O'REILLY®

Mikrousługi
Budowa i działanie

Przewodnik po budowaniu
architektury mikrousług

Ronnie Mitra,
Irakli Nadareishvili

Mikrousługi
Budowa i działanie

Przewodnik po budowaniu architektury mikrousług

Ronnie Mitra,
Irakli Nadareishvili,

przekład: Marek Włodarz

APN Promise
Warszawa 2021

O'REILLY®

Mikrousługi. Budowa i działanie

Authorized translation of English edition of
Microservices Up and Running
ISBN 978-1-492-07545-5

APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa
tel. +48 22 35 51 600, fax +48 22 35 51 699
e-mail: mspress@promise.pl

ISBN: 978-83-7541-443-1 (wyd. drukowane), 978-83-7541-447-9 (ebook)

Projekt okładki: Karen Montgomery
Ilustracje: Kate Dullea

Przekład: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Spis treści

Przedmowa

Dziesięć lat temu grupa architektów oprogramowania zebrała się razem i ukuła termin *mikrousługi*, aby zdefiniować nowy, wyłaniający się w toku ewolucji styl architektury oprogramowania. Od tego czasu nastąpiła prawdziwa eksplozja wykładów, filmów wideo i pisanych prac na temat stylu mikrousług. W istocie w roku 2016 sami wspólnie napisaliśmy *Microservice Architecture*, książkę stanowiącą wstępny przewodnik po zasadach systemu mikrousług.

Od publikacji tej książki mieliśmy możliwość pracy i życia wśród budowanych przez nas (i wielu innych) systemów mikrousług. Nasze własne doświadczenia, a także dyskusje z innymi praktykami, doprowadziły nas do lepszego poznania praktycznych problemów, przed którymi stają ludzie implementujący to podejście. Wiele tej wiedzy pochodzi z wdrożeń zakończonych sukcesami, ale niektóre z najbardziej użytecznych nauk wyciągnęliśmy z pomyłek i niepowodzeń.

Staraliśmy się opakować nasze doświadczenia jako praktyków w postaci wysoce arbitralnego przewodnika. Żyjemy w czasach, w których dostępne jest bogactwo praktycznych porad. Jednak żeglowanie po tym morzu informacji i zebranie ich w coś, co będzie działało, nie jest łatwe. Ta książka oferuje praktyczny, nakazowy model, który rozciąga się na obszary projektowania zespołów, domeny, infrastruktury, inżynierii i wydawania. Naszym celem jest danie czytelnikom zunifikowanego obrazu implementowania mikrousług i solidnego punktu startowego na drodze do adaptacji tej architektury.

Kto powinien przeczytać tę książkę

Książkę napisaliśmy z myślą o osobach implementujących mikrousługi. Choć odwołujemy się do niektórych zasad i wzorców systemu mikrousług, książka skupia się na praktycznym projektowaniu i inżynierii. Jeśli ktoś jest architektem oprogramowania lub inżynierem stojącym przed zadaniem budowania mikrousług albo całej architektury mikrousług, zdecydowanie powinien sięgnąć po tę książkę.

Jednak książka może być również użytecznym przewodnikiem dla czytelników, którzy chcieliby po prostu „zapoznać się osobiście" z implementacją mikrousług. Niezależnie od odgrywanej roli, jeśli ktoś jest zainteresowany pracą, która prowadzi do budowania systemu mikrousług, książka powinna okazać się bardzo przydatna.

Co będzie potrzebne

Ponieważ zakres mikrousług jest bardzo obszerny, będziemy używali wielu różnych narzędzi i metod. Jeśli ktoś chce samodzielnie wykonać wszystkie przedstawione przykłady, będzie potrzebował zainstalować albo uruchomić subskrypcję następujących narzędzi i platform:

- Docker
- Redis
- MySQL
- GitHub
- GitHub Actions
- Terraform
- Amazon Web Services
- kubectl
- Helm
- Argo CD

Informacje, gdzie znaleźć lub jak uzyskać dostęp do tych narzędzi, zamieściliśmy w treści książki podczas ich omawiania.

Konwencje używane w tej książce

W książce tej zostały użyte następujące konwencje typograficzne:

Kursywa
: Sygnalizuje nowe pojęcia, adresy URL, adresy email, nazwy plików oraz rozszerzenia nazw.

`Stała szerokość`
: Używane jest w listingach programów, a także w treści przy odwoływaniu się do elementów programowych, takich jak nazwy zmiennych lub funkcji, baz danych, typów danych, zmiennych środowiskowych, instrukcji i słów kluczowych.

`Stała szerokość z pogrubieniem`
: Przedstawia polecenia lub inny tekst, który powinien zostać dokładnie wpisany przez użytkownika.

`Stała szerokość z kursywą`
: Przedstawia tekst, który powinien być zastąpiony przez wartości podawane przez użytkownika lub wartości wynikające z kontekstu, a także komentarze w przykładach kodu.

Ten element oznacza wskazówkę lub sugestię.

Ten element oznacza uwagę ogólną.

Ten element wskazuje ostrzeżenie lub przestrogę.

Korzystanie w przykładów kodu

Materiał uzupełniający (przykłady kodu, ćwiczenia itp.) można pobrać ze strony dostępnej pod adresem *https://oreil.ly/MicroservicesUpandRunning*.

W przypadku pytań technicznych lub problemów przy korzystaniu z przykładów kodu można przesyłać je na adres *bookquestions@oreilly.com*.

Książka ta ma na celu pomóc w wykonaniu pracy. W ogólności, jeśli zawiera ona jakiś przykład kodu, można go użyć w swoich programach i dokumentacji. Nie ma potrzeby kontaktowania się z nami o zezwolenie, o ile ktoś nie reprodukuje znaczącej części kodu. Dla przykładu napisanie programu, który używa wielu fragmentów kodu z tej książki, nie wymaga zezwolenia. Sprzedaż lub rozpowszechnianie przykładów z książki wydawnictwa O'Reilly wymaga takiego zezwolenia. Odpowiadanie na pytanie poprzez zacytowanie tej książki i dołączenie przykładowego kodu nie wymaga zezwolenia. Włączenie znaczącej części kodu przykładowego z tej książki do dokumentacji swojego produkty wymaga zezwolenia.

Doceniamy, ale w ogólności nie wymagamy wskazania źródła. Atrybucja taka zazwyczaj zawiera tytuł, autora(ów), wydawcę oraz numer ISBN. Na przykład: „*Microservices: Up and Running* by Ronnie Mitra and Irakli Nadareishvili (O'Reilly). Copyright 2021 Mitra Pandey Consulting, Ltd. and Irakli Nadareishvili, 978-1-492-07545-5."

Jeśli ktoś uważa, że jego wykorzystanie przykładów kodu wykracza poza granice przedstawione powyżej, prosimy o kontakt pod adresem *permissions@oreilly.com*.

Jak się z nami skontaktować

Zachęcamy do przesyłania komentarzy i pytań dotyczących niniejszej książki do wydawcy:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (w USA lub Kanadzie)
707-829-0515 (międzynarodowo lub lokalnie)
707-829-0104 (faks)

Strona internetowa dla tej książki, na której zamieszczamy erratę, przykłady i inne informacje dodatkowe, dostępna jest pod adresem *https://oreil.ly/Microservices_Up_and_Running*.

Wyślij e-mail na adres *bookquestions@oreilly.com*, aby skomentować lub zadać pytania techniczne dotyczące tej książki.

Aby poznać wiadomości i informacje o naszych książkach i kursach, odwiedź *http://oreilly.com*.

Znajdź nas na Facebooku: *http://facebook.com/oreilly*
Śledź nas na Twitterze: *http://twitter.com/oreillymedia*
Oglądaj nas na YouTube: *http://youtube.com/oreillymedia*

Podziękowania

Chcielibyśmy podziękować naszym redaktorkom Melissie Potter i Deborah Baker oraz całemu zespołowi redakcyjnemu w wydawnictwie O'Reilly, bez których nigdy nie zdołalibyśmy ukończyć tej książki. Chcemy również przekazać podziękowania dla Pete'a Hodgsona, Chrisa O'Della, Lorindy Brandon, JP Morgenthala, Mike'a Amundsena i Davida Butlanda za przekazane przez nich uwagi i spostrzeżenia. Na koniec chcemy podziękować firmom Capital One i Publicis Sapient za wsparcie, którego udzieliły nam w naszych staraniach urzeczywistnienia tej książki.

Rozdział 1

W stronę architektury mikrousług

Celem tej książki jest zapewnienie pomocy w budowaniu działającej architektury mikrousług. Na jej stronach Czytelnik znajdzie przekonujące i zdecydowane wskazówki na temat budowania oprogramowania. Porady te pochodzą z zebranych przez nas doświadczeń rzeczywistych praktyków, dotyczących zarówno udanych implementacji, jak i takich, które mogłyby pójść lepiej. Nauki te dopracowaliśmy do postaci modelu, który – mamy nadzieję – pozwoli szybciej zbudować własny, działający system.

W ostatnim czasie popularność opartego na mikrousługach stylu budowania oprogramowania gwałtownie wzrosła. Termin *mikrousługi*[1] wyłonił się na początku drugiej dekady XXI wieku jako sposób opisania nowego stylu architektury oprogramowania. Aplikacje budowane w tym nowo nazwanym stylu składają się z niewielkich, niezależnych komponentów, które współpracują ze sobą. Od tego czasu liczba zastosowań stylu mikrousług po prostu eksplodowała. Startupy, wielkie przedsiębiorstwa i wszystkie inne firmy mieszczące się pomiędzy tymi skrajnościami uczyły się i wdrażały architektury w stylu mikrousług. Rosnący ekosystem narzędzi, usług i rozwiązań w tym obszarze jest naturalnym skutkiem tej rozszerzającej się popularności. W chwili pisania tych słów badanie Allied Market Research[2] przewidywało, że globalny rynek architektury mikrousług wzrośnie do 8,07 miliarda dolarów w roku 2026, z bieżącej wartości 2 miliardów. Liczby tego poziomu sygnalizują wielkie zainteresowanie, wiele adaptacji i mnóstwo, mnóstwo działających mikrousług.

Dla wielu budowanie oprogramowania w stylu mikrousług okazuje się być poważnym wyzwaniem. Prawda jest taka, że implementowanie systemu mikrousług nie jest łatwe. Sprawienie, aby wiele niezależnych części działało wspólnie, jest trudniejsze, niż mogłoby się wydawać. Koszty zarządzania, konserwacji, wsparcia i testowania rosną. Przy pewnej skali koszty te mogą stać się prohibicyjne. Jeśli nie zachowamy ostrożności, trudy zarządzania systemem mogą spowodować, że mikrousługi będą się wydawać kiepskim pomysłem.

1 W polskiej literaturze informatycznej równie często występuje termin *mikroserwisy*, jednak zdecydowałem się na stosowanie terminu *mikrousługi*, jako lepiej osadzonego w duchu języka polskiego (przyp. tłum.).

2 *https://oreil.ly/cugsz*

Jednak korzyści wynikające z budowania mikrousług sprawiają, że warto podjąć to ryzyko. Dobrze wykonane mikrousługi pozwalają na szybsze i bezpieczniejsze wprowadzanie zmian w oprogramowaniu w wielkiej skali. Szybsze i bezpieczniejsze zmiany oznaczają większą elastyczność naszej firmy. Ta *zwinność* przekłada się na lepsze wyniki naszego biznesu i organizacji.

Sztuczką, która pozwoli odblokować te wszystkie wartości, jest dysponowanie odpowiednią architekturą obsługującą te usługi. Musi ona redukować koszty systemowe bez zmniejszania wartości niezależnych usług. Aby zbudować taką architekturę, konieczne jest wczesne podjęcie ważnych decyzji. Wybory te rozciągają się na metody działania, procesy, zespoły, technologie i narzędzia. One również muszą współdziałać ze sobą, aby utworzyć wynikową, zoptymalizowaną całość.

Dobrą metodą budowania takiego systemu jest przyjęcie podejścia ewolucyjnego. Możemy rozpocząć od kilku drobnych decyzji, po czym uczyć się i rozwijać w trakcie pracy. W istocie większość wczesnych implementatorów doszło do mikrousług poprzez iteracyjne eksperymentowanie. Nie stawiali sobie celu zbudowania aplikacji opartej na mikrousługach. Zamiast tego doszli do tej architektury poprzez ciągły proces optymalizacji i ulepszeń.

Zaczynanie od zera i wykonywanie kolejnych iteracji zajmuje czas. Dobra wiadomość jest taka, że możemy wykorzystać doświadczenia tych praktyków jako pomoc w szybszym budowaniu naszego systemu. Naszą budowę rozpoczniemy na fundamencie wzorców, metod i narzędzi, które były już wspólnie wykorzystywane z powodzeniem. Następnie możemy zoptymalizować system, aby zapewnić zgodność z unikatowymi celami i ograniczeniami naszej organizacji.

W tej książce udokumentowaliśmy decyzje, które tworzą solidny fundament architektury mikrousług. Zanim jednak będziemy mogli zagłębić się w szczegóły tego modelu, musimy odpowiedzieć na jedno ważne pytanie: Co dokładnie rozumiemy pod pojęciem „mikrousług”?

Czym są mikrousługi?

Nie istnieje żadna oficjalna, kanoniczna definicja *mikrousług*. Dobrym punktem wyjścia może być przełomowy artykuł autorstwa Jamesa Lewisa i Martina Fowlera[3] na temat mikrousług z roku 2014. W tym tekście opisują oni mikrousługi jako:

> podejście do wytwarzania pojedynczej aplikacji jako zbioru małych usług, z których każda działa w swoim własnym procesie i komunikuje się z innymi za pośrednictwem lekkich mechanizmów […] budowanych wokół funkcjonalności biznesowych i wdrażanych niezależnie przy użyciu w pełni zautomatyzowanego mechanizmu wdrożeniowego.

Prawdziwym sednem artykułu Lewisa i Fowlera jest zbiór dziewięciu cech, które posiadają mikrousługi. Lista ta rozpoczyna się od kluczowej charakterystyki mikrousługi, czyli

3 *https://oreil.ly/guhCP*

dekompozycji poprzez usługi, co oznacza podział aplikacji na mniejsze usługi. Następnie przechodzą oni do omawiania rozległego zakresu funkcjonalności. Dokumentują potrzebę projektu organizacyjnego i kierowania dysponującego takimi cechami, jak *organizowanie wokół możliwości biznesowych* oraz *zdecentralizowane kierownictwo*. Wskazują praktyki dostarczania wykorzystywane w DevOps i podejściach zwinnych (*Agile*), gdy wprowadzają pojęcia *automatyzowanie infrastruktury* oraz *produkty, nie projekty*. Identyfikują również kilka kluczowych zasad architektonicznych, takich jak *logika w końcówkach* (*smart endpoints and dumb pipes*), *projektowanie pod kątem awarii* (*design for failure*) oraz projektowanie ewolucyjne.

Każda z tych charakterystyk warta jest dobrego poznania i zdecydowanie zachęcamy do przeczytania tego artykułu, jeśli ktoś jeszcze się z nim nie zapoznał. Łącznie te cechy tworzą holistyczne rozwiązanie z bardzo wielkim zbiorem uwarunkowań. Obejmują one technologię, infrastrukturę, inżynierię, operacjonalizację, zarządzanie, strukturę zespołu oraz kulturę.

Dla kontrastu istnieje też inna definicja mikrousług, sformułowana w książce *Microservice Architecture* (O'Reilly), której autorami są Irakli Nadareishvili, Ronnie Mitra, Matt McLarty i Mike Amundsen:

> *Mikrousługa* jest niezależnie wdrażanym komponentem o jednoznacznie ograniczonym zakresie funkcji, który obsługuje interoperacyjność poprzez łączność opartą na komunikatach. *Architektura mikrousług* jest stylem wytwarzania wysoce zautomatyzowanych, ewoluujących systemów oprogramowania budowanych z mikrousług o dopasowanych możliwościach.

Definicja ta jest zbliżona do tej zaproponowanej przez Lewisa i Fowlera, ale zwraca szczególną uwagę na ograniczoność zakresu, interoperacyjność i łączność opartą na komunikatach. Czyni też wyraźne rozróżnienie pomiędzy samymi mikrousługami a wykorzystującą je architekturą oprogramowania.

To jedynie dwa przykłady z całego bezmiaru rozmaitych definicji mikrousług. Podobnie jak w tych przykładach, większość definicji jest w ogólności podobna, ale różnią się tym, na co zwracają uwagę. Są jednak zwykle dostatecznie różne, aby trudno było je dopasować, gdybyśmy zechcieli zbudować podręcznikowy system mikrousług.

W świecie techniki nazwy są ważne, gdyż zapewniają prosty sposób przekazywania złożonych koncepcji. W tym przypadku nazwa „mikrousługi" pozwala nam opisać *styl* architektury oprogramowania, który ma trzy zasadnicze cechy konstrukcyjne:

1. Architektura aplikacji jest zasadniczo złożona z wywoływalnych maszynowo „usług", które są dostępne w sieci.
2. Rozmiary (albo granice) usług są ważnym czynnikiem projektowym. Granice te obejmują czynniki czasu działania, czasu projektowania, a także ludzkie.
3. System oprogramowania, jego organizacja i sposób działania są optymalizowane holistycznie, aby osiągnąć cel.

To oczywiście bardzo ogólnikowy zbiór cech konstrukcyjnych. Na przykład nie dokumentuje on stylów organizacyjnych, konkretnych narzędzi ani zasad architektonicznych, które powinny być stosowane. Nie ma tu również zdefiniowanych żadnych formalnych wzorców ani praktyk. Zamiast tego zapewnia on jedynie dość cech, abyśmy byli w stanie zidentyfikować system mikrousług, gdy go napotkamy.

Prawda jest taka, że jeśli się naprawdę postaramy, możemy nazwać architekturą mikrousług niemal dowolny system oparty na API. Jednak rzeczywistą uwagę należy zwrócić na cele naszego systemu. Naszym zdaniem pytanie (a mówiąc ściślej, odpowiedź na nie), dlaczego budujemy mikrousługi, jest znacznie bardziej pouczające, niż to, czym one są. Choć istnieje mnóstwo potencjalnych korzyści wynikających ze stosowania mikrousług, jesteśmy przekonani, że najlepszym powodem budowania oprogramowania w ten sposób jest to, że pozwala on na zredukowanie kosztów koordynacji.

Redukowanie kosztów koordynacji

Wiele firm na całym świecie odniosło sukces w implementowaniu architektury mikrousług. Niemal za każdym razem praktycy, z którymi rozmawialiśmy, informowali o zwiększeniu prędkości dostarczania oprogramowania. Jesteśmy przekonani, że to usprawnienie wynika z fundamentalnej zalety stylu mikrousług: redukcji kosztów koordynacji. Trzeba podkreślić, że w inżynierii oprogramowania istnieje wiele innych sposobów zwiększenia prędkości. Budowanie oprogramowania na sposób mikrousług jest jedynie jedną z dostępnych opcji. Możemy na przykład przyśpieszyć budowanie systemu, idąc na skróty i wprowadzające „dług techniczny”[4], którym będziemy musieli zająć się później. Alternatywnie możemy mniej skupiać się na stabilności i bezpieczeństwie, aby jak najszybciej uzyskać (prawie) gotowy produkt. W niektórych sytuacjach i dla niektórych firm są to rozsądne podejścia.

Jednak systemy opracowywane dla sektora finansowego, opieki zdrowotnej i rządowego, wśród innych, nie pozwalają na kompromis w dziedzinie bezpieczeństwa na rzecz prędkości. Jednocześnie konkurencja i rynek wymusza na tych branżach zwiększenie szybkości dostarczania, podobnie jak w każdej innej. W takim miejscu system mikrousług może prawdziwie rozbłysnąć. Zapewnia bowiem podejście architektoniczne, które pozwala zwiększyć prędkość pracy bez naruszania bezpieczeństwa. I pozwala nam to zrobić również w wielkiej skali.

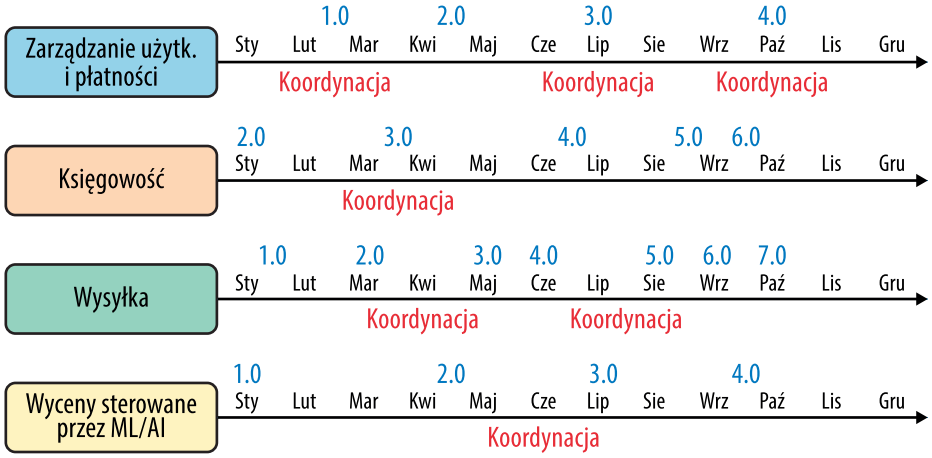
Problem kosztów koordynacji

Budowanie złożonego oprogramowania jest ciężką pracą. W filmach lub w telewizji pojedynczy genialny programista może heroicznie opracować produkt zmieniający świat w jeden bezsenny weekend. W świecie rzeczywistym osiągnięcie wyników dobrej jakości wymaga mnóstwa ludzi i jeszcze więcej czasu. Wiele zespołów pracujących przy złożonym projekcie typowo implementuje różne części danego systemu, trzymając się niezależnych

4 *https://oreil.ly/PBMHU*

planów i pracując w różnym tempie. Okresowo części te muszą być integrowane ze sobą, aby rozwiązać zależności i uwarunkowania, w którym to momencie zasadniczo autonomiczne zespoły muszą skoordynować swoją pracę (rysunek 1-1).

Wyobraźmy sobie, że Jane jest liderką zespołu odpowiedzialnego za strumień pracy Księgowość. Jej zespół właśnie ukończył sprint i pojawiła się zależność od komponentu opracowywanego przez zespół odpowiedzialny za moduł Wysyłki, kierowany przez Tyrone'a. Jako że plany pracy są niezależne, możliwe, że zespół Tyrone'a nie ukończył jeszcze pracy nad implementacją potrzebnego komponentu w strumieniu pracy Wysyłki. W tym momencie Jane ma dwie możliwości: może albo czekać na dostarczenie tego komponentu, aby wykonać niezbędne testy integracji (dając pierwszeństwo bezpieczeństwu, ale poświęcając prędkość, gdyż jej zespół musi pauzować), albo polegać na specyfikacji interfejsu uzgodnionej w kontrakcie pomiędzy jej zespołem a tym od Wysyłek, zakładając, że jego zespół dostarczy dokładnie taki komponent, jak zaplanowano. W tym drugim przypadku Jane może podążać dalej bez przerw, zwiększając szybkość działania swojego zespołu, ale potencjalnie naruszając ogólne bezpieczeństwo systemu, gdyż testy integracyjne nie zostały przeprowadzone w najwcześniejszej możliwej fazie i przyjęto założenie, że „wszystko pójdzie dobrze".



Rysunek 1-1 *Przykładowa oś czasu złożonego projektu z punktami koordynacji*

Każdy lider zespołu w złożonym, wielozespołowym środowisku regularnie staje przed tym wyborem: czy zignorować koszty koordynacji i utrzymać tempo, czy też uznać konieczność koordynacji i zwolnić. Typowo wyboru dokonujemy w oparciu o swoją intuicję i rozumienie ryzyka i korzyści, ale w ogólności, jeżeli budowany system jest dostatecznie złożony, konieczność dokonywania takich wyborów pojawia się wystarczająco często, aby stworzyć bardzo wyraźne napięcie pomiędzy prędkością a bezpieczeństwem.

To napięcie jest rzeczywiste; jednak nie jest powiązane z naszymi pierwotnymi instynktami, a więc istnieje sposób, aby je wyeliminować. Skoro to koszty koordynacji powodują spięcia, co by było, gdybyśmy mieli system specjalnie zaprojektowany w taki sposób, aby

zminimalizować te koszty? Co, jeśli zamiast wybierać jeden lub drugi sposób, zespoły niemal nigdy nie staną przed koniecznością takiego wyboru? Możemy mieć taki projekt, podkreślający minimalizowanie koordynacji, jeśli mamy autonomiczne zespoły pracujące nad małymi fragmentami izolowanej pracy. I tym właśnie, w samej swojej istocie jest architektura mikrousług.

Zrozumienie, że fundamentalną siłą budującą udane architektury mikrousług jest skupienie się na minimalizowaniu koordynacji jest niezwykle użyteczne. Daje nam coś w rodzaju uniwersalnego papierka lakmusowego. Budowanie złożonych systemów rozproszonych jako architektury mikrousług nie jest łatwe i jeśli pojawią się wątpliwości, należy zawsze postawić sobie pytanie: „Czy decyzja, którą muszę podjąć, prowadzi do zredukowania kosztów koordynacji moich zespołów, czy nie?" Znacznie łatwiejsze będzie dokonanie właściwego wyboru, gdy spojrzymy na podejmowane decyzje z perspektywy kosztów koordynacji.

Bez wątpienia mikrousługi stały się popularne, gdyż pomagają firmom odnosić sukcesy. Nowoczesne organizacje znajdują się pod niespotykanym wcześniej ciśnieniem zmian i usprawniania, które następują coraz częściej i coraz szybciej. Zainwestowanie w architekturę technologiczną, która została celowo zaprojektowana pod kątem zwiększenia szybkości i bezpieczeństwa zmian w skali wielkiej organizacji ma dużo sensu. Styl mikrousług pozwala firmom działającym w złożonych dziedzinach osiągnąć elastyczność prostszych, mniejszych firm, nie tracąc jednak możliwości wykorzystania siły i zasięgu wynikających z ich rzeczywistych rozmiarów. Jest to niezwykle atrakcyjne i pokazuje wzrost liczby zastosowań – jednak korzyści te nie przychodzą za darmo. Zbudowanie architektury mikrousług, która pozwoli odblokować tę wartość, wymaga mnóstwa wstępnej pracy, skupienia i podejmowania decyzji.

Trudne części

Jedna z największych przeszkód, które pojawiają się na drodze początkujących adaptatorów, wynika z zakresu i rozległości systemu mikrousług. Ktoś może zechcieć rozpocząć od tworzenia mniejszych, dobrze ograniczonych usług. Bardzo szybko jednak okaże się, że musi dobrać odpowiednią infrastrukturę, modele danych, platformy wykonawcze i projektowe, a także modele zespołów oraz procesy, które je wspierają. To mnóstwo zagadnień, które trzeba ogarnąć i zajmowanie się wszystkim w tak szerokim zakresie może prowadzić do pewnych unikatowych wyzwań. Oto trzy główne problemy projektowe, z którymi muszą się mierzyć architekci i inżynierowie systemów mikrousług:

Długie pętle sprzężenia zwrotnego
: Jednym z najważniejszych wyzwań jest to, że w systemie mikrousług trudno jest ocenić wpływ indywidualnych decyzji na całość systemu. Podejmowane dziś decyzje mogą okazać się źródłem problemów, ale nie ujawnią się one zbyt szybko. Możemy na przykład przyjąć początkowe założenie, że będziemy używać wspólnej biblioteki łącznościowej, aby uprościć wymianę informacji pomiędzy różnymi usługami. Z upływem czasu stanie się jasne, że utrzymywanie aktualności tej biblioteki we wszystkich

mikrousługach i zespołach okazuje się poważnym problemem. Sedno sprawy leży w tym, że trudne jest zrozumienie efektów podejmowanych decyzji, zanim nie pojawią się problemy, a tym samym trudno jest oceniać poszczególne opcje i wybierać pomiędzy nimi.

Zbyt wiele ruchomych części
: W swojej istocie system mikrousług jest złożonym systemem adaptacyjnym. Oznacza to, że każdy element systemu wpływa w jakiś sposób na inne elementy. Gdy wszystkie te części łączą się ze sobą, powstaje zachowanie wynikowego systemu. Ktoś, kto kiedykolwiek wprowadzał w organizacji nowe narzędzie lub nowy proces, zapewne widział to już na własne oczy. Niektóre zespoły reagują sprawnie na nowy bodziec i natychmiast zmieniają swój sposób działania, podczas gdy inne potrzebują pomocy, aby się dostosować. Niezależnie od tego, niemal zawsze spotkamy się z konsekwencjami dotyczącymi tego, jak ludzie pracują i jakie decyzje są podejmowane. Na przykład gdy zespół techniczny wprowadza narzędzia konteneryzacji Dockera, nieuchronnie kończy się to dostosowaniem ich cyklu wytwarzania i wydań jako skutek przyjęcia modelu wdrażania poprzez kontenery. Niekiedy te konsekwencje są zaplanowane, ale często musimy sobie radzić z niezamierzonymi skutkami wprowadzanych zmian. Ta złożoność jest tym, co sprawia, że projektowanie systemu mikrousług jest trudne. Nie jest łatwo przewidzieć konkretny wpływ wprowadzanych zmian, co prowadzi do ryzyka, że spowodujemy więcej szkód, niż korzyści, sięgając po nowy model architektury.

Paraliż analizy
: Kiedy połączymy problem długotrwałych pętli zwrotnych dla naszych decyzji ze złożonością systemu, który mamy zaprojektować, łatwo można dostrzec, dlaczego architektura mikrousług jest wyzwaniem. Decyzje, które musimy podejmować, mają dalekosiężne skutki, a zarazem są one trudne do zmierzenia. Może to prowadzić do niekończących się spekulacji, dyskusji i ocen decyzji architektonicznych ze względu na obawę przed budowaniem niewłaściwego typu systemu. Zamiast budować system, który będzie w stanie zrealizować zamierzenia biznesowe, możemy utknąć w stanie niezdecydowania, próbując wymodelować niekończące się permutacje naszych wyborów. Taki stan jest powszechnie znany jako *paraliż analityczny*. Nie pomaga też to, że sieć pełna jest strasznych opowieści, nieprzydatnych porad i sprzecznych ze sobą „najlepszych praktyk" budowania architektury mikrousług.

Bez wątpienia prawdziwym wyzwaniem jest konieczność poradzenia sobie z wielkim, skomplikowanym systemem, który rozciąga się na wielki obszar. Dobra wiadomość jest taka, że nie jest to problem, którego nie dałoby się rozwiązać. W tej książce będziemy zbierać razem i wykorzystywać zestaw praktyk i wzorców, które wyewoluowały w tej dziedzinie. Będziemy również wprowadzać i implementować narzędzia, które wcielają w życie takie sposoby działania i sprawiają, że praca nad systemem mikrousług będzie łatwiejsza, bezpieczniejsza, tańsza i szybsza.

Nauka przez praktykę

Jak dotąd, ustaliliśmy, że styl mikrousług może pomóc dostarczać oprogramowanie szybciej, bez rezygnowania z bezpieczeństwa. Wskazaliśmy też, że droga do dobrej architektury mikrousług jest trudna i najeżona wymagającymi i złożonymi decyzjami. Wielu skutecznych realizatorów mikrousług, z którymi rozmawialiśmy, budowało swoje systemy poprzez ciągłe iteracje i ulepszenia. Często zdarzało się, że zbudowali architektury, które okazały się zawodne, zanim opanowali umiejętność budowania systemu, który działa.

Gdybyśmy mieli nieograniczony czas, moglibyśmy zbudować świetną architekturę mikrousług wyłącznie w drodze eksperymentowania. Moglibyśmy przetestować niezliczone modele organizacyjne, wypróbować każdą metodologię i budować mikrousługi rozmaitych rozmiarów. O ile tylko bylibyśmy w stanie pomierzyć swoje wyniki, moglibyśmy dalej usprawniać system. Przy dostatecznej liczbie prób dotarlibyśmy do systemu, który działa zgodnie z potrzebami, jednocześnie zdobywając ogromne doświadczenie w budowaniu systemów mikrousług.

Najprawdopodobniej jednak nie mamy takiego luksusu, jak nieograniczony czas. Jak zatem zdobyć doświadczenie, którego potrzebujemy, aby budować lepsze mikrousługi?

Aby pomóc w poradzeniu sobie z tym wyzwaniem, opracowaliśmy nakazowy model mikrousług. Podjęliśmy już decyzje dotyczące kształtu zespołu, procesu wytwarzania, architektury, infrastruktury, a nawet narzędzi i technologii. Omówimy szeroki zakres tematyki, jednocześnie budując rozwiązanie, które łączy te wszystkie obszary ze sobą. Nasze decyzje opierają się na opiniach wynikających z doświadczenia w budowaniu systemów mikrousług dla wielkich organizacji. Podążając za tymi instrukcjami, na koniec lektury Czytelnik zbuduje prosty, ale w pełni funkcjonalny system mikrousług w architekturze opartej na chmurze.

Aby ułatwić odniesienie naszych przykładów mikrousług do rzeczywistego świata, będziemy używać jako modelu fikcyjnego systemu rezerwacji linii lotniczej. Będzie to oczywiście bardzo uproszczona wersja tego, jak mógłby wyglądać rzeczywisty system rezerwacji. Nasz bardzo podstawowy system rezerwacji linii lotniczej będzie zawierać dwie funkcje: usługę informacji o lotach (tylko do odczytu) oraz usługę rezerwacji miejsc.

Naszym celem jest pokierowanie Czytelnika, aby mógł zbudować swą pierwszą implementację mikrousług tak szybko, jak to możliwe. Z naszych doświadczeń akt budowania rzeczywistego systemu jest najlepszym sposobem uzyskania prawdziwego zrozumienia koniecznej do wykonania pracy i kluczowych decyzji. Nie spodziewamy się, aby Czytelnik zgadzał się ze wszystkimi naszymi wyborami. W istocie kwestionowanie tych decyzji, które podjęliśmy, jest ważną częścią procesu nauki! Mamy nadzieję, że model, który zbudujemy wspólnie, będzie zaledwie pierwszym z wielu systemów mikrousług, które Czytelnik zbuduje już samodzielnie.

Model rozwoju kompetencji braci Dreyfus
Rozpoczynanie nauki od podążania za instrukcjami jest wypróbowaną metodą zdobywania prawdziwych umiejętności. Zgodnie z artykułem Stuarta i Huberta Dreyfusów, *Five Stage Model of Adult Skill Acquisition*[5], pierwsza faza obejmuje podążanie za nakazowymi wskazówkami, zanim zostanie osiągnięta biegłość i doświadczenie.

Model mikrousług „Up and Running"

Zakres zagadnień związanych architekturami mikrousług jest dość rozległy. Niestety nie możemy go całkowicie ująć w jednej książce. Podjęliśmy jednak starania, aby omówić te obszary tematyki, które są najistotniejsze dla systemu mikrousług i mają największy wpływ na osiągnięcie sukcesu. Przyjrzyjmy się teraz pokrótce, co będziemy omawiać w naszym modelu mikrousług „Up and Running" (działającym).

Projekt zespołu
: Budowanie rozpoczniemy w rozdziale 2 od ludzkiej strony systemu mikrousług. Przedstawimy w nim wyzwania związane z efektywnym projektem zespołu oraz fundamentalne czynniki wpływające na koordynację mikrousług. Przedstawimy tu również zespoły, których będziemy używać w naszym przykładowym systemie, wraz z narzędziem Team Topologies, które pomaga w ich projektowaniu.

Projektowanie mikrousług
: Po zaprojektowaniu zespołów w rozdziale 3 przedstawimy proces SEED(S). Jest to proces projektowy, który pomoże utworzyć mikrousługi wypełniające potrzeby użytkowników i konsumentów poprzez użytkowalne interfejsy i zachowania. Następnie w rozdziale 4 zajmiemy się problemem zaprojektowania właściwych granic dla naszych przykładowych mikrousług. Wprowadzimy tu również kilka ważnych koncepcji *projektowania nakierowanego na dziedzinę* (Domain-Driven Design – DDD) i wykorzystamy proces nazywany *Event Storming* w celu „właściwego zwymiarowania" naszych usług.

Projektowanie danych
: Dane stanowią jeden z najtrudniejszych aspektów projektowania mikrousług. W rozdziale 5 przyjrzymy się czynnikom dotyczącym danych, które trzeba uwzględnić w systemie mikrousług. Wprowadzimy tu koncepcję niezależności danych i przygotujemy grunt dla architektury danych w naszym przykładowym projekcie.

Platforma chmurowa
: Nasza implementacja mikrousług zostanie zbudowana w infrastrukturze opartej na chmurze. W rozdziale 6 wprowadzimy i zaimplementujemy zasady niezmiennej infrastruktury oraz infrastruktury jako kodu (*Infrastructure as Code* – IaC) jako podstawy naszej infrastruktury mikrousług. Przedstawimy również AWS jako naszą platformę chmurową i zbudujemy potok CI/CD oparty na działaniach w GitHubie.

5 *https://oreil.ly/vs3ao*

Następnie w rozdziale 7 użyjemy tego potoku do zaprojektowania i opracowania opartej na AWS infrastruktury mikrousług, zawierającej sieć, klaster Kubernetes oraz narzędzie wdrażania GitOps.

Opracowywanie mikrousług
: Mając przygotowaną platformę infrastruktury, zagłębimy się w pracę nad inżynierią mikrousług. Rozpoczniemy od omówienia zasad i narzędzi, których będziemy potrzebować, w rozdziale 8. Następnie w rozdziale 9 zaimplementujemy dwie niezależne, heterogeniczne mikrousługi dla naszej przykładowej aplikacji.

Wydania i zmiany
: Całe rozwiązanie zbierzemy w całość w rozdziale 10, w którym wdrożymy jedną z przykładowych, opracowanych przez nas mikrousług w przygotowanej wcześniej platformie chmurowej. W tym celu użyjemy zestawu technologii obejmującego DockerHub, Kubernetes, Helm oraz Argo CD. Na koniec, po wydaniu, dokonamy retrospektywnego przeglądu systemu w rozdziale 11.

Opracowany przez nas model zbudowany jest na bazie pięciu wiodących zasad, w tym *wzorca dwunastu aspektów* aplikacji[6]. Osoby zainteresowane mogą poczytać o wiodących zasadach naszego modelu w repozytorium GitHub tej książki[7].

Mamy nadzieję, że ten krótki przegląd daje wyobrażenie o zasięgu naszego modelu i przykładowej aplikacji. W trakcie lektury książki będziemy implementować pełny, działający system. Aby do tego dotrzeć, będziemy musieli podjąć mnóstwo decyzji. Tym samym pierwsze narzędzie, którego potrzebujemy, to sposób śledzenia tych decyzji, które są naprawdę ważne.

Decyzje, decyzje …

Jeśli chodzi o budowanie oprogramowania, decyzje są ważną sprawą. Inżynierowie i architekci oprogramowania otrzymują swoje – niemałe! – wynagrodzenie za decyzje, które podejmują i problemy, które rozwiązują. Jakość oprogramowania i zapewniane przez nie wyniki biznesowe zależą od jakości tych decyzji.

Jednak decyzje te nie zawsze są łatwe. Co więcej, nie zawsze są poprawne. Dokonujemy najlepszych decyzji, jak to możliwe, opierając się posiadanych informacjach, doświadczeniu i talencie. Gdy którakolwiek z tych zmiennych się zmieni, nasze decyzje również powinny ulec zmianie. Niektóre decyzje są właściwe w chwili ich podejmowania, ale stają się przestarzałe, gdy zmieni się technologia, ludzie lub sytuacje. Pewne decyzje nigdy nie są poprawne. W każdym przypadku potrzebujemy sposobu wyłuskiwania tych decyzji, które są istotne, aby móc je ponownie rozważyć i – być może – poprawić w przyszłości.

6 *https://12factor.net*
7 *https://oreil.ly/MicroservicesUpandRunning*

Aby zaspokoić tę potrzebę, zamierzamy wykorzystać narzędzie nazywane *rekordem decyzji architektonicznych* (*architecture decision record* – ADR). Nie jest pewne, kto wymyślił termin *ADR* ani kiedy został on użyty po raz pierwszy, ale idea dokumentowania decyzji projektowych jest obecna od długiego czasu. Prawdziwym problemem jest to, że większość ludzi nie poświęca czasu, aby to robić. Z naszego doświadczenia ADR jest niezwykle użytecznym narzędziem i dobrym sposobem, aby uzyskać jasność co do decyzji, które trzeba podejmować.

Dobry rekord decyzji musi zawierać cztery ważne elementy:

Kontekst
: Przed jakim wyzwaniem stoimy? Jaki problem próbujemy rozwiązać? Jakie są ograniczenia? Rekord decyzji powinien zawierać podsumowanie tych elementów kontekstowych. W ten sposób będziemy mogli zrozumieć uzasadnienie decyzji i dlaczego może ona wymagać uaktualnienia.

Alternatywy
: Decyzja nie jest decyzją, jeśli nie ma jakiegoś wyboru. Dobry rekord decyzji powinien pomóc zrozumieć, jakie są te możliwości. Pomoże to lepiej zrozumieć kontekst i „przestrzeń wyboru" w chwili podejmowania decyzji.

Wybór
: W samym sercu decyzji znajduje się wybór. Każdy rekord decyzji musi dokumentować wybór, który został dokonany.

Wpływ
: Decyzje mają konsekwencje i rekord decyzji powinien dokumentować te ważne. Jakie kompromisy były konieczne? Jak dokonany wybór wpływa na sposób naszego działania albo inne decyzje, które trzeba podjąć?

Rekord decyzji można utworzyć w dowolny sposób. Możemy je zapisywać w plikach tekstowych, użyć narzędzia do zarządzania projektami albo rejestrować je w arkuszu kalkulacyjnych. Format i narzędzia są mniej ważne, niż zawartość. O ile tylko uwzględnimy obszary, które opisaliśmy, będziemy mieli dobry rekord decyzji.

W naszym przykładowym projekcie użyjemy istniejącego już formatu, nazywanego *lekkim rekordem decyzji architektonicznych* (*lightweight architectural decision record* – LADR). Format LADR został stworzony przez Michaela Nygarda () i jest przyjemnym, zwięzłym sposobem dokumentowania decyzji. Zbudujemy wspólnie rekord LADR, aby go poznać.

Jeśli ktoś woli używać czegoś innego niż LADR, Joel Parker Henderson udostępnia świetną listę formatów i szablonów ADR[8].

8 *https://oreil.ly/T3Tc-*

Tworzenie lekkiego rekordu decyzji architektonicznej

Pierwszą, kluczową decyzją, którą zarejestrujemy, jest decyzja o utrzymywaniu rejestru decyzji. Mówiąc prościej, utworzymy ADR, który stwierdza, że zamierzamy śledzić podejmowane przez nas decyzje. Jak wspomnieliśmy, będziemy używać formatu LADR. Przyjemną cechą LADR jest to, że został zaprojektowany jako lekki. Pozwala śledzić nasze decyzje w prostych plikach tekstowych, które możemy szybko pisać. A ponieważ będziemy zajmować się plikami tekstowymi, możemy nawet zarządzać naszymi rekordami decyzji w taki sam sposób, jak zarządzamy kodem źródłowym.

Rekordy LADR są pisane przy użyciu formatu tekstowego o nazwie Markdown[9], który zapewnia elegancki i prosty sposób pisania dokumentacji. Wielką zaletą Markdown jest to, że jest on łatwo czytelny dla człowieka w swojej surowej formie i większość popularnych narzędzi wie, jak go interpretować. Na przykład Confluence, GitLab, GitHub i SharePoint potrafią przetwarzać Markdown i prezentować go jako dokument sformatowany, czytelny dla człowieka.

Aby utworzyć nasz pierwszy LADR oparty na Markdown, wystarczy otworzyć swój ulubiony edytor tekstowy i utworzyć nowy dokument. Pierwszą rzeczą, którą zrobimy, będzie ustalenie struktury dokumentu.

Dodajemy poniższy tekst do swojego pliku LADR:

```
# OPM1: Używanie ADR do śledzenia decyzji

## Status
Zaakceptowana

## Kontekst

## Decyzja

## Konsekwencje
```

To kluczowe elementy naszego rekordu decyzji. Znaki `#` poprzedzające wiersze są tokenami Markdown, które pozwolą parserowi zidentyfikować te wiersze jako nagłówki. Zwróćmy uwagę, że nadaliśmy temu rekordowi tytuł, który odpowiada podejmowanej decyzji. W tytule tym widzimy również nieco tajemniczy akronim: „OPM1". Jest to po prostu skrótowy kod, który pomaga oznakować decyzję i rozpoznać, jakiej części systemu dotyczy ta decyzja. W tym przypadku „OPM1" oznacza, że jest to pierwsza decyzja dotycząca modelu operacyjnego (*operating model*).

Nagłówek `Status` naszego rekordu pozwala określić, w jakiej fazie cyklu życia jest dana decyzja. Jeśli na przykład przygotowujemy szkic nowej decyzji, którą trzeba będzie dopiero uzgodnić, moglibyśmy zacząć od statusu `Propozycja`. Jeśli natomiast rozważamy zmianę istniejącej decyzji, możemy zmienić jej status na `W przeglądzie`. W naszym przypadku podjęliśmy już tę decyzję, zatem ustaliliśmy jej status jako `Zaakceptowana`.

9 *https://oreil.ly/oRyx0*

Sekcja `Kontekst` opisuje problem, ograniczenia i tło podejmowanej decyzji. W naszym przypadku chcemy przechwycić potrzebę rejestrowania ważnych decyzji oraz to, dlaczego jest to ważne. Dodajemy poniższy tekst (albo własną wariację na ten temat) do sekcji `Kontekst` naszego rekordu:

```
## Kontekst
Architektura mikrousług jest złożona i będziemy musieli podejmować wiele decyzji.
Potrzebujemy sposobu śledzenia podejmowania ważnych decyzji, aby móc do nich
wrócić i je ocenić ponownie w przyszłości. Preferujemy użycie lekkiego rozwiązania
opartego na plikach tekstowych, aby nie musieć instalować żadnych nowych narzędzi
programowych.
```

Po wpisaniu kontekstu możemy przejść do rejestrowania faktycznie podjętej decyzji. Możemy tu wyliczyć kilka z rozważanych alternatyw, a także nasz wybór o użyciu LADR. Dodajemy poniższy tekst do sekcji `Decyzja`, aby udokumentować ten fakt:

```
## Decyzja
Zdecydowaliśmy się na użycie formatu LADR autorstwa Michaela Nygarda. LADR jest
oparty na plikach tekstowych i jest wystarczająco lekki, aby spełnić nasze
wymagania. Będziemy przechowywać każdy rekord LADR w jego własnym pliku tekstowym
i zarządzać tymi plikami tak samo, jak kodem. Rozważaliśmy również następujące
rozwiązania alternatywne:
* Narzędzia zarządzania projektami (nie wybrane, gdyż nie chcemy instalować
  dodatkowych narzędzi)
* Nieformalne lub "ustne" przechowywanie rekordów (nie wiarygodne)
```

Wszystko, co pozostało, to udokumentowanie konsekwencji. W naszym przypadku jedną z kluczowych konsekwencji jest to, że będziemy musieli poświęcić czas na rzeczywiste dokumentowanie decyzji i zarządzanie tymi rekordami. Możemy to zapisać następująco:

```
## Konsekwencje
* Musimy zapisywać rekordy decyzji dla kluczowych rozstrzygnięć
* Potrzebujemy rozwiązania zarządzania kodem źródłowym, aby zarządzać rekordami
  decyzji
```

To wszystko, czego potrzeba, aby utworzyć LADR. Jest to niezwykle użyteczny sposób rejestrowania naszych przemyśleń, a dodatkową zaletą jest to, że wymusza podejmowanie rozważnych, przemyślanych decyzji. W miarę budowania naszej przykładowej aplikacji rezerwowania lotów będziemy tworzyć dziennik podejmowanych kluczowych decyzji. Dla oszczędności czasu (i miejsca w książce) nie będziemy tu zapisywać całego rekordu decyzji. Zamiast tego będziemy zaznaczać fakt podjęcia kluczowej decyzji taką notą, jak poniżej.

Kluczowa decyzja: Użycie ADR do śledzenia decyzji

Będziemy używać ADR do rejestrowania kluczowych decyzji podejmowanych w trakcie projektowania i budowania systemu.

Czytelnik może znaleźć szczegółowy rekord każdej decyzji w repozytorium GitHub dla tej książki[10].

Podsumowanie

W tym rozdziale przedstawiliśmy kilka podstawowych koncepcji dotyczących tej książki. Podaliśmy zgrubną definicję systemu mikrousług, w tym zbiór trzech kluczowych cech. Zidentyfikowaliśmy redukcję kosztów koordynacji jako kluczową korzyść wynikającą ze stosowania mikrousług. Pokazaliśmy też główne wyzwania, przed jakimi stają osoby chcące stosować architekturę mikrousług, a mianowicie złożoność i paraliż analityczny.

Aby pomóc w poradzeniu sobie z tymi wyzwaniami, wprowadzamy model mikrousług – opartą na opiniach i doświadczeniu, nakazową implementację, która powinna przyśpieszyć proces nauki. Przedstawiliśmy główne aspekty modelu i zagadnienia, które zamierzamy omówić. Na koniec wprowadziliśmy koncepcję rekordów decyzji architektonicznych (ADR), którą planujemy wykorzystywać w pozostałej części książki.

Po całym tym przeglądzie wszystko, co nam pozostało, to zbudować system. Pracę rozpoczniemy w rozdziale 2 od zajęcia się tym, jak wykonywana jest praca nad mikrousługami, ze szczególnym skupieniem na koordynacji zespołu.

10 *https://github.com/implementing-microservices/ADRs*. Trzeba zwrócić uwagę, że rekordy zapisane w repozytorium są w języku angielskim (przyp. tłum.).

Rozdział 2

Projektowanie modelu operacyjnego mikrousług

W tej książce będziemy budować aplikację opartą na mikrousługach. Aby to osiągnąć, zaprojektujemy i utworzymy te mikrousługi, a także infrastrukturę i narzędzia, których będziemy potrzebować do ich obsługiwania. Jednak osiągnięcie sukcesu w tej dziedzinie wymaga czegoś więcej, niż jedynie napisania i wdrożenia kodu. Aby nasze działania przebiegły z powodzeniem, potrzebujemy właściwych ludzi, właściwych sposobów pracy i stosowania właściwych zasad, aby zapewnić sprawne działanie całego systemu. Z tego względu rozpoczniemy naszą podróż od zaprojektowania ogólnego modelu operacyjnego naszej aplikacji.

Model operacyjny to zbiór ludzi, procesów i narzędzi, które leżą u podstaw naszego systemu. Ukierunkowuje całość procesu decyzyjnego i pracy, którą trzeba wykonać podczas budowania oprogramowania. Model operacyjny może na przykład definiować odpowiedzialności poszczególnych zespołów. Może także definiować nadzór nad pracą i podejmowaniem decyzji.

Model operacyjny możemy traktować jako swoisty „system operacyjny" naszego rozwiązania. Całość pracy niezbędnej do zbudowania systemu mikrousług odbywa się w ramach struktury zespołów, procesów i granic, które definiujemy w modelu. W praktyce modele operacyjne mogą mieć wielki zakres i być bardzo szczegółowe. Jednak w naszym przykładzie zredukujemy zakres i skupimy się na najważniejszych częściach systemu mikrousług – a mianowicie na tym, jak zaprojektować zespoły i jak będą one współpracować.

To właśnie będziemy omawiać w tym rozdziale: zależności pomiędzy zespołami a implementacjami mikrousług. Przedstawimy narzędzie nazywane *Team Topologies* (topologie zespołów) i na koniec uzyskamy projekt oparty na zespołach, którego będziemy mogli użyć jako fundamentu dla reszty naszej konstrukcji.

Czytelnik nie musi rzeczywiście zbierać personelu i tworzyć zespołów, które definiujemy, aby podążać za budową naszego systemu mikrousług „Up and Running".

Rozpoczniemy od zastanowienia się, dlaczego zespoły i ich projekt organizacyjny są tak ważne.

Dlaczego ludzie i zespoły są istotne

Model, którego używamy w tej książce, koncentruje się głównie na decyzjach związanych z wyborem technologii i narzędzi. Jednak technologia sama w sobie nie zapewni wartości, której potrzebujemy od systemu mikrousług. Technologia jest ważna. Dobre wybory technologiczne sprawiają, że łatwiejsze jest realizowanie zadań, które inaczej mogłyby się okazać skrajnie trudne lub niewykonalne. W najlepszym wydaniu technologia otwiera wiele drzwi i uwalnia nowe możliwości. Jednak sama w sobie jest bezużyteczna.

Możemy mieć najlepsze w świecie narzędzia i platformy, ale poniesiemy klęskę, jeśli nie będziemy dysponowali odpowiednią kulturą i organizacją, w której będą używane. Cel, który próbujemy osiągnąć w naszym modelu, to umieszczenie dobrej technologii w rękach niezależnych, wysoko funkcjonalnych zespołów. Tym samym potrzebujemy zacząć od rozważenia typów zespołów i struktury, która będzie najlepiej sprawdzać się w modelu, jaki zamierzamy opracować.

W systemie mikrousług ważna jest kultura i struktura zespołów. Z badań, które przeprowadziliśmy na potrzeby tej książki, a także z naszych własnych doświadczeń wynika ważne spostrzeżenie: to ludzie i procesy są krytycznymi czynnikami sukcesu. Implementacja mikrousług jest wartościowa, o ile zapewnia nam swobodę łatwego i szybkiego dokonywania zmian. W praktyce jednak zmiany są produktem ubocznym zdolności naszej organizacji do podejmowania decyzji. Jeśli nie będziemy mogli szybko podejmować dobrych decyzji, uzyskanie rzeczywistej korzyści (wartości) z systemu mikrousług będzie trudne lub czasochłonne. Przypomina to budowanie samochodu wyścigowego z bardzo słabym silnikiem. Nieważne, jak dobrze samochód będzie zbudowany – nigdy nie będzie jechać tak dobrze, jak powinien.

Koncepcja, że projekt zespołu i jego kultura są ważne, nie jest nowa. Mel Conway elokwentnie ujął wpływ struktury zespołu na projektowanie systemu w swoim, obecnie dobrze znanym artykule „How Do Committees Invent?”[1]. Jego pogłębione obserwacje doprowadziły do jeszcze lepiej znanej parafrazy tych twierdzeń, nazywanej „prawem Conwaya”:

> Każda organizacja projektująca jakiś (ogólnie zdefiniowany) system utworzy projekt, którego struktura będzie kopią struktury komunikacyjnej tej organizacji.
>
> *przypisywane Fredowi Brooksowi*

Conway mówi, że produkt (wynik działań) organizacji odzwierciedla sposób, w jaki ludzie i zespoły komunikują się ze sobą. Dla przykładu rozważmy zespół mikrousługi, który musi konsultować się ze scentralizowanym zespołem ekspertów bazodanowych, ilekroć potrzebują zmian w modelu danych. Istnieje spore prawdopodobieństwo, że model danych

1 *https://www.melconway.com/Home/pdf/committees.pdf*

i jego implementacja w utworzonym systemie również będą scentralizowane. Wynikowy system ostatecznie będzie odwzorowywał schemat organizacyjny i model koordynacji przedsiębiorstwa.

Wniosek z tych rozważań jest taki, że w budowaniu i utrzymywaniu systemu mikrousług najważniejsi są ludzie. Sposób, w jaki podejmują decyzję, wykonują swoją pracę i komunikują się ze sobą ma wielki wpływ na tworzony system. Mówiąc ogólnie, można wyróżnić trzy czynniki ludzkie, które mają największy wpływ na system mikrousług: wielkość zespołu, jego umiejętności oraz koordynacja międzyzespołowa. Przyjrzyjmy się uważniej każdemu z nich, zaczynając od rozmiarów.

Wielkość zespołu

Człon „mikro" w terminie „mikrousługi" sugeruje, że rozmiar jest ważny i że im mniejsze, tym lepsze. Uczciwie mówiąc, jest to nadmierne uproszczenie. Jednak zasadnicze spostrzeżenie pozostaje prawdą: budowanie mniejszych, indywidualnie wdrażanych usług jest ważną częścią dla osiągnięcia sukcesu. Okazuje się, że wielkość zespołów budujących te usługi również ma wielkie znaczenie.

Jeśli zespół zawiera zbyt wielu ludzi, będziemy musieli poświęcić więcej czasu na komunikowanie się pomiędzy sobą. Ta wewnętrzna koordynacja ostatecznie spowalnia cały zespół, powodując rzadsze dostarczanie zmian. Z drugiej strony, jeśli będziemy mieli zbyt mało ludzi, może nam zabraknąć umysłów i rąk do wykonania pracy. „Właściwe wymiarowanie" zespołów jest ważną częścią projektowania systemu. Choć nie istnieje żaden „złoty przepis" na wskazanie rozmiaru, który sprawdzi się wszędzie i w każdej sytuacji, lata doświadczeń i badań nad rozmiarami zespołów doprowadziły do sformułowania ogólnie akceptowanych praktyk.

Bill Gore, współzałożyciel firmy Gore-Tex, ograniczył wielkość zespołów w firmie, aby utrzymać ich efektywność. Aby to zrealizować, zdefiniował „wbudowany" limit wielkości: wszyscy członkowie zespołu muszą mieć osobiste kontakty ze sobą, każdy z każdym. Gdy zespół staje się zbyt duży, aby wszyscy jego członkowie znali się wzajemnie, jednostka ta urosła nadmiernie.

Antropolog Robert Dunbar w swoich badaniach zachowań społecznych szympansów zauważył, że wielkości grup szympansów są skorelowane z rozmiarami ich mózgów. Ekstrapolując te spostrzeżenia na swoją wiedzę o mózgu człowieka określił zbiór wielkości grup dla ludzi. *Liczba Dunbara*[2] stwierdza, że jesteśmy – bazując na wielkości naszych mózgów – w stanie wygodnie utrzymać około 150 stabilnych znajomości. Dunbar ustalił też, że ludzie są w stanie utrzymać około 5 bliskich, rodzinnych związków i jedynie około 15 zaufanych przyjaciół.

Jeff Bezos, CEO i twórca Amazonu, zapewne najbardziej znany z wymienionych, dał nam „regułę dwóch pizz"[3]. Głosi ona, że zespół Amazonu powinien być dostatecznie mały, aby dwie pizze wystarczyły dla wszystkich członków. Choć nie są jasne konkretne

2 *https://oreil.ly/-DbyT*
3 *https://oreil.ly/ccT85*

szczegóły wielkości tych placków ani tego, jaki apetyt mają członkowie zespołu, dwupizzowy zespół znajdzie się zapewne gdzieś w przedziale od 5 do 15 osób przedstawionym przez Dunbara i stwarza warunki dla utrzymania osobistych związków, opisanych przez Gore.

Wszystkie te opowieści wskazują limit wielkości oparty na zdolności ludzi do efektywnego komunikowania się ze sobą. Nasze doświadczenia i badania pasują do tej intuicyjnej koncepcji. Aby utrzymać wysokie tempo zmian, musimy ograniczyć wielkość zespołów w naszym systemie. W naszym modelu mikrousług będziemy chcieli ograniczyć wielkość zespołów do czegoś pomiędzy pięcioma a ośmioma osobami.

Kluczowa decyzja: Wielkość zespołu powinna być limitowana

Zespoły wykonujące pracę w naszym systemie nie powinny zawierać po więcej niż osiem osób.

Utrzymanie niewielkiego rozmiaru zespołu pomoże ograniczyć konieczne interakcje wewnętrzne. Pojawia się tu jednak efekt domina. Mniejsze wielkości zespołów zazwyczaj oznaczają więcej zespołów. Musimy zatem zachować ostrożność przy projektowaniu reszty systemu. Nic dobrego nie wyniknie z posiadania małych zespołów, jeśli będą one poświęcać cały czas na komunikowanie się pomiędzy sobą! Aby tego uniknąć, musimy umożliwić im niezależne i autonomiczne działanie, tak bardzo, jak tylko będzie to (bezpiecznie) możliwe.

Innym ubocznym efektem zmniejszania wielkości naszych zespołów jest to, że ogranicza to liczbę specjalistów, którymi możemy dysponować. Przy mniejszej liczbie członków zespołu musimy zadbać o to, że kolektywnie będziemy mieli dość talentów, aby uzyskać wyniki dobrej jakości. To dlatego musimy rozważyć skład naszych zespołów z perspektywy potrzebnych umiejętności.

Umiejętności zespołu

Zespół może być tylko tak dobry, jak jego członkowie. Jeśli chcemy mieć zespoły o wysokiej wydajności, musimy zwrócić szczególną uwagę na to, kto powinien znaleźć się w danym zespole. Dla przykładu musimy się zastanowić, jakich ról i specjalizacji będą potrzebować nasze zespoły? Jak utalentowani i doświadczeni powinni być poszczególni członkowie? Jaka jest właściwa mieszanka umiejętności i doświadczenia?

Prawdą jest, że na pytania te trudno jest udzielić uniwersalnej odpowiedzi. Wynika to stąd, że ludzie i kultura pracy są często najbardziej unikatową cechą miejsca, w którym pracujemy. Przykładowo, potężna firma wydaje mnóstwo pieniędzy, aby zdobyć dla siebie topowych geniuszy technologicznych z całego świata. Inna (zapewne mniejsza, a na pewno nie tak zasobna) firma może głównie zatrudniać lokalnych kandydatów, skupiających się na rozwoju swojej kariery i uczących się w trakcie pracy od niewielkiej liczby ekspertów. Dobry projekt zespołu w każdej z tych dwóch kategorii firm będzie zapewne wyglądać zupełnie inaczej.

W tej książce chcemy skupić się na budowaniu implementacji mikrousług. Tym samym nie będziemy zagłębiać się zbytnio w projektowanie organizacji i kultury pracy. Dobra wiadomość jest taka, że istnieje ogólna reguła, którą możemy zastosować, aby pomóc dowolnym realizatorom mikrousług. Jest to zasada zespołu wielofunkcyjnego (*cross-functional team*).

Ludzie o różnego rodzaju umiejętnościach (albo funkcjach) pracują wspólnie w zespole wielofunkcyjnym w celu osiągnięcia tego samego celu. Ta wiedza może rozciągać się zarówno na dziedziny technologiczne, jak i biznesowe. Na przykład zespół wielofunkcyjny mógłby zawierać projektantów UX, programistów aplikacji, właścicieli produktu i analityków biznesowych.

Zespoły wielofunkcyjne spotykane są od dłuższego czasu, sięgając co najmniej lat pięćdziesiątych XX wieku, gdy rozwiązanie zostało szeroko zastosowane w firmie Northwestern Mutual Life Insurance Company.

Wielką zaletą budowania zespołu w taki sposób jest przyśpieszenie podejmowania decyzji. Określiliśmy już górną granicę wielkości zespołów, ograniczając członkostwo do maksymalnie ośmiu osób. „Właściwie zwymiarowany" zespół, z odpowiednimi ludźmi na pokładzie, może działać z dużą prędkością i odpowiedzialnością.

Ale kim są *odpowiedni ludzie*? Jeśli chodzi o wielkość zespołu, możemy opierać się na anegdotach, doświadczeniu, a nawet badaniach naukowych. Jednak w przypadku profilu zespołu znacznie trudniej będzie znaleźć spójne wskazówki. Dla przykładu, gdy przyjrzymy się pracy wielkich dostawców chmurowych nad mikrousługami, używają oni zwykle czterech do pięciu ekspertów o wiedzy z różnych dziedzin, uzupełnionych pojedynczym specjalistą od testowania. I przeciwnie, w firmach konsultingowych widywaliśmy silnie zróżnicowaną mieszankę wyspecjalizowanych inżynierów, właścicieli produktu, menedżerów projektu i ekspertów od testów w każdym zespole. Zdolności, doświadczenie i kultura organizacji mają wpływ na właściwy dobór ludzi.

Tak więc, zamiast dokładnie nakazywać, jakich ról będziemy potrzebować w swoich zespołach, w naszym modelu dokonamy dwóch ogólnych decyzji. Po pierwsze, zespoły powinny być wielofunkcyjne. Nasze doświadczenie pokazuje, że praca nad mikrousługami przebiega sprawniej, jeśli zespoły są w stanie samodzielnie podejmować właściwe decyzje. Zespoły wielofunkcyjne to umożliwiają. Po drugie, zespoły powinny składać się z członków, którzy bezpośrednio wpływają na wyniki. W ten sposób wybieramy ludzi, o których wiemy, że mogą wnieść dobry wkład w pracę zespołu. Nie potrzebujemy w zespole obserwatorów ani ludzi, którzy są tylko pobocznie związani z tą pracą i podejmowanymi decyzjami.

Kluczowa decyzja: Trzeba zdefiniować zasady członkostwa zespołu

Zespoły powinny być wielofunkcyjne i składać się tylko z takich członków, którzy są w stanie dodać wartość do produktu, usługi lub innych wyników zespołu.

Przy odpowiedniej wielkości i właściwych ludziach powinniśmy być w stanie zbudować efektywne zespoły, które zapewnią zrealizowanie pracy. W miarę wzrostu liczby zespołów będziemy musieli również rozważyć koordynowanie działań poszczególnych zespołów ze sobą. To ostatnia właściwość zespołów, którą musimy się zająć.

Koordynacja międzyzespołowa

Budowanie zespołów o odpowiedniej wielkości i wypełnienie ich właściwymi ludźmi pomoże w utworzeniu wysokowydajnych zespołów. Jednak to komunikacja pomiędzy zespołami, bardziej niż wewnątrz nich, jest tym, co może sprawić, że nasz projekt ugrzęźnie. Problem kosztów koordynacji podkreśliliśmy już w punkcie „Problem kosztów koordynacji". Jeśli zdołamy zredukować ilość zdarzeń koordynacji, które występują pomiędzy zespołami, nasze zespoły będą w stanie szybciej dokonywać zmian.

Byłoby świetnie, gdyby nasze zespoły mikrousług mogły działać w pełni autonomicznie i niezależnie. Jeśli zespoły będą mogły swobodnie podejmować swoje decyzje dotyczące projektowania, programowania, testowania i wdrażania, nie byłoby żadnego „tarcia organizacyjnego" spowalniającego pracę. Jednak z naszego doświadczenia nie jest to praktyczna metoda działania.

Wynika to stąd, że koordynacja i współpraca są ważne dla sukcesu organizacji. Moglibyśmy chcieć, aby nasze zespoły mikrousług działały niezależnie, ale chcemy też, aby utworzyły one usługi, które będą wartościowe dla klientów, użytkowników, a w konsekwencji dla organizacji. Oznacza to, że konieczna jest komunikacja dla ustalenia wspólnych celów, przekazywania żądań zmian, dostarczania informacji zwrotnych i – przede wszystkim – rozwiązywania problemów.

Na dodatek, gdy zespoły działają zupełnie niezależnie, możliwości dzielenia się są mniejsze. Zespoły mikrousług działające niezależnie mogą wybierać właściwe narzędzia dla określonej pracy i budować wysoce wydajne systemy. Jednak ta efektywność ograniczona jest do zespołu. Czasami oznacza to, że w ten sposób tracimy wydajność *na poziomie systemu*. Na przykład jeśli wszystkie nasze zespoły zaprojektują i zbudują swoje własne architektury sieciowe oparte na chmurze, tracimy okazję do wykonania tej pracy tylko raz i współużytkowania rozwiązania.

Możliwe jest zbudowanie organizacji, która zapewnia wydajną niezależność i autonomię zespołów poprzez samo-organizowanie się. Na przykład pionier rozwiązań mikrousług Fred George opisał metodę, którą nazywa Anarchią programistyczną (Programmer Anarchy – *https://oreil.ly/C1N0f*), w której zespoły techniczne mają pełną autonomię (i odpowiedzialność) formowania zespołów, wyboru pracy i projektowania swoich własnych rozwiązań. Jednak z naszych doświadczeń wynika, że większość wielkich przedsiębiorstw ma trudności ze spójnym stosowaniem takiego podejścia.

Jeśli za bardzo zapędzimy się w zapewnianiu zespołom niezależności i autonomii, wprowadzimy rozwiązania niewydajne na poziomie całego systemu, a być może również

odstępstwa od celów organizacyjnych. Jeśli przeciwnie, wprowadzimy zbyt wiele koordynacji, ryzykujemy ugrzęźnięcie całego systemu i utratę korzyści związanych z łatwością wprowadzania zmian w mikrousługach. Wyzwanie polega tu na znalezieniu odpowiedniego punktu równowagi pomiędzy niezależną pracą a skoordynowanymi wysiłkami. Wymaga to trochę eksperymentowania i ciągłego dostrajania naszego projektu zespołów.

Co najważniejsze, optymalizowanie koordynacji zespołów wymaga aktywnych wysiłków projektowych. Jednym z błędów, które, jak wielokrotnie widzieliśmy, popełniają doświadczeni praktycy, jest skupienie się wyłącznie na architekturze technicznej. Gdy tak się stanie, projekt zespołów formowany jest wokół tworzonej technologii. Trudności koordynacyjne stają się oczywiste dopiero wtedy, gdy pojawią się problemy. Jednak w tym momencie dokonanie zmian często jest już zbyt kosztowne lub zbyt trudne.

Aby uniknąć tego problemu, zajmiemy się problemem koordynacji zespołów i ich projektowaniem w pierwszym kroku naszego procesu projektowania systemu. Niektórzy nazywają to „odwrotnym manewrem Conwaya", gdyż zaprojektowana przez nas struktura komunikacyjna na koniec będzie informować, jaki system będzie budowany. Jakkolwiek zechcemy to nazwać, ustaliliśmy, że rozpoczynanie od skupienia się na projekcie zespołów i ich koordynacji może naprawdę pomóc w osiągnięciu sukcesu w projektowaniu mikrousług. W istocie ten punkt jest tak ważny, że zarejestrujemy go jako decyzję.

Kluczowa decyzja: Kiedy projektować modele zespołów i koordynacji

Projektowanie zespołów i koordynacji powinno rozpocząć się przed przystąpieniem do projektowania architektury systemu lub samych mikrousług. Modele zespołów i koordynacji muszą być stale uaktualniane i usprawniane w całym czasie życia systemu.

Zagadnieniom tym poświęcimy resztę tego rozdziału. Najpierw przedstawimy użyteczne narzędzie projektowania modeli zespołów mikrousług, o nazwie Team Topologies (Topologie zespołów).

Przedstawiamy Team Topologies

Ponieważ zamierzamy rozpocząć naszą pracę projektową od zajęcia się zespołami, potrzebujemy sposobu katalogowania i komunikowania naszych decyzji. Istnieje mnóstwo sposobów dokumentowania projektów zespołów. Na potrzeby naszego modelu użyjemy narzędzia o nazwie Team Topologies.

Team Topologies[4] jest podejściem projektowym wynalezionym przez Matthew Skeltona i Manuela Pais. Lubimy się nim posługiwać, gdyż zapewnia formalny język opisywania projektu zespołów, ze szczególnym zwróceniem uwagi na to, jak zespoły współpracują ze sobą nawzajem.

4 *https://teamtopologies.com*

W naszej pracy nie będziemy wykorzystywać wszystkich aspektów podejścia Team Topologies. Zamiast tego naszkicujemy trzy elementy: typy zespołów, tryby interakcji pomiędzy zespołami oraz tworzenie diagramów. Mając te części, będziemy w stanie zbudować prosty, ale działający projekt naszych zespołów mikrousług.

Następnie przyjrzymy się różnym częściom podejścia Team Topology, poczynając od typów zespołów, jakie możemy zdefiniować.

Typy zespołów

Jedną z kluczowych koncepcji Team Topologies są *typy zespołów*. Są to archetypy lub kategorie, które opisują podstawową naturę zespołu z perspektywy jego komunikacji z resztą organizacji. Team Topologies definiuje cztery typy zespołów: zadaniowe, umożliwiające, skomplikowanych podsystemów oraz platformowe.

Przyjrzymy się pokrótce każdemu z nich:

*Zadaniowy (**Stream-aligned**)*
: Zespół zadaniowy posiada i uruchamia możliwy do dostarczenia element pracy. Kluczową cechą tego zespołu jest ciągłe dostarczanie czegoś ważnego dla organizacji biznesowej. Zespół zadaniowy wciela w życie powiedzenie Wernera Vogela, CTO Amazonu[5] na temat odpowiedzialności zespołów w firmie Amazon: „Kto to buduje, ten to uruchamia". Zespoły zadaniowe nie są rozwiązywane po wydaniu produktu. Zamiast tego nadal są jego właścicielami i implementują „strumień" (*stream*) zmian, ulepszeń i poprawek w swoim komponencie. Zespoły mikrousług są zazwyczaj zespołami tego typu, gdyż nieustannie tworzą i publikują funkcjonalności posiadanych przez siebie usług.

Umożliwiający (Enabling)
: Zespół umożliwiający wspomaga pracę innych zespołów poprzez model zaangażowania doradczego. Zespoły takie są zwykle złożone ze specjalistów i ekspertów dziedzinowych, którzy są w stanie przekraczać luki w wiedzy lub umiejętnościach. Jednak mogą również pomagać indywidualnym zespołom w zrozumieniu ogólnego obrazu organizacji lub branży, w której działają. Na przykład zespół architektoniczny może pomóc zespołom mikrousług w poznawaniu wyłaniających się standardów technicznych i konwencji stosowanych w organizacji.

Skomplikowanych podsystemów (Complicated-Subsystem)
: Tego typu zespół pracuje w dziedzinie lub nad zagadnieniem, które jest trudne do zrozumienia, a przynajmniej jest na tyle trudne, że do jego przezwyciężenia organizacja nie ma dostatecznych zasobów. Niektóre obszary problemowe nie skalują się dobrze i nie mogą zostać powierzone dowolnemu zespołowi. Na przykład dostrajanie oprogramowania pod kątem bezpieczeństwa kryptograficznego wymaga eksperckiej wiedzy i doświadczenia szczególnego rodzaju. Zamiast próbować podnosić takie umiejętności

5 *https://oreil.ly/bIwkK*

we wszystkich zespołach, większość organizacji tworzy specjalistyczny zespół zabezpieczeń (typu skomplikowanego podystemu), który może zaangażować się w prace poszczególnych zespołów w razie potrzeby.

Platformowy (Platform)
: Podobnie jak zespoły umożliwiające, zespół platformowy zapewnia wsparcie dla reszty organizacji, ale z jedną ważną różnicą – zespoły platformowe dostarczają swoim użytkownikom samoobsługowe możliwości umożliwiające. Podczas gdy zespoły umożliwiające i skomplikowanych podsystemów są ograniczone możliwościami przerobowymi swoich ludzi, zespół platformowy poświęca się budowaniu narzędzi wspomagających i procesów, które można łatwo skalować. Wymaga to więcej wstępnych wysiłków oraz ciągłego utrzymywania i obsługi. Platforma staje się produktem, którego użytkownikami są pozostałe zespoły w organizacji. Na przykład zespoły operacyjne mogą stać się zespołami platformowymi, gdy zaoferują zbudowanie i udostępnienie narzędzi dla zespołów programistycznych.

Znając te cztery typy zespołów możemy zacząć określać, jak nasze zespoły powinny działać. Aby naprawdę przedstawić nasz projekt zespołów, potrzebujemy jeszcze jednej części modelu: sposobów, jakimi zespoły będą wchodzić w interakcje ze sobą nawzajem, czym zajmiemy się w kolejnym podpunkcie.

Tryby interakcji

Cel, który chcemy osiągnąć przy projektowaniu zespołów budujących nasze mikrousługi, to zredukowanie liczby koordynacji, które będą musiały wystąpić, aby praca mogła zostać wykonana. Typy zespołów występujące w Team Topology pomagają zidentyfikować podstawowe charakterystyki zespołu. Aby w pełni zrozumieć, jak i gdzie można zredukować koszty koordynacji, musimy wyartykułować sposoby, jakimi nasze zespoły będą koordynować swoją pracę z innymi. W tym miejscu na scenę wkracza tryb interakcji Team Topology. Skelton i Pais w swojej książce omawiają trzy tryby interakcji, które odpowiadają różnym poziomom koordynacji:

Współdziałanie (Collaboration)
: Ten tryb interakcji wymaga bliskiego współdziałania od obu zespołów. Współpraca zapewnia członkom zespołów możliwości nauki, odkrywania nowych możliwości i wprowadzania innowacji. Wymaga jednak wysokiego poziomu koordynacji ze strony każdego zespołu i jest mało skalowalna. Przykładem tego trybu może być współpraca zespołu zabezpieczeń z zespołem programistów mikrousług w celu wypracowania bezpieczniejszej wersji oprogramowania. W tym przypadku współpraca może obejmować wspólne projektowanie, pisanie i testowanie kodu.

Wspomaganie (Facilitating)
: Interakcja ułatwiająca jest podobna do współdziałania, ale jest jednokierunkowa. Zamiast łącznej pracy zespołów w celu rozwiązania wspólnego problemu jeden zespół

odgrywa rolę wsparcia, pomagając innemu zespołowi w dostarczeniu pożądanych wyników. Przykładem interakcji wspomagającej może być sytuacja, gdy zespół infrastruktury pomaga zespołowi mikrousług zrozumieć i rozwiązać problemy związane z architekturą sieciową udostępnianą przez tę infrastrukturę.

X jako usługa (X-as-a-service)
: Współpraca zespołów przybiera niekiedy formę związku „konsument-dostawca". W interakcji tego typu jeden zespół udostępnia usługi innym zespołom organizacji, przy minimalnym poziomie koordynacji. Występuje to najczęściej w sytuacji, gdy zespół wydaje współdzielony proces, dokument, bibliotekę, API albo platformę. Interakcje typu *X jako usługa* zazwyczaj skalują się bardzo dobrze, gdyż wymagają niewiele koordynacji. W sposób naturalny pasują do zespołów platformowych, ale inne typy zespołów również mogą przyjmować ten tryb. Na przykład umożliwiający zespół architektoniczny może dokumentować listę zalecanych wzorców programowych i udostępniać ją wszystkim zespołom mikrousług w modelu „wzorce jako usługa".

Koncepcja Team Topologies obejmuje znacznie więcej zagadnień, niż tu naszkicowane. Kategoryzacja typów zespołów i trybów interakcji łącznie daje nam rozległą paletę wariantów, których możemy używać do malowania obrazu naszych zespołów, ze szczególnym naciskiem na to, kiedy i jak nasze zespoły będą musiały koordynować swoje działania. W kolejnym podrozdziale wykorzystamy pojęcia zapożyczone z Team Topology, aby zaprojektować model zespołu mikrousług.

Projektowanie topologii zespołu mikrousług

Podejście Team Topology zapewnia nam język, w którym możemy omawiać koordynację pomiędzy zespołami. Tym, co czyni ten język czymś naprawdę szczególnym, jest fakt, że został on zbudowany pod kątem wizualnych reprezentacji. W tym podrozdziale zamierzamy utworzyć projekt naszych zespołów mikrousług, który pokaże, jakich zespołów potrzebujemy i jak będą one współpracować ze sobą. Po ukończeniu tej pracy będziemy mieli diagram wyróżniający najważniejsze punkty koordynacji i interakcji pomiędzy zespołami.

Aby utworzyć projekt zespołu i jego topologię, będziemy wykorzystywać następujące podejście „krok po kroku":

1. Ustanowienie zespołu projektowania systemu.
2. Utworzenie szablonu zespołu mikrousług dla przyszłych zespołów.
3. Zdefiniowanie zespołów platformowych.
4. Dodanie zespołów umożliwiających i skomplikowanych podsystemów.
5. Dodanie kluczowych zespołów konsumenckich.

W miarę przechodzenia przez każdy krok będziemy dokumentować nasz projekt i budować topologię zespołów. W każdym kroku zidentyfikujemy jeden (lub więcej) zespół, utworzymy i wypełnimy dokument projektu zespołu i nakreślimy kluczowe interakcje tego zespołu. Rozpoczniemy od skupienia się na zespole projektowania systemu.

Nie istnieje żadna pojedyncza topologia zespołów, która dobrze pasowałaby do każdej sytuacji. Niemożliwe byłoby wzięcie pod uwagę indywidualnych cech poszczególnych organizacji, takich, jak rozmiar, zatrudnieni ludzie i ich umiejętności, a nade wszystko potrzeby. Topologia, którą tu tworzymy, jest „skonsolidowaną" wersją znanych nam implementacji w skali wielkich przedsiębiorstw, które dobrze się sprawdziły.

Ustanowienie zespołu projektowania systemu

System mikrousług jest złożoną konstrukcją, zawierającą mnóstwo części i wielu pracujących ludzi. Oprogramowanie, które będzie budowane, jest wynikiem kolektywnego podejmowania decyzji i pracy tych wszystkich ludzi. Z naszych doświadczeń wiemy, że doprowadzenie wszystkiego do stanu, gdy działa wspólnie w pożądany sposób, nie jest łatwe. To dlatego konieczne jest wyznaczenie grupy ludzi, którzy ukształtują wizję i zachowanie systemu. W naszym modelu grupę tę nazywamy *zespołem projektowania systemu*.

W naszym modelu zespół projektowania systemu jest odpowiedzialny za trzy kluczowe zagadnienia:

Projektowanie struktur zespołów
: Zespół projektowania systemu jest pierwszym zespołem, który będzie tworzyć. Jest to również zespół, który – jak oczekujemy – zaprojektuje pozostałe zespoły wykonujące pracę przy budowaniu systemu. To praca, którą wykonamy w kolejnych krokach projektowania zespołu. W rezultacie odegramy tu rolę zespołu projektowania systemu.

Zdefiniowanie standardów, bodźców i „poręczy"
: Oprócz formowania zespołów, zespół projektowania systemu powinien określić kształt (rodzaj i zakres) decyzji, które mogą podejmować indywidualne zespoły. Zagwarantuje to osiąganie przez te zespoły wyników pasujących do celów naszego systemu. Jedną z metod osiągnięcia tego jest ustanowienie standardów określających, co zespoły mogą, a czego nie mogą robić. Jest to podejście nakazowe, które przyjęliśmy dla wielu decyzji prezentowanych w tej książce. W praktyce nadmierna standaryzacja jest trudna w utrzymywaniu i może okazać się zbyt restrykcyjna, aby utworzyć zdrowy, sprawny system. Dobrzy projektanci uzyskują bardziej pożądane zachowania, wprowadzając zachęty (bodźce) oraz „poręcze", które działają jako łagodne zalecenia i wskazówki, a nie jako żelazne reguły.

Ciągłe usprawnianie systemu
: Na koniec zespół projektowania systemu powinien stale ulepszać wszystkie wprowadzone wcześniej projekty zespołów, standardów, bodźców i ograniczeń. Aby to osiągnąć, musi ustanowić sposób monitorowania lub mierzenia systemu jako całości, aby móc dokonywać zmian i wprowadzać ulepszenia.

Użyteczne będzie dokumentowanie obowiązków zespołu, aby móc jasno zakomunikować, co robi każdy zespół. W istocie powinniśmy udokumentować wszystkie kluczowe

właściwości naszych zespołów, aby ułatwić ich zrozumienie i ulepszanie ich w miarę ewolucji systemu. Jako minimum, należy określić typ Team Topology, rozmiar zespołu oraz zdefiniowane wcześniej obowiązki.

Rozpoczniemy od wyboru typu zespołu według Team Topology. Po wstępnym skonfigurowaniu projektów zespołów i standardów oczekujemy, że zespół projektowania systemu skupi się na pomaganiu innym w budowaniu mikrousług i wspomagających komponentów. Spodziewamy się, że większość pracy tego zespołu polegać będzie na konsultacjach wspomagających inne zespoły. Tym samym, choć zespół projektowania systemu ma dostarczany przez siebie produkt (czyli cały system), charakter pracy, którą ma wykonać sprawia, że zaliczymy go do typu umożliwiającego.

Będziemy również chcieli, aby zespół projektowania systemu był niewielki. Powinien składać się jedynie z kilku doświadczonych liderów, architektów i projektantów systemu, którzy będą w stanie szybko podejmować wspólne decyzje dotyczące systemu jako całości. Z tego względu ograniczymy liczbę członków zespołu do od trzech do pięciu osób – jeszcze mniej, niż ogólna wielkość zespołu wybrana wcześniej.

Spróbujmy zebrać te decyzje i właściwości zespołu w postaci dokumentu decyzji dla zespołu projektowania systemu. Utworzymy plik o nazwie *system-design-team.md* i wypełnimy go następującą treścią:

```
# Zespół projektowania systemu

## Typ zespołu
Umożliwiający

## Wielkość zespołu
3-5 osób

## Obowiązki
*  Projektowanie struktur zespołów
*  Zdefiniowanie standardów, bodźców i "poręczy"
*  Ciągłe usprawnianie systemu
```

Wygoda używania plików tekstowych do celów dokumentacji zespołów polega na tym, że możemy traktować je tak samo, jak kod. Dzięki temu możemy przechowywać dokumentację w repozytorium kodu i wersjonować ją, ilekroć zajdzie potrzeba dokonywania zmian. Alternatywnie możemy wykorzystać wiki, repozytorium dokumentów albo cokolwiek innego, co najlepiej sprawdzi się w konkretnej firmie. Decyzję o tym, jak zarządzać plikami projektów zespołów, pozostawiamy Czytelnikowi. Wszystkie przykłady projektów zespołów użyte w naszym modelu można znaleźć w repozytorium GitHub tej książki[6].

Po dojściu do tego miejsca zazwyczaj zaczniemy sporządzać wizualny diagram zespołów i odwzorowywać w nim jego interakcje z innymi zespołami systemu. Jest to serce naszej pracy projektowej, pozwalające na wizualizację tego, jak zespoły będą współpracować ze sobą. Możemy na przykład oczekiwać od zespołu projektowania systemu, że będzie używać modelu ułatwiania w interakcjach z zespołami mikrousług. Jednak ponieważ jest

6 *https://oreil.ly/Microservices_UpandRunning_team-designs*

to dopiero pierwszy zdefiniowany zespół, nie mamy jeszcze nic, z czym mógłby on wchodzić w interakcje. Tym samym odłożymy na później pracę nad diagramami.

Po utworzeniu dokumentu dla zespołu projektowania systemu możemy przejść do dokumentowania naszych zespołów mikrousług i tworzenia diagramu interakcji.

Budowanie szablonu zespołu mikrousług

W budowanym przez nas modelu „Up and Running" każda mikrousługa jest własnością określonego zespołu. Ten pojedynczy zespół odpowiada za wszystkie decyzje i pracę dotyczące projektowania, budowania, dostarczenia i utrzymywania mikrousługi. W praktyce pojedynczy zespół może posiadać kilka mikrousług. To zupełnie poprawne podejście, pozwalające uniknąć niepotrzebnego rozrastania się liczby zespołów. Najważniejszym ograniczeniem w tym przypadku jest to, aby odpowiedzialność za konkretną mikrousługę nie rozciągała się na wiele zespołów. Własność mikrousługi musi być przypisana do odpowiedzialnego i wiarygodnego zespołu.

Kluczowa decyzja: Własność mikrousług

Każda mikrousługa będzie własnością pojedynczego zespołu, który będzie ją projektować, budować i uruchamiać. Zespół ten jest odpowiedzialny za mikrousługę przez cały czas jej życia.

Gdy nasz system osiągnie dojrzałość, będziemy mieli w nim bardzo wiele mikrousług. Zapewne będziemy mieli ostatecznie również wiele zespołów mikrousług. Ponieważ oczekujemy, że w naszym systemie będzie wiele takich zespołów, nie będziemy projektować indywidualnie każdego z nich. Zamiast tego zdefiniujemy szablon zespołu mikrousług, który będziemy mogli zastosować do każdego nowego zespołu. Można myśleć o tym jak o utworzeniu wykrawaczki do ciastek, której będziemy używać do „wytłaczania" jakiś zespołów w późniejszym czasie, gdy okażą się potrzebne. Dla osób z przygotowaniem programistycznym bardziej czytelna może być analogia do zdefiniowania „klasy", której „instancje" będziemy tworzyć.

Na początku musimy zrobić to samo, co w przypadku zespołu projektowania systemu – zdefiniować pewne podstawowe właściwości systemu. Tak jak poprzednio, udokumentujemy typ zespołu, jego wielkość i obowiązki. Jak wspomnieliśmy wcześniej, nasze zespoły mikrousług mają niezależnie i wyłącznie posiadać jedną mikrousługę (lub więcej). Ten stan posiadania obejmuje utrzymywanie usługi w działaniu i wydawanie w razie potrzeby ciągłego strumienia usprawnień, poprawek i zmian.

Przy takiej charakterystyce logiczne jest sklasyfikowanie zespołu mikrousług jako zadaniowego (*stream-aligned*). Pozostaniemy tu przy decyzji wymiarowania zespołu, którą przyjęliśmy wcześniej w tym rozdziale i zachowamy wielkość zespołu od pięciu do ośmiu osób. Udokumentujmy te właściwości, podobnie jak poprzednio. Tworzymy plik o nazwie *microservice-team-template.md* i wypełniamy go następującą treścią: