Politechnika Poznańska

Wydział Informatyki

Instytut Informatyki

Automatyczne testowanie interfejsów programistycznych typu RESTful

Jakub Matjanowski

Promotor

dr inż. Bartosz Walter

Poznań, 2016 r.

Streszczenie

W dobie rosnącego zapotrzebowania na skalowalne systemy informatyczne, coraz częściej stosowane są podejścia architektoniczne pozwalające na zapewnienie możliwości rozproszenia obliczeń i operacji. Popularnym ostatnimi czasy podejściem jest wyizolowanie wielu niewielkich komponentów zwanych mikrousługami, spełniających zasadę pojedynczej odpowiedzialności, które komunikując się ze sobą w środowisku sieciowym, wspólnie realizują cel systemu. Potencjalna mnogość takich komponentów wymaga od zespołu programistów wykonywania dodatkowych testów, które zweryfikują czy interfejsy wszystkich komponentów działają zgodnie z założeniami. Niniejsza praca ma na celu zamodelowanie i stworzenie narzędzia, które wygeneruje kod testów jednostkowych pokrywających połączenia między mikrousługami na podstawie formalnej specyfikacji.

Abstract

Nowadays, when we observe increasing demand for scalable systems, there are many cases of using achitectural aproaches which enable to provide posibilty to distribute calculations and operations. A proposal which gets more and more popular throughout last months is to isolate many small components, called microservices, which follows single responsibility principle. Those components communicate with each other in network environment to achive a system goal. Potential lots of such components require from development teams performing additional tests that verify whether interfaces of each service work under the dictates of initial assumptions. The present work has a purpose to model and create a tool, which generates unit tests code covering microservices boundaries, basing‘ on formal specification.

Spis treści

Rozdział 1

Wstęp

W ciągu ostatnich paru lat, w informatyce można zaobserwować wyjątkowo szybki rozwój tematyki przetwarzania w chmurze obliczeniowej (ang. Cloud Computing), a co za tym idzie, zainteresowanie stylami i wzorcami architektonicznymi, które w efektywny sposób są w stanie wykorzystać potencjał aplikacji rozproszonych. Takie podejście prezentuje między innymi idea mikro-serwisów, czyli małych, niezależnych komponentów komunikujących się między sobą prostymi protokołami sieciowymi[5]. Najczęściej stosowanym rozwiązaniem jest używanie komunikacji HTTP w konwencji REST, gdyż zostało to zarekomendowane przez prekursorów idei rozproszonych mikrousług - Martina Fowlera oraz Jamesa Levisa[2].

Oczywistym jest, że wraz ze wzrostem skomplikowania i rozmiaru systemu bazującego na architekturze mikroserwisowej, powstaje coraz więcej usług, które komunikują się ze sobą nieustannie. Nieodzownym elementem testowania takiego oprogramowania jest więc sprawdzanie punktów styku między usługami (and. end points) pod kątem ich poprawności i zgodności ze specyfikacją. Zgodnie z wiedzą autora, w systemach zgodnych z architekturą mikroserwisową, testy jednostkowe skupiają się na funkcjonalnościach każdej z usług, a walidacja poprawności zwracania i wymiany danych jest często ignorowana. W środowisku pracy zorganizowanym wokół mikroserwisów krytyczna jest komunikacja między zespołami pracującymi nad połączonymi mikrousługami. Taka komunikacja ma na celu uzgodnienie sposobów integracji wielu komponentów w postaci mikrousług, a jej efektem jest ustalenie i spisanie dokumentacji technicznej, będącej podstawą pisania nowych konsumentów danej usługi. Specyfikacja taka powinna być spójna i logiczna, dlatego często stosuje się języki formalne, które w swojej semantyce wykluczają lub minimalizują ryzyko sprzeczności lub niedomówień.

1.1 Cel pracy

Niniejsza praca ma na celu zaprezentowanie modelu narzędzia do generowania testów jednostkowych połączeń między mikrousługami na podstawie ich specyfikacji zapisanej w języku RAML, zaimplementowanie tego modelu w języku C#, a także

dokonanie weryfikacji istniejącego systemu opartego o komunikację REST w celu oceny skuteczności zaproponowanego rozwiązania.

1.2 Struktura pracy

W rozdziale pierwszym zaprezentowany jest zarys obecnej sytuacji w dziedzinie rozwoju oprogramowania oraz problemy jakie stawia przed nimi szybki rozwój aplikacji rozproszonych. Został tam też zamieszczony opis zakresu pracy, która jest odpowiedzią na problemy stające przed zespołami rozwijającymi nowoczesne systemy informatyczne. Częścią tego rozdziału jest również niniejsze przedstawienienie struktury pracy.

Rozdział drugi ma na celu wprowadzenie czytelnika w tematykę postawionego problemu oraz opisanie głównych pojęć związanych z tematyką pracy. Przeanalizowany zostanie styl architektoniczny oparty o mikrousługi, uwidaczniając przyczyny jego powstania, zasady działania oraz najbardziej znaczące zalety. Nieuniknione będzie również zidentyfikowanie słabych stron i procesów, które według publikacji, mogłyby je redukować. Pamiętając o komunikacji mikrousług, opisana zostanie najczęściej używana metoda ich komunikacji czyli interfejsy REST z użyciem protokołu HTTP. W dalszym ciągu rozdziału nastąpi przybliżenie potrzeby dokumentacji interfejsów programistycznych (*API*), korzyści z nich płynących oraz standardów. Ta sekcja zostanie opisana na przykładzie języka RAML w wersji 1.0. W końcowej części tego rozdziału zaprezentowany zostanie koncept testowania jednostkowego, zasady pisania takich testów oraz korzyści z nich płynące.

Rozdział trzeci zawiera opis proponowanego przez autora rozwiązania. Począwszy od modelu tworzonego narzędzia, poprzez projekt architektury zaprezentowane zostanie rozwiązanie problemów związanych z rosnącą popularnością komunikacji międzyusługowej. Rozdział uwzględnia rówież zmiany kultury pracy i dyscypliny deweloperskiej, która jest niezbęna przy stosowaniu takich rozwiązań procesowych. W końcowej części rozdziału spojrzymy na rezultaty implementacji, przedstawiając główne funkcjonalności.

W rozdziale czwartym skupimy się na weryfikacji proponowanego rozwiązania. Przybliżony zostanie istniejący system oparty o komunikację sieciową HTTP/REST. Następnie opisana zostanie metodologia przeprowadzania testów, czyli kryteria testowania oraz sposób weryfikowania poprawności działania stworzonego rozwiązania. W ostatniej sekcji tego rozdziału zaprezentowane zostaną otrzymane rezultaty testów i weryfikacji.

Ostatni rozdział zawiera anailizę rezulatów testów wyszczególnonych w rozdziale czwartym‘, a następnie kreślone zostaną wyciągnięte wnioski na podstawie analizy wyników. W dalszej części, wybiegając w przyszłość, zidentyfikowane zostaną najważniejsze ścieżki udoskonalenia narzędzia nakreślając tym samym kierunki rozwoju całej dziedziny poruszanej w niniejszej pracy. W ostatniej sekcji całej pracy zostanie

ona podsumowana.

Rozdział 2

Wprowadzenie

2.1 Architektura oparta o mikrousługi

2.1.1 Architektura zorientowana na usługi

Koncept architektury mikrousług, mimo, że w ostatnich kilkunastu miesiącach stał się szeroko omawianym tematem, powszechnie uważanym za innowację, to rozszerza tylko podejście, które istnieje od dawna jako SOA (ang. *Service-Oriented Architecture*). SOA to architektura łącząca wiele komponentów, których komunikacja odbywa się nie w granicach procesu, ale również przez sieć komputerową. Ściślej mówiąc, komunikacja powinna niejako ignorować fakt fizycznej lokalizacji procesu wywołującego jak i wywoływanego. System w ten sam sposób powinien zachować się zarówno uruchomiony na jednej maszynie jak i na wielu maszynach mieszczących się na różnych kontynentach.

Taka granulacja umożliwia lepsze wykorzystanie zasobów, poprzez dedykowanie ich konkretnym komponentom w zależności od nakładu pracy, który jest wykonywany. Stosuje się podejście tworzenia wielu drobnych (*ang. fine-grained*) aplikacji przeznaczonych do jednego zadania.[4] Pozwala to na skalowanie tylko obciążonej części przetwarzania, aby udrożnić wąskie gardło systemu.

2.1.2 Domain Driven Design jako źródło mikrousług

Procesem rozwoju oprogramowania rządzi szereg podstawowych zasad, które umożliwiają skuteczne pisanie, a przede wszystkich utrzymywanie istniejącego kodu. Jedną z najważniejszch w ocenie autora reguł jest zasada pojedynczej odpowiedzialności, stanowiąca o tym, że każda funkcjonalność powinna być realizowana przez jedną wydzieloną część kodu i tylko przez nią. Na poziomie programowania zorientowanego obiektowo jest to klasa, na poziomie projektu jest to pakiet, a na poziomie architektury może być to komponent. Koncepcja projektowania aplikacji w oparciu o tę zasadę została nazwana zasadą projektowania sterowanego dziedziną (ang. *Domain*

Driven Design)[1] i ostatnio bardzo często spotykanym paradygmatem prezentowania rzeczywistego świata w kodzie źródłowym.

Według tego podejścia, cały proces wytwarzania oprogramowania, począwszy od zbieranie wymagań, przez analizę, projektowanie, implementację, aż po testowanie i wdrożenie powinno być oparte o wiedzę na temat wycinka rzeczywistości, który mapowany jest na oprogramowanie w celu ułatwienia lub przyspieszenia rozwiązywania problemów z nim związanych.

2.1.3 Zasada autonomiczności jako źródło problemów

Tworzenie mikrousług zgodnie z zaleceniami twórców tego konceptu wymaga również zmian w strukturach organizacji, która wytwarza system. Zmiana taka powinna polegać na zorganizowaniu całych zespołów wokół każdej usługi. Oczywiście każdy zespół może realizować prace związane z wieloma usługami, ale ważne jest aby jedna usługa była rozwijana i ustrzymywana przez jeden zespół. Podejście takie wymaga stworzenia kilkuosobowych zespołów, w których znajdują się graficy, testerzy, projektanci i programiści. Zespół ten sam decyduje, kiedy wdroży nową wersję swojej usługi.

Problem pojawia się w momencie, kiedy niezależny zespół wprowadza zmiany, które wymagają zmian również po stronie konsumentów. Są to tzw. *zmiany przełomowe[3].* Podczas projektowania interfejsów należy statrać się jak najrzadziej wykonywać takie zmiany. Wymaga to przede wszytkim bardzo dokładnej analizy dziedziny problemu, jaki rozwiązuje dana usługa, aby uwzględnić kierunki, w których może się ona rozwijać. Unikanie zmian nie zawsze będzie możliwe, ale nie każda z nich jest zmianą przełomową. Np. dodanie pola do zwracanego obiektu nie powinno zmienić zachowania konsumanta. W tym miejscu należy też wspomnieć o tym, aby konsumentów implementować w sposób „odporny" na zmiany, które nie są przełomowe.

Niezależnie rozwijane komponenty mimo niezaprzeczalnych korzyści jakie niosą, mają też wady i z pewnością jest to dbanie o kompatybilność z oprogramowaniem z nich korzystającym.

2.2 Komunikacja między usługami

NOTE: Aby ta komunikacja była uporządkowana, warto zastosować konwencję, która definiowała będzie sposób odwoływania się do zasobów

2.3 Język RAML

2.4 Testowanie jednostkowe

[Samo przepisywanie kodu na mikrousługi jest wystarczająco trudny więc warto zaoszczędzić czas na testeowaniu] [S. G. Saez, V. Andrikopoulos, F. Wessling, and C. C. Marquezan, "Cloud Adaptation and Application (Re-)Distribution: Bridging the Two Perspectives," in 2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations. IEEE, Sep. 2014, pp. 163–172. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6975357]

Rozdział 3

Proponowane rozwiązanie

3.1 Model rozwiązania

3.2 Opis architektury

3.3 Szczegóły implementacyjne

3.4 Prezentacja narzędzia

Rozdział 4

Weryfikacja i przeprowadzone testy

4.1 Testowany system

4.2 Metodologia przeprowadzonych testów

4.3 Otrzymane wyniki i rezultaty

Rozdział 5

Wnioski i podsumowanie

5.1 Analiza rezultatów testów

5.2 Wnioski

5.3 Kierunki rozwoju

5.4 Podsumowanie

Bibliografia

[1] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[2] J. Lewis M. Fowler. Microservices - a definition of this new architectural term. http://martinfowler.com/articles/microservices.html, mar 2014. Accessed: 2016-04-25.

[3] Sam Newman. *Building Microservices*. Ó'Reilly Media, Inc.", 2015.

[4] N. Tanković, N. Bogunović, T. G. Grbac, and M. Žagar. Analyzing incoming workload in cloud business services. In *Software, Telecommunications and Computer Networks (SoftCOM), 2015 23rd International Conference on*, pages 300–304, Sept 2015.

[5] J. Thönes. Microservices. *IEEE Software*, 32(1):116–116, Jan 2015.