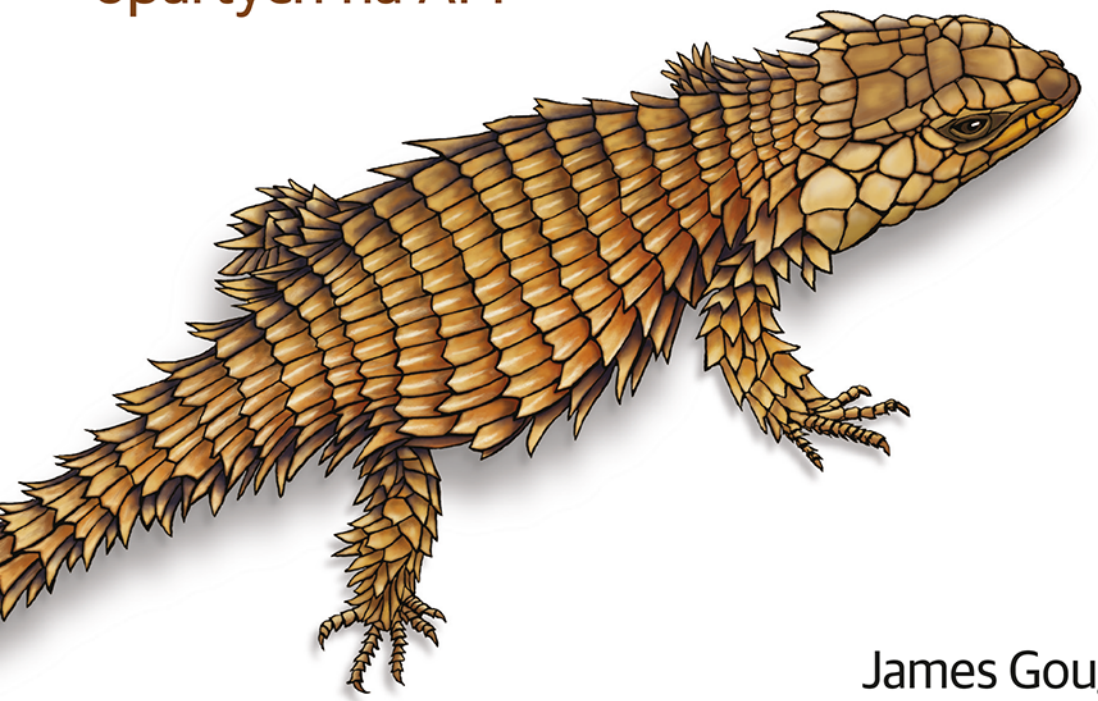
O'REILLY®

# Architektura API

## Projektowanie, używanie i rozwijanie systemów opartych na API



James Gough
Daniel Bryant
Matthew Auburn

Helion

Tytuł oryginału: Mastering API Architecture: Design, Operate, and Evolve API-Based Systems

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-0720-1



Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*https://helion.pl/user/opinie/arcapi*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

# Spis treści

ROZDZIAŁ 1

# Opracowywanie, budowanie i określanie API

Podczas opracowywania i budowania API do dyspozycji masz wiele możliwości. Wprawdzie usługę można zbudować wręcz niewiarygodnie szybko dzięki wykorzystaniu nowoczesnych technologii i frameworków, ale opracowanie trwałego podejścia wymaga dokładnego przemyślenia i przeanalizowania projektu. W rozdziale zajmiemy się analizą stylu REST i zdalnego wywoływania procedur ( ) do modelowania relacji producenta i konsumenta w przykładowym rozwiązaniu omawianym w książce.

Przekonasz się, że standardy mogą pomóc w skróceniu czasu podejmowania decyzji projektowych oraz uniknięciu potencjalnych problemów związanych z zapewnieniem zgodności. Ponadto zapoznasz się ze specyfikacją **OpenAPI**, praktycznym wykorzystaniem zespołów oraz ważną rolą wersjonowania.

Interakcje bazujące na zdalnym wywoływaniu procedur są określane z użyciem schematu. Aby porównać je i zestawić z podejściem typu REST, przeanalizujemy styl **gRPC**. Zapoznasz się z możliwością wykorzystania obu tych podejść, zarówno REST, jak i **gRPC**, w tej samej usłudze i zobaczysz, czy takie rozwiązanie będzie odpowiednie w Twoim projekcie.

## Przykład opracowywania API uczestnika

We wprowadzeniu zdecydowaliśmy o migracji dotychczasowego systemu konferencyjnego do architektury w większym stopniu bazującej na API. Pierwszym krokiem podczas tej modernizacji będzie utworzenie nowej usługi **uczestnika** (ang. *attendee*), której zadanie polega na udostępnieniu odpowiedniego API uczestnika. Ponadto zaprezentowaliśmy dość zawężoną definicję API. W celu efektywnego projektowania konieczne jest rozważenie znacznie szerszego rozwiązania w zakresie wymiany danych między producentem i konsumentem, a co ważniejsze, zdefiniowanie producenta i konsumenta. Producent będzie własnością zespołu uczestników. Ten zespół definiuje dwie kluczowe relacje:

- Zespół uczestników jest właścicielem producenta, a zespół starego systemu konferencyjnego będzie właścicielem konsumenta. Między tymi dwoma zespołami zachodzi silny związek i wszelkie zmiany w strukturze mogą być łatwo koordynowane. Możliwe jest zapewnienie dużej spójności między usługami producenta i konsumenta.
- Zespół uczestnika jest właścicielem producenta, a zespół zewnętrznego systemu **CFP** jest właścicielem konsumenta. Istnieje pewna relacja między zespołami, przy czym wszelkie zmiany muszą

być koordynowane, aby nie uszkodziły integracji. Luźne powiązanie jest wymagane i trzeba zachować ostrożność podczas zarządzania poważnymi zmianami.

W rozdziale porównamy i zestawimy podejścia w zakresie projektowania i budowy API uczestnika.

# Wprowadzenie do stylu REST

Styl REST (ang. *representational state transfer*) przedstawia zbiór architekturalnych ograniczeń, w większości stosowanych za pomocą protokołu transportowego HTTP. Przygotowane przez Roya Thomasa Fieldinga opracowanie *Architectural Styles and the Design of Network-based Software Architectures* (*https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*) zawiera pełną definicję stylu REST. Aby API zostało z perspektywy praktycznej uznane za zgodne ze stylem **RESTful**, musi spełniać następujące wymagania:

- Jest modelowana interakcja między producentem i konsumentem — w tej relacji producent modeluje zasoby, z którymi konsument może prowadzić interakcje.
- Wykonywane przez producenta żądania do konsumenta są bezstanowe. To oznacza, że producent nie buforuje szczegółów wcześniejszych żądań. Aby zbudować łańcuch żądań do danego zasobu, konsument musi przekazywać producentowi wszystkie informacje w celu ich przetworzenia.
- Żądania są możliwe do buforowania, co oznacza, że producent może zapewnić konsumentowi podpowiedzi tam, gdzie będzie to odpowiednie. W przypadku protokołu HTTP informacje są często przekazywane w nagłówku.
- Konsumentowi jest udostępniany jednolity interfejs. Wkrótce dowiesz się nieco więcej na temat używania metod HTTP, zasobów i innych wzorców.
- Mamy do czynienia z systemem bazującym na warstwach, który ukrywa złożoność systemów kryjących się za interfejsem REST. Na przykład konsument nie będzie wiedział i nawet nie będzie zainteresowany tym, czy prowadzi komunikację z bazą danych, czy też z innymi usługami.

## Oparte na przykładzie wprowadzenie do technologii REST i HTTP

Przedstawimy teraz prosty przykład użycia stylu REST poprzez protokół HTTP. W przedstawionej sytuacji mamy do czynienia z żądaniem `GET`, przy czym słowo `GET` oznacza tu metodę HTTP. Nazwa metody opisuje działanie podejmowane na określonym zasobie, tutaj będzie to zasób *uczestnika* konferencji.

Nagłówek `Accept` został przekazany w celu zdefiniowania typu treści, którą konsument chciałby otrzymać. Styl REST definiuje sposób reprezentacji danych w żądaniu i pozwala na zdefiniowanie *metadanych* w nagłówku.

W przykładach zamieszczonych w rozdziale powyżej znaków `---` znajduje się żądanie, natomiast poniżej zawsze znajduje się udzielona na nie odpowiedź.

```
GET http://mastering-api.com/attendees
Accept: application/json
---
```

```
200 OK
Content-Type: application/json
{
    "displayName": "Jim",
    "id": 1
}
```

Udzielona odpowiedź zawiera kod stanu i komunikat pochodzący z serwera, co pozwala konsumentowi na przeanalizowanie wyniku operacji przeprowadzonej przez zasób po stronie serwera. Kod stanu omawianego żądania wyniósł `200 OK`, co wskazuje na zakończone sukcesem przetworzenie żądania przez producenta. W treści odpowiedzi znajdują się dane `JSON` zawierające uczestników konferencji. Wprawdzie wiele typów treści jest poprawnych w stylu REST, ale trzeba koniecznie wziąć pod uwagę to, czy dany typ będzie mógł być przetworzony przez konsumenta. Na przykład można zwrócić treść typu `application/pdf`, przy czym dane w tym formacie nie będą mogły być łatwo użyte przez inny system. Podejścia w zakresie modelowania typów treści zostaną omówione w dalszej części rozdziału, choć skoncentrujemy się przede wszystkim na formacie `JSON`.

Styl REST jest względnie prosty do implementacji, ponieważ relacja między klientem i serwerem jest bezstanowa. Oznacza to, że serwer nie przechowuje trwale żadnych informacji dotyczących stanu klienta. W trakcie wykonywania kolejnych żądań klient musi ponownie przekazać kontekst serwerowi. Na przykład żądanie do punktu końcowego *http://mastering-api.com/attendees/1* spowoduje pobranie dokładnych informacji na temat wskazanego uczestnika konferencji.

## Model dojrzałości Richardsona

Występując na konferencji QCon (*https://www.crummy.com/writing/speaking/2008-QCon/act3.html*) w 2008 roku, Leonard Richardson podzielił się swoimi doświadczeniami związanymi z analizą wielu API REST. Richardson odkrył używane przez zespoły poziomy adopcji, które były stosowane podczas tworzenia API z perspektywy stylu REST. Martin Fowler na swoim blogu (*https://martinfowler.com/articles/richardsonMaturityModel.html*) również zdecydował się na omówienie heurystyki dojrzałości Richardsona. W tabeli 1.1 znajdziesz omówienie różnych poziomów heurystyki dojrzałości Richardsona oraz ich zastosowanie w API REST.

*Tabela 1.1. Heurystyka dojrzałości Richardsona*

| | |
|---|---|
| Poziom 0 — HTTP/RPC | Wskazuje na budowanie API za pomocą HTTP i zastosowanie notacji w postaci pojedynczego adresu URI. Z wykorzystaniem wcześniejszego przykładu */attendees* i bez użycia metody do wskazania zamiaru nastąpiłoby otworzenie punktu końcowego w celu wymiany danych. W zasadzie przedstawia to implementację RPC działającą poprzez protokół REST. |
| Poziom 1 — zasoby | Wskazuje na użycie zasobów i zaczyna wprowadzać ideę modelowania zasobów w kontekście adresu URI. W naszym przykładzie, jeśli zostałoby dodane wywołanie `GET /attendees/1` zwracające informacje na temat konkretnego uczestnika, to API wreszcie zaczęłoby przypominać API pierwszego poziomu. Martin Fowler posłużył się analogią do klasycznego świata programowania zorientowanego obiektowo i wprowadzenia w nim identyfikatora. |
| Poziom 2 — metody | Rozpoczyna się wprowadzenie do właściwego modelowania adresów URI wielu zasobów, do których dostęp uzyskują różne metody HTTP żądania, na podstawie efektu tych zasobów wywieranego na serwerze. API na poziomie 2 może zagwarantować, że metody `GET` nie będą miały wpływu na stan serwera. Możliwe jest przeprowadzanie wielu operacji na tym samym adresie URI zasobu. W naszym przykładzie dodanie wywołań `DELETE /attendees/1` i `PUT /attendees/1` spowodowałoby, że API miałoby notację poziomu drugiego — zgodnego API. |

*Tabela 1.1. Heurystyka dojrzałości Richardsona – ciąg dalszy*

| | |
|---|---|
| Poziom 3 — kontrolki hipermediów | To typowy przykład projektu REST; obejmuje API, po którym można się poruszać. Tego rodzaju rozwiązanie jest możliwe dzięki użyciu metod **HATEOAS** (ang. *hypermedia as the engine of application state*) (*https://restcookbook.com/Basics/hateoas/*). W naszym przykładzie wykonanie wywołania `GET /attendees/1` spowoduje, że odpowiedź będzie zawierała akcje możliwe do przeprowadzania na obiekcie zwróconym przez serwer. Będzie to obejmowało możliwość uaktualnienia uczestnika konferencji bądź jego usunięcia oraz to, co klient musi wywołać, aby taka operacja została przeprowadzona. W ujęciu praktycznym styl poziomu trzeciego rzadko jest używany w nowoczesnych usługach HTTP stylu RESTful. Wprawdzie możliwość nawigacji jest zaletą w elastycznych systemach interfejsu użytkownika, ale nie pasuje do wywołań API między usługami. Wykorzystanie metod HATEOAS będzie pewnym doświadczeniem, często ograniczonym przez posiadanie pełnej specyfikacji z góry określającej interakcje dozwolone podczas tworzenia rozwiązania programistycznego, które ma współpracować z producentem. |

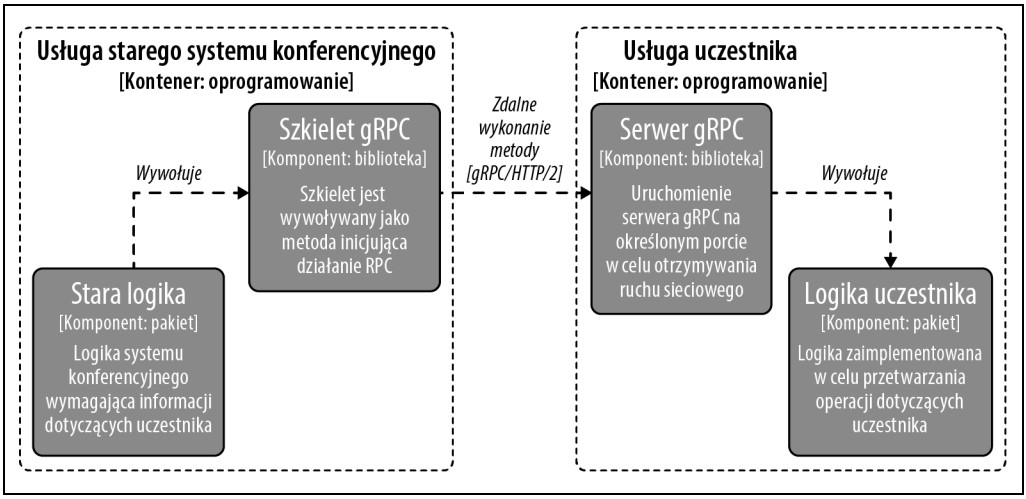
Podczas projektowania API trzeba wziąć pod uwagę różne poziomy dojrzałości Richardsona. Przejście w kierunku poziomu drugiego pozwoli na zaprojektowanie konsumentowi zrozumiałego modelu zasobów razem z właściwymi działaniami, które będą mogły być podejmowane dla tego modelu. To pozwala z kolei na zmniejszenie poziomu powiązania i ukrycie pełnych szczegółów usługi. W dalszej części rozdziału zobaczysz, jak taka abstrakcja ma zastosowanie dla wersjonowania.

Jeżeli konsumentem jest zespół CFP, wówczas modelowanie wymiany z niskim poziomem powiązania i projektowanie modelu RESTful będzie dobrym punktem wyjścia. Jeżeli konsumentem jest zespół starego systemu konferencyjnego, to wciąż można zdecydować się na użycie API RESTful, choć istnieje wówczas jeszcze inne rozwiązanie w postaci RPC. Aby rozważyć tego rodzaju tradycyjny sposób modelowania wschód-zachód, konieczne jest poznanie RPC.

# Wprowadzenie do API zdalnego wywoływania procedur

**Zdalne wywoływanie procedur** (ang. *remote procedure calls*, **RPC**) obejmuje wywoływanie metody w jednym procesie, ale wykonanie jej w innym. Podczas gdy styl REST pozwala na opracowanie projektu domeny i dostarcza konsumentowi abstrakcję technologii kryjącej się za rozwiązaniem, RPC umożliwia udostępnienie metody z jednego procesu i wywołanie jej bezpośrednio z innego procesu.

gRPC to nowoczesna, dostępna jako open source i charakteryzująca się dużą wydajnością działania implementacja RPC. Projektem gRPC zarządza fundacja Linux Foundation i stanowi on faktyczny standard zdalnego wywoływania procedur na większości platform. Na rysunku 1.1 przedstawiliśmy wywołanie RPC w standardzie gRPC. W tym przykładzie usługa starego systemu konferencyjnego wywołuje zdalną metodę w usłudze uczestnika (Attendee). Usługa gRPC Attendee zostaje uruchomiona i udostępnia serwer gRPC na określonym porcie, co pozwala na zdalne wywoływanie metod. Po stronie klienta (usługa starego systemu konferencyjnego) szkielet zostaje użyty do abstrakcji złożoności wykonania zdalnego wywołania na bibliotekę. gRPC wymaga schematu w celu pełnej obsługi interakcji między producentem i konsumentem.




*Rysunek 1.1. Przykładowy diagram C4 komponentu używającego standardu gRPC*

Podstawową różnicą między REST i RPC jest stan. Styl REST jest z definicji bezstanowy — natomiast w przypadku RPC stan zależy od implementacji. Bazujące na RPC integracje mogą w niektórych sytuacjach również tworzyć stan jako część procesu wymiany danych. Takie utworzenie stanu będzie wygodne, gdyż zapewni wysoką wydajność, zaś potencjalnym kosztem tego podejścia będą mniejsza niezawodność i złożoność routingu. W przypadku standardu RPC model ma tendencję do przekazywania na poziomie metody dokładnej funkcjonalności, która jest wymagana od usługi dodatkowej. Ten stan opcjonalny może prowadzić do wymiany, która potencjalnie spowoduje większe powiązanie między producentem i konsumentem. Powiązanie nie zawsze jest czymś złym, zwłaszcza w przypadku usług typu wschód-zachód, w których to wydajność działania ma ważne znaczenie.

# Krótkie wprowadzenie do GraphQL

Przed przejściem do dokładnego omówienia stylów REST i RPC niedopatrzeniem z naszej strony byłoby pominięcie GraphQL i wyjaśnienie jego miejsca w świecie API. Wprawdzie RPC oferuje dostęp do serii poszczególnych funkcji dostarczanych przez producenta, ale zwykle nie rozszerza modelu bądź abstrakcji dla konsumenta. Z kolei REST rozszerza model zasobów dla pojedynczego API dostarczanego przez producenta. Dzięki bramie API istnieje możliwość dostarczenia wielu API na podstawie bazowego adresu URL. Konsument musi również znać strukturę wszystkich usług wykorzystywanych w zapytaniu. Takie podejście okaże się marnotrawstwem, jeśli konsument będzie zainteresowany jedynie podzbiorem danych dostarczonych w udzielonej odpowiedzi. Urządzenia mobilne są ograniczone przez mniejsze ekrany i dostępność sieci, więc w takim środowisku GraphQL sprawdza się doskonale.

GraphQL wprowadza dla istniejących usług, magazynów danych i API warstwę technologiczną, zapewniającą język zapytań pozwalający na wykonywanie zapytań do wielu źródeł. Język zapytania pozwala również klientowi na zapytanie o wymagane pola, w tym również pola obejmujące

wiele API. GraphQL wykorzystuje język schematu GraphQL do wskazania typów poszczególnych API oraz sposobów ich łączenia. Jedną z najważniejszych zalet wprowadzenia schematu GraphQL do systemu jest możliwość dostarczenia pojedynczej wersji dla wszystkich API, co eliminuje potrzebę potencjalnie skomplikowanego zarządzania wersją po stronie klienta.

GraphQL sprawdza się świetnie, gdy klient wymaga ujednoliconego API dostępu do szerokiej gamy powiązanych ze sobą usług. Schemat zapewnia połączenie i rozbudowę modelu domeny, a to pozwala klientowi na dokładne wskazanie tego, co jest wymagane po stronie klienta. Takie podejście sprawdza się wyjątkowo dobrze podczas modelowania interfejsu użytkownika, w systemach raportujących bądź systemach typu hurtownie danych. W systemach przechowujących ogromne ilości danych w różnych podsystemach GraphQL może zapewnić idealne rozwiązanie pozwalające na abstrakcję złożoności systemu wewnętrznego.

Istnieje możliwość użycia GraphQL w starej wersji systemu i wykorzystania GraphQL jako fasady mającej na celu ukrycie złożoności. Odbywa się to przez dostarczenie GraphQL na podstawie warstwy doskonale opracowanych API, przez co fasada staje się prostsza do zaimplementowania i utrzymania. GraphQL można potraktować jako technologię uzupełniającą, której użycie warto rozważyć podczas opracowywania i budowania API. GraphQL można zastosować również jako pełne podejście do budowy całego ekosystemu API.

GraphQL sprawdza się doskonale w określonych scenariuszach. Gorąco zachęcamy do zapoznania się z pozycjami *Learning GraphQL* (O'Reilly) i *GraphQL in Action* (O'Reilly), aby znacznie dokładniej poznać tę technologię.

# Struktura i standardy API REST

Styl REST ma kilka bardzo prostych reguł, przy czym większość kwestii związanych z implementacją i projektem jest pozostawionych w gestii programisty. Na przykład jaki jest najlepszy sposób na poradzenie sobie z błędami? Jak powinno być zaimplementowane stronicowanie? Jak uniknąć przypadkowego zbudowania API, w którym zgodność często będzie niezachowywana? Na tym etapie znacznie użyteczniejsze będzie posiadanie dużo bardziej praktycznej definicji API, w celu zapewnienia jednolitości i spełnienia oczekiwań między różnymi implementacjami. Tutaj mogą pomóc standardy i wytyczne, przy czym mamy mnóstwo źródeł oferujących tego rodzaju informacje.

Na potrzeby związane z analizowaniem projektu posłużymy się *wytycznymi API REST Microsoftu* (*https://github.com/microsoft/api-guidelines*), które składają się z serii wewnętrznych wskazówek udostępnionych jako open source. Te wytyczne używają dokumentu RFC-2119 definiującego terminologię dla standardów takich jak MUST, SHOULD, SHOULD NOT, MUST NOT itd., co pozwala programiście na określenie, które wymagania są opcjonalne lub obowiązkowe.

Wraz z ewolucją standardów API REST otwarta lista standardów API jest dostępna na przygotowanej dla tej książki stronie w serwisie GitHub (*https://github.com/masteringapi/rest-api-standards*). Jeżeli znasz otwarty standard, który według Ciebie może być użyteczny także dla innych czytelników, rozważ jego zgłoszenie za pomocą tzw. żądania aktualizacji (ang. *pull lrequest*).

Przejdźmy teraz do projektu API *Attendee* utworzonego z uwzględnieniem wytycznych Microsoft API REST i dodajmy punkt końcowy pozwalający na utworzenie nowego *uczestnika*. Jeżeli masz doświadczenie w pracy ze stylem REST, na myśl natychmiast przyjdzie Ci użycie metody `POST`:

```
POST http://mastering-api.com/attendees
{
    "displayName": "Jim",
    "givenName": "James",
    "surname": "Gough",
    "email": "jim@mastering-api.com"
}
---
201 CREATED
Location: http://mastering-api.com/attendees/1
```

Nagłówek *`Location`* ujawnia położenie nowego zasobu utworzonego na serwerze. W tym API modelujemy unikatowy identyfikator dla uczestnika. Wprawdzie istnieje możliwość użycia pola *adresu e-mail* jako unikatowego identyfikatora, ale zamieszczone w sekcji 7.9 *wytyczne Microsoft API REST* zalecają, aby dane osobowe *nie* były wykorzystywane jako część adresu URL.

Powodem kryjącym się za usunięciem danych wrażliwych z adresu URL jest to, że ścieżki dostępu i parametry zapytania mogą być przypadkowo buforowane w sieci — np. w dziennikach zdarzeń serwerów itd.

Innym trudnym aspektem API może być nazewnictwo. Jak wyjaśnimy w dalszej części rozdziału, czasami bardzo prosta operacja typu zmiana nazwy może doprowadzić do uszkodzenia zgodności. W *zaleceniach Microsoft API REST* została wymieniona krótka lista nazw standardowych, których należy używać. Jednak zespoły powinny rozbudowywać tę listę, aby otrzymać słownik danych domeny, który będzie uzupełniał standard. W wielu organizacjach niezwykle pomocne okazuje się proaktywne analizowanie wymagań związanych z projektem danych i czasami nawet zarządzanie tym projektem. Organizacje zapewniające spójność we wszystkich interfejsach API oferowanych przez firmę zapewniają jednocześnie jednolitość, która z kolei ułatwia klientom zrozumienie i połączenie odpowiedzi. W niektórych domenach może istnieć powszechnie znana terminologia — używaj jej!

## Kolekcje i stronicowanie

Rozsądne wydaje się modelowanie żądania `GET /attendees` jako odpowiedzi zawierającej niezmodyfikowaną tablicę. W kolejnym fragmencie kodu pokazaliśmy, jak mogłaby wyglądać przykładowa odpowiedź udzielona na takie żądanie:

```
GET http://mastering-api.com/attendees
---
200 OK
[
    {
        "displayName": "Jim",
        "givenName": "James",
        "surname": "Gough",
        "email": "jim@mastering-api.com",
        "id": 1,
    },
    ...
]
```

Rozważmy alternatywny dla żądania `GET /attendees` model, który będzie zagnieżdżał w obiekcie tablicę uczestników. Może wydawać się dziwne zwracanie odpowiedzi w postaci tablicy umieszczonej w obiekcie, ale powodem takiego rozwiązania jest to, że pozwala ono na modelowanie większych kolekcji i zastosowanie stronicowania. Stronicowanie oznacza zwrot wyniku częściowego i jednoczesne dostarczenie konsumentowi instrukcji na temat tego, jak można żądać następnego zbioru wyników. To pozwoli na czerpanie korzyści w późniejszym czasie. Jeżeli stronicowanie zostałoby dodane później i odbyłaby się konwersja z postaci tablicy na obiekt w celu dodania `@nextLink` (zgodnie z zaleceniami standardów), wówczas spowodowałoby to niezapewnienie zgodności.

```
GET http://mastering-api.com/attendees
---
200 OK
{
    "value": [
        {
            "displayName": "Jim",
            "givenName": "James",
            "surname": "Gough",
            "email": "jim@mastering-api.com",
            "id": 1,
        }
    ],
    "@nextLink": "{opaqueUrl}"
}
```

## Filtrowanie kolekcji

Nasza konferencja będzie wyglądała skromnie, jeśli będzie w niej tylko jeden uczestnik. Jednak wraz ze wzrostem wielkości kolekcji może pojawić się potrzeba nie tylko dodania stronicowania, ale również zapewnienia jej obsługi. Standard filtrowania zapewnia język wyrażeń w REST, pozwalający na standaryzowanie sposobu, w jaki powinny działać zapytania filtrowania w oparciu o protokół *Open Data Protocol* (**OData**). Na przykład w celu odszukania wszystkich uczestników o wyświetlanym imieniu `Jim` można posłużyć się następującym zapytaniem:

```
GET http://mastering-api.com/attendees?$filter=displayName eq 'Jim'
```

Nie trzeba już od samego początku korzystać ze wszystkich oferowanych funkcjonalności filtrowania i wyszukiwania. Jednak opracowywanie API zgodnie ze standardami pozwoli programiście na obsługę ewolucji architektury API bez uszkadzania zgodności klientom. Filtrowanie i wykonywanie zapytań to funkcjonalność, w której GraphQL sprawdza się wyjątkowo dobrze, zwłaszcza jeśli operacje przeprowadzane między usługami mają znaczenie.

## Obsługa błędów

Ważną kwestią podczas rozszerzania API dla konsumentów jest zdefiniowanie tego, co powinno się zdarzyć w przypadku występowania różnych błędów. Najlepiej będzie wcześniej zdefiniować standardy dotyczące błędów (*https://github.com/microsoft/api-guidelines/blob/vNext/Guidelines.md#7102-error-condition-responses*) i udostępnić je producentom w celu zapewnienia zgodności.

Bardzo duże znaczenie ma to, aby błędy dokładnie wskazywały konsumentowi, co poszło źle podczas wykonywania żądania, ponieważ dzięki temu zmniejsza się ilość pomocy technicznej, którą trzeba będzie świadczyć dla API.

We wspomnianych wcześniej zaleceniach dotyczących API można przeczytać: „*W przypadku operacji, która nie zakończy się sukcesem, programista POWINIEN mieć możliwość utworzenia pojedynczego fragmentu kodu zapewniającego spójną obsługę błędów*". Klientowi musi być przekazany odpowiedni kod stanu, ponieważ często się zdarza, że klienci opracowują logikę bazującą właśnie na kodzie stanu przekazanym w odpowiedzi na żądanie. Spotkaliśmy się z przykładami wielu API zwracających komunikaty błędów w treści odpowiedzi typu `2xx`, który jest używany do wskazania sukcesu. Kod stanu typu `3xx` jest przeznaczony do przekierowania aktywnie używanych przez pewne implementacje bibliotek działających po stronie klienta, co pozwala dostawcom na zmianę położenia zasobów zewnętrznych i mimo to zapewnienie do nich dostępu. Z kolei kod stanu typu `4xx` zwykle wskazuje błąd po stronie klienta. W takim przypadku zawartość pola `message` będzie wyjątkowo użyteczna dla programisty bądź użytkownika końcowego. Natomiast kod stanu typu `5xx` zwykle oznacza błąd po stronie serwera; pewne biblioteki działające po stronie klienta będą ponownie wykonywały żądanie po otrzymaniu kodu stanu typu `5xx`. Bardzo ważne jest rozważenie i udokumentowanie tego, co się stanie w usłudze w przypadku wystąpienia nieoczekiwanego niepowodzenia. Na przykład: czy w systemie przetwarzania płatności kod stanu `500` oznacza, że dana płatność została zrealizowana?

Należy się upewnić, że komunikaty błędów przekazywane do zewnętrznego konsumenta nie zawierają danych stosu wywołań oraz innych informacji wrażliwych. Tego rodzaju dane mogą pomóc osobie przeprowadzającej atak w uzyskaniu dostępu do systemu. Struktura błędów w wytycznych Microsoft zawiera koncepcję tzw. **błędu wewnętrznego** (ang. *inner error*), która może okazać się użyteczna podczas zamieszczania znacznie dokładniejszych informacji dotyczących problemu i stosu wywołań. Takie rozwiązanie będzie niezwykle pomocne podczas debugowania, choć informacje wrażliwe muszą być usunięte przed przekazaniem komunikatu błędu do konsumenta zewnętrznego.

To jest zaledwie wierzchołek góry lodowej tematu obejmującego budowanie API REST. Nie ulega wątpliwości, że podczas tworzenia API konieczne będzie podjęcie wielu ważnych decyzji. Jeżeli połączymy chęć dostarczenia intuicyjnego API, które pozostanie spójne oraz pozwoli na ewolucję i zachowanie przy tym wstecznej zgodności, wówczas warto już na wczesnym etapie zaadaptować standard API.

## Wskazówki dotyczące dokumentu typu ADR — wybór standardu API

Aby podjąć decyzje dotyczące standardów API, trzeba rozważyć wiele kwestii, z których najważniejsze zostały wymienione w tabeli 1.2. Opracowano wiele zaleceń, z których można korzystać, m.in. wspomniane w rozdziale zalecenia Microsoftu. Znalezienie tych najbardziej odpowiednich dla stylu tworzonego API będzie decyzją o znaczeniu kluczowym.

*Tabela 1.2. Wskazówki dotyczące standardów API*

| | |
|---|---|
| Decyzja | Który standard API powinien być zaadaptowany? |
| Kwestie do omówienia | Czy w organizacji są już zastosowane jakieś inne standardy? Czy można je rozszerzyć na konsumentów zewnętrznych? |
| | Czy używamy jakiegokolwiek opracowanego przez podmioty zewnętrzne API, które będzie musiało być udostępnione konsumentowi (np. usługa identyfikacji) i które ma już przyjęty standard? |
| | Jaki wpływ na konsumentów będzie miała rezygnacja z przyjęcia standardu? |
| Zalecenia | Wybierz standard API najlepiej pasujący do kultury przyjętej w organizacji i sformatuj API, które być może jest już stosowane w organizacji. |
| | Przygotuj się na ewolucję i dodawanie do standardu różnych poprawek związanych z domeną i branżą, w której działa organizacja. |
| | Zacznij wcześnie, aby zapewnić spójność i uniknąć niedotrzymania zgodności. |
| | Spójrz krytycznie na istniejące API. Czy są one sformatowane w sposób zrozumiały dla konsumentów? Czy konieczne będzie włożenie większego wysiłku w celu zaoferowania dostępu do treści? |

# Określanie API REST za pomocą OpenAPI

Jak wkrótce dostrzeżesz, projekt API ma podstawowe znaczenie dla odniesienia sukcesu przez platformę API. Następną kwestią do omówienia jest udostępnianie API programistom, którzy będą z niego korzystać.

Dostępnych jest dla konsumentów wiele zarówno publicznych, jak i prywatnych rejestrów API. Programista może przeglądać dokumentację i szybko wypróbować API w przeglądarce WWW, aby sprawdzić oferowaną funkcjonalność i sposób działania. We wspomnianych rejestrach API te w stylu REST są umieszczone wyraźnie w przestrzeni klienta. Na sukces API REST wpływ mają nie tylko kwestie techniczne, ale także niskie bariery wejścia zarówno dla klienta, jak i serwera.

Wraz ze wzrostem liczby API szybko pojawiła się potrzeba opracowania mechanizmu pozwalającego na udostępnianie konsumentom *kształtu* i struktury API. Dlatego też liderzy branży API opracowali inicjatywę OpenAPI w celu stworzenia **specyfikacji OpenAPI** (ang. *openapi specification*, **OAS**). Pierwotną implementacją wzorcową **OAS** był **Swagger**, a obecnie większość narzędzi jest zgodna z OpenAPI.

Specyfikacje OpenAPI są bazującymi na `JSON` lub `YAML` reprezentacjami API, które opisują strukturę, domenę wymienianych obiektów oraz wszelkie wymagania API związane z zapewnieniem bezpieczeństwa. Poza strukturą specyfikacja przekazuje także metadane dotyczące API, m.in. wymagania prawne i licencyjne, a także dostarcza dokumentację i przykłady, które są użyteczne dla programistów wykorzystujących dane API. Specyfikacja OpenAPI to ważna koncepcja związana z nowoczesnym API REST, a wiele narzędzi i produktów zostało opracowanych na bazie OAS.

# Praktyczne zastosowanie specyfikacji OpenAPI

Po udostępnieniu OAS potężne możliwości specyfikacji stały się oczywiste. *Projekt OpenAPI.Tools* (*https://openapi.tools/*) dokumentuje pełny zakres dostępnych narzędzi, zarówno typu open source, jak i komercyjnych. W tym podrozdziale przedstawimy wybrane z praktycznych zastosowań narzędzi na podstawie ich sposobu współpracy ze specyfikacją OpenAPI.

Jeżeli zespół CFP jest konsumentem, wówczas udostępnienie OAS pozwala zespołowi na zrozumienie struktury API. Korzystając z wybranych z zamieszczonych w tym podrozdziale przykładów praktycznego zastosowania, można poprawić wrażenia programisty używającego API, a także upewnić się co do poprawności procesu wymiany danych.

## Generowanie kodu

Prawdopodobnie jedną z najbardziej użytecznych funkcjonalności OAS jest umożliwienie generowania kodu po stronie klienta w celu użycia API. Jak już wcześniej wspomniano, w API można zamieścić dokładne informacje dotyczące serwera, bezpieczeństwa oraz oczywiście struktury samego API. Dzięki tym wszystkim informacjom można wygenerować serię modeli i obiektów usługi, które przedstawiają i wywołują API. *Projekt OpenAPI Generator* (*https://openapi-generator.tech/*) zapewnia obsługę szerokiej gamy języków programowania i narzędzi. Na przykład w Javie można zdecydować się na użycie **Spring** lub **JAX-RS**, natomiast w **TypeScripcie** można wybrać połączenie języka TypeScript i ulubionego frameworka. Istnieje również możliwość wygenerowania na podstawie OAS szkieletów implementacji API.

To rodzi ważne pytanie o to, co powinno pojawić się jako pierwsze — specyfikacja czy kod działający po stronie serwera? W następnym rozdziale będziemy analizować tzw. „śledzenie kontraktu", które przedstawia bazujące na działaniu podejście do budowania i testowania API. Trudność w przypadku specyfikacji OpenAPI polega na tym, że zapewnia ona jedynie sam kształt API. Specyfikacja OpenAPI nie modeluje w pełni semantyki (oczekiwanego sposobu działania) API w różnych sytuacjach. Jeżeli zamierzasz przedstawić API użytkownikom zewnętrznym, bardzo ważne okażą się modelowanie i przetestowanie sposobu jego działania, aby w ten sposób uniknąć w przyszłości konieczności wprowadzania drastycznych zmian w sposobie działania API.

API powinno być opracowane z perspektywy konsumenta oraz z uwzględnieniem wymagań do abstrakcji używanej reprezentacji, aby tym samym zmniejszyć poziom powiązania. Duże znaczenie ma możliwość przeprowadzania dowolnej refaktoryzacji komponentów używanych w tle i jednocześnie zachowania zgodności API. W przeciwnym razie abstrakcja API traci swoje znaczenie.

## Weryfikacja OpenAPI

Specyfikacje OpenAPI są użyteczne podczas weryfikowania treści wymienianych danych, aby mieć pewność, że żądanie i odpowiedź na nie udzielona są zgodne ze specyfikacją. W pierwszej chwili może nie wydawać się oczywiste, czy takie podejście będzie użyteczne — skoro kod został wygenerowany, wymiana danych zawsze będzie odbywała się poprawnie, nieprawdaż? Jednym z praktycznych zastosowań weryfikacji OpenAPI jest zapewnienie bezpieczeństwa API i jego infrastrukturze. W wielu organizacjach stosowana jest architektura bazująca na strefach. Tak zwana **strefa zdemilitaryzowana** (ang. *demilitarized zone*, **DMZ**) jest używana do ochrony sieci przed przychodzącym ruchem sieciowym. Użyteczną funkcjonalnością jest sprawdzanie komunikatów w strefie DMZ oraz pozbywanie się ruchu sieciowego, jeśli nie odpowiada on specyfikacji. W rozdziale 6. znajdziesz dokładne omówienie modelu zapewnienia bezpieczeństwa.

Atlassian udostępnia narzędzie typu *open source* o nazwie *swagger-request-validator* (*https://bitbucket.org/atlassian/swagger-request-validator/src/master/*), które potrafi weryfikować treść `JSON REST`. W ramach

projektu są dostarczane adaptery pozwalające na integrację z różnymi frameworkami imitacji i testowania, aby pomóc w zapewnieniu zgodności specyfikacji API. Narzędzie ma komponent `OpenApiInteractionValidator`, używany do utworzenia egzemplarza `ValidationReport` podczas wymiany danych. W kolejnym fragmencie kodu pokazaliśmy przykład utworzenia na podstawie specyfikacji weryfikatora obejmującego wszelkie egzemplarze `basePathOverrides` — może to być niezbędne w przypadku wdrażania API za infrastrukturą, która zmienia ścieżkę dostępu. Raport weryfikacji zostaje wygenerowany na podstawie analizy żądania i udzielonej na nie odpowiedzi w chwili, gdy przeprowadzana jest weryfikacja:

```
// Z wykorzystaniem położenia specyfikacji następuje utworzenie weryfikatora .
// Nadpisanie bazowej ścieżki dostępu okazuje się użyteczne, gdy weryfikator
// jest używany za bramą lub proxy.
final OpenApiInteractionValidator validator = OpenApiInteractionValidator
        .createForSpecificationUrl(specUrl)
        .withBasePathOverride(basePathOverride)
        .build;

// Obiekty żądania i odpowiedzi na nie mogą być skonwertowane za pomocą budowniczego.
final ValidationReport report = validator.validate(request, response);

if (report.hasErrors()) {
    // Przechwycenie lub przetworzenie informacji o błędzie.
}
```

## Przykłady i imitacje

Specyfikacja OAS może dostarczać przykładowych odpowiedzi dla ścieżek dostępu w specyfikacji. Jak już wcześniej wspomnieliśmy, przykłady są użyteczne dla dokumentacji i pomagają programistom w zrozumieniu oczekiwanego sposobu działania API. Część produktów zaczęła używać przykładów w celu umożliwienia użytkownikom wypróbowania API, a odpowiedzi na te żądania pochodzą z usługi imitacji. Tego rodzaju rozwiązanie może być naprawdę użyteczne podczas oferowania funkcjonalności takiej jak portal programisty, ponieważ pozwala programistom na przeglądanie dokumentacji i wywoływanie API. Kolejną użyteczną funkcją imitacji i przykładów jest możliwość współdzielenia pomysłów między producentem i konsumentem przed przystąpieniem do budowy usługi. Możliwość „wypróbowania" API często ma większą wartość niż próba oceny, czy specyfikacja będzie spełniała Twoje wymagania.

Przykłady mogą potencjalnie wprowadzić interesujący problem polegający na tym, że ta część specyfikacji jest w zasadzie ciągiem tekstowym (w celu modelowania danych `XML`/`JSON` itd.). Narzędzie `openapi-examples-validator` (*https://github.com/codekie/openapi-examples-validator*) sprawdza, czy przykład odpowiada OAS dla właściwego komponentu żądania/odpowiedzi w API.

## Wykrywanie zmian

Specyfikacje OpenAPI mogą być pomocne także podczas wykrywania zmian w API. To okazuje się wręcz niewiarygodnie użyteczne jako część potoku `DevOps`. Wykrywanie zmian pod kątem zachowania wstecznej zgodności jest bardzo ważne, ale najpierw trzeba znacznie dokładniej poznać temat wersjonowania API.

# Wersjonowanie API

Przedstawiliśmy już korzyści, m.in. szybkość integracji, wynikające z udostępnienia specyfikacji OAS konsumentowi. Rozważ sytuację, w której pracę z API rozpocznie wielu konsumentów. Co się stanie po wprowadzeniu zmiany w API bądź gdy jeden z konsumentów będzie żądał dodania nowych funkcjonalności do API?

Warto wykonać krok wstecz i zastanowić się nad sytuacją, w której kod biblioteki zostałby wbudowany w aplikację podczas jej kompilacji. Wówczas każda zmiana w bibliotece wymagałaby utworzenia nowej, a dopóki kod nie będzie ponownie skompilowany i przetestowany, nie będzie miał wpływu na aplikację w środowisku produkcyjnym. Skoro API to uruchomiona usługa, mamy kilka możliwości uaktualnienia, które stają się dostępne natychmiast po wykonaniu żądania zmian.

*Wydanie nowej wersji i wdrożenie jej w nowym położeniu.*

Starsze aplikacje nadal będą działać ze starszą wersją API. Z perspektywy konsumenta jest to dobre rozwiązanie, ponieważ konsument przeprowadzi uaktualnienie do nowego położenia i API tylko wtedy, gdy będzie chciał uzyskać dostęp do nowej funkcjonalności. Jednak właściciel API musi zapewnić obsługę techniczną i zarządzać wieloma wersjami API, uwzględniając je podczas usuwania błędów i wprowadzania niezbędnych poprawek.

*Wydanie nowej wersji API, która zapewni wsteczną zgodność z poprzednią wersją API.*

Takie rozwiązanie pozwala na wprowadzanie stopniowych zmian bez wpływania na istniejących użytkowników API. Ze strony konsumenta nie są wymagane żadne zmiany. Jednak podczas uaktualniania trzeba uwzględniać czas przestoju i dostępność obu wersji, nowej i starej. W przypadku małej poprawki eliminującej drobny błąd w stylu niepoprawnej nazwy pola taka zmiana spowoduje, że zgodność API nie będzie zachowana.

*Niezachowanie zgodności z poprzednią wersją API, więc wszyscy konsumenci muszą uaktualnić kod w celu używania nowego API.*

Na początku wydaje się to kiepskim pomysłem, ponieważ może prowadzić do nieoczekiwanych skutków w środowisku produkcyjnym.[1] Może się również zdarzyć sytuacja, w której nie da się uniknąć zmiany prowadzącej do niezachowania zgodności z poprzednimi wersjami API. Tego rodzaju zmiana może doprowadzić do blokady całego systemu, co z kolei będzie wymagało koordynacji przestoju.

Trudność polega tutaj na tym, że te różne rodzaje uaktualnienia mają swoje wady i zalety zarówno dla konsumenta, jak i producenta. W rzeczywistości zwykle chcemy zapewnić obsługę wariantu będącego pewnego rodzaju połączeniem wszystkich trzech opcji. Aby można to było zrobić, konieczne jest wprowadzenie reguł dotyczących wersjonowania i wyjaśnienie, jak wersje są przekazywane konsumentowi.

[1] Wielokrotnie znaleźliśmy się takiej sytuacji. Najczęściej była to pierwsza rzecz, którą musieliśmy się zajmować w poniedziałkowy poranek.

## Wersjonowanie semantyczne

**Wersjonowanie semantyczne** (*https://semver.org/*) oferuje podejście, które można zastosować dla API REST i wykorzystać połączenie wymienionych wcześniej opcji uaktualnienia. Wersjonowanie semantyczne definiuje wartość liczbową, która będzie powiązana z wydaniem API. Będzie to liczba bazująca na zmianie w sposobie działania względem poprzedniej wersji, z zastosowaniem przy tym następujących reguł:

- Wersja *główna* wprowadza zmiany niezgodne z poprzednimi wersjami API. Na platformie API decyzja o uaktualnieniu do nowej wersji głównej będzie pozostawała w gestii konsumenta. Prawdopodobnie będzie dostarczony przewodnik migracji pomagający konsumentom w uaktualnieniu aplikacji do używania nowego API.
- Wersja *mniejsza* wprowadza zmiany zgodne z poprzednimi wersjami API. Na platformie usługi API jest akceptowane przez konsumentów otrzymywanie wersji mniejszej bez konieczności wprowadzania zmian po stronie klienta.
- Wersja *poprawki* nie wprowadza żadnych zmian i nie dodaje nowej funkcjonalności. Ta wersja jest używana podczas usuwania błędów w obecnej funkcjonalności oferowanej przez wersję *Główną.Mniejszą*.

Formatowanie wersjonowania semantycznego można przedstawić w postaci *Główna.Mniejsza.Poprawki*. Na przykład *1.5.1* oznacza wersję główną 1, wersję mniejszą 5 i wersję poprawki 1. W rozdziale 5. dowiesz się więcej o tym, jak wersjonowanie semantyczne łączy się z koncepcją wydań i cyklu życiowego API.

## Specyfikacja OpenAPI i wersjonowanie

Po zapoznaniu się z wersjonowaniem możesz spojrzeć na przykłady zmian zachowujących zgodność i niezachowujących zgodności z poprzednimi wersjami API. W tym celu posłużymy się specyfikacją API uczestnika. Dostępnych jest wiele narzędzi pozwalających na porównywanie specyfikacji. W omawianym przykładzie wykorzystamy narzędzie `openapi-diff` pochodzące z pakietu **Open APITools** (*https://github.com/OpenAPITools/openapi-diff*).

Rozpoczynamy od zmiany niezachowującej zgodności z poprzednimi wersjami API — zmieniamy nazwę pola `givenName` na `firstName`. Ta zmiana nie zachowuje zgodności, ponieważ konsumenci będą oczekiwali przetwarzania pola `givenName`, a nie `firstName`. W celu uruchomienia narzędzia typu `diff` w kontenerze Dockera możemy posłużyć się następującym poleceniem:

```
$ docker run --rm -t \
   -v $(pwd):/specs:ro \
   openapitools/openapi-diff:latest /specs/original.json /specs/first-name.json
================================================================================
...
- GET    /attendees
  Return Type:
    - Changed 200 OK
      Media types:
        - Changed */*
          Schema: Broken compatibility
          Missing property: [n].givenName (string)
```

```
-------------------------------------------------------------------------
--                              Result                                 --
-------------------------------------------------------------------------
                API changes broke backward compatibility
-------------------------------------------------------------------------
```

W drugim przypadku próbujemy dodać nowy atrybut do typu zwrotnego punktu końcowego `/attendees` w celu dodania kolejnego pola o nazwie `age`. Dodanie nowego pola nie powoduje zmiany sposobu działania istniejącej funkcjonalności i dlatego nie jest zmianą niezachowującą zgodności z poprzednimi wersjami API.

```
$ docker run --rm -t \
 -v $(pwd):/specs:ro \
openapitools/openapi-diff:latest --info /specs/original.json /specs/age.json
=========================================================================
...
- GET    /attendees
  Return Type:
    - Changed 200 OK
      Media types:
        - Changed */*
          Schema: Backward compatible
-------------------------------------------------------------------------
--                              Result                                 --
-------------------------------------------------------------------------
                  API changes are backward compatible
-------------------------------------------------------------------------
```

Warto wypróbować te przykłady i sprawdzić, które zmiany zapewniają zgodność z poprzednimi wersjami API, a które jej nie zapewniają. Wykorzystanie tego rodzaju narzędzi podczas pracy z API pomoże uniknąć nieoczekiwanych i niezgodnych zmian dla konsumentów. Specyfikacje OpenAPI są ważną częścią programu API, zaś w połączeniu z narzędziami, wersjonowaniem i cyklem życiowym okazują się bezcenne.

Narzędzia często są przeznaczone dla konkretnej wersji OpenAPI. Dlatego też duże znaczenie ma sprawdzenie, czy dane narzędzie będzie obsługiwało specyfikację, z której korzystasz. W poprzednim przykładzie zostało użyte narzędzie typu diff w wersji wcześniejszej niż specyfikacja i nie wykryło żadnych zmian niezgodnych z wcześniejszymi wersjami API.

# Implementacja RPC za pomocą gRPC

Usługi ruchu sieciowego typu wschód-zachód, takie jak `Attendee` w omawianym przykładzie, zwykle obsługują większy ruch sieciowy i mogą być zaimplementowane jako mikrousługi używane w całej architekturze. Styl gRPC dla usług ruchu sieciowego typu wschód-zachód może okazać się znacznie odpowiedniejszym narzędziem niż REST, ze względu na przekazywanie mniejszej ilości danych oraz szybkość działania w ekosystemie. Wszelkie decyzje dotyczące wydajności działania zawsze powinny być sprawdzone, aby można było podejmować je świadomie.

Przeanalizujemy teraz przykład użycia narzędzia **Spring Boot Starter** (*https://github.com/yidongnan/grpc-spring-boot-starter*) w celu szybkiego utworzenia serwera **gRPC**. Przedstawiony tutaj plik *.proto*

modeluje dokładnie ten sam obiekt `attendee`, który został przeanalizowany w przykładzie dotyczącym specyfikacji OpenAPI. Podobnie jak w przypadku specyfikacji OpenAPI wygenerowanie kodu na podstawie schematu odbywa się szybko i jest możliwe w wielu językach programowania.

Plik *.proto* usługi uczestnika definiuje puste żądanie i zwraca powtórzoną odpowiedź `Attendee`. W protokołach używanych dla reprezentacji binarnych trzeba koniecznie pamiętać, że położenie i kolejność pól mają znaczenie krytyczne, ponieważ narzucają one układ komunikatu. Wprawdzie dodanie nowej usługi lub metody to zmiana zachowująca zgodność, podobnie jak dodanie pola do komunikatu, ale i tak należy zachować ostrożność. Nowo dodawane pole nie może być obowiązkowe, ponieważ w przeciwnym razie będzie to zmiana niezachowująca zgodności.

Usunięcie pola bądź zmiana jego nazwy spowoduje niezachowanie zgodności, ponieważ będziemy mieli wówczas do czynienia ze zmianą typu danych pola. Zmiana numeru pola także jest problematyczna — te numery są używane do identyfikacji pól. Ograniczenia związane z kodowaniem za pomocą gRPC oznaczają, że definicja musi być bardzo konkretna. Specyfikacje REST i OpenAPI są dość liberalne, a sama specyfikacja to tak naprawdę jedynie zalecenie.[2] Dodatkowe pola i ich numerowanie nie mają znaczenia w OpenAPI, więc wersjonowanie i zachowanie zgodności mają jeszcze większe znaczenie w przypadku gRPC.

```
syntax = "proto3";
option java_multiple_files = true;
package com.masteringapi.attendees.grpc.server;

message AttendeesRequest {
}

message Attendee {
  int32 id = 1;
  string givenName = 2;
  string surname = 3;
  string email = 4;
}

message AttendeeResponse {
  repeated Attendee attendees = 1;
}

service AttendeesService {
  rpc getAttendees(AttendeesRequest) returns (AttendeeResponse);
}
```

Zamieszczony w kolejnym fragmencie kod Javy przedstawia prostą strukturę przeznaczoną do implementacji sposobu działania na bazie wygenerowanej klasy serwera gRPC.

```
@GrpcService
public class AttendeesServiceImpl extends
    AttendeesServiceGrpc.AttendeesServiceImplBase {

    @Override
    public void getAttendees(AttendeesRequest request,
```

[2] Weryfikacja specyfikacji OpenAPI przeprowadzana w trakcie działania aplikacji pomaga w zapewnieniu jeszcze większej ścisłości.

```
            StreamObserver<AttendeeResponse> responseObserver) {
              AttendeeResponse.Builder responseBuilder
                  = AttendeeResponse.newBuilder();

              //Umieszczenie danych w odpowiedzi.
              responseObserver.onNext(responseBuilder.build());
              responseObserver.onCompleted();
        }
    }
```

Usługę Javy modelującą ten przykład znajdziesz na stronie książki w serwisie *GitHub* (*https://github.com/masteringapi/attendees*). Do gRPC nie można wykonywać zapytań bezpośrednio z przeglądarki WWW bez użycia dodatkowych bibliotek. Jednak można zainstalować interfejs użytkownika gRPC (*https://github.com/fullstorydev/grpcui*) i wówczas wykorzystać przeglądarkę WWW podczas testów. Z kolei `grpcurl` to działające w powłoce polecenie przeznaczone do pracy z gRPC.

```
$ grpcurl -plaintext localhost:9090 \
    com.masteringapi.attendees.grpc.server.AttendeesService/getAttendees
{
  "attendees": [
    {
      "id": 1,
      "givenName": "Jim",
      "surname": "Gough",
      "email": "gough@mail.com"
    }
  ]
}
```

gRPC to jeszcze inna możliwość w zakresie wykonywania zapytań do naszej usługi. Definiuje specyfikację pozwalającą konsumentowi na wygenerowanie kodu. gRPC ma znacznie ściślejszą specyfikację niż OpenAPI i wymaga, aby metody i wewnętrzny sposób działania tego podejścia były zrozumiałe dla konsumenta.

# Modelowanie zmian i wybór formatu API

We wprowadzeniu zostały omówione koncepcja wzorców ruchu sieciowego oraz różnice między żądaniami przychodzącymi z zewnątrz ekosystemu a żądaniami wykonywanymi wewnątrz ekosystemu. Wzorce ruchu sieciowego to ważny czynnik podczas określania odpowiedniego formatu dla API. Gdy zachowujesz pełną kontrolę nad usługami i wymianą danych w architekturze bazującej na mikrousługach, możesz zdecydować się na kompromisy, które byłyby niemożliwe w przypadku konsumentów zewnętrznych.

Trzeba wiedzieć, że charakterystyka wydajności działania usługi typu wschód-zachód prawdopodobnie będzie odpowiedniejsza niż usługi typu północ południe. W przypadku wymiany danych typu północ-południe ruch sieciowy pochodzący z zewnątrz środowiska producenta, ogólnie rzecz biorąc, będzie obejmował wymianę danych poprzez internet. Internet wprowadza wysoki poziom opóźnienia, zaś w architekturze API zawsze należy rozważać skumulowane efekty powodowane przez poszczególne usługi. Z kolei w architekturze bazującej na mikrousługach istnieje prawdopodobieństwo, że żądanie typu północ-południe będzie obejmowało wiele wymian danych

typu wschód-zachód. Wysoki poziom wymiany danych typu wschód-zachód musi charakteryzować się efektywnością działania, aby uniknąć kaskadowego spowolnienia, które będzie propagowane do klienta.

## Usługi z wysokim poziomem ruchu sieciowego

W omawianym przykładzie usługa uczestnika jest usługą centralną. W architekturze bazującej na mikrousługach komponenty będą śledziły wartość `attendeeId`. API oferowane konsumentom będzie potencjalnie pobierało dane przechowywane w usłudze uczestnika, a na dużą skalę będzie to komponent generujący ogromną ilość ruchu sieciowego. Jeżeli częstotliwość wymiany danych między usługami jest duża, wówczas koszt przekazywania danych będzie się zwiększał wraz ze wzrostem wykorzystania usługi, co ma związek z wielkością danych i ograniczeniami protokołu w porównaniu z innymi protokołami. Ten koszt może być wyrażony zarówno w kwotach za wykorzystany transfer, jak i w ilości czasu potrzebnego na dotarcie komunikatu do miejsca docelowego.

## Ogromne ilości wymienianych danych

Ogromna ilość danych również staje się wyzwaniem podczas ich wymiany za pomocą API i może prowadzić do mniejszej wydajności w trakcie przekazywania danych w sieci. Dane `JSON` przekazywane przez usługę REST są czytelne dla człowieka i często będą znacznie obszerniejsze niż dane w postaci stałej bądź binarnej, a ponadto zwiększą wielkość komunikatów.

Powszechnie i błędnie przyjmuje się, że „czytelność dla człowieka" jest podstawowym powodem używania formatu JSON podczas przekazywania danych. W przypadku nowoczesnych narzędzi monitorowania nie jest argumentem to, ile razy programista będzie musiał odczytywać komunikat w porównaniu z kwestiami wydajności działania. Ponadto naprawdę rzadko się zdarza, aby obszerne pliki JSON były odczytywane od początku do końca. Lepsze rejestrowanie danych i obsługa błędów są w stanie zmniejszyć wagę argumentu czytelnego dla człowieka.

Kolejnym czynnikiem dotyczącym ogromnej ilości wymienianych danych jest ilość czasu potrzebna komponentom na przetworzenie treści komunikatu na postać obiektów domeny na poziomie języka programowania. Wydajność przetwarzania formatów danych będzie w dużej mierze zależała od języka programowania, w którym została zaimplementowana usługa. Wiele tradycyjnych języków działających po stronie serwera zmaga się z formatem `JSON`, zaś nie ma problemów z przetwarzaniem danych w postaci np. binarnej. Warto przeanalizować wpływ wydajności operacji przetwarzania i uwzględnić to podczas wyboru formatu wymiany danych.

## Korzyści związane z wydajnością działania HTTP/2

Używanie usług bazujących na protokole HTTP/2 może pomóc w poprawieniu wydajności operacji wymiany danych, co jest możliwe dzięki obsłudze kompresji binarnej i technologii *binary framing layer*. Ta technologia (*https://web.dev/performance-http2/#binary-framing-layer*) pozostaje niewidoczna dla programisty, natomiast w tle będzie dzieliła komunikat na mniejsze fragmenty i go kompresowała. Zaletą jej używania jest możliwość przekazania pełnego żądania i odpowiedzi na nie poprzez pojedyncze połączenie. Rozważ np. przetworzenie listy w innej usłudze i konieczność

pobrania informacji o 20 różnych uczestnikach. Jeżeli to zadanie będzie wykonywane za pomocą poszczególnych żądań HTTP/1, wówczas będzie wiązało się to z obciążeniem polegającym na utworzeniu 20 nowych połączeń TCP. Multiplexing pozwoli na wykonanie 20 poszczególnych żądań poprzez jedno połączenie HTTP/2.

gRPC domyślnie używa HTTP/2 oraz zmniejsza wielkość wymiany danych dzięki wykorzystaniu protokołu binarnego. Gdy przepustowość sieci jest ograniczona bądź kosztowna, wówczas zastosowanie gRPC będzie korzystne, zwłaszcza jeśli wielkość danych w komunikatach będzie się znacznie zwiększała. Użycie gRPC może być lepszym rozwiązaniem niż REST w sytuacjach, gdy przepustowość sieci jest ograniczona bądź usługa przekazuje ogromne ilości danych. W przypadku częstej wymiany ogromnego wolumenu danych warto rozważyć pewne asynchroniczne aspekty gRPC.

Opracowywany jest protokół HTTP/3, który wszystko zmieni. Będzie używał QUIC, czyli protokołu transportowego zbudowanego na bazie UDP. Na stronie *https://http3-explained.haxx.se/* zamieszczono dość dokładne omówienie protokołu HTTP/3.

### Stare formaty

Nie wszystkie usługi w architekturze będą bazowały na nowoczesnym projekcie. W rozdziale 8. wyjaśnimy, jak można odizolować i modernizować stare komponenty, ponieważ w przypadku architektury ewoluującej używanie starszych komponentów jest często rozważanym podejściem. Bardzo duże znaczenie ma to, aby osoby zaangażowane w tworzenie architektury API rozumiały ogólny wpływ starych komponentów na wydajność działania.

## Wskazówki dotyczące dokumentu typu ADR — modelowanie wymiany danych

Gdy konsumentem jest zespół starego systemu konferencyjnego, wówczas wymiana danych zwykle odbywa się w ramach relacji typu wschód-zachód. Natomiast jeśli konsumentem jest zespół CFP, wtedy wymiana danych zwykle odbywa się w ramach relacji typu północ-południe. Różnice dotyczące powiązania i wymagania w zakresie wydajności działania będą zmuszały zespoły do zastanowienia się nad sposobem modelowania wymiany danych. Wybrane aspekty z tym związane oraz wskazówki zostały zamieszczone w tabeli 1.3.

*Tabela 1.3. Wskazówki dotyczące modelowania wymiany danych*

| Decyzja | Który format powinien być użyty do modelowania API w budowanej usłudze? |
|---|---|
| Kwestie do omówienia | Z jakiego typu wymianą danych mamy do czynienia: północ-południe czy wschód-zachód? |
| | Czy mamy kontrolę nad kodem konsumenta? Czy istnieje silna domena biznesowa między wieloma usługami bądź czy chcesz pozwolić konsumentom na tworzenie własnych zapytań? |
| | Czy istnieją do rozważenia jakiekolwiek kwestie związane z wersjonowaniem? |
| | Jaka jest częstotliwość wdrażania i wprowadzania zmian w modelu danych? |
| | Czy usługa charakteryzuje się wysokim poziomem ruchu sieciowego, a kwestie dotyczące przepustowości sieci lub wydajności działania są ważne? |

*Tabela 1.3. Wskazówki dotyczące modelowania wymiany danych – ciąg dalszy*

| Decyzja | Który format powinien być użyty do modelowania API w budowanej usłudze? |
|---|---|
| Zalecenia | Jeżeli API jest przeznaczone dla użytkowników zewnętrznych, wówczas styl REST oznacza niskie bariery wejścia i dostarcza silny model domeny. Użytkownicy zewnętrzni oznaczają również, że najlepszym rozwiązaniem będzie usługa o małym stopniu powiązania i małej zależności od innych komponentów.<br><br>Jeżeli API jest przeznaczone do działania między dwiema usługami pozostającymi pod kontrolą producenta bądź usługa wiąże się z ogromnym ruchem sieciowym, wówczas należy rozważyć użycie rozwiązania bazującego na gRPC. |

# Wiele specyfikacji

W rozdziale przedstawiliśmy różne formaty API do rozważenia i zastosowania w architekturze API. Prawdopodobnie teraz nasuwa się następujące pytanie: „*Czy można dostarczyć wszystkie te formaty?*”. Odpowiedź na nie brzmi „*tak*”. Można opracować API dostarczające reprezentację RESTful, usługę gRPC i połączenia w schemacie GraphQL. Jednak takie rozwiązanie nie będzie łatwe i niekoniecznie musi być poprawnym podejściem. W ostatnim podrozdziale przedstawimy wybrane możliwości dostępne podczas tworzenia wieloformatowego API i wyzwania z tym związane.

## Czy istnieje złoty środek?

Plik *.proto* dla uczestników i specyfikacja OpenAPI przedstawiają się podobnie, zawierają te same pola i mają typy danych. Czy za pomocą narzędzia `openapi2proto` (*https://github.com/nytimes/openapi2proto*) można wygenerować plik *.proto* na podstawie OAS? Wydanie polecenia `openapi2proto --spec spec-v2.json` spowoduje utworzenie pliku *.proto* razem z polami ułożonymi domyślnie w kolejności alfabetycznej. To nie stanowi problemu aż do chwili dodania nowego pola do specyfikacji OAS zapewniającej wsteczną zgodność — nagle okazuje się, że zmianie uległy identyfikatory wszystkich pól, a tym samym nie będzie zachowana zgodność z poprzednimi wersjami API.

W kolejnym fragmencie kodu znajduje się zawartość przykładowego pliku *.proto*. Dodanie nowego pola `a_new_field` spowodowało, że zostało ono umieszczone na początku (pamiętaj o używanej domyślnie kolejności alfabetycznej), co doprowadziło do zmiany formatu binarnego i uszkodzenia istniejących usług.

```
message Attendee {
    string a_new_field = 1;
    string email = 2;
    string givenName = 3;
    int32 id = 4;
    string surname = 5;
}
```

Do rozwiązania problemu związanego z konwersją specyfikacji można wykorzystać inne narzędzia. Trzeba jednak pamiętać, że część z nich obsługuje jedynie wersję 2 specyfikacji OpenAPI. Czas potrzebny na przejście między wersjami 2 i 3 w niektórych narzędziach zbudowanych na bazie OpenAPI doprowadził do tego, że wiele produktów musiało zapewnić obsługę obu wersji OAS.

Rozwiązaniem alternatywnym jest narzędzie `grpc-gateway` (*https://github.com/grpc-ecosystem/grpc-gateway*) generujące proxy odwrotne, które zapewnia fasadę REST dla usługi gRPC. To proxy odwrotne zostaje wygenerowane i zbudowane na bazie pliku *.proto*. Wynikiem jego działania jest mapowanie na REST, podobnie jak otrzymane w przypadku użycia polecenia `openapi2proto`. Istnieje możliwość zamieszczenia rozszerzeń w pliku *.proto*, aby mapować metody gRPC na elegancką reprezentację w OAS.

```
import "google/api/annotations.proto";
//...
service AttendeesService {
  rpc getAttendees(AttendeesRequest) returns (AttendeeResponse) {
        option(google.api.http) = {
                get: "/attendees"
        };
}
```

Użycie narzędzia `grpc-gateway` to kolejna możliwość pozwalająca na dostarczenie usług REST i gRPC. Jednak to narzędzie wymaga wydania wielu poleceń w powłoce, a jego konfiguracja będzie znana jedynie programistom, którzy mają doświadczenie w pracy z językiem programowania Go lub ze środowiskiem kompilacji.

## Wyzwania związane z połączeniem specyfikacji

Ważne jest, aby wykonać krok wstecz i zastanowić się nad celem, który ma zostać osiągnięty. Podczas konwersji z OpenAPI w praktyce następuje próba konwersji naszej reprezentacji RESTful na serię wywołań gRPC. Próbujemy skonwertować rozszerzony model domeny hipermedialnej na serię niskiego poziomu wywołań między funkcjami. Mamy tutaj potencjalne połączenie różnic między RPC i API, a to prawdopodobnie doprowadzi do problemów związanych ze zgodnością.

Konwersja gRPC na OpenAPI wiąże się z podobnym problemem. Celem jest próba wykorzystania rozwiązania typu gRPC i zmiana go w taki sposób, aby przypominało API REST. Podczas ewolucji usługi prawdopodobnie doprowadzi to do powstania serii trudnych do rozwiązania problemów.

Po połączeniu specyfikacji lub wygenerowaniu jednej na podstawie innej wyzwaniem staje się wersjonowanie. Pod uwagę trzeba wziąć to, jak gRPC i specyfikacje OpenAPI będą zajmować się własnymi wymaganiami dotyczącymi zapewnienia zgodności. Należy ustalić, czy połączenie domen REST i RPC ma sens oraz czy przynosi jakąkolwiek wartość.

Zamiast wygenerować rozwiązanie RPC dla stylu typu wschód-zachód na podstawie stylu północ-południe, znacznie większy sens będzie miało staranne opracowanie komunikacji w architekturze bazującej na mikrousługach (RPC) niezależnie od reprezentacji REST. Dzięki temu oba rodzaje API będą mogły dowolnie ewoluować. Dokładnie taki wybór został dokonany w przykładzie omawianym w książce. Będzie on zarejestrowany w projekcie za pomocą dokumentu typu ADR.

# Podsumowanie

W rozdziale wyjaśniliśmy, jak opracowywać, budować i określać API, a także przedstawiliśmy różne okoliczności, które mogą wpływać na wybór między stylami REST i gRPC. Chcemy w tym miejscu wyraźnie podkreślić, że nie chodzi tutaj o starcie REST i gRPC, a raczej o wybór najlepszego podejścia w zakresie modelowania wymiany danych w konkretnej sytuacji. Oto najważniejsze wnioski płynące z tego rozdziału:

- W większości technologii mamy małe bariery wejścia podczas budowania API bazującego na REST lub RPC. Staranne przemyślenie projektu i struktury to bardzo ważna decyzja architekturalna.
- Podczas wyboru między modelami REST i RPC należy wziąć pod uwagę model dojrzałości Richardsona oraz stopień powiązania między producentem i konsumentem.
- REST to standard charakteryzujący się stosunkowo małym powiązaniem. Podczas budowy API zapewnienie zgodności ze standardem API gwarantuje spójność API i jego działanie zgodnie z oczekiwaniami konsumentów. Standardy API mogą również pomóc w eliminacji decyzji projektowych, których wynikiem byłoby API niezgodne z wcześniejszymi wersjami.
- Specyfikacje OpenAPI to użyteczny sposób na udostępnianie struktury API oraz automatyzację wielu działań związanych z tworzeniem kodu. Należy aktywnie wybierać funkcjonalności OpenAPI i decydować, które narzędzia bądź funkcje generowania znajdą zastosowanie w projektach.
- Wersjonowanie to ważny temat, który powoduje zwiększenie stopnia złożoności producenta, ale jest konieczny w celu ułatwienia konsumentowi używania API. Nieplanowanie wersjonowania w API prowadzi do zagrożenia dla konsumenta. Wersjonowanie powinno być aktywną decyzją dotyczącą zestawu funkcjonalności produktu. Wiąże się to z omówieniem mechanizmu przekazywania konsumentom informacji o wersjonowaniu.
- gRPC sprawdza się doskonale w przypadku wymiany dużej ilości danych. To również świetne rozwiązanie dla wymiany danych w stylu wschód-zachód. Narzędzia przeznaczone do pracy z gRPC mają potężne możliwości i stanowią kolejną opcję, którą warto wziąć pod uwagę podczas modelowania wymiany danych.
- Modelowanie wielu specyfikacji będzie dość trudnym zadaniem, zwłaszcza w przypadku generowania jednego typu specyfikacji na podstawie innej. Wersjonowanie jeszcze bardziej to komplikuje, choć jednocześnie stanowi ważny czynnik pomagający uniknąć zmian niezachowujących zgodności z poprzednimi wersjami API. Zespoły powinny starannie rozważyć wszystkie kwestie przed połączeniem w API reprezentacji RPC i RESTful, ponieważ między nimi istnieją poważne różnice w zakresie sposobu używania i zachowywania kontroli nad kodem konsumenta.

Wyzwaniem stojącym przed architekturą API jest spełnienie wymagań z perspektywy biznesowej konsumenta, zapewnienie programistom świetnych wrażeń podczas pracy z API oraz uniknięcie nieoczekiwanych problemów związanych z zapewnieniem zgodności. W następnym rozdziale przejdziemy do tematu testowania, które ma ogromne znaczenie, ponieważ zapewnia, że usługa spełnia zdefiniowane wymagania.

## W tej pozycji znajdziesz dokładne omówienie tematów związanych z nadawaniem struktury i ewoluowaniem API.

**Sam Newman**, autor książki *Budowanie mikrousług. Projektowanie drobnoziarnistych systemów*

Trendy w tworzeniu oprogramowania zmierzają w kierunku architektury zorientowanej na usługi. Coraz więcej organizacji decyduje się na rozwiązania chmurowe lub opate na mikrousługach. Te wzorce architekturalne wymagają API: umiejętności ich budowania i stosowania podczas integracji z systemami zewnętrznymi. Nie są to proste zagadnienia – zaprojektowanie i utworzenie platformy API jest prawdziwym wyzwaniem.

Ten przewodnik, który docenią programiści i architekci, zawiera wyczerpujące omówienie zagadnień projektowania, funkcjonowania i modyfikowania architektury API. Od strony praktycznej przedstawia strategie budowania i testowania API REST umożliwiającego połączenie oferowanej funkcjnalności na poziomie mikrousług. Opisuje stosowanie bram API i infrastruktury typu service mesh. Autorzy dokładnie przyglądają się kwestiom zapewnienia bezpieczeństwa systemów opartych na API, w tym uwierzytelnianiu, autoryzacji i szfrowaniu. Sporo miejsca poświęcają również ewolucji istniejących systemów w kierunku API i różnych docelowych platform.

## Doskonale napisana, zawiera wiele podpowiedzi, przykładów i praktycznych wskazówek.

**Stefania Chaplin**, GitLab & DevStefOps

## Najciekawsze zagadnienia:

- podstawy API i wzorce architekturalne platformy API
- wdrażanie i konfiguracja komponentów platformy API
- używanie bram API i infrastruktury typu service mesh
- bezpieczeństwo API i najczęstsze luki w zabezpieczeniach
- przekształcanie istniejących systemów w kierunku architektury bazującej na API

**James Gough** jest mistrzem Javy i architektem API. Był członkiem Java Community Process Executive Committee i rozwijał implementację OpenJDK.

**Daniel Bryant** ma bogate doświadczenie w zakresie platform chmury/kontenerów, a także implementacji mikrousług. Jest mistrzem Javy, rozwijał projekty open source.

**Matthew Auburn** opracował wiele aplikacji. Obecnie zajmuje się kwestiami bezpieczeństwa podczas tworzenia API.

---

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
*Sięgnij po więcej!*
ISBN 978-83-289-0720-1
9 788328 907201
Cena: 69,00 zł