Rozdział 1

Wnioski i podsumowanie

1.1 Analiza rezultatów testów

1.1.1 Pokrycie API przez wygenerowane testy

Testy wygenerowane przez prototyp pokryły znaczącą część informacji niesionych przez specyfikację RAML. Zaliczyć do nich można:

- Weryfikacja kodu odpowiedzi zapytania HTTP,
- Weryfikacja formatu odpowiedzi (wszystkie anazlizowane w formacie *application/json*),
- Weryfikacja struktury zwracanego obiektu z jego schematem,
- Weryfikacja konkretnej odpowiedzi na podstawie parametrów zapytań zdefiniowanych w specyfikacji RAML w postacji pól opcjonalnych *Annotations,*
- Obecność nagłówków i ich zgodność z oczekiwanymi wartościami.

Wszystkie te testy były generowane pod warunkiem istnienia w specyfikacji informacji na ich temat.

1.1.2 Niedeterminizm testów

Podczas wykonywania testów okazało się, że są one niedeterministyczne, co wymagało rzetelnej weryfikacji ich poprawności. Testy, istotnie, były wygenerowane poprawnie, ale nie uwzględniały zmieniającego się stanu aplikacji testowanej podczas wykonywania testów.

Przypadek, który został zidentyfikowany dotyczył trzech testów, które w zależności od kolejności w jakiej zostały wykonane dawały różne rezultaty. Testy te wykorzystywały następujące oparacje:

- polubienie zdjęcia (+),
- usunięcie polubienia zdjęcia (-),
- wyświetlenie wszystkich polubień danego zdjęcia (?).

Wspomnieć należy, że raz dodane zdjęcie, przy kolejnym dodaniu zwróci wynik inny niż oczekiwany. Podobne zachowanie zaobserwujemy usuwając już usunięte polubienie.

Testy podczas sześciokrotnej próby uruchomienia ich wszystkich razem wykonywały się w następujących sekwencjach:

Iteracja	Kolejność	Ilość błędów
1	+ ? -	0
2	? + -	1
3	? - +	2
4	+ - ?	2
5	- + ?	1
6	- ? +	1

W pierwszej iteracji dodajemy nowe polubienie, następnie weryfikujemy czy ono istnieje, a potem je usuwamy. W drugiej iteracji sprawdzamy, czy polubienie istnieje, ale zostało usunięte w iteracji 1., więc ten test złamie asercję. Następnie z sukcesem dodajemy i usuwamy polubienie. W trzeciej iteracji najpierw sprzewdzano wynik zapytania o polubienie, a następnie je usuwano - obie te operacje wygenerowały błąd. Dodanie się powiodło. Idąc tym schematem w iteracji czwartej wystąpiły dwa błędy a w piątej i szóstej po jednym.

Niedeterminizm jest spowodowany błędnym założeniem o stałym stanie systemu.

1.2 Wnioski

Prototyp mimo swojej prostoty, był w stanie wygenerować 27 w pełni poprawnych testów dla specyfikacji obejmującej jedynie 7 typów zasobów. Udało się stworzyć w pełni automatyczne narzędzie, które oprócz generacji, kompiluje i uruchamia testy bez jakiejkolwiek ingerencji użytkownika lub programisty. W znaczącym stopniu może usprawnić to proces weryfikacji i kontroli istniejących systemów wyposażonych w interfejsy programistycze oparte o wzorzec REST. Stworzony prototyp może być uruchamiany cyklicznie lub jako element procesu ciągłej integracji i ciągłego dostarczania oprogramowania. Wydaje się, że w znakomitej większości prototyp spełnił oczekiwania i okazał się bardzo obiecującym narzędziem w codziennym procesie wytwarzania oprogramowania.

Weryfikacja rozwiązania przy użyciu rzeczywistego systemu, obnażyło jednak jeden ważny błąd koncepcyjny. Zaproponowane rozwiązanie oparte było o założenie, że stan systemu jest zawsze taki sam, czyli po wykonaniu testowanej operacji, wróci do stanu sprzed uruchomienia testu.

Ponadto testowanie systemu w środowisku niewyizolowanym dopuszcza ingerencję osób trzecich w stan systemu. W środowisku produkcyjnym, jakie zostało przetestowane, zmiany zbioru komentarzy lub ilości polubień są na porządku dziennym, więc cykliczna jego weryfikacjia przy stałych założeniach stanu początkowego jest bezcelowe.

Powyższe problemu nie eliminują jednak zaproponowanego rozwiązania, ale wymagają pewnych modyfikacji.

1.2.1 Odpowiedzi na problemy

1.2.1.1 Zmieniający się stan aplikacji

Jak wykazano podczas testów narzędzia, problemy są powodowane głównie przez to, że operacje na tym samym zbiorze danych wykonywane są w losowej kolejności. Utrzymanie stałej kolejności wykonywania tych operacji, całkowicie eleminuje problem niedeterminizmu, a ponadto dostarcza dodatkowej weryfikacji stanu aplikacji po wykonaniu konkretnej ścieżki. Definicje takich ścieżek można opisać na dwa sposoby:

- Wyszczególnienie kolejnych kroków wraz z opisanym oczekiwanym stanem po wykonaniu każdego z kroków,
- Dobranie par operacji komplementarnych, czyli takich które się wzajemnie znoszą i wykonywanie ich zawsze w parach.

Dodanie do narzędzia takiej funkcjonalności pomagałoby zbadzć, nie tylko bezpośrednią odpowiedź serwera na konkretne żądanie, ale również zweryfikować logikę biznesową aplikacji (np. polubienie dodane w jednym momencie, umożliwia jego usunięcie w kolejnym żądaniu). W tym miejscu należy pamiętać również o częstej cesze systemów rozproszonych, które są często budowane w architekturze mikrousługowej, a mianowicie o spójności ewentualnej (*ang. eventual consistency*)[**?**]. Oznacza ona, że po wykonaniu pewnej operacji na systemie, nie możemy oczekiwać, że ta zmiana zostanie zapisana natychniast, a pozytywna odpowiedź z serwera oznacza tylko przyjędzie zadania do realizacji. Ta kwestia może zostać rozwiazana za pomocą nagłówków, a dokładniej nagłówka *Cache-Control*. Jego wartość powinna zawierać czas w jakim system gwarantuje zapisanie w systemie żądania, a wartość ta powinna wynikać bezpośrednio z umowy o gwarantowanym poziomie świadczenia usług (*ang. Service Level Agreement*).

1.2.1.2 Wpływ osób trzecich na środowisko produkcyjne. Wpływ testów na stan systemu.

Istotnie, zarówno wykonywanie testów jak i operacje użytkowników na działającym systemie wprowadzają rozbieżności, między założeniami o stanie systemu dla testów, a faktycznym jego stanem. Oczywistym rozwiązaniem w tej sytuacji jest uruchamianie testów w środowisku kontrolowanym (np. testowym). Eliminuje to jednak ważną wartość, jaką jest monitorowanie działającego systemu produkcyjnego. Aby rozwiazać ten problem, można podzielić testy na dwie grupy: inwazycjną i nieinwazycją.

Inwazyjna grupa testów rzeczywiście dokonuje zmian w systemie lub weryfikuje szczegółowo stan systemu (np. dokładna wartość obiektów zwracanych podczas zapytań). Są to zapytania, których poprawność może zostać zweryfikowana tylko w ściśle kontrolowanych systemach, gdyż wynikają z innych zapytań, które wykonano wcześniej. Grupa testów inwazyjnych, może zostać wykonana tylko na środowisku testowym.

Drugą grupą, są testy nieinwazyjne. Weryfikacja poprawności zapytań w tych testach nie polega na porównywaniu dokładnych odpowiedzi z wartością oczekiwaną, ale na przeanalizowaniu struktury odpowiedzi lub wzorców nagłówków. Rzeczywisty stan systemu nie powinien mieć wpływu na wynik testów, a sprawdzane są jedynie mechanizmy, z których korzysta cały system. Do grupy testów nieinwazycjnych można zaliczyć również testy wydajnościowe, które zweryfikują np. czas odpowiedzi serwera. Testy nieinwazyjne mogą zostać uruchomione podczas działania aplikacji w środowisku produkcyjnym, cykliczne z nich korzystanie może pomóc wykryć niepożądane rozbieżności między oczkiwaną, a rzeczywistą jakością pracy serwisów.

1.3 Kierunki rozwoju

Prototyp, mimo, że spełnił wyznaczone cele, może zostać rozwinięty. Podczas opracowywania modelu rozwiązania oraz przeprowadzania testów, ujawniło się wiele kwestii, które nie były przedmiotem pierwotnej analizy problemu, a mogą zostać rozwiązane za pomocą niewielkich modyfikacji w narzędziu stworzonym na potrzeby tej pracy. Zdaniem autora, wyszczególnić można trzy główne kierunki rozwoju:

- wsparcie dla innych typów specyfikacji formalnych (Swagger, BluePrint) oraz wsparcie dla innych języków programowania (Java, Python, C++, JavaScript, Go)
- rozwinięcie sposobu przeprowadzania testów (ścieżki wywołań, dobieranie par operacji komplementarnych, autoryzacja użytkowników OAuth 2.0, dodatki dla narzędzi ciągłej integracji - Jenkins/TeamCity)
- zaimplementowanie nowych kryteriów weryfikacji (czasa odpowiedzi, prędność przesyłu danych, wartość danych według przedziałów)

Dzięki temu, że rozwiązanie zostało zaprojektowanie modularnie, wszelkie zmiany opisane wyżej mogą zostać wprowadzone względnie niewielkim kosztem.

Potencjalnie pożądaną funkcjonalnością jest raportowanie przebiegu testów w postaci czytelnych wykresów oraz okresowych zestawień. Takie dane byłyby z pewnością wartościową informacją dla manadżerów oraz dyrektorów techniczych, sprawujących pieczę nad dużymi rozproszonymi systemami.

1.4 Podsumowanie

Niniejsza praca miała za zadanie uświadomić czytelnika, że w obliczu zmieniających się trendów w zakresie rozwoju oprogramowania, należy rzetelnie zidentyfikować problemy związane z tą zmianą a następnie opracować rozwiązania, które zminimalizują skutki tych problemów.

Jedną z trudności z coraz bardziej popularnymi modularnymi systemami webowymi (zwanymi mikrousługami) jest trudność w ich testowaniu oraz częste zmiany w specyfikacji interfejsów programistycznych. Opisano koncept mikrousług, wyjaśniono podstawy działania interfejsów typu REST, następnie podkreślono wagę spójnej dokumentacji technicznej oprogramownia, a potem

odniesiono się do kwesti testowania, kładąc nacisk na opis właściwego przygotowywania testów interfejsów programistycznych typu REST.

Aby ułatwić weryfikację spójności interfejsów programistycznych ze specyfikacją, zaprojektowano i stworzono prototyp narzędzia, które na podstawie dokumentacji interfejsów programistycznych API generuje definicje testów, a następnie na ich podstawie tworzy kod źródłowy testów oprogramowania w języku C# dla biblioteki xUnit.

W końcowej części zaprezentowano rezultaty testów systemu Instagram za pomocą stworzonego narzędzia. Zidentyfikowano kilka nieoczekiwanych problemów tego rozwiązania, oraz zaproponowano szereg usprawnień oraz rozwinięć.

Mimo, że narzędzie nie dostarcza w obecnej formie kompleksowego rozwiązania, to stanowi podwaliny pod zaawansowane oprogramowanie, które może zostać uruchamiane w celu utrzymywania jak najwyższej jakości oprogramowania.