Testowanie kodów aplikacji oraz ich bezpieczeństwa

Laboratorium 7

Wymagania

1. Środowisko programistyczne umożliwiające przygotowanie aplikacji w frameworku Spring Boot. Alternatywnie dowolne środowisko programistyczne umożliwiające przygotowanie aplikacji implementującej interfejs REST API.

Przydatne źródła

2. https://www.kodolamacz.pl/blog/wprowadzenie-do-testowania-aplikacji-w-srodowisku-java/

Wstęp

Implementacja poszczególnych modułów wchodzących w skład systemu rozproszonego wymaga dbałości o jakość dostarczanego systemu oraz zapewnienia, że przygotowane usługi pozbawione są oczywistych błędów, wad i luk bezpieczeństwa. Jednym ze sposobów maksymalizacji skuteczności implementowanego rozwiązania jest odpowiednie podejście do testowania usług.

Podstawowa klasyfikacja testów wygląda następująco:

1. **Jednostkowe** (tak zwane unit tests) - łatwe do tworzenia i modyfikacji, testują małą funkcjonalność, mały fragment kodu w oderwaniu od reszty systemu.
2. **Integracyjne** (inaczej testy warstwy serwisów) - w tych testach skupiamy się na sprawdzeniu czy nasze moduły / komponenty / fragmenty kodu potrafią ze sobą współpracować, testujemy samą ich integrację, bez wnikania czy te moduły są prawidłowe, gdyż to robią testy jednostkowe.
3. **End-to-end** (inaczej funkcjonalne lub akceptacyjne) - testy całościowe, sprawdzają jakąś funkcjonalność systemu w sposób całościowy, np. od wejścia do wyjścia gdy przetwarzamy jakieś dane, niekiedy też tutaj pojawiają się testy warstwy interfejsu dla użytkownika, czyli tak zwane testy UI (user interface), choć nie każdy system komputerowy taką warstwę w ogóle ma [1].

Dodatkowo przygotowanie odpowiednich testów wymaga nie tylko umiejętności technicznych, ale także odpowiedniego podejścia do projektowania i implementacji kodu. Jedną z koncepcji wskazujących jak powinna wyglądać praca w ramach wytwarzania poszczególnych funkcjonalności jest TDD. TDD to sposób wytwarzania oprogramowania zbudowany na trzech fazach **Red->Green->Refactor**. Sprowadza się to do powtarzania następującego cyklu:

1. Piszemy test, który kończy się niepowodzeniem, bo nie została wykonana jeszcze implementacja (Red).
2. Przygotowujemy kod, który powoduje, że test kończy się sukcesem (Green).
3. Jeżeli zauważymy taką potrzebę poprawiamy przypadki testowe i implementacje danej metody (Refactor). W tym miejscu zdarza się, że wrócimy do fazy Red. Cykl ten powtarzamy do momentu, aż skończymy implementować dany kawałek kodu.

Poza wspomnianymi rodzajami testów wykonywane są także testy penetracyjne. Test penetracyjny polega na przeprowadzenia kontrolowanego ataku na system. Jego celem jest ocena stanu bezpieczeństwa, a przede wszystkim analiza występowania w implementacji znanych podatności. W ramach testu penetracyjnego weryfikowane są luki wynikające z błędnej konfiguracji, podatności w wykorzystywanym oprogramowaniu lub modułach sprzętowych. Testy penetracyjne weryfikują także podatności na poziomie procedur dotyczących środków zabezpieczeń, a niejednokrotnie również zagrożenia wynikają z braku świadomości użytkowników systemu.

Przykładowe metodyki testów penetracyjnych [2]:

1. OWASP (The Open Web Application Security Project)
2. PTES (The Penetration Testing Execution Standard)
3. OSSTMM (Open Source Security Testing Methodology Manual)
4. NIST SP 800-42 (Guideline on Network Security Testing, 2003)
5. NIST SP 800-115 (Technical Guide to Information Security Testing")
6. ISAAF (OISSG Penetration Testing Framework, 2008)

Zadanie 1

Zadanie polega na utworzeniu prostego projektu w języku Java. Projekt powinien zawierać implementację klasy Range, która posiada dwa pola typu liczbowego o nazwie mix i max. Klasa implementuje ***metodę boolean isInRange(int number)*** która zwraca informację czy przekazana liczbą zawarta jest w przedziale min <= number <= max

1. Przygotować trzy przypadki testowe weryfikujące czy metoda działa poprawnie (np. dla liczby mniejszej, większej, znajdującej się wewnątrz przedziału, przypadki brzegowe)
2. Do przygotowanej klasy dodać konstruktor ***Range(int min, int max)***, który w przypadku gdy liczba min jest większa od liczby max rzuci odpowiedni wyjątek. Przygotować testy jednostkowe weryfikujące poprawność wyrzucania wyjątków.
3. Zmodyfikować implementację klasy testującej w taki sposób, żeby zaprezentować cykl życia klasy z testami jednostkowymi. Należy skorzystać z adnotacji @Before i odpowiednio zainicjować obiekty klasy Range.

Zadanie 2

Zadanie polega na przygotowaniu testów integracyjnych dla implementacji przygotowanej w ramach laboratorium 4 (Implementacja interfejsów REST).

1. Przygotować przypadek testowy weryfikujący dodanie użytkownika do bazy danych. W ramach tego testu należy wywołać odpowiednią metodę Controllera, która umożliwi zapis użytkownika do bazy danych. Następnie należy zweryfikować czy użytkownik o przekazanych danych jest dodany do bazy danych.
2. Przygotować przypadek testowy weryfikujący poprawność zachowania aplikacji w przypadku próby dodania użytkownika o istniejącym w bazie danych identyfikatorze.
3. Przygotować przypadki testowe weryfikujące poprawność implementacji dla próby usunięcia użytkownika (dla istniejącego i nieistniejącego identyfikatora).

Zadanie 3

Zadanie polega na odpowiedniej konfiguracji projektu przygotowanego w ramach laboratorium 4 (Implementacja interfejsów REST). Konfiguracja ta polega na dodaniu do pliku ***pom.xml*** pluginu ***dependency-check-maven***, służącego do analizy zależności wykorzystanych w projekcie.

1. Uruchomić implementacje aplikacji i w pliku pom.xml odnaleźć znacznik <plugins>…</plugins>.
2. Dodać plugin depency-check-maven

```
<plugin>
    <groupId>org.owasp</groupId>
    <artifactId>dependency-check-maven</artifactId>
    <version>5.3.2</version>
    <executions>
        <execution>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

3. Wywołać polecenie ***mvn dependency-check:check***
4. W folderze target odnaleźć i otworzyć w dowolnej przeglądarce plik ***dependency-check-report.html***.
5. Zaprezentować jedną z wykrytych podatności.
6. Zaproponować zmianę mającą na celu wykluczenie wskazanej podatności.
7. Wykonać ponownie polecenie wskazane w punkcie 3. oraz zaprezentować, że problem z wybraną podatnością został rozwiązany.

Rezultaty wykonanych zadań przedstawić w postaci sprawozdania.