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

**[Reflectoring Logo](https://reflectoring.io/)**

* [**Spring Boot**](https://reflectoring.io/categories/spring-boot)
* [**Java**](https://reflectoring.io/categories/java)
* [**Software Craft**](https://reflectoring.io/categories/craft)
* [**Book Reviews**](https://reflectoring.io/categories/book-reviews)
* [**Meta**](https://reflectoring.io/categories/meta)

[**SPRING BOOT**](https://reflectoring.io/categories/spring-boot)

1. **Testing Spring MVC Web Controllers with @WebMvcTest**

In this second part of the series on testing with Spring Boot, we’re going to look at web controllers. First, we’re going to explore what a web controller actually does so that we can build tests that cover all of its responsibilities.

Then, we’re going to find out how to cover each of those responsibilities in a test. Only with those responsibilities covered can we be sure that our controllers behave as expected in a production environment.

1. **Code Example**

This article is accompanied by a working code example [on GitHub](https://github.com/thombergs/code-examples/tree/master/spring-boot/spring-boot-testing).

1. **The “Testing with Spring Boot” Series**

This tutorial is part of a series:

1. [Unit Testing with Spring Boot](https://reflectoring.io/unit-testing-spring-boot/)
2. [Testing Spring MVC Web Controllers with @WebMvcTest](https://reflectoring.io/spring-boot-web-controller-test/)
3. [Testing JPA Queries with @DataJpaTest](https://reflectoring.io/spring-boot-data-jpa-test/)
4. [Integration Tests with @SpringBootTest](https://reflectoring.io/spring-boot-test/)
5. **Dependencies**

We’re going to use JUnit Jupiter (JUnit 5) as the testing framework, Mockito for mocking, AssertJ for creating assertions and Lombok to reduce boilerplate code:

dependencies **{**

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

compileOnly**(**'org.projectlombok:lombok'**)**

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

testCompile 'org.junit.jupiter:junit-jupiter-engine:5.2.0'

testCompile**(**'org.mockito:mockito-junit-jupiter:2.23.0'**)**

**}**

AssertJ and Mockito automatically come with the dependency to spring-boot-starter-test.

1. **Responsibilities of a Web Controller**

Let’s start by looking at a typical REST controller:

**@RestController**

**@RequiredArgsConstructor**

**class** **RegisterRestController** **{**

**private** **final** **RegisterUseCase** registerUseCase**;**

**@PostMapping(**"/forums/{forumId}/register"**)**

**UserResource** **register(**

**@PathVariable(**"forumId"**)** **Long** forumId**,**

**@Valid** **@RequestBody** **UserResource** userResource**,**

**@RequestParam(**"sendWelcomeMail"**)** **boolean** sendWelcomeMail**)** **{**

**User** user **=** **new** **User(**

userResource**.**getName**(),**

userResource**.**getEmail**());**

**Long** userId **=** registerUseCase**.**registerUser**(**user**,** sendWelcomeMail**);**

**return** **new** **UserResource(**

userId**,**

user**.**getName**(),**

user**.**getEmail**());**

**}**

**}**

The controller method is annotated with @PostMapping to define the URL, HTTP method and content type it should listen to.

It takes input via parameters annotated with @PathVariable, @RequestBody, and @RequestParam which are automatically filled from the incoming HTTP request.

Parameters my be annotated with @Valid to indicate that Spring should perform [bean validation](https://reflectoring.io/bean-validation-with-spring-boot/) on them.

The controller then works with those parameters, calling the business logic before returning a plain Java object, which is automatically mapped into JSON and written into the HTTP response body by default.

There’s a lot of Spring magic going on here. In summary, for each request, a controller usually does the following steps:

| **#** | **Responsibility** | **Description** |
| --- | --- | --- |
| 1. | **Listen to HTTP Requests** | The controller should respond to certain URLs, HTTP methods and content types. |
| 2. | **Deserialize Input** | The controller should parse the incoming HTTP request and create Java objects from variables in the URL, HTTP request parameters and the request body so that we can work with them in the code. |
| 3. | **Validate Input** | The controller is the first line of defense against bad input, so it’s a place where we can validate the input. |
| 4. | **Call the Business Logic** | Having parsed the input, the controller must transform the input into the model expected by the business logic and pass it on to the business logic. |
| 5. | **Serialize the Output** | The controller takes the output of the business logic and serializes it into an HTTP response. |
| 6. | **Translate Exceptions** | If an exception occurs somewhere on the way, the controller should translate it into a meaningful error message and HTTP status for the user. |

A controller apparently has a lot to do!  
**We should take care not to add even more responsibilities like performing business logic**. Otherwise, our controller tests will become fat and unmaintainable.

How are we going to write meaningful tests that cover all of those responsibilities?

1. **Unit or Integration Test?**

Do we write unit tests? Or integration tests? What’s the difference, anyways? Let’s discuss both approaches and decide for one.

**In a unit test, we would test the controller in isolation**. That means we would instantiate a controller object, [mocking away the business logic](https://reflectoring.io/unit-testing-spring-boot/#using-mockito-to-mock-dependencies), and then call the controller’s methods and verify the response.

Would that work in our case? Let’s check which of the 6 responsibilities we have identified above we can cover in an isolated unit test:

| **#** | **Responsibility** | **Covered in a Unit Test?** |
| --- | --- | --- |
| 1. | **Listen to HTTP Requests** | No, because the unit test would not evaluate the @PostMapping annotation and similar annotations specifying the properties of a HTTP request. |
| 2. | **Deserialize Input** | No, because annotations like @RequestParam and @PathVariable would not be evaluated. Instead we would provide the input as Java objects, effectively skipping deserialization from an HTTP request. |
| 3. | **Validate Input** | Not when depending on bean validation, because the @Valid annotation would not be evaluated. |
| 4. | **Call the Business Logic** | Yes, because we can verify if the mocked business logic has been called with the expected arguments. |
| 5. | **Serialize the Output** | No, because we can only verify the Java version of the output, and not the HTTP response that would be generated. |
| 6. | **Translate Exceptions** | No. We could check if a certain exception was raised, but not that it was translated to a certain JSON response or HTTP status code. |

In summary, **a simple unit test will not cover the HTTP layer**. So, we need to introduce Spring to our test to do the HTTP magic for us. Thus, we’re building an integration test that tests the integration between our controller code and the components Spring provides for HTTP support.

An integration test with Spring fires up a Spring application context that contains all the beans we need. This includes framework beans that are responsible for listening to certain URLs, serializing and deserializing to and from JSON and translating exceptions to HTTP. These beans will evaluate the annotations that would be ignored by a simple unit test.

So, how do we do it?

1. **Verifying Controller Responsibilities with @WebMvcTest**

Spring Boot provides the @WebMvcTest annotation to fire up an application context that contains only the beans needed for testing a web controller:

**@ExtendWith(SpringExtension.**class**)**

**@WebMvcTest(**controllers **=** **RegisterRestController.**class**)**

**class** **RegisterRestControllerTest** **{**

**@Autowired**

**private** **MockMvc** mockMvc**;**

**@Autowired**

**private** **ObjectMapper** objectMapper**;**

**@MockBean**

**private** **RegisterUseCase** registerUseCase**;**

**@Test**

**void** **whenValidInput\_thenReturns200()** **throws** **Exception** **{**

mockMvc**.**perform**(...);**

**}**

**}**

1. **@ExtendWith**

The code examples in this tutorial use the @ExtendWith annotation to tell JUnit 5 to enable Spring support. [As of Spring Boot 2.1](https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.1-Release-Notes#junit-5), we no longer need to load the SpringExtension because it's included as a meta annotation in the Spring Boot test annotations like @DataJpaTest, @WebMvcTest, and @SpringBootTest.

We can now @Autowire all the beans we need from the application context. Spring Boot automatically provides beans like an @ObjectMapper to map to and from JSON and a MockMvc instance to simulate HTTP requests.

We use @MockBean to mock away the business logic, since we don’t want to test integration between controller and business logic, but between controller and the HTTP layer. @MockBean automatically replaces the bean of the same type in the application context with a Mockito mock.

You can read more about the @MockBean annotation in [my article](https://reflectoring.io/spring-boot-mock/) about mocking.

1. **Use @WebMvcTest with or without the controllers parameter?**

By setting the controllers parameter to RegisterRestController.class in the example above, we're telling Spring Boot to restrict the application context created for this test to the given controller bean and some framework beans needed for Spring Web MVC. All other beans we might need have to be included separately or mocked away with @MockBean.

If we leave away the controllers parameter, Spring Boot will include *all* controllers in the application context. Thus, we need to include or mock away *all* beans any controller depends on. This makes for a much more complex test setup with more dependencies, but saves runtime since all controller tests will re-use the same application context.

I tend to restrict the controller tests to the narrowest application context possible in order to make the tests independent of beans that I don't even need in my test, even though Spring Boot has to create a new application context for each single test.

Let’s go through each of the responsibilities and see how we can use MockMvc to verify each of them in order build the best integration test we can.

1. **1. Verifying HTTP Request Matching**

Verifying that a controller listens to a certain HTTP request is pretty straightforward. We simply call the perform() method of MockMvc and provide the URL we want to test:

mockMvc**.**perform**(**post**(**"/forums/42/register"**)**

**.**contentType**(**"application/json"**))**

**.**andExpect**(**status**().**isOk**());**

Aside from verifying that the controller responds to a certain URL, this test also verifies the correct HTTP method (POST in our case) and the correct request content type. The controller we have seen above would reject any requests with a different HTTP method or content type.

Note that this test would still fail, yet, since our controller expects some input parameters.

More options to match HTTP requests can be found in the Javadoc of [MockHttpServletRequestBuilder](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/web/servlet/request/MockHttpServletRequestBuilder.html).

1. **2. Verifying Input Serialization**

To verify that the input is successfully serialized into Java objects, we have to provide it in the test request. Input can be either the JSON content of the request body (@RequestBody), a variable within the URL path (@PathVariable), or an HTTP request parameter (@RequestParam):

**@Test**

**void** **whenValidInput\_thenReturns200()** **throws** **Exception** **{**

**UserResource** user **=** **new** **UserResource(**"Zaphod"**,** "zaphod@galaxy.net"**);**

mockMvc**.**perform**(**post**(**"/forums/{forumId}/register"**,** 42L**)**

**.**contentType**(**"application/json"**)**

**.**param**(**"sendWelcomeMail"**,** "true"**)**

**.**content**(**objectMapper**.**writeValueAsString**(**user**)))**

**.**andExpect**(**status**().**isOk**());**

**}**

We now provide the path variable forumId, the request parameter sendWelcomeMail and the request body that are expected by the controller. The request body is generated using the ObjectMapper provided by Spring Boot, serializing a UserResource object to a JSON string.

If the test is green, we now know that the controller’s register() method has received those parameters as Java objects and that they have been successfully parsed from the HTTP request.

1. **3. Verifying Input Validation**

Let’s say the UserResource uses the @NotNull annotation to deny null values:

**@Value**

**public** **class** **UserResource** **{**

**@NotNull**

**private** **final** **String** name**;**

**@NotNull**

**private** **final** **String** email**;**

**}**

Bean validation is triggered automatically when we [add the @Valid annotation to a method parameter](https://reflectoring.io/bean-validation-with-spring-boot/#validating-input-to-a-spring-mvc-controller) like we did with the userResource parameter in our controller. So, for the happy path (i.e. when the validation succeeds), the test we created in the previous section is enough.

If we want to test if the validation fails as expected, we need to add a test case in which we send an invalid UserResource JSON object to the controller. We then expect the controller to return HTTP status 400 (Bad Request):

**@Test**

**void** **whenNullValue\_thenReturns400()** **throws** **Exception** **{**

**UserResource** user **=** **new** **UserResource(null,** "zaphod@galaxy.net"**);**

mockMvc**.**perform**(**post**(**"/forums/{forumId}/register"**,** 42L**)**

**...**

**.**content**(**objectMapper**.**writeValueAsString**(**user**)))**

**.**andExpect**(**status**().**isBadRequest**());**

**}**

Depending on how important the validation is for the application, we might add a test case like this for each invalid value that is possible. This can quickly add up to a lot of test cases, though, so you should talk to your team about how you want to handle validation tests in your project.

1. **4. Verifying Business Logic Calls**

Next, we want to verify that the business logic is called as expected. In our case, the business logic is provided by the RegisterUseCase interface and expects a User object and a boolean as input:

**interface** **RegisterUseCase** **{**

**Long** **registerUser(User** user**,** **boolean** sendWelcomeMail**);**

**}**

We expect the controller to transform the incoming UserResource object into a User and to pass this object into the registerUser() method.

To verify this, we can ask the RegisterUseCase mock, which has been injected into the application context with the @MockBean annotation:

**@Test**

**void** **whenValidInput\_thenMapsToBusinessModel()** **throws** **Exception** **{**

**UserResource** user **=** **new** **UserResource(**"Zaphod"**,** "zaphod@galaxy.net"**);**

mockMvc**.**perform**(...);**

**ArgumentCaptor<User>** userCaptor **=** **ArgumentCaptor.**forClass**(User.**class**);**

verify**(**registerUseCase**,** times**(**1**)).**registerUser**(**userCaptor**.**capture**(),** eq**(true));**

assertThat**(**userCaptor**.**getValue**().**getName**()).**isEqualTo**(**"Zaphod"**);**

assertThat**(**userCaptor**.**getValue**().**getEmail**()).**isEqualTo**(**"zaphod@galaxy.net"**);**

**}**

After the call to the controller has been performed, we use an ArgumentCaptor to capture the User object that was passed to the RegisterUseCase.registerUser() and assert that it contains the expected values.

The verify call checks that registerUser() has been called exactly once.

Note that if we do a lot of assertions on User objects, we can create [our own custom Mockito assertion methods](https://reflectoring.io/unit-testing-spring-boot/#creating-readable-assertions-with-assertj) for better readability.

1. **5. Verifying Output Serialization**

After the business logic has been called, we expect the controller to map the result into a JSON string and include it in the HTTP response. In our case, we expect the HTTP response body to contain a valid UserResource object in JSON form:

**@Test**

**void** **whenValidInput\_thenReturnsUserResource()** **throws** **Exception** **{**

**MvcResult** mvcResult **=** mockMvc**.**perform**(...)**

**...**

**.**andReturn**();**

**UserResource** expectedResponseBody **=** **...;**

**String** actualResponseBody **=** mvcResult**.**getResponse**().**getContentAsString**();**

assertThat**(**objectMapper**.**writeValueAsString**(**expectedResponseBody**))**

**.**isEqualToIgnoringWhitespace**(**actualResponseBody**);**

**}**

To do assertions on the response body, we need to store the result of the HTTP interaction in a variable of type MvcResult using the andReturn() method.

We can then read the JSON string from the response body and compare it to the expected string using isEqualToIgnoringWhitespace(). We can build the expected JSON string from a Java object using the ObjectMapper provided by Spring Boot.

Note that we can make this much more readable by using a custom ResultMatcher, [as described later](https://reflectoring.io/spring-boot-web-controller-test/#matching-json-output).

1. **6. Verifying Exception Handling**

Usually, if an exception occurs, the controller should return a certain HTTP status. 400, if something is wrong with the request, 500, if an exception bubbles up, and so on.

Spring takes care of most of these cases by default. However, if we have a custom exception handling, we want to test it. Let’s say we want to return a structured JSON error response with a field name and error message for each field that was invalid in the request. We’d create a @ControllerAdvice like this:

**@ControllerAdvice**

**class** **ControllerExceptionHandler** **{**

**@ResponseStatus(HttpStatus.**BAD\_REQUEST**)**

**@ExceptionHandler(MethodArgumentNotValidException.**class**)**

**@ResponseBody**

**ErrorResult** **handleMethodArgumentNotValidException(MethodArgumentNotValidException** e**)** **{**

**ErrorResult** errorResult **=** **new** **ErrorResult();**

**for** **(FieldError** fieldError **:** e**.**getBindingResult**().**getFieldErrors**())** **{**

errorResult**.**getFieldErrors**()**

**.**add**(new** **FieldValidationError(**fieldError**.**getField**(),**

fieldError**.**getDefaultMessage**()));**

**}**

**return** errorResult**;**

**}**

**@Getter**

**@NoArgsConstructor**

**static** **class** **ErrorResult** **{**

**private** **final** **List<FieldValidationError>** fieldErrors **=** **new** **ArrayList<>();**

**ErrorResult(String** field**,** **String** message**){**

**this.**fieldErrors**.**add**(new** **FieldValidationError(**field**,** message**));**

**}**

**}**

**@Getter**

**@AllArgsConstructor**

**static** **class** **FieldValidationError** **{**

**private** **String** field**;**

**private** **String** message**;**

**}**

**}**

If bean validation fails, Spring throws an MethodArgumentNotValidException. We handle this exception by mapping Spring’s FieldError objects into our own ErrorResult data structure. The exception handler causes all controllers to return HTTP status 400 in this case and puts the ErrorResult object into the response body as a JSON string.

To verify that this actually happens, we expand on our earlier test for failing validations:

**@Test**

**void** **whenNullValue\_thenReturns400AndErrorResult()** **throws** **Exception** **{**

**UserResource** user **=** **new** **UserResource(null,** "zaphod@galaxy.net"**);**

**MvcResult** mvcResult **=** mockMvc**.**perform**(...)**

**.**contentType**(**"application/json"**)**

**.**param**(**"sendWelcomeMail"**,** "true"**)**

**.**content**(**objectMapper**.**writeValueAsString**(**user**)))**

**.**andExpect**(**status**().**isBadRequest**())**

**.**andReturn**();**

**ErrorResult** expectedErrorResponse **=** **new** **ErrorResult(**"name"**,** "must not be null"**);**

**String** actualResponseBody **=**

mvcResult**.**getResponse**().**getContentAsString**();**

**String** expectedResponseBody **=**

objectMapper**.**writeValueAsString**(**expectedErrorResponse**);**

assertThat**(**expectedResponseBody**)**

**.**isEqualToIgnoringWhitespace**(**actualResponseBody**);**

**}**

Again, we read the JSON string from the response body and compare it against an expected JSON string. Additionally, we check that the response status is 400.

This, too, can be implemented in a much more readable manner, [as we’ll learn below](https://reflectoring.io/spring-boot-web-controller-test/#matching-expected-validation-errors).

1. **Creating Custom ResultMatchers**

Certain assertions are rather hard to write and, more importantly, hard to read. Especially when we want to compare the JSON string from the HTTP response to an expected value it takes a lot of code, as we have seen in the last two examples.

Luckily, we can create custom ResultMatchers that we can use within the fluent API of MockMvc. Let’s see how we can do this for our use cases.

1. **Matching JSON Output**

Wouldn’t it be nice to use the following code to verify if the HTTP response body contains a JSON representation of a certain Java object?

**@Test**

**void** **whenValidInput\_thenReturnsUserResource\_withFluentApi()** **throws** **Exception** **{**

**UserResource** user **=** **...;**

**UserResource** expected **=** **...;**

mockMvc**.**perform**(...)**

**...**

**.**andExpect**(**responseBody**().**containsObjectAsJson**(**expected**,** **UserResource.**class**));**

**}**

No need to manually compare JSON strings anymore. And it’s much better readable. In fact, the code is so self-explanatory that I’m going to stop explaining here.

To be able to use the above code, we create a custom ResultMatcher:

**public** **class** **ResponseBodyMatchers** **{**

**private** **ObjectMapper** objectMapper **=** **new** **ObjectMapper();**

**public** **<**T**>** **ResultMatcher** **containsObjectAsJson(**

**Object** expectedObject**,**

**Class<**T**>** targetClass**)** **{**

**return** mvcResult **->** **{**

**String** json **=** mvcResult**.**getResponse**().**getContentAsString**();**

T actualObject **=** objectMapper**.**readValue**(**json**,** targetClass**);**

assertThat**(**expectedObject**).**isEqualToComparingFieldByField**(**actualObject**);**

**};**

**}**

**static** **ResponseBodyMatchers** **responseBody(){**

**return** **new** **ResponseBodyMatchers();**

**}**

**}**

The static method responseBody() serves as the entrypoint for our fluent API. It returns the actual ResultMatcher that parses the JSON from the HTTP response body and compares it field by field with the expected object that is passed in.

1. **Matching Expected Validation Errors**

We can even go a step further to simplify our exception handling test. It took us [4 lines of code](https://reflectoring.io/spring-boot-web-controller-test/#validation_code_example) to verify that the JSON response contained a certain error message. We can to it in one line instead:

**@Test**

**void** **whenNullValue\_thenReturns400AndErrorResult\_withFluentApi()** **throws** **Exception** **{**

**UserResource** user **=** **new** **UserResource(null,** "zaphod@galaxy.net"**);**

mockMvc**.**perform**(...)**

**...**

**.**content**(**objectMapper**.**writeValueAsString**(**user**)))**

**.**andExpect**(**status**().**isBadRequest**())**

**.**andExpect**(**responseBody**().**containsError**(**"name"**,** "must not be null"**));**

**}**

Again, the code is self-explanatory.

To enable this fluent API, we must add the method containsErrorMessageForField() to our ResponseBodyMatchers class from above:

**public** **class** **ResponseBodyMatchers** **{**

**private** **ObjectMapper** objectMapper **=** **new** **ObjectMapper();**

**public** **ResultMatcher** **containsError(**

**String** expectedFieldName**,**

**String** expectedMessage**)** **{**

**return** mvcResult **->** **{**

**String** json **=** mvcResult**.**getResponse**().**getContentAsString**();**

**ErrorResult** errorResult **=** objectMapper**.**readValue**(**json**,** **ErrorResult.**class**);**

**List<FieldValidationError>** fieldErrors **=** errorResult**.**getFieldErrors**().**stream**()**

**.**filter**(**fieldError **->** fieldError**.**getField**().**equals**(**expectedFieldName**))**

**.**filter**(**fieldError **->** fieldError**.**getMessage**().**equals**(**expectedMessage**))**

**.**collect**(Collectors.**toList**());**

assertThat**(**fieldErrors**)**

**.**hasSize**(**1**)**

**.**withFailMessage**(**"expecting exactly 1 error message"

**+** "with field name '%s' and message '%s'"**,**

expectedFieldName**,**

expectedMessage**);**

**};**

**}**

**static** **ResponseBodyMatchers** **responseBody()** **{**

**return** **new** **ResponseBodyMatchers();**

**}**

**}**

All the ugly code is hidden within this helper class and we can happily write clean assertions in our integration tests.

1. **Conclusion**

Web controllers have a lot of responsibilities. If we want to cover a web controller with meaningful tests, it’s not enough to just check if it returns the correct HTTP status.

With @WebMvcTest, Spring Boot provides everything we need to build web controller tests, but for the tests to be meaningful, we need to remember to cover all of the responsibilities. Otherwise, we may be in for ugly surprises at runtime.

The example code from this article is available [on github](https://github.com/thombergs/code-examples/tree/master/spring-boot/spring-boot-testing).

Follow me on [Twitter](https://twitter.com/TomHombergs), [LinkedIn](https://www.linkedin.com/in/tom-hombergs-74048777/), or my Mailing List to be notified of new content.

**[](https://reflectoring.io/about/)**

1. **Tom Hombergs**

As a professional software engineer, consultant, architect, and general problem solver, I've been practicing the software craft for more than ten years and I'm still learning something new every day. I love sharing the things I learned, so you (and future me) can get a head start.

1. **Get 66% Off My eBook**

**[](https://reflectoring.io/get-your-hands-dirty-on-clean-architecture/)**

Liked this article? Subscribe to my mailing list to get notified about new content and get 66% off my eBook [**"Get Your Hands Dirty on Clean Architecture"**](https://reflectoring.io/e-book/).

SUBSCRIBE [**GET IT AT AMAZON**](https://www.amazon.com/gp/product/1839211962/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=1839211962&linkCode=as2&tag=reflectorin0c-20&linkId=559e54b6599c4213252259df28d1d3e3)