**Unit and Integration Tests in Spring Boot**

## 1. Overview

In this post, we'll have a look at how to write tests unit and integration in a Spring Boot environment. You can find tons of tutorials online on this topic but it is very difficult to find all the information that you need in just one page. I often noticed that junior developers are confusing between unit and integration test expecially when speaking about spring ecosystem and I'll try to clarify the usage of different annotations used in different contexts.

## 2. Unit vs. Integration tests

Wikipedia says about [unit testing](https://en.wikipedia.org/wiki/Unit_testing): "In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use."

and about [Integration testing](https://en.wikipedia.org/wiki/Integration_testing): "Integration testing (sometimes called integration and testing, abbreviated I&T) is the phase in software testing in which individual software modules are combined and tested as a group."

In simple words, when we do unit test, we test just a single unit of code, one method at a time, excluding all other components that interact with our under testing one.

In integration tests on the other side, we test the integration between components. **Thanks to unit testing, we know that components behave as required individually, but we don't know how they'll work altogether.** This is the responsibility of integration tests.

## 3. Java Test Unit

All Java developers know about **JUnit**as the main framework to perform test unit. It offers a lot of annotations to make assertions on expectations.

**Hamcrest**is an additional framework for software tests. Hamcrest allows checking for conditions in your code using existing matchers classes and it also allows you to define your custom matcher implementations. To use Hamcrest matchers in JUnit you have to use the assertThat statement followed by one or several matchers.

Here you can see simple tests using both frameworks:

import static org.hamcrest.CoreMatchers.allOf;

import static org.hamcrest.CoreMatchers.anyOf;

import static org.hamcrest.CoreMatchers.both;

import static org.hamcrest.CoreMatchers.containsString;

import static org.hamcrest.CoreMatchers.equalTo;

import static org.hamcrest.CoreMatchers.everyItem;

import static org.hamcrest.CoreMatchers.hasItems;

import static org.hamcrest.CoreMatchers.not;

import static org.hamcrest.CoreMatchers.sameInstance;

import static org.hamcrest.CoreMatchers.startsWith;

import static org.junit.Assert.assertArrayEquals;

import static org.junit.Assert.assertEquals;

import static org.junit.Assert.assertFalse;

import static org.junit.Assert.assertNotNull;

import static org.junit.Assert.assertNotSame;

import static org.junit.Assert.assertNull;

import static org.junit.Assert.assertSame;

import static org.junit.Assert.assertThat;

import static org.junit.Assert.assertTrue;

import java.util.Arrays;

import org.hamcrest.core.CombinableMatcher;

import org.junit.Test;

public class AssertTests {

@Test

public void testAssertArrayEquals() {

byte[] expected = "trial".getBytes();

byte[] actual = "trial".getBytes();

assertArrayEquals("failure - byte arrays not same", expected, actual);

}

@Test

public void testAssertEquals() {

assertEquals("failure - strings are not equal", "text", "text");

}

@Test

public void testAssertFalse() {

assertFalse("failure - should be false", false);

}

@Test

public void testAssertNotNull() {

assertNotNull("should not be null", new Object());

}

@Test

public void testAssertNotSame() {

assertNotSame("should not be same Object", new Object(), new Object());

}

@Test

public void testAssertNull() {

assertNull("should be null", null);

}

@Test

public void testAssertSame() {

Integer aNumber = Integer.valueOf(768);

assertSame("should be same", aNumber, aNumber);

}

// JUnit Matchers assertThat

@Test

public void testAssertThatBothContainsString() {

assertThat("albumen", both(containsString("a")).and(containsString("b")));

}

@Test

public void testAssertThatHasItems() {

assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));

}

@Test

public void testAssertThatEveryItemContainsString() {

assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }), everyItem(containsString("n")));

}

// Core Hamcrest Matchers with assertThat

@Test

public void testAssertThatHamcrestCoreMatchers() {

assertThat("good", allOf(equalTo("good"), startsWith("good")));

assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));

assertThat("good", anyOf(equalTo("bad"), equalTo("good")));

assertThat(7, not(CombinableMatcher.&amp;lt;Integer&amp;gt; either(equalTo(3)).or(equalTo(4))));

assertThat(new Object(), not(sameInstance(new Object())));

}

@Test

public void testAssertTrue() {

assertTrue("failure - should be true", true);

}

}

## 4. Introducing Our Example

Let’s write our simple application. The idea is to provide a basic search engine for manga.

### 4.1. Maven Dependencies

First of all, we need to add some dependency to our project



### 4.2. Define the Model

Our model is really simple; it is made up of only two classes: Manga and MangaResult.

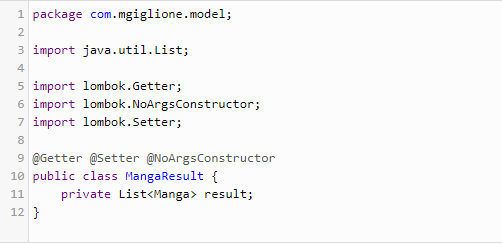
#### 4.2.1. Manga Class

Manga class represents an instance of manga as retrieved by the system. I used [Lombok](https://projectlombok.org/)to reduce boilerplate code.



#### 4.2.2. MangaResult

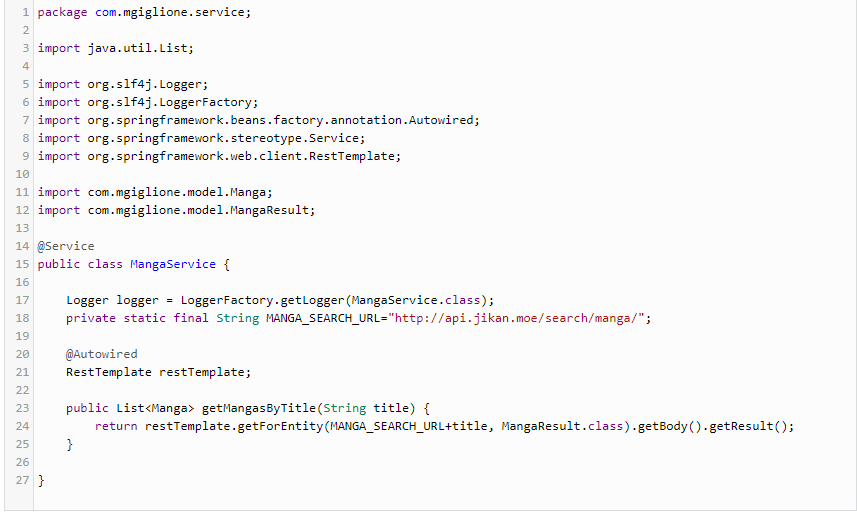
MangaResult is a wrapper class that contains a list of mangas.



### 4.3. Implementing the Service

For implementing the service, we will use API freely exposed by [Jikan Moe](https://jikan.moe/" \t "_blank).

RestTemplate is the Spring class that I use to make REST calls to the API.



### 4.4. Implementing the Controller

The next step on the list is to write down the REST controller that exposes two endpoints, one synchronous and one asynchronous, just for testing purposes. This controller makes use of the Service defined above.

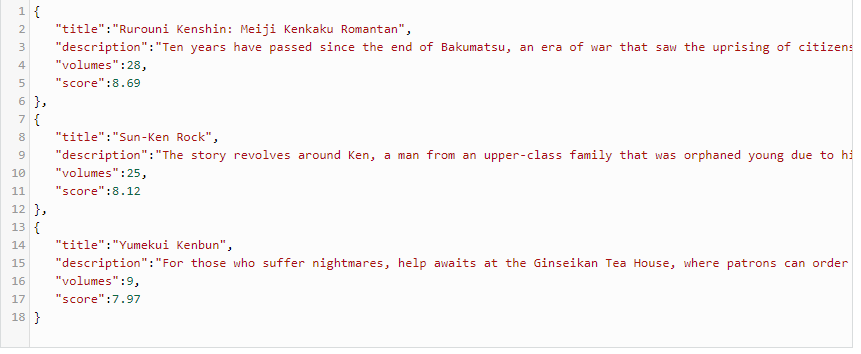


### 4.5. Launching and Testing the System

mvn spring-boot:run

Then let’s try it:  
  
curl http://localhost:8080/manga/async/ken  
curl http://localhost:8080/manga/sync/ken

Example of output:



## 5. Unit Testing the Spring Boot Application

Spring boot offers a great class to make testing easier: [@SpringBootTest](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/context/SpringBootTest.html) annotation

This annotation can be specified on a test class that runs Spring Boot based tests.  
Provides the following features over and above the regular Spring TestContext Framework:

* Uses SpringBootContextLoader as the default ContextLoader when no specific @ContextConfiguration (loader=…) is defined.
* Automatically searches for a @SpringBootConfiguration when nested @Configuration is not used, and no explicit classes are specified.
* Allows custom Environment properties to be defined using the properties attribute.
* Provides support for different web environment modes, including the ability to start a fully running web server listening on a defined or random port.
* Registers a TestRestTemplate and/or WebTestClient bean for use in web tests that are using a fully running web server.

We basically have two components to test here: MangaService and MangaController

### 5.1. Unit Testing MangaService

To test MangaService, we need to isolate it from external components. In our case, we only have one external component required: RestTemplate, which we use to call a remote API.

What we need to do is to mock the RestTemplate bean and let it always respond with a fixed given response. Spring Test incorporates and extends the Mockito library to configure mocked beans through the @MockBean annotation.

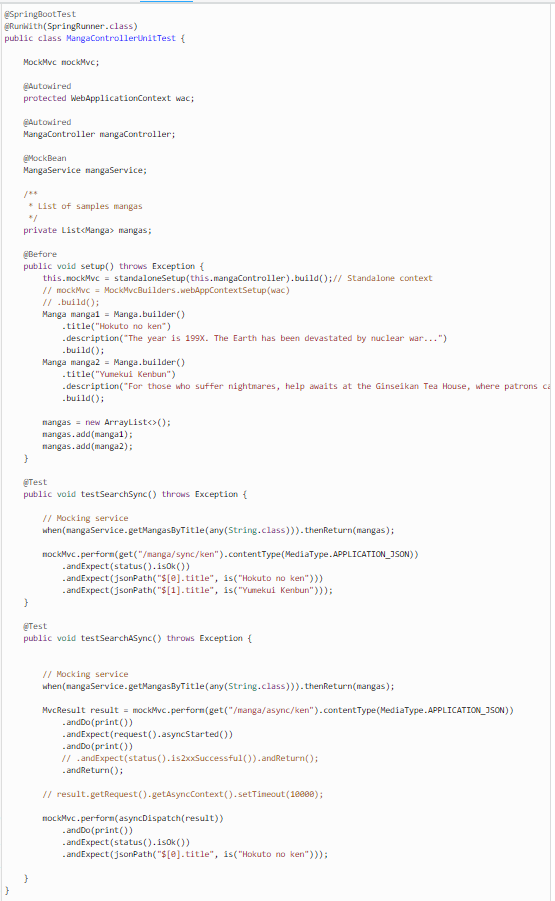


## 5.2. Unit Testing MangaController

As done in the unit testing of the service, we need to isolate components. In this case, we need to mock the MangaService bean.

Then, we have a little further problem… Controller part is the part of the system that manages HttpRequest, so we need a system to simulate this behavior without starting a full HTTP server.  
MockMvc is the Spring class that does that. It can be set up in different ways:

1. Using Standalone Context
2. Using WebApplication Context
3. Let Spring autoconfigure it by loading all context by using these annotations on test class @SpringBootTest @AutoConfigureMockMvc
4. Let Spring autoconfigure it by loading just the web layer context by using these annotations on the test class @WebMvcTest



As you can see from the code, I chose the first solution because it is the lightest one, and we have the best governance on what we load in the Spring context.

In the async test, I had to simulate the asynchronous behavior by first calling the service and then starting the asyncDispatch method.

## 6. Integration Testing the Spring Boot Application

For the integration tests, we want to check our main components with downstream communication.

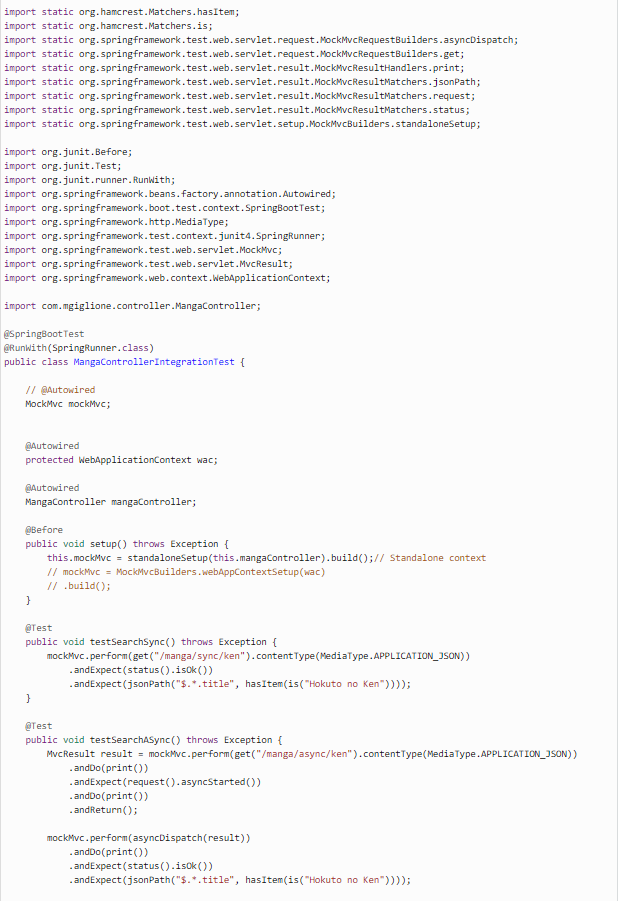
### 6.1. Integration Testing of MangaService

This test is really simple. We don’t need to mock anything because we want to call the remote mangas API.



### 6.2. Integration Testing of MangaController

This test is pretty similar to the unit test, but in this case, we haven’t a mocked service.



## 6.3 Example of Integration Testing



## 

## 7. Conclusions

We have seen the main differences between unit and integration tests in a Spring Boot environment, taking a look at frameworks like Hamcrest that simplify test writing, as well. Of course, you can find everything in my [GitHub repository](https://github.com/jdesigndev/spring-test-unit).