Technologia i rozwiązania

REST
Najlepsze praktyki i wzorce w języku Java

Usprawnij wymianę danych z usługą REST!

Bhakti Mehta

Helion

[PACKT] PUBLISHING open source community experience distilled

Tytuł oryginału: RESTful Java Patterns and Best Practices

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-0647-9

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/restja.zip

Spis treści

O autorce

Bhakti Mehta jest autorką książki *Developing RESTful Services with JAX-Rs 2.0, WebSockets and JSON* (Packt Publishing, 2013). Od ponad 13 lat zajmuje się architekturą, projektowaniem i implementowaniem programów dla platformy Java EE i innych pokrewnych technologii. Fascynuje się ruchem *open source* i jest jednym z założycieli projektu otwartego serwera aplikacji GlassFish.

Mehta ma stopień inżyniera z inżynierii komputerowej i magistra z informatyki. W swoich badaniach skupia się na rozwiązaniach dotyczących odporności, skalowalności, niezawodności i wydajności technologii działających po stronie serwera i aplikacji chmurowych.

Aktualnie Mehta jest starszym programistą w Blue Jeans Network. Do jej zadań należy tworzenie usług sieciowych typu RESTful na użytek niezależnych partnerów i różnych innych programistów. Zajmuje się też infrastrukturą zaplecza i dba o to, by udostępniane usługi działały sprawnie, niezawodnie oraz szybko.

Mehta często bierze udział w konferencjach, napisała też wiele artykułów, wpisów na blogach i wskazówek technicznych, które zostały opublikowane w różnych portalach branżowych, takich jak *https://home.java.net/* czy Dzone. W wolnym czasie uprawia kickboxing, podróżuje i czyta.

Na Twitterze można ją znaleźć pod nazwą *@bhakti_mehta*.

Podziękowania

Pisanie książki to satysfakcjonująca, ale i wymagająca praca. Dlatego dziękuję mojej rodzinie za tolerowanie moich późnych powrotów do domu i weekendów spędzonych nad książką. Dziękuję mojemu mężowi Parikshatowi i teściom za wsparcie. Dziękuję Mansi za to, że zawsze służyła mi pomocą i stała po mojej stronie, gdy było ciężko! Książka ta jest dla moich dzieci, które są dla mnie niewyczerpanym źródłem inspiracji. Chcę, aby uwierzyły, że marzenia się spełniają, tylko trzeba nad tym ciężko pracować.

Dziękuję również moim rodzicom i bratu Pranavowi z rodziną za dopingowanie mnie do pracy nad tą książką. To łaska żyć wśród tak oddanych przyjaciół, których najwięcej pojawiło się w moim życiu po ukończeniu studiów inżynierskich. Dziękuję Wam za motywację. Nie da się wyrazić słowami, jak bardzo jestem wdzięczna kolegom z Blue Jeans Network za ich entuzjastyczne wsparcie i dobre życzenia.

Dziękuję pracownikom z wydawnictwa Packt Publishing, w szczególności takim osobom jak Vinay Argekar, Adrian Raposo i Edwin Moses, za to, że skontaktowali się ze mną, przejrzeli treść i pilnowali harmonogramu. Na koniec chciałabym podziękować recenzentom za cenne rady i skrupulatność — osoby, o których mowa, to: Masoud Kalali, Dustin R. Callaway, Kausal Malladi oraz Antonio Rodrigues.

O recenzentach

Dustin R. Callaway jest konsultantem ds. oprogramowania, pisarzem, konstruktorem i wszechstronnym programistą. Aktualnie jest starszym programistą w Intuit Inc., wiodącej firmie dostarczającej oprogramowanie finansowe. Callaway ukończył studia inżynierskie z informatyki w Brigham Young University i jest autorem książki *Inside Servlets* (Addison-Wesley). Do jego zainteresowań zawodowych zalicza się tworzenie usług sieciowych typu RESTful przy użyciu języka Java i systemu Node.js oraz tworzenie aplikacji mobilnych i sieciowych.

Masoud Kalali jest konsultantem technicznym w firmie Oracle. Napisał książki *Developing RESTful Services with JAX-Rs 2.0, WebSockets and JSON* (Packt Publishing, 2013) i *GlassFish Security* (Packt Publishing, 2010). Ponadto jest autorem wielu artykułów i krótkich poradników publikowanych w różnych portalach, od Java.Net po Dzone.

Od 2001 roku, w którym rozpoczął pracę z różnymi technologiami programistycznymi, Kalali miał szczęście pracować przy wielu luźno powiązanych ze sobą rozwiązaniach architektonicznych dla wydajnych systemów przesyłania wiadomości opartych na JMS i innych elementach towarzyszących tej technologii, która stanowi dla nich główną magistralę komunikacyjną.

Wiele czasu Kalali spędził też na analizowaniu wydajności oraz udzielaniu konsultacji na temat architektury, projektowania, kodowania i rozwijania konfiguracji. Do jego specjalności należą również usługi RESTful i wykorzystywanie ich punktów końcowych do integracji danych. Podczas pracy w ChemAxon zajmował się systemami takich klientów jak IJC i TIBCO Spotfire.

Kolejną dziedziną, w której Kalali ma doświadczenie, jest integracja zabezpieczeń, a w szczególności integracja OpenSSO z systemem szkieletowym SOA do rozwijania programów przepływowych w języku BPEL. Aktualnie pracuje w firmie Oracle na stanowisku głównego programisty. Zajmuje się projektowaniem i rozwijaniem serwera aplikacji i infrastruktury PaaS usługi chmurowej firmy Oracle bazującej na rozwiązaniach wirtualizacji OVM/OVAB i Numbula.

Jeśli ktoś jest ciekaw, czym obecnie zajmuje się Masoud Kalali, może go znaleźć na Twitterze pod nazwą *@MasoudKalali*.

Kausal Malladi to inżynier programista nastawiony na efekty, chętny, by ciągle poznawać najnowsze technologie, rozwiązywać problemy informatyczne i rozwijać innowacyjne produkty. Jest magistrem informatyki w International Institute of Information Technology w Bangalore (IIIT-B). Od ponad dwóch lat zajmuje się projektowaniem i rozwijaniem oprogramowania i aktualnie pracuje w firmie Intel.

W Intelu Malladi należy do zespołu zajmującego się rozwijaniem oprogramowania graficznego dla systemu Android. Wcześniej, zanim ukończył studia magisterskie, kilka lat pracował w firmie Infosys Ltd. Należał tam do wewnętrznego zespołu zajmującego się rozwojem i badaniami efektywnych rozwiązań trudnych problemów infrastrukturalnych.

Jako zapalony naukowiec Malladi opublikował kilka artykułów w renomowanych pismach naukowych. W 2013 roku złożył też kilka wniosków patentowych w Indiach. W tym samym roku wygłosił referat pt. *ATM Terminal Services the RESTful Way* na konferencji JavaOne w Indiach.

Kausal Malladi lubi bawić się hobbystycznymi projektami dotyczącymi technologii chmurowych i uczenia maszynowego, a także jest gorącym zwolennikiem programowania sieciowego i ruchu *open source*. Uwielbia muzykę. W wolnym czasie słucha, śpiewa i gra (na skrzypcach) muzykę karnatacką. Jest też wolontariuszem w organizacji zajmującej się promocją indyjskiej muzyki klasycznej wśród młodych (SPIC MACAY). Jest to ruch młodzieżowy działający zarówno na froncie technicznym, jak i organizacyjnym.

Więcej informacji na jego temat można znaleźć na stronie *http://www.kausalmalladi.com*.

Antonio Rodrigues to inżynier oprogramowania o bogatym doświadczeniu w dziedzinie programowania po stronie serwera i aplikacji mobilnych. Przez ostatnie 17 lat pracował w wielu firmach konsultingowych IT i telekomunikacyjnych, agencjach rządowych, agencjach cyfrowych oraz start-upach. Uważa, że interfejsy API, a w szczególności usługi typu RESTful, są w dzisiejszym mobilnym świecie podstawą.

Można go znaleźć na Twitterze pod nazwą *@aaadonai*.

Wstęp

Z połączenia mediów społecznościowych, technologii chmurowych i aplikacji mobilnych wyłania się nowa generacja technologii umożliwiających różnym przyłączonym do sieci urządzeniom komunikowanie się przez internet. Kiedyś stosowano tradycyjne i zamknięte rozwiązania, polegające na użyciu różnych urządzeń i komponentów, które komunikowały się ze sobą przez zawodną sieć lub internet. Niektóre z tych technologii, jak np. RPC CORBA i usługi oparte na protokole SOAP, które powstały jako różne implementacje architektury SOA, wymagały ściślejszego powiązania elementów i były bardziej skomplikowane.

Ale dzięki rozwojowi technologii obecnie aplikacje tworzy się w oparciu o paradygmat dostarczania i używania interfejsów API, zamiast wykorzystywać do tego celu szkielety sieciowe wywołujące usługi i tworzące strony internetowe. Taka architektura oparta na API umożliwia stosowanie zwinnych technik programowania, ułatwia adaptację i rozpowszechnianie oraz skalowanie i integrację aplikacji zarówno w firmach, jak i poza nimi.

Powszechność technologii REST i JSON sprawia, że ułatwione jest wykorzystywanie funkcjonalności jednych aplikacji w innych. Technologia REST swoją popularność zawdzięcza głównie temu, że umożliwia budowanie lekkich, prostych i tanich modułowych interfejsów, z których mogą korzystać rozmaite klienty.

Pojawienie się aplikacji mobilnych wymusiło dokładniejsze wyodrębnienie modelu klient-serwer. Firmy tworzące aplikacje dla platform iOS i Android mogą używać interfejsów API typu REST oraz zwiększać zasięg swojej działalności poprzez połączenie danych z różnych platform.

Dodatkowymi zaletami technologii REST są jej bezstanowość, skalowalność, widoczność, niezawodność oraz niezależność od platform i języków programowania. Wiele firm do zabezpieczeń i zarządzania tokenami zaczyna używać protokołu OAuth 2.0.

W pierwszej części tej książki znajduje się opis architektury REST ze szczególnym uwzględnieniem wszystkich wymienionych tematów. Następnie przechodzę do bardziej szczegółowego opisu najlepszych praktyk i często stosowanych rozwiązań dotyczących budowy lekkich, skalowalnych, niezawodnych i szeroko dostępnych usług typu RESTful.

Zawartość książki

Rozdział 1., „Podstawy REST", zawiera opis podstawowych pojęć dotyczących technologii REST, projektowania usług typu RESTful oraz najlepszych metod projektowania zasobów REST. Znajduje się w nim opis interfejsu API JAX-RS 2.0 służącego do budowy usług typu RESTful w Javie.

Rozdział 2., „Projektowanie zasobów", zawiera opis różnych rodzajów odpowiedzi na żądania. Omawiam takie tematy jak negocjacja treści, wersjonowanie zasobów oraz kody odpowiedzi w REST.

Rozdział 3., „Bezpieczeństwo i wykrywalność", zawiera szczegółowy opis zaawansowanych kwestii związanych z bezpieczeństwem i identyfikacją dotyczących interfejsów API typu REST. W tym rozdziale opisuję metody kontroli dostępu, techniki uwierzytelniania przy użyciu protokołu OAuth, sposoby obsługi wyjątków oraz metody sprawdzania poprawności danych.

Rozdział 4., „Projektowanie wydajnych rozwiązań", zawiera opis zasad projektowania ze szczególnym naciskiem na uzyskanie jak najwyższej wydajności. Omawiam w nim zasady buforowania, asynchroniczne i długotrwałe zadania w REST oraz sposoby wykonywania częściowych aktualizacji metodą `PATCH`.

Rozdział 5., „Zaawansowane zasady projektowania", zawiera opis zaawansowanych problemów dotyczących ograniczania liczby żądań, stronicowania odpowiedzi oraz internacjonalizacji i lokalizacji wraz ze szczegółowymi przykładami. W rozdziale tym omawiam kwestię rozszerzalności, ograniczenia HATEOAS oraz techniki testowania i dokumentowania usług typu REST.

Rozdział 6., „Nowe standardy i przyszłość technologii REST", zawiera opis interfejsów API czasu rzeczywistego używających uchwytów sieciowych, gniazd sieciowych, modelu PuSH oraz usług zdarzeń serwerowych i porównanie ich pod różnymi względami. Dodatkowo w tym rozdziale przedstawiam studia przypadków demonstrujące sposób użycia nowych technologii, takich jak gniazda sieciowe i uchwyty sieciowe, w realnych aplikacjach, oraz opisuję rolę technologii REST w mikrousługach.

Dodatek A zawiera opis różnych API typu REST z portali GitHub, Twitter i Facebook oraz informacje o tym, w jakim zakresie zrealizowano w nich zasady opisane w rozdziałach od 2. do 5.

Co jest potrzebne

Aby skompilować i uruchomić przykłady towarzyszące tej książce, potrzebne są następujące programy:

- Apache Maven 3.0 lub nowszy — Maven to narzędzie do kompilowania programów, które można pobrać pod adresem *http://maven.apache.org/download.cgi*.
- GlassFish Server wersja open source v4.0 — jest to darmowy, rozwijany przez społeczność serwer aplikacji dostarczający implementację Javy EE 7. Program ten można pobrać ze strony *http://dlc.sun.com.edgesuite.net/glassfish/4.0/promoted/*.

Adresaci książki

Książka ta jest idealna dla programistów aplikacji, którzy chcę poznać technologię REST. Zawiera wiele szczegółowych informacji, opisy najlepszych praktyk oraz typowych wzorców programowania REST, jak również wskazówki na temat tego, jak swoje rozwiązania typu RESTful implementują portale Facebook, Twitter, PayPal, GitHub czy Stripe.

Konwencje typograficzne

W książce tej zastosowano pewne style typograficzne ułatwiające rozróżnianie poszczególnych rodzajów informacji. Poniżej znajdują się przykłady ich użycia i krótkie objaśnienia.

Nazwy folderów i plików, adresy internetowe oraz nazwy profilów Twittera są zapisywane tak: „Szczegóły znajdziesz w pliku *example.doc*, pod adresem *http://example.com*".

Fragmenty kodu w tekście, nazwy tabel baz danych, zmyślone adresy URL oraz dane wprowadzane przez użytkownika są oznaczane jak w tym przykładzie: „`GET` i `HEAD` to bezpieczne metody".

Bloki kodu wyglądają tak:

```
@GET
@Path("orders")
public List<Coffee> getOrders() {
    return coffeeService.getOrders(); }
```

Jeśli trzeba zwrócić uwagę na wybraną część bloku kodu, została ona oznaczona pogrubieniem:

```
@Path("v1/coffees")
public class CoffeesResource {
@GET
@Path("orders")
@Produces(MediaType.APPLICATION_JSON)
```

```
public List<Coffee> getCoffeeList( ){
    // implementacja
}
```

Dane z wiersza poleceń również są oznaczane czcionką o stałej szerokości znaków:

```
# curl -X GET http://api.test.com/baristashop/v1.1/coffees
```

Nowe pojęcia i **ważne słowa** oznaczono pogrubieniem.

> W takich ramkach znajdują się ostrzeżenia i bardzo ważne uwagi, jak również wskazówki i opisy sztuczek.

Pobieranie przykładów kodu

Wszystkie pliki z kodem źródłowym opisanym w tej książce można pobrać pod adresem *ftp://ftp.helion.pl/przyklady/restja.zip*.

Errata

Mimo iż dołożyliśmy wszelkich starań, aby zapewnić treść o jak najwyższej jakości, błędy zdarzają się każdemu. Jeśli znajdziesz jakiś błąd, czy to w kodzie, czy w tekście, będziemy bardzo wdzięczni za informację. W ten sposób możesz pomóc innym czytelnikom oraz nam w zapewnieniu lepszej jakości przyszłych wydań. Erratę należy zgłaszać, korzystając z formularza na stronie *http://helion.pl/erraty.htm*. Dokładnie sprawdzamy każde takie zgłoszenie i jeśli uznamy uwagę za słuszną, publikujemy ją na stronie internetowej książki. Listę błędów można obejrzeć pod adresem *http://www.helion.pl/ksiazki/restja.htm*.

Piractwo

Nielegalne rozpowszechnianie materiałów chronionych prawem autorskim to narastający problem wszystkich rodzajów mediów. Wydawnictwo Helion bardzo poważnie podchodzi do tego problemu. Jeśli znajdziesz w internecie nielegalne kopie naszych produktów, prześlij nam adres do strony, na której się one znajdują, abyśmy mogli przedsięwziąć kroki zaradcze.

Adresy podejrzanych stron internetowych można wysyłać na adres *helion@helion.pl*.

Bardzo cenimy sobie Waszą pomoc w ochronie naszych praw autorskich i zawsze staramy się dostarczać towary jak najwyższej jakości.

ROZDZIAŁ 1

Podstawy REST

Usługi sieciowe w tradycyjnej technologii SOA, umożliwiające zróżnicowaną komunikację między aplikacjami, istnieją już od pewnego czasu. Jednym ze sposobów obsługi tej komunikacji jest użycie technologii ***Simple Object Access Protocol*** (**SOAP**) i ***Web Service Description Language*** (**WSDL**). Są to standardy oparte na formacie XML doskonale sprawdzające się, gdy między usługami jest ścisły kontakt. Ale nastała era usług rozproszonych. Teraz różne klienty z internetu, urządzenia przenośne, jak również inne usługi (wewnętrzne i zewnętrzne) mogą używać interfejsów API udostępnianych przez różnych dostawców i różne platformy *open source*. To sprawia, że potrzebne są technologie łatwej wymiany informacji między usługami rozproszonymi w różnych miejscach, z przewidywalnymi, solidnymi, ściśle zdefiniowanymi interfejsami.

Protokół HTTP 1.1, zdefiniowany w dokumencie RFC 2616, jest powszechnie używany w rozproszonych systemach hipermedialnych. Technologia ***Representational State Transfer*** (**REST**) bazuje na HTTP i może być używana wszędzie tam, gdzie ten protokół. W tym rozdziale przedstawione są podstawowe wiadomości na temat projektowania usług typu RESTful oraz używania takich usług za pomocą standardowych interfejsów API Javy.

W rozdziale omówiono następujące zagadnienia:

- Wprowadzenie do technologii REST.
- Bezpieczeństwo i idempotentność.
- Zasady projektowe dotyczące budowy usług typu RESTful.
- Standardowe API Javy dla usług typu RESTful.
- Najlepsze techniki projektowania usług typu RESTful.

Wprowadzenie do REST

REST to styl architektoniczny zgodny z takimi standardami sieciowymi jak czasowniki HTTP i identyfikatory URI. Obowiązują w nim następujące zasady:

- Wszystkie zasoby określa identyfikator URI.
- Każdy zasób może mieć liczne reprezentacje.
- Każdy zasób można pobrać, zmodyfikować, utworzyć i usunąć standardowymi metodami HTTP.
- Na serwerze nie są przechowywane żadne informacje o stanie.

REST i bezstanowość

W REST obowiązuje zasada **bezstanowości**. Każde żądanie przesyłane przez klienta do serwera musi zawierać wszystkie informacje potrzebne do obsługi tego zdarzenia. To poprawia widoczność, niezawodność i skalowalność żądań.

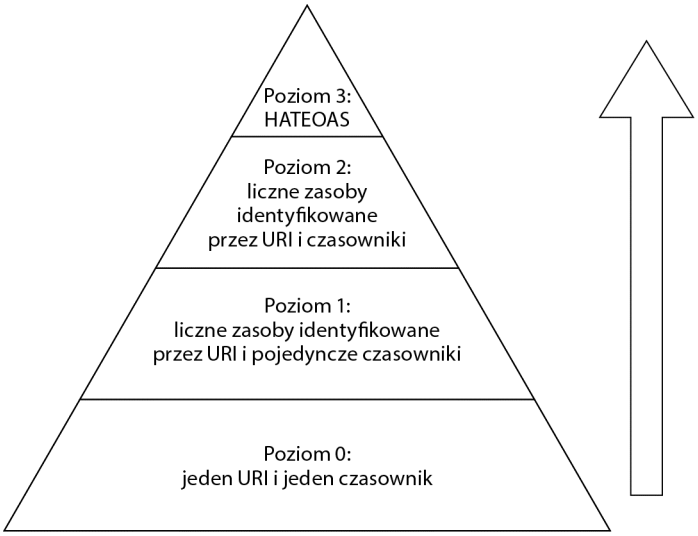
Poprawa *widoczności* wynika z tego, że system monitorujący żądania nie musi szukać szczegółów poza żądaniami. *Niezawodność* poprawia się dzięki wyeliminowaniu punktów kontrolnych i wznowienia w przypadku częściowych niepowodzeń operacji. Poprawa *skalowalności* jest efektem zwiększenia liczby żądań, które jest w stanie przetworzyć serwer, co jest możliwe dzięki temu, że serwer nie musi przechowywać informacji o stanie.

> Roy Fielding napisał doktorat na temat stylu architektonicznego REST, w którym szczegółowo opisał bezstanowość tej technologii. Więcej informacji na ten temat można znaleźć pod adresem *http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm*.

To są podstawowe wiadomości o technologii REST. Teraz zajmiemy się różnymi poziomami dojrzałości i zobaczymy, gdzie pośród nich mieści się ta technologia.

Model dojrzałości Richardsona

Model dojrzałości Richardsona to opracowany przez Leonarda Richardsona model opisujący podstawy REST pod względem zasobów, czasowników i hipermediów. Punktem początkowym tego modelu jest wykorzystanie HTTP jako warstwy transportowej. Ukazuje to poniższy schemat.



Model dojrzałości Richardsona

Poziom 0 — zdalne wywoływanie procedur

Do poziomu 0 zalicza się przesyłanie danych przy użyciu technologii SOAP i XML-RPC jako **POX** (ang. *Plain Old XML* — zwykły XML). Używana jest tylko metoda `POST`. Jest to najprostszy sposób budowania aplikacji SOA z jedną metodą `POST` i przy użyciu formatu XML do komunikacji między usługami.

Poziom 1 — zasoby REST

Na poziomie 1 używane są metody `POST`, a zamiast funkcji i przekazywania argumentów wykorzystuje się identyfikatory URI REST. Zatem nadal używana jest tylko jedna metoda HTTP. Zaletą tego poziomu w stosunku do zerowego jest podział złożonej funkcjonalności na kilka zasobów za pomocą jednej metody `POST` służącej do komunikacji między usługami.

Poziom 2 — dodatkowe czasowniki HTTP

Na poziomie drugim jest więcej czasowników, np. `GET`, `HEAD`, `DELETE`, `PUT` i oczywiście `POST`. Poziom ten reprezentuje rzeczywisty przypadek użycia technologii REST, w której wykorzystuje się różne czasowniki HTTP do wykonywania różnych żądań, a system może zawierać wiele zasobów.

Poziom 3 — HATEOAS

HATEOAS (ang. *Hypermedia as the Engine of Application State* — hipermedia jako mechanizm obsługi stanu aplikacji) reprezentuje najwyższy stopień dojrzałości w modelu Richardsona. Odpowiedzi na żądania klientów zawierają elementy hipermedialne, przy użyciu których klient może zdecydować, co robić dalej. Zasady te ułatwiają wykrywanie usług i sprawiają, że odpowiedzi są bardziej zrozumiałe. Toczą się dyskusje na temat tego, czy HATEOAS rzeczywiście spełnia wymagania RESTful, ponieważ reprezentacja zawiera o wiele więcej informacji niż tylko opis zasobu. W rozdziale 5. przedstawiam parę przykładów pokazujących, jak zaimplementowano HATEOAS jako część API niektórych platform, np. PayPal.

W następnym podrozdziale wyjaśniam pojęcia bezpieczeństwo i idempotentność, które są bardzo ważne w usługach RESTful.

Bezpieczeństwo i idempotentność

W poniższych dwóch podrozdziałach dokładniej wyjaśniam znaczenie bezpieczeństwa i idempotentności metod.

Bezpieczeństwo metod

Bezpieczna metoda to taka, która nie zmienia stanu na serwerze. Warunek ten spełnia na przykład metoda `GET /v1/coffees/orders/1234`.

> Bezpieczne metody, do których zaliczają się `GET` i `HEAD`, można buforować.
>
> Metoda `PUT` nie jest bezpieczna, ponieważ tworzy lub modyfikuje zasoby na serwerze. To samo dotyczy metody `POST`. Z kolei metoda `DELETE` nie jest bezpieczna, ponieważ usuwa zasoby z serwera.

Idempotentność metod

Metoda idempotentna to taka, która zwraca taki sam wynik niezależnie od tego, ile razy zostanie wywołana.

> Metoda `GET` jest idempotentna, ponieważ niezależnie od tego, ile razy się ją wywoła, zawsze zwraca taką samą odpowiedź.
>
> Metoda `PUT` też jest idempotentna, ponieważ wielokrotne jej wywołanie powoduje aktualizację tego samego zasobu i nie zmienia to wyniku.

Metoda POST nie jest idempotentna, ponieważ jej wielokrotne wywołanie może dawać różne skutki i powodować powstanie wielu zasobów. Metoda DELETE jest idempotentna, gdyż usunięty zasób znika i powtórne wywołanie tej samej metody niczego nie zmieni.

Zasady projektowe dotyczące budowy usług typu RESTful

Poniżej w punktach przedstawiam proces projektowania, tworzenia i testowania usług typu RESTful. Dalej znajduje się dokładniejszy opis każdego z tych etapów:

- Wyznaczenie identyfikatorów URI zasobów.

  Polega na wybraniu rzeczowników do reprezentowania zasobu.

- Identyfikacja metod obsługiwanych przez zasób.

  Polega na wybraniu metod HTTP do wykonywania operacji CRUD (ang. *create*, *read*, *update*, *delete* — utworzenie, odczytanie, aktualizacja, usunięcie).

- Identyfikacja różnych reprezentacji zasobu.

  Polega na wybraniu, czy zasób będzie reprezentowany w formacie JSON, XML, HTML, czy tekstowym.

- Implementacja usług RESTful przy użyciu API JAX-RS.

  Interfejs API należy zaimplementować na podstawie specyfikacji JAX-RS.

- Wdrożenie usług RESTful.

  Wdrożenie usługi w kontenerze aplikacji, np. Tomcat, GlassFish lub WildFly. Na przykładach pokazuję, jak tworzy się pliki WAR, i prezentuję sposób wdrożenia na serwerze GlassFish 4.0. Poza tym przedstawiony przykład działa w każdym kontenerze zgodnym z Java EE 7.

- Testowanie usług RESTful.

  Polega na napisaniu API klienta do testowania usług lub użyciu narzędzi cURL albo przeglądarkowych do testowania żądań REST.

Wyznaczenie identyfikatorów URI zasobów

Zasoby RESTful są identyfikowane przez identyfikatory URI. Dzięki temu technologia REST jest rozszerzalna.

Poniższa tabela zawiera przykłady identyfikatorów URI, które mogą reprezentować różne zasoby w systemie.

URI	Opis
/v1/library/books	Możliwa reprezentacja kolekcji zasobów książkowych w bibliotece.
/v1/library/books/isbn/12345678	Możliwa reprezentacja jednej książki identyfikowanej przez numer ISBN 123456.
/v1/coffees	Możliwa reprezentacja wszystkich kaw sprzedanych w kawiarni.
/v1/coffees/orders	Możliwa reprezentacja wszystkich zamówionych kaw w kawiarni.
/v1/coffees/orders/123	Możliwa reprezentacja pojedynczego zamówienia kawy o identyfikatorze 123.
/v1/users/1235	Możliwa reprezentacja użytkownika o identyfikatorze w systemie 1235.
/v1/users/5034/books	Możliwa reprezentacja wszystkich książek użytkownika o identyfikatorze 5034.

Wszystkie przedstawione identyfikatory są zbudowane wg jasnego wzorca, który klient może bez trudu zinterpretować. Wszystkie te zasoby mogą mieć liczne reprezentacje, np. w formacie JSON, XML, HTML lub tekstowym, i można nimi zarządzać za pomocą metod `GET`, `PUT`, `POST` i `DELETE`.

Identyfikacja metod obsługiwanych przez zasób

Czasowniki HTTP stanowią ważny składnik jednolitego ograniczenia interfejsu, które definiuje związek między czynnościami opisywanymi przez dany czasownik w stosunku do opisanego za pomocą rzeczowników zasobu REST.

Poniższa tabela zawiera zestawienie metod HTTP i opis powodowanych przez nie zdarzeń oraz prosty przykład kolekcji książek w bibliotece.

Metoda HTTP	URI zasobu	Opis
GET	/library/books	Pobiera listę książek.
GET	/library/books/isbn/12345678	Pobiera książkę o numerze ISBN 12345678.
POST	/library/books	Tworzy nowe zamówienie książki.
DELETE	/library/books/isbn/12345678	Usuwa książkę o numerze ISBN 12345678.
PUT	/library/books/isbn/12345678	Aktualizuje książkę o numerze ISBN 12345678.
PATCH	/library/books/isbn/12345678	Częściowo aktualizuje książkę o numerze ISBN 12345678.

W kolejnym podrozdziale znajduje się opis zastosowania każdego z czasowników HTTP w kontekście REST.

Czasowniki HTTP w REST

Czasowniki HTTP stanowią dla serwera informację o tym, co ma zrobić z otrzymanymi danymi.

GET

`GET` to najprostsza metoda HTTP pozwalająca uzyskać dostęp do zasobu. Gdy klient kliknie adres URL w przeglądarce internetowej, aplikacja ta wysyła żądanie `GET` pod ten właśnie adres. Metoda `GET` jest bezpieczna i idempotentna. Żądania wysyłane tą metodą są buforowane i mogą zawierać parametry.

Poniżej znajduje się proste żądanie `GET` pobierające wszystkich aktywnych użytkowników:

```
curl http://api.foo.com/v1/users/12345?active=true
```

POST

Metoda `POST` służy do tworzenia zasobów. Żądania wysyłane przy użyciu tej metody nie są idempotentne ani bezpieczne. Wielokrotne wywołania mogą spowodować utworzenie wielu zasobów.

Żądanie `POST` powinno powodować unieważnienie odpowiedniego elementu w buforze, jeśli taki istnieje. W żądaniach `POST` nie zaleca się stosowania parametrów zapytaniowych.

Poniżej znajduje się żądanie utworzenia użytkownika:

```
curl –X POST -d'{"name":"Jan Kowalski","username":"jkow", "phone":
  "412-344-5644"}' http://api.foo.com/v1/users
```

PUT

Metoda `PUT` służy do aktualizowania zasobów. Jest ona idempotentna, ale nie jest bezpieczna. Wielokrotne żądania tego typu powinny dawać taki sam efekt w postaci zaktualizowania zasobu.

Żądania `PUT` powinny unieważniać zawartość bufora, jeśli taka istnieje.

Poniżej znajduje się przykład żądania `PUT` aktualizującego użytkownika:

```
curl –X PUT -d'{ "phone":"413-344-5644"}'
http://api.foo.com/v1/users
```

DELETE

Metoda `DELETE` służy do usuwania zasobów. Jest idempotentna, ale nie jest bezpieczna. Idempotentność wynika z tego, że zgodnie ze specyfikacją RFC 2616 skutki uboczne dowolnej większej od zera liczby żądań są takie same jak jednego żądania. Oznacza to, że po usunięciu zasobu kolejne wywołania metody `DELETE` nic nie zmieniają.

Poniżej znajduje się przykładowe żądanie usuwające użytkownika:

```
curl –X DELETE http://foo.api.com/v1/users/1234
```

HEAD

Żądania typu `HEAD` są podobne do `GET`. Różnica między nimi polega na tym, że w odpowiedzi na żądanie `HEAD` zwracane są tylko nagłówki HTTP, bez treści. Metoda `HEAD` jest idempotentna i bezpieczna.

Poniżej znajduje się przykładowe żądanie `HEAD` wysyłane za pomocą narzędzia cURL:

```
curl –X HEAD http://foo.api.com/v1/users
```

> Jeśli zasób jest duży, to za pomocą żądania `HEAD` można sprawdzić, czy coś się w nim zmieniło, zanim się je pobierze przy użyciu żądania `GET`.

PUT czy POST

Zgodnie z dokumentem RFC różnica między metodami `PUT` i `POST` dotyczy identyfikatora URI żądania. W metodzie `POST` przesłany identyfikator URI definiuje jednostkę, która ma obsłużyć żądanie. W żądaniu `PUT` natomiast identyfikator URI zawiera tę jednostkę.

A zatem `POST /v1/coffees/orders` oznacza utworzenie nowego zasobu i zwrócenie opisującego go identyfikatora. `PUT /v1/coffees/orders/1234` oznacza natomiast aktualizację zasobu o identyfikatorze `1234`, jeśli taki istnieje. Jeśli nie ma takiego zasobu, zostanie utworzone nowe zamówienie, do którego identyfikacji zostanie użyty URI `orders/1234`.

> Zarówno metody `PUT`, jak i `POST` można używać do tworzenia i aktualizacji zasobów. Wybór jednej z nich zależy głównie od tego, czy potrzebna jest idempotentność metody, oraz od lokalizacji zasobu.

W następnym podrozdziale dowiesz się, jak identyfikować różne reprezentacje zasobu.

Identyfikacja różnych reprezentacji zasobu

Zasoby RESTful są jednostkami abstrakcyjnymi, które przed przesłaniem do klienta trzeba poddać serializacji do jakiegoś reprezentacyjnego formatu. Wśród najczęściej używanych reprezentacji można wymienić XML, JSON, HTML i zwykły tekst. Zasób może dostarczać klientowi reprezentację w zależności od tego, co klient ten jest w stanie przyjąć. Klient może określić preferowane przez siebie języki i typy mediów. Nazywa się to **negocjacją treści** i zostało szczegółowo opisane w rozdziale 2.

Implementowanie API

Wiesz już mniej więcej, jak projektować zasoby RESTful i jakich czasowników HTTP używać do wykonywania różnych działań na tych zasobach, więc możemy przejść do kwestii implementowania API i budowania usługi typu RESTful. Głównym tematem tego podrozdziału jest:

- API Javy dla usług RESTful (JAX-RS).

API Javy dla usług RESTful (JAX-RS)

API Javy dla usług RESTful służy do budowania i rozwijania aplikacji wg zasad technologii REST. Przy użyciu JAX-RS można udostępniać obiekty Javy jako usługi sieciowe typu RESTful, które są niezależne od podstawowej technologii i używają prostego API opartego na adnotacjach.

Najnowsza wersja specyfikacji to JAX-RS 2.0. Od wersji JAX-RS 1.0 różni się przede wszystkim:

- narzędziami do sprawdzania poprawności ziaren,
- obsługą API klienta,
- możliwością wykonywania wywołań asynchronicznych.

Implementacja specyfikacji JAX-RS nazywa się Jersey.

Wszystkie wymienione tematy zostały szczegółowo opisane w kolejnych rozdziałach. Przedstawiam prosty przykład kawiarni, w którym można utworzyć zasób REST o nazwie `CoffeesResource` o następujących umiejętnościach:

- podanie szczegółów złożonych zamówień,
- tworzenie nowych zamówień,
- sprawdzenie informacji o wybranym zamówieniu.

Tworzenie zasobu RESTful zaczniemy od utworzenia obiektu Javy o nazwie `CoffeesResource`. Poniżej znajduje się przykład zasobu JAX-RS:

```
@Path("v1/coffees")
public class CoffeesResource {

    @GET
    @Path("orders")
    @Produces(MediaType.APPLICATION_JSON)
    public List<Coffee> getCoffeeList( ){
        // implementacja
}
```

1. W powyższym kodzie został utworzony niewielki obiekt Javy o nazwie `CoffeesResource`. Klasę tę opatrzyłam adnotacją `@Path("v1/coffees")` określającą ścieżkę URI, dla której klasa ta obsługuje żądania.

2. Następnie definiujemy metodę o nazwie `getCoffeeList()`. Ma ona następujące adnotacje:
   - `@GET`: oznacza, że metoda reprezentuje żądanie HTTP `GET`.
   - `@PATH`: w tym przypadku żądania `GET` zasobu `v1/coffees/orders` będą obsługiwane przez metodę `getCoffeeList()`.
   - `@Produces`: definiuje typy mediów zwracane przez ten zasób. W omawianym przykładzie określono typ mediów `MediaType.APPLICATION_JSON`, którego wartość to `application/json`.
3. Inna metoda tworząca zamówienie wygląda tak:

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@ValidateOnExecution
public Response addCoffee(@Valid Coffee coffee) {
// implementacja
}
```

Jest to metoda o nazwie `addCoffee()` zawierająca następujące adnotacje:

- `@POST`: oznacza, że metoda reprezentuje żądanie HTTP `POST`.
- `@Consumes`: definiuje przyjmowane przez zasób typy mediów. W omawianym przykładzie określono typ mediów `MediaType.APPLICATION_JSON`, którego wartość to `application/json`.
- `@Produces`: definiuje typy mediów zwracane przez ten zasób. W omawianym przykładzie określono typ mediów `MediaType.APPLICATION_JSON`, którego wartość to `application/json`.
- `@ValidateOnExecution`: określa, dla których metod parametry i wartości zwrotne mają być sprawdzane. Szerzej o adnotacjach `@ValidateOnExecution` i `@Valid` piszę w rozdziale 3.

Jak widać, zmiana prostego obiektu Javy w usługę REST jest bardzo łatwa. Teraz obejrzymy podklasę klasy `Application`, która będzie zawierała definicje komponentów aplikacji JAX-RS włącznie z metadanymi.

Poniżej znajduje się kod źródłowy przykładowej podklasy klasy `Application` o nazwie `CoffeeApplication`:

```
@ApplicationPath("/")
public class CoffeeApplication extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(CoffeesResource.class);
        return classes;
    }
```

W podklasie tej została nadpisana metoda `getClasses()` oraz dodano `CoffesResource`. W pliku WAR podklasy klasy `Application` mogą znajdować się w katalogach *WEB-INF/classes* i *WEB-INF/lib*.

Wdrażanie usług typu RESTful

Następnym krokiem po utworzeniu zasobu i dodaniu metadanych do podklasy klasy `Application` jest utworzenie pliku WAR, który można wdrożyć w każdym kontenerze serwletów.

Kod źródłowy opisywanych przykładów znajduje się w plikach do pobrania z serwera FTP. Dodatkowo można w nich znaleźć szczegółowe instrukcje, jak uruchomić te przykłady.

Testowanie usług typu RESTful

Teraz możemy użyć funkcjonalności API klienta JAX-RS 2.0 w celu uzyskania dostępu do zasobów.

W tym podrozdziale opisane są następujące tematy:

- API klienta w JAX-RS 2.0,
- uzyskiwanie dostępu do zasobów RESTful przy użyciu narzędzia cURL lub rozszerzenia przeglądarki internetowej o nazwie Postman.

API klienta w JAX-RS 2.0

W JAX-RS 2.0 dodano nowe API klienckie służące do uzyskiwania dostępu do zasobów RESTful. Jego punkt początkowy to `javax.ws.rs.client.Client`.

Z tego nowego API można korzystać w następujący sposób:

```
Client client = ClientFactory.newClient();
WebTarget target = client.target("http://. . ./coffees/orders");
String response = target.request().get(String.class);
```

Jak widać w tym przykładzie, domyślny egzemplarz klienta tworzy się przy użyciu metody `ClientFactory.newClient()`. Za pomocą metody `target()` został utworzony obiekt `WebTarget`. Z wykorzystaniem obiektów tego typu przygotowuje się żądanie przez dodanie metody i parametrów zapytania.

Zanim nie pojawiły się nowe API, dostęp do zasobów REST uzyskiwało się w następujący sposób:

```
URL url = new URL("http://. . ./coffees/orders");
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setRequestMethod("GET");
conn.setDoInput(true);
conn.setDoOutput(false);
BufferedReader br = new BufferedReader(new InputStreamReader(conn.
  getInputStream()));
String line;
while ((line = br.readLine()) != null) {
  //...
}
```

Przykład ten wyraźnie pokazuje, jak dużego postępu dokonano w API klienckim JAX-RS 2.0 — wyeliminowano konieczność używania klasy `HttpURLConnection`, zamiast której można używać API `Client`.

Jeśli żądanie jest typu `POST`:

```
Client client = ClientBuilder.newClient();
Coffee coffee = new Coffee(...);
WebTarget myResource = client.target("http://foo.com/v1/coffees");
myResource.request(MediaType.APPLICATION_XML) .post(Entity.xml(coffee),
  Coffee.class);
```

metoda `WebTarget.request()` zwraca obiekt `javax.ws.rs.client.InvocationBuilder`, który za pomocą metody `post()`wywołuje żądanie HTTP `POST`. Metoda `post()` pobiera jednostkę z egzemplarza `Coffee` i określa typ mediów jako `APPLICATION_XML`.

W kliencie zostaje zarejestrowana implementacja klas `MessageBodyReader` i `MessageBodyWriter`. Szerzej na temat tych klas piszę w rozdziale 2.

W poniższej tabeli znajduje się zestawienie opisanych do tej pory najważniejszych klas i adnotacji JAX-RS.

Nazwa	Opis
`javax.ws.rs.Path`	Określa ścieżkę URI, dla której zasób serwuje metodę.
`javax.ws.rs.ApplicationPath`	Jest używana przez podklasę klasy `Application` jako bazowy URL wszystkich identyfikatorów URI dostarczanych przez zasoby w aplikacji.
`javax.ws.rs.Produces`	Definiuje typ mediów, jaki może zostać zwrócony przez dany zasób.
`javax.ws.rs.Consumes`	Definiuje typ mediów przyjmowany przez zasób.
`javax.ws.rs.client.Client`	Definiuje punkt wejściowy dla żądań klienta.
`javax.ws.rs.client.WebTarget`	Definiuje cel zasobu identyfikowany przez URI.

Klienty to ciężkie obiekty do obsługi infrastruktury komunikacyjnej po stronie klienta. Ponieważ ich tworzenie i usuwanie to dość czasochłonne operacje, powinno się tworzyć jak najmniej tych obiektów. Ponadto egzemplarz klienta zawsze trzeba poprawnie zamknąć, aby nie dopuścić do wycieku zasobów.

Uzyskiwanie dostępu do zasobów RESTful

W tym podrozdziale znajduje się opis różnych sposobów uzyskiwania dostępu do zasobów REST i testowania ich przez klienty.

cURL

cURL to popularne narzędzie wiersza poleceń do testowania API REST. Za jego pomocą użytkownik może tworzyć żądania, wysyłać je do API i analizować otrzymane odpowiedzi. Poniżej znajduje się parę przykładowych żądań `curl` wykonujących podstawowe czynności:

Żądanie curl	Opis
`curl http://api.foo.com/v1/coffees/1`	Proste żądanie `GET`.
`curl -H "foo:bar" http://api.foo.com/v1/coffees`	Żądanie z dodatkiem nagłówka za pomocą parametru `-H`.
`curl -i http://api.foo.com/v1/coffees/1`	Żądanie z wyświetleniem nagłówków odpowiedzi HTTP za pomocą parametru `-i`.
`curl –X POST -d'{"name":"JanKowalski", "username":"jkow","phone":"412-344-5644"} http://api.foo.com/v1/users`	Żądanie metodą `POST` utworzenia nowego użytkownika.

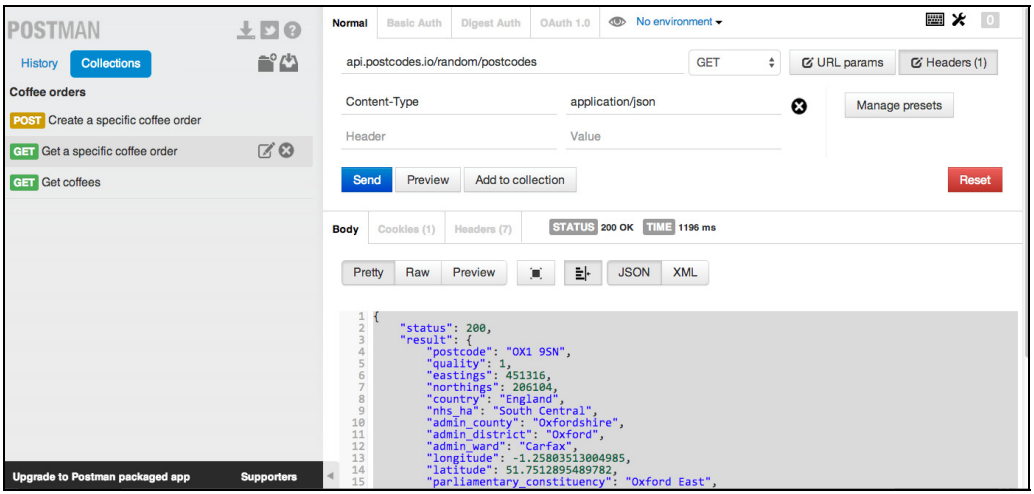
Choć narzędzie cURL jest bardzo pomocne, ma wiele opcji, które trzeba zapamiętać. Dlatego czasami lepszym rozwiązaniem jest użycie narzędzia przeglądarkowego, np. Postman albo Advanced REST client.

Postman

Postman dla przeglądarki Chrome to doskonałe narzędzie do testowania i rozwijania API REST. Zawiera przeglądarkę danych w formatach JSON i XML oraz umożliwia podglądanie żądań HTTP 1.1, jak również ich wielokrotne wysyłanie i zapisywanie na przyszłość. Postman działa w środowisku przeglądarki internetowej i umożliwia też przeglądanie danych cookie.

Zaletą narzędzia Postman w porównaniu z cURL jest przyjazny interfejs użytkownika do wprowadzania parametrów, dzięki czemu nie trzeba wpisywać całych poleceń ani skryptów. Ponadto program ten obsługuje różnego rodzaju metody uwierzytelniania, takie jak uwierzytelnianie podstawowe czy przy użyciu skrótów (tzw. *digest access authentication*).

Poniżej znajduje się zrzut ekranu przedstawiający sposób wysyłania zapytań w narzędziu Postman.



Na powyższym zrzucie ekranu przedstawiono okno aplikacji Postman. Najprostszym sposobem na przetestowanie tego programu jest uruchomienie go w przeglądarce Chrome.

Następnie należy wybrać metodę HTTP `GET` i wpisać adres URL `api.postcodes.io/random/postcodes`. (PostCodes to darmowa otwarta usługa, której działanie opiera się na danych geograficznych).

Otrzymasz odpowiedź JSON podobną do poniższej:

```
{
    "status": 200,
    "result": {
        "postcode": "OX1 9SN",
        "quality": 1,
        "eastings": 451316,
        "northings": 206104,
        "country": "England",
        "nhs_ha": "South Central",
        "admin_county": "Oxfordshire",
        "admin_district": "Oxford",
        "admin_ward": "Carfax",
...}
}
```

Po lewej stronie okna znajdują się różne zapytania, które zostały dodane do kolekcji, np. pobranie wszystkich zamówień kawy, pobranie jednego konkretnego zamówienia, utworzenie zamówień itd. na podstawie różnych przykładów z tej książki. Możesz też tworzyć własne kolekcje zapytań.

Pobieranie przykładów kodu

Pliki z przykładowym kodem źródłowym można pobrać z serwera FTP wydawnictwa Helion, pod adresem *ftp://ftp.helion.pl/przyklady/restja.zip*.

Więcej informacji na temat narzędzia Postman znajduje się na stronie *http://www.getpostman.com/*.

Inne narzędzia

Oto parę innych narzędzi, które również mogą być przydatne w pracy z zasobami REST.

Advanced REST client

Advanced REST client to kolejne rozszerzenie przeglądarki Chrome oparte na Google Web-Toolkit i służące do testowania oraz tworzenia API REST.

JSONLint

JSONLint to proste internetowe narzędzie do sprawdzania poprawności danych w formacie JSON. Gdy wysyła się dane w tym formacie, dobrze jest sprawdzić, czy są sformatowane zgodnie ze specyfikacją. Można to zrobić właśnie za pomocą narzędzia JSONLint. Więcej informacji na jego temat znajduje się na stronie *http://jsonlint.com/*.

Najlepsze praktyki projektowania zasobów

W tym podrozdziale znajduje się opis niektórych najlepszych praktyk projektowania zasobów RESTful:

- Programista API powinien używać rzeczowników, aby ułatwić użytkownikowi poruszanie się po zasobach, a czasowników tylko jako metod HTTP. Na przykład URI `/user/1234/books` jest lepszy niż `/user/1234/getBook`.
- Do identyfikacji podzasobów używaj asocjacji. Na przykład aby pobrać autorów książki 5678 dla użytkownika 1234, powinno się użyć URI `/user/1234/books/5678/authors`.
- Do pobierania specyficznych wariacji używaj parametrów zapytań. Na przykład aby pobrać wszystkie książki mające 10 recenzji, użyj URI `/user/1234/books?reviews_counts=10`.
- W ramach parametrów zapytań w razie możliwości zezwalaj na częściowe odpowiedzi. Przykładem może być pobranie tylko nazwy i wieku użytkownika. Klient może wysłać w URI parametr zapytania `?fields` zawierający listę pól, które chce otrzymać od serwera, np. `/users/1234?fields=name,age`.

- Zdefiniuj domyślny format odpowiedzi na wypadek, gdyby klient nie podał, jaki format go interesuje. Większość programistów jako domyślnego formatu używa JSON.
- W nazwach atrybutów stosuj `notacjęWielbłądzią` lub ze znakami podkreślenia _.
- Dla kolekcji zapewnij standardowe API liczące, np. `users/1234/books/count`, aby klient mógł sprawdzić, ilu obiektów może się spodziewać w odpowiedzi.

  Będzie to też pomocne dla klientów używających stronicowania. Szerzej na temat stronicowania piszę w rozdziale 5.
- Zapewnij opcję eleganckiego drukowania — `users/1234?pretty_print`. Ponadto nie powinno się buforować zapytań z parametrem drukowania.
- Staraj się minimalizować komunikację przez dostarczenie jak najpełniejszych informacji w pierwszej odpowiedzi. Chodzi o to, że jeśli serwer nie dostarczy wystarczającej ilości danych w odpowiedzi, klient będzie musiał wysłać kolejne żądania, aby zdobyć potrzebne mu informacje. W ten sposób marnuje się zasoby i wyczerpuje limit żądań klienta. Szerzej na temat ograniczania liczby żądań klienta piszę w rozdziale 5.

Zalecana lektura

- RFC 2616: *http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html*.
- Model dojrzałości Richardsona: *http://www.crummy.com/writing/speaking/2008-QCon/act3.html*.
- Implementacja JAX-RS Jersey: *https://jersey.java.net/*.
- InspectB.in: *http://inspectb.in/*.
- Postman: *http://www.getpostman.com/*.
- Advanced REST Client: *https://code.google.com/p/chrome-rest-client/*.

Podsumowanie

W tym rozdziale przedstawiłam podstawowe założenia technologii REST, opisałam API CRUD oraz pokazałam, jak projektować zasoby RESTful. W przykładach użyte zostały adnotacje JAX-RS 2.0, za pomocą których można reprezentować metody HTTP, i API klienckie, przy użyciu których można odnosić się do zasobów. Ponadto zrobiłam przegląd najlepszych praktyk projektowania usług typu RESTful.

W następnym rozdziale znajduje się rozszerzenie tych wiadomości. Bardziej szczegółowo poznasz zasady negocjowania treści, dostawców jednostek w JAX-RS 2.0, techniki obsługi błędów, sposoby kontrolowania wersji oraz kody odpowiedzi REST. Ponadto dowiesz się, w jaki sposób serwer może wysyłać do klienta odpowiedzi przy użyciu strumieniowania i kawałkowania.

ROZDZIAŁ 2

Projektowanie zasobów

W rozdziale 1. zostały opisane podstawy technologii REST i najlepsze praktyki projektowania zasobów typu RESTful. W tym rozdziale rozszerzymy tę wiedzę o rodzaje odpowiedzi na żądania, sposoby obsługi różnych reprezentacji zasobów, strategie wersjonowania API oraz standardowe kody HTTP zwracane w odpowiedziach na żądania. W poszczególnych podrozdziałach omawiane są następujące tematy:

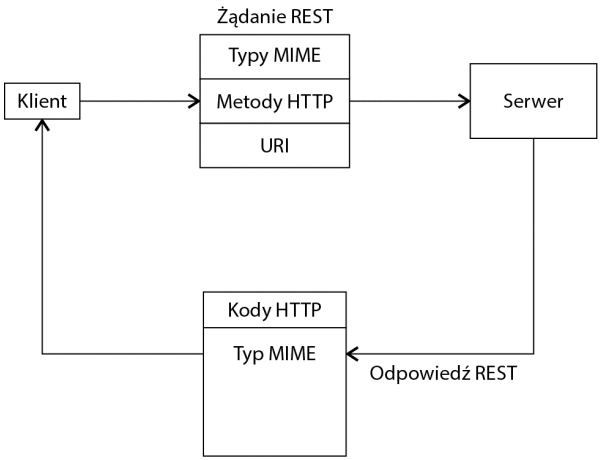
- Rodzaje odpowiedzi REST.
- Negocjacja treści.
- Dostawcy jednostek i różne reprezentacje.
- Wersjonowanie API.
- Kody odpowiedzi i wzorce REST.

Ponadto nauczysz się tworzyć własnych dostawców jednostek do serializowania i deserializowania żądań i odpowiedzi oraz poznasz inne techniki, takie jak strumieniowanie i kawałkowanie.

Rodzaje odpowiedzi REST

W poprzednim rozdziale pokazałam, jak przy użyciu danych domenowych tworzyć czytelne identyfikatory URI, używać metod HTTP do tworzenia, odczytywania, aktualizowania i usuwania zasobów oraz przesyłać dane między klientem i serwerem z wykorzystaniem typów MIME i kodów odpowiedzi HTTP.

Na poniższym schemacie przedstawiony jest standardowy model żądań i odpowiedzi REST:



Jak widać na tym rysunku, klient wysyła za pomocą jednej ze standardowych metod HTTP żądanie REST zawierające definicje typów MIME i docelowy URI. Serwer przetwarza to żądanie i odsyła odpowiedź zawierającą standardowy kod HTTP i typy MIME. Metody HTTP i adnotacje JAX-RS zostały opisane wcześniej. Ponadto omówiłam najlepsze praktyki projektowania identyfikatorów URI zasobów. W tym rozdziale opisuję najczęściej używane kody odpowiedzi HTTP oraz sposoby obsługi różnych typów MIME.

Negocjacja treści

Negocjacja treści to dostarczenie różnych reprezentacji zasobu pod jednym URI, aby klient mógł wybrać tę, która mu najbardziej odpowiada.

> *HTTP zapewnia kilka mechanizmów negocjacji treści, która polega na wyborze najlepszej reprezentacji w danym przypadku, gdy do wyboru jest kilka reprezentacji.*
>
> *— RFC 2616, Fielding i in.*

Są dwie następujące techniki negocjacji treści:

- przy użyciu nagłówków HTTP,
- przy użyciu adresów URL.

Negocjacja treści przy użyciu nagłówków HTTP

Gdy klient wysyła żądania utworzenia lub zaktualizowania zasobu, to przesyła pewne informacje do serwera. Analogicznie serwer wraz z odpowiedzią również może przesłać do klienta jakieś dane. Przesyłanie tych informacji jest obsługiwane przez jednostki HTTP, które są wysyłane jako część wiadomości HTTP.

Jednostki są wysyłane poprzez żądania, najczęściej metodą `POST` lub `PUT`, lub odbierane w odpowiedzi. Do określania typu MIME przesyłanej przez serwer jednostki służy nagłówek HTTP `Content-Type`. Wśród najczęściej używanych typów treści znajdują się `"text/plain"`, `"appli➥cation/xml"`, `"text/html"`, `"application/json"`, `"image/gif"` oraz `"image/jpeg"`.

Klient może wysłać żądanie do serwera i określić, jakie typy mediów obsługuje w kolejności od najbardziej pożądanego w nagłówku HTTP `Accept`. Ponadto za pomocą nagłówka `Accept-Language` klient może określić, w jakim języku chciałby otrzymać odpowiedź. Jeżeli w żądaniu nie ma nagłówka `Accept`, serwer może wybrać, jaką reprezentację wysłać.

Specyfikacja JAX-RS zawiera standardowe adnotacje dotyczące negocjacji treści: `javax.ws.rs.➥Produces` i `javax.ws.rs.Consumes`. Poniżej znajduje się przykład użycia adnotacji `@Produces` w metodzie do obsługi zasobów:

```
@GET
@Path("orders")
@Produces(MediaType.APPLICATION_JSON)
public List<Coffee> getCoffeeList(){
    return CoffeeService.getCoffeeList();
}
```

Metoda `getCoffeeList()` zwraca listę kaw i ma adnotację `@Produces(MediaType.APPLICATION_JSON)`. Adnotacja ta służy do określania, jakie typy MIME zasób może zwracać do klienta, i powinna odpowiadać zawartości nagłówka `Accept`.

Powyższa metoda utworzy następującą odpowiedź:

```
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source
Edition 4.0 Java/Oracle Corporation/1.7)
Server: GlassFish Server Open Source Edition 4.0
Content-Type: application/json
Date: Thu, 31 Jul 2014 15:25:17 GMT
Content-Length: 268
{
    "coffees": [
        {
            "Id": 10,
            "Name": "Cappuccino",
            "Price": 3.82,
            "Type": "Iced",
            "Size": "Medium"
        },
        {
            "Id": 11,
            "Name": "Americano",
            "Price": 3.42,
            "Type": "Brewed",
            "Size": "Large"
        }
    ]
}
```

Jeśli w zasobie nie ma żadnej metody mogącej zwrócić typ MIME zażądany przez klienta, system wykonawczy JAX-RS wysyła błąd HTTP `406 Not Acceptable`.

Poniżej znajduje się metoda z adnotacją `@Consumes`:

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response addCoffee(Coffee coffee) {
    // implementacja
}
```

Adnotacja `@Consumes` określa typy mediów akceptowane przez zasób. Gdy klient wyśle żądanie, JAX-RS wyszukuje wszystkie metody pasujące do ścieżki i wywołuje jedną z nich na podstawie typu treści wysłanej przez tego klienta.

Jeżeli zasób nie potrafi zinterpretować typu MIME zażądanego przez klienta, JAX-RS zwraca błąd HTTP `415 Unsupported Media Type`.

W adnotacjach `@Produces` i `@Consumes` można określać po kilka typów MIME, np. `@Produces (MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML)`.

Oprócz statycznej negocjacji treści JAX-RS obsługuje też negocjację treści w czasie wykonywania poprzez klasę `javax.ws.rs.core.Variant` i obiekty `javax.ws.rs.core.Request`. Obiekt klasy `Variant` w specyfikacji JAX-RS jest kombinacją typów mediów, języka treści i kodowania treści, jak również znaczników `ETag`, nagłówków określających datę modyfikacji itp. Obiekt ten definiuje reprezentację zasobu obsługiwaną przez serwer. Klasa `Variant.VariantListBuilder` służy do tworzenia listy wariantów reprezentacji.

Poniżej znajduje się przykład utworzenia listy wariantów reprezentacji zasobu:

```
List<Variant> variants = Variant.mediatypes("application/xml",
  "application/json").build();
```

W kodzie tym znajduje się wywołanie metody `build()` klasy `VariantListBuilder`. Metoda `Request.selectVariant()` pobiera listę obiektów klasy `Variant` i wybiera jeden z nich na podstawie nagłówka `Accept` wysłanego przez klienta, jak pokazano poniżej:

```
@GET
public Response getCoffee(@Context Request r) {
    List<Variant> vs = ...;
    Variant v = r.selectVariant(vs);
    if (v == null) {
        return Response.notAcceptable(vs).build();
    } else {
        Coffee coffee = .. // wybór reprezentacji na podstawie v
        return Response.ok(coffee, v);
    }
}
```

Negocjacja treści poprzez adres URL

Inną techniką negocjacji treści stosowaną w niektórych API jest wysyłanie reprezentacji zasobu na podstawie rozszerzenia przekazanego w adresie URL. Na przykład klient może zażądać szczegółów za pomocą adresu *http://foo.api.com/v2/library/books.xml* lub *http://foo.api.com/v2/library/books.json*. Na serwerze do obsługi każdego z tych adresów służy inna metoda. Mimo to oba są reprezentacjami tego samego zasobu.

```
@Path("/v1/books/")
public class BookResource {
@Path("{resourceID}.xml")
@GET
public Response getBookInXML(@PathParam("resourceID") String resourceID) {
    // zwraca odpowiedź w formacie XML
        }

@Path("{resourceID}.json")
@GET
public Response getBookInJSON(@PathParam("resourceID") String resourceID) {
    // zwraca odpowiedź w formacie JSON
        }
}
```

Jak widać, w powyższym kodzie znajdują się definicje dwóch metod: `getBookInXML()` i `getBook ↪InJSON()`, a rodzaj odpowiedzi zależy od ścieżki URL.

> Dobrą praktyką jest używanie do negocjacji treści nagłówka HTTP `Accept`, ponieważ w ten sposób oddziela się warstwę biznesową aplikacji od jej warstwy technicznej. Inną zaletą tego nagłówka jest to, że istnieje tylko jedna metoda dla wszystkich reprezentacji.

W poniższym podrozdziale znajduje się opis technik serializacji i deserializacji zasobów między różnymi reprezentacjami przy użyciu dostawców jednostek z JAX-RS.

Dostawcy jednostek i różne reprezentacje

W poprzednich przykładach przekazywaliśmy parametry do metody obsługującej zasób ze ścieżki URI i parametrów zapytania żądania. Ale czasami trzeba przesłać dane w treści żądania, np. typu `POST`. W JAX-RS dostępne są dwa interfejsy, które to umożliwiają: jeden do obsługi przychodzącej reprezentacji jednostki o nazwie `javax.ws.rs.ext.MessageBodyReader`, a drugi do obsługi wychodzącej reprezentacji jednostki o nazwie `javax.ws.rs.ext.MessageBodyWriter`.

`MessageBodyReader` deserializuje jednostki z reprezentacji treści wiadomości do postaci klas Javy. `MessageBodyWriter` natomiast serializuje klasy Java na określony format reprezentacyjny.

W poniższej tabeli znajduje się opis metod, które należy zaimplementować:

Metoda interfejsu MessageBodyReader	Opis
`isReadable()`	Metoda ta służy do sprawdzania, czy klasa `MessageBodyReader` obsługuje konwersję ze strumienia na typ Javy.
`readFrom()`	Metoda służąca do odczytu typu z klasy `InputStream`.

Jak widać, metoda `isReadable()` klasy implementującej interfejs `MessageBodyReader` jest wywoływana w celu sprawdzenia, czy klasa ta może obsłużyć określone dane. Z kolei metoda `readFrom()` konwertuje dane ze strumienia wejściowego na zwykły obiekt Javy.

Poniższa tabela zawiera zestawienie obowiązkowych metod interfejsu `MessageBodyWriter` wraz ze zwięzłymi opisami ich zastosowania.

Metoda interfejsu MessageBodyWriter	Opis
`isWritable()`	Metoda ta służy do sprawdzania, czy klasa implementująca interfejs `MessageBodyWriter` obsługuje konwersję z określonego typu Javy.
`getSize()`	Metoda służąca do sprawdzania długości w bajtach i zwracająca –1, jeśli nie da się określić rozmiaru z góry.
`writeTo()`	Zapisuje typ do strumienia.

Metoda `isWriteable()` klasy implementacyjnej interfejsu `MessageBodyWriter` służy do sprawdzania, czy klasa ta jest w stanie obsłużyć określony rodzaj danych wejściowych. Metoda `writeTo()` konwertuje obiekt Javy na strumień wyjściowy. Przykłady użycia metod klas `MessageBodyReader` i `MessageBodyWriter` znajdują się w plikach z przykładowym kodem źródłowym do pobrania z serwera FTP.

Istnieją też lekkie implementacje, takie jak klasy implementujące interfejsy `StreamingOutput` i `ChunkedOutput`. W kolejnych podrozdziałach znajduje się opis obsługi podstawowych formatów, takich jak tekst, JSON i XML, przez implementację JAX-RS Jersey.

StreamingOutput

Klasa `javax.ws.rs.core.StreamingOutput` to wywołanie zwrotne, które można zaimplementować w taki sposób, aby wysyłać jednostki w odpowiedzi, gdy aplikacja zażąda strumieniowania danych wyjściowych. Klasa `StreamingOutput` jest lekką alternatywą dla klasy `javax.ws.rs.ext.MessageBodyWriter`.

Poniżej znajduje się przykład przedstawiający sposób użycia klasy `StreamingOutput` jako części odpowiedzi:

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("/orders/{id}")
public Response streamExample(@PathParam("id") int id) {
    final Coffee coffee = CoffeeService.getCoffee(id);
    StreamingOutput stream = new StreamingOutput() {
        @Override
        public void write(OutputStream os) throws IOException,
                WebApplicationException {
            Writer writer = new BufferedWriter(new OutputStreamWriter(os));
            writer.write(coffee.toString());
            writer.flush();
        }
    };
    return Response.ok(stream).build();
}
```

Jak widać w tym przykładzie, metoda `write()` klasy `StreamingOutput` została przesłonięta tak, aby zapisywała dane do strumienia wyjściowego. Klasa `StreamingOutput` jest pomocna, gdy trzeba strumieniowo przesyłać dane binarne. Więcej informacji znajduje się w plikach z kodem źródłowym dostępnych na serwerze FTP.

ChunkedOutput

W implementacji Jersey JAX-RS serwer może za pomocą klasy `org.glassfish.jersey.server.ChunkedOutput` wysyłać klientom odpowiedzi w kawałkach natychmiast, gdy staną się dostępne, nie czekając na pozostałe kawałki. W celu poinformowania, że odpowiedź nadejdzie w kawałkach, w nagłówku `Content-Length` odpowiedzi zostaje wysłana wartość `-1` obiektu `size`. Klient będzie wiedział, że otrzyma pokawałkowaną odpowiedź, więc każdy z kawałków wczyta i przetworzy oddzielnie. Serwer będzie wysyłał po jednym kawałku, a gdy wyśle wszystkie, zamknie połączenie, co będzie oznaczało koniec przetwarzania odpowiedzi.

Poniżej znajduje się przykładowy kod ilustrujący sposób użycia klasy `ChunkedOutput`:

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("/orders/{id}/chunk")
public ChunkedOutput<String> chunkExample(final @PathParam("id")
int id) {
        final ChunkedOutput<String> output = new
                ChunkedOutput<String>(String.class);

        new Thread() {
            @Override
            public void run() {
                try {
                    output.write("foo");
                    output.write("bar");
```

```
                    output.write("test");
                } catch (IOException e) {
                  e.printStackTrace();
                } finally {
                    try {
                        output.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }.start();
        return output;

    }
}
```

Jak widać w tym przykładzie, metoda `chunkExample` zwraca obiekt `ChunkedOutput`.

Po stronie klienta klasy `org.glassfish.jersey.client.ChunkedInput` można używać do odbierania wiadomości w kawałkach o określonym typie. Za pomocą tej klasy można konsumować częściowe odpowiedzi z dużych lub ciągłych strumieni danych wejściowych. Poniżej znajduje się przykład wczytywania danych przez klienta z klasy `ChunkedInput`:

```
ChunkedInput<String> input = target().path("..").request().get(new
  GenericType<ChunkedInput<String>>() {
        });
while ((chunk = chunkedInput.read()) != null) {
    // robi coś
}
```

Różnice między klasami ChunkedOutput i StreamingOutput

`ChunkedOutput` to wewnętrzna klasa Jersey umożliwiająca serwerowi wysyłanie **kawałków** danych bez zamykania połączenia z klientem. Wykorzystuje szereg wygodnych wywołań metod `ChunkedOutput.write()`, które pobierają obiekt Javy i typ mediów, a następnie przy użyciu klasy JAX-RS `MessageBodyWriter` konwertują te obiekty na bajty. Operacje zapisu klasy `ChunkedOutput` nie mają charakteru blokującego.

Klasa `StreamingOutput` to niskopoziomowe API JAX-RS pracujące bezpośrednio na bajtach. Serwer musi implementować klasę `StreamingOutput`, a jej metoda `write(OutputStream)` jest wywoływana tylko raz przez system wykonawczy JAX-RS i wywołanie to ma charakter blokujący.

Jersey i JSON

W Jersey z reprezentacjami danych w formacie JSON można pracować na następujące sposoby.

Powiązanie danych JSON oparte na POJO

Obsługa powiązania danych JSON oparta na POJO jest bardzo ogólna i umożliwia mapowanie obiektów Javy na format JSON. W tym procesie wykorzystywany jest egzemplarz biblioteki Jackson `org.codehaus.jackson.map.ObjectMapper`. Aby na przykład wczytać dane w formacie JSON do obiektu `Coffee`, należy użyć następującego kodu:

```
ObjectMapper objectMapper = new ObjectMapper();
Coffee coffee = objectMapper.readValue(jsonData, Coffee.class);
```

Więcej informacji można znaleźć na stronie *https://jersey.java.net/documentation/1.18/json.html*.

Powiązanie danych JSON oparte na JAXB

Obsługa powiązania danych JSON oparta na JAXB jest przydatna, gdy zasób może tworzyć i konsumować dane w formacie XML lub JSON. Implementacja polega na dodaniu do prostego obiektu Javy adnotacji `@XMLRootElement`, jak pokazano w poniższym przykładzie:

```
@XMLRootElement
public class Coffee {
    private String type;
    private String size;
}
```

Utworzenie danych w formacie JSON z metody zasobu przy użyciu powyższego ziarna JAXB jest teraz bardzo proste:

```
@GET
@Produces("application/json")
public Coffee getCoffee() {
    // implementacja
}
```

Adnotacja `Produces` spowoduje konwersję reprezentacji zasobu `Coffee` na format JSON.

Niskopoziomowe przetwarzanie danych JSON

Ta metoda jest przydatna, gdy chce się dokładnie kontrolować format JSON przy użyciu klas `JSONArray` i `JSONObject` służących do tworzenia reprezentacji JSON. Zaletą tej techniki jest to, że programista ma pełną kontrolę nad tworzonym i konsumowanym formatem JSON. Poniżej znajduje się przykład użycia klasy `JSONArray`:

```
JsonObject myObject = Json.createObjectBuilder()
        .add("name", "Mocha")
        .add("size", "Large")
        .build();
```

Z drugiej strony praca z obiektami modelu danych jest z reguły trochę bardziej skomplikowana. Na przykład poniższy kod ilustruje sposób zastosowania techniki przetwarzania strumieniowego przy użyciu klasy `JSONParser`:

```
JsonParser parser = Json.createParser(…)
Event event = parser.next(); // START_OBJECT
event = parser.next();       // END_OBJECT
```

W następnym podrozdziale znajduje się opis metod wersjonowania API zgodnie z tokiem jego ewolucji oraz w taki sposób, aby zmiany wprowadzane na serwerze nie powodowały uszkodzenia podstawowej funkcjonalności aplikacji klienta.

Wersjonowanie API

Ze względu na ewolucję aplikacji projekt URI powinien mieć pewne ograniczenia dotyczące identyfikowania wersji. Trudno jest przewidzieć wszystkie zasoby, które mogą zmieniać się w czasie cyklu istnienia aplikacji. Celem wersjonowania API jest zdefiniowanie punktów końcowych zasobów i sposobów adresowania oraz powiązanie z nimi różnych wersji. Programista powinien zadbać o to, aby zmiany wersji nie miały wpływu na semantykę czasowników HTTP i kody statusu. Wersje aplikacji ewoluują i niektóre API mogą zostać wycofane. Żądania wysyłane do starszych wersji API można przekierowywać do najnowszych ścieżek lub można zwracać odpowiednie kody błędów informujące o tym, że API jest przestarzałe.

Problem określania różnych wersji API można rozwiązać na kilka sposobów. Oto one:

- Określenie wersji w identyfikatorze URI.
- Określenie wersji w parametrze zapytaniowym żądania.
- Określenie wersji w nagłówku `Accept`.

Wszystkie te metody są dobre. W następnych paru podrozdziałach znajduje się ich bardziej szczegółowy opis ze wskazaniem największych zalet i wad.

Określanie wersji w identyfikatorze URI

W tej technice numer wersji jest częścią identyfikatora URI zasobu udostępnianego przez serwer.

Na przykład w poniższym adresie URL znajduje się numer wersji *v2*:

```
http://api.foo.com/v2/coffees/1234
```

Ponadto programiści API mogą dostarczyć ścieżkę domyślnie kierującą do najnowszej wersji API. W takim przypadku poniższe dwa URI powinny zachowywać się identycznie:

- *http://api.foo.com/coffees/1234*
- *http://api.foo.com/v2/coffees/1234*

Z tego wynika, że najnowsza wersja API ma numer *v2*. Jeśli klient skieruje żądanie do starszej wersji, to powinien zostać poinformowany, że istnieje nowsza wersja, za pomocą jednego z następujących kodów HTTP oznaczających przekierowanie:

- `301 Moved permanently` — ten kod oznacza, że zasób o podanym URI został przeniesiony na stałe pod inny URI. Za jego pomocą można informować, że dana wersja API jest przestarzała lub nieobsługiwana i że należy skorzystać z innego URI.
- `302 Found` — ten kod oznacza, że żądany zasób jest tymczasowo przeniesiony w inne miejsce, ale URI, pod który wysłano żądanie, nadal jest obsługiwany.

Numer wersji w parametrze zapytaniowym żądania

Innym sposobem na określanie wersji API jest wysyłanie numeru w parametrze żądania. Na podstawie tej wartości metoda zasobu może wybrać odpowiednią gałąź kodu. Na przykład w adresie URL *http://api.foo.com/coffees/1234?version=v2* znajduje się parametr `?version` o wartości *v2*.

Wadą tego rozwiązania jest brak możliwości buforowania odpowiedzi. Ponadto kod źródłowy implementacji zasobu musi mieć różne rozgałęzienia dla poszczególnych wartości parametru zapytaniowego, co nie jest zbyt intuicyjnym ani łatwym w obsłudze rozwiązaniem.

> Więcej informacji na temat najlepszych technik buforowania znajduje się w rozdziale 4.

Z drugiej strony, jeśli URI zawiera numer wersji, jest to bardziej przejrzyste i czytelne. Ponadto wersja URI może mieć standardowo określony czas ważności, po upływie którego wszystkie żądania do starszych wersji będą przekierowywane do najnowszej wersji.

> W API portali Facebook, Twitter i Stripe numer wersji wchodzi w skład URI. W Facebooku okres ważności wersji wynosi dwa lata i po jego upływie następuje wydanie nowej wersji. Jeżeli klient wyśle żądanie bez określenia numeru wersji, serwer automatycznie użyje najnowszej dostępnej wersji API.
>
> W API Twittera stosowany jest sześciomiesięczny cykl przemiany od wersji 1.0 do 1.1.
>
> Szerzej na temat tych API piszę jeszcze w „Dodatku A".

Określanie numeru wersji w nagłówku Accept

W niektórych API numer wersji jest przekazywany w nagłówku `Accept`. Spójrz na przykład na poniższy fragment kodu:

```
Accept: application/vnd.foo-v1+json
```

W kodzie tym `vnd` to specjalny typ MIME zdefiniowany przez programistę. W ten sposób eliminuje się konieczność ukazywania numeru wersji w adresie URL i jest to metoda preferowana przez niektórych programistów.

> W API portalu GitHub zalecane jest wysyłanie nagłówka `Accept`, jak pokazano poniżej:
>
> ```
> Accept: application/vnd.github.v3+json
> ```
>
> Więcej informacji na ten temat znajduje się na stronie *https://developer.github.com/v3/media/*.

W następnym podrozdziale zamieszczam opis standardowych kodów odpowiedzi protokołu HTTP, które powinny być wysyłane do klienta w różnych sytuacjach.

Kody odpowiedzi i wzorce REST

W protokole HTTP zdefiniowano standardowy zestaw kodów odpowiedzi, które mogą być zwracane w reakcji na każde żądanie. W poniższej tabeli znajduje się zestawienie wzorców odpowiedzi REST bazujących na API CRUD. W zależności od operacji i tego, czy treść jest wysyłana jako część odpowiedzi, czy nie, mogą występować pewne drobne różnice.

Grupa	Kod odpowiedzi	Opis
Powodzenie — *2XX*	`200 OK`	Ten kod może zostać zwrócony w odpowiedzi na żądania utworzenia, aktualizacji lub usunięcia zasobu metodami `PUT`, `POST` lub `DELETE`. Jako część takiej odpowiedzi zwracana jest treść.
	`201 Created`	Ten kod może zostać zwrócony po utworzeniu zasobu metodą `PUT`. Musi zawierać nagłówek `Location` zasobu.
	`204 No Content`	Ten kod może zostać zwrócony po wykonaniu operacji `DELETE`, `POST` lub `PUT`. W odpowiedzi nie jest zwracana żadna treść.
	`202 Accepted`	Kod ten oznacza, że odpowiedź zostanie wysłana później, ponieważ przetwarzanie jeszcze się nie zakończyło. Jest on używany w przypadku operacji asynchronicznych. Powinno się dodatkowo zwracać nagłówek `Location` określający, gdzie klient może spodziewać się odpowiedzi.
Przekierowanie — *3XX*	`301 Permanent`	Kod ten służy do przekierowywania wszystkich żądań do nowej lokalizacji.
	`302 Found`	Kod ten może wskazywać, że zasób już istnieje i jest poprawny.
Błędy klienta — *4XX*	`401 Unauthorized`	Kod oznaczający, że nie można przetworzyć żądania z powodu niepoprawnych danych poświadczających.

Grupa	Kod odpowiedzi	Opis
Błędy klienta — 4*XX*	`404 Not Found`	Ten kod oznacza, że nie znaleziono zasobu. Dobrym zwyczajem jest wysyłanie tego kodu także wtedy, gdy wysyłający żądanie użytkownik nie ma uprawnień dostępu do zasobu, aby nie ujawniać informacji.
	`406 Not Acceptable`	Kod oznaczający, że zasób nie może wyprodukować typu MIME żądanego przez klienta. Sytuacja taka ma miejsce, gdy typ MIME określony w nagłówku `Accept` nie odpowiada żadnemu z typów mediów w metodzie zasobu lub klasie oznaczonej adnotacją `@Produces`.
	`415 Unsupported Media Type`	Kod ten może zostać zwrócony, gdy klient wyśle typ mediów nierozpoznawany przez zasób. Sytuacja taka ma miejsce, gdy typ MIME podany w nagłówku `Content-Type` nie pasuje do typu mediów w metodzie zasobu lub klasie oznaczonej adnotacją `@Consumes`.
Błędy serwera — 5*XX*	`500 Internal Server error`	Kod oznaczający wewnętrzny błąd serwera. Jest to ogólna informacja zwracana, gdy brak jest jakichkolwiek szczegółów.
	`503 Service Unavailable`	Ten kod może zostać zwrócony, gdy trwają prace przy obsłudze serwera lub serwer jest zbyt zajęty, aby obsługiwać żądania.

JAX-RS definiuje klasę `javax.ws.rs.core.Response` zawierającą metody statyczne do tworzenia egzemplarzy przy użyciu klasy `javax.ws.rs.core.Response.ResponseBuilder`:

```
@POST
 Response addCoffee(...) {
   Coffee coffee = ...
   URI coffeeId = UriBuilder.fromResource(Coffee.class)...
   return Response.created(coffeeId).build();
}
```

W tym kodzie została użyta metoda `addCoffee()`, która zwraca kod odpowiedzi `201 Created` przy użyciu metody `Response.created()`. Więcej informacji o metodach odpowiedzi znajduje się na stronie *https://jersey.java.net/apidocs/latest/jersey/javax/ws/rs/core/Response.html*.

Zalecana lektura

- *https://jersey.java.net/documentation/latest/representations.html*: dokumentacja negocjacji treści w Jersey.
- *http://docs.jboss.org/resteasy/docs/2.2.1.GA/userguide/html/JAX-RS_Content_Negotiation.html*: negocjacja treści przy użyciu RESTEasy i adresów URL.

- *https://dev.twitter.com/docs/api/1.1/overview*: opis API REST Twittera i sposobu określania numerów wersji.
- *https://developers.facebook.com/docs/apps/versions*: API i wersjonowanie Facebooka.

Podsumowanie

W tym rozdziale zostały opisane takie tematy jak negocjacja treści, wersjonowanie API oraz kody odpowiedzi REST. Po jego lekturze przede wszystkim należy zapamiętać, jak ważne jest obsługiwanie różnych reprezentacji jednego zasobu, aby klient mógł wybrać najodpowiedniejszą dla siebie w określonym przypadku. Opisałam, czym różni się strumieniowanie od kawałkowania danych wyjściowych oraz jak można wykorzystać te nieskomplikowane techniki w połączeniu z własnymi dostawcami jednostek, takimi jak np. `MessageBodyReader` i `MessageBodyWriter`. Przedstawiłam przykłady firm stosujących wersjonowanie w swoich rozwiązaniach oraz najlepsze praktyki i zasady projektowania.

W następnym rozdziale poznasz zaawansowane techniki dotyczące bezpieczeństwa, wykrywalności i weryfikacji poprawności danych w modelach programowania REST.

ROZDZIAŁ 3

Bezpieczeństwo i wykrywalność

W czasach otwartych platform programiści mogą budować aplikacje, które bardzo łatwo odłączyć od cyklu biznesowego pierwotnej platformy. Taka oparta na interfejsach API architektura umożliwia programowanie zwinne, ułatwia adaptację, pozwala na rozpowszechnianie rozwiązań oraz umożliwia skalowanie i integrację aplikacji zarówno z aplikacjami wewnętrznymi firmy, jak i zewnętrznymi. Jedną z najważniejszych kwestii do rozpatrzenia jest bezpieczeństwo. Programista nie powinien przejmować się danymi poświadczającymi użytkowników. Ponadto z usług REST mogą korzystać także inne klienty, między innymi przeglądarki internetowe i aplikacje mobilne. Klient może działać w imieniu innych użytkowników i musi być uwierzytelniony, aby móc wykonywać za nich różne działania bez przekazywania mu danych poświadczających, takich jak nazwa użytkownika i hasło. Do tego potrzebny jest protokół OAuth 2.0.

Innym ważnym aspektem, który należy rozważyć podczas budowy aplikacji rozproszonych, jest wykrywalność. Należy rejestrować dane związane z żądaniami w celach diagnostycznych w środowisku obejmującym wiele mikrousług, które mogą być geograficznie rozproszone i mają obsługiwać tysiące żądań. Żądania wysyłane do zasobów REST i kody statusu należy rejestrować po to, by móc je wykorzystać w diagnozowaniu problemów w produkcji oraz do audytów. W tym rozdziale opisane są zaawansowane techniki zabezpieczeń i wykrywalności w modelach programowania REST. Oto lista opisywanych tematów:

- Rejestrowanie informacji w API REST.
- Obsługa wyjątków w usługach typu RESTful.
- Metody weryfikacji poprawności danych.
- Federacyjne zarządzanie tożsamością:

- SAML 2.0,
- OAuth 2.0,
- OpenID Connect.

Na końcu rozdziału dowiesz się, jak budować skalowalne i wydajne usługi typu RESTful.

Rejestrowanie informacji w API REST

W skomplikowanych aplikacjach rozproszonych jest wiele możliwości wystąpienia awarii. Ponadto trudno jest wykryć i naprawić błędy, co wydłuża czas reakcji na zdarzenia i może prowadzić do kosztownej eskalacji problemu. Programiści i administratorzy mogą nie mieć bezpośredniego dostępu do danych z tego komputera, z którego potrzebują.

Rejestracja danych jest bardzo ważnym aspektem budowy usług typu RESTful, zwłaszcza gdy przyjdzie do diagnozowania problemów w środowisku produkcyjnym na rozproszonych węzłach reprezentujących różne mikrousługi. Dobrym pomysłem jest połączenie zdarzeń lub transakcji między różnymi komponentami tworzącymi aplikację lub usługę biznesową. Kompletna linia dzienników może być bardzo pomocna w odtworzeniu sekwencji zdarzeń, które miały miejsce w systemie. Ponadto dzienniki mogą pomóc w indeksowaniu i dzieleniu danych oraz analizowaniu wzorców przychodzących żądań, jak również dostarczyć mnóstwo potencjalnie przydatnych informacji.

Poniżej znajduje się przykład prostego filtra rejestrującego, który można zintegrować z zasobami REST. Filtr ten rejestruje dane dotyczące żądań — znacznik czasu, łańcuch zapytania i dane wejściowe:

```
@WebFilter(filterName = "LoggingFilter",
          urlPatterns = {"/*"}
)
public class LoggingFilter implements Filter {
    static final Logger logger = Logger.getLogger(LoggingFilter.class);
    @Override
    public void doFilter(ServletRequest servletRequest,
      ServletResponse servletResponse,
            FilterChain filterChain) throws IOException,
              ServletException {

        HttpServletRequest httpServletRequest = (HttpServletRequest)
          servletRequest;

logger.info("request"  +httpServletRequest.getPathInfo().toString());
        filterChain.doFilter(servletRequest, servletResponse);
}
```

Klasa `LoggingFilter` jest prostym filtrem implementującym interfejs `javax.servlet.Filter`. Rejestrator ten rejestruje wszystkie wiadomości zawierające ścieżkę żądania i dane wejściowe. W przedstawionym przykładzie do budowy rejestratora użyto biblioteki Apache Log4j.

> Więcej informacji o Apache Log4j znajduje się na stronie *http://logging.apache.org/log4j/2.x/*.

Dzienniki można pobrać z rozproszonej serwerowej aplikacji dziennikowej, np. Splunk (*http://www.splunk.com/*). Programista może w nich znaleźć informacje dotyczące przyczyn awarii lub spowolnienia wydajności aplikacji. Przykładem w odniesieniu do naszej kawiarni może być problem dotyczący przetwarzania zamówień kawy. Jeśli zarejestrujemy szczegółowe informacje o żądaniu w rozproszonej aplikacji serwerowej, takiej jak Splunk, to będziemy mogli sprawdzić dane z określonego czasu, aby dowiedzieć się, co dokładnie klient wysyłał i dlaczego nie udało się tego obsłużyć.

W następnym podrozdziale znajduje się opis kilku najlepszych praktyk dotyczących rejestrowania informacji w API REST.

Najlepsze praktyki rejestrowania informacji w API REST

W dużych rozproszonych środowiskach dane z dziennika mogą być jedyną dostępną informacją, którą programista może wykorzystać do celów diagnostycznych. Audyty i dzienniki, jeśli zostaną poprawnie wykonane, mogą być bardzo pomocne przy szukaniu problemów z działaniem aplikacji i przy odtwarzaniu sekwencji zdarzeń, które doprowadziły do awarii. Poniżej opisuję niektóre najlepsze praktyki dotyczące rejestrowania danych, dzięki którym można uzyskać informacje pozwalające zrozumieć, co się dzieje w systemie i jakie jest źródło problemów z wydajnością lub innych.

Zastosowanie spójnego szczegółowego wzorca we wszystkich dziennikach

Wzór dziennika powinien zawierać przynajmniej następujące elementy:

- bieżącą datę i godzinę,
- poziom rejestracji,
- nazwę wątku,
- prostą nazwę rejestratora,
- szczegółową wiadomość.

Zaciemnianie danych poufnych

Dane poufne w dziennikach produkcyjnych koniecznie należy zamaskować lub zaciemnić, aby uniemożliwić wyciek ważnych informacji klientów. W filtrze rejestrującym można zastosować algorytm do zaciemniania haseł, który uniemożliwi odczytanie haseł, numerów kart kredytowych

itp. **Dane osobowe** to takie informacje, które samodzielnie lub w połączeniu z innymi danymi pozwalają zidentyfikować osobę. Przykładami takich informacji są nazwisko, adres e-mail, numer karty kredytowej itd. Wszelkie takie dane powinny być zamaskowane przy użyciu różnych technik, np. zastępowania, mieszania, szyfrowania itd.

> Więcej informacji na temat maskowania danych można znaleźć na stronie *http://en.wikipedia.org/wiki/Data_masking*.

Identyfikacja wywołującego lub inicjatora w dziennikach

Dobrym pomysłem jest zidentyfikowanie w dziennikach osoby, która wysłała żądanie. Z API mogą korzystać różne klienty, np. urządzenia mobilne, strony internetowe i inne usługi. Jeśli problem dotyczy konkretnie jednego rodzaju klienta, dodanie możliwości zidentyfikowania go może pomóc w postawieniu diagnozy.

Nie rejestruj treści głównej domyślnie

Zaimplementuj opcję rejestrowania treści głównej żądań i domyślnie ją wyłącz. Dzięki temu, jeśli zasoby będą zawierały poufne dane, nie zostaną one domyślnie ujawnione.

Identyfikacja metainformacji dotyczących żądania

Dla każdego żądania powinny być zapisane informacje na temat czasu jego wykonywania, statusu oraz rozmiaru. Pomaga to w wykryciu opóźnień oraz wszelkich problemów z wydajnością, które mogą wystąpić w przypadku, gdy wiadomości są bardzo duże.

Powiązanie systemu rejestrującego z systemem monitorującym

Dane z dzienników powinno dać się też powiązać z systemem monitorującym, który może zbierać dane dotyczące gwarantowanego poziomu świadczenia usług i innych statystyk.

> **Studia przypadków dotyczące systemów rejestrowania danych w rozporoszonych środowiskach na różnych platformach**
>
> Portal Facebook opracował własne rozwiązanie o nazwie Scribe. Jest to serwer do gromadzenia strumieniowo przesyłanych danych dzienników. Jest w stanie obsłużyć dużą liczbę żądań dziennie na różnych serwerach rozrzuconych po całym świecie. Serwery te wysyłają dane, które mogą następnie zostać przetworzone, poddane diagnostyce, zaindeksowane, podsumowane lub zapisane. Scribe z zasady powinien dać się skalować do bardzo dużej liczby węzłów. Jest tak zaprojektowany, aby przetrwać nawet awarie sieci i węzłów. Serwer ten działa na każdym węźle w systemie. W większych grupach wiadomości są gromadzone i wysyłane do centralnego serwera. Jeśli serwer ten ulegnie awarii, wiadomości są zapisywane w plikach na dyskach lokalnych i przesyłane do serwera centralnego, gdy ten znowu zacznie działać. Więcej informacji o Scribe można znaleźć na stronie *https://github.com/facebookarchive/scribe*.

Dapper to system śledzenia firmy Google, który pobiera próbki danych z tysięcy żądań i dostarcza wystarczającą ilość informacji do śledzenia. Ślady są gromadzone w lokalnych dziennikach i pobierane do bazy danych Google BigTable. Firma Google odkryła, że próbkowanie informacji dotyczących typowych przypadków pomaga w śledzeniu szczegółów. Więcej na ten temat znajduje się na stronie *http://research.google.com/pubs/pub36356.html*.

W następnym podrozdziale znajduje się opis technik sprawdzania poprawności żądań wysyłanych do API REST i jednostek odpowiedzi.

Sprawdzanie poprawności usług REST

Przed udostępnieniem usług REST — lub jakiejkolwiek innej usługi opartej na protokole HTTP — należy sprawdzić, czy działa poprawnie i zwraca dane w odpowiednim formacie. Na przykład powinno się sprawdzić dane wejściowe wprowadzane do usługi typu RESTful, przykładowo: czy wysyłane w treści adresy e-mail są zgodne ze standardem, czy w treści znajdują się określone wartości, czy kod pocztowy ma prawidłowy format itd.

JAX-RS obsługuje specyfikację Bean Validation do weryfikowania klas zasobów JAX-RS. Obsługa ta składa się z następujących elementów:

- Dodanie adnotacji ograniczających parametrów metod zasobów.
- Zapewnienie poprawności danych jednostek przekazywanych jako parametry.

Poniżej znajduje się kod źródłowy klasy `CoffeesResource` zawierającej adnotację `@Valid`:

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@ValidateOnExecution
public Response addCoffee(@Valid Coffee coffee) {
    ...
        }
```

Adnotacja `javax.validation.executable.ValidateOnExecution` może pomóc w określeniu, w przypadku której metody lub którego konstruktora parametry i wartości zwrotne powinny zostać sprawdzone przed wykonaniem. Adnotacja `javax.validation.Valid` przy treści żądania zapewnia poprawność obiektu `Coffee` jako zwykłego obiektu Javy.

Poniżej znajduje się definicja klasy `Coffee`:

```
@XmlRootElement
public class Coffee {

    @VerifyValue(Type.class)
    private String type;
```

```
    @VerifyValue(Size.class)
    private String size;

    @NotNull
    private String name;
    // metody pobierające i ustawiające
}
```

Nazwa pola ma adnotację `javax.validation.constrains.NotNull` oznaczającą, że nazwa kawy w zamówieniu nie może być pusta. Ponadto zdefiniowaliśmy własne adnotacje weryfikujące typ i rozmiar oraz sprawdzające, czy wartości w treści żądania są w odpowiednim formacie.

Na przykład pole rozmiaru `Size` może mieć jedną z następujących wartości: `Small`, `Medium`, `Large` lub `ExtraLarge`:

```
public enum Size {
    Small("S"), Medium("M"), Large("L"), ExtraLarge("XL");
    private String value;
}
```

Definicja niestandardowej adnotacji `@VerifyValue(Size.class)` znajduje się w przykładowych plikach do pobrania z serwera FTP.

Obsługa wyjątków i kodów odpowiedzi związanych z weryfikacją poprawności danych

W poniższej tabeli znajduje się zwięzły opis typów kodów odpowiedzi, jakie mogą zostać zwrócone, gdy wystąpi wyjątek dotyczący weryfikacji poprawności danych. Rodzaj kodu błędu zależy od typu wyjątku oraz tego, czy weryfikacja jest przeprowadzana na żądaniu, czy odpowiedzi metody HTTP.

Kod odpowiedzi HTTP	Typ wyjątku
`500 Internal Server Error`	Ten kod błędu jest zwracany po tym, gdy zostanie zgłoszony wyjątek typu `javax.validation.ValidationException` lub którejkolwiek z podklas klasy `ValidationException`, włącznie z `ConstraintValidationException` podczas sprawdzania poprawności typu zwrotnego metody.
`400 Error`	Kod zwracany dla wyjątków typu `ConstraintViolationException` we wszystkich pozostałych przypadkach weryfikacji metody.

W następnym podrozdziale znajduje się opis sposobów zgłaszania wyjątków specyficznych dla aplikacji i wysyłania odpowiednich kodów błędów HTTP w zależności od ich typów.

Obsługa błędów w usługach typu RESTful

API typu RESTful powinno zgłaszać specyficzne wyjątki i dostarczać specyficzne odpowiedzi HTTP zawierające szczegółowe informacje o tych wyjątkach. Poniżej wyjaśniam, jak obsługiwać wyjątki zdefiniowane przez użytkownika oraz jak odwzorowywać je na odpowiedzi HTTP i kody statusu. Klasy `javax.ws.rs.ext.ExceptionMapper` to niestandardowe elementy aplikacji przechwytujące zgłoszone wyjątki i zapisujące odpowiedzi HTTP. Klasy mapowania wyjątków oznacza się adnotacją `@Provider`.

Poniżej znajdują się przykłady ilustrujące sposób budowy własnego mapera wyjątków:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("/orders/{id}")
public Response getCoffee(@PathParam("id") int id) {
    Coffee coffee = CoffeeService.getCoffee(id);
    if (coffee == null)
        throw new CoffeeNotFoundException("Nie znaleziono kawy dla
          zamówienia " + id);
    return Response.ok(coffee).type(MediaType.APPLICATION_JSON_TYPE).build();
}
```

Jak widać w powyższym kodzie, metoda `getCoffee()` zwraca obiekt klasy `Coffee` o identyfikatorze podanym w parametrze ścieżki. Jeśli nie zostanie znaleziona kawa o określonym identyfikatorze, następuje zgłoszenie wyjątku `CoffeeNotFoundException`.

Poniżej znajduje się implementacja klasy `ExceptionMapper`:

```
@Provider
public class MyExceptionMapper implements ExceptionMapper<Exception> {

    public Response toResponse(Exception e) {
        ResourceError resourceError = new ResourceError();

        String error = "Wystąpił wewnętrzny błąd usługi.";
        if (e instanceof CoffeeNotFoundException) {
            resourceError.setCode(Response.Status.NOT_FOUND.getStatusCode());
            resourceError.setMessage(e.getMessage());

            return Response.status(
              Response.Status.NOT_FOUND).entity(resourceError)
                  .type(MediaType.APPLICATION_JSON_TYPE).build();
        }
        return Response.status(503).entity(resourceError)
                   .type(MediaType.APPLICATION_JSON_TYPE).build();
    }
}
```

Jest to implementacja interfejsu `ExceptionMapper` zawierająca definicję metody `toResponse()`. Kod ten sprawdza, czy zgłoszony wyjątek jest egzemplarzem klasy `CoffeeNotFoundException`, i zwraca odpowiedź, której jednostka jest typu `ResourceError`.

`ResourceError` to zwykła klasa Javy z adnotacją `@XMLRootElement`, której obiekt jest wysyłany w odpowiedzi:

```
@XmlRootElement
public class ResourceError {

    private int code;
    private String message;
    // metody pobierające i ustawiające
...}
```

Kod ten można uruchomić w ramach przykładów do pobrania z FTP. Wynik powinien być następujący:

```
HTTP/1.1 404 Not Found
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source
Edition 4.0 Java/Oracle Corporation/1.7)
Server: GlassFish Server Open Source Edition 4.0
Content-Type: application/json
Content-Length: 54
{"code":404,"message":" Nie znaleziono kawy dla zamówienia 100"}
```

Uwierzytelnianie i autoryzacja

Kiedyś organizacje potrzebowały metody na ujednolicenie sposobów uwierzytelniania użytkowników w całym przedsiębiorstwie. Pojedyncze logowanie jest rozwiązaniem polegającym na przechowywaniu jednego składu nazw użytkowników i haseł, którego można używać w różnych aplikacjach.

Wraz z rozwojem różnych rodzajów architektury usługowej pojawiła się potrzeba umożliwienia korzystania z API różnym partnerom i uproszczenia procesu logowania w różnych aplikacjach i platformach. Potrzeba ta stała się jeszcze bardziej paląca z uwagi na rozpowszechnienie się mediów społecznościowych i otwarcie różnych platform, pojawienie się rozmaitych API i ekosystemu złożonego z niezliczonej ilości aplikacji i urządzeń korzystających m.in. z Twittera, Facebooka i LinkedIn.

Dlatego coraz ważniejsze jest oddzielenie funkcji uwierzytelniania i autoryzacji od aplikacji konsumenckich. Ponadto nie każda aplikacja musi znać dane poświadczające tożsamość użytkownika. W tym podrozdziale znajduje się opis technologii do obsługi autoryzacji SAML 2.0 i OAuth 2.0 jako narzędzi do obsługi tożsamości federacyjnych mających uprościć logowanie i zwiększyć bezpieczeństwo.

W kolejnych podrozdziałach opisane są następujące tematy:

- SAML.
- OAuth.
- Tokeny odświeżania a tokeny dostępu.
- Jersey i OAuth 2.0.
- Kiedy używać SAML, a kiedy OAuth.
- OpenID Connect.

Co to jest uwierzytelnianie

Uwierzytelnianie to proces, którego celem jest zweryfikowanie, czy osoba używająca przeglądarki internetowej lub innej aplikacji jest rzeczywiście tym, za kogo się podaje.

SAML

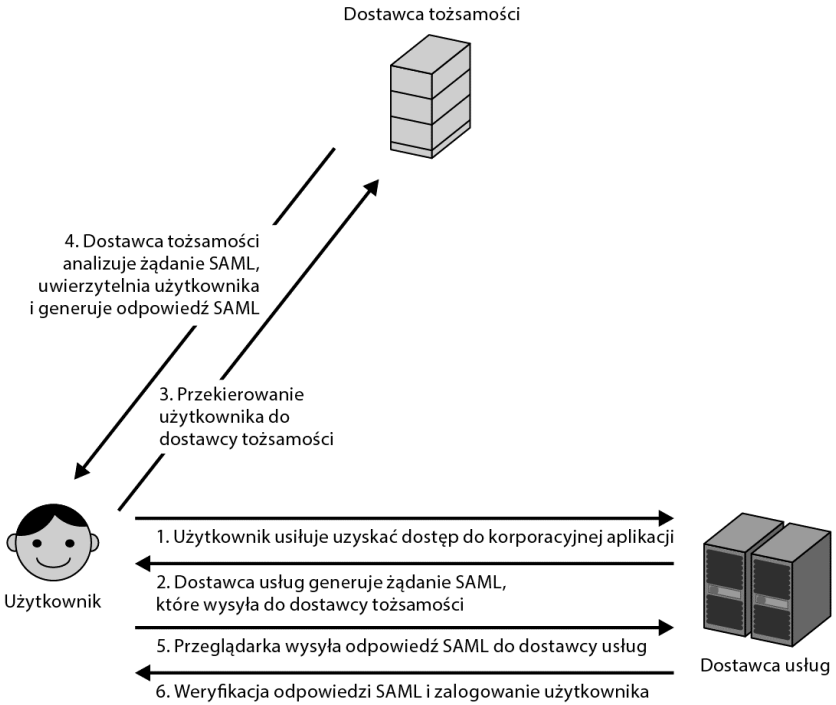
SAML (ang. *Security Assertion Markup Language*) to standard obejmujący profile, wiązania oraz konstrukcje do zarządzania **logowaniem pojedynczym** (ang. *Single Sign-On* — SSO), federacyjnym i tożsamościowym.

Specyfikacja SAML 2.0 dostarcza przeglądarkowy profil SSO określający sposób pojedynczego logowania w aplikacjach sieciowych. Profil ten definiuje trzy role:

- ***Principal*** (podmiot zabezpieczeń) — gdzie użytkownik weryfikuje swoją tożsamość.
- ***Identity provider*** **(IdP)** (dostawca tożsamości) — jednostka mogąca zweryfikować tożsamość użytkownika końcowego.
- ***Service provider*** **(SP)** (dostawca usług) — jednostka używająca dostawcy tożsamości w celu zweryfikowania tożsamości użytkownika końcowego.

Na poniższym schemacie przedstawiono typowy przykład użycia protokołu SAML. Powiedzmy, że pracownik chce uzyskać dostęp do korporacyjnej witryny podróżniczej. Korporacyjna aplikacja podróżnicza zażąda dostawcy tożsamości związanego z tym pracownikiem, aby zweryfikować jego tożsamość, i na podstawie otrzymanych informacji podejmie odpowiednie działania.

1. Użytkownik próbuje użyć korporacyjnej aplikacji, np. aplikacji podróżniczej.
2. Program ten generuje żądanie SAML i przekierowuje użytkownika do **dostawcy tożsamości** pracodawcy.
3. Użytkownik zostaje przekierowany do dostawcy tożsamości pracodawcy w celu pobrania asercji uwierzytelniającej SAML.
4. Dostawca tożsamości analizuje żądanie SAML, uwierzytelnia użytkownika i generuje odpowiedź SAML.
5. Przeglądarka wysyła odpowiedź SAML do aplikacji podróżniczej.



6. Aplikacja podróżnicza odbiera token dostępowy i otrzymuje dostęp do zasobu sieciowego przez przekazanie tego tokenu w nagłówku żądania HTTP. Token ten pełni podobną rolę do tokenu sesji zawierającego informację, że aplikacja podróżnicza działa w imieniu użytkownika.

Protokół SAML ma specyfikacje wiązań dla przeglądarek internetowych, SSO, SOAP i WS-Security, ale nie ma formalnego wiązania dla API REST.

W następnym podrozdziale znajduje się opis protokołu OAuth, który jest powszechnie używany do autoryzacji na takich platformach jak Twitter, Facebook i Google.

Co to jest autoryzacja

Autoryzacja to proces polegający na sprawdzeniu, czy nadawca żądania ma uprawnienia do wykonania określonych czynności.

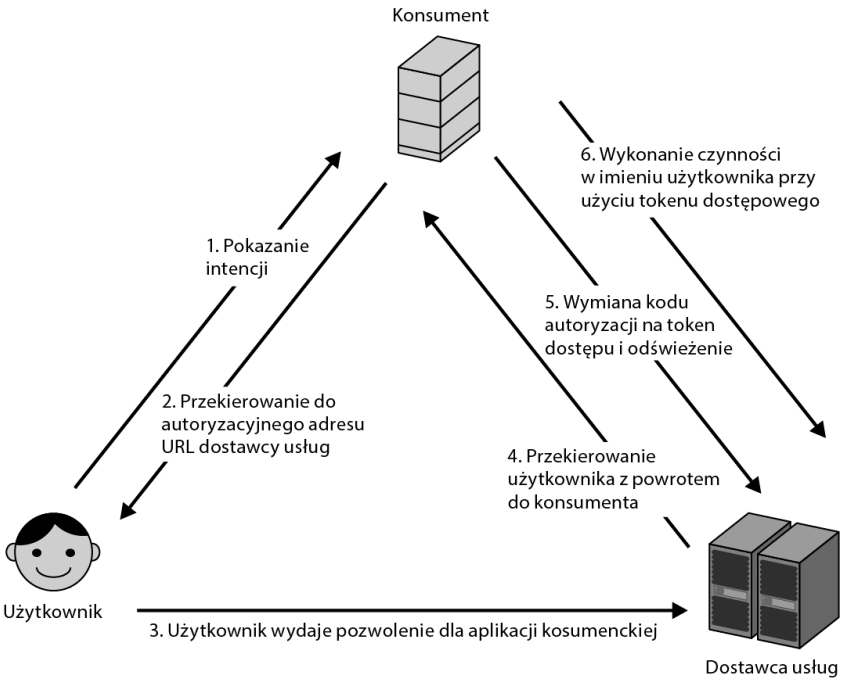
OAuth

Skrót OAuth pochodzi od nazwy *Open Authorization* (otwarta autoryzacja). Jest to więc technologia do autoryzacji użytkowników w aplikacjach w celu uzyskania dostępu do danych z konta bez ujawniania nazwy użytkownika i hasła.

W tradycyjnym uwierzytelnianiu klient-serwer klient, chcąc uzyskać dostęp do swoich zasobów na serwerze, przekazuje swoje dane poświadczające. Dla serwera nie ma znaczenia, czy żądanie przyjdzie z klienta, czy też klient żąda zasobów dla jakiejś innej jednostki. Jednostką tą może być inna aplikacja lub osoba, w związku z czym klient nie uzyskuje dostępu do własnego zasobu, tylko do zasobu kogoś innego. Każdy, kto chce uzyskać dostęp do zasobu podlegającego ochronie i wymagającego uwierzytelnienia, musi otrzymać autoryzację od właściciela tego zasobu. Protokół OAuth to technologia otwierająca API REST dla takich portali jak Twitter, Facebook, Google+, GitHub itp. i niezliczonych działających na ich podstawie aplikacji. OAuth 2.0 w pełni bazuje na SSL.

Liczba elementów w żądaniu OAuth zależy od liczby zaangażowanych stron. Jeśli jest klient, serwer i właściciel zasobu, to znaczy, że są trzy elementy. Jeżeli klient działa we własnym imieniu, mówimy o autoryzacji dwuelementowej (dwustronnej).

Funkcjonowanie protokołu OAuth opiera się na użyciu tokenów dostępowych. Są one czymś w rodzaju kluczy dających ograniczony dostęp do funkcjonalności na pewien czas. Tokeny mają ograniczony termin ważności, który może wynosić od paru godzin do paru dni. Na poniższym schemacie pokazano sposób użycia protokołu OAuth.



Schemat ten przedstawia proces autoryzacji kodu.

W przykładzie tym użytkownik przechowuje zdjęcia w serwisie będącym dostawcą usług, takim jak np. Flickr. Użytkownik ten chce wywołać usługę drukowania, aby wydrukować swoje zdjęcia, np. Snapfish, która jest aplikacją konsumencką. Zamiast podawać swoją nazwę użytkownika i hasło tej aplikacji, użytkownik może przy użyciu OAuth udostępnić swoje zdjęcia na ograniczony czas.

W tym przykładzie wystąpiły trzy role:

- **Użytkownik, czyli właściciel zasobu** — użytkownik jest właścicielem, który chce wydrukować swoje zdjęcia.
- **Aplikacja konsumencka, czyli klient** — w tym przypadku jest to aplikacja oferująca usługę drukowania, która będzie działać w imieniu użytkownika.
- **Dostawca usług, czyli serwer** — dostawca usług jest serwerem zasobów, na którym przechowywane są zdjęcia użytkowników.

W przykładzie można wyodrębnić poszczególne etapy działania poprzez protokół OAuth:

1. Użytkownik chce umożliwić aplikacji wykonanie pewnych czynności w jego imieniu. Może to być na przykład wydrukowanie zdjęć znajdujących się na serwerze przy użyciu aplikacji konsumenckiej.
2. Aplikacja konsumencka przekierowuje użytkownika do autoryzacyjnego adresu URL dostawcy usługi.
3. Użytkownik zgadza się na przyznanie aplikacji dostępu do swoich zasobów.
4. Dostawca usług przekierowuje użytkownika z powrotem do aplikacji (przy użyciu przekierowującego URI), przekazując kod autoryzacji jako parametr.
5. Aplikacja wymienia kod autoryzacji na prawo dostępu. Dostawca usług wydaje aplikacji zezwolenie na dostęp. Zezwolenie to zawiera token dostępowy i token odświeżania.
6. Po ustanowieniu połączenia aplikacja konsumencka uzyskuje referencję do API usługi i może wywoływać dostawcę w imieniu użytkownika. Dzięki temu usługa drukowania może pobrać zdjęcia użytkownika z serwisu dostawcy usług.

Zaletą protokołu OAuth jest to, że nawet jeśli dojdzie do złamania zabezpieczeń aplikacji, nie spowoduje to wielkich szkód, ponieważ w użyciu są tokeny dostępu, a nie prawdziwe dane poświadczające. Metoda SAML z tokenem autoryzującym (ang. *bearer token*) jest bardzo podobna do opisanej klasycznej trójstronnej metody OAuth. Ale zamiast przekierowywać przeglądarkę użytkownika do serwera autoryzacji, dostawca usług współpracuje z dostawcą tożsamości w celu otrzymania prostej asercji uwierzytelniania. Aplikacja dostawcy usług zamiast kodu autoryzacji wymienia asercję autoryzującą (ang. *bearer assertion*) SAML na użytkownika.

Różnice między OAuth 2.0 i OAuth 1.0

W specyfikacji OAuth 2.0 wyraźnie napisano, jak używać protokołu OAuth w całości w przeglądarce internetowej za pomocą JavaScriptu, w którym nie ma możliwości bezpiecznego przechowywania tokenu. Ponadto ogólnie wyjaśniono, jak używać OAuth w telefonach komórkowych, a nawet urządzeniach pozbawionych przeglądarki internetowej, i opisano interakcje z **aplikacjami** i **aplikacjami macierzystymi** zarówno w smartfonach, jak i w tradycyjnych urządzeniach komputerowych.

OAuth 2.0 definiuje następujące trzy typy profili:

- Aplikacja sieciowa. (W tym przypadku hasło klienta jest przechowywane na serwerze i używany jest token dostępu).
- Klient w postaci przeglądarki internetowej. (W tym przypadku dane poświadczające OAuth nie są zaufane; niektórzy dostawcy nie ujawniają sekretów klienta. Przykładem może być JavaScript w przeglądarce).
- Aplikacja macierzysta. (W tym przypadku wygenerowane tokeny dostępu lub tokeny odświeżania mogą zapewnić wystarczający poziom ochrony. Przykładem mogą być aplikacje mobilne).

Protokół OAuth 2.0 nie wymaga szyfrowania i używa HTTPS, a nie HMAC. Ponadto OAuth 2.0 umożliwia ograniczenie terminu ważności tokenu dostępu.

Grant autoryzacji

Grant autoryzacji to informacja reprezentująca właściciela zasobu lub autoryzację użytkownika, pozwalająca klientowi uzyskać dostęp do chronionych zasobów w celu pobrania tokenu dostępu. W specyfikacji OAuth 2.0 zdefiniowane są cztery typy grantów:

- grant kodu autoryzacji,
- grant niejawny,
- grant hasła właściciela zasobu,
- grant danych poświadczających klienta.

Ponadto OAuth 2.0 zawiera mechanizm umożliwiający definiowanie dodatkowych typów.

Tokeny odświeżania a tokeny dostępu

Token odświeżania to dane poświadczające umożliwiające uzyskanie tokenu dostępu. Tokenu odświeżania używa się do pobierania tokenu dostępu, gdy bieżący token stanie się nieważny lub minie jego termin ważności. Wydanie tokenu odświeżającego leży w gestii serwera.

W odróżnieniu od tokenu dostępu, token odświeżania jest przeznaczony do użycia tylko z serwerami autoryzacji i nie jest nigdy wysyłany do serwerów zasobów w celu uzyskania dostępu do zasobów.

Jersey i OAuth 2.0

Choć OAuth 2.0 jest powszechnie używany przez różne przedsiębiorstwa, dokument RFC OAuth 2.0 opisuje tylko szkielet, na podstawie którego można budować rozwiązania. W dokumencie tym znajduje się wiele luk, w których specyfikacja daje wolną rękę implementatorowi. Zespół pracujący nad OAuth 2.0 miał problemy z dojściem do porozumienia w pewnych kwestiach, takich jak wymagany typ tokenu, wymóg wskazania terminu wygaśnięcia tokenu czy określenie rozmiaru tokenu.

> Więcej informacji na ten temat można znaleźć na stronie *http://hueniverse.com/2012/07/26/oauth-2-0--and-the-road-to-hell/*.

Aktualnie Jersey obsługuje OAuth 2.0 tylko od strony klienta. W specyfikacji OAuth 2.0 zdefiniowano wiele punktów rozszerzeń, których implementacja należy do obowiązków dostawców usług. Ponadto w OAuth 2.0 zdefiniowano więcej niż jeden sposób autoryzacji. Jersey aktualnie obsługuje tylko metodę z użyciem grantu kodu autoryzacji. Więcej informacji na ten temat znajduje się na stronie *https://jersey.java.net/documentation/latest/security.html*.

Najlepsze praktyki przy implementacji OAuth w API REST

Poniżej znajduje się opis niektórych najlepszych praktyk w zakresie implementacji protokołu OAuth 2.0.

Ograniczanie czasu życia tokenu dostępu

Parametr protokołu `expires_in` umożliwia serwerowi autoryzacji ograniczenie czasu życia tokenu dostępu i przekazanie tej informacji do klienta. Mechanizm ten można wykorzystać do wydawania krótkotrwałych tokenów.

Dostarczanie tokenów odświeżania na serwerze autoryzacji

Token odświeżania można wysyłać wraz z krótkotrwałym tokenem dostępu w celu przyznania dostępu do zasobów na dłuższy czas bez potrzeby autoryzacji przez użytkownika. Zaletą tego podejścia jest to, że można rozdzielić serwery zasobów i autoryzacji. Na przykład w środowisku rozproszonym token odświeżania jest zawsze wymieniany na serwerze autoryzacji.

Użycie SSL i zastosowanie szyfrowania

Protokół OAuth 2.0 w dużym stopniu bazuje na HTTPS. Dzięki temu system jest prostszy, ale i mniej bezpieczny.

W poniższej tabeli zostały zebrane informacje na temat tego, kiedy używać SAML, a kiedy OAuth.

Scenariusz	SAML	OAuth
Jeśli jedna ze stron jest przedsiębiorstwem	Użyj SAML.	
Jeśli aplikacja musi dać tymczasowy dostęp do niektórych zasobów		Użyj OAuth.
Jeśli aplikacja potrzebuje niestandardowego dostawcy tożsamości	Użyj SAML.	
Jeśli z aplikacji korzystają aplikacje mobilne		Użyj OAuth.
Jeśli aplikacja nie ma ograniczeń dotyczących transportu, np. SOAP i JMS	Użyj SAML.	

OpenID Connect

W fundacji OpenID trwają prace nad technologią OpenID Connect. Jest to prosty, oparty na REST i JSON protokół zbudowany na bazie OAuth 2.0. OpenID Connect jest prostszy niż SAML, łatwy w obsłudze i nadaje się do użycia na różnych poziomach zabezpieczeń, od sieci społecznościowych przez aplikacje biznesowe po aplikacje rządowe o restrykcyjnych wymaganiach dotyczących bezpieczeństwa. OpenID Connect i OAuth są przyszłością uwierzytelniania i autoryzacji. Więcej informacji o OpenID Connect można znaleźć na stronie *http://openid.net/connect/*.

Studia przypadków firm używających OAuth 2.0 i OpenID Connect

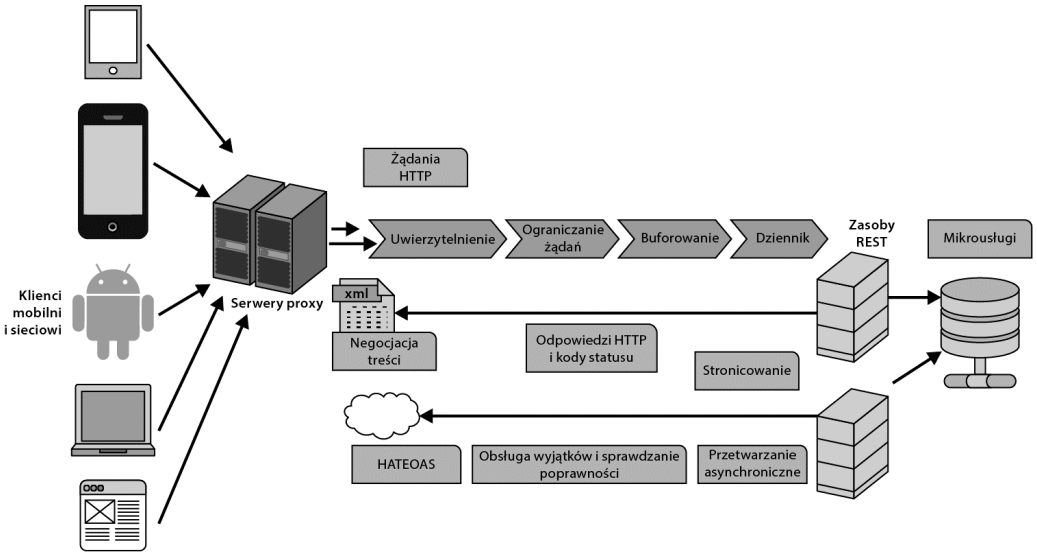
Mechanizm logowania portalu Google+ jest zbudowany na bazie protokołów OAuth i OpenID Connect. Umożliwia instalację bez użycia kabla (ang. *over-the-air installation*), ma funkcje do obsługi mediów społecznościowych oraz udostępnia widżet logowania zbudowany na bazie standardowych technik logowania OpenID Connect.

W następnym podrozdziale znajduje się podsumowanie informacji o różnych opisanych do tej pory elementach usług typu RESTful.

Elementy architektury REST

W tym podrozdziale znajduje się opis różnych elementów, które należy uwzględnić przy budowaniu API typu RESTful. Szersze informacje na ich temat znajdują się w innych podrozdziałach. Ponadto przedstawiam najlepsze praktyki pozwalające uniknąć rozmaitych pułapek przy projektowaniu i rozwijaniu API REST. Na poniższym schemacie (patrz rysunek na następnej stronie) zobaczysz elementy architektury REST.

Jak widać na tym schemacie, usługi REST mogą być wykorzystywane przez wiele różnych klientów działających na różnych platformach i w rozmaitych urządzeniach, np. w telefonach czy przeglądarkach internetowych.



Żądania są przesyłane przez serwer proxy. Przedstawione elementy architektury REST można łączyć ze sobą, jak to pokazano na schemacie. Na przykład można utworzyć łańcuch filtrujący składający się z elementów *Uwierzytelnianie*, *Ograniczanie żądań*, *Buforowanie* i *Dziennik*. Łańcuch taki obsłuży uwierzytelnianie użytkowników, sprawdzanie, czy nie został przekroczony limit liczby żądań, sprawdzanie, czy żądanie można obsłużyć z bufora, oraz rejestrowanie informacji o żądaniu.

Jeśli chodzi o odpowiedź, *Stronicowanie* pozwala na wysyłanie przez serwer tylko podzbioru wyników. Ponadto serwer może wykonywać *Przetwarzanie asynchroniczne*, pozwalające poprawić interaktywność i skalowalność aplikacji. W odpowiedzi mogą też być umieszczane łącza, co wiąże się z HATEOAS.

Oto niektóre opisane do tej pory elementy architektury REST:

- Żądania HTTP umożliwiające korzystanie z API REST poprzez czasowniki HTTP w celu zapewnienia jednolitego ograniczonego interfejsu.
- Negocjacja treści mająca na celu wybór reprezentacji odpowiedzi w przypadku, gdy dostępnych jest wiele reprezentacji.
- Rejestrowanie danych w dziennikach w celu zapewnienia informacji do analizy w przypadku wystąpienia awarii.
- Obsługa wyjątków pozwalająca wysyłać wyjątki specyficzne dla aplikacji z kodami HTTP.
- Uwierzytelnianie i autoryzacja OAuth 2.0 zapewniające kontrolę dostępu do innych aplikacji i umożliwiające wykonywanie czynności bez potrzeby podawania przez użytkownika danych poświadczających.

- Mechanizmy sprawdzania danych mające zapewnić klientowi szczegółowe informacje zwrotne o błędach oraz zweryfikować dane wejściowe otrzymane w żądaniach.

W kolejnych paru rozdziałach skupiam się na zaawansowanych zagadnieniach i najlepszych praktykach dotyczących tematów wymienionych poniżej. Przedstawiam też przykłady kodu ilustrujące implementację opisywanych elementów przy użyciu JAX-RS.

- Ograniczenie liczby żądań zapobiegające przeciążeniu serwera zbyt dużą liczbą żądań od jednego klienta.
- Buforowanie w celu zapewnienia lepszej interaktywności.
- Przetwarzanie asynchroniczne pozwalające na przesyłanie przez serwer odpowiedzi do klienta w sposób asynchroniczny.
- Mikrousługi pozwalające podzielić jedną wielką usługę na kilka mniejszych.
- HATEOAS w celu poprawienia walorów użytkowych, czytelności i właściwości nawigacyjnych poprzez zwrócenie listy łączy w odpowiedzi.
- Stronicowanie umożliwiające klientowi określenie, które elementy zbioru danych go interesują.

Ponadto dowiesz się, jakie rozwiązania w odniesieniu do przedstawionych kwestii zastosowano w API REST największych platform, takich jak Facebook, Google, GitHub i PayPal.

Zalecana lektura

Poniżej znajduje się lista adresów internetowych, pod którymi można znaleźć dodatkowe informacje na tematy poruszone w tym rozdziale.

- *https://developers.google.com/oauthplayground/*: system firmy Google do tworzenia i testowania podpisanych żądań OAuth.
- *http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/*: kłopoty podczas pracy nad specyfikacją OAuth 2.0.
- *https://developers.google.com/accounts/docs/OAuth2Login*: uwierzytelnianie i autoryzacja na kontach Google.
- *https://github.com/facebookarchive/scribe*: serwer logowania Scribe Facebooka.
- *http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/36356.pdf*: Google Dapper — infrastruktura do śledzenia dużych systemów rozproszonych.

Podsumowanie

Na początku rozdziału zwięźle opisałam podstawy rejestrowania informacji w dziennikach API typu RESTful oraz najważniejsze zasady i najlepsze praktyki rejestrowania danych żądań z uwzględnieniem kwestii bezpieczeństwa. Pokazałam, jak sprawdzać poprawność zasobów JAX-RS 2.0 przy użyciu Bean Validation. Ponadto w tym rozdziale wyjaśniłam, jak napisać ogólny maper wyjątków dla aplikacji.

Pokazałam, że tożsamości federacyjne przy aktualnym stanie połączonych ze sobą systemów hybrydowych, protokołów i urządzeń są koniecznością. Opisałam podobieństwa i różnice technologii SAML i OAuth 2.0 oraz najlepsze praktyki przy używaniu protokołu OAuth.

W następnym rozdziale znajduje się opis technik buforowania oraz asynchronicznych API REST, które pozwalają polepszyć wydajność i skalowalność. Potem przyjrzymy się sposobom wykonywania częściowych aktualizacji przy użyciu metody `PATCH` protokołu HTTP i nowszej metody JSON Patch.

4

ROZDZIAŁ

Projektowanie wydajnych rozwiązań

REST to styl architektury odnoszący się do architektury sieciowej, który trzeba poprawnie zaprojektować i zaimplementować, aby móc w pełni wykorzystać możliwości skalowalnej sieci. W tym rozdziale znajduje się opis zaawansowanych zasad projektowania pozwalających uzyskać jak najwyższą wydajność, które każdy programista budujący usług typu RESTful powinien znać.

W rozdziale opisane są następujące zagadnienia:

- Zasady buforowania.
- Asynchroniczne i długotrwałe zadania w REST.
- HTTP `PATCH` i częściowe aktualizacje.

Szerzej opiszę różne nagłówki buforowe HTTP oraz pokażę, jak wysyłać żądania warunkowe, aby sprawdzić, czy powinna zostać zwrócona nowa, czy zbuforowana treść. Potem przeanalizujesz parę przykładów implementacji buforowania przy użyciu JAX-RS.

Ponadto dowiesz się, w jaki sposób w API portalu Facebook używane są znaczniki ETag do buforowania. Później przyjrzymy się technikom i najlepszym praktykom obsługi żądań i odpowiedzi asynchronicznych za pomocą JAX-RS. Na zakończenie poznasz metodę HTTP `PATCH` i dowiesz się, jak typowo zaimplementować częściowe aktualizacje.

W rozdziale przedstawione są różne fragmenty kodu. Całe działające przykłady znajdują się w plikach z kodem źródłowym umieszczonych na serwerze FTP wydawnictwa.

Zasady buforowania

W tym podrozdziale znajduje się opis różnych zasad projektowania usług typu RESTful. Jednym z poruszanych tematów jest **buforowanie**. Polega ono na zapisywaniu odpowiedzi na żądania w tymczasowym kontenerze i przechowywaniu ich tam przez pewien czas. Dzięki temu w przyszłości serwer nie będzie musiał przetwarzać tych żądań ponownie, tylko będzie mógł wysłać informacje z bufora.

Elementy przechowywane w buforze mogą tracić ważność po upływie określonego czasu lub w momencie, gdy zapisane obiekty ulegną zmianie, bo np. ktoś zmodyfikuje lub usunie zasób.

Buforowanie ma wiele zalet. Pomaga zredukować *czas odpowiedzi* i zwiększyć interaktywność aplikacji. Zmniejsza liczbę żądań do obsłużenia przez serwer, dzięki czemu serwer może obsłużyć więcej klientów, które otrzymają odpowiedzi w krótszym czasie.

Ogólnie rzecz biorąc, bardzo dobrze buforuje się takie zasoby jak obrazy, pliki JavaScript i arkusze stylów. Ponadto zaleca się buforowanie odpowiedzi, których wygenerowanie wymaga wykonania dużej ilości obliczeń.

Szczegóły buforowania

W tym podrozdziale szczegółowo opisuję techniki buforowania. Aby buforowanie było skuteczne, należy wykorzystać nagłówki HTTP do określenia terminu ważności i daty ostatniej modyfikacji zasobów.

Typy nagłówków buforowania

W następnych paru podrozdziałach opisuję typy nagłówków buforowania i przedstawiam przykłady ich użycia. Wyróżnia się dwa rodzaje takich nagłówków:

- silne nagłówki buforowania,
- słabe nagłówki buforowania.

Silne nagłówki buforowania

Silne nagłówki buforowania określają czas ważności buforowanego zasobu i przez cały ten okres przeglądarka nie musi wysyłać żadnych dodatkowych żądań `GET`. Do tej grupy należą nagłówki `Expires` i `Cache-Control: max-age`.

Słabe nagłówki buforowania

Słabe nagłówki buforowania pomagają przeglądarce w podjęciu decyzji, czy pobrać element z bufora poprzez wysłanie warunkowego żądania `GET`. Do tej grupy należą nagłówki `Last-Modified` i `ETag`.

Nagłówki Expires i Cache-Control: max-age

Nagłówki `Expires` i `Cache-Control` określają termin, w jakim przeglądarka internetowa może używać zasobów z bufora bez sprawdzania, czy pojawiła się nowsza wersja. Jeśli nagłówki te zostaną zdefiniowane, nowszy zasób zostanie pobrany dopiero po upływie określonej daty ważności lub po tym, gdy zasób osiągnie określony wiek. Nagłówek `Expires` wyznacza datę, po której dany zasób staje się nieważny. Atrybut `max-age` określa natomiast, ile czasu zasób jest ważny od chwili jego pobrania.

Nagłówek Cache-Control i dyrektywy

W protokole HTTP 1.1 nagłówek `Cache-Control` określa sposób buforowania zasobu oraz maksymalny czas przetrzymywania zasobu w buforze. W poniższej tabeli przedstawiono dyrektywy tego nagłówka:

Dyrektywa	Opis
`private`	Oznacza, że przeglądarka może buforować obiekt, ale nie mogą robić tego serwery proxy i sieci dostarczania treści (CDN).
`public`	Oznacza, że obiekt mogą buforować wszystkie przeglądarki, serwery proxy i sieci dostarczania treści (CDN).
`no-cache`	Oznacza, że obiektu nie można buforować.
`no-store`	Oznacza, że obiekt może być buforowany w pamięci, ale nie na dysku twardym.
`max-age`	Określa czas ważności zasobu.

Poniżej znajduje się przykładowa odpowiedź zawierająca nagłówek HTTP 1.1 `Cache-Control`:

```
HTTP/1.1 200 OK Content-Type: application/json
Cache-Control: private, max-age=86400
Last-Modified: Thur, 01 Apr 2014 11:30 PST
```

Odpowiedź ta zawiera nagłówek `Cache-Control` z dyrektywą `private` i dyrektywą `max-age` ustawioną na 24 godziny, czyli 86 400 sekund.

Gdy zasób straci ważność z powodu ustawień dyrektywy `max-age` lub nagłówka `Expires`, klient może ponownie wysłać żądanie zasobu lub warunkowe żądanie `GET`, które pobierze zasób tylko pod warunkiem, że coś się w nim zmieniło. Do tego należy użyć słabych nagłówków buforowania: `Last-Modified` i ETag, których opis znajduje się w następnym podrozdziale.

Nagłówki Last-Modified i ETag

Nagłówki `Last-Modified` i ETag umożliwiają przeglądarce sprawdzenie, czy zasób uległ zmianie od czasu wysłania ostatniego żądania `GET`. W nagłówku `Last-Modified` przesyłana jest data ostatniej modyfikacji zasobu. W nagłówku ETag może znajdować się dowolna wartość iden-

tyfikująca zasób (np. skrót). Przy ich użyciu przeglądarka może efektywnie aktualizować swoje zbuforowane zasoby przez wysyłanie warunkowych żądań `GET`. Żądania takie zwracają pełną odpowiedź tylko w sytuacji, gdy zasób na serwerze uległ zmianie. Dzięki temu warunkowe żądania `GET` charakteryzują się niższymi opóźnieniami niż pełne żądania wysłane przy użyciu tej metody.

Nagłówek Cache-Control i API REST

Poniżej znajduje się przykład dodania nagłówka `Cache-Control` do odpowiedzi JAX-RS. Cały kod można znaleźć w plikach źródłowych dostępnych na serwerze FTP wydawnictwa.

```
@Path("v1/coffees")
public class CoffeesResource {

    @GET
    @Path("{order}")
    @Produces(MediaType.APPLICATION_XML)
    @NotNull(message = "Coffee does not exist for the order id requested")
    public Response getCoffee(@PathParam("order") int order) {
        Coffee coffee = CoffeeService.getCoffee(order);
        CacheControl cacheControl = new CacheControl();
        cacheControl.setMaxAge(3600);
        cacheControl.setPrivate(true);
        Response.ResponseBuilder responseBuilder = Response.ok(coffee);
        responseBuilder.cacheControl(cacheControl);
        return responseBuilder.build();
}
```

JAX-RS zawiera klasę `javax.ws.rs.core.Cache-Control`, która jest abstrakcją nagłówka HTTP 1.1 `Cache-Control`. Metoda `setMaxAge()` obiektu `cacheControl` odpowiada dyrektywie `max-age`, a `setPrivate(true)` — dyrektywie `private`. Odpowiedź jest budowana przy użyciu metody `responseBuilder.build()`. Obiekt `cacheControl` jest dodawany do obiektu `Response`, który jest zwracany przez metodę `getCoffee()`.

Poniżej znajduje się odpowiedź z nagłówkami wygenerowana przez tę aplikację:

```
curl -i http://localhost:8080/caching/v1/coffees/1
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source
Edition 4.0 Java/Oracle Corporation/1.7)
Server: GlassFish Server Open Source Edition 4.0
Cache-Control: private, no-transform, max-age=3600
Content-Type: application/xml
Date: Thu, 03 Apr 2014 06:07:14 GMT
Content-Length: 143
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<coffee>
<name>Mocha</name>
<order>1</order>
<size>Small</size>
<type>Chocolate</type>
</coffee>
```

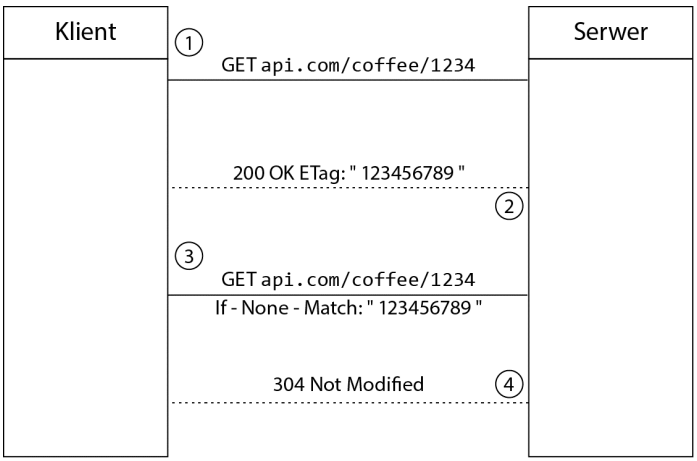
Znaczniki ETag

W protokole HTTP zdefiniowano znakomity mechanizm buforowania zawierający następujące elementy:

- nagłówek `ETag`,
- nagłówek `If-Modified-Since`,
- kod odpowiedzi `304 Not Modified`.

Podstawowe wiadomości o znacznikach ETag

Poniżej przedstawiam podstawowe informacje o tym, jak działają znaczniki ETag. Zacznijmy od przeanalizowania następującego schematu:



Zasada działania znaczników ETag

Przyjrzyjmy się każdemu z etapów przedstawionego procesu po kolei:

1. Klient wysyła żądanie `GET` do zasobu REST `http://api.com/coffee/1234`.
2. Serwer zwraca odpowiedź `200 OK` z wartością ETag, np. `123456789`.
3. Po pewnym czasie klient wysyła kolejne żądanie `GET` do zasobu REST `api.com/coffee/1234` z nagłówkiem `If-None-Match: "123456789"`.

4. Serwer sprawdza, czy skrót MD5 zasobu nie został zmodyfikowany, a następnie wysyła odpowiedź `304 Not-Modified` bez treści głównej.

Jeżeli zasób zmienił się, w odpowiedzi zostanie przesłany kod statusu `200 OK`. Dodatkowo w odpowiedzi serwer wysyła nowy znacznik ETag.

Nagłówek ETag i API REST

Poniżej znajduje się przykładowy kod ilustrujący sposób dodania nagłówka `ETag` do odpowiedzi JAX-RS:

```
@GET
@Path("/etag/{order}")
@Produces(MediaType.APPLICATION_JSON)
@NotNull(message = "Nie ma kawy dla zamówienia o żądanym identyfikatorze.")
public Response getCoffeeWithEtag(@PathParam("order")
  int order, @Context Request request
) {
    Coffee coffee = CoffeeService.getCoffee(order);
    EntityTag et = new EntityTag("123456789");
    Response.ResponseBuilder responseBuilder =
      request.evaluatePreconditions(et);
    if (responseBuilder != null) {
        responseBuilder.build();
    }
    responseBuilder = Response.ok(coffee);
    return responseBuilder.tag(et).build();
```

W kodzie został utworzony obiekt typu `javax.ws.core.EntityTag` przy użyciu skrótu zasobu, który dla uproszczenia zdefiniowano jako `123456789`.

Metoda `request.evalautePreconditions` sprawdza wartość obiektu `EntityTag`. Jeżeli warunki wstępne są spełnione, zwraca odpowiedź z kodem statusu `200 OK`.

Obiekt typu `EntityTag`, `et`, zostaje następnie wysłany wraz z odpowiedzią, która jest zwracana przez metodę `getCoffeeWithETag`. Bardziej szczegółowe informacje można znaleźć w plikach z kodem źródłowym dostępnych na serwerze FTP.

Typy znaczników ETag

Dopasowanie znacznika ETag silnej weryfikacji oznacza, że treść dwóch zasobów jest identyczna z dokładnością co do bajta i że wszystkie inne pola jednostkowe (np. `Content-Language`) także są niezmienione.

Dopasowanie znacznika ETag słabej weryfikacji oznacza tylko, że dwa zasoby są semantycznie równoważne i że można używać zbuforowanych kopii.

Buforowanie pozwala zredukować liczbę żądań od klienta i pełnych odpowiedzi, zmniejszając obciążenie łącza oraz czas potrzebny na wykonanie różnych obliczeń. Wszystko to można osiągnąć przy użyciu warunkowych żądań `GET` i znaczników ETag, nagłówków `If-None-Match` oraz kodów statusu `304 Not Modified`.

> Wraz z nagłówkami `Last-Modified` i `ETag` w odpowiedzi HTTP dobrze jest dodatkowo zdefiniować nagłówek `Expires` lub `Cache-Control max-age`. Nie powinno się jednak wysyłać obu tych nagłówków naraz. Analogicznie sprawa wygląda z nagłówkami `Last-Modified` i `ETag` — wystarczy jeden lub drugi.

API REST Facebooka i nagłówki ETag

API marketingowe portalu Facebook oparte na API Graph obsługuje nagłówki `ETag`. Gdy konsument wysyła żądanie do API Graph, w odpowiedzi otrzymuje nagłówek `ETag`, którego wartością jest skrót danych zwróconych w wyniku tego wywołania. Jeśli klient jeszcze raz wykona takie samo wywołanie, może dołączyć nagłówek `If-None-match` z wartością nagłówka `ETag` zapisaną w poprzedniej operacji. Jeżeli dane się nie zmieniły, w odpowiedzi otrzyma kod statusu `304 Not Modified`, bez żadnych danych.

Jeśli dane na serwerze zmieniły się od czasu wysłania ostatniego żądania, zostaną zwrócone z nowym nagłówkiem `ETag`. Nagłówek ten może potem zostać wykorzystany w następnych wywołaniach. Więcej informacji na ten temat znajduje się na stronie *http://developers.facebook.com*.

RESTEasy i buforowanie

RESTEasy to projekt JBoss zapewniający różne systemy szkieletowe do budowy usług sieciowych i aplikacji Java typu RESTful. Technologia ta może działać w każdym kontenerze serwletów, ale najlepiej zintegrowana jest z serwerem aplikacji JBoss Application Server.

RESTEasy ma rozszerzenie do JAX-RS umożliwiające automatyczne ustawianie nagłówków `Cache-Control` dla poprawnie obsłużonych żądań `GET`.

Ponadto RESTEasy dostarcza bufor serwerowy i lokalny przechowywany w pamięci, które można postawić przed usługami JAX-RS. Automatycznie buforuje poddane szeregowaniu odpowiedzi z wywołań JAX-RS HTTP `GET`, jeśli metoda zasobu JAX-RS definiuje nagłówek `Cache-Control`.

Gdy nadejdzie żądanie HTTP `GET`, bufor serwerowy RESTEasy sprawdza, czy ma już zapisany dany identyfikator URI. Jeśli tak, zwraca poddaną szeregowaniu odpowiedź bez wywoływania metody JAX-RS.

Więcej informacji można znaleźć na stronie *http://resteasy.jboss.org/*.

Rady dotyczące buforowania na serwerze

Unieważniaj znajdujące się w buforze elementy dotyczące żądań PUT i POST. Nie buforuj żądań zawierających parametry zapytaniowe, ponieważ po zmianie parametru odpowiedź zapisana w buforze może stracić ważność.

Asynchroniczne i długotrwałe operacje w REST

Twórcy API typu RESTful często muszą mierzyć się z asynchronicznymi i długotrwałymi operacjami. Tworzą zasoby, których wygenerowanie może zajmować bardzo dużo czasu, ale nie mogą kazać klientowi czekać, aż API zakończy działanie.

Pomyśl o zamawianiu kawy w kawiarni. Dane zamówienia są zapisywane w kolejce i gdy barman ma chwilę czasu, przetwarza je po kolei. Wcześniej dostajemy kwitek potwierdzający złożenie zamówienia, ale kawę otrzymamy za chwilę.

Podobnymi prawami rządzi się przetwarzanie asynchroniczne. Zasoby asynchroniczne to takie, których nie można utworzyć natychmiast. Mogą zostać wstawione do kolejki zadań lub wiadomości obsługującej ich generowanie albo obsłużone w jakiś inny sposób.

Spójrz na poniższe żądanie zamówienia małej kawy:

```
POST v1/coffees/order HTTP 1.1 with body
<coffee>
<size>SMALL</coffee>
<name>ESPRESSO</name>
<price>3.50</price>
<coffee>
```

Można odesłać następującą odpowiedź:

```
HTTP/1.1 202 Accepted
Location: /order/12345
```

W odpowiedzi tej przesłano nagłówek `202 Accepted` i nagłówek `Location` wskazujący miejsce przechowywania szczegółów o zasobie.

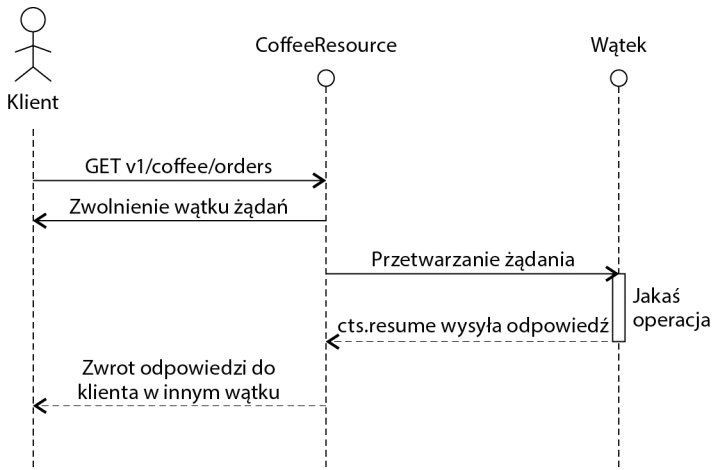
Asynchroniczne przetwarzanie żądań i odpowiedzi

Przetwarzanie asynchroniczne jest wykonywane zarówno przez API klienckie, jak i serwerowe JAX-RS 2.0. Technika ta umożliwia interakcję między klienckimi i serwerowymi elementami aplikacji. Poniżej znajduje się lista nowych interfejsów i klas związanych właśnie z przetwarzaniem asynchronicznym po obu stronach:

- Serwer
  - `AsyncResponse` — wstrzykiwalna asynchroniczna odpowiedź JAX-RS umożliwiająca asynchroniczne przetwarzanie odpowiedzi po stronie serwera.
  - `@Suspend` — adnotacja informująca kontener, że żądanie HTTP należy przetworzyć w drugim wątku.
  - `CompletionCallback` — wywołanie zwrotne odbierające zdarzenia ukończenia przetwarzania żądań.
  - `ConnectionCallback` — wywołanie zwrotne cyklu asynchronicznego przetwarzania żądań odbierające zdarzenia dotyczące cyklu życia asynchronicznych odpowiedzi dotyczących połączeń.
- Klient
  - `InvocationCallback` — wywołanie zwrotne, które można zaimplementować w celu odbierania zdarzeń przetwarzania asynchronicznego od operacji przetwarzania wywołania.
  - `Future` — umożliwia klientowi sondowanie zakończenia operacji asynchronicznej lub zablokowanie się i poczekanie.

> Wprowadzony w Javie SE interfejs `Future` zapewnia dwa różne mechanizmy uzyskiwania wyniku operacji asynchronicznej. Jeden z nich polega na wywołaniu różnych wersji metody `Future.get()`, która blokuje się i czeka, aż pojawi się wynik lub upłynie limit czasu. Drugi polega na sprawdzeniu, czy operacja się zakończyła, za pomocą wywołania metod `isDone()` i `isCancelled()`. Zwracają one wartości logiczne oznaczające aktualny status obiektu `Future`. Więcej informacji na ten temat można znaleźć na stronie *http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Future.html*.

Poniższy schemat przedstawia proces asynchronicznego przetwarzania żądań w JAX-RS:



Asynchroniczne przetwarzanie żądań i odpowiedzi w JAX-RS

Klient wysyła żądanie do asynchronicznej metody obiektu klasy `CoffeeResource`. Klasa `CoffeeResource` tworzy nowy wątek, który może wykonać intensywne operacje i zwrócić odpowiedź. W międzyczasie zostaje zwolniony wątek żądań i można obsługiwać inne żądania. Gdy wątek wykonujący operację zakończy działanie, zwraca odpowiedź do klienta.

Poniżej znajduje się przykład utworzenia asynchronicznego zasobu przy użyciu API JAX-RS 2.0:

```
@Path("/coffees")
@Stateless
public class CoffeeResource {
  @Context private ExecutionContext ctx;
  @GET @Produce("application/json")
  @Asynchronous
  public void order() {
           Executors.newSingleThreadExecutor().submit( new Runnable()
             {
           public void run() {
               Thread.sleep(10000);
               ctx.resume("Hello async world! Coffee Order is 1234");
           } });
ctx.suspend();
return;
  }
}
```

Klasa `CoffeeResource` to bezstanowe ziarno sesyjne zawierające metodę o nazwie `order()`. Metoda ta ma adnotację `@Asynchronous`, którą wystarczy zastosować raz i można o niej zapomnieć. Gdy klient zażąda zasobu poprzez ścieżkę zasobu metody `order()`, następuje utworzenie nowego wątku, którego zadaniem jest przygotowanie odpowiedzi na to żądanie. Wątek ten zostaje przedstawiony egzekutorowi do wykonania i wątek przetwarzający żądanie klienta zostaje zwolniony (poprzez metodę `ctx.suspend`), aby można było przetworzyć inne żądania przychodzące.

Kiedy wątek roboczy, którego celem jest przygotowanie odpowiedzi, zakończy swoją pracę, wywołuje metodę `ctx.resume`, która informuje kontener, że odpowiedź jest gotowa do wysłania do klienta. Jeśli metoda `ctx.resume` zostanie wywołana przed metodą `ctx.suspend` (wątek roboczy przygotował wynik, zanim wykonywanie doszło do metody `ctx.suspend`), zawieszenie jest ignorowane i wynik zostaje wysłany do klienta.

Taką samą funkcjonalność można uzyskać przy użyciu widocznej poniżej adnotacji `@Suspended`:

```
@Path("/coffees")
@Stateless
public class CoffeeResource {
@GET @Produce("application/json")
@Asynchronous
  public void order(@Suspended AsyncResponse ar) {
    final String result = prepareResponse();
    ar.resume(result)
  }
}
```

Kod z adnotacją `@Suspended` jest bardziej przejrzysty, ponieważ nie zawiera zmiennej `ExecutionContext` nakazującej kontenerowi zawieszenie, a potem wznowienie wątku komunikacyjnego, gdy wątek roboczy, w tym przypadku metoda `prepareResponse()`, zakończy działanie. Kod kliencki konsumujący zasób asynchroniczny może wykorzystać mechanizm wywołań zwrotnych lub sondowanie. Poniżej pokazano przykład sondowania przy użyciu interfejsu `Future`:

```
Future<Coffee> future = client.target("/coffees")
               .request()
               .async()
               .get(Coffee.class);
try {
  Coffee coffee = future.get(30, TimeUnit.SECONDS);
} catch (TimeoutException ex) {
 System.err.println("Timeout occurred");
}
```

Na początku utworzono żądanie do zasobu `Coffee`. Użyto egzemplarza klasy `javax.ws.rs.client.Client` w celu wywołania metody `target()`, która tworzy egzemplarz klasy `javax.ws.rs.client.WebTarget` dla zasobu `Coffee`. Metoda `Future.get()` zostaje zablokowana do momentu, aż nadejdzie odpowiedź od serwera, ale nie na dłużej niż 30 sekund.

Innym rozwiązaniem jest użycie egzemplarza interfejsu `javax.ws.rs.client.InvocationCallback`. Jest to wywołanie zwrotne, które można zaimplementować w taki sposób, aby odbierało asynchroniczne zdarzenia z wywołania. Więcej informacji na ten temat można znaleźć na stronie *https://jax-rs-spec.java.net/nonav/2.0/apidocs/javax/ws/rs/client/InvocationCallback.html*.

Najlepsze praktyki pracy z zasobami asynchronicznymi

W paru kolejnych podrozdziałach znajduje się opis najlepszych praktyk dotyczących pracy z asynchronicznymi zasobami typu RESTful.

Wysyłanie kodu statusu 202 Accepted

W przypadku obsługi żądań i odpowiedzi asynchronicznych, jeśli żądanie jest poprawne i zasób może zostać zwrócony w ciągu kilku sekund, API powinno zwracać kod statusu `202 Accepted`. Kod ten oznacza, że żądanie zostało zaakceptowane do przetworzenia i zasób będzie wkrótce dostępny. Dodatkowo powinno się przesłać nagłówek `Location` z informacją dla klienta o miejscu udostępnienia tego zasobu. Jeśli odpowiedź nie jest dostępna natychmiast, nie należy zwracać kodu `201 Created`.

Ustawianie terminu wygaśnięcia dla obiektów w kolejce

Programista API powinien ustawiać termin wygaśnięcia obiektów znajdujących się w kolejce. Dzięki temu obiekty te nie będą się gromadzić i co jakiś czas będą usuwane.

Asynchroniczne obsługiwanie zadań przy użyciu kolejek wiadomości

Programista API powinien rozważyć możliwość użycia kolejek wiadomości do wykonywania operacji asynchronicznych, aby wiadomości oczekujące na odbiór przez adresata były przechowywane w kolejce. Niezawodnym standardem do trasowania, kolejkowania, publikowania i subskrybowania wiadomości jest protokół **AMQP** (ang. *Advanced Messaging Queing Protocol*). Więcej informacji o tym protokole można znaleźć na stronie *http://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol*.

Na przykład gdy zostanie wywołana metoda zasobu asynchronicznego, użyj kolejki wiadomości do wysyłania wiadomości i obsługi różnych zadań na podstawie wiadomości i zdarzeń.

W przedstawionym przykładzie, jeśli ktoś złoży zamówienie na kawę, można wysłać wiadomość przy użyciu narzędzia RabbitMQ (*http://www.rabbitmq.com/*) w celu wyzwolenia zdarzenia `COMPLETED`. Po wykonaniu zamówienia szczegóły można przenieść do systemu inwentarzowego.

W następnym podrozdziale znajduje się opis kolejnej ważnej techniki związanej z usługami typu RESTful — częściowych aktualizacji.

Metoda HTTP PATCH i częściowe aktualizacje

Typowym problemem, z którym mierzą się programiści interfejsów API, jest implementacja częściowych aktualizacji. Technika ta jest przydatna, gdy klient wyśle żądanie zmiany tylko części stanu zasobu. Na przykład wyobraź sobie następującą reprezentację zasobu `Coffee` w formacie JSON:

```
{
 "id": 1,
 "name": "Mocha"
 "size": "Small",
 "type": "Latte",
 "status":"PROCESSING"
}
```

Po zrealizowaniu zamówienia status PROCESSING trzeba zmienić na COMPLETED.

W API typu RPC można by było dodać następującą metodę:

```
GET myservice/rpc/coffeeOrder/setOrderStatus?completed=true&coffeeId=1234
```

W REST, jeśli użyto by metody PUT, konieczne byłoby wysłanie wszystkich informacji, co oznaczałoby marnowanie transferu i pamięci.

```
PUT /coffee/orders/1234
{
 "id": 1,
 "name": "Mocha"
 "size": "Small",
 "type": "Latte",
 "status": "COMPLETED"
}
```

Aby uniknąć konieczności wysyłania wszystkich danych, gdy potrzebna jest tylko drobna zmiana, można zastosować metodę PATCH do częściowej aktualizacji:

```
PATCH /coffee/orders/1234
{
 "status": "COMPLETED"
}
```

Ale nie wszystkie serwery i aplikacje klienckie obsługują tę metodę, przez co implementuje się także częściowe aktualizacje przy użyciu metod POST i PUT:

```
POST /coffee/orders/1234
{
 "status": "COMPLETED"
}
```

Użycie metody PUT:

```
PUT /coffee/orders/1234
{
 "status": "COMPLETED"
}
```

Podsumowując, do częściowych aktualizacji można używać zarówno metody PUT, jak i POST. W API Facebooka wykorzystywana jest metoda POST. Użycie metody PUT byłoby bardziej spójne ze sposobem implementacji zasobów RESTful i metod takich jak operacje CRUD.

Poniżej znajduje się adnotacja, którą należy dodać do JAX-RS w celu zaimplementowania obsługi metody PATCH:

```
@Target({ElementType.METHOD})
  @Retention(RetentionPolicy.RUNTIME)
```

```
  @HttpMethod("PATCH")
  public @interface PATCH {
}
```

Kod ten ilustruje sposób powiązania adnotacji `javax.ws.rs.HTTPMethod` z nazwą `PATCH`. Po dodaniu tego kodu można używać adnotacji `@PATCH` do oznaczania dowolnej metody zasobu JAX-RS.

JSON Patch

JSON Patch to metoda opisana w dokumencie RFC 6902. Jest to standard definiujący wykonywanie działań na dokumentach JSON. Metoda ta może być stosowana w połączeniu z metodą HTTP `PATCH` w celu częściowego aktualizowania dokumentów w formacie JSON. Do identyfikacji takich dokumentów używany jest typ mediów `application/json-patch+json`.

Oto lista składowych:

- `op` — określa rodzaj operacji, jaka ma zostać wykonana na dokumencie. Dostępne są następujące wartości: `add`, `replace`, `move`, `remove`, `copy` oraz `test`. Każda inna wartość oznacza błąd.
- `path` — wskaźnik JSON reprezentujący lokalizację w dokumencie JSON.
- `value` — oznacza wartość, która ma zostać zmieniona.

Operacja `move` przyjmuje składową `from`, określającą lokalizację w dokumencie docelowym, z którego ma być przeniesiona wartość.

Poniżej znajduje się przykład dokumentu JSON Patch wysłanego w żądaniu HTTP `PATCH`:

```
PATCH /coffee/orders/1234 HTTP/1.1
Host: api.foo.com
Content-Length: 100
Content-Type: application/json-patch

[
  {"op":"replace", "path": "/status", "value": "COMPLETED"}
]
```

Żądanie to ilustruje sposób użycia metody JSON Patch do zmiany statusu zamówienia kawy identyfikowanego przez zasób `coffee/orders/1234`. Zastosowano operację (`op`) zamiany (`replace`), która ustawia status obiektu w reprezentacji JSON na wartość `COMPLETED`.

Metoda JSON Patch jest bardzo przydatna w aplikacjach jednostronicowych, przy współpracy na bieżąco, gdy trzeba wprowadzać zmiany offline, oraz znajduje zastosowanie w programach wprowadzających drobne zmiany w dużych dokumentach. Szersze informacje na jej temat można znaleźć na stronie *http://jsonpatchjs.com/*, która zawiera implementację metod JSON Patch (RFC 6902) i JSON Pointer (RFC 6901) na licencji MIT License.

Zalecana lektura

Poniżej znajduje się lista dostępnych w internecie tekstów, w których można znaleźć dodatkowe informacje na tematy opisane w tym rozdziale:

- RESTEasy: *http://resteasy.jboss.org/*.
- Couchbase: *http://www.couchbase.com/*.
- Eksplorator API Graph portalu Facebook: *https://developers.facebook.com/*.
- RabbitMQ: *https://www.rabbitmq.com/*.
- Dokument RFC 6902 JSON Patch: *http://tools.ietf.org/html/rfc6902*.
- Dokument RFC 6901 JSON Pointer: *http://tools.ietf.org/html/rfc6901*.

Podsumowanie

Niniejszy rozdział zawiera podstawowe wiadomości na temat buforowania i używania nagłówków buforowania HTTP, takich jak `Cache-Control`, `Expires` itd. Ponadto pokazałam, jak używa się nagłówków `ETag` i `Last-Modified` do wysyłania warunkowych żądań `GET` w celu zoptymalizowania wydajności aplikacji. Opisałam też najlepsze praktyki buforowania, buforowanie po stronie serwera w RESTEasy oraz sposób wykorzystania nagłówków ETag w API Facebooka. W rozdziale tym zostało poruszone zagadnienie asynchronicznych zasobów RESTful oraz zostały przedstawione najlepsze praktyki dotyczące pracy z asynchronicznymi API. Na końcu znalazł się opis technik częściowego aktualizowania zasobów za pomocą metody JSON Patch (RFC 6902).

W następnym rozdziale przechodzę do zaawansowanych tematów, które powinien opanować każdy programista interfejsów API typu RESTful: ograniczania liczby żądań, stronicowania odpowiedzi oraz internacjonalizacji zasobów. Dodatkowo piszę parę słów o HATEOAS, REST i ich rozszerzalności.

ROZDZIAŁ 5

Zaawansowane zasady projektowania

W tym rozdziale opisane są zaawansowane zasady projektowania, które powinien znać każdy programista projektujący usługi typu RESTful. Ponadto znajdziesz tu praktyczne porady na temat budowy złożonych aplikacji z API REST.

W rozdziale opisane są następujące zagadnienia:

- Techniki ograniczania liczby żądań.
- Stronicowanie odpowiedzi.
- Internacjonalizacja i lokalizacja.
- Wtyczki i rozszerzalność usług REST.
- Inne kwestie dotyczące budowy API REST.

W rozdziale przedstawione są różne fragmenty kodu. Całe działające przykłady znajdują się w plikach z kodem źródłowym umieszczonych na serwerze FTP wydawnictwa.

Podobnie jak w poprzednich rozdziałach, staram się maksymalnie skracać opisy i pozostawiać tylko tyle informacji, abyś dobrze zrozumiał opisywane zagadnienia, a jednocześnie chcę podać wystarczająco dużo szczegółów, by można było natychmiast rozpocząć posługiwanie się przedstawianymi technikami.

Techniki ograniczania liczby żądań

Ograniczanie liczby żądań polega na wyznaczeniu maksymalnej liczby żądań, jaką może wykonać klient w określonym czasie. Do identyfikacji klienta można użyć tokenu dostępu zgodnie z opisem zamieszczonym w rozdziale 3. albo adresu IP.

Aby zapobiec nadmiernemu obciążeniu serwera, API musi mieć zaimplementowane techniki ograniczania liczby żądań. Aplikacja ograniczająca może zdecydować, czy przepuścić żądanie wysłane przez danego klienta, czy nie.

Dopuszczalna liczba żądań może być określona przez serwer i może wynosić np. 500 żądań na godzinę. Klient wysyła żądanie do serwera poprzez wywołanie API. Serwer sprawdza, czy nie został przekroczony limit, i jeśli nie, przepuszcza je oraz zwiększa wartość licznika o jeden. Jeżeli klient przekroczył limit, serwer może zgłosić błąd `429`.

Ewentualnie serwer może dodać do odpowiedzi nagłówek `Retry-After`, aby poinformować klienta, ile czasu ma odczekać, zanim wyśle kolejne żądanie.

Każde żądanie z aplikacji może przejść przez dwie przepustnice: z tokenem dostępu i bez tokenu dostępu. Przydział żądań dla aplikacji z tokenem dostępu może być inny od przydziału dla aplikacji niemającej takiego tokenu.

Oto szczegółowe informacje o kodzie błędu HTTP `429 Too Many Requests`.

> **429 Too Many Requests (RFC 6585)**
>
> Użytkownik wysłał za dużo żądań w określonej jednostce czasu. Kodu tego powinno się używać w mechanizmach ograniczania liczby żądań.

W odpowiedzi z kodem `429 Too Many Requests` może znajdować się dodatkowy nagłówek `Retry-After` określający, ile czasu klient powinien odczekać przed wysłaniem kolejnego żądania. Poniżej znajduje się przykład takiej odpowiedzi:

```
HTTP/1.1 429 Too Many Requests
Content-Type: text/html
Retry-After: 3600
 <html>
    <head>
    <title>Zbyt duża liczba żądań</title>
    </head>
 <body>
    <h1>Zbyt duża liczba żądań</h1>
    <p>Maksymalna liczba żądań do tej strony wynosi 100 na godzinę.</p>
 </body>
</html>
```

W odpowiedzi tej ustawiono nagłówek `Retry-After` na 3600 sekund, aby poinformować klienta, że powinien spróbować później. Ponadto serwery mogą wysyłać nagłówek `X-RateLimit-Remaining` informujący o tym, ile jeszcze żądań pozostało klientowi.

Wiesz już mniej więcej, na czym polega ograniczanie liczby żądań oraz do czego służą nagłówki `Retry-After` i `X-RateLimit-Remaining`, więc możemy przejść do przykładów kodu JAX-RS.

W poniższym podrozdziale znajdują się przykłady ilustrujące sposób implementacji prostego filtru ograniczającego liczbę żądań w JAX-RS.

Układ projektu

Rozkład katalogów projektu ma standardową strukturę Maven, której bardzo zwięzły opis znajduje się w poniższej tabeli. W przykładzie jest tworzony plik WAR, który można wdrożyć na każdym serwerze aplikacji obsługującym Javę EE 7, np. GlassFish 4.0.

Przykład demonstruje prostą kawiarnię, w której klient może zapytać o złożone przez siebie zamówienie.

Kod źródłowy	Opis
src/main/java	Katalog zawierający wszystkie zasoby wymagane przez aplikację kawiarni.

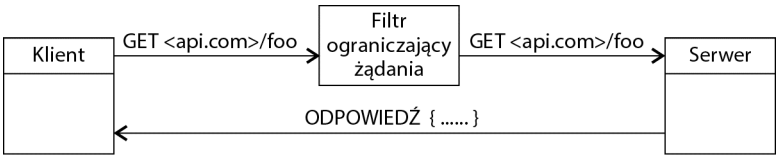
Klasa `CoffeesResource` to prosty zasób JAX-RS:

```
@Path("v1/coffees")
public class CoffeesResource {
    @GET
    @Path("{order}")
    @Produces(MediaType.APPLICATION_XML)
    @NotNull(message="Nie istnieje kawa dla zamówienia o identyfikatorze
      requested")
    public Coffee getCoffee(@PathParam("order") int order) {
        return CoffeeService.getCoffee(order);
    }
}
```

Projekt zawiera klasę `CoffeesResource` służącą do pobierania informacji o zamówieniach kawy. Metoda `getCoffee()` zwraca obiekt klasy `Coffee` zawierający dane zamówienia.

W celu wymuszenia ograniczeń liczby żądań dodamy klasę `RateLimiter`, która będzie prostym filtrem serwletów, jak pokazano na poniższym schemacie.

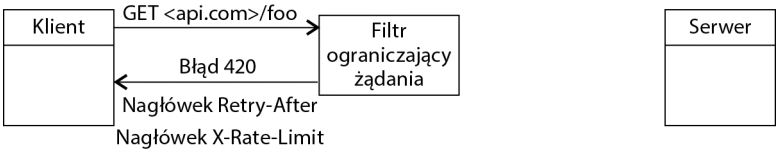
Klasa `RateLimiter` sprawdza adres IP klienta oraz to, czy klient ten nie wysłał zbyt dużej liczby żądań. Poniżej znajduje się schematyczne ujęcie tego procesu:



Przypadek 1. Żądanie klienta nie przekracza limitu

Na tym schemacie przedstawiono proces wysłania przez klienta żądania `GET` na adres *http://api.com/foo*. **Filtr ograniczający żądania** sprawdza, ile żądań wykonał już klient reprezentowany przez dany adres IP. Ponieważ nie przekroczył on jeszcze limitu, żądanie zostaje przekazane do serwera, który może zwrócić odpowiedź w formacie XML, JSON lub tekstowym.

Poniższy schemat również przedstawia proces wysłania żądania `GET` na adres *http://api.com/foo*. **Filtr ograniczający żądania** sprawdza, ile żądań wykonał już klient reprezentowany przez dany adres IP. Jako że klient ten przekroczył limit, żądanie nie zostaje przekazane do serwera, tylko następuje zwrócenie przez filtr w odpowiedzi HTTP kodu błędu `429 Too Many Requests`.



Przypadek 2. Żądanie klienta przekracza limit

Szczegółowa analiza przykładu ograniczania liczby żądań

Aby zaimplementować mechanizm ograniczania liczby żądań w JAX-RS, należy zaimplementować klasę `Filter`, jak pokazano w poniższym przykładzie:

```
@WebFilter(filterName = "RateLimiter",
       urlPatterns = {"/*"}
       )
public class RateLimiter implements Filter {
    private static final int REQ_LIMIT = 3;
    private static final int TIME_LIMIT = 600000;
    private static AccessCounter accessCounter = AccessCounter.getInstance();
}
```

Jest to implementacja interfejsu `WebFilter` z pakietu `javax.servlet.annotation`. Adnotacja `@WebFilter` oznacza, że klasa jest filtrem dla aplikacji.

Adnotacja `@WebFilter` musi mieć przynajmniej jeden z atrybutów `urlPatterns` lub `value`.

Stała `REQ_LIMIT` oznacza dozwoloną liczbę żądań w określonym czasie. Stała `TIME_LIMIT` oznacza czas, po jakim klientowi jest przyznawana nowa pula żądań.

Dla uproszczenia w przykładach ustawiono niewielkie limity. W prawdziwych aplikacjach byłyby to wartości rzędu 60 żądań na minutę albo 1000 dziennie. Jeżeli limit żądań zostanie wyczerpany, w nagłówku `Retry-After` zostanie przesłana informacja, za ile czasu klient może znowu wysyłać żądania do serwera.

Do zapamiętywania, ile żądań wykonał każdy klient, została utworzona klasa o nazwie `AccessCounter`, której kod źródłowy znajduje się poniżej. Jest to klasa singletonowa z adnotacją `@Singleton`. Tworzy ona strukturę `ConcurrentHashMap`, w której przechowuje adresy IP jako klucze i dane klientów `AccessData` jako wartości.

```
@Singleton
public class AccessCounter {

    private static AccessCounter accessCounter;

    private static ConcurrentHashMap<String,AccessData> accessDetails =
      new ConcurrentHashMap<String, AccessData>();
}
```

Klasa `AccessData` służy do zapisywania danych klientów, a konkretnie liczby wykonanych żądań i terminu ostatniego żądania. Jest to zwykła klasa Javy, której kod źródłowy pokazano poniżej:

```
public class AccessData {
    private long lastUpdated;
    private AtomicInteger count;

    public long getLastUpdated() {
        return lastUpdated;
    }

    public void setLastUpdated(long lastUpdated) {
        this.lastUpdated = lastUpdated;
    }

    public AtomicInteger getCount() {
        return count;
    }

    public void setCount(AtomicInteger count) {
        this.count = count;
    }

...
```

Jak widać na powyższym listingu, klasa `AccessData` zawiera pola o nazwach `count` i `lastUpdated`. Gdy pojawia się nowe żądanie, następuje zwiększenie wartości pola `count` oraz ustawienie wartości pola `lastUpdated` na bieżącą godzinę.

Metoda `doFilter()` klasy `RateLimiter` jest używana w poniższym fragmencie kodu:

```
@Override
    public void doFilter(ServletRequest servletRequest,
        ServletResponse servletResponse,
            FilterChain filterChain) throws IOException, ServletException {

        HttpServletRequest httpServletRequest = (HttpServletRequest)
          servletRequest;
        HttpServletResponse httpServletResponse = (HttpServletResponse)
          servletResponse;

        String ipAddress = getIpAddress(httpServletRequest);
        if (accessCounter.contains(ipAddress)) {
        if (!requestLimitExceeded(ipAddress)) {
            accessCounter.increment(ipAddress);
            accessCounter.getAccessDetails(ipAddress)
              .setLastUpdated(System.currentTimeMillis());

        } else {

            httpServletResponse.addIntHeader("Retry-After",TIME_LIMIT);
            httpServletResponse.sendError(429);

        }
    } else {
        accessCounter.add(ipAddress);
    }
    filterChain.doFilter(servletRequest, servletResponse)

}
```

Kod ten ilustruje sposób użycia metody `doFilter()` interfejsu `javax.servlet.Filter` zaimplementowanej w klasie `RateLimiter`. Pierwszą czynnością, jaką wykonuje ta metoda, jest sprawdzenie adresu IP użytkownika.

Jeśli obiekt klasy `accessCounter` zawiera adres IP klienta, następuje sprawdzenie za pomocą metody `requestLimitExcedeed()`, czy nie doszło do przekroczenia limitu żądań.

Jeżeli limit został przekroczony, w odpowiedzi `httpServletResponse` zostaje zwrócony nagłówek `Retry-After` wraz z kodem błędu `429 Too Many Requests`. Jeśli nadejdzie nowe żądanie od tego samego klienta po pewnym czasie dłuższym niż wartość `TIME_LIMIT`, licznik zostaje wyzerowany i znowu żądania tego klienta są obsługiwane.

Poniżej znajduje się lista nagłówków używanych w technikach ograniczania liczby żądań, które można wysyłać w odpowiedziach do klientów:

- `X-RateLimit-Limit` — maksymalna liczba żądań, jaką klient może wysłać w określonym czasie.
- `X-RateLimit-Remaining` — pozostała liczba żądań, jaką można wysłać w bieżącym okresie.

Bardziej szczegółowy przykład znajduje się w plikach z kodem źródłowym do pobrania z serwera FTP wydawnictwa. Gdy program zostanie wdrożony na serwerze aplikacji, klient może wysłać wiele żądań danych zamówień kaw.

Dla ułatwienia ustawimy limit trzech żądań i 10 minut. Poniżej znajduje się przykładowe żądanie `curl`:

```
curl -i http://localhost:8080/ratelimiting/v1/coffees/1
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source
Edition 4.0 Java/Oracle Corporation/1.7)
Server: GlassFish Server Open Source Edition 4.0
Content-Type: application/json
Date: Mon, 23 Jun 2014 23:27:34 GMT
Content-Length: 57

{
  "name":"Mocha",
  "order":1,
  "size":"Small",
  "type":"Brewed"
}
```

Po przekroczeniu limitu żądań pojawia się błąd `429`:

```
curl -i http://localhost:8080/ratelimiting/v1/coffees/1
HTTP/1.1 429 CUSTOM
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source
Edition 4.0 Java/Oracle Corporation/1.7)
Server: GlassFish Server Open Source Edition 4.0
Retry-After: 600000
Content-Language:
Content-Type: text/html
Date: Mon, 23 Jun 2014 23:29:04 GMT
Content-Length: 1098
```

W przykładzie pokazano, jak tworzy się niestandardowe filtry implementujące mechanizm ograniczania liczby żądań. Inną możliwością jest użycie otwartego projektu o nazwie **Repose**. Jest to skalowalna i rozbudowana implementacja do ograniczania liczby żądań. Repose to kolejna usługa pośrednicząca HTTP umożliwiająca ograniczanie liczby żądań, uwierzytelnianie klientów, wersjonowanie itd. Więcej informacji o niej można znaleźć na stronie *http://openrepose.org/*.

W następnym podrozdziale znajduje się opis najlepszych praktyk pozwalających zapobiec przekroczeniu limitu żądań przez klienty API REST.

Najlepsze praktyki pozwalające uniknąć przekroczenia limitu żądań przez klienty

W podrozdziale przedstawiam najlepsze praktyki, których stosowanie pozwala uniknąć przekroczenia limitu żądań przez klienty API REST.

Buforowanie

Buforowanie odpowiedzi API na serwerze pozwala zredukować liczbę żądań. Jeśli ustawi się odpowiedni termin ważności elementów w buforze, można zapobiec zasypaniu bazy danych lawiną zapytań, ponieważ odpowiedzi zawierające niezmienione zasoby będą wysyłane z bufora. Na przykład aplikacja wyświetlająca tweety z Twittera może buforować odpowiedzi z API lub wykorzystać API Streaming Twittera (opisane w następnym podrozdziale). Klienty API nie powinny wysyłać identycznych żądań częściej niż raz na minutę, ponieważ w większości przypadków jest to marnotrawstwo transferu.

Niewysyłanie żądań w pętlach

Żądań API nie powinno się wysyłać w pętli. API serwera powinno dostarczać jak najwięcej informacji, tak aby klient nie musiał wysyłać zbyt wielu żądań w celu zdobycia wszystkich potrzebnych mu danych. Oznacza to, że klient może otrzymać w odpowiedzi na jedno żądanie całą kolekcję obiektów, zamiast pobierać je po kolei za pomocą żądań wysyłanych przy użyciu pętli.

Rejestrowanie żądań w dzienniku

Dobrym zwyczajem jest rejestrowanie żądań u klienta, aby było wiadomo, ile zostało ich już wysłanych. Analizując dzienniki, klient może sprawdzić, które żądania są zbędne i tylko wyczerpują limit.

Unikanie sondowania

Konsumenci nie powinni też sondować zasobów po to, by dowiedzieć się, czy zaszły jakieś zmiany. Zamiast tego lepiej jest użyć uchwytów sieciowych (*http://en.wikipedia.org/wiki/Webhook*) lub powiadomień serwerowych (*http://en.wikipedia.org/wiki/Push_technology*). Szerzej na temat uchwytów sieciowych piszę w rozdziale 6.

Wykorzystanie API strumieniowania

Programiści API mogą wykorzystać strumieniowanie. Technika ta również pozwala zredukować liczbę żądań wysyłanych przez klienty. Portal Twitter udostępnia zestaw API strumieniowania o krótkim czasie dostępu do globalnego strumienia tweetów. Klient strumienia nie musi marnować czasu na oczekiwanie związane z sondowaniem punktu końcowego REST i otrzymuje wiadomości o tweetach i innych zdarzeniach.

Gdy aplikacja ustanowi połączenie ze strumieniowym punktem końcowym, zacznie otrzymywać tweety bez potrzeby sondowania i uwzględniania limitów liczby żądań.

Studium przypadku limitów liczby żądań w API REST Twittera

W Twitterze jest ustawiony limit 150 żądań na godzinę dla nieuwierzytelnionych klientów.

Wywołania OAuth mają limit 350 na godzinę na podstawie tokenu w żądaniu.

Jeśli aplikacja przekroczy limit API wyszukiwania, otrzyma w odpowiedzi kod HTTP 420. Najlepszym rozwiązaniem jest sprawdzanie tego błędu i honorowanie zwracanego z nim nagłówka `Retry-After`. Wartość tego nagłówka oznacza liczbę sekund, jaką aplikacja kliencka musi odczekać, zanim po raz kolejny wyśle żądanie do API wyszukiwania. Jeżeli klient wyśle więcej żądań niż dozwolone w ciągu godziny, otrzyma w odpowiedzi kod `420 Enhance Your Calm`.

Kod 420 Enhance Your Calm w Twitterze

Kod ten nie należy do standardu HTTP, ale jest zwracany przez API wyszukiwania i trendów Twittera, gdy klient przekracza limit liczby żądań. Zamiast niego w aplikacjach powinno się implementować kod `429 Too Many Requests`.

Stronicowanie odpowiedzi

API REST są wykorzystywane przez rozmaite systemy, od klientów sieciowych po urządzenia przenośne, i dlatego odpowiedzi zwrotne zawierające wiele elementów powinny być dzielone na strony z określoną liczbą pozycji. Technikę dzielenia na strony nazywa się stronicowaniem odpowiedzi. Do odpowiedzi powinno się też zawsze dołączać informację o tym, ile jest wszystkich elementów, stron, oraz łącza do następnych zbiorów wyników. Konsumenci mogą definiować indeksy stron do pytania o wyniki i liczbę wyników na stronie.

Zaleca się zdefiniowanie i opisanie w dokumentacji domyślnej liczby wyników na stronę, która zostanie zwrócona, jeśli klient nie określi jej sam. Na przykład w API REST portalu GitHub domyślny rozmiar strony wynosi 30 rekordów, a maksymalny — 100. Jest też ustawiony limit zapytań do API. Jeżeli API ma określony domyślny rozmiar strony, to łańcuch zapytania może określać tylko indeks strony.

W paru następnych podrozdziałach znajduje się opis różnych technik stronicowania. Programista API w zależności od potrzeby może zaimplementować jedną lub wiele z nich.

Rodzaje stronicowania

Wyróżnia się trzy techniki stronicowania:

- offsetowe,
- czasowe,
- kursorowe.

Stronicowanie offsetowe

Stronicowanie offsetowe polega na zwracaniu klientowi określonego numeru strony z konkretną liczbą wyników. Na przykład jeśli klient chce pobrać wszystkie informacje o wypożyczonych książkach albo zamówieniach kawy, może wysłać następujące żądanie:

```
GET v1/coffees/orders?page=1&limit=50
```

W poniższej tabeli znajduje się opis parametrów zapytania, jakich można by było używać w stronicowaniu offsetowym:

Parametr zapytania	Opis
`page`	Określa numer strony do zwrócenia.
`limit`	Maksymalna liczba wyników na stronie, jaka może zostać przesłana w odpowiedzi.

Stronicowanie czasowe

Technika stronicowania czasowego ma zastosowanie wtedy, gdy klient zażąda zbioru wyników z określonego okresu.

Na przykład aby pobrać listę zamówień kawy złożonych w określonym czasie, klient może wysłać następujące zapytanie:

```
GET v1/coffees/orders?since=140358321&until=143087472
```

W poniższej tabeli znajduje się opis parametrów zapytania, jakich można by było używać w stronicowaniu czasowym:

Parametr zapytania	Opis
`until`	Uniksowy znacznik czasu oznaczający koniec okresu.
`since`	Uniksowy znacznik czasu oznaczający początek okresu.
`limit`	Maksymalna liczba wyników na stronie, jaka może zostać przesłana w odpowiedzi.

Stronicowanie kursorowe

Stronicowanie kursorowe to technika polegająca na dzieleniu wyników na strony przez kursor. Po wynikach takich można poruszać się w przód i do tyłu za pomocą kursorów dostarczonych w odpowiedzi.

API stronicowania kursorowego unika zwracania duplikatów rekordów w przypadkach, gdy między żądaniami stronicowania zostaną dodane nowe zasoby. Dzieje się to za sprawą tego, że parametr kursora jest wskaźnikiem oznaczającym miejsce, od którego należy wznowić zwracanie zasobów w kolejnym żądaniu.

Twitter i stronicowanie kursorowe

Poniżej przedstawiam przykład użycia stronicowania kursorowego w portalu Twitter. Wynik zapytania pobierającego identyfikatory użytkowników, którzy mają dużą liczbę obserwatorów, mógłby być stronicowany i zostać zwrócony w następującym formacie:

```
{
    "ids": [
        385752029,
        602890434,
        ...
        333181469,
        333165023
    ],
    "next_cursor": 1374004777531007833,
    "next_cursor_str": "1374004777531007833",
    "previous_cursor": 0,
    "previous_cursor_str": "0"
}
```

Wartość `next_cursor` można przekazać do następnego zapytania, aby pobrać kolejny zbiór wyników:

```
GET https://api.twitter.com/1.1/followers/ids.json?screen_name=someone
&cursor=1374004777531007833
```

Przy użyciu wartości `next_cursor` i `previous_cursor` można łatwo poruszać się między zbiorami wyników.

Znasz już ogólne zasady stronicowania wyników, więc możemy przejść do szczegółowej analizy konkretnego przykładu. W następnym podrozdziale znajduje się opis prostej implementacji techniki stronicowania offsetowego przy użyciu JAX-RS.

Układ projektu

Rozkład katalogów projektu ma standardową strukturę Maven, której bardzo zwięzły opis znajduje się w poniższej tabeli.

Przykład ten demonstruje prostą usługę kawiarni, do której można wysłać zapytanie o wszystkie złożone do tej pory zamówienia.

Kod źródłowy	Opis
src/main/java	Katalog zawierający wszystkie zasoby wymagane przez aplikację kawiarni.

Poniżej znajduje się kod źródłowy klasy `CoffeesResource`:

```
@Path("v1/coffees")
public class CoffeesResource {
    @GET
    @Path("orders")
    @Produces(MediaType.APPLICATION_JSON)
    public List<Coffee> getCoffeeList(@QueryParam("page") @DefaultValue("1")
      int page, @QueryParam("limit") @
        DefaultValue("10") int limit ) {
        return CoffeeService.getCoffeeList( page, limit);
    }
}
```

Metoda `getCoffeeList()` pobiera dwie wartości typu `QueryParam`: `page` i `limit`. Wartość `page` odpowiada indeksowi strony, a `limit` — liczbie wyników na stronę. Adnotacja `@DefaultValue` określa domyślne wartości, które zostaną użyte w przypadku braku parametrów zapytania.

Poniżej znajduje się wynik wykonania przykładowego zapytania. Element `metadata` zawiera wartość `totalCount` oznaczającą liczbę wszystkich rekordów, atrybut `links` tablicy `JSONArray` z elementami `self`, oznaczającymi bieżącą stronę, i `next`, który jest łączem do pobrania większej liczby wyników.

```
{
    "metadata": {
        "resultsPerPage": 10,
        "totalCount": 100,
        "links": [
            {
                "self": "/orders?page=1&limit=10"
            },
            {
                "next": "/orders?page=2&limit=10"
            }
        ]
```

```
    },
    "coffees": [
        {
            "Id": 10,
            "Name": "Espresso",
            "Price": 2.77,
            "Type": "Hot",
            "Size": "Large"
        },
        {
            "Id": 11,
            "Name": "Cappuccino",
            "Price": 0.14,
            "Type": "Brewed",
            "Size": "Large"
        },
...
        ......
    ]
}
```

Cały kod można znaleźć w plikach źródłowych do pobrania z serwera FTP.

> W API REST zawsze dobrym pomysłem jest zdefiniowanie domyślnej liczby wyników na stronę. Ponadto zaleca się dodawanie do odpowiedzi metadanych, przy użyciu których konsument będzie mógł bez problemu pobrać następny zbiór wyników.

Internacjonalizacja i lokalizacja

Usługi często działają w środowiskach międzynarodowych, więc odpowiedzi powinny być dostosowane do kraju i miejsca, w którym ich zażądano. Parametry lokalizacji można zdefiniować w:

- nagłówkach HTTP,
- parametrach zapytań,
- treści odpowiedzi REST.

Negocjacja języka przypomina negocjację treści. Wartością nagłówka HTTP `Accept-Language` może być dwuliterowy kod oznaczający język w standardzie ISO-3166 (*http://www.iso.org/iso/country_codes.htm*). Nagłówek ten jest podobny do nagłówka `Content-Type` i może określać język odpowiedzi.

Na przykład poniżej znajduje się nagłówek `Content-Language` wysłany w odpowiedzi na żądanie przesłane przez klienta:

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source
Edition 4.0 Java/Oracle Corporation/1.7)
Server: GlassFish Server Open Source Edition 4.0
Content-Language: en
Content-Type: text/html
Date: Mon, 23 Jun 2014 23:29:04 GMT
Content-Length: 1098
```

W tej odpowiedzi nagłówek `Content-Language` ma wartość `en`.

W JAX-RS do obsługi negocjacji treści w trakcie działania programu służą obiekty typu `Request` i klasa `javax.ws.rs.core.Variant`. Klasa `Variant` może zawierać typ mediów, język i kodowanie. Klasa `Variant.VariantListBuilder` służy do tworzenia listy wariantów reprezentacji.

Poniższy fragment kodu ilustruje sposób utworzenia listy wariantów reprezentacji zasobu:

```
List<Variant> variantList = Variant.
        .languages("en", "fr").build();
```

W kodzie znajduje się wywołanie metody `build()` klasy `VariantListBuilder` z językami `en` i `fr`.

Parametry zapytania mogą zawierać informacje o lokalizacji, na podstawie których serwer może zwrócić dane w odpowiednim języku, np.:

```
GET v1/books?locale=fr
```

Jest to przykład zapytania pobierającego dane książek zawierającego w parametrze informację o lokalizacji. Ponadto treść odpowiedzi REST również może zawierać informacje dotyczące kraju, np. kod waluty i inne szczegóły zdobyte na podstawie nagłówków HTTP i parametrów zapytania żądań.

Różne tematy

Poniżej opisuję parę różnych zagadnień, takich jak HATEOAS i rozszerzalność technologii REST.

HATEOAS

HATEOAS (ang. *Hypermedia as the Engine of Application State*) to ograniczenie architektury aplikacji REST.

API bazujące na hipermediach przekazuje informacje o dostępnych interfejsach API i tym, jakie czynności może wykonać konsument, poprzez dostarczenie łączy hipertekstowych w odpowiedzi przesłanej przez serwer.

Na przykład reprezentacja książki dla zasobu REST zawierająca takie dane jak tytuł i numer ISBN może wyglądać tak:

```
{
    "Name": "Developing RESTful Services with JAX-RS 2.0,
             WebSockets, and JSON",
    "ISBN": "1782178120"
}
```

Implementacja HATEOAS mogłaby zwrócić następujące dane:

```
{
    "Name": "Developing RESTful Services with JAX-RS 2.0,
             WebSockets, and JSON",
    "ISBN": "1782178120"
    "links": [
      {
       "rel": "self",
       "href": "http://packt.com/books/123456789"
      }
    ]
}
```

W tym przykładzie element `links` zawiera obiekty JSON `rel` i `href`.

Atrybut `rel` w tym przypadku jest hiperłączem odnoszącym się do tego samego dokumentu. W bardziej skomplikowanych systemach mogłyby znaleźć się także inne relacje. Na przykład zamówienie książki mogłoby mieć relację `"rel":"customer"` łączącą zamówienie książki z klientem. Atrybut `href` zawiera adres URL jednoznacznie identyfikujący zasób.

Zaletą HATEOAS jest to, że pomaga programistom klientów w eksplorowaniu protokołu. Łącza stanowią wskazówkę na temat tego, jaką czynność można wykonać w następnej kolejności. Choć nie ma standardu opisującego kontrolki hipermedialne, zaleca się przestrzeganie zasad opisanych w standardzie RFC ATOM (4287).

> Zgodnie z modelem dojrzałości Richardsona HATEOAS uważa się za ostatni poziom REST. Oznacza to, że każde łącze powinno implementować standardowe czasowniki REST: `GET`, `POST`, `PUT` i `DELETE`. Dodatkowe informacje, takie jak zawarte w elemencie `links` widocznym w poprzednim przykładzie kodu, są dla klienta wskazówką, jak poruszać się po usłudze i jakie kolejne czynności podejmować.

API REST portalu PayPal i HATEOAS

API REST portalu PayPal obsługuje HATEOAS, więc w każdej odpowiedzi znajduje się zbiór łączy, przy użyciu których klient może zdecydować, co robić dalej.

Poniżej przedstawiam przykładową odpowiedź z API REST portalu PayPal zawierającą obiekty JSON:

```
{
    "href": "https://www.sandbox.paypal.com/webscr?cmd=
      _express-checkout&token=EC-60U79048BN7719609",
    "rel": "approval_url",
    "method": "REDIRECT"
  },
  {
    "href": "https://api.sandbox.paypal.com/v1/payments/
      payment/PAY-6RV70583SB702805EKEYSZ6Y/execute",
    "rel": "execute",
    "method": "POST"
}
```

Krótki opis atrybutów:

- `href` — zawiera adresy URL, przy użyciu których można wykonywać kolejne wywołania API.
- `rel` — relacja tego łącza z poprzednimi wywołaniami API.
- `method` — określa metodę, przy użyciu której ma zostać wykonane wywołanie API.

> Więcej informacji na ten temat znajduje się na stronie *https://developer.paypal.com/docs/integration/direct/paypal-rest-payment-hateoas-links/*.

REST i rozszerzalność

Aplikacje typu RESTful są bardziej rozszerzalne i łatwiejsze w obsłudze. Jeśli dodatkowo są zbudowane na podstawie ograniczeń stylu projektowego, łatwo je zrozumieć i z nimi pracować. Są proste i przewidywalne, gdyż wszystko kręci się wokół zasobów. Ponadto z aplikacjami typu RESTful łatwiej się pracuje niż z aplikacjami XML-RPC, w których konsument musi analizować skomplikowane dokumenty WSDL, aby w ogóle zrozumieć, co się dzieje.

Inne tematy związane z API REST

Poniżej znajduje się opis paru innych zagadnień, które mogą interesować programistów REST. W poprzednich rozdziałach zostały opisane takie tematy jak projektowanie usług typu RESTful, obsługa błędów, weryfikacji danych, uwierzytelnianie, buforowanie oraz ograniczanie liczby żądań. W tym podrozdziale skupiam się na dodatkowych narzędziach pomocnych przy testowaniu i dokumentowaniu API REST.

Testowanie usług typu RESTful

Zawsze powinno się mieć zestaw automatycznych testów do weryfikowania odpowiedzi otrzymywanych od serwera. Jednym z systemów szkieletowych do budowy takich testów dla usług typu RESTful jest REST Assured.

REST Assured to specjalistyczny język oparty na Javie służący do testowania usług RESTful. Obsługuje metody `GET`, `PUT`, `POST`, `HEAD`, `OPTIONS` i `PATCH` i można go używać do weryfikowania poprawności odpowiedzi przesyłanych przez serwer.

Poniżej znajduje się przykładowy kod pobierający zamówienie kawy i sprawdzający zwrócony w odpowiedzi identyfikator:

```
get("order").
then().assertThat().
body("coffee.id",equalTo(5));
```

W przykładzie tym zostało wysłane wywołanie w celu pobrania zamówienia i sprawdzono, czy wartość `coffee.id` wynosi `5`.

Za pomocą języka REST Assured można określać i sprawdzać np. parametry, nagłówki, ciasteczka i treść główną. Ponadto możliwe jest mapowanie obiektów Javy na formaty JSON i XML i odwrotnie. Więcej informacji na temat tego narzędzia znajduje się na stronie *https://code.google.com/p/rest-assured/*.

Dokumentowanie usług typu RESTful

Każda usługa typu RESTful powinna mieć dokumentację niezależnie od tego, czy jest przeznaczona na użytek wewnętrzny firmy, czy dla klientów zewnętrznych w postaci aplikacji sieciowych lub urządzeń przenośnych. Poniżej przedstawiam system szkieletowy ułatwiający tworzenie dobrej jakości dokumentacji usług typu RESTful.

Swagger to system szkieletowy do opisywania, tworzenia, konsumowania i wizualizowania usług sieciowych typu REST. Dokumentacja metod, parametrów i modeli jest ściśle zintegrowana z kodem na serwerze. Swagger jest niezależny od języka programowania i istnieją jego implementacje dla języków Scala, Java oraz HTML 5.

Na stronie *https://github.com/wordnik/swagger-core/wiki/Adding-Swagger-to-yourAPI* znajduje się samouczek na temat dodawania Swaggera do API REST.

Zalecana lektura

Poniżej znajduje się lista dostępnych w internecie tekstów, w których można znaleźć dodatkowe informacje na tematy opisane w tym rozdziale:

- *https://dev.twitter.com/docs*:dokumentacja API Twittera.
- *https://dev.twitter.com/console*: konsola programistyczna Twittera.
- *https://dev.twitter.com/docs/rate-limiting/1.1*: ograniczenia liczby żądań w API v1.1 Twittera.
- *https://dev.twitter.com/docs/misc/cursoring*: API Twittera i kursory.
- *https://dev.twitter.com/docs/api/streaming*: API strumieniowe Twittera.
- *https://developers.facebook.com/docs/reference/ads-api/api-rate-limiting/*: ograniczenia liczby żądań w API Facebooka.
- *https://developer.github.com/v3/rate_limit/*: ograniczenia liczby żądań w API portalu GitHub.
- *https://developers.facebook.com/docs/opengraph/guides/internationalization/*: lokalizacja Facebooka.

Podsumowanie

W tym rozdziale zostały opisane zaawansowane zagadnienia, które powinien znać każdy programista tworzący API REST. Na początku pokazałam przykład ograniczania liczby żądań demonstrujący sposób kontrolowania komunikacji z serwerem tak, aby zapobiec jego przeciążeniu. Następnie przyjrzeliśmy się rozwiązaniom z zakresu ograniczania liczby żądań zastosowanym w API portali Twitter, GitHub i Facebook. Dalej opisałam różne techniki stronicowania i przedstawiłam parę prostych przykładów ich zastosowania. W kolejnej części rozdziału poruszone zostały różne zagadnienia, takie jak internacjonalizacja czy HATEOAS, czyli następny poziom API REST, a także rozszerzalność.

W następnym rozdziale opisuję nowe standardy, które dopiero zdobywają popularność (gniazda sieciowe i uchwyty sieciowe), oraz omawiam rolę technik REST w odniesieniu do przyszłych standardów sieciowych.

ROZDZIAŁ 6

Nowe standardy i przyszłość technologii REST

W tym rozdziale znajduje się opis nowych standardów i pojawiających się technologii, które mogą pozwolić na wzbogacenie funkcjonalności usług RESTful i rzucenie okiem na przyszłość technologii REST oraz inne rodzaje API na bieżąco współpracujące z użytkownikiem. Znajdziesz tu opis paru API i dowiesz się, jak mogą pomóc wyeliminować niektóre starsze techniki, takie jak np. sondowanie. Zważywszy na wielką popularność takich platform jak Twitter, Facebook czy Stripe, nie ma co się dziwić, że ich twórcy podążają za najnowszymi trendami i publikują działające na bieżąco interfejsy API, które dostarczają klientowi informacji w reakcji na zdarzenia.

W rozdziale opisane są następujące zagadnienia:

- API reagujące na bieżąco.
- Sondowanie.
- Uchwyty sieciowe.
- Gniazda sieciowe.
- Inne API i technologie do komunikacji na bieżąco:
  - PubSubHubbub,
  - zdarzenia wysyłane przez serwer,
  - XMPP,
  - BOSH za pośrednictwem XMPP.

- Studia przypadków firm używających uchwytów i gniazd sieciowych.
- Porównanie uchwytów i gniazd sieciowych.
- REST i mikrousługi.

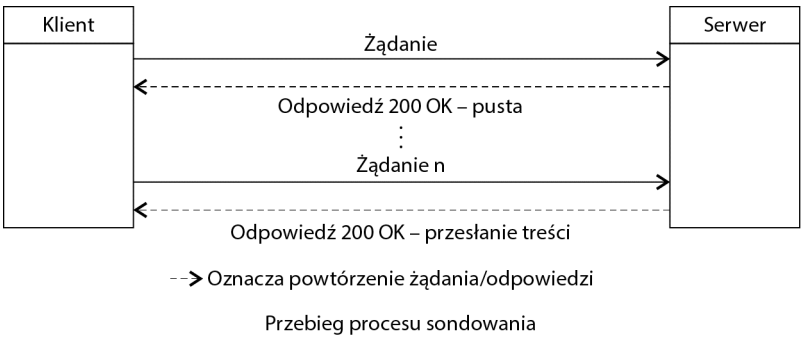
Zaczniemy od definicji interfejsu API reagującego na bieżąco, a potem przejdziemy do sondowania i wad tej techniki. Później przejrzymy różne modele powszechnie stosowane do asynchronicznej komunikacji na bieżąco i szczegółowo przeanalizujemy praktyczne rozwiązania z wykorzystaniem gniazd i uchwytów sieciowych.

API reagujące na bieżąco

W naszym kontekście API reagujące na bieżąco to taki interfejs, który pozwala konsumentowi na odbieranie interesujących go zdarzeń wtedy, gdy wystąpią. Przykładem tego typu aktualizacji jest sytuacja, gdy ktoś publikuje odnośnik na Facebooku albo ktoś obserwuje na Twitterze tweety na określony temat. Innym przykładem jest subskrypcja kanału wiadomości o zmianach cen akcji firmy.

Sondowanie

Sondowanie (ang. *polling*) to najbardziej tradycyjna metoda pobierania danych ze źródła, które wytwarza strumień zdarzeń i aktualizacji. Klient co pewien czas wysyła żądanie, a serwer w odpowiedzi przekazuje dane, jeśli są. Jeżeli nie ma nic do wysłania, zostaje zwrócona pusta odpowiedź. Proces ciągłego sondowania został przedstawiony na poniższym schemacie.



Przebieg procesu sondowania

Sondowanie ma wiele wad, jedną z nich jest np. wysyłanie pustych odpowiedzi, gdy na serwerze nic się nie zmieniło, przez co marnuje się transfer i moc obliczeniową. Jeśli klient będzie sondował ze zbyt małą częstotliwością, może stracić jakieś ważne informacje, a jeśli będzie to robił za często, może marnować zasoby i wyczerpać ustawiony na serwerze limit liczby żądań.

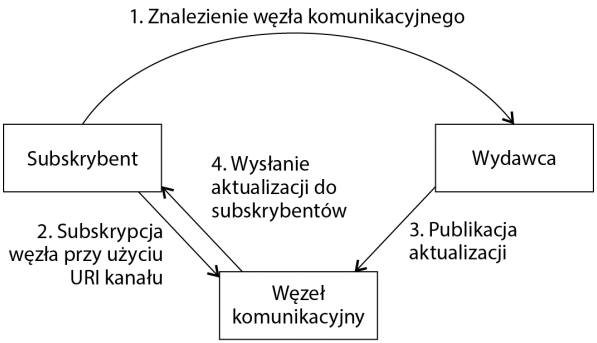
Wymienione problemy można wyeliminować, stosując:

- model PuSH — PubSubHubbub,
- model strumieniowy.

Model PuSH — PubSubHubbub

PuSH to prosta technika oparta na protokole typu publikacja/subskrypcja, który z kolei bazuje na protokole ATOM/RSS. Jej celem jest przeistoczenie kanałów Atom w dane przesyłane na bieżąco i wyeliminowanie konieczności sondowania, które negatywnie odbija się na odbiorcach kanałów. Użytkownik subskrybuje określony temat i wydawca informuje go, jeśli pojawi się coś nowego.

W dystrybucji zadań publikacji i rozprowadzania treści istnieje pojęcie **węzła komunikacyjnego** (ang. *hub*). Węzeł rozsyła treść do subskrybentów. Na poniższym schemacie przedstawiono model PubSubHubbub:



Zasada działania PubSubHubbub

Przyjrzyjmy się dokładniej temu modelowi:

1. **Subskrybent** znajduje węzeł komunikacyjny poprzez pobranie kanału od **dostawcy**.
2. Subskrybent subskrybuje zawartość węzła komunikacyjnego przy użyciu identyfikatora URI kanału, który go interesuje.
3. Gdy **wydawca** ma aktualizacje do wysłania, pozwala je pobrać węzłowi.
4. Węzeł wysyła aktualizacje do wszystkich subskrybentów.

Zaletą tego modelu jest to, że wydawca nie musi wysyłać aktualizacji do wszystkich subskrybentów. Dla subskrybentów korzystne jest natomiast to, że otrzymują aktualizacje od węzła komunikacyjnego bez konieczności sondowania wydawcy.

Protokół ten jest wykorzystywany w paradygmacie uchwytów sieciowych opisanym w następnych podrozdziałach.

Model strumieniowania

Model strumieniowania w komunikacji asynchronicznej polega na utworzeniu otwartego kanału i wysyłaniu pojawiających się danych. W takim przypadku musi być cały czas otwarte połączenie gniazdowe.

Zdarzenia wysyłane przez serwer

Zdarzenia wysyłane przez serwer (ang. *Server-Send Events* — SSE) to bazująca na modelu strumieniowym technologia, w której przeglądarka otrzymuje automatyczne aktualizacje od serwera poprzez połączenie HTTP. Organizacja W3C opracowała standard tych zdarzeń w postaci API o nazwie EventSource w ramach technologii HTML5.

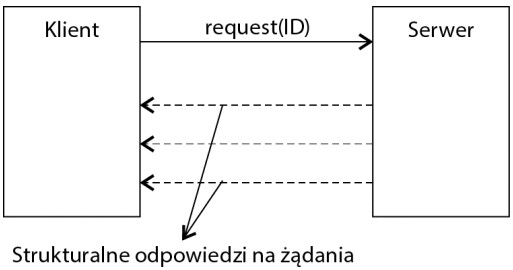
W technologii tej klient inicjuje żądanie do serwera przy użyciu typu MIME `text/eventstream`. Po wstępnych uzgodnieniach serwer może wysyłać zdarzenia do klienta na bieżąco, w miarę jak się pojawiają. Zdarzenia te mają postać zwykłych wiadomości tekstowych przesyłanych od serwera do klientów. Mogą zawierać dane do użycia przez procedury nasłuchu klientów, które mogą je odpowiednio interpretować i na nie reagować.

Zdarzenia wysyłane przez serwer mogą definiować format przesyłanych informacji. Format ten składa się z linijki tekstu, której granice wyznaczają sekwencje znaków. Na przykład linijki zawierające treść lub dane zaczynają się od napisu `data:` i kończą znakami `\n\n`, jak widać w poniższym przykładzie:

```
data: Moja wiadomość \n\n
```

Linijki zawierające dyrektywy **QoS** (ang. *Quality of Service* — np. `retry` i `id`) zaczynają się od nazwy atrybutu QoS, po którym jest dwukropek i wartość. Standardowy format pozwala tworzyć ogólne biblioteki na bazie SSE ułatwiające pisanie programów.

Na poniższym schemacie przedstawiono, jak działa wysyłanie zdarzeń przez serwer.



Proces wysyłania zdarzeń przez serwer

Jak widać na tym schemacie, klient subskrybuje źródło zdarzeń, po czym serwer wysyła mu aktualizacje natychmiast po wystąpieniu zdarzeń.

Dodatkowo wraz z całymi wiadomościami serwer może wiązać i wysyłać identyfikatory, jak pokazano poniżej:

```
id: 12345\n
data: Wiadomość 1\n
data: Wiadomość 2\n\n
```

W tym przykładzie pokazano sposób wysyłania z identyfikatorem zdarzenia wiadomości składających się z kilku linijek. Zwróć uwagę, że ostatnia linijka kończy się znakami `\n\n`.

Identyfikator umożliwia klientowi sprawdzenie, jakie było ostatnie zdarzenie, dzięki czemu w razie utraty połączenia z serwerem w następnym żądaniu klient ustawi specjalny nagłówek HTTP (`Last-Event-ID`).

W kolejnych punktach opisuję sposoby wiązania identyfikatorów ze zdarzeniami wysyłanymi przez serwer, sposoby postępowania w przypadku utraty połączenia oraz metody wiązania nazw ze zdarzeniami wysyłanymi przez serwer.

Wiązanie identyfikatora ze zdarzeniem

Każda wiadomość SSE może zawierać identyfikator, który może być przydatny na wiele sposobów, np. do śledzenia wiadomości odebranych przez klienta oraz do ich kontrolowania. Jeśli SSE zawiera identyfikator, klient może przekazać go do serwera jako jeden z parametrów połączenia, aby nakazać wznowienie pracy od konkretnej wiadomości. Oczywiście na serwerze powinna znajdować się implementacja odpowiedniej procedury do wznawiania komunikacji z identyfikatora wiadomości na żądanie klienta.

Poniżej znajduje się przykładowa wiadomość SSE z identyfikatorem:

```
id: 123 \n
data: To jest jednolinijkowe zdarzenie. \n\n
```

Ponawianie próby w przypadku utraty połączenia

Przeglądarki Firefox, Chrome, Opera i Safari obsługują zdarzenia wysyłane przez serwer. Jeśli więc nastąpi zerwanie połączenia z serwerem, przeglądarka może spróbować ponownie je nawiązać. Istnieje dyrektywa `retry`, którą można skonfigurować na serwerze, aby pozwolić na ponawianie prób nawiązania połączenia przez klienta w przypadku utraty połączenia. Domyślnie próby te podejmowane są co trzy sekundy. Aby wydłużyć ten czas do pięciu sekund, serwer może wysłać następujące zdarzenie:

```
retry: 5000\n
data: Dane mieszczące się w jednej linijce.\n\n
```

Wiązanie nazw ze zdarzeniami

Inną dyrektywą SSE jest nazwa zdarzenia. Każde źródło zdarzeń może generować różne typy zdarzeń i klient może sam zdecydować, w jaki sposób wykorzystać każdy z nich. Poniższy fragment kodu ilustruje sposób użycia dyrektywy `name` w wiadomości:

```
event: bookavailable\n
data: {"name" : "Gra o tron"}\n\n
event: newbookadded\n
data: {"name" :"Nawałnica mieczy"}\n\n
```

Zdarzenia wysyłane przez serwer i JavaScript

Dla programistów JavaScript podstawowym API SSE po stronie klienta jest interfejs `EventSource`. Zawiera on sporo funkcji i atrybutów, a w poniższej tabeli znajduje się opis najważniejszych z nich:

Nazwa funkcji	Opis
`addEventListener`	Funkcja dodająca procedurę nasłuchu zdarzeń do obsługi przychodzących zdarzeń określonego typu.
`removeEventListener`	Funkcja usuwająca zarejestrowaną procedurę nasłuchu zdarzeń.
`onmessage`	Funkcja wywoływana w momencie przybycia wiadomości.
`onerror`	Funkcja wywoływana, gdy z połączeniem stanie się coś złego.
`onopen`	Funkcja wywoływana przy otwieraniu połączenia.
`onclose`	Funkcja wywoływana przy zamykaniu połączenia.

Poniżej znajduje się przykład ilustrujący sposób subskrypcji różnych typów zdarzeń emitowanych przez jedno źródło. Przyjęto założenie, że wiadomości przychodzą w formacie JSON. Na przykład istnieje aplikacja wysyłająca strumieniowo aktualizacje do użytkowników, gdy w pewnym kontenerze pojawią się nowe książki. Procedura nasłuchowa `bookavailable` analizuje otrzymywane dane przy użyciu prostego parsera JSON.

Następnie przy użyciu aktualizacji zmienia GUI, podczas gdy procedura nasłuchowa `newbookadded` używa funkcji wznawiającej do filtrowania i selektywnego przetwarzania par JSON.

```
var source = new EventSource('books');
source.addEventListener('bookavailable', function(e) {
  var data = JSON.parse(e.data);
  // Aktualizuje element GUI przy użyciu otrzymanych danych...
}, false);

source.addEventListener('newbookadded', function(e) {
  var data = JSON.parse(e.data, function (key, value) {
    var type;
    if (value && typeof value === 'string') {
        return "Wartość łańcucha: "+value;
    }
    return value;
```

Zdarzenia wysyłane przez serwer i Jersey

Zdarzenia wysyłane przez serwer nie należą do standardowej specyfikacji JAX-RS, ale są obsługiwane przez implementację Jersey tej biblioteki. Więcej informacji na ten temat znajduje się na stronie *https://jersey.java.net/documentation/latest/sse.html*.

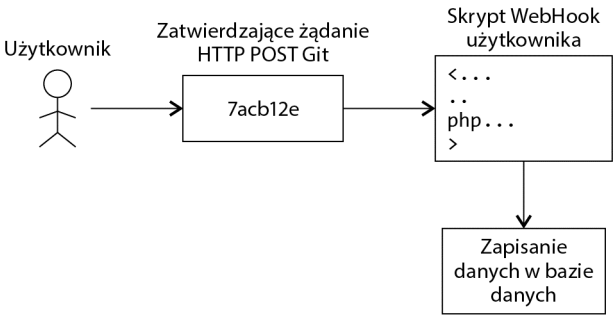
Uchwyty sieciowe

Uchwyty sieciowe (ang. *WebHook*) to rodzaj definiowanych przez użytkownika niestandardowych wywołań zwrotnych HTTP. W tym modelu klient wskazuje producentowi zdarzeń punkt końcowy, do którego można *wysyłać* zdarzenia. Gdy do punktu końcowego zostanie wysłane zdarzenie, „zainteresowana" aplikacja kliencka może wykonać odpowiednie czynności. Przykładem użycia uchwytów sieciowych jest wywoływanie zdarzeń, takich jak np. zadanie Hudson, przy użyciu uchwytu GIT `post-receive`.

Na potwierdzenie, że subskrybent otrzymał uchwyt sieciowy, punkt końcowy subskrybenta powinien wysyłać kod statusu HTTP `200 OK`. Producent zdarzeń zignoruje treść żądania i wszystkie nagłówki oprócz statusu. Każdy kod odpowiedzi spoza zakresu `200`, wliczając zakres `3xx`, oznacza, że nie otrzymano uchwytu, i API może ponowić próbę wysłania żądania HTTP `POST`.

Zdarzenia uchwytów sieciowych generowane przez GitHub zawierają informacje o aktywności w repozytorium. Uchwyty sieciowe mogą wyzwalać zdarzenia dla wielu różnych akcji. Na przykład konsument może zażądać informacji przy każdym zatwierdzeniu, rozwidleniu repozytorium lub zgłoszeniu problemu.

Poniższy schemat przedstawia sposób współpracy uchwytów sieciowych z GitHub i GitLab:



Zasada działania uchwytów sieciowych

Przyjrzyjmy się uważnie temu procesowi:

1. Użytkownik wysyła dane do repozytorium Git.
2. Istnieje niestandardowy URL uchwytu sieciowego do wysyłania obiektów zdarzeń zarejestrowanych przez konsumenta w GitHub. Gdy pojawia się zdarzenie, np. zatwierdzenie, usługa GitHub wysyła informacje dotyczące tego zatwierdzenia w wiadomości `POST` do punktu końcowego dostarczonego przez konsumenta.
3. Aplikacja konsumencka może zapisać te dane w bazie danych lub podjąć jakieś inne czynności, np. rozpocząć kompilację w ramach ciągłej integracji.

Parę popularnych studiów przypadków dotyczących uchwytów sieciowych

Twilio używa uchwytów sieciowych do wysyłania SMS-ów. GitHub używa uchwytów sieciowych do wysyłania powiadomień o zmianach w repozytorium i dodatkowo przekazuje pewne informacje.

PayPal używa usługi **IPN** (ang. *Instant Payment Notification*) automatycznie powiadamiającej klientów o zdarzeniach dotyczących transakcji. Usługa ta również bazuje na uchwytach sieciowych.

W API reagującym na bieżąco Facebooka używane są uchwyty sieciowe i jest ono zbudowane na bazie PubSubHubbub (PuSH).

Jak już wspominałam, jeśli API nie wysyła powiadomień przy użyciu jakiejś technologii bazującej na uchwytach sieciowych, klienty muszą dowiadywać się, czy są nowe dane, za pomocą sondowania, co jest nieefektywne i powoduje straty czasu.

Gniazda sieciowe

Gniazdo sieciowe (***WebSocket***) to dwukierunkowy kanał komunikacyjny działający na bazie jednego połączenia TCP.

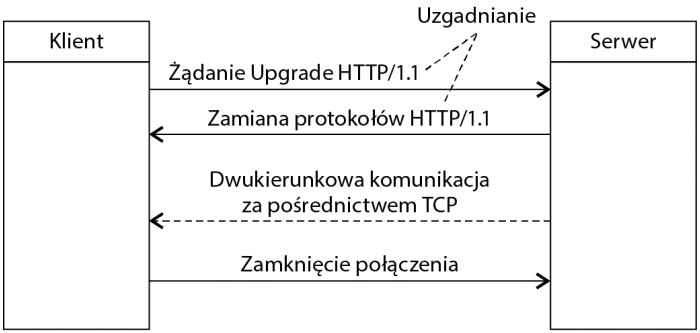
Protokół ten jest niezależny od TCP, a z HTTP wiąże go tylko to, że uzgadnianie w celu przełączenia na gniazda sieciowe jest interpretowane przez serwery HTTP jako żądanie `Upgrade`.

Protokół WebSocket umożliwia nawiązanie dwukierunkowej komunikacji na bieżąco między klientami (np. przeglądarką internetową) a punktem końcowym bez stałego kosztu ustanawiania połączenia lub sondowania zasobu. Gniazda sieciowe są powszechnie używane w kanałach nowości mediów społecznościowych, narzędziach do pracy zespołowej itp.

Poniżej znajduje się przykład uzgadniania przy użyciu protokołu WebSocket, które rozpoczyna się od żądania `Upgrade`:

```
GET /text HTTP/1.1\r\n Upgrade: WebSocket\r\n Connection:
  Upgrade\r\n Host: www.websocket.org\r\n …\r\n
HTTP/1.1 101 WebSocket Protocol Handshake\r\n
Upgrade: WebSocket\r\n
Connection: Upgrade\r\n
...\r\n
```

Poniższy schemat przedstawia przykład uzgadniania za pośrednictwem żądania HTTP/1.1 `Upgrade` i odpowiedzi zmiany protokołów HTTP/1.1.



Zasada działania gniazd sieciowych

Po ustanowieniu połączenia między klientem i serwerem przy użyciu żądania `Upgrade` i odpowiedzi HTTP/1.1 można zacząć przesyłać pakiety danych WebSocket, w formie binarnej lub tekstowej, między klientem i serwerem w obu kierunkach.

Minimalny rozmiar ramki w WebSocket wynosi dwa bajty. To radykalnie redukuje narzut w porównaniu z tym, co jest przesyłane w nagłówkach HTTP.

Poniżej przedstawiono bardzo prosty przykład użycia API gniazd sieciowych języka JavaScript:

```
// utworzenie obiektu WebSocket
var websocket = new WebSocket("coffee");
// ustawienie funkcji zdarzenia wiadomości
websocket.onmessage = function(evt) {
onMessageFunc(evt)
};
// funkcja onMessageFunc wywoływana po nadejściu wiadomości
function onMessageFunc (evt) {
// aktualizacja GUI na podstawie treści wiadomości
}
// wysłanie wiadomości do serwera
websocket.send("coffee.selected.id=1020");
// ustawienie procedury nasłuchu zdarzeń dla zdarzeń typu open
addEventListener('open', function(e){
    onOpenFunc(evt)});
// zamknięcie połączenia
websocket.close();
```

Poniższa tabela zawiera szczegółowy opis funkcjonalności gniazd sieciowych i ich różnych funkcji:

Nazwa funkcji	Opis
`send`	Funkcja do wysyłania wiadomości na określony adres URL serwera.
`onopen`	Funkcja wywoływana po utworzeniu połączenia. Obsługuje zdarzenia typu `open`.
`onmessage`	Funkcja wywoływana po pojawieniu się nowej wiadomości w celu obsługi zdarzenia typu `message`.
`onclose`	Funkcja wywoływana przy zamykaniu połączenia. Obsługuje zdarzenia typu `close`.
`onerror`	Funkcja wywoływana w celu obsługi zdarzeń typu `error`, które są zgłaszane, gdy w kanale komunikacyjnym wystąpi błąd.
`close`	Funkcja zamykająca gniazdo komunikacyjne i kończąca interakcję klienta z serwerem.

Studia znanych przypadków użycia gniazd sieciowych

Jedną z pierwszych gier, w których wykorzystano gniazda sieciowe na masową skalę, jest *Zynga Poker*. Zastosowanie technologii WebSocket i HTML5 pozwoliło zapewnić płynną i szybką rozgrywkę synchroniczną na urządzeniach przenośnych. Oczywiście wszystko zależy od jakości samego połączenia, ale gra ładuje się i odświeża prawie natychmiast.

Inne API i technologie do komunikacji na bieżąco

Istnieje jeszcze kilka innych protokołów komunikacyjnych i API działających na bieżąco lub prawie na bieżąco, z których większość jest używana poza przeglądarką internetową. Niektóre z nich są opisane w paru następnych podrozdziałach.

XMPP

Protokół XMPP umożliwia przesyłanie wiadomości tekstowych i jest często wykorzystywany do budowy czatów internetowych. Bazuje na modelu komunikacji klient-serwer, serwer-serwer oraz serwer-klient. W związku z tym definiuje protokół klient-serwer i serwer-serwer do wysyłania wiadomości XML za pośrednictwem TCP.

XMPP to dojrzała technologia mająca wiele implementacji w różnych językach i na rozmaitych platformach. Jej największą wadą jest długotrwałe sondowanie i otwarcie gniazd do obsługi komunikatów przychodzących i wychodzących.

BOSH poprzez XMPP

BOSH (ang. *Bidirectional Streams Over Synchronous HTTP*) to standard zdefiniowany w dokumencie XEP-0124 umożliwiający wykorzystanie XMPP poprzez HTTP. W przypadku połączeń inicjowanych przez klienta klient wysyła pakiety XMPP poprzez HTTP, a w przypadku połączeń inicjowanych przez serwer serwer stosuje technikę **spowolnionego sondowania** (ang. *long polling*) przy połączeniu otwartym na z góry określony czas.

Największą zaletą technologii BOSH jest możliwość wykorzystania przeglądarki internetowej jako klienta XMPP przez użycie jednej z implementacji JavaScript standardu BOSH. Wśród bibliotek obsługujących BOSH można wymienić Emite, JSJaC i Xmpp4Js.

Porównanie uchwytów sieciowych, gniazd sieciowych i zdarzeń wysyłanych przez serwer

Zdarzenia wysyłane przez serwer w odróżnieniu od gniazd sieciowych działają na bazie protokołu HTTP. SSE umożliwiają tylko jednostronną komunikację od serwera do klienta, gniazda sieciowe natomiast zapewniają komunikację w dwie strony. SSE mogą automatycznie podejmować próby ponownego nawiązania połączenia i mają identyfikatory zdarzeń, które można wiązać z wiadomościami w celu zapewnienia funkcjonalności QoS. W gniazdach sieciowych nie ma takiej możliwości.

Z drugiej strony gniazda sieciowe zapewniają komunikację w obie strony, redukują czas opóźnień i poprawiają jakość przesyłania danych, ponieważ na początku dokonują **uzgodnienia** (ang. *handshake*) poprzez HTTP, a później przesyłają wiadomości między punktami końcowymi za pośrednictwem TCP.

W porównaniu z tymi dwoma protokołami uchwyty sieciowe są łatwiejsze w obsłudze i integracji z aplikacjami oraz usługami. To pozwala na tworzenie wzajemnie połączonych i wymiennych, choć luźno powiązanych, usług chmurowych, które komunikują się ze sobą za pomocą żądań HTTP.

Poniższa tabela zawiera zwięzłe porównanie uchwytów sieciowych, gniazd sieciowych i zdarzeń wysyłanych przez serwer.

W następnym podrozdziale dowiesz się, dlaczego aplikacje chmurowe ewoluują w kierunku architektury mikrousług.

Kryterium	Uchwyty sieciowe	Gniazda sieciowe	Zdarzenia wysyłane przez serwer
Asynchroniczna komunikacja na bieżąco	Tak	Tak	Tak
Rejestracja zwrotnego adresu URL	Tak	Nie	Nie
Otwarcie długotrwałego połączenia	Nie	Tak	Tak
Dwukierunkowość	Nie	Tak	Nie
Obsługa błędów	Nie	Tak	Tak
Łatwość obsługi i implementacji	Tak	Wymaga obsługi przez przeglądarki i serwer proxy	Tak
Wymaga sondowania w razie problemów	Nie	Tak	Nie

REST i mikrousługi

Sen SOA urzeczywistnił się w postaci architektury **mikrousług**, której głównym założeniem jest podzielenie dużych monolitycznych aplikacji na zestawy mniejszych, lepiej dostosowanych usług. W tym podrozdziale przedstawiam, jakie są zalety mikrousług w porównaniu z monolitycznymi usługami.

Prostota

Programiści doszli do wniosku, że zamiast budować skomplikowane tradycyjne rozwiązania, lepiej jest tworzyć aplikacje przy użyciu lekkich API usługowych. Powstałe w ten sposób programy są bardziej niezawodne, skalowalne i łatwiejsze w obsłudze. Metoda ta nazywa się architekturą mikrousługową. Stoi ona w opozycji do starszych technologii RPC, takich jak CORBA i RMI oraz ciężkie protokoły typu SOAP.

Wyodrębnienie problemów

W monolitycznych aplikacjach wszystkie elementy usługi są ładowane w jednym komponencie aplikacji (pliku WAR, EAR lub JAR), który wdraża się na jednej maszynie wirtualnej Javy. Z tego powodu jeśli serwer aplikacji ulegnie awarii, przestają działać wszystkie usługi.

W architekturze mikrousługowej natomiast usługi mogą być niezależne od plików WAR/EAR. Mogą się ze sobą komunikować przy użyciu technologii REST oraz JSON i XML. Innym sposobem komunikacji są też protokoły do przesyłania wiadomości, takie jak AMQP i Rabbit MQ.

Skalowalność

W usługach monolitycznych nie wszystkie usługi z wdrożonego pliku aplikacji mogą wymagać skalowania, ale i tak wszystkie muszą być przeskalowane w taki sam sposób na poziomie wdrożenia.

W architekturze mikrousługowej aplikacje mogą składać się z mniejszych usług, które można wdrażać i skalować niezależnie od pozostałych. Dzięki temu rozwiązania architektoniczne są mniej awaryjne, skalowalne i zwinne, a proces tworzenia i rozwoju od fazy definicji do produkcji jest znacznie skrócony.

Wyraźny podział funkcjonalności

W architekturze mikrousługowej usługi mogą być uporządkowane wg funkcjonalności biznesowej. Na przykład usługa przechowywania może być oddzielona od usługi rachunkowej i transportowej. Jeśli jedna z tych usług ulegnie awarii, pozostałe nadal będą działać, dzięki cechom opisanym w punkcie „Wyodrębnienie problemów".

Niezależność od języka programowania

Kolejną zaletą architektury mikrousługowej jest to, że usługi mają proste i łatwe w użyciu API typu REST oparte na formacie JSON, z których łatwo jest korzystać w dowolnym języku programowania i systemie szkieletowym, np. PHP, Ruby on Rails, Python czy Node.js.

Pionierami stosowania architektury mikrousługowej są portale Amazon i Netflix. Portal eBay udostępnia Turmeric, otwartą platformę SOA, przy użyciu której można tworzyć, wdrażać, zabezpieczać, uruchamiać i monitorować usługi i konsumentów SOA.

Zalecana lektura

Poniżej znajduje się lista adresów internetowych, pod którymi znajdziesz więcej informacji na temat przypadków opisanych w tym rozdziale.

- *https://stripe.com/docs/webhooks*: opis uchwytów sieciowych w portalu Stripe.
- *https://github.com/sockjs*: strona projektu SockJs w portalu GitHub.
- *https://developer.github.com/webhooks/testing/*: uchwyty sieciowe w portalu GitHub.
- *http://www.twilio.com/platform/webhooks*: uchwyty sieciowe w portalu Twilio.
- *http://xmpp4js.sourceforge.net/*: biblioteka BOSH Xmpp4Js.
- *https://code.google.com/p/emite/*: biblioteka BOSH Emite.

Podsumowanie

W tym rozdziale zostały opisane zaawansowane zagadnienia, takie jak uchwyty sieciowe, zdarzenia wysyłane przez serwer i gniazda sieciowe oraz sposoby i zasady ich wykorzystania. Najważniejszą informacją jest to, jak bardzo ważne są interfejsy API współpracujące na bieżąco i eliminacja konieczności sondowania. Przedstawiłam przykłady firm wykorzystujących w swoich rozwiązaniach zarówno uchwyty, jak i gniazda sieciowe. W różnych rozdziałach książki opisałam rozmaite najlepsze techniki i zasady projektowania. W ostatnim rozdziale przedstawiłam, jak może wyglądać przyszłość technologii REST i komunikacji asynchronicznej. Rosnące zasoby informacji społecznościowych mogą być świetnym katalizatorem rozwoju sieci semantycznej, która umożliwi wykonywanie różnych czynności w naszym imieniu oraz przeprowadzanie błyskawicznych aktualizacji przy użyciu opisanych technologii.

Ponadto pokazałam, dlaczego twórcy aplikacji chmurowych coraz bardziej skłaniają się ku modelowi sieciowemu polegającemu na budowie mikrousług, które można wdrażać i skalować niezależnie od siebie. Więcej informacji o metodach tworzenia usług typu RESTful znajduje się w książce *Developing RESTful Services with JAX-RS2.0, WebSockets, and JSON* (Bhakti Mehta i Masoud Kalali, Packt Publishing).

A

DODATEK

W erze sieci społecznościowych, chmury i aplikacji mobilnych ludzie chcą się komunikować, wygłaszać swoje opinie, wspólnie tworzyć rozwiązania, dzielić się z innymi swoimi informacjami oraz zadawać pytania. Takie wnioski płyną z raportu opublikowanego na stronie *http://www.statisticbrain.com/twitter-statistics/*, z którego wynika, że portal Twitter ma 6,5 miliona użytkowników, publikujących 58 milionów wpisów dziennie. Dane portalu Facebook są jeszcze bardziej imponujące. Portal ten ma 1,3 miliarda użytkowników i stanowi najważniejszą platformę społecznościową na świecie. Portal GitHub natomiast wyrósł na podstawową platformę społecznościową dla programistów. Te trzy portale, Twitter, Facebook i GitHub, są najpopularniejszymi platformami do budowy aplikacji, wyszukiwania danych oraz analizowania informacji.

W poprzednich rozdziałach głównym tematem było tworzenie usług typu RESTful oraz techniki optymalizacji ich wydajności, buforowania, zabezpieczania i skalowania. Tematem tego dodatku są z kolei popularne platformy REST oraz sposób realizacji w nich różnych technik opisanych w poprzednich rozdziałach.

W dodatku znajduje się:

- Przegląd API REST portalu GitHub.
- Przegląd API Open Graph portalu Facebook.
- Przegląd API REST portalu Twitter.

Przegląd API REST portalu GitHub

Portal GitHub stał się najpopularniejszą platformą społecznościową do wspólnego pisania programów i pomagania sobie w rozwoju różnych projektów. Korzystają z niego programiści chcący tworzyć i wdrażać oprogramowanie. Można w nim znaleźć zarówno indywidualne, jak i firmowe projekty. GitHub ma rozbudowany interfejs API, którego dokumentacja znajduje się na stronie *https://developer.github.com/v3/*.

W następnym podrozdziale znajduje się opis tego, jak w serwisie GitHub zastosowano różne techniki omówione w poprzednich rozdziałach tej książki.

Pobieranie informacji z portalu GitHub

Poniżej znajdują się przykładowe polecenia cURL ilustrujące sposób pobierania danych na temat użytkownika, repozytoriów itd. bez uwierzytelniania.

Poniższe polecenie pobiera informacje o użytkowniku `javaee-samples`:

```
curl https://api.github.com/users/javaee-samples
{
  "login": "javaee-samples",
  "id": 6052086,
  "avatar_url": "https://avatars.githubusercontent.com/u/6052086?",
  "gravatar_id": null,
  "url": "https://api.github.com/users/javaee-samples",
  "html_url": "https://github.com/javaee-samples",
  "followers_url": "https://api.github.com/users/javaee-samples/followers",
  "following_url": "https://api.github.com/users/
     javaee-samples/following{/other_user}",
  "gists_url": "https://api.github.com/users/javaee-samples/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/
     javaee-samples/starred{/owner}{/repo}",
  "subscriptions_url": "https://api.github.com/users/
     javaee-samples/subscriptions",
  "organizations_url": "https://api.github.com/users/javaee-samples/orgs",
  "repos_url": "https://api.github.com/users/javaee-samples/repos",
  "events_url": "https://api.github.com/users/javaee-samples/events{/privacy}",
  "received_events_url": "https://api.github.com/users/
     javaee-samples/received_events",
  "type": "Organization",
  "site_admin": false,
  "name": "JavaEE Samples",
  "company": null,
  "blog": "https://arungupta.ci.cloudbees.com/",
  "location": null,
  "email": null,
  "hireable": false,
  "bio": null,
  "public_repos": 11,
  "public_gists": 0,
  "followers": 0,
  "following": 0,
  "created_at": "2013-11-27T17:17:00Z",
  "updated_at": "2014-07-03T16:17:51Z"
```

W powyższej odpowiedzi znajdują się różne adresy URL, przy użyciu których można zdobyć różne szczegółowe informacje, takie jak liczba obserwujących, zatwierdzenia itd. Ten sposób prezentacji adresów URL różni się od wcześniej opisanych przykładów zastosowania HATEOAS z wykorzystaniem atrybutów `links`, `href`, `rel` itp. Przykład stanowi ilustrację tego, jak usługi są implementowane na różnych platformach.

Aby pobrać repozytoria użytkownika z podziałem na strony, można wysłać następujące żądanie:

```
curl https://api.github.com/users/javaee-samples/repos?page=1&per_page=10
...
```

W interfejsie API portalu GitHub do uwierzytelniania użytkowników stosowany jest protokół OAuth2. Każdy programista pracujący z tym API musi zarejestrować swoją aplikację, której zostaje przydzielony identyfikator i tajny klucz.

Więcej informacji na temat wysyłania uwierzytelnionych żądań znajduje się na stronie *https://developer.github.com/v3/oauth/*.

Czasowniki i akcje zasobów

W poniższej tabeli znajduje się zestawienie czasowników używanych przez API GitHub do różnych celów:

Czasownik	Opis
`HEAD`	Służy do pobierania informacji o nagłówkach HTTP.
`GET`	Służy do pobierania zasobów, np. informacji o użytkowniku.
`POST`	Służy do tworzenia zasobów, np. scalania żądań ściągnięcia.
`PATCH`	Służy do częściowego aktualizowania zasobów.
`PUT`	Służy do zmieniania zasobów, np. aktualizowania użytkowników.
`DELETE`	Służy do usuwania zasobów, np. współpracowników.

Wersjonowanie

W identyfikatorze URI interfejsu API portalu GitHub znajduje się numer v3. Domyślna wersja API może się w przyszłości zmienić. Jeśli działanie klienta zależy od konkretnej wersji, zaleca się wysłanie nagłówka `Accept`, jak pokazano poniżej:

```
Accept: application/vnd.github.v3+json
```

Obsługa błędów

Jak napisałam w rozdziale 2., błędy po stronie klienta są oznaczane kodami z serii `4xx`. W portalu GitHub stosowana jest podobna taktyka oznaczania błędów.

Jeżeli klient korzystający z API wyśle nieprawidłowy kod JSON, otrzyma w odpowiedzi kod `400 Bad Request`. Jeśli klient API nie wyśle jakiegoś pola w żądaniu, w odpowiedzi otrzyma kod błędu `422 Unprocessable Entity`.

Ograniczanie liczby żądań

W interfejsie API portalu GitHub stosowane jest też ograniczanie liczby żądań uniemożliwiające przeciążenie serwera jakiemuś niecnemu klientowi. W przypadku żądań stosujących uwierzytelnianie podstawowe lub OAuth limit wynosi 5000 żądań na godzinę. Dla żądań nieuwierzytelnionych limit ten wynosi tylko 60 na godzinę. API portalu GitHub informuje o statusie limitów za pomocą nagłówków `X-RateLimit-Limit`, `X-RateLimit-Remaining` oraz `X-RateLimit-Reset`.

Wiesz już mniej więcej, jakie rozwiązania zastosowano w interfejsie API portalu GitHub w zakresie realizacji zasad technologii REST opisanych w poprzednich rozdziałach tej książki. W następnym podrozdziale poznasz API REST o nazwie Open Graph portalu Facebook. Ponownie przeanalizujemy kwestie wersjonowania, obsługi błędów, ograniczania liczby żądań itd.

Przegląd API Graph portalu Facebook

API Graph portalu Facebook umożliwia wydobywanie informacji z danych dostarczanych przez ten portal. Przy użyciu żądań HTTP API REST klient może wykonać wiele czynności, takich jak wysyłanie zapytań, publikowanie aktualizacji i obrazów, pobieranie i tworzenie albumów, sprawdzanie liczby polubień węzła, pobieranie komentarzy itd. Poniżej znajduje się opis metod korzystania z API Graph portalu Facebook.

> Do uwierzytelniania i autoryzacji portal Facebook wykorzystuje wariant protokołu OAuth 2.0. Macierzysta aplikacja Facebook jest używana w systemach iOS i Android.

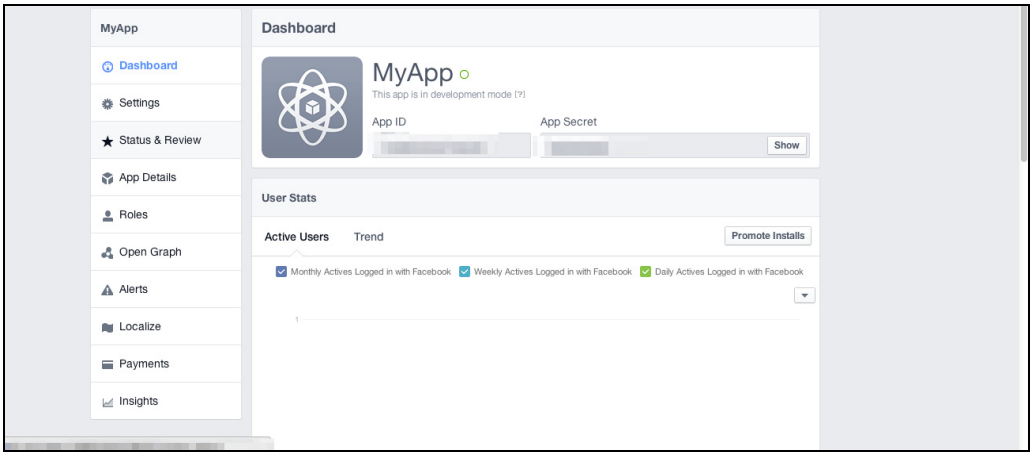
Aby skorzystać z API Facebooka, klient musi zdobyć token dostępu wymagany przez protokół OAuth 2.0. Poniżej znajduje się opis procesu tworzenia identyfikatora aplikacji i klucza tajnego oraz pobierania tokenu dostępu pozwalającego na wysyłanie zapytań o dane Facebooka:

1. Wejdź na stronę *developers.facebook.com/apps*, na której możesz utworzyć nową aplikację. Po utworzeniu aplikacji otrzymasz identyfikator i klucz tajny, jak pokazano na poniższym zrzucie ekranu (patrz pierwszy rysunek na następnej stronie).
2. Gdy ma się identyfikator aplikacji i klucz tajny, można pobrać token dostępu i zacząć wysyłać zapytania do Facebooka.

> Facebook ma specjalny punkt końcowy o nazwie `/me`, odnoszący się do użytkownika, którego token dostępowy jest używany. Aby np. pobrać zdjęcia swojego użytkownika, można wysłać następujące żądanie:
>
> ```
> GET /graph.facebook.com/me/photos
> ```

3. Aby wysłać wiadomość, użytkownik może użyć prostego wywołania API:

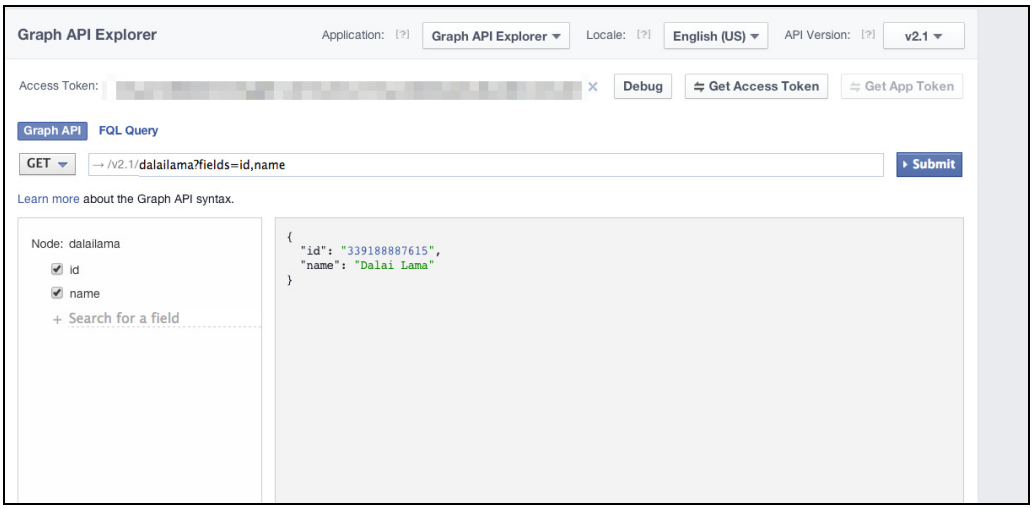
```
POST /graph.facebook.com/me/feed?message="foo" &access_token="..."
```



4. By pobrać szczegóły dla identyfikatora, nazwę i zdjęcia przy użyciu **eksploratora API Graph** (ang. *Graph API Explorer*), należy wysłać następujące zapytanie:

   ```
   https://developers.facebook.com/tools/explorer?method=GET&path=
   me%3Ffields=id,name
   ```

5. Na poniższym zrzucie ekranu widać zapytanie eksploratora API Graph z węzłem `dalailama`. Kliknięcie identyfikatora powoduje wyświetlenie dodatkowych informacji o węźle.



Wiesz już, jak za pomocą aplikacji Graph API Explorer tworzyć zapytania dotyczące węzłów w API Social Graph. Można wysyłać zapytania dotyczące różnych pól, np. identyfikatora i nazwy, oraz używać metod `GET`, `POST` i `DELETE`.

Czasowniki i czynności zasobów

W poniższej tabeli znajduje się zestawienie najczęściej używanych czasowników API Facebook Graph:

Czasownik	Opis
GET	Służy do pobierania zasobów, np. albumów, wpisów itd.
POST	Służy do tworzenia zasobów, np. wpisów, albumów itd.
PUT	Służy do zmieniania zasobów.
DELETE	Służy do usuwania zasobów.

> Warto podkreślić, że w API Graph portalu Facebook do aktualizowania zasobów używana jest metoda POST, a nie PUT.

Wersjonowanie

API Graph ma aktualnie wersję 2.1 opublikowaną dnia 7 sierpnia 2014 roku. Klient może określić inną wersję w adresie URL żądania. Jeśli tego nie zrobi, domyślnie stosowana jest najnowsza wersja. Każda wersja działa przez dwa lata, po upływie których wszystkie żądania do starszych wersji zostają przekierowane do najnowszej.

Obsługa błędów

Poniżej znajduje się przykładowa odpowiedź zawierająca informację o błędzie dotyczącym niepoprawnego żądania wysłanego do API:

```
{
    "error": {
      "message": "Opis błędu.",
      "type": "OAuthException",
      "code": 190 ,
    "error_subcode": 460
  }
}
```

W kodzie tym widać obiekty JSON o nazwach `code` i `error_subcode`, przy użyciu których można dowiedzieć się, na czym polega problem i jakie czynności naprawcze należy podjąć. W tym przypadku `code` ma wartość `190` typu `OAuthException`, a `error_subcode` ma wartość `460` oznaczającą, że mogło się zmienić hasło, przez co token dostępu stracił ważność.

Ograniczanie liczby żądań

API Graph Facebooka stosuje różne zasady ograniczania liczby żądań w zależności od tego, czy klientem jest użytkownik, aplikacja, czy reklama. Jeśli limit żądań przekroczy użytkownik, to zostaje zablokowany na 30 minut. Więcej informacji na ten temat znajduje się na stronie *https://developers.facebook.com/docs/reference/ads-api/api-rate-limiting/*. W następnym podrozdziale znajduje się opis API REST Twittera.

Przegląd API portalu Twitter

Interfejs API portalu Twitter składa się z API REST i strumieniowego, przy użyciu których programista może pobrać najważniejsze informacje, takie jak oś czasu, dane statusu, dane użytkownika itd.

W Twitterze stosowana jest trójelementowa autoryzacja OAuth.

Ważne aspekty dotyczące protokołu OAuth w API Twittera

Aplikacja kliencka nie musi zapisywać identyfikatora i hasła. Zamiast tego w każdym żądaniu wysyła token dostępu reprezentujący użytkownika.

Aby żądanie zakończyło się powodzeniem, zmienne `POST`, parametry zapytania i adres URL żądania muszą być nienaruszone.

To użytkownik decyduje, jakie aplikacje mogą działać w jego imieniu, i może cofnąć autoryzację w dowolnym momencie.

W każdym żądaniu używany jest identyfikator (`oauth_nonce`) zapobiegający wielokrotnemu wykonaniu tego samego żądania w przypadku ataku hakera.

Dla większości programistów podstawowe narzędzia do wysyłania żądań do Twittera mogą być niejasne. Na stronie *https://blog.twitter.com/2011/improved-oauth-10a-experience* opisano, jak zbudować aplikację, wygenerować klucze oraz utworzyć żądanie przy użyciu narzędzia OAuth.

Poniżej znajduje się przykładowe żądanie wygenerowane przez narzędzie OAuth Twittera. Zawiera ono zapytanie pobierające statusy dla uchwytu `twitterapi`:

API Twittera nie obsługuje żądań nieuwierzytelnionych i ma bardzo restrykcyjne ograniczenia dotyczące liczby żądań.

```
curl --get 'https://api.twitter.com/1.1/statuses/user_timeline.json'
--data 'screen_name=twitterapi' --header 'Authorization: OAuth
oauth_consumer_key="w2444553d23cWKnuxrlvnsjWWQ",

oauth_nonce="dhg2222324b268a887cdd900009ge4a7346",
oauth_signature="Dqwe2jru1NWgdFIKm9cOvQhghmdP4c%3D",
oauth_signature_method="HMAC-SHA1", oauth_timestamp="1404519549",
oauth_token="456356j901-A88OLMupyw4iCnVAm24t33HmnuGOCuNzABhg5QJ3SN8Y",
oauth_version="1.0"'–verbose.
```

Wynik tego jest następujący:

```
GET /1.1/statuses/user_timeline.json?screen_name=twitterapi HTTP/1.1
Host: api.twitter.com
Accept: */*
HTTP/1.1 200 OK
...
"url":"http:\/\/t.co\/78pYTvWfJd","entities":{"url":{"urls":[{"url":"http:\/\/
t.co\/78pYTvWfJd","expanded_url":"http:\/\/dev.twitter.com","display_url":
"dev.twitter.com","indices":[0,22]}]},"description":{"urls":[]}},"protected":
false,"followers_count":2224114,"friends_count":48,"listed_count":12772,
"created_at":"Wed May 23 06:01:13 +0000 2007","favourites_count":26,
"utc_offset":-25200,"time_zone":"Pacific Time (US & Canada)","geo_enabled":
true,"verified":true,"statuses_count":3511,"lang":"en","contributors_enabled":
false,"is_translator":false,"is_translation_enabled":false,"profile_background_
color":"C0DEED","profile_background_image_url":"http:\/\/pbs.twimg.com\/
profile_background_images\/656927849\/miyt9dpjz77sc0w3d4vj...
```

Czasowniki i działania na zasobach

W poniższej tabeli znajduje się zestawienie najczęściej używanych czasowników API REST Twittera:

Czasownik	Opis
`GET`	Służy do pobierania zasobów, np. użytkowników, obserwujących, ulubionych, subskrybentów itd.
`POST`	Służy do tworzenia zasobów, np. użytkowników, obserwujących, ulubionych, subskrybentów itd.
`POST` z czasownikiem `update`	Służy do zmieniania zasobów. Na przykład aby zaktualizować przyjaźnie, należy użyć adresu URL *friendships/update*.
`POST` z czasownikiem `destroy`	Służy do usuwania zasobów, np. wiadomości, wyłączenia obserwowania kogoś itd. Przykładowy adres URL: *direct_messages/destroy*.

Wersjonowanie

Aktualnie API Twittera ma numer wersji 1.1. Obsługuje tylko format JSON, a obsługa formatów XML, RSS i Atom została zarzucona. W API tym wszystkie klienty chcące wysyłać żądania muszą być uwierzytelnione przy użyciu protokołu OAuth. API 1.0 jest wycofywane z użytku. Po upływie sześciomiesięcznego okresu przejściowego zostanie ono wyłączone.

Obsługa błędów

API Twittera zwraca standardowe kody HTTP w odpowiedzi na żądania REST. W przypadku powodzenia operacji zwraca więc kod `200 OK`; jeśli nie ma danych do zwrócenia, zwraca kod `304 Not Modified`; jeżeli użytkownik nie podał danych poświadczających lub podał niepoprawne dane, zwraca kod `401 Not Authorized`; gdy wystąpi jakaś awaria i trzeba to opublikować na forum, zwraca kod `500 Internal Server Error` itd. Wraz z informacjami o błędach API Twittera zwraca też kody błędów czytelne dla komputera. Na przykład kod błędu `32` w odpowiedzi oznacza, że serwer nie uwierzytelnił użytkownika. Więcej informacji na ten temat znajduje się na stronie *https://dev.twitter.com/docs/error-codes-responses*.

Zalecana lektura

Poniżej znajduje się parę przydatnych odnośników:

- Narzędzia Facebooka: *https://developers.facebook.com/tools/*.
- Twurl (cURL z obsługą OAuth dla Twittera): *https://github.com/twitter/twurl*.
- Dokumentacja API GitHub: *https://developer.github.com/v3/*.
- Dokumentacja API Twittera: *https://dev.twitter.com/docs/api/1.1*.
- Dokumentacja API Stripe: *https://stripe.com/docs/api*.

Podsumowanie

Dodatek ten zawiera opis implementacji interfejsów API w niektórych platformach internetowych — GitHub, Facebook i Twitter — oraz zastosowanych w nich technik realizacji zasad technologii REST. Choć użytkownik może wykorzystać informacje otrzymane z API REST na niezliczoną ilość sposobów, wszystkie te systemy łączą REST i JSON. API tych platform są wykorzystywane przez klientów internetowych i urządzenia przenośne. W tym dodatku znajduje się opis metod wersjonowania, czasowników, obsługi błędów oraz uwierzytelniania i autoryzacji żądań na podstawie protokołu OAuth 2.0.

Na początku książki opisałam podstawy technologii REST i sposoby tworzenia własnych usług typu RESTful. Potem przedstawiłam wiele różnych tematów, porad i najlepszych praktyk dotyczących tworzenia skalowalnych i wydajnych usług REST. Ponadto wymieniłam liczne biblioteki i narzędzia pomocne w testowaniu i dokumentowaniu usług REST oraz dodałam parę słów o pojawiających się nowych technologiach budowy API działających na bieżąco. W ostatnim rozdziale nie zabrakło także paru studiów przypadków użycia nowych technologii gniazd sieciowych i uchwytów sieciowych oraz kilku słów na temat przyszłości technologii REST.

Mamy nadzieję, że nasza skromna publikacja pomoże Ci zrozumieć API REST, a w przyszłości projektować oraz budować jeszcze lepsze.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

X

Z

Ż