JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 28 (2023) 236–241
Received: 2 June 2023
Accepted: 20 June 2023

Comparative analysis of methods for testing web applications

Analiza porównawcza sposobów testowania aplikacji internetowych

Tomasz Smyk*, Wojciech Superson, Małgorzata Plechawska-Wójcik

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

The aim of the study was to conduct a comparative analysis of testing approaches for web applications in the two most popular architectures: monolithic and microservices. For the purpose of the study, the server-side of the application (backend) was implemented twice with identical functionalities for each of these architectures, allowing for a precise comparison of testing differences for the same program capabilities. The results revealed that the monolithic application was easier and faster to test. However, the microservices architecture requires more energy spent on testing, but allows better scalability and elasticity for independent teams to develop applications. Each of the examined architectures certainly has its own advantages and drawbacks. Furthermore, the conducted research indicates that unit tests require significantly less time to execute. However, when it comes to comprehensive code analysis, integration tests outperform unit tests by covering a substantial portion of the application's code with a single test. Nonetheless, the best comprehensive code analysis and protection against unwanted functional changes can be achieved by employing all known types of tests.

Keywords: unit testing; integration testing; microservices architecture; monolithic architecture

Streszczenie

Celem pracy była analiza porównawcza sposobów testowania aplikacji internetowych w dwóch najpopularniejszych architekturach, tj. monolitycznej oraz mikroserwisowej. Na potrzeby badania strona serwerowa aplikacji (*ang. backend*) została napisana dwukrotnie z identycznymi funkcjonalnościami w każdej z tych architektur, aby móc dokładnie zestawić różnice w testowaniu tych samych możliwości programu. Wyniki ukazały aplikację monolityczną, jako łatwiejszą i szybszą do testowaniu. Architektura mikroserwisowa natomiast wymaga większego nakładu pracy związanej z testowaniem, ale pozwala na uzyskanie większej skalowalności i elastyczność w rozwijaniu aplikacji przez niezależne zespoły. Każda z badanych architektur z pewnością ma swoje wady i zalety. Z przeprowadzonych badań wynika również, że testy jednostkowe potrzebują dużo mniej czasu na wykonanie, jednak jeśli chodzi o kompleksowość analizy kodu testy integracyjne zwyciężają, pokrywając jednym testem znaczną część kodu aplikacji. Najlepszą jednak kompleksowość analizy kodu i zabezpieczenie aplikacji przed niechcianymi zmianami funkcjonalności uzyskamy stosując wszystkie znane nam rodzaje testów.

Słowa kluczowe: testy jednostkowe; testy integracyjne; architektura mikroserwisowa; architektura monolityczna

*Corresponding author

Email address: **tomasz.smyk@pollub.edu.pl** (T. Smyk)

1. Wstęp

Celem niniejszej pracy jest analiza i porównanie sposobów testowania aplikacji internetowych. Przedstawione zostaną różne sposoby, którymi obecnie rynek komercyjny zwykł testować aplikacje przed, w trakcie i po wdrożeniu. Testowanie aplikacji znacząco różni się w zależności od zastosowanej architektury oraz czy jest to warstwa serwerowa aplikacji, czy też warstwa kliencka. Podstawowe pytanie badawcze pracy brzmi: które rodzaje testów są bardziej nastawione na wydajność wykonania, a które na kompleksową analizę aplikacji kosztem właśnie wydajności. Kolejnym pytaniem, które nasuwa się podczas analizy tematu jest czy obecnie popularny, mikroserwisowy, podział systemu aplikacji wpływa na dywersyfikację typów testów inaczej w przypadku klasycznego monolitycznego sposobu podziału aplikacji.

Każda z metod testowania aplikacji internetowych ma swoje wady, ale również posiada zalety. Skuteczność danego rodzaju testu może być zależna od wielu czynników - od stopnia skomplikowania systemu aż po stopień zintegrowania danej aplikacji z innymi. Celem niniejszej pracy jest przeprowadzenie analizy porównawczej sposobów testowania aplikacji internetowych. W ten sposób autorzy postarają się udowodnić, że testy kontraktowe pozwalają na lepsze i bardziej kompleksowe przetestowanie kodu aplikacji mikroserwisowych w systemie informatycznym w porównaniu do zwykłych testów integracyjnych.

2. Przegląd literatury

Obecnie standardem można nazwać tworzenie systemów informatycznych w architekturze mikroserwisów [1]. Jest to spowodowane skalowalnością, prostotą utrzymania, łatwością wdrażania poprawek itp. Natomiast niewątpliwym problemem tego sposobu tworzenia systemów informatycznych jest trudność testowania sposobu komunikacji [2, 3]. Obecnie standardem powoli stają się testy oparte o kontrakty komunikacji między

modułami systemu [4]. Jednocześnie, ciągle największy nacisk kładziony jest na testy jednostkowe i integracyjne, a im większa aplikacja tym bardziej rozkład testów jest zbliżony do klasycznego piramidowego znanego z aplikacji monolitycznych [5].

Obecnie popularność zyskuje budowanie aplikacji internetowych z gotowych komponentów, które zostały wcześniej przygotowane w taki sposób, by były niezależne i umożliwiały możliwie najprostsze rozwiązania programistyczne pod względem rozszerzalności kodu produkcyjnego tak, aby podłączony komponent można było z łatwością dostosować do wymagań klienta. Jest to niesamowite ułatwienie w porównaniu do archaicznych już modeli aplikacji opartych o architekturę monolityczną. Jednak, aby upewnić się o poprawności działania takiego nowego, dopiero, co podłączonego komponentu, należy do niego napisać odpowiednie testy, w taki sposób, aby po podłączeniu testy automatycznie zweryfikowały poprawność działania aplikacji w sposób automatyczny, a przynajmniej pół-automatyczny [2].

Taki trend tworzenia aplikacji z gotowych komponentów, czyli w architekturze mikroserwisowej zapanował z powodu możliwości zwinnego tworzenia komponentów do tych aplikacji, niezależności pisania jednego kodu produkcyjnego od drugiego, możliwości łatwego wdrażania aplikacji i w następstwie łatwych poprawek programistycznych a także skalowalności systemu zbudowanego z takich komponentów [1]. Takie zalety prowadzą jednak do konieczności zmiany postrzegania testowania aplikacji z zachowaniem klasycznej piramidy [3]. Przy architekturze klasycznego monolitu, testy powinny mieć rozkład prezentowany przez piramidę, jednak w architekturze mikroserwisowej powinno się skupiać się zdecydowanie na komunikacji wewnętrznej i zewnętrznej między serwisami, a testy jednostkowe zostawić na niezbędne pokrycie ścisłej logiki biznesowej aplikacji.

Nie należy jednak tutaj zaprzeczać, że testy jednostkowe dalej odgrywają kluczową rolę w procesie rozwoju oprogramowania [6]. Pozwalają one na niezależne, w pewien sposób hermetyczne, zapewnienie programisty o tym, że logika biznesowa, która zaimplementował w swojej aplikacji, jest zgodna z założeniami [7]. Tak napisane testy są swoistym zabezpieczeniem nie tylko przed błędem podczas pierwotnej implementacji wymagań, ale także podczas późniejszego rozwijania aplikacji, kiedy to inni programiści mogą upewnić się o poprawności napisanego przez siebie kodu w stosunku do już istniejącego w aplikacji [8]. Bez takich zapewnień ciężko jest tworzyć kod bez wad.

Mimo, że testy jednostkowe są tak niezaprzeczalnie ważne, to, jak zostało już wspomniane, nie gwarantują one niezawodności działania całego systemu informatycznego, szczególnie takiego, który opiera swoje działanie na komunikacji między swoimi komponentami. Uzupełnieniem testów jednostkowych o zapewnienie o poprawności komunikacji są testy integracyjne [9]. Pozwalają one przetestować przetwarzanie danych między modułami aplikacji lub między aplikacjami. Są one jednak wolniejsze od jednostkowych, bardziej kosztowne w utrzymaniu a także mniej czytelne dla programisty, co jest logiczne, jeśli weźmie się pod uwagę, że testują większe części aplikacji w porównaniu do testów jednostkowych. Istnieje jednak rodzaj testów integracyjnych, który nie jest dużo wolniejszy od testów jednostkowych, a który gwarantuje poprawność komunikacji między komponentami systemu. Są to testy kontraktowe, czyli testy swoistej umowy zawartej między 2 modułami systemu. Innymi słowy, test kontraktowy wykorzystuje zdefiniowany kontrakt, aby automatycznie zweryfikować, czy nie została złamana komunikacja między komponentami [9]. W efekcie otrzymuje się wydajny test, którego zaletą jest też możliwość wykonania po obu stronach komunikacji.

Internetowa treść sieci przekształciła się z prostych stron internetowych w złożone aplikacje. Aby zapewnić jakość oprogramowania i sprawność interfejsu użytkownika, aplikacja powinna być testowana za pomocą testów jednostkowych i integracyjnych [10]. Jednak modyfikacje aplikacji mogą prowadzić do niezamierzonych zmian wizualnych, które nie są wykrywane przez testy funkcjonalne. Można przetestować tę regresję wizualną, przechwytując stan aplikacji jako zrzut ekranu i porównując go z obrazami z poprzednich wersji aplikacji.

Można przyjąć, że jest mało testów sprawdzających wydajność kodu, a dużo testów jednostkowych sprawdzających logikę biznesową (funkcjonalność) [5]. Można to zauważyć, licząc wystąpienia w publicznych repozytoriach kodu wykorzystywanego do analizy wydajności kodu w popularnych szkieletach programistycznych służących do pisania testów w języku Java.

Obsługa różnych przeglądarek i ich różnych wersji jest jednym z głównych wyzwań dla wielu aplikacji internetowych [11,12]. Kod JavaScript uruchamiany w przeglądarce Safari może nie działać poprawnie w innych przeglądarkach, takich jak Internet Explorer, Firefox lub Google Chrome. To wyzwanie wynika z braku testów jednostkowych kodu JavaScript. Jednak za pomocą popularnych szkieletów programistycznych, takich jak Jasmine, YUI Test, QUnit i JsTestDriver [13], można skutecznie tworzyć i automatyzować testy JavaScript dla aplikacji internetowych [14].

Test Driven Development (TDD) [15] to metoda programowania, w której najpierw definiuje się testy jednostkowe dla kodu, a następnie implementuje się sam kod. Celem TDD jest zapewnienie jakości i wydajności kodu poprzez ciągłe testowanie go i dostosowywanie do wymagań. Podejście TDD polega na tym, że testowanie staje się głównym czynnikiem napędowym projektowania, dokumentacji, łatwości utrzymania i jakości kodu. W przypadku języków dynamicznych, na przykład JavaScript, gdzie nie otrzymuje się ostrzeżeń o błędach podczas kompilacji, testowanie staje się kluczowym elementem łączącym duże aplikacje.

W dzisiejszych czasach jednym z kluczowych czynników sukcesu każdej firmy jest jej widoczność w Internecie [16]. Dlatego tak ważne jest, aby każda firma posiadała odpowiednią stronę internetową, która może być np. sklepem internetowym. Projektowanie takich stron nie jest łatwe, ponieważ trudno zapewnić, że będą one działać poprawnie we wszystkich nowoczesnych przeglądarkach [17]. Również dla testerów oprogramowania stanowi to wyzwanie - muszą oni sprawdzić, czy strona działa poprawnie we wszystkich przeglądarkach. W artykule [18] autorzy skupili się na różnych narzędziach do automatyzacji testów dostępnych na rynku, zarówno tych otwartych, jak i płatnych. Porównują je pod kątem ich przydatności i wyciągają wnioski. Badanie to pokazuje, że skuteczne testowanie w różnych przeglądarkach jest możliwe dzięki wykorzystaniu narzędzi do automatyzacji.

3. Teoria testów aplikacji internetowych

Wykorzystywanie różnych rodzajów testów jest kluczowe dla prawidłowego zrozumienia procesu testowania oprogramowania. Jest wiele rodzajów testów dedykowanych różnym celom: testy jednostkowe, testy integracyjne, testy kontraktowe oraz inne testy, które są stosowane w procesie weryfikacji jakości oprogramowania.

Testy jednostkowe to rodzaj testów oprogramowania, które umożliwiają pozbycie się zależności wynikających z naturalnego sposobu pisania kodu produkcyjnego. Testy te skupiają się na testowaniu najmniejszych wydzielonych komponentów kodu, takich jak klasy w programowaniu obiektowym, w celu sprawdzenia poprawności realizacji logiki biznesowej zleconego zadania programistycznego. Testy jednostkowe są niezależne od innych komponentów systemu oraz kontekstu aplikacyjnego, co umożliwia ich szybkie wykonanie i łatwe zrozumienie. Ich wykonanie na etapie tworzenia oprogramowania pozwala na szybkie wykrycie i poprawienie błędów w kodzie, co prowadzi do oszczędności czasu i pieniędzy w dalszych fazach projektu. Ponadto, testy jednostkowe są ważne dla utrzymania jakości kodu oraz ułatwienia jego późniejszej rozbudowy, co umożliwia wprowadzenie zmian bez ryzyka powstawania błędów w innych częściach systemu.

Testy integracyjne są kluczowe dla weryfikacji poprawności interakcji pomiędzy poszczególnymi modułami lub komponentami, które tworzą całość systemu. Testy te są przeprowadzane w środowisku testowym, w którym stosowane są specjalnie przygotowane dane testowe, aby zapewnić spójność i niezmienną jakość wyników testów. W odróżnieniu od testów jednostkowych, które skupiają się na weryfikacji poprawności kodu każdego modułu oddzielnie, testy integracyjne są bardziej skomplikowane i czasochłonne. Jednakże, ich wartość jest nieoceniona, ponieważ pozwalają na wczesne wykrywanie i naprawianie błędów w interakcjach pomiędzy modułami, co minimalizuje ryzyko wystąpienia poważniejszych problemów w systemie w przyszłości. Testy integracyjne powinny być przeprowadzane systematycznie i regularnie, co znacznie zwiększa stabilność, niezawodność i wydajność systemu, a także zapewnia spójność danych i interakcji między różnymi modułami.

Testy kontraktowe są ważnym narzędziem testowania oprogramowania, które służy do zapewnienia poprawnej i spójnej interakcji pomiędzy różnymi modułami lub serwisami. Ich celem jest przetestowanie, czy kontrakty, czyli umowy między nimi, są zachowane i czy komunikacja między nimi jest zgodna z oczekiwaniami. Testy kontraktowe pozwalają na wczesne wykrycie błędów w kodzie oraz na szybsze znalezienie i rozwiązanie problemów związanych z integracją oprogramowania. Testy kontraktowe opierają się na zdefiniowaniu interfejsów API między modułami lub serwisami, a następnie na przetestowaniu, czy te interfejsy są zgodne z założeniami. W przypadku testów jednostkowych, programista testuje poszczególne funkcje lub metody w izolacji, podczas gdy testy kontraktowe uwzględniają całą sieć powiązań między różnymi modułami.

Ważne jest, aby testy kontraktowe były pisane w sposób czytelny i łatwy do zrozumienia dla innych członków zespołu. Dzięki temu testy kontraktowe mogą służyć nie tylko jako narzędzie testowania, ale również jako dokumentacja dla projektu. Testy kontraktowe są szczególnie przydatne w projektach, w których wiele zespołów pracuje nad różnymi modułami lub serwisami, a integracja między nimi jest kluczowa dla poprawnego funkcjonowania systemu jako całości. W takich projektach testy kontraktowe pozwalają na szybsze wykrycie i rozwiązanie problemów związanych z integracją.

Przetestowanie aplikacji monolitycznych zgodnie z dobrymi praktykami i nowoczesnymi technologiami może być nie tylko przyjemne, ale także satysfakcjonujące. Dzięki nowoczesnym narzędziom testowym i podejściu do testowania, deweloperzy są w stanie szybko przeprowadzić testy jednostkowe i integracyjne, co pozwala na wykrycie błędów i szybsze wprowadzenie poprawek.

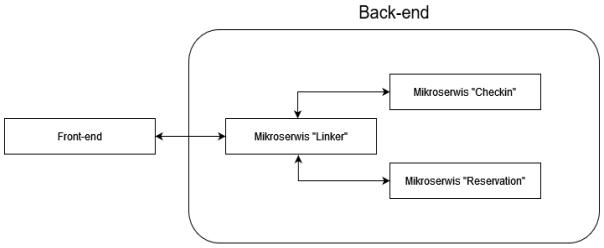
Testowanie jest także kluczowym elementem architektury mikroserwisowej. Wymaga ono podejścia, które uwzględnia testowanie integracyjne oraz testowanie jednostkowe każdego z mikroserwisów, ponieważ system składa się z wielu mniejszych usług. Testy kontraktowe są ważnym rodzajem testów w architekturze mikroserwisowej. Są one szczególnie istotne, gdy wiele mniejszych usług komunikuje się ze sobą za pomocą API.

4. Plan badań

Badanie miało na celu przeprowadzenie analizy porównawczej testowania aplikacji internetowych z uwzględnieniem różnych rodzajów testów, takich jak jednostkowe, integracyjne i kontraktowe. Jednak oprócz samego porównania wydajnościowego, skupiono się również na roli, jaką architektura aplikacji odgrywa w ilości i rodzaju testów pisanych podczas fazy rozwoju

aplikacji. Aplikacja miała na celu umożliwienie rezerwacji biletów na filmy (wybranie sali oraz miejsca na niej) oraz odprawę w kinie za pomocą kodu odprawy otrzymanego podczas rezerwacji.

W pierwszej aplikacji wykorzystano architekturę mikroserwisową, która składała się z trzech mikroserwisów: "linker", "reservation" i "checkin". Każdy z tych mikroserwisów miał architekturę Domain-Driven Design (DDD). Mikroserwis "linker" pełnił rolę pośrednika, komunikującego się zarówno z aplikacją front-end, jak i pozostałymi mikroserwisami. Pozostałe mikroserwisy, "reservation" i "checkin", nie komunikowały się bezpośrednio z zewnętrznym światem, udostępniając swoje API tylko dla "linkera". Opisana komunikację przedstawia Rysunek 1.



Rysunek 1: Graficzna reprezentacja komunikacji w aplikacji.

W drugiej aplikacji zaimplementowano tę samą funkcjonalność, ale jako aplikację monolityczną. Cała logika i funkcjonalność zostały skonsolidowane w jednym projekcie.

Przeprowadzenie porównawczej analizy tych dwóch aplikacji pozwoliło na zrozumienie wpływu architektury aplikacji na proces testowania. W przypadku aplikacji mikroserwisowej, testowanie było bardziej skomplikowane ze względu na rozproszenie logiki biznesowej i konieczność zarządzania komunikacją między mikroserwisami. Testy integracyjne miały kluczowe znaczenie, aby upewnić się, że poszczególne mikroserwisy komunikują się ze sobą poprawnie i spełniają oczekiwane wymagania.

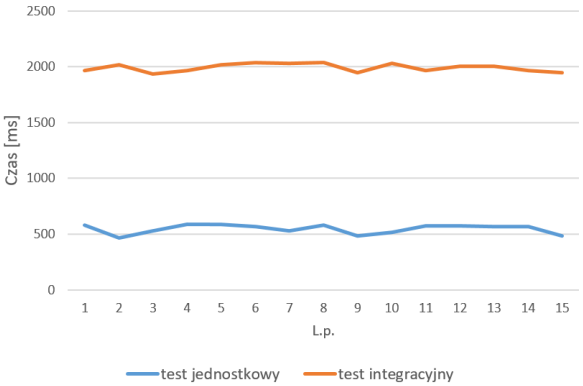
5. Analiza wyników badań

5.1. Wydajność różnych rodzajów testów

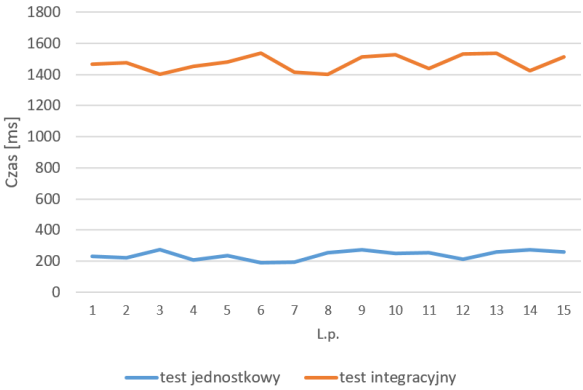
Przeprowadzona analiza dotycząca czasów wykonania różnych rodzajów testów w każdej z zaimplementowanych aplikacji w architekturze mikroserwiowej dała wyniki przedstawione graficznie na Rysunku 2.

Takie samo badanie zostało przeprowadzone w aplikacji w architekturze monolitycznej i dane przedstawione są na Rysunku 3.

W mikroserwisach stosunek czasu wykonania pojedynczego testu jednostkowego do czasu wykonania pojedynczego testu integracyjnego wynosi około 1:4, natomiast w monolicie stosunek ten oscylował w granicach 1:6. Oznaczałoby to, że w architekturze monolitycznej testy jednostkowe wykonują się szybciej, patrząc na dane testy integracyjne wykazują podobną tendencję. Wynika to oczywiście z faktu, że każdy z mikroserwisów musi przygotować kontekst frameworka Spring, skompilować kod osobno i wykonać testy w ‘świeżym’ środowisku. W aplikacji monolitycznej dzieje się to raz i wszystkie testy mogą zostać wykonane.



Rysunek 2: Graficzna reprezentacja czasów w milisekundach potrzebnych do wykonania pojedynczego testu w aplikacji w architekturze mikroserwisowej.



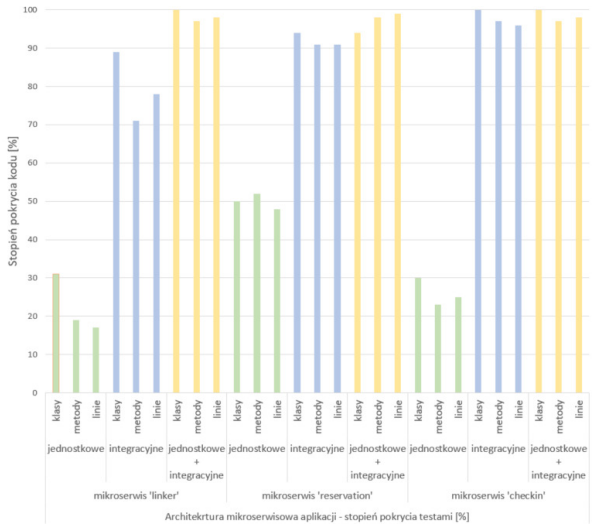
Rysunek 3: Graficzna reprezentacja czasów w milisekundach potrzebnych do wykonania pojedynczego testu w aplikacji w architekturze monolitycznej.

5.2. Kompleksowość analizy kodu

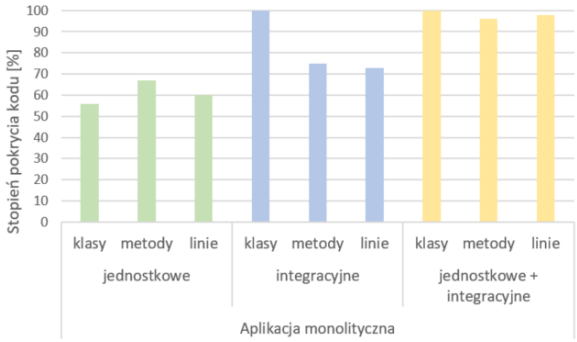
Analiza kompleksowości analizy kodu została przeprowadzona na podstawie stopnia pokrycia kodu testami. Wyniki badań przedstawiono na Rysunkach 4 oraz 5.

We wszystkich aplikacjach pokrycie kodu testami jest niemal stuprocentowe, co najlepiej pozwoliło ukazać, który rodzaj testów jest niezbędny. Mimo tego, że testy integracyjne ukazują większy stopień pokrycia kodu nie są idealne i nie można osiągnąć nimi wszystkiego. Testami jednostkowymi dałoby się oczywiście osiągnąć stuprocentowe pokrycie, jednak w tym przypadku lepiej sprawdziły się jako uzupełnienie do sprawdzania miejsc ciężko dostępnych przy testowaniu integracyjnym. Z analizy wyników można wywnioskować, że najlepszym wyjściem jest stosowanie obu rodzajów testów symultanicznie. Przeprowadzenie badań kompleksowości analizy kodu zostało wykonane przy użyciu narzędzia ‘Code

coverage' programu IntelliJ IDEA. Jest to narzędzie pokazujące miejsca, które już są pokryte testami oraz wyszczególniające miejsca, których testy na ten moment nie sprawdzają.



Rysunek 4: Graficzna reprezentacja stopnia pokrycia kodu w procentach w aplikacjach w architekturze mikroserwisowej.

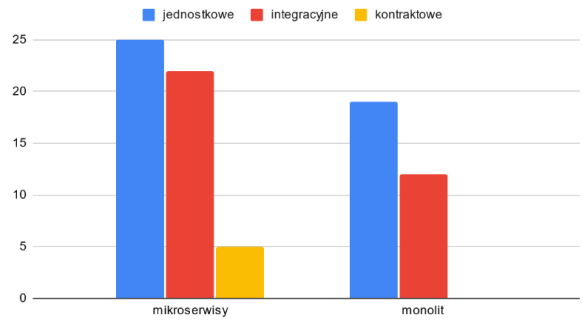


Rysunek 5: Graficzna reprezentacja stopnia pokrycia kodu w procentach w aplikacji w architekturze monolitycznej.

5.3. Zróżnicowanie testów pod względem architektury

Jak wspomniano wcześniej, realizując badania napisano aplikację dwukrotnie, raz w architekturze mikroserwisowej a raz w architekturze monolitycznej, jednak z identyczną funkcjonalnością. W ramach badań test kompleksowo przetestowano testami jednostkowymi, integracyjnymi i kontraktowymi. Następnie policzono wystąpienia każdego rodzaju testu w każdej architekturze. Wyniki przedstawia wykres na Rysunku 6.

Analizując przedstawiony wykres można zauważyć, że, mimo iż logika biznesowa w obu badanych architekturach jest identyczna, to ilość testów jest całkowicie inna. Architektura oparta o mikroserwisy ma zdecydowanie więcej testów jednostkowych i integracyjnych, a dodatkowo decydując się na tego rodzaju architekturę trzeba również wykonać testy kontraktowe, których w monolicie nie ma potrzeby pisać.



Rysunek 6: Graficzna reprezentacja ilości różnych rodzajów testach w aplikacji w dwóch różnych architekturach.

5.4. Porównanie testów kontraktowych i integracyjnych

Jako, że oba rodzaje testów - testy kontraktowe i integracyjne - służą zapewnieniu pewności w poprawności działania komunikacji między różnymi częściami systemu, aby odpowiedzieć na pytanie czy skuteczność testów kontraktowych jest większa w porównaniu z testami integracyjnymi, należy zawęzić zakres testowalności dla obu tych rodzajów testów. Można z całą pewnością stwierdzić, że zakres zastosowalności testów integracyjnych jest o wiele szerszy niż testów kontraktowych. Przykładem takiej komunikacji może być testowanie komunikacji między modułami lub między aplikacją a bazą danych. Testy kontraktowe nie sprawdzą takiego rodzaju komunikacji, nie były też tworzone z taką myślą. Zatem w celu poprawnego porównania autorzy tekstu zdecydowali się zawęzić oba rodzaje do jednego celu: sprawdzania programistycznego interfejsu aplikacji.

Podczas rozwoju systemu napisanych zostało kilka testów kontraktowych oraz kilka testów integracyjnych spełniających założenia. Oczywiście, należy mieć na uwadze, że ich realizacja następuje po obu stronach aplikacji całkowicie inaczej.

Po stronie dostawcy API test kontraktowy jest prostszy w napisaniu - wymaga tylko zdefiniowania testowych danych w pliku kontraktu. Test integracyjny natomiast zawiera logikę komunikacji, co utrudnia jego pisanie. Dodatkowo, właściwy, czyli wykonywalny, test kontraktowy jest automatycznie generowany z napisanego kontraktu i ma postać czytelną, bardzo podobną do tego, co w branży przyjęło się pisać w testach integracyjnych sprawdzających komunikację API. Natomiast test integracyjny można uruchomić w dowolnym momencie, tak jak każdy inny test, w przeciwieństwie do testu kontraktowego, który można uruchomić tylko w fazie budowania aplikacji, która musi się zakończyć, aby uzyskać rezultat wykonania wszystkich testów, i na tej podstawie, albo w logach budowania, sprawdzić czy dany test kontraktowy wykonał się pomyślnie.

Po stronie konsumenta API test kontraktowy jest o wiele łatwiejszy w utrzymaniu, nie wymaga zasobów oraz pracy nad utrzymaniem dostępności testowej usługi. Aby test integracyjny nie był zależny od testowej instancji dostarczyciela API, musi posiadać logikę, która faktycznie nie sprawdza implementacji komunikacji między mikroserwisami, a więc nie spełnia swojego podstawowego założenia. Staje się w takim wypadku testem sprawdzającym wewnętrzną komunikację aplikacji. W ten sposób powstaje luka w zabezpieczeniu systemu przed zmianami. Ma jednak przewagę nad testem kontraktowym w momencie, kiedy integracja następuje z zewnętrznym serwisem, który nie dostarcza definicji kontraktów. W tym momencie nie da się napisać testu kontraktowego, a test integracyjny jest jedynym wyjściem.

6. **Wnioski**

Testy integracyjne są zdecydowanie bardziej nastawione na kompleksową analizę funkcjonalności aplikacja w porównaniu do testów jednostkowych, które są skupione bardziej na pojedynczym uniezależnionym od kontekstu aplikacji procesie, przez co są zdecydowanie bardziej wydajne. Taki wniosek pozwala przyjąć podstawowe pytanie badawcze postawione w pracy. Architektura mikroserwisowa wymaga większego nakładu pracy związanej z testowaniem, ale pozwala na większą modularność i elastyczność w rozwijaniu aplikacji. Natomiast architektura monolityczna może być bardziej efektywna pod względem czasu wykonania testów, ale może ograniczać skalowalność i separację odpowiedzialności w systemie. Ostateczny wybór architektury zależy od konkretnych wymagań i preferencji projektowych. Z tego powodu można stwierdzić, że wykorzystana w systemie architektura wpływa dosyć znacznie na dywersyfikację rodzajów testów wykorzystanych w komponentach systemu. Jednocześnie, mając tylko na uwadze integrację między identycznymi warstwami dostępu, testy kontraktowe okazały się lepszym wyborem, jeśli tylko mogły być zastosowane. Stosowalność tego rodzaju testów jest ograniczona jednak dosyć mocno od integracyjnych, jednak porównując tylko zakres wspólnej zastosowalności testy kontraktowe okazują się lepszym wyborem.

Literatura

[1] J. P. Sotomayor, S. C. Allala, P. Alt, J. Phillips, T. M. King, P. J. Clarke, Comparison of runtime testing tools for microservices, Annual Computer Software and Applications Conference (COMPSAC) 43(2) (2019) 356-361.

[2] H. G. Gross, C. Atkinson, F. Barbier, Component integration through built-in contract testing, Component-based software quality, Lecture Notes in Computer 2693 (2003) 159-183.

[3] H. Fischer, Testing in microservice systems: a repository mining study on open-source systems using contract testing, GUPEA, Gothenburg, 2021.

[4] F. Selleby, Creating a Framework for Consumer-Driven Contract Testing of Java APIs, Bachelor's degree, Linköping University, Linköping, 2018.

[5] P. Stefan, V. Horky, L. Bulej, P. Tuma, Unit testing performance in java projects: Are we there yet?, Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (2017) 401-412.

[6] R. Pawlak, Testowanie oprogramowania. Podręcznik dla początkujących, Helion, Gliwice, 2014.

[7] R. Dahiya, A. Shahid, Importance of Manual and Automation Testing, CS & IT Conference Proceedings 9(17) (2019) 6-13.

[8] G. Fink, F. Ido, JavaScript Unit Testing, Pro Single Page Application Development: Using Backbone, JS and ASP. Net, Apress, Berkeley, 2014.

[9] D. Raghuvanshi, Introduction to Software Testing, International Journal of Trend in Scientific Research and Development (IJTSRD) 4(3) (2020) 797-800.

[10] M. Vesikkala, Visual regression testing for web applications, Master's thesis, Aalto University, Espoo, 2014.

[11] H. Achkar, Model Based Testing of Web Applications, The Science Technicians' Association of New Zealand Conference (2010) 11-19.

[12] Z. Qian, M. Huaikou, Z. Hongwei, a practical web testing model for web application testing, 2007 third international IEEE conference on signal-image technologies and internet-based system (2007) 434-441.

[13] N. Antunes, M. Vieira, Penetration testing for web services, Computer 47(2) (2013) 30-36.

[14] H. Saleh, JavaScript Unit Testing, Packt Publishing, Mumbai 2013.

[15] T. Kleivane, Unit Testing with TDD in JavaScript, Master's thesis, Institutt for datateknikk og informasjonsvitenskap, Trondheim, 2011.

[16] B. Kaalra, and K. Gowthaman, Cross Browser Testing Using Automated Test Tools, International Journal of Advanced Studies in Computers, Science and Engineering 3(10) (2014) 7-13.

[17] P. Tonella, R. Filippo, Web Testing: a Roadmap for the Empirical Research, Seventh IEEE International Symposium on Web Site Evolution (2005) 63-70.

[18] H. V. Gamido, M. V. Gamido, Comparative review of the features of automated software testing tools, International Journal of Electrical and Computer Engineering (IJECE) 9(5) (2019) 4473-4480.