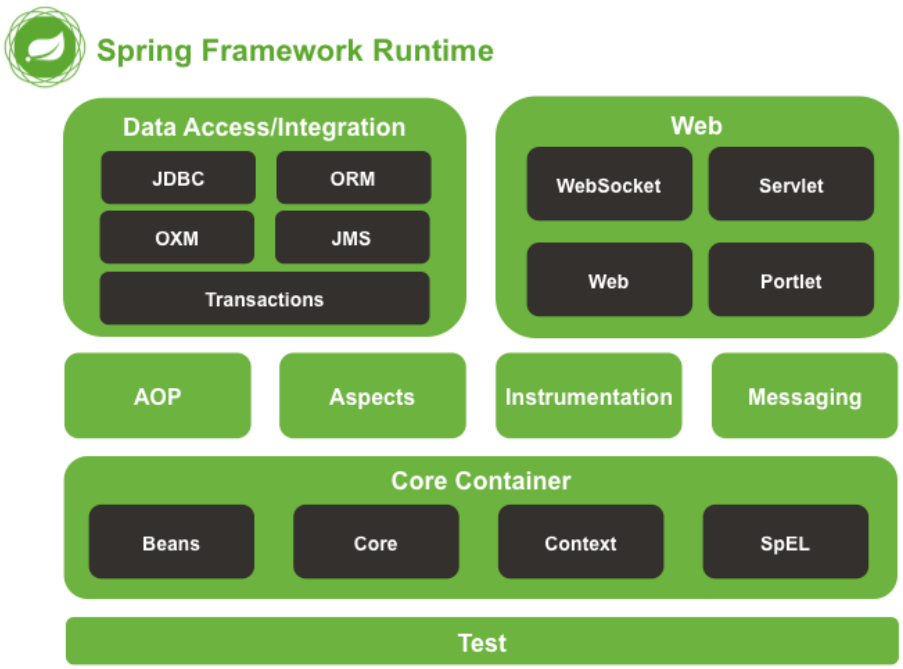
Spring with Mockito

Spis treści

Testowanie aplikacji Springowych

Spring Framework dostarcza wsparcie dla testów jednostkowych, ale także testów integracyjnych. Moduł, który jest za to odpowiedzialny to Spring TestContext Framework (spring-test). Działa on niezależnie od zastosowanego frameworka testowego, dlatego może być używany razem z **TestNG**, czy z **JUnit**. Rozszerza on możliwości testów o funkcjonalności przydatne przy pracy z aplikacjami Springowymi.

Patrząc na ogólny zarys modułów Spring Framework, można zauważyć wydzielony moduł *Test*.



Obraz 1. Moduły wraz z modułem Test

To właśnie o nim będziemy rozmawiać ☺

Żeby móc wykorzystać wspomniane funkcjonalności w praktyce, musimy dodać do projektu odpowiednie zależności.

Zależność do biblioteki spring-test:

```
// https://mvnrepository.com/artifact/org.springframework/spring-test
testImplementation "org.springframework:spring-test:$springVersion"
```

Żeby w teście JUnit5 mieć dostęp do modułu *Spring TestContext Framework*, to podobnie, jak w przypadku Mockito, klasę testową trzeba oznaczyć adnotacją `@ExtendWith`, ale tym razem ze wskazaniem na klasę `SpringExtension.class`, która integruje *Spring TestContext Framework* z JUnit5.

Adnotacja @ExtendWith:

```
import org.junit.jupiter.api.extension.ExtendWith; ①
import org.springframework.test.context.junit.jupiter.SpringExtension; ②

@ExtendWith(SpringExtension.class)
```

① Gradle: org.junit.jupiter:junit-jupiter-api

② Gradle: org.springframework:spring-tes

Porównanie różnych rodzajów adnotacji

"Samo" Mockito

Dotychczas pracowaliśmy w testach z adnotacją `@ExtendWith(MockitoExtension.class)`. Zwróć uwagę, że adnotacja ta była przez nas używana zanim powiedzieliśmy cokolwiek na temat Springa w testach. Wynika to z tego, że korzystając z tej adnotacji wykorzystujemy Javę, Mockito oraz JUnit5 i zapominamy o Springu. Nie tworzymy wtedy Spring contextu, nie korzystamy ze Springa. Możemy wtedy wykorzystywać adnotacje `@Mock` i `@InjectMocks` i pracujemy wtedy w zestawieniu Java + JUnit5 + Mockito. Dlatego można ten przypadek nazwać "samo Mockito", w domyśle mamy tutaj właśnie to zestawienie Java + JUnit5 + Mockito. Można ewentualnie nazwać taki przypadek: "czysty test Mockito". Jak zwał tak zwał, chodzi o to, że nie korzystamy tutaj ze Springa.

Gdy zaczniesz szukać przykładów w internecie, spotkasz również taki zapis:

```
@RunWith(MockitoJUnitRunner.class)
```

Jest to zapis, który był potrzebny do korzystania z Mockito w zestawieniu z JUnit4. My teraz używamy JUnit5, dlatego możemy traktować ten zapis jako *deprecated*. Teraz posługiwaliśmy się inną adnotacją, czyli:

```
@ExtendWith(MockitoExtension.class)
```

Wracając do Springa

Zwróć uwagę, że gdy zaczęliśmy rozmawiać o Springu, to zmieniliśmy argument wykorzystywany w adnotacji `@ExtendWith`. Teraz zamiast zapisu:

```
@ExtendWith(MockitoExtension.class)
```

będziemy stosowali zapis:

```
@ExtendWith(SpringExtension.class)
```

Gdy zastosujemy tę adnotację, będzie to oznaczało, że test ma stworzyć cały kontekst Springa, razem ze wstrzyknięciem zależności Springowych itp. Będziemy wtedy musieli określić konfigurację Beanów dla testów. Korzystamy wtedy z funkcjonalności, która są nam zapewniane przez `spring-test`.

W internecie możesz spotkać przykłady z zapisem:

```
@RunWith(SpringJUnit4ClassRunner.class)
```

Jest to zapis, który był potrzebny przy korzystaniu z JUnit4, natomiast my omawiamy materiał w oparciu o JUnit5. Oznacza to, że adnotacja *@RunWith(SpringJUnit4ClassRunner.class)* może być traktowana przez nas jako *@Deprecated* i zamiast niej używamy teraz *@ExtendWith(SpringExtension.class)*.

Testy jednostkowe

Prawdopodobne jest, że jeżeli będziesz pisać test jednostkowy (a nie integracyjny), to nie będziesz potrzebować korzystać ze Spring contextu. Najczęściej w takim przypadku zupełnie wystarczające będzie wykorzystanie zestawienia Java + JUnit5 + Mockito. Spring context byłby nam potrzebny gdybyśmy potrzebowali mieć skonfigurowane zależności między Beanami, ale mówimy o teście jednostkowym - testujemy jednostkę. W takim przypadku najczęściej potrzebujemy "zaślepić" zależności jakie ma klasa i w tym celu zastosowalibyśmy wspomniane zestawienie Java + JUnit5 + Mockito. Najczęściej Spring context nie jest nam w takim przypadku potrzebny.

Mogą natomiast wystąpić takie sytuacje, że pisząc testy jednostkowe, będziemy mieli potrzebę skorzystania z *Spring TestContext Framework* (będziemy wracać do takich przypadków w kolejnych częściach kursu). Warto jednak zaznaczyć, że *Spring TestContext Framework* wspiera testy jednostkowe o dodatkowe opcje mockowania. Znajdują się one w pakiecie *org.springframework.mock* i są podzielone na części dotyczące:

- **Environment** - mockownia obiektów *org.springframework.core.env.Environment* i *org.springframework.core.env.PropertySource* - byłoby to potrzebne przy pracy np. ze zmiennymi środowiskowymi,
- **Servlet API** - wsparcia mockowania *Servlet API*, co jest pomocne przy testach *Spring Web MVC* framework - o tym będziemy rozmawiać później. Link do strony z dokumentacją modułu Spring Web MVC,
- **Spring Web Reactive** - wsparcia mockowania requestów i responsów http przy stosowaniu

podejścia rekatywnego, co jest pomocne przy testach *Spring WebFlux* - to już jest wyższa szkoła jazdy i nie planujemy poruszać tej tematyki w obrębie ścieżki Zajavkowej. Link do strony z dokumentacją modułu Spring WebFlux.

Jeżeli zastanawiasz się po co podajemy tyle informacji jednocześnie, już spieszymy z odpowiedzią. Projekt Spring jest ogromny, będziemy uczyć się tylko jego wycinka, gdyż większą część i tak będziesz poznawać w praktyce. Chcemy Ci jednak dać pewnego rodzaju obraz tego jak duży jest to projekt, stąd pokazujemy pewne wycinki. Nie oznacza to jednak, że masz teraz uczyć się teraz na pamięć wszystkiego o czym wpsominamy. Na wszystko przyjdzie czas i pora. My przechodzimy przez świat Springa stopniowo.

Testy integracyjne

Główną zaletą *Spring TestContext Framework* jest wsparcie testów integracyjnych aplikacji Springowej. Dzięki niemu można przetestować wstrzykiwanie zależności do kontekstu Springa, a także dostęp do danych poprzez np. JDBC. Czyli korzystając z *Spring TestContext Framework* uruchamiamy aplikację z poziomu testu i jednocześnie tworzymy Spring context. *Spring TestContext Framework* jest oparty o adnotacje, które będziemy poznawać stopniowo. Stosując adnotacje dedykowane do testów, nadal możemy korzystać z adnotacji Spring, które poznaliśmy już wcześniej.

Poniżej znajdziesz przykład, bazujący na wcześniejszych przykładach. Mamy dwa serwisy `ExampleBeanService` i `InjectedBeanService` oznaczone stereotypami `@Component` i `@Service` oraz klasę z konfiguracją `ConfigScanBean`.

Przykład kodu Springowego:

```
package pl.zajavka;

public interface ExampleBeanService {
    boolean sampleMethod();
}
```

```
package pl.zajavka;

import org.springframework.stereotype.Component;
import lombok.AllArgsConstructor;

@Component
@AllArgsConstructor
public class ExampleBeanServiceImpl implements ExampleBeanService {

    private InjectedBeanService injectedBeanService;

    @Override
    public boolean sampleMethod() {
        return injectedBeanService.anotherSampleMethod();
    }
}
```

```
package pl.zajavka;

public interface InjectedBeanService {
    boolean anotherSampleMethod();
}
```

```
package pl.zajavka;

import org.springframework.stereotype.Service;

@Service
public class InjectedBeanServiceImpl implements InjectedBeanService {
    @Override
    public boolean anotherSampleMethod() {
        System.out.println("InjectedBeanServiceImpl#anotherSampleMethod");
        return false;
    }
}
```

```
package pl.zajavka;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("pl.zajavka")
public class ConfigScanBean {}
```

Zacznijmy od przykładu testu (przykład poniżej), który korzysta jedynie z Junit5 (nie korzystamy ze `spring-test`). Metoda testowa `testSampleMethod()` wywołuje metodę `exampleBeanService.sampleMethod()`, która to wywołuje metodę `anotherSampleMethod()` ze wstrzykniętego serwisu `InjectedBeanService`. W teście tworzymy Spring context manualnie, a następnie test sprawdza, czy kontekst Springa został prawidłowo przygotowany i czy zależności są wstrzykiwane poprawnie. Nie ma tutaj żadnego mockowania, Spring context łączy ze sobą rzeczywiste Spring komponenty. Dlatego oczekiwanym wynikiem testu jest wartość `false`, czyli to, co zwraca implementacja metody `anotherSampleMethod()`.

Test bez użycia Spring TestContext Framework:

```
package pl.zajavka;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import pl.zajavka.ConfigScanBean;
import pl.zajavka.ExampleBeanService;

class NoSpringExampleTest {

    private ExampleBeanService exampleBeanService;
```

```
    @BeforeEach ①
    public void setUp() {
        ApplicationContext applicationContext =
                new AnnotationConfigApplicationContext(ConfigScanBean.class);
        exampleBeanService = applicationContext.getBean(ExampleBeanService.class);

        Assertions.assertNotNull(exampleBeanService);
    }

    @AfterEach ②
    public void tearDown() {
        exampleBeanService = null;

        Assertions.assertNull(exampleBeanService);
    }

    @Test ③
    void testSampleMethod() {
        boolean result = exampleBeanService.sampleMethod();
        Assertions.assertFalse(result);
    }
}
```

① Przed każdym testem testowany serwis jest tworzony na podstawie nowo utworzonego kontekstu Springa,

② Po każdym teście instancja testowanego serwisu jest nullowana,

③ Test kończy się sukcesem, bo metoda `anotherSampleMethod()` zwraca wartość `false`.

W przykładzie powyżej utworzyliśmy Spring context ręcznie. Teraz kolej na test napisany z użyciem *Spring TestContext Framework*. Metoda sprawdza to samo co poprzednio, ale tym razem kontekst Springa tworzony jest przez *spring-test*, a testowany serwis `ExampleBeanService` jest wstrzykiwany z użyciem adnotacji `@Autowired`.

Test z użyciem Spring TestContext Framework:

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import pl.zajavka.ExampleBeanService;

@ExtendWith(SpringExtension.class) ①
@ContextConfiguration(classes = { TestConfigScanBean.class }) ②
class SpringExampleTest {

    @Autowired ③
    private ExampleBeanService exampleBeanService;

    @BeforeEach ④
    public void setUp() {
        Assertions.assertNotNull(exampleBeanService);
```

```
    }

    @Test ⑤
    void testSampleMethod() {
        boolean result = exampleBeanService.sampleMethod();
        Assertions.assertFalse(result);
    }
}
```

① Integrujemy Springa z JUnit5,

② Wskazujemy miejsce definicji Beanów, w tym przypadku mamy klasę konfiguracyjną skanującą beany,

③ Wstrzykujemy serwis to przetestowania,

④ Sprawdzenie, czy bean jest gotowy do użycia,

⑤ Test kończy się sukcesem, bo metoda `anotherSampleMethod()` zwraca wartość `false`.

```
@Configuration
@ComponentScan("pl.zajavka")
public class TestConfigScanBean {}
```

Adnotacja `@ContextConfiguration` informuje, gdzie znajduje się konfiguracja kontekstu.

Przykład użycia adnotacji @ContextConfiguration:

```
import org.springframework.test.context.ContextConfiguration;

@ContextConfiguration(classes = { TestConfigScanBean.class }) ①
@ContextConfiguration(locations = {"SpringExampleTest-config.xml"}) ②
```

Konfiguracja Beanów może być pobrana z:

① Klasy konfiguracyjnej `TestConfigScanBean.class`,

② Pliku konfiguracyjnego *SpringExampleTest-config.xml*.

Konfiguracja klasy testowej może zostać jeszcze bardziej uproszczona poprzez użycie adnotacji `@SpringJUnitConfig`, która łączy w sobie obie `@ExtendWith(SpringExtension.class)` i `@ContextConfiguration`. Dlatego można je zastąpić jak poniżej:

Przykład użycia adnotacji `@SpringJUnitConfig`:

```
import org.springframework.test.context.junit.jupiter.SpringJUnitConfig;

@SpringJUnitConfig(classes = {TestConfigScanBean.class})
```

Powyższy przykład pokazał nam jak stworzyć Spring context w testach i przetestować "całą" aplikację, razem ze wszystkimi zdefiniowanymi zależnościami.

Co natomiast zrobić w przypadku, gdzie będziemy potrzebowali napisać test integracyjny, oparty o Spring context, ale będziemy potrzebowali zamockować część zależności. Czyli innymi słowy,

potrzebujemy utworzyć Spring context razem ze wszystkimi powiązaniami pomiędzy Beanami, ale tylko niektóre z Beanów mają być zamockowane. Już do tego przechodzimy.

Mockownaie

Przejdźmy zatem do przykładu, gdzie tworzymy cały Spring context, ale mamy potrzebę, żeby kilka zależności było mockami, a pozostałe mają być rzeczywistymi zależnościami . Poniżej znajdziesz źródła z katalogu `src/main/java`.

```
package pl.zajavka;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackageClasses = Marker.class)
public class BeanConfiguration {}
```

```
package pl.zajavka;

import java.math.BigDecimal;

public interface CapacityCalculationService {
    BigDecimal someCalculation(final InputData inputData);
}
```

```
package pl.zajavka;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;

@Service
@RequiredArgsConstructor
public class CapacityCalculationServiceImpl implements CapacityCalculationService {

    private final WidthCalculationService widthCalculationService;

    private final HeightCalculationService heightCalculationService;

    private final DepthCalculationService depthCalculationService;

    @Override
    public BigDecimal someCalculation(final InputData inputData) {
        BigDecimal height = heightCalculationService.calculate(inputData);
        BigDecimal width = widthCalculationService.calculate(inputData);
        BigDecimal depth = depthCalculationService.calculate(inputData);
        return height.multiply(width).multiply(depth);
    }
}
```

```
package pl.zajavka;
```

```
import java.math.BigDecimal;

public interface DepthCalculationService {

    BigDecimal calculate(InputData inputData);
}
```

```
package pl.zajavka;

import org.springframework.stereotype.Service;

import java.math.BigDecimal;

@Service
public class DepthCalculationServiceImpl implements DepthCalculationService {

    @Override
    public BigDecimal calculate(final InputData inputData) {
        return new BigDecimal(inputData.getDepth());
    }
}
```

```
package pl.zajavka;

import java.math.BigDecimal;

public interface HeightCalculationService {

    BigDecimal calculate(InputData inputData);

}
```

```
package pl.zajavka;

import org.springframework.stereotype.Service;

import java.math.BigDecimal;

@Service
public class HeightCalculationServiceImpl implements HeightCalculationService {

    @Override
    public BigDecimal calculate(final InputData inputData) {
        return new BigDecimal(inputData.getHeight());
    }
}
```

```
package pl.zajavka;

import lombok.Builder;
import lombok.Value;

import java.math.BigDecimal;
```

```
@Value
@Builder
public class InputData {
    String width;
    String height;
    String depth;
}
```

```
package pl.zajavka;

public interface Marker {}
```

```
package pl.zajavka;

import java.math.BigDecimal;

public interface WidthCalculationService {

    BigDecimal calculate(InputData inputData);

}
```

```
package pl.zajavka;

import org.springframework.stereotype.Service;

import java.math.BigDecimal;

@Service
public class WidthCalculationServiceImpl implements WidthCalculationService {

    @Override
    public BigDecimal calculate(final InputData inputData) {
        return new BigDecimal(inputData.getWidth());
    }
}
```

Dodajmy do tego kod źródłowy z katalogu `src/test/java`:

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import java.math.BigDecimal;

@ExtendWith(SpringExtension.class)
```

```
@ContextConfiguration(classes = { TestBeanConfiguration.class }) ①
public class CapacityCalculationServiceTest {

    @Autowired
    private CapacityCalculationService capacityCalculationService;

    @Autowired
    private WidthCalculationService widthCalculationService;

    @Autowired
    private HeightCalculationService heightCalculationService;

    @Autowired
    private DepthCalculationService depthCalculationService;

    @BeforeEach
    public void setUp() {
        Assertions.assertNotNull(capacityCalculationService);
        Assertions.assertNotNull(widthCalculationService);
        Assertions.assertNotNull(depthCalculationService);
        Assertions.assertNotNull(heightCalculationService);
    }

    @Test
    void testSampleMethod() {
        // given
        final var inputData = someInputData();
        Mockito.when(widthCalculationService.calculate(Mockito.any(InputData.class)))
            .thenReturn(BigDecimal.TEN); ②
        Mockito.when(depthCalculationService.calculate(Mockito.any(InputData.class)))
            .thenReturn(new BigDecimal("20")); ②

        // when
        final var result = capacityCalculationService.someCalculation(inputData);

        // then
        Assertions.assertEquals(new BigDecimal("600"), result);
    }

    private InputData someInputData() {
        return InputData.builder()
            .depth("1")
            .width("2")
            .height("3")
            .build();
    }
}
```

① Tutaj zaczyna się cała zabawa. W testach możemy wykorzystać konfigurację zdefiniowaną w klasie `BeanConfiguration`. Przy czym ta klasa nie ma dodanej żadnej konfiguracji mocków. Czyli jeżeli zastosowalibyśmy w tym miejscu konfigurację `BeanConfiguration` oraz pozbyli się konfiguracji mocków oznaczonych w kodzie cyferką 2, to test podniósłby Spring context i wykonał się na rzeczywistych zależnościach. Jeżeli natomiast wykorzystamy konfigurację `TestBeanConfiguration`, to będziemy mogli skorzystać z konfiguracji mocków zdefiniowanych przy cyferce 2. Definicja klasy `TestBeanConfiguration` jest poniżej.

② W tym miejscu mamy zdefiniowane zachowania dla mocków.

```java
package pl.zajavka;

import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.MockitoAnnotations;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackageClasses = Marker.class)
public class TestBeanConfiguration {

    @Mock
    private DepthCalculationService depthCalculationService;

    public TestBeanConfiguration() { ①
        try (AutoCloseable autoCloseable = MockitoAnnotations.openMocks(this)) {
            System.out.println("Correctly opened mocks");
        } catch (Exception e) {
            System.out.println("Unable to open mocks: " + e);
            throw new RuntimeException(e);
        }
    }

    @Bean
    public DepthCalculationService depthCalculationService() {
        return depthCalculationService; ②
    }

    @Bean
    public WidthCalculationService widthCalculationService() {
        return Mockito.mock(WidthCalculationService.class); ③
    }
}
```

W testowych plikach źródłowych (`src/test/java`) została dodana klasa `TestBeanConfiguration`, która jest wykorzystana w teście. Klasa ta nie może być wykorzystana do konfiguracji Spring context w kodzie źródłowym, klasa ta jest stworzona specjalnie na potrzeby testów. Zdefiniowane są w niej 2 Beany, które mają być mockami. Zostało to zrobione na dwa sposoby. Sposób pierwszy jest związany z cyferkami 1 i 2. Sposób drugi to cyferka 3.

① W konstruktorze musimy wywołać metodę `MockitoAnnotations.openMocks(this)`, żeby móc skorzystać z adnotacji `@Mock`, która została zapisana wyżej.

② Tutaj możemy wykorzystać pole, do którego zostanie przypisany mock, który będzie zainicjowany dzięki adnotacji `@Mock`.

③ Tutaj mamy drugi sposób stworzenia mocka, wykorzystujemy metodę `Mockito.mock()`.

Wykorzystując przedstawioną konfigurację w testach integracyjnych, możemy korzystać jednocześnie z prawdziwych zależności jak i z tych zamockowanych.

Zawsze warto jest się zastanowić czy lepiej jest napisać test oparty o Spring context czy nie. Wspominam o tym dlatego, że często testy oparte o Spring context będą się wykonywały dłużej niż "czyste testy w Mockito". Wynika to z czasu jaki jest potrzebny,

żeby utworzyć cały Spring context. Przypomnijmy sobie piramidę testów i to, że testy jednostkowe były u podstawy piramidy, a testy integracyjne były nad nimi. Testy integracyjne często trwają o wiele dłużej niż testy jednostkowe, czego przykładem jest test integracyjny oparty o Spring context.

Jeżeli zaczniesz wyszukiwać w internecie przykładów testów napisanych przy wykorzystaniu Spring i/lub mockito, spotkasz się ze stwierdzeniem **spring-boot**. Jest to jeden z projektów Springowych, który ma upraszczać konfigurację Springa. Poświęcimy na Spring Boot oddzielny warsztat.

Podsumowanie

Testy to obszerny temat, do którego będziemy jeszcze wracać. To, co jest ważne do zapamiętania z tego warsztatu to przede wszystkim zagadnienia dotyczące mockowania i Mockito. To dzięki nim możemy, w wygodny sposób, zapewnić izolacje testu. Na pierwszy rzut oka pisanie testów może wydawać się zbędne, ale docenia się je z czasem, kiedy uratują nas przed błędami, które niewykryte mogłyby wybuchnąć w najmniej oczekiwanym momencie. Podejście do tworzenia aplikacji oparte na pisaniu testów wymusza dodatkowo poprawną architekturę. Na przykład, w przypadku starych systemów, jeżeli czegoś nie da się przetestować, to najprawdopodobniej jest to źle zaprojektowane. Do tematyki testów będziemy wracać cały czas, gdyż jest to nieodzowny element wytwarzania oprogramowania.