

**UNIWERSYTET GDAŃSKI**  
**Wydział Matematyki, Fizyki i Informatyki**

**Artur Rybak**

nr albumu: 179796

# **Projektowanie RESTful API na potrzeby aplikacji mobilnych**

Praca magisterska na kierunku:

**INFORMATYKA**

Promotor:

**dr Włodzimierz Bzył**

Gdańsk 2015



## Streszczenie

Praca jest o komunikacji między urządzeniami mobilnymi a serwerem, który narzuca projekt interfejsu w postaci WebAPI. Przeanalizowałem konsekwencje, które niesie za sobą obranie podejścia RESTful jako wzorca projektowego w przypadku, gdy grupa klientów ograniczona zostanie do aplikacji mobilnych, tj. takich, które uruchamiane są z poziomu urządzeń przenośnych (telefonów, tabletów, zegarków, opasek, odzieży). Okazało się, że zastosowanie tego wzorca, wymusza odseparowanie z istniejącej aplikacji webowej logiki prezentacji i funkcjonalności serwowania danych. Efektem takiego podziału jest ograniczenie duplikacji kodu - od których zaczęły się problemy i dyskusje. Z nowopowstałego bytu docelowo mogą korzystać niezależni deweloperzy, a właściwie działały deweloperskie klientów naszej firmy, z wykupionymi subskrypcjami (z reguły duże firmy z rynków energetycznych, chemicznych, motoryzacyjnych, nowych technologii). API ma szansę stać się odrębnym produktem.

W poniższej pracy przedstawiłem argumenty świadczące o tym, że proces projektowania i struktura interfejsu musi zostać podparta przypadkami użycia, zaś syntaktyka powinna wpasowywać się w intuicję programistów. Moja propozycja rozwiązania problemu polega na wydzieleniu niewielkiego wycinka funkcjonalności i udostępnieniu API jako osobnego serwisu. Dzięki temu udało się w zapewnić interfejs o strukturze wpasowującej się we wszystkie nasze istniejące aplikacje, a co za tym idzie - brak spójności i duplikacje zostały ograniczone do minimum.

Aplikacja webowa serwująca API została stworzona w oparciu o lekki framework *Nancy* dla platformy .NET. Ponieważ pozostałe produkty firmy również korzystają z tej platformy, łatwiej jest integrować poszczególne serwisy. Źródła można znaleźć na dołączonej płytce oraz w repozytorium pod adresem <http://github.com/wedkarz/mgr-nancy-demo>. Instrukcja uruchomienia aplikacji znajduje się w dodatku A. Do hostowania usługi użyłem kontenerów *Docker* wyposażonych w środowisko uruchomieniowe *Mono* i chmury *Amazona*. To rozwiązanie zapewniło łatwość deployowania i zarządzania instancjami aplikacji. Z pomocą przyszły tu narzędzia konsolowe *ElasticBeanstalk*. W osobnej instancji zapewniłem źródło danych - bazę MSSQL 2008, dzięki czemu oba komponenty zyskały niezależność i możliwość skalowania, a także balansowania ruchu.

## Słowa kluczowe

WebAPI, RESTful, aplikacje, mobilne, telefony, komunikacja, wzorce, serwer, żądanie, interfejs, API, HTML

# Spis treści

<b>Wprowadzenie</b>	7
<b>1. Rozumienie WebAPI</b>	9
1.1. Jakie API jest użyteczne?	12
1.2. Praca z Web API	14
1.2.1. Podstawowa struktura składniowa	17
1.2.2. Inspekcja i symulowanie żądań przychodzących i wychodzących	19
1.3. API First	22
1.3.1. API jako produkt poboczny	23
1.3.2. API jako jeden z wielu interfejsów	24
1.3.3. API first	25
1.3.4. API w codziennej pracy	28
<b>2. Projekt WebAPI</b>	33
2.1. Wartość biznesowa	37
2.2. Budowa poprawnego modelu	39
2.2.1. Rzecznowniki - zasoby, kontra czasowniki - akcje	40
<b>3. Wykonanie WebAPI</b>	45
3.1. Implementacja na wybranej platformie	46
3.2. Składnia zapytań i odpowiedzi przy użyciu protokołu HTTP	49
3.2.1. Konstrukcja adresów URL	49
3.2.2. Filtrowanie, sortowanie, przeszukiwanie tekstu	50
<b>Zakończenie</b>	53
<b>A. Aplikacja RESTful Web api</b>	55
<b>Bibliografia</b>	57

<b>Spis tabel</b> . . . . .	59
<b>Spis rysunków</b> . . . . .	61
<b>Oświadczenie</b> . . . . .	67

# Wprowadzenie

*The number one benefit of information technology is that it empowers people to do what they want to do. It lets people be creative. It lets people be productive. It lets people learn things they didn't think they could learn before, and so in a sense it is all about potential.*

- Steve Ballmer, Microsoft CEO

Od trzech lat wnikliwie obserwuję współczesny świat przez pryzmat telefonu i jego możliwości. Dotychczas dostrzegłem różne próby komunikacji, gdzie jako główne medium służy smartfon. Jego pierwotna funkcja (możliwość wykonywania połączeń głosowych za pośrednictwem sieci GSM) jest dobrze zdefiniowana i wykorzystana, natomiast w zakresie funkcjonalności jest ona mało rozszerzalna. Krótkie wiadomości tekstowe urozmaiciliły formę komunikacji. Jednakże prawdziwą rewolucję przyniósł tani, mobilny Internet oraz upowszechnienie się darmowych punktów dostępowych sieci WiFi. Procentowy udział w ruchu internetowym, który jest generowany przez platformy mobilne wzrasta wraz z upływem czasu. Dzieje się to za sprawą coraz doskonalszych aplikacji, przemyślanych interfejsów i wygodzie dostępu do informacji, z której korzystamy, będąc w podróży lub odpowiadając po ciężkim dniu na kanapie, bo i ta nie jest naturalnym środowiskiem, w którym używamy komputera stacjonarnego czy laptopa.

Jako deweloper aplikacji mobilnych jestem świadomy, że sukces napisanych przeze mnie programów nie ma jednego źródła. Składają się na niego: przemyślany i estetyczny design aplikacji, rozpoznanie, zrozumienie i prawidłowe modelowanie potrzeb użytkownika oraz przypadków użycia. To i tak jedynie wierzchołek góry lodowej. Natomiast doświadczenie podpowiada, że dobrze zaprojektowana komunikacja klient (aplikacja mobilna) - serwer (w postaci WebAPI) wiele upraszcza i pozwala się skupić na problemach funkcjonalnych. Tym samym oszczędza czas, który zazwyczaj poświęcany jest na kolejne techniczne *gotchas*. Postanowiłem zatem wyciągnąć wnioski z toczących się w firmie rozmów i dokonałem próby przekucia tychże w namacalną formę API.

W ciągu kilku lat swojego istnienia produkt, nad którym pracuję (platforma

integrującą serwisy kupowanych przez firmę spółek) przeszedł kilka metamorfoz. Często z powodu braku decyzyjności, czystej wizji, wyobraźni albo doświadczenia ów twór zyskiwał kolejną „poskręcaną odnogę”, którą czasami udawało się nastawić albo zostawała po prostu odcinana. Taką odnogą było stworzenie (oprócz istniejącego serwisu *webowego*) natywnej aplikacji mobilnej. Traktowana jako wabik dla inwestorów i *nice-to-have* dla klientów, nie była nigdy na świeczniku. Stąd implementacja dostarczająca dane dla tabletów, która dotychczas była traktowana po macoszemu, wrosła w strukturę serwisu web. W konsekwencji pojawiły się duplikacje kodu, problemy z utrzymaniem i zapewnieniem niezależności kontraktów (rozumianych tu jako przesyłane modele) oraz utrudnienia związane z różnym cyklem życia aplikacji natywnych i webowych.

Wszystkie nasze rozmowy mające na celu poprawić sytuację sprowadziły się do dwóch skrajnych podejść: **rozbicia serwisu web na pomniejsze „mikroserwisy”** dające *de facto* rozbicie integrującej platformy oraz **stworzenia API**, które byłoby wspólnym, jednolitym interfejsem odseparowującym warstwę logiki od prezentacji. Poniższe rozdziały w moim odczuciu mogą służyć jako argumenty za obraniem drugiego podejścia. Począwszy od zrozumienia, gdzie stawiać znaki podziału i jak konstruować poprawne syntaktycznie i semantycznie API aż do wydzielenia i zmierzenia konkretnej, przeprojektowanej funkcjonalności.

W poniższej pracy korzystałem z opracowania Kirsten Hunter [1] oraz testów przeprowadzonych przez niezależnych twórców aplikacji. Przykłady wykorzystują znane modele API *Facebooka*, *Twittera* czy *Google'a* oraz moje własne zasoby i doświadczenie z pracy w firmie. Tam, gdzie było to możliwe, zmierzone zostały czasy żądań i wielkości odpowiedzi, a także czas oczekiwania na wykonanie zapytań SQL. Miało to na celu udowodnienie, że za pomocą kilku dobrych wzorców jesteśmy w stanie zapewnić całkiem wydajny i responsywny interfejs.

## **ROZDZIAŁ 1**

# **Rozumienie WebAPI**

Zanim skupię się na kwestiach technicznych oraz na projektowaniu API, warto rozważyć, jaki jest cel stworzenia możliwości do manipulowania posiadanymi zasobami za pomocą zewnętrznego interfejsu. Pytanie wydaje się być zgoła podstawowe. Istotne jest jednak to, aby wracać do niego w miarę często w trakcie projektowania lub implementacji. Łatwo bowiem zgubić ideę, zanim do pojawią się problemy techniczne lub dylematy związane ze strukturą interfejsu.

Projektowanie API nie jest wyjątkiem. Podobnie jest w przypadku niniejszej pracy magisterskiej, a także niemal każdego pomysłu na biznes. Cel powinien nie tylko uświetać środki, ale i wyznaczać dalszą drogę. Powinno się mieć go zatem zawsze na uwadze.

By zrozumieć, w jaki sposób projektować API, należy najpierw zrozumieć, jakie są wymagania i oczekiwania, które są stawiane przed jego twórcami.

**"Wielu chciało GROM rozwiązać. Bo był inny, a w naszym kraju na inność patrzy się podejrzliwie..."**

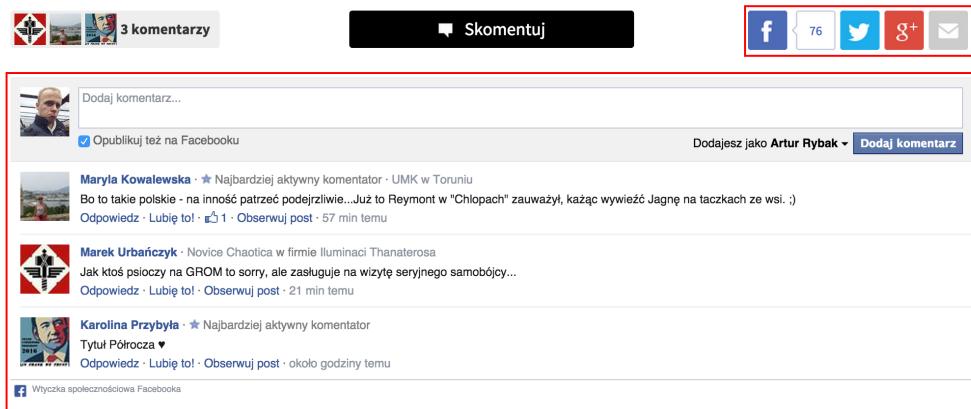


Rysunek 1.1: Strona artykułu w serwisie *na:temat* z przyciskami społecznościowymi

**Interpretacja:** Jak wiele podobnych stron informacyjnych *na:temat* oferuje przyciski społecznościowe, z których każdy powiązany jest z API serwisu macierzystego, tutaj: *Facebook*, *Twitter*, *Google+*

**Źródło:** *na:temat* [online], [dostęp: 16 czerwca 2015] dostępny w internecie: <http://natemat.pl/145775>

Na przykładzie rysunku 1.1 można dostrzec, że twórcy serwisu *na:temat* umożliwiają swoim czytelnikom rozpowszechnianie treści za pomocą przycisków społecznościowych. Jest to jeden z najprostszych sposobów na budowanie zasięgu treści i docieranie do szerokiego grona odbiorców (exposure). W ten sposób autor artykułu lub wydawca stawia na treściwe i wzbudzające emocje artykuły, z którymi czytelnik się utożsamia. Co za tym idzie - sam staje się przekaźnikiem, poprzez który serwis dociera do znajomych i obserwatorów swoich dotychczasowych odbiorców.



**Rysunek 1.2:** Sekcja komentarzy do artykułu w serwisie *na:temat*

**Interpretacja:** Niektóre z serwisów zrzucają całe funkcjonalności i interakcje z użytkownikami na baki API społecznościowych. Tutaj: wykorzystanie formularza komentarzy zintegrowanego z *Facebookiem*

**Źródło:** *na:temat* [online], [dostęp: 16 czerwca 2015] dostępny w internecie: <http://natemat.pl/145775>

*na:temat* postanowił oddać w ręce zewnętrznego serwisu całą sekcję komentarzy, która nieudolnie powielana jest przez wiele portali. Jak można zaobserwować na rysunku 1.2, serwis zyskuje gotowe rozwiązanie, z całym dobrodziejstwem inventarza, ale i pewnymi ograniczeniami. Zdecydowanie rzadziej pojawiają się tu komentarze o charakterze wulgarnym lub obraźliwym, ponieważ każdy użytkownik sygnuje je swoim imieniem i nazwiskiem. W ten sposób ograniczona została moderacja, a jej ciężar spoczywa na barkach Facebooka. Minusem tego rozwiązania jest mała elastyczność wyglądu tej sekcji i brak kontroli oraz niechciane treści reklamowe, które Facebook może dołączać do swojego produktu.

## 1.1. Jakie API jest użyteczne?

Dobrze zaprojektowane API pozwala deweloperom na tworzenie aplikacji  *mashup*, tzn. takich, które łączą w sobie cechy kilku serwisów. Z pewnością istnieje uzasadnienie dla istnienia aplikacji, które oferują funkcjonalność specyficzną dla ich dziedziny. Nie potrzebują one elastyczności i gdyby zastosować porównanie do garniturów, najtrajniej byłoby w tym przypadku użyć określenia „skrojone na miarę”. Takie aplikacje mogą bez problemów komunikować się z interfejsami bazującymi na akcjach lub za pomocą SOAP API. Pewną nadmiarowością byłoby także tworzenie generycznego serwisu tylko na potrzeby jednej aplikacji. Niesie to za sobą wymierne korzyści, gdy tworzymy system mocno powiązany. Jednocześnie pomaga to *ścinać zakręty* i skraca proces powstawania aplikacji.

Jeśli jednak chcemy pobudzić społeczność i dać jej narzędzie, które będzie mogła kreatywnie wykorzystać, to wystawienie REST API może się okazać najtrajniejszym wyborem. Do najbardziej rozchwytywanych API należą te, które oferowane są przez serwisy społecznościowe, tj. *Facebook*, *Twitter* i *LinkedIn*. Deweloper, który otrzymuje takie narzędzie, posiada natychmiastowy dostęp do danych o milionach (a nawet miliardach) użytkowników. Oczywiście „za darm” otrzymujemy jedynie dane publiczne, tzn. takie, które użytkownik sam zgodził się udostępniać zawsze i wszystkim internautom. Istnieje jednak kilka sposobów, by poprosić użytkownika o bardziej wrażliwe i szczegółowe informacje. Wiąże się to z reguły z autentykacją (m.in OAuth) i założeniem konta w serwisie, który będzie potrafił jednoznacznie określić to, która aplikacja odpowiada za aktywność w serwisie. Przydaje się to szczególnie wtedy, gdy twórcy aplikacji zaczynają obchodzić regulamin lub wprost go lekceważyć i wykorzystują naiwność, słabość i ciekawość internautów w celu pozyskania cennych informacji marketingowych, pieniędzy lub udostępnieniem możliwości do przesyłania spamu. Twórcy API, którym zależy na budowaniu pozytywnego wizerunku swojego produktu, mogą w takim wypadku uniknąć tego rodzaju sytuacji poprzez włączenie dostępu do API dla konkretnych aplikacji. Zdarza się również tak, że w przypadku serwisów o największej liczbie wizyt jest to proces zautomatyzowany, który sterowany jest za pomocą algorytmów z dziedziny sztucznej inteligencji.

## MOST POPULAR APIs

1. Facebook	<a href="#">Track this API</a>	6. LinkedIn	<a href="#">Track this API</a>
2. Google Maps	<a href="#">Track this API</a>	7. Kayak	<a href="#">Track this API</a>
3. Skype	<a href="#">Track this API</a>	8. Waze	<a href="#">Track this API</a>
4. Netflix	<a href="#">Track this API</a>	9. Yahoo Weather...	<a href="#">Track this API</a>
5. Telegram	<a href="#">Track this API</a>	10. Pinterest	<a href="#">Track this API</a>

Rysunek 1.3: Ranking najpopularniejszych API z których korzystają deweloperzy.

**Interpretacja:** Najpopularniejsze serwisy webowe oferują najbardziej rozchwytywane API. Zarówno ze względu na bogatą treść jaką za ich pomocą „wyciągnąć” ale i środki jakie czołowi gracze inwestują w rozwój swoich platform. Owocuje to przemyślanym i responsywnym interfejsem a także znaczną penetracją rynku.

**Źródło:** ProgrammableWeb [online], [dostęp: 16 czerwca 2015] dostępny w internecie:  
<http://www.programmableweb.com/apis>

## 1.2. Praca z Web API

Praca z Web API wymaga od programisty zrozumienia platformy - jej struktury danych, sposobu interakcji, kroków wymaganych do wydobycia lub przetworzenia interesujących go w danym momencie informacji. Niezbędne jest zapewnienie w takim wypadku miejsca, gdzie zagubiony programista będzie mógł usystematyzować swoją wiedzę o możliwościach, ograniczeniach, wymaganiach i najprostszych sposobach korzystania z danego API. Wszelkie narzędzia, przykładowy kod, tutoriale, samouczki są nieodzowną pomocą w takich sytuacjach. Często, w przypadku rzędu wielkości Facebooka strony deweloperskie urastają do całkiem sporego rozmiaru, tworząc cały ekosystem. W ten sposób powstaje wirtualna społeczność (virtual community), wśród której możemy bez skrępowania zadawać pytania i poszukiwać odpowiedzi.

Rysunek 1.4 przedstawia narzędzie deweloperskie stworzone przez Twilio. Jest to firma, która zajmuje się dostarczaniem API dla rozmów telefonicznych i SMSów. Celem Twilio jest, by każdy, kto odwiedza stronę główną mógł w ciągu 5 minut wykonać testową rozmowę lub wysłać smsa za pomocą oferowanego API. To doskonały sposób na zaangażowanie potencjalnych klientów i deweloperów, którzy jak się okazuje - z niezwykłą łatwością mogą korzystać z dobrodziejstw SMSów i rozmów głosowych w swoich produktach. Być może także początkowy plan by tylko 5-10 minut rozejrzeć się na tronie Twilio zaowocuje w programiście chęcią zainwestowania czasu, który pozwoli mu na poznanie i zrozumienie „co oni mogą dla mnie zrobić?”.

## Request

The screenshot shows a web-based interface for making a POST request to the Twilio API. At the top, there are tabs for Curl, Ruby, PHP, Python, Node.js, Java, and C#. Below the tabs, a note says "Toggle showing your Auth Token by clicking here" and "Check out the curl manpage". A code block contains a curl command:

```
curl -X POST 'https://api.twilio.com/2010-04-01/Accounts/ACb92118e87631a105a67f326a74d508ff/Messages.json' \
--data-urlencode 'To=+48[REDACTED]8' \
--data-urlencode 'From=+48[REDACTED]8' \
--data-urlencode 'Body=Testing API for master thesis' \
-u ACb92118e87631a105a67f326a74d508ff:[AuthToken]
```

At the bottom, a red button says "Make Request". Below it, a note says "This request costs money. You can find pricing information [here](#)".

Rysunek 1.4: Podgląd pierwszego testowego żądania z Twilio API

**Interpretacja:** Twilio umożliwia „wyklikanie” i wypełnienie formularza online, który zostaje „w locie” przetłumaczony na żądanie. Możemy zobaczyć przykładową implementację w kilku najbardziej popularnych językach programowania a także za pomocą programu *curl*.

**Źródło:** własne

Wysłanie pierwszego SMSa za pomocą Twilio API zajęło mi około 7 minut. To prawdopodobnie wynik poniżej oczekiwania firmy. Niemniej, cel został osiągnięty, a także pojawiło się kilka pomysłów na to, w jaki sposób takie API można byłoby wykorzystać. Przykładowe wykorzystanie zostało zaprezentowane na Rysunku 1.5. Ponadto w informacji zwróconej otrzymujemy kod informujący o utworzonej encji (201) CREATED oraz samą encję w postaci JSONa.

Tak łatwe wprowadzenie jak w przypadku Twilio, generuje wymierną wartość marketingową. W pozostałych sytuacjach deweloper zastanawiałby się nad tym, dlaczego w ogóle ma sobie tym zaprzątać głowę. W trakcie poszukiwania zasadności i sposobu utworzenia tego typu rozwiązania, natknąłby się nie na kolejne przeszkody a nie klocki, budulce jego przyszłych aplikacji.

## Response 201

CREATED - The request was successful. We created a new resource and the response body contains the representation.

```
{  
    "sid": "SM1b2e48d4cca24833949d1a0cfb85c654",  
    "date_created": "Mon, 24 Aug 2015 13:40:19 +0000",  
    "date_updated": "Mon, 24 Aug 2015 13:40:19 +0000",  
    "date_sent": null,  
    "account_sid": "ACb92118e87631a105a67f326a74d508ff",  
    "to": "+485 [REDACTED] 8",  
    "from": "+487 [REDACTED]",  
    "body": "Testing API for master thesis",  
    "status": "queued",  
    "num_segments": "1",  
    "num_media": "0",  
    "direction": "outbound-api",  
    "api_version": "2010-04-01",  
    "price": null,  
    "price_unit": "USD",  
    "error_code": null,  
    "error_message": null,  
    "uri": "/2010-04-  
01/Accounts/ACb92118e87631a105a67f326a74d508ff/Messages/SM1b2e48d4cca24833949d1a0cfb85c654.json",  
    "subresource_uris": {  
        "media": "/2010-04-  
01/Accounts/ACb92118e87631a105a67f326a74d508ff/Messages/SM1b2e48d4cca24833949d1a0cfb85c654/Media.json"  
    }  
}
```

Rysunek 1.5: Odpowiedź z API Twilio, która potwierdza wysłanie testowego SMSa

**Interpretacja:** W odpowiedzi widzimy, czego możemy się spodziewać w zwrocie z Twilio API. Warto zwrócić uwagę na to, że nawet bez zagłębiania się w strukturę łatwo domniemać znaczenie poszczególnych pól.

**Źródło:** własne

### 1.2.1. Podstawowa struktura składniowa



Rysunek 1.6: Elementy składowe żądania HTTP

**Interpretacja:** Podstawowa struktura żądania zawiera informację o nagłówkach, metodzie, URLu (adresie) oraz ciele.

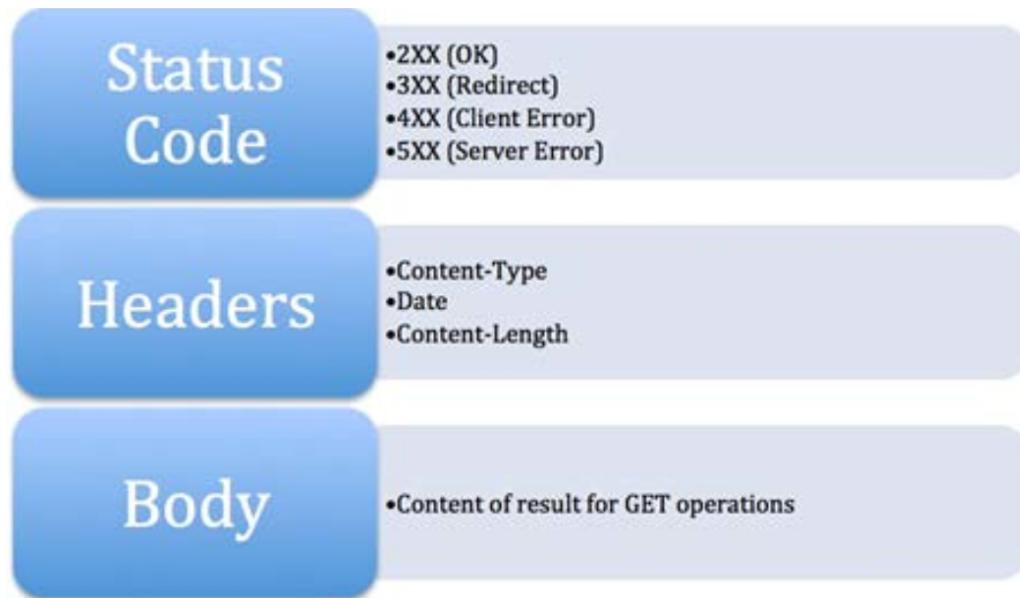
**Źródło:** [1], wersja IV, s. 27

W przypadku API można mieć na uwadze skomplikowane systemy, natomiast Web API są interfejsami tak prostymi, jak to tylko możliwe. Podstawowymi oferowanymi przez nie akcjami są *CRUD* - Create Read Update Delete. Ponadto interfejsy Web API wykorzystują cechy protokołu HTTP, a co za tym idzie – można się spodziewać, że żądanie (tak jak na rysunku 1.6) będzie zawierało informacje o adresie, nagłówkach, metodzie oraz - w przypadku metod PUT/PATCH/POST - ciele żądania. Oczywiście, jest to tylko konwencja i nikt nie zabrania obsługi żądań na własną rękę w zupełnie dowolny sposób. Należy jednak pamiętać, że jest to niezgodne z intuicją i powszechną praktyką. W związku z tym utrudnione jest zrozumienie intencji danego API i wzrasta „próg wejścia” który wymagany jest do tego, by z niego korzystać. Największe zdziwienie może tutaj wzbudzić odniesienie do

metody PATCH, która nie była pierwotnie częścią standardu HTTP. Została jednak wprowadzona dokumentem RFC 5789 [2] w marcu 2010 roku. Jest ona szczególnie ważna ze względu na platformy mobilne, których zadaniem jest minimalizacja transferu.

Postępowanie według wzorca pozwala deweloperom poczynić pewne założenia. Jak chociażby takie, że metoda POST stworzy nowy zasób, PUT uaktualni już istniejący, PATCH uaktualni niektóre właściwości/wartości zasobu, GET dokona odczytu, a DELETE usunie go całkowicie. Programista będzie również domniemywać unikalności zasobu występującego pod danym adresem.

Założenia co do wartości zwracanych również będą bazować na protokole HTTP, tj. na kodzie zwracanym, nagłówkach oraz ciele odpowiedzi (Rysunek 1.7).



Rysunek 1.7: Element składowe odpowiedzi na żądanie HTTP

**Interpretacja:** Ustrukturyzowana odpowiedź na żądanie HTTP zawiera kod HTTP informujący o statusie, nagłówki oraz w niektórych przypadkach ciało.

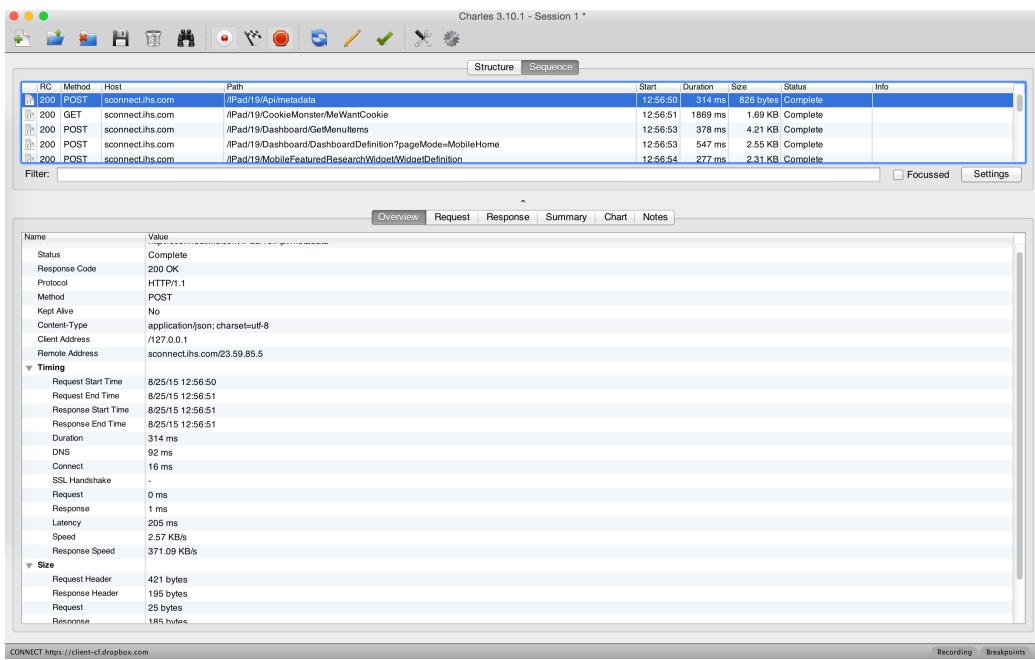
**Źródło:** [1], wersja IV, s. 28

### 1.2.2. Inspekcja i symulowanie żądań przychodzących i wychodzących

Codzienna praca z API, nawet jeśli nie jest ono dopiero rozwijane, wymaga użycia narzędzi, które umożliwiają podgląd żądań i odpowiedzi wraz z wszelkimi metadanymi. Wiele różnych побudek determinuje wybór odpowiedniego narzędzia. Jednak w niniejszej pracy posłużę się tym samym zestawem narzędzi, z którego korzystam w codziennej pracy nad aplikacjami mobilnymi.

Po pierwsze **Charles** [3]. Jest to proxy, poprzez które przechodzą wszelkie żądania z lokalnej maszyny a także wszelkie żądania urządzeń, symulatorów i emulatorów, na których ustawione zostało proxy w taki sposób, by wskazywało na maszynę z zainstalowanym Charlesem. Domyślnie program instaluje się na porcie 8888 i cechuje go prosty i elegancki wygląd, a także duże możliwości konfiguracji - ograniczanie śledzonych żądań do wskazanych domen; możliwość symulowania niewydajnego łącza internetowego; stawianie breakpointów (zarówno na żądaniach, jak na odpowiedziach) oraz ich modyfikacja „w locie”; przepisywanie żądań i odpowiedzi według ustalonych reguł, a także bardzo przydatne mapowanie (lokalne lub zdalne), które polega na odsyłaniu do wybranego pliku lub strony w przypadku spełnienia kryteriów. W praktyce pozwala to pracować z niestabilnymi środowiskami deweloperskimi poprzez przekierowywanie niektórych żądań do środowisk stabilniejszych. Oczywiście, wszystko to wymaga rozwagi i rozumnego wykorzystania, gdyż narzędzie może okazać się mieczem obosiecznym i doprowadzić do niestabilnego działania zarówno aplikacji klienckiej, jak backendu. Alternatywami dla Charlesa mogą być *HTTPScoop* [4], *Wireshark* [5] czy *Fiddler* [6].

Na Rysunku 1.8 przedstawiono przykładowy obraz, jaki uzyskuje programista, który chce zbadać parametry żądań wysyłanych przez aplikację oraz wszelkie detale dotyczące struktury i czasu wykonania. Umożliwia to łatwe wychwytywanie błędów, literówek, nadmiarowości, a także jest doskonałym narzędziem do profilowania czasu oczekiwania na odpowiedź. Dzięki temu widoczne staje się, jakie żądania są obciążające dla naszej aplikacji i gdzie należy szukać usprawnień w wydajności.



Rysunek 1.8: Charles podczas pracy z jednym z API

**Interpretacja:** W górnej części znajdują się chronologicznie wykonywane żądania, które można zaznaczyć, by u dołu otrzymać szczegóły. Sposób prezentacji w poszczególnych zakładkach pozwala na czytelny, ukierunkowany obraz, gdzie skupić możemy się na parametrach żądania, parametrach odpowiedzi, nagłówkach, sposobie autentakcji, ciasteczkach, wykresach związanych z długością trwania poszczególnych żądań.

**Źródło:** własne

Po drugie **cURL** [7] i **Apache™ JMeter**. Funkcja, której najbardziej brak w przypadku Charlesa, to możliwość łatwego komponowania żądań - ustawiania ich parametrów, zapisywanie i tworzenie scenariuszy. W przypadku prostych rozwiązań wybrałem wbudowany w każdy uniksowopochodny system *cURL*. Wzięty z życia przykład żądania wygląda jak na listingu 1. Przy czym parametry \$1 i \$2 to login i hasło, które z powodu wymaganej poufności zostały ukryte. W zwrocie otrzymujemy JSONa wraz informacjami statusie, co obrazuje Rysunek 1.9.

### Listing 1. curl - podstawowe użycie

```

1 curl -v --basic --user $1:$2
→ https://sconnect.ihs.com/IPad/19/Dashboard/DashboardDefinition?pageMode=MobileHome

```

Konstrukcja tego szczególnego żądania, jak i całego API, którego używam by najmniej nie należy do wzorów i nie powinna być naśladowana. W dalszych rozdziałach uzasadnię powyższe twierdzenie.

```
curl -v --url http://127.0.0.1:5000/api/listings
$ ./curl_1.sh
> Trying 23.59.85.5...
> Connected to sconnect.ihs.com (23.59.85.5) port 80 (#0)
> Server auth using Basic with user "XXXXXXXXXX"
> GET /Ipad/19/dashboard?dashboardDefinition?pageMode=MobileHome HTTP/1.1
> Host: sconnect.ihs.com
> Authorization: Basic XXXXXXXXXXXXXXXXXX
> User-Agent: curl/7.43.0
> Accept: */*
< HTTP/1.1 200 OK
< Cache-Control: private, s-maxage=0
< Content-Type: application/json; charset=utf-8
< X-Powered-By: ASP.NET
< Date: Tue, 25 Aug 2015 18:09:47 GMT
< Content-Length: 4260
< Connection: keep-alive
< Set-Cookie: IHS_SSO_SESS=0sx2FVBgCBAlCAQjZMCKMsZyqPGoYbsRX-cx2dR0x2Bpv4x2B918x30roPw4dBcIC65j0n5J3E30x30x30-2F4b0pFIWGNzAmRcrAalwx30x30-TRBuJy8S019imjh0I2FkXa30x30;
< Set-Cookie: IHS_SSO_UU=x13z-i1pn2PAPh3-/rEMl75po7SndNSNxTu7/KSbg5qlbxAMSb7jDWRZy830f; domain=.ihs.com; path=/; HttpOnly
< Set-Cookie: IHS_CONNECT_SESS_STAGE=xhFR0q1PmG0HhQD9p9x0A93c93c92c2hJ0nOrvKKd5JH0wIRScSm382ba1XtxAH05wZt082hU5cb5WTXNl3BgcrcRNKw9cbs9YHRd6jZ2npz6zNDNTQCQ1qmz3M2W7oW
EMwRds2blC07rePTF7z2h005c2bx9b9d0hCUT0c88ID2r2W0r11hpMBpdtLppwrf6ImVPvxz2h9zFhSwEv18Vm9wZhaufgcaia1qJN0ZgxhUGRoFE152h69JPKPGEk5wdkR6arBrxG9sozCEpDFM0lWw0oX1VbHZ
Umdbh8p8d2gtW3dn0fIAKWr1DvBDMPt2PostSUK35h5cbVlwj0134jW1xkFNKCRHjBu0PGCrPs6f0WtWZczo13y0v0v4eU93fkjZompeBhP1lk0pWj0DheeZE0vP30R140jEs2bvhehosu7TNP1AFjXByukMVWTPV3B
1k1M8SbszJ0FU9rxdec0U0%2FLfJz1H5s2baUXHZ3MKRyW6GCT1TW3FwAMPd7nnHtqzY2FFJGnrlhwqs3d; domain=.ihs.com; expires=Tue, 25-Aug-2015 18:39:46 GMT; path=/; HttpOnly
<
{"Data": {"TaxonomyDefinition": {"TaxonomyTypeMobileDefinitions": [{"DependentOn": null, "Description": null, "Min": null, "Max": null, "Name": "Domain", "PresentationStyle": "List", "TaxonomyKey": "d", "Final": false, "ExcludeFromCurrentSelectionDescription": true, "ExcludeFromInitialMessage": true}, {"DependentOn": null, "Description": null, "Min": null, "Max": null, "Name": "Region", "PresentationStyle": "List", "TaxonomyKey": "r", "Final": false, "ExcludeFromCurrentSelectionDescription": true, "ExcludeFromInitialMessage": true}, {"SelectEdtTaxonomy": {"d": {"Id": "A1", "Name": "All"}, "EntitledTaxonomy": "MobileSectorAndRegion", "Final": false}, {"PageMode": "MobileHome", "Name": null, "Folded": true, "SavedDashboardId": null, "SubMenuDefinition": null, "AvailableWidgets": [{"Id": "MobileFeaturedResearchWidget", "Title": "Featured Research", "IsOpen": true, "IsTemplated": true, "ShowTitle": true, "BorderMargin": true}, {"Id": "MobileHotTopicsWidget", "Title": "Hot Topics", "IsOpen": true, "IsTemplated": false, "ShowTitle": true, "BorderMargin": false}, {"Id": "MobileHeadlinePerspectiveWidget", "Title": "Headline Perspective", "IsOpen": true, "IsTemplated": false, "ShowTitle": true, "BorderMargin": true}, {"Id": "MobileAnalysisWidget", "Title": "Analysis", "IsOpen": true, "IsTemplated": false, "ShowTitle": true, "BorderMargin": true}, {"Id": "MobileHeadlineWidget", "Title": "Headline", "IsOpen": true, "IsTemplated": false, "ShowTitle": true, "BorderMargin": true}, {"Id": "MobileLatestResearchWidget", "Title": "Latest Research", "IsOpen": true, "IsTemplated": true, "ShowTitle": true, "BorderMargin": true}, {"Id": "MobileMostPopularWidget", "Title": "Most Popular", "IsOpen": true, "IsTemplated": true, "ShowTitle": true, "BorderMargin": true}, {"Id": "MobileMostPopularOnConnect", "Title": "Most Popular on Connect", "IsOpen": true, "IsTemplated": true, "ShowTitle": true, "BorderMargin": true}], "LayoutDefinition": [{"Id": "A", "Children": [{"Id": "A1", "Children": [{"Id": "A1", "Children": [{"SplitRatio": 100, "HorizontalSplit": true, "WidgetId": "MobilefeaturedResearchWidget"}, {"SplitRatio": 60, "HorizontalSplit": true, "WidgetId": "MobileheadlinePerspectiveWidget"}], "HorizontalSplit": true, "WidgetId": "MobileheadlineWidget"}]}]}]}]
```

Rysunek 1.9: Wynik działania programu z listingu 1.

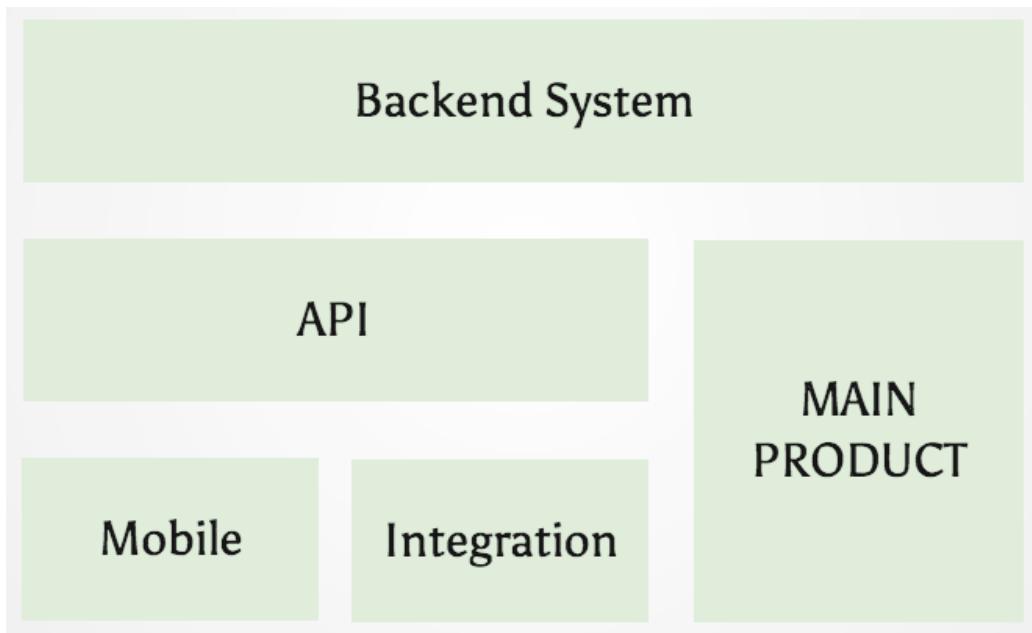
**Interpretacja:** W odpowiedzi otrzymaliśmy ciało w postaci json'a oraz informacje o statusie.

**Źródło:** własne

### 1.3. API First

*API first* to strategia planowania całej linii produktów danej firmy, gdzie API są podstawą każdego produktu, w przeciwieństwie do bycia osobnym, pobocznym produktem. By zrozumieć, dlaczego *API first* jest dobrym pomysłem, należy najpierw zapoznać się z uzasadnieniem celu tworzenia API. Istnieje kilka modeli tworzenia interfejsów. Jednak generalizując, API serwuje to, co otrzyma z systemu stanowiącego backend i komunikuje się z nim bezpośrednio, równolegle z innymi produktami. Wynika z tego, że jeśli celem jest stworzenie kolejnych aplikacji, należy napisać więcej systemów, które będą bezpośrednio dotykać backendu. Drugim rozwiązaniem jest rozszerzenie API w taki sposób, by wspierało oba alternatywne produkty. Co więcej, API jest często uważane za „wabik” lub „czynnik wyróżniający”, który pomaga przyciągnąć klientów. Jednak traktowane jest jako *nice-to-have*, a nie ważny składnik ekosystemu produktów w firmie. Takie podejście generuje poważne problemy, szczególnie przy uwzględnieniu zasobów, ze względu na fakt, że konkuuuje ono z produktami przynoszącymi widoczne przychody.

### 1.3.1. API jako produkt poboczny



Rysunek 1.10: API jako produkt poboczny, podejście standardowe

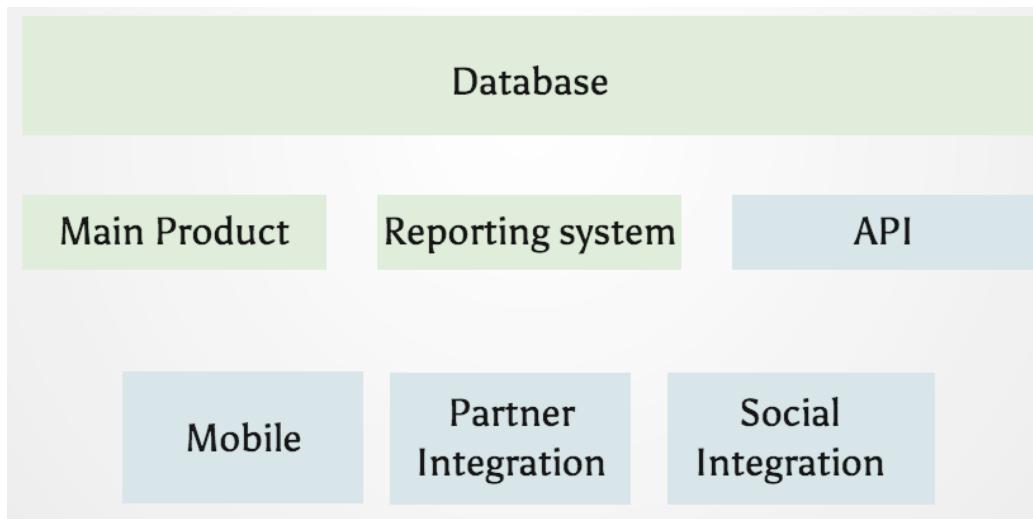
**Interpretacja:** API jest osobną ścieżką w linii produktów firmy i używane jest do zapewnienia działania jednego lub kilku integracji, czasem klientów mobilnych.

**Źródło:** własne

Rysunek 1.10 jest przykładem typowego zastosowania API w firmie. Główny produkt i produkty poboczne mają własne ścieżki. Nawet jeśli wszystkie produkty poboczne komunikują się poprzez API, wciąż widoczna będzie różnica w odbiorze i użytkowania tych aplikacji. Teoretycznie możliwe jest, by wszystkie ścieżki konsekwentnie oferowały taki sam *user experience* (UX), jednakże wymaga to większego nakładu pracy w porównaniu do restrukturyzacji, która uczyni z API jądro systemu.

Sceptycy argumentują, że odseparowanie API od głównego produktu może ochronić go przed atakami za pośrednictwem API. Rzeczywista obserwacja wskazuje jednak, że to backend jest krytycznym elementem, a proponowana separacja utrudnia jedynie naprawę błędów. Utrzymywanie wszystkiego w jednym porządku pozwala zapewnić niezawodność i spójność produktu, który się rozwija. Efektem ubocznym, acz pożądanym, jest ułatwienie skalowalności i ekspansji.

### 1.3.2. API jako jeden z wielu interfejsów



Rysunek 1.11: API jako jeden z wielu interfejsów

**Interpretacja:** API jest tylko jednym z interfejsów, jakie komunikują się z backendem i choć ma swoich klientów, to równolegle działają systemy komunikujące się z backendem w podobny lub wręcz zupełnie odmienny sposób.

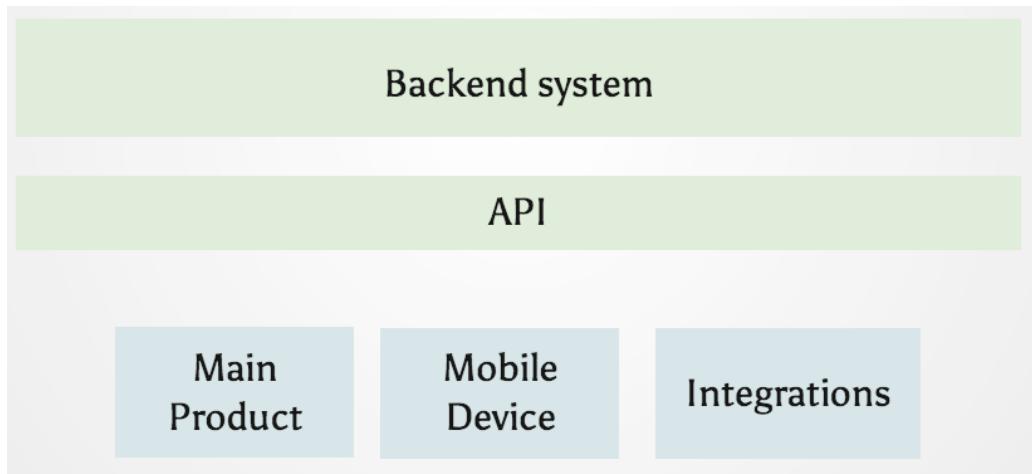
**Źródło:** własne

Podobną, choć nieco bardziej rozbudowaną architekturą jest traktowanie API jako jednego z wielu interfejsów. W obrazowanym przez Rysunek 1.11 schemacie można zauważać bardzo niepokojącą tendencję. Każdy z systemów powiela tutaj wysiłek komunikacji z backendem i duplikuje interfejsy. Już na pierwszy rzut oka można stwierdzić, że prowadzi to do powstawania *długu technicznego*. Łatwo wyobrazić sobie sytuację, gdzie w systemie backendowym dodane zostają nowe pola, dzięki czemu dostępne są dodatkowe informacje. Jednakże ze względu na podziały i różne cykle wdrożenia aplikacji nowa funkcja w najlepszym przypadku zostaje wdrożona w różnym czasie do każdego z tych systemów. O wiele gorzej i częściej zdarzają się sytuacje, że z powodu „dziurawej komunikacji” nie wszystkie systemy zostają zaktualizowane lub dzieje się to w sposób niepożądany. Gdy udało się określić, jaki system będzie najbardziej intuicyjny dla użytkowników, należy dążyć to tego, by wspierać ogólny rozwiązań w każdym miejscu.

Rzeczną architekturą objawia się szczególnie, gdy kilka zespołów próbuje two-

rzyć produkty bez jednej, spójnej wizji. U podłożu problemu zazwyczaj leży słaba komunikacja pomiędzy tymi zespołami.

#### 1.3.3. API first



Rysunek 1.12: API first

**Interpretacja:** API jest pośrednikiem pomiędzy wszystkimi aplikacjami klienckimi a backendem, stanowi warstwę bez której komunikacja między tymi systemami nie może mieć miejsca.

**Źródło:** własne

Rysunek 1.12 przedstawia model API first. Wszystkie produkty korzystają tu z jednego interfejsu. Zasoby, na których operuje API będą takie same w przypadku każdego z produktów. Co za tym idzie - system jest spójny. Warto zwrócić uwagę na fakt, że nie wymusza to na systemie tego, by wszystkie zasoby były dostępne dla każdej z aplikacji klienckich. Możemy ograniczyć dostępność zasobów do aplikacji wewnętrznych, partnerskich lub otworzyć dostęp dla wszystkich. API first wymusza komunikację między zespołem odpowiedzialnym za backend a wszystkimi zespołami frontendowymi. API stanowi tu warstwę pośredniczącą, którą każdy z tych zespołów będzie próbował pociągnąć w swoją stronę. Jednakże żadne rozwiązanie nie dojdzie do skutku, jeśli nie będzie mogło posłużyć pozostałym zespołom.

Dodatkową korzyścią jest redukcja redundancji w kodzie. Należy bowiem pamiętać, że powstają one nie tylko na potrzeby funkcjonowania produktu ale także w celach testowania i sprawdzania integralności. W przypadku schematu przedsta-

wionym na Rysunku 1.11 założmy, że zmieni się baza danych. Wymagana jest praca po stronie każdego z zespołów i weryfikacja tego, czy zmiany nie wprowadzają błędów. Systemy te mogą być skonstruowane w taki sposób, że niemal niemożliwym będzie wykrycie wszystkich zależności. Zatem centralizacja testowania i wpływu na aplikacje za pośrednictwem API przynosi oszczędności czasowe.

Z pewnością API nie rozwiąże wszystkich problemów systemu, jednak może ułatwić jego projektowanie i rozwój. By osiągnąć ten pożądanego model API, powinno być ono dobrze udokumentowane i całkowicie zabezpieczone przed dostępem klientów innymi środkami. Mając to na uwadze, będziemy można wykorzystać fakt, że:

### **1. funkcjonalności są równe na każdej platformie**

Nie ma potrzeby inwestowania czasu i angażowania osób we wdrażanie nowych funkcji na każdej z platform z osobna. Są one dobrze przetestowane, a zmiana dokonywana jest tylko po jednej stronie. Możliwe jest również przeforumułowanie (bez wpływu na pracę deweloperów) wewnętrznej implementacji systemu, z którego API czerpie.

### **2. zwiększamy prędkość**

Nieintuicyjnym może się wydawać, że dodanie warstwy do systemu może zredukować ogólny nakład pracy, którą muszą wykonać deweloperzy. Wspólny interfejs pozwala na dzielenie bibliotek i schematów. Chociaż na początku deweloperzy - szczególnie backendowi - mogą ze sceptyczmem podchodzić do API, to po pewnym czasie zauważają, że w konsekwencji mają mniej pracy w utrzymaniu systemu i dostosowywaniu go do potrzeb poszczególnych produktów.

### **3. regulujemy dostęp**

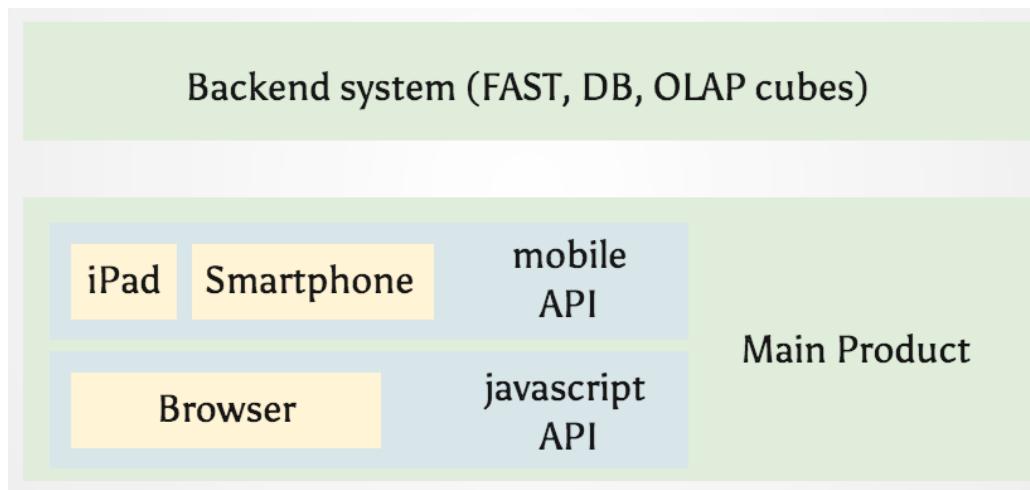
Autentykacja i authoryzacja dostępu do zasobów to narzędzia, dzięki czemu możemy decydować o tym, dla których aplikacji lub użytkowników poszczególne funkcje mają być dostępne, a dla których dostęp ten chcemy ograniczać. Jednym ze sposobów radzenia sobie z tym problemem jest implementacja OAuth. Jedną z największych zalet w tej implementacji jest możliwość blokowania złośliwych, niewydajnych lub niezgodnych z regulaminem aplikacji.

kacji. Otwarcie API dla zewnętrznych deweloperów ma także tę korzyść, że weryfikują oni użyteczność interfejsu i wykrywają na wczesnym etapie potencjalne zagrożenia. Poza tym tworzenie aplikacji dla użytkowników wewnętrznych, twórcy nie są zwolnieni z troski o to, czy system jest funkcjonalny i prosty w obsłudze.

Istnieje wiele dobrych przykładów na to, że oferowanie API jest krokiem w stronę rozwoju. *Twilio* za pomocą swojego API do rozmów głosowych i SMSów uprościło bardzo trudne zagadnienie, z którego po prostu opłaca się skorzystać. Firma ta angażuje także cały zespół *ewangelistów*, którzy uczestniczą w *hakatonach* i ułatwiają deweloperom start w ich serwisie, a także adoptują coraz to nowsze pomysły i spostrzeżenia.

*Instagram* początkowo był aplikacją jedynie mobilną. Jednak z czasem użytkownicy tego kanału zaczęli domagać się strony web i integracji. Firma nie dysponowała wówczas jeszcze zasobami, które pozwoliłyby na te zmiany. W związku z tym sfrustrowani niezależni programiści zajęli się *inżynierią wstępna* i „zhakowali” Instagram. To zmusiło Facebooka, który przejął Instagram, do upublicznienia przy najmniej części API.

#### 1.3.4. API w codziennej pracy



Rysunek 1.13: API z mojej perspektywy

**Interpretacja:** API widziane z mojej perspektywy. Zaimplementowane jako *sideproduct* w strategii firmy. Służy do komunikacji backendu i platform mobilnych, przy czym samo nie jest osobnym byteam a wrasta w główny produkt - aplikację webową.

**Źródło:** własne

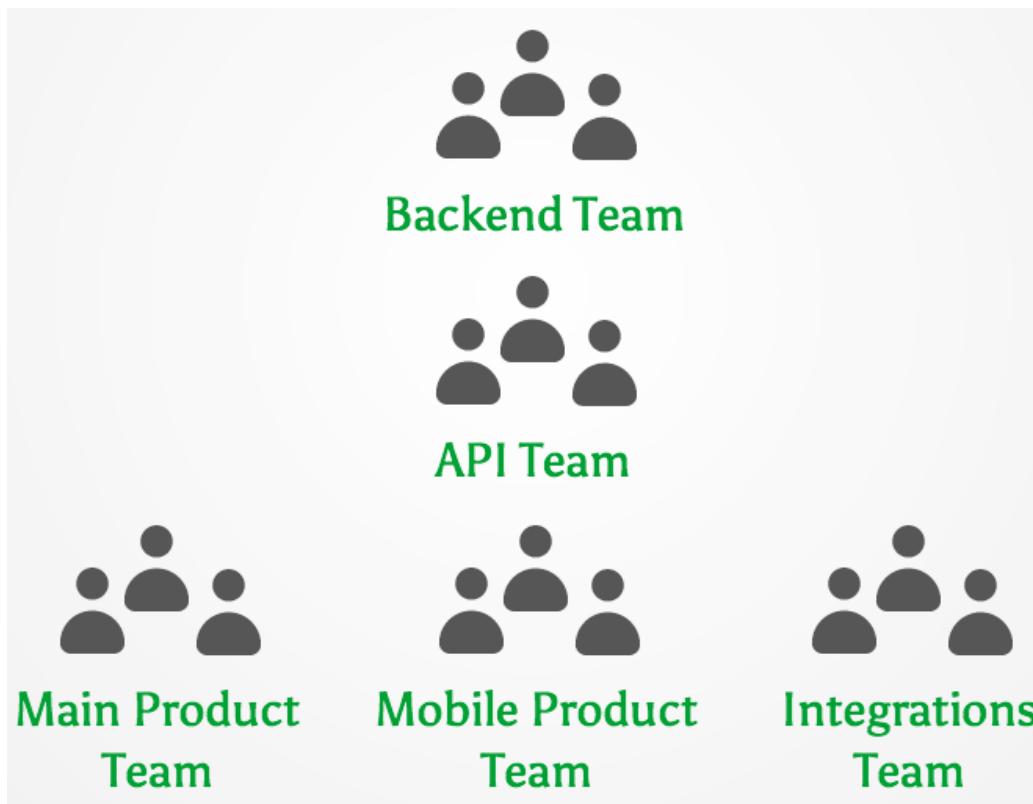
Inspiracją do napisania tej pracy stało się moje środowisko pracy, gdzie architektura wyewoluowała w sposób przewyższający standardy największych przedsiębiorstw z branży IT. Nie było to problemem, dopóki system nie zaczął się rozrastać. Tymczasem po lekturze poprzednich rozdziałów na pierwszy rzut oka widać, że można lepiej – wydajniej i bezpieczniej.

Obecną architekturę systemu, nad którym pracuję, przedstawia Rysunek 1.13. W tym wypadku przedstawiono systemy backendu, t.j. bazę danych, silnik wyszukiwania (FAST) i kostki OLAP. Z backendem komunikuje się aplikacja główna, która w praktyce korzysta z wielu niezależnych serwisów. Odgrywa ona rolę pośrednika, agregującego i korzystającego z poszczególnych serwisów. Niestety, aplikacja ta serwuje również treść użytkownikom końcowym w postaci całego portalu internetowego, a dodatkowo zaszyta jest w niej obsługa dwóch niezależnych API: mobile API, które jest utylizowane przez platformy takie jak iPad, iPhone, smartfony Androidowe; JavaScript API, które służy web portalowi do asynchronicznej komunikacji za pośrednictwem JavaScriptu.

Bardzo odczuwalny jest szczególnie podział na wersję webową i mobilną. Po pewnym czasie zaczęto dociekać, jaka może być tego przyczyna i jakie są realne konsekwencje posiadania oddzielnych API. Niestety, okazało się, że powód tkwi bardzo głęboko - mianowicie, w wewnętrznej strukturze organizacyjnej projektu. Nad obecnym rozwiązaniem pracuje 5 zespołów stricte zajmujących się wersją przeglądarkową. Ich kompetencje dublują się nieco z obowiązkami osobnego (szóstego) zespołu zajmującego się dostarczaniem backendu. Każdy z tych zespołów ma swojego *analityka biznesowego* i często (już na poziomie biznesu) brakuje zgodności i spójności.

Wraz ze swoim siódmym zespołem także znalazłem się w takiej nieszczęśliwej sytuacji. Otrzymujemy bowiem zlecenia od wszystkich analityków (a właściwie od żadnego). Tak się dzieje, gdy odpowiedzialność za produkt *nice-to-have* produkt jest zbiorowa. Próbujemy się wpasować w ramy, które zostały już ugruntowane. Co gorsza, każdy z zespołów webowych okazuje się mieć inny na te ramy pomysł, więc bardzo cierpimy z powodu braku platformowości. Wielokrotnie prostujemy *hard-kodowane* implementacje, a także dostosowujemy się do tego, by pokrętnie móc współdziałać z każdym z zespołów.

Nie jest to sytuacja komfortowa ani pożądana, a na dodatek powoduje sytuacje, w których jedna platforma degraduje drugą. Podejście API first wymusza przede wszystkim poprawny schemat komunikacyjny na zespołach. Obrazuje to Rysunek 1.14.



Rysunek 1.14: API z mojej perspektywy

**Interpretacja:** API widziane z mojej perspektywy. Zaimplementowane jako *sideproduct* w strategii firmy. Służy do komunikacji backendu i platform mobilnych, przy czym samo nie jest osobnym byteam a wrasta w główny produkt - aplikację webową.

**Źródło:** własne

Każdy z zespołów musi brać pod uwagę istnienie aplikacji zależnych podczas wdrażania nowych funkcjonalności, a tym, by API było spójne kieruje zespół explicit do tego przeznaczony. Systemy i patologie jednak bardzo szybko zwiększały swoją objętość. Trudno zatem przeprowadzić generalną reformę. Postanowiłem więc w tej pracy, jak można poprawnie skonstruować pewien bardzo mały wycinek API, który zadowoli każdą z platform. Dowodem niepotrzebnych duplikacji niech będą poniższe dwa Rysunki: 1.15 i 1.16. Pośród dzielenia i zabezpieczania własnego interesu należy zauważać, że obie implementacje korzystają z tego samego repozytorium. Różni je jedynie typ zwracany i związane z tym mapowanie. Niesie to za sobą konsekwencje w postaci różnic powstających pomiędzy modela-

mi. Można więc odważnie stwierdzić, że funkcjonalnie oba kontrolery należałoby połączyć w jeden, który serwowałby API dla wszystkich platform.

Do problemu „łączenia” należy podchodzić jednak z rezerwą. Jak się bowiem okazuje, aplikacja webowa nie ma narzutu związanego z utrzymywaniem starszych wersji API. Każdorazowe wdrożenie niesie za sobą pozbycie się starego kodu. W przypadku platformy mobilnej jest to niemal niewykonalne ze względu na obowiązkowy proces zatwierdzenia przez właściciela sklepu (np. *App Store*, który jest narucany na każdą aplikację). Poza tym im więcej komponentów od siebie zależy, tym trudniej zsynchronizować ich publikację, dlatego stosuje się wersjonowanie. Dla aplikacji mobilnej rzecz nieodzowna. Dla platformy webowej nowość, do której należy się przyzwyczaić.

```
public class ChemicalHeadlineAnalysisWidgetController
    : WidgetBaseController<ChemicalHeadlineAnalysisWidgetSetup,
                           ChemicalHeadlineAnalysisViewModel>
{
    private readonly IChemicalHeadlineAnalysisRepository _repository;

    public ChemicalHeadlineAnalysisWidgetController(
        IWidgetServicesProvider services,
        IChemicalHeadlineAnalysisRepository repository)
        : base(services)
    {
        _repository = repository;
    }

    protected override ChemicalHeadlineAnalysisViewModel
        GetViewModel(WidgetContext context, NullSettings settings)
    {
        var entitlementsOptions = _userSettings.GetUserProfile(UserIdentity.Id)
            .EntitlementsOptions().ByGlobalSwitch();

        return _repository.Fetch(context.EffectiveTaxonomy, entitlementsOptions);
    }
}
```

Rysunek 1.15: Kontroler aplikacji webowej

**Interpretacja:** Wyszczególnione elementy stanowią różnicę w stosunku do korespondującego kontrolera 1.16

**Źródło:** własne

```
class ChemicalHeadlineAnalysisWidgetIPadController :  
    WidgetListContextIPadController<ChemicalHeadlineAnalysisWidgetSetup,  
    ListResult<Document>, NullSettings>  
{  
    private readonly IChemicalHeadlineAnalysisRepository _repository;  
  
    public ChemicalHeadlineAnalysisWidgetIPadController(  
        IWidgetServiceProvider serviceProvider,  
        IWidgetIPadSettingsRepository<NullSettings> widgetIPadSettingsRepository,  
        IChemicalHeadlineAnalysisRepository repository)  
        : base(serviceProvider, widgetIPadSettingsRepository)  
    {  
        _repository = repository;  
    }  
  
    protected override IPadResult<ListResult<Document>>  
        GetDataModel(WidgetWithPageContext context, NullSettings settings)  
    {  
        var entitlementsOptions = _userSettings.GetUserProfile(UserIdentity.Id)  
            .EntitlementsOptions().ForIPad();  
        var data = _repository.Fetch(context.EffectiveTaxonomy, entitlementsOptions);  
        return this.IPadResult(new ListResult<Document>(  
            ConceptMapper.Map(data.Documents, this.GetApiVersion())));  
    }  
}
```

Rysunek 1.16: Kontroler aplikacji mobilnej

**Interpretacja:** Wyszczególnione elementy stanowią różnicę w stosunku do korespondującego kontrolera 1.15

**Źródło:** własne

W tej części pracy po krótce zdefiniowano sens istnienia, a także mechanizmy determinujące wygląd i strukturę API. W następnym rozdziale poruszana zostanie kwestia konstruowania poprawnych syntaktycznie i semantycznie interfejsów. Znane są już oczekiwania programistów i to, jak wygląda ich interakcja z interfejsami Web API. Dodatkowo, ustaliłem, jakie modele sprzyjają lub utrudniają budowę skalowalnego produktu czy linii produktów. Nadszedł zatem czas, by odnaleźć wartość biznesową i postawić linię podziału, która wyznaczy obszar, w jakim API będzie się poruszało.

## ROZDZIAŁ 2

# Projekt WebAPI

Myślenie o dobrze zaprojektowanym API najczęściej determinuje wiele złożonych sytuacji. Najpopularniejsze z nich to:

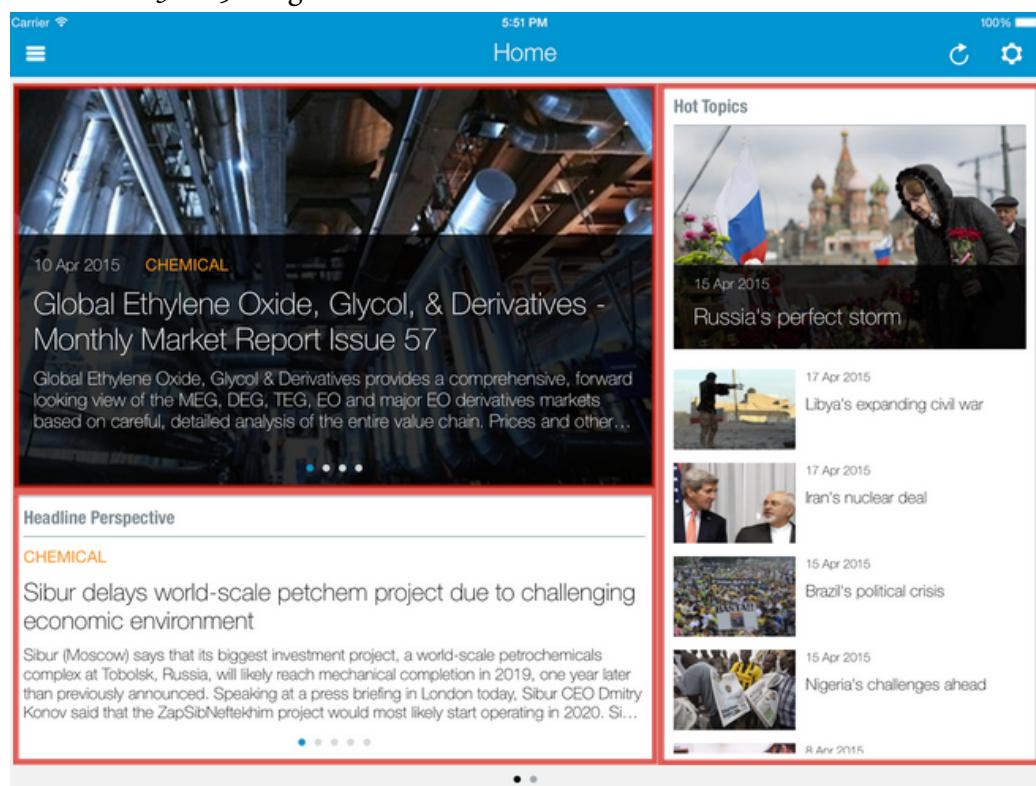
1. **pierwsze projektowanie i planowanie** - co wydaje się być naturalne, choć nieoczywiste. Wielokrotnie pojawiają się interfejsy, które wyrosły organicznie w oparciu o intuicję i wcześniejsze doświadczenia deweloperów.
2. **poprawa wydajności** - okazuje się bowiem, że API narzuca na aplikację konieczność częstego komunikowania się i synchronizowania żądań lub przypadek, gdy oczekiwanie na odpowiedź trwa zbyt długo.
3. **pożądana funkcjonalność** okazuje się być bardzo skomplikowana do użyczenia, a co za tym idzie - wymaga od wykonania wielu kroków.
4. **widoczne są redundancje w kodzie i dług techniczny**, co powiązane jest z nieudolną architekturą.

Bardzo ważne jest, by pierwszą wersję API dobrze zaprojektować - szczegółowo, gdy w założeniu będzie używane przez wiele aplikacji. Nie ma wtedy obowiązku myślenia o utrzymywaniu wcześniejszych wersji, co umożliwia bardzo elastyczne modyfikowanie podejścia. Z czasem jednak, gdy narzuć istniejących wersji powstaje, należy bardzo ostrożnie migrować pojawiające się pomysły i płynnie przeходить pomiędzy jedną architekturą a drugą, mając na uwadze to, że przez kolejny tydzień, miesiąc, rok, niezależnie wydane aplikacje mogą nie zostać zaktualizowane. Tym samym – będą wciąż bazowały na starej wersji API. Niezbędne jest tu zapewnienie odpowiedniego wsparcia i dokumentacji dotyczącej tego, w jaki sposób dokonać migracji z wersji na wersję.

Moją motywacją do przyjrzenia się strukturze była architektura API, z którego obecnie korzystam w projekcie. Dotychczas rozwijałem API o nowe zasoby. Tym

razem jednak chciałem wejść głębiej, poznać mechanizmy, z których korzystam i wysnuć przypuszczenia odnośnie ich genezy i zastosowania. Doszczętnie do następujących wniosków.

Aplikacja jest zbudowana z kilkunastu *dashboardów*, które zawierają od jednej do kilku stron, na których rozmieszczone są aplikacje. Schemat ten obrazuje Rysunek 2.1. Każdy z widgetów dysponuje autonomiczną powierzchnią na ekranie i na tej powierzchni prezentuje swoje treści. Zazwyczaj liczba widgetów na stronie zawiera od 3 do 5 widgetów.



**Rysunek 2.1:** Układ strony startowej w aplikacji mobilnej dla platformy iPad

**Interpretacja:** W układzie strony, która jest częścią *dashboardu* widzimy kilka wyodrębnionych *widgetów* ułożonych w „klockową strukturę”

**Źródło:** własne

Autonomiczność widgetów poniosła za sobą konsekwencję w postaci osobnych żądań do serwera. Jednak nadal poszczególne widgety są częścią większego bytu, stąd dodatkowy narzut żądań zapewniających właściwe metadane o strukturze strony oraz dane niezbędne do komunikacji z całością dashboardu. Nie byłoby w

tym nic zlego, gdyby nie protokół HTTP 1.1, który „rozdmuchuje” ilość przesłanych pakietów o własne nagłówki.

200 POST connect.ihc.com	/iPad/16/Api/metadata	539 ms	749 bytes	Complete
200 POST [REDACTED]	[REDACTED] json/Login	670 ms	1.22 KB	Complete
200 POST connect.ihc.com	/iPad/16/Dashboard/GetMenuItems	801 ms	7.36 KB	Complete
200 POST connect.ihc.com	/iPad/16/Saved/GetReadLaterFolder	1764 ms	10.32 KB	Complete
200 POST connect.ihc.com	/iPad/16/LandingPageFilters/Index	967 ms	3.83 KB	Complete
200 POST connect.ihc.com	/iPad/16/Dashboard/GetSubMenuItems?pageMode=LandingPage	404 ms	3.26 KB	Complete
200 POST connect.ihc.com	/iPad/16/LandingPage/GetTopStories	2624 ms	6.15 KB	Complete
200 POST connect.ihc.com	/iPad/16/LandingPage/GetFeaturedResearch	1038 ms	4.73 KB	Complete
200 POST connect.ihc.com	/iPad/16/LandingPageHotTopics/Content	776 ms	5.31 KB	Complete
200 POST connect.ihc.com	/iPad/HotTopicViewer>Show?source=gi&docid=2617787	748 ms	6.95 KB	Complete
200 POST connect.ihc.com	//iPad/DisplayDocument>Show?source=gi&docid=1030847	596 ms	4.59 KB	Complete
302 GET connect.ihc.com	//iPad/16/ImageResize/Render?path=%5C%5Cihdeneclp01%5CCastle_prod_CIFS1%5CStaticAttachments%5C3f%5C345.jpg&title=image&width=748&height=497&crop=true	524 ms	3.64 KB	Complete
302 GET connect.ihc.com	//iPad/16/ImageResize/Render?path=%5C%5Cihdeneclp01%5CCastle_prod_CIFS1%5CStaticAttachments%5C8b%5C503.jpg&title=image&width=748&heigh=497&crop=true	406 ms	3.64 KB	Complete
302 GET connect.ihc.com	//iPad/16/ImageResize/Render?path=%5C%5Cihdeneclp01%5CCastle_prod_CIFS1%5CStaticAttachments%5Cc9%5C518.jpg&title=image&width=748&height=497&crop=true	366 ms	3.64 KB	Complete
200 GET connectfiles.ihc.com	/FileDownload.ashx?token=cAm4tE5r78UfmPL5JxbT++3SbQg8t4TfN2fkAbA/B4k0Dx/V1TEU24y9M1WaoOzjuFuqtbYqTxpOhSeuzsSMims0CU7pn6jMFs+9pKTSU8dYyrTw+yhZEXP5fyS	497 ms	65.02 KB	Complete
302 GET connect.ihc.com	//iPad/16/ImageResize/Render?path=%5C%5Cihdeneclp01%5CCastle_prod_CIFS1%5CStaticAttachments%5Cd8%5C525.jpg&title=image&width=748&height=497&crop=true	419 ms	3.64 KB	Complete
200 GET connectfiles.ihc.com	/FileDownload.ashx?token=Dwoy3b0BTzVdN/G4zP4anXjTQfONlqDG9ogTLozguZJDTKiDIXV+oPLAnpdAbt7RRDvYdl+GOWYNyqSyAv4t6WVTA9FYAc23jdQzLEXwpWA6YF7DIEZbzBZ1Dai	798 ms	129.03 KB	Complete
200 POST connect.ihc.com	//iPad/CeraViewViewer>Show?source=gi&docid=2796542	772 ms	8.18 KB	Complete
200 GET connectfiles.ihc.com	/FileDownload.ashx?token=d+wA1KBA5zD4HD/NcVSG/CzTzNkmnU7yL3Dluui7j3lQJT/9i82zYYkfZcRNv+sTFUpKkZkOsP+D0oJshf7yBlcAjkeUoLY6i6P7YvELHvw1ozk8u05FWhrEvK	725 ms	129.03 KB	Complete
200 GET connectfiles.ihc.com	/FileDownload.ashx?token=LmWYoMS10i2XgZlDavgPZUKM1IYhQQvChicjlJ7zip68vVA+v4oEb7cvkil.rQ2cQ5L511Ef0REYEALhwPtff/KxsPT3+q+oJv07cc1ZPDR9piKxLmkdLc35kGCVdGe	452 ms	65.02 KB	Complete

Rysunek 2.2: Zrzut żądań wysyłanych przez aplikację iPadową, po to by uwierzytelnić/autoryzować użytkownika oraz wyświetlić treść pierwszego ekranu.

**Interpretacja:** Na zielono oznaczone zostały żądanie niezwiązane z konkretnym ekranem a ze strukturą API. Na niebiesko żądanie autentykacyjne. Pozostałe służą wyświetleniu strony startowej.

**Źródło:** własne

200 POST connect.ihc.com	/iPad/16/Dashboard/DashboardTaxonomy?pageMode=OrganizationGeneral	1231 ms	3.31 KB	Complete
200 POST connect.ihc.com	/iPad/16/Dashboard/GetSubMenuItems?pageMode=OrganizationGeneral	823 ms	3.35 KB	Complete
200 POST connect.ihc.com	/iPad/16/Dashboard/DashBoardWidgetsForTaxonomy?pageMode=OrganizationGeneral&	485 ms	4.87 KB	Complete
200 POST connect.ihc.com	/iPad/16/OrganizationKeyReportsWidget/Content	2770 ms	5.81 KB	Complete
200 POST connect.ihc.com	/iPad/16/OrganizationProfileBasicWidget/Content	700 ms	4.23 KB	Complete
200 POST connect.ihc.com	/iPad/16/OrganizationProfileLatestWidget/Research	875 ms	9.74 KB	Complete
200 POST connect.ihc.com	/iPad/16/OrganizationSupportCompaniesWidget/Content	669 ms	3.72 KB	Complete
200 POST connect.ihc.com	/iPad/16/OrganizationHeadquartersWidget/Content	570 ms	3.77 KB	Complete
200 POST connect.ihc.com	/iPad/16/OrganizationPeerInformationWidget/Content	921 ms	3.65 KB	Complete
200 POST connect.ihc.com	/iPad/16/OrganizationContactsWidget/Content	583 ms	3.93 KB	Complete
200 POST connect.ihc.com	/iPad/16/OrganizationMarketDataWidget/Content	1291 ms	3.71 KB	Complete

Rysunek 2.3: Zrzut żądań wysyłanych, by wyświetlić jedną ze stron aplikacji.

**Interpretacja:** Pierwsze trzy żądania odpowiadają za pobranie struktury strony i poszczególnych jej modułów. Pozostałe żądania pobierają treść dla każdego z modułów.

**Źródło:** własne

Wprawione oko zauważa także, że większość z żądań na Rysunkach 2.2 i 2.3

zwraca kod HTTP 200 i jest wykonywana za pomocą metody POST. Dodatkowo, każda z odpowiedzi zwraca własny rodzaj informacji o błędach, a co za tym idzie - obudowuje nieudane żądania w niestandardowy obiekt ze statusem, na który nie będą w stanie automatycznie zareagować nawet najpopularniejsze biblioteki *networkingowe*. Wynika to z faktu, że zostaną oszukane przez kod HTTP 200, który świadczy o sukcesie wykonywanej operacji.

Jest to zaledwie ogólna analiza kilku grzechów projektu, w którym uczestniczę. Być może w dalszej części uda mi się te błędy odpowiednio rozliczyć. W pozostałych przypadkach pozostanie przyznać się do winy i zabrać się za naprawę stanu rzeczy przynajmniej częściowo.

## 2.1. Wartość biznesowa

Kluczowym dla zapewnienia sukcesu API jest określenie celu jego powstania i rozwoju oraz wyznaczenie sobie kamieni milowych, które chcemy osiągać. Często ma miejsce sytuacja, gdy trzeba będzie wy tłumaczyć radzie nadzorczej, szefowi, menadżerom istotę istnienia API. Mając to na uwadze, dobrze jest mieć przygotowaną odpowiedź w każdym momencie. Jeśli mamy sprecyzowaną wizję naszego produktu lub rodziny produktów, odpowiedź na takie pytanie nie sprawi nam problemu. Inaczej jest, gdy podjęta zostaje próba bieżącego usprawiedliwiania rozwoju interfejsu, wobec której późno znaleźć potwierdzenie w realnych i mierzalnych danych. Najszczeliwsza sytuacja to ta, gdy API jest głównym produktem, a jego rentowność i stosowność można wyrazić w konkretnych dochodach firmy. Kryje się tu jednak pułapka dla API, będącego wsparciem głównego produktu. Zaczyna bowiem ono konkurować z aplikacją, której wpływ na zysk firmy jest mierzalny, podczas, gdy sam fakt istnienia i użytkowania API nie jest przekładalny na dochody.

*Twilio* jest w tym wypadku adekwatnym przykładem, gdzie API jest głównym produktem i zarabia samo na siebie. Każde żądanie wysiane do API generuje niewielki, określony koszt. Stąd developer integrujący API Twilio do swojej aplikacji dzieli się pewną częścią zysku z usługodawcą. Przekłada się to jednak na świetne wsparcie i przemyślany, intuicyjny portal deweloperski, jako że bezpośrednimi klientami *Twilio* są programiści.

Znacznie częściej będzie trzeba określić wartość biznesową API, którego celem jest zwiększenie zaangażowania użytkowników na platformie, tak jak ma to miejsce w przypadku *Facebooka* czy *Twittera*. Wykorzystanie zasobów tych serwisów i intensywne zaangażowanie użytkowników zapewnia proporcjonalne wpływy z reklam oraz podnosi wartość danych, którymi rzeczone platformy dysponują. Jeżeli jednak nie produkt nie jest potentatem na rynku portali społecznościowych, dobrą motywacją jest zacieśnianie relacji z klientami. Oferowanie interfejsu, który pozwoli łatwo zintegrować oferowane systemy z systemami klienta będzie korzystnie wpływać na decyzję podczas wyboru spośród podobnych rozwiązań proponowanych przez konkurencję.

Można się zastanowić nad tym, jak udostępnienie API przez uczelnię wpłynę-

łoby na rozwój systemów administracji, ocenianie studentów, prowadzenie kursów. W Uniwersytecie Gdańskim system *Fast* pokrywa w znaczącym stopniu zapotrzebowanie na wsparcie informatyczne większości pracowników administracji, studentów i wykładowców. W szczególnym przypadku wydziału MFI wykładowcy pomimo tego inwestują w swoje metody przekazu i organizacji zajęć. Związane jest to z rozwojem technologii oraz wiąże się z prowadzonymi badaniami. Jest także dodatkową okazją wdrożenia studentów w systemy informatyczne. API w tym przypadku pozwoliłoby pracownikom i studentom na produkowanie własnych aplikacji i interfejsów, które bazowałyby na realnych danych, bez potrzeby ręcznego przenoszenia ich pomiędzy źródłami. Owocem takich integracji z reguły są pomysły, które pierwotnie nie zostały wzięte pod uwagę podczas projektowania systemów, zapotrzebowanie spowodowało jednak ich rozwój. Taki schemat działania przyjął chociażby *Evernote*, który inwestuje środki *hackathony* i kreatywność deweloperów i kreatywność deweloperów, a jednocześnie najlepsze rozwiązania włącza do swojej platformy. API jest tutaj furtką do kreatywności.

Firma na rzecz, której pracuję zajmuje się sporządzaniem analiz rynków (energetycznych, chemicznych, nowych technologii, motoryzacyjnych i innych). Jej kapitałem są dokumenty, serie danych, informacja (nie tylko ta zagregowana, ale też przetworzona i opisana), narzędzia, które służą dużym podmiotom gospodarczym, do podejmowania krytycznych decyzji o rozwoju i zachowaniu na rynku. Produktami mojej firmy są zatem **gotowe rozwiązania**: aplikacje webowe, mobilne, desktopowe. Z pewnością są one dużym ułatwieniem dla analityków firm klienckich. Jednakże warto zastanowić się nad tym, czy oprócz większych elementów w postaci aplikacji w ofercie firmy nie powinny się również znaleźć bloki składowe? Dzięki temu daliśmy naszym klientom możliwość współpracy i współtworzenia naszej platformy, a co za tym idzie - zwiększyliśmy ich zaangażowanie. Stąd motywacja, by zająć się tematem API. Kolejne rozdziały zobrazują to, w jaki sposób zbudowany jest nasz model aplikacji i co moglibyśmy zaoferować klientom powstającego API, a także jak powinniśmy ustrukturyzować powstający interfejs.

## 2.2. Budowa poprawnego modelu

Jest kilka cech, które charakteryzują dobrze zaprojektowany model API. Powinien on być przede wszystkim zrozumiały i przewidywalny dla klienta. Jako przykład, autorka książki *Irresible APIs* [1], podaje API serwisu *Flickr*. Sugeruje, że użycie jedynie metod *POST* i *GET* portokołu HTTP, a także obecność parametru *method* w ścieżce wywołania na listingu 2 świadczy o tym, że pomimo tego, iż adres w domniemaniu odnosi się do API RESTowego, z tym rodzajem API nie mamy do czynienia w przypadku portalu *Flickr*.

Listing 2. Jeden z adresów API *Flickr*

```
1 https://api.flickr.com/services/rest/?method=flickr.photos.delete&photo_id=value
```

Aby wywołać powyższe żądanie, serwis oczekuje metody *POST*. W zwrocie otrzymujemy kod błędu oraz status 200. Świadczy to o pozytywnym wyniku operacji. Dla wielu moich kolegów z branży może to wyglądać nieintuicyjnie. Jednakże warto zwrócić uwagę również na fakt, że podczas integrowania wielu API w jednej aplikacji należy każde obsłużyć osobno, ponieważ nawet informacje o błędach nie są standardowe. Oczywiście powyższe jest wykonalne, jednak wymaga dodatkowej uwagi i ostrożności. Nie jest to także droga na skróty jak np. możliwość korzystania z gotowych frameworków.

Projekt API powinien zakładać zróżnicowanie form klienckich, a co za tym idzie - ich szczególne wymagania. W przypadku aplikacji mobilnych najprawdopodobniej do listy wymagań należałoby dopisać: autentykację na platformie, zapobieganie nieprzewidzianym *crashom* aplikacji, krótki czas oczekiwania. By sprostać tym wymaganiom powinno się uwzględnić w architekturze chociażby możliwość pobrania wszystkich informacji dla jednego ekranu w pojedynczym żądaniu, minimalny rozmiar przesyłanych danych, a także możliwość sprecyzowania, które sekcje danych są potrzebne.

Wymagania te biorą się z ograniczonego pasma sieciowego oraz częstych przerw w połączeniach (np. po wejściu do tunelu, windy lub lasu). Nadal większość aplikacji i urządzeń nie jest w stanie zapewnić wystarczająco efektywnego przetwarzania

równoległego, więc konieczność synchronizacji kilku żądań dla jednej strony będzie skutkowała kiepskim zachowaniem interfejsu użytkownika.

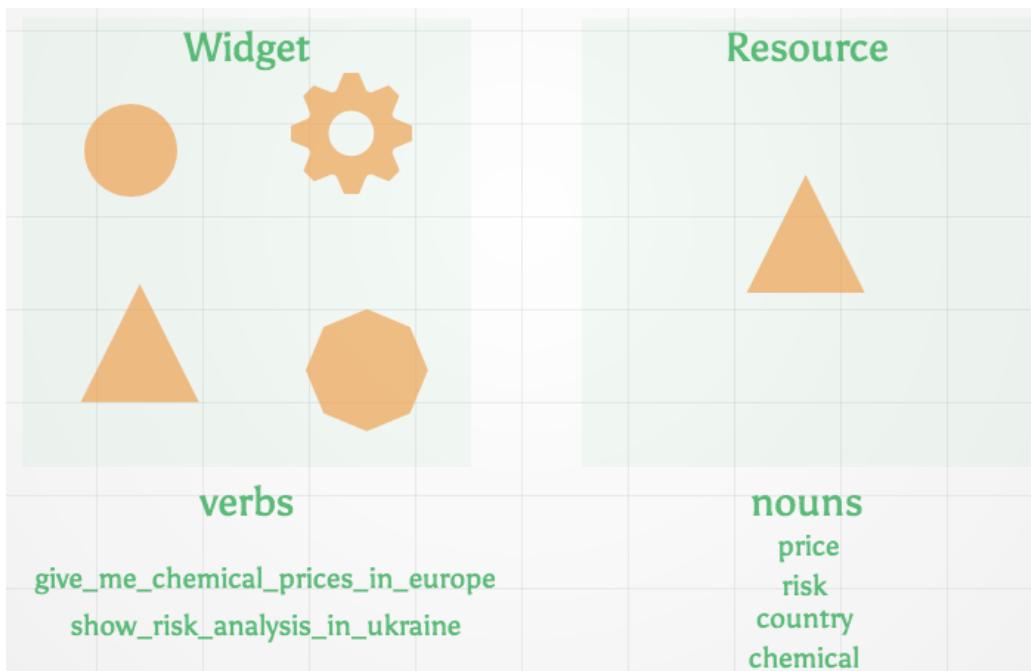
### 2.2.1. Rzeczowniki - zasoby, kontra czasowniki - akcje

Zmotywowani chęcią usprawnień w istniejących aplikacjach mobilnych i ich integracji z częścią serwerową usiedliśmy z zespołem pewnego dnia do rozmów. Właściwie inicjatywa „poprawiania” została zapoczątkowana przez odnalezione duplikacje, wspomniane już w rozdziale 1.3. Zaczęliśmy się zastanawiać nad tym, skąd się one biorą i jak należałoby podejść do tej kwestii. Otóż efektem ponad dwugodzinnej rozmowy był wniosek: *Powinniśmy mieć do tego API*. Zamiast „rozmuchiwać” aplikację webową należałoby wydzielić część odpowiedzialną stricte za prezentację interfejsu użytkownika i część (aplikację) odpowiedzialną za serwowanie danych - zarówno dla serwisu web, jak dla klientów mobilnych i integracji. Przy spojrzeniu na istniejącą architekturę i rozmach projektu (dziesiątki serwisów, bibliotek, podprojektów, miliony klas), było to stwierdzenie bardzo górnolotne, a wręcz utopijne. Gdyby jednak stopniowo wydzielać konkretne części aplikacji i oferować API dla nich, to jest szansa na powolne, mozołne budowanie przyjaznego interfejsu.

Dlaczego właściwie potrzeba nowego interfejsu? Postaram się to zobrazować za pomocą przykładu.

Nasze aplikacje w obecnym kształcie składają się z poukładanych w większą strukturę widgetów (Rysunek 2.1). rosząc o dane dla nich, wysyłamy żądania w postaci: *Daj mi ceny chemikaliów XYZ w Europie albo Pokaż mi analizę ryzyka na dla Ukrainy*. Spowoduje to wysłanie paczki danych skrojonych na potrzeby konkretnego widgetu. Gdyby jednak rozważyć to, co można z obecnego API skonstruować, to okazałoby się, że istnieje możliwość stworzenia jedynie podobnych aplikacji, składających się z takich samych części, różniących się jedynie kwestiami wizualnymi i warstwą prezentacji. Szybko zaczęłoby brakować dostępu do bardziej granularnych danych, takich jak informacje o poszczególnych *krajach, ryzykach, cenach, chemikaliach*. Jeśli więc chcemy wyzwolić kreatywność naszych klientów, powinniśmy oferować zasoby w postaci „rzeczownikowej”. Twitter bardzo rozwinął swoje API

dzięki pomysłom użytkowników (włączając w ramy własnej platformy najlepsze z nich i uniemożliwiając wykonywanie tych samych zapytań w inny sposób).



Rysunek 2.4: Porównanie architektury metod i zasobów

**Interpretacja:** Architektura oparta na czasownikach wykonuje bardzo określoną czynność i serwuje jej rezultat. Architektura oparta na zasobach operuje rzeczownikami, nie mówi nic o sposobie ich wykorzystania.

**Źródło:** własne

Jest to doskonały sposób na „ożywienie” naszej platformy i jej rozwój. Nie ma bowiem produktów skończonych. Często okazuje się, że projekt bez pomysłu na rozwój lub produkt, który został wstrzymany, ginie bezpowrotnie zastąpiony alternatywami umożliwiającymi kreatywność.

Rank	Country	12 Month Trend	Overall Rating	Last Change	Political %25	Economic %25	Legal %15	Tax %15	Operational %10	Security %10
13	United States	■	■ 1.60	28 Jan 2013	■ 1.50	■ 1.75	■ 1.00	■ 1.50	■ 1.50	■ 2.25
18	Germany	■	■ 1.67	18 Jul 2014	■ 1.75	■ 1.50	■ 1.25	■ 1.50	■ 2.00	■ 2.25
24	France	■	■ 1.83	18 Jul 2014	■ 1.75	■ 1.75	■ 1.50	■ 1.50	■ 2.00	■ 2.75
32	Poland	■	■ 1.96	10 Oct 2013	■ 2.00	■ 2.00	■ 1.75	■ 2.00	■ 2.00	■ 2.00
119	Russian Federation	■	■ 3.13	17 Mar 2015	■ 3.25	■ 3.25	■ 2.75	■ 2.50	■ 3.75	■ 3.25
170	Ukraine	■	■ 3.68	17 Mar 2015	■ 4.00	■ 4.00	■ 3.25	■ 2.50	■ 3.50	■ 4.25

Rysunek 2.5: Jeden z widgetów, prezentujący kilka wybranych aspektów i ocen ryzyka dla wybranych krajów.

**Interpretacja:** Widgety w aplikacji prezentują informację przygotowaną, przetworzoną i wycelowaną w specyfczną analizę. Zawierają ściśle zdefiniowany zestaw informacji, który ma docelowo ułatwić analizę sytuacji klientom.

**Źródło:** własne

W ten oto sposób z pojedynczego widgetu 2.5, możemy wydzielić kilka niezależnych lub wzajemnie linkujących do siebie zasobów. Zasoby te mogą następnie znaleźć zastosowanie we wdrożeniach, o których nawet nie pomyśleliśmy wcześniej (Rysunek 2.6). Warto wymienić tu chociaż szybko rozwijający się rynek systemów *embedded*, *wearables* czy coraz popularniejsze *smartwatches* i opaski nafaszerowane czujnikami i nadajnikami. Bardzo mało firm jest sobie w stanie pozwolić na wypróbowanie wszystkiego, dlatego otwartość i pomoc przeróżnych społeczności jest nieoceniona w akceleracji rozwoju naszych produktów. W przypadku zasobów generycznych istnieje możliwość dowolnego miksuowania danych i łączenia ich z innymi dostępnymi API. Można zapewnić własne przetwarzanie i skupić

się tylko na tym, co jest obiektem zainteresowania, w czasie, gdy platforma zyskuje na ruchu i wzbogaca się w nowe *Use Case'y*.



Rysunek 2.6: Wykorzystanie zasobów API w całym przekroju platform i nośników.

**Interpretacja:** Szybko rozwijający się Internet rzeczy - *Internet of Things* jest potencjalnym medium przekazu informacji, którego badanie i testowanie warto powierzyć klientom w przypadku braku własnych zasobów.

**Źródło:** własne



## **ROZDZIAŁ 3**

# **Wykonanie WebAPI**

W poprzednich częściach omówiłem biznesowe pobudki, które skutkują powstaniem API, a także koncepcje kreowania zasobów i konsekwencje używania RESTful API opartego na rzeczownikach. Nadszedł czas, by przyjrzeć się szczegółowo implementacyjnym, czyli tzw. smaczkom, które mogą nie wydawać się oczywiste i zauważalne na pierwszy rzut oka. W poniższych rozdziałach skupię się na konstruowaniu poprawnego syntaktycznie interfejsu, który umożliwi wydajną komunikację z aplikacjami mobilnymi. Jednocześnie przypadki użycia ograniczą do podzbioru cen chemikaliów i związanych z nimi serii, zawierających historyczne i przewidywane ceny. W kilku przykładach zobrazuję, w jaki sposób można ograniczyć czas żądania i wielkość odpowiedzi, a także zapewnić dodatkową funkcjonalność w postaci sortowania przy niewielkim koszcie. W tym rozdziale naturalnie można by próbować opisać znacznie więcej aspektów, takich jak autentykacja, wersjonowanie, rozszerzalność czy cache, jednak moją intencją jest jedynie wzbudzenie ciekawości czytelnika i zachęcenie do zagłębiania się w konstrukcję API w świetle przedstawionych przykładów.

### 3.1. Implementacja na wybranej platformie

Każdy programista ma swój ulubiony język programowania i framework, w którym najłatwiej jest mu wyrazić swój projekt. W poniższych przykładach posłużę się frameworkm *Nancy*, który jest dostępny na platformę .NET i pozwala na bardzo szybkie i proste skonstruowanie serwera API. By umożliwić korzystanie z API, opublikowałem je w chmurze *Amazona* pod adresem <http://mgr.arturrybak.com>, w kontenerze *Dockerowym*. Dzięki temu bardzo łatwo jest *deployować* nowe wersje, a dostępność aplikacji szacuje się na 99.5%. Do przechowywania danych użyłem bazy MSSQL Server 2008.

Wszystkie projekty dostępne są w repozytorium pod adresem <http://github.com/wedkarz/mgr> (należy uprzednio uzyskać dostęp) a także na załączonej płycie. Nie należy tu oczekwać fajerwerków graficznych, gdyż aplikacje zostały stworzone jedynie celem zmierzenia wydajności komunikacji klient-serwer.

\*\*\*

Modyfikację istniejącego modelu rozpoczęłem od bardzo głębokich warstw – a mianowicie od zmiany struktury tabel bazy danych. Dotychczasowy model serii chemicznych zakładał ich znormalizowane przechowywanie w bazie danych w postaci czterech tabel:

- *TimeseriesLocal* - tabela zawierająca identyfikator serii danych, wartości w postaci JSON, tytuł, grupę i subskrypcje, dla których seria jest dostępna (ponad 100 tys. elementów);
- *TimeseriesAttribute* - tabela zawierające wszystkie dodatkowe typy atrybutów serii (obecnie 40 elementów);
- *TimeseriesAttributeValue* - tabela zawierające wszystkie możliwe wartości wszystkich atrybutów serii (97 tys. elementów);
- *TimeseriesAttributeValueToLocalTimeseries* - tabela, która zawiera klucze obce, łączące *TimeseriesLocal* i *TimeseriesAttributeValue*.

Podejście to okazało się nieefektywne dla użytkownika, który chciałby w pojedynczym zapytaniu uzyskać możliwie najwięcej informacji i metadanych serii

iteracja	czas wykonania żądania (w ms)		
	przed denormalizacją	dla widoku	po denormalizacji
1	12248	8370	5925
2	7704	10688	5776
3	7278	6784	3825
4	8809	6784	4627
5	9696	8112	4845
6	9220	9486	3489
7	7415	9675	4017
8	7215	9219	4319
9	7578	7034	4430
10	9110	7133	4444
<b>średnia</b>	<b>8627.3</b>	<b>8328</b>	<b>4569.7</b>

Tabela 3.1: Porównanie czasów zapytań sql dotyczących Benzenu przed i po denormalizacji

chemicznej (zmian cen chemikalium w czasie). Okazuje się, że by otrzymać cenę oraz wszystkie metadane jednego chemikalium w jednym zapytaniu SQL konieczne było wywołanie *SELECT*, w którym **13 razy** musieliśmy złączać po trzy tabele (ze względu na rozmiar, skrypt został dołączony na płytce CD). Jest to co najmniej nieefektywne i skutkuje długimi czasami oczekiwania i zajętości bazy. Dlatego też zdecydowałem się utworzyć widok w bazie, który grupuje tabele. Jednakże takie działanie również nie rozwiązało problemu wydajności, o ile nie został założony dodatkowy indeks. Drugim rozwiązaniem, które stało się również wyjściem ostatecznym, było zduplikowanie danych.

Redundancja i denormalizacja bazy była w tym przypadku uzasadniona, jako że spodziewamy się częstych zapytań o serie chemiczne, które będą zawierały wszystkie metadane. W oparciu o powyższe powstała nowa tabela *TimeseriesDetails*. Dla porównania zmierzyłem czasy oczekiwania. Dla porównania zmierzyłem czasy oczekiwania. Wyniki prezentuje Tabela 3.1. Na jej przykładzie zysk z utworzenia nowej tabeli jest niepodważalny. Średnio niemal dwukrotne przyspieszenie zapytań, a w szczególnym przypadku pierwszego zapytania różnica po denormalizacji to ponad dwukrotne przyspieszenie. Powyższy test można przeprowadzić we

własnym zakresie, korzystając ze skryptów dołączonych do pracy a także dostępnych pod adresem `http://mgr.arturrybak.com/test_select` lub w wersji wieloiteracyjnej `http://mgr.arturrybak.com/test_select/liczba_iteracji`, gdzie parametr `liczba_iteracji` należy zastąpić liczbą. Jedna iteracja to czas oczekiwania rzędu 25 sekund.

Przygotowując się do stworzenia API warto pamiętać o tym, że będzie ono spełniało swoją rolę tylko wtedy, gdy wydajność nie będzie stwarzała problemu. Jednakże w tym wypadku można się natknąć na przeszkodę w postaci modelu danych, czyli bazy, której nie można modyfikować. Warto zatem pomyśleć o tym, czy dla tej specyficznej części systemu, która będzie objęta API, nie należałoby rozważyć przeprojektowania albo wydzielenia części bazy.

## 3.2. Składnia zapytań i odpowiedzi przy użyciu protokołu HTTP

### 3.2.1. Konstrukcja adresów URL

Jak już wcześniej wspominałem, elementarną kwestią jest wydzielenie zasobów w API. Wokół nich będziemy bowiem konstruować żądania HTTP za pomocą odpowiednich metod (GET, POST, PUT, DELETE). W rozdziale 2.2.1 bszernie omówiłem, dlaczego w tym wypadku powinno się korzystać z rzeczowników. Decyzja o tym, czy używać liczby pojedynczej czy mnogiej zależy już wyłącznie od preferencji. Niewielką zaletą liczby mnogiej będzie uwolnienie deweloperów od tworzenia wielu akcji dla różnych form, jak np. *person/people*, *goose/geese*.

Moje wyjściowe API będzie ze względu na swoją prostotę wydaje się być optymalnym rozwiązaniem. Wynika to z faktu, że będzie zawierało tylko dwie metody jak na listingu 3.

Listing 3. Wyjściowe API cen chemicznych

```
1 GET /prices  
2 GET /prices/{chemical_code}
```

Zdefiniowane przypadki użycia przewidują jedynie akcje *read-only*. Oznacza to, że możliwe będzie jedynie „wyciąganie” informacji z API. Wszystko jednak zaczyna się rozmywać, gdy nasze akcje nie wpadają w typowe operacje CRUD (Create, Read, Update, Delete). Możemy tutaj podjąć kilka kroków celem wykłarowania sytuacji:

- przemianowanie akcji na pole zasobu, np. dla akcji *activate* może zostać zmiana na pole *activated* zasobu;
- traktowanie akcji jako podzasobów, np. API Github'a oferuje oznaczanie gwiazdką *gist'ów* jak na listingu 4;
- czasami usilne obstawianie przy pryncypach RESTful po prostu mija się z censem. Jak bowiem przełożyć pożądaną akcję *search*, skoro dotyczy ona wielu

zasobów? Zatem nie należy tego uwzględniać w każdym przypadku. Najlepszym rozwiążaniem okazuje się metoda najwygodniejsza z punktu widzenia klienta.

Listing 4. Zaznaczanie/odznaczanie gwiazdką w API Githuba

```
1 PUT /gists/:id/star
2 DELETE /gists/:id/star
```

### 3.2.2. Filtrowanie, sortowanie, przeszukiwanie tekstu

Dotychczasowy wysiłek włożony w API pozwala na wylistowanie wszystkich serii danych dla wszystkich chemikaliów bądź z uwzględnieniem kodu chemikaliów. Takie rozwiązanie nie jest wydajne ani w pełni satysfakcjonujące ze względu na ogrom danych przesyłanych w każdej odpowiedzi. Przykładowo zapytanie o ceny Benzenu (Rysunek 3.1) skutkuje około **9 sekundowym (sic!)** czasem oczekiwania i ponad 3 megabajtową odpowiedzią. To wszystko przy użyciu środowiska lokalnego. Wykorzystanie w tym przypadku maszyn (stosunkowo mało wydajnych) na chmurze Amazona pogarsza sprawę i często prowadzi do *timeoutów*. Po części może to być spowodowane niedostateczną mocą serwera (zarówno bazy danych, jak i API), niedostatecznym indeksowaniem bazy, kosztownymi operacjami parsowania.

	RC	Met...	Host	Path	Start	Duration	Size ▲	Status
	200	GET	win:8080	/prices/BZE	13:43:37	8471 ms	3.21 MB	Complete
	200	GET	win:8080	/prices/BZE	13:41:25	5466 ms	3.21 MB	Complete
	200	GET	win:8080	/prices/BZE	13:42:56	10865 ms	3.21 MB	Complete
	200	GET	win:8080	/prices/BZE	13:43:12	12489 ms	3.21 MB	Complete
	200	GET	win:8080	/prices/BZE	13:43:27	9060 ms	3.21 MB	Complete

Rysunek 3.1: Żądania o ceny chemiczne benzenu

**Interpretacja:** Czas oczekiwania na zwrot wynosi średnio 9 sekund w przypadku środowiska lokalnego. Wielkość odpowiedzi wynosi 3.21 MB, co jest dużym narzutem dla platform mobilnych.

**Źródło:** własne

Przyglądając się strukturze modelu cen chemicznych, łatwo zauważać, że pole *values* zawiera zserializowanego JSONa z danymi o wartościach cen w czasie. Jest to

długi *string*, który powoduje przesyłanie znacznej ilości danych. Dobrą praktyką jest udostępnianie interfejsu, który pozwoli sprecyzować, które pola mają być włączane do odpowiedzi.

#	RC	Method	Host	Path	Start	Duration	Size	Status
1	200	GET	win:8080	/prices/BZE	18:37:51	7813 ms	3.21 MB	Complete
2	200	GET	win:8080	/prices/BZE?fields=title	18:38:38	117 ms	5.23 KB	Complete
3	200	GET	win:8080	/prices/BZE?fields=title,type,status,region,concept	18:39:39	4309 ms	11.00 KB	Complete

Rysunek 3.2: Żądania z limitowaną liczbą pól

**Interpretacja:** Czasy trwania żądań i wielkość odpowiedzi dzięki limitowaniu tylko potrzebnych pól, mogą zostać znacznie zredukowane.

**Źródło:** własne



Rysunek 3.3: Żądania z limitowaną liczbą pól, wykres

**Interpretacja:** Porównanie czasu trwania żądań z limitowaną liczbą pól obrazuje ile możemy zyskać na oszczędnej komunikacji z serwerem

**Źródło:** własne

Rysunki 3.2 i 3.3 ujmują, jak wiele można zaoszczędzić na transferze danych do aplikacji mobilnej, gdy aktywność jest ograniczona jedynie do proszenia o wykorzystywane w danym momencie pola. Jeśli API jest dobrze zaprojektowane, to już w momencie wykonywania zapytania do bazy danych ilość żądanych pól będzie ograniczona. Spowoduje to oszczędność czasu na wyciągnięcie danych z bazy, przetworzenie na serwerze i przesyłanie do klienta. W moim przypadku, gdy ograniczę się tylko do proszenia o tytuły serii, zredukuję czas z 7,8 sek. do 0,1 sek., a wielkość odpowiedzi z 3,21 MB do 5,23 KB. Można zatem uznać, że jest to znaczne usprawnienie. Dodatkowo, twierdzenie to potwierdza porównanie w przeglądarce poniższych linków <http://mgr.arturrybak.com/prices/BZE> i <http://mgr.arturrybak.com/prices/BZE?fields=title,type,status,region,concept>

Oprócz limitowania zwrotów powinniśmy zapewnić swoim użytkownikom interfejs, który umożliwi im sortowanie czy filtrowanie zwrotów. Wprawdzie nie będzie to miało znaczenia przy oszczędności transferu lub czasów zwrotów, z pewnością jednak będzie pomocnym narzędziem w rękach deweloperów. Pomoże im

ograniczyć wysiłek wkładany w przetwarzanie danych po stronie aplikacji końcowej, a w efekcie „odchudzi” aplikacje klienckie. Rysunek 3.4 jest dowodem na to, że nie ma kosztów po stronie serwera w dobrze zaimplementowanym sortowaniu.

RC	Method	Host	Path	Start	Duration	Size	Status
200	GET	win:8080	/prices/BZE?fields=title,type,status,region,concept	19:52:05	107 ms	11.03 KB	Complete
200	GET	win:8080	/prices/BZE?fields=title,type,status,region,concept&sort=-concept	19:52:14	109 ms	11.01 KB	Complete

Rysunek 3.4: Zapasy z wykorzystaniem sortowania i bez niego

**Interpretacja:** Dobrze zaimplementowane sortowanie nie ma wpływu na czas i wielkość odpowiedzi, pozwala za to już na etapie komunikacji z serwerem przygotować dane do prezentacji.

Źródło: własne

# Zakończenie

Wierzę, że w tej pracy znalazło się przynajmniej kilka wskazówek, które okażą się przydatne dla osób tworzących aplikację. Ufam także, że każde API, które po wstanie po lekturze tej pracy będzie przemyślane, a oparcie się o organiczne tworzenie API zostanie zamienione na metodę podpartą uzasadnionymi przypadkami użycia. Moją intencją było przybliżenie nie tyle implementacji, co konstrukcji logicznej i semantycznej tworzonego interfejsu, tak by był on nie tylko sztuką dla sztuki, ale również praktycznym narzędziem do wykorzystania na co dzień.

Dla mnie to dopiero początek zmian, które pojawią się w strukturze serwisu. Praca ta jednak wiąże się ogromnym wysiłkiem, który wpłynie na wiele warstw abstrakcji: począwszy od bazy danych - przez logikę biznesową - po oddzielenia warstwy prezentacji. Niech ta praca będzie przestrogą i nauczką dla architektów, deweloperów i analityków biznesowych, by ich myślenie już na początkowym etapie tworzenia aplikacji obejmowało wykorzystanie kilku platform - szczególnie przy tak dużym udziale rynku aplikacji mobilnych.



## DODATEK A

# Aplikacja RESTful Web api

Aplikacja serwująca API na potrzeby pracy dyplomowej. Architektura zakłada uruchomienie projektu jako kontenera Docker. Serwis został opublikowany pod adresem <http://mgr.arturrybak.com/>

W stworzonej aplikacji zamodelowałem pewien wycienek większego API, który pozwala na uzyskiwanie informacji o cenach chemikaliów i ich zmianach w czasie. Można dzięki temu uzyskać informacje o wszystkich chemikaliach (a właściwie seriach danych związanych z chemikaliami), co nie jest polecane ze względu na wydajność i rozmiar odpowiedzi. Dla porównania dostępne jest żądanie o ceny chemikalium. Może ono zostać wzbogacone o filtry i parametr sortowania, które potrafią znacznie skrócić czas oczekiwania i rozmiar odpowiedzi. Odpowiednie testy zostały opisane w rozdziale 3. pracy dyplomowej.

Kilka przykładów wywołań API:

Listing 5. Przykłady wywołań API

```
1 GET http://mgr.arturrybak.com/chemicals
2 GET http://mgr.arturrybak.com/prices
3 GET http://mgr.arturrybak.com/prices/ETH
4 GET http://mgr.arturrybak.com/prices/BZE?fields=title,type,status,region,concept
   ↳ &sort=-concept
```

Instrukcja deploymentu:

Listing 6.

```
1 $ docker build -t github.com/benhall/nancy-demo-hosting-docker .
2   $ docker run -d --name nancy-demo -p 8080:8080
   ↳ github.com/benhall/nancy-demo-hosting-docker
3   $ docker port nancy-demo 8080 | curl "xargs` :8080/prices/BZE"
```

Jeżeli używane jest boot2docker należy podać adres IP maszyny wirtualnej jako część wywołania cURLa.

Listing 7.

```
1 $ docker port nancy-demo 8080 | curl "boot2docker ip":8080/prices/BZE"
```

# Bibliografia

- [1] Kirsten L. Hunter. *Irresistible APIs. Create Web APIs that developers will love.* Manning Publications, 2015.
- [2] Patch method for http [online], [dostęp: 25 sierpnia 2015] dostępny w internecie: <http://tools.ietf.org/html/rfc5789>.
- [3] Charles proxy [online], [dostęp: 25 sierpnia 2015] dostępny w internecie: <https://www.tuffcode.com/>.
- [4] Httpscoop [online], [dostęp: 25 sierpnia 2015] dostępny w internecie: <https://www.wireshark.org/>.
- [5] Wireshark [online], [dostęp: 25 sierpnia 2015] dostępny w internecie: <https://www.wireshark.org/>.
- [6] Fiddler [online], [dostęp: 25 sierpnia 2015] dostępny w internecie: <http://www.telerik.com/fiddler>.
- [7] curl [online], [dostęp: 25 sierpnia 2015] dostępny w internecie: <http://curl.haxx.se/>.
- [8] Kore.io [online], [dostęp: 15 czerwca 2015] dostępny w internecie: <http://kore.io/>.
- [9] Best practices for designing a pragmatic restful api [online], [dostęp: 15 czerwca 2015] dostępny w internecie: <http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>.
- [10] Chen-Che Huang, Jiun-Long Huang, Chin-Liang Tsai, Guan-Zhong Wu, Chia-Min Chen, and Wang-Chien Lee. Energy-efficient and cost-effective web api invocations with transfer size reduction for mobile mashup applications. *Wireless Networks*, 20(3):361–378, April 2014.

- [11] How to design rest apis for mobile? [online], [dostęp: 15 czerwca 2015] dostępny w internecie: <http://www.redotheweb.com/2012/08/09/how-to-design-rest-apis-for-mobile.html>.
- [12] Efficient mobile apis using apache thrift [online], [dostęp: 15 czerwca 2015] dostępny w internecie: <http://blog.whitepages.com/reducing-data-with-apache-thrift/>.
- [13] Rest and the promise of secure and efficient application delivery [online], [dostęp: 15 czerwca 2015] dostępny w internecie: <https://blog.akana.com/rest-and-the-promise-of-secure-and-efficient-application-delivery/>.
- [14] Creating an efficient rest api with http [online], [dostęp: 15 czerwca 2015] dostępny w internecie: <http://mark-kirby.co.uk/2013/creating-a-true-rest-api/>.
- [15] How to handle challenges with api security and efficiency [online], [dostęp: 15 czerwca 2015] dostępny w internecie: <http://searchsoa.techtarget.com/feature/How-to-handle-challenges-with-API-security-and-efficiency>.
- [16] Ilya Grigorik. *High Performance Browser Networking*. O'Reilly, 2013.
- [17] Walter Binder. *Designing and Implementing a Secure, Portable, and Efficient Mobile Agent Kernel: The J-SEAL2 Approach*. PhD thesis, der Technischen Universität Wien, April 2001.
- [18] How we moved our api from ruby to go and saved our sanity [online], [dostęp: 18 czerwca 2015] dostępny w internecie: <http://blog.parse.com/learn/how-we-moved-our-api-from-ruby-to-go-and-saved-our-sanity/>.

# Spis tabel

3.1. Porównanie czasów zapytań sql dotyczących <i>Benzenu</i> przed i po denormalizacji . . . . .	47
---	----



# Spis rysunków

1.1.	Strona artykułu w serwisie <i>na:temat</i> z przyciskami społecznościowymi	10
	<b>Interpretacja:</b> Jak wiele podobnych stron informacyjnych <i>na:temat</i> oferuje przyciski społecznościowe, z których każdy powiązany jest z API serwisu macierzystego, tutaj: <i>Facebook</i> , <i>Twitter</i> , <i>Google+</i>	
	<b>Źródło:</b> <i>na:temat</i> [online], [dostęp: 16 czerwca 2015] dostępny w internecie: <a href="http://natemat.pl/145775">http://natemat.pl/145775</a>	10
1.2.	Sekcja komentarzy do artykułu w serwisie <i>na:temat</i>	11
	<b>Interpretacja:</b> Niektóre z serwisów zrzucają całe funkcjonalności i interakcje z użytkownikami na baki API społecznościowych. Tutaj: wykorzystanie formularza komentarzy zintegrowanego z <i>Facebook'iem</i>	
	<b>Źródło:</b> <i>na:temat</i> [online], [dostęp: 16 czerwca 2015] dostępny w internecie: <a href="http://natemat.pl/145775">http://natemat.pl/145775</a>	11
1.3.	Ranking najpopularniejszych API z których korzystają deweloperzy.	13
	<b>Interpretacja:</b> Najpopularniejsze serwisy webowe oferują najbardziej rozchwytywane API. Zarówno ze względu na bogatą treść jaką za ich pomocą „wyciągnąć” ale i środki jakie czołowi gracze inwestują w rozwój swoich platform. Owocuje to przemyślanym i odpowiednim interfejsem a także znaczną penetracją rynku.	
	<b>Źródło:</b> ProgrammableWeb [online], [dostęp: 16 czerwca 2015] dostępny w internecie: <a href="http://www.programmableweb.com/apis">http://www.programmableweb.com/apis</a>	

1.4. Podgląd pierwszego testowego żądania z Twilio API <i>Interpretacja:</i> Twilio umożliwia „wyklikanie” i wypełnienie formularza online, który zostaje „w locie” przetłumaczony na żądanie. Możemy zobaczyć przykładową implementację w kilku najbardziej popularnych językach programowania a także za pomocą programu <i>curl</i> . Źródło: własne . . . . .	15
1.5. Odpowiedź z API Twilio, która potwierdza wysłanie testowego SMSa <i>Interpretacja:</i> W odpowiedzi widzimy, czego możemy się spodziewać w zwrocie z Twilio API. Warto zwrócić uwagę na to, że nawet bez zagłębiania się w strukturę łatwo domniemać znaczenie poszczególnych pól. Źródło: własne . . . . .	16
1.6. Elementy składowe żądania HTTP <i>Interpretacja:</i> Podstawowa struktura żądania zawiera informacje o nagłówkach, metodzie, URLu (adresie) oraz ciele. Źródło: [1], wersja IV, s. 27 . . . . .	17
1.7. Element składowy odpowiedzi na żądanie HTTP <i>Interpretacja:</i> Ustrukturyzowana odpowiedź na żądanie HTTP zawiera kod HTTP informujący o statusie, nagłówki oraz w niektórych przypadkach ciało. Źródło: [1], wersja IV, s. 28 . . . . .	18
1.8. Charles podczas pracy z jednym z API <i>Interpretacja:</i> W górnej części znajdują się chronologicznie wykonywane żądania, które można zaznaczyć, by u dołu otrzymać szczegółowe. Sposób prezentacji w poszczególnych zakładkach pozwala na czytelny, ukierunkowany obraz, gdzie skupić możemy się na parametrach żądania, parametrach odpowiedzi, nagłówkach, sposobie autentakcji, ciasteczkach, wykresach związanych z dłużością trwania poszczególnych żądań. Źródło: własne . . . . .	20

1.9. Wynik działania programu z listingu 1.	
<b>Interpretacja:</b> W odpowiedzi otrzymaliśmy ciało w postaci json'a oraz informacje o statusie.	
<b>Źródło:</b> własne . . . . .	21
1.10. API jako produkt poboczny, podejście standardowe	
<b>Interpretacja:</b> API jest osobną ścieżką w linii produktów firmy i używane jest do zapewnienia działania jednego lub kilku integracji, czasem klientów mobilnych.	
<b>Źródło:</b> własne . . . . .	23
1.11. API jako jeden z wielu interfejsów	
<b>Interpretacja:</b> API jest tylko jednym z interfejsów, jakie komunikują się z backendem i choć ma swoich klientów, to równolegle działają systemy komunikujące się z backendem w podobny lub wręcz zupełnie odmienny sposób.	
<b>Źródło:</b> własne . . . . .	24
1.12. API first	
<b>Interpretacja:</b> API jest pośrednikiem pomiędzy wszystkimi aplikacjami klienckimi a backendem, stanowi warstwę bez której komunikacja między tymi systemami nie może mieć miejsca.	
<b>Źródło:</b> własne . . . . .	25
1.13. API z mojej perspektywy	
<b>Interpretacja:</b> API widziane z mojej perspektywy. Zaimplementowane jako <i>sideproduct</i> w strategii firmy. Służy do komunikacji backendu i platform mobilnych, przy czym samo nie jest osobnym byteam a wrasta w główny produkt - aplikację webową.	
<b>Źródło:</b> własne . . . . .	28
1.14. API z mojej perspektywy	
<b>Interpretacja:</b> API widziane z mojej perspektywy. Zaimplementowane jako <i>sideproduct</i> w strategii firmy. Służy do komunikacji backendu i platform mobilnych, przy czym samo nie jest osobnym byteam a wrasta w główny produkt - aplikację webową.	
<b>Źródło:</b> własne . . . . .	30

1.15. Kontroler aplikacji webowej Interpretacja: Wyszczególnione elementy stanowią różnicę w stosunku do korespondującego kontrolera 1.16 Źródło: własne . . . . .	31
1.16. Kontroler aplikacji mobilnej Interpretacja: Wyszczególnione elementy stanowią różnicę w stosunku do korespondującego kontrolera 1.15 Źródło: własne . . . . .	32
2.1. Układ strony startowej w aplikacji mobilnej dla platformy iPad Interpretacja: W układzie strony, która jest częścią <i>dashboardu</i> widzimy kilka wyodrębnionych <i>widgetów</i> ułożonych w „klockową strukturę” Źródło: własne . . . . .	34
2.2. Zrzut żądań wysyłanych przez aplikację iPadową, po to by uwierzytelnić/autoryzować użytkownika oraz wyświetlić treść pierwszego ekranu. Interpretacja: Na zielono oznaczone zostały żądanie niezwiązane z konkretnym ekranem a ze strukturą API. Na niebiesko żądanie autentykacyjne. Pozostałe służą wyświetleniu strony startowej. Źródło: własne . . . . .	35
2.3. Zrzut żądań wysyłanych, by wyświetlić jedną ze stron aplikacji. Interpretacja: Pierwsze trzy żądania odpowiadają za pobranie struktury strony i poszczególnych jej modułów. Pozostałe żądania pobierają treść dla każdego z modułów. Źródło: własne . . . . .	35
2.4. Porównanie architektury metod i zasobów Interpretacja: Architektura oparta na czasownikach wykonuje bardzo określoną czynność i serwuje jej rezultat. Architektura oparta na zasobach operuje rzeczownikami, nie mówi nic o sposobie ich wykorzystania. Źródło: własne . . . . .	41

- 2.5. Jeden z widgetów, prezentujący kilka wybranych aspektów i ocen ryzyka dla wybranych krajów.

**Interpretacja:** Widgety w aplikacji prezentują informację przygotowaną, przetworzoną i wycelowaną w specyfczną analizę. Zawierają ścisłe zdefiniowany zestaw informacji, który ma docelowo ułatwiać analizę sytuacji klientom.

**Źródło:** własne . . . . . 42

- 2.6. Wykorzystanie zasobów API w całym przekroju platform i nośników.

**Interpretacja:** Szybko rozwijający się Internet rzeczy - *Internet of Things* jest potencjalnym medium przekazu informacji, którego badanie i testowanie warto powierzyć klientom w przypadku braku własnych zasobów.

**Źródło:** własne . . . . . 43

- 3.1. Żądania o ceny chemiczne benzenu

**Interpretacja:** Czas oczekiwania na zwrot wynosi średnio 9 sekund w przypadku środowiska lokalnego. Wielkość odpowiedzi wynosi 3.21 MB, co jest dużym narzutem dla platform mobilnych.

**Źródło:** własne . . . . . 50

- 3.2. Żądania z limitowaną liczbą pól

**Interpretacja:** Czasy trwania żądań i wielkość odpowiedzi dzięki limitowaniu tylko potrzebnych pól, mogą zostać znacznie zredukowane.

**Źródło:** własne . . . . . 51

- 3.3. Żądania z limitowaną liczbą pól, wykres

**Interpretacja:** Porównanie czasu trwania żądań z limitowaną liczbą pól obrazuje ile możemy zyskać na oszczędnej komunikacji z serwerem

**Źródło:** własne . . . . . 51

3.4. Zaptyania z wykorzystaniem sortowania i bez niego	
<b>Interpretacja:</b> Dobrze zaimplementowane sortowanie nie ma wpływu na czas i wielkość odpowiedzi, pozwala za to już na etapie komunikacji z serwerem przygotować dane do prezentacji.	
<b>Źródło:</b> własne	52

## Oświadczenie

Ja, niżej podpisany(a) oświadczam, iż przedłożona praca dyplomowa została wykonana przeze mnie samodzielnie, nie narusza praw autorskich, interesów prawnych i materialnych innych osób.

.....

data

.....

podpis