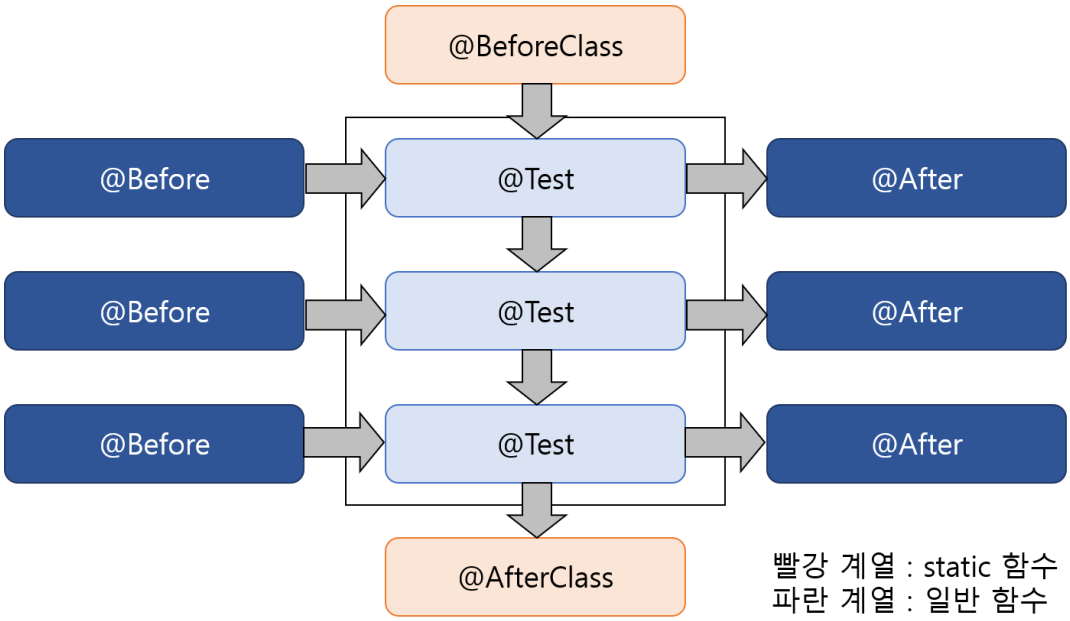




그 다음에 추가할 Library 이름을 JUnit로 맞춘 이후에 버전은 JUnit 4를 선택하여 라이브러리 추가를 적용시키면 JUnit를 이용해서 Unit Test가 가능해진다.

B. JUnit Annotation & Mechanism

우선 JUnit의 각 단위 테스팅은 아래와 같은 메커니즘을 통해서 이뤄진다. JUnit에서 쓰이는 어노테이션은 @BeforeClass, @Before, @Test, @After, @AfterClass 5가지로 나뉘게 된다. 큰 순서로 보면 BeforeClass, Before, Test, After, AfterClass 순으로 실행된다고 볼 수 있다.



@BeforeClass : 이는 Unit 클래스 내부에서 데이터베이스와 연동을 시키기 위해서 드라이버 로딩 부분에서 쓰지만 이는 어차피 Config 클래스를 이용해서 썼으니 넘어가도 상관은 없지만 추후에 클러스터링 작업을 새로 추가할 때에는 상황이 달라질 수도 있으니 일단은 이 어노테이션이 존재한다는 점을 인지하고 넘어가자.

@Before : 이는 @Test 함수가 실행이 되기 이전에 실행되는 함수인데 MongoDB에서는 이미 알고 있는 사실이지만 Transaction 기능을 제공하지 않기 때문에 우리는 여기서 각 데이터를 받아서 Java 내부에 미리 저장을 해 두고 난 후에 @Test 함수를 마치고 난 후에 다시 저장을 해주는 설정만 하겠다.

@Test : 말 그대로 Test 함수이다. assert 함수는 각 Database에서 가져온 데이터들의 비교나 연산 등으로 테스팅의 승낙을 결정해주는 역할을 하기도 한다. 그래서 테스팅 결과의 제대로 된 승낙 여부를 인지하기 위해서는 assert 함수를 작성을 해서 비교를 해 줘야 하는데 대표적으로 assertTrue, assertFalse, assertEquals 함수 등이 존재한다.

@After : @Test 함수 실행 완료 이후에 Transaction 기능과 유사하게 각 데이터를 다시 MongoDB 서버에 저장을 시키게끔 할 때 쓰는 함수로 쓰게 되는데 이따가 새로 배우는 Mock MVC에서는 @Before, @Test 어노테이션을 주로 쓰기 때문에 이러한 어노테이션이 존재한다고 생각하면 된다.

@AfterClass : 모든 @Test 함수를 마치고 난 후에 클래스가 종료되는 시점에서 실행되는 함수인데 이는 예를 들어 데이터베이스 연동을 종결 시킬 때(즉 자원 반납) 쓰이는 사례로 볼 수 있다.

1. JUnit Config 설정

지난 시간에 작성한 REST API에 대해서는 그대로 사용을 해도 무방하다. 이번 예제에서 살펴보는 소스 코드들의 위치는 src > test 에 존재한다는 점을 감안하고 넘어가자. 여기에 JUnit 테스팅 관련 소스 코드들을 올리면서 실행을 할 수가 있다.

```
package net.kang;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@EnableAutoConfiguration(exclude = {JpaRepositoriesAutoConfiguration.class})
@RunWith(SpringRunner.class)
@SpringBootTest
public class Example02ApplicationTests {

	@Test
	public void contextLoads() {
	}

	@Test
	@ConfigurationProperties(prefix = "spring.data.db-main")
```

```
        public void mainDataSource() {
                DataSourceBuilder.create().build();
        }

        @Test
        @ConfigurationProperties(prefix = "spring.data.db-log")
        public void contractDataSource() {
                DataSourceBuilder.create().build();
        }

}
```

[mongoDB_JPA_Start02 > src > test > java > net > kang > Example02ApplicationTests.java]

Example02ApplicationTests.java 파일은 원래 Spring Starter Application을 생성할 시에 자동적으로 생성이 된다. 이전 예제에서는 MongoDB와 연동을 시키는데 있어서 DataSource를 Bean으로 설정하여 관련 함수를 생성하였다. 이는 Example02Application.java에도 이미 기재가 되어 있는데 여기서는 Pivotal 서버(혹은 Tomcat 서버) 자체에서 실행을 하는 것이고, JUnit Test Class는 JUnit 자체에서 실행을 하여 테스팅을 하기 때문에(즉 실행 주체가 달라진다.) 테스팅 패키지 내부에서 MongoDB와의 연동에 이상이 없게 하기 위해 이 문장을 추가하였다.

```
package net.kang.config;

import org.springframework.beans.factory.config.PropertyPlaceholderConfigurer;
import org.springframework.boot.test.context.ConfigFileApplicationContextInitializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.test.context.ContextConfiguration;

@Configuration
@ContextConfiguration(initializers = ConfigFileApplicationContextInitializer.class)
@ComponentScan({"net.kang.domain", "net.kang.repository", "net.kang.service", "net.kang.controller"})
public class JUnitConfig {
        @Bean
        public static PropertyPlaceholderConfigurer propertyPlaceholderConfigurer() {
                PropertyPlaceholderConfigurer propertyPlaceholderConfigurer = new
PropertyPlaceholderConfigurer();
            propertyPlaceholderConfigurer.setLocations(new
ClassPathResource("application.properties"));
            propertyPlaceholderConfigurer.setIgnoreUnresolvablePlaceholders(true);
            return propertyPlaceholderConfigurer;
        }
}
```

[mongoDB_JPA_Start02 > src > test > java > net > kang > config > JUnitConfig.java]

우리가 application.properties에서 MongoDB와 Spring Data MongoDB를 연동시키기 위해서 Database, 사용자 정보, 로컬 정보, 포트 정보 등을 기재를 하였지만, 테스팅 패키지 내부에서 이를 읽어 들이는데 있어서 Placeholder 관련 문제가 자주 발생을 하기 때문에 방지하기 위하여 Configuration을 생성하였으며 또한 ComponentScan은 Main 패키지에 현존하는 Domain, Repository, Service, Controller 등의 위치를 설정을 해 두어 이를 이용해서 데이터를 불러오는데 지장이 없게끔 하기 위해 소스 코드를 추가하였다.

2. Repository 단위 JUnit Testing

Example01에서 이용했던 music_test_01 Database를 사례로 계속 이용을 하지만 여기서도 마찬가지로 music Collection에 대해서만 다루겠다. album Collection에 대해 다룬 내용은 GitHub를 참고하길 바란다.

```
package net.kang.unit.repository;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import java.util.List;
import java.util.Optional;
import java.util.Random;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.data.mongodb.repository.config.EnableMongoRepositories;
import org.springframework.test.annotation.Rollback;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import net.kang.config.JUnitConfig;
import net.kang.config.MongoConfig;
import net.kang.domain.Music;
import net.kang.repository.MusicRepository;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {JUnitConfig.class, MongoConfig.class})
@EnableMongoRepositories(basePackageClasses = net.kang.repository.MusicRepository.class)
@EntityScan(basePackageClasses = net.kang.domain.Music.class)
public class MusicRepositoryTest {
	@Autowired MusicRepository musicRepository;
	static final int QTY=15;
	static final int MIN_YEAR=1900;
	static final int MAX_YEAR=2000;
	static Random random=new Random();
	List<Music> currentMusicList;

	@Before
	public void initialize() {
		currentMusicList=musicRepository.findAll();
		musicRepository.deleteAll();
		for(int k=0;k<QTY;k++) {
			int newYear=MIN_YEAR+random.nextInt(100);
			Music music=new Music();
			music.setTitle(String.format("Title%02d", k));
			music.setSinger(String.format("Singer%02d", k));
			music.setGenre(String.format("Genre%02d", k));
			music.setYear(newYear);
			musicRepository.insert(music);
		}
	}

	@Test
	public void findAllTest() {
		List<Music> musicList=musicRepository.findAll();
		assertEquals(QTY, musicList.size());
	}
```

```
	@Test
	public void findOneTest() {
		int getIndex=random.nextInt(QTY);
		List<Music> musicList=musicRepository.findAll();
		Music findOneMusic=musicList.get(getIndex);
		Optional<Music> resultMusic=musicRepository.findById(findOneMusic.getId());
		assertEquals(findOneMusic, resultMusic.get());
	}

	@Test
	public void findBySingerTest() {
		int getIndex=random.nextInt(QTY);
		String singerName=String.format("Singer%02d", getIndex);
		List<Music> findBySingerMusic=musicRepository.findBySinger(singerName);
		assertTrue(findBySingerMusic.size()>0);
	}

	@Test
	public void findByYearBetweenTest() {
		List<Music> findByYearBetweenMusic=musicRepository.findByYearBetween(MIN_YEAR-1,
MAX_YEAR+1);
		assertEquals(findByYearBetweenMusic.size(), QTY);
	}

	@Test
	@Rollback(true)
	public void insertTest() {
		int newYear=MIN_YEAR+random.nextInt(100);
		Music newMusic=new Music();
		newMusic.setTitle(String.format("Title%02d", QTY));
		newMusic.setSinger(String.format("Singer%02d", QTY));
		newMusic.setGenre(String.format("Genre%02d", QTY));
		newMusic.setYear(newYear);
		musicRepository.insert(newMusic);
		List<Music> findAllMusic=musicRepository.findAll();
		assertEquals(findAllMusic.size(), QTY+1);
	}

	@Test
	@Rollback(true)
	public void updateTest() {
		int getIndex=random.nextInt(QTY);
		int newYear=MIN_YEAR+random.nextInt(100);
		List<Music> findAllMusic=musicRepository.findAll();
		Music updateMusic=findAllMusic.get(getIndex);
		updateMusic.setTitle("tempMusic");
		updateMusic.setSinger("tempSinger");
		updateMusic.setGenre("tempGenre");
		updateMusic.setYear(newYear);
		musicRepository.save(updateMusic);
		Optional<Music> findOneMusic=musicRepository.findById(updateMusic.getId());
		assertEquals(updateMusic, findOneMusic.get());
	}

	@Test
	@Rollback(true)
	public void deleteTest() {
		int getIndex=QTY-1;
		List<Music> findAllMusic=musicRepository.findAll();
		Music deleteMusic=findAllMusic.get(getIndex);
		musicRepository.deleteById(deleteMusic.getId());
		Optional<Music> findOneMusic=musicRepository.findById(deleteMusic.getId());
		assertEquals(findOneMusic.orElse(new Music()), new Music());
```

```
        }

        @Test
        public void countTest() {
                long length=musicRepository.count();
                assertEquals(QTY, length);
        }

        @Test
        public void existsTest() {
                int getIndex=random.nextInt(QTY-1);
                List<Music> findAllMusic=musicRepository.findAll();
                Music existMusic=findAllMusic.get(getIndex);
                assertTrue(musicRepository.existsById(existMusic.getId()));
        }

        @After
        public void afterTest() {
                musicRepository.deleteAll();
                musicRepository.saveAll(currentMusicList);
        }
}
```

[mongoDB_JPA_Start02 > src > test > java > net > kang > unit > repository > MusicRepositoryTest.java]

우선은 MongoDB에서는 Transaction 작업에 대해서는 지원을 하지 않기 때문에 JUnit Before 함수에서는 현재 보유하고 있는 데이터를 currentMusicList에 저장을 하고 난 이후에 Mock 데이터를 임시로 넣어주고 난 후에 각 테스트 별로 작동을 하도록 하였다. 그리고 각 테스트가 완료되는 시점에서는 currentMusicList에 있는 데이터들을 다시 복원을 해서 실제 MongoDB 데이터베이스 내부에서 이상이 없게끔 임시 방편으로 처리를 했는데 JUnit에서는 테스팅을 하는 작업 도중에 실제 MongoDB 서버로 접속이 되어서 Mock 데이터를 임시로 넣게 된다면 music_test_01의 music Collection에 저장이 되기 때문에 한 편으로는 JUnit은 불편하다. 그렇지만 어떠한 원리로 돌아가는지 알기 위해서 Service 객체 까지는 이로 작성을 하였다.

@RunWith(SpringJUnit4ClassRunner.**class**)

JUnit 4 테스팅을 실행하기 위해서는 이 어노테이션을 추가해서 Testing 작업을 할 수 있도록 해야 한다.

@ContextConfiguration(classes = {JUnitConfig.**class**, MongoConfig.**class**})

이는 JUnitConfig, MongoConfig의 각 Configuration 클래스 정보들을 가져와서 MusicRepositoryTest에서 이 Configuration을 설정할 수 있도록 불러오게 하였다.

@EnableMongoRepositories(basePackageClasses = net.kang.repository.MusicRepository.**class**)

@EntityScan(basePackageClasses = net.kang.domain.Music.**class**)

Repository와 Entity 클래스는 main 패키지에 있기 때문에 basePackageClasses에서는 문자열(""로 묶은 데이터)로

패키지 주소를 쓰는 것이 아니라 실제로 위치한 데이터 패키지를 통해 클래스 정보를 가져와서 어노테이션을 이용해 설정을 해 뒀다.

각 함수에 하는 역할들을 나뉘어서 설명한다면 아래와 같이 작성을 할 수 있다.

@Before 함수 : 현재 music Collection에 존재하는 음반들에 대해서 멤버 변수에 저장을 해 두고 난 후에 Mock 데이터를 새로 추가를 하되 각 음악의 멤버 변수를 토대로 작성을 했는데 연도는 1900년에서 2000년대 랜덤 숫자를 넣도록 하였다.

@Test 함수

@Test 함수는 각각 시작되기 이전에 @Before 함수를 시작하고 난 후에 작동을 한다. 각 함수를 마치게 된다면 @After 함수로 넘어가게 된다.

- findAllTest() : Repository에서 제공하는 findAll() 함수를 실험하기 위해 작성하였다. Mock 데이터들은 15개가 들어갔기 때문에 findAll()에서 나온 데이터의 목록이 15개이면 테스팅 완료를 하게끔 만들었다.
- findOneTest() : Mock 데이터에서 0~14 번째에 있는 임의의 데이터 중에서 하나를 가져오고 난 후에 Repository에서 기본적으로 제공하는 findOne() 함수에서 가져온 데이터와 비교하여 같으면 테스팅 완료를 하게끔 만들었다.
- findBySingerTest() : Mock 데이터들 중에서 임의의 가수(Singer00, Singer01, ... Singer14)들 중에서 가수 이름 하나를 뽑아서 Repository에서 새로 추가한 findBySinger 함수를 이용해서 임의의 가수로 검색을 하고 난 후에 데이터 목록 사이즈가 0보다 크면(임의의 가수는 0~14중에 하나를 가리키게끔 생성하여서 대부분 크기는 1이 나온다.) 테스팅 완료를 하게끔 만들었다.
- findByYearBetweenTest() : Mock 데이터의 각 임의의 연도는 1900년에서 2000년 사이로 설정을 하여 Repository에 추가한 findByYearBeween 함수의 작동 여부를 확인하여 모든 데이터가 나오게 된다면 테스팅 완료를 하게끔 만들었다.
- insertTest() : 새로운 Music 데이터를 추가하고 난 후에 findAll를 이용해서 데이터의 목록이 1이 늘었다면 테스팅 완료를 하게끔 만들었다. 또한 exists 함수를 이용하는 방법도 있지만 필자는 정확한 수치를 확인하기 위해서 이를 적용시켰다.
- updateTest() : 한 인덱스를 지목하고 난 후에 데이터의 Field 값들을 각각 바꾸고 난 후에 수정을 하게끔 진행하고 난 후에 findOne을 이용해서 현재 수정 할 때 쓴 객체와 findOne을 통해 새로 가져온 객체를 확인시키고 서로 같다면 테스팅 완료를 하게끔 만들었다.
- deleteTest() : 한 인덱스를 지목하고 데이터를 삭제 완료한 후에 findOne을 이용해서 데이터 존재 여부를 비교하고 난 후에 데이터가 존재하지 않으면 테스팅 완료를 하게끔 만들었다.

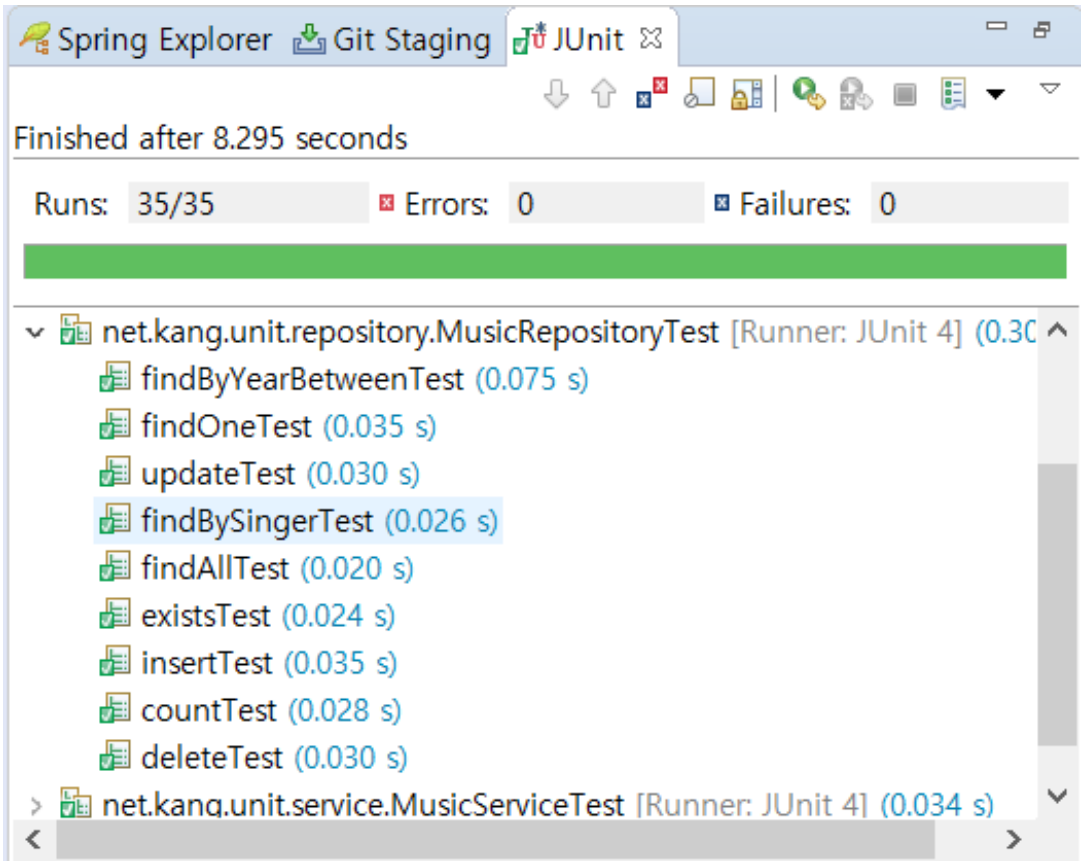
- countTest() : 현재 music Collection에 있는 데이터들의 수를 측정하고 난 후에 비교를 한 이후에 서로 같다면 테스팅 완료를 하게끔 만들었다.
- existsTest() : 이는 Mock 데이터들 중에 추가된 데이터 중 일부를 가져와서 존재하는 것을 확인하는 테스팅이다.

@After 함수

테스팅이 끝나면 Mock 데이터들은 싹 다 삭제하고, 원래 데이터들을 다시 넣어서 REST API에서도 지장이 없게끔 설정을 한 함수이다. 이를 안 쓰고 넘어간다면 REST API에서도 Mock 데이터들이 보이게 되는 상황이 발생한다.

assertEquals, assertTrue : JUnit의 테스팅 완료는 assert가 당락을 결정한다. assertEquals는 두 객체가 객체 주소와는 상관 없이 서로 존재하는 멤버 변수들의 값들이 각각 같다면 테스팅 완료로 나오게 하고, assertTrue는 주어진 조건에 대해서 true가 나오게 된다면 테스팅 완료로 나오게 한다.

이를 기반으로 한 테스팅의 결과는 JUnit 콘솔 창에서 확인을 할 수 있다.



[그림] 위의 함수들을 기반으로 테스팅을 완료한 결과. 초록색은 성공, 검은색은 실패, 빨간색은 에러 발생이다.

3. Service 단위 JUnit Testing

이제 Repository에서 주로 쓰는 함수들에 대한 Testing이 완료 되었으니 이번에는 Service 클래스들에 대한 Testing 작업을 거쳐보도록 하자. Service 클래스에 존재하는 함수들 중에서 음영으로 칠해진 메소드를 새로 추가를 하였다. 그 대신에 파란색으로 음영을 칠하지 않은 함수들에 대해서는 Repository를 이용한 함수이기 때문에 굳이 테스팅을 거치지 않고 넘어가겠다.

```
package net.kang.service;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Service;

import net.kang.domain.Music;
import net.kang.model.MusicForm;
import net.kang.repository.MusicRepository;

@Service
public class MusicService {
	@Autowired MusicRepository musicRepository;
	public List<Music> findAll(){
		return musicRepository.findAll();
	}
	public Optional<Music> findById(String id){
		return musicRepository.findById(id);
	}
	public List<Music> findByYearBetween(int year1, int year2){
		return musicRepository.findByYearBetween(year1, year2);
	}
	public List<Music> findBySinger(String singer){
		return musicRepository.findBySinger(singer);
	}
	public void insert(Music music) {
		musicRepository.insert(music);
	}
	public void saveAll(List<Music> musicList) {
		musicRepository.saveAll(musicList);
	}
	public void update(Music music) {
		musicRepository.save(music);
	}
	public void delete(String id) {
		musicRepository.deleteById(id);
	}
	public void deleteAll() {
		musicRepository.deleteAll();
	}
	public boolean exists(Music music) {
		return musicRepository.existsById(music.getId());
	}
```

```
	public List<Music> insertAfterFindAll(MusicForm musicForm){
		Music newMusic=new Music();
		newMusic.setTitle(musicForm.getTitle());
		newMusic.setSinger(musicForm.getSinger());
		newMusic.setYear(musicForm.getYear());
		newMusic.setGenre(musicForm.getGenre());
		musicRepository.insert(newMusic);
		return musicRepository.findAll();
	}
	public Optional<Music> updateAfterFindOne(String id, MusicForm musicForm){
		Optional<Music> findOneMusic=musicRepository.findById(id);
		Music updateMusic=findOneMusic.orElse(new Music());
		updateMusic.setTitle(musicForm.getTitle());
		updateMusic.setSinger(musicForm.getSinger());
		updateMusic.setYear(musicForm.getYear());
		updateMusic.setGenre(musicForm.getGenre());
		musicRepository.save(updateMusic);
		return musicRepository.findById(id);
	}
	public Music findTopByYearDesc() {
		List<Music> musicList=musicRepository.findAll(new Sort(Sort.Direction.DESC,
"year"));
		return musicList.get(0);
	}
}
```

[mongoDB_JPA_Start02 > src > main > java > net > kang > service > MusicService.java]

그리고 model 패키지를 새로 생성하여 아래와 같은 클래스를 하나 더 생성하도록 한다.

```
package net.kang.model;

import lombok.Data;

@Data
public class MusicForm {
	String title;
	String singer;
	int year;
	String genre;
}
```

[mongoDB_JPA_Start02 > src > main > java > net > kang > model > MusicForm.java]

이제 Service 단위 테스팅은 파란색 음영으로 칠해진 함수들에 대하여 테스팅을 거쳐보도록 하자.

```
package net.kang.unit.service;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotEquals;
import static org.junit.Assert.assertTrue;

import java.util.List;
import java.util.Optional;
import java.util.Random;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.data.mongodb.repository.config.EnableMongoRepositories;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import net.kang.config.JUnitConfig;
import net.kang.config.MongoConfig;
import net.kang.domain.Music;
import net.kang.model.MusicForm;
import net.kang.service.MusicService;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {JUnitConfig.class, MongoConfig.class})
@EnableMongoRepositories(basePackageClasses = net.kang.repository.MusicRepository.class)
@EntityScan(basePackageClasses = net.kang.domain.Music.class)
public class MusicServiceTest {
	@Autowired MusicService musicService;
	private List<Music> currentMusicList;
	static Random random=new Random();

	@Before
	public void initialize() {
		currentMusicList=musicService.findAll();
	}

	@Test
	public void insertAfterFindAllTest() {
		MusicForm musicForm=new MusicForm();
		musicForm.setTitle("insertTitle");
		musicForm.setSinger("insertSinger");
		musicForm.setYear(2000);
		musicForm.setGenre("insertGenre");
		List<Music> insertAfter=musicService.insertAfterFindAll(musicForm);
		assertEquals(currentMusicList.size()+1, insertAfter.size());
	}

	@Test
	public void updateAfterFindOneTest() {
		int getIndex=random.nextInt(currentMusicList.size());
		Music updateMusic=currentMusicList.get(getIndex);
		String updateId=updateMusic.getId();
		MusicForm musicForm=new MusicForm();
		musicForm.setTitle("updateTitle");
		musicForm.setSinger("updateSinger");
		musicForm.setYear(1000);
		musicForm.setGenre("updateGenre");
```

```
                Optional<Music> updateAfterMusic=musicService.updateAfterFindOne(updateId, musicForm);
                assertNotEquals(updateMusic, updateAfterMusic.get());
        }

        @Test
        public void findTopByYearDescTest() {
                int getIndex=random.nextInt(currentMusicList.size());
                Music compareMusic=currentMusicList.get(getIndex);
                Music latestMusic=musicService.findTopByYearDesc();
                assertTrue(compareMusic.getYear()<=latestMusic.getYear());
        }

        @After
        public void afterTest() {
                musicService.deleteAll();
                musicService.saveAll(currentMusicList);
        }
}
```

[mongoDB_JPA_Start02 > src > test > java > net > kang > unit > service > MusicServiceTest.java]

클래스 위에서 쓴 어노테이션에 대해서는 크게 설명을 하지 않고 넘어가겠다. 다만 Service 내부에서도 Repository가 이용되기 때문에 이를 꼭 써줘야 작동이 된다.

@Before 함수 : currentMusicList에는 현재 music_test_01에 현존하는 Document들을 보관을 하고 각 테스팅이 완료되는 시점에서 데이터베이스에 복구를 하게끔 @After 함수에서 저장을 하게 만들었다.

@Test 함수

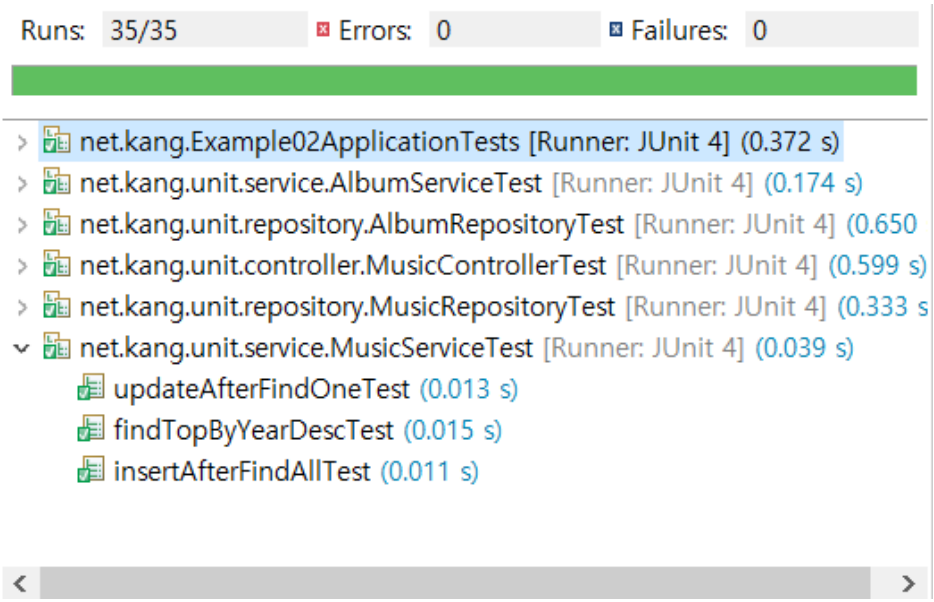
insertAfterFindAllTest : 이는 MusicForm을 이용해서 insert 작업을 하고 난 후에 findAll를 한 데이터의 결과를 비교하는 것이다. MusicService 클래스 내에 있는 함수인 insertAfterFindAll를 테스팅하는 과정인데 데이터를 새로 하나 추가하고 난 이후에는 이전에 findAll를 한 목록보다 1씩 증가하기 때문에 이를 확인하는 결과로 테스팅을 하였다.

updateAfterFindOneTest : MusicForm을 이용해서 update 작업을 한 뒤에 findOne을 한 데이터의 결과를 비교하게끔 만들었다. 역시 updateAfterFindOne을 테스팅하는 과정인데 임의의 데이터를 1개 갱신하고 난 뒤에 현재 데이터와 같지 않다면 테스팅을 완료하게끔 만들었다.

findTopByYearDescTest : 이는 현재 저장된 음악들 중에서 연도가 가장 최신인 음악을 가져오는 함수이다. 물론 이는 Repository에서 생성을 해도 무관하지만 일부러 Service 클래스에 함수를 형성해서 제작을 해봤다. 그러면 임의의 음악 한 개를 뽑아서 어느 음악들 보다 연도가 최신인 음악이 있을 수도 있고 없을 수도 있기 때문에 이를 통해 연도 확인을 통해 최신 년도 이하이면 테스팅 완료를 하게끔 만들었다.

@After 함수 : 각 Service 함수가 테스팅 완료가 된 시점에서 테스팅 이전에 있던 데이터들에 대해서 저장을 하도록 작성한 문장이다. 데이터베이스에서 Rollback 개념으로 인지하면 된다.

이와 같은 함수들을 @Before -> @Test -> @After 함수 순으로 각각 진행을 한다면 테스팅 결과는 아래와 같이 정상 작동을 하는 것을 확인할 수 있다.



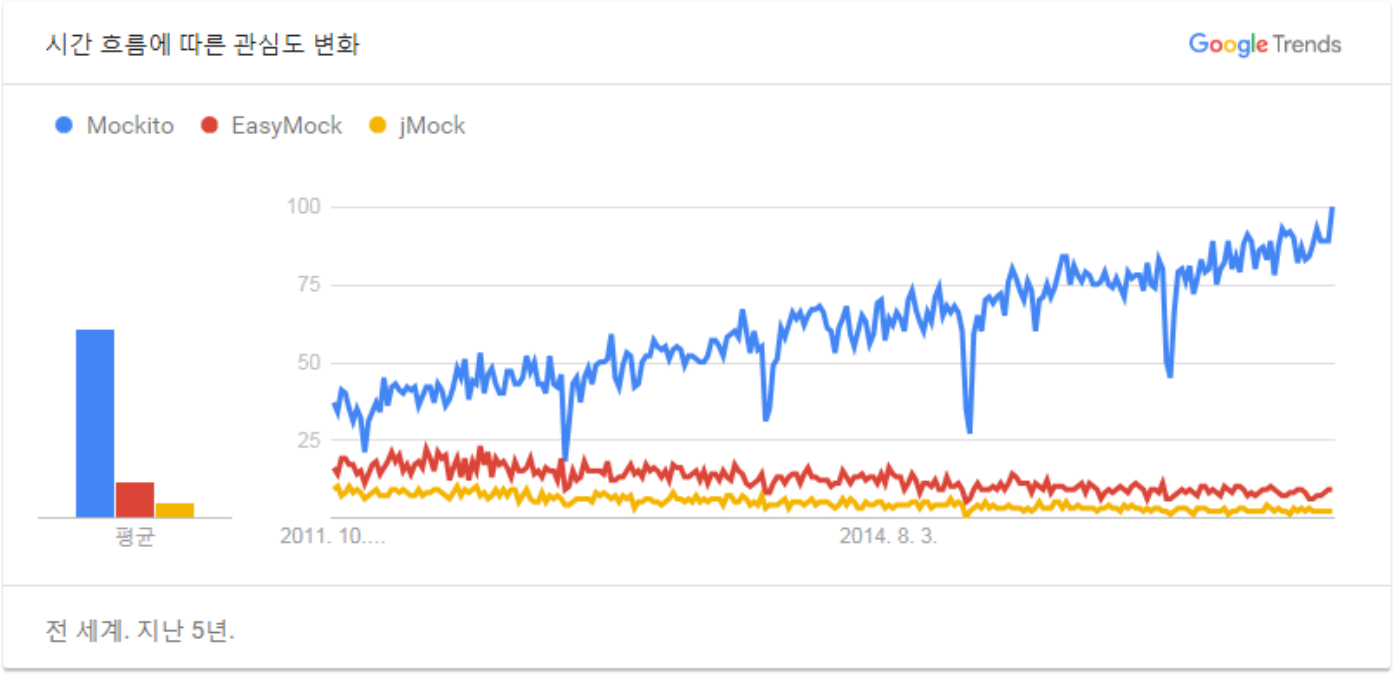
[그림] MusicServiceTest에 대한 테스팅 결과.

4. Mockito Mock MVC 정의와 소개

우리가 방금까지 JUnit를 이용한 테스팅을 연습 해 봤다. 그렇지만 JUnit를 계속 이용해서 테스팅을 하는 것은 실제로 데이터베이스에 적용이 되는데 있어서 Rollback 과정까지 생각하면 무리수로 볼 수 있다. 이번에는 Controller 클래스에서는 Mock MVC를 생성해서 이를 방지해보도록 하겠다. Mock MVC에서 쓰이는 테스팅 과정은 JUnit와 다르기 때문에 이를 어떻게 테스팅을 하는지에 대하여 한 번 탐구해보도록 하겠다.

A. Mock MVC 정의

Mock MVC는 정확히 무슨 역할을 하는 것일까? 우리가 일반적으로 Spring Web MVC를 이용해서 요청 내용들에 대하여 Controller에서 결과를 바로 얻어서 확인하는 경우와는 달리 status 코드를 기반으로 테스트 통과 여부를 결정을 하는데 있어서 도움을 주도록 하기 위해 Mock MVC를 이용을 하는 경우가 있다. 또한 기존 Service나 Repository 테스트의 경우에는 Front-End 작업에서 View에 대하여 연결을 하기 이전에 Back-End 개발 측의 데이터 연동 오류를 정확하게 잡을 수 있도록 도움을 주기 위한 개념을 잡아주는 역할을 한다고 보면 된다. Mock MVC를 이용하는 대표적인 Framework는 Mockito Mock MVC를 주로 활용하는데 Easy Mock도 또한 존재하지만 이는 과거에 많이 이용을 했는데 최근에는 Mockito를 더욱 이용을 하는 추세로 이해할 수 있다.



[그림] Mockito, EasyMock, JMock 등의 이용 추세를 나타낸 도표. 시간이 지날수록 Mockito의 이용 추세가 더욱 올라가고 있다.

B. JUnit VS Mockito

JUnit과 Mockito Mock MVC는 둘 다 공통점은 테스팅 작업을 각 Repository, Service, Controller를 단위 별로 한다는 점이 공통점이다. 그렇지만 이 둘은 어떠한 특징에서 차이가 있는지 알아보기 위해 작성을 해 뒀다.

- Mockito는 JUnit과는 달리 특정 종류의 테스팅을 효율적으로 작성하기 위해 쓰는 Mock MVC 프레임워크의 일종이다. JUnit에서는 각 단위 테스트를 작성하고 실행을 하는데 도움을 주지만 Mockito Mock MVC는 특정 테스팅을 더욱 효율적으로 추측을 하고 난 후에 판결을 내려주는 역할을 한다고 볼 수 있다.
- Unit 테스팅의 핵심 요소는 본래 테스팅을 진행하는 클래스에 대해서 다른 특정한 요소로 분리를 하게 되는 사례를 볼 수 있는데 테스트 중인 클래스에 대한 객체를 제공하는 테스트 더블을 만들어서 수동적으로 판단을 하는 목적을 가지게 될 개념이 바로 Mock MVC가 하는 역할로 볼 수 있다. 즉, JUnit로만 테스팅을 하게 된다면 MongoDB 실제 데이터베이스에 적용하게 되어서(방금 전 Repository, Service 클래스의 사례), 데이터베이스에도 영향을 주는 단점이 있지만, Mock MVC를 이용한다면 트랜잭션 작업에 대해 고려할 필요 없이 특정 클래스의 객체(Repository, Service, Controller)를 Bean 객체를 통해 실제로 데이터베이스에 영향이 없도록 사전에 방지하는 역할을 도와준다.
- 그렇지만 Mock MVC Framework를 반대하는 개발자들의 의견을 들어 본다면 Mock MVC Framework를 사용을 하는 것보단 JUnit을 더욱 활용하자는 의견도 없지 않나 있는데 JUnit은 실제로 존재하는 특정 클래스의 객체(Repository, Service, Controller)들의 테스팅을 직접 하는 점과 Mock MVC에서는 가짜 클래스 객체(Mock Bean)을 이용하여 테스팅을 하는 점을 비교하는 점에서 나온 이슈인데 가짜 클래스 객체를 이용한다면 결과 추측을 하는데 있어서 아직까지 부족함이 있지 않나 싶어서 나온 의견인 듯 하다.

C. Mockito Mock MVC Mechanism

Mockito Mock MVC의 메커니즘을 알아보기 위해서 여기서 주로 쓰는 어노테이션의 소개를 하면 아래와 같다.

@Mock

특정 클래스 객체(Controller, Service, Repostiory etc.)들에 대해서 큰 개념의 클래스에 의존성 주입으로 선언이 되어 있는 경우에는 Mock Bean으로 변환하여 테스팅을 하고 난 이후에 이용을 할 수 있을 때 쓰는 어노테이션이다. 예를 들어 Controller 클래스 내부에는 Service 클래스 객체가 존재하기 마련인데 Controller 클래스에서는 의존성 주입된 Service 클래스가 Mock Bean으로 작동을 하기 위해서는 이를 붙어주면 의존성 주입으로 적용되는 클래스인데도 불구하고 Mock Bean으로 작동을 할 수 있기 때문에 정상적으로 Mocking Test가 가능하게 된다.

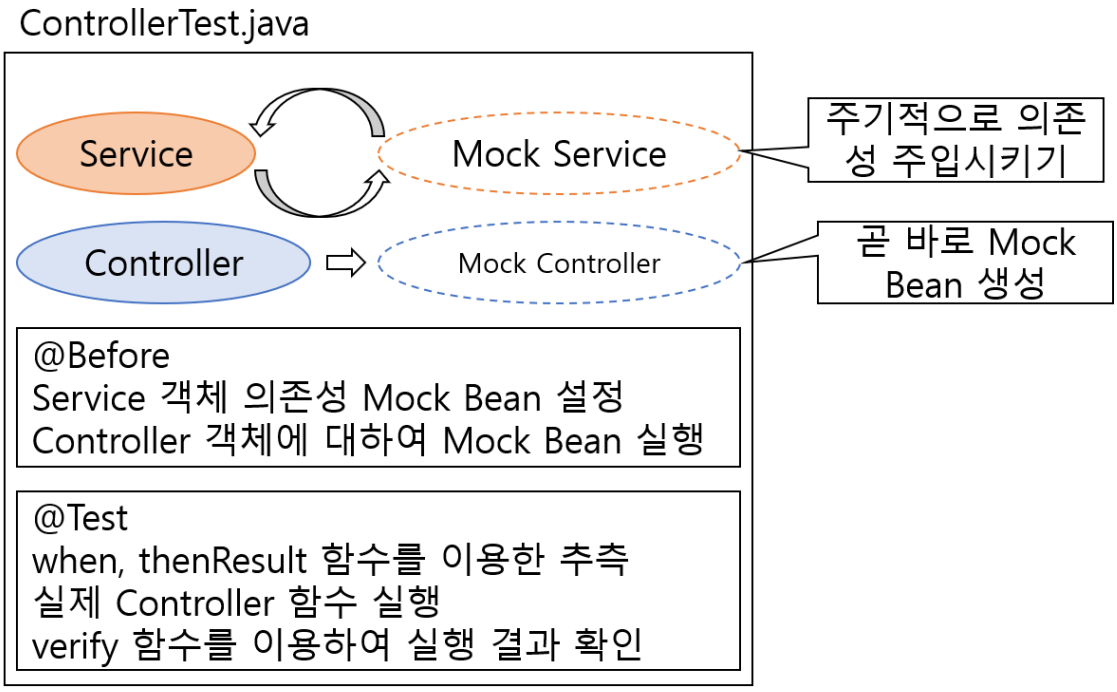
@InjectMock

특정 클래스 객체들에 대해 주기적으로 Mock Bean 객체를 주입시키는 것이 아닌 큰 개념의 클래스 내부에서 Mock Bean 객체로 이용을 할 수 있도록 설정을 하기 위해서는 InjectMock 어노테이션을 붙어주면 된다. 예를 들어 Testing 클래스의 주 객체인 Controller 클래스를 InjectMock으로 설정을 한다면 이 클래스 내에서 Mock Bean 객체로 테스팅을 하는데 지장이 없어지게 된다.

@Spy

이는 Mock Bean 객체와는 달리 실제 특정 클래스 객체 내부에 있는 함수를 쓸 수 있도록 설정을 하는 어노테이션이다. 이를 작성하게 된다면 실제로 데이터베이스에 연동이 될 수 있으니 이를 주의하고 생성을 하자.

이 3가지 어노테이션이 주요 설정인데 우선 Mock Bean 객체를 이용한 테스팅 작업은 대표적으로 어떻게 이뤄지는지에 대해서는 아래 그림을 참고해서 이해하도록 하자.



[그림] Mock MVC의 간략한 메커니즘. 이의 순서는 아래로 내려가면서 실행이 된다.

5. Controller 단위 Mock MVC Testing

이번에는 JUnit만 이용을 하지 않고 Mock MVC을 활용하여 실행 결과를 분석하면서 테스팅을 해 보도록 하겠다. 소스 코드는 아래와 같다.

```
package net.kang.unit.controller;

import static org.hamcrest.CoreMatchers.equalTo;
import static org.mockito.Mockito.doNothing;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.verifyNoMoreInteractions;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.delete;
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.request.MockMvcRequestBuilders.put;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.Random;

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.mockito.junit.MockitoJUnitRunner;
import org.springframework.http.MediaType;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

import net.kang.config.JUnitConfig;
import net.kang.config.MongoConfig;
import net.kang.controller.MusicController;
import net.kang.domain.Music;
import net.kang.service.MusicService;

@RunWith(MockitoJUnitRunner.class)
@ContextConfiguration(classes = {JUnitConfig.class, MongoConfig.class})
@WebAppConfiguration
public class MusicControllerTest {
        static int QTY=10;
        static Random random=new Random();
        MockMvc mockMvc;
        @Mock MusicService musicService;
        @InjectMocks MusicController musicController;

        private Music initMusic(String id, String title, String singer, int year, String genre) {
                Music music=new Music();
                music.setId(id);
                music.setTitle(title);
```

```
                music.setSinger(singer);
                music.setYear(year);
                music.setGenre(genre);
                return music;
        }

        private String jsonStringFromObject(Object object) throws JsonProcessingException {
       ObjectMapper mapper = new ObjectMapper();
       return mapper.writeValueAsString(object);
    }

        private List<Music> makeTempData(){
                List<Music> tmpList=new ArrayList<Music>();
                for(int k=0;k<QTY;k++) {
                        tmpList.add(initMusic(Integer.toString(k+1), String.format("Music%02d", k),
String.format("Singer%02d", k), 1900+random.nextInt(100), String.format("Genre%02d", k)));
                }
                return tmpList;
        }

        private List<Music> findBySingerTempData(){
                List<Music> tmpList=new ArrayList<Music>();
                for(int k=0;k<QTY;k++) {
                        tmpList.add(initMusic(Integer.toString(k+1), String.format("Music%02d", k),
"findSinger", 1900+random.nextInt(100), String.format("Genre%02d", k)));
                }
                return tmpList;
        }

        private List<Music> findByYearTempData(){
                List<Music> tmpList=new ArrayList<Music>();
                for(int k=0;k<QTY;k++) {
                        tmpList.add(initMusic(Integer.toString(k+1), String.format("Music%02d", k),
String.format("Singer%02d", k), 1980+random.nextInt(10), String.format("Genre%02d", k)));
                }
                return tmpList;
        }

        @Before
        public void setup() {
                MockitoAnnotations.initMocks(this);
                mockMvc=MockMvcBuilders.standaloneSetup(musicController).build();
        }

        @Test
        public void findAllTest() throws Exception{
                List<Music> findAllResult=makeTempData();
                when(musicService.findAll()).thenReturn(findAllResult);
                String toJSON=this.jsonStringFromObject(findAllResult);
                mockMvc.perform(get("/music/findAll"))
                .andExpect(status().isOk())
                .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
                .andExpect(content().string(equalTo(toJSON)))
                .andDo(print());

                verify(musicService, times(1)).findAll();
                verifyNoMoreInteractions(musicService);
        }

        @Test
        public void findOneTest() throws Exception{
                Music findOneResult=makeTempData().get(1);

        when(musicService.findById(Integer.toString(1))).thenReturn(Optional.of(findOneResult));
```

```
                String toJSON=this.jsonStringFromObject(findOneResult);
                mockMvc.perform(get("/music/findOne/{id}", 1))
                .andExpect(status().isOk())
                .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
                .andExpect(content().string(equalTo(toJSON)))
                .andDo(print());

                verify(musicService, times(1)).findById("1");
                verifyNoMoreInteractions(musicService);
        }

        @Test
        public void findBySingerTest() throws Exception{
                List<Music> findBySingerResult=findBySingerTempData();
                when(musicService.findBySinger("findSinger")).thenReturn(findBySingerResult);
                String toJSON=this.jsonStringFromObject(findBySingerResult);

                mockMvc.perform(get("/music/findBySinger/{singer}", "findSinger"))
                .andExpect(status().isOk())
                .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
                .andExpect(content().string(equalTo(toJSON)))
                .andDo(print());

                verify(musicService, times(1)).findBySinger("findSinger");
                verifyNoMoreInteractions(musicService);
        }

        @Test
        public void findByYearBetweenTest() throws Exception{
                List<Music> findByYearResult=findByYearTempData();
                when(musicService.findByYearBetween(1980, 1990)).thenReturn(findByYearResult);
                String toJSON=this.jsonStringFromObject(findByYearResult);

                mockMvc.perform(get("/music/findByYearBetween/{year1}/{year2}", 1980, 1990))
                .andExpect(status().isOk())
                .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
                .andExpect(content().string(equalTo(toJSON)))
                .andDo(print());

                verify(musicService, times(1)).findByYearBetween(1980, 1990);
                verifyNoMoreInteractions(musicService);
        }

        @Test
        public void insertTest() throws Exception{
                Music insertMusic=initMusic(Integer.toString(QTY+1), String.format("Title%02d", QTY),
String.format("Singer%02d", QTY), 1900+random.nextInt(100), String.format("Genre%02d", QTY));
                when(musicService.exists(insertMusic)).thenReturn(false);
                doNothing().when(musicService).insert(insertMusic);

                mockMvc.perform(
                                post("/music/insert")
                                .contentType(MediaType.APPLICATION_JSON_UTF8)
                                .content(this.jsonStringFromObject(insertMusic)))
                .andExpect(status().isOk())
                .andDo(print()).andReturn();

                verify(musicService, times(1)).exists(insertMusic);
                verify(musicService, times(1)).insert(insertMusic);
                verifyNoMoreInteractions(musicService);
        }

        @Test
        public void insertTestFail() throws Exception{
```

```
                Music insertMusic=initMusic(Integer.toString(QTY+1), String.format("Title%02d", QTY),
String.format("Singer%02d", QTY), 1900+random.nextInt(100), String.format("Genre%02d", QTY));
                when(musicService.exists(insertMusic)).thenReturn(true);

                mockMvc.perform(
                                post("/music/insert")
                                .contentType(MediaType.APPLICATION_JSON_UTF8)
                                .content(this.jsonStringFromObject(insertMusic)))
                .andExpect(status().isOk())
                .andDo(print()).andReturn();

                verify(musicService, times(1)).exists(insertMusic);
                verifyNoMoreInteractions(musicService);
        }

        @Test
        public void updateTest() throws Exception{
                Music updateMusic=initMusic("1", "tmpTitle", "tmpSinger", 1900+random.nextInt(100),
"tmpGenre");
                when(musicService.exists(updateMusic)).thenReturn(true);
                doNothing().when(musicService).update(updateMusic);

                mockMvc.perform(
                                put("/music/update")
                                .contentType(MediaType.APPLICATION_JSON_UTF8)
                                .content(this.jsonStringFromObject(updateMusic)))
                .andExpect(status().isOk())
                .andDo(print()).andReturn();

                verify(musicService, times(1)).exists(updateMusic);
                verify(musicService, times(1)).update(updateMusic);
                verifyNoMoreInteractions(musicService);
        }

        @Test
        public void updateTestFail() throws Exception{
                Music updateMusic=initMusic("1", "tmpTitle", "tmpSinger", 1900+random.nextInt(100),
"tmpGenre");
                when(musicService.exists(updateMusic)).thenReturn(false);

                mockMvc.perform(
                                put("/music/update")
                                .contentType(MediaType.APPLICATION_JSON_UTF8)
                                .content(this.jsonStringFromObject(updateMusic)))
                .andExpect(status().isOk())
                .andDo(print()).andReturn();

                verify(musicService, times(1)).exists(updateMusic);
                verifyNoMoreInteractions(musicService);
        }

        @Test
        public void deleteTest() throws Exception{
                Music deleteMusic=initMusic("1", "tmpTitle", "tmpSinger", 1900+random.nextInt(100),
"tmpGenre");
                when(musicService.findById(deleteMusic.getId())).thenReturn(Optional.of(deleteMusic));
                doNothing().when(musicService).delete(deleteMusic.getId());

                mockMvc.perform(
                                delete("/music/delete/{id}", deleteMusic.getId()))
                .andExpect(status().isOk())
                .andDo(print()).andReturn();

                verify(musicService, times(1)).findById(deleteMusic.getId());
```

```
                verify(musicService, times(1)).delete(deleteMusic.getId());
                verifyNoMoreInteractions(musicService);
        }

        @Test
        public void deleteTestFail() throws Exception{
                when(musicService.findById("1")).thenReturn(Optional.of(new Music()));

                mockMvc.perform(
                                delete("/music/delete/{id}", "1"))
                .andExpect(status().isOk())
                .andDo(print()).andReturn();

                verify(musicService, times(1)).findById("1");
                verifyNoMoreInteractions(musicService);
        }
}
```

[mongoDB_JPA_Start02 > src > test > java > net > kang > unit > controller > MusicControllerTest.java]

Controller 단위 JUnit Test는 지난 번에 작성한 RequestMapping을 이용을 한다면 아래와 같이 나뉘어지게 된다.

RequestMethod	RequestMapping	Testing Method
music/findAll	GET	findAllTest()
music/findOne/{id}	GET	findOneTest()
music/findBySinger/{singer}	GET	findBySingerTest()
music/findByYearBetween/{year1}/{year2}	GET	findByYearBetweenTest()
music/insert	POST	insertTest()
music/update	PUT	updateTest()
music/delete	DELETE	deleteTest()

그렇지만 여기서 insert, update, delete 작업에 대해서는 데이터베이스의 논리 상 문제가 발생할 수 있기 때문에 사전에 방지하고자 각각 insertTestFail(), updateTestFail(), deleteTestFail() 함수를 더 추가하였다. 이를 통해서 어떻게 테스팅이 진행되는지에 대해서 소스 코드의 순서를 살펴보면서 알아보도록 하겠다.

```
@RunWith(MockitoJUnitRunner.class)
@ContextConfiguration(classes = {JUnitConfig.class, MongoConfig.class})
@WebAppConfiguration
```

SpringJUnit4ClassRunner와는 달리 여기서는 MockitoJUnitRunner로 실행을 해야 Mockito에서 제공하는 테스팅을 진행할 수 있다.

그리고 우리가 REST API에서 이용하는 설정들을 Testing에서도 이용을 할 수 있도록 WebAppConfiguration 어노테이션을 새로 추가하였다.

@Before 함수

```
MockitoAnnotations.initMocks(this);
```
Mock Bean 객체들(MusicService, MusicController)를 쓰기 위해서 이 함수를 작성을 하고 넘어가야 한다.

```
mockMvc=MockMvcBuilders.standaloneSetup(musicController).build();
```
Mock MVC를 실행할 최종적인 큰 단위 객체인 MusicController를 이용해서 실행을 할 필요성이 있는데 각 URL를 이용해서 연동을 할 수 있도록 하게끔 만드는 역할로 보면 된다.

@Test 함수

이는 모든 데이터 목록들에 대한 Controller 테스팅으로 볼 수 있는데 소스 코드 별로 살펴본다면 다음과 같은 메커니즘으로 돌아가게 된다.

```
@Test
public void findAllTest() throws Exception{
        List<Music> findAllResult=makeTempData(); // 1
        when(musicService.findAll()).thenReturn(findAllResult); // 2
        String toJSON=this.jsonStringFromObject(findAllResult); // 3
        mockMvc.perform(get("/music/findAll")) // 4
        .andExpect(status().isOk()) // 5
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON)) // 6
        .andExpect(content().string(equalTo(toJSON))) // 7
        .andDo(print()); // 8

        verify(musicService, times(1)).findAll(); // 9
        verifyNoMoreInteractions(musicService); // 10
}
```
[MusicControllerTest.java의 일부]

```
@RequestMapping("findAll")
public List<Music> findAll(){
        return musicService.findAll();
}
```
[MusicController.java의 일부]

우선 MusicController에 있는 findAll 함수에는 MusicService 함수들 중에 findAll 함수 밖에 이용을 하지 않는다. 그래서 이 데이터 값이 나온다는 것을 추측하기 위해서는 findAll 함수만 이용을 하면 된다. 그렇지만 이전 예제와는 달리 MusicController 함수가 일부 바뀐 곳도 있다.(insert, update, delete etc.) 예를 들어 insert 함수에는 exists 함수와 insert 함수를 쓰기 때문에 insertTest() 함수에서 Mocking Test를 할 때에 insert, exists 함수를 이용을 하였다.

1 : 임의의 음악 10개를 만들어서 이를 findAll를 한 결과가 이와 같다는 증명을 내기 위해 쓰는 데이터들이다.

2 : when 함수는 Mock Bean 객체인 MusicService 객체에서 findAll 함수를 실행한다면 1에서 생성한 데이터 목록들이 나온다는 것을 추측하는 것이다. 이 추측 값에 대해 테스팅 결과는 9번에서 확인을 할 수 있다. 그렇지만 when 함수에서 반환하는 데이터가 확정이 났다면 그냥 when 함수를 이용을 해서 그 데이터 형에 맞춘 추측 값을 입력하면 되지만, insert, update, delete 등 void 함수인 경우에는 doNothing()을 앞에 붙어줘야 정상적으로 테

스팅이 가능하다. 그리고 when() 함수는 Controller 함수에서 썼던 Service 함수들에 대해서 모두 기재를 해야 된다는 사실을 잃지 말고 넘어가도록 한다. 안 그러면 테스팅을 하는데 예외가 존재하여 종료된다.

3 : 이는 7번에서 혹여 상황에 JSON 데이터와 같은 여부를 확인하기 위해 toJSON에 JSON으로 변환한 값을 저장하였다.

4 : music/findAll의 요청 메소드 형은 GET이기 때문에 get으로 작성하였다. insert, update, delete 등은 각각 post(), put(), delete() 함수를 이용해서 작성을 해야 한다. 또한 Mapping URL에 필요로 한 매개 변수는 중괄호를 이용해서 구분해서 작성을 하면 된다. 그리고 그 옆에 매개 변수 이름들을 적어서 이용하면 되겠다.

5 : status()에서 isOK() 함수를 통해 status가 200인지 확인을 해야 한다. 우리가 작성한 Controller에서 나오는 status는 대부분 200이 나온다. 그렇지만 데이터 수정, 삭제, 삽입 등에 따라서 값이 언제든지 달라질 수 있다. 이를 RequestObject를 이용해서 설정을 해야 하는데 이에 대해서 추후에 반영을 하겠다.

6 : Response의 데이터 형을 JSON으로 바꾸기 위해서 이를 설정하였는데 UTF-8 환경이 아닌 경우에는 ***APPLICATION_JSON_UTF8, APPLICATION_JSON_UTF8_VALUE*** 등의 설정을 해 줘야 한다.

7 : Response를 통한 데이터의 값들을 방금 추측한 데이터와 같은지에 대해서 확인을 하기 위해 equalTo 함수를 작성하였다. 실제로 equalTo 함수는 hamcrest에서 제공하는 함수인데 이는 JSON에서 포함된 개행 문자, 탭 문자 등을 고려해서 비교를 해주는 역할을 한다.

8 : print() 함수는 우리가 여태동안 실행해온 흔적들을 Console 창에서 볼 수 있도록 하는 역할을 한다. 실행 결과는 아래와 같다.

```
MockHttpServletRequest:
        HTTP Method = GET
        Request URI = /music/findAll
         Parameters = {}
            Headers = {}
               Body = <no character encoding set>
    Session Attrs = {}

Handler:
               Type = net.kang.controller.MusicController
             Method = public java.util.List<net.kang.domain.Music> net.kang.controller.MusicController.findAll()

Async:
    Async started = false
     Async result = null

Resolved Exception:
               Type = null

ModelAndView:
          View name = null
               View = null
              Model = null

FlashMap:
        Attributes = null
```

```
MockHttpServletResponse:
           Status = 200
    Error message = null
          Headers = {Content-Type=[application/json;charset=UTF-8]}
     Content type = application/json;charset=UTF-8
             Body = [{"id":"1","title":"Music00","singer":"Singer00","year":1907,"genre":"Genre00"},{"id":"2","title":"Music01","singer":"Singer01","year":1984,"genre":"Genre01"},{"id":"3","title":"Music02","singer":"Singer02","year":1991,"genre":"Genre02"},{"id":"4","title":"Music03","singer":"Singer03","year":1924,"genre":"Genre03"},{"id":"5","title":"Music04","singer":"Singer04","year":1902,"genre":"Genre04"},{"id":"6","title":"Music05","singer":"Singer05","year":1932,"genre":"Genre05"},{"id":"7","title":"Music06","singer":"Singer06","year":1925,"genre":"Genre06"},{"id":"8","title":"Music07","singer":"Singer07","year":1927,"genre":"Genre07"},{"id":"9","title":"Music08","singer":"Singer08","year":1991,"genre":"Genre08"},{"id":"10","title":"Music09","singer":"Singer09","year":1912,"genre":"Genre09"}]
    Forwarded URL = null
   Redirected URL = null
          Cookies = []
```

9 : Controller를 통한 결과가 올바르게 나왔는지 확인을 할 필요가 있다. 이를 실행하기 위해서는 musicService 객체를 이용해서 findAll를 하면서 위에서 Response한 값들에 대해서 정확하게 일치하는지에 대해 확인을 시켜주는 함수가 바로 verify 함수이다. verify 함수에서 이용할 수 있는 Service 함수들은 when에서 언급했던 함수들로 이용을 해서 정상 작동을 하는 곳에 확인을 할 때 쓰는 것으로 이해할 수 있다.

10 : 테스팅이 끝났으면 MusicService 객체와 MusicController 객체의 비교를 종료 시켜주는 문장으로 보면 된다.

여기에 나온 순서들은 findOne, insert, update, delete 등의 Testing에도 모두 해당되는 이야기인데 Controller 객체의 Request Method를 맞춰서 작성을 해야 하는 것이 관건이다. 이를 충족해서 어떻게 돌아가는지에 대해서는 여기서 작성하면 길어지면서 메커니즘은 비슷하기 때문에 생략하겠다.

Controller Testing에서는 Service 객체와 Controller 객체가 서로 상호 작용을 하면서 MongoDB에 존재하는 데이터들을 직접 Mock MVC 내부에서 넣어보면서 같이 테스팅을 할 수 있는 기능을 제공을 한다. 그래서 추후에 Mock MVC를 이용해서 테스팅이 완료되었다면 REST API를 이용한 JSON 결과를 한 눈에 볼 수 있도록 하기 위해 Postman을 이용하지 않고 Swagger API를 이용해서 REST API를 최종 점검하는데 이용을 할 예정이다. Swagger API를 이용하는 방법에 대해서는 추후에 후술하도록 하겠다.

<참고> JUnit4를 이용하기 위해서 방금 전에 설명한 설정대로 안 된다면 Maven Dependencies 저장소를 이용해서 직접 불러올 수도 있다. pom.xml 파일에 아래와 같은 문장들을 첨가해서 작동하는데 이상이 없도록 설정을 해 두도록 하자.

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.10.19</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

[소스 코드 GitHub 확인]

https://github.com/tails5555/mongoDB_JPA_Start02

이번 스터디 노트에서 작성한 MongoDB JPA Testing 예제 결과물.

[출처]

http://www.nextree.co.kr/p11104/ - JUnit 기초 개념

http://using.tistory.com/54 - JUnit 기초 개념과 활용 사례

http://blog.saltfactory.net/create-and-test-rest-conroller-in-spring/ - JUnit를 이용한 구현 사례(1)

https://www.blazemeter.com/blog/spring-boot-rest-api-unit-testing-with-junit - JUnit를 이용한 구현 사례(2)

http://blog.leocat.kr/notes/2016/01/22/mongodb-cannot-find-mongorepository-bean - MongoRepository Bean 찾을 수 없을 때 해결하는 방법

https://stackoverflow.com/questions/29669393/override-default-spring-boot-application-properties-settings-in-junit-test - JUnit Test에서 application.properties 관련 문제 해결 방안

http://thswave.github.io/java/2015/03/02/spring-mvc-test.html - Controller JUnit 테스팅 방법

http://jdm.kr/blog/222 - Mockito 소개와 메커니즘 안내

http://hyeonjae-blog.logdown.com/posts/679308 - Mockito Mock MVC Framework 개념

https://modernjava.io/testing-spring-mvc-mockmvc/ - Mockito Mock MVC 메소드 소개

https://memorynotfound.com/unit-test-spring-mvc-rest-service-junit-mockito/ - Mockito Mock MVC를 이용한 구현

사례

https://code.google.com/archive/p/mockito/wikis/MockitoFeaturesInKorean.wiki - Mockito 한국어 위키. 여기는 Mockito Mock MVC Framework를 주로 이용하는 한국 유저들의 번역본이다.