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[24]. Najczęściej stosowanym rozwiązaniem jest używanie komunikacji HTTP w konwencji REST, gdyż zostało to zarekomendowane przez prekursorów idei rozproszonych mikrousług - Martina Fowlera oraz Jamesa Levisa[14].

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.[23] 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*)[9] 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[17].* 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

### 2.2.1 Podział monolitu

Krytyczną decyzją podczas projektowania architektury opartej o mikrousługi jest wybór punktów podziału systemu na związane konteksty (*ang. bounded context*)[9, 14] 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.[17]

Drugim wariantem jest wyekstrahowanie modułów. Moduł jest fragmentem kodu, który realizować powininen 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[17]. Ważne jest, aby nie unifikować pojęcia związanego kontekstu z modułem[9]. Związany kontekst jest raczej zbiorem modułów, istniejących w tym samym celu w sensie domeny. Dobrą praktyką jest więc tworzenie kilku warst aplikacji w osobnych modułach zawierających się w tej samej przestrzeni nazw, więc wydzielonym kontekście[9].

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 rozdziale2.1.1, do komunikacji między mikrousługami powinno stosować się lekkich i szybkich rozwiązań, więc podejście REST, które jest sześciokrotnie szybsze niż starsze protokoły SOAP[25], 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.[11]

Niezmienniki wyspecyfikowane 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.[11]
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 skalownia systemu, gdyż nie trzeba skalować magazynu przechowującego stany aplikacji wszystkich klientów.[11]
3. Możliwość przechowywanie 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.[11]
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.[11]
5. Ujednolicony interfejs - ta zasada umożliwia podążanie za konwencją korzystając z usług aplikacji serwerowej[7]. 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[12]. 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[4, 11]:

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[7]. 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[11]. 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[11].

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[7].

### 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 trudnos 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*)[14]. Społeczność skupiona na rozwijaniu konceptu mikrousług 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[28].

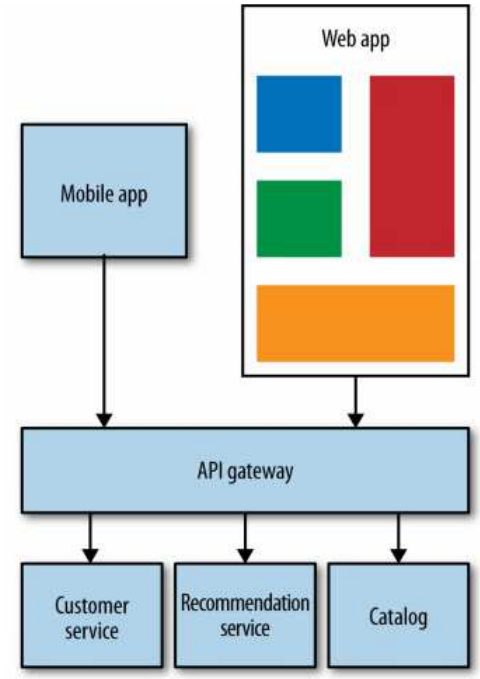
Praktyka pokazuje jednak, że istnieją przypadki, kiedy konieczne jest wyposażenie warstwy komunikacyjnej w komponent wspomagający, tzw. bramę interfejsów (*ang. API gateway*). Brama taka ma za zadanie agregować żądania, możliwie mała ich ilość była wymieniana między aplikacją kliencką, a usługami[17]. 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

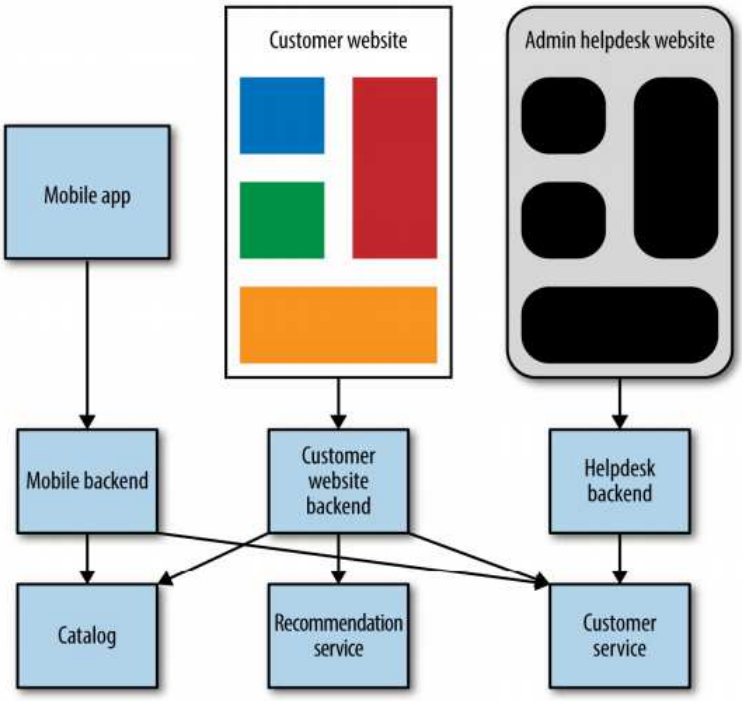
Dokumentacja jest nieodłączną częścią procesu wytwarzania oprogramowania[26, 5]. Prowadzenie dokumentacji ma oczywiście wiele zalet, jednak jest również źródłem dyskusji na temat opłacalności pod względem biznesowym[27].

### 2.3.1 Definicja terminu oraz rola dokumentacji

Dokumentacja to artefakt, którego celem jest komunikowanie innym informacji o systemie, którego dotyczy[13]. Dokumentacja może również odnosić się do dokumentów stworzonych przez programistów dla użytkowników nietechnicznych, znanego wtedy instrukcją[27]. Wyróżnić można też specyfikację, która jest technicz-




**Rysunek 2.1:** Monolityczna brama interfejsów[17].




**Rysunek 2.2:** Wiele bram dla aplikacji końcowych[17].

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[17].

W przypadku zbioru usług, często wykorzystywane są centralne dynamiczne rejesty 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[17]. Taki typ żądania jest oferowany przez protokół *HTTP* - chodzi o metodą *OPTIONS*[10]. 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[27]. W sekcji 2.3.1 opisano możliwość udostępniania dokumentacji technicznej poprzez wywołanie odpowiedniego żądania. Pobrna dokumentacja może potem zostać wczytana automatycznie przez parser, który jest w stanie zinterpretować jej treść. Trudno byloby analizować dokumentację pisaną w języku naturalnym, więc powstało kilka technologii, które opisują interfersy w zrozumiały dla maszyny sposób i pozwalają na wykorzystanie tej wiedzy przez program komputerowy. Przykładadami takich rozwiązań są *Swagger*, *HAL*[17] lub *RAML*.

### 2.3.3 Koszty rozwoju i utrzymania

Koszty wytworzenia dokumentacji szacunkowo wynoszą 11%[20]. Jest to oczywiście znaczący koszt, który z biznesowego punktu widzenia powinien być minimalizowany, zachowując jakość, a ściśle mówiąc wartość informacji, która zawarta jest w dokumentacji. Wraz z wzrostem popularności zwinnych metodologii [15] zespoły deweloperskie coraz częściej stawiają działające oprogramowanie i samoopisujący się kod ponad dokumentają[6]. 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 architekonicznym *REST* w oparciu o protokół *HTTP*. Jest on oparty na składni *YAML* i pozwala na intuicyjne odwzorowanie hierarchicznego modelu zasobów w *REST*[1].

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 wywałania metody; ta własność nadpisuje jakiekolwiek wartości protokołów definiowane wyżej w hierarchi,
- 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[19], 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[17]. 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[3]. Testowanie funkcjonalne znane jest również jako stesowanie czarnoskrzynkowe lub testowanie oparte o specyfikację[18].

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[18]. Testy można również podzielić na systematyczne lub losowe[18]. Przypadki testowe w kontekście testownia funkcjonalnego powinny być testami systematycznymi, czyli wartości argumentów oraz spodziewany wynik powinien zostać jasno określony podczas definiowania testu[18]. Źródłem takich informacji są *wyrocznie* czyli byty (dokumenty, artefakty, wiedza ekspercka)[22], 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 zaproponnowanego przez Prezze 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 określi również przy pomocy wyroczni, więc zgodnie z zasadami testowania czarnoskrzynkowego, można stworzyć specyfikację przypadku testowego bez wiedzy eksperckiej. Ostatnim krokiem jest wygenerownie przypadku testowego na podstawie jego specyfikacji. Wynikiem, może być na przykład dokument definiujący testy lub kod źródłowy tesów.

### 2.4.2 Przedmiot testowania w przypadku interfejsów REST

Rozwijanie aplikacji z interfejsami programistycznmi polega tak naprawdę na dodaniu dodatkowej warstwy oprogramowania. Jest to więc dodatkowa warstwa, którą należy objąć testami[21].

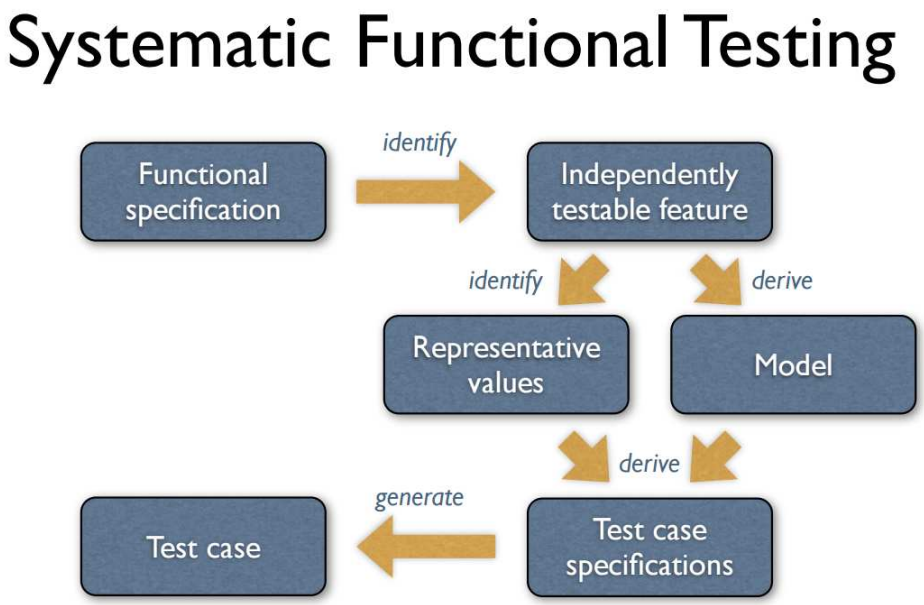
Testy interfejsów REST posiadają specyfikę wymagającą podejścia do testów w taki sposób, aby pokryć przypadki wynikające z założeń stylu architektonicznego REST[21]. Takimi aspektami są:

- Zgodność interfejsu z zasadami REST,
- Zgodność formatu zserializowanych obiektów z zadanym,
- 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[2]. Jest to niejako mechanizm samoobrony procesu rozwoju oprogramownia[8]. 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




**Rysunek 2.3:** Proces tworzenia testów funkcjonalnych

zostaje zaniedbana. Powoduje to powstawanie rozbieżności i niespójności pomiędzy dokumentacją, specyfikacją, a rzeczywiście używanym oprogramowaniem[16].

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] Raml version 1.0: Restful api modeling language. Online, 2016.

[2] 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.

[3] 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.

[4] Michael Athanasopoulos and Kostas Kontogiannis. Extracting REST resource models from procedure-oriented service interfaces. *Journal of Systems and Software*, 100:149–166, feb 2015.

[5] 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.

[6] 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, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001.

[7] Bruno Costa, Paulo F. Pires, Flavia C. Flávia C. Delicato, and Paulo Merson. Evaluating REST architectures-Approach, tooling and guidelines, apr 2014.

[8] 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.

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

[10] 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.

[11] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887.

[12] 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.

[13] Andrew Forward. Software documentation – building and maintaining artefacts of communication. Master's thesis, University of Ottawa, 2002.

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

[15] Project Management Institute. Pim's pulse of the profession. Technical report, Project Management Institute, mar 2012.

[16] Robert C. Martin. *Clean Code - a Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.

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

[18] Mauro Pezzè and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

[19] 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.

[20] 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.

[21] 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.

[22] Phil Stocks and David A. Carrington. A framework for specification-based testing. *IEEE Trans. Software Eng.*, 22(11):777–793, 1996.

[23] 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.

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

[25] Adam Trachtenberg. Php web services without soap. Online, oct 2003.

[26] 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.

[27] 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.

[28] 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.