

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 Bzyl

Gdańsk 2015

Streszczenie

Problematyka tej pracy dotyczy schematu komunikacji między urządzeniami mobilnymi a serwerem, jaki narzuca projekt interfejsu w postaci WebAPI. Przeanalizuję konsekwencje, jakie niesie za sobą obranie podejścia RESTful jako wzorca projektowego oraz gdy grupa klientów ograniczona zostanie do aplikacji mobilnych, tj. takich, które uruchamiane są z poziomu urządzeń przenośnych: telefonów, tabletów, zegarków, opasek, odzieży. Kierując się potrzebami oraz wymaganiami wpisanymi w sprzęt, przedstawię oraz zutylizuję w istniejącym projekcie praktyki, powszechnie uważane za właściwe, m.in. konstruowanie interfejsu, który silnie utylizuje protokół HTTP i zapewnia interfejs ograniczający wielkość odpowiedzi. Słuszność rzeczonych praktyk potwierdzona będzie oszczędnym gospodarowaniem dostępnego pakietu danych - minimalizacja transferu osiągnięta w moim przypadku to nawet kilka megabajtów w jednym żądaniu - czy poprawnym reagowaniem na osiągalność i topologię sieci.

Modelowe przykłady API które oferują firmy takie, jak Facebook, Twitter, Flickr czy Google posłużą jako punkt odniesienia oraz źródło wzorców, którymi powiniśmy się kierować podczas tworzenia własnego WebAPI. Przykłady tych firm pokazują dobrze skonstruowany interfejs, oparty o zasoby w formie rzeczników można przekuć w rozwijającą się platformę, czy ekosystem produktów. Skupię się na najczęstszych problemach i możliwych rozwiązańach struktury in-

terfejsu, oraz powszechnych rozwiązań dla sztandardowych problemów takich jak rozszerzalność czy obsługa błędów.

Wreszcie, zajmę się analizą wdrożonego interfejsu, na kilku przykładach postaram się przetransformować API istniejącej aplikacji w ten sposób, by intuicyjnie spełniało założenia REST. Zmierzone czasy żądań i wielkości odpowiedzi a także czas oczekiwania na wykonanie zapytań SQL zobrazują, że z pomocą kilku dobrych wzorców jesteśmy w stanie zapewnić całkiem wydajny i responsywny produkt.

Słowa kluczowe

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

Spis treści

Wprowadzenie	8
1. Rozumienie WebAPI	11
1.1. Jakie API jest użyteczne?	14
1.2. Praca z Web API	16
1.2.1. Podstawowa struktura składniowa	19
1.2.2. Inspekcja i symulowanie żądań przychodzących i wychodzących	21
1.3. API First	25
1.3.1. API jako produkt poboczny	26
1.3.2. API jako jeden z wielu interfejsów	27
1.3.3. API first	29
1.3.4. API w codziennej pracy	33
2. Projekt WebAPI	39
2.1. Wartość biznesowa	44
2.2. Budowa poprawnego modelu	47
2.2.1. Rzeczowniki - zasoby, kontra czasowniki - akcje	48

<i>Spis treści</i>	7
3. Wykonanie WebAPI	53
3.1. Implementacja na wybranej platformie	54
3.2. Składnia zapytań i odpowiedzi przy użyciu protokołu HTTP	58
3.2.1. Konstrukcja adresów URL	58
3.2.2. Filtrowanie, sortowanie, przeszukiwanie tekstu	59
Zakończenie	63
Bibliografia	64
Spis tabel	67
Spis rysunków	68
Oświadczenie	77

Wprowadzenie

Urządzenia mobilne opanowały sposób w jaki komunikujemy się ze światem. Smartfon jest nam bliższy niż niejeden z członków rodziny. Nie ma w tym jednak nic dziwnego, skoro stał się naszą bramą do poznania, doświadczania i dzielenia się światem i przeżyciami. Mobilność sięga jednak coraz dalej. Telefon, tablet, zegarek, opaska na nadgarstku, okulary, buty, a niedługo pozostałe części naszej garderoby są lub będą częścią świata mobilnego. Niezaprzeczalny jest wpływ telefonu i tabletu na nasz rytm życia, wobec czego, biorąc pod uwagę powszechność rozwiązań mobilnych, postanowiłem przyjrzeć się kluczowej kwestii, jaką jest wydajny, dobrze zaprojektowany interfejs komunikacyjny, pozwalający trzymać w ryzach nawet najbardziej zaawansowany system.

Smartfon - będący naszym osobistym centrum sterowania musi w sposób efektywny porozumiewać się z użytkownikiem oraz światem zewnętrznym. Ubogie byłyby bowiem aplikacje mobilne z których korzystamy gdyby nie pomoc web serwisów. Z jednej strony dostarczają nam one niezbędnych informacji o pogodzie, ruchu drogowym, zmianach cen ropy czy naszych osiągnięciach na siłowni. Z drugiej konsumują to wszystko, co chcemy zachować dla siebie lub czym chcemy pochwalić się przed innymi. Zmieniają sposób w jaki jesteśmy połączeni z ludźmi z najbliższego otoczenia ale i najdalszych zakątków globu.

Intuicyjnie, jako ludzie, potrafimy się porozumiewać. W większości sytuacji wiemy, czy to, co i o czym mówimy kogoś rani, czy bawi. Wiemy, czy dialog z naszym rozmówcą "klei się" w całość, czy raczej jest to nerwowe odbijanie pytań i odpowiedzi. Całą naszą umiejętności i intuicję w tym zakresie opieramy na doświadczeniu. O ile nie jest więc dla nas problemem wysyłanie i odbieranie prostych komunikatów, o tyle zdolność do porozumiewania się między narodami wspólnym językiem biznesowym z zachowaniem wrażliwości na kulturę i poglądy rozmówców jest niezaprzeczalnie wyzwaniem.

Podobnie jest z aplikacjami mobilnymi: o ile łatwo jest z pomocą dostępnych narzędzi wysłać wiadomość w świat a może nawet odebrać odpowiedź na zadane pytanie, o tyle trudniej robić to w sposób wydajny i dobrze ustrukturyzowany. W ciągu ostatnich lat udało się nam jednak wypracować kilka reguł i wytycznych, które czynią komunikację szybką, czytelną, bezpieczną. Reguły, które czynią komunikację wydajną.

Od trzech lat usilnie przyglądam się światu przez pryzmat telefonu i jego możliwości. Przez ten czas dostrzegłem różne próby komunikacji, gdzie jako medium służy smartfon. Pierwotna jego funkcja - możliwość wykonywania połączeń głosowych za pośrednictwem sieci GSM jest dobrze zdefiniowana i wykorzystana ale funkcjonalnie mało rozszerzalna. Krótkie wiadomości tekstowe urozmaicili formę komunikacji ale prawdziwą rewolucję przyniósł tani mobilny internet oraz upo-

wszechnienie się darmowych punktów dostępowych sieci WiFi. Procentowy udział w ruchu internetowym, jaki jest kreowany przez platformy mobilne wzrasta z upływem czasu. Dzieje się to za sprawą coraz doskonalszych aplikacji, przemyślanych interfejsów i wygodzie dostępu do informacji, jaką zyskujemy będąc w podróży czy odpoczywając po ciężkim dniu na kanapie, bo i ta nie jest naturalnym środowiskiem w którym korzystamy z komputera stacjonarnego czy laptopa.

Jako developer aplikacji mobilnych jestem świadomy, że sukces napisanych przeze mnie programów nie ma jednego źródła. Składają się na niego: przemyślany i przyjemny design aplikacji, rozpoznanie, zrozumienie i prawidłowe modelowanie potrzeb użytkownika oraz przypadków użycia. To i tak tylko wierzchołek góry lodowej. Doświadczenie jednak podpowiada, że dobrze zaprojektowana komunikacja klient (aplikacja mobilna) - serwer (w postaci WebAPI) wiele rzeczy upraszcza i pozwala się skupić na problemach funkcjonalnych, zamiast marnotrawić czas na kolejne techniczne *gotchas*.

ROZDZIAŁ 1

Rozumienie WebAPI

Zanim skupimy się na kwestiach technicznych oraz na projektowaniu API, powinniśmy wiedzieć, po co w ogóle chcemy stwarzać możliwość na manipulację naszymi zasobami za pomocą zewnętrznego interfejsu. Pytanie zgoła podstawowe. Istotne jest jednak byśmy wracali do niego często w trakcie projektowania czy implementacji. Łatwo bowiem zgubić ideę którą chcieliśmy się podzielić podczas gdy pochłonie do reszty walka z przeciwnościami technicznymi oraz dylematy związane ze strukturą naszego interfejsu.

Projektowanie API nie jest żadnym wyjątkiem. Podobnie jest z tą pracą magisterską, podobnie z każdym pomysłem na biznes. Cel powinien nie tylko uświetcać środki ale i wyznaczać dalszą drogę. Cel powinien być zawsze widoczny.

By zrozumieć jak dobrze projektować API, postarajmy się więc najpierw zrozumieć jakie są wymagania i oczekiwania stawiane przed jego twórcami.

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

The screenshot shows a news article from the website 'na:temat'. At the top left is a small profile picture of Jakub Noch. Next to it is his name 'JAKUB NOCH' and the text '2 godziny temu'. Below this is a vertical column of social sharing icons: Facebook (with a red border), Twitter, Google+, and Email. The main image is a portrait of a man in a camouflage military uniform. To the right of the image is a sidebar titled 'ZOBACZ TAKŻE:' containing four links to other articles with small thumbnail images.

JAKUB NOCH
2 godziny temu

f
76
Twitter
g+
Email

ZOBACZ TAKŻE:

- Jak zarabiać na swoich pasjach? Zainwestowała w siebie – odniasta sukces**
- Wino i sery idealnym połączeniem? To kulinarny trend, który pokochali Polacy**
- Samochody elektryczne przyszłością polskich miast?**
- Jak w 3 krokach rozpocząć biznes w internecie?**
- Kup bilet y i jedź do Sopotu. Na Dancing. I koniecznie zabierz babcię**

- W GROM najistotniejszy nie jest stopień, a to, co kto ma w głowie – mówi naTemat dowódca tej jednostki, płk Piotr Gaśtał. • Zrzut ekranu z kanalu YouTube.com/GRAK

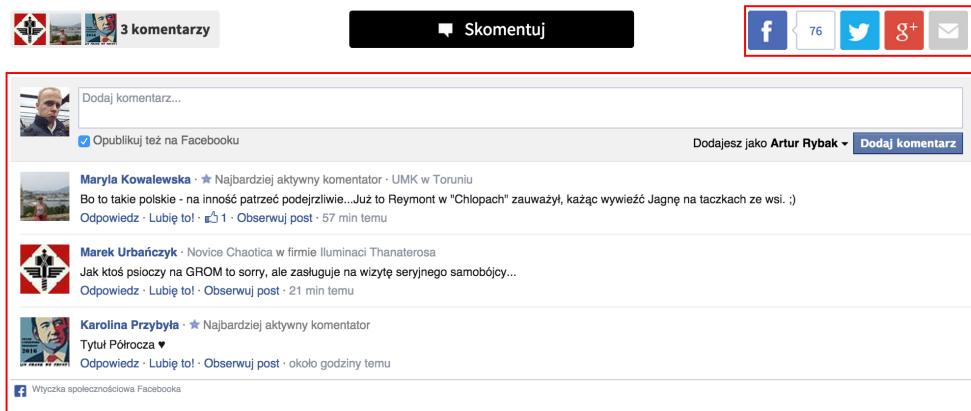
- W GROM na przykład nie ma typowego wojskowego drylu, stukania obcasami na każdym kroku. Jesteśmy dla siebie partnerami, rozmawiamy ze

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>

W powyższym przykładzie widzimy, że twórcy serwisu *na:temat* umożliwiają swoim czytelnikom rozpowszechnianie treści za pomocą przycisków społecznościowych. Jest to jeden z najprostszych sposobów budowanie zasięgu treści i docieranie do szerokiego grona odbiorców. W ten sposób autor artykułu, czy wydawca stawia na treściwe i wzruszające emocje artykuły z którymi czytelnik się identyfikuje. 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 postanowiło oddać w ręce zewnętrznego serwisu całą sekcję komentarzy, która z mozołem powielana jest przez wiele portali. Jak można zaoberwować na rysunku 1.2, serwis zyskuje gotowe rozwiązanie, z całym dobrodziejstwem inwentarza ale i ograniczeniami. Zdecydowanie mniej tu komentarzy, które są wulgarnie, opryskliwe i niekulturalne, ponieważ każdy z imienia i nazwiska podpisuje się pod swoimi słowami. Ogranicza to ilość potrzebnej moderacji i przenosi tę odpowiedzialność na Facebooka. Minusem 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 *marshup*, tj. takich, które łączą w sobie cechy kilku serwisów. Z pewnością istnieje uzasadnienie dla istnienia aplikacji, oferujących funkcjonalność specyficzną dla ich dziedziny. Nie potrzebują one elastyczności i gdyby porównać je do garniturów, to mielibyśmy do czynienia jedynie z szytymi 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 budujemy system mocno powiązany. Pozwala *ścinac zakręty* i skracac proces powstawania aplikacji.

Jeśli jednak chcemy pobudzić społeczność i dać jej narzędzie, które będzie mogła KREATYWNIĘ wykorzystać, to być może wystawienie REST API jest dla nas najlepszym wyborem. Do najbardziej rozchwytywanych API należą te, oferowane przez serwisy społecznościowe takie, jak *Facebook*, *Twitter* czy *LinkedIn*. Deweloper, który otrzymuje takie narzędzie, ma natychmiast dostęp do danych o milionach a nawet miliardach użytkowników. Oczywiście „za darmo” otrzymujemy jedynie dane publiczne, tj. takie, które użytkownik sam zgodził się udostępniać zarówno i wszystkim. Istnieje jednak kilka sposobów, by poprosić użytkownika o bardziej newralgiczne, dokładniejsze informacje. Wiąże się to z reguły z autentykacją

(m.in OAuth o którym w kolejnych rozdziałach) i założeniem konta w serwisie, który będzie potrafił jednoznacznie określić, 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 internautów, ich słabości, ciekawość tylko po to by przechwycić cenne informacje, pieniądze lub zaatakować kolejną, spersonalizowaną dawką spamu. Twórcy API, którym zależy na budowaniu pozytywnego wizerunku swojego produktu mogą wówczas takie wystąpienia szybko ukröcić, wyłączając dostęp do API dla konkretnych aplikacji. Bywa, że w przypadku bardzo obieganych serwisów jest to proces zautomatyzowany, sterowany algorytmami z dziedziny sztucznej inteligencji.

MOST POPULAR APIs



1. Facebook	Track this API	6. LinkedIn	Track this API
2. Google Maps	Track this API	7. Kayak	Track this API
3. Skype	Track this API	8. Waze	Track this API
4. Netflix	Track this API	9. Yahoo Weather...	Track this API
5. Telegram	Track this API	10. Pinterest	Track this API

Rysunek 1.3: Ranking najpopularniejszych API z których korzystają deweloperzy.

Interpretacja: Najpopularniejsze serwisy webowe oferują najbardziej rozchwitzywane API. Zarówno ze względu na bogatą treść jaką za ich pomocą „wyciągnąć” ale i środki jakie członkowie 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. Wszelakie narzędzia, przykładowy kod, tutoriale, samouczki są nieodzowną pomocą w takich przypadkach. Często, w przypadku raczy wielkości Facebooka strony deweloperskie urastają do całkiem sporego rozmiaru, tworząc cały ekosystem. Społeczność, 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 doskonaly 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

Curl Ruby PHP Python Node.js Java C#

Note: Toggle showing your **Auth Token** by [clicking here](#) Check out the curl manpage

```
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]
```

Make Request

Note: 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

Mnie zajęło 7 minut wysłanie pierwszego SMSa, za pomocą Twilio API. I choć może to poniżej celu jaki sobie ta firma stawia, to z pewnością efekt został osiągnięty. W głowie zakiełkowało już co najmniej 10 pomysłów, jak takie API można wykorzystać i gdzie mógłbym je wpleść do moich aplikacji. Rysunek 1.5 jest dowodem na to, że można. Ponadto w informacji zwróconej otrzymujemy kod informujący o utworzonej encji (**201**) **CREATED** oraz samą encję w postaci JSONa.

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"
  }
}
```

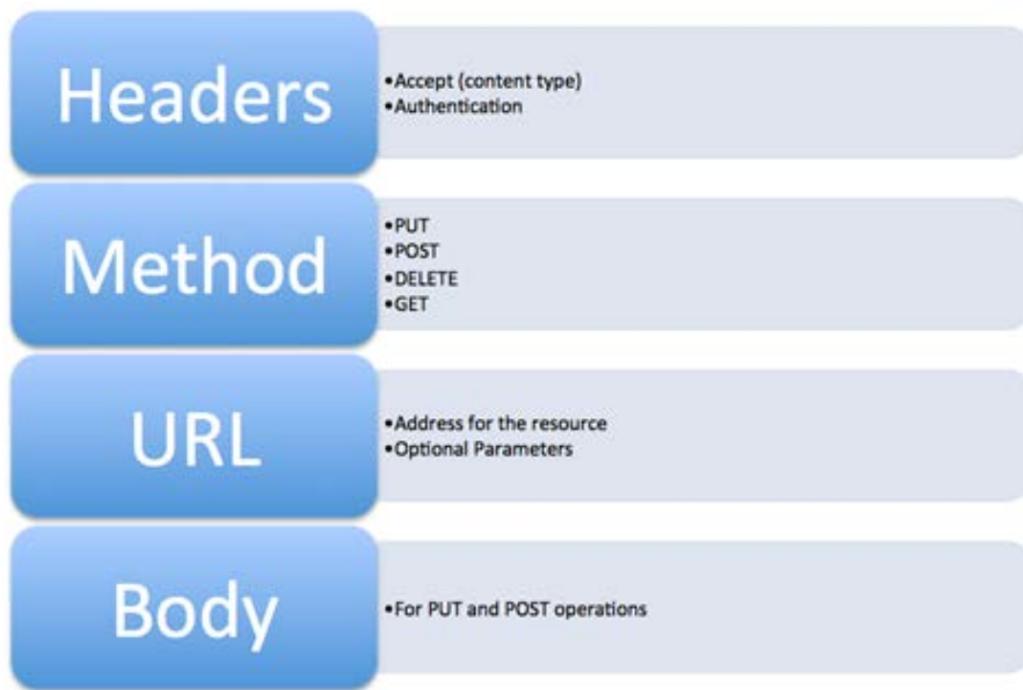
Rysunek 1.5: Odpowiedź z API Twilio, która potwierdza wysłanie testowego SMSa

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

Tak łatwe wprowadzenie jak w przypadku Twilio, generuje wymierną wartość marketingową. Deweloper, który w pozostałych sytuacjach zaprośczyłby „Dlaczego ma mnie to obchodzić?”, co sprowadza się do dwóch innych pytań: „Co mogę Z TYM zrobić” i „Jak mogę TO zrobić?” w tym momencie zobaczy nie kolejne przeszkody a klocki, budulec jego przyszłych aplikacji.

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.

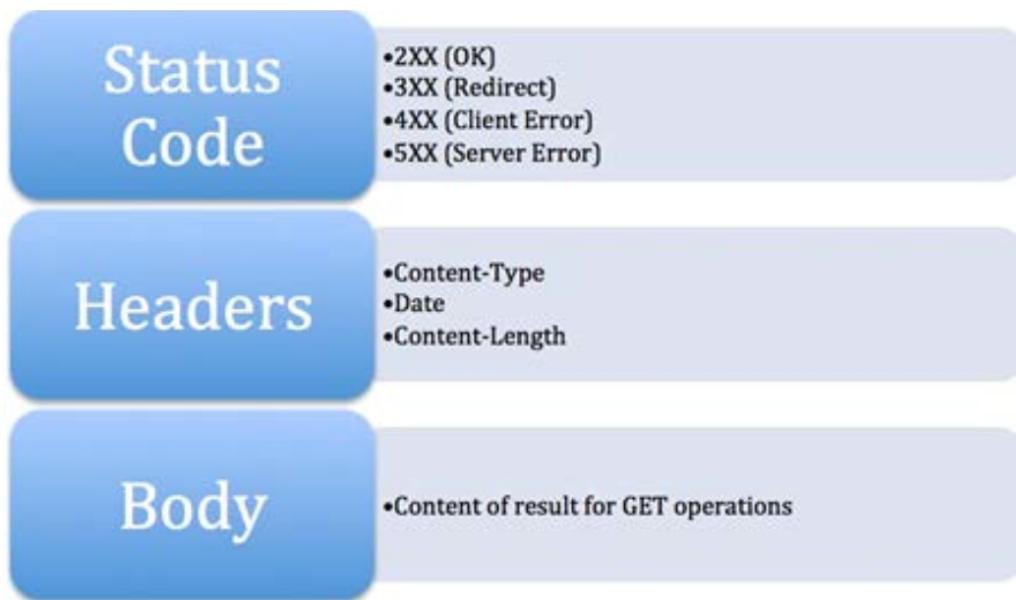
Źródło: [1], wersja IV, s. 27

Podczas gdy mówiąc o API możemy mieć na uwadze skomplikowane systemy, Web API są interfejsami tak prostymi jak to tylko możliwe. Podstawowymi akcjami przezeń oferowanymi są akcje *CRUD* - Create Read Update Delete. Ponadto interfejsy Web API wykorzystują cechy protokołu HTTP a co za tym idzie spodziewają się, że żądanie, tak jak na rysunku 1.6 będzie zawierało informacje o adresie, nagłówkach, metodzie oraz w przypadku metod PUT/PATCH/POST ciało żądana-

nia. Oczywiście jest to tylko konwencja i nikt nie zabroni nam obsłużyć żądań na własną rękę w zupełnie dowolny sposób. Należy jednak pamiętać, że jest to niezgodne z intuicją i powszechną praktyką. Co za tym idzie utrudnia zrozumienie intencji naszego API i podwyższa „próg wejścia” potrzebny do tego by z API korzystać. Największe zdziwienie może tutaj budzić odniesienie do metody PATCH, która nie była pierwotnie częścią standardu HTTP, została jednak wprowadzona dokumentem RFC 5789 [2] w marcu 2010 roku. Szczególnie ważna ze względu na platformy mobilne, których zadaniem jest minimalizacja transferu.

Postępowanie według wzorca pozwala deweloperom poczynić pewne założenia. Jak choćby te, 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 unikalność zasobu występującego pod danym adresem.

Założenia co do wartości zwracanych również będą bazować na protokole HTTP, tj. na kodzie zwracanym, nagłówkach oraz ciele odpowiedzi tak jak obrazuje to 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

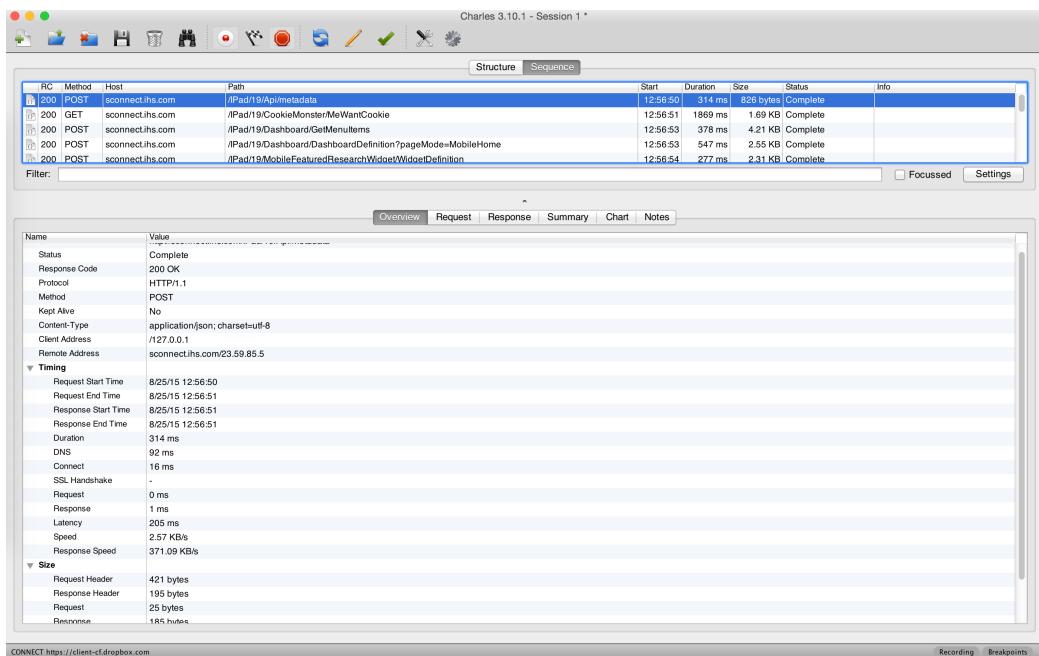
wychodzących

Codzienna praca z API, nawet jeśli nie jest ono przez nas rozwijane wymaga użycia narzędzi, które pozwolą na podglądarkanie żądań i odpowiedzi wraz z wszelkimi metadanymi. Każdy ma swoje ulubione narzędzia, jednak w tej pracy pozwolę sobie korzystać z takiego samego zestawu narzędzi, jak w przypadku mojej codziennej pracy nad aplikacjami mobilnymi.

Po pierwsze Charles [3]. Jest to proxy, poprzez które przechodzą wszelkie żą-

dania z lokalnej maszyny a także wszelkie żądania urządzeń, symulatorów i emulatorów na których ustawiliśmy proxy w ten sposób, by wskazywało na maszynę z zainstalowanym Charlesem. Domyslnie program instaluje się na porcie 8888 a cechuje go prosty i elegancki wygląd a także duże możliwości konfiguracji: ograniczenie 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, polegające 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. Naturalnie wszystko wymaga rozwagi i rozumnego wykorzystania, gdyż narzędzie może okazać się mieczem obosiecznym i doprowadzić do niestabilnego działania zarówno aplikacji klienckiej jak backendu. Alternatywami dla Charles'a mogą być *HTTPStoop* [4], *Wireshark* [5] czy *Fiddler* [6]

Na rysunku 1.8 widzimy przykładowy obraz jaki uzyskuje programista, który chce zbadać jakie są 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 czasów oczekiwania na odpowiedź. Dzięki temu widzimy 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 brakuje mi w Charles'ie to możliwość łatwego komponowania żądań - ustawiania ich parametrów, zapisywanie, tworzenie scenariuszy. Tutaj moim wyborem do prosty rozwiązań jest zdecydowanie 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 json'a wraz informacjami statusu, 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 dowiesz się dlaczego.

```
$ curl -L 1.5h :: ~/Projects/PRIVATE/mqr/listings (master *% u)
* Trying 23.59.85.5...
* Connected to sconnect.ihs.com (23.59.85.5) port 80 (#0)
> GET /IPad/19/Dashboard/DashboardDefinition?pageMode=MobileHome HTTP/1.1
Host: sconnect.ihs.com
Authorization: Basic [REDACTED]
User-Agent: curl/7.4.0
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
< Content-Type: application/json; charset=utf-8
< Connection: Keep-Alive
< Set-Cookie: IHS_SSID_SESS=px2FB0cBAUCA0jZMKMSzypGanYbsRX-cx2dRN0zBpv4x2B9178xbroPw14dbC1G5j0n5J3E3Qx3Dx3D-ZF4b0pFIWGN7AmRcrAa1wx3Dx3D-TRBuJy8S019imjH012F0kAx3Dx3D
SSD_Cookie=IHS_UT=xxj32-LJm2PAph2-nEMfZSpzSmSN0x0x/KSbgc5q1bxAMg7j0MRAZY830f; domain=.ihs.com; path=/; HttpOnly
< Set-Cookie: IHS_SSID_SESS_STAGE=vhFR0qPmG0OH0p9pxeQAS3d3d2cHl0nvnKKd5JhWURScSm%2sb1XtxAH05m2f0B2Us2b8WtXHUbGrcnDqz2F5hLqggwF6JmVpx%2b%2fSwkEv18VmVs2b1gfcJa1qjJN0ZgxUGR0rEc152H69JKpPGKs5wdkRganDrx0s0z2CePDMoUNVQoX1wBzUmh8p82gj1W3dN0rakNr1vB0M#E2Pos150u5Ns2bVw1w0134jWtkfNkCM1jBqQGCrFy5f0W1W1Zc0t3y0kvJ4E093fkqZomphdPfNkpw1y0hreeZEz0P3bR140jE%2bvhehsu77NP1AFxByukKvvWlPV3BfKMsbs3J0fu93Xec0JU08z2fJFizAm2bA0XuM23McKyWcT1W3NcPd7nmII1qzYFFJgnHhwgs3d; domain=.ihs.com; expires=Tue, 25-Aug-2015 18:39:46 GMT; path=/; HttpOnly
```

Rysunek 1.9: Wynik działania programu z listingu 1.

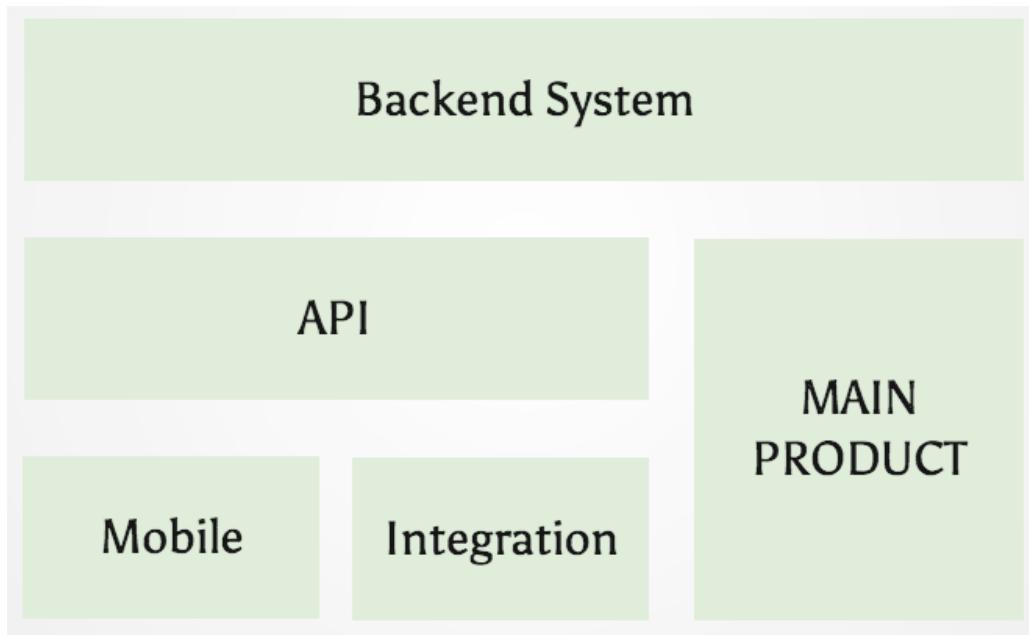
Interpretacja: W odpowiedzi otrzymaliśmy ciało w postaci json'a oraz informacje o statusie.

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, najpierw trzeba zrozumieć, dlaczego API są tworzone w ogóle. 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 chcemy stworzyć kolejne aplikacje, musimy napisać więcej systemów, które będą bezpośrednio dotykać backendu ALBO rozszerzyć API w ten sposób, by wspierało oba alternatywne produkty. Co więcej, API jest często uważane za „wabik” lub „czynnik wyróżniający”, który pozwala zwabić klientów, jednak traktowane jest jako *nice to have* zamiast być ważnym składnikiem ekosystemu produktów w firmie. Takie podejście generuje poważne problemy, szczególnie uwzględniając zasoby, jako, że konkuruje z produktami przynoszącymi zauważalne przychody.

1.3.1. API jako produkt poboczny



Rysunek 1.10: API jako produkt poboczny, podejście standardowe

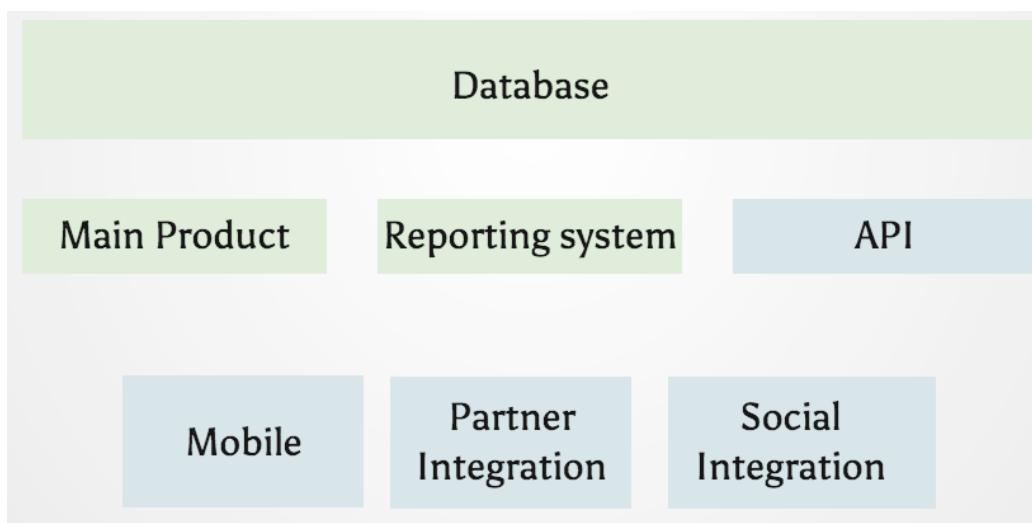
Interpretacja: API jest osobną ścieżką w linii produktów firmy i używane jest do zapewnienia działania jednego lub kilku integracji, czasem klientów mobilnych.

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, nadal będziemy mieli do czynienia z różnicą w odbiorze i użytkowania tych aplikacji. Teoretycznie wykonalne jest by wszystkie ścieżki konsekwentnie oferowały taki sam *user experience*, jednakże wymaga to znacznie więcej pracy w porównaniu do restrukturyzacji, która uczyni z API jądro systemu.

Sceptycy argumentują, że odseparowanie API i głównego produktu może ochronić tenże, przed atakami za pośrednictwem API. Szczera 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 rośnie. Efektem ubocznym, choć pożądany 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.

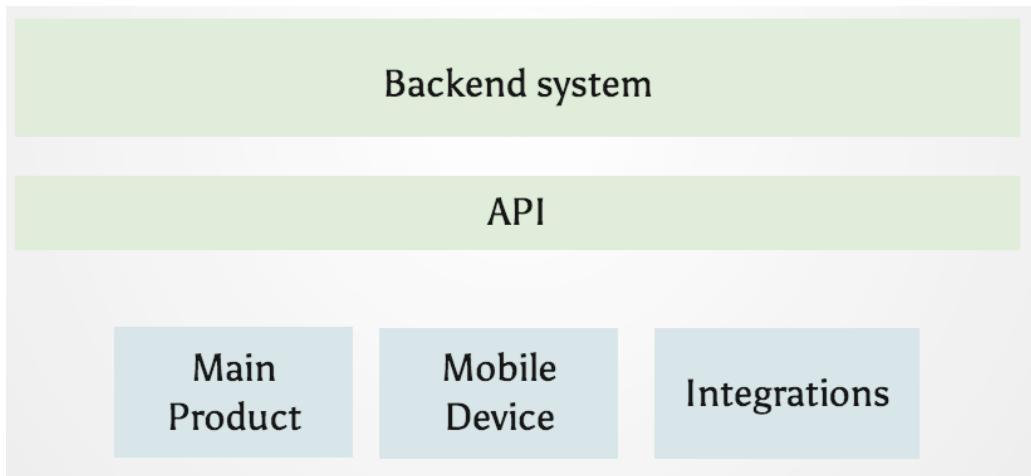
Źródło: własne

Podobną, choć nieco bardziej rozbudowaną architekturą jest traktowanie API jako jednego z wielu interfejsów. W obrazowanym przez rysunek 1.11 schemacie

można zauważyc coś bardzo niepokojącego. 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 ktoś dodał 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 zdarza się, że z powodu „dziurawej komunikacji” nie wszystkie systemy zostają zaktualizowane lub dzieje się to w sposób niepożądany. Podczas gdy raz udało nam się określić, jak naszym użytkownikom będzie najwygodniej korzystać z systemu, powinniśmy dążyć to tego by wspierać nasze rozwiązańe w każdym miejscu.

Rzeczona architektura objawia się szczególnie, gdy kilka zespołów próbuje tworzyć produkty bez wspólnej, spójnej wizji. U podłoża 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, że nie wymusza to na systemie, by wszystkie zasoby były dla każdej z aplikacji klienckich dostępne. 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 odpowiedzialnymi za frontend. API stanowi tu warstwę pośredniczącą, którą każdy z tych zespołów będzie próbował pociągnąć w swoją stronę. Jednakże żad-

ne rozwiązanie nie dojdzie do skutku, jeśli nie będzie mogło posłużyć pozostałym zespołom.

Dodatkowym benefitem 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. Podczas, gdy w przypadku schmatu z rysunku 1.11 gdy zmieni się baza danych, wymagana jest praca po stronie każdego z zespołów i weryfikacja czy zmiany nie wprowadzają błędów. Systemy mogą te być skonstruowane w ten sposób, że niemal niemożliwym będzie wykrycie wszystkich zależności. Stąd 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 pożądany model API powinno być dobrze udokumentowane i całkowicie zabezpieczone przed dostępem klientów jakimkolwiek innymi środkami. Mając to na uwadze będziemy mogli korzystać z tego, że:

1. funkcjonalności są równe na każdej platformie

Nie ma potrzeby inwestowania czasu i osób we wdrażanie nowych funkcjonalności na każdej z platform z osobna. Są one dobrze przetestowane a zmiana dokonywana jest tylko po jednej stronie. Możliwym jest również zmiana

wewnętrznej implementacji systemu z którego API czerpie, bez wpływania na pracę deweloperów.

2. zwiększymy szybkość

Nieintuicyjnym może się wydawać, że dodanie warstwy do systemu może zredukować ilość pracy w ogóle, jaką deweloperzy muszą wykonać. Wspólny interfejs pozwala na dzielenie bibliotek i schematów. Chociaż początkowo 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

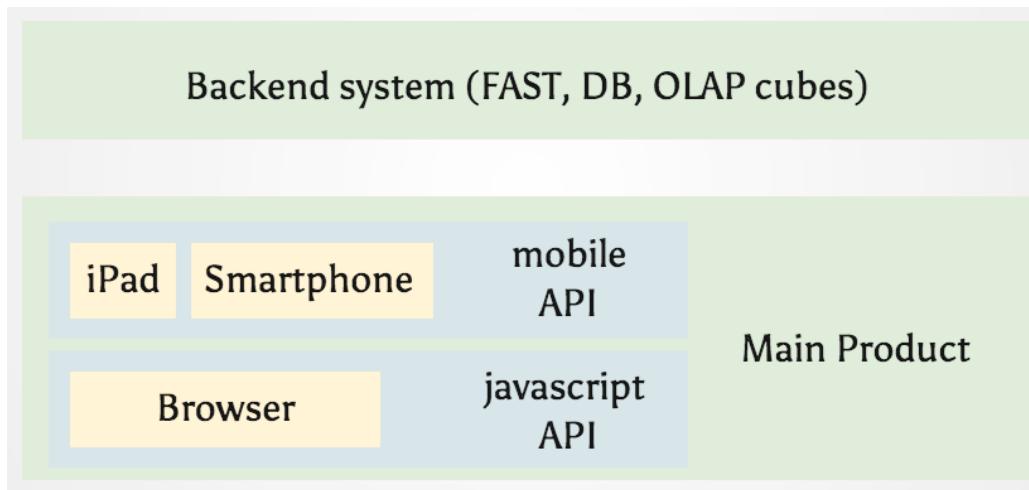
Autentykacja i authoryzacja dostępu do zasobów to narzędzia dzięki którym możemy decydować dla jakich aplikacji czy użytkowników poszczególne funkcje mają być dostępne a dla których dostęp ten chcemy ograniczać. Jednym ze sposobów radzenia sobie z tym problemem jest implementacja OAuth o której w dalszych rozdziałach. Jedną z największych zalet w tej implementacji jest możliwość blokowania złośliwych, niewydajnych lub łamiących regulamin aplikacji. 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 nie zwalnia nas z troski o to czy system jest funkcjonalny i łatwy 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 i absorbową coraz to nowe pomysły i spostrzeżenia.

Instagram początkowo był aplikacją jedynie mobilną. Użytkownicy jednak z czasem zaczęli domagać się strony web i integracji. Firma nie miała zasobów, które pozwoliłyby na te implementacje. Sfrustrowani niezależni programiści zajęli się więc *inżynierią wsteczną* i „zhackowali” Instagram. To zmusiło firmę do upublicznienia przynajmniej 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

Motywacją do napisania tej pracy stało się moje środowisko pracy, gdzie architektura wyewoluowała w sposób odstający od standardów przyświecających największym firmom. Nie było to problemem, dopóki system nie zaczął się rozrastać. Tymczasem po lekturze poprzednich rozdziałów każdy już na pierwszy rzut oka zauważ, że można lepiej: wydajniej, bezpieczniej.

Obecną architekturę systemu nad którym pracuję przedstawia rysunek 1.13. Mamy tu do czynienia z systemami backendu tj. bazą danych, silnikiem wyszukiwania (FAST), kostkami OLAP. Z backendem komunikuje się aplikacja główna. Jest to aplikacja webowa, korzystająca wpradzie z wielu niezależnych serwi-

sów. Stanowi ona role 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.

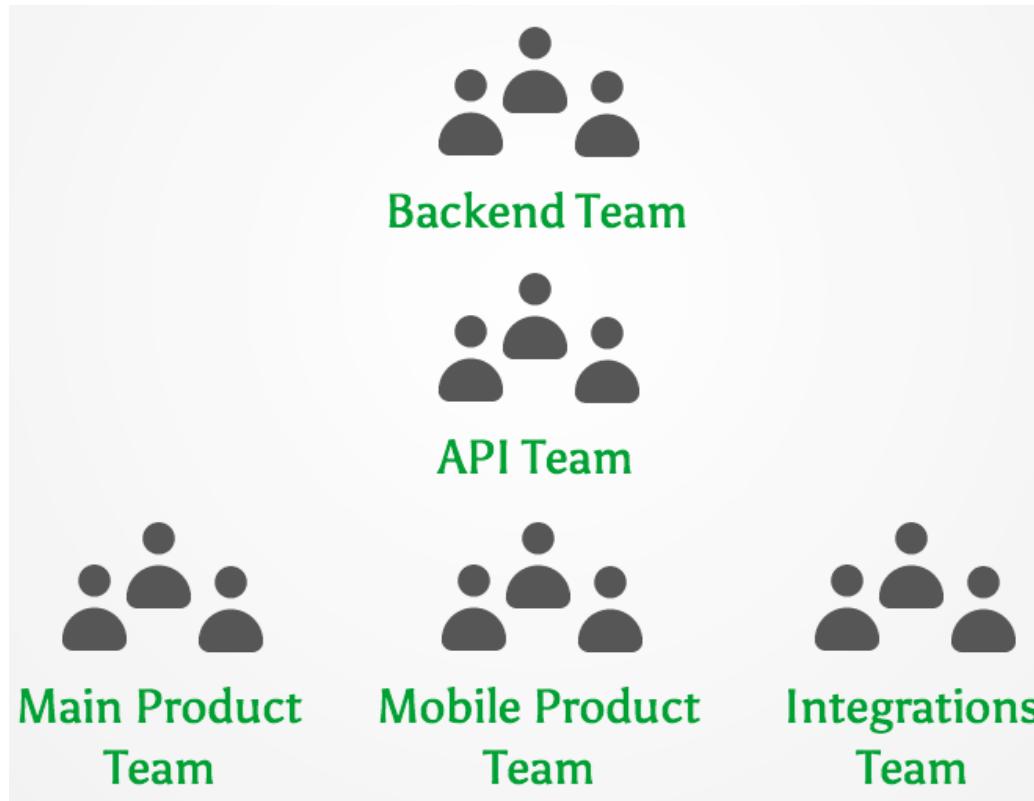
Bardzo odczuwalny jest szczególnie podział na wersję webową i mobilną. Po pewnym czasie zaczeliśmy dociekać, jaka może być tego przyczyna i jakie są realne konsekwencje posiadania oddzielnego API. Niestety okazało się, że przyczyna leży bardzo głęboko, mianowicie, w wewnętrznej strukturze organizacyjnej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

W tej nieszczęśliwej sytuacji jestem także ja, wraz z moim (siódmym) zespołem, którzy mamy zlecenia od wszystkich analityków (a właściwie od żadnego), tak bowiem dzieje się gdy odpowiedzialność za *nice to have* produkt jest zbiorowa. Próbujemy się wpasować w ramy, które zostały już ufundamentowane. Co gorsza, każdy z zespołów webowych okazuje się mieć inny na te ramy pomysł, więc bardo cierpimy z powodu breaku platformowości. Wielokrotnie prostujemy *hard-*

kodowane implementacje ale i wpasowujemy się by pokrętnie móc współdziałać z każdym z zespołów.

Nie jest to sytuacja komfortowa ani pożądana, a na dodatek powoduje sytuacje, w których jedna platforma psuje 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 expli-
cite do tego przeznaczony. Systemy i patologie jednak bardzo szybko rosną ob-
jętościowo, trudno więc przeprowadzić generalną reformę. Postanowiłem więc w
tej pracy, jak można poprawnie skonstruować pewien bardzo mały wycinek API,
które zadowoli każdą z platform. Dowodem niepotrzebnych duplikacji niech bę-
dą poniższe dwa rysunki: 1.15 i 1.16. Pośród dzielenia i zabezpiecznia własnego
interesu należy zauważyc, że obie implementacje korzystają z tego samego repo-
zytorium. Różni je jedynie typ zwracany i związane z tym mapowanie. Niesie to
za sobą konsekwencje w postaci różnic powstających pomiędzy modelami. Można
więc odważnie stwierdzić, że funkcjonalnie, oba kontrolery należałoby połączyć w
jeden, serwujący API dla wszystkich platform.

Do problemu „łączenia” należy podchodzić ostrożnie. 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 przy-
padku platformy mobilnej jest to trudno wykonalne, choćby ze względu na obo-
wiązkowy proces zatwierdzenia przez właściciela sklepu (np. *App Store*, jaki jest
narzucony na każdą aplikację). Poza tym im więcej komponentów od siebie zależy,
tym bardziej trudniej zsynchronizować ich publikację, dlatego stosuje się wersjonowanie.
Dla aplikacji mobilnej rzecz nieodzowna. Dla platformy webowej novum, do któ-
rego należy się przyzwyczaić.

```

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

    public ChemicalHeadlineAnalysisWidgetController(
        IWidgetServiceProvider 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 servicesProvider,
        IWidgetIPadSettingsRepository<NullSettings> widgetIPadSettingsRepository,
        IChemicalHeadlineAnalysisRepository repository)
        : base(servicesProvider, 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

Podczas gdy w tym rozdziale staraliśmy się zrozumieć sens istnienia i mechanizmy determinujące wygląd i strukturę API, w następnej części zastanowimy się, jak konstruować poprawne syntaktycznie i semantycznie interfejsy. Wiemy już, czego oczekują programiści i jak wygląda ich interakcja z interfejsami Web API. Znamy modele które sprzyjają lub utrudniają budowę skalowalnego produktu czy linii produktów. Czas by odnaleźć wartość biznesową i postawić linie podziału, które wyznaczą obszar w jakim nasze API będzie się poruszało.

ROZDZIAŁ 2

Projekt WebAPI

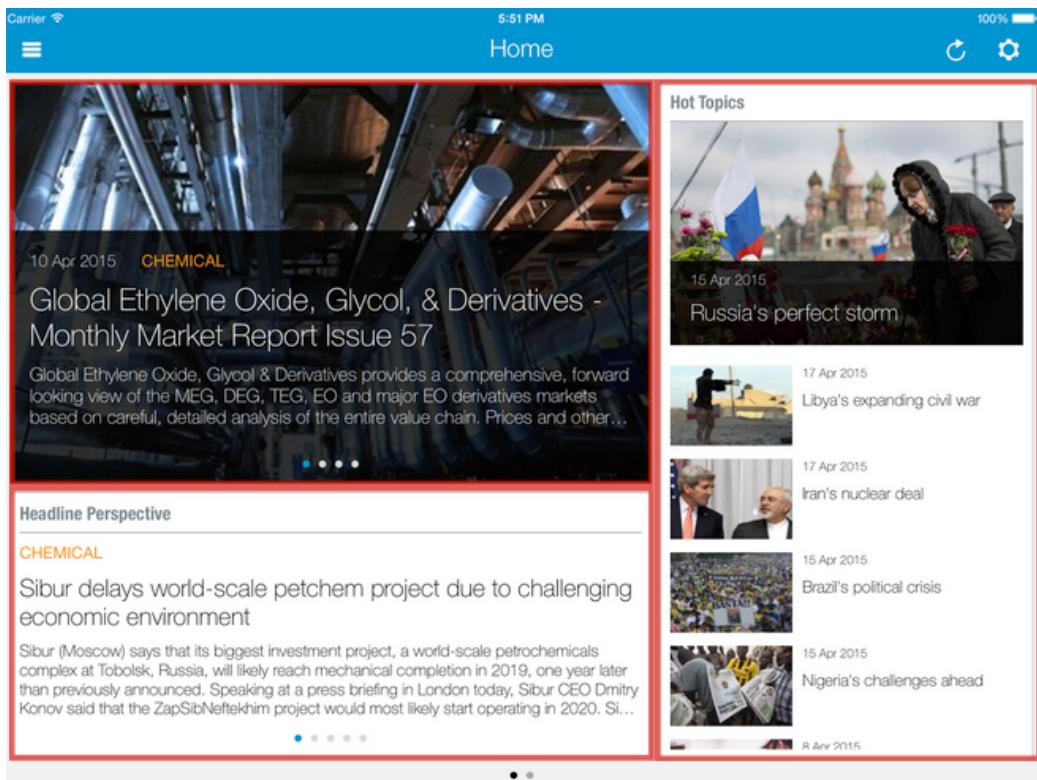
Do myślenia o dobrze zaprojektowanym API najczęściej skłaniamy się w kilku sytuacjach. Naczęściej:

- 1. podczas jego pierwszego planowania i projektowania**, co jest naturalne, choć nieoczywiste. Wielokrotnie mamy do czynienia z interfejsami, które wyrosły organicznie na intuicji i wcześniejszych doświadczeniach deweloperów.
- 2. gdy usiłujemy poprawić wydajność**, okazuje się bowiem, że API narzuca na aplikację konieczność częstego komunikowania się i synchronizowania żądań lub gdy oczekiwanie na odpowiedź trwa zbyt długo.
- 3. gdy funkcjonalność, której potrzebujemy** okazuje się być bardzo skomplikowana do uzyskania co za tym idzie wymaga od nas wykonania wielu kroków.
- 4. gdyauważamy redundancje w kodzie i dług techniczny** co powiązane jest z nieudolną architekturą

Bardzo ważne jest, by pierwszą wersję API dobrze zaprojektować - szczególnie, gdy wiemy, że będzie używane przez wiele aplikacji. Nie mamy wtedy obowiązku myślenia o utrzymywaniu wcześniejszych wersji co umożliwia bardzo elastyczne modyfikowanie naszego podejścia. Z czasem jednak, gdy narzuć istniejących wersji powstaje, musimy bardzo ostrożnie migrować nasze pomysły i płynnie przechodzić pomiędzy jedną architekturą a drugą, mając na uwadze, że przez kolejny tydzień, miesiąc, rok, niezależnie wydane aplikacje mogą nie zostać zaktualizowane a co za tym idzie, będą bazowały cały czas na starej wersji naszego API. Niezbędne jest tu zapewnienie odpowiedniego wsparcia i dokumentacji dotyczącej tego, jak dokonać migracji z wersji na wersję.

Moją motywacją do przyjrzenia się strukturze była architektura API z którego korzystam w projekcie. Dotychczas rozwijałem API o nowe zasoby, tym razem jednak chciałem wejść głębiej, poznać mechanizmy z których korzystamy i wysnuć przypuszczenia odnośnie ich genezy i zastosowania. Oto co odkryłem.

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 mieści się 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 złego, gdyby nie protokół HTTP 1.1, który „rozdmuchuje” ilość przesyłanych pakietów o własne nagłówki.

200 POST connect.ih.s.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.ih.s.com	/iPad/16/Dashboard/GetMenuItems		801 ms	7.36 KB	Complete
200 POST connect.ih.s.com	/iPad/16/Saved/GetReadLaterFolder		1764 ms	10.32 KB	Complete
200 POST connect.ih.s.com	/iPad/16/LandingPageFilters/Index		967 ms	3.83 KB	Complete
200 POST connect.ih.s.com	/iPad/16/Dashboard/GetSubMenuItems?pageMode=LandingPage		404 ms	3.26 KB	Complete
200 POST connect.ih.s.com	/iPad/16/LandingPage/GetTopStories		2624 ms	6.15 KB	Complete
200 POST connect.ih.s.com	/iPad/16/LandingPage/GetFeaturedResearch		1038 ms	4.73 KB	Complete
200 POST connect.ih.s.com	/iPad/16/LandingPageHotTopics/Content		776 ms	5.31 KB	Complete
200 POST connect.ih.s.com	//iPad/HotTopicViewer/Show?source=gi&docid=2617787		748 ms	6.95 KB	Complete
200 POST connect.ih.s.com	//iPad/Display/Show?source=gi&docid=1030847		596 ms	4.59 KB	Complete
302 GET connect.ih.s.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.ih.s.com	/iPad/16/ImageResize/Render?path=%5C%5Cihdeneclp01%5CCastle_prod_CIFS1%5CStaticAttachments%5C8b%5C503.jpg&title=image&width=748&height=497&crop=true		406 ms	3.64 KB	Complete
302 GET connect.ih.s.com	/iPad/16/ImageResize/Render?path=%5C%5Cihdeneclp01%5CCastle_prod_CIFS1%5CStaticAttachments%5C9e%5C518.jpg&title=image&width=748&height=497&crop=true		366 ms	3.64 KB	Complete
200 GET connectfiles.ih.s.com	/FileDownload.ashx?token=cAm4ufE5r78UfmPL5jlxrT++3Sb0g8t4tTfN2fkAbA/B4k0Dx/V1Te24y9M1WaoOzjuFqdqYqTxOhSEuzsSMims0CU7pn6jMFt+9pKTSU8dYyrTw+yhZEXP5fYS		497 ms	65.02 KB	Complete
302 GET connect.ih.s.com	/iPad/16/ImageResize/Render?path=%5C%5Cihdeneclp01%5CCastle_prod_CIFS1%5CStaticAttachments%5Cd%5C525.jpg&title=image&width=748&height=497&crop=true		419 ms	3.64 KB	Complete
200 GET connectfiles.ih.s.com	/FileDownload.ashx?token=dDwy3b0BTzVdN/4zP4anJXjTQfONlqgDG9ogfLTozguZJDTKdIXV+oPLIAmpdAbt7RRRdvYdI+GOWWNYnySyAv4t6WVITA9FYAc23jdQzLEXwpWA6YF7bIEzbBZ1Dai		798 ms	129.03 KB	Complete
200 POST connect.ih.s.com	//iPad/CeraViewViewer/Show?source=gi&docid=2796542		772 ms	8.18 KB	Complete
200 GET connectfiles.ih.s.com	/FileDownload.ashx?token=YkfjZcRNv+tfSFUpKkZkS0eP+D0oJshf7yBlAjkeUoLY6fP7YvELHwv10zk8u05FWhrEvK		725 ms	129.03 KB	Complete
200 GET connectfiles.ih.s.com	/FileDownload.ashx?token=LMWYoMS10i2XgZlDaqvPZUKM1IYhQqvChicjlJ7zip68vVA+V4oEb7cvklLrQ2cQ51_51IeoREYEALhwFtfKxsPT3+q+8oJv0o7cc1ZPDR9pKxLmkdfc35kGCVdGe		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.ih.s.com	/iPad/16/Dashboard/DashboardTaxonomy?pageMode=OrganizationGeneral		1231 ms	3.31 KB	Complete
200 POST connect.ih.s.com	/iPad/16/Dashboard/GetSubMenuItems?pageMode=OrganizationGeneral		823 ms	3.35 KB	Complete
200 POST connect.ih.s.com	/iPad/16/Dashboard/WidgetsForTaxonomy?pageMode=OrganizationGeneral&		485 ms	4.87 KB	Complete
200 POST connect.ih.s.com	/iPad/16/OrganizationKeyReportsWidget/Content		2770 ms	5.81 KB	Complete
200 POST connect.ih.s.com	/iPad/16/OrganizationProfileBasicWidget/Content		700 ms	4.23 KB	Complete
200 POST connect.ih.s.com	/iPad/16/OrganizationProfileLatestWidget/Research		875 ms	9.74 KB	Complete
200 POST connect.ih.s.com	/iPad/16/OrganizationSupportCompaniesWidget/Content		669 ms	3.72 KB	Complete
200 POST connect.ih.s.com	/iPad/16/OrganizationHeadquartersWidget/Content		570 ms	3.77 KB	Complete
200 POST connect.ih.s.com	/iPad/16/OrganizationPeerInformationWidget/Content		921 ms	3.65 KB	Complete
200 POST connect.ih.s.com	/iPad/16/OrganizationContactsWidget/Content		583 ms	3.93 KB	Complete
200 POST connect.ih.s.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

Wprawne oko zauważa także, że większość 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*. Najczęściej dlatego, że zostaną oszukane przez kod HTTP 200 świadczący o sukcesie wykonywanej operacji.

Jest to zaledwie zgrubne przyjrzenie się kilku grzechom projektu, w którym uczestniczą. Być może w dalszej części uda mi się te grzechy rozgrzeszyć. W pozostałych przypadkach pozostanie przyznać się do winy i pokutnie zabrać się za naprawę stanu rzeczy, przynajmniej w jakiejś małej części.

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 staniemy w obliczu sytuacji, gdy trzeba będzie wytłumaczyć 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 próbujemy na bieżąco usprawiedliwiać rozwój naszego interfejsu i nie znajdujemy potwierdzenia w realnych, mierzalnych danych. Najszerzsza sytuacja to ta, gdy API jest naszym głównym produktem a jego rentowność i stosowność możemy wyrazić w konkretnych przychodach dla firmy. Kryje się tu jednak pułapka dla API, będącego wsparciem głównego produktu. Za czyna bowiem ono konkurować z aplikacją, której wpływ na przychody firmy jest mierzalny, podczas, gdy sam fakt istnienia i użytkowania API nie jest przekładalny na przychody.

Twilio jest tutaj dobrym przykładem, gdzie API jest głównym produktem i samo na siebie zarabia. Każde żądanie wysłane do API kosztuje niewielką, określoną ilość pieniędzy. 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 przyjdzie nam określenie wartości biznesowej dla API, którego celem jest zwiększenie zaangażowania użytkowników w platformę, tak jak ma to miejsce w przypadku *Facebooka* czy *Twitter'a*. Wykorzystanie zasobów tych serwisów i intensywne zaangażowanie użytkowników zapewnia proporcjonalne wpływy z reklam oraz podnosi wartość danych, jakimi rzeczone platformy dysponują. Jeżeli jednak nie jesteśmy potentatem na rynku portali społecznościowych, dobrą motywacją jest zacieśnianie naszej relacji a klientami. Oferowanie interfejsu, który pozwoli łatwo zintegrować nasze systemy z systemami klienta będzie korzystnie wpływać na decyzję podczas wyboru spośród podobnych rozwiązań oferowanych przez konkurencję.

Możemy się zastanowić, jak udostępnienie API przez uczelnię wpłynęłoby na rozwój systemów administracji, oceniania studentów, prowadzenia kursów. Na 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. Związanego jest to z rozwojem technologii oraz powiązane z prowadzonymi badaniami. Jest także do datkową okazją wdrożenia studentów w systemy informatyczne. API w tym przypadku pozwoliłoby pracownikom i studentom na produkowanie własnych aplikacji

cji i interfejsów, które bazowały na realnych danych, bez potrzeby ręcznego przesłania 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 w *hackathon'y* i kreatywność deweloperów a najlepsze rozwiązania włącza do swojej platformy. API jest tutaj furtką do kreatywności.

W moim przypadku, w firmie, która zajmuje się sporządzaniem analiz rynków energetycznych, chemicznych, nowych technologii, motoryzacyjnych i innych. Gdzie kapitałem firmy są dokumenty, serie danych, informacja, nie tylko ta zagradowana, ale też przetworzona i opisana. Gdzie oferowane nas narzędzia służą dużym firmom, do podejmowania krytycznych decyzji o rozwoju i zachowaniu na rynku oferujemy **gotowe rozwiązania**: aplikacje webowe, mobilne, desktopowe. Z pewnością są one dużym ułatwieniem dla analityków firm klienckich ale czy oprócz większych elementów w postaci aplikacji nie powinniśmy oferować także bloków składowych? 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, jak zbudowany jest nasz model aplikacji i co mogliśmy zaoferować klientom powstającego API, a także jak powinniśmy ustrukturyzować powstający interfejs.

2.2. Budowa poprawnego modelu

Jest kilka cech, którymi dobrze zaprojektowany model API powinien się cechować. Powinien być zrozumiały dla klientów i niepotrzebnie ich nie zaskakiwać.

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 listingu² świadczy iż pomimo adres w domniemaniu odnosi się do API RESTowego, to z tym rodzajem API nie mamy do czynienia w przypadku *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, świadczący o pozytywnym wyniku operacji. Dla wielu moich kolegów może to być nieintuicyjne. Wyobraźmy sobie również, że podczas integrowania wielu API w jednej aplikacji musimy każde obsłużyć z osobna, ponieważ nawet informacje o błędach nie są standardowe. Oczywiście wszystko jest wykonalne, jednak wymaga od nas dodatkowej uwagi, zamiast po prostu skorzystania z gotowych frameworków.

Projekt API powinien zakładać zróżnicowanie form klienckich, a co za tym idzie, ich szczególne wymagania. W przypadku aplikacji mobilnych prawie na pew-

no do listy wymagań będziemy musieli dopisać: autentykację na platformie, zapobieganie nieprzewidzianym *crashom* aplikacji, krótki czas oczekiwania. By sprostać tym wymaganiam będziemy musieli uwzględnić w naszej architekturze chociażby możliwość pobrania wszystkich informacji dla jednego ekranu w pojedynczym żądaniu, minimalny rozmiar przesyłanych danych, możliwość sprecyzowania, które sekcje danych są potrzebne.

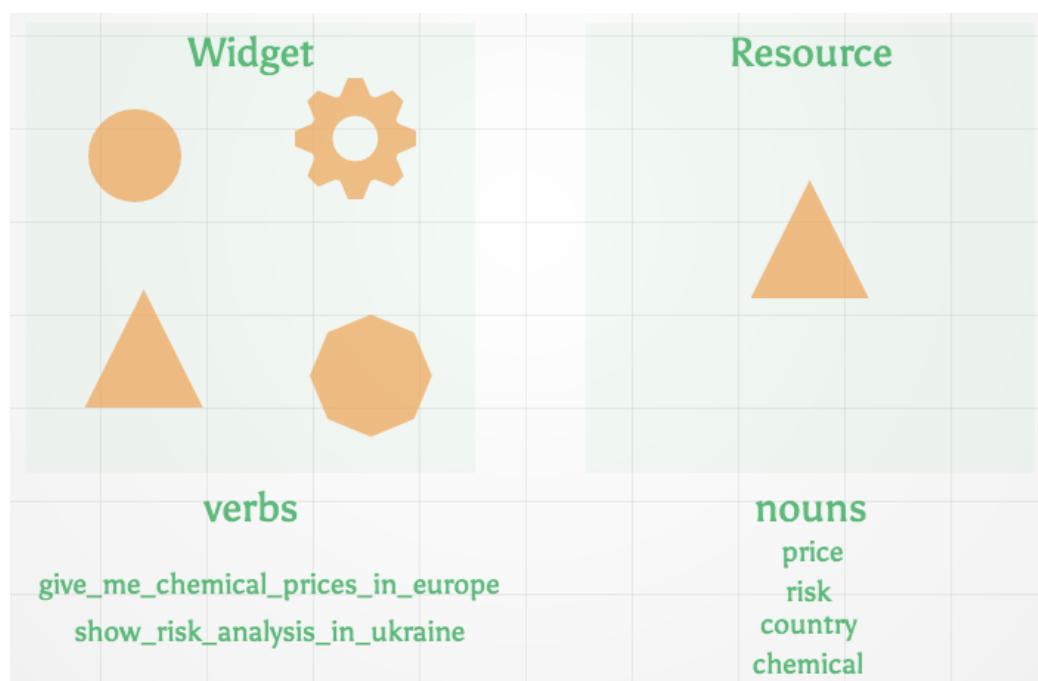
Wymagania te biorą się z ograniczonego pasma sieciowego, częstych przerwach w połączeniach, choćby przy wejściu do tunelu, windy czy lasu. Nadal większość aplikacji i urządzeń nie jest w stanie zapewnić wystarczająco efektywnego przetwarzania równoległego, więc konieczność synchronizacji kilku żądań dla jednej strony będzie skutkowała kiepskim zachowaniem interfejsu użytkownika.

2.2.1. Rzecznowniki - 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, skąd biorą się one biorą i jak powinniśmy podejść do tej kwestii. Otóż efektem ponad dwugodzinnej rozmowy był wniosek: *Powinniśmy mieć do tego API. Zamiast „rozdmuchiwać” aplikację webową powinniśmy wydzielić część odpowiedzialną stricte za prezentację interfejsu użytkownika i część (aplikację) odpowiedzialną za serwo-*

wanie 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, wręcz utopijne. O ile patrzyć na całość projektu. Gdyby jednak stopniowo wydziełać konkretne części aplikacji i oferować API dla nich, to jest szansa na powolne, mozolne budowanie przyjaznego interfejsu.

Dlaczego właściwie potrzeba nowego interfejsu? Postaram się to zobrazować przykładem.



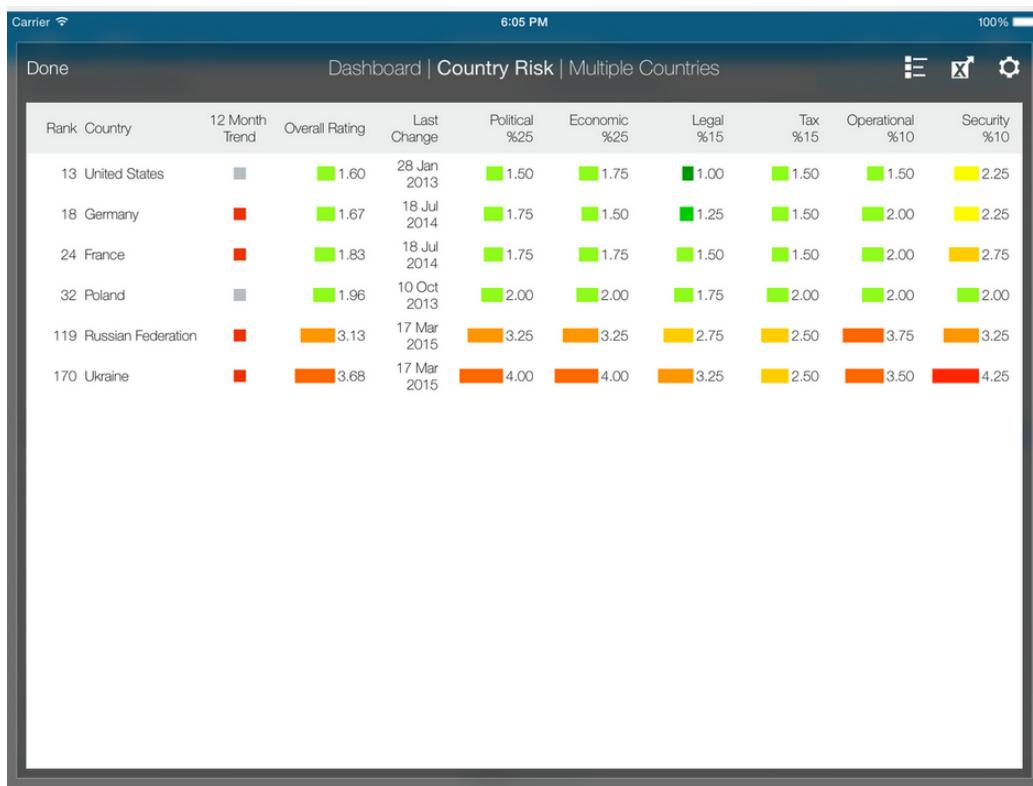
Rysunek 2.4: Porównanie architektury metod i zasobów

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

Nasze aplikacje w obecnym kształcie korzystają składającą się z poukładanych w większą strukturę widgetów (rys. 2.1). Prosząc 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. Gdybyśmy się jednak zastanowili, co możemy z obecnego API skonstruować, to okazałoby się, że możemy stworzyć jedynie podobne aplikacje, składające się z takich samych części, różniących się jedynie kwestiami wizualnymi, warstwą prezentacji. Szybko zaczęłoby nam 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).

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, produkt, który zatrzymał się, ginie bezpowrotnie zastąpiony alternatywami umożliwiającymi kreatywność.



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 (rys. 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 wy-

próbowanie wszystkiego, dlatego otwartość i pomoc przeróżnych *community* jest nieoceniona w akceleracji rozwoju naszych produktów. Gdy mamy do czynienia z generycznymi zasobami mamy dowolność mikowania danych i łączenia ich z innymi dostępnymi API. Możemy zapewnić własne przetwarzanie i skupić się tylko na tym, co nas interesuje, podczas 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

Skoro znamy już biznesowe pobudki, które skutkują powstaniem API. Wiemy jak koncepcyjnie powinniśmy kreować zasoby i jakie konsekwencje niesie za sobą używanie RESTful API opartego na rzeczownikach. Czas przyjrzeć się szczegółowo implementacyjnym „Smaczkom”, które mogą nam nie wydawać się oczywiste ale i na które nie zwracamy uwagi, gdy tworzymy organiczne interfejsy bazując na intuicji i doświadczeniu. W poniższych rozdziałach skupię się na konstruowaniu poprawnego syntaktycznie interfejsu, który pozwoli na wydajną komunikację z aplikacjami mobilnymi, przypadki użycia ograniczę natomiast do podzbioru cen chemikaliów i związanych z nimi serii, zawierających historyczne i przewidywane ceny. W kilku przykładach zobrazuję, jak można ograniczyć czas żądania i wielkość odpowiedzi a także zapewnić dodatkową funkcjonalność w postaci sortowania, niewielkim kosztem. 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 zazębia do zagłębiania się w konstrukcję API w świetle przedstawionych przykładów.

3.1. Implementacja na wybranej platformie

Każdy z nas 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, a który pozwala bardzo szybko i łatwo skonstruować serwer API. By umożliwić korzystanie z API, opublikowałem je w chmurze *Amazona* po adresem mgr.arturrybak.com, w kontenerze *Dockerowym*, dzięki czemu bardzo łatwo jest *deployować* nowe wersje, a dostępność aplikacji szacuje się na 99.5%. Do przechowywania danych użyłem bazy MSSQL Server 2008.

Testowe aplikacje klienckie zostały wykonane na dwóch platformach: **iOS** i **Android**. Przetestowane zaś zostały na urządzeniach **iPhone 5s** oraz **HTC One**. Wszystkie projekty dostępne są w repozytorium pod adresem <http://github.com/wedkarz/mgr> (należy uprzednio uzyskać dostęp) a także na załączonej płytce. Nie należy tu oczekwać fajerwerków graficznych, gdyż aplikacje zostały stworzone jedynie celem zmierzenia wydajności komunikacji klient-serwer.

Modyfikację istniejącego modelu rozpoczęłem bardzo głęboko, 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 dużo informacji i metadanych serii chemicznej (zmian cen chemikalium w czasie). Otóż by otrzymać cenę oraz wszystkie metadane jendego chemikalium w jednym zapytaniu SQL konieczne było wywołanie *SELECT'a* w którym 13 razy musielibyśmy łączyć po trzy tabele (ze względu na rozmiar, skrypt został załączony w dodatku ??). Jest to co najmniej nieefektywne i skutkuje długimi czasami oczekiwania i zajętości bazy. Z tego zdecydowałem się utworzyć widok na bazie, grupujący tabele, który jak się okazało nie rozwiązywał problemu wydajności, przynajmniej, jeśli nie został założony dodatkowy indeks. Drugim rozwiązaniem, które stało się również rozwiązaniem ostatecznym, było zduplikowanie danych.

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
średnia	8627.3	8328	4569.7

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

Redundancja i denormalizacja bazy była w tym przypadku uzasadniona, jako, że spodziewamy się częstych zapytań o serie chemiczne, które będą zawierały wszystkie metadane. Stąd powstała nowa tabela *TimeseriesDetails*. Dla porównania zmierzyłem czasy oczekiwania. Wyniki prezentuje tabela 3.1. Na jej przykładzie zysk z utworzenia nowej tabeli jest ewidentny. 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 pod adresem http://mgr.arturrybak.com/test_select lub w wersji wie-loiteracyjnej 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 musimy pamiętać, że swoją rolę będzie ono spełniało tylko wtedy, gdy wydajność nie będzie problemem. Często napotkamy na ścianę, którą jest model danych, baza, której nie będziemy mogli modyfikować. Warto pomyśleć, czy dla tej specyficznej części systemu, która będzie objęta naszym API nie powinniśmy 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 wspomniałem, powinniśmy wydzielić w naszym API zasoby. Wokół tych zasobów będziemy konstruować żądania HTTP ze adekwatnymi metodami (GET, POST, PUT DELETE). W rozdziale 2.2.1 dużo pisałem o tym, dla czego powinny to być rzeczowniki. Decyzja o tym, czy używać liczby pojedynczej czy mnogiej zależy już wyłącznie od preferencji. Niewielką zaletą formy mnogiej będzie uwolnienie deweloperów od tworzenia wielu akcji dla różnych form, jak np.

person/people, goose/geese

Moje wyjściowe API będzie tutaj wspaniałe w swej prostocie, jako, 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*, tzn. 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 wyjaśnienia 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 nie ma sensu. Jak bowiem przełożyć pożądaną akcję *search*, skoro dotyczy ona wielu zasobów? Nie należy się tym przejmować w takich wypadkach zawsze róby to, co będzie najwygodniejsze z punktu widzenia konsumenta API.

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 chemicznych kodów chemicznych. Takie rozwiązanie nie jest wydajne ani w pełni satysfakcyjne ze względu na ograniczoną ilość danych przesyłanych w każdej odpowiedzi. Przykładowo zapytanie

o ceny Benzenu (zobrazowane na rysunku 3.1) skutkuje około **9 sekundowym** (!) czasem oczekiwania i ponad 3 MB odpowiedzią. To wszystko z użyciem środowiska lokalnego. Użycie 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 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 zauważyc, że pole *values* zawiera zserializowanego JSON'a z danymi o wartościach cen w czasie. Jest to długi *string*, który powoduje przesłanie 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 możemy zaoszczędzić na transferze danych do aplikacji mobilnej, gdy ograniczamy się do proszenia jedynie 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 przesłanie do klienta. W moim przypadku, gdy będziemy prosili tylko o tytuły serii zredukowaliśmy czas z 7,8 sek. do 0,1 sek. a wielkość odpowiedzi z 3,21 MB do 5,23 KB. To znaczne usprawnienie, wystarczy porównać w przeglądarce wywołanie <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 in-

terfejs, który pozwoli im posortować czy filtrować zwroty. Wprawdzie nie będzie to miało znaczenia jeśli chodzi o oszczędność transferu czy czasy zwrotów, jednak z pewnością będzie pomocnym narzędziem w rękach deweloperów. Pozwoli 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
1	200	GET	win:8080	/prices/BZE?fields=title,type,status,region,concept	19:52:05	107 ms	11.03 KB	Complete
2	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: Zaptryania 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 każdy w swojej aplikacji będzie mógł zaaplikować. Ufam także, że każde API, które powstanie po lekturze tej pracy będzie przemyślane i zamiast powstawać w sposób organiczny zostanie podparte 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 też praktycznym narzędziem do wykorzystania każdego dnia.

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

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	56
---	----

Spis rysunków

1.1. Strona artykułu w serwisie <i>na:temat</i> z przyciskami społecznościowymi	12
Interpretacja: 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>	
Źródło: na:temat [online], [dostęp: 16 czerwca 2015] dostępny w internecie: http://natemat.pl/145775	12
1.2. Sekcja komentarzy do artykułu w serwisie <i>na:temat</i>	13
Interpretacja: Niektóre z serwisów zrzucają całe funkcjonalności i interakcje z użytkownikami na baki API społecznościowych. Tużaj: wykorzystanie formularza komentarzy zintegrowanego z <i>Facebookiem</i>	
Źródło: na:temat [online], [dostęp: 16 czerwca 2015] dostępny w internecie: http://natemat.pl/145775	13

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 investują 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>

..... 15

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 17

- 1.5. Odpowiedź z API Twilio, która potwierdza wysłanie testowego
SMSa

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

Źródło: własne 18

- 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 19

- 1.7. Element składowe odpowiedzi na żądanie HTTP

Interpretacja: Ustrukturyzowana odpowiedź na żądanie HTTP
zawiera kod HTTP informujący o statusie, nagłówki oraz w nie-
których przypadkach ciało.

Źródło: [1], wersja IV, s. 28 21

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 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 23

1.9. Wynik działania programu z listingu 1.

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

Źródło: własne 24

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 26

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 27

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 29

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 33

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 35

1.15. Kontroler aplikacji webowej

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

Źródło: własne 37

1.16. Kontroler aplikacji mobilnej

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

Źródło: własne 37

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 41

- 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 niezwiązanne z konkretnym ekranem a ze strukturą API. Na niebiesko żądanie autentykacyjne. Pozostałe służą wyświetleniu strony startowej.

Źródło: własne 42

- 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 42

- 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 49

- 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 51

- 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 52

- 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 60

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 61

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 61

3.4. Zaptyania 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 62

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