

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

1	Wstęp	6
1.1	Cel pracy	6
1.2	Struktura pracy	7
2	Wprowadzenie	9
2.1	Architektura oparta o mikrousługi	9
2.1.1	Architektura zorientowana na usługi	9
2.1.2	Domain Driven Design jako źródło mikrousług	9
2.1.3	Zasada autonomiczności jako źródło problemów	10
2.2	Komunikacja między usługami	10
2.2.1	Podział monolitu	10
2.2.2	Projektowanie interfejsów REST	11
2.2.3	Połączenia między usługami	13
2.3	Dokumentacja oprogramowania	14
2.3.1	Definicja terminu oraz rola dokumentacji	14
2.3.2	Formalna specyfikacja usług sieciowych	16
2.3.3	Koszty rozwoju i utrzymania	16
2.3.4	RAML	16
2.4	Testowanie	18
2.4.1	Testowanie funkcjonalne	18
2.4.2	Przedmiot testowania w przypadku interfejsów REST	19
2.4.3	Ciągła integracja	19
3	Proponowane rozwiązanie	21
3.1	Model rozwiązania	21
3.2	Opis działania aplikacji	22
3.3	Szczegóły implementacyjne	23
3.3.1	Generacja kodu pośredniego	23
3.3.2	Generacja kodu testów	25
3.3.3	Uruchamianie testów	27

4	Weryfikacja i przeprowadzone testy	29
4.1	Testowany system	29
4.2	Metodologia przeprowadzonych testów	30
4.2.1	Przygotowanie do wykonania testów	30
4.2.1.1	Instagram	30
4.2.1.2	Dokumentacja techniczna	31
4.2.2	Przeprowadzenie testów.	31
4.3	Otrzymane wyniki i rezultaty	33
5	Wnioski i podsumowanie	36
5.1	Analiza rezultatów testów	36
5.1.1	Pokrycie API przez wygenerowane testy.	36
5.1.2	Niedeterminizm testów	36
5.2	Wnioski.	37
5.2.1	Odpowiedzi na problemy	38
5.2.1.1	Zmieniający się stan aplikacji	38
5.2.1.2	Wpływ osób trzecich na środowisko produkcyjne. Wpływ testów na stan systemu.	38
5.3	Kierunki rozwoju.	39
5.4	Podsumowanie.	40
	Bibliografia	42

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[30]. Najczęściej stosowanym rozwiązaniem jest używanie komunikacji HTTP w konwencji REST, gdyż zostało to zarekomendowane przez prekursorów idei rozproszonych mikrousług - Martina Fowlera oraz Jamesa Levisa[17].

Oczywistym jest, że wraz ze wzrostem skomplikowania i rozmiaru systemu bazującego na architekturze mikroservisowej, powstaje coraz więcej usług, które komunikują się ze sobą nieustannie. Nieodzownym elementem testowania takiego oprogramowania jest więc sprawdzanie punktów styku między usługami (ang. end points) pod kątem ich poprawności i zgodności ze specyfikacją. Zgodnie z wiedzą autora, w systemach zgodnych z architekturą mikroservisową, testy jednostkowe skupiają się na funkcjonalnościach każdej z usług, a walidacja poprawności zwracania i wymiany danych jest często ignorowana. W środowisku pracy zorganizowanym wokół mikroservisów 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 nieдомówień.

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 stojące przed zespołami rozwijającymi nowoczesne systemy informatyczne. Częścią tego rozdziału jest również niniejsze przedstawienie 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 również zmiany kultury pracy i dyscypliny deweloperskiej, która jest niezbędna 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 analizę rezultatów testów wyszczególnionych 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.

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.[29] 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 wszystkim utrzymywanie istniejącego kodu. Jedną z najważniejszych 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 dziedziny (ang.

Domain Driven Design)[11] i ostatnio bardzo często spotykanym paradygmatem prezentowania rzeczywistego świata w kodzie źródłowym.

Według tego podejścia, cały proces wytwarzania oprogramowania, począwszy od zbierania 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 utrzymywana przez jeden zespół. Podejście takie wymaga stworzenia kiluosobowych 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*[22]. Podczas projektowania interfejsów należy starać się jak najrzadziej wykonywać takie zmiany. Wymaga to przede wszystkim 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 konsumenta. 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

2.2.1 Podział monolitu

Krytyczną decyzją podczas projektowania architektury opartej o mikrousługi jest wybór punktów podziału systemu na związane konteksty (*ang. bounded context*)[11, 17] oraz sposobu współdzielenia tych jednostek.

Według Sama Newmana, rozróżnia się dwie techniki dekompozycji systemu monolitycznego. Pierwszym są pakiety udostępniane w postaci bibliotek współdzielo-

nych, a drugim moduły, które mogą mieć wydzielony stos technologiczny. Każde z tych rozwiązań ma zarówno szereg zalet jak i wad, co sprawia, że powinny być zastosowane do różnych celów.

Biblioteki współdzielone mogą zostać wykorzystane w znakomitej większości języków i technologii, np. C#, Java, lub PHP. Dzielą one kod w logiczne jednostki, które mogą być importowane do wielu projektów. Takie biblioteki mogą być dostarczane zarówno przez zewnętrzne instytucje jak i udostępniane wewnątrz firmy. Mimo łatwości w używaniu takich bibliotek, należy wziąć pod uwagę zalety z których rezygnujemy. Najważniejszą z nich jest oczywiście heterogeniczność - biblioteki zazwyczaj mogą być uruchomione tylko przez aplikację skompilowaną dla tej samej platformy i w tym samym języku. Oprócz tego, aktualizacja takiej biblioteki wymaga ponownej kompilacji i instalacji wszystkich aplikacji z niej korzystających. Skalowanie takich pojedynczych komponentów również nie jest możliwe, bez duplikacji całych korzystających z niej aplikacji. Najlepszym zastosowaniem dla tego typu komponentów są współdzielone elementy systemu, które nie są kluczowe z punktu widzenia domeny, np. zbiory funkcji matematycznych lub klasy pomocnicze dla refleksji.[22]

Drugim wariantem jest wyekstrahowanie modułów. Moduł jest fragmentem kodu, który realizować powinien proces określony w wymaganiach biznesowych systemu. Jest to wydzielona część domeny, która realizuje określone przez nią zadania. W szczególnym przypadku powinny posiadać własną warstwę dostępu do danych lub warstwę prezentacji[22]. Ważne jest, aby nie unifikować pojęcia związanego kontekstu z modułem[11]. Związany kontekst jest raczej zbiorem modułów, istniejących w tym samym celu w sensie domeny. Dobrą praktyką jest więc tworzenie kilku warstw aplikacji w osobnych modułach zawierających się w tej samej przestrzeni nazw, więc wydzielonym kontekście[11].

Drugie z tych podejść jest bardziej uwarunkowane różnicami w procesach biznesowych mapowanych na oprogramowanie, więc to taki podział tworzy podwaliny pod wydzielenie pełnoprawnych mikrousług.

2.2.2 Projektowanie interfejsów REST

REST (*ang. Representational State Transfer*) to styl architektoniczny przeznaczony dla rozproszonych systemów opartych o hipermedia. Ma on na celu ułatwienie dostępu do zasobów (*ang. resources*) dla systemów w modelu typu klient-serwer, ze znaczną liczbą aktywnych aplikacji klienckich. Jak wspomniano w rozdziale 2.1.1, do komunikacji między mikrousługami powinno stosować się lekkich i szybkich rozwiązań, więc podejście REST, które jest sześciokrotnie szybsze niż starsze protokoły SOAP[31], nadają się idealnie do tego celu. Autor konceptu REST określił pięć wymagań, które umożliwiają bezpieczny dostęp do zasobów zachowując spójny stan systemu. Dodatkowo opisano dodatkowe wymaganie, które jest opcjonalne. Opcjonalne wymaganie wydaje się być oksymoronem, ale w dalszej części tej sekcji zostanie

ono jasno scharakteryzowane.[14]

Niezmienne wytyczne określone w opisie REST to:

1. Architektura klient-serwer - zastosowanie tej zasady pozwala rozróżnić pojęcia interfejsu użytkownika oraz źródła danych, a także umożliwia skalowanie i przenośność poszczególnych komponentów niezależnie, w zależności od potrzeb.[14]
2. Bezstanowość - niniejsza zasada wprowadza wymóg przekazywania wszystkich informacji potrzebnych do realizacji zadania przez serwer w ramach jednego żądania; nie jest możliwe przechowywanie stanu po stronie serwera. Zasada ta znacząco poprawia niezawodność systemu, gdyż w razie jakichkolwiek awarii, nie ma potrzeby odtwarzania stanu przetwarzania aby zrealizować poprawnie żądanie. Oprócz tego realizacja żądania jest przejrzysta, gdyż nie ma potrzeby analizowania jego zawartości, aby określić naturę wywołania. Poprawiona jest w końcu możliwość skalowania systemu, gdyż nie trzeba skalować magazynu przechowującego stany aplikacji wszystkich klientów.[14]
3. Możliwość przechowywania w pamięci o krótkim czasie dostępu (*ang. cachability*) - jest to zasada wykorzystująca idempotentność operacji, aby zaoszczędzić zasoby obliczeniowe serwera, a tym zwiększyć wydajność współpracy aplikacji klienckiej i serwerowej. Pamiętać należy jednak, że wprowadza to do systemu ewentualną zgodność danych odczytanych przez użytkownika i rzeczywistych danych na serwerze. Projektując więc interfejsy REST, należy poświęcić dużo uwagi na określenie polityki częstotliwości wymuszania odświeżania danych, aby dostosować czas uzyskania pełnej zgodności danych do wymagań biznesowych.[14]
4. Ukrycie warstwowej architektury serwera - jest to zasada polegająca na ukrywaniu przed aplikacją kliencką faktu, że aplikacja kliencka jest w rzeczywistości zagregowaniem wielu mniejszych aplikacji serwerowych. Daje to wiele korzyści, takich jak chociażby umożliwienie skalowania systemów, poprzez dodanie komponentu równoważącego obciążenie (*ang. load balancer*). Jest on wtedy równocześnie serwerem proxy, który w kolejnych warstwach może odwoływać się do różnych instancji tej samej aplikacji. Możliwe jest również mapowanie zasobów na różne aplikacje serwerowe, bez informowania o tym użytkownika lub dokonywanie zmian w aplikacji serwerowej bez zmiany implementacji aplikacji klienckich.[14]
5. Ujednolicony interfejs - ta zasada umożliwia podążanie za konwencją korzystając z usług aplikacji serwerowej[9]. W tej zasadzie ujawnia się największa zaleta systemów opartych o REST, gdyż zapewnia uniwersalność aplikacji serwerowej i proste zasady, które spełniać powinna aplikacja kliencka, aby komunikować się z serwerem. Właśnie to zadecydowało o tak dużej popularności interfejsów typu REST[15]. Wadą jednolitego interfejsu jest to, że każda apli-

kacja uzyskuje pełną informację o zasobie, bez względu na to jaki ich podzbiór jest przez nią wymagany. W ramach tego niezmiennika wyróżnia się cztery punkty, opisujące dokładniej zasadę ujednoliconego interfejsu[5, 14]:

- a) Identyfikacja zasobów - używanie stałych identyfikatorów zasobu (np. URI w interfejsach REST opartych o sieć WWW).
 - b) Manipulacja zasobami przez ich reprezentację - zasada mówiąca o tym, że aplikacja kliencka może manipulować zasobami tylko w zakresie informacji uzyskanych przez odczyt.
 - c) Samoopisujące się wiadomości - każda wiadomość powinna zawierać wystarczającą informację opisującą jak przetworzyć żądanie. Przykładem może być identyfikator MIME w nagłówku 'Content-type' w przypadku protokołu HTTP
 - d) Hipermedium jako silnik stanu aplikacji (*ang. Hypermedia as the engine of application state*) (HATEOAS) - zasada umożliwiająca zrozumienie zasobu i ich zależności dzięki rozróżnianiu ich przez stałe identyfikatory oraz reprezentowanie zależności przez przekazywanie tych identyfikatorów[9]. Niezaprzeczalną wadą tego rozwiązania, jest potrzeba wykonania większej ilości zapytań, aby uzyskać pełen obraz bardziej złożonego zasobu.
6. Udostępnianie kodu na żądanie* - jest to zasada opcjonalna, która polega na umożliwieniu aplikacji klienckiej uzyskiwanie kodu, który można wykonać po stronie klienta. Przykładem przychodzącym na myśl w pierwszej kolejności jest pobieranie kodu JavaScript przez przeglądarki internetowe[14]. Pozorna sprzeczność pojęcia opcjonalnego wymagania, może zostać wyjaśniona przez podkreślenie, że system implementujący REST powinien dać sposobność wykonania kodu do wykonania, ale wykorzystywanie też możliwości należy ograniczyć to szczególnych przypadków przy sprawdzonych warunkach[14].

Specyfika interfejsu typu REST uwydatnia, kwestie które powinny zostać rozważone podczas weryfikacji oprogramowania opartego o ten styl architektoniczny. Przetestować można wówczas nie tylko funkcjonalne właściwości serwisu, ale również zgodność z założeniami stylu w którym jest on realizowany[9].

2.2.3 Połączenia między usługami

Wizja ogromnego systemu złożonego z wielu komponentów, realizującego skomplikowane zadania nasuwa na myśl skomplikowane połączenia między komórkami systemu. Istnieje wiele systemów, w których istnieją dodatkowe komponenty odpowiadające tylko i wyłącznie za realizację komunikacji. Komponenty takie zaimplementowane mają często skomplikowane reguły przekazywania wiadomości w zależności od logiki biznesowej. Przykładem są produkty typu ESB (*ang. Enterprise Service Bus*), które zawierają skomplikowane funkcje trasowania wiadomości, kapsułkowa-

nia lub transformacji danych. Komponenty takie, jako że są wspólne dla całego systemu, są trudne skalowalne, więc generują duże ryzyko wystąpienia wąskiego gardła. Mechanizm komunikacji powinien być prostą częścią systemu spełniając zasadę „inteligentne punkty końcowe i naiwne kanały komunikacji” (*ang. smart endpoints and dumb pipes*)[17]. Społeczność skupiona na rozwijaniu konceptu mikro-usług proponuje proste wywołania HTTP w celu wymiany danych między usługami. Niższe warstwy sieciowego modelu OSI zrealizują więc komunikację, a zadaniem „inteligentnych punktów końcowych” jest analiza żądań i generowanie odpowiedzi.

Wykorzystywanie prostych protokołów komunikacyjnych zrzuca niejako z programistów, ciężar projektowania i implementowania mechanizmów komunikacji oraz dostarcza całą gamę gotowych i sprawdzonych rozwiązań, które nie obciążają niepotrzebnie systemu.

Proste protokoły sieciowe, mimo niezaprzeczalnych zalet w sensie wydajnościowym, mogą być niewystarczające dla niektórych zastosowań. Protokół HTTP nie jest niezawodny, więc dla krytycznych punktów systemu wprowadzić można kolejki komunikatów, które nie realizują logiki biznesowej, a tylko zapewniają niezawodność komunikacji asynchronicznej niewielkim kosztem wydajności[34].

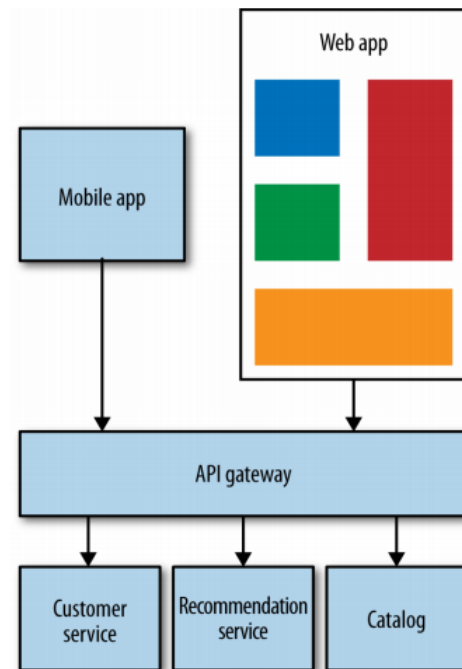
Praktyka pokazuje jednak, że istnieją przypadki, kiedy konieczne jest wyposażenie warstwy komunikacyjnej w komponent wspomagający, tzw. bramę interfejsów (*ang. API gateway*). Brama taka ma za zadanie agregować żądania, możliwie mała ich ilość była wymieniana między aplikacją kliencką, a usługami[22]. Podejście to stosowane jest często w przypadku aplikacji mobilnych, gdzie użytkownicy końcowi dysponują limitowaną ilością przesyłanych danych w ramach ich pakietu internetowego. Ta koncepcja może być zastosowana w dwóch wariantach. Pierwszym jest jedna, monolityczna brama interfejsów, która używana jest kiedy mamy niewiele (najczęściej jedną) aplikację z niej korzystającą. Wariant drugi to odseparowane bramy dla wielu aplikacji końcowych (*ang. backends for frontends*). Oba podejścia zaprezentowane są na rysunkach 2.1 oraz 2.2.

2.3 Dokumentacja oprogramowania

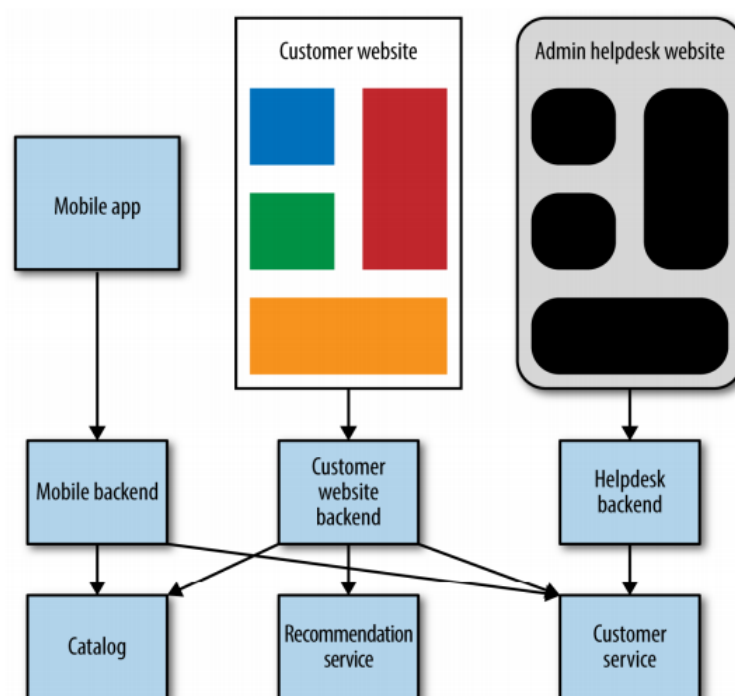
Dokumentacja jest nieodłączną częścią procesu wytwarzania oprogramowania[32, 6]. Prowadzenie dokumentacji ma oczywiście wiele zalet, jednak jest również źródłem dyskusji na temat opłacalności pod względem biznesowym[33].

2.3.1 Definicja terminu oraz rola dokumentacji

Dokumentacja to artefakt, którego celem jest komunikowanie innym informacji o systemie, którego dotyczy[16]. Dokumentacja może również odnosić się do dokumentów stworzonych przez programistów dla użytkowników nietechnicznych, znanego wtedy instrukcją[33]. Wyróżnić można też specyfikację, która jest technicz-



Rysunek 2.1: Monolityczna brama interfejsów[22].



Rysunek 2.2: Wiele bram dla aplikacji końcowych[22].

nym opisem oprogramowania, przeznaczonym dla użytkowników, którymi są inni programiści. Korzystają oni więc tylko z interfejsów udostępnionych przez usługi, ale rzadko uczestniczą w ich rozwoju[22].

W przypadku zbioru usług, często wykorzystywane są centralne dynamiczne rejestry usług, które udostępniają informację o tym jakie, usługi są dostępne i w jaki sposób nawiązać z nimi łączność. Zazwyczaj nie udostępniają jednak informacji o tym jakie funkcje są przez te usługi oferowane. Funkcję tę może pełnić techniczna dokumentacja, którą można pobrać wykonując ustalone żądanie[22]. Taki typ żądania jest oferowany przez protokół *HTTP* - chodzi o metodą *OPTIONS*[13]. Po wywołaniu takiej metody serwer może zwrócić plik dokumentacji.

2.3.2 Formalna specyfikacja usług sieciowych

Zauważyć należy, że dokumentacja techniczna nie musi być dokumentem pisanym w języku naturalnym, a jedynie precyzyjną informacją dla inżynierów pracujących nad systemem lub chcących uzyskać do niego dostęp[33]. W sekcji 2.3.1 opisano możliwość udostępniania dokumentacji technicznej poprzez wywołanie odpowiedniego żądania. Pobrana dokumentacja może potem zostać wczytana automatycznie przez parser, który jest w stanie zinterpretować jej treść. Trudno byłoby analizować dokumentację pisaną w języku naturalnym, więc powstało kilka technologii, które opisują interfejsy w zrozumiałym dla maszyny sposób i pozwalają na wykorzystanie tej wiedzy przez program komputerowy. Przykładami takich rozwiązań są *Swagger*, *HAL*[22] lub *RAML*.

2.3.3 Koszty rozwoju i utrzymania

Koszty wytworzenia dokumentacji szacunkowo wynoszą 11%[25]. Jest to oczywiście znaczący koszt, który z biznesowego punktu widzenia powinien być minimalizowany, zachowując jakość, a ściślej mówiąc wartość informacji, która zawarta jest w dokumentacji. Wraz z wzrostem popularności zwinnych metodologii [20] zespoły deweloperskie coraz częściej stawiają działające oprogramowanie i samoopisujący się kod ponad dokumentacją[7]. Siłą rzeczy, pisanie dokumentacji jest pomijane lub zaniedbywane, a często istniejąca dokumentacja nie jest uaktualniana, więc staje się coraz bardziej rozbieżna z oprogramowaniem.

2.3.4 RAML

RAML (*RESTful API Modeling Language*) to język służący do dokumentowania interfejsów usług sieciowych zbudowanych w zgodzie z opisanym w sekcji 2.2.2, stylem architektonicznym *REST* w oparciu o protokół *HTTP*. Jest on oparty na składni *YAML* i pozwala na intuicyjne odwzorowanie hierarchicznego modelu zasobów w *REST*[2].

Specyfikacja *RAML* podzielona jest na dwie główne części: pierwsza to zbiór metadanych na temat całego serwisu, a drugi to zbiór zasobów oferowanych przez daną usługę wraz z ich szczegółową specyfikacją. Na część pierwszą składają się np. tytuł, wersję, bazowy adres internetowy, schemat struktury danych, informacje na temat uzyskiwania autoryzacji lub typy danych. Szczegółowa specyfikacja zawarta w części drugiej dokumentacji podzielona jest głównie na hierarchicznie uporządkowane segmenty, z których każdy opisuje jakiś zasób. Na opis zasobu może składać się z następujących właściwości:

- `displayName` - nazwa zasobu,
- `description` - szczegółowy opis zasobu,
- metoda dostępu do danych (`get`, `patch`, `put`, `post`, `delete`, `options`, `head`),
- `type` - typ zasobu; realizuje niejako dziedziczenie kopiując wszystkie własności z wybranego typu,
- `is` - cecha zasobu; realizuje dziedziczenie w węższym zakresie niż typ, a mianowicie odnosi się do specyfikacji metod,
- `securedBy` - definiuje schemat autoryzacji dostępu do zasobu,
- `uriParameters` - lista możliwych parametrów przekazywanych w adresie URI,
- `/<relativeUri>` - podrzędny zasób, który opisany może zostać w ten sam sposób,
- `<annotationName>` - dodatkowe adnotacje, pozwalające na zawarcie dodatkowych informacji na temat zasobu.

Każda metoda na liście, również opisana jest wieloma własnościami:

- `displayName` - alternatywna nazwa metody, odnosząca się do rzeczywistej operacji przez nią dokonywanej w kontekście wiedzy domenowej,
- `description` - dłuższy opis metody,
- (`<annotationName>`) - dodatkowe adnotacje, pozwalające na zawarcie dodatkowych informacji na temat zasobu,
- `queryParameters` - szczegółowe informacje o parametrach zapytania używanych w ramach metody; wzajemnie wykluczające się z własnością *queryString*,
- `headers` - szczegółowe informacje na temat nagłówków zapytania *HTTP*,
- `queryString` - zagregowany ciąg znaków zawierający parametry; wzajemnie wykluczające się z własnością `queryParameters`,
- `responses` - informacja o możliwych odpowiedziach z serwera identyfikowane kodem odpowiedzi,
- `body` - informacja o strukturze, zawartości, możliwych wartościach oraz formacie zawartości sekcji *body* zapytania,

- protocols - lista protokołów używanych do wywołania metody; ta własność nadpisuje jakiegokolwiek wartości protokołów definiowane wyżej w hierarchii,
- is - lista cech, które opisują szczegóły metody,
- securedBy - definiuje schemat autoryzacji dostępu do zasobu.

Składnia RAML jest oczywiście dużo szersza i umożliwia jeszcze bardziej dokładne i szczegółowe definiowanie interfejsów REST. Z punktu widzenia tej pracy, są one jednak nieistotne a opisane wyżej elementy szczegółowo omówione będą w rozdziale 3.

2.4 Testowanie

Implementacja mikrousług a tym bardziej przepisywanie istniejącego monolitycznego systemu na architekturę mikroserwisową jest bardzo trudne i czasochłonne[24], więc aby zredukować koszty zespołu deweloperskiego należy oszczędzać czas programistów, między innymi rezygnując z pisania części testów. Oczywiście, nie można pozwolić sobie na obniżenie jakości oprogramowania, więc testy pominięte przez programistów można generować automatycznie. Idealnym źródłem danych wydaje się być dokumentacja techniczna interfejsów usług.

W ostatnich latach odchodzi się od testowania ręcznego na rzecz pełnej automatyzacji testów[22]. Kolejnym krokiem może być również automatyzacja generowania takich testów, a następnie automatyczne ich uruchomienie i weryfikacja.

2.4.1 Testowanie funkcjonalne

Wśród wielu rodzajów i technik testowania oprogramowania, na uwagę w kontekście tej pracy zasługuje testowanie funkcjonalne. Polega ono na takim wyborze przypadków testowych, aby pokryć wymagania funkcjonalne, które ma spełniać oprogramowanie[4]. Testowanie funkcjonalne znane jest również jako stesowanie czarnoskrzynkowe lub testowanie oparte o specyfikację[23].

Wyodrębnić można wiele różnych podziałów technik testowania, takich jak podział ze względu na zakres, ze względu na przedmiot testów albo ze względu na sposób ich wykonywania[23]. Testy można również podzielić na systematyczne lub losowe[23]. Przypadki testowe w kontekście testownia funkcjonalnego powinny być testami systematycznymi, czyli wartości argumentów oraz spodziewany wynik powinien zostać jasno określony podczas definiowania testu[23]. Źródłem takich informacji są *wyrocznice* czyli byty (dokumenty, artefakty, wiedza ekspercka)[28], więc skoro testy te są oparte na specyfikacji, ona powinna służyć jako wyrocznia. Powinna ona w takim razie definiować nie tylko sposób operowania na danych, ale również przykładowe dane wejściowe oraz oczekiwane wyniki.

Rysunek 2.3 pokazuje proces powstawania przypadku testowego zaproponowanego przez Prezza i Younga. Pierwszym krokiem jest zidentyfikowanie niezależnej funkcjonalności lub cechy systemu informatycznego. Będzie ona przedmiotem testu. Następnie należy wybrać wartości wejściowe oraz przeanalizować zasady zachowania funkcjonalności, w celu określenia oczekiwanego wyniku. Analiza w celu określenia rezultatu jest opcjonalna. Wynik można określić również przy pomocy wyroczni, więc zgodnie z zasadami testowania czarnoskrzynkowego, można stworzyć specyfikację przypadku testowego bez wiedzy eksperckiej. Ostatnim krokiem jest wygenerowanie przypadku testowego na podstawie jego specyfikacji. Wynikiem, może być na przykład dokument definiujący testy lub kod źródłowy testów.

2.4.2 Przedmiot testowania w przypadku interfejsów REST

Rozwijanie aplikacji z interfejsami programistycznymi polega tak naprawdę na dodaniu dodatkowej warstwy oprogramowania. Jest to więc dodatkowa warstwa, którą należy objąć testami[26].

Testy interfejsów REST posiadają specyfikę wymagającą podejścia do testów w taki sposób, aby pokryć przypadki wynikające z założeń stylu architektonicznego REST[26]. Takimi aspektami są:

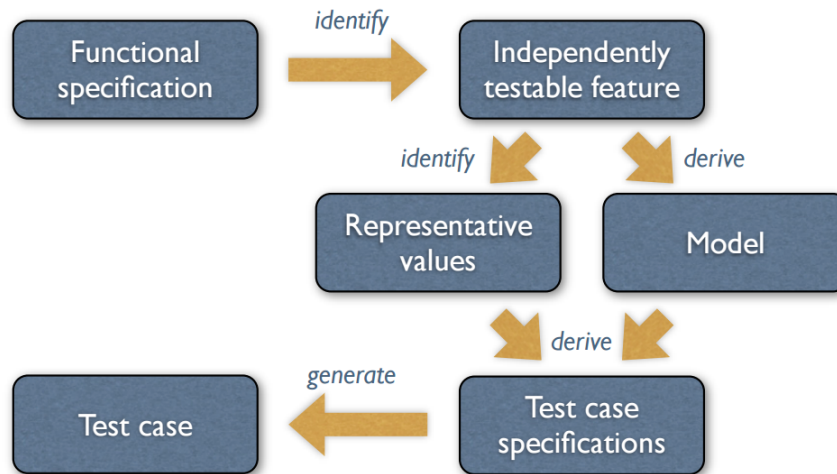
- Zgodność interfejsu z zasadami REST,
- Zgodność formatu zserializowanych obiektów z zadaniem,
- Testy scenariuszy (najpierw dodawanie, potem usuwanie, weryfikacja czy usunięcie się powiodło),
- Testy bezpieczeństwa,
- Testy wydajności komunikacji.

O ile pierwsze trzy punkty realizują założenia testów funkcjonalnych, to kolejnych dwóch nie można za takie uznać. W ramach tej pracy, skupimy się więc na pierwszych trzech aspektach testowania.

2.4.3 Ciągła integracja

W kontekście testowania funkcjonalnego, proces ciągłej integracji jest ważnym elementem weryfikacji oprogramowania poprzez testy regresyjne. Każda zmiana w kodzie źródłowym programu, skutkuje rozpoczęciem iteracji procesu budowania oprogramowania, a następnie wykonania testów zdefiniowanych wcześniej[3]. Jest to niejako mechanizm samoobrony procesu rozwoju oprogramowania[10]. Skoro oprogramowanie jest cały czas testowane możemy być pewni, że kod jest spójny ze specyfikacją. Częstym bowiem problemem jest fakt, że kod ewoluuje, a jego specyfikacja

Systematic Functional Testing



Rysunek 2.3: Proces tworzenia testów funkcjonalnych

zostaje zaniedbana. Powoduje to powstawanie rozbieżności i niespójności pomiędzy dokumentacją, specyfikacją, a rzeczywiście używanym oprogramowaniem[21].

Proponowane rozwiązanie

Rozdział drugi powinien uswiadomic czytelnika, ze obecny kierunek rozwoju inzynierii oprogramowania wymaga nowych narzedzi, które odciaza zespoły programistyczne od powtarzalnych czynności, które mimo, ze przynosza korzyści, sa często zaniebdywane, wprowadzając dodatkowy koszt, nie przekładający się na wymierny zysk.

Rozwiązaniem jest narzędzie automatyzujące proces analizy dokumentacji w celu generowania testów. W opinii autora, przyniesie ono zysk w postaci weryfikacji dwóch artefaktów naraz: kodu źródłowego interfejsów oraz dokumentacji technicznej tych interfejsów.

3.1 Model rozwiązania

Rozwiązanie należało zaprojektować na tyle uniwersalnie, aby spełnić dwa założenia.

1. Skoro jednym z podstawowych filarów architektury miroservisowej jest niezależność od technologii, to rozwiązanie musi wspierać generowanie kodu testów niezależnie od języka programowania.
2. Specyfikacja RAML jest tylko jednym z kilku języków technicznego dokumentowania interfejsów, więc rozwiązanie powinno być na tyle elastyczne, aby wspierać jego alternatywy.

Wychodząc z tych dwóch zasad, zaprojektowano rozwiązanie, które bazuje na zasadzie abstrakcji programowania obiektowego. W prosty sposób można zatem wymienić fragmenty kodu, aby umożliwić generowanie kodu testów w różnych technologiach i dla wielu różnych języków specyfikacji technicznej.

Program opierając się na dokumentacji interfejsów programistycznych generuje kod testów, które pokrywają odwołania HTTP do zasobów zgodne z zaimportowanym plikiem. Aby jednak rozwiązanie było elastyczne i otwarte na zmiany, generacja nie może być bezpośrednia. Na podstawie dokumentacji, generowany jest zestaw przypadków testowych, który w sposób zserializowany zostaje zapisany do pliku w

formacie XML. Ten plik jest w dalszej części wczytywany przez kolejną jednostkę przetwarzającą, która generuje kod testów, specyficzny dla wybranej technologii. Narzędzie wykonuje zatem swoje zadanie w następujących krokach:

1. Wczytanie pliku specyfikacji i parsowanie go.
2. Analiza zawartości i generacja uniwersalnego opisu przypadków testowych.
3. Zapis przypadków testowych do tymczasowego pliku XML.
4. Odczyt i analiza przypadków testowych.
5. Generacja specyficznego dla technologii kodu testów.

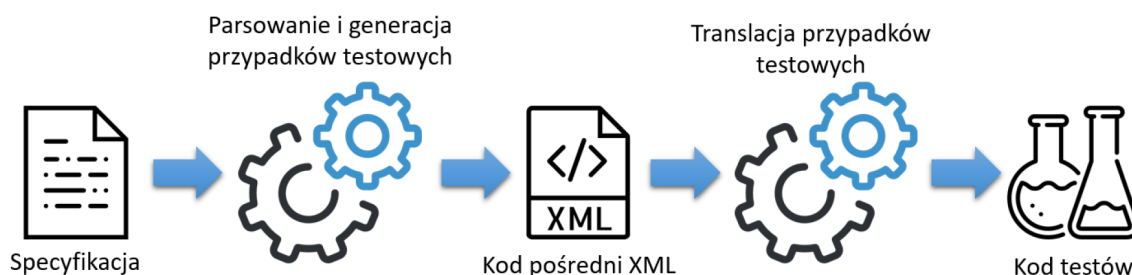
3.2 Opis działania aplikacji

Aplikacja zbudowana jest z dwóch głównych komponentów: generatora przypadków testowych i translatora przypadków testowych z XML do testów w wybranym języku. Przepływ danych przedstawiony jest na rysunku 3.1.

Główne zadanie jest realizowane przez pierwszy komponent, więc nad nim warto się chwilę pochylić.

Na 2.3 przedstawiono proces generowania przypadku testowego. Na nim oparty jest główny algorytm generowania przypadków testowych, który po sparsowaniu dokumentacji wykonuje następujące kroki:

1. Identyfikacja testowanej funkcjonalności (*ang. identify independently testable feature*) - aplikacja implementuje kilka klas, zgodnych z interfejsem *IUseCaseBuilder*, budujących przypadki testowe. Każda z tych klas odpowiada innej cesze opisanej w ramach specyfikacji RAML; np. sprawdzanie czy odpowiedź na zapytanie o danej wartości nagłówka „Content-Type” zwraca odpowiedź o oczekiwanym formacie odpowiedzi.
2. Identyfikacja reprezentatywnych wartości wejściowych i wyjściowych (*ang. identify representative values*) - implementacje interfejsu *IUseCaseBuilder* w swoich ramach rozpoznają dane wejściowe dla testowanej funkcjonalności. Wartości wejściowe w wielu przypadkach mogą zostać zidentyfikowane wprost ze



Rysunek 3.1: Przepływ danych

specyfikacji (np. nagłówki zapytania), z kolei w innych należy podać je jawnie (wartości parametrów URL). W tym celu wykorzystano cechę języka RAML umożliwiającą definiowanie opcjonalnych pól (*ang. Annotations*), gdzie należy umieścić wartość, która zostanie użyta.

3. Analiza w celu określenia wyniku operacji* (*ang. derive model*) - jak napisano w podsekcji 2.4.1, ten element procesu jest opcjonalny. Może być bowiem zastąpiony danymi uzyskanymi przez wyrocznie, a ściślej mówiąc, zawartych w specyfikacji jako wartości pól „*Annotations*” lub wynikających wprost ze specyfikacji.
4. Specyfikacja przypadku testowego (*ang. derive test case specifications*) - na podstawie powyższych kroków, można wydedukować specyfikację przypadku testowego, którą aplikacja zapisuje w jednoznacznej formie jako węzeł dokumentu XML.

W ramach drugiego komponentu realizowany jest ostatni krok procesu zaproponowanego przez Prezze i Younga[23]:

1. Generacja implementacji przypadku testowego (*ang. generate test case*) - translacja przypadków testowych na kod źródłowy testów.

Pamiętać należy, że testy aplikacji sieciowych takich jak mikrousługi, wymagają zawarcia w nich dodatkowego kodu, pozwalającego na zestawienie połączenia HTTP, wysłanie zapytania oraz obsługę odpowiedzi. Ten fragment musi być jednak w pełni realizowany w procesie translacji, gdyż może wyglądać zupełnie inaczej dla różnych technologii.

3.3 Szczegóły implementacyjne

Proponowane rozwiązanie zostało zaimplementowane w języku C# dla platformy .NET, niemniej jednak, jak wspomniano już wcześniej, może on generować testy dla wielu innych języków i platform. Przykład zaprezentowany w ramach tego rozwiązania generuje testy również w języku C#, dla biblioteki wspomagającej testowanie xUnit.net¹.

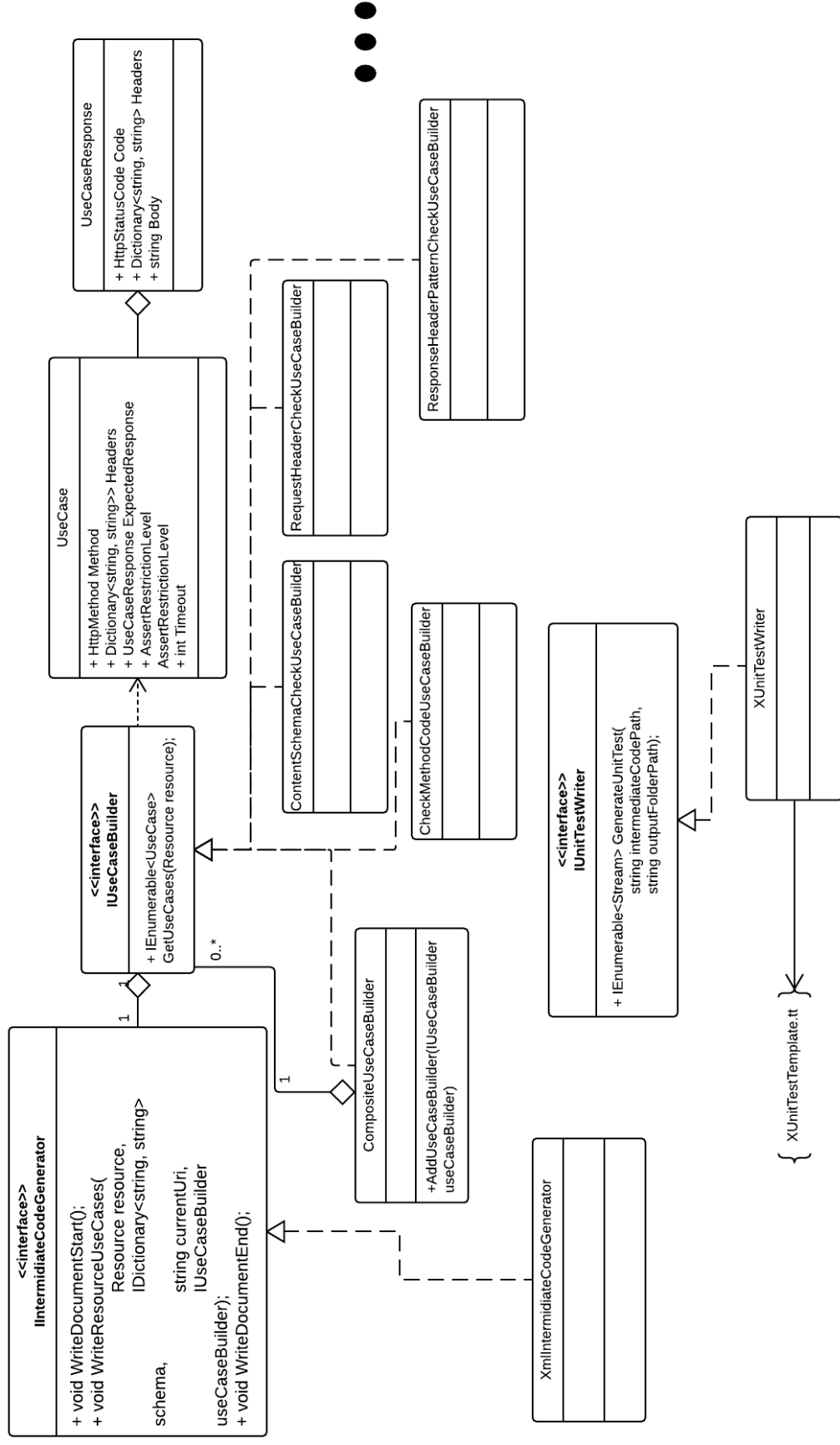
Diagram klas zaimplementowanego rozwiązania widoczny jest na rysunku 3.2.

3.3.1 Generacja kodu pośredniego

Do implementacji wykorzystano parser do języka RAML² udostępniony na licencji Apache 2.0. Aplikacja wczytuje z dysku twardego specyfikację RAML, parsuje ją, a następnie zwraca ją w przyjaznej dla programisty strukturze. Podczas procesu

¹<https://xunit.github.io>

²<https://github.com/raml-org/raml-dotnet-parser>



Rysunek 3.2: Diagram klas zaprezentowanego rozwiązania

implementacji okazało się, że wykorzystany parser nie wspiera pól opcjonalnych typu „Annotations”, więc należało zaimplementować obsługę tych pól.

Wygenerowany rezultat przetwarza klasa *XmlIntermediateCodeGenerator*. Operuje ona w pętli, działając dla każdego zasobu pierwszego rzędu zgodnie z kodem 3.1.

Lista przypadków testowych *useCases* generowana jest na podstawie zdefiniowanych wcześniej klas budowniczych, implementujących wspólny interfejs *IUseCaseBuilder*. W ramach przedstawionych przykładów, dla każdego zasobu analizowane są następujące testy:

1. Dla każdej metody zapytania kody odpowiedzi zdefiniowane w sekcji „Annotations” zgadzają się z parami [parametr, kod odpowiedzi]
2. Dla każdego zapytania zwracana jest odpowiedź o strukturze zgodnej z wartością w sekcji *Example*:
 - a) Wszystkie wymagane pola istnieją
 - b) Wszystkie pola istniejące w odpowiedzi są zadeklarowane w schemacie
3. Dla każdego ze zdefiniowanych typów odpowiedzi (wartość nagłówka *Content-Type*), wynik odpowiada formatowi zdefiniowanemu w zapytaniu.
4. Wszystkie nagłówki HTTP odpowiedzi są zgodne ze specyfikacją:
 - a) występujące nagłówki są zgodne z wzorcem
 - b) wymagane nagłówki istnieją

Jeśli jakikolwiek przypadek testowy (nawet jeśli jego asercja nie polega na weryfikacji kodu odpowiedzi) zwróci kod błędu z klasy 500 (błąd serwera), rezultatem testu będzie błąd.

3.3.2 Generacja kodu testów

Każdy język, technologia i biblioteka przy pomocy których realizowane są testy, wymaga kompletnie innego kodu źródłowego. Cały czas jednak bazuje on na modelu przypadku testowego, który jest wspólny dla wszystkich technologii.

Aby efektywnie zrealizować generację dokumentów tego typu, użyto techniki firmy Microsoft znanej jako szablony tekstowe T4 (*ang. T4 Text Templates*). Szablon taki wczytać można do pamięci podczas działania aplikacji, a następnie wygenerować kod na podstawie przekazanego modelu. Dużą zaletą takiego podejścia jest łatwość wymiany używanego szablonu, co pozwala na wygenerowanie kodu w praktycznie każdym języku.

Model pojedynczego przypadku testowego przedstawiono w kodzie 3.2.

Zawiera on następujące pola:

- *Name* - tekst nazywający zwięźle przypadek testowy

Algorytm 3.1 Przetwarzanie zasobu zdefiniowanego w RAML

Wejście:

resource- zasoby

useCaseBuilder- kompozyt agregujący wybranych budowniczych przypadków testowych

Wyjście:

plik XML zawierający definicje testów dla danego zasobu

```
1 public void WriteResourceUseCases(Resource resource, IUseCaseBuilder
    useCaseBuilder)
2 {
3     var useCases = useCaseBuilder.GetUseCases(resource);
4     foreach (var useCase in useCases)
5     {
6         PrintUseCase(resourceUri, useCase);
7     }
8     foreach (var innerResource in resource.Resources)
9     {
10        WriteResourceUseCases(innerResource, schema, resourceUri,
            useCaseBuilder);
11    }
12 }
```

Algorytm 3.2 Model pojedynczego przypadku testowego

```
1 public class UnitTestModel
2 {
3     public string Name { get; set; }
4     public string Link { get; set; }
5     public HttpMethod Method { set; get; }
6     public AssertRestrictionLevel AssertRestrictionLevel { get; set;}
7     public HttpStatusCode ResponseCode { get; set; }
8     public IDictionary<string,string> Headers { get; set; }
9     public IDictionary<string,string> ResponseHeaders { get; set; }
10    public string Body { get; set; }
11    public string ResponseBody { get; set; }
12 }
```

- *Link* - adres URL do wywołania
- *Method* - metoda zapytania HTTP
- *AssertRestrictionLevel* - poziom weryfikacji odpowiedzi
 - 0 - jakakolwiek odpowiedź z kodem innym niż z klasy 500
 - 1 - kod odpowiedzi zgodny z oczekiwanym
 - 2 - nagłówki odpowiedzi są zgodne z wzorcem o ile istnieje, oraz ich istnieją jeśli zostały oznaczone jako obowiązkowe
 - 3 - format odpowiedzi jest zgodny z zadaną wartością nagłówka *Content-Type*
 - 4 - struktura odpowiedzi jest zgodna ze zdefiniowanym schematem
 - 5 - treść odpowiedzi jest w pełni zgodna z oczekiwaną
- *ResponseCode* - oczekiwany kod odpowiedzi
- *Headers* - lista nagłówków HTTP zapytania
- *ResponseHeaders* - lista oczekiwanych wartości nagłówków odpowiedzi HTTP (z ewentualnymi wzorcami)
- *Body* - zawartość zapytania HTTP
- *ResponseBody* - zawartość odpowiedzi HTTP

3.3.3 Uruchamianie testów

Samo wygenerowanie kodu testów stanowi już bardzo dużą automatyzację procesu weryfikacji oprogramowania. Wciąż jednak testy takie muszą zostać manualnie wprowadzone do bazy kodu źródłowego, skompilowane oraz uruchomione. Proces ten jest jednak na tyle powtarzalny, że możliwe jest jego wykonanie za pomocą programu komputerowego.

W tym celu stworzono plik wykonywalny, który w celu pełnej automatyzacji wykonuje następujące kroki:

1. wczytuje ścieżkę pliku specyfikacji przekazany jako parametr wywołania aplikacji;
2. uruchamia prototyp w celu generacji kodu testów jednostkowych;
3. kompiluje kod w celu wygenerowania pliku wykonywalnego (biblioteka .dll);
4. kopiuje z repozytorium zestaw narzędzi niezbędnych do przeprowadzenia testów; w tym przypadku są to pliki biblioteki xUnit;
5. uruchamia osobny proces wykonujący wszystkie testy zawarte w wygenerowanej bibliotece

Narzędzie posiadające wszystkie te komponenty powinno stanowić kompletny apa-

rat programistyczny pomagający weryfikować typ oprogramowania jakim są usługi sieciowe.

Weryfikacja i przeprowadzone testy

W celu wykazania przydatności i poprawności działania zaproponowanego rozwiązania przeprowadzono testy. Prototyp narzędzia został przetestowany w rzeczywistym środowisku infrastruktury sieci Internet. Pozyskano powszechnie dostępną specyfikację RAML istniejącego systemu, a następnie uruchomiono program podając ją jako plik wejściowy.

4.1 Testowany system

6 listopada 2010 roku w sieci Internet pojawił się serwis Instagram założony przez Kevina Systroma and Mike’a Kriegera[1]. Stał się on na tyle popularny, że w grudniu 2014 roku był użytkowany przez 300 mln użytkowników[12].

Aplikacja Instagram umożliwia umieszczanie na zdalnych serwerach fotografii, na które można nakładać różne filtry fotograficzne, nadające zdjęciom przyjemny dla oka wygląd. Każde zdjęcie lub film (*ang. media*) jest może zostać przypisane tylko do jednego użytkownika systemu, przy czym każdy użytkownik może posiadać wiele takich mediów. Każde medium może posiadać zdefiniowane przez autora słowa kluczowe (*ang. hash tags*) opisujące opublikowany przez niego zasób. Oprócz tego, na każdy z tych zasobów mogą reagować inni użytkownicy serwisu. Na Instagram zaimplementowano dwa rodzaje takich reakcji: polubienia (*ang. likes*) oraz komentarze (*ang. comments*). Każdy zasób może zostać również przypisany do jakiejś rzeczywistej lokalizacji (*ang. location*). Co więcej można uzyskać również informacje o powiązaniach pomiędzy użytkownikami (*ang. relationships*). Powiązanie jest zdefiniowane za pomocą pary uporządkowanej dwóch użytkowników oraz typu relacji. Instagram zdefiniował następujące relacje[19]:

- brak (*ang. none*) - brak powiązania lub relacja między dwoma tymi samymi użytkownikami
- śledzenie (*ang. follows*) - relacja śledzenia, czyli możliwości wyświetlania w przypadku użytkownika prywatnego, lub uzyskiwania powiadomień o aktywności użytkownika w przypadku kiedy użytkownik jest publiczny

- bycie śledzonym (*ang. followed-by*) - relacja odwrotna do relacji śledzenia, tj. użytkownik będący drugim argumentem relacji ma możliwość śledzenia aktywności użytkownika będącego pierwszym argumentem relacji
- wysłano prośbę (*ang. requested-by*) - relacja mówiąca o tym, że użytkownik wysłał prośbę o umożliwienie śledzenia do innego użytkownika

Informacje o wszystkich tych zasobach można pobrać na dwa sposoby. Pierwszym jest uzyskanie dostępu do strony internetowej za pomocą panelu logowania, drugim natomiast jest wykonywanie zapytań do serwisu znajdującego się pod adresem <https://api.instagram.com> w celu pozyskania w odpowiedzi samych danych (bez całości strony HTML).

Sposób pierwszy jest polecany dla użytkowników korzystających z przeglądarek internetowych, które interpretują wizualnie zawartość plików HTML. Sposób drugi jest natomiast polecany dla programistów, którzy projektują aplikacje użytkowe korzystające z tych danych[19].

4.2 Metodologia przeprowadzonych testów

4.2.1 Przygotowanie do wykonania testów

4.2.1.1 Instagram

Serwis Instagram istotnie umożliwia połączenia zdalne z udostępnionym interfejsem programistycznym, niemniej jednak w celu uzyskania dostępu do *api.instagram.com* należy przejść przez specjalnie przygotowaną procedurę.

W pierwszej kolejności należy stworzyć klienta (*ang. client*), w celu uzyskania identyfikatora aplikacji oraz tajnego klucza (*ang. Client Secret*). Dane te są konieczne, aby uzyskać żeton autoryzacyjny (*ang. token*). Ten wykorzystany będzie przy okazji każdego zapytania w celu potwierdzenia tożsamości. Mechanizm autoryzacyjny wykorzystany w systemie Instagram to OAuth 2.0[18, 19]. Kwestia autoryzacji nie mieści się w zakresie tej pracy, więc założono stałość tokenu autoryzacyjnego i użyto go jako zmiennej całego procesu, przekazywanej jako parametr.

Konfiguracja klienta przeznaczona dla celów testowych determinuje użycie go w trybie „piaskownicy” (*ang. sandbox mode*). Tryb ten minimalizuje ryzyko wykonania niepożądanych akcji, ograniczając widoczne informacje jedynie do zbioru użytkowników, którzy dołączyli do danego klienta. Wymagało to więc wykorzystania istniejących kont Instagram, których właściciele zgodzili się udostępnić dane swoich kont dla badawczych celów niniejszej pracy. Nazwa klienta, który został utworzony w celach testów to „*RestTestingFramework*”, a nazwy trzech kont wykorzystanych w celach testów to „*ensorceler*”, „*matjanos*” oraz „*kamil518*”.

4.2.1.2 Dokumentacja techniczna

Twórcy systemu Instagram nie udostępnił dokumentacji swoich interfejsów programistycznych w formie dokumentu RAML, a jedynie stronę internetową, która w języku naturalnym opisuje wszystkie udostępniane zasoby oraz sposób ich udostępniania[19]. Społeczność tworząca projekt RAML udostępnia jednak kolekcję dokumentacji wielu popularnych serwisów internetowych, a na ich liście znalazł się również Instagram[8]. Okazało się jednak, że udostępniona specyfikacja nie jest aktualna i nie do końca odpowiada obecnemu stanowi interfejsu programistycznego[19, 8], więc wymagane było wprowadzenie uaktualnień zgodnie z aktualną specyfikacją umieszczoną na stronie internetowej.

Modyfikacje dotyczyły dwóch aspektów:

- uspoźnienie specyfikacji z aktualną wersją interfejsu opisanego przez Instagram,
- dodanie pól typu „*Annotations*”, w celu zdefiniowania wyroczeni (metoda opisana w sekcji 3.2).

Wartości pól typu „*Annotations*” definiowano na podstawie faktycznego stanu aplikacji znanego autorowi. Były to następujące parametry:

- *mediaId* = 392839750589921094_287676068,
- *userId* = 391387866,
- *commentId* = 17851823476094686,
- *tagName* = „*computerscience*”,
- *locId* = 217492538.

Opisano tam również oczekiwane obiekty, aby móc zweryfikować ich strukturę oraz wartości.

4.2.2 Przeprowadzenie testów

Przygotowany prototyp narzędzia do automatycznego testowania został przygotowany jako plik wykonywalny dla systemu Windows 7 / Windows 8/8.1 / Windows 10 z zainstalowanym środowiskiem .NET 4.6. Narzędzia należało użyć wykonując następującą komendę:

```
1 ./MasterThesis.DemoRunner.exe ./TestFiles/Instagram1.0/api.raml
```

Uruchamia ona aplikację, korzystając z pliku specyfikacji *api.raml* znajdującego się w podfolderze *./TestFiles/Instagram1.0*. Przykładowe wyjście programu zaprezentowano na obrazie 4.1. Oznacza ono, że podczas generowania przypadków testowych znaleziono 2 błędy oraz jedno ostrzeżenie, a także aplikacja wykonała 8 testów, z których 3 zakończyły się niepowodzeniem.

```

1 Loaded file: ./TestFiles/Instagram1.0/api.raml
2 2016-09-14 00:28:58.3643|INFO|MasterThesis.RestTestsGenerator.RestTestGenerator|File D:\kmatj_000\Documents\St
3 udia\MasterThesis\out\Soft\TestFiles\Instagram1.0\api.raml loaded
4 2016-09-14 00:28:58.4728|ERROR|MasterThesis.RestTestsGenerator.UseCaseGenerators.ContentSchemaCheckUseCaseBuil
5 der|There is no definition of response for request Content-Type: application/x-www-form-urlencoded
6 2016-09-14 00:28:58.5173|WARN|MasterThesis.RestTestsGenerator.IntermediateCodeGenerator.XmlIntermediateCodeGen
7 erator|There is no placeholder defined for geoId
8 2016-09-14 00:28:58.5413|ERROR|MasterThesis.RestTestsGenerator.UseCaseGenerators.ContentSchemaCheckUseCaseBuil
9 der|There is no definition of response for request Content-Type: application/x-www-form-urlencoded
10 Finished generation.
11 Compiling.
12 Finished generation. D:\kmatj_000\Desktop\tests\OutputAssembly.dll
13 Copy tools...
14 Executing tests...
15 Finished testing. Code: 3
16 xUnit.net Console Runner (64-bit .NET 4.0.30319.42000)
17   Discovering: OutputAssembly
18   Discovered:   OutputAssembly
19   Starting:     OutputAssembly
20     Assert.Equal() Failure
21       Expected: OK
22       Actual:   BadRequest
23       Stack Trace:
24         at MasterThesis.RestTestsGenerator.Tests.TemplateTest.<Test_7625c9941b7344ff9f52a166db270605>d__3c.
25 MoveNext()
26       --- End of stack trace from previous location where exception was thrown ---
27         at System.Runtime.CompilerServices.AsyncMethodBuilderCore.<>c.<ThrowAsync>b__6_0(Object state)
28     Assert.Equal() Failure
29       Expected: OK
30       Actual:   BadRequest
31       Stack Trace:
32         at MasterThesis.RestTestsGenerator.Tests.TemplateTest.<Test_4a0db796d6cd4900a702f977898ab883>d__46.
33 MoveNext()
34       --- End of stack trace from previous location where exception was thrown ---
35         at System.Runtime.CompilerServices.AsyncMethodBuilderCore.<>c.<ThrowAsync>b__6_0(Object state)
36     Assert.Equal() Failure
37       Expected: OK
38       Actual:   BadRequest
39       Stack Trace:
40         at MasterThesis.RestTestsGenerator.Tests.TemplateTest.<Test_faf3b69d1b1544bd9ad97e077cca5aaf>d__32.
41 MoveNext()
42       --- End of stack trace from previous location where exception was thrown ---
43         at System.Runtime.CompilerServices.AsyncMethodBuilderCore.<>c.<ThrowAsync>b__6_0(Object state)
44   Finished:   OutputAssembly
45 === TEST EXECUTION SUMMARY ===
46   OutputAssembly Total: 8, Errors: 0, Failed: 3, Skipped: 0, Time: 3,273s
47

```

Rysunek 4.1: Przykładowe wyjście narzędzia

4.3 Otrzymane wyniki i rezultaty

Prototyp narzędzia wygenerował 30 testów. Pokrywają one 7 następujących zasobów:

- Użytkownicy (*ang. Users*) -
- Powiązania między użytkownikami (*ang. Relationships*)
- Zdjęcia lub filmy (*ang. Media*)
- Komentarze (*ang. Comments*)
- Polubienia (*ang. Likes*)
- Słowa kluczowe (*ang. Tags*)
- Lokalizacje (*ang. Locations*)

Testy objęły wykonanie wielu operacji w serwisie Instagram, na które złożyły się:

- wyświetlanie zasobów,
- usuwanie zasobów,
- dodawanie zasobów.

W ramach testów zweryfikowano następujące informacje:

- zgodność kodu HTTP odpowiedzi,
- zgodność odpowiedzi ze schematem,
- zgodność nagłówek odpowiedzi,
- zgodność formatu odpowiedzi.

Test został przeprowadzony w 6 powtórzeniach przy czym zapewniono, aby kolejność wywoływania testów była losowa. Wyniki zaprezentowano w tabeli 4.1.

Niedeterministyczne testy dotyczyły dodawania, usuwania oraz wyświetlania po-

Iteracja	Wynik		
1	Wykonano: 30	Sukces: 30	Porażka: 0
2	Wykonano: 30	Sukces: 29	Porażka: 1
3	Wykonano: 30	Sukces: 28	Porażka: 2
4	Wykonano: 30	Sukces: 28	Porażka: 2
5	Wykonano: 30	Sukces: 29	Porażka: 1
6	Wykonano: 30	Sukces: 28	Porażka: 2

Tablica 4.1: Wyniki przeprowadzonych testów

lubień, a więc następujących operacji HTTP pod wskazanym adresem (usunięto parametr *access_token*, który w tym przypadku nie ma znaczenia):

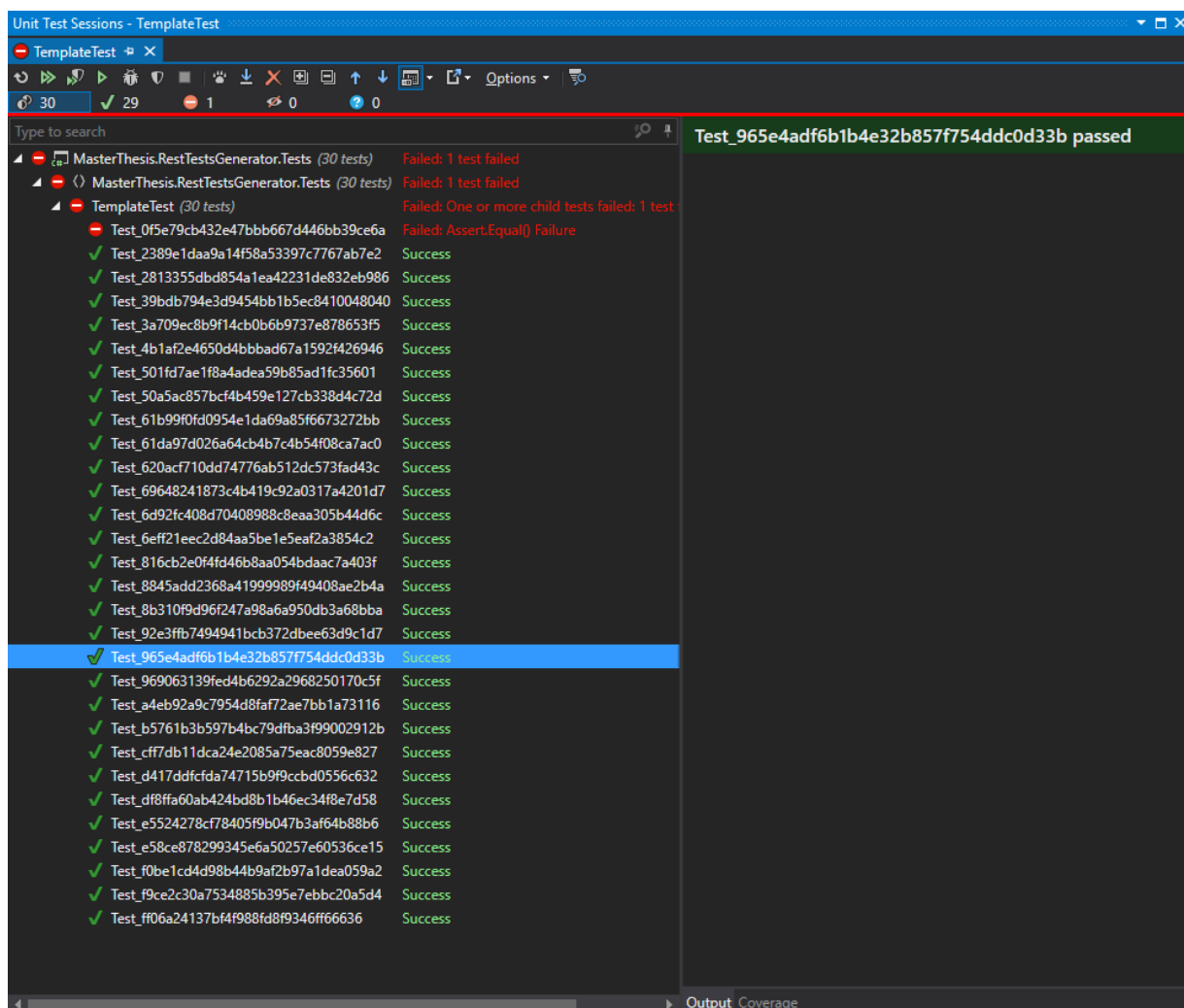
Były to jedyne testy, które manipulowały danymi. Wszystkie inne zapytania w zbiorze wygenerowanych testów były wywołaniami typu GET, więc nie zmieniały stanu systemu online.

W celu wizualizacji testów uruchomiono je również w graficznym środowisku programistycznym Visual Studio 2015, a ich wynik zaprezentowany jest na rysunku 4.2.

GET https://api.instagram.com/v1/media/1338612569593807801_287676068/likes

POST https://api.instagram.com/v1/media/1338612569593807801_287676068/likes

DELETE https://api.instagram.com/v1/media/1338612569593807801_287676068/likes



Rysunek 4.2: Testy uruchomione w środowisku programistycznym Visual Studio 2015.

Wnioski i podsumowanie

5.1 Analiza rezultatów testów

5.1.1 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 analizowane w formacie *application/json*),
- Weryfikacja struktury zwracanego obiektu z jego schematem,
- Weryfikacja konkretnej odpowiedzi na podstawie parametrów zapytań zdefiniowanych w specyfikacji RAML w postaci 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.

5.1.2 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 operacje:

- 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 próbując usunąć nieistniejące 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 sprawdzano 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.

5.2 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 jakiegokolwiek ingerencji użytkownika lub programisty. W znaczącym stopniu może usprawnić to proces weryfikacji i kontroli istniejących systemów wyposażonych w interfejsy programistyczne 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 niewyzolowanym dopuszcza ingerencję osób trzecich w stan systemu. W środowisku produkcyjnym, jakie zostało prze-

testowane, zmiany zbioru komentarzy lub ilości polubień są na porządku dziennym, więc cykliczna jego weryfikacja przy stałych założeniach stanu początkowego jest bezcelowe.

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

5.2.1 Odpowiedzi na problemy

5.2.1.1 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 eliminuje 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 zbadać, 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 mikro-usługowej, a mianowicie o spójności ewentualnej (*ang. eventual consistency*)[27]. Oznacza ona, że po wykonaniu pewnej operacji na systemie, nie możemy oczekiwać, że ta zmiana zostanie zapisana natychmiast, a pozytywna odpowiedź z serwera oznacza tylko przyjeździe zadania do realizacji. Ta kwestia może zostać rozwiązana 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*).

5.2.1.2 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 rozwiązać ten problem, można podzielić testy na dwie grupy: inwazyjną i nieinwazyjną.

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 nieinwazyjnych 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 oczekiwaną, a rzeczywistą jakością pracy serwisów.

5.3 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 (czasu odpowiedzi, prędkość przesyłu danych, wartość danych według przedziałów)

Dzięki temu, że rozwiązanie zostało zaprojektowane 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 menadżerów oraz dyrektorów technicznych, sprawujących pieczę nad dużymi rozproszonymi systemami.

5.4 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 oprogramowania, a potem odniesiono się do kwestii 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.

Bibliografia

- [1] Instagram press. <https://www.instagram.com/press>. Accessed: 2016-09-01.
- [2] Raml version 1.0: Restful api modeling language. Online, 2016.
- [3] Ruth Ablett, Ehud Sharlin, Frank Maurer, Jörg Denzinger, and Craig Schock. Buildbot: Robotic monitoring of agile software development teams. In *IEEE RO-MAN 2007, 16th IEEE International Symposium on Robot & Human Interactive Communication, August 26-29, 2007, Jeju Island, Korea, Proceedings*, pages 931–936, 2007.
- [4] I. Arsie, G. Betta, D. Capriglione, A. Pietrosanto, and P. Sommella. Functional testing of measurement-based control systems: An application to automotive. *Measurement*, 54:222 – 233, 2014.
- [5] Michael Athanasopoulos and Kostas Kontogiannis. Extracting REST resource models from procedure-oriented service interfaces. *Journal of Systems and Software*, 100:149–166, feb 2015.
- [6] Joachim Bayer and Dirk Muthig. A view-based approach for improving software documentation practices. In *13th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2006), 27-30 March 2006, Potsdam, Germany*, pages 269–278, 2006.
- [7] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Ro-

- bert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001.
- [8] RAML Community. Raml apis. <https://github.com/raml-apis/Instagram>, 2016.
- [9] Bruno Costa, Paulo F. Pires, Flavia C. Flávia C. Delicato, and Paulo Merson. Evaluating REST architectures-Approach, tooling and guidelines, apr 2014.
- [10] John Downs, John G. Hosking, and Beryl Plimmer. Status communication in agile software teams: A case study. In *The Fifth International Conference on Software Engineering Advances, ICSEA 2010, 22-27 August 2010, Nice, France*, pages 82–87, 2010.
- [11] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.
- [12] Seth Fiegerman. Instagram tops 300 million active users, likely bigger than twitter. Online, dec 2014.
- [13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999.
- [14] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887.
- [15] Marios Fokaefs and Eleni Stroulia. Using WADL specifications to develop and maintain REST client applications. In *2015 IEEE International Conference on Web Services, ICWS 2015, New York, NY, USA, June 27 - July 2, 2015*, pages 81–88, 2015.
- [16] Andrew Forward. Software documentation – building and maintaining artefacts of communication. Master’s thesis, University of Ottawa, 2002.
- [17] J. Lewis M. Fowler. Microservices - a definition of this new architectural term. <http://martinfowler.com/articles/microservices.html>, mar 2014. Accessed: 2016-04-25.

- [18] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2015.
- [19] Instagram Inc. Instagram developer documentation. Online.
- [20] Project Management Institute. Pim's pulse of the profession. Technical report, Project Management Institute, mar 2012.
- [21] Robert C. Martin. *Clean Code - a Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.
- [22] Sam Newman. *Building Microservices*. Ó'Reilly Media, Inc.", 2015.
- [23] Mauro Pezzè and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.
- [24] Santiago Gómez Sáez, Vasilios Andrikopoulos, Florian Wessling, and Clarissa Cassales Marquezan. Cloud adaptation and application (re-)distribution: Bridging the two perspectives. In *18th IEEE International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, EDOC Workshops 2014, Ulm, Germany, September 1-2, 2014*, pages 163–172, 2014.
- [25] Isaac Sánchez-Rosado, Pablo Rodríguez-Soria, Borja Martín-Herrera, Juan Jose Cuadrado-Gallego, José Javier Martínez-Herráiz, and Alfonso González. Assessing the documentation development effort in software projects. In *Software Process and Product Measurement, International Conferences IWSM 2009 and Mensura 2009, Amsterdam, The Netherlands, November 4-6, 2009. Proceedings*, pages 337–346, 2009.
- [26] Pablo Lamela Seijas, Huiqing Li, and Simon J. Thompson. Towards property-based testing of restful web services. In *Proceedings of the Twelfth ACM SIGPLAN Erlang Workshop, Boston, Massachusetts, USA, September 28, 2013*, pages 77–78, 2013.
- [27] Marc Shapiro. A principled approach to eventual consistency. In *20th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, WETICE 2011, Paris, France, 27-29 June 2011, Proceedings*, page 1, 2011.

- [28] Phil Stocks and David A. Carrington. A framework for specification-based testing. *IEEE Trans. Software Eng.*, 22(11):777–793, 1996.
- [29] 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.
- [30] J. Thönes. Microservices. *IEEE Software*, 32(1):116–116, Jan 2015.
- [31] Adam Trachtenberg. Php web services without soap. Online, oct 2003.
- [32] Marcello Visconti and Curtis R. Cook. Software system documentation process maturity model. In *Proceedings of the ACM 21th Conference on Computer Science, CSC '93, Indianapolis, IN, USA, February 16-18, 1993*, pages 352–357, 1993.
- [33] Junji Zhi, Vahid Garousi-Yusifoglu, Bo Sun, Golara Garousi, S. M. Shahnewaz, and Günther Ruhe. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, 99:175–198, 2015.
- [34] Judicael A. Zounmevo and Ahmad Afsahi. A fast and resource-conscious MPI message queue mechanism for large-scale jobs. *Future Generation Comp. Syst.*, 30:265–290, 2014.