

Budowanie mikroustug

WYKORZYSTAJ POTENCJAŁ ARCHITEKTURY USŁUG!



Tytuł oryginału: Building Microservices

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-1381-1

© 2015 Helion S.A.

Authorized Polish translation of the English edition of Building Microservices, ISBN 9781491950357 © 2015 Sam Newman.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/budmik>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Przedmowa	13
1. Mikrouslugi	19
Czym są mikrouslugi?	20
Niewielkie, skoncentrowane na dobrym wykonywaniu jednej rzeczy	20
Autonomiczne	21
Najważniejsze korzyści	22
Niejednorodność technologii	22
Odporność na błędy	23
Skalowanie	23
Łatwość wdrażania	24
Dopasowanie do organizacji zespołów	25
Interoperatywność	25
Optymalizacja w kierunku wymienności	25
Architektura zorientowana na usługi	26
Inne techniki dekompozycji	27
Biblioteki współdzielone	27
Moduły	28
Nie istnieje panaceum na wszystko	29
Podsumowanie	29
2. Ewolucyjny architekt	31
Niedokładne porównania	31
Ewolucyjna wizja architekta	33
Podział na strefy	34
Pryncypialne podejście	35
Cele strategiczne	36
Zasady	36
Praktyki	37

Łączenie zasad i praktyk	37
Praktyczny przykład	37
Wymagane standardy	38
Monitorowanie	39
Interfejsy	39
Bezpieczeństwo architektury	39
Zarządzanie za pośrednictwem kodu	40
Przykładowe egzemplarze	40
Spersonalizowany szablon usługi	40
Dług techniczny	42
Obsługa wyjątków	42
Zarządzanie i przewodnictwo od środka	43
Budowanie zespołu	44
Podsumowanie	45
3. Jak modelować usługi?	47
Przedstawiamy firmę MusicCorp	47
Co decyduje o tym, że usługa jest dobra?	48
Luźne sprzężenia	48
Wysoka spójność	48
Ograniczony kontekst	49
Modele współdzielone i ukryte	49
Moduły i usługi	51
Przedwczesna dekompozycja	51
Możliwości biznesowe	52
Żółwie aż do spodu	52
Komunikacja w kategoriach pojęć biznesowych	54
Granice techniczne	54
Podsumowanie	55
4. Integracja	57
Poszukiwanie idealnej technologii integracji	57
Unikanie wprowadzania przełomowych zmian	57
Dbanie o niezależność interfejsów API od technologii	57
Dbałość o zapewnienie prostoty usługi dla konsumentów	58
Ukrycie wewnętrznych szczegółów implementacji	58
Interfejs z klientami	58
Wspólna baza danych	59
Komunikacja synchroniczna kontra asynchroniczna	60
Aranżacja kontra choreografia	61

Zdalne wywołania procedur	64
Sprzężenia technologii	64
Wywołania lokalne różnią się od zdalnych	65
Kruchość	65
Czy wywołania RPC są złym rozwiązaniem?	67
REST	67
REST a HTTP	68
Hipermedium jako silnik stanu aplikacji	68
JSON, XML czy coś innego?	70
Uważaj na zbyt wielkie wygodę	71
Wady interfejsu REST przez HTTP	72
Implementacja współpracy asynchronicznej, bazującej na zdarzeniach	73
Opcje wyboru technologii	73
Zawiłości architektur asynchronicznych	74
Usługi jako maszyny stanów	76
Rozszerzenia reaktywne	76
DRY i perypetie wielokrotnego wykorzystania kodu w świecie mikrousług	77
Biblioteki klienckie	77
Dostęp przez referencję	78
Zarządzanie wersjami	80
Odkładaj modyfikowanie interfejsu tak długo, jak to możliwe	80
Wczesne wychwytywanie zmian naruszających zgodność interfejsu	81
Zastosowanie semantycznej kontroli wersji	81
Współistnienie różnych punktów końcowych	82
Korzystanie z wielu równoległych wersji usługi	83
Interfejsy użytkownika	84
W stronę środowiska cyfrowego	85
Ograniczenia	85
Kompozycja interfejsów API	86
Kompozycja fragmentu interfejsu użytkownika	87
Zaplecza dla frontonów	89
Podejście hybrydowe	90
Integracja z oprogramowaniem zewnętrznych producentów	91
Brak kontroli	92
Personalizacja	92
Makaron integracji	92
Personalizacja na własnych warunkach	93
Wzorzec Dusiciel	95
Podsumowanie	96

5. Dzielenie monolitu	97
To wszystko są szwy	97
Podział systemu w firmie MusicCorp	98
Powody dzielenia monolitu	99
Tempo zmian	99
Struktura zespołu	99
Bezpieczeństwo	99
Technologia	100
Splątane zależności	100
Baza danych	100
Zlikwidowanie problemu	100
Przykład: eliminowanie relacji kluczy obcych	101
Przykład: wspólne statyczne dane	103
Przykład: współdzielone dane	104
Przykład: wspólne tabele	105
Refaktoryzacja baz danych	106
Podział na etapy	106
Granice transakcyjne	107
Spróbuj ponownie później	108
Odrzucenie całej operacji	109
Transakcje rozproszone	109
Jakie rozwiązanie wybrać?	110
Raportowanie	111
Bazy danych raportowania	111
Pobieranie danych za pośrednictwem wywołania usługi	112
Pompy danych	114
Alternatywne lokalizacje docelowe	115
Pompa danych sterowana zdarzeniami	116
Pompa danych z kopii zapasowej	117
W stronę czasu rzeczywistego	117
Koszty zmiany	117
Zrozumieć przyczyny	118
Podsumowanie	119
 6. Wdrażanie	 121
Krótkie wprowadzenie do ciągłej integracji	121
Czy rzeczywiście to robisz?	122
Mapowanie ciągłej integracji na mikrousługi	123
Potoki kompilacji a ciągłe dostawy	125
Nieuniknione wyjątki	126

Artefakty specyficzne dla platformy	127
Artefakty systemu operacyjnego	128
Spersonalizowane obrazy	129
Obrazy jako artefakty	131
Serwery niezmiennie	131
Środowiska	131
Konfiguracja usługi	133
Odwzorowanie usługi-host	133
Wiele usług na hoście	134
Kontenery aplikacji	136
Jedna usługa na host	137
Platforma jako usługa	138
Automatyzacja	139
Dwa studia przypadków na potwierdzenie potęgi automatyzacji	139
Od świata fizycznego do wirtualnego	140
Tradycyjna wirtualizacja	140
Vagrant	141
Kontenery w Linuksie	142
Docker	144
Interfejs instalacji	145
Definicja środowiska	146
Podsumowanie	147
7. Testowanie	149
Rodzaje testów	149
Zakres testów	150
Testy jednostkowe	152
Testy usług	152
Testy od końca do końca	153
Kompromisy	153
Ile?	154
Implementacja testów usług	154
Makiety lub namiastki	155
Inteligentniejsza namiastka usługi	155
Kłopotliwe testy od końca do końca	156
Wady testowania od końca do końca	157
Testy kruche i łamliwe	158
Kto pisze te testy?	159
Jak długo?	159
Piętrzące się zaległości	160
Metawersje	161

Testuj ścieżki, a nie historie	161
Testy sterowane potrzebami konsumentów	162
Pact	163
Konwersacje	165
Czy należy używać testów od końca do końca?	165
Testowanie po opublikowaniu systemu do produkcji	166
Oddzielenie wdrożenia od publikacji	166
Publikacje kanarkowe	167
Średni czas do naprawy kontra średni czas między awariami	168
Testy współzależności funkcjonalnych	169
Testy wydajności	170
Podsumowanie	171
8. Monitorowanie	173
Jedna usługa, jeden serwer	174
Jedna usługa, wiele serwerów	174
Wiele usług, wiele serwerów	175
Logi, logi i jeszcze raz logi...	176
Śledzenie metryk dotyczących wielu usług	177
Metryki usług	178
Monitorowanie syntetyczne	178
Implementacja monitorowania semantycznego	179
Identyfikatory korelacji	180
Kaskada	182
Standaryzacja	182
Weź pod uwagę użytkowników	183
Przyszłość	184
Podsumowanie	185
9. Bezpieczeństwo	187
Uwierzytelnianie i autoryzacja	187
Popularne implementacje pojedynczego logowania	188
Brama pojedynczego logowania	189
Szczegółowa autoryzacja	190
Uwierzytelnianie i autoryzacja w trybie usługa-usługa	191
Zezwalaj na wszystko wewnątrz obszaru	191
Podstawowe uwierzytelnianie HTTP(S)	192
Korzystanie z SAML lub OpenID Connect	192
Certyfikaty klienta	193
HMAC przez HTTP	194
Klucze API	195
Problem zastępcy	195

Zabezpieczanie danych w spoczynku	197
Korzystaj ze sprawdzonych sposobów	198
Wszystko dotyczy kluczy	198
Wybierz swoje cele	199
Odszyfruj dane na żądanie	199
Szyfruj kopie zapasowe	199
Obrona wielostrefowa	199
Zapory firewall	199
Rejestrowanie	200
Systemy wykrywania włamań (i zapobiegania im)	200
Podział sieci	200
System operacyjny	201
Praktyczny przykład	201
Bądź oszczędny	204
Czynnik ludzki	204
Złota reguła	204
Wdrażanie zabezpieczeń	205
Zewnętrzna weryfikacja	206
Podsumowanie	206
10. Prawo Conwaya a projektowanie systemów	207
Dowody	207
Organizacje sprzężone luźno i ściśle	208
Windows Vista	208
Netflix i Amazon	208
Co można z tym zrobić?	209
Dostosowanie się do ścieżek komunikacyjnych	209
Własność usługi	210
Powody współdzielenia usług	211
Zbyt trudne do rozdzielania	211
Zespoły funkcyjne	211
Wąskie gardła dostaw	212
Wewnętrzne Open Source	212
Rola opiekunów	213
Dojrzałość	213
Narzędzia	214
Konteksty ograniczone a struktura zespołów	214
Usługa osierocona?	214
Studium przypadku: RealEstate.com.au	215
Odwrócone prawo Conwaya	216
Ludzie	217
Podsumowanie	218

11. Mikrouslugi w projektach dużej skali	219
Awarie zdarzają się wszędzie	219
Jak wiele jest zbyt wiele?	220
Degradowanie funkcjonalności	221
Środki bezpieczeństwa architektury	222
Antykrucha organizacja	224
Limity czasu	225
Bezpieczniki	225
Grodzie	227
Izolacja	229
Idempotencja	229
Skalowanie	230
Zwiększenie rozmiarów	230
Podział obciążeń	231
Rozłożenie ryzyka	231
Równoważenie obciążenia	232
Systemy bazujące na wątkach roboczych	234
Zaczynanie od nowa	235
Skalowanie baz danych	236
Dostępność usługi kontra trwałość danych	236
Skalowanie do obsługi operacji odczytu	236
Skalowanie do obsługi operacji zapisu	237
Wspólna infrastruktura bazy danych	238
CQRS	238
Buforowanie	239
Buforowanie po stronie klienta, na serwerze proxy i po stronie serwera	239
Buforowanie w HTTP	240
Buforowanie operacji zapisu	242
Buforowanie w celu poprawy niezawodności	242
Ukrywanie źródła	242
Zachowaj prostotę	243
Zatrucie pamięcią podręczną: historia ku przestrodze	244
Autoskalowanie	245
Twierdzenie CAP	246
Poświęcenie spójności	247
Poświęcenie dostępności	247
Poświęcenie tolerancji podziału?	248
AP czy CP?	249
To nie jest zasada „wszystko albo nic”	249
Świat rzeczywisty	250

Wykrywanie usług	250
DNS	251
Dynamiczne rejestry usług	252
Zookeeper	252
Consul	253
Eureka	254
Tworzenie własnych rozwiązań	254
Nie zapomnij o ludziach!	255
Dokumentowanie usług	255
Swagger	256
HAL i przeglądarka HAL	256
System samoopisujący się	257
Podsumowanie	258
12. Podsumowanie	259
Zasady dotyczące mikrousług	259
Wzorowanie na koncepcjach działania biznesu	260
Przyjęcie kultury automatyzacji	260
Ukrywanie wewnętrznych szczegółów implementacji	261
Decentralizacja wszystkich operacji	261
Możliwość niezależnej instalacji	262
Izolowanie awarii	262
Łatwe do obserwacji	263
Kiedy nie należy używać mikrousług?	263
Ostatnie słowo	264
Skorowidz	265

Mikrouслуги

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

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

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

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

Czym są mikrouслуги?

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

Niewielkie, skoncentrowane na dobrym wykonywaniu jednej rzeczy

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

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

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

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

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

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

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

Autonomiczne

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

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

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

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

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

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

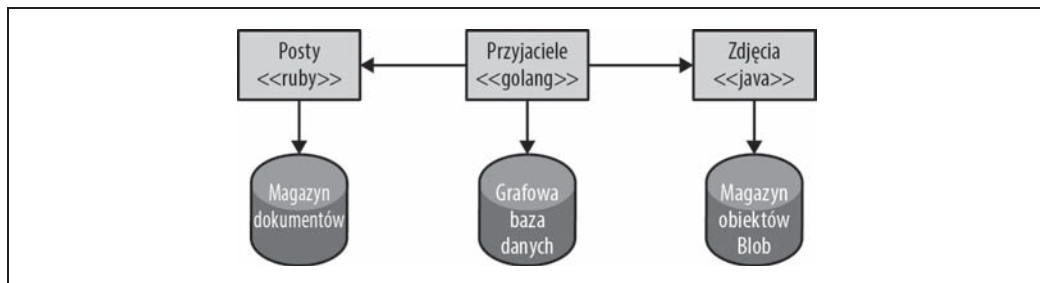
Najważniejsze korzyści

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

Niejednorodność technologii

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

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



Rysunek 1.1. Mikrousługi pozwalają na łatwe wykorzystywanie różnych technologii

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

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

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

Odporność na błędy

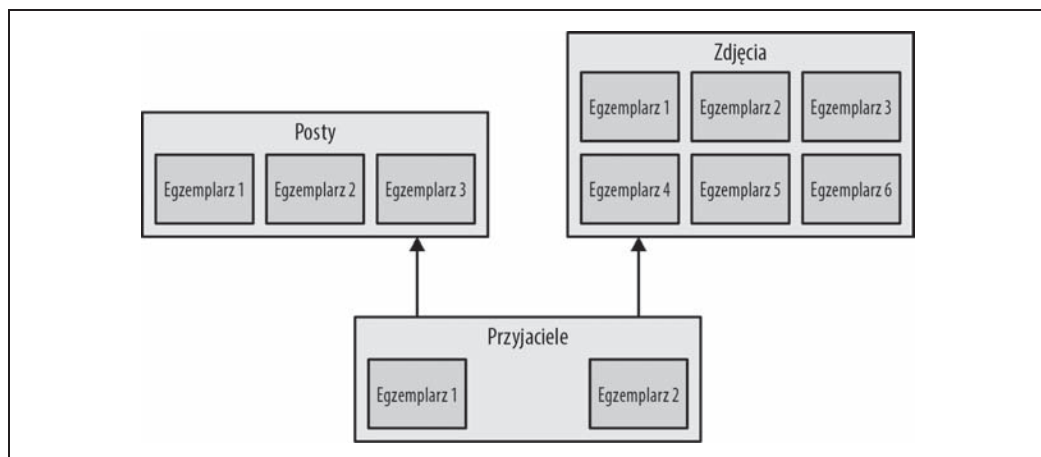
Kluczowym pojęciem w inżynierii odporności na błędy są tzw. przegrody (*bulkhead*). Jeśli jeden składnik systemu ulegnie awarii, ale ta awaria nie rozprzestrzenia się kaskadowo, to można wyizolować problem, a reszta systemu może kontynuować pracę. Granice usług stają się oczywistymi przegrodami. W usłudze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

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

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

Skalowanie

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



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

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

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

Łatwość wdrażania

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

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

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

Dopasowanie do organizacji zespołów

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

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

Interoperatywność

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

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

Optymalizacja w kierunku wymienności

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

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

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

Architektura zorientowana na usługi

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

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

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

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

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

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

Inne techniki dekompozycji

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

Biblioteki współdzielone

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

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

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

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

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

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

Moduły

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

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

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

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

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

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

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

Nie istnieje panaceum na wszystko

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

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

Podsumowanie

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

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

A

- abstrakcje repozytoriów, 49
- aktualizacja tabel, 108
- Amazon, 208
- antykrucha organizacja, 224
- AP, 249
- API, 21
- aplikacja-dusiciel, 222
- aranżacja, 61, 62
- architekt, 32
 - IT, 33
 - kodowania, 35
- architektura
 - na cebulkę, 54
 - SOA, 48
 - zorientowana na usługi, 26
- artefakty
 - specyficzne dla platformy, 127
 - systemu operacyjnego, 128
- atak man-in-the-middle, 191
- atrapy, 155
- automatyzacja, 139, 260
- autonomia, 21, 45
- autoryzacja, 187, 191
 - szczegółowa, 190
- autoskalowanie, 245
- awarie, 219
 - kaskadowe, 182

B

- baza danych, 59, 100
 - raportowania, 111
- bezpieczeństwo, 99, 187
 - architektury, 39, 222

- bezpieczniki, 225
- BFF, backends for frontends, 89
- biblioteki
 - klienckie, 77
 - współdzielone, 27
- blokada usług, 203
- brama pojedynczego logowania, 189
- budowanie zespołu, 44
- buforowanie, 239
 - dla poprawy niezawodności, 242
 - na serwerze proxy, 239
 - operacji zapisu, 242
 - po stronie klienta, 239
 - po stronie serwera, 239
 - w HTTP, 240

C

- CaaS, Containers as a Service, 144
- CD, continuous delivery, 125
- CDC, consumer-driven contract, 162
- cele COBIT, 43
- cele strategiczne, 36
- certyfikaty klienta, 193
- CFR, cross-functional requirements, 169
- chmury VPC, 200
- choreografia, 61, 63
- CI, continuous integration, 121
- ciągła
 - dostawa, CD, 125
 - integracja, CI, 121
- CMS jako usługa, 93
- Conway Melvin, 207
- CP, 249
- CQRS, 238
- CRM, Customer Relationship Management, 94

CRUD, create, read, update, delete, 52
czas reakcji, 221
czynniki ludzkie, 204

D

Datensparsamkeit, 204
DDD, domain-driven design, 19
decentralizacja wszystkich operacji, 261
definicja
 maszyny, 21
 środowiska, 146
degradowanie funkcjonalności, 221
dekompozycja, 51
dług techniczny, 42
długi ogon danych, 113
dni ćwiczeń, 224
DNS, 251
Docker, 144
dokumentowanie usług, 255
dostawca
 tożsamości, 188
 usług, 188
dostęp przez referencję, 78
dostępność, 221, 246, 247
 usługi, 236
DRY, don't repeat yourself, 77
dynamiczne rejestry usług, 252
działanie bezpieczników, 226

E

eliminowanie relacji kluczy obcych, 101
empatia, 45

F

fałszywki, 155
firma MusicCorp, 47
format
 HTML, 71
 JSON, 70
 JSONPATH, 71
 XML, 70
 XPath, 71
framework, 41
 OSGi, 28
 REST, 72
fronton, 89

G

granice
 techniczne, 54
 transakcyjne, 107
 usług, 55
grodzie, 227

H

HAL, Hypertext Application Language, 71
hipermedium, 68
hipernadzorca, 141
HMAC, hash-based messaging code, 194

I

IaaS, Infrastructure as a Service, 144, 177
idempotencja, 229
identyfikator
 korelacji, 180
 URI, 83
IDS, intrusion detection systems, 200
implementacja, 58
 grodzi, 228
 monitorowania semantycznego, 179
 pojedynczego logowania, 188
 RPC, 65
 testów usług, 154
 współpracy asynchronicznej, 73
 zabezpieczeń, 201
informacje o klientach, 59
instalacja
 niezależna, 262
 usługi, 145
integracja, 57, 92
interfejs, 39
 API, 86
 instalacji, 145
 programowania aplikacji, 21
 REST, 72, 96
 RPC, 96
 użytkownika, 84, 87
 z klientami, 58
interoperatywność, 25
IPS, intrusion prevention systems, 200
izolacja, 229
izolowanie awarii, 262

J

Java RMI, 65
jedna usługa, 174
 na host, 137
język
 Erlang, 28
 HAL, 71, 256
JVM, Java Virtual Machine, 23
JWT, JSON Web Token, 194

K

kaskada, 182
klucze, 198
 API, 195
 obce, 101
kompilacja CI, 124
 na mikrousługę, 125
komponenty interfejsu użytkownika, 87
kompozycja interfejsów API, 86
komunikacja, 54
 asynchroniczna, 60, 74
 synchroniczna, 60
konfiguracja
 ostatecznie spójna, 236
 usługi, 133
konta usług, 193
kontekst, 49, 98
 klienta, 105
 ograniczony, 214
kontenery, 142
 aplikacji, 136
kontrola wersji, 81, 92
konwersacje, 165
końcowa spójność, 109
kończenie sesji SSL, 233
kopia zapasowa, 117
korelacje, 180
koszty zmiany, 117
kwadrant testowania, 150

L

limity czasu, 225
LOB, line of business, 215
logi, 176
lokalizacje docelowe, 115
luźne sprzężenia, 48

Ł

łączenie zasad i praktyk, 37

M

mapowanie ciągłej integracji, 123
maskowanie systemu CRM, 95
maszyna
 JVM, 23, 136
 stanów, 76
 wirtualna, 130
mechanizm
 semantycznego monitorowania, 263
 SSO, 189
 wykonawczy, 28
 XPath, 80
menedżer transakcji, 109
metadane, 202
metryki usług, 178
mieszanie hasła z solą, 198
mikrousługi, 13, 19, 123, 260
 w projektach dużej skali, 219
modele
 ukryte, 49
 współdzielone, 49
modelowanie usług, 47, 54
moduły, 28
moduły i usługi, 51
modyfikowanie interfejsu, 80
monitorowanie, 39, 173
 semantyczne, 165, 263
 syntetyczne, 178
monolityczna usługa, 99
możliwości
 biznesowe, 52
 przystosowawcze, 45

N

namiatka usługi, 155
narzędzie
 ELB, 147
 LXC, 142
 Pact, 163
Netflix, 208
Newman Sam, 265
niezależność interfejsów API, 57
niezawodność, 242

0

- obiekty fikcyjne, 155
- obowiązki ewolucyjnego architekta, 45
- obraz maszyny wirtualnej, 130
- obrazy jako artefakty, 131
- obrona wielostrefowa, 190, 199
- obsługa
 - operacji odczytu, 236
 - operacji zapisu, 237
 - systemu e-commerce, 234
 - wyjątków, 42
 - wywołań, 89
- oddzielanie wdrożenia od publikacji, 167
- odporność na błędy, 23
- odrzućcie operacji, 109
- odwrócone prawo Conwaya, 216
- odzworowanie usługi-host, 133
- ograniczenia, 85
- ograniczone konteksty, 49, 53
- opcje wyboru technologii, 73
- OpenID Connect, 192
- operacje CRUD, 52
- oprogramowanie zewnętrznych producentów, 91
- optymalizacja, 25
- organizacja zespołów, 25
- organizacje sprzężone, 208

P

- PaaS, Platform as a Service, 21, 119, 138, 144
- pakiet, 98
- personalizacja, 92, 93
- pierścień, 238
- piramida testów, 151, 163
- platforma
 - jako usługa, PaaS, 21, 119, 138, 144
 - wdrożeń, 141
- pobieranie danych, 112
- podział
 - obciążeń, 231
 - sieci, 200
 - systemu, 98
- pojedyncza
 - operacja, 108
 - transakcja, 108
 - usługa, 174
- pojedyncze logowanie, SSO, 188
- pojęcia biznesowe, 54

- pompa danych, 114
 - sterowana zdarzeniami, 116
 - z kopii zapasowej, 117
- poświęcenie
 - dostępności, 247
 - spójności, 247
 - tolerancji podziału, 248
- potoki kompilacji, 125
- poziom usług, 232
- pozyskiwanie zdarzeń, 239
- praktyki, 37
- prawo Conwaya, 207, 216
- prezentowanie interfejsu użytkownika, 86
- problem
 - wspólnej tabeli, 106
 - zastępcy, 195, 196
 - zdezorientowanego zastępcy, 196
- projekt
 - dużej skali, 219
 - OSGi, 28
 - sterowany testami, TDD, 152
- projektowanie
 - systemów, 207
 - tematyczne, 19
- prostota usługi, 58
- protokół
 - HTTP, 68
 - TLS, 193
- przeglądarka HAL, 256
- przykładowe egzemplarze, 40
- publikacja, 166
 - kanarkowa, 167
 - niebieska-zielona, 166
- punkt końcowy, 82
 - usługi, 65

R

- raportowanie, 111
- RDBMS, 236
- redukcja obciążenia, 228
- refaktoryzacja baz danych, 106
- referencja, 78
- rejestrwanie, 200
- relacje kluczy obcych, 101
- relacyjna baza danych, 236
- replikacja
 - odczytu, 111
 - w trybie multiprimary, 247

repliki odczytu, 236
repozytorium kodu źródłowego, 124, 125
REST, REpresentational State Transfer, 67
rodzaje testów, 149
rozdzielanie warstw repozytorium, 101
rozłożenie ryzyka, 231
rozszerzenia reaktywne, 76
równoległe wersje usługi, 83
równoważenie obciążenia, 232
RPC, Remote Procedure Call, 54, 67
rywalizujący konsument, 74

S

SAML, 188, 192
schemat raportowania, 115
semantyczna kontrola wersji, 81
serwery niezmiennie, 131
sieci WiFi, 202
silnik stanu aplikacji, 68
skalowanie, 23, 230
 baz danych, 236
 operacji odczytu, 236
 operacji zapisu, 237
 systemu, 220
SLA, Service Level Agreement, 232
SOA, service-oriented architecture, 26
SOAP, 64
specyfikacja ATOM, 74
spersonalizowane obrazy, 129
spójność, 246, 247
sprzężenia technologii, 64
SSO, single sign on, 188
standard ATOM, 74
standardy, 38
standaryzacja, 182
stos technologii, 22
stosowanie mikrousług, 22
strefy, 34
struktura zespołu, 99, 214
system
 AP, 247, 249
 bezpieczny, 203
 Consul, 253
 CP, 249
 CRM, 94
 DNS, 251
 Docker, 144
 Eureka, 254

Kibana, 176
kontroli wersji, 92
niezabezpieczony, 202
operacyjny, 201
samoopisujący się, 257
Swagger, 256
Zookeeper, 252
systemy
 wykrywania włamań, IDS, 200
 zapobiegania włamaniom, IPS, 200
szablony usług, 40
szew, seam, 97
szyfrowanie kopii zapasowych, 199

Ś

ścieżki komunikacyjne, 209
śledzenie
 łańcuchów wywołań, 181
 metryk, 177
śnieżny stożek testów, 154
średni
 czas do naprawy, 168
 czas między awariami, 168
środki bezpieczeństwa architektury, 222
środowisko, 131, 146

T

tabele współdzielone, 105
TDD, test-driven design, 152
TDE, Transparent Data Encryption, 198
techniki dekompozycji, 27
technologia, 22, 100
 HAL, 257
 JWT, 194
 REST, 67
 TDE, 198
technologie integracji, 57
tempo zmian, 99
testowanie, 149
 po opublikowaniu systemu, 166
 ścieżki, 161
 według potrzeb konsumenta, 162, 163
testy
 CDC, 162
 dymne, 166
 jednostkowe, 152
 kruche i łamliwe, 158

testy

- od końca do końca, 153, 156
- metawersje, 161
- obsługa standardowa, 157
- używanie, 165
- wady, 157
- usług, 152, 154
- współzależności funkcjonalnych, 169
- wydajności, 170

TLS, Transport Layer Security, 193

tolerancja podziału, 246, 248

transakcja

- kompensacyjna, 109
- rozproszona, 109
- syntetyczna, 179

trwałość danych, 221, 236

twierdzenie CAP, 246

tworzenie klienta, 62, 63

U

ukrywanie

- systemu CMS, 94
- szczegółów implementacji, 261
- źródła, 242

usługa

- AWS, 234
- katalogowa, 188
- monolityczna, 99
- osierocona, 214

usługa-host, 133

usługa-usługa, 191

usługi

- AWS, 143
- jako maszyny stanów, 76
- kleiste, 94
- na hoście, 134
- RESTFul, 71
- sidecar, 41
- w osobnych kontenerach, 143

uwierzytelnianie, 187, 191

- podstawowe HTTP, 192

używanie mikrousług, 263

V

Vagrant, 141

VLAN, 233

VMware, 130

VPC, virtual private cloud, 200

W

wady interfejsu REST, 72

warstwy repozytorium, 101

wąskie gardła dostaw, 212

wątki robocze, 234

wczesne wychwytywanie zmian, 81

wdrażanie, 24, 121, 166

- zabezpieczeń, 205

wewnętrzne Open Source, 212

wiele

- serwerów, 174, 175

- usług, 175, 177

wielokrotne wykorzystanie kodu, 77

Windows Vista, 208

wirtualizacja, 140

- typu 2, 141

wirtualna chmura prywatna, VPC, 200

wizja, 45

własność usługi, 210

wskaźnik MTTR, 171

wspólna baza danych, 59

wspólne statyczne dane, 103

współdzielenie usług, 211

współdzielone dane, 104

współdzielony model, 50

współlistnienie punktów końcowych, 82

współpraca, 45

- sterowana zdarzeniami, 61

wszechobecny język, 49

wyjątki, 126

wykrywanie usług, 250

wymagania нефunkcjonalne, 169

wymogi współzależności funkcjonalnych, CFR, 169

wysoka spójność, 48

wywołania

- lokalne, 65

- RPC, 67

- usług, 112

- zdalne, 65

wzorzec

- BFF, 89

- CQRS, 238

- Dusiciel, 95

Z

zabezpieczanie danych, 197

zakres testów, 150

- jednostkowych, 152, 153

- zależności, 100
- zaplecza dla frontonów, 89, 90
- zapory firewall, 199
- zarządzanie, 40, 43, 45
 - połączeniami SSL, 233
 - wersjami, 80
- zasada pojedynczej odpowiedzialności, 20
- zasady, 36
 - dotyczące mikrousług, 259

- zatrucie pamięcią podręczną, 244
- zatwierdzanie dwufazowe, 109
- zdalne wywoływanie procedury, 54, 64
- zdarzenia, 73, 116
- zespoły funkcyjne, 211
- zestaw testów dymnych, 166
- zewnętrzna weryfikacja, 206
- złota reguła, 204

Ż

- żądanie GET, 241
- żądanie-odpowiedź, 61

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

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

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

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

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

Dzięki tej książce:

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

Helion

37819 numer katalogowy

księgarnia internetowa

<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Sprawdź najnowsze promocje:

- <http://helion.pl/promocje>
- Książki najchętniej czytane:
- <http://helion.pl/bestsellery>
- Zamów informacje o nowościach:
- <http://helion.pl/nawosci>

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

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-1381-1



Informatyka w najlepszym wydaniu

cena: 59,00 zł