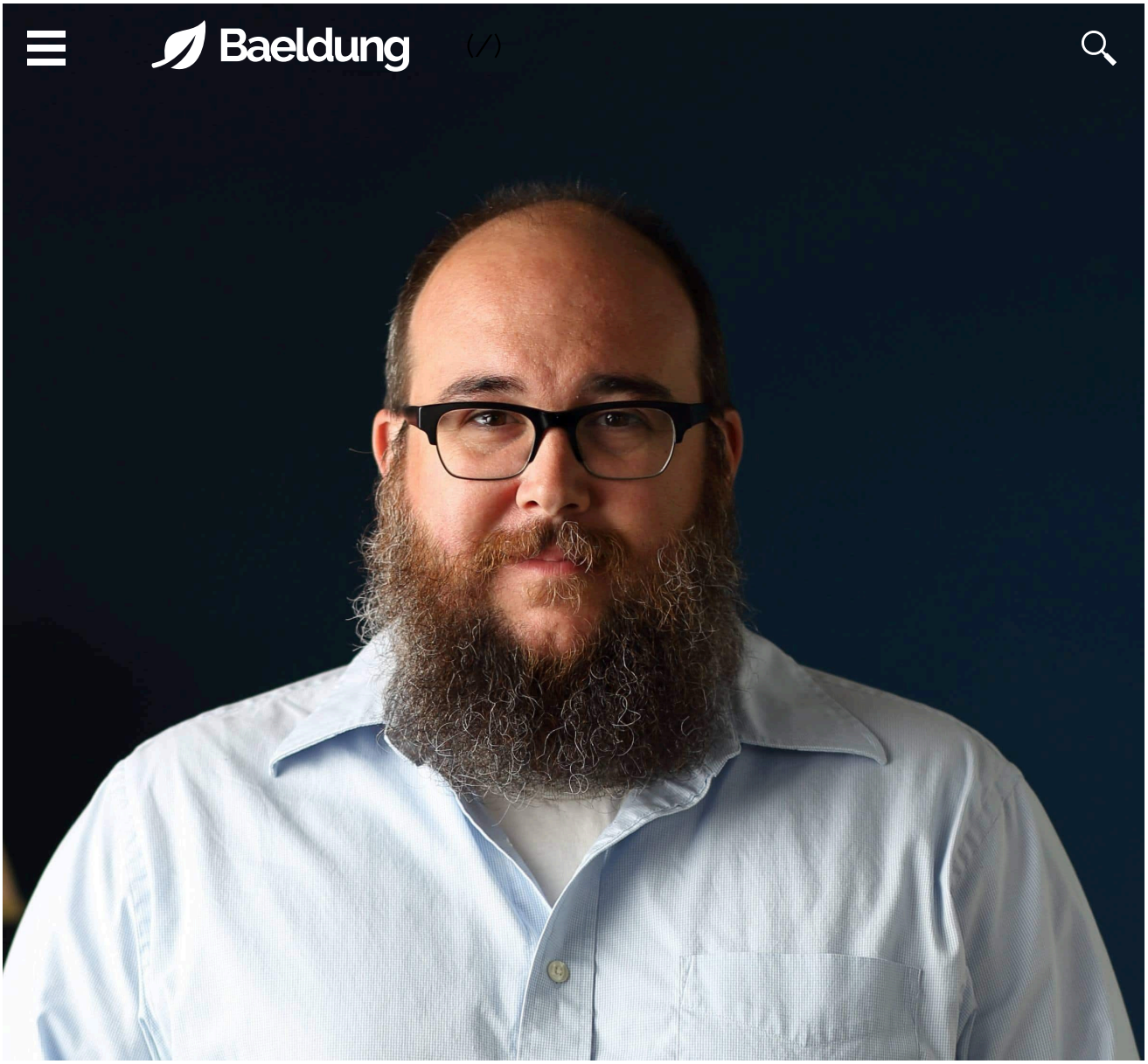
Mock @Value in Spring Boot Test

Last updated: October 6, 2024

Written by: Neetika Khandelwal
(https://www.baeldung.com/author/neetikakhandelwal)



Reviewed by: Eric Martin (https://www.baeldung.com/editor/eric-editor)

Spring Boot (https://www.baeldung.com/category/spring/spring-boot)

Testing (https://www.baeldung.com/category/testing)

Spring Annotations (https://www.baeldung.com/tag/spring-annotations)

End-to-end testing is a very useful method to make sure that your application works as intended. This highlights **issues in the overall functionality** of the software, that the unit and integration test stages may miss.

Playwright is an easy-to-use, but powerful tool that **automates end-to-end testing**, and supports all modern browsers and platforms.

1. Overview

When writing unit tests in Spring Boot, it's common to encounter scenarios where we must mock external configurations or properties loaded using the *@Value* (/spring-value-annotation) annotation (/spring-boot-annotations). **These properties are often loaded from *application.properties* or *application.yml* files and injected into our Spring components.** However, we typically don't want to load the full Spring context (/spring-application-context) with external files. Instead, we'd like to mock these values to keep our tests fast and isolated.

In this tutorial, we'll learn why and how to mock *@Value* in Spring Boot tests to ensure smooth and effective testing without loading the entire application context.

2. How to Mock *@Value* in Spring Boot Tests

Let's have a service class *ValueAnnotationMock* that gets us the value of an external API URL from the *application.properties* file using *@Value*:

```
@Service                                    (/)
public class ValueAnnotationMock {
    @Value("${external.api.url}")
    private String apiUrl;

    public String getApiUrl() {
        return apiUrl;
    }

    public String callExternalApi() {
        return String.format("Calling API at %s", apiUrl);
    }
}
```

Also, this is our *application.properties* file:

```
external.api.url=http://dynamic-url.com
```

So, how can we mock this property in our test class?

There are multiple ways to mock *@Value* annotation. Let's see each one by one.

2.1. Using the *@TestPropertySource* Annotation

The simplest way to mock a *@Value* property in a Spring Boot test is by using the *@TestPropertySource* (/spring-test-property-source) annotation. **This allows us to define properties directly in our test class.**

Let's walk through an example to understand better:

```
@RunWith(SpringJUnit4ClassRunner.class) //
@ContextConfiguration(classes = ValueAnnotationMock.class)
@SpringBootTest
@TestPropertySource(properties = {
    "external.api.url=http://mocked-url.com"
})
public class ValueAnnotationMockTestPropertySourceUnitTest {
    @Autowired
    private ValueAnnotationMock valueAnnotationMock;

    @Test
    public void
givenValue_whenUsingTestPropertySource_thenMockValueAnnotation() {
        String apiUrl = valueAnnotationMock.getApiUrl();
        assertEquals("http://mocked-url.com", apiUrl);
    }
}
```

In this example, the *@TestPropertySource* provides a mock value for the *external.api.url* property, i.e., *http://mocked-url.com*. It's then injected into the *ValueAnnotationMock* bean.

Let's explore some key advantages of using *@TestPropertySource* to mock *@Value:*

- This approach allows us to simulate the real behavior of the application by using Spring's property injection mechanism, which means the code being tested is close to what will run in production.
- We can specify test-specific property values directly in our test class, simplifying the mocking of complex properties.

Now, let's switch and look into some drawbacks of this approach:

- Tests using *@TestPropertySource* load the Spring context, which can be slower than unit tests that don't require context loading.
- This approach can add unnecessary complexity for simple unit tests since it brings in a lot of Spring's machinery, making tests less isolated and slower.

2.2. Using *ReflectionTestUtils*

For cases where we want to directly inject mock values into *private* fields such as those annotated with *@Value*, we can use the *ReflectionTestUtils* (/spring-reflection-test-utils)class of Spring to set the value manually.

Let's see the implementation:

```
public class ValueAnnotationMockReflectionUtilsUnitTest {
    @Autowired
    private ValueAnnotationMock valueAnnotationMock;

    @Test
    public void
givenValue_whenUsingReflectionUtils_thenMockValueAnnotation() {
        valueAnnotationMock = new ValueAnnotationMock();
        ReflectionTestUtils.setField(valueAnnotationMock, "apiUrl",
"http://mocked-url.com");
        String apiUrl = valueAnnotationMock.getApiUrl();
        assertEquals("http://mocked-url.com", apiUrl);
    }
}
```

This method bypasses the entire context of Spring, making it ideal for pure unit tests where we don't want to involve Spring Boot's dependency injection mechanism at all.

Let's see some of the benefits of this approach:

- We can directly manipulate *private* fields, even those annotated with *@Value*, without modifying the original class. This can be helpful for legacy code that cannot be easily refactored.
- It avoids loading the Spring application context, making tests faster and isolated.
- We can test the exact behaviors by changing the internal state of an object dynamically during testing.

There are some downsides to using *ReflectionUtils:*

- Directly accessing or modifying private fields through reflection breaks encapsulation, which goes against best practices for object-oriented design.
- While it doesn't load Spring's context, reflection is slower than standard access due to the overhead of inspecting and modifying class structures at runtime.

2.3. Using Constructor Injection (✓)

This approach uses constructor injection (/constructor-injection-in-spring) to handle *@Value* properties, providing a clean solution for Spring context injection and unit testing without needing reflection or Spring's full environment.

We have a main class where properties like *apiUrl* and *apiPassword* are injected using the *@Value* annotation in the constructor (/java-constructors):

```
public class ValueAnnotationConstructorMock {
    private final String apiUrl;
    private final String apiPassword;

    public ValueAnnotationConstructorMock(@Value("#{myProps['api.url']}")
String apiUrl,
        @Value("#{myProps['api.password']}") String apiPassword) {
        this.apiUrl = apiUrl;
        this.apiPassword = apiPassword;
    }

    public String getApiUrl() {
        return apiUrl;
    }

    public String getApiPassword() {
        return apiPassword;
    }
}
```

Instead of injecting these values via field injection which might require reflection or additional setup in testing, they're passed directly through the constructor. This allows the class to easily instantiate with specific values for testing purposes.

In the test class, we don't need Spring's context to inject these values. Instead, we can simply instantiate the class with arbitrary values:

```
public class ValueAnnotationMockConstructorUnitTest {
    private ValueAnnotationConstructorMock
valueAnnotationConstructorMock;

    @BeforeEach
    public void setUp() {
        valueAnnotationConstructorMock = new
ValueAnnotationConstructorMock("testUrl", "testPassword");
    }

    @Test
    public void testDefaultUrl() {
        assertEquals("testUrl",
valueAnnotationConstructorMock.getApiUrl());
    }

    @Test
    public void testDefaultPassword() {
        assertEquals("testPassword",
valueAnnotationConstructorMock.getApiPassword());
    }
}
```

This makes testing more straightforward because we're not dependent on Spring's initialization process. We can pass whatever values we need directly, such as *anyUrl* and *anyPassword*, to mock the *@Value* properties.

Now, moving on to some key advantages of using Constructor Injection:

- This approach is ideal for unit tests since it allows us to bypass the need for Spring's context. We can directly inject mock values in tests, making them faster and more isolated.
- It simplifies class construction and ensures that all required dependencies are injected at construction time, making the class easier to reason about.
- Using *final* fields with constructor injection ensures immutability, making the code safer and easier to debug.

Next, let's review some of the disadvantages of using Constructor Injection:

- We may need to modify our existing code to adopt constructor injection, which might not always be feasible, especially for legacy applications.
- We need to explicitly manage all dependencies in tests, which could result in verbose test setups if the class has many dependencies.

- If our tests need to handle dynamically changing values or a large number of properties, constructor injection can become cumbersome.

3. Conclusion

In this article, we saw that mocking *@Value* in Spring Boot tests is a common requirement to keep our unit tests focused, fast, and independent of external configuration files. By leveraging tools like *@TestPropertySource* and *ReflectionTestUtils*, we can efficiently mock configuration values and maintain clean, reliable tests.

With these strategies, we can confidently write unit tests that isolate the logic of our components without relying on external resources, ensuring better test performance and reliability.

The code backing this article is available on GitHub. Once you're **logged in as a Baeldung Pro Member (/members/)**, start learning and coding on the project.

You can also ask questions and leave feedback on the Azure Container Apps GitHub page (/microsoft-NPI-EA-4-9jJ51).

COURSES

ALL COURSES (/COURSES/ALL-COURSES)
BAELDUNG ALL ACCESS (/COURSES/ALL-ACCESS)
BAELDUNG ALL TEAM ACCESS (/COURSES/ALL-ACCESS-TEAM)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
LEARN SPRING BOOT SERIES (/SPRING-BOOT)
SPRING TUTORIAL (/SPRING-TUTORIAL)
GET STARTED WITH JAVA (/GET-STARTED-WITH-JAVA-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
REST WITH SPRING SERIES (/REST-WITH-SPRING-SERIES)
ALL ABOUT STRING IN JAVA (/JAVA-STRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS/)
PARTNER WITH BAELDUNG (/PARTNERS/WORK-WITH-US)
EBOOKS (/LIBRARY/)
FAQ (/LIBRARY/FAQ)
BAELDUNG PRO (/MEMBERS/)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)

PRIVACY MANAGER