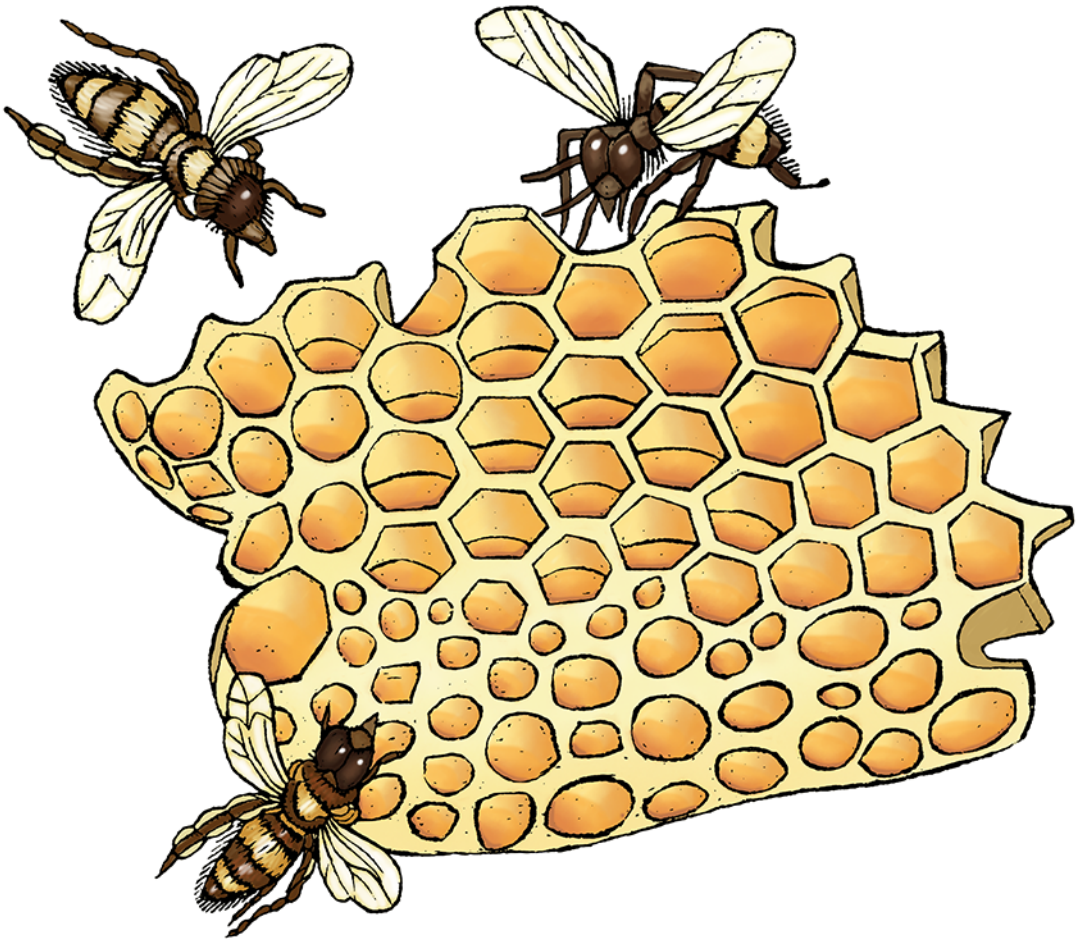
O'REILLY®

Wydanie II

Budowanie mikrousług

Projektowanie drobnoziarnistych systemów



Helion

Sam Newman

Tytuł oryginału: Building Microservices: Designing Fine-Grained Systems, 2nd Edition

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-8800-0

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/budmi2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 5.

Implementacja komunikacji mikrousług

Jak opisałem w poprzednim rozdziale, wybór technologii powinien być w dużej mierze sterowany wybranym stylem komunikacji. Wybór między połączeniami blokującymi i synchronicznymi a nieblokującymi i asynchronicznymi, stylem żądanie – odpowiedź a komunikacją sterowaną zdarzeniami pomoże Ci skrócić to, co w przeciwnym razie mogłoby być bardzo długą listą technologii. W tym rozdziale omówię wybrane technologie powszechnie używane do komunikacji mikrousług.

Poszukiwanie idealnej technologii

Istnieje oszałamiająca gama opcji dotyczących sposobów komunikowania się jednej mikrousługi z inną. Który z nich jest właściwy: SOAP, XML-RPC, REST, gRPC? A przecież ciągle pojawiają się nowe opcje. Zanim więc przystąpię do omówienia konkretnej technologii, zastanówmy się, co chcielibyśmy uzyskać z każdej z nich.

Łatwość zachowania zgodności wstecz

Podczas wprowadzania zmian do mikrousług trzeba zadbać o to, aby nie złamać zgodności z żadną z mikrousług korzystających z mikrousługi, którą zmieniamy. W związku z tym chcemy mieć pewność, że niezależnie od wybranej technologii wprowadzanie zmian w sposób zapewniający zgodność wstecz będzie łatwe. Takie proste operacje jak dodawanie nowych pól nie powinny powodować przerw w działaniu klientów. Idealnie byłoby również, gdyby istniała możliwość sprawdzenia, czy wprowadzone zmiany są zgodne wstecz oraz gdyby był sposób na uzyskanie tych informacji przed wdrożeniem mikrousługi do produkcji.

Zdefiniuj interfejs w sposób jawny

Istotne znaczenie ma to, aby interfejs, który mikrousługa udostępnia światu zewnętrznemu, był wyraźny. Oznacza to, że dla konsumenta mikrousługi powinno być jasne, jakie funkcjonalności udostępnia ta mikrousługa. Ale oznacza także, że dla programisty pracującego nad mikrousługą powinno być jasne, jakie funkcjonalności muszą pozostać nienaruszone dla podmiotów zewnętrznych — chcemy uniknąć sytuacji, w której modyfikacja mikrousługi spowoduje przypadkowe naruszenie zgodności.

W zapewnieniu jawności interfejsu udostępnianego przez mikrousługę mogą bardzo pomóc jawne schematy. W przypadku niektórych technologii użycie schematu jest wymagane, natomiast w innych przypadkach jest ono opcjonalne. Tak czy inaczej, zdecydowanie zachęcam do korzystania z jawnych schematów, a także do utworzenia pomocniczej dokumentacji wystarczającej do tego, aby jasno określić funkcjonalności, jakich konsument może oczekiwać od mikrousługi.

Zachowaj niezależność technologii interfejsów API

Jeśli jesteś w branży IT dłużej niż 15 minut, nie muszę Ci mówić, że pracujemy w obszarze, który szybko się zmienia. Jedyną rzeczą pewną są zmiany. Cały czas pojawiają się nowe narzędzia, frameworki i języki, powstają implementacje coraz to nowych pomysłów, które mogą pomóc pracować szybciej i skuteczniej. Być może w tej chwili programujesz w .NET, ale co będzie za rok lub za pięć lat? Co zrobić, jeśli chcesz eksperymentować z alternatywnymi stosami technologicznymi, które mogą spowodować, że staniesz się bardziej wydajny?

Jestem fanem utrzymywania otwartych opcji, dlatego jestem tak wielkim zwolennikiem mikrousług. Dlatego też uważam, że bardzo ważne jest zapewnienie niezależności interfejsów API używanych do komunikacji między technologiami mikrousług. Oznacza to unikanie korzystania z takich technologii integracji, które dyktują, jakich stosów technologicznych możemy użyć do zaimplementowania naszych mikrousług.

Spraw, aby Twoja usługa była prosta dla konsumentów

Chcemy, aby korzystanie z mikrousługi było dla konsumentów łatwe. Posiadanie pięknie „skrojonej" mikrousługi nie będzie miało większego znaczenia, jeśli koszt korzystania z niej będzie dla konsumenta niebotycznie wysoki! Zastanów się więc, co ułatwia konsumentom korzystanie z naszej wspaniałej, nowej usługi. Byłoby idealnie, gdybyśmy umożliwili klientom pełną swobodę w wyborze technologii. Z drugiej zaś strony udostępnienie biblioteki klienckiej może ułatwić przyjęcie konkretnej opcji. Często jednak takie biblioteki są niekompatybilne z innymi celami, które chcemy osiągnąć. Na przykład skorzystanie z bibliotek klienckich może ułatwić konsumentom integrację usługi, ale jednocześnie może powodować ściślejsze sprzężenia.

Ukryj szczegóły wewnętrznej implementacji

Nie chcemy, aby konsumentów naszej usługi wiązały jej wewnętrzne szczegóły implementacji, ponieważ prowadzi to do zwiększonych sprzężeń. To z kolei oznacza, że jeśli chcemy coś zmienić w naszej mikrousłudze, możemy naruszyć zgodność z jej konsumentami, co spowoduje konieczność wprowadzenia zmian również wewnątrz nich. Taka sytuacja zwiększa koszty zmian — a właśnie tego staramy się uniknąć. Oznacza to również, że z obawy przed koniecznością aktualizacji konsumentów będziemy skłonni do rzadszego wprowadzania zmian. W konsekwencji może to prowadzić do zwiększenia długu technicznego w ramach usługi. Z tych powodów należy unikać wszelkich technologii, które zmuszają do ujawniania wewnętrznych szczegółów implementacji.

Wybór technologii

Istnieje mnóstwo technologii, na które moglibyśmy spojrzeć, ale zamiast patrzeć szeroko na długą listę opcji, skupię uwagę na kilku spośród najbardziej popularnych i interesujących wyborów. Oto technologie, którym przyjrzymy się bliżej:

Zdalne wywołania procedur

Frameworki umożliwiające lokalnym metodom wywoływanie metod w zdalnym procesie. Typowe opcje to SOAP i gRPC.

REST

Styl architektury, w którym udostępnia się zasoby (Klient, Zamówienie itp.), do których można uzyskać dostęp za pomocą powszechnie znanego zestawu czasowników (`GET`, `POST`). REST to nieco więcej niż ta wzmianka, ale zajmę się tym wkrótce.

GraphQL

Stosunkowo nowy protokół, który pozwala konsumentom definiować niestandardowe zapytania pozwalające na pobieranie informacji z wielu mikrousług działających w dole strumienia przetwarzania. Protokół ten pozwala także na filtrowanie wyników, tak by użytkownik mógł odbierać tylko to, co jest mu potrzebne.

Brokery wiadomości

Oprogramowanie middleware umożliwiające asynchroniczne komunikowanie się za pośrednictwem kolejek lub tematów.

Zdalne wywołania procedur

Zdalne wywołania procedur (ang. *Remote Procedure Call* — RPC) odnoszą się do techniki polegającej na wykonywaniu lokalnego wywołania, które jest uruchamiane w usłudze zdalnej. Istnieje wiele różnych implementacji RPC. Większość technologii w tej przestrzeni wymaga jawnego schematu, takiego jak SOAP lub gRPC. W kontekście RPC schemat jest często określany jako język definicji interfejsu (ang. *Interface Definition Language* — IDL), a w technologii SOAP schemat jest określany jako WSDL (ang. *Web Service Definition Language* — dosłownie: język definicji usługi sieciowej). Użycie osobnego schematu ułatwia generowanie namiastek[1] (ang. *stubs*) klienta i serwera dla różnych stosów technologicznych — na przykład mogę zdefiniować serwer Javy udostępniający interfejs SOAP i klienta .NET wygenerowanego z tej samej definicji interfejsu w formacie WSDL. Inne technologie, na przykład RMI dla Javy, wymagają ściślejszego sprzężenia między klientem a serwerem, w związku z czym istnieje wymaganie, by zarówno klient, jak i serwer używały tej samej technologii. Nie muszą one jednak jawnie definiować usług, ponieważ definicja usługi jest domyślnie dostarczana przez definicje typu w Javie. Wszystkie te technologie mają jednak tę samą podstawową cechę: sprawiają, że połączenie zdalne wygląda jak połączenie lokalne.

[1] Źródło: *https://www.computerworld.pl/slownik/termin/47582/stub.html — przyp. tłum.*

Zazwyczaj korzystanie z technologii RPC oznacza konieczność skorzystania z protokołu serializacji. Framework RPC definiuje sposób serializacji i deserializacji danych. Na przykład w gRPC do tego celu używa serializacji za pomocą tzw. buforów protokołu (ang. *protocol buffers*). Niektóre implementacje są powiązane z określonym protokołem sieciowym (na przykład SOAP, który nominalnie korzysta z protokołu HTTP), podczas gdy inne pozwalają na korzystanie z różnych typów protokołów sieciowych, które mogą zapewniać dodatkowe funkcjonalności. Na przykład TCP zapewnia gwarancje dostawy, podczas gdy UDP ich nie zapewnia, ale wiąże się ze znacznie niższymi kosztami ogólnymi. Dzięki temu możliwe jest stosowanie różnych technologii sieciowych dla różnych przypadków użycia.

Frameworki RPC, które korzystają z jawnego schematu, bardzo ułatwiają generowanie kodu klienta. Pozwala to uniknąć konieczności korzystania z bibliotek klienckich, ponieważ każdy klient może po prostu wygenerować własny kod zgodnie ze specyfikacją usługi. Aby generowanie kodu po stronie klienta działało, klient potrzebuje jakiegoś sposobu na wyprowadzenie schematu — innymi słowy, konsument musi mieć dostęp do schematu, zanim zaplanuje wykonanie połączenia. Interesującą technologią jest w tym obszarze Avro RPC, ponieważ ma opcję wysyłania wraz z ładunkiem danych pełnego schematu, co umożliwia klientom dynamiczną interpretację schematu.

Łatwość generowania kodu po stronie klienta jest jednym z głównych cech różniących produkty RPC między sobą. Możliwość wykonania zwyczajnego wywołania metody i teoretycznie zignorowania reszty to ogromna korzyść.

Wyzwania

Jak widzieliśmy, technika RPC ma kilka wspaniałych zalet, ale niestety nie jest pozbawiona wad. Niektóre implementacje RPC mogą stwarzać więcej problemów niż inne. Wiele z tych kwestii można rozwiązać, ale warto przyjrzeć im się nieco bliżej.

Sprzężenia technologiczne. Niektóre mechanizmy RPC, takie jak Java RMI, są ściśle powiązane z określoną platformą, co może wprowadzać ograniczenia co do technologii dozwolonych do wykorzystania po stronie klienta i serwera. Technologie Thrift i gRPC charakteryzują się imponującym poziomem wsparcia dla języków alternatywnych, co może nieco złagodzić tę wadę. Należy jednak pamiętać, że technologia RPC czasami wiąże się z ograniczeniami interoperacyjności.

W pewnym sensie sprzężenie technologiczne można uznać za rodzaj ujawnienia wewnętrznych, technicznych szczegółów implementacji. Na przykład użycie usługi RMI wiąże z JVM nie tylko klienta, ale także serwer.

Aby być uczciwym, należy dodać, że istnieje wiele implementacji RPC, które nie mają takich ograniczeń — gRPC, SOAP i Thrift to przykłady mechanizmów pozwalających na współpracę między różnymi stosami technologii.

Wywołania lokalne to nie to samo co zdalne. Podstawową ideą RPC jest ukrycie złożoności zdalnego wywołania. Może to jednak prowadzić do sytuacji, w której ukrywamy zbyt wiele. Dążenie w niektórych technologiach RPC, aby zdalne wywołania metod wyglądały jak lokalne wywołania metod, ukrywa fakt, że te dwa rodzaje wywołań bardzo się między sobą różnią. Mogę wykonywać dużą liczbę lokalnych, wewnątrzprocesowych wywołań, nie martwiąc się zbytnio o wydajność.

Jednak w przypadku RPC koszty serializacji i deserializacji ładunków danych mogą być znaczące, nie wspominając o czasie poświęconym na wysyłanie informacji przez sieć. Oznacza to, że o projektowaniu interfejsów API dla interfejsów zdalnych musisz myśleć inaczej niż w przypadku interfejsów lokalnych. Próba bezmyślnego uczynienia z lokalnego interfejsu API granicy usługi może wpędzić Cię w kłopoty. W niektórych, najgorszych przykładach, jeśli abstrakcja nie jest odpowiednio przejrzysta, programiści mogą używać zdalnych wywołań, nawet o tym nie wiedząc.

Należy pomyśleć o samej sieci. Jednym z największych błędnych założeń przetwarzania rozproszonego jest stwierdzenie „sieć jest niezawodna" (*https://oreil.ly/8J4Vh*). Sieci *nie są* niezawodne. Mogą zawieść i czasami zawodzą, nawet jeśli klient i serwer, z którym się komunikujesz, działają bez zarzutu. Mogą zawieść nagle i mogą psuć się powoli, a nawet mogą zniekształcać przesyłane pakiety. Powinieneś założyć, że Twoje sieci są nękane przez wrogie istoty gotowe wyzwolić swój gniew z kaprysu. Dlatego powinieneś się spodziewać różnych rodzajów trybów awarii, z którymi być może nigdy nie miałeś do czynienia w prostszym, monolitycznym oprogramowaniu. Awaria może być spowodowana przez zdalny serwer zwracający błąd lub przez nieprawidłowe wywołanie. Czy potrafisz spostrzec tę różnicę, a jeśli tak, to czy możesz coś z tym zrobić? A co zrobić, gdy zdalny serwer dopiero zaczyna odpowiadać powoli? Temat ten omówię szerzej przy okazji opisywania niezawodności w rozdziale 12.

Kruchość. Niektóre z najpopularniejszych implementacji RPC mogą prowadzić do nieprzyjemnych form kruchości. Dobrym przykładem jest technologia Java RMI. Rozważmy bardzo prosty interfejs Javy, dla którego zdecydowaliśmy się stworzyć zdalny interfejs API do usługi `Klient`. Na listingu 5.1 znajdują się deklaracje metod, które mają być dostępne zdalnie. Następnie Java RMI generuje namiastki kodu klienta i serwera dla tworzonej metody.

Listing 5.1. Definiowanie punktu końcowego usługi przy użyciu Java RMI

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface CustomerRemote extends Remote {
  public Customer findCustomer(String id) throws RemoteException;

public Customer createCustomer(
  String firstname, String surname, String emailAddress)
  throws RemoteException;
}
```

W tym interfejsie metoda `createCustomer` pobiera imię, nazwisko i adres e-mail. Co zrobić, jeśli zdecydujemy, aby utworzenie obiektu `Customer` było możliwe również za pomocą samego adresu e-mail? W tym momencie dość łatwo moglibyśmy dodać nową metodę, na przykład:

```
...
public Customer createCustomer(String emailAddress) throws RemoteException;
...
```

Problem polega na tym, że teraz musimy również ponownie wygenerować namiastki klienta. Klienty, które zechcą skorzystać z nowej metody, potrzebują nowych namiastek, a w zależności od charakteru zmian w specyfikacji konsumenci, którzy nie potrzebują nowej metody, również mogą potrzebować aktualizacji namiastek. Jest to oczywiście możliwe do opanowania, ale tylko do pewnego stopnia. Rzeczywistość jest taka, że takie zmiany są dość powszechne. Punkty końcowe RPC

często mają dużą liczbę metod dla różnych sposobów tworzenia obiektów lub interakcji z nimi. Wynika to częściowo z faktu, że nadal myślimy o tych zdalnych wywołaniach tak, jakby były lokalne.

Istnieje jednak również inny rodzaj kruchości. Rzućmy okiem na to, jak wygląda obiekt `Customer`:

```
public class Customer implements Serializable {
    private String firstName;
    private String surname;
    private String emailAddress;
    private String age;
}
```

Co się stanie, jeśli okaże się, że chociaż w obiektach `Customer` zostało udostępnione pole `age`, to żaden z konsumentów usługi nigdy z niego nie korzysta? Decydujemy, że chcemy usunąć to pole. Ale jeśli w implementacji serwera usuniemy `age` z definicji typu, a nie zrobimy tego samego we wszystkich konsumentach, to nawet jeśli nigdy nie korzystali z pola, kod związany z deserializacją obiektu `Customer` po stronie konsumenta przestanie działać. Aby wprowadzić tę zmianę, trzeba by wprowadzić w kodzie klienta zmiany w celu obsługi nowej definicji i wdrożyć te zaktualizowane klienty jednocześnie z wdrożeniem nowej wersji serwera. Jest to kluczowe wyzwanie w przypadku każdego mechanizmu RPC, który promuje korzystanie z binarnego generowania namiastek: nie można rozdzielić wdrożenia klientów od serwera. Jeśli korzystasz z tej technologii, musisz liczyć się z koniecznością publikacji typu *lockstep*.

Podobne problemy występują w przypadku, gdy chcemy zmodyfikować strukturę obiektu `Customer`, nawet bez usuwania pól — na przykład gdybyśmy dla ułatwienia posługiwania się imieniem i nazwiskiem chcieli zhermetyzować pola `firstName` i `surname` w nowym typie `naming`. Moglibyśmy oczywiście to naprawić, przekazując w roli parametrów wywołań typy słownikowe, ale w tym momencie stracilibyśmy wiele korzyści z wygenerowanych namiastek, ponieważ nadal musielibyśmy ręcznie dopasowywać i wyodrębniać potrzebne pola.

W praktyce obiekty używane jako część serializacji binarnej przez sieć powinny być uważane za typy „tylko do rozszerzania". Z powodu opisanej kruchości typy są odsłonięte w komunikacji sieciowej. Z biegiem czasu mogą przyjąć formę zbioru pól, z których niektóre nie są już używane, ale nie można ich bezpiecznie usunąć.

Gdzie korzystać z tej technologii?

Pomimo pewnych niedociągnięć w gruncie rzeczy bardzo lubię RPC. Bardziej nowoczesne implementacje, takie jak gRPC, są doskonałe, podczas gdy z innymi występują istotne problemy, z powodu których najchętniej omijałbym je szerokim łukiem. Na przykład Java RMI ma wiele problemów dotyczących kruchości i ograniczonych wyborów technologicznych, a SOAP jest dość trudny z punktu widzenia programisty, szczególnie w porównaniu z technologiami bardziej nowoczesnymi.

Jeśli zamierzasz skorzystać z modelu RPC, pamiętaj o potencjalnych pułapkach związanych z tą technologią. Nie twórz abstrakcji dla zdalnych połączeń do takiego stopnia, że sieć stanie się całkowicie ukryta, i zadbaj o możliwość rozwijania interfejsu serwera bez konieczności wykonywania aktualizacji lockstep klientów. Bardzo ważne jest znalezienie odpowiedniej równowagi dla kodu klienta. Zadbaj o to, aby klienty miały świadomość faktu wykonywania połączenia sieciowego. W kontekście RPC często są używane biblioteki klienckie, a jeśli nie mają odpowiedniej struktury, mogą stwarzać problemy. Wkrótce zajmę się tym bardziej szczegółowo.

Gdybym poszukiwał opcji w tym obszarze, technologia gRPC byłaby na początku mojej listy. Została zbudowana z myślą o wykorzystaniu protokołu HTTP/2 i ma imponującą charakterystykę wydajności oraz dobrą ogólną łatwość użycia. Doceniam również ekosystem wokół technologii gRPC, w tym takie narzędzia jak Protolock (*https://protolock.dev*), które omówię w dalszej części tego rozdziału, przy okazji opisywania schematów.

gRPC dobrze pasuje do synchronicznego modelu żądanie – odpowiedź, ale może również działać w połączeniu z rozszerzeniami reaktywnymi. Jest wysoko na mojej liście, gdy jestem w sytuacjach, w których mam dużą kontrolę zarówno nad klientem, jak i serwerem. Jeśli musisz obsługiwać wiele innych aplikacji, które mogą wymagać komunikacji z mikrousługami, potrzeba skompilowania kodu po stronie klienta względem schematu po stronie serwera może stwarzać problemy. W takim przypadku jakaś forma API REST przez HTTP prawdopodobnie byłaby lepsza.

REST

REST (*Representational State Transfer*) to styl architektoniczny inspirowany siecią WWW. Styl REST obejmuje wiele zasad i ograniczeń. Tutaj skupię się na tych, które najbardziej nam pomagają, gdy napotykamy wyzwania związane z integracją w świecie mikrousług oraz gdy szukamy alternatywy dla RPC dla naszych interfejsów usług.

Najważniejsze w technologii REST jest pojęcie zasobów. Zasób możesz porównać do podmiotu, którego usługa jest świadoma — może nim być na przykład obiekt `Customer`. Serwer na żądanie tworzy różne reprezentacje tego obiektu. Sposób, w jaki zasób jest prezentowany na zewnątrz, jest całkowicie oddzielony od sposobu, w jaki jest przechowywany wewnętrznie. Klient może na przykład poprosić o reprezentację JSON obiektu `Customer`, nawet jeśli jest ona przechowywana w zupełnie innym formacie. Gdy klient posiada odpowiednią reprezentację obiektu `Customer`, może kierować żądania modyfikacji obiektu, a serwer może, choć nie musi, te żądania spełniać.

Istnieje wiele różnych stylów REST. W tym punkcie poprzestanę jedynie na krótkiej wzmiance na ich temat. Gorąco polecam zapoznanie się z artykułem „Richardson Maturity Model" (*https://oreil.ly/AlDzu*), w którym zostały porównane różne style REST.

Sam REST w gruncie rzeczy nie mówi niczego o wykorzystywanych protokołach, chociaż najczęściej jest wykorzystywany za pośrednictwem protokołu HTTP. Spotykałem implementacje REST korzystające z bardzo różnych protokołów, chociaż taka implementacja może wymagać dużo pracy. Niektóre funkcjonalności, które protokół HTTP daje nam w ramach specyfikacji, na przykład czasowniki, ułatwiają implementację REST przez HTTP, podczas gdy w przypadku innych protokołów trzeba te funkcjonalności zaimplementować samodzielnie.

REST i HTTP

Sam protokół HTTP definiuje kilka przydatnych własności, które bardzo dobrze pasują do stylu REST. Na przykład czasowniki HTTP (takie jak `GET`, `POST` i `PUT`) mają w specyfikacji HTTP dobrze rozumiane znaczenie, dlatego można je łatwo wykorzystać do pracy z zasobami. Styl architektoniczny REST faktycznie mówi nam, że czasowniki powinny zachowywać się w taki sam sposób dla wszystkich zasobów, natomiast specyfikacja HTTP definiuje kilka czasowników, których można

użyć do tego celu. Na przykład czasownik `GET` pobiera zasób w sposób idempotentny, a czasownik `POST` tworzy nowy zasób. Oznacza to, że możemy uniknąć wielu różnych metod w stylu `createCustomer` lub `editCustomer`. Zamiast tego możemy po prostu zastosować czasownik `POST` w celu opublikowania reprezentacji klienta, aby zażądać od serwera stworzenia nowego zasobu, a następnie zainicjować żądanie `GET` w celu pobrania reprezentacji zasobu. Koncepcyjnie w takich przypadkach istnieje jeden *punkt końcowy* w postaci zasobu `Customer`, a operacje, które możemy na nim wykonać, są wbudowane w protokół HTTP.

Protokół HTTP niesie ze sobą również rozbudowany ekosystem narzędzi i technologii pomocniczych. Korzystamy z serwerów proxy buforowania HTTP, takich jak Varnish, a także z modułów równoważenia obciążenia, takich jak `mod_proxy`. Wiele narzędzi do monitorowania ma gotowe wsparcie dla protokołu HTTP. Wspomniane bloki konstrukcyjne pozwalają obsługiwać ruch HTTP o dużym natężeniu i inteligentnie nim kierować w dość przejrzysty sposób. Można również skorzystać ze wszystkich dostępnych dla protokołu HTTP mechanizmów kontroli bezpieczeństwa, co pozwala odpowiednio zabezpieczyć komunikację. Ekosystem HTTP oferuje wiele narzędzi ułatwiających proces zabezpieczania zasobów — począwszy od podstawowego uwierzytelniania, po certyfikaty klientów. Temat ten omówię bardziej szczegółowo w rozdziale 11. Należy jednak pamiętać, że aby uzyskać te korzyści, trzeba korzystać z HTTP w odpowiedni sposób. Jeśli będziesz używać go źle, może być równie niepewny i trudny do skalowania jak każda inna technologia.

Warto zwrócić uwagę, że protokół HTTP można również wykorzystać do implementacji mechanizmów RPC. Na przykład SOAP działa przez HTTP, ale niestety używa bardzo niewielkiego fragmentu specyfikacji HTTP. Czasowniki, podobnie jak proste mechanizmy takie jak kody błędów HTTP, są ignorowane. Z drugiej strony gRPC został zaprojektowany w taki sposób, aby korzystał z możliwości protokołu HTTP/2 — na przykład wysyłania wielu strumieni żądanie – odpowiedź za pośrednictwem jednego połączenia. Ale, co oczywiste, gdy korzystasz z gRPC, nie korzystasz z REST tylko dlatego, że używasz HTTP!

Hipermedia jako silnik stanu aplikacji

Inną zasadą wprowadzoną w REST, która może pomóc w uniknięciu sprzężeń pomiędzy klientem a serwerem, jest koncepcja *hipermediów jako silnika stanu aplikacji* (często określana skrótem HATEOAS — od ang. *hypermedia as the engine of application state*, choć nie rozumiem, po co ten skrót). To trochę niejasny termin i dość interesująca koncepcja, więc spróbuję przyjrzeć się jej nieco dokładniej.

Hipermedia to koncepcja, zgodnie z którą treść zawiera linki do różnych innych treści w różnych formatach (np. tekst, ilustracje, dźwięki). Powinno to być wszystkim dość dobrze znane, ponieważ tak wygląda większość stron internetowych: podążasz za linkami będącymi formą hipermedialnych kontrolek, aby obejrzeć powiązane treści. Ideą HATEOAS jest to, że klienci powinni wykonywać interakcje z serwerem (potencjalnie prowadzące do zmian stanu) za pośrednictwem linków do innych zasobów. Aplikacja kliencka nie musi wiedzieć, gdzie na serwerze znajdują się dane kontrahentów, poprzez zapamiętywanie adresów URI, pod które ma trafić. Zamiast tego, aby znaleźć to, co jest jej potrzebne, szuka linków i nawiguje po nich.

Jest to trochę dziwna koncepcja, więc najpierw zróbmy krok wstecz, by zastanowić się, w jaki sposób ludzie komunikują się ze stronami internetowymi, na których — co już ustaliliśmy — znajduje się mnóstwo hipermedialnych kontrolek.

Weźmy za przykład witrynę zakupową *Amazon.com*. Lokalizacja koszyka na zakupy zmieniała się w czasie. Grafika uległa zmianie. Link uległ zmianie. Ale ludzie są na tyle inteligentni, że nadal widzą ikonę koszyka na zakupy, wiedzą, co to jest, i z niego korzystają. Ludzie rozumieją, co oznacza koszyk na zakupy, nawet jeśli zmieniła się jego forma oraz kontrolka używana do jego reprezentowania. Wiemy, że jeśli chcemy przejrzeć zawartość koszyka, to powinniśmy odszukać właściwą kontrolkę i wejść z nią w interakcję. Dzięki temu strony internetowe mogą stopniowo zmieniać się w czasie. Dopóki spełnione są niejawne kontrakty pomiędzy klientem a witryną, zmiany nie muszą powodować problemów z komunikacją.

Dzięki hipermedialnym kontrolkom staramy się osiągnąć ten sam poziom „inteligencji" dla naszych elektronicznych konsumentów. Weźmy za przykład hipermedialną kontrolkę, która mogłaby występować w systemie firmy MusicCorp. Za pomocą kodu z listingu 5.2 uzyskaliśmy dostęp do zasobu reprezentującego pozycję katalogu dla wybranego albumu. Wraz z informacjami o albumie widzimy szereg hipermedialnych kontrolek.

Listing 5.2. Kontrolki hipermedialne używane dla listy albumów

```
<album>
    <name>Give Blood</name>
    <link rel="/artist" href="/artist/theBrakes" /> ❶
    <description>
        Niesamowite, krótkie, brutalne, zabawne i głośne. Musisz go kupić!
    </description>
    <link rel="/instantpurchase" href="/instantPurchase/1234" /> ❷
</album>
```

❶ Ta kontrolka hipermedialna pokazuje nam, gdzie znaleźć informacje o artyście.

❷ A jeśli chcesz kupić album, wiesz już, gdzie się udać.

W tym dokumencie mamy dwie kontrolki hipermedialne. Aplikacja kliencka czytająca taki dokument musi wiedzieć, że aby uzyskać informacje o artyście, musi znaleźć kontrolkę z relacją `artist`, a `instantpurchase` to część protokołu używanego do zakupu albumu. Aplikacja kliencka musi rozumieć semantykę API w taki sam sposób, w jaki człowiek musi rozumieć, że w sklepie internetowym koszyk jest miejscem, w którym znajdą się kupowane towary.

Jako klient nie muszę wiedzieć, do którego schematu URI uzyskać dostęp, aby *kupić* album. Muszę tylko uzyskać dostęp do zasobu, znaleźć kontrolkę zakupu i do niej przejść. Kontrolka zakupu może zmienić lokalizację, może zmienić się jej identyfikator URI, a nawet witryna może wysłać mnie do innej usługi, ale mnie, jako klienta, to nie obchodzi. Daje to nam świetny poziom oddzielenia klienta od serwera.

Abstrahujemy tutaj od szczegółów. Moglibyśmy całkowicie zmienić implementację sposobu prezentacji kontrolki, pod warunkiem że klient nadal będzie potrafił ją znaleźć. Kontrolka ta musi jedynie pasować do sposobu, w jaki klient rozumie protokół, na takiej samej zasadzie, na jakiej kontrolka koszyka na zakupy może zmienić się z prostego łącza do bardziej skomplikowanej kontrolki JavaScript. Możemy również dodawać do dokumentu nowe kontrolki, być może reprezentujące

nowe zmiany stanu możliwe do wykonania na określonym zasobie. Działanie klientów uniemożliwilibyśmy tylko wtedy, gdybyśmy w zasadniczy sposób zmienili semantykę jednej z kontrolek, tak aby zaczęła zachowywać się zupełnie inaczej, lub gdybyśmy całkowicie ją usunęli.

Teoretycznie, używając kontrolek do oddzielenia klienta od serwera, z czasem uzyskujemy znaczne korzyści, które — przynajmniej takie jest założenie — powinny równoważyć wydłużenie czasu potrzebnego do skorzystania z wymaganych protokołów. Niestety, chociaż wszystkie te koncepcje wydają się w teorii sensowne, okazuje się, że z powodów, których nie do końca rozumiem, ta forma interfejsu REST jest rzadko praktykowana. To sprawia, że w szczególności koncepcja HATEOAS jest dla mnie znacznie trudniejsza do promowania wśród osób, które już zaangażowały się w korzystanie z technologii REST. Ogólnie rzecz biorąc, wiele pomysłów w REST opiera się na tworzeniu rozproszonych systemów hipermedialnych, a niestety większość systemów niestety tak nie działa.

Wyzwania

Jeśli chodzi o łatwość użycia, z powodów historycznych nie jest możliwe wygenerowanie kodu po stronie klienta dla interfejsu REST przez HTTP, tak jak w przypadku implementacji RPC. Często prowadzi to do tworzenia interfejsów API REST, które udostępniają biblioteki klienckie dla konsumentów. Wspomniane biblioteki klienckie zapewniają powiązanie z interfejsem API, co ułatwia integrację z klientami. Problem polega na tym, że biblioteki klienckie mogą powodować pewne sprzężenia pomiędzy klientami a serwerem. Zagadnienia te omówię w punkcie „Zasada DRY i niebezpieczeństwa wielokrotnego wykorzystywania kodu w świecie mikrousług".

W ostatnich latach problem ten został nieco złagodzony. Specyfikacja OpenAPI (*https://oreil.ly/Idr1p*), która wyrosła z projektu Swagger, zapewnia możliwość zdefiniowania w punkcie końcowym REST wystarczającej ilości informacji do tego, aby umożliwić generowanie kodu klienckiego w różnych językach. Z moich doświadczeń wynika, że niezbyt wiele zespołów faktycznie korzysta z tej funkcjonalności, nawet jeśli zespoły te używają systemu Swagger do obsługi dokumentacji. Podejrzewam, że może to wynikać z trudności związanych z integracją z istniejącym API. Mam również obawy dotyczące specyfikacji, która wcześniej była używana tylko do dokumentacji, a teraz jest używana do definiowania bardziej wyraźnego kontraktu. Może to prowadzić do znacznie bardziej złożonej specyfikacji — na przykład porównanie schematu OpenAPI ze schematem buforów protokołu daje dość ostry kontrast. Pomimo swoich zastrzeżeń uważam, że dobrze, iż taka opcja już istnieje.

Problemem może być również wydajność. Ładunki danych w REST przez HTTP mogą być bardziej kompaktowe niż w protokole SOAP, ponieważ REST obsługuje alternatywne formaty, takie jak JSON lub nawet formaty binarne. Nadal jednak nie jest tak zwięzłym protokołem binarnym jak Thrift. Narzut HTTP dla każdego żądania może również stanowić problem w przypadku, gdy są wymagane niskie opóźnienia. Wszystkie współcześnie stosowane odmiany protokołu HTTP wymagają użycia protokołu TCP (ang. *Transmission Control Protocol*), który ma niską wydajność w porównaniu z innymi protokołami sieciowymi, natomiast niektóre implementacje RPC umożliwiają korzystanie z innych niż TCP protokołów sieciowych, na przykład UDP (ang. *User Datagram Protocol*).

Obecnie trwają prace nad pokonaniem ograniczeń protokołu HTTP wynikających z wymogu korzystania z TCP. W protokole HTTP/3, który jest obecnie w trakcie finalizacji, nastąpi przejście na korzystanie z nowszego protokołu QUIC. QUIC zapewnia takie same możliwości jak TCP

(na przykład poprawiona niezawodność w stosunku do UDP), ale obejmuje pewne znaczące ulepszenia, które — jak pokazały badania — zapewniają mniejsze opóźnienia i zmniejszenie zużycia pasma. Zanim protokół HTTP/3 zacznie wywierać powszechny wpływ na publiczny internet, prawdopodobnie upłynie kilka lat, ale wydaje się rozsądne założenie, że firmy w swoich wewnętrznych sieciach mogą zacząć czerpać korzyści z nowego protokołu znacznie wcześniej.

Jeśli chodzi o HATEOAS, to można oczekiwać dodatkowych problemów z wydajnością. Konieczność nawigowania po wielu kontrolkach, aby znaleźć odpowiednie punkty końcowe dla danej operacji, może prowadzić do bardzo gadatliwych protokołów — dla każdej operacji może być wymagana komunikacja w dwie strony. Ostatecznie jest to kompromis. Jeśli zdecydujesz się przyjąć REST w stylu HATEOAS, to sugeruję, abyś zaczął od tego, by aplikacje klienckie najpierw w ogóle poruszały się po wymaganych kontrolkach, a następnie, jeśli zajdzie taka potrzeba, możesz dokonać odpowiednich optymalizacji. Należy zapamiętać, że zgodnie z tym, co powiedziałem wcześniej, korzystanie z HTTP pomaga w implementacji REST. Zło przedwczesnej optymalizacji zostało dobrze udokumentowane, więc nie muszę tutaj mówić o tym zbyt wiele. Należy również wziąć pod uwagę, że wiele ze wspomnianych technik opracowano w celu stworzenia rozproszonych systemów hipertekstowych i nie wszystkie z nich pasują do każdej sytuacji. Czasami może się okazać, że najlepsza będzie po prostu dobra, staromodna technika RPC.

Pomimo wymienionych wad REST za pośrednictwem HTTP to rozsądny, domyślny wybór dla interakcji między usługami. Jeśli chcesz dowiedzieć się więcej, polecam książkę *REST in Practice: Hypermedia and Systems Architecture* (O'Reilly) autorstwa Jima Webbera, Savasa Parastatidisa i Iana Robinsona, w której obszernie omówiono temat API REST przez HTTP.

Gdzie korzystać z tej technologii?

Ze względu na szerokie zastosowanie w branży interfejs API oparty na technice REST przez HTTP to oczywisty wybór dla synchronicznego interfejsu żądanie – odpowiedź. Komunikacja tego typu sprawdza się zwłaszcza wtedy, kiedy chcesz zapewnić dostęp do usługi z jak najszerszej gamy klientów. Błędem byłoby myślenie o REST API jako o wyborze „wystarczająco dobrym do większości zastosowań", ale w tym stwierdzeniu jest dużo prawdy. Jest to powszechnie rozumiany styl interfejsu znany większość osób, który gwarantuje interoperacyjność wielu różnych technologii.

W dużej mierze ze względu na możliwości protokołu HTTP oraz fakt, że REST w dużym stopniu opiera się na tych możliwościach (zamiast je ukrywać), interfejsy API bazujące na technice REST doskonale sprawdzają się w sytuacjach, w których zależy nam na wydajnym i wielkoskalowym buforowaniu żądań. Z tego powodu są one oczywistym wyborem do udostępniania interfejsów API aplikacjom zewnętrznym lub interfejsom klienckim. Mogą jednak okazać się gorsze w porównaniu z bardziej wydajnymi protokołami komunikacyjnymi i chociaż można konstruować protokoły interakcji asynchronicznych na bazie interfejsów API opartych na REST, nie jest to zbyt dobre dopasowanie, zwłaszcza w porównaniu z alternatywami dla generycznej komunikacji mikrousługi z innymi mikrousługami.

Pomimo szczytnych celów stojących za HATEOAS nie zaobserwowałem zbyt wielu dowodów na to, że dodatkowa praca nad implementacją tego stylu REST przynosi w dłuższej perspektywie odpowiednie korzyści. Nie przypominam również sobie, abym w ciągu ostatnich kilku lat rozmawiał

z wdrażającymi architekturę mikrousług zespołami, które wspominałyby o wartości korzystania z HATEOAS. Moje własne doświadczenia są oczywiście tylko jednym zestawem danych i nie wątpię, że istnieją osoby, dla których technika HATEOAS mogła się sprawdzić. Wydaje mi się jednak, że koncepcja ta nie przyciągnęła tak wielu zwolenników, jak się spodziewałem. Możliwe, że pojęcia stojące za HATEOAS są dla nas zbyt obce lub może zniechęcająco działa brak w tym obszarze narzędzi lub standardów, a może ten model po prostu nie sprawdza się dla systemów, które budowaliśmy. Możliwe jest również, że pojęcia stojące za HATEOAS nie komponują się dobrze z tym, w jaki sposób budujemy mikrousługi.

Tak więc można przyjąć, że technika ta sprawdza się do wykorzystania „na granicy" mikrousług oraz dla synchronicznego interfejsu opartego na żądaniach i odpowiedziach między mikrousługami.

GraphQL

W ostatnich latach GraphQL (*https://graphql.org*) zyskał większą popularność, w dużej mierze ze względu na to, że wyróżnia się w jednym konkretnym obszarze. Mianowicie umożliwia urządzeniu po stronie klienta zdefiniowanie zapytań, które mogą uniknąć konieczności wykonywania wielu żądań w celu pobrania tych samych informacji. Pozwala to osiągnąć znaczną poprawę wydajności dla ograniczonego zbioru urządzeń klienckich, a także uniknąć konieczności implementacji wspomnianych agregacji po stronie serwera.

W ramach prostego przykładu wyobraź sobie urządzenie mobilne, które chce wyświetlić stronę zawierającą przegląd najnowszych zamówień klienta. Strona powinna zawierać pewne informacje o kliencie wraz z danymi o jego pięciu ostatnich zamówieniach. Ekran wymaga tylko kilku pól z rekordu klienta oraz daty, wartości i statusu wysyłki każdego zamówienia. Urządzenie mobilne mogłoby w celu pobrania potrzebnych informacji nawiązywać połączenia z dwiema mikrousługami w dole strumienia przetwarzania, ale wymagałoby to wykonywania wielu połączeń, w tym pobierania informacji, które w rzeczywistości nie są niezbędne. Zwłaszcza w przypadku urządzeń mobilnych może to być marnotrawstwem — zużywa z dostępnego dla urządzenia mobilnego limitu więcej danych, niż jest to potrzebne, i może dłużej trwać.

GraphQL pozwala urządzeniu mobilnemu na wysłanie jednego zapytania, które pozwala pobrać wszystkie potrzebne informacje. Aby to zadziałało, potrzebna jest mikrousługa, która udostępnia urządzeniu klienckiemu punkt końcowy GraphQL. Ten punkt końcowy GraphQL jest punktem wejścia dla wszystkich zapytań klienta i udostępnia schemat do wykorzystania przez urządzenia klienckie. Schemat ten eksponuje dostępne dla klienta typy. Można również skorzystać z estetycznego konstruktora zapytań, ułatwiającego ich tworzenie. Dzięki zmniejszeniu liczby wywołań oraz ilości danych pobieranych przez urządzenie klienckie można sobie świetnie poradzić z niektórymi wyzwaniami występującymi podczas budowania interfejsów użytkownika korzystających z architektury mikrousług.

Wyzwania

W początkowej fazie rozwoju tej technologii jednym z wyzwań był brak obsługi dla specyfikacji GraphQL w językach programowania. Język JavaScript był początkowo jedynym wyborem. Obecnie sytuacja znacznie się poprawiła, a wsparcie dla tej specyfikacji mają wszystkie najważniejsze języki.

Wprowadzono istotne ulepszenia w GraphQL oraz różnych jej implementacjach, dzięki czemu jest to teraz znacznie mniej ryzykowna technologia niż jeszcze kilka lat temu. Niemniej warto mieć świadomość istnienia kilku wyzwań związanych ze stosowaniem tej technologii.

Po pierwsze urządzenie klienckie może wysyłać dynamicznie zmieniające się zapytania. Słyszałem o zespołach, które ze względu na tę możliwość doświadczały problemów z zapytaniami GraphQL, objawiających się znacznymi obciążeniami po stronie serwera. Kiedy porównamy GraphQL z taką technologią jak SQL, możemy dostrzec podobny problem. Kosztowna instrukcja SQL może spowodować poważne problemy dla bazy danych i potencjalnie może mieć duży wpływ na szerszy system. To samo dotyczy GraphQL. Różnica polega na tym, że w SQL mamy narzędzia, takie jak planery zapytań do baz danych, które mogą pomóc w zdiagnozowaniu problematycznych zapytań, podczas gdy podobne problemy z GraphQL mogą być trudniejsze do wyśledzenia. Jednym z potencjalnych rozwiązań jest dławienie żądań po stronie serwera, ale ponieważ realizacja wywołania może obejmować wiele mikrousług, to rozwiązanie nie należy do prostych.

W porównaniu ze zwykłymi interfejsami API HTTP opartymi na REST buforowanie jest również bardziej złożone. W przypadku interfejsów API opartych na REST można ustawić jeden z wielu nagłówków odpowiedzi, co może pomóc urządzeniom po stronie klienta lub pośrednim pamięciom podręcznym, takim jak sieci dostarczania zawartości (ang. *Content Delivery Networks* — CDN), buforować odpowiedzi, tak aby nie trzeba było ponownie wysyłać takich samych żądań. W przypadku technologii GraphQL nie jest to możliwe na takim samym poziomie. Rozwiązania tego problemu, z jakimi się spotykałem, bazują na powiązaniu z każdym zwracanym zasobem identyfikatora (trzeba pamiętać, że zapytanie GraphQL może zawierać wiele zasobów), a następnie buforowaniu żądania przez urządzenie klienckie na podstawie tego identyfikatora. To sprawia, że korzystanie z CDN lub buforowanie odwrotnych serwerów proxy, bez dodatkowej pracy, jest niezwykle trudne.

Chociaż spotkałem się z kilkoma rozwiązaniami tego problemu specyficznymi dla implementacji (na przykład Apollo w implementacji JavaScript), buforowanie w początkowej fazie rozwoju GraphQL wydaje się być świadomie lub nieświadomie zignorowane. Jeśli zapytania, które wysyłasz, mają bardzo specyficzny charakter dla konkretnego użytkownika, ten brak buforowania na poziomie żądania może oczywiście nie być zbyt znaczący, ponieważ współczynnik „trafień" w odwołaniach do pamięci podręcznej prawdopodobnie będzie niski. Zastanawiam się jednak, czy to ograniczenie oznacza, że nadal trzeba stosować hybrydowe rozwiązania dla urządzeń klienckich, z niektórymi (bardziej generycznymi) żądaniami przekazywanymi przez zwykłe interfejsy API HTTP oparte na REST oraz innymi żądaniami przekazywanymi przez GraphQL.

Inną kwestią jest to, że chociaż GraphQL teoretycznie może obsługiwać operacje zapisu, wydaje się, że technika ta nie pasuje tak dobrze do zapisów, jak do odczytów. Prowadzi to do sytuacji, w których zespoły używają GraphQL do odczytu, ale REST do zapisu.

Ostatnia kwestia jest czymś, co może być całkowicie subiektywne, ale nadal uważam, że warto ją poruszyć. GraphQL sprawia, że czujesz się tak, jakbyś po prostu pracował z danymi, co może wzmocnić wrażenie, że mikrousługi, z którymi się komunikujesz, są tylko opakowaniami dla baz danych. W rzeczywistości spotykałem wiele osób porównujących GraphQL do Odata — technologii zaprojektowanej jako generyczny interfejs API do uzyskiwania dostępu do danych z baz danych. Jak już wielokrotnie wspominałem, pomysł traktowania mikrousług jako opakowań dla baz danych

może być bardzo problematyczny. Mikrousługi udostępniają funkcjonalności za pośrednictwem interfejsów sieciowych. Niektóre z tych funkcjonalności mogą wymagać ujawniania danych lub je powodować, ale nadal powinny mieć własną wewnętrzną logikę i zachowanie. Nie wpadnij w pułapkę myślenia o swoich mikrousługach jako o niewiele więcej niż interfejsie API do baz danych tylko dlatego, że używasz GraphQL — ważne jest to, aby interfejs API GraphQL nie był sprzężony z bazowymi magazynami danych Twoich mikrousług.

Gdzie korzystać z tej technologii?

GraphQL świetnie nadaje się do wykorzystania na granicy systemu w celu udostępniania funkcjonalności klientom zewnętrznym. Klienty zazwyczaj są wyposażone w GUI. W oczywisty sposób, biorąc pod uwagę ograniczenia pod względem zdolności do prezentowania danych użytkownikowi oraz charakter sieci komórkowych, GraphQL pasuje do urządzeń mobilnych. Ale GraphQL jest również stosowany w zewnętrznych interfejsach API, a jednym z pierwszych zastosowań był serwis GitHub. Jeśli masz zewnętrzny interfejs API, który często wymaga od klientów zewnętrznych wielokrotnego wywoływania w celu uzyskania potrzebnych informacji, dzięki GraphQL możesz uzyskać interfejs API znacznie bardziej wydajny i przyjazny użytkownikom.

Ogólnie rzecz biorąc, GraphQL jest mechanizmem agregacji i filtrowania połączeń, więc w kontekście architektury mikrousług jest wykorzystywany do agregowania wywołań w wielu mikrousługach działających w dole strumienia przetwarzania. W związku z tym GraphQL nie jest technologią, która może zastąpić generyczną komunikację między mikrousługami.

Alternatywą dla użycia GraphQL jest zastosowanie innego wzorca, na przykład backendu dla frontendu (BFF). Temu wzorcowi wraz z porównaniem go z GraphQL i innymi technikami agregacji przyjrzymy się w rozdziale 14.

Brokery wiadomości

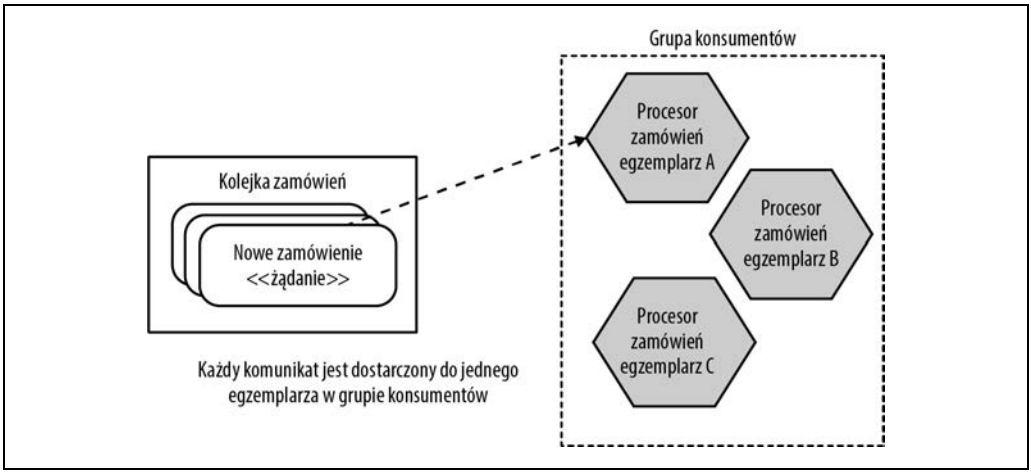
Brokery wiadomości to oprogramowanie pośredniczące (middleware), wykorzystywane pomiędzy procesami w celu zarządzania komunikacją między nimi. Ponieważ oferują wiele zaawansowanych funkcjonalności, są popularnym wyborem ułatwiającym implementację asynchronicznych mechanizmów komunikacji między mikrousługami.

Jak powiedziałem wcześniej, wiadomość (nazywana również komunikatem) jest generycznym pojęciem i definiuje informację wysyłaną przez brokera wiadomości. Wiadomość może zawierać żądanie, odpowiedź lub zdarzenie. Zamiast komunikować się bezpośrednio z inną mikrousługą, mikrousługa przekazuje komunikat do brokera wiadomości z informacją o tym, w jaki sposób ma on zostać wysłany.

Tematy i kolejki

Brokery zwykle dostarczają kolejki, tematy lub jedno i drugie. Kolejki zazwyczaj implementują komunikaty punkt – punkt. Nadawca umieszcza wiadomość w kolejce, a konsument ją z niej odczytuje. W systemach opartych na tematach wielu konsumentów może subskrybować temat, a każdy konsument, który dokonał subskrypcji, otrzymuje kopię komunikatu.

Konsument może reprezentować jedną lub więcej mikrousług — zazwyczaj są one modelowane jako grupa konsumentów. Taka konfiguracja jest przydatna w sytuacji, gdy mamy wiele egzemplarzy mikrousługi i chcemy, aby każdy z nich mógł odebrać komunikat. Na rysunku 5.1 widzimy przykład, w którym są trzy egzemplarze usługi `Procesor zamówień`. Wszystkie stanowią część tej samej grupy konsumentów. Gdy w kolejce zostanie umieszczona wiadomość, tylko jeden członek grupy konsumentów ją otrzyma. Oznacza to, że kolejka działa jako mechanizm dystrybucji obciążenia. Jest to przykład wzorca rywalizujących konsumentów, o których krótko wspominałem w rozdziale 4.



Rysunek 5.1. Kolejka pozwala na jedną grupę konsumentów

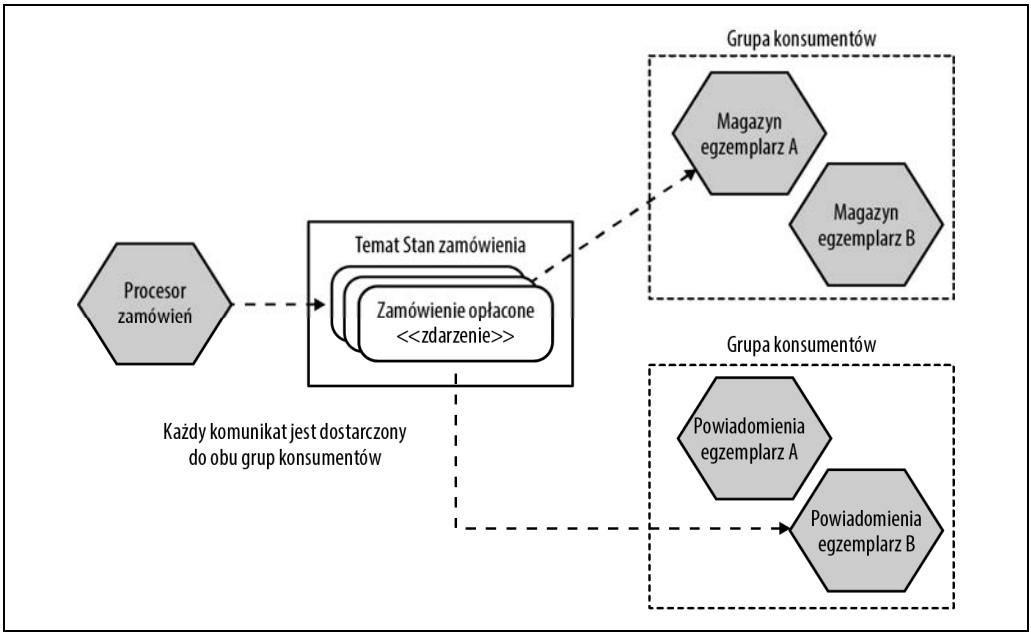
Tematy pozwalają stworzyć wiele grup konsumentów. Na rysunku 5.2 zdarzenie reprezentujące opłacone zamówienie zostało umieszczane w temacie `Stan zamówienia`. Kopię tego zdarzenia odbierze zarówno mikrousługa `Magazyn`, jak i mikrousługa `Powiadomienia`, należące do oddzielnych grup konsumentów. Zdarzenie to zobaczy tylko jeden egzemplarz każdej grupy konsumentów.

Na pierwszy rzut oka kolejka wygląda jak temat z jedną grupą konsumentów. Największa różnica między tymi dwoma pojęciami polega na tym, że gdy wiadomość jest wysyłana za pośrednictwem kolejki, to nadawca wie, gdzie wysyła wiadomość. W przypadku tematu informacje te są ukryte przed nadawcą wiadomości — nie jest on świadomy, jaki konsument otrzyma wiadomość (i czy w ogóle jakiś konsument ją otrzyma).

Tematy świetnie nadają się do współpracy opartej na zdarzeniach, podczas gdy kolejki lepiej sprawdzają się do komunikacji żądanie – odpowiedź. Należy to jednak traktować jako ogólne wytyczne, a nie jako ścisłą zasadę.

Gwarantowana dostawa

Po co więc korzystać z brokera? Ogólnie rzecz biorąc, brokery dają pewne możliwości, które mogą być bardzo przydatne w komunikacji asynchronicznej. Różne brokery charakteryzują się różnymi właściwościami, jednak najciekawszą cechą jest gwarantowana dostawa — własność, którą w pewien sposób wspierają wszystkie powszechnie wykorzystywane brokery. Gwarantowana dostawa opisuje zobowiązanie brokera do zapewnienia dostawy wiadomości.



Rysunek 5.2. Tematy umożliwiają wielu subskrybentom odbieranie tych samych wiadomości, co przydaje się do transmisji zdarzeń

Cecha ta może być bardzo przydatna z punktu widzenia mikrousługi wysyłającej wiadomość. Nie ma problemu, jeśli miejsce docelowe komunikatu jest niedostępne — broker będzie przechowywał wiadomość dopóty, dopóki nie zostanie dostarczona. Pozwala to zmniejszyć liczbę rzeczy, o które powinna zadbać mikrousługa działająca w górze strumienia przetwarzania. Warto porównać tę cechę z synchronicznym wywołaniem bezpośrednim — na przykład żądaniem HTTP: jeśli miejsce docelowe w dole strumienia przetwarzania nie jest osiągalne, mikrousługa działająca w górnej części strumienia musi ustalić, co zrobić z żądaniem — czy powinna ponowić próbę połączenia, czy zrezygnować?

Aby zagwarantować dostawę, broker musi zadbać o to, aby wszystkie wiadomości, które jeszcze nie zostały dostarczone, były w trwały sposób przechowywane — dopóki nie zostaną dostarczone. Aby spełnić tę obietnicę, broker zwykle działa jako rodzaj systemu opartego na klastrach — dba o to, aby utrata jednej maszyny nie spowodowała utraty całej wiadomości. Zazwyczaj prawidłowe działanie brokera wymaga sporo zachodu, częściowo ze względu na wyzwania związane z zarządzaniem oprogramowaniem opartym na klastrach. Często obietnica gwarantowanej dostawy nie może zostać spełniona, jeśli broker nie jest poprawnie skonfigurowany. Na przykład RabbitMQ wymaga, aby egzemplarze należące do klastra komunikowały się w sieciach o stosunkowo niskich opóźnieniach; w przeciwnym razie egzemplarze mogą posiadać nieaktualne informacje dotyczące bieżącego stanu obsługiwanych wiadomości, co może powodować utratę danych. Nie wymieniam tego konkretnego ograniczenia jako dowodu na to, że RabbitMQ jest w jakikolwiek sposób zły — wszystkie brokery mają ograniczenia co do tego, jak muszą być skonfigurowane, aby mogły spełnić obietnicę gwarantowanej dostawy. Jeśli planujesz samodzielnie uruchomić brokera wiadomości, to pamiętaj, żeby dokładnie zapoznać się z dokumentacją.

Warto również zauważyć, że dla każdego z brokerów znaczenie pojęcia gwarantowanej dostawy może być inne. Jak już wspominałem, warto zacząć od dokładnego zapoznania się z dokumentacją.

Zaufanie

Jedną z największych zalet brokerów jest własność gwarantowanej dostawy. Ale aby skorzystać z gwarantowanej dostawy, musisz zaufać nie tylko osobom, które stworzyły brokera, ale także sposobowi, w jaki on działa. Jeśli zbudowałeś system oparty na założeniu, że dostawa jest gwarantowana, a okazuje się, że z powodu problemu z wykorzystywanym brokerem tak nie jest, możesz wpaść w poważne tarapaty. Oczywiście decydując się na skorzystanie z brokera, mamy nadzieję, że powierzamy pracę oprogramowaniu stworzonemu przez ludzi potrafiących wykonać tę pracę lepiej, niż sami to potrafimy. Ostatecznie musisz zdecydować, jak bardzo chcesz zaufać brokerowi, z którego korzystasz.

Inne cechy

Oprócz gwarantowanej dostawy brokery mogą zapewnić inne własności, które mogą okazać się przydatne.

Większość brokerów potrafi zagwarantować kolejność dostarczania wiadomości. Cecha ta nie jest jednak uniwersalna, a nawet wtedy, gdy jest dostępna, zakres świadczonych gwarancji może być ograniczony. Na przykład w przypadku systemu Kafka kolejność jest gwarantowana tylko w ramach jednej partycji. Jeśli nie możesz mieć pewności, że wiadomości zostaną odebrane we właściwej kolejności, konsumenci Twojej usługi mogą potrzebować odpowiedniej rekompensaty, na przykład odroczenia przetwarzania wiadomości odebranych poza kolejnością do momentu otrzymania brakujących wiadomości.

Niektóre brokery zapewniają transakcje zapisu — na przykład system Kafka pozwala pisać do wielu tematów w jednej transakcji. Niektóre brokery mogą również zapewniać transakcyjność operacji odczytu. Z tej własności korzystałem dla wielu brokerów za pośrednictwem interfejsów API Java Message Service (JMS). Własność ta może się przydać w przypadku, gdy przed usunięciem wiadomości z brokera chcesz mieć pewność, czy może ona zostać przetworzona przez konsumenta.

Inną, nieco kontrowersyjną funkcjonalnością obiecywaną przez niektóre brokery jest dostawa dokładnie jednorazowa. Jednym z łatwiejszych sposobów zapewnienia gwarantowanej dostawy jest umożliwienie ponownego przesłania wiadomości. Może to spowodować, że konsument zobaczy tę samą wiadomość więcej niż raz (choć sytuacje tego rodzaju zdarzają się rzadko). Większość brokerów będzie dążyć do zminimalizowania zagrożenia wystąpienia takiej sytuacji lub będzie dążyć do ukrycia tego faktu przed konsumentem. Niektóre brokery zapewniają jednak więcej — gwarantują dostawę dokładnie jeden raz. Jest to złożony temat. Z moich rozmów z kilkoma ekspertami wynika, że zagwarantowanie dokładnie jednej dostawy we wszystkich przypadkach jest niemożliwe. Istnieją także eksperci, którzy twierdzą, że w zasadzie jest to możliwe do osiągnięcia za pomocą kilku prostych obejść. Tak czy inaczej, jeśli wybrany przez Ciebie broker implementuje tę cechę, zwróć *szczególną* uwagę na sposób jej implementacji. Będzie jeszcze lepiej, jeśli zbudujesz elektronicznych konsumentów w taki sposób, aby byli przygotowani na ewentualność otrzymania wiadomości więcej niż jeden raz i aby potrafili poradzić sobie z tą sytuacją. Bardzo prostym rozwiązaniem mogłoby

być przydzielenie do każdej wiadomości identyfikatora, który mógłby być sprawdzany przez konsumenta za każdym razem, gdy zostanie odebrana wiadomość. Jeśli wiadomość o danym identyfikatorze została już przetworzona, nowszą wiadomość można zignorować.

Opcje do wyboru

Istnieje wiele brokerów wiadomości. Popularne przykłady to RabbitMQ, ActiveMQ i Kafka (który wkrótce omówię dokładniej). Główni dostawcy chmury publicznej zapewniają również różnorodne produkty, które pełnią tę rolę — począwszy od wersji zarządzanych, które można zainstalować we własnej infrastrukturze, po implementacje indywidualne, specyficzne dla danej platformy. Na przykład dla usługi AWS istnieją opcje Simple Queue Service (SQS), Simple Notification Service (SNS) i Kinesis. Wszystkie one są różnymi odmianami w pełni zarządzanych brokerów. SQS był drugim w historii produktem wydanym przez AWS, który został wprowadzony na rynek w 2006 roku.

Kafka

Warto zauważyć, że system Kafka to specyficzny broker w dużej mierze ze względu na jego obecną popularność. Po części popularność ta wynika z wykorzystania systemu Kafka do wspomagania zadań transportu dużych ilości danych w ramach implementacji potoków przetwarzania strumieniowego. Zastosowanie systemu Kafka może pomóc w przejściu od przetwarzania wsadowego do przetwarzania w czasie rzeczywistym.

Istnieje kilka cech Kafki, na które warto zwrócić uwagę. Po pierwsze system ten został zaprojektowany z myślą o rozwiązaniach na bardzo dużą skalę — został zbudowany na potrzeby systemu LinkedIn w celu zastąpienia wielu istniejących klastrów wiadomości jedną platformą. System Kafka stworzono po to, aby umożliwić współdziałanie wielu konsumentom i producentom. Rozmawiałem z jednym ekspertem w dużej firmie technologicznej, która korzystała z ponad pięćdziesięciu tysięcy producentów i konsumentów pracujących w tym samym klastrze. Trzeba przyznać, że istnieje bardzo niewiele organizacji, które obsługiwałyby problemy na taką skalę, jednak w przypadku niektórych organizacji możliwość łatwego skalowania Kafki (relatywnie rzecz biorąc) może być bardzo przydatna.

Inną dość niepowtarzalną cechą Kafki jest utrwalanie wiadomości. Gdy ostatni konsument odbierze wiadomość, zwykły broker nie ma potrzeby dłużej tę wiadomość utrzymywać. Dzięki systemowi Kafka wiadomości mogą być przechowywane przez konfigurowalny czas. Oznacza to, że wiadomości mogą być przechowywane w nieskończoność. Właściwość ta pozwala konsumentom na ponowne pozyskiwanie wiadomości, które już zostały przez nich przetworzone. Pozwala też nowo zainstalowanym klientom przetwarzać wiadomości, które zostały wysłane wcześniej.

I na koniec: w systemie Kafka wprowadzono wbudowaną obsługę przetwarzania strumieniowego. Zamiast używać Kafki do wysyłania wiadomości do dedykowanego narzędzia przetwarzania strumieniowego, takiego jak Flink, niektóre zadania można wykonać wewnątrz samej Kafki. Za pomocą języka KSQL można zdefiniować instrukcje przypominające SQL, które mogą obsługiwać „w locie" jeden lub większą liczbę tematów. W ten sposób możemy uzyskać mechanizm podobny do dynamicznej aktualizacji zmaterializowanego widoku bazy danych, w którym źródłem danych zamiast bazy danych są tematy Kafki. Funkcjonalności te otwierają bardzo interesujące możliwości

zarządzania danymi w systemach rozproszonych. Chcącym zapoznać się z tymi pojęciami bardziej szczegółowo polecam książkę Bena Stopforda *Designing Event-Driven Systems* (O'Reilly) — muszę polecić książkę Bena, ponieważ napisałem do niej przedmowę. Aby dokładniej zapoznać się z systemem Kafka, warto przeczytać książkę Nehy Narkhede, Gwena Shapira i Todda Palino *Kafka: The Definitive Guide* (O'Reilly).

Formaty serializacji

Niektóre z opcji technologicznych, którym się przyjrzeliśmy — w szczególności niektóre implementacje RPC — umożliwiają dokonywanie wyborów dotyczących sposobu serializacji i deserializacji danych. Na przykład w przypadku gRPC wszystkie wysłane dane są konwertowane na format bufora protokołu. Wiele opcji technologicznych daje nam jednak większą swobodę w zakresie przesyłania danych w połączeniach sieciowych. Zastosowanie systemu Kafka jako brokera wiadomości pozwala wysyłać wiadomości w różnych formatach. Oto kilka możliwości do wyboru.

Formaty tekstowe

Zastosowanie standardowych formatów tekstowych daje klientom dużą elastyczność w sposobie wykorzystywania zasobów. W systemach korzystających z interfejsów API REST najczęściej używa się formatów tekstowych dla treści żądań i odpowiedzi, nawet jeśli teoretycznie można bez trudu wysyłać przez HTTP dane binarne. W rzeczywistości tak właśnie działa gRPC — używa pod spodem HTTP, ale stosuje do komunikacji protokół binarny.

Format JSON w dużej mierze zastąpił XML w zadaniach serializacji tekstu. Jest wiele powodów, dla których tak się stało, ale głównym jest to, że jednym z ważniejszych konsumentów interfejsów API są często przeglądarki, w których JSON doskonale się sprawdza. JSON stał się popularny częściowo w wyniku sprzeciwu wobec XML, a zwolennicy tego formatu powołują się na jego względną zwięzłość i prostotę w porównaniu z XML. W rzeczywistości jednak rzadko można dostrzec dużą różnicę między rozmiarem ładunku JSON a rozmiarem ładunku XML, zwłaszcza że ładunki te są zazwyczaj kompresowane. Warto również podkreślić, że część prostoty JSON ma swoją cenę — dążąc do przyjmowania prostszych protokołów, zapominamy o schematach (więcej na ten temat później).

Interesującym formatem serializacji jest Avro. Przyjmuje JSON jako strukturę bazową i używa go do zdefiniowania formatu opartego na schemacie. Avro zyskał dużą popularność jako format ładunków wiadomości, częściowo ze względu na charakterystyczną dla niego zdolność do wysyłania schematu jako części ładunku, co może znacząco ułatwić obsługę wielu różnych formatów wiadomości.

Osobiście jednak nadal jestem fanem XML. Nadal dla XML istnieje lepsze wsparcie ze strony narzędzi. Na przykład, aby wyodrębnić wybrane części ładunku (technikę tę omówię dokładniej w punkcie „Obsługa zmian między mikrousługami"), mogę skorzystać z XPATH — dobrze znanego standardu z dużą ilością dostępnych narzędzi, a nawet z selektorów CSS, które wiele osób uważa za jeszcze prostsze. Dla JSON istnieje JSONPath, ale nie ma on równie obszernego wsparcia. Uważam, że dziwne jest to, że ludzie wybierają JSON, ponieważ jest estetyczny i lekki, a potem próbują wepchnąć do niego takie pojęcia jak kontrolki hipermedialne, które już istnieją w XML. Zgadzam się jednak, że prawdopodobnie jestem tu w mniejszości i że formatem z wyboru dla wielu osób jest JSON!

Formaty binarne

O ile formaty tekstowe mają swoje zalety, na przykład łatwość ich czytania przez ludzi oraz zapewnienie wysokiego poziomu interoperacyjności z różnymi narzędziami i technologiami, to jeśli zaczniesz przejmować się rozmiarami ładunku lub wydajnością pisania i odczytywania ładunków, będziesz musiał zwrócić się w kierunku świata protokołów serializacji binarnej. Bufory protokołów istnieją już od jakiegoś czasu i często są używane poza zakresem gRPC — prawdopodobnie reprezentują najpopularniejszy format serializacji binarnej dla komunikacji opartej na mikrousługach.

Obszar formatów binarnych jest jednak rozległy — z myślą o różnych wymaganiach zostało opracowanych wiele innych formatów. Na myśl przychodzą proste kodowanie binarne (*https://oreil.ly/p8UbH*), Cap'n Proto (*https://capnproto.org*) czy FlatBuffers (*https://oreil.ly/VdqVB*). Chociaż dla każdego z tych formatów istnieje wiele testów porównawczych, które podkreślają ich istotne zalety w porównaniu z buforami protokołów, JSON lub innymi formatami, z tymi testami wiąże się podstawowy problem, ponieważ niekoniecznie prezentują one sposób, w jaki zamierzasz korzystać z określonych formatów. Jeśli chcesz pobrać kilka ostatnich bajtów z formatu serializacji lub skrócić czas na odczyt lub zapisanie ładunków danych o kilka mikrosekund, zdecydowanie sugeruję przeprowadzenie własnego porównania różnych formatów. Z moich doświadczeń wynika, że w zdecydowanej większości systemów rzadko musimy martwić się o takie optymalizacje, ponieważ często bardziej znaczące ulepszenia można osiągnąć dzięki wysyłaniu mniejszej ilości danych lub dzięki całkowitej rezygnacji z wywołania. Jeśli jednak budujesz system rozproszony o bardzo niskich opóźnieniach, pamiętaj o przygotowaniu się do zagłębienia się w świat formatów serializacji binarnych.

Schematy

Raz po raz pojawiają się dyskusje na temat tego, czy do definiowania danych udostępnianych przez punkty końcowe oraz tych, które one akceptują, należy stosować schematy. Schematy mogą być dostępne w wielu różnych rodzajach, a to, której technologii schematu można użyć, zazwyczaj określa wybrany format serializacji. Jeśli pracujesz z nieprzetworzonym formatem XML, najlepiej użyć definicji schematu XML (XSD); dla surowego JSON najlepszy będzie JSON Schema. Niektóre z opcji technologicznych, o których pisałem (w szczególności spory podzbiór opcji RPC), wymagają jawnego użycia schematów, więc jeśli wybierzesz te technologie, będziesz musiał skorzystać z powiązanych z nimi schematów. SOAP korzysta z WSDL, natomiast gRPC wymaga użycia specyfikacji bufora protokołu. W przypadku innych opcji technologicznych, które omawiałem, korzystanie ze schematów jest opcjonalne. W tych przypadkach istnieją ciekawsze opcje.

Jak już pisałem, jestem zwolennikiem korzystania z jawnych schematów dla punktów końcowych mikrousług z dwóch kluczowych powodów. Po pierwsze są one formą wyraźnej reprezentacji tego, co punkt końcowy mikrousługi udostępnia oraz co może pobierać. To ułatwia życie zarówno programistom pracującym nad mikrousługą, jak i konsumentom mikrousługi. Schematy raczej nie zastąpią dobrej dokumentacji, ale z pewnością mogą pomóc zmniejszyć ilość wymaganej dokumentacji.

Innym powodem, dla którego wolę jawne schematy, jest możliwość ich wykorzystania do wykrywania przypadkowych naruszeń kontraktu dla punktów końcowych mikrousług. Sposobami obsługi zmian w mikrousługach zajmę się za chwilę, najpierw jednak warto zbadać różne typy naruszeń oraz role, jakie mogą odgrywać schematy.

Strukturalne i semantyczne naruszenia kontraktu

Ogólnie rzecz biorąc, naruszenia kontraktów możemy podzielić na dwie kategorie — naruszenia *strukturalne* i *semantyczne*. Strukturalne naruszenie kontraktu to sytuacja, w której struktura punktu końcowego zmienia się w taki sposób, że konsument przestaje być z nim zgodny. Taką zmianą może być usuwanie pól lub metod albo dodanie nowych pól obowiązkowych. Z kolei naruszenia semantyczne odnoszą się do sytuacji, w której struktura punktu końcowego mikrousługi pozostaje niezmieniona, ale zachowanie zmienia się w sposób, który powoduje naruszenie oczekiwań konsumentów.

Weźmy prosty przykład. Masz bardzo złożoną mikrousługę `Złożone obliczenia`, która w punkcie końcowym udostępnia metodę `oblicz`. Ta metoda `oblicz` przyjmuje dwie liczby całkowite, z których obie są polami obowiązkowymi. Zmiana usługi `Złożone obliczenia` w taki sposób, że metoda `oblicz` będzie przyjmować tylko jedną liczbę całkowitą, spowoduje problemy z działaniem konsumentów usługi — będą wysyłać żądania z dwiema liczbami całkowitymi, które mikrousługa `Złożone obliczenia` odrzuci. Jest to przykład zmiany strukturalnej. Ogólnie rzecz biorąc, takie zmiany nie są zbyt trudne do zauważenia.

Zmiany semantyczne stwarzają więcej problemów. W tym przypadku struktura punktu końcowego się nie zmienia, ale zmienia się zachowanie punktu końcowego. Wracając do naszej metody `oblicz`, wyobraź sobie, że w pierwszej wersji dwie podane liczby całkowite są do siebie dodawane i zwracana jest suma. Na razie wszystko jest w porządku. Teraz modyfkujemy mikrousługę `Złożone obliczenia` w taki sposób, żeby metoda `oblicz` mnożyła przekazane liczby całkowite i zwracała wynik. Semantyka metody `oblicz` zmieniła się w sposób, który może naruszyć oczekiwania konsumentów.

Czy należy używać schematów?

Kiedy korzystamy ze schematów, to dzięki porównaniom pomiędzy ich różnymi wersjami możemy wykryć naruszenia struktury. Wykrywanie naruszeń semantycznych wymaga użycia testów. Jeśli nie korzystasz ze schematów lub jeśli ich używasz, ale zdecydowałeś się nie porównywać zmian schematu pod kątem zgodności, ciężar wykrywania naruszeń strukturalnych przed przekazaniem systemu do produkcji również spada na testowanie. Sytuacja jest trochę podobna do statycznego i dynamicznego typowania w językach programowania. W przypadku języka typowanego statycznie typy są ustalane w czasie kompilacji — jeśli kod robi z egzemplarzem typu coś, co jest niedozwolone (np. wywołanie metody, która nie istnieje), wówczas kompilator może wykryć ten błąd. Dzięki temu możesz skoncentrować wysiłki testowe na innych rodzajach problemów. Jednak w przypadku języka typowanego dynamicznie niektóre testy będą musiały wykrywać błędy, które dla języków typowanych statycznie wykrywa kompilator.

Obecnie dość swobodnie posługuję się językami zarówno statycznie, jak i dynamicznie typowanymi. Odkryłem, że jestem bardzo produktywny (relatywnie mówiąc) w posługiwaniu się obydwoma rodzajami języków. Z pewnością języki typowane dynamicznie dają pewne znaczące korzyści, które w opinii wielu osób uzasadniają rezygnację z bezpieczeństwa fazy kompilacji. Wracając na chwilę do interakcji z mikrousługami, zaznaczę, że nie odkryłem podobnie zrównoważonego kompromisu, jeśli chodzi o stosowanie schematu w porównaniu z komunikacją „bez schematu”.

Mówiąc prościej, myślę, że korzystanie z jawnego schematu z nawiązką kompensuje wszelkie postrzegane korzyści płynące z komunikacji bez schematu.

W gruncie rzeczy prawdziwym problemem nie jest to, czy korzystasz ze schematu, czy nie — chodzi raczej o to, czy ten schemat jest *jawny*. Jeśli korzystasz z danych z interfejsu API bez schematu, nadal masz oczekiwania co do tego, jakie dane powinny się tam znajdować i jaką powinny mieć strukturę. Kod, który ma obsługiwać dane, będzie napisany z pewną liczbą założeń dotyczących struktury danych. W takim przypadku twierdzę, że korzystasz ze schematu, ale jest on po prostu ukryty[2]. Uważam, że należy korzystać z jawnego schematu także dlatego, że mikrousługa powinna w maksymalnie jawny sposób ujawniać to, co robi (lub czego nie robi).

Głównym argumentem przemawiającym za korzystaniem z punktów końcowych bez schematu wydaje się być to, że schematy wymagają więcej pracy i nie dają zbyt wiele wartości. Moim skromnym zdaniem jest to częściowo brak wyobraźni, a częściowo brak dobrego narzędzia, które pomogłoby uzyskać ze schematów więcej wartości, jeśli chodzi o ich wykorzystanie do wykrywania naruszeń struktury.

Ostatecznie w dużej części schematy zapewniają wyraźną reprezentację części kontraktu struktury pomiędzy klientem a serwerem. Pomagają ujawnić tę strukturę oraz mogą znacznie pomóc w komunikacji między zespołami, a także spełniają rolę siatki bezpieczeństwa. W sytuacjach, w których koszty zmian nie są duże — na przykład gdy zarówno za klienta, jak i za serwer odpowiada ten sam zespół — jestem skłonny zgodzić się na rezygnację z korzystania ze schematów.

Obsługa zmian między mikrousługami

Jedne z najczęściej zadawanych pytań dotyczących mikrousług to: „Jak duże powinny być?”, „Jak radzisz sobie z wersjowaniem?”. Kiedy to pytanie jest zadawane, pytający rzadko mają na myśli rodzaj schematu numerowania, którego należy użyć, a raczej sposób obsługi zmian w kontraktach między mikrousługami.

Sposób obsługi zmian w gruncie rzeczy sprowadza się do dwóch zagadnień. Za chwilę przyjrzymy się, co się stanie, jeśli będziesz musiał dokonać zmiany naruszającej kontrakt. Ale zanim to zrobimy, przyjrzyjmy się, co można zrobić, aby uniknąć konieczności wprowadzania takich zmian.

Unikanie zmian naruszających kontrakt

Istnieje kilka kluczowych pojęć, którym warto się przyjrzeć w kontekście dbania o unikanie wprowadzania zmian naruszających kontrakt między mikrousługami. O wielu spośród tych pojęć wspominałem na początku tego rozdziału. Jeśli uda Ci się wprowadzić w życie niżej opisane koncepcje, niezależne modyfikowanie mikrousług stanie się o wiele łatwiejsze.

Zmiany rozszerzające

Dodawaj do interfejsu mikrousług nowe elementy; nie usuwaj starych.

Tolerancyjny konsument

Gdy korzystasz z interfejsu mikrousług, bądź elastyczny, jeśli chodzi o to, czego oczekujesz.

[2] Martin Fowler wyjaśnia to bardziej szczegółowo w kontekście przechowywania danych bez schematu (*https://oreil.ly/Ew8Jq*).

Właściwa technologia

Wybierz technologię, która ułatwia dokonywanie w interfejsie zmian zgodnych wstecz.

Jawny interfejs

Staraj się jasno określić, co udostępnia mikrousługa. Ułatwia to klientowi korzystanie z mikrousługi, a jej opiekunom ustalenie elementów, które można swobodnie zmieniać.

Wczesne wykrywanie zmian naruszających kontrakt

Stwórz mechanizmy umożliwiające wykrywanie zmian interfejsów, które spowodują naruszenie kontraktu z konsumentami, zanim wdrożysz te zmiany do produkcji.

Wymienione koncepcje wzajemnie się wzmacniają. Wiele z nich opiera się na kluczowej koncepcji ukrywania informacji, o której często wspominałem. Przyjrzyjmy się po kolei każdej z koncepcji na liście.

Zmiany rozszerzające

Prawdopodobnie najłatwiejsze jest jedynie dodawanie do kontraktu mikrousług nowych rzeczy i nieusuwanie niczego. Rozważmy przykład dodania do ładunku danych nowego pola. Przy założeniu, że klient jest w jakiś sposób tolerancyjny wobec takich zmian, nie powinno to mieć istotnego znaczenia. Na przykład nie powinno być problemu z dodaniem do rekordu klienta nowego pola `dateOfBirth`.

Tolerancyjny konsument

Wprowadzanie zmian kompatybilnych wstecz może znacznie ułatwić implementację konsumenta mikrousługi. W szczególności należy unikać zbyt ścisłego wiązania kodu klienta z interfejsem mikrousługi. Rozważmy mikrousługę `Email`, której zadaniem jest wysyłanie wiadomości e-mail do kontrahentów. Do usługi przesyłane jest żądanie o wysłanie wiadomości e-mail „zamówienie wysłane" do klienta o identyfikatorze 1234. Mikrousługa pobiera rekord klienta z tym identyfikatorem i zwraca coś w rodzaju odpowiedzi pokazanej na listingu 5.3.

Listing 5.3. Przykładowa odpowiedź z usługi Klient

```
<customer>
    <firstname>Sam</firstname>
    <lastname>Newman</lastname>
    <email>sam@magpiebrain.com</email>
    <telephoneNumber>555-1234-5678</telephoneNumber>
</customer>
```

Aby wysłać wiadomość e-mail, mikrousługa `Email` potrzebuje pól `firstname`, `lastname` i `email`. Nie musi znać pola `telephoneNumber`. Powinniśmy po prostu wyciągnąć te pola, na których nam zależy, a resztę zignorować. Niektóre technologie wiązania, zwłaszcza te używane przez języki silnie typowane, mogą próbować wiązać *wszystkie* pola, niezależnie od tego, czy konsument tego chce, czy nie. Co się stanie, jeśli zdamy sobie sprawę, że nikt nie używa pola `telephoneNumber` i zdecydujemy się je usunąć? Może to spowodować niepotrzebną awarię konsumentów.

Zastanówmy się również, co by się stało, gdybyśmy chcieli zrestrukturyzować obiekt `Customer` w taki sposób, aby obsłużył więcej szczegółów — być może poprzez dodanie nowego pola, jak na listingu 5.4? Dane, których potrzebuje usługa `Email`, nadal tam są i mają takie same nazwy, ale jeśli kod przyjmuje bardzo ścisłe założenia dotyczące tego, gdzie są przechowywane pola `firstname` i `lastname`, ponownie może dojść do jego awarii. W tym przypadku do wyodrębnienia pól, na których nam zależy, moglibyśmy skorzystać z Xpath. Dzięki temu nie musielibyśmy martwić się tym, gdzie znajdują się pola, pod warunkiem że potrafilibyśmy je znaleźć. Ten wzorzec — implementacja czytelnika zdolnego do ignorowania zmian, na których nam nie zależy — Martin Fowler nazywa tolerancyjnym konsumentem (ang. *tolerant reader*) — *https://oreil.ly/G65yf*.

Listing 5.4. Zrestrukturyzowany zasób `customer`: wszystkie dane wciąż tam są, ale czy aplikacje konsumentów będą potrafiły je znaleźć?

```
<customer>
    <naming>
        <firstname>Sam</firstname>
        <lastname>Newman</lastname>
        <nickname>Magpiebrain</nickname>
        <fullname>Sam "Magpiebrain" Newman</fullname>
    </naming>
    <email>sam@magpiebrain.com</email>
</customer>
```

Przykład klienta starającego się być w konsumowaniu usługi tak elastycznym, jak to tylko możliwe, pokazuje prawo Postela (*https://oreil.ly/GVqeI*) (znane także jako *zasada solidności*), które mówi: „Bądź konserwatywny w tym, co robisz, bądź liberalny w tym, co akceptujesz od innych”. Pierwotnym kontekstem dla tej mądrości była interakcja urządzeń w sieciach, gdzie należy spodziewać się wszelkiego rodzaju dziwnych rzeczy. W kontekście interakcji opartych na mikrousługach prowadzi to do próby ustrukturyzowania kodu klienta w taki sposób, aby był tolerancyjny na zmiany wewnątrz ładunków danych.

Właściwa technologia

Jak już pisałem, niektóre technologie mogą być bardziej kruche, jeśli chodzi o możliwość wprowadzania zmian w interfejsach. Wcześniej wspominałem o moich osobistych frustracjach związanych z Java RMI. Z drugiej strony niektóre implementacje mechanizmów integracji robią wszystko, aby można było jak najłatwiej wprowadzać zmiany bez naruszania kontraktu z klientami. Prostym rozwiązaniem jest stosowane w buforach protokołów — formacie serializacji używanym w ramach gRPC — pojęcie numeru pola. Każdy wpis w buforze protokołu musi zawierać definicję numeru pola, którego oczekuje kod klienta. Jeśli zostaną dodane nowe pola, klient nie będzie się nimi przejmował. Technologia Avro pozwala na wysyłanie wraz z ładunkiem schematu, co umożliwia klientom potencjalną interpretację ładunku w taki sposób, jakby był typem dynamicznym.

Znacznie trudniejszym rozwiązaniem jest koncepcja HATEOAS dla interfejsów REST, która w dużej części polega na umożliwieniu klientom korzystania z punktów końcowych REST nawet wtedy, gdy one się zmienią. Do tego celu używane są omówione wcześniej łącza hipermedialne. To oczywiście wymaga zastosowania wszystkich pojęć związanych z techniką HATEOAS.

Jawny interfejs

Jestem *wielkim* fanem mikrousług udostępniających jawny schemat — określający to, co robią jej punkty końcowe. Posiadanie jawnego schematu w czytelny sposób komunikuje konsumentom, czego mogą oczekiwać, ale sprawia również, że programiści pracujący nad mikrousługą mogą znacznie łatwiej zorientować się, co powinno pozostać nietknięte, aby nie dopuścić do naruszenia kontraktu z konsumentami. Innymi słowy, jawny schemat znacznie ułatwia rozpoznawanie granic ukrywania informacji — to, co jest ujawnione w schemacie, z definicji nie jest ukryte.

Istnienie jawnego schematu dla RPC jest od dawna ustalone i w gruncie rzeczy jest w przypadku wielu implementacji RPC obowiązkowe. Z kolei w technice REST koncepcja schematu jest zazwyczaj postrzegana jako opcjonalna. Pogląd ten jest do tego stopnia powszechny, że jawne schematy punktów końcowych REST są niezwykle rzadkie. Sytuacja ta zmienia się. Takie mechanizmy jak wspomniana wcześniej specyfikacja OpenAPI zyskują na popularności, a dzięki specyfikacji JSON Schema stają się również bardziej dojrzałe.

Wersjonowanie semantyczne

Czyż nie byłoby wspaniale, gdyby aplikacja kliencka mogła na podstawie numeru wersji usługi stwierdzić, czy może się z nią zintegrować? *Wersjonowanie semantyczne* (*http://semver.org*) to specyfikacja, która na to pozwala. W przypadku wersjonowania semantycznego każdy numer wersji ma postać `GŁÓWNA.POMOCNICZA.POPRAWKA`. Zwiększona wartość numeru wersji `GŁÓWNA` oznacza wprowadzenie zmian naruszających zgodność z poprzednimi wersjami. Zwiększenie wartości wersji `POMOCNICZA` oznacza dodanie nowych funkcji, które powinny być zgodne z poprzednimi wersjami. Wreszcie zmiana numeru wersji `POPRAWKA` oznacza wprowadzenie poprawek błędów do istniejących funkcjonalności.

Aby przekonać się, jak przydatne może być wersjonowanie semantyczne, przyjrzyjmy się prostemu przypadkowi użycia. Aplikacja helpdesk jest zbudowana tak, aby działała z wersją 1.2.0 usługi `Klient`. Jeśli zostanie dodana nowa funkcjonalność, powodująca zmianę usługi `Klient` do wersji 1.3.0, aplikacja helpdesk nie powinna zauważyć żadnych zmian w zachowaniu i nie należy od niej oczekiwać wprowadzania jakichkolwiek zmian. Nie moglibyśmy jednak zagwarantować, że aplikacja będzie działać z wersją 1.1.0 usługi `Klient`, ponieważ być może korzysta ona z funkcjonalności dodanych w wersji 1.2.0. Możemy również spodziewać się konieczności wprowadzania zmian w naszej aplikacji, jeśli pojawi się nowa wersja 2.0.0 usługi `Klient`.

Możesz zdecydować się na zastosowanie wersjonowania semantycznego dla usługi, a nawet dla pojedynczego punktu końcowego w usłudze, jeśli ze sobą współistnieją. Opisałem to w następnym punkcie.

Wspomniany schemat wersjonowania pozwala spakować wiele informacji i oczekiwań w zaledwie trzech polach. Pełna specyfikacja w bardzo prosty sposób określa oczekiwania klientów wobec zmian w tych liczbach i może uprościć proces komunikowania, czy zmiany powinny mieć wpływ na konsumentów. Niestety, nie zaobserwowałem, aby to podejście było wystarczająco szeroko stosowane w systemach rozproszonych, tak aby można było zdać sobie sprawę z jego skuteczności w tym kontekście. Ta sytuacja w gruncie rzeczy nie zmieniła się od pierwszego wydania tej książki.

Większe problemy występują w przypadku asynchronicznych protokołów przesyłania wiadomości. Schemat dla ładunku wiadomości można stworzyć dość łatwo — do tego celu często wykorzystywany jest system Avro. Jednak posiadanie jawnego interfejsu to znacznie więcej. W przypadku mikrousługi, która zgłasza zdarzenia, ważne jest wskazanie zdarzeń, które mikrousługa ujawnia. Obecnie prowadzonych jest kilka projektów mających na celu stworzenie jawnych schematów dla punktów końcowych opartych na zdarzeniach. Jednym z nich jest AsyncAPI (*https://www.asyncapi.com*) — mechanizm, który zdobył wielu znanych użytkowników. Jednak tym, który zyskuje najwięcej zwolenników, wydaje się być CloudEvents (*https://cloudevents.io*) — to specyfikacja wspierana przez Cloud Native Computing Foundation (CNCF). Format CloudEvents obsługuje produkt Azure Event Grid, co jest oznaką zainteresowania formatem przez różnych dostawców. Powinno to ułatwić interoperacyjność pomiędzy różnymi usługami. Jest to wciąż dość nowa przestrzeń, warto więc obserwować rozwój wydarzeń na przestrzeni najbliższych kilku lat.

Wczesne wykrywanie zmian naruszających kontrakt

Jest bardzo ważne, abyśmy jak najszybciej wychwycili zmiany, które spowodują naruszenie zgodności z konsumentami, ponieważ nawet jeśli wybierzemy najlepszą możliwą technologię, niewinna zmiana mikrousługi może spowodować, że aplikacje konsumenckie przestaną działać. Jak wspominałem, w wykrywaniu zmian strukturalnych może nam pomóc używanie schematów. Oczywiście będzie to możliwe przy założeniu, że używamy pewnego rodzaju narzędzi wspomagających proces porównywania wersji schematów. Istnieje szeroka gama narzędzi, które można zastosować do tego celu dla różnych typów schematów. Dla buforów protokołu dostępny jest system Protolock (*https://oreil.ly/wwxBx*), dla JSON Schema można skorzystać z narzędzia json-schema-diff-validator (*https://oreil.ly/COSIr*) oraz z narzędzia openapi-diff dla specyfikacji OpenAPI[3]. Co jakiś czas powstają również nowe narzędzia. System, którego szukasz, to mechanizm, który nie tylko zgłosi różnice między dwoma schematami, ale także zwróci pozytywny lub negatywny wynik testu na zgodność. Cecha ta umożliwia odrzucenie kompilacji przez system CI w przypadku znalezienia niezgodnych schematów, co zablokuje wdrożenie niezgodnej mikrousługi.

System open source Confluent Schema Registry (*https://oreil.ly/qcggd*) obsługuje formaty JSON Schema, Avro i bufory protokołu i potrafi porównywać nowo przesłane wersje pod kątem zgodności z poprzednimi wersjami. Chociaż został zbudowany jako część ekosystemu, w którym jest używany system Kafka, i potrzebuje Kafki do uruchomienia, nic nie stoi na przeszkodzie, aby używać go do przechowywania i sprawdzania poprawności używanych do komunikacji schematów niewykorzystujących systemu Kafka.

Narzędzia do porównywania schematów mogą pomóc wykryć zmiany naruszające strukturę. Co jednak zrobić z naruszeniami semantycznymi? A co wtedy, kiedy w ogóle nie korzystasz ze schematów? Wtedy należy sięgnąć po testy. Temat ten omówię bardziej szczegółowo w punkcie „Testy kontraktu oraz kontrakty konsumenckie". Chcę jednak wspomnieć o kontraktach CDC, które jawnie pomagają w tej dziedzinie. Doskonałym przykładem narzędzia ukierunkowanego

[3] Należy zwrócić uwagę, że istnieją trzy różne narzędzia o tej samej nazwie! Narzędzie openapi-diff dostępne pod adresem *https://github.com/Azure/openapi-diff* wydaje się najbliższe narzędziu, które faktycznie wykrywa zmiany w kompatybilności.

na ten problem jest Pact. Powinieneś jednak pamiętać, że jeśli nie korzystasz ze schematów, Twoje testy będą musiały być może wykonać więcej pracy, aby wykryć zmiany naruszające zgodność wstecz.

Jeśli obsługujesz wiele różnych bibliotek klienckich, możesz skorzystać z techniki polegającej na uruchamianiu testów z wykorzystaniem każdej obsługiwanej biblioteki oraz najnowszej wersji usługi. Kiedy wykryjesz zmianę, która narusza zgodność z konsumentem, masz wybór — możesz spróbować całkowicie uniknąć tego naruszenia albo zastosować ją i nawiązać komunikację z osobami odpowiedzialnymi za usługi konsumenckie.

Zarządzanie zmianami naruszającymi zgodność wstecz

Zrobiłeś wszystko, co było możliwe, aby nie wprowadzić w interfejsie mikrousługi zmian powodujących naruszenie zgodności wstecz, ale zdałeś sobie sprawę, że po prostu musisz wprowadzić zmianę, która naruszy tę zgodność. Co możesz zrobić w takiej sytuacji? Dostępne są trzy główne opcje:

Wdrażanie lockstep

Konieczność przeprowadzenia w tym samym czasie zmiany mikrousługi udostępniającej interfejs wraz ze wszystkimi jej konsumentami.

Współistnienie niezgodnych ze sobą wersji mikrousług

Korzystaj równolegle ze starej i nowej wersji mikrousługi.

Emulowanie starego interfejsu

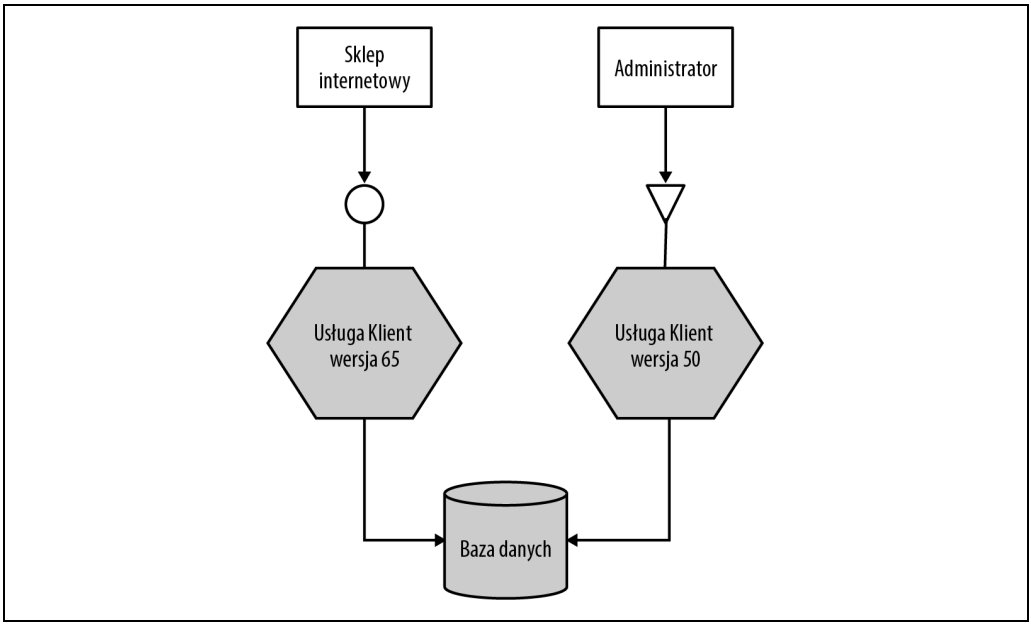
Opublikowanie mikrousługi z nowym interfejsem oraz emulowanie starego interfejsu.

Wdrażanie lockstep

Oczywiście wdrażanie lockstep stoi w sprzeczności z możliwością niezależnego wdrażania. Jeśli chcesz mieć możliwość wdrażania nowej wersji mikrousługi ze zmianą interfejsu naruszającą zgodność wstecz, ale nadal chcesz to robić w niezależny sposób, musisz dać konsumentom czas na przeprowadzenie aktualizacji do nowego interfejsu. Prowadzi to do kolejnych dwóch opcji możliwych do zastosowania.

Współistnienie niezgodnych ze sobą wersji mikrousług

Innym często wykorzystywanym rozwiązaniem wersjonowania jest wykorzystywanie w produkcji różnych wersji usługi i w przypadku starszych konsumentów kierowanie ruchu do starszej wersji usługi. Dla nowych konsumentów będzie widoczna nowa wersja, co pokazałem na rysunku 5.3. Jest to podejście stosowane w niektórych przypadkach przez Netflix w sytuacjach, w których koszt zmiany starszych konsumentów jest zbyt wysoki. Wykorzystuje się je szczególnie wtedy, gdy starsze urządzenia są nadal powiązane ze starszymi wersjami API. Osobiście nie jestem fanem tego pomysłu i rozumiem, dlaczego firma Netflix korzysta z niego rzadko. Po pierwsze, jeśli trzeba naprawić wewnętrzny błąd w usłudze, trzeba naprawić i wdrożyć dwa różne zestawy usług. Zwykle oznacza to konieczność stworzenia rozgałęzienia bazy kodu dla usługi, a to zawsze stwarza problemy.



Rysunek 5.3. Współistnienie wielu wersji tej samej usługi w celu obsługi starych wersji punktów końcowych

Po drugie oznacza to konieczność wykorzystania specjalnej logiki odpowiedzialnej za kierowanie konsumentów do właściwej mikrousługi. Takie zachowanie nieuchronnie kończy się koniecznością stosowania warstwy middleware lub zbioru skryptów `nginx`, co utrudnia wnioskowanie na temat zachowania systemu. Na koniec rozważmy stany trwałe, którymi może zarządzać usługa. Klienci utworzeni przez którąkolwiek z wersji usługi muszą być przechowywani i widoczni dla wszystkich usług, bez względu na to, która wersja została użyta do utworzenia danych. Może to wprowadzać dodatkową złożoność.

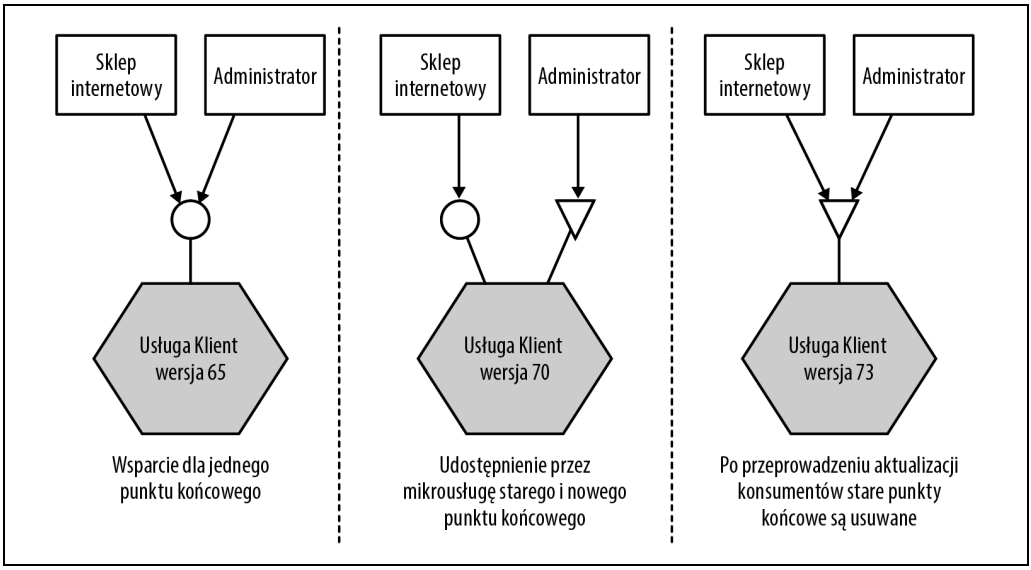
Współistnienie różnych wersji usług przez krótki czas może mieć sens, zwłaszcza gdy tworzymy coś w rodzaju wydania kanarkowego (wzorzec ten omówię dokładniej w punkcie „Na drodze do dostaw progresywnych"). W takich sytuacjach różne wersje mogą współistnieć tylko przez kilka minut lub godzin i zazwyczaj będziemy korzystać tylko z dwóch różnych wersji usługi w tym samym czasie. Im dłużej trwa uaktualnienie konsumentów do nowszej wersji i opublikowanie ich w nowych wersjach, tym bardziej należy szukać rozwiązania polegającego na współistnieniu różnych punktów końcowych tej samej mikrousługi zamiast współistnienia zupełnie różnych wersji. Nie jestem przekonany, czy dla przeciętnego projektu potrzebny wysiłek jest wart zachodu.

Emulowanie starego interfejsu

Jeśli zrobiłeś wszystko, co w Twojej mocy, aby uniknąć wprowadzenia zmiany interfejsu naruszającej zgodność wstecz, ale Ci się to nie udało, Twoim następnym zadaniem powinno być ograniczenie wpływu tej zmiany na konsumentów usługi. Nie chcemy dopuścić do sytuacji, w której konsumenci usługi będą zmuszeni do przeprowadzenia aktualizacji razem z usługą, ponieważ zawsze powinniśmy dążyć do zachowania możliwości publikowania mikrousług niezależnie od siebie.

Jednym z podejść, które z powodzeniem stosowałem, było współistnienie zarówno starego, jak i nowego interfejsu w tej samej działającej usłudze. Jeśli więc chcemy wydać zmianę naruszającą zgodność wstecz, wdrażamy nową wersję usługi, która udostępnia zarówno starą, jak i nową wersję punktu końcowego.

Takie podejście pozwala na szybkie opublikowanie nowej mikrousługi wraz z nowym interfejsem, a jednocześnie daje konsumentom czas na dokonanie przejścia. Gdy wszyscy konsumenci przestaną korzystać ze starego punktu końcowego, możemy usunąć go wraz z powiązanym z nim kodem, jak pokazałem na rysunku 5.4.



Rysunek 5.4. Mikrousługa emulująca stary punkt końcowy i udostępniająca nowy punkt końcowy niezgodny z poprzednimi wersjami

Kiedy ostatnio zastosowałem to podejście, wpakowaliśmy się w pewne kłopoty związane z liczbą obsługiwanych konsumentów oraz pewną liczbą wprowadzonych zmian, które naruszały zgodność wstecz. Z tego powodu wykorzystywaliśmy jednocześnie trzy różne wersje punktu końcowego. Nie polecam takiego podejścia! Utrzymywanie kodu wszystkich wersji oraz powiązanych z nimi testów niezbędnych do zyskania pewności, że wszystkie działają, było sporym obciążeniem. Aby ułatwić zarządzanie tym problemem, wewnętrznie przekształciliśmy wszystkie żądania do punktu końcowego V1 oraz żądania do punktu końcowego V2 na żądanie do punktu końcowego V3. Dzięki temu mogliśmy łatwo wskazać kod, który powinien być wycofany po wycofaniu starych punktów końcowych.

Jest to w gruncie rzeczy przykład wzorca rozszerzania i ograniczania, który pozwala na stopniowe wprowadzanie zmian naruszających zgodność wstecz. *Rozszerzamy* oferowane funkcjonalności, wspierając zarówno stare, jak i nowe sposoby wykonywania działań. Gdy starzy konsumenci usługi zaczynają wykonywać działania w nowy sposób, *ograniczamy* interfejs API poprzez usunięcie starej funkcjonalności.

Jeśli decydujesz się na współistnienie punktów końcowych, potrzebujesz sposobu, aby wywołujący odpowiednio kierowali swoje żądania. W przypadku systemów korzystających z protokołu HTTP widziałem zarówno rozwiązania bazujące na wykorzystaniu numerów wersji w nagłówkach żądań, jak i w samym identyfikatorze URI — na przykład */v1/customer/* lub */v2/customer/*. Mam rozterki co do tego, które podejście ma największy sens. Z jednej strony wolę, aby identyfikatory URI były niejawne, co powinno zniechęcić klientów do kodowania „na twardo" szablonów URI, ale z drugiej strony takie podejście sprawia, że wszystko staje się bardzo oczywiste, co pozwala uprościć routing żądań.

W przypadku RPC sytuacja może być nieco trudniejsza. Poradziłem sobie z tym dla buforów protokołów poprzez umieszczenie metod w różnych przestrzeniach nazw — na przykład `v1.createCustomer` i `v2.createCustomer`, ale gdy próbujesz obsługiwać różne wersje tych samych typów wysyłanych przez sieć, takie podejście może stać się bardzo kłopotliwe.

Jakie podejście preferuję?

W sytuacjach, w których ten sam zespół zarządza zarówno mikrousługą, jak i wszystkimi konsumentami, nie mam nic przeciwko ograniczonemu zastosowaniu wdrażania lockstep. Przy założeniu, że jest to rzeczywiście jednorazowa sytuacja, zdecydowanie się na to rozwiązanie w przypadku, kiedy jego wpływ jest ograniczony do jednego zespołu, może być uzasadnione. Podchodzę jednak do tego bardzo ostrożnie, ponieważ istnieje niebezpieczeństwo, że jednorazowa działalność stanie się standardową praktyką, co zagrozi możliwości niezależnego wdrażania. Jeśli z kolei będziesz korzystać z wdrożeń lockstep zbyt często, wkrótce skończysz z rozwiązaniem przypominającym rozproszony monolit.

Jak wspominałem, współistnienie różnych wersji tej samej mikrousługi może stwarzać problemy. Decydowałbym się na to rozwiązanie tylko w sytuacjach, w których istnieją plany uruchomienia dwóch wersji mikrousługi tylko przez krótki czas. Rzeczywistość jest taka, że kiedy zachodzi konieczność dania konsumentom czasu na aktualizację, trzeba się przygotować na to, że stan ten będzie trwać kilka tygodni lub jeszcze dłużej. W innych sytuacjach, w których zachodzi prawdopodobieństwo współistnienia wielu wersji mikrousług, na przykład w ramach wdrożenia niebiesko-zielonego[4] lub wydania kanarkowego, czas trwania tego stanu jest znacznie krótszy, co kompensuje wady tego podejścia.

Moją osobistą preferencją jest używanie tam, gdzie to możliwe, emulacji starych wersji punktów końcowych. Wyzwania związane z implementacją emulacji są moim zdaniem znacznie łatwiejsze do obsługi niż te, które występują w przypadku współistnienia kilku wersji mikrousługi.

Umowa społeczna

To, które podejście wybierzesz, będzie w dużej mierze wynikało z oczekiwań konsumentów co do sposobu wprowadzania zmian. Utrzymywanie starej wersji interfejsu wiąże się z kosztami. Najlepiej byłoby ten interfejs wyłączyć i usunąć powiązany z nim kod wraz z infrastrukturą tak szybko, jak to możliwe. Z drugiej strony trzeba dać konsumentom tyle czasu na zmianę, ile potrzebują. Trzeba pamiętać, że w wielu przypadkach wprowadzanie zmian niezgodnych wstecz to często

[4] Zobacz: *https://ichi.pro/pl/niebiesko-zielony-z-wdrozeniem-canary-nowatorskie-podejscie-49402727061069 — przyp. tłum.*

rzeczy, o które proszą konsumenci i (lub) które faktycznie przyniosą im korzyści. Istnieje oczywiście konieczność zachowania równowagi pomiędzy potrzebami opiekunów mikrousług a potrzebami konsumentów.

Z moich doświadczeń wynika, że w wielu przypadkach sposób, w jaki planowane zmiany zostaną obsłużone, nigdy nie był omawiany. To prowadziło do różnego rodzaju wyzwań. Podobnie jak jest w przypadku schematów, zapewnienie jakiegoś stopnia jawności co do sposobu wprowadzenia zmian niezgodnych z poprzednimi wersjami może wiele rzeczy znacząco uprościć.

Aby osiągnąć porozumienie w sprawie sposobu obsługi zmian, niekoniecznie potrzebujesz mnóstwa papieru i wielu spotkań. Zakładając jednak, że nie chcesz iść drogą wdrożeń lockstep, sugerowałbym, że zarówno właściciel, jak i konsument mikrousługi powinni mieć jasność co do kilku kwestii:

- W jaki sposób zasygnalizujesz, że interfejs musi się zmienić?
- W jaki sposób będą ze sobą współpracować zespoły konsumentów i zespoły mikrousług, aby uzgodnić sposób przeprowadzenia zmiany?
- Od kogo oczekuje się wykonania prac związanych z aktualizacją konsumentów?
- Ile czasu, po uzgodnieniu zmian, będą mieć konsumenci na przejście do nowego interfejsu, zanim zostanie on usunięty?

Pamiętaj, że jednym z sekretów skutecznej architektury mikrousług jest przyjęcie podejścia „najpierw konsument". Twoje mikrousługi istnieją po to, aby mogły być wywoływane przez innych konsumentów. Potrzeby konsumentów są najważniejsze, a jeśli wprowadzasz w mikrousługach zmiany, które spowodują problemy konsumentów w górze strumienia przetwarzania, powinieneś wziąć to pod uwagę.

Oczywiście w niektórych sytuacjach zmiana konsumentów może nie być możliwa. Słyszałem od pracowników firmy Netflix o trudnościach (przynajmniej dawniej) ze starymi dekoderami korzystającymi ze starszych wersji interfejsów API Netflix. Tych dekoderów nie można łatwo zaktualizować, więc stare punkty końcowe muszą pozostać dostępne, dopóki liczba starszych dekoderów nie spadnie do poziomu, na którym będzie można wyłączyć wsparcie dla starych wersji. Decyzje o zablokowaniu dostępu do punktów końcowych przez starych konsumentów mogą czasami mieć skutki finansowe — musisz zrównoważyć kwotę kosztów związanych z obsługą starego interfejsu w stosunku do tej, jaką zarabiasz od tych konsumentów.

Śledzenie użycia

Nawet jeśli zgadzasz się co do czasu, w którym konsumenci powinni przestać korzystać ze starego interfejsu, musisz być pewien co do tego, czy faktycznie przestali z niego korzystać. W uzyskaniu tego rodzaju danych mogą pomóc mechanizmy logowania dla każdego punktu końcowego udostępniającego mikrousługi. Podobnie pomocne może być przypisanie klientom identyfikatorów pozwalających na nawiązanie komunikacji z zespołami, aby skłonić ich do migracji ze starego interfejsu. Może to być coś tak prostego jak poproszenie konsumentów o umieszczenie identyfikatora w nagłówku `agenta użytkownika` podczas wykonywania żądań HTTP. Możesz również wymagać od nich, aby wszystkie wywołania przechodziły przez jakąś bramę API, w której klienci potrzebują kluczy w celach identyfikacji.

Środki ekstremalne

Załóżmy, że już wiesz, że konsument nadal używa starego interfejsu, który chcesz usunąć, i ociąga się z przejściem do nowej wersji. Co możesz z tym zrobić? Cóż, pierwszą rzeczą, jaką można zrobić, jest rozmowa. Być może trzeba zaoferować klientom pomoc w przeprowadzeniu zmian. Jeśli wszystko inne zawiedzie, a konsumenci nadal nie przeprowadzą migracji nawet po uzgodnieniach, możesz sięgnąć do pewnych ekstremalnych technik.

Dyskutowałem o sposobie poradzenia sobie z takim problemem z przedstawicielami jednej dużej firmy technologicznej. Wewnętrznie firma bardzo hojnie wyznaczyła okres jednego roku do czasu wycofania starych interfejsów. Zapytałem, skąd wiedzą, że konsumenci nadal korzystają ze starych interfejsów. Odpowiedziano mi, że tak naprawdę nikt nie zawraca sobie głowy śledzeniem tych informacji — po roku po prostu wyłączyli stary interfejs. Wewnętrznie uznano, że jeśli spowodowało to awarię konsumenta, była to wina zespołu konsumującego mikrousługę — mieli rok na przeprowadzenie zmiany i tego nie zrobili. Oczywiście takie podejście nie sprawdzi się w wielu przypadkach (wspominałem, że jest ekstremalne!). Prowadzi również do sporych nieefektywności. Nie wiedząc, czy stary interfejs był używany, firma zablokowała sobie możliwość usunięcia go przed upływem roku. Osobiście, nawet gdybym miał zasugerować proste wyłączenie punktu końcowego po pewnym czasie, nadal zdecydowanie wolałbym wiedzieć, na jakie klienty będzie to miało wpływ.

Inny ekstremalny środek, który obserwowałem, w gruncie rzeczy dotyczył kontekstu wycofywania bibliotek, ale teoretycznie można go było również wykorzystać w odniesieniu do punktów końcowych mikrousług. Podany przykład dotyczył starej biblioteki, którą próbowano wycofać z użytku wewnątrz organizacji na rzecz nowszej, lepszej. Pomimo dużych nakładów pracy związanej z przeniesieniem kodu do nowej biblioteki niektóre zespoły konsumentów nadal się ociągały z przejściem. Rozwiązaniem było zastosowanie opóźnień w starej bibliotece, aby wolniej reagowała na wywołania (monitorowane przez mechanizmy logowania tak, aby było widać, co się dzieje). Z biegiem czasu zespół zajmujący się wycofywaniem starej biblioteki wydłużał czas opóźnień, aż w końcu do zespołów konsumentów dotarł właściwy komunikat. Oczywiście musisz mieć pewność, że wyczerpałeś inne rozsądne środki, aby skłonić konsumentów do aktualizacji, zanim rozważysz zastosowanie czegoś takiego!

Zasada DRY i niebezpieczeństwa wielokrotnego wykorzystywania kodu w świecie mikrousług

Jednym z akronimów, którymi często posługują się programiści, jest DRY (ang. *Don't Repeat Yourself* — nie powtarzaj się). Chociaż jego definicja często jest upraszczana do dążenia, by nie powielać kodu, DRY w gruncie rzeczy oznacza, że należy unikać powielania *zachowań systemu i wiedzy*. Ogólnie rzecz biorąc, jest to bardzo rozsądna rada. Zarządzanie wieloma wierszami kodu, które robią to samo, sprawia, że baza kodu staje się większa, niż trzeba, a zatem trudniejsza do wnioskowania. Kiedy chcesz zmienić jakieś zachowanie, a to zachowanie jest powielone w wielu częściach systemu, łatwo jest zapomnieć o wszystkich miejscach, w których trzeba wprowadzić zmianę, co może prowadzić do błędów. Tak więc powtarzanie zasady DRY jak mantry ma sens.

DRY jest czymś, co prowadzi do tworzenia kodu, który można wykorzystać wielokrotnie. Wyodrębniamy powielony kod do abstrakcji, które następnie możemy wywołać z wielu miejsc. Możemy nawet zdecydować o stworzeniu współdzielonej biblioteki, z której można korzystać wszędzie! Okazuje się jednak, że udostępnianie kodu w środowisku mikrousług jest nieco bardziej złożone. Jak zawsze mamy do rozważenia więcej niż jedną opcję.

Udostępnianie kodu za pośrednictwem bibliotek

Jedną z rzeczy, których chcemy uniknąć za wszelką cenę, jest nadmierne sprzęganie mikrousługi z konsumentami w taki sposób, że każda niewielka zmiana w samej mikrousłudze może spowodować niepotrzebne zmiany u konsumenta. Czasami jednak użycie współdzielonego kodu może stworzyć takie sprzężenie. Na przykład jeden z moich klientów korzystał z biblioteki współdzielonych obiektów domeny, które reprezentowały podstawowe jednostki używane w systemie. Biblioteka ta była używana przez wszystkie nasze usługi. Kiedy jednak wprowadziliśmy zmianę w jednej z nich, musieliśmy zaktualizować wszystkie usługi. System komunikował się za pośrednictwem kolejek wiadomości, które również musiały zostać opróżnione z ich teraz *nieprawidłowej* zawartości. Gdybyś o tym zapomniał, wpadłbyś w kłopoty.

Jeśli użycie współdzielonego kodu kiedykolwiek „wycieknie" poza granice usługi, to znaczy, że wprowadziłeś potencjalną formę sprzężenia. Nie ma niczego złego w używaniu współdzielonego kodu, na przykład bibliotek logowania, ponieważ są to pojęcia wewnętrzne, które są niewidoczne dla świata zewnętrznego. Strona internetowa *realestate.com.au* korzysta ze spersonalizowanego szablonu usługi, który ma ułatwić tworzenie nowych usług. Zamiast udostępniać ten kod, firma kopiuje go do każdej nowej usługi, aby zyskać pewność, że nie dojdzie do wycieku sprzężenia.

Bardzo istotną kwestią dotyczącą udostępniania kodu za pośrednictwem bibliotek jest brak możliwości zaktualizowania wszystkich zastosowań biblioteki jednocześnie. Chociaż wiele mikrousług może korzystać z tej samej biblioteki, zwykle odbywa się to poprzez uwzględnienie instalacji tej biblioteki w procesie wdrażania mikrousługi. Aby uaktualnić używaną wersję biblioteki, należy ponownie wdrożyć mikrousługę. Dążenie do zaktualizowania wybranej biblioteki we wszystkich miejscach dokładnie w tym samym czasie może doprowadzić do konieczności powszechnego wdrożenia w tym samym czasie wielu różnych mikrousług, ze wszystkimi powiązanymi z tym kłopotami.

Jeśli więc używasz bibliotek do wielokrotnego wykorzystywania kodu w granicach mikrousług, musisz zaakceptować fakt, że w tym samym czasie może być dostępnych wiele różnych wersji tej samej biblioteki. Możesz oczywiście z czasem zaktualizować je wszystkie do najnowszej wersji, ale dopóki akceptujesz ten fakt, za wszelką cenę staraj się wielokrotnie wykorzystywać kod za pośrednictwem bibliotek. Jeśli naprawdę musisz zaktualizować kod dla wszystkich użytkowników dokładnie w tym samym czasie, powinieneś zastanowić się nad zastosowaniem mechanizmu wielokrotnego wykorzystywania kodu za pośrednictwem dedykowanej mikrousługi.

Istnieje jednak jeden konkretny przypadek użycia związany z wielokrotnym wykorzystywaniem kodu za pośrednictwem bibliotek, któremu warto przyjrzeć się bliżej.

Biblioteki klienckie

Podczas konsultacji z więcej niż jednym zespołem zdarzało mi się słyszeć zdanie, według którego tworzenie bibliotek klienckich jest istotnym elementem tworzenia usług. Przytaczano argument, że to ułatwia korzystanie z usługi i pozwala uniknąć powielania kodu wymaganego do skorzystania z usługi.

Problem oczywiście polega na tym, że jeśli ci sami ludzie utworzyli zarówno interfejs API serwera, jak i interfejs API klienta, to istnieje niebezpieczeństwo, że logika, która powinna być umieszczona na serwerze, zacznie wyciekać do klienta. Powinienem to wiedzieć: sam tak zrobiłem. Im więcej logiki wkradnie się do biblioteki klienckiej, tym gorsza spójność. W końcu zachodzi konieczność zmiany wielu klientów w celu uwzględnienia poprawek na serwerze. Powstają również ograniczenia wyboru technologii, zwłaszcza jeśli wprowadzimy wymaganie użycia biblioteki klienckiej.

Osobiście podoba mi się model bibliotek klienckich zastosowany dla usług Amazon Web Services (AWS). Wywołania usługi sieciowej na bazie SOAP lub REST mogą być wykonywane bezpośrednio, ale w praktyce użytkownicy korzystają z jednego spośród wielu dostępnych zestawów SDK, zapewniających abstrakcje nad podstawowym interfejsem API. Te pakiety SDK są jednak pisane przez członków społeczności lub pracowników firmy Amazon Web Services innych niż ci, którzy pracują nad samym interfejsem API. Wydaje się, że ten poziom separacji sprawdza się i pozwala uniknąć niektórych pułapek związanych z wykorzystaniem bibliotek klienckich. Jednym z powodów, dla których ten układ się sprawdza, jest fakt, że to klient odpowiada za moment, w którym następuje uaktualnienie. Jeśli zdecydujesz się podążać ścieżką bibliotek klienckich, upewnij się, że tak właśnie jest.

Firma Netflix w pewnych miejscach kładzie szczególny nacisk na biblioteki klienckie, ale obawiam się, że jest to odbierane wyłącznie przez pryzmat unikania powielania kodu. W rzeczywistości biblioteki klienckie używane przez firmę Netflix w równym stopniu (jeśli nie w większym) dotyczą zapewnienia niezawodności i skalowalności systemów. Biblioteki klienckie firmy Netflix obsługują wykrywanie usług, tryby awarii, rejestrowanie oraz inne funkcje, które w istocie nie mają związku z naturą samej usługi. Bez współdzielonych klientów trudno byłoby zapewnić właściwe działanie wszystkich elementów komunikacji klient – serwer na tak masową skalę, na jaką działa Netflix. Ich wykorzystanie w firmie Netflix z pewnością ułatwiło uruchomienie usługi oraz zwiększyło produktywność przy jednoczesnym zapewnieniu właściwego działania systemu. Jednak zdaniem co najmniej jednej osoby, która pracuje w firmie Netflix, z czasem doprowadziło to do powstania pewnego stopnia sprzężeń pomiędzy klientami a serwerem, które okazały się problematyczne.

Jeśli myślisz o zastosowaniu podejścia bibliotek klienckich, przede wszystkim powinieneś wydzielić kod klienta do obsługi podstawowego protokołu transportu. Ten protokół będzie odpowiedzialny za obsługę takich funkcji jak odkrywanie usług i obsługa awarii oraz będzie pozbawiony funkcji dotyczących docelowej usługi. Należy zdecydować, czy warto upierać się przy zastosowaniu samej biblioteki klienckiej oraz czy pozwolimy konsumentom korzystać z innych stosów technologii w celu kierowania wywołań do podstawowego interfejsu API. Na koniec należy zadbać o to, aby to klienty decydowały o chwili, w której należy zaktualizować biblioteki klienckie: należy zachować możliwość publikowania usług niezależnie od siebie.

Wykrywanie usług

Jeśli mamy do dyspozycji więcej niż kilka mikrousług, koniecznie chcielibyśmy się o nich dowiedzieć. Być może chcielibyśmy wiedzieć, jakie usługi działają w danym środowisku, aby wiedzieć, co należy monitorować. Być może chcemy wiedzieć, gdzie działa usługa `Konta`, aby mikrousługi, które z niej korzystają, mogły ją odszukać. Może się również zdarzyć, że chcemy ułatwić programistom pracującym w naszej firmie znalezienie dostępnych interfejsów API, tak by nie musieli wyważać otwartych drzwi. Ogólnie rzecz biorąc, wszystkie te przypadki użycia można opisać wspólnym szyldem *wykrywanie usług*. Jak zawsze w przypadku architektury mikrousług mamy kilka różnych możliwości obsługi tego mechanizmu.

Wszystkie rozwiązania, którym się przyjrzymy, obsługują dwie strony problemu. Po pierwsze zapewniają egzemplarzom usługi mechanizm rejestracji, tak by mogły powiedzieć „Jestem tutaj!". Po drugie dostarczają sposobu wyszukiwania zarejestrowanej usługi. Wykrywanie usług staje się jednak bardziej skomplikowane, jeśli rozważymy środowiska, w których stale niszczymy i wdrażamy nowe egzemplarze usług. W idealnej sytuacji chcielibyśmy, aby mogło to być obsłużone niezależnie od przyjętego rozwiązania.

Przyjrzyjmy się niektórym z najczęściej stosowanych rozwiązań wykrywania usług i rozważmy istniejące możliwości.

DNS

Jest miło, jeśli można rozpocząć od prostego rozwiązania. System DNS pozwala nam skojarzyć nazwę z adresem IP jednej maszyny lub większej liczby maszyn. Na przykład możemy zdecydować, że nasza usługa `Konta` zawsze będzie dostępna pod adresem *accounts.musiccorp.com*. Na tej podstawie moglibyśmy znaleźć adres IP hosta, na którym działa ta usługa, lub być może rozwiązać tę nazwę na mechanizm równoważenia obciążenia odpowiedzialny za dystrybucję obciążenia na wiele egzemplarzy usługi. Oznacza to, że musielibyśmy zadbać o obsługę aktualizacji tych wpisów w ramach wdrażania naszej usługi.

Podczas obsługi egzemplarzy usługi w różnych środowiskach zaobserwowałem, że dobrze się sprawdza szablon domeny bazującej na przyjętej konwencji. Na przykład możemy mieć szablon zdefiniowany jako *<nazwausługi>-<środowisko>.musiccorp.com*. W ten sposób mogą powstać wpisy w postaci *accounts-uat.musiccorp.com* lub *accounts-dev.musiccorp.com*.

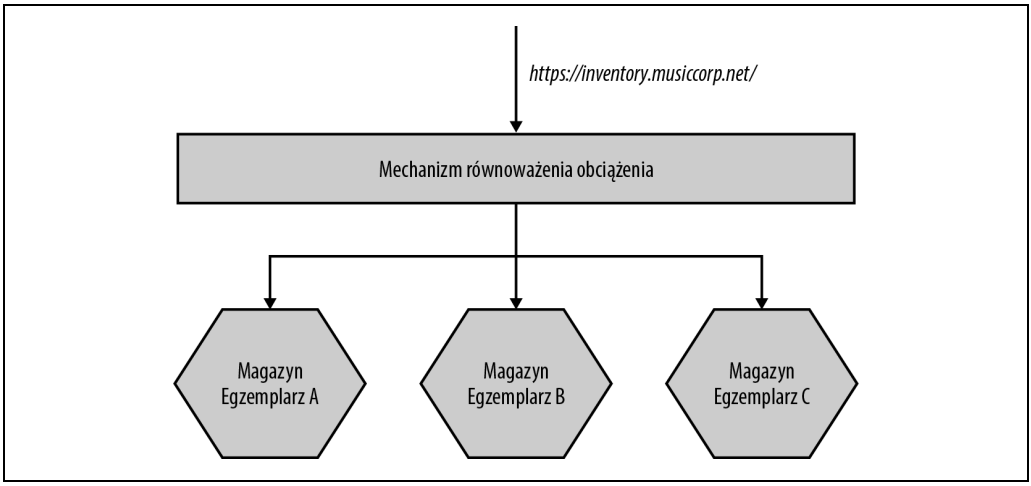
Bardziej zaawansowanym sposobem obsługi różnych środowisk jest wyznaczenie różnych serwerów nazw domen dla różnych środowisk. Można więc założyć, że pod adresem *accounts.musiccorp.com* zawsze znajduje się usługa `Konta`, ale adres ten może być przetłumaczony na różne hosty w zależności od tego, gdzie realizuję wyszukiwanie. Jeśli już rozmieściliśmy środowiska w różnych segmentach sieci i potrafimy wygodnie zarządzać własnymi serwerami DNS i wpisami, to może to być interesujące rozwiązanie. Jeśli jednak nie mamy innych korzyści z tej instalacji, to trzeba pamiętać, że jest ona czasochłonna.

Stosowanie systemu DNS ma mnóstwo zalet. Najważniejsza polega na tym, że jest to dobrze rozumiany i szeroko wykorzystywany standard, który jest wspierany przez prawie każdy stos technologii. Niestety, chociaż istnieje szereg usług zarządzania DNS wewnątrz organizacji, niewiele z nich zaprojektowano z myślą o środowiskach, w których mamy do czynienia z hostami dynamicznie

konfigurowanymi i usuwanymi na żądanie, co bardzo utrudnia wprowadzanie aktualizacji we wpisach DNS. Dość dobrym rozwiązaniem jest usługa Route53 firmy Amazon. Jak dotąd nie spotkałem jednak opcji systemu z własnym hostem (ang. *self-hosted*), która by sprawdzała się równie dobrze. W tym obszarze dobry może okazać się system Consul (który omówimy wkrótce). Oprócz problemów z uaktualnianiem wpisów DNS sama specyfikacja DNS może sprawiać pewne kłopoty.

Wpisy DNS dla nazw domen często są charakteryzowane przez parametr czasu życia (ang. *time to live* — TTL). Jest to czas, który opisuje okres, przez jaki klient może uznawać wpis za aktualny. Gdy chcemy zmienić hosta, do którego odnosi się nazwa domeny, możemy zaktualizować ten wpis, ale musimy założyć, że klienty będą utrzymywać stary adres IP *co najmniej* tak długo, jak wynika z parametru TTL. Wpisy DNS mogą być buforowane w wielu miejscach (nawet platforma JVM buforuje wpisy DNS, jeśli ta opcja nie zostanie wyłączona, a im więcej miejsc, w których wpisy są buforowane, tym bardziej nieaktualny może być wpis.

Jednym ze sposobów obejścia tego problemu jest skierowanie wpisu z nazwą domeny usługi do mechanizmu równoważenia obciążenia, który z kolei kieruje go do egzemplarza usługi, tak jak pokazano na rysunku 5.5. W przypadku instalowania nowego egzemplarza usługi można usunąć stary wpis z mechanizmu równoważenia obciążenia i dodać nowy. Niektórzy używają mechanizmu *DNS round-robin*, którego działanie polega na tym, że wpisy DNS odnoszą się do grupy maszyn. Technika ta sprawia sporo problemów, ponieważ klient jest ukryty przed swoim hostem i dlatego w przypadku problemów nie można łatwo zatrzymać routingu ruchu do jednego z takich hostów.



Rysunek 5.5. Wykorzystanie systemu DNS do przetłumaczenia nazwy domeny na mechanizm równoważenia obciążenia w celu uniknięcia utrzymywania przestarzałych wpisów

Jak już wspomniano, system DNS jest dobrze rozumiany i powszechnie obsługiwany. Jednak ma również kilka wad. Przed wybraniem bardziej złożonego systemu warto zastanowić się nad tym, czy DNS gwarantuje wystarczająco dobre rozwiązanie. W systemach, w których działają tylko pojedyncze węzły, zastosowanie DNS w celu odwołania się bezpośrednio do hostów prawdopodobnie będzie dobrym rozwiązaniem. Jednak w tych przypadkach, w których jest potrzebny więcej niż jeden egzemplarz hosta, warto zastosować tłumaczenie wpisów DNS na mechanizm równoważenia obciążenia zdolny do umieszczania hostów w usłudze i usuwania ich z niej według potrzeb.

Dynamiczne rejestry usług

Wady systemu DNS jako sposobu na znalezienie węzłów w bardzo dynamicznych środowiskach doprowadziły do powstania wielu systemów alternatywnych. Większość z nich obejmuje mechanizm rejestracji usługi w jakimś centralnym rejestrze, w którym można później wyszukać tę usługę. Często systemy te robią więcej niż tylko dostarczanie usług rejestracji i wykrywania. Może to być korzystne w pewnych sytuacjach. Istnieje wiele narzędzi zapewniających takie funkcje, dlatego omówimy tylko kilka opcji, aby dać Czytelnikom obraz tego, co jest dostępne.

Zookeeper

Zookeeper (*http://zookeeper.apache.org/*) pierwotnie został opracowany w ramach projektu Hadoop. System jest wykorzystywany dla oszałamiająco szerokiego zbioru przypadków użycia. Realizuje takie zadania jak zarządzanie konfiguracją, synchronizowanie danych między usługami, wybór lidera, kolejki komunikatów i (co jest przydatne dla nas) pełni funkcję usługi nazw.

Zookeeper, tak jak wiele podobnych rodzajów systemów, bazuje na uruchamianiu pewnej liczby węzłów w klastrze w celu zapewnienia różnych gwarancji. Oznacza to, że należy spodziewać się konieczności uruchomienia co najmniej trzech węzłów systemu Zookeeper. Większa część logiki systemu Zookeeper sprowadza się do zapewnienia bezpiecznej replikacji danych pomiędzy tymi węzłami oraz zapewnienia spójności danych podczas awarii węzłów.

Centralnym komponentem systemu Zookeeper jest hierarchiczna przestrzeń nazw do przechowywania informacji. Klienty mogą wstawiać nowe węzły w tej hierarchii, zmieniać je lub odpytywać. Ponadto mogą obserwować węzły tak, aby uzyskać informację, jeśli węzeł się zmieni. Oznacza to, że klient może zapisać informację o tym, gdzie w tej strukturze są nasze usługi, i zostanie powiadomiony, gdy dane te się zmienią. Zookeeper jest często używany w roli ogólnego magazynu konfiguracji. Możemy więc wykorzystać go do przechowywania konfiguracji poszczególnych usług. Dzięki temu możemy wykonywać takie zadania jak dynamiczne zmiany poziomu rejestrowania lub wyłączać funkcje działającego systemu.

W rzeczywistości istnieją lepsze rozwiązania dla dynamicznej rejestracji usług, do tego stopnia, że obecnie aktywnie unikałbym systemu Zookeeper w tym przypadku użycia.

Consul

Podobnie jak Zookeeper, system Consul (*https://www.consul.io/*) obsługuje zarówno zarządzanie konfiguracją, jak i wykrywanie usług. W systemie Consul zastosowano jednak inne podejście niż w systemie Zookeeper, polegające na zapewnieniu większego wsparcia dla kluczowych przypadków użycia. System Consul udostępnia na przykład interfejs HTTP do wykrywania usług, a jedną z najciekawszych funkcji tego systemu jest wbudowany serwer DNS. W szczególności pozwala on na serwowanie rekordów SRV, które pozwalają ustalić na podstawie przekazanej nazwy zarówno adres IP, jak i port. Oznacza to, że jeśli w części systemu używamy DNS i potrafimy obsługiwać rekordy SRV, możemy po prostu włączyć system Consul i zacząć go używać bez wprowadzania żadnych zmian w istniejącym systemie.

Consul zawiera również wbudowane inne funkcje, które mogą być użyteczne — na przykład zdolność wykonywania testów kondycji węzłów. Oznacza to, że system Consul może dobrze pokrywać możliwości oferowane przez inne dedykowane narzędzia do monitorowania, chociaż częściej system Consul może pełnić funkcję źródła informacji, które można później wykorzystać w bardziej wszechstronnych systemach monitorowania.

System Consul wykorzystuje interfejs HTTP REST do wykonywania wielu zadań — począwszy od rejestracji usługi, odpytywania magazynu klucz-wartość, a skończywszy na kontroli kondycji. Dzięki temu integracja systemu Consul z różnymi stosami technologii staje się bardzo prosta. Z systemem Consul jest również związany zestaw narzędzi, które dobrze z nim współpracują, dodatkowo poprawiając jego przydatność. Jednym z przykładów jest *consul-template* (*https://oreil.ly/llwVQ*) — narzędzie zapewniające aktualizację plików tekstowych na podstawie wpisów w systemie Consul. Na pierwszy rzut oka nie wydaje się to zbyt interesujące, dopóki nie weźmiesz pod uwagę faktu, że za pomocą programu *consul-template* możesz teraz zmienić wartości w Consul — być może lokalizację mikrousługi lub wartość konfiguracyjną — i dynamicznie aktualizować pliki konfiguracyjne w całym systemie. Nagle każdy program, który odczytuje swoją konfigurację z pliku tekstowego, może dynamicznie aktualizować swoje pliki tekstowe bez konieczności wiedzy o samym systemie Consul. Doskonałym przypadkiem użycia byłoby dynamiczne dodawanie węzłów do puli modułu równoważenia obciążenia lub usuwanie ich stamtąd za pomocą programowego modułu równoważenia obciążenia, takiego jak HAProxy.

Innym narzędziem, które dobrze integruje się z Consul, jest Vault — program do zarządzania informacjami poufnymi, do którego wrócę w punkcie „Sekrety". Zarządzanie informacjami poufnymi (tzw. sekretami) może stwarzać wiele problemów, ale połączenie systemów Consul i Vault z pewnością może wiele ułatwić.

etcd i Kubernetes

Jeśli korzystasz z platformy do zarządzania kontenerami, prawdopodobnie masz już zapewniony mechanizm odkrywania usług. Kubernetes nie różni się pod tym względem od pozostałych i korzysta częściowo z mechanizmu etcd (*https://etcd.io*) — wykorzystywanego w pakiecie z Kubernetes magazynu zarządzania konfiguracją. System etcd ma możliwości podobne do systemu Consul, a Kubernetes używa go do zarządzania szeroką gamą informacji konfiguracyjnych.

Kubernetes omówię bardziej szczegółowo w punkcie „Kubernetes i orkiestracja kontenerów". W skrócie: odkrywanie usług w Kubernetes polega na wdrożeniu kontenera wewnątrz poda. Następnie usługa dynamicznie identyfikuje te pody, które powinny stać się częścią usługi przez dopasowywanie wzorców do metadanych skojarzonych z określonym podem. Jest to dość elegancki mechanizm, który może oferować spore możliwości. Po wdrożeniu kontenera wewnątrz poda żądania do usługi są kierowane do jednego z podów tworzących tę usługę.

Szeroki zakres możliwości, które otrzymujesz z Kubernetes „po wyjęciu z pudełka", może skłaniać do wykorzystywania tego, co jest dostarczane wraz z podstawową platformą, i unikania dedykowanych narzędzi, takich jak Consul. W wielu przypadkach ma to sens, zwłaszcza jeśli nie jesteś zainteresowany szerszym ekosystemem narzędzi związanych z systemem Consul. Jeśli jednak korzystasz

ze środowiska mieszanego, w którym pewne usługi działają na platformie Kubernetes, a inne w innych miejscach, warto zainteresować się dedykowanym narzędziem do wykrywania usług, które może być używane na obu platformach.

Tworzenie własnych rozwiązań

Jednym z możliwych sposobów podejścia, z którego sam korzystałem, a także obserwowałem używanie go w innych miejscach, było stworzenie własnego systemu. W jednym z projektów intensywnie korzystaliśmy z usługi AWS, która oferuje możliwość dodawania tagów do egzemplarzy usług. Podczas uruchamiania egzemplarzy usług stosowałem tagi, aby pomóc zdefiniować egzemplarz oraz ułatwić ustalenie, do czego był używany. Pozwoliło to na przykład na powiązanie bogatego zbioru metadanych z określonym hostem:

- usługa = konta;
- środowisko = produkcja;
- wersja = 154.

Następnie można było skorzystać z interfejsów API usługi AWS w celu odpytania wszystkich egzemplarzy powiązanych z określonym kontem AWS, aby znaleźć maszyny, które mnie interesowały. W takiej konfiguracji sama usługa AWS odpowiada za przechowywanie metadanych skojarzonych z każdym egzemplarzem i zapewnia możliwość odpytywania o te dane. Następnie stworzyłem narzędzia wiersza polecenia do interakcji z tymi wystąpieniami oraz narzędzia GUI do monitorowania stanu. Było to dość łatwe, zwłaszcza jeśli każdy egzemplarz usługi udostępnia programowo informacje dotyczące interfejsów usług.

Ostatnim razem, gdy tworzyłem takie rozwiązanie, nie posunęliśmy się do tego, by usługi same korzystały z interfejsu API usługi AWS do wyszukiwania swoich zależności, ale nie ma żadnego powodu, dla którego nie można by tego zrobić. Oczywiście, jeśli chcemy, aby usługi wyższej warstwy uzyskały informację o zmianie lokalizacji usług warstwy niższej, jesteśmy zdani na samych siebie.

W dzisiejszych czasach nie jest to droga, którą bym poszedł. Dostępne narzędzia w tej przestrzeni są na tyle dojrzałe, że byłby to przypadek nie tylko ponownego wynalezienia koła, ale odtwarzania znacznie gorszego koła.

Nie zapomnij o ludziach

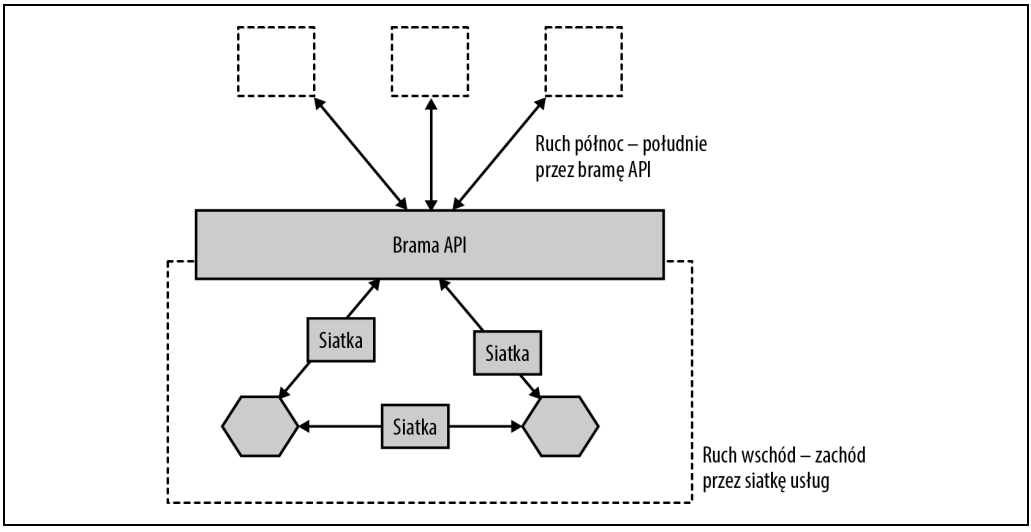
Systemy, które dotychczas omawialiśmy, umożliwiały łatwą rejestrację egzemplarza usługi, by później wyszukiwać inne usługi, z którymi trzeba się skomunikować. Jednak czasami ludzie także potrzebują tych informacji. Niezależnie od wybranego systemu należy zadbać o zebranie narzędzi, które umożliwiają czytanie ich przez ludzi. Można to zrobić na przykład za pośrednictwem interfejsu API, który pozwala na pobieranie tych szczegółów do rejestrów czytanych przez ludzi (tematem tym zajmę się za chwilę).

Siatki usług i bramy interfejsów API

Istnieje niewiele związanych z mikrousługami obszarów technologii, którym poświęcono by tyle uwagi i wokół których byłoby tyle szumu i zamieszania, ile jest w przypadku siatek usług i bram API. Oba rozwiązania mają swoje miejsce, ale — co może być mylące — niektóre spełniane przez nie obowiązki mogą się na siebie nakładać. W szczególności bramy API są podatne na niewłaściwe użycie (oraz niewłaściwą sprzedaż), dlatego ważne jest, by zrozumieć, w jaki sposób tego typu technologie mogą pasować do architektury mikrousług. Zamiast próby przedstawienia szczegółowego obrazu tego, co można zrobić z tymi produktami, chcę przedstawić przegląd tego, gdzie pasują i jak mogą pomóc, oraz zaprezentować kilka pułapek, których należy unikać.

W typowym języku związanym z centrami danych mówimy o ruchu „wschód – zachód" jako o ruchu wewnątrz centrum danych oraz o ruchu „północ – południe" związanym z interakcjami centrum danych ze światem zewnętrznym. Z perspektywy sieci to, czym jest centrum danych, stało się koncepcją nieco rozmytą, więc dla naszych celów porozmawiamy szerzej o sieciowych granicach. Może to odnosić się do całego centrum danych, klastra Kubernetes lub po prostu koncepcji wirtualnej sieci, na przykład grupy maszyn działających w tej samej wirtualnej sieci LAN.

Ogólnie rzecz biorąc, brama API znajduje się na granicy systemu i obsługuje ruch północ – południe. Jej głównym problemem jest zarządzanie dostępem do wewnętrznych mikrousług ze świata zewnętrznego. Z drugiej strony siatka usług zajmuje się, bardzo wąsko, komunikacją między mikrousługami w granicach mikrousługi, tzn. — jak pokazuje rysunek 5.6 — ruchem wschód – zachód.



Rysunek 5.6. Ogólny obraz miejsc, w których są używane bramy API i siatki usług

Siatki usług i bramy API mogą potencjalnie umożliwiać mikrousługom udostępnianie kodu bez konieczności tworzenia nowych bibliotek klienckich lub nowych mikrousług. W dużym uproszczeniu siatki usług i bramy API mogą działać jako serwery proxy między mikrousługami. Może to oznaczać, że mogą być używane do implementacji pewnych zachowań, na przykład wykrywania usług lub logowania, które w przeciwnym razie musiałyby być zrealizowane w kodzie.

Jeśli używasz bramy API lub siatki usług do implementacji wspólnego zachowania dla mikrousług, ważne jest, aby to zachowanie było całkowicie generyczne — innymi słowy, aby zachowanie na serwerze proxy nie miało związku z żadnym konkretnym zachowaniem pojedynczej mikrousługi.

Po wyjaśnieniu tego problemu muszę również wyjaśnić, że świat nie zawsze jest tak jednoznaczny. Wiele bram API próbuje również zapewnić możliwości dla ruchu wschód – zachód, ale jest to temat, którym zajmę się wkrótce. Najpierw spójrzmy na bramy API oraz zadania, które można realizować za ich pomocą.

Bramy API

Bramy API są skoncentrowane przede wszystkim na ruchu północ – południe, a głównym rozwiązywanym przez nie problemem w środowisku mikrousług jest mapowanie żądań od podmiotów zewnętrznych do wewnętrznych mikrousług. Odpowiedzialność ta jest podobna do tego, co można osiągnąć za pomocą prostego serwera proxy HTTP. W gruncie rzeczy bramy API zazwyczaj budują dodatkowe funkcjonalności na istniejących produktach proxy HTTP i w dużej mierze działają jako odwrotne serwery proxy. Ponadto bramy API mogą być używane do implementowania takich mechanizmów jak klucze API dla podmiotów zewnętrznych, logowanie, ograniczanie pasma i tym podobne. Niektóre produkty bram API udostępniają również portale dla programistów, często ukierunkowane na konsumentów zewnętrznych.

Część zamieszania wokół bram API ma podłoże historyczne. Jakiś czas temu istniało ogromne zainteresowanie tym, co nazywano „ekonomią API". Branża zaczęła rozumieć możliwości wynikające z oferowania interfejsów API do rozwiązań zarządzanych — od produktów SaaS, takich jak Salesforce, po platformy takie jak AWS — ponieważ stało się jasne, że interfejs API zapewnia klientom znacznie większą elastyczność w sposobie wykorzystania ich oprogramowania. Z tego powodu wiele osób zaczęło przyglądać się wykorzystywanemu przez siebie oprogramowaniu i rozważać korzyści płynące z ujawnienia tych funkcjonalności swoim klientom nie tylko za pośrednictwem GUI, ale także za pośrednictwem API. Liczono na to, że takie podejście otworzy większe możliwości rynkowe, a w związku z tym pozwoli zarobić więcej pieniędzy. W obliczu tego zainteresowania pojawiła się grupa produktów — implementacji bram API, które pomogły w osiągnięciu tych celów. Ich zestaw funkcji opierał się w dużej mierze na zarządzaniu kluczami API dla podmiotów zewnętrznych, egzekwowaniu limitów i śledzeniu wykorzystania do celów tzw. obciążeń zwrotnych (ang. *chargeback*). Rzeczywistość jest taka, że chociaż interfejsy API absolutnie okazały się doskonałym sposobem dostarczania usług niektórym klientom, rozmiar wpływu API na ekonomię nie był tak duży, jak niektórzy się spodziewali, a wiele firm odkryło, że kupiło produkty bram API dostarczające funkcjonalności, których nigdy nie potrzebowały.

W większości przypadków brama API jest faktycznie używana do zarządzania dostępem do mikrousług organizacji z poziomu własnych klientów GUI (stron internetowych, natywnych aplikacji mobilnych) za pośrednictwem publicznego internetu. Nie ma tutaj „podmiotu zewnętrznego". Potrzeba istnienia jakiejś formy bramy API dla Kubernetes jest kluczowa, ponieważ Kubernetes natywnie obsługuje sieć tylko w obrębie klastra i nie zajmuje się obsługą komunikacji do klastra i z klastra. Jednak w takim przypadku brama API zaprojektowana do zewnętrznego dostępu stron trzecich jest ogromną przesadą.

Jeśli więc chcesz zastosować bramę API, postaraj się w czytelny sposób określić, czego od niej oczekujesz. Właściwie poszedłbym nieco dalej i powiedział, że prawdopodobnie powinieneś unikać korzystania z bramy API, która robi zbyt wiele. Tym tematem zajmę się w kolejnym podpunkcie.

Gdzie używać bram API?

Gdy zdasz sobie sprawę z istniejących przypadków użycia, nieco łatwiej będzie Ci zobaczyć, jakiego typu bramy są Ci potrzebne. Jeśli celem jest wyłącznie udostępnienie mikrousług działających na platformie Kubernetes, możesz skorzystać z własnych odwróconych serwerów proxy lub jeszcze lepiej: możesz zastosować dedykowany produkt, taki jak Ambassador, który został zbudowany od podstaw z myślą o tym przypadku użycia. Jeśli naprawdę musisz zarządzać dużą liczbą użytkowników zewnętrznych uzyskujących dostęp do Twojego interfejsu API, prawdopodobnie istnieją inne produkty, na które należy zwrócić uwagę. W rzeczywistości możliwe jest, że będziesz zmuszony skorzystać z więcej niż jednej bramy, aby lepiej obsłużyć podział obowiązków, i zdaję sobie sprawę, że jest to rozsądne w wielu sytuacjach, chociaż nadal mają tu zastosowanie zwykłe zastrzeżenia dotyczące zwiększania ogólnej złożoności systemu i zwiększania liczby sieciowych przeskoków.

Kilkakrotnie byłem zaangażowany w bezpośrednią współpracę z dostawcami, aby pomóc im w wyborze narzędzi. Mogę bez wahania powiedzieć, że w przestrzeni bram API doświadczyłem więcej błędnych i złych lub nieodpowiednich zachowań niż w jakimkolwiek innym miejscu. Z tego powodu w tym rozdziale nie zamieściłem odwołań do produktów niektórych dostawców. Wiele z tych doświadczeń dotyczyło start-upów wspieranych przez inwestycje *venture capital*, które zbudowały produkt w czasach boomu pojęcia ekonomii API tylko po to, aby odkryć, że rynek nie istnieje, a więc walczą na dwóch frontach: po pierwsze o niewielką liczbę użytkowników, którzy faktycznie potrzebują tego, co oferują bardziej złożone bramy, a po drugie tracą biznes na rzecz bardziej skoncentrowanych produktów bram API, które zostały zbudowane z myślą o zdecydowanej większości prostszych potrzeb.

Czego unikać?

Częściowo z powodu pewnej desperacji niektórych dostawców bram API składano bardzo różne deklaracje dotyczące tego, co te produkty mogą robić. Doprowadziło to do wielu niewłaściwych zastosowań tych produktów, a w konsekwencji do niefortunnej nieufności wobec tego, co zasadniczo jest dość prostą koncepcją. Dwa kluczowe przykłady niewłaściwego wykorzystania bram API, z którymi się spotkałem, to agregacja wywołań i przepisywanie protokołów. Spotkałem się jednak również z większym naciskiem na używanie bram API na granicy mikrousług (dla ruchu wschód – zachód).

W tym rozdziale przyglądaliśmy się już pokrótce przydatności takich protokołów jak GraphQL, który może nam pomóc w sytuacji, gdy trzeba wykonać kilka wywołań, a następnie zagregować i odfiltrować wyniki. Często można jednak spotkać się z dążeniami do rozwiązywania tego problemu również w warstwie bram API. Zaczyna się dość niewinnie: łączysz kilka wywołań, które zwracają pojedynczy ładunek danych. Następnie w ramach tego samego zagregowanego przepływu zaczynasz wykonywanie kolejnego wywołania usługi w dole strumienia. Później odczuwasz potrzebę wprowadzenia dodatkowej logiki warunkowej i wkrótce zdajesz sobie sprawę, że wbudowałeś implementację podstawowych procesów biznesowych w narzędziu zewnętrznego podmiotu, które nie jest przystosowane do tego zadania.

Jeśli odkryjesz, że musisz skorzystać z agregacji wywołań i mechanizmów filtrowania, wykorzystaj potencjał GraphQL lub wzorca BFF, który omówię w rozdziale 14. Jeśli agregacja wywołań, którą wykonujesz, jest w gruncie rzeczy procesem biznesowym, lepiej zaimplementować go za pomocą jawnie zamodelowanej sagi, którą zajmę się w rozdziale 6.

Oprócz zadań agregacji wywołań często wskazuje się przepisywanie protokołów jako zadanie, do którego powinny być używane bramy API. Pamiętam, jak pewien dostawca, którego nazwy tu nie wymienię, bardzo intensywnie promował ideę, że jego produkt może „zmienić dowolny interfejs API SOAP w interfejs API REST". Po pierwsze REST to cały sposób myślenia architektonicznego, którego nie można po prostu zaimplementować w warstwie proxy. Po drugie przepisywanie protokołów, co zasadniczo próbuje się robić, nie powinno odbywać się w warstwach pośrednich, ponieważ powoduje to umieszczanie wielu zachowań w niewłaściwym miejscu.

Głównym problemem dotyczącym zarówno funkcjonalności przepisywania protokołu, jak i implementacji agregacji wywołań wewnątrz bram API jest naruszenie zasady, według której potoki powinny być nieinteligentne, a punkty końcowe inteligentne. „Inteligencja" w naszym systemie powinna być zaimplementowana w naszym kodzie, nad którym możemy mieć pełną kontrolę. Brama API w tym przykładzie jest potokiem — chcemy, aby była jak najprostsza. Dzięki zastosowaniu mikrousług dążymy do modelu, w którym dzięki niezależnemu wdrażaniu możemy łatwiej wprowadzać zmiany i je publikować. Pomaga w tym utrzymywanie inteligencji wewnątrz mikrousług. Jeśli wprowadzanie zmian będzie konieczne również w warstwach pośrednich, sprawy nieco się skomplikują. Biorąc pod uwagę kluczowe znaczenie bram API, wprowadzane w nich zmiany powinny być ściśle kontrolowane. Wydaje się mało prawdopodobne, aby poszczególne zespoły otrzymały wolną rękę do dokonywania samoobsługowych zmian w tych często centralnie zarządzanych usługach. Co to oznacza? Stosowanie systemu ticketów. Aby wprowadzić zmiany w oprogramowaniu, zespół bramy interfejsu API wprowadza zmiany za Ciebie. Im więcej zachowań wycieka do bram API (lub do korporacyjnych magistral usług), tym większe ryzyko konieczności przekazywania pracy, zwiększonych nakładów na koordynację i spowolnione dostawy.

Ostatnim problemem jest użycie bramy API jako pośrednika dla wszystkich wywołań wewnątrz mikrousługi. Może to stwarzać olbrzymie problemy. Wstawienie bramy API lub zwykłego sieciowego serwera proxy między dwiema mikrousługami zwykle oznacza dodanie co najmniej jednego przeskoku sieciowego. Wywołanie z mikrousługi A do mikrousługi B najpierw przechodzi z A do bramy API, a następnie z bramy API do B. Trzeba wziąć pod uwagę wpływ opóźnień związanych z dodatkowym wywołaniem sieciowym oraz dodatkowe nakłady związane z działaniem serwera proxy. Znacznie lepiej przygotowane do rozwiązania tego problemu są siatki usług, które omówię w następnym punkcie.

Siatki usług

W przypadku siatki usług typowe funkcje związane z komunikacją wewnątrz mikrousługi są przekazane do siatki. Zmniejsza to zakres funkcjonalności, którą trzeba zaimplementować wewnątrz mikrousługi, co jednocześnie zapewnia spójność w sposobie wykonywania określonych czynności.

Typowe funkcje implementowane przez siatki usług obejmują wzajemne zabezpieczenia TLS, identyfikatory korelacji, odkrywanie usług, równoważenie obciążenia oraz wiele innych. Często ten typ funkcjonalności jest dość generyczny, więc ostatecznie prowadzi do wykorzystania do jej obsługi wspólnej biblioteki dla wielu mikrousług. Potem jednak trzeba poradzić sobie z tym, co się stanie, jeśli różne mikrousługi będą miały uruchomione różne wersje bibliotek, lub co się stanie, jeśli mamy kilka mikrousług napisanych z wykorzystaniem różnych środowisk wykonawczych.

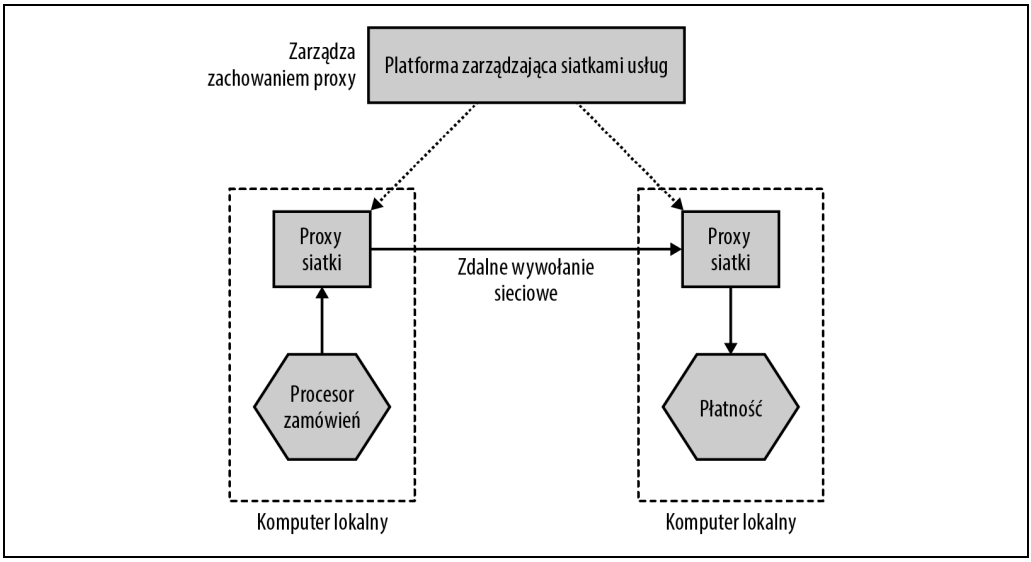
Dawniej w firmie Netflix obowiązywała zasada, zgodnie z którą cała nielokalna komunikacja sieciowa musiała być wykonywana w trybie JVM do JVM. Miało to na celu zadbanie o to, aby były wykorzystywane wypróbowane i przetestowane biblioteki, które są istotną częścią zarządzania efektywną komunikacją między mikrousługami. Zastosowanie siatki usług pozwala jednak na wielokrotne wykorzystanie wspólnych funkcji w mikrousługach napisanych w różnych językach programowania. Siatki usług mogą być również niezwykle przydatne w implementacji standardowego zachowania w mikrousługach tworzonych przez różne zespoły. Korzystanie z siatki usług, szczególnie na platformie Kubernetes, staje się coraz częściej zakładaną częścią każdej platformy, którą można utworzyć do samoobsługowego wdrażania i zarządzania mikrousługami.

Jedną z ważniejszych zalet siatki usług jest ułatwienie implementacji typowych zachowań w mikrousługach. Gdyby ta wspólna funkcjonalność została zaimplementowana wyłącznie za pośrednictwem bibliotek współdzielonych, zmiana zachowania wymagałaby pobrania nowej wersji wspomnianych bibliotek dla każdej mikrousługi i jej wdrożenia przed opublikowaniem zmiany. Dzięki siatce usług zyskujemy znacznie większą elastyczność wdrażania zmian dotyczących komunikacji wewnątrz mikrousług bez konieczności przebudowy i ponownego wdrażania.

Jak działają siatki usług?

Ogólnie rzecz biorąc, w architekturze mikrousług możemy się spodziewać mniejszego ruchu północ – południe niż wschód – zachód. Pojedyncze wywołanie północ – południe, na przykład złożenie zamówienia, może spowodować wiele połączeń wschód – zachód. Oznacza to, że gdy rozważasz zastosowanie jakiegokolwiek rodzaju serwera proxy dla wywołań wewnątrz mikrousługi, musisz zdawać sobie sprawę z narzutu, jaki mogą powodować te dodatkowe wywołania. Jest to podstawowa kwestia brana pod uwagę przy budowaniu siatek usług.

Siatki usług mają różne kształty i rozmiary, ale łączy je to, że ich architektura opiera się na próbie ograniczenia wpływu wywołań do i z serwera proxy. Osiąga się to przede wszystkim poprzez dystrybucję procesów serwera proxy do uruchamiania na tych samych fizycznych komputerach co wystąpienia mikrousług, a to zapewnia ograniczoną liczbę zdalnych wywołań sieciowych. Schemat takiego działania przedstawiono na rysunku 5.7 — usługa `Procesor zamówień` wysyła żądanie do mikrousługi `Płatność`. To wywołanie jest najpierw kierowane lokalnie do egzemplarza proxy działającego na tym samym komputerze co `Procesor zamówień`, a następnie jest kierowane do mikrousługi `Płatność` za pośrednictwem lokalnego egzemplarza proxy. Z perspektywy usługi `Procesor zamówień` wszystko wygląda tak, jakby było wykonywane zwykłe połączenie sieciowe — jest nieświadoma, że połączenie jest kierowane lokalnie na maszynie, co jest znacznie szybsze (a także mniej podatne na podziały).



Rysunek 5.7. Siatka usług służy do obsługi bezpośredniej komunikacji między mikrousługami

Na szczycie lokalnych proxy siatek działa platforma zarządzająca, i to zarówno jako miejsce, w którym można zmienić zachowanie proxy, jak i miejsce, w którym można zbierać informacje o tym, co robią poszczególne egzemplarze proxy.

W przypadku wdrażania na platformie Kubernetes każdy egzemplarz mikrousług należy wdrożyć wewnątrz poda wraz z własnym, lokalnym proxy. Pojedynczy pod zawsze jest wdrażany jako pojedyncza jednostka, dzięki czemu zawsze wiemy, że proxy jest dostępne. Co więcej, awaria pojedynczego proxy ma wpływ tylko na ten jeden pod. Taka konfiguracja umożliwia również skonfigurowanie każdego proxy inaczej, do różnych celów. Bardziej szczegółowo omówię te pojęcia w punkcie „Kubernetes i orkiestracja kontenerów".

Wiele implementacji siatek usług jako podstawy lokalnie uruchomionych procesów używa proxy Envoy (*https://www.envoyproxy.io*). Envoy to lekki, napisany w C++ proxy często używany jako blok konstrukcyjny dla siatek usług i innych typów oprogramowania opartego na proxy. Jest to bardzo ważny blok konstrukcyjny na przykład dla systemów Istio i Ambassador.

Egzemplarze proxy są z kolei zarządzane przez platformę zarządzającą (ang. *control plane*). To oprogramowanie, które ułatwia obserwację tego, co się dzieje, i pozwala kontrolować wykonywane operacje. Na przykład w przypadku zastosowania siatki usług do implementacji wzajemnego protokołu TLS platforma zarządzająca jest wykorzystywana do dystrybucji certyfikatów klienta i serwera.

Czy siatki usług nie są inteligentnymi potokami?

Całe to omówienie wypychania wspólnych zachowań do siatek usług mogło spowodować, że w głowach niektórych z Was zaczęły dzwonić dzwonki alarmowe. Czy takie podejście nie jest otwarte na tego samego rodzaju problemy, na jakie są narażone korporacyjne magistrale usług lub nadmiernie rozdęte bramy API? Czy nie grozi nam wciśnięcie zbyt wiele „inteligencji" do siatki usług?

Kluczowe do zapamiętania jest tutaj to, że wspólne zachowanie, które implementujemy w siatce usług, nie jest specyficzne dla żadnej mikrousługi. Żadna funkcjonalność biznesowa nie powinna wyciec na zewnątrz. Konfigurujemy generyczne operacje, takie jak sposób obsługi limitów czasu żądań. Jeśli chodzi o typowe zachowanie, które może wymagać dostosowania dla konkretnej mikrousługi, jest to zazwyczaj coś, co jest dobrze obsłużone wewnątrz mikrousługi i nie ma potrzeby wykonywania działań na centralnej platformie. Na przykład dzięki systemowi Istio można zdefiniować wymagania dotyczące obsługi limitów czasu z wykorzystaniem samoobsługi, po prostu zmieniając definicję usługi.

Czy potrzebujesz siatki usług?

Kiedy korzystanie z siatek usług zaczęło stawać się popularne, tuż po ukazaniu się pierwszego wydania tej książki, dostrzegłem wiele zalet w tym pomyśle. Widziałem jednak również wiele zagrożeń. Zaproponowano i zbudowano różne modele wdrażania, a następnie zrezygnowano z nich, a liczba firm oferujących rozwiązania w tej przestrzeni znacząco wzrosła; ale nawet w przypadku tych narzędzi, które istniały od dłuższego czasu, widoczny był brak stabilności. Firma Linkerd (*https://linkerd.io*), która prawdopodobnie zrobiła więcej niż inni, aby stać się pionierem w tym obszarze, przechodząc od wersji v1 do v2, całkowicie przebudowała swój produkt. Firma Istio (*https://istio.io*), oferująca rozwiązanie siatki usług namaszczone przez Google, potrzebowała lat, aby opublikować wydanie 1.0, a nawet po tym wydaniu była zmuszona wprowadzić znaczące zmiany w swojej architekturze (przechodząc nieco ironicznie, choć rozsądnie, do bardziej monolitycznego modelu wdrażania dla swojej platformy zarządzającej).

Przez ostatnie pięć lat, kiedy pytano mnie: „Czy powinniśmy zastosować siatkę usług?", moja rada brzmiała: „Jeśli możesz sobie pozwolić na odczekanie sześciu miesięcy przed dokonaniem wyboru, poczekaj sześć miesięcy". Byłem przekonany co do pomysłu, ale martwiłem się o stabilność. Taki mechanizm jak siatka usług nie jest miejscem, w którym chciałbym podejmować zbyt wielkie ryzyko — to zbyt kluczowy mechanizm konieczny do tego, aby wszystko dobrze działało. Należy ustawić go na ścieżce krytycznej. Wybór siatki usług radzę potraktować równie poważnie jak wybór brokera wiadomości lub dostawcy chmury.

Mogę z zadowoleniem stwierdzić, że ta przestrzeń dojrzała. Zmiany do pewnego stopnia spowolniły, ale nadal dostępna jest pewna (rozsądna) liczba dostawców. Trzeba jednak podkreślić, że siatki usług nie są dla wszystkich. Po pierwsze, jeśli nie korzystasz z platformy Kubernetes, Twoje opcje są ograniczone. Po drugie siatki usług zwiększają złożoność. Jeśli masz pięć mikrousług, to nie sądzę, abyś mógł łatwo uzasadnić zastosowanie siatki usług (można się zastanawiać, czy jeśli masz tylko pięć mikrousług, możesz uzasadnić zastosowanie Kubernetes). W przypadku organizacji, które mają więcej mikrousług, zwłaszcza jeśli chcą, aby dla tych mikrousług była dostępna opcja pisania ich w różnych językach programowania, siatki usług są warte zastanowienia. Trzeba jednak odrobić pracę domową — przełączanie się między siatkami usług może być kłopotliwe!

Monzo to jedna z organizacji, która otwarcie mówi o roli, jaką odegrało zastosowanie siatki usług do umożliwienia działania ich architektury w skali. Wykorzystanie do zarządzania wywołaniami RPC między mikrousługami wersji 1. systemu Linkerd okazało się niezwykle korzystne. Co ciekawe, firma Monzo musiała poradzić sobie z problemami związanymi z migracją siatki usług, aby

pomóc im osiągnąć właściwą skalę, gdy starsza architektura Linkerd w wersji 1. nie spełniała już jej wymagań (*https://oreil.ly/5dLGC*). Ostatecznie dzięki wykorzystaniu proxy Envoy udało się skutecznie dokonać przejścia na wewnętrzną siatkę usług.

A co z innymi protokołami?

Bramy API i siatki usług są używane głównie do obsługi wywołań związanych z protokołem HTTP. Tak więc za pomocą tych produktów mogą być zarządzane techniki REST, SOAP, gRPC i tym podobne. Sprawy jednak nieco bardziej się komplikują, gdy zaczynamy przyglądać się komunikacji z wykorzystaniem innych protokołów, na przykład brokerów wiadomości, takich jak Kafka. Zazwyczaj w tym momencie siatka usług zostaje pominięta — komunikacja odbywa się bezpośrednio z brokerem. Oznacza to, że nie można założyć, że siatka usług może działać jako pośrednik dla wszystkich wywołań między mikrousługami.

Dokumentowanie usług

Dzięki dekompozycji systemów na bardziej drobnoziarniste mikrousługi mamy nadzieję udostępnić wiele szwów w postaci interfejsów API, które ludzie mogą wykorzystywać do realizacji wielu często bardzo interesujących zadań. Jeśli wykrywanie usług przebiegnie bez problemów, to wiemy, gdzie są poszczególne usługi. Skąd jednak można się dowiedzieć, jakie operacje realizują te usługi lub jak można z nich korzystać? Jedną z opcji jest oczywiście skorzystanie z dokumentacji dotyczącej interfejsów API. Oczywiście dokumentacja często może być nieaktualna. W sytuacji idealnej powinniśmy zadbać o to, by dokumentacja zawsze była na bieżąco z interfejsem API mikrousługi oraz by można było łatwo z niej skorzystać, jeśli wiemy, gdzie jest punkt końcowy usługi.

Jawne schematy

Posługiwanie się jawnymi schematami znacznie ułatwia zrozumienie tego, co ujawnia dany punkt końcowy, ale same schematy często nie wystarczą. Jak już pisałem, schematy pomagają pokazać strukturę, ale nie pomagają zbytnio, jeśli chodzi o komunikowanie zachowania punktu końcowego. Nadal więc może być potrzebna dobra dokumentacja, która pomoże konsumentom zrozumieć, w jaki sposób należy korzystać z punktu końcowego. Warto oczywiście zauważyć, że jeśli zdecydujesz się nie używać jawnego schematu, Twoja dokumentacja będzie zmuszona wykonać *więcej* pracy. Musisz wyjaśnić, co robi punkt końcowy, a także udokumentować strukturę i szczegóły interfejsu. Co więcej, bez jawnego schematu wykrycie, czy dokumentacja jest aktualna i odpowiada rzeczywistym punktom końcowym, jest trudniejsze. Nieaktualna dokumentacja jest powszechnym problemem, ale jawny schemat zwiększa szansę na jej aktualność.

Wcześniej wprowadziłem już OpenAPI jako format schematu, ale technologia ta jest również bardzo skuteczna w dostarczaniu dokumentacji. Obecnie istnieje wiele narzędzi open source i komercyjnych, które obsługują korzystanie z deskryptorów OpenAPI. Za ich pomocą można stworzyć przydatne portale umożliwiające czytanie dokumentacji przez programistów. Warto zauważyć, że portale open source do przeglądania OpenAPI wydają się dość proste — na przykład starałem się znaleźć taki, który obsługiwał funkcje wyszukiwania. Dla użytkowników Kubernetesa szczególnie interesujący jest portal programistów systemu Ambassador (*https://oreil.ly/8pg12*). Ambassador

jest już popularnym wyborem jako brama API dla Kubernetes, a system Developer Portal należący do tego pakietu oprogramowania zapewnia możliwość automatycznego wykrywania dostępnych punktów końcowych OpenAPI. Pomysł wdrożenia nowej mikrousługi i automatycznego udostępniania jej dokumentacji bardzo do mnie przemawia.

W przeszłości brakowało nam dobrego wsparcia dla dokumentowania interfejsów opartych na zdarzeniach. Teraz przynajmniej mamy kilka opcji. Format AsyncAPI rozpoczął się jako adaptacja OpenAPI, a teraz mamy również CloudEvents, który jest projektem CNCF. Nie używałem żadnego z nich (to znaczy w rzeczywistej aplikacji), ale bardziej pociąga mnie CloudEvents i to tylko dlatego, że wydaje się zapewniać wiele możliwości i wsparcia, w dużej mierze ze względu na związek z CNCF. Przynajmniej historycznie CloudEvents wydawał się być bardziej restrykcyjny pod względem formatu zdarzeń w porównaniu z AsyncAPI, przy czym tylko JSON był prawidłowo obsługiwany, aż do niedawna, kiedy to po wcześniejszym usunięciu została przywrócona obsługa bufora protokołu. Zatem być może jest to opcja warta rozważenia.

System samoopisujący się

Podczas wczesnego rozwoju architektury SOA pojawiły się takie standardy jak UDDI (ang. *Universal Description, Discovery and Integration*). Miały one na celu pomóc użytkownikom zrozumieć sens działających usług. Były to dość skomplikowane specyfikacje, co skłoniło do poszukiwania alternatywnych technik odkrywania systemów. Martin Fowler opisał pojęcie rejestru ludzkiego (ang. *humane registry — https://oreil.ly/UI0YJ*). Zastosowano w nim znacznie prostsze podejście, polegające na wyznaczeniu miejsca, w którym członkowie organizacji mogliby rejestrować informacje — na przykład w formie czegoś tak prostego, jak strony Wiki.

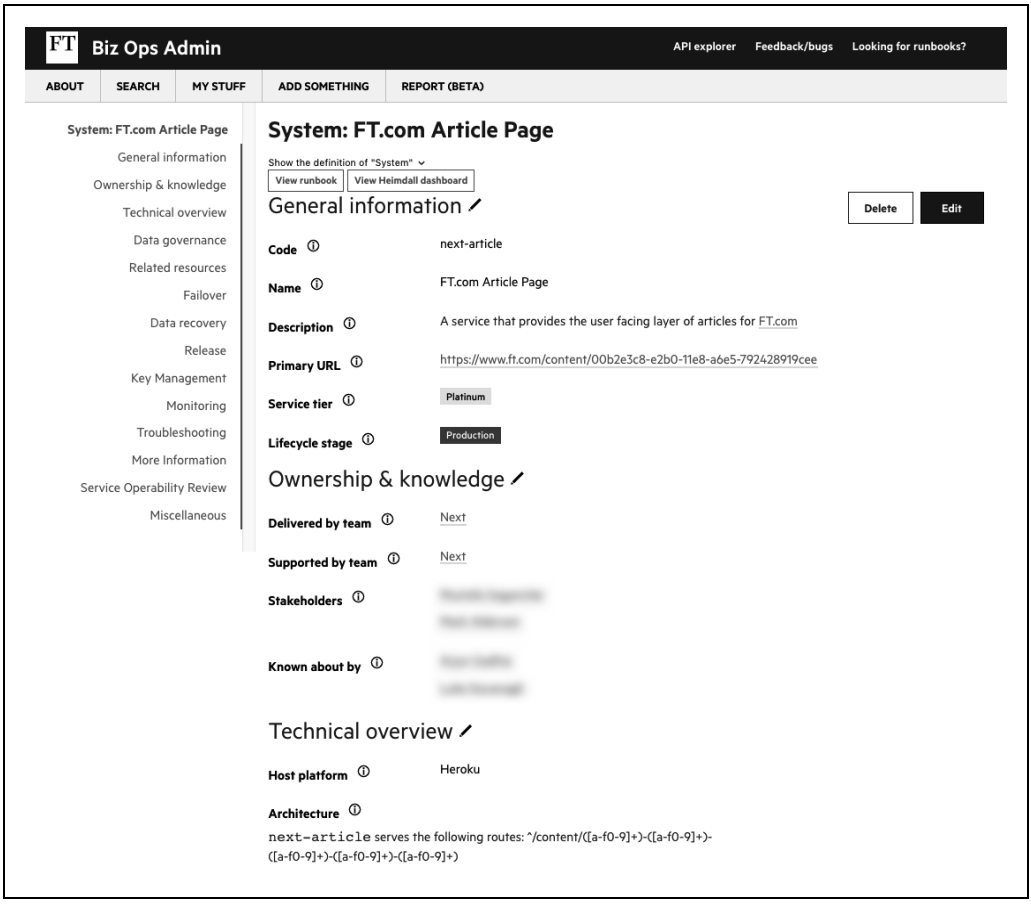
Uzyskanie obrazu systemu i sposobu, w jaki się zachowuje, jest bardzo ważne — zwłaszcza w przypadku systemów dużej skali. Wcześniej opisaliśmy szereg różnych technik, które pomagają w uzyskaniu wiedzy o systemie bezpośrednio z tego systemu. Dzięki śledzeniu kondycji usług działających w dole strumienia przetwarzania oraz zastosowaniu identyfikatorów korelacji pomagających w zaobserwowaniu łańcuchów wywołań możemy uzyskać prawdziwe dane dotyczące sposobu wzajemnych powiązań pomiędzy usługami. Za pomocą systemów wykrywania usług, takich jak Consul, możemy stwierdzić, gdzie działają mikrousługi. Mechanizmy takie jak OpenAPI i CloudEvents pozwalają ustalić funkcje oferowane za pośrednictwem konkretnych punktów końcowych, natomiast strony sprawdzania kondycji i systemy monitorowania pozwalają określić kondycję zarówno całego systemu, jak i pojedynczych usług.

Te informacje są dostępne programowo. Wszystkie dane pozwalają rozszerzyć możliwości ludzkiego rejestru tak, by stał się bardziej „potężny" niż proste strony Wiki, które z pewnością będą traciły aktualność. Strony Wiki można natomiast wykorzystać do wyświetlenia wszystkich informacji generowanych przez system. Dzięki stworzeniu niestandardowych paneli możemy wyodrębnić szeroką gamę dostępnych informacji, które pomagają zrozumieć sens naszego ekosystemu.

Za wszelką cenę należy dążyć do stworzenia takiego systemu. Warto zacząć od czegoś tak prostego jak statyczne strony WWW lub strony Wiki, które pobierają trochę danych z żywego systemu. Z czasem jednak warto zapewnić napływ coraz większych ilości informacji. Udostępnienie tych informacji jest kluczowym narzędziem zarządzania coraz większą złożonością, która wynika z działania systemów na dużą skalę.

Rozmawiałem z przedstawicielami wielu firm, które doświadczały podobnych problemów i które ostatecznie stworzyły proste wewnętrzne rejestry służące do tworzenia zestawień metadanych dotyczących usług. Niektóre z tych rejestrów, aby utworzyć listę usług, po prostu przeszukują repozytoria kodu źródłowego w poszukiwaniu plików z metadanymi. Informacje te można łączyć z rzeczywistymi danymi pochodzącymi z systemów wykrywania usług, takich jak Consul lub etcd. W ten sposób można stworzyć bogatszy obraz tego, co działa i z kim można o tym porozmawiać.

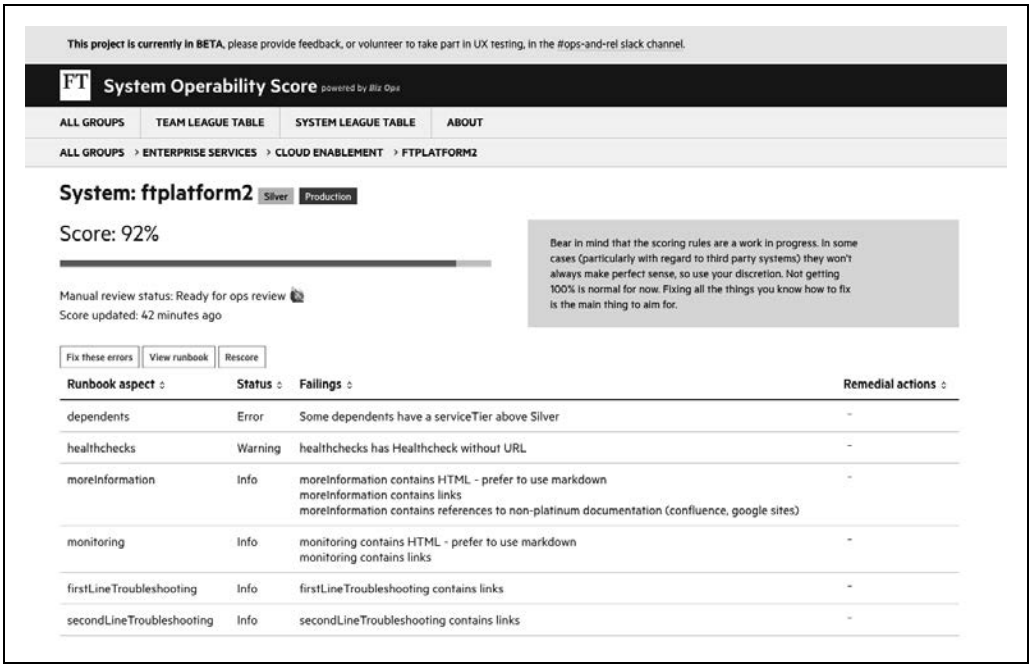
Aby pomóc rozwiązać ten problem, firma *Financial Times* stworzyła system Biz Ops. Firma ta zarządza kilkaset usługami opracowanymi przez zespoły na całym świecie. Narzędzie Biz Ops (rysunek 5.8) tworzy jedno miejsce, w którym oprócz informacji o innych usługach infrastruktury IT, takich jak sieci i serwery plików, można znaleźć wiele przydatnych informacji o mikrousługach. Zbudowany na bazie bazy danych wykresów Biz Ops zapewnia dużą elastyczność w zakresie rodzaju gromadzonych danych i sposobu modelowania informacji.



Rysunek 5.8. Narzędzie Financial Times Biz Ops, które zestawia informacje o swoich mikrousługach

Narzędzie Biz Ops idzie jednak dalej niż większość podobnych narzędzi, które widziałem. Program ten oblicza wartość, którą określa terminem *System Operability Score* (dosłownie: ocena sprawności systemu), co pokazałem na rysunku 5.9. Chodzi o to, że są pewne działania, które usługi oraz ich zespoły powinny wykonywać w celu zapewnienia łatwej obsługi usług. Może to obejmować takie operacje jak zadbanie o dostarczenie przez zespoły prawidłowych informacji w rejestrze, a także sprawdzenie, czy usługi są wyposażone w odpowiednie mechanizmy kontroli kondycji. Obliczona wartość współczynnika *System Operability Score* pozwala zespołom zobaczyć na pierwszy rzut oka, czy są elementy, które wymagają naprawy.

Ten obszar dynamicznie się rozwija. W świecie open source dostępne jest narzędzie Backstage (*https://backstage.io*) firmy Spotify, które podobnie jak Biz Ops oferuje mechanizm budowania katalogu usług. Narzędzie to jest wyposażone w model wtyczek umożliwiający wprowadzanie wyrafinowanych dodatków, takich jak możliwość zainicjowania tworzenia nowej mikrousługi lub pobierania na żywo informacji z klastra Kubernetes. Własny katalog usług systemu Ambassador (*https://oreil.ly/7o649*) jest skoncentrowany na widoczności usług w systemie Kubernetes, co oznacza, że może nie być tak dobrym narzędziem generycznym jak Biz Ops firmy Financial Times. Pomimo to warto zapoznać się z nowym podejściem do rozwiązań, które są bardziej dostępne.

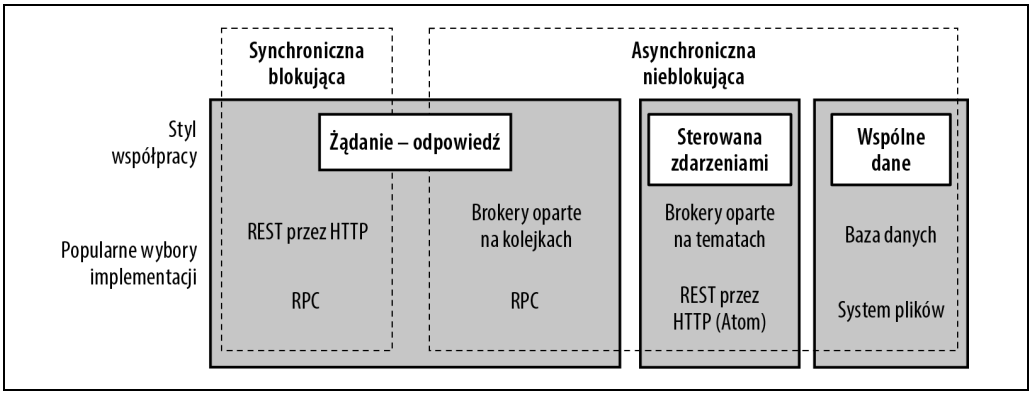


Rysunek 5.9. Przykład współczynnika Service Operability Score dla mikrousługi w Financial Times

Podsumowanie

W tym rozdziale omówiłem wiele pojęć — oto podsumowanie niektórych z nich:

- Na początek upewnij się, że o wyborze technologii decyduje problem, który próbujesz rozwiązać. Wybierz technologię, która jest dla Ciebie najbardziej odpowiednia, na podstawie kontekstu i preferowanego stylu komunikacji. Nie wpadnij w pułapkę wybierania technologii najpierw. W podjęciu decyzji może pomóc podsumowanie stylów komunikacji między mikrousługami, po raz pierwszy wprowadzone w rozdziale 4. i pokazane ponownie na rysunku 5.10. Samo podążanie za tym modelem nie zastępuje jednak dokładnego przeanalizowania własnej sytuacji.



Rysunek 5.10. Różne style komunikacji między mikrousługami wraz z przykładowymi technologiami implementacji

- Niezależnie od tego, jakiego wyboru dokonasz, rozważ użycie schematów, częściowo po to, aby pomóc w doprecyzowaniu kontraktów, ale także w celu wychwycenia przypadkowych zmian.
- Tam, gdzie to możliwe, staraj się wprowadzać zmiany, które są zgodne wstecz. Dzięki temu zapewnisz możliwość niezależnego wdrażania.
- Jeśli musisz wprowadzić zmiany niezgodne z poprzednimi wersjami, znajdź sposób, aby dać konsumentom czas na uaktualnienie. Pozwoli Ci to uniknąć wdrożeń typu lockstep.
- Zastanów się, co możesz zrobić, aby pomóc w udostępnieniu ludziom informacji o punktach końcowych — rozważ użycie ludzkich rejestrów i podobnych mechanizmów pozwalających zrozumieć chaos.

W tym rozdziale pokazałem sposób, w jaki można zaimplementować wywołanie między dwiema mikrousługami. Co się jednak stanie, gdy zajdzie potrzeba skoordynowania działań pomiędzy wieloma mikrousługami? Na tym skupię się w następnym rozdziale.

Architektura mikrousług: naucz się podejmować najlepsze decyzje projektowe!

Mikrousługi są dla wielu organizacji wspaniałą alternatywą. Systemy rozproszone złożone ze współpracujących ze sobą mikrousług pozwalają na efektywne i elastyczne dostarczanie oprogramowania, które ściśle odpowiada na wymagania użytkowników. Dodatkową zaletą jest możliwość relatywnie szybkiego wprowadzania modyfikacji w systemie, co ułatwia płynne zaspokajanie zmieniających się potrzeb. Pewnym wyzwaniem dla programistów jednak może być złożoność powodowana przez właściwości architektury systemów rozproszonych, a także pojawiające się wciąż nowe technologie i metodyki, które znacząco zmieniają sposób korzystania z mikrousług.

Oto drugie wydanie praktycznego przewodnika po projektowaniu, tworzeniu, wdrażaniu, skalowaniu i utrzymaniu systemów opartych na drobnoziarnistych mikrousługach. Publikacja została uzupełniona o informacje dotczące najnowszych trendów i technologii związanych z mikrousługami. Sporo miejsca poświęcono na staranne przeanalizowanie przykładów dotyczących opisywanych koncepcji, a także pokazanie optymalnych sposobów rozwiązywania różnych problemów. Opisano również najnowsze rozwiązania dotyczące modelowania, integracji, testowania, wdrażania i monitorowania autonomicznych usług. Bardzo interesującą częścią są studia przypadków, w których przeanalizowano, jak organizacjom udaje się w praktyce w pełni wykorzystywać możliwości mikrousług.

Dzięki książce dowiesz się, jak:

- przeprowadzać orkiestrację kontenerów i wdrażać rozwiązania bezserwerowe
- dostosowywać projekt systemu do potrzeb organizacji
- wybrać najlepszy sposób integracji usługi z systemem
- samodzielnie wdrażać mikrousługi
- skutecznie testować i monitorować usługi rozproszone
- zarządzać zabezpieczeniami dla rozszerzonej zawartości

Sam Newman jest niezależnym konsultantem, autorem i prelegentem. Od ponad 20 lat pracuje ze stosami technologicznymi, w różnych branżach. Jego głównym celem jest pomagać organizacjom w szybszym i bezpieczniejszym dostarczaniu oprogramowania do produkcji, a także w poruszaniu się po złożoności mikrousług.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-8800-0
9 788328 388000
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 109,00 zł