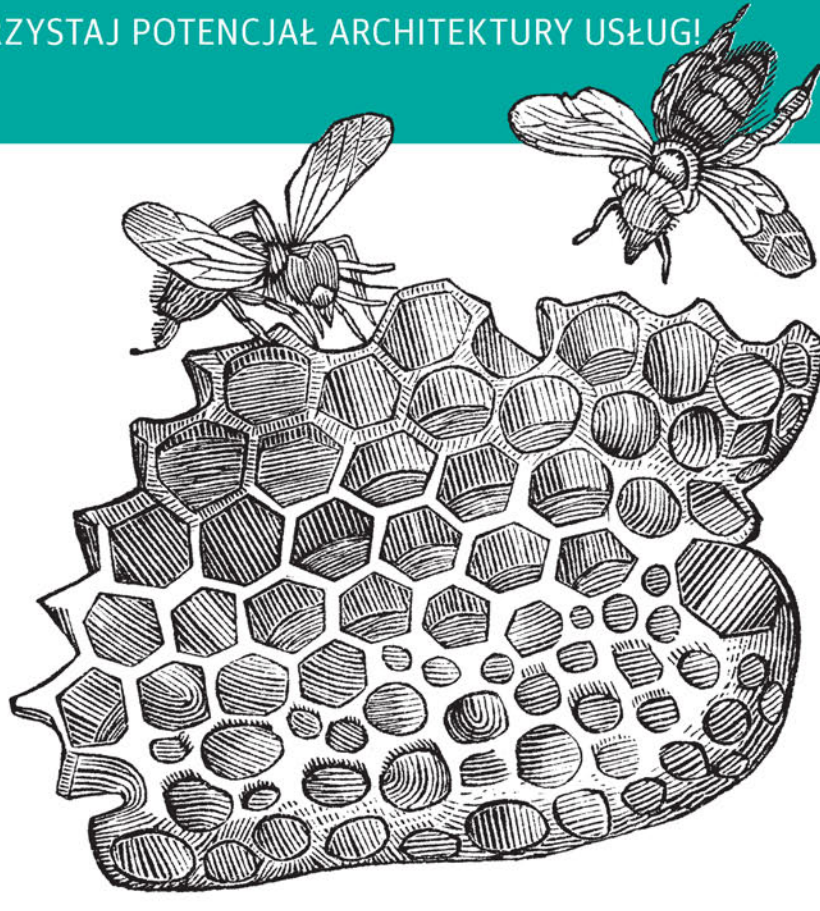
O'REILLY®

Budowanie mikrousług

Szybkie wprowadzenie

WYKORZYSTAJ POTENCJAŁ ARCHITEKTURY USŁUG!



Helion

Sam Newman

Tytuł oryginału: Building Microservices

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-1381-1

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 1.
Mikrousługi

Od wielu lat próbujemy znaleźć lepsze sposoby budowania systemów. Uczymy się na podstawie tego, co było wcześniej, adoptujemy nowe technologie i obserwujemy różne sposoby działania przedstawicieli nowej fali firm po to, aby tworzyć systemy informatyczne, które uszczęśliwią zarówno klientów, jak i twórców systemów.

Książka Erica Evansa *Domain-Driven Design* (Addison-Wesley) pomogła nam zrozumieć znaczenie reprezentowania w kodzie rzeczywistego świata oraz pokazała lepsze sposoby modelowania naszych systemów. Koncepcja ciągłych dostaw (*continuous delivery*) pokazała, jak można skuteczniej i wydajniej wdrażać tworzone oprogramowanie. Zaszczepiła w nas pomysł, aby wersję z każdej operacji wgrania kodu do repozytorium (*check-in*) traktować jako potencjalną wersję do publikacji. Zrozumienie sposobu działania sieci WWW doprowadziło nas do opracowania lepszych sposobów komunikowania się maszyn. Koncepcja architektury sześciokątnej Alistaira Cockburna (*http://alistair.cockburn.us/Hexagonal+architecture*) odwiodła nas od architektur warstwowych, które pozwalały ukryć logikę biznesową. Platformy wirtualizacji pozwoliły na konfigurowanie i zmianę rozmiarów maszyn, na żądanie gwarantując automatyzację infrastruktury, która umożliwia obsługę tych maszyn w skali. Niektóre duże organizacje, takie jak Amazon i Google, odniosły sukces, forsując wizję niewielkich zespołów odpowiedzialnych za pełny cykl życia swoich usług. A w ostatnim czasie firma Netflix zaproponowała sposoby budowania elastycznych systemów (*antifragile* — dosł. 'antykruchych') w skali, która zaledwie 10 lat temu byłaby trudna do zrozumienia.

Projektowanie tematyczne (*domain-driven design* — DDD). Ciągłe dostarczanie. Wirtualizacja na żądanie. Automatyzacja infrastruktury. Małe autonomiczne zespoły. Systemy w skali. Z tego świata wyłoniły się mikrousługi. Nie zostały wymyślone lub opisane „z góry", przed wdrożeniem — pojawiły się one jako trend lub wzorzec z praktycznych zastosowań. Jednak istnieją tylko dlatego, że wcześniej stosowano wszystkie wymienione powyżej techniki. W tej książce będę wyciągał wnioski z tych wcześniejszych prac po to, by wskazać sposób, w jaki można budować, zarządzać i rozwijać mikrousługi.

W wielu firmach zdano sobie sprawę, że wykorzystanie drobnoziarnistych architektur mikrousług pozwala stosować nowe technologie i szybciej dostarczać oprogramowanie. Mikrousługi dają nam znacznie więcej swobody reakcji oraz podejmowania różnych decyzji, co pozwala nam szybciej reagować na nieuniknione zmiany, które mają wpływ na nas wszystkich.

Czym są mikrousługi?

Mikrousługi są niewielkimi, współpracującymi ze sobą, autonomicznymi usługami. Spróbujmy rozbić tę definicję na mniejsze części i przeanalizować cechy, które wyróżniają mikrousługi spośród innych architektur.

Niewielkie, skoncentrowane na dobrym wykonywaniu jednej rzeczy

W wyniku pisania kodu, który dodaje nowe funkcje, bazy kodu rozrastają się. Z czasem ze względu na to, że baza kodu rozrośnie się do olbrzymich rozmiarów, stwierdzenie, gdzie należy wprowadzić zmiany, może być trudne. Pomimo że dąży się do czytelnych, modułowych, monolitycznych baz kodu, nazbyt często te arbitralne granice wewnątrz procesów się rozpływają. Kod pełniący podobne funkcje zaczyna rozsiewać się po całym projekcie. W związku z tym usuwanie błędów lub implementacja stają się trudniejsze.

W ramach monolitycznego systemu zwalczamy te trendy, starając się zapewnić większą spójność kodu. Często w tym celu tworzymy nowe abstrakcje lub moduły. Spójność — dążenie do grupowania powiązanego ze sobą kodu — to bardzo ważne pojęcie, kiedy zaczynamy myśleć o mikrousługach. Cechę tę wzmacnia **zasada pojedynczej odpowiedzialności** (*http://programmer.97things.oreilly.com/wiki/index.php/The_Single_Responsibility_Principle*) Roberta C. Martina, która mówi „Pogrupuj ze sobą te elementy, które będą się zmieniać z tego samego powodu, i rozdziel te, które zmieniają się z różnych powodów".

Mikrousługi pozwalają zastosować to samo podejście do niezależnych usług. Wyznaczając granice usług, koncentrujemy się na kategoriach biznesowych. Dzięki temu lokalizacja fragmentu kodu odpowiedzialnego za określone funkcjonalności staje się oczywista. Dzięki zapewnieniu czytelnych granic usługi możemy uniknąć pokusy zbytniej rozbudowy kodu — wraz ze wszystkimi związanymi z tym trudnościami.

Pytanie, które często jest mi zadawane, brzmi: *ile to jest mało*? Podanie dokładnej liczby wierszy kodu stwarza problemy, ponieważ niektóre języki są bardziej ekspresywne od innych, dlatego pozwalają na wyrażenie więcej za pomocą mniejszej liczby wierszy kodu. Trzeba także wziąć pod uwagę fakt wielu możliwych zależności, które same w sobie zawierają wiele wierszy kodu. Ponadto niektóre części dziedziny mogą być bardziej złożone od innych, co może wymagać dodatkowych ilości kodu. Jon Eaves z firmy RealEstate.com.au z Australii charakteryzuje mikrousługę jako kod, który można przepisać w dwa tygodnie — to reguła „spod dużego palca", która ma sens w tym konkretnym kontekście.

Inna, trochę banalna odpowiedź, jakiej mogę udzielić, brzmi: *małe jest wystarczająco małe, ale nie mniejsze*. Wygłaszając referaty na konferencjach, prawie zawsze zadaję pytanie: *kto utrzymuje system, który jest zbyt duży i który należałoby rozdzielić*? Prawie wszyscy podnieśli ręce. Wydaje się, że mamy bardzo dobre poczucie tego, co jest zbyt duże, a więc można argumentować, że jeśli fragment kodu nie będzie nam się wydawał zbyt duży, to prawdopodobnie jest wystarczająco mały.

Bardzo pomocne w udzieleniu odpowiedzi na pytanie o to, jak mały jest fragment kodu, jest porównanie kodu do struktury zespołu. Jeśli baza kodu jest zbyt duża, aby mogła być zarządzana przez mały zespół, dążenie do jej podzielenia jest bardzo rozsądne. Tematykę organizacji zespołów zgodnie z rozmiarami bazy kodu poruszymy w dalszej części tej książki.

Przy próbach rozstrzygania, jak małe jest wystarczająco małe, myślę w następujących kategoriach: im mniejsza usługa, tym łatwiej zmaksymalizować korzyści i zminimalizować wady architektury mikrousług. W miarę jak usługi stają się coraz mniejsze, zwiększają się korzyści wynikające ze współzależności. Jednak wprowadza to również pewną złożoność. Jej źródłem jest istnienie coraz większej liczby ruchomych części — opowiemy o tym w dalszej części tej książki. Im lepiej uda się obsłużyć tę złożoność, tym łatwiej będzie można stosować coraz mniejsze usługi.

Autonomiczne

Mikrousługa jest odrębnym podmiotem. Może być wdrożona jako odizolowana usługa w ramach infrastruktury PaaS (*Platform as a Service*) lub może być procesem systemu operacyjnego. Staramy się unikać pakowania zbyt wielu usług na tej samej maszynie, mimo że definicja **maszyny** w dzisiejszym świecie jest dość mglista! Zgodnie z tym, co powiemy później, chociaż ta izolacja może wprowadzić pewien narzut, to wynikowa prostota sprawia, że nasz rozproszony system jest znacznie łatwiejszy do analizowania, a zastosowanie nowszych technologii pozwala złagodzić wiele wyzwań związanych z tą formą wdrażania.

Cała komunikacja pomiędzy samym usługami odbywa się za pośrednictwem wywołań sieci. To pozwala wymusić podział między usługami i unikać niebezpieczeństw związanych ze zbyt ścisłymi sprzężeniami.

Usługi muszą pozwalać na wprowadzanie zmian niezależnie od innych usług oraz pozwalać na odrębne wdrażanie — bez konieczności zmiany konsumentów. Powinniśmy pomyśleć o tym, co nasze usługi mają udostępniać i jakie cechy mają być możliwe do ukrycia. Jeśli istnieje zbyt wiele współdzielenia, usługi konsumenckie stają się sprzężone z wewnętrznymi reprezentacjami. To obniża autonomię, ponieważ wymaga dodatkowej koordynacji z konsumentami podczas wprowadzania zmian.

Usługa udostępnia interfejs programowania aplikacji (API), przez który komunikują się z nią usługi współpracujące. Musimy także pomyśleć o tym, jaka technologia jest odpowiednia do tego, aby nie wprowadzała sprzężeń z konsumentami. Może to oznaczać wybieranie interfejsów API agnostycznych pod względem technologii, tak by nie ograniczać możliwości wyboru technologii. Do tego, jak ważne są dobre, pozbawione sprzężeń interfejsy API, będziemy powracać w całej książce.

Bez eliminacji sprzężeń cała architektura upada. Istnieje złota zasada: czy potrafisz wprowadzić zmiany w usłudze i wdrożyć ją osobno bez zmieniania czegokolwiek innego? Jeśli odpowiedź na to pytanie brzmi „nie”, to wiele korzyści, które omawiamy na kartach całej tej książki, będzie trudne do osiągnięcia.

Aby dobrze rozdzielić interfejsy API, należy właściwie zamodelować usługi oraz odpowiednio zaprojektować interfejsy API. Ten temat będzie często poruszany w tej książce.

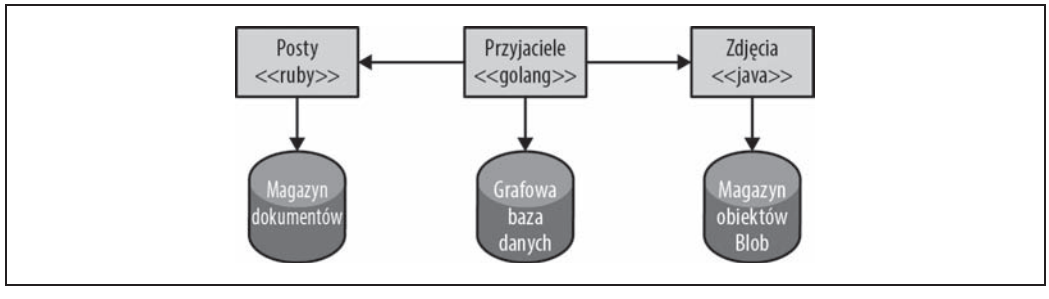
Najważniejsze korzyści

Korzyści ze stosowania mikrousług jest wiele i są one zróżnicowane. Wiele z nich uzyskuje się dla każdego systemu rozproszonego. Stosowanie mikrousług pozwala jednak na osiągnięcie tych korzyści w większym stopniu. Przede wszystkim ze względu na poziom wykorzystania systemów rozproszonych i architektury zorientowanej na usługi.

Niejednorodność technologii

W systemie złożonym z wielu współpracujących ze sobą usług do zaimplementowania każdej z tych usług można zastosować inną technologię. Pozwala to na wybranie odpowiednich narzędzi dla każdego zadania. Nie trzeba stosować bardziej znormalizowanego, jednego uniwersalnego narzędzia, które często pełni funkcję najmniejszego wspólnego mianownika.

Jeśli w jednej części systemu trzeba poprawić wydajność, możemy zdecydować się na wykorzystanie innego stosu technologii — takiego, który pozwala łatwiej osiągnąć wymagany poziom wydajności. Możemy też zadecydować że sposób przechowywania danych w różnych częściach systemu będzie różny. Na przykład w sieci społecznościowej interakcje pomiędzy użytkownikami mogą być przechowywane w bazie danych o strukturze grafu. Taka baza danych odzwierciedla charakter wielu wewnętrznych powiązań występujących w społeczności, ale posty, które użytkownicy publikują, mogą być przechowywane w magazynie danych zorientowanym na dokumenty. W ten sposób powstaje architektura heterogeniczna, podobna do takiej, którą pokazano na rysunku 1.1.



Rysunek 1.1. Mikrousługi pozwalają na łatwe wykorzystywanie różnych technologii

Dzięki zastosowaniu mikrousług jesteśmy również w stanie szybciej adaptować technologie i lepiej zrozumieć, jak mogą nam pomóc nowe usprawnienia. Jedną z największych barier podczas próbowania i adoptowania nowych technologii są związane z nimi zagrożenia. W przypadku monolitycznej aplikacji wypróbowanie nowego języka programowania, bazy danych lub frameworka przy każdej zmianie będzie wpływać na duży fragment systemu. W przypadku systemu składającego się z wielu usług jest wiele miejsc, w których można wypróbować nowe technologie. Można wybrać usługę, której modyfikacja wiąże się z najmniejszym ryzykiem, i tam spróbować nowej technologii, mając świadomość możliwości ograniczenia wszelkich potencjalnych negatywnych skutków. W wielu firmach przyjmuje się stanowisko, że zdolność szybszej absorpcji nowych technologii przynosi realne korzyści.

Wykorzystywanie wielu technologii oczywiście nie przychodzi bez kosztów. W niektórych firmach stosowane są pewne ograniczenia odnośnie wyboru języka. Na przykład w firmach Netflix i Twitter najczęściej w roli platformy wykorzystywana jest maszyna wirtualna Javy (*Java Virtual Machine* — JVM) ze względu na doświadczenia tych firm w zakresie niezawodności i wydajności tego systemu. Firmy te tworzą również biblioteki i narzędzia dla platformy JVM, dzięki którym skalowanie systemów staje się znacznie łatwiejsze, a jednocześnie jest trudne do wykorzystania przez usługi i aplikacje klienckie, które nie są napisane w Javie. Ale ani Twitter, ani Netflix nie stosują tylko jednego stosu technologii do realizacji wszystkich zadań. Innym czynnikiem, który należy wziąć pod uwagę w przypadku mieszania różnych technologii, jest skala systemu. Jeśli naprawdę można przepisać mikrousługę w ciągu dwóch tygodni, z łatwością może również zmniejszyć zagrożenia związane z zastosowaniem nowej technologii.

Podczas lektury tej książki przekonamy się, że w wielu aspektach projektowania systemów bazujących na mikrousługach najistotniejsze jest znalezienie odpowiedniej równowagi. Sposoby dokonywania wyboru technologii omówimy w rozdziale 2., który skupia się na ewolucji architektury, a także w rozdziale 4., poświęconym zagadnieniom integracji. Dowiemy się z nich, w jaki sposób zapewnić ewolucję stosowanych technologii wewnątrz usług niezależnie od siebie — bez zbędnych sprzężeń.

Odporność na błędy

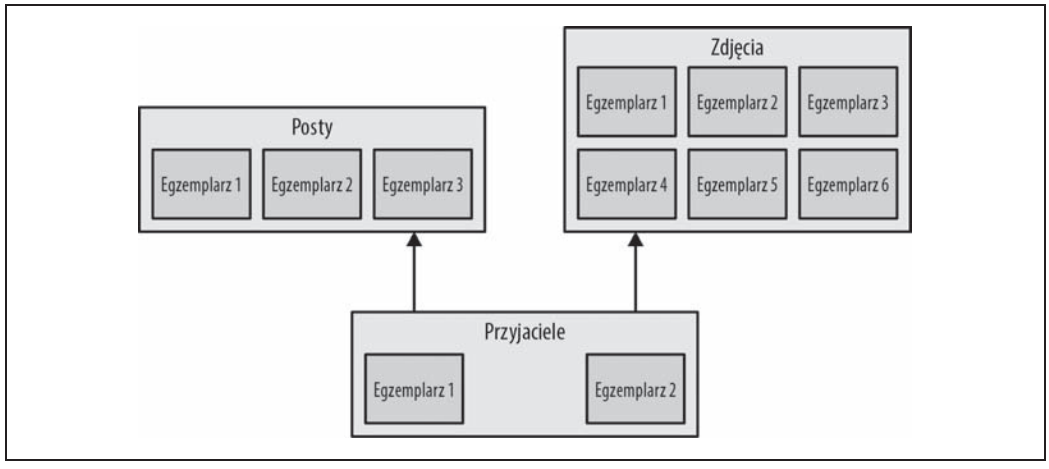
Kluczowym pojęciem w inżynierii odporności na błędy są tzw. przegrody (*bulkhead*). Jeśli jeden składnik systemu ulegnie awarii, ale ta awaria nie rozprzestrzenia się kaskadowo, to można wyizolować problem, a reszta systemu może kontynuować pracę. Granice usług stają się oczywistymi przegrodami. W usłudze monolitycznej awaria powoduje, że wszystko przestaje działać. System monolityczny można uruchomić na wielu maszynach, aby zmniejszyć ryzyko awarii, ale w przypadku mikrousług możemy budować systemy, które obsługują całkowitą awarię usług i odpowiednią degradację funkcjonalności.

Trzeba jednak zachować ostrożność. Aby upewnić się, że systemy bazujące na mikrousługach właściwie wykorzystują tę poprawioną odporność, musimy zdać sobie sprawę z nowych źródeł awarii, z którymi muszą sobie radzić systemy rozproszone. Sieci, podobnie jak komputery, mogą ulegać awariom i to czasami się zdarza. Trzeba wiedzieć, jak sobie z tym poradzić i jaki wpływ (jeśli w ogóle) powinna mieć awaria na użytkowników naszego oprogramowania.

Więcej informacji na temat obsługi odporności na błędy oraz sposobów obsługi trybów awarii można znaleźć w rozdziale 11.

Skalowanie

W dużej monolitycznej usłudze wszystkie elementy muszą być skalowane razem. Jedna mała część całego systemu ma ograniczoną wydajność, ale jeśli to zachowanie jest zamknięte w gigantycznej monolitycznej aplikacji, musimy obsłużyć skalowanie wszystkiego jako komponentu. W przypadku korzystania z mniejszych usług możemy skalować tylko te usługi, które wymagają skalowania. Dzięki temu inne części systemu mogą być uruchamiane na słabszym sprzęcie (patrz rysunek 1.2).



Rysunek 1.2. Skalowanie może dotyczyć tylko tych mikrousług, które wymagają skalowania

Gilt, firma zajmująca się internetową sprzedażą detaliczną odzieży, zastosowała mikrousługi dokładnie w tym celu. W 2007 roku system miał postać monolitycznej aplikacji Rails. Do 2009 roku system firmy Gilt nie był w stanie obsłużyć przychodzącego obciążenia. Dzięki podzieleniu podstawowych części swojego systemu firmie Gilt udało się lepiej obsłużyć szczytowe natężenie ruchu. Obecnie system składa się z ponad 450 mikrousług, z których każda działa na kilku odrębnych komputerach.

W przypadku systemów konfigurowania na żądanie, podobnych do tych, które są stosowane w usługach Amazon Web Services, można nawet stosować skalowanie na żądanie tylko dla tych fragmentów, które tego wymagają. To pozwala skuteczniej kontrolować koszty. Sytuacja, w której zmiany w architekturze są tak ściśle skorelowane z niemal natychmiastowymi oszczędnościami, nie jest zbyt częsta.

Łatwość wdrażania

Opublikowanie zmiany jednego wiersza w monolitycznej aplikacji zawierającej milion wierszy kodu wymaga przygotowania wydania całej aplikacji. To może być operacja wywierająca duży wpływ na użytkowników i obarczona wysokim ryzykiem. W praktyce takie operacje są wykonywane rzadko ze względu na zrozumiałe obawy. Niestety, to oznacza, że liczba zmian pomiędzy wydaniami kumuluje się. Kiedy nowa wersja aplikacji trafi do produkcji, zawiera olbrzymią liczbę zmian w porównaniu z poprzednią wersją. A im większe różnice pomiędzy wydaniami, tym większe ryzyko popełnienia błędów!

W przypadku zastosowania mikrousług możemy wprowadzić zmiany w pojedynczej usłudze i wdrożyć ją niezależnie od reszty systemu. Ta własność pozwala nam na szybsze wdrażanie kodu. Jeśli wystąpi problem, może być szybko wyizolowany do pojedynczej usługi. Dzięki temu można łatwo cofnąć wprowadzoną zmianę. Oznacza to również, że możemy łatwiej udostępnić nową funkcjonalność klientom. Jest to jeden z głównych powodów, dla których takie firmy jak Amazon i Netflix stosują takie architektury — robią to po to, aby usunąć jak najwięcej przeszkód związanych z publikowaniem oprogramowania.

W ciągu ostatnich kilku lat technologia w tym obszarze znacznie się zmieniła. Więcej informacji na temat wdrażania mikrousług można znaleźć w rozdziale 6.

Dopasowanie do organizacji zespołów

Wielu z nas doświadczyło problemów związanych z dużymi zespołami i rozbudowanymi bazami kodu. Problemy te mogą się dodatkowo nasilić w przypadku, gdy zespół jest rozproszony. Wiemy również, że mniejsze zespoły pracujące na mniejszych bazach kodu zwykle są bardziej wydajne.

Mikrousługi pozwalają na lepsze dopasowanie architektury do organizacji oraz pomagają zminimalizować liczbę osób pracujących nad określoną bazą kodu. W ten sposób można łatwiej dostosować proporcje pomiędzy wielkością zespołu a jego wydajnością. Możemy również dokonać zmiany własności usług pomiędzy zespołami, aby osoby zajmujące się jedną usługą pracowały we wspólnej lokalizacji. Znacznie więcej szczegółów na ten temat podamy przy okazji omawiania prawa Conwaya w rozdziale 10.

Interoperatywność

Jedną z głównych korzyści, jaką powinno przynieść zastosowanie systemów rozproszonych i architektury zorientowanej na usługi, jest możliwość wielokrotnego wykorzystania funkcjonalności. W przypadku systemu bazującego na mikrousługach funkcjonalności mogą być wykorzystywane w różny sposób w różnych celach. Może to być szczególnie ważne w przypadku, gdy weźmiemy pod uwagę sposób, w jaki konsumenci używają naszego oprogramowania. Dawno minęły czasy, kiedy można było postrzegać aplikację w wąskiej perspektywie — jako aplikację desktop, witrynę WWW lub aplikację mobilną. Obecnie trzeba wziąć pod uwagę mnóstwo sposobów kombinowanego wykorzystania możliwości oprogramowania w internecie, aplikacjach natywnych, mobilnych witrynach Web, aplikacjach na tablety, telefony komórkowe lub urządzenia przenośne. W miarę jak organizacje odchodzą od myślenia w kategoriach wąskich kanałów i skłaniają się do bardziej holistycznych koncepcji zaangażowania klienta, potrzebujemy architektur, które będą mogły sprostać nowym wymaganiom.

Zastosowanie mikrousług pozwala nam zwolnić szwy w naszym systemie, które mogą być wykorzystane do integracji systemów zewnętrznych. W przypadku zmian okoliczności możemy zbudować system w inny sposób. Aplikacje monolityczne często są wyposażone w jeden „gruby szew", który może być użyty z zewnątrz. Aby zerwać ten szew w celu uzyskania czegoś bardziej pożytecznego, potrzebny jest młot! W rozdziale 5. zostaną omówione sposoby podziału istniejących monolitycznych systemów i ich modyfikowania do postaci mikrousług umożliwiających wielokrotne użytkowanie i komponowanie w dowolnych konfiguracjach.

Optymalizacja w kierunku wymienności

W dużych lub średnich firmach często działają duże, rozbudowane i przestarzałe systemy. Są to systemy, których nikt nie chce dotykać. Mają kluczowe znaczenie dla działania firmy, ale często zostały napisane w jakimś dziwnym dialekcie Fortrana i działają wyłącznie na sprzęcie, którego nie produkuje się od co najmniej 25 lat. Dlaczego taki system nie został zastąpiony innym? Wiadomo dlaczego: jest zbyt duży, a jego wymiana zbyt ryzykowna.

W przypadku pojedynczych usług niewielkich rozmiarów koszty zastąpienia wybranej usługi inną, lepszą implementacją, lub nawet całkowita rezygnacja z pojedynczej usługi, są znacznie niższe, a operacja jest znacznie łatwiejsza do przeprowadzenia. Przypomnijmy sobie, jak często zdarza się nam usunąć ponad sto wierszy kodu jednego dnia, nie martwiąc się zbytnio tym, jakie zmiany to spowoduje. Ponieważ mikrousługi często mają podobne rozmiary, opór przed przepisaniem lub całkowitym usunięciem usługi jest bardzo mały.

Zespoły stosujące architektury bazujące na mikrousługach nie wahają się przepisywać od podstaw usług, jeśli zachodzi taka konieczność, lub po prostu eliminować pojedynczych usług, jeśli przestaną one być potrzebne. Jeśli baza kodu obejmuje zaledwie kilkaset wierszy, trudno emocjonalnie się do niej przywiązać, a koszty zastąpienia jej inną są dość małe.

Architektura zorientowana na usługi

Architektura zorientowana na usługi (*service-oriented architecture* — SOA) to podejście do projektowania, w którym wiele usług współpracuje ze sobą w celu świadczenia pewnego skończonego zbioru możliwości. Usługa zazwyczaj oznacza w tym przypadku całkowicie odrębny proces systemu operacyjnego. Komunikacja pomiędzy tymi usługami odbywa się za pośrednictwem wywołań przez sieć zamiast wywołań metod w granicach procesu.

Architektury SOA pojawiły się jako podejście do zwalczania wyzwań właściwych dla dużych, monolitycznych aplikacji. Jest to podejście, które ma na celu promowanie wielokrotnego wykorzystania oprogramowania. Na przykład dwie lub większa liczba aplikacji użytkowych mogą jednocześnie korzystać z tych samych usług. Architektury SOA mają na celu ułatwienie utrzymania lub przepisania oprogramowania. Teoretycznie można zastąpić jedną usługę inną bez niczyjej wiedzy, pod warunkiem że semantyka nowej usługi zbytnio się nie zmieni.

SOA w swojej istocie jest bardzo sensownym pomysłem. Jednak pomimo wielu wysiłków brakuje dobrego konsensusu co do tego, w jaki sposób *dobrze* budować architektury SOA. Moim zdaniem wśród większości przedstawicieli branży brakuje dostatecznie holistycznego spojrzenia na ten problem. Prezentowane są przekonujące alternatywy wyrażane przez różnych dostawców.

Wiele problemów przyporządkowywanych do architektury SOA to w rzeczywistości problemy dotyczące takich elementów jak protokoły komunikacyjne (na przykład SOAP), dostawcy warstwy middleware, brak wytycznych na temat ziarnistości usługi lub niewłaściwe wytyczne dotyczące wyboru miejsc podziału systemu. Każdym z tych problemów zajmiemy się po kolei w dalszej części książki. Cynik mógłby zasugerować, że producenci wykorzystali (a w niektórych przypadkach zmodyfikowali) ruch SOA do zwiększenia sprzedaży swoich produktów. Takie podejście w gruncie rzeczy pozostaje w sprzeczności z celem SOA.

Znaczna część wiedzy na temat SOA nie pozwala zrozumieć tego, jak podzielić duży system w taki sposób, by uzyskać mały. Nie wynika z niej odpowiedź na pytanie: *jak dużo, to za dużo*. Nie mówi wystarczająco dużo na temat rzeczywistych, praktycznych sposobów eliminacji niepotrzebnych sprzężeń pomiędzy usługami. Te wszystkie niedopowiedziane rzeczy są źródłem wielu pułapek związanych z architekturami SOA.

Podejście do projektowania bazujące na mikrousługach pochodzi z zastosowań w rzeczywistym świecie, uwzględnia lepsze zrozumienie systemów i architektury tak, by właściwie projektować architekturę SOA. Zatem o mikrousługach powinniśmy myśleć raczej jak o szczególnym podejściu do SOA, na podobnej zasadzie, na jakiej XP lub Scrum prezentują specyficzne podejście do zwinnego wytwarzania oprogramowania.

Inne techniki dekompozycji

Kiedy przyjrzymy się mikrousługom bliżej, okaże się, że wiele zalet architektury bazującej na mikrousługach wynika z jej granularnej natury oraz tego, że gwarantuje ona dużo więcej możliwości rozwiązywania problemów. Jednak czy można osiągnąć te same korzyści, stosując inne techniki dekompozycji?

Biblioteki współdzielone

Standardową techniką dekompozycji, która jest wbudowana w praktycznie każdym języku programowania, jest podzielenie bazy kodu na wiele bibliotek. Biblioteki te mogą być dostarczane przez producentów zewnętrznych lub tworzone we własnym zakresie.

Biblioteki gwarantują sposób współdzielenia funkcjonalności pomiędzy zespołami i usługami. Na przykład można utworzyć kolekcję przydatnych narzędzi lub bibliotekę obliczeń statystycznych, która może być wykorzystywana wielokrotnie.

Wokół tych bibliotek można organizować zespoły, a same biblioteki mogą być wykorzystywane wielokrotnie. Są jednak pewne wady takiego podejścia.

Po pierwsze, tracimy prawdziwą różnorodność technologii. Biblioteki zazwyczaj muszą być napisane w tym samym języku lub przynajmniej działać na tej samej platformie. Po drugie, łatwość, z jaką można skalować części systemu niezależnie od siebie, jest ograniczona. Poza tym, jeśli nie używamy bibliotek łączonych dynamicznie, nie możemy zainstalować nowej biblioteki bez ponownej instalacji całego procesu, zatem zdolność do wdrażania zmian w izolacji jest ograniczona. I być może najgorszy jest brak oczywistych „szwów", które zapewniają architekturze środki bezpieczeństwa uodparniające system na awarie.

Biblioteki współdzielone mają swoje zastosowania. Czasami tworzy się kod dla typowych zadań, które nie są specyficzne dla domeny biznesowej, a które mają być wielokrotnie wykorzystywane w różnych miejscach organizacji. Taki kod jest oczywistym kandydatem do tego, by stać się biblioteką wielokrotnego użytku. Trzeba jednak zachować ostrożność. Współdzielony kod używany do komunikacji między usługami może stać się punktem sprzężeń — czymś, co będziemy omawiali w rozdziale 4.

W usługach można i należy korzystać z zewnętrznych bibliotek w celu wielokrotnego wykorzystywania wspólnego kodu. Jednak nie są one rozwiązaniem wszystkich problemów.

Moduły

W niektórych językach dostępne są odrębne techniki dekompozycji bazujące na modułach, których możliwości wykraczają poza proste biblioteki. Pozwalają one na zarządzanie cyklem życia modułów — na przykład możliwością instalowania wewnątrz działających procesów, co pozwala na wprowadzanie zmian bez zatrzymywania całego procesu.

Warto wymienić projekt OSGi (*Open Source Gateway initiative*) jako przykład specyficznego dla technologii podejścia do dekompozycji na moduły. W Javie nie istnieje rzeczywista koncepcja modułów. Na dodanie tej konstrukcji do języka trzeba będzie poczekać co najmniej do publikacji Java 9. Obecnie, aby dodać koncepcję modułów w Javie za pośrednictwem biblioteki, wykorzystywany jest komponent OSGi, który pojawił się jako framework umożliwiający instalowanie wtyczek w środowisku IDE języka Java Eclipse.

Problem z OSGi polega na tym, że framework ten stara się wprowadzić takie elementy jak zarządzanie cyklem życia modułów bez wystarczającego wsparcia w obrębie samego języka. W efekcie autorzy modułów, aby zapewnić właściwą izolację, muszą wykonać więcej pracy. Ponadto w ramach procesu znacznie łatwiej wpaść w pułapkę tworzenia modułów zbyt mocno sprzężonych ze sobą, co sprawia różnego rodzaju problemy. Moje doświadczenia z OSGi (pokrywają się one z doświadczeniami moich kolegów z branży) są takie, że nawet w przypadku dobrych zespołów łatwo doprowadzić do sytuacji, w której framework OSGi staje się znacznie większym źródłem złożoności niż gwarantem korzyści.

W języku Erlang zastosowano inne podejście — moduły są wbudowane w mechanizm wykonawczy (*runtime*) języka. Tak więc w języku Erlang zastosowano bardzo dojrzałe podejście do dekompozycji na moduły. Moduły Erlanga bez problemu można zatrzymać, zrestartować i zaktualizować. Erlang obsługuje nawet uruchamianie więcej niż jednej wersji modułu w danym momencie, co pozwala na łatwiejszą aktualizację modułów.

Możliwości modułów Erlanga są rzeczywiście imponujące, ale nawet jeśli mamy szczęście korzystać z platformy gwarantującej takie możliwości, nadal musimy znosić te same wady jak w przypadku zastosowania standardowej współdzielonej biblioteki. Jesteśmy ściśle ograniczeni w zdolności do korzystania z nowych technologii. Ograniczeni możliwościami niezależnego skalowania, sposobami dryfowania w kierunku technik integracji, które są nadmiernie sprzężone, i pozbawieni szwów spełniających rolę środków bezpieczeństwa dla architektury.

Mam jedno spostrzeżenie — według mnie godne uwagi. Z technicznego punktu widzenia powinno być możliwe tworzenie dobrze ukształtowanych, niezależnych modułów w obrębie pojedynczego monolitycznego procesu. Pomimo to rzadko można spotkać tego rodzaju architektury. Same moduły szybko stają się ściśle powiązane z pozostałą częścią kodu, co zaprzecza jednej z ich najważniejszych zalet. Uzyskanie separacji granic procesu egzekwuje czystą higienę pod tym względem (lub przynajmniej utrudnia stosowanie niewłaściwych praktyk). Oczywiście nie sugeruję, że to powinno być głównym motorem dla separacji procesów, ale interesujące jest to, że separacja modułów w granicach procesu jest rzadko spotykana w rzeczywistym świecie.

O ile zatem dekompozycja na moduły w granicach procesu może być czymś, co chcielibyśmy zrobić razem z dekompozycją systemu na usługi, o tyle sama dekompozycja na moduły nie pomoże rozwiązać wszystkich problemów. Jeśli programujemy wyłącznie w języku Erlang, jakość implementacji modułów w Erlangu może nam bardzo pomóc, ale nie podejrzewam, aby zbyt wielu Czytelników było w takiej sytuacji. Reszta z nas powinna postrzegać moduły jako mechanizmy oferujące te same korzyści co biblioteki współdzielone.

Nie istnieje panaceum na wszystko

Zanim zakończymy ten rozdział, powinienem podkreślić, że mikrousługi nie są „darmowym obiadem" czy „srebrną kulą" i nie sprawdzą się jako złoty młotek. Mają wszystkie zawiłości związane z systemami rozproszonymi i o ile dowiedzieliśmy się sporo o tym, jak należy właściwie zarządzać systemami rozproszonymi (które będziemy omawiać w całej książce), o tyle zarządzanie nimi jest pomimo wszystko trudne. Jeżeli nasze spojrzenie na systemy ewoluuje z perspektywy systemu monolitycznego, będziemy musieli znacznie więcej uwagi poświęcić zagadnieniom obsługi wdrażania, testowania i monitorowania tak, aby odblokować korzyści, o których opowiedzieliśmy do tej pory. Będziemy również musieli inaczej podejść do skalowania systemów i zapewnić ich odporność na awarie. Nie powinniśmy również być zaskoczeni koniecznością uwzględnienia takich elementów jak transakcje rozproszone lub twierdzenie CAP!

Każda firma, organizacja i system mają inną specyfikę. Na to, czy mikrousługi są właściwe w danej sytuacji oraz na jaką agresywność można sobie pozwolić podczas ich wdrażania, wpływa szereg czynników. W każdym rozdziale w tej książce spróbuję przekazać wskazówki podkreślające potencjalne pułapki. Wskazówki te powinny pomóc w wytyczeniu stabilnej ścieżki postępowania.

Podsumowanie

Mam nadzieję, że po lekturze tego rozdziału Czytelnik wie, czym są mikrousługi i co odróżnia je od innych technik kompozycji, a także jakie są niektóre ich główne atuty. W każdym z następnych rozdziałów będziemy prezentowali więcej szczegółów na temat możliwości wykorzystania tych atutów, a także powiemy, jak uniknąć niektórych typowych pułapek.

Jest szereg tematów do omówienia, ale trzeba od czegoś zacząć. Jednym z głównych wyzwań, które stawiają przed nami mikrousługi, jest zmiana roli tych, którzy często sterują ewolucją naszych systemów: architektów. W następnym rozdziale przyjrzymy się kilku różnym aspektom tej roli — takim, które pozwolą uzyskać z opisywanej nowej architektury jak najwięcej korzyści.

Skorowidz

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Ż

O'REILLY®

Przekonaj się, jak architektura mikrousług zmieni Twoje spojrzenie na aplikacje!

Architektura mikrousług to sposób na odejście od dużych, monolitycznych aplikacji. Wyspecjalizowane usługi realizujące konkretne zadania i komunikujące się z otoczeniem pozwalają na lepsze zapanowanie nad kodem, są łatwiejsze do przetestowania oraz bardziej elastyczne. Jednak oprócz zalet mają wady. Sięgnij po tę książkę i dowiedz się, jak najlepiej radzić sobie z architekturą mikrousług!

Autor przedstawia w książce skuteczne techniki projektowania i korzystania z architektury mikrousług. W trakcie lektury kolejnych rozdziałów poznasz w szczegółach ideę mikrousług, korzyści z ich stosowania, sposoby modelowania usług oraz skuteczne techniki dzielenia dużej aplikacji na mikrousługi. Ponadto zapoznasz się z możliwymi sposobami integracji: zdalne wywołanie procedur, REST i zdarzenia — to tylko niektóre z poruszanych kwestii. Na sam koniec zaznajomisz się z najlepszymi metodami testowania i monitorowania usług, zapewnisz im bezpieczeństwo dzięki kluczom API oraz innym technikom. Ta książka jest obowiązkową lekturą dla wszystkich osób chcących tworzyć nowoczesne systemy bazujące na architekturze mikrousług.

Sam Newman — technolog w firmie ThoughtWorks odpowiedzialny za wspomaganie klientów oraz architekturę wewnętrznych systemów. Prelegent, autor artykułów dla wydawnictwa O'Reilly. Programista języków Java oraz Python.

Dzięki tej książce:

- Poznasz możliwości i zalety architektury usług
- Wybierzesz odpowiedni sposób integracji
- Wdrożysz stworzoną aplikację
- Przygotujesz dokumentację, korzystając z narzędzia Swagger
- Zapewnisz bezpieczeństwo Twoim usługom

Helion

37819 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-283-1381-1

9 788328 313811

Informatyka w najlepszym wydaniu

cena: 59,00 zł