Programowanie aplikacji w Java

Maciej Gowin

Zjazd 10 - dzień 2

Mockowanie

Uruchomienie aplikacji wiąże się z dostarczeniem wszystkich wymaganych zależności do klasy poddawanej testowi.

Zamiast używać rzeczywistych implementacji obiektów (np. repozytorium), można zastąpić je obiektami imitującymi ich działanie.

Taki sposób ułatwia pisanie testów oraz umożliwia skupienie się na testowaniu funkcjonalności danej klasy.

Mockowanie

Rodzaje mockowania:

- **Dummy** to obiekt w teście, który jest nam potrzebny jako wypełnienie. Najczęściej w formie pustej klasy.
- **Stub** to obiekt mający minimalną implementację interfejsu, bez skomplikowanej logiki.
- **Mock** to obiekt, któremu wskazujemy dokładne zachowania dla określonych metod. Najlepiej skorzystać już z dostępnych wchodzących w skład bibliotek (Mockito)

Mockito

Mockito - biblioteka dostarczająca mechanizmy służące do mockowania obiektów i definiowania ich zachowania.

Świetnie współgrająca z JUnit.

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>4.5.1</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>4.5.1</version>
    <scope>test</scope>
</dependency>
```

Zależności są już dostarczone wraz z paczkami Spring Boot

Mockowanie zależności z Mockito

```
@ExtendWith(MockitoExtension.class)
class ExampleServiceTest {
    @InjectMocks
    private ExampleService exampleService;
    @Mock
    private DummyRepository dummyRepository;
    @Test
    void shouldCallSaveBook() {
        // given
        when(dummyRepository.save(any(Book.class))).thenReturn(4);
        // when
        int result = exampleService.saveRandomBook();
        // then
        assertThat(result).isEqualTo(4);
        verify(dummyRepository, times(1)).save(any(Book.class));
    }
}
```

Mockowanie zależności z Mockito

Zalety użycia Mockito:

- Chcemy napisać testy sprawdzający działanie pojedynczej metody, nie zależy nam na stawianiu całego kontekstu Springa
- Testy wykonują się bardzo szybko
- Jesteśmy w stanie sprawdzić wykonanie kodu linijka po linijce, śledząc stan obiektów na każdym kroku
- Można zamockować wszystko lub jedynie pojedyncze beany (np. repozytoria, co eliminuje potrzebę stawiania bazy danych)

Mockowanie zależności z Mockito

Wady użycia Mockito:

- Test będzie tak dobry, jak dobre będą mocki (jeśli będziemy zwracać nierealne dane to taki test nie będzie pomocny)
- Zielone testy z użyciem Mockito nie gwarantują, że kontekst Springa wstanie (nie sprawdzają poprawności konfiguracji)

Mockowanie zależności z Mockito

Mockowanie polega na ustaleniu odpowiedzi metody danej klasy w formie:

JEŚLI zostanie wykonana metoda **X**, **WTEDY** zwróć **Y**

```
when(object.method()).thenReturn(objectOrValue);
```

Podobna składnia jest w przypadku chęci zwrócenia wyjątku:

```
when(object.method()).thenThrow(ex);
```

Weryfikacja działania metody z Mockito

Oprócz mockowania odpowiedzi method istnieje możliwość weryfikacji czy dana metoda faktycznie została wykonana oraz ile razy.

Przykład (sprawdzenie, czy metoda `dummyRepository.save()` została wykonana 1 raz):

```
verify(dummyRepository, times(1)).save(any(Book.class));
```

Weryfikacja działania metody z Mockito

Możemy również "złapać" dowolny obiekt przekazywany jako parametr do metody i sprawdzić, czy jest poprawnie ustawiony.
Ten mechanizm umożliwia dotarcie do konkretnej linijki metody, gdzie "coś się psuje".

Przykład:

```
@Test
void shouldCallSaveBook() {
    // given
    when(dummyRepository.save(any(Book.class))).thenReturn(4);
    // when
    int result = exampleService.saveRandomBook();
    // then
    ArgumentCaptor<Book> argumentCaptor = ArgumentCaptor.forClass(Book.class);
    verify(dummyRepository, times(1)).save(argumentCaptor.capture());
    Book book = argumentCaptor.getValue();
    assertThat(book).isNotNull();
}
```

Testy integracyjne

Testy integracyjne służą sprawdzeniu połączeń i interakcji między poszczególnymi modułami lub innymi systemami.

W przypadku testów jednostkowych możemy sprawdzić, że:

- zapis do bazy działa poprawnie
- metoda serwisu poprawnie przepisuje obiekty i wykonuje skomplikowaną logikę

Czego nie wiemy?

- Czy aplikacja w ogóle się włączy?
- Czy request HTTP zakończy się poprawnym zapisem danych do bazy danych?

Na tego typu pytania odpowiadają testy integracyjne. Taki test polega na **uruchomieniu** aplikacji oraz wykonaniu na niej testu (np. wykonanie zapytania HTTP).

Testy Spring

W przypadku użycia Springa jest możliwość uruchomienia całej aplikacji podczas wykonywanie testu, włącznie z podłączeniem jej do bazy danych.

W zależności co chcemy przetestować, możemy uruchomić całość bądź skupić się jedynie na części aplikacji (np. tylko test repozytorium).

- `@SpringBootTest`
  - uruchamia cały kontekst Spring
- `@WebMvcTest`
  - używany do testowania pojedynczej klasy @Controller
  - nie przeskanuje beanów takich jak `@Service`, `@Repository` (należy pamiętać, aby dostarczyć odpowiednie zależności)

Testy Spring

- `@DataJpaTest`
  - używany do testowania warstwy persistence (repozytoria, dao)
  - uruchomi jedynie część aplikacji w warstwie persistence (JPA)
  - uruchomi testową bazę in-memory H2 (należy dostarczyć zależność w `pom.xml`)
  - przestawi aplikację w tryb logowania SQL na konsoli

Testy Spring

Jednym z przykładów testu integracyjnego jest wykonanie zapytania HTTP do jednego z endpointów naszej aplikacji oraz sprawdzenie, czy wynik jest zgodny z oczekiwanym.

Istnieje mechanizm umożliwiający w prosty sposób wykonanie zapytania HTTP w teście - `MockMvc`. Aby skonfigurować `MockMvc` można użyć adnotacji `@AutoConfigureMockMvc`, która utworzy beana typu `MockMvc`.

Testy Spring

Przykład testu:

```
@SpringBootTest
@AutoConfigureMockMvc
class ExampleControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @Test
    void shouldReturnString() {  // wykonanie HTTP GET /example
        // when & then
        MvcResult result = mockMvc.perform(get("/example"))
                .andExpect(status().isOk())
                .andExpect(content().string("example String"))
                .andDo(print()).andReturn();
    }
}
```

Testy Spring

Testy integracyjne mogą korzystać z:

- rzeczywistych implementacji
- mocków

Aby zamockować bean należy użyć adnotacji `@MockBean`, dla przykładu:

```
@SpringBootTest
@AutoConfigureMockMvc
class ExampleControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private ExampleService exampleService;  // serwis jest zamockowany
    @Test
    void shouldReturnString() {  // wykonanie HTTP GET /example
        // given
        when(exampleService.returnHello()).thenReturn("Siema");  // definicja zachowania metody
        // when & then
        MvcResult result = mockMvc.perform(get("/example"))
                .andExpect(status().isOk())
                .andExpect(content().string("example String"))
                .andDo(print()).andReturn();
    }
}
```

Testy Spring - zależności Maven

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <version>2.5.0</version>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

Warto zwrócić uwagę na *scope* zależności

Test Driven Development

W klasycznym podejściu rozwoju oprogramowania właściwy kod programu powstaje przed powstaniem testów, w tym testów jednostkowych.

Możliwe jest odwrócenie tego procesu, a podejście to jest opisywane jako Test Driven Development.

Test Driven Development

Test Driven Development (ang. TDD) to podejście do tworzenia oprogramowanie, w którym głównym nacisk kładzie się na rozwijanie testów.

Opiera się ono na tworzeniu testów przed stworzeniem właściwego kodu programu.

Test Driven Development

Cykl tworzenia opiera się na kilku krokach:

- dodaj test,
- uruchom test, który powinien zwrócić błąd,
- zaimplementuj funkcjonalność w najprostszy możliwy sposób,
- wszystkie testy powinny zakończyć się sukcesem,
- zrefaktoruj kod w razie potrzeby, upewniając się, że wszystkie testy zawracają sukces.

Test Driven Development

Oczywiście podejście TDD może być używane nie tylko przy okazji pisania testów jednostkowych, ale też testów integracyjnych.

Definiuje ono jedynie model pracy i tworzenia oprogramowania.

Behavioral Driven Development

BDD (behavior-driven development) jest rozszerzeniem podejścia TDD. Podobnie jak w przypadku TDD rozwój oprogramowania opiera się na początkowym tworzeniu testów. Główną różnicą jest samo podejście do opisu testów.

W podejściu BDD testy opisują zachowanie i oczekiwane efekty tego zachowania - w przeciwieństwie do prostych testów jednostkowych, które opisuje mały wycinek funkcjonalności, która jest przedmiotem testów.

Behavioral Driven Development

Istnieje szereg narzędzi, które pozwalają na opisywanie zachowań oraz testów przy pomocy podejścia BDD w sposób zrozumiały dla użytkowników nietechnicznych takich jak analitycy biznesu czy testerzy manualni.

Dzięki temu możemy oddzielić definicję zachowań aplikacji od samej implementacji testów oraz umożliwić definicję testów behawioralnych nie tylko na poziomie kodu.

Cucumber

Narzędzie `Cucumber` jest narzędziem wspierającym rozwój oprogramowania na podstawie koncepcji BDD. Testy definiowane są za pomocą specjalnego języka `Gherkin`.

Gherkin pozwala na definiowanie zachowań oprogramowania w sposób zrozumiały nie tylko dla programistów, ale też użytkowników. Dzięki temu opis testów może posłużyć jako definicja funkcjonalności danego oprogramowania.

W wielu przypadkach Cucumber jest używany wraz z innymi narzędziami wspierającymi testowania oprogramowania. Dodatkowo wspiera wiele języków programowania w połączeniu, z którymi może zostać użyty.

Cucumber - Gherkin

Testy definiujemy w plikach `.feature`, które opisują daną funkcjonalność. Format pliku oraz słowa kluczowe są definiowane przez język Gherkin.

```
Feature: bank account operations

    Back account operations

    Scenario: Customer wants to add money to his bank account
        Given account balance is 100
        When customer adds 50
        Then account balance is 150

    Scenario: Customer wants to withdraw money from his bank account
        ...
```

Cucumber - Gherkin

Do słów kluczowych zaliczamy:

`Feature` - każda z funkcjonalności może zawierać jeden lub więcej scenariuszy oraz opisuje daną funkcjonalność oprogramowania.
`Scenario` - opisuje konkretny test, który zostanie wykonany. Scenariusze składają się z kroków, które sprawdzają dane zdarzenie.

Oraz kroki (ang. steps):

`Given` - opisuje warunki początkowe oraz stan przed uruchomieniem testów
`When` - opisuje akcje wykonane przez użytkownika podczas testu
`Then` - opisuje rezultaty testów

Cucumber - Gherkin

Kroki budujące scenariusze mogą być rozumiane jako wywołania funkcji. Aby mechanizm Cucumber mógł wykonać dany scenariusz, należy zdefiniować sposób, w jaki krok powinien zostać przetworzony.

Definicja kroków odbywa się na poziomie kodu źródłowego testów.

Cucumber - Gherkin

```
    @Given("account balance is set to {value}")
    public void account_balance_is_set_to(double value) {
        /* ... */
    }

    @When("customer adds {value}")
    public void customer_adds(double value) {
        /* ... */
    }

    @Then("account balance is {value}")
    public void account_balance_is(double value) {
        /* ... */
    }
```

Programowanie: przyklad-cucumber

Implementacja testów przy pomocy biblioteki Cucumber. Do wykonania testów użyj standardowego kroku:

```
mvn test
```

Testy manualne

Testowanie manualne opierają się na manualnym sprawdzeniu aplikacji.

Testy te najczęściej dotyczą stron internetowych oraz aplikacji mobilnych - czyli interfejsów użytkownika. Podczas ich wykonywania implicite testowane są też rozwiązania backendowe, które są konsumowane przez aplikacje frontendowe.

Błędy pojawiające się podczas testów manualnych mogą wynikać z błędów na poziomie FE oraz BE.

Testy manualne - problemy

Wyobraźmy sobie aplikacje do logowania użytkowników.

Przy dowolnej zmianie ekranu logowania lub też systemu wspierającego zapytania logowania wymagane jest przetestowanie aplikacji w celu sprawdzenia poprawności jej działania.

Krok ten powinien być wykonany przy każdej nowej wersji aplikacji. W przypadku testów manualnych musimy wykonać powtarzalną czynność, której kroki będą identyczne w każdej iteracji rozwoju oprogramowania.

Testy automatyczne

Aby rozwiązać problem powtarzalności manualnych testów, wprowadzamy testy automatyczne.

Testy automatyczne mają na celu zautomatyzowanie czynności, które musielibyśmy wykonać przy każdorazowej weryfikacji oprogramowania.

Testy te są oparte na skryptach, które imitują czynności wykonywane przez testera manualnego i mogą zostać uruchomione w dowolnym momencie.

Testy automatyczne

Aby zdefiniować test automatyczny, w większości przypadków definiujemy scenariusz testu, czyli kroki, które test powinien wykonać.

Skrypt definiujący kroki imituje zachowanie testera manualnego oraz stara się wykonać analogiczne czynności.

Uruchomienie skryptu równoznaczne jest z wykonaniem testu.

Testy automatyczne

Istnieje szereg systemów ułatwiających wykonywanie testów automatycznych.

Dla przykładu: do tej pory używaliśmy narzędzia Cucumber do testów jednostkowych aplikacji. Równie dobrze moglibyśmy użyć tego narzędzia do stworzenia scenariuszy testów automatycznych dla istniejącej aplikacji backend. Kroki odpowiadałyby wywołaniom API zdefiniowanego przez aplikację webową.

Testy automatyczne

Testowanie systemu backendowego takiego jak aplikacja backend wydaje się stosunkowo proste. Wywołania API mogą zostać wykonane z poziomu kodu programu.

Istnieje też możliwość automatycznego testowania aplikacji webowych, w tym stron internetowych. Testy te imitują zachowanie użytkownika przechodzącego po kolejnych stronach oraz wykonujących działania na stronie.

Testy automatyczne - Selenium

Przykładem narzędzia, które wspiera automatyczne testowanie aplikacji webowych - stron internetowych jest narządzie Selenium.

Selenium pozwala na definiowanie testów przy pomocy wielu języków - w tym Java.

Programowanie: przyklad-selenium

Przykładowa automatyzacja akcji na stornie internetowej.

Testy automatyczne - łączenie rozwiązań

Jak widać rozwiązania i narzędzia testujące łączą się w łatwy sposób. Używając poznanych bibliotek, moglibyśmy zdefiniować scenariusze testów przy pomocy Cucumber, gdzie kroki testów wykorzystywałyby Selenium do wykonywania akcji na stornie internetowej.

Testy wydajnościowe

Aplikacje backendowa uruchomiona na serwerze jest w stanie obsłużyć ograniczoną ilość zapytań.

Ograniczenia te mogą wynikać z różnych czynników takich jak:

- zasobów serwera (procesor, pamięć),
- wymogów pamięciowych aplikacji,
- złożoności obliczeniowej aplikacji,
- ilości przypisanych wątków
- czasu przetworzenia zapytania
- wydajności zależności.

Testy sprawdzające zachowanie aplikacji dla określonego obciążenia to testy wydajnościowe.

Testy wydajnościowe

Testy wydajnościowe przyjmują formę eksperymentowania i sprawdzania aplikacji w różnych warunkach.

Ich analiza opera się na 3 głównych aspektach:

- zachowanie w czasie: zdolność systemu do reagowania na dane wejściowe użytkownika lub systemu w określonym czasie i w określonych warunkach,
- wykorzystanie zasobów: ilość zasobów wymaganych przez system do poprawnego działania,
- przepustowość: ograniczenie systemu w zakresie limitów wydajnościowych (np. ograniczenia w ilości zapytań).

Testy wydajnościowe - Gatling

Gatling jest narzędziem pozwalającym na łatwe tworzenie testów wydajnościowych oraz ich automatyzację.

Testy opierają się na symulacjach wywołań aplikacji oraz zapisywaniu rezultatów dla danych warunków wykonywania.

Programowanie: przyklad-gatling

Test wydajnościowy dla RESTful API.

Uruchomienie:

```
mvn gatling:test
```

Rezultaty:

```
/target/gatling
```