O'REILLY®

Mikrousługi

Wdrażanie i standaryzacja systemów w organizacji inżynierskiej

Helion

Susan J. Fowler

Tytuł oryginału: Production-Ready Microservices:
Building Standardized Systems Across an Engineering Organization

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-3682-7

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)

Spis treści

ROZDZIAŁ 1.

Mikrousługi

W ciągu ostatnich kilku lat branża technologiczna doświadczyła gwałtownych zmian w stosowanej praktycznej architekturze systemów rozproszonych. Zmiany te odwiodły gigantów branży (takich jak Netflix, Twitter, Amazon, eBay i Uber) od budowania monolitycznych aplikacji. Zamiast nich zaczęto stosować architekturę mikrousług. O ile podstawowe pojęcia dotyczące mikrousług nie są nowością, o tyle współczesne zastosowanie architektury mikrousług nią jest. Motywację do korzystania z takiej architektury częściowo tworzą wyzwania skalowalności, niewystarczająca efektywność systemów, wolne tempo rozwoju oraz trudności z przyjęciem nowych technologii powstające w przypadku próby objęcia i wdrożenia złożonych systemów oprogramowania w dużej monolitycznej aplikacji.

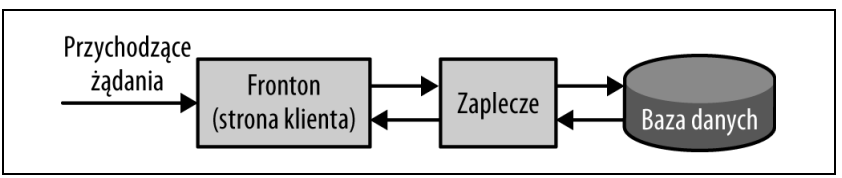
Przyjęcie architektury mikrousług — czy to od podstaw, czy przez podział istniejącej monolitycznej aplikacji na niezależnie rozwijane i wdrażane mikrousługi — rozwiązuje te problemy. Dzięki zastosowaniu architektury mikrousług aplikacja może być łatwo skalowana zarówno w poziomie, jak i w pionie, znacznie zwiększa się wydajność i szybkość jej rozwoju, a stare technologie łatwo mogą być wymieniane na nowsze.

Jak zobaczymy w tym rozdziale, zastosowanie architektury mikrousług można uznać za naturalny krok w skalowaniu aplikacji. Motywem do dzielenia monolitycznych aplikacji na mikrousługi są obawy dotyczące skalowalności i wydajności, ale mikrousługi wprowadzają swoje własne wyzwania. Udany, skalowalny ekosystem mikrousług wymaga obecności stabilnej i wyszukanej infrastruktury. Ponadto, aby wesprzeć architekturę mikrousług, należy radykalnie zmodyfikować strukturę organizacyjną firmy. Struktury zespołów, które są efektem takiej reorganizacji, mogą prowadzić do powstawania silosów

i nadmiernego rozrastania się. Największym wyzwaniem związanym z zastosowaniem architektury mikrousług jest jednak potrzeba standaryzacji architektury samych usług oraz określenie dla każdej mikrousługi wymagań dotyczących zaufania i dostępności.

Od monolitów do mikrousług

Prawie każde współcześnie napisane oprogramowanie można podzielić na trzy odrębne elementy: *fronton* (ang. *frontend*), czyli *stronę klienta*, *zaplecze* (ang. *backend*) oraz pewnego rodzaju magazyn danych (rysunek 1.1). Żądania są kierowane do aplikacji za pośrednictwem strony klienta, kod zaplecza realizuje główną pracę, a potrzebne dane, które muszą być składowane lub udostępniane (bez względu na to, czy tymczasowo w pamięci, czy trwale w bazie danych) są wysyłane lub pozyskiwane z miejsc, w których są przechowywane dane. Nazywamy to **architekturą trójwarstwową**.



Rysunek 1.1. Architektura trójwarstwowa

Istnieją trzy różne sposoby łączenia tych elementów w celu stworzenia aplikacji. W większości aplikacji pierwsze dwie części mają wspólną bazę kodu (lub repozytorium), w której jest przechowywany cały kod warstw strony klienta i zaplecza. Części te są uruchamiane jako jeden plik wykonywalny z oddzielną bazą danych. Innym sposobem jest rozdzielenie kodu frontonu od kodu zaplecza i zapisanie ich w postaci oddzielnych logicznie plików wykonywalnych, którym towarzyszy zewnętrzna bazy danych. W aplikacjach, które nie wymagają zewnętrznej bazy danych i przechowują wszystkie dane w pamięci, zwykle wszystkie trzy elementy są w jednym repozytorium. Niezależnie od sposobu, w jaki te elementy są podzielone lub połączone, sama *aplikacja* jest sumą tych trzech odrębnych elementów.

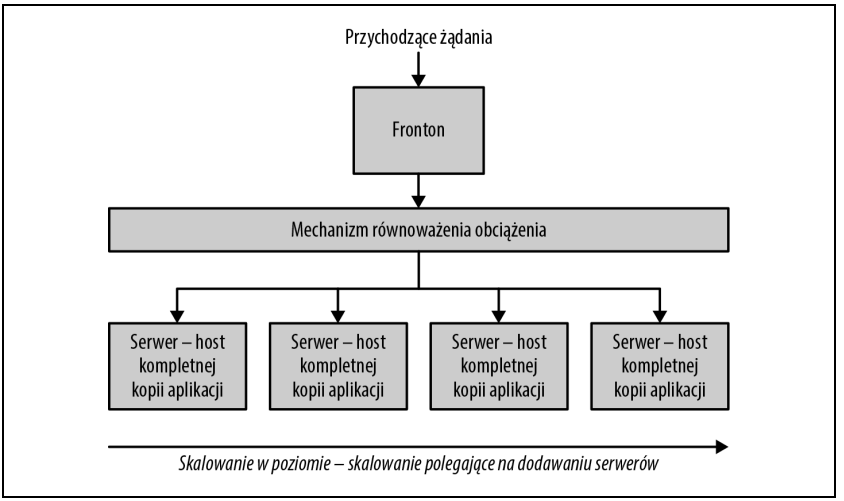
Aplikacje są zwykle zaprojektowane, zbudowane i uruchamiane w ten sposób od początku cyklu ich życia, a architektura aplikacji przeważnie jest niezależna od produktów oferowanych przez firmę bądź celu samej aplikacji. Wymienione trzy elementy, które składają się na architekturę trójwarstwową, są obecne w każdej witrynie internetowej, każdej aplikacji telefonicznej, każdym zapleczu i frontonie oraz dziwnych ogromnych aplikacjach korporacyjnych. Wszystkie one mają formę jednej z opisanych wcześniej permutacji.

Na początkowych etapach, gdy firma jest młoda, jej aplikacje są proste, a liczba programistów biorących udział w tworzeniu bazy kodu jest mała i zazwyczaj dzielą oni obowiązki związane z tworzeniem i utrzymywaniem bazy kodu. W miarę rozwoju firmy zatrudnianych jest więcej programistów, a do aplikacji dodawane są nowe funkcjonalności. Wtedy mogą wydarzyć się trzy istotne rzeczy.

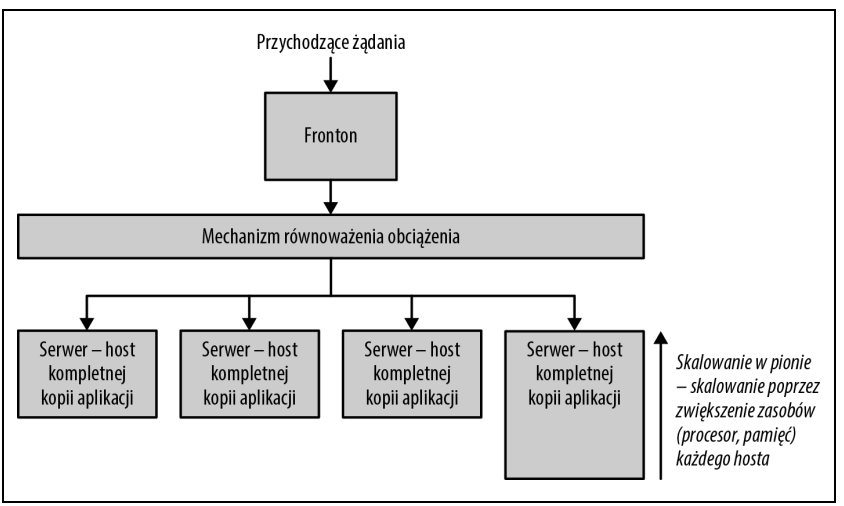
Pierwsza to obserwowany wzrost obciążenia operacyjnego. Praca operacyjna, ogólnie rzecz biorąc, obejmuje zadania związaną z uruchamianiem i utrzymaniem aplikacji. Wzrost ilości pracy operacyjnej zazwyczaj prowadzi do zatrudnienia inżynierów operacyjnych (administratorów systemów, inżynierów TechOps i tzw. inżynierów DevOps), którzy przejmują większość zadań operacyjnych — związanych ze sprzętem, monitorowaniem oraz obsługą wezwań.

Druga to rezultat prostej arytmetyki: dodawanie nowych funkcji do aplikacji zwiększa zarówno liczbę wierszy kodu w aplikacji, jak i jej złożoność.

Trzecia natomiast to niezbędne skalowanie aplikacji w poziomie i (lub) w pionie. Wzrost ruchu wymusza znaczną skalowalność i wydajność aplikacji, co pociąga za sobą potrzebę użycia większej liczby serwerów — hostów aplikacji. Po dodaniu większej liczby serwerów na każdym z nich instalowana jest kopia aplikacji. Instalowane są też mechanizmy równoważenia obciążenia tak, aby żądania były równomiernie rozkładane pomiędzy serwerami aplikacji (rysunek 1.2, przedstawiający fronton, który może zawierać własny mechanizm równoważenia obciążenia, mechanizm równoważenia obciążenia zaplecza oraz serwery zaplecza). Skalowanie w pionie staje się koniecznością, ponieważ aplikacja rozpoczyna przetwarzanie większej liczby zadań związanych z jej zróżnicowanym zestawem funkcji. W związku z tym aplikacja jest instalowana na większych, bardziej wydajnych serwerach, które mogą obsłużyć większe wymagania procesora i pamięci (rysunek 1.3).



Rysunek 1.2. Skalowanie aplikacji w poziomie



Rysunek 1.3. Skalowanie aplikacji w pionie

W miarę rozwoju firmy i wzrostu liczby inżynierów, gdy nie jest ona już jedno-, dwu- ani nawet trzycyfrowa, sprawy zaczynają się trochę bardziej komplikować. Z powodu wielu funkcji, łatek i poprawek dodawanych do bazy kodu przez programistów aplikacja rozrasta się do wielu tysięcy wierszy kodu. Złożoność aplikacji stale rośnie. Trzeba napisać setki (jeśli nie tysiące) testów,

aby sprawdzić, czy wprowadzone zmiany (polegające na modyfikacji choćby tylko jednej lub dwóch linijek) nie naruszyły integralności istniejących tysięcy wierszy kodu. Rozwijanie i wdrażanie aplikacji zmienia się w koszmar, testowanie staje się uciążliwe i jest przeszkodą dla wdrożenia nawet najbardziej istotnych poprawek, a dług techniczny szybko rośnie. Aplikacje, których cykl życia pasuje do tego wzorca (mniej lub bardziej), są w środowisku programistów czule (i adekwatnie) nazywane *monolitami*.

Oczywiście nie wszystkie monolityczne aplikacje są złe i nie każda monolityczna aplikacja sprawia wymienione problemy, ale monolity, które nie mają w którymś momencie swojego cyklu życia takich przypadłości, są (jak wynika z mojego doświadczenia) dość rzadkie. Powodem, dla którego większość monolitów jest podatna na te problemy, jest charakter takich aplikacji — natura monolitów jest zaprzeczeniem *skalowalności* w najbardziej ogólnym znaczeniu. Skalowalność wymaga *współbieżności* i *partycjonowania* — dwóch rzeczy, które są trudne do osiągnięcia w przypadku monolitu.

Skalowanie aplikacji

Spróbujmy przeprowadzić krótką analizę.

Celem każdego oprogramowania jest przetwarzanie pewnego rodzaju zadań. Bez względu na to, jakie są te zadania, można przyjąć ogólne założenia co do sposobu, w jaki aplikacja ma sobie z nimi radzić: musi je efektywnie wykonywać.

Aby zadania były skutecznie przetwarzane, aplikacja musi obsługiwać pewien rodzaj *współbieżności*. Oznacza to, że nie może to być tylko jeden proces, który wykonuje całą pracę. W takim przypadku ten proces wybierałby jedno zadanie, wykonywałby jego wszystkie niezbędne działania (lub nie!), a następnie przechodziłby do zadania następnego. Taki sposób przetwarzania jest bardzo niewydajny! Aby aplikacja była wydajna, należy zastosować współbieżność tak, aby każde zadanie mogło być podzielone na mniejsze fragmenty.

Aby wydajnie przetwarzać zadania, można także dzielić i zarządzać poprzez wprowadzenie *partycjonowania*. W tym przypadku każde zadanie jest nie tylko podzielone na małe fragmenty, ale również przetwarzane równolegle. Jeśli mamy kilka zadań, możemy przetwarzać je wszystkie w tym samym czasie, wysyłając je zbioru procesów roboczych, które mogą przetwarzać je równolegle. Aby przetwarzać więcej zadań, można z łatwością skalować rozwiązanie poprzez dodawanie procesów roboczych do przetwarzania nowych zadań bez wywierania wpływu na wydajność systemu.

Współbieżność i partycjonowanie są trudne do obsłużenia, gdy mamy jedną dużą aplikację, którą trzeba zainstalować na każdym serwerze i która musi przetwarzać wszelkiego rodzaju zadania. Jeśli aplikacja jest choćby odrobinę skomplikowana, jedynym sposobem na jej skalowanie w obliczu rosnącej listy funkcji i zwiększonego ruchu jest skalowanie sprzętu, na którym jest zainstalowana.

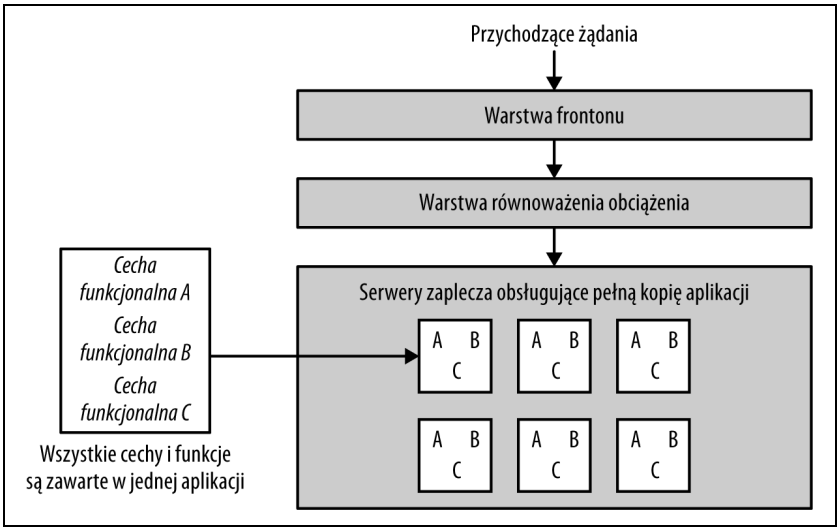
Najlepszym sposobem skalowania aplikacji w celu zapewnienia jej odpowiedniej wydajności jest podzielenie jej na wiele małych, niezależnych aplikacji, z których każda wykonuje jeden typ zadania. Potrzebny jest kolejny krok w procesie? To proste: wystarczy stworzyć nową aplikację, która wykonuje tylko ten krok! Trzeba poradzić sobie z większym obciążeniem? Żaden problem: dodajemy więcej procesów roboczych do każdej aplikacji!

Współbieżność i partycjonowanie są trudne do uzyskania w aplikacjach monolitycznych, przez co architektura monolitycznej aplikacji nie może być tak wydajna, jak tego oczekujemy.

Ten wzorzec pojawił się w takich firmach jak Amazon, Twitter, Netflix, eBay i Uber — firmach, które uruchamiają aplikacje nie na setkach, lecz na tysiącach, a nawet setkach tysięcy serwerów, i których aplikacje wyewoluowały do postaci monolitów i natknęły się na wyzwania skalowalności. Wyzwaniom, jakie napotkały te firmy, udało się sprostać dzięki porzuceniu architektury monolitycznej aplikacji na rzecz *mikrousług*.

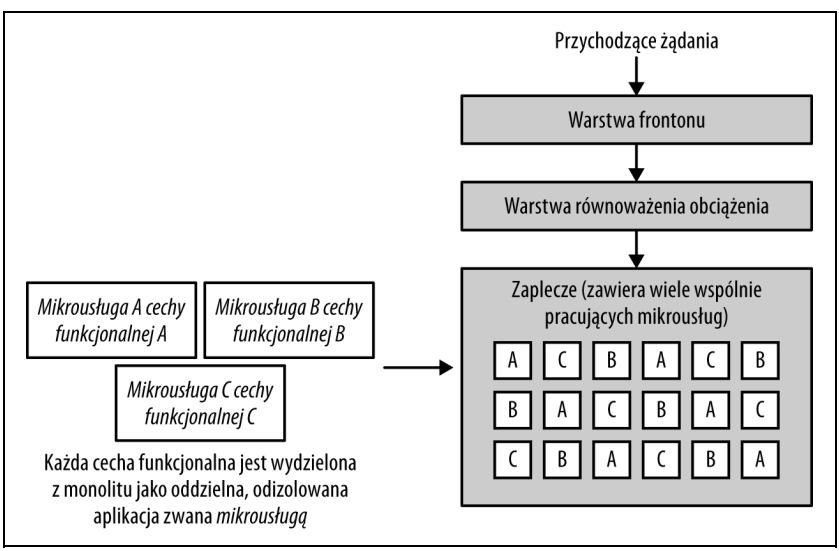
Podstawowe pojęcie mikrousługi jest proste: to mała aplikacja, która dobrze wykonuje tylko jedno działanie. Mikrousługa jest małym komponentem, który łatwo wymienić, można go niezależnie rozwijać i niezależnie instalować. Mikrousługa nie może jednak istnieć samodzielnie. Żadna mikrousługa nie jest wyspą. Jest to część większego systemu, uruchomionego i działającego razem z innymi mikrousługami dla osiągnięcia tego, co normalnie byłoby obsługiwane przez jedną dużą, samodzielną aplikację.

Celem architektury mikrousług jest stworzenie zbioru małych aplikacji, z których każda jest odpowiedzialna za wykonywanie jednej funkcji (w przeciwieństwie do tradycyjnego sposobu budowania jednej aplikacji, która robi wszystko), a także zapewnienie autonomii, niezależności i samodzielności każdej mikrousługi. Fundamentalna różnica między monolityczną aplikacją i mikrousługą jest następująca: monolityczna aplikacja (rysunek 1.4) zawiera wszystkie cechy i funkcje w jednej aplikacji i jednej bazie kodu, wszystkie są instalowane w tym samym czasie, każdy serwer jest hostem dla kompletnej kopii aplikacji, natomiast



Rysunek 1.4. Monolit

mikrousługa (rysunek 1.5) zawiera tylko jedną funkcję lub cechę i funkcjonuje w *ekosystemie mikrousług* wraz z innymi mikrousługami, z których każda realizuje tylko jedną funkcję lub cechę funkcjonalną.



Rysunek 1.5. Mikrousługi

Istnieje wiele korzyści z zastosowania architektury mikrousług — w tym (ale nie tylko) zmniejszenie długu technicznego, poprawa wydajności pracy programistów, lepsza wydajność testowania, zwiększona skalowalność i łatwość wdrażania. Firmy, które stosują architekturę mikrousług, zwykle decydują się na to po zbudowaniu jednej aplikacji i napotkaniu wyzwań skalowalności i problemów organizacyjnych. Zaczynają od aplikacji monolitycznej, a następnie *dzielą monolit* na mikrousługi.

Trudności z podziałem monolitu na mikrousługi całkowicie zależą od złożoności monolitycznej aplikacji. Podział monolitycznej aplikacji z wieloma funkcjami na mikrousługi wymaga sporo wysiłku architektonicznego i starannej analizy. Dodatkową komplikację stanowi potrzeba reorganizacji i restrukturyzacji zespołów. Decyzja o przejściu na mikrousługi zawsze musi stać się zadaniem dla całej firmy.

Rozbijanie monolitów na części wymaga kilku kroków. Pierwszym jest zidentyfikowanie komponentów, które należy napisać w postaci niezależnych usług. To chyba najtrudniejszy krok w całym procesie, ponieważ o ile może istnieć wiele prawidłowych sposobów podziału monolitu na komponenty, o tyle istnieje znacznie więcej sposobów nieprawidłowych. Prostą i praktyczną zasadą w identyfikacji komponentów jest wskazanie kluczowych funkcjonalności monolitu, a następnie podzielenie ich na małe, niezależne komponenty. Mikrousługi muszą być jak najprostsze. W przeciwnym razie firma będzie ryzykowała prawdopodobieństwo zastąpienia jednego monolitu kilkoma mniejszymi, a w miarę rozwoju firmy wszystkie one będą narażone na takie same problemy, jakie generowała pierwotna aplikacja.

Po zidentyfikowaniu kluczowych funkcji i prawidłowym przydzieleniu ich do niezależnych mikrousług należy przebudować strukturę organizacyjną firmy tak, aby każda mikrousługa była obsługiwana przez zespół inżynierów. Istnieje kilka sposobów, aby to zrobić. Pierwszą metodą reorganizacji firmy przy przyjęciu architektury mikrousług jest wyznaczenie dedykowanego zespołu do każdej mikrousługi. Wielkość zespołu w całości zależy od złożoności i obciążenia mikrousługi. W zespole powinna się znaleźć wystarczająca liczba programistów i inżynierów niezawodności witryny, aby można było właściwie obsłużyć zarówno rozwój funkcjonalności, jak i bieżącą rotację usługi. Drugą metodą jest przypisanie kilku usług jednemu zespołowi i powierzenie mu rozwijania usług równolegle. Ten sposób sprawdza się najlepiej w przypadku, gdy zespoły są zorganizowane wokół konkretnych produktów lub dziedzin biznesowych i są

odpowiedzialne za rozwijanie usług związanych z tymi produktami lub dziedzinami. Jeśli firma zdecyduje się na drugą metodę reorganizacji, powinna zadbać, aby programiści nie mieli nadmiernej ilości pracy i nie musieli zmagać się ze zmęczeniem zadaniem, awariami lub wypaleniem zawodowym.

Kolejnym ważnym elementem zastosowania mikrousług jest stworzenie *ekosystemu mikrousług*. Zazwyczaj firma obsługująca dużą aplikację monolityczną posiada dedykowaną organizację infrastruktury odpowiedzialnej za projektowanie, budowanie i utrzymanie środowiska, w którym działa aplikacja. Gdy monolit zostaje podzielony na mikrousługi, rośnie znaczenie obowiązków związanych z organizacją infrastruktury w celu dostarczenia stabilnej platformy dla rozwijania i utrzymania mikrousług. Zespoły infrastruktury muszą zapewnić zespołom mikrousług stałą infrastrukturę, która ukrywa większość złożonych interakcji pomiędzy mikrousługami.

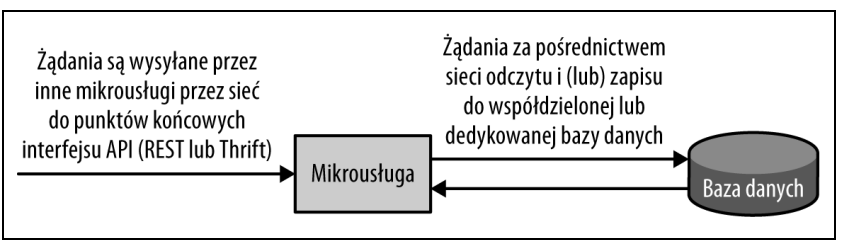
Po wykonaniu tych trzech kroków — podziale aplikacji na komponenty, restrukturyzacji zespołów inżynieryjnych w celu przydzielenia osób odpowiedzialnych za każdą mikrousługę oraz rozwinięciu organizacji infrastruktury w firmie — można rozpocząć migrację. Niektóre zespoły decydują się na przeniesienie właściwego kodu dla mikrousług bezpośrednio z monolitu do oddzielnej usługi i tworzą „cień" ruchu do monolitu, dopóki nie zyskają przekonania, że mikrousługa może samodzielnie realizować pożądane funkcjonalności. Inne zespoły decydują się na tworzenie usługi od podstaw — zaczynają od czystego konta i generują równoległy ruch monolitu lub przekierowują usługę po pomyślnym przejściu odpowiednich testów. Najlepsze podejście do migracji zależy od funkcjonalności mikrousługi. Z moich doświadczeń wynika, że w większości przypadków oba podejścia sprawdzają się tak samo dobrze. Jednak prawdziwym kluczem do pomyślnej migracji jest dokładny, szczegółowy i starannie udokumentowany plan oraz świadomość, że pełna migracja dużego monolitu może zająć kilka długich lat.

Biorąc pod uwagę ogrom pracy związanej z podziałem monolitu na mikrousługi, lepszą strategią może wydawać się rozpoczęcie od architektury mikrousług, pominięcie wszystkich wyzwań skalowalności i uniknięcie w ten sposób dramatu związanego z migracją do mikrousług. Takie podejście może okazać się dobre w przypadku niektórych firm. Chciałabym jednak wypowiedzieć kilka słów przestrogi. Otóż małe przedsiębiorstwa często nie mają niezbędnej infrastruktury do utrzymania mikrousług, nawet na bardzo małą skalę, a dobra architektura mikrousług wymaga stabilnej, często bardzo złożonej infrastruktury, ta zaś wymaga dużego, dedykowanego zespołu, którego koszty utrzymania są

akceptowalne tylko dla takich firm, które napotkały wyzwania skalowalności uzasadniające przejście na architekturę mikrousług. Małe firmy po prostu nie mają wystarczających możliwości operacyjnych do utrzymania ekosystemu mikrousług. Ponadto gdy firma jest we wczesnym stadium rozwoju, to niezwykle trudne jest zidentyfikowanie kluczowych obszarów i komponentów, które można wbudować w mikrousługi — aplikacje nowych firm nie mają zbyt wielu funkcji ani zbyt wielu oddzielnych obszarów funkcjonalności, które mogłyby zostać odpowiednio podzielone na mikrousługi.

Architektura mikrousług

Architektura mikrousług (rysunek 1.6) nie różni się zbytnio od architektury standardowej aplikacji opisanej w pierwszej części tego rozdziału (rysunek 1.1). Każda mikrousługa składa się z trzech komponentów: frontonu (strony klienta), jakiegoś kodu zaplecza odpowiedzialnego za zasadnicze zadania oraz sposobu przechowywania lub pobierania istotnych danych.



Rysunek 1.6. Elementy architektury mikrousług

Fronton — część mikrousługi działająca po stronie klienta — nie jest typową aplikacją z interfejsem użytkownika, lecz raczej interfejsem programowania aplikacji (API) ze statycznymi **punktami końcowymi** (ang. *endpoints*). Dobrze zaprojektowane interfejsy API mikrousługi umożliwiają łatwe i skuteczne komunikowanie się z mikrousługami oraz wysyłanie żądania do odpowiednich punktów końcowych API. Na przykład mikrousługa odpowiedzialna za dane klienta może mieć kilka punktów końcowych: *get_customer_information*, do którego inne usługi mogą wysyłać żądania w celu pobrania informacji o klientach, *update_customer_information*, do którego inne usługi wysyłają żądania w celu zaktualizowania informacji dla konkretnego klienta, oraz punkt końcowy *delete_customer_information*, którego usługi mogą używać do usuwania informacji na temat klienta.

Te punkty końcowe są wydzielone w architekturze i teoretycznie samodzielne. W praktyce współistnieją obok siebie w ramach wspólnego kodu zaplecza, który przetwarza każde żądanie. W naszym przykładzie mikrousługi odpowiedzialnej za dane klienta żądanie wysłane do punktu końcowego *get_customer_information* wygeneruje zadanie, które przetworzy przychodzące żądanie, określi specyficzne filtry lub opcje, które zostały zastosowane w żądaniu, pobierze informacje z bazy danych, sformatuje informacje, a następnie prześle je do klienta (mikrousługi), który tego zażądał.

Większość mikrousług przechowuje jakiś rodzaj danych — albo w pamięci (być może przy użyciu pamięci podręcznej), albo w zewnętrznej bazie danych. Jeśli dane są przechowywane w pamięci, nie jest konieczne połączenie sieciowe z zewnętrzną bazą danych, a mikrousługi mogą łatwo przekazać potrzebne dane do klienta. Jeśli dane są przechowywane w zewnętrznej bazie danych, mikrousługa jest zmuszona przekazać osobne żądanie do bazy danych, poczekać na odpowiedź, a następnie kontynuować przetwarzanie zadania.

Taka architektura jest konieczna, jeśli mikrousługi mają dobrze współpracować ze sobą. Paradygmat architektury mikrousług wymaga, aby zbiór mikrousług współpracował ze sobą w celu stworzenia czegoś, co w przeciwnym wypadku istniałoby jako jedna duża aplikacja. Stąd jeśli zestaw mikrousług ma współpracować ze sobą skutecznie i wydajnie, to są pewne elementy tej architektury, które muszą być znormalizowane w całej organizacji.

Punkty końcowe interfejsu API mikrousług powinny być ujednolicone w całej organizacji. Nie znaczy to, że wszystkie mikrousługi powinny mieć takie same konkretne punkty końcowe, ale że typ punktu końcowego powinien być taki sam. Dwoma popularnymi typami punktów końcowych API mikrousług są REST i Apache Thrift. Spotykałam mikrousługi korzystające z obu typów punktów końcowych (choć jest to rzadkie, komplikuje monitorowanie i, ogólnie rzez biorąc, tego sposobu nie polecam). Wybór typu punktu końcowego jest odzwierciedleniem wewnętrznego działania samej mikrousługi. Typ dyktuje także jej architekturę: trudno jest stworzyć asynchroniczną mikrousługę, która komunikuje się za pośrednictwem protokołu HTTP przez punkty końcowe REST, ponieważ taka usługa wymagałaby również dodania punktu końcowego bazującego na komunikatach.

Mikrousługi komunikują się ze sobą za pośrednictwem mechanizmu *zdalnego wywoływania procedur* (ang. *remote procedure call* — RPC). Są to wywołania za pośrednictwem sieci, zaprojektowane w taki sposób, aby wyglądały i działały

dokładnie tak, jak lokalne wywołania procedur. Stosowane protokoły zależą od wyborów architektonicznych i organizacyjnego wsparcia, a także od używanych punktów końcowych. Na przykład mikrousługa z punktami końcowymi REST prawdopodobnie będzie współdziałać z innymi mikrousługami za pośrednictwem protokołu HTTP, podczas gdy mikrousługa z punktami końcowymi Thrift może komunikować się z innymi mikrousługami przez HTTP lub za pomocą bardziej spersonalizowanych, wewnętrznych rozwiązań.

Unikaj wersjonowania mikrousług i punktów końcowych

Mikrousługa nie jest biblioteką (nie jest ładowana do pamięci w czasie kompilacji lub w czasie wykonywania programu), lecz niezależną aplikacją programową. Ze względu na szybkie tempo rozwoju mikrousług ich wersjonowanie może łatwo stać się koszmarem firmy. Może ono implikować sytuacje, w których programiści usług klienckich są związani używaniem konkretnych (nieaktualnych, nieutrzymywanych) wersji mikrousługi we własnym kodzie. Mikrousługi powinny być traktowane jako żywe, zmieniające się komponenty, a nie jak statyczne wydania bądź biblioteki. Wersjonowanie punktów końcowych API jest kolejnym antywzorcem, którego z tych samych powodów należy unikać.

Każdy typ punktu końcowego i każdy protokół używany do komunikacji z innymi mikrousługami ma swoje zalety i wady. Decyzje architektoniczne dotyczące mikrousług nie powinny być podejmowane przez indywidualnych programistów budujących mikrousługi, lecz powinny być częścią projektu architektury ekosystemu mikrousług jako całość (do tego tematu powrócimy w następnym podrozdziale).

Pisanie mikrousług daje programistom dużo swobody — poza wyborami organizacyjnymi dotyczącymi punktów końcowych API i protokołów komunikacyjnych programiści mają swobodę tworzenia wewnętrznej implementacji mikrousług. Mikrousługi mogą być zaimplementowane w dowolnym języku — w Go, Javie, Erlangu lub Haskellu. Istnieje tylko jeden warunek: programista musi wziąć pod uwagę punkty końcowe i protokoły komunikacyjne. Rozwijanie mikrousług nie różni się zbytnio od tworzenia autonomicznych aplikacji. Istnieją do tego pewne zastrzeżenia, o czym przekonamy się w ostatniej części niniejszego rozdziału („Wyzwania organizacyjne”). Swoboda programistów w zakresie wyboru języka wiąże się bowiem z ogromnymi kosztami ponoszonymi przez organizację.

Mikrousługa może być traktowana jako czarna skrzynka: wprowadzamy do niej pewne informacje poprzez wysyłanie żądań do punktów końcowych i otrzymujemy wyniki. Jeśli to, czego chcemy i potrzebujemy, otrzymamy od mikrousługi w rozsądnym czasie i bez absurdalnych błędów, to znaczy, że spełniła swoje zadanie i nie trzeba rozumieć niczego więcej poza znajomością punktów końcowych i oceną poprawności działania usługi.

Na tym kończy się opis specyfiki architektury mikrousług — nie dlatego, że temat został wyczerpany, ale ponieważ każdy z kolejnych rozdziałów książki został poświęcony opisom sposobu doprowadzenia mikrousług do doskonałego stanu czarnej skrzynki.

Ekosystem mikrousług

Mikrousługi nie istnieją w izolacji. Środowisko, w którym mikrousługi są budowane i uruchamiane i w którym odbywają się interakcje między nimi, to miejsce, gdzie one *żyją*. Złożoność wielkoskalowego środowiska mikrousług można porównać do zawiłości ekologicznych tropikalnego lasu, pustyni lub oceanu. Uznanie tego środowiska za ekosystem — *ekosystem mikrousług* — pomaga w przyjęciu architektury mikrousług.

W dobrze zaprojektowanych, zrównoważonych ekosystemach mikrousługi są wyabstrahowane z infrastruktury. Są oddzielone od sprzętu, sieci, kompilacji i potoku wdrożeń, a także mechanizmu wykrywania usług i równoważenia obciążenia. To wszystko jest częścią infrastruktury ekosystemu mikrousług, a budowanie, standaryzacja i utrzymywanie tej infrastruktury w stanie stabilnym, skalowalnym, odpornym na awarie i niezawodnym ma kluczowe znaczenie dla sukcesu działania mikrousług.

Infrastruktura musi podtrzymywać ekosystem mikrousług. Celem wszystkich inżynierów infrastruktury i architektów powinno być wyeliminowanie konieczności uwzględniania niskopoziomowych szczegółów działania infrastruktury podczas rozwijania mikrousług. Powinni oni dążyć do zbudowania stabilnej infrastruktury, którą można skalować — takiej, na której bazie programiści mogą łatwo zbudować i uruchomić mikrousługi. Rozwijanie mikrousługach w ramach stabilnego ekosystemu mikrousług powinno przypominać tworzenie niewielkich, samodzielnych aplikacji. Wymaga to bardzo wyrafinowanej, najwyższej klasy infrastruktury.

Ekosystem mikrousług można podzielić na cztery warstwy (rysunek 1.7), choć granice każdej z tych warstw nie zawsze są jasno zdefiniowane, niektóre elementy infrastruktury dotykają bowiem wszystkich części stosu. Trzy niższe warstwy to warstwy infrastruktury: na dole stosu znajduje się warstwa sprzętu, a nad nią warstwa komunikacji (która przenika do czwartej warstwy) oraz platforma aplikacji. Pojedyncze mikrousługi mieszczą się w czwartej warstwie (najwyższej).

Rysunek 1.7. Czterowarstwowy model ekosystemu mikrousług

Warstwa 1.: sprzęt

Na samym dole ekosystemu mikrousług znajduje się *warstwa sprzętu*. Są to rzeczywiste maszyny: materialne, fizyczne komputery, na których działają wszystkie narzędzia i mikrousługi. Serwery te są zamontowane na półkach w centrach danych, chłodzone za pomocą drogich systemów HVAC i zasilane energią elektryczną. To serwery wielu różnych typów: niektóre są zoptymalizowane pod kątem baz danych, natomiast inne — dla zadań intensywnie wymagających procesora. Mogą być własnością firmy lub być wynajęte od dostawców chmury, takich jak Amazon Web Services' Elastic Compute Cloud (AWS EC2), Google Cloud Platform (GCP) lub Microsoft Azure.

Wybór konkretnego sprzętu jest zdeterminowany tym, kim są właściciele serwerów. Jeśli nasza firma prowadzi własne centra danych, wybór sprzętu należy do nas. Możemy zoptymalizować wybór serwerów zgodnie ze specyficznymi potrzebami. Jeśli korzystamy z serwerów w chmurze (co jest najczęściej spotykanym scenariuszem), wybór jest ograniczony do tego, co oferuje dostawca usług w chmurze. Wybór pomiędzy tzw. *gołym metalem* a *dostawcą w chmurze* (lub dostawcami) nie jest łatwą decyzją. Trzeba wziąć pod uwagę koszty, dostępność, niezawodność i wydatki operacyjne.

Zarządzanie serwerami jest częścią warstwy sprzętu. Każdy serwer musi mieć zainstalowany *system operacyjny*, który powinien być ustandaryzowany na wszystkich serwerach. Nie istnieje jedyna słuszna odpowiedź na pytanie o system operacyjny, którego należy użyć dla ekosystemu mikrousług. Odpowiedź na to pytanie zależy całkowicie od budowanej aplikacji, języków, w jakich będzie ona zapisana, oraz bibliotek i narzędzi wymaganych przez mikrousługi. Większość ekosystemów mikrousług działa na jakiejś wersji Linuxa. Zazwyczaj jest to CentOS, Debian lub Ubuntu, choć firmy posługujące się platformą .NET oczywiście dokonają innego wyboru. Nad warstwą sprzętu mogą być umieszczone warstwy złożone z dodatkowych abstrakcji: izolacja i abstrakcja zasobów (w postaci oferowanej przez takie technologie jak Docker i Apache Mesos) również należą do tej warstwy, podobnie jak bazy danych (dedykowane lub współdzielone).

Mechanizmy instalowania systemu operacyjnego i *konfigurowania* (ang. *provisioning*) sprzętu należą do pierwszej warstwy nad samymi serwerami. Każdy host musi być przygotowany i skonfigurowany, a po zainstalowaniu systemu operacyjnego należy wykorzystać *narzędzie do zarządzania konfiguracją* (na przykład Ansible, Chef lub Puppet) w celu zainstalowania wszystkich aplikacji i wprowadzenia wszystkich koniecznych ustawień konfiguracyjnych.

Hosty wymagają właściwych mechanizmów *monitorowania* (na przykład za pomocą takich narzędzi jak Nagios) oraz *rejestrowania*. Dzięki tym mechanizmom, jeśli cokolwiek się wydarzy (awaria dysku bądź sieci lub znaczny wzrost wykorzystania procesora), problemy będzie można łatwo zdiagnozować, złagodzić i rozwiązać. Mechanizmy monitorowania na poziomie hosta szczegółowo omówiłam w rozdziale 6., „Monitorowanie".

Warstwa 1.: sprzęt — podsumowanie

Warstwa sprzętu ekosystemu mikrousług obejmuje:

- serwery fizyczne
  (własność firmy lub wynajęte od dostawców usługw chmurze);
- bazy danych (dedykowane lub współdzielone);
- system operacyjny;
- mechanizmy izolacji i abstrakcji zasobów;
- zarządzanie konfiguracją;
- monitorowanie na poziomie hosta;
- rejestrowanie na poziomie hosta.

Warstwa 2.: komunikacja

Druga warstwa ekosystemu mikrousług to *warstwa komunikacji*. Warstwa komunikacji przenika do wszystkich innych warstw ekosystemu (w tym platformy aplikacji i warstwy mikrousług), ponieważ to ona obsługuje całą komunikację między usługami. Granice pomiędzy warstwą komunikacji a wszystkimi innymi warstwami ekosystemu mikrousług są słabo zdefiniowane. O ile granice nie są zbyt wyraźne, o tyle elementy *są* wyraźne: druga warstwa ekosystemu mikrousług zawsze zawiera punkty końcowe sieci, DNS, RPC i API, a także wykrywanie usług, rejestr usług i równoważenie obciążenia.

Omawianie elementów sieci i DNS warstwy komunikacji wykracza poza zakres tej książki. Dlatego tutaj skoncentrujemy się na punktach końcowych RPC i API, wykrywaniu usługi, rejestrze usług i równoważeniu obciążenia.

RPC, punkty końcowe i przesyłanie komunikatów

Mikrousługi komunikują się ze sobą przez sieć za pomocą *zdalnych wywołań procedury* (RPC) lub *przesyłania komunikatów* (ang. *messaging*) do punktów końcowych API innych mikrousług (lub, w przypadku przesyłania komunikatów, do brokera komunikatów, który kieruje komunikat do odpowiedniego adresata). Podstawowa idea jest prosta: za pomocą określonego protokołu mikrousługa wysyła dane w standardowym formacie przez sieć do innej usługi (być może do punktu końcowego API innej mikrousługi) albo do brokera komunikatów, który dba o przesłanie danych do punktu końcowego API innej mikrousługi.

Istnieje kilka paradygmatów komunikacji mikrousług. Pierwszy z nich, *HTTP+REST/THRIFT*, jest najczęściej spotykany. W przypadku paradygmatu HTTP+REST/THRIFT usługi komunikują się ze sobą przez sieć za pomocą protokołu HTTP (ang. *Hypertext Transfer Protocol*). Wysyłają żądania do specyficznych punktów końcowych REST (od ang. *representational state transfer*) — za pomocą różnych metod, np. `GET`, `POST` itp. — lub specyficznych punktów końcowych *Apache Thrift* (albo jednych i drugich) i odbierają z nich odpowiedzi. Dane są zwykle formatowane i wysyłane jako *JSON* (lub jako tzw. *bufory protokołu* — ang. *protocol buffers*) przez protokół HTTP.

HTTP + REST to najbardziej dogodna forma komunikacji mikrousług. Nie ma dla niej żadnych niespodzianek, jest łatwa w konfiguracji oraz najbardziej stabilna i niezawodna — głównie dlatego, że trudno ją nieprawidłowo zaimplementować. Minusem przyjęcia tego paradygmatu jest to, że metoda ta jest z konieczności synchroniczna (blokująca).

Drugim paradygmatem komunikacyjnym jest *przesyłanie komunikatów* (ang. *messaging*). Metoda ta jest asynchroniczna (nieblokująca), ale nieco bardziej skomplikowana. Mechanizm przesyłania komunikatów działa w następujący sposób: mikrousługa wysyła dane (*komunikat*) przez sieć (HTTP lub za pomocą innego protokołu) do *brokera komunikatów*, który kieruje komunikację do innych mikrousług.

Mechanizm przesyłania komunikatów występuje w kilku odmianach. Dwie najbardziej popularne to *publikuj-subskrybuj* (*pubsub*) i *żądanie-odpowiedź* (ang. *request-response*). W modelu pubsub klienty dokonują *subskrypcji tematu* (ang. *topic*) i otrzymują wiadomość za każdym razem, gdy *wydawca opublikuje* wiadomość w ramach tego tematu. Modele żądanie-odpowiedź są prostsze: klient wysyła *żądanie* do usługi (lub *brokera komunikatów*) i otrzymuje *odpowiedź* zawierającą żądane informacje. Istnieją technologie przesyłania komunikatów, które są unikatową mieszanką obu modeli. Przykładem może być Apache Kafka. Dla mikrousług napisanych w Pythonie można wykorzystać systemy przesyłania komunikatów (i przetwarzania zadań) Celery i Redis (lub Celery z RabbitMQ): Celery przetwarza zadania i (lub) komunikaty, wykorzystując w roli brokera systemy Redis lub RabbitMQ.

Mechanizm przesyłania komunikatów ma kilka istotnych wad, które trzeba złagodzić. Rozwiązania bazujące na przesyłaniu komunikatów, jeśli są od początku projektowane z myślą o skalowalności, mogą być w takim samym stopniu skalowalne (albo nawet bardziej) jak te, które bazują na HTTP + REST. Ze swojej natury mechanizmy przesyłania komunikatów nie są tak samo łatwe do zmiany i aktualizacji, a ich scentralizowany charakter (choć może się wydawać korzystny) może prowadzić do powstawania kolejek. Brokery stają się natomiast punktami awarii dla całego ekosystemu. Jeśli nie zostaną zastosowane odpowiednie środki zaradcze, to asynchroniczny charakter mechanizmu przesyłania komunikatów może prowadzić do wyścigów i nieskończonych pętli. Jeżeli mechanizm przesyłania komunikatów zostanie zaimplementowany wraz z zabezpieczeniami przeciwko tym problemom, to może osiągnąć taką samą stabilność i wydajność jak rozwiązanie synchroniczne.

Wykrywanie usługi, rejestr usług i równoważenie obciążenia

W architekturze monolitycznej ruch musi być wysyłany tylko do jednej aplikacji i odpowiednio dystrybuowany do serwerów będących hostami aplikacji. W architekturze mikrousług ruch musi być odpowiednio kierowany do dużej liczby różnych aplikacji, a następnie odpowiednio rozprowadzany do serwerów obsługujących każdą konkretną mikrousługę. Aby można było sprawnie i skutecznie to zrealizować, architektura mikrousług wymaga zaimplementowania w warstwie komunikacji trzech technologii: *wykrywania usługi*, *rejestru usług* i *równoważenia obciążenia*.

Ogólnie rzecz biorąc, gdy mikrousługa A chce skierować żądanie do mikrousługi B, to musi znać adres IP i port egzemplarza, na którym działa mikrousługa B. Dokładniej mówiąc, warstwa komunikacji pomiędzy mikrousługami musi znać adresy IP i porty mikrousług, tak aby mogła odpowiednio kierować żądania. Ta funkcjonalność jest realizowana przez mechanizm wykrywania usługi (np. etcd, Consul, Hyperbahn lub ZooKeeper), który dba o to, aby żądania były kierowane dokładnie tam, gdzie powinny być wysyłane, oraz aby (co bardzo ważne) były kierowane tylko do egzemplarzy o dobrej kondycji. Mechanizm wykrywania usługi potrzebuje *rejestru usług*, który jest bazą danych portów i adresów IP wszystkich mikrousług w ekosystemie.

Dynamiczne skalowanie i przypisane porty

W architekturze mikrousług porty i adresy IP mogą się zmieniać (i zmieniają się) cały czas — zwłaszcza w przypadku skalowania i instalowania mikrousług (w szczególności z wykorzystaniem warstwy abstrakcji sprzętu, na przykład Apache Mesos). Jednym ze sposobów podejścia do zadań wykrywania i routingu jest przypisanie do każdej mikrousługi statycznych portów (zarówno w warstwie frontonu, jak i zaplecza).

Jeśli wszystkie mikrousługi nie występują wyłącznie w jednym egzemplarzu (co jest bardzo mało prawdopodobne), to w różnych częściach warstwy komunikacji ekosystemu mikrousług potrzebne są mechanizmy *równoważenia obciążenia*. Na bardzo ogólnym poziomie równoważenie obciążenia działa w następujący sposób: jeśli mikrousługa jest zainstalowana na dziesięciu różnych egzemplarzach, to oprogramowanie równoważenia obciążenia (i/lub sprzętu) dba o dystrybucję ruchu pomiędzy wszystkimi egzemplarzami.

Równoważenie obciążenia sieciowego jest potrzebne w każdym miejscu ekosystemu, w którym wysyłane jest żądanie do aplikacji. To oznacza, że każdy duży ekosystem mikrousług jest złożony z wielu warstw równoważenia obciążenia. Powszechnie używanymi do tego celu systemami równoważenia obciążenia są Amazon Web Services Elastic Load Balancer, Netflix Eureka, HAProxy i Nginx.

Warstwa 2.: komunikacja — podsumowanie

Warstwa komunikacji ekosystemu mikrousług obejmuje:

- sieć;
- DNS;
- zdalne wywołania procedur (RPC);
- punkty końcowe;
- przesyłanie komunikatów;
- wykrywanie usług;
- rejestr usług;
- równoważenie obciążenia.

Warstwa 3.: platforma aplikacji

Platforma aplikacji jest trzecią warstwą ekosystemu mikrousług. Zawiera wszystkie wewnętrzne narzędzia i usługi, które są niezależne od mikrousług. Ta warstwa jest wypełniona scentralizowanymi, działającymi w całym ekosystemie narzędziami i usługami, które muszą być budowane w taki sposób, aby zespoły rozwoju mikrousług nie były zmuszone do projektowania, budowania lub utrzymywania niczego poza własną mikrousługą.

Dobra platforma aplikacji jest wyposażona w *samoobsługowe* wewnętrzne narzędzia dla programistów, standardowy proces wytwarzania oprogramowania, scentralizowany i zautomatyzowany *system kompilacji i publikowania*, *automatyczne testowanie*, ustandaryzowane i scentralizowane *rozwiązanie wdrażania* oraz scentralizowany *mechanizm rejestrowania* i *monitorowania na poziomie mikrousługi*. Wiele szczegółów dotyczących wymienionych elementów omówiłam w kolejnych rozdziałach. Tutaj opiszę zwięźle kilka z nich, aby zaprezentować wprowadzenie podstawowych pojęć.

Samoobsługowe wewnętrzne narzędzia programistyczne

Do kategorii *samoobsługowych wewnętrznych narzędzi programistycznych* można zakwalifikować sporo elementów. Przydział składników, które należą do tej kategorii, zależy nie tylko od potrzeb programistów, ale również od poziomu abstrakcji zarówno infrastruktury, jak i ekosystemu jako całości. Kluczem do określenia narzędzi, które muszą zostać zbudowane, jest w pierwszej kolejności podzielenie stref odpowiedzialności, a następnie określenie zadań, które programiści powinni móc zrealizować w celu zaprojektowania, zbudowania i utrzymania swoich usług.

W firmie, która przyjęła architekturę mikrousług, obowiązki powinny być uważnie oddelegowane do różnych zespołów inżynierskich. Łatwym sposobem osiągnięcia tego celu jest stworzenie podorganizacji inżynierskiej dla każdej warstwy ekosystemu mikrousług, wraz z innymi zespołami, które tworzą pomosty dla poszczególnych warstw. Każda z tych organizacji inżynierskich, funkcjonując quasi-niezależnie, jest odpowiedzialna za wszystko, co dzieje się w jej warstwie: zespoły TechOps są odpowiedzialne za warstwę 1., zespoły infrastruktury są odpowiedzialne za warstwę 2., zespoły platformy aplikacji są odpowiedzialne za warstwę 3., natomiast zespoły mikrousług są odpowiedzialne za warstwę 4. (to jest oczywiście bardzo uproszczony widok, ale pozwala zaprezentować ogólną koncepcję).

W ramach tego schematu organizacyjnego, za każdym razem, gdy inżynier pracujący w którejś z wyższych warstw musi stworzyć, skonfigurować lub wykorzystać jakiś element z niższej warstwy, powinien mieć możliwość skorzystania z samoobsługowego narzędzia. Na przykład zespół pracujący nad mechanizmem przesyłania komunikatów dla ekosystemu powinien zbudować samoobsługowe narzędzie, które może być wykorzystane przez programistę zespołu mikrousługi do łatwego skonfigurowania mechanizmu komunikatów dla swojej usługi bez konieczności rozumienia wszystkich zawiłości systemu obsługi komunikatów.

Istnieje wiele powodów, dla których warto dysponować scentralizowanymi, samoobsługowymi narzędziami dla każdej warstwy. W zróżnicowanym ekosystemie mikrousług przeciętny inżynier w danym zespole nie posiada wiedzy (lub ma bardzo niewielką wiedzę) na temat sposobu działania usług i systemów w innych zespołach. Nie jest możliwe, aby każdy inżynier mógł stać się ekspertem w każdej usłudze i w każdym systemie w czasie, gdy pracuje nad własnymi. Poszczególni programiści będą wiedzieli bardzo niewiele poza swoimi

własnymi usługami, ale razem, wszyscy programiści pracujący w ramach ekosystemu, będą kolektywnie wiedzieć wszystko. Zamiast próby edukowania każdego programisty w zakresie zawiłości każdego narzędzia i każdej usługi w ekosystemie, lepiej zbudować trwałe, łatwe w użyciu interfejsy użytkownika dla każdej części ekosystemu, a następnie przeszkolić innych ze sposobu korzystania z nich. Należy zamienić wszystkie narzędzia w czarne skrzynki, a następnie dokładnie udokumentować, w jaki sposób działają i jak należy ich używać.

Drugim powodem dobrego budowania tych narzędzi jest fakt, że nie chcemy dopuścić do tego, aby programista z innego zespołu mógł wprowadzić znaczące zmiany do usługi lub systemu — zwłaszcza jeśli te zmiany mogłyby spowodować awarię. Jest to szczególnie prawdziwe i istotne dla usług i systemów należących do niższych warstw (warstwa 1., warstwa 2. i warstwa 3.). Zezwolenie nieekspertom na wprowadzanie zmian w obrębie tych warstw lub wymaganie (albo, co gorsza, żądanie) od nich, by stali się ekspertami w tych dziedzinach, to przepis na katastrofę. Jednym z obszarów, w których może dojść do nieprzewidzianych sytuacji, jest zarządzanie konfiguracją: zezwolenie programistom zespołów mikrousług na wprowadzanie zmian w konfiguracji systemu bez odpowiedniej wiedzy fachowej może — w przypadku, gdy zmiana wpływa na inny element niż ich własna usługa — doprowadzić do przestojów produkcyjnych na dużą skalę.

Cykl rozwoju

Gdy programiści wprowadzają zmiany w istniejących mikrousługach lub tworzą nowe mikrousługi, warto zadbać o bardziej skuteczny proces rozwoju poprzez uproszczenie i ujednolicenie procesu oraz zautomatyzowanie jak największej liczby jego elementów. Szczegóły standaryzacji stabilnego i niezawodnego procesu rozwoju omówiłam w rozdziale 4., „Skalowalność i wydajność”. Jest jednak kilka elementów, które powinny się znaleźć w trzeciej warstwie ekosystemu mikrousług, aby stabilny i niezawodny rozwój był możliwy.

Pierwszym wymaganiem jest *scentralizowany system kontroli wersji*, w którym jest przechowywany, śledzony, wersjonowany i przeszukiwany cały kod. Zazwyczaj funkcjonalność ta jest realizowana za pomocą systemu podobnego do GitHub albo samodzielnego repozytorium git lub svn powiązanego z pewnego rodzaju narzędziem współpracy takim jak Phabrictor. Wykorzystanie tych narzędzi pozwala na łatwe utrzymywanie i przeglądnie kodu.

Drugie wymaganie to stabilne, wydajne *środowisko programistyczne*. Środowiska programistyczne w ekosystemach mikrousług są bardzo trudne do zaimplementowania ze względu na skomplikowane zależności każdej z mikrousług od innych usług, ale są absolutnie niezbędne. Niektóre organizacje inżynierskie preferują, aby wszystkie zadania programistyczne były realizowane lokalnie (na laptopie programisty), ale to może prowadzić do niepoprawnych instalacji, ponieważ programista nie uzyskuje dokładnego obrazu działania jego zmian w kodzie w środowisku produkcyjnym. Najbardziej stabilnym i niezawodnym sposobem zaprojektowania środowiska programistycznego jest stworzenie lustrzanej kopii środowiska produkcyjnego (takiego, które nie jest środowiskiem przedprodukcyjnym, kanarkowym ani produkcyjnym) uwzględniającej wszystkie skomplikowane łańcuchy zależności.

Testowanie, kompilowanie, pakietowanie i wydawanie

Etapy *testowania*, *kompilowania*, *pakietowania* i *wydawania* występujące pomiędzy fazami rozwoju i wdrażania powinny być w maksymalnym stopniu ustandaryzowane i scentralizowane. Po zakończeniu cyklu rozwoju i zaewidencjonowaniu wszystkich zmian w kodzie należy uruchomić wszystkie niezbędne testy oraz automatycznie zbudować i utworzyć pakiety dla wszystkich nowych wydań. Dokładnie do tego celu służą *narzędzia ciągłej integracji*, a gotowe rozwiązania (takie jak Jenkins) są bardzo zaawansowane i proste w konfiguracji. Narzędzia te ułatwiają automatyzację całego procesu, pozostawiając bardzo mało miejsca na ludzki błąd.

Potok wdrożeń

Potok wdrożeń (ang. *deployment pipeline*) jest procesem, w ramach którego nowy kod trafia na serwery produkcyjne po przeprowadzeniu cyklu rozwoju oraz następujących po nim krokach testowania, kompilowania, pakietowania i wydawania. Wdrożenia w ekosystemie mikrousług, w którym setki wdrożeń dziennie nie są niczym niezwykłym, mogą się bardzo szybko skomplikować. Zbudowanie oprzyrządowania wdrażania oraz standaryzacja praktyk wdrażania dla wszystkich zespołów programistycznych nierzadko są koniecznością. Zasady budowy stabilnych i niezawodnych (gotowych do produkcji) potoków wdrożeń omówiłam szczegółowo w rozdziale 3., „Stabilność i niezawodność”.

Rejestrowanie i monitorowanie

Każda mikrousługa powinna być wyposażona w mechanizm rejestrowania na poziomie mikrousługi wszystkich żądań kierowanych do mikrousługi (włącznie z wszystkimi ważnymi informacjami) oraz odpowiedzi udzielanych na te żądania. Ze względu na naturę programowania mikrousług — szybkie tempo — często nie jest możliwe odtworzenie błędów w kodzie, ponieważ nie ma możliwości odtworzenia stanu systemu w momencie wystąpienia awarii. Dobry mechanizm rejestrowania na poziomie mikrousług dostarcza programistom informacji, których potrzebują do pełnego zrozumienia stanu usługi w określonym czasie w przeszłości lub teraźniejszości. Mechanizm *monitorowania na poziomie mikrousługi* wszystkich *kluczowych charakterystyk* mikrousług jest niezbędny z podobnych powodów: dokładne, przeprowadzane w czasie rzeczywistym monitorowanie pozwala programistom zapoznać się z kondycją i stanem ich usługi. Mechanizmy rejestrowania i monitorowania na poziomie mikrousługi omówiłam bardziej szczegółowo w rozdziale 6., „Monitorowanie".

Warstwa 3.: platforma aplikacji — podsumowanie

Warstwa platformy aplikacji ekosystemu mikrousług obejmuje:

- samoobsługowe wewnętrzne narzędzia programistyczne;
- środowisko programistyczne;
- narzędzia do testowania, kompilowania, pakietowaniai i wydawania;
- potok wdrożeń;
- rejestrowanie na poziomie mikrousługi;
- monitorowanie na poziomie mikrousługi.

Warstwa 4.: mikrousługi

Na samej górze ekosystemu mikrousług jest *warstwa mikrousług*. W tej warstwie mieszczą się mikrousługi oraz wszystko, co jest z nimi związane. Są one całkowicie wyabstrahowane od niższych warstw infrastruktury. Są oddzielone od sprzętu, wdrażania, wykrywania usług, równoważenia obciążenia i komunikacji. Od warstwy mikrousług nie są oddzielone jedynie konfiguracje specyficzne dla każdej usługi, umożliwiające posługiwanie się narzędziami.

Powszechną praktyką w inżynierii oprogramowania jest centralizacja wszystkich konfiguracji aplikacji, tak aby konfiguracje dla konkretnego narzędzia lub zestawu narzędzi (np. zarządzanie konfiguracją, izolacja zasobów lub narzędzia wdrażania) były przechowywane razem z narzędziem. Na przykład niestandardowe konfiguracje wdrażania aplikacji często są przechowywane nie razem z kodem aplikacji, lecz z kodem narzędzia wdrażania. Praktyka ta dobrze się sprawdza dla architektury monolitycznej, a nawet dla małych ekosystemów mikrousług, ale w rozbudowanych ekosystemach mikrousług złożonych z setek mikrousług i dziesiątek wewnętrznych narzędzi (z których każde ma oddzielną niestandardową konfigurację) praktyka ta prowadzi raczej do bałaganu: programiści zespołów mikrousług muszą wprowadzać zmiany w bazach kodu narzędzi w niższych warstwach, a czasami zapominają, gdzie są przechowywane określone konfiguracje (lub czy w ogóle one istnieją). W celu złagodzenia tego problemu wszystkie konfiguracje specyficzne dla mikrousług mogą być przechowywane w repozytorium mikrousług i powinny być dostępne dla narzędzi i systemów z warstw znajdujących się niżej.

Warstwa 4.: mikrousługi — podsumowanie

Warstwa mikrousług ekosystemu mikrousług obejmuje:

- mikrousługi;
- wszystkie konfiguracje specyficzne dla mikrousługi.

Wyzwania organizacyjne

Zastosowanie architektury mikrousług rozwiązuje najpilniejsze wyzwania charakterystyczne dla architektury aplikacji monolitycznych. Mikrousług nie dotyczą wyzwania skalowalności, braku wydajności bądź trudności w przyjęciu nowych technologii, są one bowiem zoptymalizowane pod kątem skalowalności, wydajności oraz szybkości pracy programisty. W branży, w której nowe technologie szybko trafiają na rynek, czysto organizacyjne koszty utrzymywania i usprawniania skomplikowanych aplikacji monolitycznych są po prostu niepotrzebne. Biorąc to pod uwagę, trudno sobie wyobrazić powody, dla których ktoś miałby być niechętny podzieleniu monolitu na mikrousługi albo mieć obiekcje co do budowania ekosystemu mikrousług od podstaw.

Mikrousługi wydają się rozwiązaniem magicznym (i w pewnym sensie oczywistym), ale my wiemy o nich więcej. W książce *The Mythical Man-Month* Frederick Brooks wyjaśnił, dlaczego nie istnieją „srebrne kule" w inżynierii oprogramowania. Koncepcję tę podsumował w następujący sposób: „Nie istnieje żaden proces rozwoju ani technologia czy technika zarządzania, które same w sobie obiecują poprawę — w wydajności, niezawodności i prostocie — choćby o jeden rząd wielkości na dekadę".

Kiedy ktoś zaprezentuje nam technologię, która obiecuje drastyczne ulepszenia, powinniśmy zwrócić uwagę na kompromisy, z jakimi wiąże się jej zastosowanie. Zastosowanie mikrousługi oferuje większą skalowalność i wydajność. Powinniśmy jednak wiedzieć, że cechy te są związane z kosztami dla pewnej części całego systemu.

Istnieją cztery szczególnie ważne kompromisy wynikające z zastosowania architektury mikrousług. Pierwszym jest zmiana w strukturze organizacyjnej, która zmierza do izolacji i słabej komunikacji pomiędzy zespołami — jest to konsekwencja odwróconego *prawa Conwaya*. Drugi to dramatyczny rozrost technologiczny. Jest on niezwykle kosztowny nie tylko dla całej organizacji, ale również wnosi znaczne koszty dla każdego inżyniera. Trzecim kompromisem jest zwiększone *ryzyko awarii całego systemu*. Czwarty kompromis to *rywalizacja o zasoby infrastruktury i zasoby inżynieryjne*.

Odwrócone prawo Conwaya

Koncepcja *prawa Conwaya* (nazwa pochodzi od nazwiska programisty Melvina Conwaya) sformułowanego w 1968 roku jest następująca: architektura systemu jest określona przez strukturę komunikacyjną i organizacyjną firmy. *Odwrócone prawo Conwaya* również jest prawdziwe i w szczególności odnosi się do ekosystemu mikrousług: struktura organizacyjna firmy jest określona przez architekturę jej produktu. Ponad 40 lat po opublikowaniu prawa Conwaya zarówno ono, jak i jego odwrotność nadal wydają się prawdziwe. Struktura organizacyjna firmy Microsoft — gdybyśmy spróbowali naszkicować ją w formie architektury systemu — przypomina architekturę jej produktów. To samo dotyczy Google'a, Amazona i wszystkich innych dużych firm technologicznych. Firmy, które przyjmują architekturę mikrousług, nie są wyjątkiem od tej reguły.

Architektura mikrousług składa się z dużej liczby małych, odizolowanych od siebie, niezależnych mikrousług. Odwrócone prawo Conwaya wymaga, aby struktura organizacyjna dowolnej firmy stosującej architekturę mikrousług składała się z dużej liczby bardzo małych, odizolowanych od siebie i niezależnych zespołów. Struktury zespołu, które się z tego wywodzą, nieuchronnie prowadzą do powstawania silosów i rozrastania się. Problemy stają się coraz większe w miarę wzrostu zaawansowania, współbieżności i wydajności ekosystemu mikrousług.

Odwrócone prawo Conwaya oznacza również, że programiści pod pewnymi względami będą przypominali mikrousługi: będą zdolni do wykonywania jednej rzeczy i (miejmy nadzieję) będą ją wykonywać bardzo dobrze, ale będą odseparowani (pod względem odpowiedzialności, wiedzy dziedzinowej i doświadczenia) od reszty ekosystemu. Łącznie wszyscy programiści *wspólnie* działający w ramach ekosystemu mikrousług wiedzą wszystko, co trzeba wiedzieć na temat tego ekosystemu, ale indywidualnie są bardzo wyspecjalizowani — znają tylko fragmenty ekosystemu, za które są odpowiedzialni.

Stwarza to nieunikniony problem organizacyjny: mimo że mikrousługi muszą być tworzone w izolacji (co prowadzi do powstania odizolowanych, autonomicznych zespołów), to nie funkcjonują w izolacji, a jeśli produkt ma dobrze działać, muszą ze sobą współpracować. To wymaga, aby te odizolowane, niezależnie działające zespoły współpracowały ze sobą ściśle i często. Jest to trudne do osiągnięcia, zważywszy na fakt, że cele i projekty większości zespołów (skodyfikowane w ich celach i kluczowych rezultatach — ang. *objectives and key results*, OKR) są specyficzne dla mikrousługi, nad którą zespół pracuje.

Istnieje również olbrzymia luka komunikacyjna pomiędzy zespołami mikrousług a zespołami infrastruktury. Trzeba zadbać o jej wypełnienie. Na przykład zespoły platformy aplikacji powinny budować usługi i narzędzia platformy, które będą używane przez wszystkie zespoły mikrousług, ale zebranie wymagań i potrzeb od setek zespołów mikrousług przed stworzeniem jednego niewielkiego projektu może potrwać miesiące (a nawet lata). Nakłonienie zespołów programistów i zespołów infrastruktury do wspólnej pracy nie jest łatwym zadaniem.

Istnieje pokrewny problem wynikający z odwróconego prawa Conwaya, który rzadko występuje w firmach stosujących architekturę monolityczną, a mianowicie trudność działania organizacji operacyjnych. W przypadku monolitu można z łatwością obsadzić personel organizacji operacyjnej i ustanowić dyżury

pracy personelu z aplikacją. Jest to natomiast bardzo trudne do osiągnięcia w architekturze mikrousług, ponieważ wymaga przydzielenia do każdej mikrousługi zarówno zespołu programistycznego, *jak i* zespołu operacyjnego. W konsekwencji zespoły programistów mikrousług muszą być odpowiedzialne za realizację obowiązków operacyjnych i zadań powiązanych z ich mikrousługami. Nie ma oddzielnej organizacji operacyjnej, która mogłaby przyjmować zgłoszenia, nie ma również dedykowanych inżynierów operacyjnych odpowiedzialnych za monitorowanie — zgłoszenia dotyczące swoich usług muszą przyjmować sami programiści.

Techniczny rozrost

Drugi minus, *techniczny rozrost* (ang. *technical sprawl*), jest powiązany z pierwszym. O ile prawo Conwaya i jego odwrócona wersja przewidują rozrost organizacyjny i tworzenie silosów mikrousług, o tyle drugi rodzaj rozrostu (związany z technologiami, narzędziami i tym podobnymi elementami) także jest nieunikniony w architekturze mikrousług. Techniczny rozrost może przejawiać się na wiele różnych sposobów. W tym rozdziale opiszę kilka najczęstszych.

Przyczyny, dla których stosowanie architektury mikrousług prowadzi do technicznego rozrostu, są łatwe do zauważenia, jeśli przyjrzymy się dużemu ekosystemowi składającemu się z tysiąca mikrousług. Załóżmy, że każdą z tych mikrousług obsługuje zespół złożony z sześciu programistów, a każdy programista używa swojego własnego zestawu ulubionych narzędzi i bibliotek i pracuje w swoich własnych ulubionych językach programowania. Każdy z tych zespołów programistycznych stosuje własny sposób wdrażania, własne parametry monitorowania i alertów, własne biblioteki zewnętrzne i zależności wewnętrzne. Wykorzystuje spersonalizowane skrypty do uruchamiania na komputerach produkcyjnych. I tak dalej.

Jeśli tych zespołów jest kilka tysięcy, oznacza to, że w ramach jednego systemu stosowanych jest tysiące sposobów na wykonanie jednej rzeczy. Istnieje tysiąc sposobów na wdrażanie, tysiąc bibliotek do utrzymania, tysiąc różnych sposobów ostrzegania, monitorowania, testowania i obsługiwania awarii. Jedynym sposobem na zmniejszenie rozrostu technicznego jest standaryzacja na wszystkich poziomach ekosystemu mikrousług.

Inny rodzaj technicznego rozrostu jest związany z wyborem języka. Mikrousługi słyną z obietnicy większej swobody dla programistów — swobody wyboru dowolnych języków i bibliotek. Ogólnie rzecz biorąc, jest to możliwe i może

być stosowane w praktyce, ale w miarę rozwoju ekosystemu mikrousług często staje się niepraktyczne, kosztowne i niebezpieczne. Aby zobaczyć, dlaczego może to stać się problemem, rozważmy następujący scenariusz. Załóżmy, że mamy ekosystem mikrousług zawierający dwieście usług. Wyobraźmy sobie, że niektóre z tych mikrousług są napisane w Pythonie, inne w JavaScripcie, jeszcze inne w Haskellu, kilka innych w Go, a jeszcze kilka innych w Ruby, Javie i C++. Dla każdego wewnętrznego narzędzia, dla każdego systemu i dla każdej usługi w ramach każdej warstwy ekosystemu trzeba napisać biblioteki dla każdego z tych języków.

Zastanówmy się przez chwilę, ile pracy związanej z utrzymaniem i programowaniem trzeba wykonać dla każdego z tych języków, aby zapewnić wymagany poziom wsparcia. Jest jej ogromnie dużo. Tylko nieliczne organizacje inżynierskie mogą sobie pozwolić na przydzielenie zasobów umożliwiających sprostanie tym zadaniom. Bardziej realistyczny od obsługi dużej liczby języków jest wybór ich niewielkiej liczby i zadbanie o to, aby wszystkie biblioteki i narzędzia były dostępne i zgodne z tymi językami.

Ostatnim rodzajem technicznego rozrostu, jaki omówię w tym rozdziale, jest dług techniczny, który zwykle odnosi się do pracy, jaką trzeba wykonać ze względu na to, że pewną pracę wykonano tak, aby zrobić ją szybko, ale nie w najlepszy i najbardziej optymalny sposób. Zważywszy na to, że zespoły programowania mikrousług publikują nowe funkcjonalności w szybkim tempie, dług techniczny często narasta niepostrzeżenie w tle. Gdy pojawiają się przestoje z powodu awarii, wtedy prace wykonywane w rezultacie przeprowadzanych analiz incydentu rzadko są rozwiązaniem optymalnym — zespoły programowania mikrousług zazwyczaj stosują poprawki, które rozwiązują problem szybko i są wystarczająco dobre w danej chwili. Wszelkie lepsze rozwiązania są odkładane na później.

Większe ryzyko awarii

Mikrousługi są dużymi, złożonymi, rozproszonymi systemami składającymi się z wielu małych, niezależnych fragmentów, które stale się zmieniają. Realia pracy ze złożonymi systemami tego rodzaju są takie, że pojedyncze komponenty ulegają awariom. Awarie komponentów są częste i przebiegają w sposób, którego nikt nie potrafi przewidzieć. Tu objawia się trzecia wada architektury mikrousług: większe ryzyko awarii systemów.

Istnieją sposoby przygotowania się do awarii, łagodzenia błędów, gdy one wystąpią, i testowania limitów i granic zarówno poszczególnych komponentów, jak i ekosystemu jako całości. Opiszę je w rozdziale 5., „Odporność na awarie i przygotowanie na katastrofy". Warto jednak zapamiętać, że bez względu na to, jak wiele testów odporności uruchomimy, niezależnie od liczby scenariuszy błędów i katastrof, które weźmiemy pod uwagę, nie możemy zaprzeczyć faktowi, że system kiedyś *ulegnie* awarii. Jedyne, co można zrobić, to jak najlepiej przygotować się na tę ewentualność.

Rywalizacja o zasoby

Podobnie jak w innych ekosystemach w świecie przyrody, w ekosystemie mikrousług występuje ostra konkurencja o zasoby. Każda organizacja inżynieryjna ma ograniczone zasoby: skończoną liczbę zasobów inżynierskich (zespoły, programiści) oraz skończone zasoby sprzętu i infrastruktury (fizycznych komputerów, sprzętu w chmurze, przestrzeni w bazach danych itd.), a każdy z zasobów kosztuje firmę mnóstwo pieniędzy.

Gdy ekosystem mikrousług zawiera dużą liczbę mikrousług i wiele rozbudowanych i skomplikowanych platform aplikacji, rywalizacja pomiędzy zespołami o zasoby sprzętu i infrastruktury staje się nieunikniona: każda usługa i każde narzędzie będą prezentowane jako równie ważne. Ich skalowanie będzie przedstawiane jako to, które ma najwyższy priorytet.

Na podobnej zasadzie, gdy zespoły platformy aplikacji proszą o specyfikacje i potrzeby od zespołów mikrousług, aby mogły właściwie zaprojektować swoje systemy i narzędzia, każdy zespół programowania mikrousług twierdzi, że jego potrzeby są najważniejsze, i będzie rozczarowany (i potencjalnie bardzo sfrustrowany), jeśli jego potrzeby nie zostaną uwzględnione. Tego rodzaju rywalizacja o zasoby inżynierskie może prowadzić do niechęci między zespołami.

Ostatni typ rywalizacji o zasoby jest najbardziej oczywisty: rywalizacja między menedżerami, zespołami oraz różnymi działami inżynieryjnymi (organizacjami) o inżynierów. Pomimo wzrostu liczby absolwentów kierunków informatycznych oraz dostępności licznych kursów programowania (tzw. bootcampów) znalezienie naprawdę dobrych programistów jest bardzo trudne. Dobrzy programiści należą do najbardziej niezastąpionych i ograniczonych zasobów. Gdy istnieją setki lub tysiące zespołów, w których mogą pracować nieliczni nowi inżynierowie, wtedy każdy zespół będzie się upierał, że to on potrzebuje dodatkowych inżynierów bardziej niż inne zespoły.

Nie ma sposobu, aby rywalizacji o zasoby uniknąć, choć istnieją sposoby, aby nieco ją złagodzić. Najbardziej skuteczne wydaje się organizowanie lub kategoryzowanie zespołów pod względem ich znaczenia i ważności dla ogólnej działalności oraz udzielenie zespołom dostępu do zasobów na podstawie ich priorytetu lub znaczenia. Istnieją wady takiego podejścia, ponieważ jego stosowanie może powodować niewystarczające dbanie o obsadę zespołów pracujących nad narzędziami programistycznymi oraz porzucanie projektów, których znaczenie polega na kształtowaniu przyszłości (np. przyjęcie nowych technologii w infrastrukturze).

Skorowidz

N

O

P

R

S

Ś

T

W

Z

Notatki

O'REILLY®

Wdrażaj mikrousługi w najlepszym sprawdzonym standardzie!

W odróżnieniu od aplikacji monolitycznych systemy oparte na mikrousługach są bardziej skalowalne, efektywniejsze, a także łatwiejsze w implementacji, rozwijaniu i utrzymaniu. Dzięki zastosowaniu architektury mikrousługi kontenerów programiści skupiają się jedynie na elemencie, a nie na całości aplikacji. Jednak przejście na architekturę mikrousług nie jest akie proste. Problemem jest brak gotowych standardów architektonicznych, operacyjnych i organizacyjnych, które ułatwiałyby pełne wykorzystanie niewątpliwych zalet architektury mikrousług.

Oto praktyczny poradnik dla inżynierów, menedżerów i architektów oprogramowania odpowiedzialnych za przygotowanie i funkcjonowanie systemów w firmach inżynierskich. Wyjaśniono tu niezbędne pojęcia i pokazano zasady budowania mikrousług. Opisano również szereg strategii ich implementacji. Docenisz także podejście oparte na standaryzacji: dzięki temu łatwiej jest zaprojektować mikrousługi, które są stabilne, niezawodne, skalowalne, odporne na uszkodzenia, wydajne, monitorowane i udokumentowane.

Najważniejsze zagadnienia:

- mikrousługi, ich budowa i ekosystem
- standardy gotowości bazujące na dostępności mikrousług
- standardy projektowania cyklu życiowego mikrousług
- budowa odporności na awarie i strategie wykrywania błędów
- konserwacja systemu i strategie zarządzania zależnościami

Susan J. Fowler – pracuje w firmie Uber Technologies, gdzie zajmuje się przygotowaniem mikrousług do wdrożenia do produkcji. Przed dołączeniem do Ubera pracowała nad platformami aplikacji i infrastrukturą w kilku małych firmach. Wcześniej studiowała fizykę cząstek elementarnych na Uniwersytecie Pensylwanii. Interesuje się nie tylko technologiami informatycznymi i fizyką, ale też matematyką oraz filozofią. Jednym z jej marzeń jest napisanie symfonii i... scenariusza sitcomu.

ISBN 978-83-283-3682-7

9 788328 336827

Informatyka w najlepszym wydaniu

cena: 49,00 zł