Mockito 5

Spis treści

I co?

I Mockito! Znaczy Mockito 5! Mockito w wersji 5.X ukazało się w styczniu 2023.

Chcemy jednocześnie przyzwyczajać Cię, do tego, że niektóre zajavkowe materiały będą Ci udostępnione tylko w formie pisemnej i tak samo będzie w tym przypadku. Zmian dodanych w Mockito 5 (oczywiście z perspektywy naszej ścieżki), nie ma aż tak wiele, dlatego ten materiał będzie omówiony w postaci artykułu / notatki.

Zależność

Żeby skorzystać z Mockito 5, należy dodać do Maven taki wpis:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>5.0.0</version> ①
    <scope>test</scope>
</dependency>
```

① Szok i niedowierzanie...

Natomiast w przypadku Gradle, możemy dodać Mockito do projektu w taki sposób:

```
testImplementation 'org.mockito:mockito-junit-jupiter:5.0.0'
```

Dodam też, że w przyszłości jak dojdziemy do omówienia Spring Boot, to Mockito będzie dodawane w pewnym sensie automatycznie przez zależności "Springowe", ale przekonasz się o tym, jak będziemy już omawiać ten materiał.

Moduł mockito-inline

Najważniejsza zmiana z naszej perspektywy dotyczy wykorzystania modułu `mockito-inline`. Przypominasz sobie, do czego to służyło?

Do staticów! No właśnie, a Mockito od wersji 5.X pozwala na mockowanie konstruktorów, metod statycznych i klas `final` z pudełka (*out-of-the-box*). Nie musisz dodawać dodatkowych zależności.

Spójrzmy na poniższy przykład:

Klasa SomeFinalClass

```
package pl.zajavka;

public final class SomeFinalClass {

    public String someResult() {
        return "result";
    }
}
```

Klasa SomeFinalClassTest

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

import static org.mockito.Mockito.when;

public class SomeFinalClassTest {

    @Test
    void testMockito() {
        SomeFinalClass someFinalClassMock = Mockito.mock(SomeFinalClass.class);
        String answer = "Some mocked answer";
        when(someFinalClassMock.someResult()).thenReturn(answer);
        Assertions.assertEquals(answer, someFinalClassMock.someResult());
    }
}
```

Jeżeli uruchomisz teraz ten projekt, mając dodane do classpath takie zależności:

```
testImplementation "org.mockito:mockito-junit-jupiter:4.8.0"
testImplementation "org.mockito:mockito-inline:4.8.0"
```

Test zostanie uruchomiony i zakończony poprawnie.

Jeżeli natomiast nie dodasz do classpath zależności `mockito-inline`, na ekranie zobaczysz taki błąd:

```
Cannot mock/spy class pl.zajavka.SomeClass
Mockito cannot mock/spy because :
 - final class
org.mockito.exceptions.base.MockitoException:
Cannot mock/spy class pl.zajavka.SomeClass
Mockito cannot mock/spy because :
 - final class
```

Jeżeli teraz zmienisz zależności na classpath na takie:

```
testImplementation "org.mockito:mockito-junit-jupiter:5.0.0"
```

Test ponownie zakończy się powodzeniem. Wniosek? Mockito 5 pozwala na mockowanie klas `final`.

A co ze `static`? Spójrz na poniższy przykład:

Klasa StaticMethodExample

```
package pl.zajavka;

import java.time.LocalTime;

public class StaticMethodExample {

    public int getNano() {
        LocalTime now = LocalTime.now();
        return now.getNano();
    }
}
```

Klasa StaticMethodExampleTest

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.mockito.MockedStatic;
import org.mockito.Mockito;

import java.time.LocalTime;

class StaticMethodExampleTest {

    private StaticMethodExample staticMethodExampleSpy = new StaticMethodExample();

    @Test
    void testGetNanoNow() {
        // given
        LocalTime now = LocalTime.now();
        int nanoNow = now.getNano();
        LocalTime nowEarlier = now.minusNanos(100);
        int nanoEarlier = nowEarlier.getNano();

        int result;
        try(MockedStatic<LocalTime> timeMock = Mockito.mockStatic(LocalTime.class)) {
            timeMock.when(LocalTime::now).thenReturn(nowEarlier);
            // when
            result = staticMethodExampleSpy.getNano();
        }

        // then
        Assertions.assertNotEquals(result, nanoNow);
        Assertions.assertEquals(result, nanoEarlier);
    }
```

```
}
```

Jeżeli uruchomisz ten test z Mockito 5.X, test zakończy się powodzeniem. Natomiast z Mockito 4.X, test do pozytywnego zakończenia potrzebuje zależności `mockito-inline`.

ArgumentMatcher

Kolejna zmiana dotyczy interfejsu `ArgumentMatcher` i metody `any()` w odniesieniu do **varargs**. Zacznijmy od poniższej klasy:

```
package pl.zajavka;

public class SomeVarArgsExample {

    public int count(String... elements) {
        return elements.length;
    }
}
```

Przed Mockito 5, możliwe było napisanie takich stubbingów:

```
package pl.zajavka;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

public class SomeVarArgsExampleTest {

    @Test
    void testMockito() {
        SomeVarArgsExample exampleMock = mock(SomeVarArgsExample.class);

        when(exampleMock.count()).thenReturn(0);
        when(exampleMock.count(any(), any())).thenReturn(2);
        when(exampleMock.count(any(), any(), any())).thenReturn(3);

        assertEquals(0, exampleMock.count());
        assertEquals(2, exampleMock.count("a", "b"));
        assertEquals(3, exampleMock.count("a", "b", "c"));
    }
}
```

Natomiast jeżeli napisalibyśmy taki kod:

```
package pl.zajavka;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
```

```
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

public class SomeVarArgsExampleTest {

    @Test
    void testMockito() {
        SomeVarArgsExample exampleMock = mock(SomeVarArgsExample.class);

        when(exampleMock.count(any())).thenReturn(3);

        assertEquals(3, exampleMock.count());
        assertEquals(3, exampleMock.count("a"));
        assertEquals(3, exampleMock.count("a", "b"));
        assertEquals(3, exampleMock.count("a", "b", "c"));
    }
}
```

To zdefiniowanie stubbingu przy wykorzystaniu jednego argumentu `any()` pozwalało na zwrócenie wyniku dla dowolnej liczby argumentów, co z resztą jest widoczne w powyższym przykładzie.

Zabawnie zaczynało robić się, gdy napisaliśmy taki kod:

```
public class SomeVarArgsExampleTest {

    @Test
    void testMockito() {
        SomeVarArgsExample exampleMock = mock(SomeVarArgsExample.class);

        when(exampleMock.count()).thenReturn(0);
        when(exampleMock.count(any())).thenReturn(1);
        when(exampleMock.count(any(), any())).thenReturn(2);
        when(exampleMock.count(any(), any(), any())).thenReturn(3);

        assertEquals(0, exampleMock.count()); ①
        assertEquals(1, exampleMock.count("a"));
        assertEquals(2, exampleMock.count("a", "b"));
        assertEquals(3, exampleMock.count("a", "b", "c"));
    }
}
```

① Zgodnie ze stubbingami, w tej linijce powinniśmy dostać wartość `0`, natomiast test kończył się takim wynikiem: *expected: <0> but was: <1>*.

Co się zmieniło w Mockito 5? Powyższy test zaczyna działać poprawnie. Czyli możliwe jest określenie stubinngu dla *varargs*, przy przekazaniu tylko jednego argumentu wywołania metody. Tylko że teraz (Mockito 5) taki zapis zakończy się niepowodzeniem:

```
public class SomeVarArgsExampleTest {

    @Test
    void testMockito() {
        SomeVarArgsExample exampleMock = mock(SomeVarArgsExample.class);

        when(exampleMock.count(any())).thenReturn(1);
```

```
        assertEquals(1, exampleMock.count()); ①
        assertEquals(1, exampleMock.count("a"));
        assertEquals(1, exampleMock.count("a", "b"));
        assertEquals(1, exampleMock.count("a", "b", "c"));
    }
}
```

① Na ekranie będzie wydrukowane: *expected: <1> but was: <0>*.

Jeżeli natomiast chcemy, żeby powyższy przypadek znowu działał tak jak w przypadku Mockito 4.X, musimy napisać kod w taki sposób:

```
public class SomeVarArgsExampleTest {

    @Test
    void testMockito() {
        SomeVarArgsExample exampleMock = mock(SomeVarArgsExample.class);

        when(exampleMock.count(any(String[].class))).thenReturn(1); ①

        assertEquals(1, exampleMock.count());
        assertEquals(1, exampleMock.count("a"));
        assertEquals(1, exampleMock.count("a", "b"));
        assertEquals(1, exampleMock.count("a", "b", "c"));
    }
}
```

① Zmiana nastąpiła w definicji metody `any()`.

Podsumowanie

Opisanych zmian nie jest dużo, ale takie właśnie smaczki często zmieniają się przy wdrażaniu kolejnych wersji jakiejś biblioteki, czy frameworka. Oczywiście oprócz tego, co zostało tutaj omówione, twórcy zmieniają też często inne części biblioteki, natomiast my omawiamy tylko te zagadnienia, które są istotne z punktu widzenia ścieżki Zajavka.