Moduł 9: Testowanie

Rozdział ten z premedytacją nie został nazwany testowanie jednostkowe, tym bardziej testy integracyjne, systemowe – czy jakkolwiek by ich nie nazywać. Celem tego rozdziału nie jest prowadzenie świętej wojny o testowaniu aplikacji, ale dokładne opisanie jak wykorzystywany framework jest w stanie pomóc w codziennych testach programistycznych.

Nie ulega kwestii, że testy stanową integralną część procesu wytwarzania oprogramowania. Z takiego założenia wyszli także twórcy Springa, wbudowując we framework szereg programistycznych ułatwień, aby testy aplikacji rozpocząć na możliwie najwcześniejszym etapie. Nie chodzi zatem tylko i wyłączenie o testy w rozumienia zespołu QA, który pod koniec procesu weryfikuje zgodność aplikacji z wymaganiami, ale o znaczące pokrycie kodu aplikacji testami automatycznymi, przygotowywanymi przez programistów i uruchamianych w sposób ciągły, przez cały czas pracy nad systemem. Stąd też mowa zarówno o testach jednostkowych, jak i integracyjnych – Spring oferuje szereg narzędzi wspierających oba etapy.

Dodatkową i niezaprzeczalną zaletą testowania już na wczesnym etapie prac nad aplikacją jest możliwość weryfikacji poprawności pewnych założeń, bez konieczności całościowej integracji systemu. Efektywnie wykorzystując testy, bardzo szybko otrzymujemy zwrotną informację o postępie prac i poprawności działania poszczególnych komponentów. Dzieląc system na małe, luźno powiązane ze sobą komponenty (ang. *loose coupling*), otrzymujemy możliwość selektywnego testowania poszczególnych bloków, nawet w sytuacjach, gdy pozostałe części aplikacji nie są jeszcze gotowe, nie kompilują się bądź w ogóle ich nie ma. Taka modularna architektura jest najważniejszym paradygmatem frameworka, stąd też bardzo duże wsparcie dla testów nie jest zaskakujące.

Testy jednostkowe

Jak już zaznaczyłem na początku, niniejszy moduł obejmuje dwa obszary: testy jednostkowe oraz integracyjne – sama dyskusja o testowaniu złożonych aplikacji wykracza zdecydowanie poza zakres tego szkolenia. Test jednostkowy (co sama nazwa wskazuje) obejmuje swoim zakresem jedną element – jednostkę. Z reguły jest to jeden komponent (czyli jedna klasa Java), której działania weryfikujemy poprzez podanie odpowiednich danych wejściowych. Test jest także klasą języka Java, którą jednak przygotowujemy nieco inaczej niż zwykła klasę. W zależności od tego, które narzędzie do testowania preferujemy: JUnit czy TestNG, występują drobne różnice w konstrukcji klasy testowej. Nie będziemy jednak teraz się na tym skupiać – wszystkie testy wykorzystywać będą narzędzie JUnit.

W najprostszej postaci, testowany komponent nie jest w żaden sposób zależny od innych elementów systemu. W takie sytuacji test jest trywialny i nie warto się nim zajmować. W bardziej złożonych przepadkach, klasa posiada pewne zależności, konieczne do poprawnego działania aplikacji.

```
public class PrettyStringTaskPrinter implements TaskPrinter {

	@Value("${date.pattern}")
	String pattern;

	@Autowired
	TasksService tasksService;

	public void printCurrentTasks() throws Exception {
		SimpleDateFormat format = new SimpleDateFormat(pattern);

		List<Task> tasks = tasksService.getTasks();
		for (Task t : tasks) {
			StringBuilder b = new StringBuilder();
			b.append(" * ").append(t.getTitle())
				.append(" (")
				.append(t.getDescription()).append(")\n")
				.append("\t").append("[")
				.append(format.format(t.getDate())).append(" for ")
				.append(t.getDuration()/1000/60/60)
				.append(" hour(s)")
				.append("]");
			System.out.println(b.toString());
		}
	}
}
```

Zacznijmy od prostego przykładu klasy `PrettyStringTaskPrinter`. Klasa wyświetla na ekran zadania, poprawnie sformatowane, używające odpowiedniego formatu daty. Jeżeli spróbujemy zastanowić się nad testem tej klasy, sprawdzającym czy formatowanie jest poprawne, zauważymy że taki ten niespecjalnie jest możliwy, że mamy do czynienia ze ścisłymi powiązaniami. Widzimy, że klasa zależy od dwóch komponentów: `TasksService` (pobierającego dane z bazy) oraz `System.out` (poprzez który wypisywane są informacje na ekran). Wczesne testowanie ma właśnie za zadanie wyeliminować takie sytuacje, a należy je wyeliminować, ponieważ mimo iż w momencie tworzenia klasa realizowała postawione przed nią zadanie, jej projekt utrudnia jakiekolwiek modyfikacje.

Poprawnie zbudowana klasa nie musi zależeć od komponentu pobierającego zadania z bazy danych; wystarczy przekazać listę takich zadań jako parametr. Dodatkowe, zamiast zapisywać bezpośrednio do konsoli systemowej, klasa mogłaby albo zwracać dane w postaci ciągu znaków, albo przyjmować dodatkowy parametr typu `PrintStream`, do którego zapisywane byłby dane. Wtedy można by stworzyć test, który byłby prawdziwie jednostkowy.

Namiastki (ang. mocks)

Niestety, rzadko kiedy poruszamy się w świecie tak idealnym i z sytuacjami opisanymi powyżej należy sobie radzić za pomocą namiastek obiektów zależnych (czyli nazywając z angielska: mocków). Mock Object jest to obiekt, który udaje oryginalny obiekt, zachowując z nim kompatybilność na poziomie interfejsu (oferuje dostęp do takich samych metod publicznych), jednocześnie umożliwiając utworzenie dowolnej odpowiedzi. Co za tym idzie, z pomocą namiastek, programista piszący testy dostaje narzędzie umożliwiające dowolne wpłynięcie na dane przekazane do testowanej metody i poprzez to możliwość dokładnego sprawdzenia wszystkich możliwych stanów w obrębie metody.

Spring jako taki nie skupia się jednak na narzędziach do testowania, polecając w wyżej wymienionych sytuacjach wykorzystanie specjalistycznych narzędzi takich jak Mockito lub EasyMock. Spring oferuje

jednak kilka wyspecjalizowanych klas narzędziowych szczególnie dobrze sprawdzających się w testach na granicy samego frameworka i serwera aplikacji, umożliwiając testowanie klas wykorzystujących serwlety bądź odwołania do drzewa JNDI.

```
public class PrettyStringTaskPrinterTest {

	PrettyStringTaskPrinter printer;
	PrintStream sysout;

	@Before
	public void setup() {
		printer = new PrettyStringTaskPrinter();
		printer.setPattern("dd-MM-yyyy");
		TasksService service = mock(TasksService.class);
		printer.setTasksService(service);

		sysout = mock(PrintStream.class);
		System.setOut(sysout);
	}

	@Test
	public void printerTest() throws Exception {
		Task t = new Task.Builder().withTitle("test task")
				.withDescription("Description")
				.withDate(new Date())
				.withDuration(1000*60*60).build();

		when(printer.getTasksService().getTasks())
			.thenReturn(Arrays.asList(t));


		printer.printCurrentTasks();
		verify(sysout)
		.println(" * test task (Description)\n\t[11-03-2012 for 1 hour(s)]");
	}

}
```

Przykładowy test wykorzystujący Mockito mógłby wyglądać jak powyżej. Jak widać, ze względu na wadliwą konstrukcję klasy jest on nietrywialny – stąd też bardzo duży nacisk kładziony jest na poprawną budowę klas oraz ich testowanie. Zapewnia to bardzo szybkie sprzężenie zwrotne pozwalające uniknąć problemów takich jak powyższe.

Testy integracyjne

Wróćmy jednak do Springa i do wsparcia testów oferowanego przez framework. Jak już pisałem, Spring oferuje szereg namiastek związanych z tworzeniem testów na pograniczu kontekstu zarządzanego przez framework oraz zewnętrznym otoczeniem (np. serwerem aplikacji). Dla przykładu, możemy w całości testować kontrolery, bez konieczności uruchamiania kontenera serwletów.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("file:src/main/webapp/WEB-INF/spring/appServlet/servlet-
context.xml")
public class TaskServletTest {

	@Autowired
	ApplicationContext ctx;

	@Autowired
	TaskController controller;

	@Test
	public void servletTest() throws Exception {
		MockHttpServletRequest req =
			new MockHttpServletRequest("GET", "/tasks");
		MockHttpServletResponse resp = new MockHttpServletResponse();
		HandlerAdapter handlerAdapter =
			ctx.getBean(AnnotationMethodHandlerAdapter.class);
		final ModelAndView model = handlerAdapter.handle(req, resp,
				controller);

		assertViewName(model, "tasks");
		assertAndReturnModelAttributeOfType(model, "task", Task.class);
		assertAndReturnModelAttributeOfType(model, "tasks", List.class);
	}

}
```

W powyższym teście sprawdzamy czy wywołując adres /tasks napisany przez nas kontroler zachowa się poprawnie, tj. przekierowuje użytkownika do widoku o nazwie „tasks” oraz doda do modelu dwa atrybuty o wymienionych powyżej typach. Wykorzystujemy w tym celu obiekty `MockHttpServletRequest` oraz `MockHttpServletResponse`, które to dostarczane są przez framework. Natomiast dodatkowe asercje (assertViewName) pozwalają operować bezpośrednio na obiekcie `ModelAndView` – przez co test staje się niezwykle prosty oraz czytelny (w porównaniu do zaprezentowanego wcześniej testu opartego o generyczne narzędzie).

Najważniejsze jednak w tym teście jest to, że poprzez wykorzystanie adnotacji `@RunWith(SpringJUnit4ClassRunner.class)` – test staje się pełnoprawnym komponentem, umożliwiającym m.in. wstrzykiwanie zależności (co zresztą zostało uczynione z kontrolerem). Poprzez podanie w parametrach adnotacji `@ContextConfiguration` lokalizacji pliku konfiguracyjnego, zainicjowany został kontekst wraz ze wszystkimi zależnościami (w tym połączeniem do bazy danych). Nic więc nie stoi na przeszkodzie, aby testować nie tylko pojedyncze komponenty, ale także całe grupy komponentów. Na tym polegają właśnie testy integracyjne.

Modularyzacja aplikacji

Pewną niedogodnością poprzedniego testu jest uruchomianie pełnego kontekstu – wraz z pełnym dostępem do bazy danych. Nie jest to jednak konieczność, a wynika raczej z konstrukcji plików konfiguracyjnych. Wracamy tutaj do modularyzacji i sposobów dzielenia plików konfiguracyjnych, o czym pisałem w poprzednim module. Jeżeli skorzystamy z tej techniki, to bez większych problemów, zamiast „produkcyjnej” bazy danych, skorzystamy w osobnego pliku konfiguracyjnego dla testów, który korzystać będzie np. z bazy danych przechowywanej w pamięci. W tym celu utworzymy dodatkowy plik konfiguracyjny test-context-db.xml (będący kopią oryginalnego app-context-db.xml), który będzie zawierał odpowiednią konfigurację bazy danych.

```
    <jdbc:embedded-database id="dataSource">
        <jdbc:script location="classpath:schema.sql"/>
    </jdbc:embedded-database>
```

Tak podzielone pliki konfiguracyjne możemy użyć w naszym teście, modyfikując atrybuty adnotacji `@ContextConfiguration`.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({
        "classpath:META-INF/spring/app-properties.xml",
        "classpath:test-context-db.xml",
        "classpath:META-INF/spring/app-context.xml",
        "file:src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml"
})
public class TaskServletTest {
        //..
}
```

Dla zmienionej konfiguracji test niestety nie działa. Wynika to z faktu zmiany bazy danych. Z jednej strony bardzo dobrze, że przestajemy w testach korzystać z „produkcyjnej" bazy danych i zaczynamy korzystać z instancji dedykowanej testom. Możemy teraz dowolnie manipulować danymi, bez ryzyka, że uszkodzone zostaną zewnętrzne dane. Minusem natomiast jest to, że na obecną chwilę baza nie istnieje i nie ma w niej żadnych danych – wszystko musimy przygotować samodzielnie.

Wbrew pozorom, mimo wydawać by się mogło większego nakładu pracy, jest to pozytywny aspekt testów. Klasa testowa staje się całością, jest niezależna od otoczenia, od danych w innych systemach – przez co testy stają się pewniejsze. Nie ma prawa zaistnieć sytuacja, w której testy przestają działać na skutek zmian w środowisku dokonanych przez osoby trzecie. Jeśli dodatkowo przygotowanie danych testowych sprowadza się do kilku dodatkowych linijek SQL – nie stanowi to chyba zbytniego wyzwania.

Dla kontekstu skonfigurowanego tak jak omówiono powyżej, metoda testująca nie tylko typy zwracane przez kontroler, ale także wartości, przedstawiona jest poniżej.

```
@Test
public void servletTest() throws Exception {
	MockHttpServletRequest req =
		new MockHttpServletRequest("GET", "/tasks");
	MockHttpServletResponse resp = new MockHttpServletResponse();
	HandlerAdapter handlerAdapter =
		ctx.getBean(AnnotationMethodHandlerAdapter.class);
	final ModelAndView model = handlerAdapter.handle(req, resp,
			controller);

	assertViewName(model, "tasks");
	assertAndReturnModelAttributeOfType(model, "task", Task.class);
	assertAndReturnModelAttributeOfType(model, "tasks", List.class);

	LocalDate date = LocalDate.now();
	final Task testTask = new Task.Builder().withId(1l)
			.withTitle("test task")
			.withDescription("Description")
			.withDate(date.toDate())
			.withDuration(1000*60*60).build();

	assertModelAttributeValue(model, "tasks", Arrays.asList(testTask));
	assertModelAttributeValue(model, "task", new Task());
}
```

Podsumowanie

Na powyższych przykładach widać, że dobry framework to nie tylko narzędzie w którym implementujemy wymagania funkcjonalne. Jest to także zestaw pomocniczych klas, które wspomagają cały proces wytwórczy – aż po omówione tutaj testy skończywszy. Testy to właśnie jest jedną z cech, która bardzo mocno odróżniała Springa od świata opartego na komponentach EJB. Twórcy Springa wyszli z założenia, że testy pisane przez programistów są koniecznością i zaoferowali im bardzo dobrą platformę umożliwiającą wydajne ich pisanie i zarządzanie nimi. W świecie EJB takie narzędzia zaczęły pojawiać się dopiero niedawno, natomiast w Spring są od zawsze. Co więcej, autorzy frameworka od samego początku wyszli z założenia, że środowisko testowe i produkcyjne to dwa różne systemy, a co za tym idzie wymagają one odmiennej konfiguracji. Takie trudno rozwiązywalne w świecie EJB niuanse w Springu są na wyciągnięcie ręki.