[Testing in Spring boot 2](#_Toc69746477)

[1) General steps for testing 2](#_Toc69746478)

[2) Integration test vs Unit test 3](#_Toc69746479)

[3) What are the testing feature of spring for different layers? 4](#_Toc69746480)

[4) @SpringBootTest 5](#_Toc69746481)

[5) Junit 4 and 5 6](#_Toc69746482)

[6) How to test if spring has created controller or other components? 6](#_Toc69746483)

[7) @MockBean usage 7](#_Toc69746484)

[8) @RunWith(SpringRunner.class) 8](#_Toc69746485)

[Testing web layer or Rest Controllers 8](#_Toc69746486)

[9) Using TestRestTemplate 9](#_Toc69746487)

[10) Spring MockMVC 10](#_Toc69746488)

[11) @WebMvcTest 11](#_Toc69746489)

[Data JPA layer test 15](#_Toc69746490)

[12) How to test data layer using @DataJpaTest? 15](#_Toc69746491)

[13) More example: 16](#_Toc69746492)

[14) testEntityManager: 17](#_Toc69746493)

[JDBC TEST 18](#_Toc69746494)

[15) example of jdbc test 18](#_Toc69746495)

[RestClientTest 18](#_Toc69746496)

[16) example of restClientTest 18](#_Toc69746497)

[SpringBootTest Customization 21](#_Toc69746498)

[17) How to customize SpringBootTest 21](#_Toc69746499)

[18) @TestConfiguration 22](#_Toc69746500)

[Reference: 23](#_Toc69746501)

# Testing in Spring boot

## General steps for testing

### The Spring Framework includes a dedicated test module for such integration testing. You can declare a dependency directly to org.springframework:spring-test or use the spring-boot-starter-test “Starter” to pull it in transitively.

### <dependency>

### <groupId>org.springframework.boot</groupId>

### <artifactId>spring-boot-starter-test</artifactId>

### <scope>test</scope>

### </dependency>

### The spring-boot-starter-test “Starter” (in the test scope) contains the following provided libraries:

### JUnit 4: The de-facto standard for unit testing Java applications.

### Spring Test & Spring Boot Test: Utilities and integration test support for Spring Boot applications.

### AssertJ: A fluent assertion library.

### Hamcrest: A library of matcher objects (also known as constraints or predicates).

### Mockito: A Java mocking framework.

### JSONassert: An assertion library for JSON.

### JsonPath: XPath for JSON.

### Unit Tests should be written under the src/test/java directory and classpath resources for writing a test should be placed under the src/test/resources directory.

### One of the major advantages of dependency injection is that it should make your code easier to unit test. You can instantiate objects by using the new operator without even involving Spring. You can also use mock objects instead of real dependencies.

### Write test cases for SpringBootApplication class, data layer, web layer, unit test, integration test and other test depends on the framework you are using in the projects.

## Integration test vs Unit test

### Before we start into integration tests with Spring Boot, let’s define what sets an integration test apart from a unit test.

### A unit test covers a single “unit”, where a unit commonly is a single class, but can also be a cluster of cohesive classes that is tested in combination.

### An integration test can be any of the following:

### a test that covers multiple “units”. It tests the interaction between two or more clusters of cohesive classes.

### a test that covers multiple layers. This is actually a specialization of the first case and might cover the interaction between a business service and the persistence layer, for instance.

### a test that covers the whole path through the application. In these tests, we send a request to the application and check that it responds correctly and has changed the database state according to our expectations.

### Spring Boot provides the @SpringBootTest annotation which we can use to create an application context containing all the objects we need for all of the above test types. Note, however, that overusing @SpringBootTest might lead to very long-running test suites.

### So, for simple tests that cover multiple units we should rather create plain tests, very similar to unit tests, in which we manually create the object graph needed for the test and mock away the rest. This way, Spring doesn’t fire up a whole application context each time the test is started.

### For tests that cover integration with the web layer or persistence layer, we can use @WebMvcTest or @DataJpaTest instead. For integration with other layers, have a look at Spring Boot’s other test slice annotations. Note that these test slices will also take some time to boot up, though.

### Finally, for tests that cover the whole Spring Boot application from incoming request to database, or tests that cover certain parts of the application that are hard to set up manually, we can and should use @SpringBootTest.

## What are the testing feature of spring for different layers?

### **@JdbcTest** – can be used for a typical [jdbc](https://howtodoinjava.com/jdbc-tutorials/" \t "_blank) test when a test focuses only on jdbc-based components. It disables full auto-configuration and instead apply only configuration relevant to jdbc tests.

### By default, tests annotated with @JdbcTest are transactional and roll back at the end of each test. The annotation configures an in-memory embedded database and **JdbcTemplate**.

### **@JooqTest** – It can be used when a test focuses only on jOOQ-based components. Beware that by default, tests annotated with @JooqTest use the application configured database. To use embedded in-memory database, **@AutoConfigureTestDatabase** annotation can be used to override these settings.

### **@JsonTest** – It is used when a test focuses only on JSON serialization. It initializes the @JsonComponent, JacksonTester, JsonbTester and GsonTester fields.

### [**@DataJpaTest**](https://howtodoinjava.com/spring-boot2/testing/datajpatest-annotation/) – It can be used to test JPA applications. By default, it scans for @Entity classes and configures Spring Data JPA repositories. If an embedded database is available on the classpath, it configures one as well.

### By default, data JPA tests are transactional and roll back at the end of each test.

### Data JPA tests may also inject a **TestEntityManager** bean, which provides an alternative to the standard JPA EntityManager that is specifically designed for tests.

### **@DataMongoTest** – is used to test MongoDB applications. By default, it configures an in-memory embedded MongoDB (if available), configures a **MongoTemplate**, scans for @Document classes, and configures Spring Data MongoDB repositories.

### **@DataRedisTest** – is used to test Redis applications. By default, it scans for @RedisHash classes and configures Spring Data Redis repositories.

### **@DataLdapTest** – is used to test LDAP applications. By default, it configures an in-memory embedded LDAP (if available), configures an **LdapTemplate**, scans for @Entry classes, and configures Spring Data LDAP repositories.

### [**@RestClientTest**](https://howtodoinjava.com/spring-boot2/testing/restclienttest-test-services/) – is used to test REST clients. By default, it auto-configures Jackson, GSON, and Jsonb support, configures a RestTemplateBuilder, and adds support for MockRestServiceServer.

## @SpringBootTest

### @SpringBootTest

### public class TestingWebApplicationTests {

### @Test

### public void contextLoads() {

### }

### }

### Often, you need to move beyond unit testing and start integration testing (with a Spring ApplicationContext). It is useful to be able to perform integration testing without requiring deployment of your application or needing to connect to other infrastructure.

### The first thing you can do is write a simple sanity check test that will fail if the application context cannot start. The @SpringBootTest annotation tells Spring Boot to look for a main configuration class (one with @SpringBootApplication, for instance) and use that to start a Spring application context.

### By default, @SpringBootTest will not start a server. You can use the webEnvironment attribute of @SpringBootTest to further refine how your tests run:

### **MOCK(Default) :** Loads a web ApplicationContext and provides a mock web environment. Embedded servers are not started when using this annotation. If a web environment is not available on your classpath, this mode transparently falls back to creating a regular non-web ApplicationContext. It can be used in conjunction with @AutoConfigureMockMvc or @AutoConfigureWebTestClient for mock-based testing of your web application.

### **RANDOM\_PORT:** Loads a WebServerApplicationContext and provides a real web environment. Embedded servers are started and listen on a random port.

### **DEFINED\_PORT:** Loads a WebServerApplicationContext and provides a real web environment. Embedded servers are started and listen on a defined port (from your application.properties) or on the default port of 8080.

### **NONE**: Loads an ApplicationContext by using SpringApplication but does not provide any web environment (mock or otherwise).

### \*\*\*If your test is @Transactional, it rolls back the transaction at the end of each test method by default. However, as using this arrangement with either RANDOM\_PORT or DEFINED\_PORT implicitly provides a real servlet environment, the HTTP client and server run in separate threads and, thus, in separate transactions. Any transaction initiated on the server does not roll back in this case.

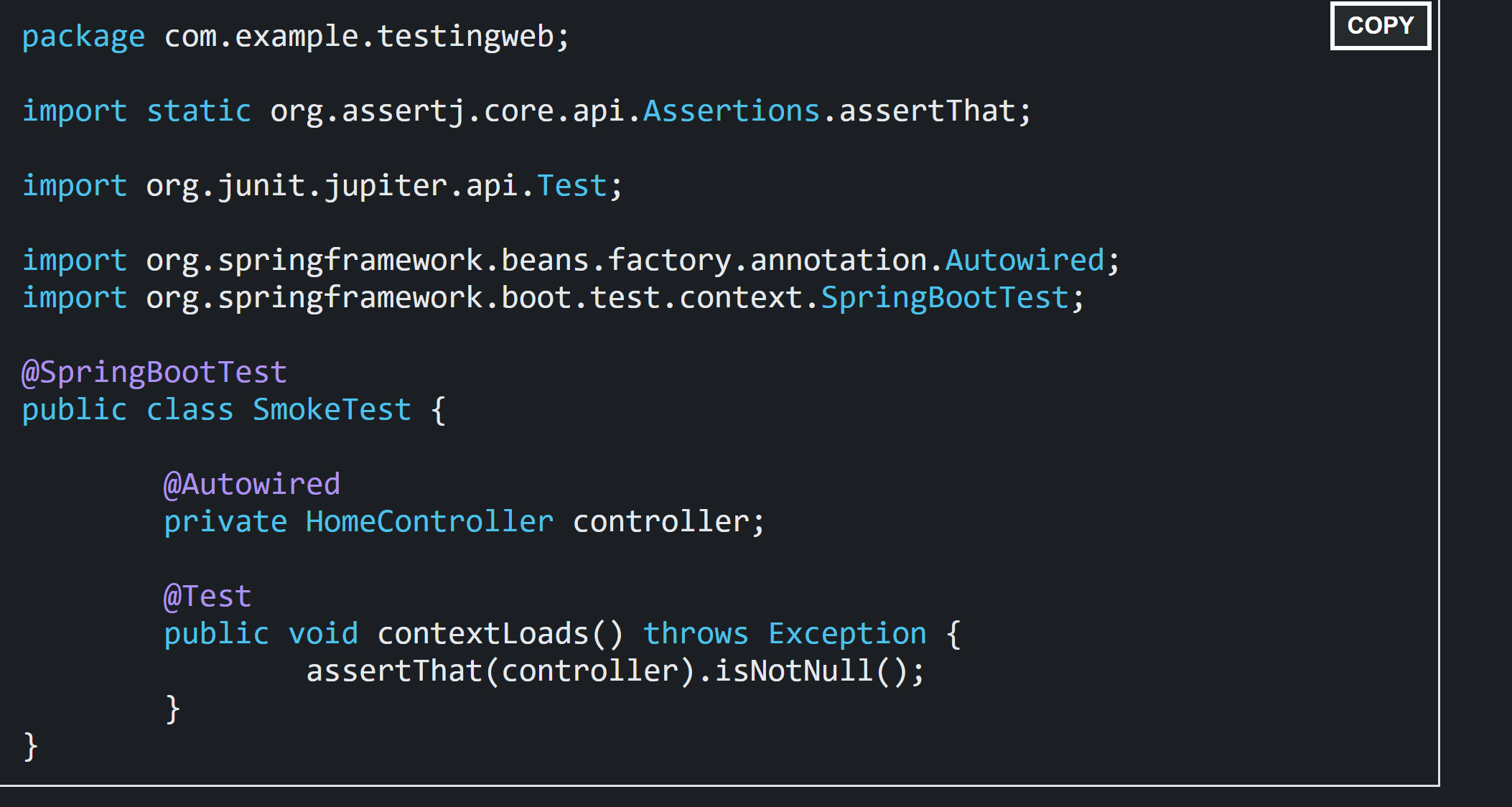
### \*\*\*@SpringBootTest with webEnvironment = WebEnvironment.RANDOM\_PORT will also start the management server on a separate random port if your application uses a different port for the management server.

## Junit 4 and 5

### If you are using JUnit 4, don’t forget to also add @RunWith(SpringRunner.class) to your test, otherwise the annotations will be ignored.

### If you are using JUnit 5, there’s no need to add the equivalent @ExtendWith(SpringExtension.class) as @SpringBootTest and the other @…Test annotations are already annotated with it.

## How to test if spring has created controller or other components?



Spring interprets the @Autowired annotation, and the controller is injected before the test methods are run. We use AssertJ (which provides assertThat() and other methods) to express the test assertions.

A nice feature of the Spring Test support is that the application context is cached between tests. That way, if you have multiple methods in a test case or multiple test cases with the same configuration, they incur the cost of starting the application only once. You can control the cache by using the @DirtiesContext annotation.

## @MockBean usage

### Our Service layer code is dependent on our Repository:

@Service

**public** **class** **EmployeeServiceImpl** **implements** **EmployeeService** {

@Autowired

**private** EmployeeRepository employeeRepository;

@Override

**public** Employee **getEmployeeByName**(String name) {

**return** employeeRepository.findByName(name);

}

}

### However, to test the Service layer, we don't need to know or care about how the persistence layer is implemented. Ideally, we should be able to write and test our Service layer code without wiring in our full persistence layer.

### To achieve this, **we can use the mocking support provided by Spring Boot Test.**

### Let's have a look at the test class skeleton first:

\*\* use of @TestConfiguration is described in this document in later section.

@RunWith(SpringRunner.class)

**public** **class** **EmployeeServiceImplIntegrationTest** {

@TestConfiguration

**static** **class** **EmployeeServiceImplTestContextConfiguration** {

@Bean

**public** EmployeeService **employeeService**() {

**return** **new** EmployeeServiceImpl();

}

}

@Autowired

**private** EmployeeService employeeService;

@MockBean

**private** EmployeeRepository employeeRepository;

// write test cases here

}

### To check the Service class, we need to have an instance of the Service class created and available as a @Bean so that we can @Autowire it in our test class. We can achieve this configuration using the @TestConfiguration annotation.

### Another interesting thing here is the use of @MockBean. It [creates a Mock](https://www.baeldung.com/mockito-mock-methods) for the EmployeeRepository, which can be used to bypass the call to the actual EmployeeRepository:

@Before

**public** **void** **setUp**() {

Employee alex = **new** Employee("alex");

Mockito.when(employeeRepository.findByName(alex.getName()))

.thenReturn(alex);

}

### Since the setup is done, the test case will be simpler:

@Test

**public** **void** **whenValidName\_thenEmployeeShouldBeFound**() {

String name = "alex";

Employee found = employeeService.getEmployeeByName(name);

assertThat(found.getName())

.isEqualTo(name);

}

## @RunWith(SpringRunner.class)

### @RunWith(SpringRunner.class) provides a bridge between Spring Boot test features and JUnit. Whenever we are using any Spring Boot testing features in our JUnit tests, this annotation will be required.

# Testing web layer or Rest Controllers

There are multiple ways to test web layer

1. Rest api – TestRestTemplate
2. MockMVC
3. @webMvcTest

## Using TestRestTemplate



Note the use of webEnvironment=RANDOM\_PORT to start the server with a random port (useful to avoid conflicts in test environments) and the injection of the port with @LocalServerPort. Also, note that Spring Boot has automatically provided a TestRestTemplate for you. All you have to do is add @Autowired to it.

## Spring MockMVC

### Another useful approach is to not start the server at all but to test only the layer below that, where Spring handles the incoming HTTP request and hands it off to your controller. That way, almost of the full stack is used, and your code will be called in exactly the same way as if it were processing a real HTTP request but without the cost of starting the server. To do that, use Spring’s MockMvc and ask for that to be injected for you by using the @AutoConfigureMockMvc annotation on the test case.



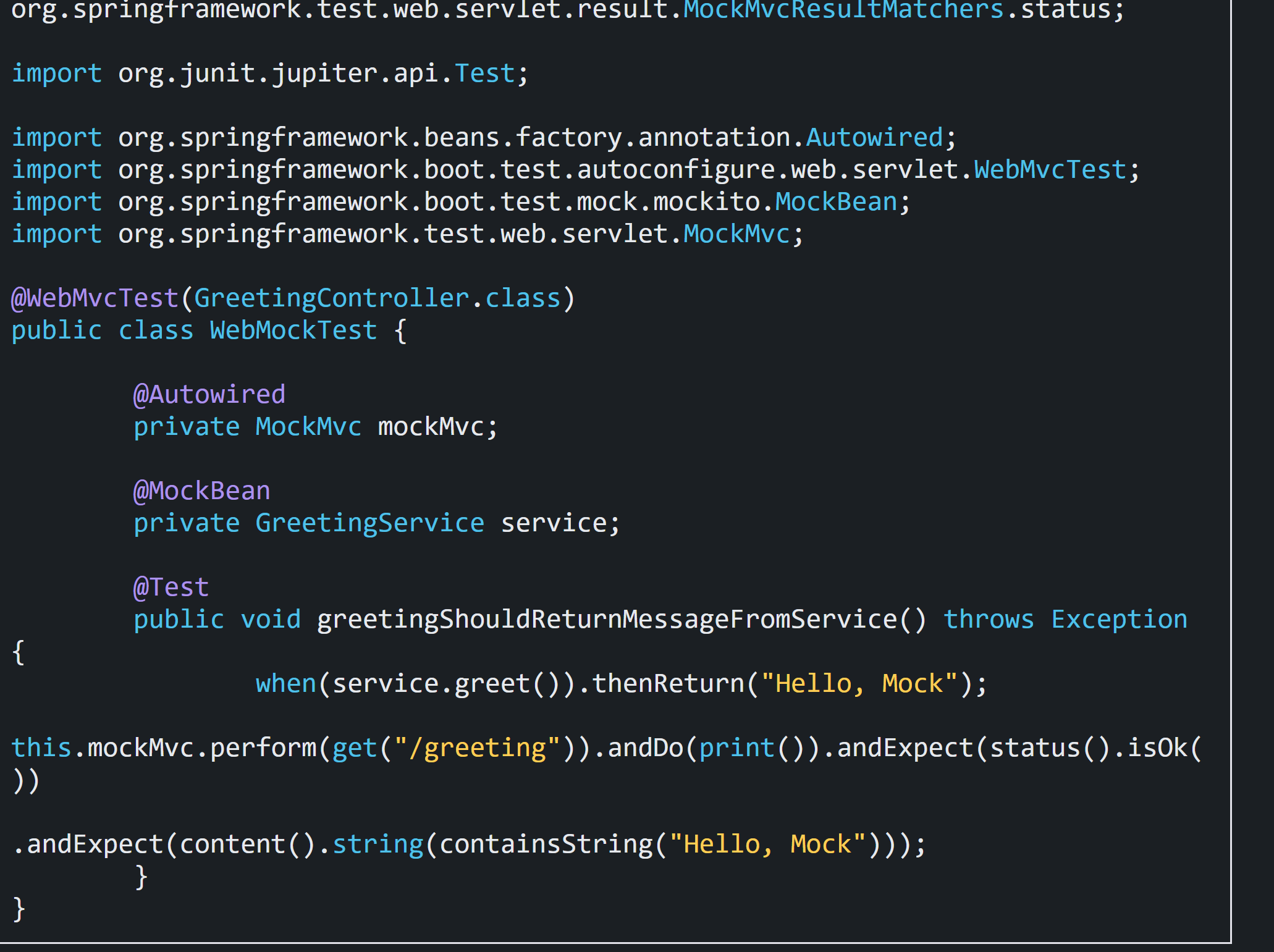
In this test, the full Spring application context is started but without the server

## @WebMvcTest

### The test assertion is the same as in the previous case. However, in this test, Spring Boot instantiates only the web layer rather than the whole context. In an application with multiple controllers, you can even ask for only one to be instantiated by using, for example, @WebMvcTest(HomeController.class).We can narrow the tests to only the web layer by using @WebMvcTest.



### **If your controller contains dependencies , then use @MockBean.**



One more example – Assume there is StudentController class, which uses StudentService to get data.

@RunWith(SpringRunner.class)

@WebMvcTest(value = StudentController.class)

@WithMockUser

public class StudentControllerTest {

@Autowired

private MockMvc mockMvc;

@MockBean

private StudentService studentService;

Course mockCourse = new Course("Course1", "Spring", "10Steps",

Arrays.asList("Learn Maven", "Import Project", "First Example",

"Second Example"));

String exampleCourseJson = "{\"name\":\"Spring\",\"description\":\"10Steps\",\"steps\":[\"Learn Maven\",\"Import Project\",\"First Example\",\"Second Example\"]}";

@Test

public void retrieveDetailsForCourse() throws Exception {

Mockito.when(

studentService.retrieveCourse(Mockito.anyString(),

Mockito.anyString())).thenReturn(mockCourse);

RequestBuilder requestBuilder = MockMvcRequestBuilders.get(

"/students/Student1/courses/Course1").accept(

MediaType.APPLICATION\_JSON);

MvcResult result = mockMvc.perform(requestBuilder).andReturn();

System.out.println(result.getResponse());

String expected = "{id:Course1,name:Spring,description:10Steps}";

*// {"id":"Course1","name":"Spring","description":"10 Steps, 25 Examples and 10K Students","steps":["Learn Maven","Import Project","First Example","Second Example"]}*

JSONAssert.assertEquals(expected, result.getResponse()

.getContentAsString(), false);

}

@Test

public void createStudentCourse() throws Exception {

Course mockCourse = new Course("1", "Smallest Number", "1",

Arrays.asList("1", "2", "3", "4"));

*// studentService.addCourse to respond back with mockCourse*

Mockito.when(

studentService.addCourse(Mockito.anyString(),

Mockito.any(Course.class))).thenReturn(mockCourse);

*// Send course as body to /students/Student1/courses*

RequestBuilder requestBuilder = MockMvcRequestBuilders

.post("/students/Student1/courses")

.accept(MediaType.APPLICATION\_JSON).content(exampleCourseJson)

.contentType(MediaType.APPLICATION\_JSON);

MvcResult result = mockMvc.perform(requestBuilder).andReturn();

MockHttpServletResponse response = result.getResponse();

assertEquals(HttpStatus.CREATED.value(), response.getStatus());

assertEquals("http://localhost/students/Student1/courses/1",

response.getHeader(HttpHeaders.LOCATION));

}

}

# Data JPA layer test

## How to test data layer using @DataJpaTest?

### @DataJpaTest is used to test JPA repositories. It is used in combination with @RunWith(SpringRunner.class).

### The annotation disables full auto-configuration and applies only configuration relevant to JPA tests.

### By default, tests annotated with @DataJpaTest use an embedded in-memory database. If you don’t want that ,we can configure for a real database with @AutoConfigureTestDatabase annotation:

@DataJpaTest

@AutoConfigureTestDatabase(replace=Replace.NONE)

class YourRepositoryTests {

}

### In our tests, we can inject a DataSource, @JdbcTemplate, @EntityManager or any Spring Data repository from our application.

### The application context containing all these components, including the in-memory database, is shared between all test methods within all test classes annotated with @DataJpaTest. Therefore, each test method runs in its own transaction, which is rolled.

### By default, tests annotated with @DataJpaTest are transactional and roll back at the end of each test.

### If you don’t want it, you can disable transaction management for a test or for the whole class using @Transactional annotation:

@DataJpaTest

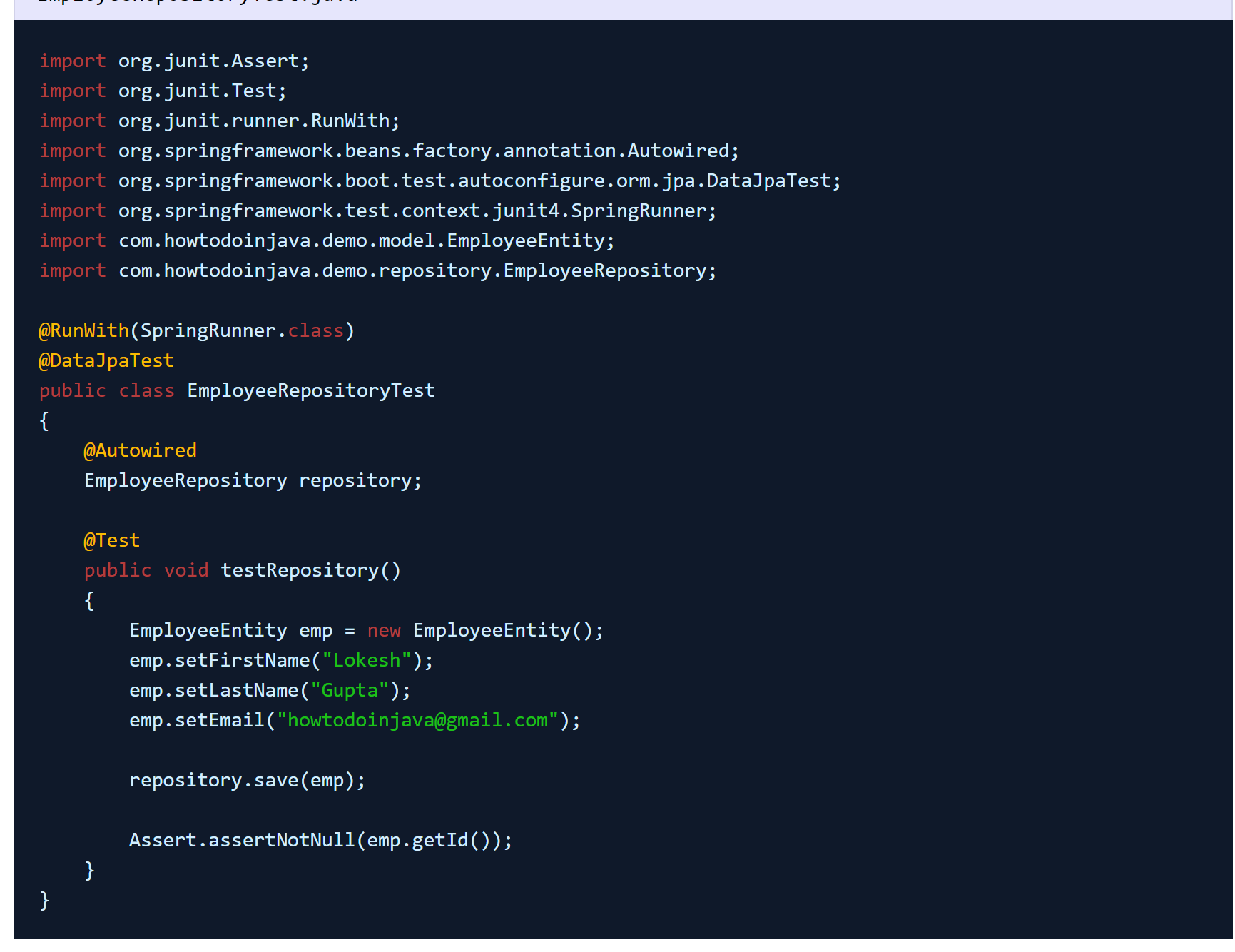
@Transactional(propagation = Propagation.NOT\_SUPPORTED)

class YourNonTransactionalTests {

}

### By default, it scans for @Entity classes and configures Spring Data JPA repositories annotated with @Repository annotation.

### A nice feature is that these tests may also inject a [TestEntityManager](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/orm/jpa/TestEntityManager.html) bean, which provides an alternative to the standard JPA EntityManager that is specifically designed for tests.



Watch the program output in console. You will find that only JPA related configuration has been loaded into context and then testcase execution starts.

## More example:

import static org.junit.Assert.assertEquals;

import static org.junit.Assert.assertNotNull;

import org.junit.Test;

import org.junit.runner.RunWith;

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

import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

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

@RunWith(SpringRunner.class)

@DataJpaTest

public class EmployeeRepositoryTests {

@Autowired

private EmployeeRepository employeeRepository;

@Test

public void testSaveEmployee() {

Employee employee = new Employee("admin", "admin", "admin@gmail.com");

employeeRepository.save(employee);

Employee employee2 = employeeRepository.findByFirstName("admin");

assertNotNull(employee);

assertEquals(employee2.getFirstName(), employee.getFirstName());

assertEquals(employee2.getLastName(), employee.getLastName());

}

@Test

public void testGetEmployee() {

Employee employee = new Employee("admin", "admin", "admin@gmail.com");

employeeRepository.save(employee);

Employee employee2 = employeeRepository.findByFirstName("admin");

assertNotNull(employee);

assertEquals(employee2.getFirstName(), employee.getFirstName());

assertEquals(employee2.getLastName(), employee.getLastName());

}

@Test

public void testDeleteEmployee() {

Employee employee = new Employee("admin", "admin", "admin@gmail.com");

employeeRepository.save(employee);

employeeRepository.delete(employee);

}

@Test

public void findAllEmployees() {

Employee employee = new Employee("admin", "admin", "admin@gmail.com");

employeeRepository.save(employee);

assertNotNull(employeeRepository.findAll());

}

@Test

public void deletByEmployeeIdTest() {

Employee employee = new Employee("admin", "admin", "admin@gmail.com");

Employee emp = employeeRepository.save(employee);

employeeRepository.deleteById(emp.getId());

}

}

<https://www.javaguides.net/2018/09/spring-data-jpa-repository-testing-using-spring-boot-datajpatest.html>

<https://bezkoder.com/spring-boot-unit-test-jpa-repo-datajpatest/>

## testEntityManager:

package net.codejava;

import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.Test;

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

import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;

@DataJpaTest

public class ProductRepositoryTests {

    @Autowired

    private TestEntityManager entityManager;

    @Autowired

    private ProductRepository repository;

    @Test

    public void testSaveNewProduct() {

        entityManager.persist(new Product("iPhone 10", 1099));

        Product product = repository.findByName("iPhone 10");

        assertThat(product.getName()).isEqualTo("iPhone 10");

    }

}

in this test class we can inject a TestEntityManager and ProductRepository. TestEntityManager is a subset of JPA EntityManager. It allows us to quickly test JPA without the need to manually configure/instantiating an EntityManagerFactory and EntityManager.

<https://www.codejava.net/frameworks/spring-boot/junit-tests-for-spring-data-jpa>

# JDBC TEST

## example of jdbc test

https://www.baeldung.com/spring-jdbctemplate-testing

# RestClientTest

## example of restClientTest

### **REST Client Support in Spring Boot Pre-1.4**

### Spring Boot is a handy framework that provides many auto-configured Spring beans with typical settings that allow you to concentrate less on configuration of a Spring application and more on your code and business logic.

### But in version 1.3 we don't get a lot of help when we want to create or test REST services clients. Its support for REST clients is not very deep.

### To create a client for a REST API – a RestTemplate instance is typically used. Usually it has to be configured before usage and its configuration may vary, so Spring Boot does not provide any universally configured RestTemplate bean.

### Same goes for testing REST clients. Before Spring Boot 1.4.0, the procedure of testing a Spring REST client was not very different than in any other Spring-based application. You would create a MockRestServiceServer instance, bind it to RestTemplate instance under test and provide it with mock responses to requests, like this:

RestTemplate restTemplate = **new** RestTemplate();

MockRestServiceServer mockServer =

MockRestServiceServer.bindTo(restTemplate).build();

mockServer.expect(requestTo("/greeting"))

.andRespond(withSuccess());

// Test code that uses the above RestTemplate ...

mockServer.verify();

### You would also have to initialize the Spring container and make sure that only the needed components are loaded into the context, to speed up the context load time (and consequently, the test execution time).

**New REST Client Features in Spring Boot 1.4+**

### In Spring Boot 1.4, the team has made a solid effort to simplify and speed up the creation and testing of REST clients.

### So, let's check out the new features.

### **-> Adding Spring Boot to Your Project**

### First, you'll need to make sure your project is using Spring Boot 1.4.x or higher:

<**parent**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-parent</**artifactId**>

<**relativePath**/> <!-- lookup parent from repository -->

</**parent**>

<**dependencies**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-web</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-test</**artifactId**>

<**scope**>test</**scope**>

</**dependency**>

</**dependencies**>

Newest release versions can be found [here](https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-parent).

### **->**RestTemplateBuilder

### Spring Boot brings both the auto-configured RestTemplateBuilder to simplify creating RestTemplates, and the matching @RestClientTest annotation to test the clients built with RestTemplateBuilder. Here’s how you can create a simple REST client with RestTemplateBuilder auto-injected for you:

@Service

**public** **class** **DetailsServiceClient** {

**private** **final** RestTemplate restTemplate;

**public** **DetailsServiceClient**(RestTemplateBuilder restTemplateBuilder) {

restTemplate = restTemplateBuilder.build();

}

**public** Details **getUserDetails**(String name) {

**return** restTemplate.getForObject("/{name}/details",

Details.class, name);

}

}

### Notice that we did not explicitly wire the RestTemplateBuilder instance to a constructor. This is possible thanks to a new Spring feature called implicit constructor injection, which is discussed in [this article](https://www.baeldung.com/whats-new-in-spring-4-3).

### RestTemplateBuilder provides convenience methods for registering message converters, error handlers, URI template handlers, basic authorization and also use any additional customizers that you need.

### **->**@RestClientTest

### For testing such a REST client built with RestTemplateBuilder, you may use a SpringRunner-executed test class annotated with @RestClientTest. This annotation disables full auto-configuration and only applies configuration relevant to REST client tests, i.e. Jackson or GSON auto-configuration and @JsonComponent beans, but not regular @Component beans.

### @RestClientTest ensures that Jackson and GSON support is auto-configured, and also adds pre-configured RestTemplateBuilder and MockRestServiceServer instances to the context. The bean under test is specified with value or components attribute of the @RestClientTest annotation:

@RunWith(SpringRunner.class)

@RestClientTest(DetailsServiceClient.class)

**public** **class** **DetailsServiceClientTest** {

@Autowired

**private** DetailsServiceClient client;

@Autowired

**private** MockRestServiceServer server;

@Autowired

**private** ObjectMapper objectMapper;

@Before

**public** **void** **setUp**() **throws** Exception {

String detailsString =

objectMapper.writeValueAsString(**new** Details("John Smith", "john"));

**this**.server.expect(requestTo("/john/details"))

.andRespond(withSuccess(detailsString, MediaType.APPLICATION\_JSON));

}

@Test

**public** **void** **whenCallingGetUserDetails\_thenClientMakesCorrectCall**()

**throws** Exception {

Details details = **this**.client.getUserDetails("john");

assertThat(details.getLogin()).isEqualTo("john");

assertThat(details.getName()).isEqualTo("John Smith");

}

}

### Firstly, we need to ensure that this test is run with SpringRunner by adding the @RunWith(SpringRunner.class) annotation.

### **So, what's new?**

### **First** – the @RestClientTest annotation allows us to specify the exact service under test – in our case it is the DetailsServiceClient class. This service will be loaded into the test context, while everything else is filtered out.

### This allows us to autowire the DetailsServiceClient instance inside our test and leave everything else outside, which speeds up the loading of the context.

### **Second** – as the MockRestServiceServer instance is also configured for a @RestClientTest-annotated test (and bound to the DetailsServiceClient instance for us), we can simply inject it and use.

### **Finally** – JSON support for @RestClientTest allows us to inject the Jackson’s ObjectMapper instance to prepare the MockRestServiceServer’s mock answer value.

### **All that is left to do is to execute the call to our service and verify the results.**

# SpringBootTest Customization

## How to customize SpringBootTest

<https://reflectoring.io/spring-boot-test/>

### A code base with a lot of @SpringBootTest-annotated tests may take quite some time to run. The Spring test support is smart enough to only create an application context once and re-use it in following tests, but if different tests need different application contexts, it will still create a separate context for each test, which takes some time for each test.

### All of the customizing options described above will cause Spring to create a new application context. So, we might want to create one single configuration and use it for all tests so that the application context can be re-used.

## @TestConfiguration

### *@SpringBootTest* will bootstrap the full application context, which means we can *@Autowire* any bean that's picked up by component scanning into our test:

@RunWith(SpringRunner.class)

@SpringBootTest

**public** **class** **EmployeeServiceImplIntegrationTest** {

@Autowired

**private** EmployeeService employeeService;

// class code ...

}

### However, we might want to avoid bootstrapping the real application context but use a special test configuration. We can achieve this with the *@TestConfiguration* annotation. There are two ways of using the annotation. Either on a static inner class in the same test class where we want to *@Autowire* the bean:

@RunWith(SpringRunner.class)

**public** **class** **EmployeeServiceImplIntegrationTest** {

@TestConfiguration

**static** **class** **EmployeeServiceImplTestContextConfiguration** {

@Bean

**public** EmployeeService **employeeService**() {

**return** **new** EmployeeService() {

// implement methods

};

}

}

@Autowired

**private** EmployeeService employeeService;

}

### Alternatively, we can create a separate test configuration class:

@TestConfiguration

**public** **class** **EmployeeServiceImplTestContextConfiguration** {

@Bean

**public** EmployeeService **employeeService**() {

**return** **new** EmployeeService() {

// implement methods

};

}

}

### Configuration classes annotated with *@TestConfiguration* are excluded from component scanning, therefore we need to import it explicitly in every test where we want to *@Autowire* it. We can do that with the *@Import* annotation:

@RunWith(SpringRunner.class)

@Import(EmployeeServiceImplTestContextConfiguration.class)

**public** **class** **EmployeeServiceImplIntegrationTest** {

@Autowired

**private** EmployeeService employeeService;

// remaining class code

}

# Reference:

<https://docs.spring.io/spring-boot/docs/2.1.5.RELEASE/reference/html/boot-features-testing.html>

<https://www.springboottutorial.com/unit-testing-for-spring-boot-rest-services>

<https://www.baeldung.com/spring-boot-testing>

data jpa

<https://bezkoder.com/spring-boot-unit-test-jpa-repo-datajpatest/>