TestNG

Co to jest, po co to jest i co w tym jest?

Co to jest?

TestNG jest biblioteką programistyczną służącą do pisania testów dla języka Java.

Została stworzona przez Cedrica Beusta.

Inne przykładowe biblioteki do testowania:

- JUnit
- Mockito
- Spock

Po co to jest?

Biblioteka TestNG powstała ze względu na liczne ograniczenia JUnit oraz NUnit. W momencie powstania oferowała nowe funkcje nieobecne wówczas w innych bibliotekach (np. JUnit 3).

Taką nową funkcjonalnością były przykładowo adnotacje, dodane w JUnit dopiero w wersji 4, tj. jakiś czas po opublikowaniu TestNG

TestNG wspiera takie funkcjonalności jak:

- Testy jednostkowe
- Testy integracyjne
- Adnotacje
- Testy parametryzowane
- Testy grupowe
- Zależności
- Oraz wiele innych

Przykładowe adnotacje w TestNG

@Test	Oznacza klasę lub metodę jako część testu.
@BeforeSuit	Metoda zostanie uruchomiona, zanim wszystkie testy w tym pakiecie zostaną uruchomione.
@AfterSuite	Metoda zostanie uruchomiona po uruchomieniu wszystkich testów w tym pakiecie.
@BeforeTest	Metoda zostanie uruchomiona, zanim zostanie uruchomiona jakakolwiek metoda testowa należąca do klas wewnątrz znacznika <test>.
@AfterTest	Metoda zostanie uruchomiona po uruchomieniu wszystkich metod testowych należących do klas wewnątrz znacznika <test>.
@BeforeGroups	Lista grup, dla których ta metoda konfiguracji będzie wcześniej uruchamiana. Ta metoda jest uruchamiana na krótko przed wywołaniem pierwszej metody testowej należącej do którejkolwiek z tych grup.

@AfterGroups	Lista grup, dla których ta metoda konfiguracji będzie wcześniej uruchamiana. Ta metoda jest uruchamiana na krótko po wywołaniu ostatniej metody testowej należącej do którejkolwiek z tych grup.
@BeforeClass	Metoda zostanie uruchomiona przed wywołaniem pierwszej metody testowej w bieżącej klasie.
@AfterClass	Metoda zostanie uruchomiona po uruchomieniu wszystkich metod testowych w bieżącej klasie.
@BeforeMethod	Metoda zostanie uruchomiona przed uruchomieniem każdej metody testowej w bieżącej klasie.
@AfterMethod	Metoda zostanie uruchomiona po każdej metodzie testu.

Pisanie testów

TestNG opiera się na odpowiednio przygotowanych asercjach w Javie dodanych w wersji 4. Skoro zatem możemy bezpośrednio użyć asercji to po co korzystać z biblioteki?

Otóż w trakcie rozwoju projektu, testowanie staje się trudnym w utrzymaniu procesem i w takiej sytuacji staje się bardziej sensowne używanie biblioteki w celu lepszego zarządzania projektem. Zarówno JUnit, jak i TestNG są zgodne z konwencjami xUnit , ale mają kilka elementów, o których warto tutaj wspomnieć:

- Grupy
- Paralelizm
- Sparametryzowany test
- Zależności.

Grupy

TestNG oferuje różne adnotacje. Prawdopodobnie najbardziej zauważalna jest możliwość stworzenia grup testowych i uruchomienia kodu przed / po grupach przypadków testowych. Ponadto pojedyncze testy mogą należeć do wielu grup, a następnie działać w różnych kontekstach (np. wolne lub szybkie testy). Jest to nieformalna warstwa pomiędzy przypadkiem testowym a scenariuszem testowym.

```
@Test(groups = { "sanity", "insanity" })
```

Grupy

Podobna funkcja istnieje w kategoriach JUnit, ale brakuje jej adnotacji @BeforeGroups / @AfterGroups TestNG. W JUnit natomiast dopiero w wersji 5 wprowadzana jest nowa koncepcja, która nazywa się @Tag i ma podobne zastosowanie co grupy w TestNG.

```
@Tag(“sanity”)
@Tag(“insanity)
void sampleTest()
{ … }
```

Testy równoległe

Jeśli chcesz uruchomić ten sam test równolegle na wielu wątkach, TestNG oferuje prostą w użyciu adnotację:

```
@Test(threadPoolSize = 3, invocationCount = 9)
public void sampleTest()
{ … }
```

TestNG daje również możliwość uruchomienia całych pakietów testów równolegle, jeśli są one określone w pliku konfiguracyjnym XML.

Testy sparametryzowane

Jest to problem podawania różnych danych wejściowych do tego samego przypadku testowego. Podstawowa idea jest taka: stworzenie tablicy dwuwymiarowej, Object [] [], która zawiera parametry.

Jednak oprócz dostarczania parametrów za pomocą kodu (@DataProvider), TestNG może również obsługiwać XML do podawania danych, plików CSV, a nawet plików tekstowych.

Zależności między grupami / metodami

Ponieważ TestNG miał na uwadze szerszy zakres testów niż tylko testy jednostkowe, daje on kilka niespotykanych chociażby w JUnit funkcji.

Przykładowo pozwala zadeklarować zależności między testami i pominąć je, jeśli test zależności nie przeszedł.

```
@Test(dependsOnMethods = { "dependOnSomething" })
public void sampleTest()
{ … }
```

Uruchomienie testów

Podczas tworzenia nowego testu nia ma konieczności uwzględniania metody main w naszym kodzie, ponieważ biblioteki zapewniają własną główną metodę która zarządza wykonaniem poszczególnych testów.

Jeśli potrzebujesz niestandardowego rozwiązania, TestNG pozwala użyć własnego runnera. Jednakże nie jest to takie proste. Warto zauważyć, że TestNG obsługuje konfiguracje XML, która okazuje się przydatna w wielu przypadkach.

Jeśli chodzi o faktycznie przeprowadzane testy, TestNG ma wsparcie CLI, działające przez ANT i wtyczki dostępne dla wybranych IDE z całkiem podobną funkcjonalnością.

Raportowanie wyników

Raportowanie wyników testów jest ważną częścią całego procesu, w tym temacie TestNG dostarcza nam pewne rozwiązania.

Raporty TestNG są generowane domyślnie w folderze wyjściowym, który zawiera dokument w formacie html z tabelami zawierającymi wszystkie dane testowe, informację o testach udanych / nieudanych / pominiętych, czasie trwania testów, informację o wykorzystanych danych wejściowych, oraz dzienniki testów.

Ponadto eksportuje wszystko do pliku XML, który można wykorzystać do stworzenia własnego szablonu raportu.

Automatyzacja przebiegu testów

TestNG współpracuje z takimi narzędziami jak np. Jenkins. Zostały dla niego udostępnione wtyczki wspomagające tworzenie raportów m.in.: w formie graficznej. Tak wygenerowane raporty może być bezpośrednio wysłany do odbiorcy wybranym przez Ciebie sposobem. Prostym przykładem może być użycie: TestNG z narzędziem Jenkins i integracja z pocztą elektroniczną oraz serwisem Slack.

Społeczność

TestNG zrzesza wielu użytkowników, a odpowiedzi na nawet najbardziej nurtujące pytania są dostępne na wielu serwisach oraz grupach internetowych.

Zalety TestNG w stosunku do JUnit5 (5.2)

Prostota

JUnit 5 jest zbudowany z wielu modułów, do pisania zwykłych testów potrzeba chociażby JUnit Platform i JUnit Jupiter. Do kolejnych funkcjonalności trzeba dołączyć kolejne moduły (np. testy parametryzowane - moduł junit-jupiter-params).

TestNG w jednym swoim module zawiera wszystkie potrzebne funkcjonalności.

Raport HTML

W JUnit 5 do stworzenia raportu HTML potrzebny jest dodatkowy plugin (maven-surefire-report-plugin).

W TestNG, chociaż raporty wydają się być trochę przestarzałe, są tworzone automatycznie przy uruchomieniu testów.

Testy równoległe

W JUnit 5 (przynajmniej do wersji 5.2, obecnie jest to funkcja eksperymentalna) nie ma wsparcia dla testowania równoległego.

W TestNG można odpowiednio skonfigurować plik XML aby wykonanie testów odbywało się równolegle.

Przykłady

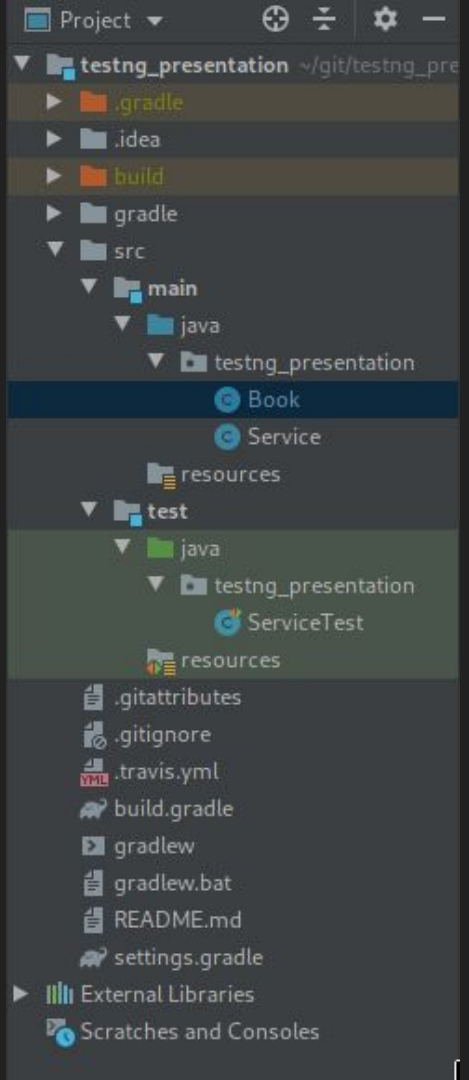
W ramach tej prezentacji pokażę podstawowe funkcjonalności TestNG z pominięciem grup (wszystkie pliki dostępne w ramach projektu na githubie).

Do tego utworzymy małą aplikację przy użycia gradle pozwalająca zarządzać książkami w naszym sklepie.

Będzie ona zawierać klasę Book posiadającą pola: name, price oraz odpowiednie gettery i settery. Druga klasa Service będzie udostępniać odpowiednie metody do zarządzania naszą listą książek (dodawanie, edycja, sumowanie cen, itd.).

W ramach testów zaś przetestujemy klasę service używając TestNG.

Struktura projektu



Klasa Book

```java
package testng_presentation;

public class Book {
    private String name;
    private double price;

    public Book(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public void setName(String bName) {
        this.name = bName;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return "Book{" +
                "bName='" + name + '\'' +
                ", price=" + price +
                '}';
    }
}
```

Klasa Service

```java
public class Service {
    private final List<Book> bookList = new ArrayList<>();

    //Add book
    public void addBook(Book book) {
        bookList.add(book);
    }

    //Edit book
    public void editBook(String name, Book book) {
        if (name == null) {
            throw new NullPointerException();
        }

        if (checkIfExists(name)) {
            for (Book temp : bookList) {
                if (temp.getName().equals(name)) {
                    temp.setName(book.getName());
                    temp.setPrice(book.getPrice());
                }
            }
        } else {
            System.out.println("ERROR: BOOK COULD NOT BE FOUND");
        }
    }

    //Delete book
    public void deleteBook(String name) {
        if (name == null) {
            throw new NullPointerException();
        }

        if (checkIfExists(name)) {
            bookList.removeIf(next -> next.getName().equals(name));
        } else {
            System.out.println("ERROR: BOOK COULD NOT BE FOUND");
        }
    }
```

```java
//Get book by name
public Book getBookByName(String name) {
    if (name == null) {
        throw new NullPointerException();
    }

    for (Book temp : bookList) {
        if (temp.getName().equals(name)) {
            return temp;
        }
    }
    return null;
}

//Count books
public int countBooks() {
    return bookList.size();
}

//Print books
public String printBooks() {
    return bookList.stream()
            .map(String::valueOf)
            .collect(Collectors
                    .joining( delimiter: "\n{"
                            , prefix: "{"
                            , suffix: "}"));
}

//Count sum
public double sumBook() {
    double sum = 0;
    for (Book book : bookList) {
        sum += book.getPrice();
    }
    return sum;
}
```

```java
//Count sum
public double sumBook() {
    double sum = 0;
    for (Book book : bookList) {
        sum += book.getPrice();
    }
    return sum;
}

//Check if book exists
public boolean checkIfExists(String name) {
    if (name == null) {
        throw new NullPointerException();
    }

    for (Book temp : bookList) {
        if (temp.getName().equals(name)) {
            return true;
        }
    }
    return false;

}
```

Początek klasy testowej

```java
public class ServiceTest {
    private Service service;
    private Book bookAlpha;
    private Book bookBeta;

    @BeforeMethod
    public void setUp() {
        service = new Service();
        bookAlpha = new Book( name: "Alpha",  price: 21);
        bookBeta = new Book( name: "Beta",  price: 37);
    }

    @AfterMethod
    public void tearDown() {
        service = null;
        bookAlpha = null;
        bookBeta = null;
    }
```

Na początku za pomocą adnotacji @BeforeMethod i @AfterMethod przygotowujemy testy, tworząc obiekt klasy Service oraz tworząc podstawowe 2 książki na których będziemy opierać większość testów. Adnotacje te jak zostało wspomniane wcześniej pozwalają nam na uruchomienie od nowa kodu umieszczonego w metodach na początku i końcu każdego testu.

Podstawowe testy

```java
@Test
public void addBookPositiveTest() {
    service.addBook(bookBeta);

    Assert.assertNotNull(service.getBookByName("Beta"));
}

@Test
public void addBookNegativeTest() {
    service.addBook(bookBeta);

    Assert.assertNull(service.getBookByName("Alpha"));
}

@Test
public void addOneBookPositiveCountTest() {
    service.addBook(bookAlpha);

    Assert.assertEquals(service.countBooks(), expected: 1 );
}

@Test
public void addTwoBooksNegativeCountTest() {
    service.addBook(bookAlpha);
    service.addBook(bookBeta);

    Assert.assertNotEquals(service.countBooks(), 0);
}
```

Najpierw testujemy podstawową metodę - “addBook”, po czym weryfikujemy ją na różne sposoby używając assercji: “NotNull, Null, Equals, NotEquals”, tj. odpowiednio sprawdzamy poprawność książki poprzez zbadanie czy w efekcie dodania książki zwracany (bądź nie) jest null lub czy też parametry dodanych książek się zgadzają. Warto zwrócić uwagę na to, że w przypadku assercji (Not)Equals najpierw podajemy faktyczny wynik, a potem spodziewany.

Podstawowe testy

```java
@Test
public void getBookByNamePositiveNameTest() {
    service.addBook(bookAlpha);
    service.addBook(bookBeta);
    Book temp = service.getBookByName("Beta");

    Assert.assertEquals(temp.getName(), expected: "Beta");
}

@Test
public void getBookByNamePositivePriceTest() {
    service.addBook(bookAlpha);
    Book temp = service.getBookByName("Alpha");

    Assert.assertEquals(temp.getPrice(), expected: 21.00);
}

@Test
public void editBookPositiveTest() {
    service.addBook(bookAlpha);
    service.editBook( name: "Alpha", bookBeta);
    Book temp = service.getBookByName("Beta");
    String name = temp.getName();

    Assert.assertEquals(name, expected: "Beta");
}
```

```java
@Test
public void deleteBookPositiveTest() {
    service.addBook(bookAlpha);
    service.addBook(bookBeta);
    service.deleteBook( name: "Alpha");

    Assert.assertEquals(service.countBooks(), expected: 1);
}

@Test
public void sumBooksPositiveTest() {
    service.addBook(bookAlpha);
    service.addBook(bookBeta);
    double sum = service.sumBook();

    Assert.assertEquals(sum, expected: 58.00);
}
```

Kontynuujemy podstawowe testy różnych metod używając najbardziej podstawowych dostępnych asercji.

Inne testy

```
@Test(enabled = false)
public void printBooksDisabledTest() {
    service.addBook(bookBeta);
    service.addBook(bookAlpha);

    Assert.assertEquals(service.printBooks(), expected: "To się nie wykona");
}
```

W tym teście korzystamy z możliwości ignorowania danego testu. Jeśli nie chcemy aby się wykonał wraz z innymi używamy funkcji enabled ustawiając ją na na false. Dzięki temu nie musimy się męczyć z wykomentowaniem testu lub jego usuwaniem, gdy w danej chwili nie jest nam potrzebny.

Inne testy

```java
@Test(expectedExceptions = NullPointerException.class)
public void nullExceptionOnGetBookByNameTest() {
    service.getBookByName(null);
}
```

Jeśli nasze metoda może wyrzucać wyjątek, możemy również i to przetestować poprzez odpowiedni dopisek przy adnotacji, gdzie także możemy ustawić rodzaj wyjątku na jaki oczekujemy. W kodzie testu nie musimy już ustawiać żadnej asercji, wystarczy by została wywołana metoda wyrzucająca wyjątek.

Inne testy

```java
@Test
@Parameters("Gamma")
public void parametrizedCheckIfExistsBookTest(@Optional("Gamma") String name) {
    service.addBook(bookAlpha);
    service.addBook(bookBeta);

    Assert.assertFalse(service.checkIfExists(name));
}
```

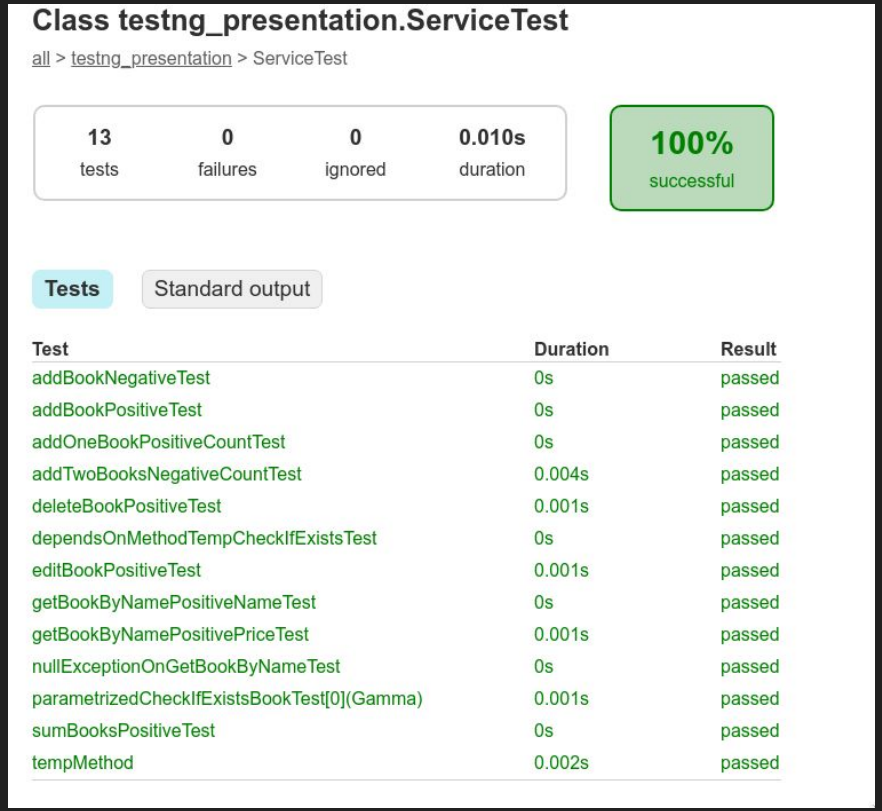
W przypadku gdy chcemy użyć sparametryzowanych testów należy dodać odpowiednią adnotację i argument w funkcji. Jeżeli nie chcemy grzebać w pliku XML i testować na miejscu musimy dodać adnotację “@Optional” przed argumentem, w innym wypadku test się nie powiedzie. Można użyć także adnotacji @DataProvider i stworzyć odpowiednią metodą z odpowiednimi parametrami, lub użyć pliku csv/tekstowego.

Inne testy

```java
@Test(dependsOnMethods = { "tempMethod" })
public void dependsOnMethodTempCheckIfExistsTest() {

    service.addBook(bookAlpha);

    Assert.assertTrue(service.checkIfExists( name: "Alpha"));
}

@Test
public void tempMethod() {
    System.out.println("Metoda potrzebna do ukazania dependsOnMethods");
    Assert.assertTrue( condition: true);
}
```

Jeżeli chcemy uzależnić test od pozytywnego wykonania od innego testu najpierw, wskazujemy przy adnotacji nazwę koniecznej metody. Jeżeli spróbujemy wywołać ręcznie jedynie pierwszy test, nie powiedzie się on. Jeżeli tempMethod zakończyłaby się niepowodzeniem, metoda zależna od niej nie wykona się.

Raport



Class testng_presentation.ServiceTest

all > testng_presentation > ServiceTest

13	0	0	0.010s
tests	failures	ignored	duration

100% successful

Tests | Standard output

Test	Duration	Result
addBookNegativeTest	0s	passed
addBookPositiveTest	0s	passed
addOneBookPositiveCountTest	0s	passed
addTwoBooksNegativeCountTest	0.004s	passed
deleteBookPositiveTest	0.001s	passed
dependsOnMethodTempCheckIfExistsTest	0s	passed
editBookPositiveTest	0.001s	passed
getBookByNamePositiveNameTest	0s	passed
getBookByNamePositivePriceTest	0.001s	passed
nullExceptionOnGetBookByNameTest	0s	passed
parametrizedCheckIfExistsBookTest[0](Gamma)	0.001s	passed
sumBooksPositiveTest	0s	passed
tempMethod	0.002s	passed

W folderze build/reports/tests/test znajdziemy plik index.html zawierający raport wykonania testów: ilość wykonanych testów, testy zakończone niepowodzeniem, testy zignorowane oraz procentowa ilość testów zakończonych sukcesem.

Podsumowanie

TestNG jest ciekawym oraz przydatnym narzędziem do testowania oprogramowania, w pewnych miejscach lepszy od JUnita a w innych gorszy. Należy pamiętać że celem TestNG jest nieco szerszy zakres testów niż jest to w przypadku JUnita. Jak zwykle użycie tego lub innego narzędzia powinno być uzależnione od aktualnych potrzeb testera.

Juliusz Korczakowski
246817