

O'REILLY®

Infrastruktura jako kod

Dynamiczne systemy w epoce chmury



Kief Morris

Infrastruktura jako kod

Dynamiczne systemy w epoce chmury

Kief Morris

przekład: Janusz Machowski

APN Promise
Warszawa 2021

O'REILLY®

Infrastruktura jako kod

© 2021 APN PROMISE SA

Authorized translation of English edition of
Infrastructure as Code, Second Edition
ISBN 978-1-098-11467-1

Copyright © 2021 Kief Morris. All rights reserved.

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

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

Wszystkie prawa zastrzeżone. Żadna część niniejszej książki nie może być powielana ani rozpowszechniana w jakiejkolwiek formie i w jakikolwiek sposób (elektroniczny, mechaniczny), włącznie z fotokopiowaniem, nagrywaniem na taśmy lub przy użyciu innych systemów bez pisemnej zgody wydawcy.

Logo O'Reilly jest zarejestrowanym znakiem towarowym O'Reilly Media, Inc. Ilustracja z okładki i powiązane elementy są znakami towarowymi O'Reilly Media, Inc.

Wszystkie inne nazwy handlowe i towarowe występujące w niniejszej publikacji mogą być znakami towarowymi zastrzeżonymi lub nazwami zastrzeżonymi odpowiednich firm odnośnych właścicieli.

Przykłady firm, produktów, osób i wydarzeń opisane w niniejszej książce są fikcyjne i nie odnoszą się do żadnych konkretnych firm, produktów, osób i wydarzeń. Ewentualne podobieństwo do jakiejkolwiek rzeczywistej firmy, organizacji, produktu, nazwy domeny, adresu poczty elektronicznej, logo, osoby, miejsca lub zdarzenia jest przypadkowe i niezamierzone.

APN PROMISE SA dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji. Jednakże nikomu nie udziela się rękojmi ani gwarancji.

APN PROMISE SA nie jest w żadnym wypadku odpowiedzialna za jakiegokolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-442-4 (wyd. drukowane), 978-83-7541-446-2 (ebook)

Projekt okładki: Karen Montgomery
Ilustracje: John Francis Amalanathan
Ilustracja na okładce: Jose Marzan

Przekład: Janusz Machowski
Redakcja: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Spis treści

Przedmowa	xiii
-----------------	------

Podziękowania.....	xxi
--------------------	-----

Część I. Podstawy

1. Co to znaczy infrastruktura jako kod?.....	3
Od epoki żelaza do epoki chmury	4
Infrastruktura jako kod	5
Korzyści z infrastruktury jako kodu	6
Używanie infrastruktury jako kodu do optymalizacji pod kątem zmian.	6
Zarzut: nie dokonujemy zmian tak często, aby była uzasadniona ich automatyzacja.	7
Zarzut: najpierw trzeba utworzyć, a potem automatyzować	7
Zarzut: musimy wybierać między szybkością i jakością	8
Cztery kluczowe wskaźniki	10
Trzy podstawowe praktyki dotyczące infrastruktury jako kodu	11
Podstawowa praktyka: definiowanie wszystkiego jako kodu	11
Podstawowa praktyka: stałe testowanie i dostarczanie wszystkiego na bieżąco .	12
Podstawowa praktyka: tworzenie małych, prostych elementów, które można zmieniać niezależnie.	12
Podsumowanie	12
2. Zasady infrastruktury w epoce chmury	13
Zasada: zakładaj, że systemy są zawodne	13
Zasada: rób tak, aby wszystko było odtwarzalne	14
Pułapka: systemy śnieżynki	15
Zasada: twórz rzeczy zastępowalne	15
Zasada: minimalizuj różnicowanie	16
Dryf konfiguracji	17
Zasada: pilnuj, abyś mógł powtórzyć każdy proces	18
Podsumowanie	20

3. Platformy infrastruktury	21
Części systemu infrastruktury	21
Platformy infrastruktury	22
Zasoby infrastruktury	25
Zasoby obliczeniowe	26
Zasoby pamięci masowej	27
Zasoby sieciowe	28
Podsumowanie	30
4. Podstawowa praktyka: definiuj wszystko jako kod	31
Dlaczego należy definiować infrastrukturę jako kod	31
Co można zdefiniować jako kod	32
Wybieraj narzędzia z eksternalizacją konfiguracji	32
Zarządzaj kodem w systemie kontroli wersji	33
Języki kodowania infrastruktury	34
Skrypty infrastruktury	35
Deklaratywne języki infrastruktury	37
Programowalne, imperatywne języki infrastruktury	39
Języki deklaratywne czy imperatywne do infrastruktury	40
Języki dziedziczne infrastruktury	40
Języki ogólnego przeznaczenia czy DSL infrastruktury	42
Zasady implementacji w przypadku definiowania infrastruktury jako kodu	42
Oddzielaj kod deklaratywny od imperatywnego	43
Traktuj kod infrastruktury jak prawdziwy kod	43
Podsumowanie	44

Część II. Praca ze stosami infrastruktury

5. Tworzenie stosów infrastruktury jako kodu	47
Co to jest stos infrastruktury?	47
Kod stosu	49
Instancja stosu	49
Konfigurowanie serwerów w stosie	49
Języki infrastruktury niskiego poziomu	50
Języki infrastruktury wysokiego poziomu	51
Wzorce i antywzorce konstruowania stosów	52
Antywzorzec: stos monolityczny	52
Wzorzec: stos grupy aplikacji	54
Wzorzec: stos usług	56
Wzorzec: mikrostos	57
Podsumowanie	58

6. Tworzenie środowisk przy użyciu stosów	59
O co chodzi w tych środowiskach	59
Środowiska dostarczania	59
Wiele środowisk produkcyjnych	60
Środowiska, spójność i konfiguracja	61
Wzorce budowania środowisk	62
Antywzorzec: stos wielu środowisk	62
Antywzorzec: środowiska kopiuj-wklej	63
Wzorzec: stos wielokrotnego użytku	65
Tworzenie środowisk z wieloma stosami	67
Podsumowanie	69
7. Konfigurowanie instancji stosu	71
Używanie parametrów stosu do tworzenia unikatowych identyfikatorów	72
Przykładowe parametry stosu	73
Wzorce konfigurowania stosów	74
Antywzorzec: ręczne parametry stosu	74
Wzorzec: zmienne środowiskowe stosu	76
Wzorzec: parametry skryptowe	78
Wzorzec: pliki konfiguracyjne stosu	81
Wzorzec: stos opakowujący	84
Wzorzec: parametry stosu potokowego	87
Wzorzec: rejestr parametrów stosu	90
Rejestr konfiguracji	93
Implementowanie rejestru konfiguracji	93
Jeden czy wiele rejestrów konfiguracji	95
Obsługa wpisów tajnych jako parametrów	96
Szyfrowanie wpisów tajnych	96
Autoryzacja bez wpisu tajnego	97
Wstrzykiwanie wpisów tajnych podczas wykonywania	97
Wpisy tajne jednorazowe	98
Podsumowanie	98
8. Podstawowa praktyka: ciągle testuj i dostarczaj	99
Po co ciągle testować kod infrastruktury?	100
Co oznacza ciągle testowanie	100
Co należy testować w przypadku infrastruktury?	102
Wyzwania związane z testowaniem kodu infrastruktury	104
Wyzwanie: testy kodu deklaratywnego mają często małą wartość	105
Wyzwanie: testowanie kodu infrastruktury jest powolne	107
Wyzwanie: zależności komplikują testowanie infrastruktury	109
Testowanie progresywne	109

Piramida testów	110
Model testowania „ser szwajcarski”	112
Potoki dostarczania infrastruktury	113
Etapu potoku	114
Zakres komponentów testowanych w ramach etapu	115
Zakres zależności używanych w etapie	115
Elementy platformy wymagane przez etap	116
Usługi i oprogramowanie potoków dostarczania	117
Testowanie w środowisku produkcyjnym	119
Czego nie można powielić poza środowiskiem produkcyjnym	120
Zarządzanie ryzykiem testowania w środowisku produkcyjnym	121
Podsumowanie	122
9. Testowanie stosów infrastruktury	123
Przykładowa infrastruktura	123
Przykładowy stos	124
Potok dla przykładowego stosu	125
Etapu testowania offline dla stosów	125
Sprawdzanie składni	126
Statyczna analiza kodu w trybie offline	126
Statyczna analiza kodu z użyciem API	127
Testowanie z użyciem atrapy API	127
Etapu testowania online dla stosów	128
Podgląd: patrzenie, jakie zmiany zostaną dokonane	128
Weryfikacja: stosowanie asercji dotyczących zasobów infrastruktury	129
Wyniki: potwierdzanie prawidłowego działania infrastruktury	131
Używanie warunków początkowych testu do obsługi zależności	132
Atrapy dla zależności nadrzędnych	133
Warunki początkowe testu dla zależności podrzędnych	134
Refaktoryzacja komponentów w celu umożliwienia ich izolowania	135
Wzorce cyklu życia dla testowych instancji stosów	136
Wzorzec: trwały stos testowy	136
Wzorzec: efemeryczny stos testowy	137
Antywzorzec: podwójny etap stosu – trwały i efemeryczny	138
Wzorzec: okresowa odbudowa stosu	140
Wzorzec: ciągle resetowanie stosu	141
Orkiestracja testów	143
Wspomaganie lokalnego testowania	143
Unikanie silnego sprzężenia z narzędziami potoku	144
Narzędzia do orkiestracji testów	144
Podsumowanie	145

Część III. Praca z serwerami i innymi platformami wykonawczymi aplikacji

10. Środowiska wykonawcze aplikacji	149
Infrastruktura natywna dla chmury i oparta na aplikacjach	150
Cele środowiska wykonawczego aplikacji.	151
Wdrażalne części aplikacji.	151
Pakiety wdrożenia	152
Wdrażanie aplikacji na serwerach	153
Pakowanie aplikacji do kontenerów	153
Wdrażanie aplikacji w klastrach serwerów	154
Wdrażanie aplikacji w klastrach aplikacji.	155
Pakiety do wdrażania aplikacji w klastrach	156
Wdrażanie aplikacji bezserwerowych na platformach FaaS.	157
Dane aplikacji.	158
Schematy i struktury danych	158
Infrastruktura magazynu aplikacji natywna dla chmury	159
Łączność aplikacji	159
Odnajdywanie usług	160
Podsumowanie	162
11. Budowanie serwerów jako kodu	163
Co jest trzymane na serwerze.	164
Skąd pochodzą rzeczy trzymane na serwerze	165
Kod konfiguracji serwera	166
Moduły kodu konfiguracji serwera	167
Projektowanie modułów kodu konfiguracji serwera	168
Wersjonowanie i promowanie kodu serwera	169
Role serwerów.	169
Testowanie kodu serwera	171
Testowanie progresywne kodu serwera.	171
Co testować w przypadku kodu serwera.	172
Jak testować kod serwera.	172
Tworzenie nowej instancji serwera	173
Ręczne tworzenie nowej instancji serwera	174
Tworzenie serwera przy użyciu skryptu	175
Tworzenie serwera przy użyciu narzędzia zarządzania stosem	175
Konfigurowanie automatycznego tworzenia serwerów przez platformę	176
Tworzenie serwera przy użyciu sieciowego narzędzia wyposażania.	177
Wstępne budowanie serwerów.	178
Klonowanie serwera „na gorąco”	178
Używanie migawek serwera	179

Tworzenie czystego obrazu serwera	179
Konfigurowanie nowej instancji serwera	180
Smażenie instancji serwera	181
Pieczenie obrazów serwera	182
Łączenie pieczenia i smażenia	182
Stosowanie konfiguracji serwera podczas tworzenia serwera	183
Podsumowanie	184
12. Zarządzanie zmianami w serwerach.	185
Wzorce zarządzania zmianami: kiedy stosować zmiany	186
Antywzorzec: stosowanie każdej zmiany oddzielnie	186
Wzorzec: ciągła synchronizacja konfiguracji	187
Wzorzec: serwer niezmienny	189
Jak stosować kod konfiguracji serwera	191
Wzorzec: wypychanie konfiguracji serwera	192
Wzorzec: pobieranie konfiguracji serwera	193
Pozostałe zdarzenia w cyklu życia serwera	196
Zatrzymywanie i restartowanie instancji serwera	196
Zastępowanie instancji serwera	197
Odzyskiwanie serwera po awarii	198
Podsumowanie	199
13. Obrazy serwerów jako kod	201
Budowanie obrazu serwera	201
Po co budować obraz serwera?	202
Jak zbudować obraz serwera	203
Narzędzia do budowania obrazów serwerów	203
Proces budowania obrazu online	204
Proces budowania obrazu offline	207
Zawartość źródłowa obrazu serwera	208
Budowanie przy użyciu stockowego obrazu serwera	208
Budowanie obrazu serwera od podstaw	209
Pochodzenie obrazu serwera i jego zawartości	209
Zmienianie obrazu serwera	210
Ogrzewanie czy pieczenie świeżego obrazu	210
Wersjonowanie obrazu serwera	211
Aktualizowanie instancji serwera po zmianie obrazu	212
Udostępnianie obrazu serwera do wykorzystania przez wiele zespołów	213
Obsługa istotnych zmian w obrazie	214
Używanie potoku do testowania i dostarczania obrazu serwera	215
Etap budowy obrazu serwera	215
Etap testowania obrazu serwera	217

Etap dostarczania obrazu serwera	218
Używanie wielu obrazów serwera	218
Obrazy serwera dla różnych platform infrastruktury	219
Obrazy serwera dla różnych systemów operacyjnych	219
Obrazy serwera dla różnych architektur sprzętowych	219
Obrazy serwera dla różnych ról	220
Warstwowanie obrazów serwera	220
Współdzielenie kodu przez obrazy serwerów	221
Podsumowanie	222
14. Budowanie klastrów jako kodu	223
Rozwiązania dla klastrów aplikacji	224
Klaster jako usługa	224
Spakowana dystrybucja klastra	225
Topologie stosu dla klastrów aplikacji	226
Stos monolityczny wykorzystujący klaster jako usługę	227
Stos monolityczny dla spakowanego rozwiązania klastrowego	228
Potok dla monolitycznego stosu klastra aplikacji	229
Przykład wielu stosów dla klastra	232
Strategie współdzielenia dla klastrów aplikacji	234
Jeden duży klaster do wszystkiego	235
Oddzielne klastry dla etapów dostarczania	236
Klastry dla nadzoru	237
Klastry dla zespołów	238
Siatka usług	238
Infrastruktura dla bezserwerowej usługi FaaS	240
Podsumowanie	242

Część IV. Projektowanie infrastruktury

15. Podstawowa praktyka: małe, proste elementy	245
Projektowanie pod kątem modularności	246
Cechy dobrze zaprojektowanych komponentów	246
Zasady projektowania komponentów	247
Używanie testowania do podejmowania decyzji projektowych	250
Modularyzacja infrastruktury	250
Komponenty stosów a stosy jako komponenty	250
Używanie serwera w stosie	252
Wytaczanie granic między komponentami	255
Dopasowywanie granic do naturalnych wzorców zmian	256
Dopasowywanie granic do cykli życia komponentów	256

Dopasowywanie granic do struktur organizacyjnych	258
Wytaczanie granic wspierających odporność	258
Tworzenie granic wspierających skalowanie	259
Dopasowywanie granic do zasad bezpieczeństwa i nadzoru	262
Podsumowanie	263
16. Budowanie stosów z komponentów	265
Języki infrastruktury dla komponentów stosu	266
Ponowne używanie kodu deklaratywnego za pomocą modułów	266
Dynamiczne tworzenie elementów stosu za pomocą bibliotek	267
Wzorce dla komponentów stosu	268
Wzorzec: moduł fasady	268
Antywzorzec: moduł zaciemniający	270
Antywzorzec: moduł niewspółdzielony	272
Wzorzec: moduł pakietowy	273
Antywzorzec: moduł spaghetti	274
Wzorzec: jednostka domeny infrastruktury	277
Budowanie warstwy abstrakcji	279
Podsumowanie	280
17. Używanie stosów jako komponentów	281
Wykrywanie zależności między stosami	281
Wzorzec: dopasowywanie zasobów	282
Wzorzec: wyszukiwanie danych w stosie	285
Wzorzec: wyszukiwanie w rejestrze integracji	288
Wstrzykiwanie zależności	290
Podsumowanie	293

Część V. Dostarczanie infrastruktury

18. Organizowanie kodu infrastruktury	297
Organizowanie projektów i repozytoriów	297
Jedno repozytorium czy wiele?	298
Jedno repozytorium na wszystko	298
Oddzielne repozytorium dla każdego projektu (mikrorepo)	300
Wiele repozytoriów z wieloma projektami	301
Organizowanie różnych rodzajów kodu	302
Pliki pomocnicze projektu	302
Testy wielu projektów	303
Dedykowane projekty testów integracji	305
Organizowanie kodu według koncepcji domeny	305

Organizowanie plików z wartościami konfiguracyjnymi	306
Zarządzanie kodem infrastruktury i aplikacji	307
Dostarczanie infrastruktury i aplikacji	308
Testowanie aplikacji razem z infrastrukturą	309
Testowanie infrastruktury przed integracją	310
Używanie kodu infrastruktury do wdrażania aplikacji	310
Podsumowanie	312
19. Dostarczanie kodu infrastruktury	313
Dostarczanie kodu infrastruktury	313
Budowanie projektu infrastruktury	314
Pakowanie kodu infrastruktury jako artefaktu	315
Używanie repozytorium do dostarczania kodu infrastruktury	315
Integrowanie projektów	317
Wzorzec: integracja projektów podczas budowania	319
Wzorzec: integracja projektów podczas dostarczania	322
Wzorzec: integracja projektów podczas stosowania	324
Używanie skryptów do opakowywania narzędzi infrastruktury	327
Zbieranie wartości konfiguracyjnych	328
Upraszczenie skryptów opakowujących	329
Podsumowanie	330
20. Przepływy pracy zespołowej	331
Ludzie	332
Kto pisze kod infrastruktury?	334
Stosowanie kodu do infrastruktury	336
Stosowanie kodu z poziomu lokalnej stacji roboczej	336
Stosowanie kodu z poziomu scentralizowanej usługi	337
Prywatne instancje infrastruktury	338
Gałęzie kodu źródłowego w przepływach pracy	340
Zapobieganie dryfowi konfiguracji	341
Minimalizacja opóźnień automatyzacji	341
Unikanie stosowania ad hoc	342
Ciągłe stosowanie kodu	342
Infrastruktura niezmiennalna	342
Nadzór w przepływie pracy opartym na potoku	343
Reorganizowanie obowiązków	344
Przesunięcie w lewo	345
Przykładowy proces w przypadku infrastruktury jako kodu podlegającej nadzorowi	345
Podsumowanie	346

21. Bezpieczne zmienianie infrastruktury	347
Ograniczanie zasięgu zmiany	347
Małe zmiany	349
Przykład refaktoryzacji	351
Wypychanie niekompletnych zmian do produkcji	352
Instancje równoległe	353
Transformacje kompatybilne wstecz	356
Przełączniki funkcji	357
Zmiana działającej infrastruktury	360
Chirurgia infrastruktury	361
Powiększanie i zmniejszanie	363
Zmiany bez przestojów	366
Ciągłość	367
Ciągłość poprzez zapobieganie błędom	368
Ciągłość poprzez szybkie odzyskiwanie	369
Ciągłe odzyskiwanie po awarii	370
Inżynieria chaosu	371
Planowanie na wypadek awarii	371
Ciągłość danych w zmieniającym się systemie	373
Blokowanie	373
Segregowanie	374
Replikacja	374
Ponowne ładowanie	374
Mieszane podejścia do ciągłości danych	375
Podsumowanie	375
Indeks	377
O autorze	395
Kolofon	396

Przedmowa

Dziesięć lat temu dyrektor ds. informatyki w globalnym banku wyśmiał mnie, gdy zaproponowałem, aby przyjrano się technologiom chmur prywatnych i narzędziom do automatyzacji infrastruktury: „Takie rzeczy mogą być dobre dla start-upów, a my jesteśmy zbyt duzi i mamy zbyt skomplikowane wymagania”. Jeszcze kilka lat temu wiele przedsiębiorstw uważało, że korzystanie z chmur publicznych jest wykluczone.

Dzisiaj technologia chmury jest wszechobecna. Nawet największe, najbardziej zatwardziałe organizacje błyskawicznie przyjmują strategię „najpierw chmura”. Te organizacje, które uznają za niemożliwe korzystanie z chmur publicznych, wdrażają w swoich centrach danych dynamicznie udostępniane platformy infrastrukturalne¹. Możliwości tych platform ewoluują i są ulepszone tak, że trudno je ignorować nie ryzykując pozostania w tyle za innymi.

Chmura i technologie automatyzacji usuwają bariery utrudniające wprowadzanie zmian w systemach produkcyjnych, a to stawia nowe wyzwania. Choć większość organizacji chce przyspieszyć tempo zmian, nie mogą one pozwolić sobie na ignorowanie ryzyka i potrzeby nadzoru. Tradycyjne procesy i techniki bezpiecznej zmiany infrastruktury nie są przystosowane do szybkiego wprowadzania zmian. Tego typu metody pracy ograniczają zwykle korzyści płynące z nowoczesnych technologii „epoki chmury” – spowalniają pracę i szkodzą stabilności².

W rozdziale 1 używam terminów „epoka żelaza” i „epoka chmury” („Od epoki żelaza do epoki chmury” na stronie 4), aby opisać różne filozofie zarządzania infrastrukturą fizyczną, w których poprawianie błędów jest wolne i kosztowne, oraz zarządzanie infrastrukturą wirtualną, w której wykrywanie i naprawa błędów odbywa się szybko.

Narzędzia infrastruktury jako kodu dają możliwość pracy w sposób, który pomaga wprowadzać zmiany częściej, szybciej i skuteczniej, poprawiając ogólną jakość systemów.

1 Na przykład wiele organizacji rządowych i finansowych w krajach, w których nie ma chmury, ma prawny zakaz hostingu danych i transakcji zagranicą.

2 Z badań opublikowanych przez DORA w *State of DevOps Report* (<https://oreil.ly/ysk9n>) wynika, że rozbudowane procesy zarządzania zmianami cechuje słaba jakość pod względem niepowodzenia zmian i innych mierników efektywności dostarczania oprogramowania.

Ale korzyści nie są efektem samych narzędzi. Są wynikiem sposobu używania tych narzędzi. Sztuka polega na tym, aby wykorzystać technologię do wbudowania jakości, niezawodności i zgodności w proces dokonywania zmian.

Dlaczego napisałem tę książkę

Pierwsze wydanie tej książki napisałem dlatego, że nie widziałem nigdzie spójnego zbioru wskazówek, jak zarządzać infrastrukturą jako kodem. Było mnóstwo porad porozrzuconych po blogach, dyskusjach konferencyjnych oraz dokumentacjach różnych produktów i projektów. Ale zwykły praktyk musiał przejrzeć to wszystko, aby ułożyć z tego strategię dla siebie, a większość ludzi po prostu nie ma na to czasu.

Wrażenia z pisania pierwszego wydania były niesamowite. Była to dla mnie okazja do podróży i rozmów z ludźmi na całym świecie o ich własnych doświadczeniach. Rozmowy te dały mi nowe spojrzenie i postawiły przede mną nowe wyzwania. Dowiedziałem się, że pisanie książki, przemawianie na konferencjach i konsultowanie się z klientami sprzyja rozmowom. Jako branża wciąż gromadzimy, udostępniamy i rozwijamy nasze pomysły zarządzania infrastrukturą jako kodem.

Co dodałem i co zmieniłem w tym wydaniu

Zaszły zmiany od pierwszego wydania tej książki w czerwcu 2016. Tamto wydanie miało podtytuł „Zarządzanie serwerami w chmurze”, co odzwierciedlało fakt, że większość automatyzacji infrastruktury do tamtego momentu była skoncentrowana na konfigurowaniu serwerów. Od tamtej pory kontenery i klastry stały się znacznie ważniejsze, a działania dotyczące infrastruktury przeniosły się na zarządzanie kolekcjami zasobów infrastruktury, udostępnianymi przez platformy chmurowe – które nazywam w tej książce *stosami*.

W efekcie to nowe wydanie zawiera szersze omówienie tworzenia stosów, co jest zadaniem narzędzi takich, jak CloudFormation i Terraform. Przyjąłem pogląd, że wykorzystujemy narzędzia zarządzania stosami do tworzenia kolekcji infrastruktur zapewniających środowisko wykonawcze dla aplikacji. Te środowiska wykonawcze mogą obejmować serwery, klastry i bezserwerowe środowiska wykonawcze.

Zmieniłem sporo na podstawie tego, czego się dowiedziałem o zmieniających się wyzwaniach i potrzebach zespołów tworzących infrastrukturę. Jak już wspomniałem w tej przedmowie, kluczową zaletą infrastruktury jako kodu jest według mnie pokazanie bezpiecznej i łatwej zmiany infrastruktury. Uważam, że ludzie nie doceniają wagi tego, myśląc, że infrastruktura to coś, co się tworzy i potem zapomina o tym.

Ale zbyt wiele zespołów, które spotykam, walczy o zaspokojenie potrzeb swoich organizacji; nie potrafią zapewnić wystarczająco szybkiego rozwoju i skalowania, dostarczania oprogramowania ani oczekiwanej niezawodności i bezpieczeństwa. A kiedy zagłębiamy się w szczegóły zadań, okazuje się, że ludzie są przytłoczeni potrzebą aktualizacji, naprawiania i ulepszania swoich systemów. Dlatego poświęciłem temu dwa razy więcej miejsca, czyniąc głównym tematem książki.

W tym wydaniu przedstawione są trzy podstawowe praktyki używania infrastruktury jako kodu, dzięki którym zmiany powinny być łatwe i bezpieczne:

Definiuj wszystko jako kod

To wynika oczywiście z tytułu i pozwala uzyskać powtarzalność i spójność.

Nieustannie testuj i dostarczaj sukcesywnie

Każda zmiana zwiększa bezpieczeństwo. Pozwala również iść naprzód szybciej i z większą pewnością.

Twórz małe, proste elementy, które można zmieniać niezależnie od innych

Zmienianie ich jest łatwiejsze i bezpieczniejsze niż dużych elementów.

Te trzy praktyki wzajemnie się wzmacniają. Kod jest łatwy do śledzenia, wersjonowania i dostarczania między różnymi etapami procesu zarządzania zmianą. Łatwiej jest nieustannie testować mniejsze elementy. Ciągłe, niezależne testowanie każdego elementu zmusza do zachowania luźno sprzężonego projektu.

Te praktyki i szczegóły ich stosowania są znane ze świata tworzenia oprogramowania. W pierwszym wydaniu tej książki przybliżyłem praktyki tworzenia i dostarczania oprogramowania zwinnego. W tym wydaniu przybliżam dodatkowo reguły i praktyki projektowania efektywnego.

W ciągu ostatnich kilku lat widziałem zespoły borykające się z dużymi i skomplikowanymi systemami infrastruktury i dostrzegłem korzyści płynące ze stosowania wiedzy opartej na wzorcach i regułach projektowania oprogramowania, dlatego zamieściłem w tej książce kilka rozdziałów na ten temat.

Zauważyłem też, że organizacja i praca nad kodem infrastruktury stanowi dla wielu zespołów problem, dlatego omówiłem różne bolączki. Opisałem, jak można dobrze organizować bazy kodu, jak zapewniać instancje programowania i testów dla infrastruktury i jak zarządzać współpracą wielu osób, łącznie z tymi, które zajmują stanowiska kierownicze.

Co dalej

Nie sądzę, abyśmy jako branża osiągnęli dojrzałość w zarządzaniu infrastrukturą. Mam nadzieję, że ta książka daje przyzwoity obraz tego, jakie zespoły są dzisiaj skuteczne. I odrobinę aspiracji, co możemy zrobić lepiej.

Jestem przekonany, że w ciągu następnych pięciu lat łańcuchy narzędzi i podejścia będą ewoluować. Możemy zobaczyć więcej języków ogólnego przeznaczenia używanych do tworzenia bibliotek i możemy dynamicznie generować infrastrukturę, zamiast definiować statyczne szczegóły środowisk na niskim poziomie. Z pewnością musimy lepiej zarządzać zmianami w działającej już infrastrukturze. Większość znanych mi zespołów boi się stosować kod do działającej infrastruktury. (Jeden z zespołów określił Terraform jako „Terrorform,” ale użytkownicy innych narzędzi też tak czują).

Czym jest i czym nie jest ta książka

Teza tej książki jest taka, że badanie różnych sposobów wykorzystywania narzędzi do implementowania infrastruktury może pomóc nam poprawić jakość świadczonych usług. Naszym celem jest wykorzystanie szybkości i częstotliwości dostaw do poprawy niezawodności i jakości tego, co dostarczamy.

Dlatego książka mniej się skupia na konkretnych narzędziach, a bardziej na sposobie ich używania.

Chociaż wymieniam przykłady narzędzi do wykonywania konkretnych funkcji, jak konfigurowanie serwerów i udostępnianie stosów, nie ma tu szczegółowych informacji o sposobie korzystania z konkretnych narzędzi albo platform chmurowych. Są natomiast wzorce, praktyki i techniki odpowiednie dla wszystkich narzędzi i platform.

Nie ma w książce przykładów kodu ze świata rzeczywistych narzędzi i chmur. Narzędzia zmieniają się zbyt szybko w tej dziedzinie, aby przykładowy kod zachował aktualność. Natomiast zawarte porady powinny się starzeć wolniej i pasować do różnych narzędzi. Dla zilustrowania koncepcji w przykładach posługuję się pseudokodem i fikcyjnymi narzędziami. Odniesienia do przykładowych projektów i kodu można znaleźć na związanej z książką stronie <https://infrastructure-as-code.com>.

Ta książka nie zawiera wskazówek na temat używania systemu operacyjnego Linux ani konfigurowania klastra Kubernetes czy routingu sieciowego. Zakres książki obejmuje natomiast sposoby udostępniania zasobów infrastruktury w celu utworzenia tych elementów oraz sposoby wykorzystania kodu do ich dostarczania. Pokazuję różne wzorce topologii klastra i podejścia do definiowania klastrów i zarządzania nimi jak kodem. Opisuję wzorce udostępniania, konfigurowania i zmieniania instancji serwerów za pomocą kodu.

Praktyki przedstawione w tej książce należy uzupełnić zasobami konkretnych systemów operacyjnych, technologii klastrów i platform chmurowych. Podkreślam raz jeszcze, ta książka wyjaśnia podejście do wykorzystywania narzędzi i technologii bez względu na to, jakie to są narzędzia.

Książka traktuje również lekko tematy związane z obsługą, takie jak monitorowanie i obserwowanie, agregację dzienników, zarządzanie tożsamościami i inne elementy potrzebne do wspierania usług w środowisku chmurowym. To, co w niej jest, powinno pomóc w zarządzaniu infrastrukturą potrzebną dla tych usług jak kodem, ale szczególnie konkretnych usług są znowu czymś, co można znaleźć w bardziej konkretnych źródłach.

Nieco historii infrastruktury jako kodu

Narzędzia i praktyki infrastruktury jako kodu pojawiły się na długo przed tym terminem. Administratorzy systemów od samego początku używali skryptów do zarządzania systemami. Mark Burgess stworzył pionierski system CFEngine³ w 1993 roku. Po raz pierwszy nauczyłem się praktyk używania kodu do pełnej automatyzacji udostępniania i aktualizacji serwerów z witryny Infrastructures.org na początku lat 2000⁴.

Infrastruktura jako kod rozwijała się wraz z ruchem DevOps. Andrew ClayShafer i Patrick Debois zapoczątkowali ruch DevOps podczas przemówienia na konferencji Agile 2008⁵. Pierwsze przypadki użycia terminu Infrastruktura jako kod pochodzą z wykładu zatytułowanego „Agile Infrastructure”⁶, wygłoszonego przez Claya-Shafera na konferencji Velocity w 2009 roku oraz artykułu John Willisa⁷ podsumowującego ten wykład. Adam Jacob, współzałożyciel firmy Chef, i Luke Kanies, założyciel Puppet, również używali tego zwrotu w tym czasie.

Dla kogo jest ta książka

Książka jest dla osób związanych z udostępnianiem i wykorzystywaniem infrastruktury przeznaczonej do dostarczania i uruchamiania oprogramowania. Czytelnik może mieć doświadczenie w zakresie systemów i infrastruktury lub w tworzeniu i dostarczaniu oprogramowania. Jego dziedziną może być inżynieria, testowanie, architektura albo zarządzanie. Zakładam pewne doświadczenie z chmurą lub infrastrukturą zwirtualizowaną i narzędziami do automatyzacji infrastruktury przy użyciu kodu.

Czytelnicy, dla których infrastruktura jako kod jest czymś nowym, powinni uznać tę książkę za dobre wprowadzenie do tematu, chociaż najwięcej korzyści przyniesie ona osobom znającym działanie platform chmurowych infrastruktury i podstawy co najmniej jednego narzędzia kodowania infrastruktury.

Ci, którzy mają większe doświadczenie z tymi narzędziami, znajdą tu mieszankę znanych oraz nowych koncepcji i podejść. Staram się stworzyć wspólny język oraz przedstawić wyzwania i rozwiązania w taki sposób, który doświadczeni praktycy i zespoły uznają za przydatny.

³ <https://cfengine.com>

⁴ Oryginalna treść nadal była dostępna na tej stronie w lecie 2020 i nie była aktualizowana od 2007 r. (<http://www.infrastructures.org>).

⁵ <https://oreil.ly/ermR3>

⁶ <https://oreil.ly/qnJKX>

⁷ https://oreil.ly/2F6y_

Zasady, praktyki i wzorce

Stosuję terminy *zasady*, *praktyki* i *wzorce* (i *antywzorce*) na określenie podstawowych pojęć. Oto znaczenie każdego z tych terminów:

Zasada

Zasada to reguła, która pomaga wybierać między potencjalnymi rozwiązaniami.

Praktyka

Praktyka to sposób na wdrożenie czegoś. Podana praktyka nie zawsze jest jedynym sposobem na zrobienie czegoś, a nawet może nie być najlepszym sposobem zrobienia tego w konkretnej sytuacji. Należy korzystać z zasad w celu wybrania najbardziej odpowiedniej praktyki w danej sytuacji.

Wzorzec

Wzorzec jest potencjalnym rozwiązaniem problemu. Jest bardzo podobny do praktyki pod tym względem, że różne wzorce mogą być bardziej skuteczne w różnych sytuacjach. Każdy wzorzec jest opisany w formacie, który powinien pomóc w ocenie, na ile jest on stosowny w przypadku danego problemu.

Antywzorzec

Antywzorzec jest potencjalnym rozwiązaniem, którego należy unikać w większości sytuacji. Zwykle jest to coś, co wydaje się dobrym pomysłem lub coś, co zaczynamy realizować nie zdając sobie sprawy z konsekwencji.

Dlaczego nie używam terminu „Najlepsza praktyka”

Ludzie z naszej branży uwielbiają mówić o „najlepszych praktykach”. Problem z tym terminem jest taki, że wywołuje on często przekonanie, że istnieje tylko jedno rozwiązanie problemu, bez względu na kontekst.

Wolę opisywać praktyki oraz wzorce i zwracać uwagę, kiedy są przydatne i jakie mają ograniczenia. Niektóre z nich opisuję jako skuteczniejsze lub bardziej odpowiednie, ale staram się być otwarty na alternatywy. W przypadku praktyk, które uważam za mniej skuteczne, mam nadzieję, że wyjaśniam, skąd się bierze to przekonanie.

Przykłady z ShopSpinner

Do zilustrowania koncepcji zawartych w tej książce posługuję się fikcyjną firmą o nazwie ShopSpinner. ShopSpinner tworzy i uruchamia sklepy internetowe dla swoich klientów.

ShopSpinner wykorzystuje do działania FCS, Fictional Cloud Service (fikcyjną usługę chmurową), publicznego dostawcę IaaS z usługami obejmującymi FSI (Fictional

Server Images) i FKS (Fictional Kubernetes Service). Do definiowania infrastruktury w chmurze i zarządzania nią ShopSpinner używa narzędzia *Stackmaker* – analogicznego do Terraform, CloudFormation i Pulumi. Serwery są konfigurowane za pomocą narzędzia *Servermaker*, podobnego do Ansible, Chef czy Puppet.

Infrastruktura i projekt systemu ShopSpinner mogą się różnić w zależności od tego, do czego są mi potrzebne, podobnie jak składnia kodu i argumenty wiersza polecenia dla jego fikcyjnych narzędzi.

Konwencje stosowane w tej książce

W książce tej stosowane są następujące konwencje typograficzne:

Kursywa

Wskazuje nowe terminy, adresy URL, adresy email, nazwy i rozszerzenia plików.

Czcionka o stałej szerokości

Jest używana w listingach programów, a także w akapitach w przypadku odniesień do elementów programu, takich jak nazwy zmiennych czy funkcji, bazy danych, typy danych, zmienne środowiskowe, instrukcje i słowa kluczowe.

Czcionka o stałej szerokości pogrubiona

Wskazuje polecenia lub inne porcje tekstu, które należy wpisać dosłownie.

Kursywa o stałej szerokości czcionki

Wskazuje tekst, który należy zastąpić wartościami podanymi przez użytkownika lub wartościami wynikającymi z kontekstu.



Ten element oznacza wskazówkę lub sugestię.



Ten element oznacza ogólną uwagę.



Ten element oznacza ostrzeżenie lub przestrożę.

Nauka online od O'Reilly

Od ponad 40 lat firma *O'Reilly Media* udostępnia szkolenia, wiedzę i informacje w zakresie technologii i biznesu, aby pomóc firmom w odniesieniu sukcesu.

Nasza wyjątkowa sieć ekspertów i innowatorów dzieli się swoją wiedzą i doświadczeniem za pośrednictwem książek, artykułów i naszej platformy nauczania online. Platforma nauczania online O'Reilly oferuje dostęp na żądanie do kursów szkoleniowych na żywo, pogłębionych ścieżek nauczania, interaktywnych środowisk kodowania oraz ogromnej kolekcji tekstów i filmów pochodzących od O'Reilly i ponad 200 innych wydawców. Więcej informacji można znaleźć na stronie <http://oreilly.com>.

Jak się z nami skontaktować

Komentarze i pytania dotyczące tej książki prosimy kierować do wydawcy:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (USA lub Kanada)

707-829-0515 (połączenia międzynarodowe lub lokalne)

707-829-0104 (faks)

Książka ma swoją stronę internetową z erratą, przykładami i dodatkowymi informacjami. Jej adres to <https://oreil.ly/infra-as-code-2e>.

Komentarze i pytania techniczne dotyczące książki można wysyłać mailem na adres bookquestions@oreilly.com.

Wiadomości i informacje o naszych książkach i kursach są na stronie <http://oreilly.com>.

Znajdź nas na Facebooku: <http://facebook.com/oreilly>

Śledź nas na Twitterze: <http://twitter.com/oreillymedia>

Oglądaj nas na YouTube: <http://www.youtube.com/oreillymedia>

Podziękowania

Podobnie jak w przypadku pierwszego wydania, książka ta nie jest wyłącznie moim dziełem. Jest to efekt zestawienia i konsolidacji, najlepiej jak umiem, tego, czego się nauczyłem od innych osób – tak wielu, że nie jestem w stanie ich zapamiętać i prawidłowo skojarzyć. Przepraszam i podziękowania dla tych wszystkich, których zapomniałem tutaj wymienić.

Zawsze lubię szukać pomysłów z Jamesem Lewisem; nasze rozmowy oraz jego teksty i przemówienia miały bezpośredni i pośredni wpływ na dużą część tej książki. Życzliwie dzielił się swoim doświadczeniem w projektowaniu oprogramowania i szeroką znajomością wielu innych tematów i wyraził opinię na temat niemal skończonej wersji tej książki. Jego sugestie pomogły uściślić powiązania, które próbowałem nakreślić między inżynierią oprogramowania a infrastrukturą jako kodem.

Martin Fowler od samego początku życzliwie wspierał moje wysiłki. Zawsze inspirowała mnie jego umiejętność czerpania z doświadczeń innych ludzi, wykorzystywania swojej wiedzy i spostrzeżeń oraz przekształcania tego wszystkiego w jasne, pomocne rady.

Thierry de Pauw był bardzo roważnym i pomocnym recenzentem. Przeczytał wiele wersji roboczych i podzielił się swoimi refleksjami, wskazując mi, co uznał za nowe i przydatne, które pomysły pokrywały się z jego własnymi doświadczeniami i które fragmenty nie były dla niego zrozumiałe.

Muszę podziękować Abigail Bangser, Jonowi Barberowi, Maksowi Griffithsowi, Anne Simmons i Claire Walkley za ich zachętę i inspirację.

Osoby, z którymi pracowałem, przekazały mi swoje uwagi i pomysły, które poprawiły jakość książki. James Green podzielił się spostrzeżeniami na temat inżynierii danych i nauczania maszynowego w kontekście infrastruktury. Pat Downey wyjaśnił swój sposób wykorzystania rozszerzania się i kurczenia w odniesieniu do infrastruktury. Vincenzo Fabrizi zwrócił mi uwagę na znaczenie inwersji sterowania dla zależności infrastruktury. Effy Elden to niewyczerpane źródło wiedzy o oprzyrządowaniu infrastruktury. Moritz Heiber bezpośrednio i pośrednio wpłynął na zawartość tej książki, choć nie mam zbyt wielkiej nadziei, że zgadza się z nią w 100%.

W ramach ThoughtWorks miałem okazję do kontaktów i omawiania Infrastruktury jako kodu i tematów pokrewnych z wieloma kolegami i klientami podczas warsztatów,

projektów i na forach internetowych. Niektóre z tych osób to: Ama Asare, Nilakhya Chatterjee, Audrey Conceicao, Patrick Dale, Dhaval Doshi, Filip Fafara, Adam Fahie, John Feminella, Mario Fernandez, Louise Franklin, Heiko Gerin, Jarrad „Barry” Goodwin, Emily Gorcenski, James Gregory, Col Harris, Prince M Jain, Andrew Jones, Aiko Klostermann, Charles Korn, Vishwas Kumar, Punit Lad, Suyu Liu, Tom Clement Oketch, Gerald Schmidt, Boss Supanat Pothivarakorn, Rodrigo Rech, Florian Sellmayr, Vladimir Sneblic, Isha Soni, Widyasari Stella, Paul Valla, Srikanth Venugopalan, Ankit Wal, Paul Yeoh i Jiayu Yi. Dziękuję również Kentowi Spillnerowi – tym razem pamiętam dłaczego.

Mnóstwo osób przejrzało różne wersje robocze tego wydania książki i podzieliło się swoimi uwagami, w tym Artashes Arabajyan, Albert Attard, Simon Bisson, Phillip Campbell, Mario Cecchi, Carlos Conde, Bamdad Dashtban, Marc Hofer, Willem van Ketwich, Barry O’Reilly, Rob Park, Robert Quinlivan, Wasin Watthanasrisong i Rebecca Wirfs-Brock.

Wielkie podziękowania dla Virginii Wilson, mojej redaktorki, która pomogła mi przejść przez długi i wyczerpujący proces powstawania tej książki. Mój kolega, John Amalanathan, z wielką cierpliwością i starannością zamienił moje mizerne diagramy w zgrabną grafikę, którą można tutaj oglądać.

Mój pracodawca, firma ThoughtWorks, stanowiła ogromne wsparcie. Po pierwsze, tworząc dla mnie środowisko, w którym mogłem się uczyć od fenomenalnych ludzi, po drugie, wspierając kulturę, która zachęca jej członków do dzielenia się pomysłami z branżą, a po trzecie, wspierając mnie podczas pracy z innymi osobami z ThoughtWorks oraz klientami w celu odkrywania i testowania nowych sposobów pracy. Ashok Subramanian, Ruth Harrison, Renee Hawkins, Ken Mugrage, Rebecca Parsons i Gayathri Rao, między innymi, pomogli mi uczynić z tego coś więcej niż osobisty projekt.

Na koniec i najbardziej dziękuję moim ukochanym, Ozlem i Erelowi, którzy wytrzymali moją obsesję na punkcie tej książki. Kolejny raz.

CZĘŚĆ I

Podstawy

Co to znaczy infrastruktura jako kod?

Jeśli ktoś pracuje w zespole tworzącym i obsługującym infrastrukturę IT, to technologie chmury i automatyzacji infrastruktury powinny mu pomagać w dostarczaniu bardziej wartościowych produktów w krótszym czasie i w sposób bardziej niezawodny. Ale w praktyce powodują wzrost wielkości, złożoności i różnorodności zarządzanych obiektów.

Technologie te stają się szczególnie istotne wraz z cyfryzacją organizacji. „Cyfryzacja” to termin, za pomocą którego ludzie biznesu tłumaczą, że systemy oprogramowania są niezbędne do tego, co robią ich organizacje¹. Przejście na technologię cyfrową zwiększa presję, aby robić więcej i szybciej. Trzeba dodawać i obsługiwać więcej usług. Więcej działań biznesowych. Więcej pracowników. Więcej klientów, dostawców i innych interesariuszy.

Narzędzia chmury i automatyzacji są pomocne, ponieważ znacznie ułatwiają dodawanie i zmienianie infrastruktury. Jednak wiele zespołów ma trudności, aby znaleźć dość czasu na nadążenie za już posiadaną infrastrukturą. Ułatwienie tworzenia jeszcze większej liczby rzeczy do zarządzania wcale nie poprawia sytuacji. Jak powiedział mi jeden z klientów: „Użycie chmury zburzyło ściany, za którymi krył się pożar opon”².

Wiele osób reaguje na groźbę chaosu zaostrzając procesy zarządzania zmianami. Mają one nadzieję, że zapobiegną chaosowi ograniczając i kontrolując zmiany. W rezultacie zakuwają chmurę w łańcuchy.

Są z tym dwa problemy. Po pierwsze, traci się korzyści płynące z używania technologii chmury; a po drugie, użytkownicy *chcą* czerpać korzyści z tej technologii. Dlatego użytkownicy unikają osób usiłujących ograniczyć chaos. W najgorszych przypadkach ludzie całkowicie ignorują zarządzanie ryzykiem, uznając, że nie ma to znaczenia w nowym,

1 Jest to przeciwieństwo tego, co wiele z tych osób mówiło kilka lat temu o oprogramowaniu, że „nie jest częścią naszej podstawowej działalności”. Po zastosowaniu się do tej rady i outsourcingu IT, organizacje zdały sobie sprawę, że zostały wyprzedzone przez inne, zarządzane przez ludzi postrzegających lepsze oprogramowanie jako sposób na konkurowanie, a nie koszty do obciążenia.

2 Jak podaje Wikipedia (https://oreil.ly/1kDu_), pożar opon (ang. *tire fire*) miewa dwie formy: „Zdarzenie prowadzące do niemal natychmiastowej utraty kontroli albo powolna piroliza, która może trwać ponad dekadę”.

wspinałym świecie chmury. Efektem tego jest IT rodem z Dzikiego zachodu, co stwarza kolejne problemy³.

Założeniem tej książki jest to, że można wykorzystać technologie chmury i automatyzacji do łatwego, bezpiecznego, szybkiego i odpowiedzialnego wprowadzania zmian. Te korzyści nie wyskakują z pudełka z narzędziami do automatyzacji ani z platform chmurowych. Zależą od sposobu korzystania z technologii.



DevOps a infrastruktura jako kod

DevOps to ruch mający na celu zredukowanie barier i tarcia między silosami organizacyjnymi – działem rozwoju, operacji i innymi interesariuszami zaangażowanymi w planowanie, tworzenie i uruchamianie oprogramowania. Chociaż technologia jest najbardziej widocznym i pod pewnymi względami najprostszym obliczem DevOps, kultura tego ruchu, ludzie i procesy mają największy wpływ na jego rozwój i skuteczność. Technologia i praktyki inżynierskie, podobnie jak infrastruktura jako kod, powinny być wykorzystywane do wspierania wysiłków na rzecz wypełniania luk i poprawy współpracy.

W tym rozdziale wyjaśniam, że nowoczesna, dynamiczna infrastruktura wymaga nastawienia na „epokę chmury”. Takie nastawienie różni się zasadniczo od tradycyjnego podejścia „epoki żelaza”, stosowanego w statycznych systemach z czasów poprzedzających chmurę. Definiuję trzy podstawowe praktyki implementowania infrastruktury jako kodu: definiowanie wszystkiego jako kodu, nieustanne testowanie wszystkiego i dostarczanie na bieżąco oraz tworzenie systemu z małych, luźno sprzężonych elementów.

W tym rozdziale opisuję również powody stojące za podejściem „epoki chmury” do infrastruktury. Takie podejście odrzuca fałszywą dychotomię kompromisu między szybkością i jakością. Zamiast tego używa szybkości jako sposobu na poprawę jakości, a jakości jako sposobu na umożliwienie szybkiej dostawy.

Od epoki żelaza do epoki chmury

Technologie epoki chmury umożliwiają szybsze od tradycyjnych, z epoki żelaza, udostępnianie i zmienianie infrastruktury (tabela 1-1).

Jednak technologie te niekoniecznie ułatwiają zarządzanie systemami i ich rozwijanie. Przeniesienie systemu z długim technicznym⁴ do nieograniczonej infrastruktury chmury przyspiesza chaos.

3 Przez „IT z Dzikiego zachodu” rozumiem osoby budujące systemy IT bez żadnej konkretnej metody ani brania pod uwagę przyszłych konsekwencji. Często ludzie, którzy nigdy nie obsługiwali systemów produkcyjnych, wybierają najkrótszą drogę realizacji bez uwzględniania kwestii bezpieczeństwa, konserwacji, wydajności i innych problemów związanych z obsługą.

4 <https://oreil.ly/3AqHB>

Tabela 1-1 *Zmiany technologii w Epoce chmury*

Epoka żelaza	Epoka chmury
Sprzęt fizyczny	Zasoby wirtualne
Udostępnianie zajmuje tygodnie	Udostępnianie zajmuje minuty
Procesy ręczne	Procesy zautomatyzowane

Być może przydałby się sprawdzony, tradycyjny model nadzoru, aby kontrolować szybkość i chaos uwalniany przez nowsze technologie. Dokładny, wstępny projekt, rygorystyczny przegląd zmian i ściśle rozdzielone obowiązki narzucają porządek! Niestety, modele te są optymalne dla epoki żelaza, w której zmiany są powolne i kosztowne. Przysparzają z góry dodatkową pracę, licząc na skrócenie czasu potrzebnego na późniejsze wprowadzanie zmian. Prawdopodobnie ma to sens, ponieważ późniejsze wprowadzanie zmian jest powolne i kosztowne. Ale chmura sprawia, że zmiany są tanie i szybkie. Należy wykorzystać tę szybkość, aby stale uczyć się i ulepszać swój system. Sposoby pracy w epoce żelaza nakładają ogromny podatek od nauki i doskonalenia.

Zamiast używać powolnych procesów epoki żelaza z szybko zmieniającą się technologią epoki chmury, należy przestawić się na nowy sposób myślenia i wykorzystać szybszą technologię do ograniczenia ryzyka i poprawy jakości. Aby to osiągnąć, konieczna jest fundamentalna zmiana podejścia i nowy sposób myślenia o zmianach i ryzyku (tabela 1-2).

Tabela 1-2 *Sposoby pracy w epoce chmury*

Epoka żelaza	Epoka chmury
Koszt zmian jest wysoki	Koszt zmian jest niski
Zmiany oznaczają porażkę (zmiany muszą być „zarządzane”, „kontrolowane”)	Zmiany oznaczają naukę i ulepszanie
Ograniczenie możliwości niepowodzenia	Maksymalizacja szybkości ulepszania
Dostarczanie dużych porcji, testowanie na końcu	Dostarczanie małych zmian, ciągłe testowanie
Długie cykle wydań	Krótkie cykle wydań
Architektury monolityczne (mniej liczne, większe części)	Architektury mikrouslug (bardziej liczne, mniejsze części)
Konfiguracja fizyczna lub oparta na GUI	Konfiguracja jako kod

Infrastruktura jako kod to podejście epoki chmury do zarządzania systemami, które podlegają ciągłym zmianom w celu zapewnienia wysokiej niezawodności i jakości.

Infrastruktura jako kod

Infrastruktura jako kod to podejście do automatyzacji infrastruktury oparte na praktykach zaczerpniętych z programowania. Kładzie ono nacisk na spójne, powtarzalne procedury udostępniania i zmiany systemów i ich konfiguracji. Najpierw wprowadza się

zmiany w kodzie, a potem wykorzystuje automatyzację do testowania i wprowadzenia tych zmian w systemach.

W całej tej książce wyjaśniam, jak wykorzystywać praktyki programowania zwinnego, takie jak TTD (Test Driven Development), CI (Continuous Integration) i CD (Continuous Delivery), aby zmiany infrastruktury były szybkie i bezpieczne. Opisuję również, jak nowoczesny projekt oprogramowania może zapewnić odporną, dobrze utrzymaną infrastrukturę. Takie praktyki i podejścia do projektowania wzajemnie się wzmacniają. Dobrze zaprojektowana infrastruktura jest łatwiejsza do testowania i dostarczania. Zautomatyzowane testowanie i dostarczanie przyczyniają się do prostszych i bardziej czytelnych projektów.

Korzyści z infrastruktury jako kodu

Podsumowując, organizacje wdrażające infrastrukturę jako kod do zarządzania dynamiczną infrastrukturą mają nadzieję na osiągnięcie korzyści, takich jak:

- Wykorzystywanie infrastruktury IT jako czynnika umożliwiającego szybkie dostarczanie
- Zmniejszenie wysiłku i ryzyka wprowadzania zmian w infrastrukturze
- Umożliwienie użytkownikom infrastruktury uzyskania zasobów, gdy będą im potrzebne
- Zapewnienie wspólnych narzędzi dla działu programowania, operacyjnego i innych interesariuszy
- Tworzenie systemów niezawodnych, bezpiecznych i opłacalnych
- Uwidocznienie nadzoru, bezpieczeństwa i kontroli zgodności
- Poprawa szybkości rozwiązywania problemów i usuwania awarii

Używanie infrastruktury jako kodu do optymalizacji pod kątem zmian

Biorąc pod uwagę, że zmiany stanowią największe ryzyko dla systemu produkcyjnego, a ciągle zmiany są nieuniknione i wprowadzanie ich jest jedynym sposobem na ulepszanie systemu, sensowne jest zoptymalizowanie zdolności do wprowadzania zmian zarówno szybko, jak i niezawodnie. Potwierdzają to badania opisane w raporcie *Accelerate State of DevOps*. Częste i niezawodne wprowadzanie zmian jest skorelowane z sukcesem organizacyjnym⁵.

Kiedy zalecam jakiemuś zespołowi zaimplementowanie automatyzacji w celu optymalizacji pod kątem zmian, spotykam się z kilkoma zarzutami. Uważam, że wynikają one z niezrozumienia, jak można i należy stosować automatyzację.

5 Raporty z badań *Accelerate* są dostępne w corocznym raporcie *State of DevOps* (<https://oreil.ly/0Q3FE>) oraz w książce *Accelerate* autorstwa Dr. Nicole Forsgren, Jez Humble i Gene Kim (IT Revolution Press).

Zarzut: nie dokonujemy zmian tak często, aby była uzasadniona ich automatyzacja

Chcemy myśleć, że jak zbudujemy system, to na tym koniec. Tak patrząc zakładamy, że nie będziemy wprowadzać wielu zmian, więc ich automatyzacja jest stratą czasu.

W rzeczywistości bardzo niewiele systemów przestaje się zmieniać, zanim zostaną wycofane. Niektórzy ludzie uważają, że aktualny poziom zmian jest tymczasowy. Inni z kolei opracowują skomplikowane procesy zgłaszania potrzeby zmian, aby zniechęcić do wysuwania takich żądań. Ci ludzie przeczą faktom. Większość zespołów, które aktywnie wspierają używane systemy, obsługuje ciągły strumień zmian.

Rozważmy typowe przykłady zmian infrastruktury:

- Ważna funkcja nowej aplikacji wymaga dodania nowej bazy danych.
- Nowa funkcja aplikacji wymaga aktualizacji serwera aplikacji.
- Poziom wykorzystania rośnie szybciej niż oczekiwano. Potrzebne są dodatkowe serwery, nowe klastry oraz bardziej rozbudowana sieć i większa pamięć.
- Profilowanie wydajności pokazuje, że obecna architektura wdrażania aplikacji ogranicza wydajność. Trzeba ponownie wdrożyć aplikacje na różnych serwerach aplikacji. To wymaga zmian w klastrach i w architekturze sieci.
- Niedawno ogłoszono lukę w zabezpieczeniach pakietów systemowych naszego systemu operacyjnego. Trzeba zaktualizować dziesiątki serwerów produkcyjnych.
- Trzeba zaktualizować serwery z przestarzałą wersją systemu operacyjnego i pakietami krytycznymi.
- Serwery WWW ulegają sporadycznym awariom. Aby zdiagnozować problem, trzeba wykonać szereg zmian konfiguracyjnych. Następnie trzeba zaktualizować moduł, aby usunąć problem.
- Odkryliśmy zmianę konfiguracji, która poprawi wydajność naszej bazy danych.

Fundamentalna prawda o epoce chmury brzmi: *stabilność jest wynikiem wprowadzania zmian*. Systemy bez poprawek nie są stabilne; są podatne na błędy. Jeśli nie możemy rozwiązywać problemów zaraz po ich wykryciu, to nasz system nie jest stabilny. Jeśli nie możemy przywrócić działania zaraz po awarii, to nasz system nie jest stabilny. Jeśli wprowadzanie zmian powoduje dłuższe przestoje, to nasz system nie jest stabilny. Jeśli zmiany mają często błędy, to nasz system nie jest stabilny.

Zarzut: najpierw trzeba utworzyć, a potem automatyzować

Rozpoczęcie pracy z infrastrukturą jako kodem to stroma krzywa. Skonfigurowanie narzędzi, usług i metod działania w celu zautomatyzowania dostarczania infrastruktury oznacza mnóstwo pracy, zwłaszcza jeśli wdrażamy również nową platformę infrastruktury. Wartość tej pracy jest trudna do wykazania przed rozpoczęciem tworzenia i wdrażania usług za pomocą nowej infrastruktury. A nawet wtedy wartość może nie być widoczna dla osób, które nie pracują bezpośrednio z tą infrastrukturą.

Interesariusze często wywierają presję na zespoły zajmujące się infrastrukturą, aby szybko i ręcznie tworzyły nowe systemy hostowane w chmurze, a ich automatyzację odkładały na później.

Istnieją trzy powody, dla których późniejsza automatyzacja jest złym pomysłem:

- Automatyzacja powinna umożliwiać szybsze dostarczanie, nawet w przypadku nowych rzeczy. Wdrożenie automatyzacji po wykonaniu większości prac powoduje utratę wielu korzyści.
- Automatyzacja ułatwia pisanie zautomatyzowanych testów tego, co robimy. Ułatwia też szybką naprawę i przebudowę w przypadku wykrycia problemów. Wykonanie tego w ramach procesu tworzenia pomaga uzyskać lepszą infrastrukturę.
- Automatyzacja istniejącego systemu jest bardzo trudna. Automatyzacja to część projektowania i implementacji systemu. Aby dodać automatyzację do systemu zbudowanego bez niej, trzeba znacząco zmienić projekt i implementację tego systemu. Dotyczy to również zautomatyzowanego testowania i wdrażania.

Infrastruktura chmury utworzona bez automatyzacji staje się kupą złomu szybciej niż się spodziewamy. Koszt ręcznej konserwacji i naprawy takiego systemu potrafi szybko rosnąć. A jeśli usługa, którą system zapewnia, cieszy się powodzeniem, interesariusze będą naciskać na rozszerzanie i dodawanie funkcji, zamiast pozwolić na przerwę i jego przebudowę.

To samo dotyczy tworzenia systemu w ramach eksperymentu. Gdy mamy już gotowy dowód słuszności naszej koncepcji, pojawia się presja, aby przejść do następnej rzeczy, zamiast cofnąć się i zbudować go dobrze. Tak naprawdę automatyzacja powinna być częścią eksperymentu. Jeśli zamierzamy używać automatyzacji do zarządzania infrastrukturą, musimy rozumieć, jak będzie działać, więc powinna być częścią dowodu słuszności koncepcji.

Rozwiązaniem jest stopniowe tworzenie systemu, z automatyzacją na bieżąco. Należy zapewnić stały strumień wartości, jednocześnie budując możliwość robienia tego w sposób ciągły.

Zarzut: musimy wybierać między szybkością i jakością

Myślenie, że można poruszać się szybko tylko kosztem jakości, a jakość można zapewnić tylko poruszając się powoli, jest całkiem naturalne. Można postrzegać to jako kontinuum przedstawione na rysunku 1-1.



Rysunek 1-1 *Pomysł, że szybkość i jakość są na dwóch przeciwległych końcach widma, jest fałszywą dychotomią*

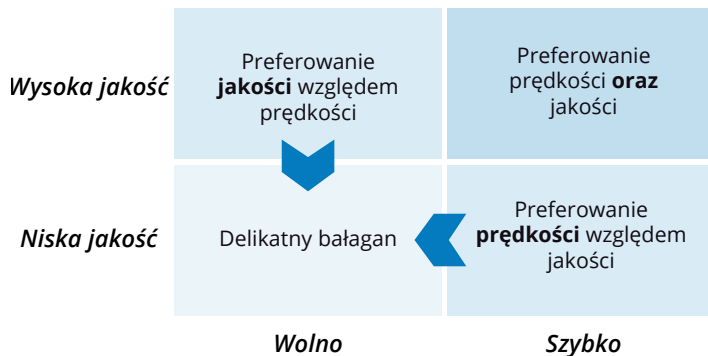
Jednak badanie *Accelerate*, o którym wspomniałem wcześniej („Używanie infrastruktury jako kodu do optymalizacji pod kątem zmian” na stronie 6) dowodzi czegoś przeciwnego:

Wyniki te pokazują, że nie ma kompromisu między poprawą wydajności a osiągnięciem wyższego poziomu stabilności i jakości. W rzeczywistości dobrzy wykonawcy radzą sobie lepiej pod każdym względem. To jest dokładnie to, co przewidują ruchy Agile i Lean, ale wiele dogmatów w naszej branży nadal opiera się na fałszywym założeniu, że poruszanie się szybciej odbywa się kosztem innych aspektów wydajności, a nie umożliwia je i wspiera.

dr Nicole Forsgren, Accelerate

Krótko mówiąc, organizacje nie wybierają między optymalizacją pod kątem zmian a stabilnością. W efekcie są albo dobre w jednym i drugim albo w obu złe.

Wolę postrzegać jakość i szybkość jako kwadrant niż kontinuum⁶, jak to jest pokazane na rysunku 1-2.



Rysunek 1-2 Kwadrant szybkości i jakości

Ten model kwadrantu pokazuje, dlaczego próba wyboru między szybkością i jakością prowadzi do przeciętności pod obu względami:

Prawa dolna ćwiartka: ważniejsza jest szybkość niż jakość

To jest filozofia „działaj szybko i taranuj”. Zespoły stosujące optymalizację pod kątem szybkości kosztem jakości tworzą niechlujne i kruche systemy. Wpadają do lewego dolnego kwadrantu, ponieważ ich tandetne systemy spowalniają ich pracę. Wiele startupów, które od jakiegoś czasu działają w ten sposób, narzeka na utratę „atrakcyjności”. Proste zmiany, które w danych czasach wymagałyby chwili, teraz zajmują dni i tygodnie, ponieważ system jest zagmatwanym kłębkim.

⁶ Tak, pracuję w firmie konsultingowej, dlaczego pytasz?

Lewa górna ćwiartka: ważniejsza jest jakość niż szybkość

Podejście znane również jako „robimy rzeczy poważne i doniosłe, musimy więc robić je *właściwie*”. Ale presja terminu wykonania zmusza do „obejść”. Duże procesy stawiają bariery przed ulepszaniem, więc dług techniczny rośnie wraz z listami „znanych problemów”. Takie zespoły wpadają do lewego dolnego kwadrantu. Kończą z systemami niskiej jakości, *ponieważ* zbyt trudno je ulepszać. Generują jeszcze więcej procesów w odpowiedzi na awarie. Procesy te jeszcze bardziej utrudniają wprowadzanie ulepszeń i zwiększają kruchość i ryzyko. Prowadzi to do zwiększenia liczby niepowodzeń i procesów. Wiele osób pracujących w organizacjach działających w ten sposób uznaje to za normalne⁷, zwłaszcza ci, którzy pracują w branżach uwzględniających ryzyko⁸.

Prawy górny kwadrant jest celem nowoczesnego podejścia, takiego jak Lean, Agile czy DevOps. Możliwość szybkiego poruszania się i utrzymania wysokiego poziomu jakości może wydawać się fantazją. Jednak badanie Accelerate udowadnia, że wiele zespołów potrafi to osiągnąć. W tym kwadrancie można znaleźć „najlepszych wykonawców”.

Cztery kluczowe wskaźniki

Zespół badawczy DORA z *Accelerate* wymienia cztery kluczowe wskaźniki dostarczania oprogramowania i wydajności operacyjnej⁹. Badanie obejmowało różne miary i okazało się, że te cztery mają najsilniejszy związek z tym, jak dobrze organizacja realizuje swoje cele:

Czas realizacji dostawy

Czas potrzebny na implementację, przetestowanie i dostarczenie zmian do systemu produkcyjnego

Częstotliwość wdrażania

Jak często są wdrażane zmiany w systemach produkcyjnych

Procent nieudanych zmian

Jaki procent zmian powoduje pogorszenie jakości usługi lub konieczność natychmiastowej korekty, takiej jak wycofanie lub poprawka awaryjna

Średni czas naprawy (Mean Time to Repair – MTTR)

Ile czasu zajmuje przywrócenie usługi w przypadku nieplanowanego przestoju lub jej utraty

⁷ Jest to przykład „normalizacji dewiacji”, ponieważ ludzie przyzwyczajają się do sposobów pracy zwiększających ryzyko. Diane Vaughan zdefiniowała ten termin w *The Challenger Launch Decision* (University Of Chicago Press).

⁸ To ironiczne (i przerażające), że tak wielu ludzi w branżach takich jak finanse, administracja i opieka zdrowotna uważają delikatne systemy IT – oraz procesy, które utrudniają ich ulepszenie – za normalne, a nawet pożądane.

⁹ DORA, obecnie część firmy Google, jest zespołem, który opracował raport *Accelerate State of DevOps* (<https://oreil.ly/ysk9n>).

Organizacje, które osiągają dobre wyniki w realizacji swoich celów – bez względu na to, czy chodzi o przychód, cenę akcji czy inne kryteria – równie dobrze wypadają pod względem tych czterech wskaźników i na odwrót. Pomysły zawarte w tej książce mają na celu pomóc zespołom i organizacjom osiągnąć dobre wartości tych wskaźników. Ułatwią to trzy podstawowe praktyki dotyczące infrastruktury jako kodu.

Trzy podstawowe praktyki dotyczące infrastruktury jako kodu

Koncepcja epoki chmury wykorzystuje dynamiczny charakter nowoczesnej infrastruktury i platform aplikacji do częstego i niezawodnego wprowadzania zmian. Infrastruktura jako kod to podejście do tworzenia infrastruktury, które obejmuje ciągłe zmiany jako środek do zapewnienia wysokiej niezawodności i jakości. Jak więc można to osiągnąć?

Są trzy podstawowe praktyki wdrażania infrastruktury jako kodu:

- Definiowanie wszystkiego jako kodu
- Stałe testowanie i dostarczanie wszystkiego na bieżąco
- Tworzenie małych, prostych elementów, które można zmieniać niezależnie

Podsumuję teraz każdą z nich, aby przygotować kontekst do dalszej dyskusji. Później poświęcę rozdział zasadom wdrażania każdej z tych praktyk.

Podstawowa praktyka: definiowanie wszystkiego jako kodu

Definiowanie wszystkiego „jako kodu” jest podstawową praktyką umożliwiającą szybkie i niezawodne wprowadzanie zmian. Jest kilka powodów, dla których to pomaga:

Możliwość wielokrotnego wykorzystania

Jeśli jakaś rzecz jest zdefiniowana jako kod, to można utworzyć wiele jej instancji. Można szybko naprawić i ponownie utworzyć taką rzecz, a inni mogą tworzyć jej identyczne instancje.

Spójność

Rzeczy zbudowane z kodu są za każdym razem budowane tak samo. To sprawia, że zachowanie systemu jest przewidywalne, testowanie bardziej niezawodne, a do tego możliwe jest ciągłe testowanie i dostarczanie.

Przejrzystość

Patrząc na kod każdy może przekonać się, jak jest zbudowana dana rzecz. Ludzie mogą przeglądać kod i sugerować ulepszenia. Mogą dowiedzieć się, jak wykorzystać tę rzecz w innym kodzie, jak jej użyć do rozwiązywania problemów, a ponadto wykonać inspekcję pod kątem zgodności.

O koncepcjach i zasadach implementacji w przypadku definiowania rzeczy jako kodu opowiem szerzej w rozdziale 4.

Podstawowa praktyka: stałe testowanie i dostarczanie wszystkiego na bieżąco

Efektywne zespoły infrastruktury podchodzą rygorystycznie do testowania. Używają automatyzacji do wdrażania i testowania każdego składnika swojego systemu oraz integrują efekty pracy, którą wszyscy wykonują. Testują na bieżąco, zamiast czekać na koniec.

Chodzi o to, aby budować jakość, a nie próbować *testować jakość*.

Jednym z elementów, o którym ludzie często zapominają, jest integracja i testowanie wszystkich prac na bieżąco. W wielu zespołach ludzie pracują nad kodem w oddzielnych działach i integrują go dopiero po zakończeniu. Z badań Accelerate wynika jednak, że zespoły osiągają lepsze wyniki, gdy każdy przynajmniej co dzień integruje swoją pracę. CI stosuje scalanie i testowanie kodu wszystkich osób w trakcie programowania. CD idzie dalej, utrzymując scalony kod zawsze gotowy do produkcji.

Ciągłe testowanie i dostarczanie kodu infrastruktury omówię bardziej szczegółowo w rozdziale 8.

Podstawowa praktyka: tworzenie małych, prostych elementów, które można zmieniać niezależnie

Zespoły borykają się z problemami, gdy ich systemy stają się duże i silnie sprzężone. Im większy system, tym trudniej go zmienić i tym łatwiej go popsuć.

Gdy patrzymy na bazę kodu wydajnego zespołu, widzimy różnicę. System składa się z małych, prostych elementów. Każdy element jest zrozumiały i ma wyraźnie określone interfejsy. Zespół może łatwo zmienić każdy składnik niezależnie od innych oraz wdrożyć go i oddzielnie przetestować.

Zasady implementacji tej podstawowej praktyki omówię bardziej szczegółowo w rozdziale 15.

Podsumowanie

Aby czerpać korzyści z chmury i automatyzacji infrastruktury, potrzebne jest podejście z epoki chmury. Oznacza to wykorzystywanie szybkości do poprawy jakości i budowanie jakości w celu zwiększenia szybkości. Automatyzacja infrastruktury wymaga pracy, zwłaszcza gdy dopiero uczymy się to robić. Ale takie działania pomagają wprowadzić zmiany, w tym przede wszystkim zbudować system.

Opisałem części typowego systemu infrastruktury, ponieważ stanowią one podstawę rozdziałów, w których wyjaśnię, jak wdrażać infrastrukturę jako kod.

Na koniec zdefiniowałem trzy podstawowe praktyki dotyczące infrastruktury jako kodu: definiowanie wszystkiego jako kodu, ciągłe testowanie i dostarczanie oraz tworzenie małych elementów.

Zasady infrastruktury w epoce chmury

Zasoby komputerowe w epoce żelaza branży IT były silnie sprzężone ze sprzętem fizycznym. Wkładaliśmy procesor, pamięć i dyski twarde do obudowy, umieszczaliśmy ją w szafie rackowej i podłączaliśmy do przełączników i routerów. Instalowaliśmy i konfigurowaliśmy system operacyjny oraz oprogramowanie użytkowe. Umieliśmy wskazać, w którym miejscu centrum danych znajduje się serwer aplikacji: na którym piętrze, w którym rzędzie, w której szafie, na jakiej wysokości.

Epoka chmury oddzieliła zasoby obliczeniowe od sprzętu fizycznego, na którym działają. Sprzęt oczywiście nadal istnieje, ale serwery, dyski twarde i routery są rozsiane po różnych miejscach. Nie ma już rzeczy fizycznych- zostały one przekształcone w wirtualne konstrukcje, które tworzymy, duplikujemy, zmieniamy lub usuwamy wedle uznania.

Ta transformacja zmusiła nas do zmiany sposobu myślenia, projektowania i wykorzystywania zasobów obliczeniowych. Nie możemy zakładać, że fizyczne atrybuty naszego serwera aplikacji będą niezmiennie. Musimy być w stanie szybko dodawać i usuwać instancje naszych systemów oraz łatwo utrzymywać ich spójność i jakość, nawet w przypadku ich gwałtownego rozrostu.

Istnieje kilka zasad projektowania i wdrażania infrastruktury w platformach chmurowych. Zasady te wyjaśniają powody stosowania trzech podstawowych praktyk (definiowanie wszystkiego jako kodu, ciągle testowanie i dostarczanie, tworzenie małych elementów). Wymieniam również kilka typowych pułapek, które czyhają na zespoły w przypadku dynamicznej infrastruktury.

Te zasady i pułapki leżą u podstaw bardziej szczegółowych porad dotyczących praktyk implementowania infrastruktury jako kodu w całej tej książce.

Zasada: zakładaj, że systemy są zawodne

W epoce żelaza zakładaliśmy, że nasze systemy działają na niezawodnym sprzęcie. W epoce chmury musimy zakładać, że nasz system działa na zawodnym sprzęcie¹.

1 Nauczyłem się tego z artykułu Sama Johnsona „Simplifying Cloud: Reliability” (<https://oreil.ly/S3VRT>).

Infrastruktura chmury obejmuje setki tysięcy urządzeń, jeśli nie więcej. W tej skali awarie zdarzają się nawet w przypadku używania niezawodnego sprzętu – a większość dostawców chmury wykorzystuje tani, mniej niezawodny sprzęt, który po wykryciu awarii jest wymieniany na nowy.

Trzeba przełączać fragmenty systemów w tryb offline nie tylko z powodu nieplanowanych awarii. Trzeba łączyć i aktualizować systemy. Trzeba zmieniać wielkość, rozkładać obciążenie i rozwiązywać problemy.

W przypadku statycznej infrastruktury robienie tych rzeczy oznacza przełączanie systemów w tryb offline. Ale dla wielu współczesnych organizacji przejście systemów w tryb offline oznacza przejście biznesu w tryb offline.

Nie można więc traktować infrastruktury, w której działa system, jako stabilnego fundamentu. Zamiast tego trzeba tak projektować, aby zapewnić nieprzerwane świadczenie usług w przypadku zmiany podstawowych zasobów².

Zasada: rób tak, aby wszystko było odtwarzalne

Jednym ze sposobów zapewnienia odtwarzalności systemu jest zadbanie, aby można było zawsze odbudować jego fragmenty, bez wysiłku i niezawodnie.

Bez wysiłku oznacza, że nie trzeba podejmować żadnych decyzji na temat tego, jak należy odbudowywać. Trzeba tylko zdefiniować ustawienia konfiguracji, wersje oprogramowania i zależności jako kod. Odbudowa jest wtedy prostą decyzją „tak/nie”.

Odtwarzalność nie tylko ułatwia odzyskanie uszkodzonego systemu, ale także pomaga:

- Zapewnić zgodność środowisk testowych z produkcyjnymi
- Replikować systemy w różnych regionach, aby zapewnić dostępność
- Dodawać instancje na żądanie, aby poradzić sobie z dużym obciążeniem
- Replikować systemy, aby zapewnić każdemu klientowi dedykowaną instancję

Oczywiście system generuje dane, zawartość i dzienniki, których nie można zdefiniować z wyprzedzeniem. Trzeba je zidentyfikować i znaleźć sposoby zachowywania w ramach swojej strategii replikacji. Może to być tak proste, jak kopiowanie lub strumieniowanie danych do kopii zapasowej, a następnie przywracanie ich podczas odbudowy. Opiszę opcje, jak to robić, w podrozdziale „Ciągłość danych w zmieniającym się systemie” na stronie 373.

Możliwość budowania i przebudowywania bez wysiłku dowolnej części infrastruktury ma bardzo duże znaczenie. Eliminuje ryzyko i lęk przed wprowadzaniem zmian i pozwala bez obaw radzić sobie z awariami. Umożliwia szybkie udostępnianie nowych środowisk i usług.

² Zasada zakładania, że systemy są zawodne, jest motorem inżynierii chaosu (<https://oreil.ly/7fvio>), w której w sposób kontrolowany wywołuje się awarie, aby przetestować i podnieść niezawodność usług. Mówię o tym więcej w punkcie „Inżynieria chaosu” na stronie 371.

Pułapka: systemy śnieżynki

Śnieżynka to instancja systemu lub części systemu, którą trudno odbudować. Może to być również środowisko, które powinno być podobne do innych środowisk, takie jak środowisko tymczasowe, ale różni się w sposób, którego zespół w pełni nie rozumie.

Ludzie nie chcą budować systemów śnieżynek. Są one zjawiskiem naturalnym. Tworząc coś po raz pierwszy za pomocą nowego narzędzia uczymy się przy okazji, z czym wiąże się popełnianie błędów. Ale jeśli ludzie zaczną już używać zbudowanej przez nas rzeczy, możemy nie mieć czasu, aby się cofnąć i przebudować ją lub ulepszyć na podstawie tego, czego się nauczyliśmy. Ulepszanie już zbudowanych rzeczy jest szczególnie trudne, gdy nie ma mechanizmów i praktyk sprawiających, że taka zmiana jest łatwa i bezpieczna.

Innym powodem występowania śnieżynek jest to, że ludzie wprowadzają zmiany w jednej instancji systemu, a nie robią tego w pozostałych. Mogą być pod presją, aby usunąć problem, który występuje tylko w jednym systemie, albo mogą rozpocząć dużą aktualizację w środowisku testowym, ale nie mieć czasu na wdrożenie jej w innych.

System jest śnieżynką, jeśli nie mamy pewności, że można go bezpiecznie zmienić lub ulepszyć. Co gorsza, jeśli system się zepsuje, trudno go naprawić. Dlatego ludzie unikają wprowadzania zmian w takim systemie, przez co pozostaje on nieaktualny, bez poprawek, a może nawet częściowo popsuty.

Systemy śnieżynki stwarzają ryzyko i marnują czas zarządzających nimi zespołów. Prawie zawsze warto spróbować zastąpić je systemami odtwarzalnymi. Jeśli system śnieżynka nie jest wart ulepszania, to może nie jest w ogóle wart zachowania.

Najlepszym sposobem na zastąpienie systemu śnieżynki jest napisanie kodu, który potrafi replikować system, uruchamiając równolegle nowy system i czekając, aż będzie gotowy. Należy skorzystać ze zautomatyzowanych testów i potoków, aby udowodnić, że nowy system jest poprawny i odtwarzalny oraz że można go łatwo zmienić.

Zasada: twórz rzeczy zastępowalne

Budowa systemu, który radzi sobie z dynamiczną infrastrukturą, to pierwszy poziom. Następny poziom to budowa systemu, który sam w sobie jest dynamiczny. Taki system powinien umożliwiać eleganckie dodawanie, usuwanie, uruchamianie, zatrzymywanie, zmienianie i przenoszenie jego części. W ten sposób zapewniamy mu elastyczność operacyjną, dostępność i skalowalność. Ponadto upraszczamy dokonywanie zmian i zmniejszamy towarzyszące im ryzyko.

Zapewnienie elastyczności elementów systemu to główna idea natywnego oprogramowania chmury. Chmura oddziela zasoby infrastruktury (obliczeniowe, sieciowe i pamięciowe) od sprzętu fizycznego. Natywne oprogramowanie chmury całkowicie rozdziela funkcjonalność aplikacji i infrastrukturę, na której działa³.

3 Zobacz „Infrastruktura natywna dla chmury i oparta na aplikacjach” na stronie 150



Bydło hodowlane, a nie domowi ulubieńcy

„Traktuj swoje serwery jak bydło hodowlane, a nie ulubieńców domowych” to popularne wyrażenie odnoszące się do zastępowalności⁴. Brakuje mi nadawania zabawnych imion wszystkim tworzonym przeze mnie serwerom. Ale nie marzę o konieczności dopieszczania każdego serwera w naszym środowisku.

Jeśli systemy są dynamiczne, to narzędzia używane do zarządzania nimi muszą sobie z tym radzić. Na przykład monitoring nie powinien podnosić alarmu za każdym razem, gdy przebudowujemy część naszego systemu. Natomiast powinien zgłosić ostrzeżenie, jeśli coś zacznie odbudowywać się w pętli.

Przypadek znikającego serwera plików

Ludzie potrzebują zwykle trochę czasu, żeby przyzwycząić się do ulotnej infrastruktury. Pracowałem z zespołem, który skonfigurował zautomatyzowaną infrastrukturę za pomocą narzędzi VMware i Chef. W razie potrzeby zespół usuwał i odbudowywał maszyny wirtualne.

Nowy programista w zespole potrzebował serwera do hostingu plików i udostępniania ich członkom zespołu, więc ręcznie zainstalował serwer HTTP na serwerze programistycznym i tam umieścił pliki. Kilka dni później odbudowałem maszynę wirtualną i jego serwer WWW zniknął.

Po chwili zdezorientowania programista zrozumiał, dlaczego tak się stało. Dodał swój serwer WWW do kodu Chefa i zamieścił swoje pliki w sieci SAN. Zespół miał teraz niezawodną usługę udostępniania plików.

Zasada: minimalizuj różnicowanie

Wraz z rozwojem systemu coraz trudniej jest go zrozumieć, zmieniać i naprawiać. Wymagana praca rośnie wraz z liczbą elementów, a także z liczbą ich rodzajów. Dlatego dobrym sposobem na utrzymanie możliwości zarządzania systemem jest dbanie o małą liczbę elementów – małą różnorodność. Łatwiej jest zarządzać stu identycznymi serwerami niż pięcioma zupełnie różnymi serwerami.

Zasada odtwarzalności (zobacz „Zasada: rób tak, aby wszystko było odtwarzalne” na stronie 14) uzupełnia tę ideę. Jeśli zdefiniujemy prosty komponent i utworzymy wiele identycznych jego instancji, będzie można go łatwo zrozumieć, zmieniać i naprawiać.

⁴ Po raz pierwszy usłyszałem to wyrażenie podczas prezentacji Gavina McCance’a „CERN Data Centre Evolution” (<https://oreil.ly/cDt47>). Randy Bias ceni prezentację Billa Bakera „Architectures for Open and Scalable Clouds” (https://oreil.ly/_SG96). Obie te prezentacje stanowią doskonałe wprowadzenie do tych zasad.

Aby to działało, należy stosować wszelkie zmiany do wszystkich instancji danego komponentu. W przeciwnym razie powstaje dryf konfiguracji.

Oto kilka rodzajów odmian, które występują w systemach:

- Wiele systemów operacyjnych, środowisk wykonawczych aplikacji, baz danych i innych technologii. Każdy z tych składników wymaga w zespole ludzi z odpowiednimi umiejętnościami i wiedzą.
- Wiele wersji oprogramowania, takiego jak system operacyjny czy baza danych. Nawet jeśli jest używany tylko jeden system operacyjny, to i tak każda wersja może wymagać innej konfiguracji i narzędzi.
- Różne wersje pakietów. Gdy niektóre serwery mają nowszą wersję pakietu, narzędzia lub biblioteki niż inne, pojawia się ryzyko. Polecenia mogą nie działać jednako we wszystkich wersjach, a starsze wersje mogą zawierać luki lub błędy.

Organizacje stoją przed dylematem, czy pozwalać każdemu zespołowi na wybór technologii i rozwiązań, które są odpowiednie do jego potrzeb, czy też utrzymywać stopień zróżnicowania w organizacji na możliwym do zarządzania poziomie.



Lekki nadzór

Nowoczesne, cyfrowe organizacje uczą się wartości *Lekkiego nadzorowania* IT, które równoważy autonomię i scentralizowane sterowanie. Jest to kluczowy element modelu EDGE dla zwinnych organizacji. Więcej na ten temat można znaleźć w książce *EDGE: Value-Driven Digital Transformation*⁵ Jima Highsmitha, Lindy Luu i Davida Robinsona (Addison-Wesley Professional) oraz w wystąpieniu Jonny'ego LeRoy'a „The Goldilocks Zone of Lightweight Architectural Governance”⁶.

Dryf konfiguracji

Dryf konfiguracji to różnice pojawiające się z czasem w systemach, które kiedyś były identyczne. Widać to na rysunku 2-1. Dryf konfiguracji może być efektem ręcznych zmian. Może również nastąpić, jeśli używamy narzędzi automatyzacji do wprowadzania zmian ad hoc tylko w niektórych instancjach. Dryf konfiguracji utrudnia zachowanie spójnej automatyzacji.

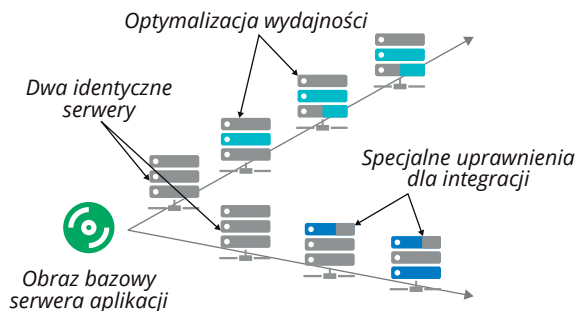
Jako przykład dywergencji konfiguracji w czasie rozważmy przypadek naszego przykładowego zespołu firmy ShopSpinner⁷.

ShopSpinner uruchamia osobną instancję swojej aplikacji dla każdego sklepu, przy czym każda instancja jest tak skonfigurowana, aby korzystała z niestandardowego brandingu i zawartości katalogu produktów. Na początku zespół ShopSpinner uruchamiał

⁵ <https://oreil.ly/Eg3Mu>

⁶ <https://oreil.ly/d29qg>

⁷ ShopSpinner to fikcyjna firma, która pozwala innym firmom zakładać i prowadzić sklepy internetowe. Będę używał jej w całej tej książce do ilustrowania koncepcji i praktyk.



Rysunek 2-1 Dryf konfiguracji ma miejsce w sytuacji, gdy wystąpienia tej samej rzeczy z upływem czasu zaczynają się różnić

skrypty, aby stworzyć nowy serwer aplikacji dla każdego nowego sklepu. Zespół zarządzał infrastrukturą ręcznie albo pisząc skrypty i dostrajając je za każdym razem, gdy była potrzebna zmiana.

Jeden z klientów, Water Works⁸, ma znacznie większy ruch w swojej aplikacji do zarządzania zamówieniami niż pozostali, więc zespół dostroił konfigurację serwera dla Water Works. Zmiany nie zostały wprowadzone u innych klientów, ponieważ zespół był zajęty i nie uznał tego za konieczne.

Później zespół ShopSpinner przystosował narzędzie Servermaker do automatyzacji konfiguracji swojego serwera aplikacji⁹. Najpierw przetestowano serwer dla Palace Pens¹⁰, mniejszego klienta, a następnie udostępniono pozostałym. Niestety, kod nie uwzględniał optymalizacji wydajności dla Water Works, więc ulepszenia przepadły. Serwer Water Works zwolnił znacząco, aż zespół zauważył to i naprawił błąd.

Sposobem na rozwiązanie problemu okazała się parametryzacja kodu Servermakera. Teraz można ustawiać różne poziomy zasobów dla poszczególnych klientów. W ten sposób zespół może nadal stosować jednaki kod dla wszystkich, optymalizując go jednocześnie dla każdego klienta. W rozdziale 7 są opisane niektóre wzorce i antywzorce parametryzacji kodu infrastruktury dla różnych instancji.

Zasada: pilnuj, abys mógł powtórzyć każdy proces

Jeśli stosujemy zasadę odtwarzalności, powinniśmy być w stanie powtórzyć wszystko, co robimy z naszą infrastrukturą. Łatwiej jest powtarzać czynności za pomocą skryptów i narzędzi do zarządzania konfiguracją, niż robić to ręcznie. Ale automatyzacja może wymagać dużo pracy, zwłaszcza jeśli nie jesteśmy do tego przyzwyczajeni.

⁸ Water Works wysyła co miesiąc butelki z wodą innego wytwórcy.

⁹ Servermaker to fikcyjne narzędzie do konfigurowania serwerów, podobne do Ansible, Chef i Puppet.

¹⁰ Palace Pens sprzedaje najlepsze na świecie luksusowe przybory do pisania.

Spirala strachu przed automatyzacją

Spirala strachu przed automatyzacją opisuje, jak wiele zespołów wpada w dryf konfiguracji i dług techniczny.

Na sesji poświęconej automatyzacji konfiguracji podczas konferencji DevOpsDays (<https://oreil.ly/x8G0C>) zapytałem grupę, ile osób korzysta z narzędzi automatyzacji, takich jak Ansible, Chef czy Puppet. Większość podniosła rękę w górę. Zapytałem, ile osób uruchamia te narzędzia bez nadzoru, używając automatycznego harmonogramu. Większość opuściła rękę.

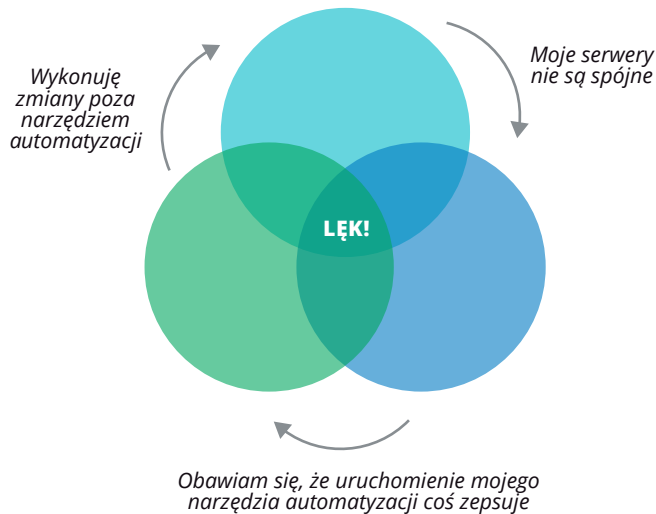
Wiele osób ma ten sam problem, który miałem na początku korzystania z narzędzi do automatyzacji. Używałem automatyzacji wybiórczo – na przykład, aby tworzyć nowe serwery lub dokonać określonej zmiany konfiguracji. Dostrajałem konfigurację za każdym razem, gdy ją uruchamiałem, aby odpowiadała konkretnemu zadaniu, które wykonywałem.

Bąłem się zostawiać moje narzędzia automatyzacji bez nadzoru, ponieważ nie miałem pewności, co zrobią.

Brakowało mi zaufania do mojej automatyzacji, ponieważ moje serwery nie były spójne.

Moje serwery nie były spójne, ponieważ nie uruchamiałem automatyzacji dość często ani konsekwentnie.

To właśnie jest spirala lęku przed automatyzacją, pokazana na rysunku 2-2. Zespoły ds. infrastruktury muszą przerwać tę spiralę, aby skutecznie używać automatyzacji.



Rysunek 2-2 Spirala lęku przed automatyzacją

Najskuteczniejszym sposobem przerywania spirali jest zmierzenie się z własnymi lękami. Trzeba zacząć od jednego zestawu serwerów. Upewnić się, że można zastosować, a następnie ponownie zastosować kod infrastruktury na tych serwerach. Następnie zaplanować godzinowy proces, który będzie w sposób ciągły stosować kod do tych serwerów. Następnie wybrać inny zestaw serwerów i powtórzyć proces. I kontynuować, aż każdy serwer będzie ciągle aktualizowany.

Dobry monitoring i zautomatyzowane testowanie budują pewność ciągłej synchronizacji kodu. A ta ujawnia dryf konfiguracji, gdy ma miejsce, więc można go szybko naprawić.

Przypuśćmy, na przykład, że muszę jednorazowo podzielić dysk na partycje. Napisanie i przetestowanie skryptu wymaga znacznie więcej pracy niż po prostu zalogowanie się i uruchomienie polecenia `fdisk`. Dlatego robię to ręcznie.

Problem pojawia się później, gdy ktoś z mojego zespołu, Priya, musi podzielić inny dysk. Dochodzi do tego samego wniosku co ja i robi to ręcznie, zamiast napisać skrypt. Podejmuje jednak nieco inną decyzję co do sposobu podziału dysku. Ja zrobiłem na moim serwerze partycję `/var` typu `ext3` 80 GB, a Priya tworzy u siebie partycję typu `xfs` 100 GB. Powodujemy dryf konfiguracyjny, który osłabia naszą zdolność do zaufanej automatyzacji.

Efektywne zespoły infrastruktury cechuje silna kultura skryptowa. Jeśli możesz napisać skrypt dla zadania, napisz go¹¹. Jeśli napisać skrypt jest trudno, zbadaj problem dokładnie. Może jest jakaś technika lub narzędzie, które mogą pomóc, a może da się uprościć zadanie albo podejść do niego inaczej. Rozbijanie pracy na zadania realizowane za pomocą skryptów sprawia, że całość staje się prostsza, bardziej przejrzysta i bardziej niezawodna.

Podsumowanie

Zasady infrastruktury w epoce chmury odzwierciedlają różnice między tradycyjną, statyczną infrastrukturą, a nowoczesną, dynamiczną infrastrukturą:

- zakładaj, że systemy są zawodne
- rób tak, aby wszystko było odtwarzalne
- twórz rzeczy zastępowalne
- minimalizuj zróżnicowanie
- pilnuj, abyś mógł powtórzyć każdy proces

Zasady te stanowią klucz do wykorzystania natury platform chmurowych. Zamiast wzbraniać się przed możliwością robienia zmian przy minimalnym wysiłku, wykorzystaj tę możliwość do zapewnienia jakości i niezawodności.

¹¹ Mój kolega Florian Sellmayr powiada „jeśli warto to dokumentować, to warto też automatyzować”.

Platformy infrastruktury

Istnieje szeroki wachlarz narzędzi związanych na różne sposoby z nowoczesną infrastrukturą chmury. Pytanie, które technologie wybrać i jak je połączyć, może być przytłaczające. W tym rozdziale przedstawiam model myślenia o problemach związanych z platformą na wyższym poziomie, jej możliwościach oraz zasobach infrastruktury, które można gromadzić, aby zapewnić te możliwości.

To nie jest autorytatywny model ani architektura. Czytelnik może mieć własny sposób na opisanie części swojego systemu. Ustalenie właściwego miejsca na diagramie dla każdego używanego narzędzia lub technologii jest mniej ważne niż rozmowa.

Celem tego modelu jest stworzenie kontekstu do dyskusji o pojęciach, praktykach i podejściach przedstawionych w tej książce. Szczególnie ważne jest upewnienie się, że te dyskusje są odpowiednie dla Czytelnika, niezależnie od stosu technologii, narzędzi lub platform, których używa. Dlatego model definiuje sposoby grupowania i terminologię, której w późniejszych rozdziałach używam do opisu, na przykład, udostępniania serwerów na platformie wirtualizacyjnej, takiej jak VMware lub w chmurze IaaS, takiej jak AWS.

Części systemu infrastruktury

Nowoczesna infrastruktura chmury składa się z wielu różnych części wielu różnych typów. Uważam, że wygodnie jest zgrupować te części w trzy warstwy platformy (rysunek 3-1):

Aplikacje

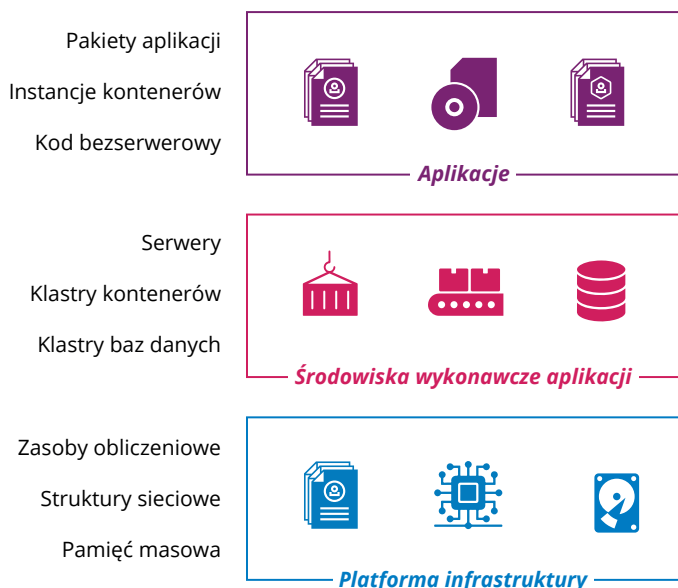
Aplikacje i usługi zapewniają organizacji i jej użytkownikom różnego rodzaju możliwości. Cała reszta tego modelu istnieje po to, żeby służyć tej warstwie.

Środowiska wykonawcze aplikacji

Środowiska wykonawcze aplikacji zapewniają usługi i różne możliwości warstwie aplikacji. Przykładami usług i konstrukcji na platformie wykonawczej aplikacji są klastry kontenerów, środowiska bezserwerowe, serwery aplikacji, systemy operacyjne i bazy danych. Warstwa ta nosi też nazwę platforma jako usługa (Platform as a Service – PaaS).

Platforma infrastruktury

Platforma infrastruktury jest zbiorem zasobów infrastruktury oraz narzędzi i usług, które nimi zarządzają. Chmury i platformy wirtualizacji dostarczają zasoby infrastruktury, w tym zasoby obliczeniowe, pamięciowe i podstawowe funkcje sieciowe. Jest to również znane jako Infrastruktura jako usługa (Infrastructure as a Service – IaaS). Zasoby zapewniane przez tę warstwę omówię szczegółowo w podrozdziale „Zasoby infrastruktury” na stronie 25.



Rysunek 3-1 Warstwy elementów systemu

Przesłaniem tej książki jest wykorzystywanie warstwy platformy infrastruktury do gromadzenia zasobów infrastruktury w celu utworzenia warstwy środowiska wykonawczego aplikacji.

W rozdziale 5 i pozostałych rozdziałach części II opisuję, jak używać kodu do budowy stosów infrastruktury. Stos infrastruktury jest kolekcją zasobów definiowaną i zarządzaną wspólnie za pomocą takich narzędzi, jak Ansible, CloudFormation, Pulumi i Terraform.

W rozdziale 10 i pozostałych rozdziałach części III opisuję, jak używać kodu do definiowania i zarządzania środowiskami wykonawczymi aplikacją. Należą do nich środowiska wykonawcze serwerowe, klastrowe i bezserwerowe.

Platformy infrastruktury

Infrastruktura jako kod wymaga dynamicznej platformy infrastruktury, czegoś, czego można używać do udostępniania i zmiany zasobów na żądanie za pomocą API. Na rysunku 3-2 widać model platformy z wyróżnioną warstwą platformy infrastruktury. Jest

to podstawowa definicja chmury¹. Gdy mówię w tej książce o „platformie infrastruktury”, można zakładać, że mam na myśli dynamiczną platformę typu IaaS².



Rysunek 3-2 Platforma infrastruktury jest podstawową warstwą modelu platformy

W dawnych czasach – epoce żelaza komputerów – infrastruktura była fizycznym sprzętem. Wirtualizacja oddzieliła systemy od sprzętu, na którym działały, a chmura dodała interfejsy API do zarządzania tymi zwirtualizowanymi zasobami³. Tak się zaczęła epoka chmury.

Istnieją różne rodzaje platform infrastruktury, od w pełni rozwiniętych chmur publicznych po chmury prywatne, od dostawców komercyjnych po platformy open source. W tym rozdziale przedstawię te rodzaje, a następnie opiszę różne typy udostępnianych przez nie zasobów infrastruktury. W tabeli 3-1 podane są przykłady dostawców, produktów i narzędzi dla każdego rodzaju platformy infrastruktury chmury.

- 1 Amerykański Narodowy Instytut Standardów i Technologii (NIST) ma doskonałą definicję przetwarzania w chmurze (<https://oreil.ly/U8qkE>).
- 2 Oto jak NIST definiuje IaaS: „Możliwości zapewniane konsumentowi to udostępnienie przetwarzania, przechowywania, sieci i innych podstawowych zasobów obliczeniowych, w ramach których konsument może wdrażać i uruchamiać dowolne oprogramowanie, w tym systemy operacyjne i aplikacje. Konsument nie zarządza infrastrukturą chmury ani jej nie kontroluje, natomiast sprawuje kontrolę nad systemami operacyjnymi, pamięcią i wdrożonymi aplikacjami; i prawdopodobnie ograniczoną kontrolę nad wybranymi składnikami sieci (np. zaporami hostów)”.
- 3 Technologia wirtualizacji istnieje już od lat 60. zeszłego wieku, ale w głównym nurcie serwerów opartych na architekturze x86 pojawiła się dopiero na początku tego wieku.

Tabela 3-1 Przykłady dynamicznych platform infrastruktury

Rodzaj platformy	Dostawcy lub produkty
Publiczne usługi chmurowe IaaS	AWS, Azure, Digital Ocean, GCE, Linode, Oracle Cloud, OVH, Scaleway i Vultr
Prywatne produkty chmurowe IaaS	CloudStack, OpenStack i VMware vCloud
Narzędzia chmurowe bare-metal	Cobbler, FAI i Foreman

Na poziomie podstawowym platformy infrastruktury zapewniają zasoby obliczeniowe, pamięciowe i sieciowe. Platformy udostępniają te zasoby w różnych formatach. Wykonywanie obliczeń może mieć postać na przykład serwera wirtualnego, środowiska wykonawczego kontenera czy kodu wykonywanego bez użycia serwera.



PaaS

Większość dostawców chmur publicznych udostępnia zasoby lub pakiety zasobów, które razem zapewniają usługi wyższego poziomu do wdrażania aplikacji i zarządzania nimi. Przykładami hostowanych usług PaaS są Heroku⁴, AWS Elastic BeanStalk⁵ i Azure DevOps⁶.

Różni dostawcy mogą pakować i oferować te same zasoby w różny sposób albo przynajmniej pod inną nazwą. Na przykład AWS object storage, Azure blob storage i GCP cloud storage to praktycznie jedno i to samo. W tej książce staram się używać nazw ogólnych, mających zastosowanie do różnych platform. Dlatego zamiast *VPC* i *Subnet* używam terminów *blok adresów sieci* i *VLAN*.

Wielochmurowość

Wiele organizacji dochodzi z czasem do hostingu na wielu platformach. Pojawiło się kilka terminów opisujących różne odmiany takiej sytuacji:

Chmura hybrydowa (ang. hybrid cloud)

Hosting aplikacji i usług dla systemu w ramach prywatnej infrastruktury i publicznej usługi chmurowej jednocześnie. Ludzie robią to często z powodu starych systemów, których nie mogą łatwo przenieść do publicznej usługi chmurowej (na przykład usług działających na komputerach typu mainframe). W innych przypadkach organizacje mają wymagania, których dostawcy chmur publicznych nie są aktualnie w stanie spełnić. Chodzi na przykład o wymóg prawny hostingu danych w kraju, w którym dostawca nie jest obecny.

⁴ <https://www.heroku.com>

⁵ <https://oreil.ly/rbFTp>

⁶ <https://oreil.ly/r0mPZ>

Agnostyk chmurowy (ang. cloud agnostic)

Budowanie systemów w taki sposób, aby mogły działać na wielu publicznych platformach chmurowych. Ludzie często tak robią mając nadzieję, że nie uzależnią się od jednego dostawcy. W praktyce kończy się to uzależnieniem od oprogramowania, które stara się ukryć różnice między chmurami, albo prowadzi do budowy i utrzymywania dużych ilości niestandardowego kodu, albo jednym i drugim.

Polichmura (ang. polycloud)

Uruchamianie różnych aplikacji, usług i systemów na więcej niż jednej publicznej platformie chmurowej. Zwykle ma to na celu wykorzystanie różnych mocnych stron poszczególnych platform.

Zasoby infrastruktury

Są trzy podstawowe zasoby, które udostępnia platforma infrastruktury: obliczenia, pamięć masowa i sieć. Różne platformy łączą ze sobą i pakują te zasoby na różne sposoby. Na przykład można mieć możliwość udostępniania maszyn wirtualnych i instancji kontenerów na swojej platformie. Można też mieć możliwość udostępnienia instancji baz danych, co obejmuje obliczenia, pamięć masową i sieć.

Bardziej elementarne zasoby infrastruktury nazywam prymitywami. Każdy z zasobów – obliczenia, pamięć masowa i sieć – opisanych w dalszej części tego rozdziału jest prymitywem. Platformy chmurowe łączą prymitywy infrastruktury w zasoby złożone, takie jak:

- Baza danych jako usługa (DBaaS)
- Równoważenie obciążeń
- DNS
- Zarządzanie tożsamościami
- Zarządzanie wpisami tajnymi (sekretami)

Linia oddzielająca zasób pierwotny od złożonego jest arbitralna, podobnie jak linia oddzielająca zasób infrastruktury złożonej od usługi środowiska wykonawczego aplikacji. Nawet podstawowa usługa pamięci masowej, taka jak magazyn obiektów (zasobniki AWS S3), obejmuje zasoby obliczeniowe i sieciowe do odczytu i zapisu danych. Ale jest to przydatne rozróżnienie, którego użyję teraz, aby wymienić popularne formy prymitywów infrastruktury. Są to trzy podstawowe typy zasobów: zasoby obliczeniowe, zasoby pamięci masowej i zasoby sieciowe.

Zasoby obliczeniowe

Zasoby obliczeniowe wykonują kod. W sensie najbardziej podstawowym jest to czas wykonywania przez rdzeń procesora fizycznego serwera. Ale platformy zapewniają obliczenia w bardziej przydatnych formach. Typowe rodzaje zasobów obliczeniowych obejmują:

Maszyny wirtualne (VM)

Platforma infrastruktury zarządza pulą fizycznych serwerów hostów i uruchamia instancje maszyn wirtualnych pod kontrolą hiperwizorów na tych hostach. Rozdział 11 zawiera bardziej szczegółowe informacje o udostępnianiu serwerów, konfigurowaniu i zarządzaniu nimi.

Serwery fizyczne

Zwane również *bare metal* – platforma udostępnia serwery fizyczne dynamicznie na żądanie. Podrozdział „Tworzenie serwera przy użyciu sieciowego narzędzia wyposażania” na stronie 177 zawiera opis mechanizmu automatycznego udostępniania serwerów fizycznych jako chmury bare metal.

Klastry serwerów

Klaster serwerów to pula instancji serwerów – maszyn wirtualnych lub serwerów fizycznych – udostępnianych i zarządzanych przez platformę infrastruktury w formie grupy. Przykłady to AWS Auto Scaling Group (ASG), Azure Virtual Machine Scale Set oraz Google Managed Instance Groups (MIGs).

Kontenery

Większość platform chmurowych oferuje CaaS (Containers as a Service) do wdrażania i uruchamiania instancji kontenerów. Często budujemy obraz kontenera w standardowym formacie (np. Docker), a platforma używa go do uruchamiania instancji. Wiele platform środowiska wykonawczego aplikacji opiera się na konteneryzacji. Bardziej szczegółowo jest to omówione w rozdziale 10. Kontenery wymagają do działania serwerów hostów. Niektóre platformy udostępniają te hosty w sposób przezroczysty, ale wiele z nich wymaga samodzielnego zdefiniowania klastra i jego hostów.

Klastry hostingu aplikacji

Klaster hostingu aplikacji to pula serwerów, na których platforma wdraża wiele aplikacji i zarządza nimi⁷. Przykłady to Amazon Elastic Container Services (ECS), Amazon Elastic Container Service for Kubernetes (EKS), Azure Kubernetes Service (AKS) i Google Kubernetes Engine (GKE). Można również wdrożyć pakiet hostingu aplikacji i zarządzać nim na platformie infrastruktury. Więcej szczegółów w podrozdziale „Rozwiązania dla klastrów aplikacji” na stronie 224.

7 Nie należy mylić klastra hostingu aplikacji z PaaS. Taki klaster zarządza udostępnianiem zasobów obliczeniowych aplikacjom, co jest jedną z podstawowych funkcji PaaS. Ale pełny PaaS zapewnia wiele innych usług poza obliczeniami.

Bezserwerowe środowiska wykonawcze kodu typu FaaS

Bezserwerowa platforma infrastruktury typu FaaS (Function as a Service) wykonuje kod na żądanie w odpowiedzi na zdarzenie lub harmonogram, a następnie kończy go po wykonaniu działania. Więcej informacji zawartych jest w podrozdziale „Infrastruktura dla bezserwerowej usługi FaaS” na stronie 240. Oferta bezserwerowych platform infrastruktury obejmuje AWS Lambda, Azure Functions i Google Cloud Functions.



Nie tylko kod bezserwerowy

Bezserwerowa koncepcja wykracza poza uruchamianie kodu. Przykładami bezserwerowych baz danych są Amazon DynamoDB i Azure Cosmos DB – serwery zaangażowane w hosting bazy danych są przezroczyste, w przeciwieństwie do bardziej tradycyjnych ofert DBaaS, takich jak AWS RDS.

Niektóre platformy chmurowe oferują wyspecjalizowane środowiska wykonywania aplikacji na żądanie. Na przykład AWS SageMaker, Azure ML Services i Google ML Engine pozwalają wdrażać i uruchamiać modele uczenia maszynowego.

Zasoby pamięci masowej

Wiele systemów dynamicznych, takich jak woluminy dyskowe, bazy danych czy centralne repozytoria plików, potrzebuje pamięci masowej. Nawet jeśli nasza aplikacja nie korzysta bezpośrednio z pamięci masowej, wiele usług to robi i będzie jej potrzebować, choćby do przechowywania obrazów obliczeń (np. migawek maszyn wirtualnych i obrazów kontenerów).

Prawdziwie dynamiczna platforma zarządza pamięcią masową i udostępnia ją aplikacjom w sposób przezroczysty. Ta funkcja różni się od klasycznych systemów wirtualizacji, w których trzeba jawnie określić pamięć fizyczną do przydzielenia i dołączyć ją do każdej instancji obliczeń.

Typowe zasoby pamięci masowej spotykane na platformach infrastruktury to:

Magazyn bloków (*wirtualny wolumin dysku*)

Wolumin magazynu bloków można dołączyć do pojedynczego serwera lub instancji kontenera, tak jakby to był dysk lokalny. Przykładami usług magazynu bloków oferowanymi przez platformy chmurowe są AWS EBS, Azure Page Blobs, OpenStack Cinder i GCE Persistent Disk.

Magazyn obiektów

Magazyn obiektów można wykorzystywać do zapisywania i dostępu do plików z wielu lokalizacji. Przykładami są Amazon S3, Azure Block Blobs, Google Cloud Storage i OpenStack Swift. Magazyn obiektów jest zwykle tańszy i bardziej niezawodny od magazynu bloków, ale ma większe opóźnienie.

Sieciowe systemy plików (udostępnione woluminy sieciowe)

Wiele platform chmurowych udostępnia dzielone woluminy magazynów, które można montować do wielu instancji obliczeń za pomocą standardowych protokołów, takich jak NFS, AFS czy SMB/CIFS⁸.

Magazyn danych strukturalnych

Większość platform infrastruktury udostępnia usługę DBaaS (Database as a Service), którą można zdefiniować jako kod i tak nią zarządzać. Może to być standardowa aplikacja bazy danych, komercyjna albo typu open source, taka jak MySQL, Postgres czy SQL Server. Może to też być niestandardowa usługa magazynu danych strukturalnych, na przykład magazyn danych klucz-wartość albo magazyn sformatowanych dokumentów o zawartości JSON lub XML. Główni dostawcy chmur oferują ponadto magazyny danych i usługi przetwarzania do zarządzania, przekształcania i analizowania dużych ilości danych. Należą do nich usługi wsadowego przetwarzania danych, map-reduce, streamingu, indeksowania i wyszukiwania oraz wiele innych.

Zarządzanie wpisami tajnymi

Każdy zasób pamięci masowej można zaszyfrować, aby móc przechowywać w nim hasła, klucze i inne informacje, które w wyniku ataku mogłyby zostać wykorzystane do uzyskania uprzywilejowanego dostępu do systemów i zasobów. Usługa zarządzania wpisami tajnymi dodaje funkcjonalność zaprojektowaną specjalnie w celu ułatwienia zarządzania tego typu zasobami. Techniki zarządzania wpisami tajnymi i kodem infrastruktury są opisane w podrozdziale „Obsługa wpisów tajnych jako parametrów” na stronie 96.

Zasoby sieciowe

Podobnie jak w przypadku innego typu zasobów infrastruktury, zdolność platform dynamicznych do udostępniania i zmiany sieci na żądanie z poziomu kodu stwarza ogromne możliwości. Te możliwości wykraczają poza szybszą zmianę sieci; obejmują również znacznie bezpieczniejsze korzystanie z sieci.

Część bezpieczeństwa wynika z możliwości szybkiego i dokładnego przetestowania zmiany konfiguracji sieci przed jej zastosowaniem w krytycznym środowisku. Ponadto, SDN (Software Defined Networking) umożliwia tworzenie bardziej szczegółowych konstrukcji zabezpieczeń sieciowych, niż można to zrobić ręcznie. Jest to prawdą zwłaszcza w przypadku systemów, w których elementy są tworzone i niszczone dynamicznie.

Typowe konstrukcje i usługi sieciowe oferowane przez platformę infrastruktury obejmują:

Bloki adresów sieciowych

Bloki adresów sieci są podstawową strukturą grupowania zasobów w celu sterowania trasowaniem ruchu między nimi. Blokiem najwyższego poziomu w AWS jest VPC

8 Odpowiednio Network File System (<https://oreil.ly/OLv7D>), Andrew File System (<https://oreil.ly/c8nh3>) oraz Server Message Block (<https://oreil.ly/DEXyZ>).

(Virtual Private Cloud). W Azure, GCP i innych jest to sieć wirtualna. Blok najwyższego poziomu jest często podzielony na mniejsze bloki, takie jak podsieci lub sieci VLAN. Określona struktura sieciowa, taka jak podsieci AWS, może być skojarzona z fizycznymi lokalizacjami, takimi jak centrum danych, których można używać do zarządzania dostępnością.

Nazwy, takie jak wpisy DNS

Nazwy są mapowane na adresy sieciowe niższego poziomu, zazwyczaj adresy IP.

Trasy

Konfigurują ruch, który jest dozwolony między blokami adresów i w ich obrębie.

Bramy

Mogą być potrzebne do kierowania ruchem do i z bloków.

Reguły równoważenia obciążenia

Przekierowują połączenia przychodzące na pojedynczy adres do puli zasobów.

Serwery proxy

Akceptują połączenia i za pomocą reguł przekształcają je lub wybierają dla nich trasy.

Bramy API

Serwery proxy, zwykle dla połączeń HTTP/S, które mogą wykonywać czynności w celu obsługi innych niż podstawowe aspektów API, takich jak uwierzytelnianie i ograniczanie szybkości. Zobacz także „Siatka usług” na stronie 238.

Sieci VPN (wirtualne sieci prywatne)

Łączą różne bloki adresów w różnych lokalizacjach, tak aby wydawały się częścią jednej sieci.

Połączenie bezpośrednie

Dedykowane połączenie między siecią chmury a inną lokalizacją, zwykle centrum danych lub siecią biurową.

Reguły dostępu do sieci (reguły zapory)

Reguły, które ograniczają lub dopuszczają ruch między lokalizacjami sieciowymi.

Asynchroniczne przesyłanie wiadomości

Kolejki dla procesów do wysyłania i odbierania wiadomości.

Pamięć podręczna

Pamięć podręczna dystrybuje dane w ramach lokalizacji sieciowych, aby zmniejszyć opóźnienia. CDN (Content Distribute Network) to usługa, która potrafi dystrybuować zawartość statyczną (a w niektórych przypadkach kod wykonywalny) do wielu lokalizacji geograficznych, zwykle w odniesieniu do zawartości dostarczanej przy użyciu HTTP/S.

Siatka usług

Zdecentralizowana sieć usług, która dynamicznie zarządza łącznością między częściami systemu rozproszonego. Przenosi możliwości sieciowe z warstwy infrastruktury do warstwy środowiska wykonawczego aplikacji w modelu opisanym w podrozdziale „Części systemu infrastruktury” na stronie 21.

Szczegóły sieci wykraczają poza zakres tej książki, dlatego odsyłam czytelnika do dokumentacji dostawcy jego platformy, a także książki Craiga Hunta *TCP/IP Network Administration*⁹ (O'Reilly).

Model bezpieczeństwa zerowego zaufania z SDN

W modelu bezpieczeństwa zerowego zaufania wszystkie usługi, aplikacje i inne zasoby są zabezpieczone na najniższym poziomie¹⁰. Różni się to od tradycyjnego modelu bezpieczeństwa opartego na koncepcji brzegu (perymetru), który zakłada, że każdemu urządzeniu wewnątrz bezpiecznej sieci można ufać.

Implementacja modelu zerowego zaufania dla nietrywialnego systemu jest możliwa tylko poprzez zdefiniowanie tego systemu jako kodu. Ręczna praca związana z zarządzaniem kontrolą każdego procesu w takim systemie byłaby przytłaczająca.

W przypadku systemu z zerowym zaufaniem każda nowa aplikacja jest opatrzona adnotacją wskazującą, do których aplikacji i usług musi mieć dostęp. Platforma wykorzystuje to do automatycznego umożliwiania potrzebnego dostępu i zapewnienia, że wszystko inne jest zablokowane. Oto zalety tego typu podejścia:

- Każda aplikacja i usługa mają tylko takie uprawnienia i dostęp, których jawnie wymagają, co jest zgodne z zasadą najmniejszych uprawnień.
- Zero zaufania, czyli bezpieczeństwo *bez perymetru*, oznacza nakładanie bardziej zawężonych barier i regulacji na zasoby, które wymagają ochrony. Takie podejście pozwala uniknąć konieczności zezwalania na strefy zaufania o szerokim zasięgu, na przykład przyznawania statusu zaufania każdemu urządzeniu podłączonemu do sieci fizycznej.
- Relacje bezpieczeństwa między elementami systemu są widoczne, co umożliwia walidację, inspekcję i raportowanie.

Podsumowanie

Różne rodzaje zasobów i usług infrastruktury omówione w tym rozdziale są elementami, z których można składać użyteczne systemy – część „infrastrukturuw” infrastruktury jako kodu.

⁹ <https://oreil.ly/2jIVeYe>

¹⁰ Google dokumentuje swoje podejście do zerowego zaufania (znanego również jako bezbrzegowy – *perimeterless*) za pomocą modelu bezpieczeństwa BeyondCorp (<https://oreil.ly/0Zonv>).

Podstawowa praktyka: definiuj wszystko jako kod

W rozdziale 1 wymieniłem trzy podstawowe praktyki, które pomagają szybko i niezawodnie zmieniać infrastrukturę: definiuj wszystko jako kod, ciągle testuj i dostarczaj wszystko na bieżąco oraz twórz małe, proste elementy.

Ten rozdział zagłębia się w pierwszą z tych podstawowych praktyk, zaczynając od banalnych pytań. Dlaczego mielibyśmy definiować infrastrukturę jako kod? Jakie rodzaje rzeczy można i należy definiować jako kod?

Na pierwszy rzut oka „definiuj wszystko jako kod” może wydawać się oczywiste w kontekście tej książki. Ale cechy różnego typu języków sprawiają, że są one odpowiednie do różnych rzeczy, co pokażę w następnych rozdziałach. I tak w rozdziale 5 opiszę używanie języków deklaracyjnych do definiowania stosów niskiego poziomu oraz wysokiego poziomu, a w rozdziale 16 wyjaśnię, kiedy kod deklaracyjny lub programistyczny jest najbardziej odpowiedni do tworzenia modułów i bibliotek kodu wielokrotnego użytku.

Dlaczego należy definiować infrastrukturę jako kod

Istnieją prostsze sposoby udostępnienia infrastruktury od pisania porcji kodu, a następnie wprowadzenia go do narzędzia. Wystarczy przejść do internetowego interfejsu użytkownika platformy, kliknięciem powołać do istnienia klaster serwerów aplikacji, a następnie wyświetlić wiersz polecenia i wykorzystując swój kunszt władania odpowiednim narzędziem CLI (command-line interface) wykuć niezniszczalną barierę sieci.

Ale na poważnie, w poprzednich rozdziałach wyjaśniłem, dlaczego lepiej jest używać kodu do budowania systemów, zapewniających możliwości ponownego wykorzystania, spójność i przejrzystość (patrz „Podstawowa praktyka: definiowanie wszystkiego jako kodu” na stronie 11).

Implementowanie systemów i zarządzanie nimi jako kodem umożliwia wykorzystanie szybkości do poprawy jakości. To sekret, który napędza wysoką wydajność mierzoną czterema kluczowymi wskaźnikami (patrz „Cztery kluczowe wskaźniki” na stronie 10).

Co można zdefiniować jako kod

Każde narzędzie infrastruktury ma inną nazwę dla swojego kodu źródłowego – na przykład podręcznik, książka kucharska, manifest czy szablon. Mówię o nich w ogólnym sensie jako *kodzie infrastruktury*, a czasami jako *definicji infrastruktury*.

Kod infrastruktury określa zarówno elementy infrastruktury, których potrzebujemy, jak i sposób ich konfigurowania. Narzędzie infrastruktury jest uruchamiane w celu zastosowania kodu do instancji infrastruktury. W efekcie narzędzie albo tworzy nową infrastrukturę, albo modyfikuje istniejącą, aby była zgodna z definicją zawartą w kodzie. Oto niektóre rzeczy, które należy definiować jako kod:

- Stos infrastruktury, który jest zbiorem elementów udostępnionych z platformy chmurowej infrastruktury. Więcej informacji o platformach infrastruktury można znaleźć w rozdziale 3, a wprowadzenie do koncepcji stosu infrastruktury znajduje się w rozdziale 5.
- Elementy konfiguracji serwera, takie jak pakiety, pliki, konta użytkowników i usługi (rozdział 11).
- Rola serwera to kolekcja elementów serwera, które są stosowane razem do pojedynczej instancji serwera („Role serwerów” na stronie 169).
- Definicja obrazu serwera generuje obraz do budowy wielu instancji serwera („Narzędzia do budowania obrazów serwerów” na stronie 203).
- Pakiet aplikacji definiuje sposób budowy możliwych do wdrożenia artefaktów aplikacji, w tym kontenerów (rozdział 10).
- Konfiguracja i skrypty usług dostarczania, które obejmują potoki i wdrażanie („Narzędzia do budowania obrazów serwerów” na stronie 203).
- Konfiguracja usług operacyjnych, takich jak testy monitorowania.
- Reguły walidacji, które obejmują zarówno testy zautomatyzowane, jak i reguły zgodności (rozdział 8).

Wybieraj narzędzia z eksternalizacją konfiguracji

Infrastruktura jako kod z definicji oznacza specyfikowanie infrastruktury w plikach tekstowych. Zarządzanie tymi plikami odbywa się niezależnie od narzędzi używanych do stosowania ich w systemie. Można je czytać, edytować, analizować i manipulować nimi za pomocą dowolnych narzędzi.

Bezkodowe narzędzia do automatyzacji infrastruktury przechowują definicje infrastruktury jako dane, do których nie można uzyskać bezpośredniego dostępu. Zamiast tego można wykorzystywać i edytować specyfikacje tylko za pomocą samego narzędzia. Narzędzie może oferować pewną kombinację interfejsów GUI, API i wiersza poleceń.

Problem z narzędziami typu „zamknięta skrzynka” polega na tym, że ograniczają one praktyki i przepływy pracy, które można stosować:

- Można stosować wersje specyfikacji infrastruktury tylko wtedy, gdy narzędzie ma wbudowane wersjonowanie.
- Można korzystać z podejścia CI tylko wtedy, gdy narzędzie umożliwia automatyczne wyzwalanie zadania po wprowadzeniu zmiany.
- Można tworzyć potoki dostarczania tylko wtedy, gdy narzędzie ułatwia wersjonowanie i promowanie specyfikacji infrastruktury.



Wnioski z kodu źródłowego oprogramowania

Schemat eksternalizacji konfiguracji odzwierciedla sposób działania większości kodu źródłowego oprogramowania. Niektóre środowiska programistyczne ukrywają kod źródłowy, na przykład Visual Basic for Applications. Ale w przypadku nietrywialnych systemów programiści uważają, że przechowywanie kodu źródłowego w plikach zewnętrznych jest bardziej wydajne.

Trudno jest stosować praktyki inżynierskie Agile, takie jak TDD, CI i CD, z narzędziami do zarządzania infrastrukturą typu „zamknięta skrzynka”.

Narzędzie, które wykorzystuje kod zewnętrzny do swoich specyfikacji, nie narzuca stosowania określonego przepływu pracy. Można użyć standardowego w branży systemu nadzorowania kodu źródłowego, edytora tekstu, serwera CI i zautomatyzowanego środowiska testowania. Można budować potoki dostarczania za pomocą narzędzia, które jest najbardziej odpowiednie.

Zarządzaj kodem w systemie kontroli wersji

Jeśli rzeczy są definiowane jako kod, to umieszczenie go w systemie kontroli wersji (VCS) jest proste i wydajne. W ten sposób zyskuje się:

Identyfikowalność

VCS zapewnia historię zmian, plus kto je zrobił i kontekst „dlaczego”¹. Taka historia jest bezcenna w przypadku debugowania problemów.

Wycofywanie

Gdy zmiana coś psuje – a zwłaszcza gdy wiele zmian coś psuje – wygodnie jest móc przywrócić wszystko do stanu dokładnie takiego, w jakim było wcześniej.

Skorelowanie

Trzymanie skryptów, specyfikacji i konfiguracji w systemie kontroli wersji pomaga śledzić i rozwiązywać poważne problemy. Można korelować elementy za pomocą tagów i numerów wersji.

Widoczność

Każdy może zobaczyć dowolną zmianę wykonaną w systemie kontroli wersji, przez co zespół ma pełny obraz sytuacji. Ktoś może zauważyć, że jakaś zmiana pominęła coś

1 Kontekst „dlaczego” zależy od tego, czy ludzie piszą przydatne komunikaty o zmianach.

ważnego. Jeśli zdarzy się wpadka, ludzie znają ostatnie wprowadzone zmiany, które mogły to spowodować.

Gotowość do działania

VCS może automatycznie wyzwać działanie na skutek każdej dokonanej zmiany. Wyzwalacze umożliwiają stosowanie zadań CI i potoków CD.

Jedynym, czego nie należy wprowadzać do systemu kontroli wersji, to niezaszyfrowane wpisy tajne, takie jak hasła i klucze. Nawet jeśli repozytorium kodu źródłowego jest prywatne, historia i wersje kodu zbyt łatwo wyciekają. Wpisy tajne, które wyciekły z kodu źródłowego, są jedną z najczęstszych przyczyn naruszeń bezpieczeństwa. Zobacz „Obsługa wpisów tajnych jako parametrów” na stronie 96, aby poznać lepsze sposoby zarządzania tymi danymi.

Języki kodowania infrastruktury

Administratorzy systemów od dziesięcioleci używają skryptów do automatyzacji zadań związanych z zarządzaniem infrastrukturą. Języki skryptowe ogólnego przeznaczenia, takie jak Bash, Perl, PowerShell, Ruby i Python, są nadal istotną częścią zestawu narzędzi zespołu infrastruktury.

CFEngine² jest pionierem wykorzystania deklaratywnych języków specyficznych dla domeny (DSL; patrz „Języki dziedzinowe infrastruktury” na stronie 40) do zarządzania infrastrukturą. Puppet³, a następnie Chef⁴ pojawiły się wraz z popularną wirtualizacją serwerów i chmurą IaaS. A po nich przyszło Ansible⁵, Saltstack⁶ i inne.

Narzędzia zorientowane na stos, takie jak Terraform⁷ i CloudFormation⁸ powstały kilka lat później, wykorzystując ten sam deklaratywny model DSL. Języki deklaratywne uprościły kod infrastruktury, oddzielając definicję tego, jaka ma być infrastruktura od sposobu jej implementacji.

Ostatnio występuje nowy trend w narzędziach infrastruktury, polegający na wykorzystywaniu języków programowania ogólnego przeznaczenia do definiowania infrastruktury⁹. Pulumi¹⁰ i AWS CDK¹¹ (Cloud Development Kit) obsługują języki, takie jak Typescript, Python i Java. Narzędzia te powstały w celu rozwiązania niektórych ograniczeń języków deklaratywnych.

2 <https://cfengine.com>

3 <https://puppet.com>

4 <https://www.chef.io>

5 <https://www.ansible.com>

6 <https://www.saltstack.com>

7 <https://www.terraform.io>

8 <https://oreil.ly/wt4pC>

9 „Ostatnio”, gdy piszę to w połowie roku 2020.

10 <https://www.pulumi.com>

11 <https://aws.amazon.com/cdk>

Mieszanie kodu deklaratywnego i imperatywnego

Kod imperatywny to zbiór instrukcji, które określają, jak sprawić, aby coś nastąpiło. Kod deklaratywny określa, czego chcemy, bez określania, jak do tego doprowadzić¹².

Zbyt wielu twórców kodu infrastruktury produkowanego obecnie cierpi z powodu mieszania kodu deklaratywnego i imperatywnego. Uważam, że naleganie, aby jeden z tych dwóch paradygmatów językowych był używany dla całego kodu infrastruktury, jest błędem.

Baza kodu infrastruktury obejmuje wiele różnych kwestii, od definiowania zasobów infrastruktury, przez konfigurowanie różnych instancji podobnych do siebie zasobów, po orkiestrację udostępniania wielu współzależnych elementów systemu. Niektóre z tych rzeczy można wyrazić najprościej za pomocą języka deklaratywnego. Inne są bardziej złożone i lepiej radzić sobie z nimi za pomocą języka imperatywnego.

Jako praktycy wciąż młodej dziedziny kodu infrastruktury nadal badamy, gdzie wyznaczyć granice między tymi zagadnieniami. Mieszanie ich może prowadzić do kodu, który łączy paradygmaty językowe. Jedną z przyczyn błędów bywa rozszerzanie deklaratywnej składni, takiej jak YAML, w celu dodania warunków i pętli. Drugą przyczyną błędów jest osadzanie prostych danych konfiguracyjnych („2GB RAM”) w kodzie proceduralnym, czyli mieszanie tego, co chcemy, ze sposobem implementacji.

W odpowiednich częściach tej książki wskazuję, gdzie, moim zdaniem, mogą być różne podejścia i gdzie jeden lub drugi paradygmat językowy może być najbardziej odpowiedni. Ale nasza dziedzina wciąż się rozwija. Wiele moich rad może okazać się błędnych lub niepełnych. Dlatego moją intencją jest zachęcenie czytelnika do przemyślenia tych pytań i pomożenia nam wszystkim w ustaleniu, co działa najlepiej.

Skrypty infrastruktury

Zanim pojawiły się standardowe narzędzia do deklaratywnego udostępniania infrastruktury chmury, pisaliśmy skrypty w językach proceduralnych ogólnego przeznaczenia. Skrypty te używały zwykle SDK (software development kit) do interakcji z API dostawcy usług w chmurze.

W przykładzie 4-1 występuje pseudokod, podobny do skryptów, które pisałem w Ruby używając AWS SDK. Tworzy on serwer o nazwie `my_application_server`, a następnie uruchamia narzędzie (fikcyjne) `Servermaker` do jego skonfigurowania.

¹² Czasami piszę o kodzie lub języku imperatywnym jako programowalnym, mimo że nie jest to najtrafniejsze określenie, ponieważ jest bardziej intuicyjne niż „imperatywny”.

Przykład 4-1 Przykład kodu proceduralnego, tworzącego serwer

```
import 'cloud-api-library'

network_segment = CloudApi.find_network_segment('private')

app_server = CloudApi.find_server('my_application_server')
if(app_server == null) {
    app_server = CloudApi.create_server(
        name: 'my_application_server',
        image: 'base_linux',
        cpu: 2,
        ram: '2GB',
        network: network_segment
    )
    while(app_server.ready == false) {
        wait 5
    }
    if(app_server.ok != true) {
        throw ServerFailedError
    }
    app_server.provision(
        provisioner: servermaker,
        role: tomcat_server
    )
}
```

Ten skrypt łączy to, co ma stworzyć, z tym, *jak* ma to stworzyć. Określa atrybuty serwera, łącznie z zasobami pamięci i procesora, które ma otrzymać, obraz systemu operacyjnego, od którego należy zacząć, oraz rolę, którą należy zastosować do serwera. Implementuje również logikę: sprawdza, czy serwer o nazwie `my_application_server` już istnieje, aby uniknąć zdublowania, a następnie czeka z zastosowaniem konfiguracji, aż serwer będzie gotowy.

Ten przykładowy kod nie obsługuje zmian atrybutów serwera. A co, jeśli zajdzie potrzeba zwiększenia RAM-u? Można zmienić skrypt tak, aby w sytuacji istnienia serwera sprawdzał każdy atrybut i zmieniał go w razie potrzeby. Można też napisać nowy skrypt do wykrywania i modyfikowania istniejących serwerów.

W bardziej realistycznych scenariuszach występuje wiele serwerów różnych typów. Oprócz naszego serwera aplikacji mój zespół miał serwery WWW i serwery baz danych. Mieliśmy także wiele środowisk, co oznaczało wiele instancji każdego serwera.

Zespoły, z którymi pracowałem, często zmieniały proste skrypty, jak ten w przykładzie, w skrypt wielofunkcyjne. Tego typu skrypt pobierał argumenty określające typ serwera i środowisko, i na ich podstawie tworzył odpowiednią instancję serwera. Ostatecznie powstawał skrypt, który wczytywał pliki konfiguracyjne, określające różne atrybuty serwera.

Podczas pracy nad takim skrypcem zastanawiałem się, czy nie byłoby warto opublikować go w postaci narzędzia open source, ale wtedy firma HashiCorp wypuściła pierwszą wersję Terraform.

Deklaratywne języki infrastruktury

Wiele narzędzi kodu infrastruktury, w tym Ansible, Chef, CloudFormation, Puppet i Terraform, używa języków deklaratywnych. Kod infrastruktury definiuje jej pożądany stan, na przykład, które pakiety i konta użytkowników powinny być na serwerze albo ile przydzielić mu pamięci RAM i procesora. Narzędzie obsługuje logikę tego, jak sprawić, aby uzyskać taki stan.

W przykładzie 4-2 mamy kod tworzący taki sam serwer, jak w przykładzie 4-1. Ten kod (jak większość przykładów w tej książce) jest napisany w języku fikcyjnym¹³.

Przykład 4-2 Przykład kodu deklaratywnego

```
virtual_machine:
  name: my_application_server
  source_image: 'base_linux'
  cpu: 2
  ram: 2GB
  network: private_network_segment
  provision:
    provisioner: servermaker
    role: tomcat_server
```

Ten kod nie zawiera żadnej logiki do sprawdzania, czy serwer już istnieje albo do czekania z uruchomieniem jego udostępniania, aż zacznie działać. Dbą o to narzędzie używane do zastosowania kodu. Narzędzie to porównuje ponadto aktualne atrybuty infrastruktury z zadeklarowanymi i ustala zmiany, które należy wprowadzić, aby odpowiednio dostosować infrastrukturę. Tak więc, aby zwiększyć pamięć RAM serwera aplikacji w tym przykładzie, trzeba dokonać zmiany w pliku i ponownie uruchomić narzędzie.

Deklaratywne narzędzia infrastruktury, takie jak Terraform i Chef, oddzielają to, co trzeba zrobić, od tego, jak to zrobić. W efekcie kod jest przejrzystszy i bardziej bezpośredni. Ludzie opisują czasem deklaratywny kod infrastruktury jako bliższy konfiguracji niż programowaniu.



Czy kod deklaratywny jest prawdziwym kodem?

Niektórzy ludzie nie uznają języków deklaratywnych, uważając je za zwykłą konfigurację, a nie „prawdziwy” kod.

¹³ Używam tego języka pseudokodu do ilustrowania koncepcji, które próbuję wyjaśnić, bez wiązania ich z konkretnym narzędziem.

Używam określenia *kod* w odniesieniu zarówno do języków deklaracyjnych, jak i imperatywnych. Kiedy muszę rozróżnić między jednym i drugim, mówię konkretnie „deklaratywny” lub „programowalny” albo stosuję jakiś inny wariant.

Nie uważam za przydatną debaty na temat tego, czy język kodowania musi być kompletny w sensie Turinga. Uważam nawet, że wyrażenia regularne są przydatne do pewnych celów, a one również nie są kompletne w sensie Turinga. Może po prostu nie jestem w 100% „prawdziwym” programistą.

Idempotencja

Ciągłe stosowanie kodu jest ważną praktyką, zapewniającą utrzymanie spójności kodu i kontrolę kodu infrastruktury, jak to zostało opisane w podrozdziale „Ciągłe stosowanie kodu” na stronie 342. Ta praktyka polega na ponawianiu stosowania kodu do infrastruktury w celu zapobieżenia dryfowi. Kod musi być *idempotentny*, aby mógł być ciągle bezpiecznie stosowany.

Kod idempotentny można uruchamiać dowolną liczbę razy nie zmieniając danych wyjściowych ani wyniku. W przypadku wielokrotnego uruchomienia narzędzia, które nie jest idempotentne, może dojść do powstania bałaganu.

Oto przykład skryptu powłoki, który nie jest idempotentny:

```
echo "spock:*:1010:1010:Spock:/home/spock:/bin/bash" \  
>> /etc/passwd
```

Jeśli uruchomimy ten skrypt raz, to uzyskamy pożądany wynik: użytkownik *spock* zostanie dodany do pliku */etc/passwd*. Ale jeśli uruchomimy go dziesięć razy, to powstanie dziesięć identycznych wpisów dla tego samego użytkownika.

W przypadku idempotentnego narzędzia infrastruktury określamy, czego oczekujemy:

```
user:  
  name: spock  
  full_name: Spock  
  uid: 1010  
  gid: 1010  
  home: /home/spock  
  shell: /bin/bash
```

Niezależnie od tego, ile razy uruchomimy narzędzie z tym kodem, w pliku */etc/passwd* na pewno będzie obecny tylko jeden wpis dla użytkownika *spock*. Bez niepożądanych efektów ubocznych.

Programowalne, imperatywne języki infrastruktury

Kod deklaracyjny jest dobry, gdy zawsze chcemy otrzymywać ten sam wynik. Ale są sytuacje, w których potrzebne są różne wyniki, zależnie od okoliczności. Na przykład kolejny kod powoduje utworzenie zbioru sieci VLAN. Dostawca chmury zespołu ShopSpinner ma w każdym kraju inną liczbę centrów danych i zespół chce, aby jego kod tworzył jedną sieć VLAN dla każdego z tych centrów. W efekcie kod musi dynamicznie rozpoznawać liczbę centrów danych i w każdym z nich tworzyć sieć VLAN:

```
this_country = getArgument("country")
data_centers = CloudApi.find_data_centers(country: this_country)
full_ip_range = 10.2.0.0/16

vlan_number = 0
for $DATA_CENTER in data_centers {
    vlan = CloudApi.vlan.apply(
        name: "public_vlan_${DATA_CENTER.name}"
        data_center: $DATA_CENTER.id
        ip_range: Networking.subrange(
            full_ip_range,
            data_centers.howmany,
            data_centers.howmany++
        )
    )
}
```

Ten kod przydziela również zakres adresów IP dla każdej sieci VLAN, używając fikcyjnej, ale wygodnej metody o nazwie `Networking.subrange()`. Metoda pobiera przestrzeń adresów zadeklarowaną w `full_ip_range`, dzieli ją na kilka mniejszych przestrzeni na podstawie wartości `data_centers.howmany` i zwraca jedną z nich, o indeksie wskazywanym przez `data_centers.howmany`.

Tego typu logiki nie można wyrazić przy użyciu kodu deklaracyjnego, dlatego większość deklaracyjnych narzędzi infrastruktury rozszerza swoje języki, dodając możliwości programowania imperatywnego. Na przykład Ansible dodaje do YAML pętle i instrukcje warunkowe¹⁴. HCL, język konfiguracyjny Terraform, jest często opisywany jako deklaracyjny, ale w rzeczywistości stanowi kombinację trzech języków podrzędnych¹⁵. Jednym z nich są wyrażenia¹⁶ zawierające pętle i instrukcje warunkowe.

Nowsze narzędzia, takie jak Pulumi i AWS CDK powracają do używania języków proceduralnych dla infrastruktury. Ich atrakcyjność w dużym stopniu wynika właśnie

¹⁴ <https://oreil.ly/-4wWs>

¹⁵ <https://oreil.ly/dFgG4>

¹⁶ <https://oreil.ly/qJQrd>

ze wsparcia dla języków programowania ogólnego przeznaczenia. Jednak są one również cenne ze względu na możliwość implementowania bardziej dynamicznego kodu infrastruktury.

Zamiast postrzegać deklaratywne lub imperatywne języki infrastruktury jako ten właściwy paradygmat, powinniśmy zająć się tym, do jakiego rodzaju zagadnień każdy z nich nadaje się najbardziej.

Języki deklaratywne czy imperatywne do infrastruktury

Kod deklaracyjny jest wygodny do definiowania pożądanego stanu systemu, zwłaszcza gdy nie ma dużej zmienności oczekiwanych wyników. Często definiujemy kształt jakiejś infrastruktury, którą chcemy powtarzać zachowując wysoki poziom spójności.

Na przykład zazwyczaj chcemy, aby wszystkie środowiska wspierające proces wydawania były niemal identyczne (patrz „Środowiska dostarczania” na stronie 59). Dlatego kod deklaracyjny jest dobry do definiowania środowisk lub fragmentów środowisk wielokrotnego użytku (zgodnie z wzorcem stosów wielokrotnego użytku omówionym w podrozdziale „Wzorec: stos wielokrotnego użytku” na stronie 65). Można nawet dopuszczać ograniczone różnice między instancjami infrastruktury zdefiniowanymi za pomocą kodu deklaracyjnego, wykorzystując parametry konfiguracyjne instancji, jak to jest opisane w rozdziale 7.

Czasami jednak chcemy napisać udostępniany kod wielokrotnego użytku, który może dawać różne wyniki w zależności od sytuacji. Na przykład zespół ShopSpinner pisze kod, który potrafi tworzyć infrastrukturę dla różnych serwerów aplikacji. Niektóre z tych serwerów są dostępne publicznie, więc potrzebują odpowiednich bram, reguł zapory, tras i rejestrowania. Inne serwery są używane wewnętrznie, więc mają inne wymagania dotyczące łączności i zabezpieczeń. Infrastruktura może się także różnić w przypadku aplikacji korzystających z przesyłania wiadomości, przechowywania danych i innych opcjonalnych elementów.

Ponieważ kod deklaracyjny obsługuje bardziej złożone warianty, potrzebuje zwiększonej ilości logiki. W pewnym momencie musimy zadać sobie pytanie, dlaczego piszemy złożoną logikę w YAML, JSON, XML lub innym języku deklaracyjnym.

Dlatego języki imperatywne, programowalne, są bardziej odpowiednie do budowania bibliotek i warstw abstrakcji, co jest omówione bardziej szczegółowo w rozdziale 16. I te języki mają zwykle lepsze możliwości pisania, testowania i zarządzania bibliotekami.

Języki dziedziczne infrastruktury

Oprócz możliwości deklaracyjnych wiele narzędzi infrastruktury wykorzystuje własny język DSL (*domain-specific language*), czyli język dziedziczny¹⁷.

17 Martin Fowler i Rebecca Parsons w swojej książce *Domain-Specific Languages* (<https://oreil.ly/hlyHx>) (Addison-Wesley Professional) definiują DSL jako „mały język, skoncentrowany na konkretnym aspekcie systemu oprogramowania”.

DSL jest językiem zaprojektowanym w celu modelowania określonej dziedziny, w naszym przypadku infrastruktury. Ułatwia to pisanie kodu i czyni go bardziej zrozumiałym, ponieważ ściśle odwzorowuje rzeczy, które definiujemy.

Na przykład każde z narzędzi Ansible, Chef i Puppet ma język DSL do konfigurowania serwerów. Języki te zapewniają konstrukcje dla takich pojęć, jak pakiety, pliki, usługi i konta użytkowników. Przykład pseudokodu języka DSL do konfiguracji serwera wygląda następująco:

```
package: jdk
package: tomcat

service: tomcat
  port: 8443
  user: tomcat
  group: tomcat

file: /var/lib/tomcat/server.conf
  owner: tomcat
  group: tomcat
  mode: 0644
  contents: $TEMPLATE(/src/appserver/tomcat/server.conf.template)
```

Ten kod powoduje instalację dwóch pakietów oprogramowania – jdk i tomcat. Definiuje też usługę, która powinna być uruchamiana, w tym port nasłuchiwania oraz użytkownika i grupę do „uruchamiania jako”. I wreszcie definiuje utworzenie pliku konfiguracyjnego serwera na podstawie pliku z szablonem.

Przykładowy kod jest dość prosty dla kogoś znającego się na administrowaniu serwerami, nawet jeśli nie miał do czynienia z tym konkretnym narzędziem lub językiem. Używanie języków służących do konfigurowania serwerów omówimy w rozdziale 11.

Języki DSL są również używane przez wiele narzędzi do zarządzania stosami, w tym Terraform i CloudFormation. Udostępniają one pojęcia z własnej dziedziny, platform infrastruktury, aby można było pisać bezpośrednio kod odwołujący się do maszyn wirtualnych, woluminów dysków i tras sieciowych. Więcej informacji o korzystaniu z tych języków i narzędzi zawiera rozdział 5.

Inne języki DSL infrastruktury modelują pojęcia dotyczące platformy środowiska wykonawczego aplikacji. Należą do nich klastry aplikacji, siatki usług i aplikacje. Przykłady to Helm Charts¹⁸ i manifesty aplikacji CloudFoundry¹⁹.

Wiele języków DSL powstało jako rozszerzenie istniejących języków znaczników, takich jak YAML (Ansible, CloudFormation, wszystko związane z Kubernetes) i JSON (Packer, CloudFormation). Niektóre z nich to wewnętrzne języki DSL, napisane jako podzbiór (lub nadzbiór) języka programowania ogólnego przeznaczenia. Chef jest przykładem wewnętrznego języka DSL, napisanego jako kod Ruby. Inne są zewnętrznymi

¹⁸ <https://oreil.ly/F14uU>

¹⁹ <https://oreil.ly/SBpsV>

językami DSL, interpretowanymi przez kod napisany w jeszcze innym języku. Terraform HCL to zewnętrzny język DSL; jego kod nie ma związku z językiem Go, w którym został napisany jego interpreter.

Języki ogólnego przeznaczenia czy DSL infrastruktury

Większość języków DSL infrastruktury to raczej języki deklaratywne niż imperatywne. Wewnętrzny język DSL, taki jak Chef, stanowi wyjątek, choć nawet on jest głównie deklaratywny²⁰.

Jedną z największych zalet używania języka ogólnego przeznaczenia, takiego jak JavaScript, Python, Ruby czy TypeScript, jest ekosystem narzędzi. Narzędzia te są bardzo dobrze obsługiwane przez środowiska IDE²¹, z zaawansowanymi funkcjami zwiększającymi produktywność, takimi jak wyróżnianie składni czy refaktoryzacja kodu. Szczególnie przydatną częścią ekosystemu języka programowania jest obsługa testowania.

Istnieje wiele narzędzi do testowania infrastruktury. Niektóre z nich są wymienione w podrozdziale „Weryfikacja: stosowanie asercji dotyczących zasobów infrastruktury” na stronie 129 oraz „Testowanie kodu serwera” na stronie 171. Ale niewiele z nich integruje się z językami w celu obsługi testowania jednostkowego. Jak się okaże w podrozdziale „Wyzwanie: testy kodu deklaratywnego mają często małą wartość” na stronie 105, może to nie stanowić problemu dla kodu deklaratywnego. Ale w przypadku kodu dającego bardziej zmienne wyniki, takiego jak biblioteki i warstwy abstrakcji, testy jednostkowe są kluczowe.

Zasady implementacji w przypadku definiowania infrastruktury jako kodu

Aby móc łatwo i bezpiecznie aktualizować i rozwijać systemy infrastruktury, baza kodu musi być dobrze uporządkowana: łatwa do zrozumienia, testowania, konserwacji i ulepszania. Jakość kodu to temat dobrze znany w inżynierii oprogramowania. Podane dalej zasady implementacji stanowią wskazówki, jak projektować i organizować kod, aby osiągnąć ten cel.

20 Imperatywny kod Ruby można wstawiać do „przepisów” Chefa, ale prowadzi to do bałaganu. Chef interpretuje „przepisy” w dwóch etapach, najpierw kompiluje kod Ruby, a następnie uruchamia kod w celu zastosowania zmian w serwerze. Kod proceduralny jest zwykle wykonywany podczas kompilacji. To sprawia, że kod Chef’a łączący kod proceduralny z imperatywnym jest trudny do zrozumienia. Z drugiej strony kod imperatywny jest przydatny przy pisaniu dostawców Chef’a, będących rodzajem biblioteki. To potwierdza tylko opinię, że język imperatywny nadaje się do kodu bibliotek, a język deklaratywny do definiowania infrastruktury.

21 *Integrated Development Environment* – specjalistyczny edytor języków programowania.

Oddzielaj kod deklaracyjny od imperatywnego

Kod będący połączeniem kodu deklaracyjnego i imperatywnego ma zapach, który sugeruje, że należy podzielić ten kod na oddzielne zagadnienia²².

Traktuj kod infrastruktury jak prawdziwy kod

Wiele baz kodu infrastruktury zaczyna się od zbioru plików konfiguracyjnych i skryptów narzędziowych, a z czasem zmienia się w niemożliwy do zarządzania bałagan. Zbyt często ludzie nie uważają kodu infrastruktury za „prawdziwy” kod. Nie stosują do niego tego samego poziomu dyscypliny inżynierskiej, co do kodu aplikacji. Aby zachować bazę kodu infrastruktury w postaci umożliwiającej zarządzanie, należy traktować ją jako sprawę najwyższej wagi.

Trzeba projektować kod infrastruktury i zarządzać nim tak, aby był łatwy do zrozumienia i utrzymania. Należy postępować zgodnie z praktykami dotyczącymi jakości kodu, takimi jak przeglądy kodu, programowanie w parach i zautomatyzowane testowanie. Zespół powinien zdawać sobie sprawę z długu technicznego i starać się go minimalizować.

W rozdziale 15 opisuję, jak stosować różne zasady projektowania oprogramowania do infrastruktury, takie jak poprawianie spójności i redukowanie sprzężenia. W rozdziale 18 wyjaśniam, jak organizować i zarządzać bazami kodu infrastruktury, aby praca z nimi była łatwiejsza.

Kod jako dokumentacja

Pisanie dokumentacji i dbanie o jej aktualność wymaga czasem zbyt dużo pracy. Do pewnych celów kod infrastruktury jest bardziej przydatny niż pisemna dokumentacja. Stanowi on zawsze dokładny i aktualny zapis systemu:

- Nowe osoby mogą przeglądać kod, aby poznać system.
- Członkowie zespołu mogą czytać kod i przeglądać zatwierdzenia, aby zobaczyć, co zrobili inni.
- Recenzenci techniczni mogą sięgać do kodu, aby ocenić, co należy poprawić.
- Audytorzy mogą przeglądać kod i historię wersji, aby uzyskać dokładny obraz systemu.

²² Termin *zapach projektu* pochodzi od *zapachu kodu* (<https://oreil.ly/meMKB>). „Zapach” to cecha obserwowanego systemu, która sugeruje istnienie ukrytego problemu. W zamieszczonym przykładzie kod łączący konstrukcje deklaracyjne z imperatywnymi ma zapach. Ten zapach sugeruje, że kod usiłuje wykonać wiele rzeczy i może lepiej by było rozbić je na oddzielne fragmenty, prawdopodobnie napisane w różnych językach.

Kod infrastruktury rzadko jest jedyną wymaganą dokumentacją. Dokumentacja wysokiego poziomu jest pomocna dla kontekstu i strategii. Mogą występować interesariusze, którzy muszą zrozumieć aspekty systemu, ale nie znają stosu technologicznego.

Również tymi innymi rodzajami dokumentacji można próbować zarządzać jako kodem. Wiele zespołów zapisuje rekordy decyzji dotyczących architektury (ADR – *architecture decision records*) w języku znaczników i trzyma je w systemie kontroli wersji.

Na podstawie kodu można automatycznie generować przydatne materiały, jak diagramy architektury i odniesienia do parametrów. Można robić to w potoku zarządzania zmianami, aby aktualizować dokumentację za każdym razem, gdy ktoś zmieni kod.

Podsumowanie

W tym rozdziale opisałem szczegółowo podstawowe praktyki definiowania systemu jako kodu. Obejmowało to zbadanie, dlaczego należy definiować rzeczy jako kod i jakie części systemu można definiować jako kod. Zasadnicza część rozdziału była poświęcona różnym paradygmatom języka infrastruktury. Może się to wydawać tematem abstrakcyjnym, jednak używanie właściwych języków we właściwy sposób jest kluczowym wyzwaniem przy tworzeniu efektywnej infrastruktury, wyzwaniem, na które branża nie znalazła jeszcze odpowiedzi. Dlatego pytanie o rodzaj języka, którego należy używać w różnych częściach systemu i jakie są konsekwencje takich decyzji, jest tematem, który będzie przewijał się przez całą książkę.

CZĘŚĆ II

Praca ze stosami infrastruktury

Tworzenie stosów infrastruktury jako kodu

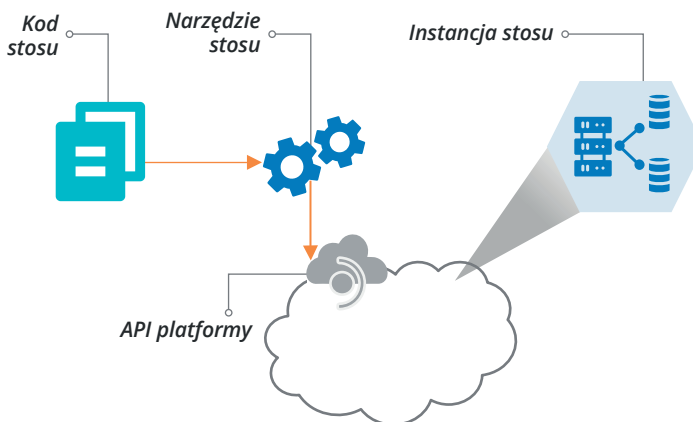
Ten rozdział łączy wiedzę z rozdziałów 3 i 4. Opisuję w nim sposoby używania kodu do zarządzania zasobami infrastruktury udostępnianymi przez platformę infrastruktury.

Do omówienia tego tematu wykorzystuję koncepcję tak zwanego stosu infrastruktury. Stos jest zbiorem zasobów infrastruktury, definiowanych i zmienianych jako całość. Zasoby zawarte w stosie są wyposażane łącznie, za pomocą narzędzi do zarządzania stosem. W efekcie powstaje nowa instancja stosu. Zmiany są wprowadzane w kodzie stosu i stosowane do instancji za pomocą tego samego narzędzia.

Rozdział zawiera opisy wzorców grupowania zasobów infrastruktury w stosy.

Co to jest stos infrastruktury?

Stos infrastruktury to zbiór zasobów infrastruktury definiowanych, wyposażanych i aktualizowanych jako jednostka (rysunek 5-1).



Rysunek 5-1 Stos infrastruktury to zbiór elementów infrastruktury zarządzanych jako grupa

Piszemy kod źródłowy, który definiuje elementy stosu, będące zasobami i usługami udostępnianymi przez platformę infrastruktury. Na przykład stos może obejmować maszynę wirtualną („Zasoby obliczeniowe” na stronie 26), wolumin dysku („Zasoby pamięci masowej” na stronie 27) oraz podsieć („Zasoby sieciowe” na stronie 28).

Następnie uruchamiamy narzędzie do zarządzania stosem, które czyta kod źródłowy stosu i wykorzystując API platformy chmury zbiera elementy zdefiniowane w kodzie w celu wyposażenia („skonstruowania”) instancji stosu.

Przykładowe narzędzia zarządzania stosem to:

- HashiCorp Terraform¹
- AWS CloudFormation²
- Azure Resource Manager³
- Google Cloud Deployment Manager⁴
- OpenStack Heat⁵
- Pulumi⁶
- Bosh⁷

Niektóre narzędzia do konfigurowania serwerów (znacznie więcej o nich powiem w rozdziale 11) mają rozszerzenia do pracy ze stosami infrastruktury. Na przykład Ansible Cloud Modules⁸, Chef Provisioning⁹ (obecnie wycofywane), Puppet Cloud Management¹⁰ oraz Salt Cloud¹¹.



„Stos” jako termin

Większość narzędzi do zarządzania stosem nie nazywa siebie narzędziami do zarządzania stosem. Każde narzędzie stosuje własną terminologię na określenie zarządzanej przez siebie jednostki infrastruktury. W tej książce opisuję wzorce i praktyki, które powinny być odpowiednie dla każdego z tych narzędzi.

Zdecydowałem się używać wyrazu *stos* (ang. *stack*).

Różne osoby mówiły mi, że są dużo lepsze określenia na to pojęcie. Każda z nich miała inną propozycję. W chwili, gdy to piszę, nadal nie ma w branży zgody, jaką nazwę należy przyjąć. Więc dopóki nic się nie pojawi, nadal będę używać wyrazu *stos*.

1 <https://oreil.ly/cvc1p>

2 <https://oreil.ly/drMmU>

3 <https://oreil.ly/eL26w>

4 <https://oreil.ly/1Wf2b>

5 <https://oreil.ly/MCK-d>

6 <https://www.pulumi.com>

7 <https://bosh.io/docs>

8 <https://oreil.ly/5grn4>

9 <https://oreil.ly/7DoV4>

10 <https://oreil.ly/I72s5>

11 https://oreil.ly/z4_eB

Kod stosu

Każdy stos jest zdefiniowany przez kod źródłowy, deklarujący elementy infrastruktury, które powinien zawierać. Kod Terraform (pliki `.tf`) i szablony CloudFormation są przykładami kodu stosu infrastruktury. Projekt stosu zawiera kod źródłowy definiujący infrastrukturę dla stosu.

W przykładzie 5-1 pokazana jest struktura folderów dla projektu kodu źródłowego stosu na użytek fikcyjnego narzędzia *Stackmaker*.

Przykład 5-1 Struktura folderów projektu stosu na użytek fikcyjnego narzędzia

```
stack-project/  
├── src/  
│   ├── dns.infra  
│   ├── load_balancers.infra  
│   ├── networking.infra  
│   └── webserver.infra  
└── test/
```

Instancja stosu

Używając jednego projektu stosu można wyposażyć więcej niż jedną instancję stosu. Po uruchomieniu narzędzia stosu dla projektu wykorzystuje ono API platformy, aby się upewnić, czy instancja stosu już istnieje i ewentualnie dopasować ją do kodu projektu. Jeśli instancja stosu nie istnieje, narzędzie ją utworzy. Jeśli zaś istnieje, ale nie jest idealnie zgodna z kodem, narzędzie modyfikuje ją, aby zapewnić tę zgodność.

Często opisując ten proces jako „stosowanie” kodu do instancji.

Jeśli zmienimy kod i ponownie uruchomimy to narzędzie, zmieni ono instancję stosu, aby była zgodna ze zmianami. Jeśli jeszcze raz uruchomimy narzędzie bez dokonywania zmian w kodzie, to instancja stosu pozostanie w tym stanie, w jakim była.

Konfigurowanie serwerów w stosie

Bazy kodu infrastruktury dla systemów, które nie są w pełni oparte na kontenerach lub bezserwerowej architekturze aplikacji, zwykle zawierają dużo kodu potrzebnego do udostępniania i konfigurowania serwerów. Nawet systemy oparte na kontenerach muszą tworzyć serwery hostów, aby uruchamiać kontenery. Do konfigurowania serwerów były używane pierwsze popularne narzędzia infrastruktury jako kodu, takie jak CFEngine, Puppet i Chef.

Należy oddzielać kod tworzący serwery od kodu tworzącego stosy. Takie podejście ułatwia zrozumienie kodu, upraszcza dokonywanie zmian przez ich rozdzielenie oraz sprzyja ponownemu wykorzystywaniu i testowaniu kodu serwera.

Kod stosu określa zwykle, jakie serwery mają zostać utworzone i przekazuje informacje o środowisku, w którym będą działać (poprzez wywołanie narzędzia konfiguracyjnego

serwera). W przykładzie 5-2 pokazana jest definicja stosu, która wywołuje fikcyjne narzędzie w celu skonfigurowania serwera.

Przykład 5-2 *Przykład definicji stosu wywołującego narzędzie do konfiguracji serwera*

```
virtual_machine:
  name: appserver-waterworks-${environment}
  source_image: shopspinner-base-appserver
  memory: 4GB
  provision:
    tool: servermaker
    parameters:
      maker_server: maker.shopspinner.xyz
      role: appserver
      environment: ${environment}
```

Ten stos definiuje instancję serwera aplikacji, tworzoną z obrazu serwera o nazwie shopspinner-appserver, z 4 GB pamięci RAM. Definicja zawiera klauzulę wyzwalającą proces udostępniania, który uruchamia narzędzie Servermaker. Kod przekazuje również kilka parametrów do wykorzystania przez narzędzie Servermaker. Parametry te obejmują adres serwera konfiguracji (maker_server) z plikami konfiguracyjnymi hostów oraz rolę, appserver, której Servermaker używa do ustalania, które pliki konfiguracyjne zastosować do danego serwera. Ponadto przekazuje nazwę środowiska, które może być wykorzystywane podczas konfiguracji do dostosowania serwera.

Języki infrastruktury niskiego poziomu

Większość języków popularnych narzędzi do zarządzania stosami to języki infrastruktury niskiego poziomu. Oznacza to, że taki język bezpośrednio udostępnia zasoby infrastruktury zapewniane przez platformę infrastruktury, z którą pracujemy (typy zasobów są wymienione w podrozdziale „Zasoby infrastruktury” na stronie 25).

Zadaniem osoby zajmującej się kodem infrastruktury jest napisanie kodu łączącego te zasoby w coś użytecznego.

Przykład 5-3 *Przykład kodu stosu infrastruktury niskiego poziomu*

```
address_block:
  name: application_network_tier
  address_range: 10.1.0.0/24"
  vlans:
    - appserver_vlan_A
      address_range: 10.1.0.0/16

virtual_machine:
  name: shopspinner_appserver_A
  vlan: application_network_tier.appserver_vlan_A
```

```
gateway:
  name: public_internet_gateway
  address_block: application_network_tier

inbound_route:
  gateway: public_internet_gateway
  public_ip: 192.168.99.99
  incoming_port: 443
  destination:
    virtual_machine: shopspinner_appserver_A
    port: 8443
```

Ten wymyślony i uproszczony przykład pseudokodu definiuje maszynę wirtualną, blok adresów i sieć VLAN oraz bramę internetową. Następnie wiąże je ze sobą definiując połączenie przychodzące, które kieruje połączenia przychodzące na adres *https://192.168.99.99* do portu 8443 maszyny wirtualnej¹².

Platforma może sama zapewniać wyższą warstwę abstrakcji, na przykład udostępniając klaster hostingu aplikacji. W takiej sytuacji element klastra udostępniany przez platformę mógłby automatycznie udostępniać instancje serwera i trasy sieciowe. Natomiast kod infrastruktury niskiego poziomu stosuje bezpośrednie mapowanie na zasoby i opcje eksponowane przez API platformy.

Języki infrastruktury wysokiego poziomu

Język infrastruktury wysokiego poziomu definiuje jednostki, które nie są mapowane bezpośrednio na zasoby udostępniane przez odpowiednią platformę. Na przykład wersja kodu wysokiego poziomu z przykładu 5-3 mogłaby deklarować podstawy serwera aplikacji, jak to jest pokazane w przykładzie 5-4.

Przykład 5-4 *Przykład kodu stosu infrastruktury wysokiego poziomu*

```
application_server:
  public_ip: 192.168.99.99
```

W tym przykładzie zastosowanie kodu powoduje albo udostępnienie zasobów sieciowych i serwerowych z poprzedniego przykładu, albo wykrycie istniejących zasobów do wykorzystania. To narzędzie lub biblioteka, którą ten kod wywołuje, decyduje o wartościach, których należy użyć dla portów sieciowych i sieci VLAN, oraz jak utworzyć serwer wirtualny.

Wiele rozwiązań hostingu aplikacji, takich jak platforma PaaS lub spakowany klaster (patrz „Spakowana dystrybucja klastra” na stronie 225), oferuje ten poziom abstrakcji. Wystarczy napisać deskryptor wdrożenia dla swojej aplikacji, a platforma przydzieli zasoby infrastruktury potrzebne do wdrożenia.

¹² Nie, tak naprawdę to nie jest publiczny adres IP.

W innych przypadkach można utworzyć własną warstwę abstrakcji pisząc odpowiednie biblioteki i moduły. Więcej na ten temat znajdziemy w rozdziale 16.

Wzorce i antywzorce konstruowania stosów

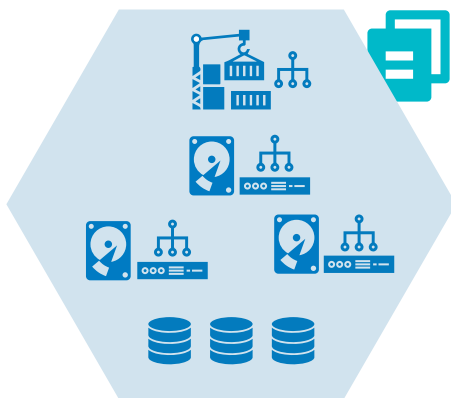
Jednym z wyzwań towarzyszących projektowaniu infrastruktury jest ustalenie rozmiaru i struktury stosów. Można utworzyć projekt kodu z jednym stosem do zarządzania całym systemem. Okazuje się to jednak nieporęczne w przypadku rozbudowy systemu. W tej części opiszę wzorce i antywzorce konstruowania stosów infrastruktury.

Wszystkie poniższe wzorce opisują sposoby grupowania elementów systemu w jeden lub więcej stosów. Można traktować je jako kontinuum:

- *Stos monolityczny* umieszcza cały system w jednym stosie.
- *Stos grup aplikacji* grupuje powiązane elementy systemu w oddzielne stosy.
- *Stos usług* umieszcza całą infrastrukturę dla jednej aplikacji w jednym stosie.
- *Mikrostos* rozбивa infrastrukturę dla danej aplikacji lub usługi na wiele stosów.

Antywzorzec: stos monolityczny

Stos monolityczny to stos infrastruktury obejmujący zbyt wiele elementów, przez co jest trudny do obsługi (rysunek 5-2).



Rysunek 5-2 *Stos monolityczny to stos infrastruktury obejmujący zbyt wiele elementów*

Tym, co odróżnia stos monolityczny od innych wzorców, jest to, że na skutek liczby zawartych w nim elementów lub ich powiązań trudno jest nim dobrze zarządzać.

Motywacja

Ludzie tworzą stosy monolityczne, ponieważ najprostszym sposobem dodania nowego elementu do systemu jest dodanie go do istniejącego projektu. Każdy nowy stos oznacza więcej części, wymagających orkiestracji, integracji i testowania.

Pojedynczy stos jest prosty w zarządzaniu. W przypadku zbioru elementów systemu o niewielkich rozmiarach stos monolityczny ma sens. Ale najczęściej stos monolityczny nieuchronnie rośnie, wymykając się spod kontroli.

Zastosowanie

Stos monolityczny bywa odpowiedni dla małych i prostych systemów. Nie nadaje się, gdy system się rozrasta, przez co udostępnianie i aktualizacja zajmują więcej czasu.

Konsekwencje

Zmienianie dużego stosu jest bardziej ryzykowne niż małego. Więcej rzeczy może pójść nie tak – mamy większy promień wybuchu. Skutki błędnej zmiany mogą być rozleglejsze, ponieważ taki stos obejmuje więcej usług i aplikacji. Większe stosy oznaczają również wolniejsze wyposażanie i modyfikowanie, przez co trudniej się nimi zarządza.

W efekcie wolniejszego tempa i większego ryzyka zmian w stosie monolitycznym ludzie rzadziej się decydują na wprowadzanie zmian, a ich dokonywanie zajmuje więcej czasu. Te dodatkowe problemy mogą prowadzić do wyższego poziomu długu technicznego.



Promień wybuchu

Bezpośredni *promień wybuchu* to zakres kodu objęty przez polecenie wprowadzające zmiany¹³. Na przykład bezpośredni promień wybuchu dla polecenia terraform apply obejmuje cały kod projektu. Pośredni promień wybuchu obejmuje inne elementy systemu, te które zależą od zasobów w bezpośrednim promieniu wybuchu i które mogą odczuć uszkodzenie tych zasobów.

Implementacja

Stos monolityczny powstaje przez utworzenie projektu stosu infrastruktury, a następnie ciągle dodawanie kodu, zamiast rozbijania go na wiele stosów.

Powiązane wzorce

Przeciwnieństwem wzorca stosu monolitycznego jest mikrostos (patrz „Wzorzec: mikrostos” na stronie 57), którego celem jest zachowanie małych rozmiarów stosów, łatwiejszych do utrzymania i ulepszania. Stosem monolitycznym może stać się stos grupy aplikacji (patrz „Wzorzec: stos grupy aplikacji” na stronie 54), którego rozrost wymknie się spod kontroli.

13 Charity Majors (<https://oreil.ly/8KcSk>) spopularyzowała termin *promień wybuchu* w kontekście infrastruktury jako kodu. W swoim poście „Terraform, VPC, and Why You Want a tfstate File Per env” (<https://oreil.ly/pWONA>) opisuje zasięg potencjalnych szkód, które dana zmiana mogłaby wyrządzić w systemie.

Czy mój stos jest monolitem?

To, czy stos infrastruktury jest monolitem, jest kwestią oceny. Oto kilka symptomów stosu monolitycznego:

- Trudno jest zrozumieć, jak elementy stosu łączą się ze sobą (może panować w nich za duży nieład, żeby je ogarnąć, albo nie pasują do siebie zbyt dobrze).
- Nowe osoby potrzebują trochę czasu na poznanie bazy kodu stosu.
- Debugowanie problemów ze stosem jest trudne.
- Zmiany stosu powodują często problemy.
- Zbyt dużo czasu trzeba poświęcać na utrzymywanie systemów i procesów, których celem jest zarządzanie złożonością stosu.

Kluczowym wskaźnikiem tego, czy stos staje się monolityczny, jest liczba osób pracujących nad jego zmianami w danym momencie. Im częściej nad stosem pracuje wiele osób jednocześnie, tym więcej czasu trzeba poświęcać na koordynowanie zmian. Jeszcze gorsza jest sytuacja, gdy zmiany w jednym stosie dokonywane są przez wiele zespołów. Jeśli wdrażanie zmian w danym stosie powoduje często błędy i konflikty, to być może jest on za duży.

Strategią radzenia sobie z tym jest rozgałęzianie funkcji¹⁴, ale może to prowadzić do dodatkowych utrudnień i zwiększenia kosztów dostarczania. Zwyczajowe używanie rozgałęzień funkcji podczas pracy ze stosem sugeruje, że stał się on monolitem.

Bardziej zrównoważonym sposobem zapewnienia bezpiecznej pracy wielu osób nad jednym stosem jest CI, czyli ciągła integracja¹⁵. Ale w miarę jak stos staje się coraz bardziej monolityczny, wydłuża się czas budowy CI i trudniej jest utrzymać dobrą dyscyplinę budowy. Jeśli praktyka CI zespołu jest niechlujna, to kolejny znak, że stos jest monolitem.

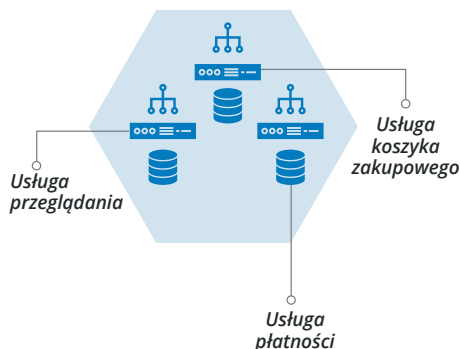
Problemy te dotyczą pojedynczego zespołu pracującego nad stosem infrastruktury. Wiele zespołów pracujących nad wspólnym stosem to wyraźny znak, że należy rozważyć jego podział na łatwiejsze do zarządzania części.

Wzorzec: stos grupy aplikacji

Stos grupy aplikacji zawiera infrastrukturę dla wielu powiązanych ze sobą aplikacji lub usług. Infrastruktura dla tych wszystkich aplikacji jest udostępniana i zarządzana jako grupa, jak widać na rysunku 5-3.

¹⁴ <https://oreil.ly/025IQ>

¹⁵ <https://oreil.ly/uJwYF>



Rysunek 5-3 Stos grupy aplikacji obsługuje wiele procesów w jednej swojej instancji

Na przykład stos aplikacji produktów ShopSpinner obejmuje oddzielne usługi do przeglądania produktów, wyszukiwania ich i zarządzania koszykiem. Serwery i pozostała infrastruktura dla tego wszystkiego stanowią wspólnie jedną instancję stosu.

Motywacja

Zdefiniowanie wspólnej infrastruktury dla wielu powiązanych usług może ułatwić zarządzanie aplikacją jako pojedynczą jednostką.

Zastosowanie

Wzorzec ten może się dobrze sprawdzać, gdy za infrastrukturę i wdrażanie wszystkich elementów aplikacji odpowiada jeden zespół. Stos grupy aplikacji pozwala dopasować granice stosu do zakresu odpowiedzialności zespołu.

Jako element pośredni między stosem monolitycznym a stosem usług bywa czasem przydatny stos wielousługowy.

Konsekwencje

Grupowanie infrastruktury dla wielu aplikacji wpływa również na czas, ryzyko i tempo zmian. Zespół musi zarządzać ryzykiem dla całego stosu przy każdej zmianie, nawet jeśli dotyczy ona tylko jednej jego części. Ten wzorzec jest nieefektywny, jeśli niektóre części stosu zmieniają się częściej niż inne.

Czas wyposażania, wprowadzania zmian i testowania zależy od całego stosu. Więc znowu, jeśli zmiana dotyczy zwykle tylko jednej części stosu, grupowanie powoduje tylko niepotrzebne obciążenie i ryzyko.

Implementacja

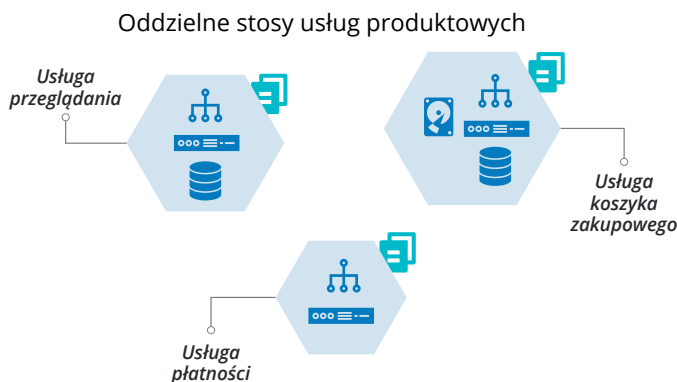
Aby utworzyć stos grupy aplikacji, trzeba zdefiniować projekt infrastruktury tworzący infrastrukturę dla całego zbioru usług. Wszystkie te elementy można wyposażać i niszczyć za pomocą jednego polecenia.

Powiązane wzorce

Wraz z rozbudową wzorec ten niesie ryzyko przekształcenia się w stos monolityczny (patrz „Antywzorec: stos monolityczny” na stronie 52). Z kolei podział usług w stosie grupy aplikacji pomiędzy oddzielne stososy powoduje powstanie stosu usług.

Wzorec: stos usług

Stos usług zarządza infrastrukturą dla każdego możliwego do wdrożenia składnika aplikacji w oddzielnym stosie infrastruktury (rysunek 5-4).



Rysunek 5-4 Stos usług zarządza infrastrukturą dla każdego możliwego do wdrożenia składnika aplikacji w oddzielnym stosie infrastruktury

Motywacja

Stosy usług dopasowują granice infrastruktury do oprogramowania, które w niej działa. Takie dopasowanie ogranicza promień wybuchu dla zmiany jednej usługi, co upraszcza proces planowania zmian. Zespoły usług mogą dysponować infrastrukturami związanymi z ich oprogramowaniem.

Zastosowanie

Stosy usług mogą dobrze działać z architekturami aplikacji mikrousług. Pomagają również organizacjom z autonomicznymi zespołami zapewnić każdemu z nich jego własną infrastrukturę¹⁶.

Konsekwencje

Jeśli mamy wiele aplikacji, każda ze stosem infrastruktury, może zachodzić niepotrzebne duplikowanie kodu. Na przykład każdy stos może zawierać kod określający sposób

¹⁶ Patrz „Mikrousługi” (<https://oreil.ly/NRoab>) Jamesa Lewisa. Warto przeczytać również „The Art of Building Autonomous Teams” (<https://oreil.ly/cKcgK>) Johna Fergusona Smarta.

wyposażania serwera aplikacji. Powielanie może prowadzić do niespójności, takiej jak używanie różnych wersji systemu operacyjnego albo różnych konfiguracji sieci. Można temu zaradzić, udostępniając kod za pomocą modułów (jak w rozdziale 16).

Implementacja

Każda aplikacja lub usługa ma oddzielny projekt kodu infrastruktury. Podczas tworzenia nowej aplikacji zespół może skopiować kod z infrastruktury innej aplikacji. Może też użyć projektu referencyjnego ze standardowym kodem do tworzenia nowych stosów.

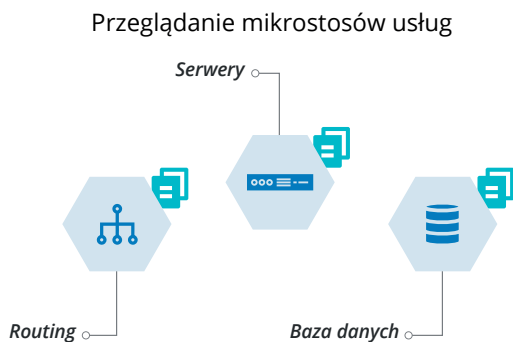
W niektórych przypadkach każdy stos może być kompletny, bez infrastruktury dzielonej z innymi stosami aplikacji. W innych przypadkach zespoły mogą tworzyć stosy z infrastrukturą obsługującą wiele stosów aplikacji. Więcej o wzorcach dla tego typu sytuacji można dowiedzieć się w rozdziale 17.

Powiązane wzorce

Wzorzec stosu usług plasuje się między stosem grup aplikacji („Wzorzec: stos grupy aplikacji” na stronie 54), który zawiera wiele aplikacji w jednym stosie, a mikrostosem, który rozbija infrastrukturę dla jednej aplikacji na wiele stosów.

Wzorzec: mikrostos

Wzorzec mikrostosu dzieli infrastrukturę dla jednej aplikacji na wiele stosów (rysunek 5-5).



Rysunek 5-5 Mikrostos dzieli infrastrukturę dla jednej aplikacji na wiele stosów

Na przykład można mieć oddzielny projekt stosu dla sieci, oddzielny dla serwerów i oddzielny dla bazy danych.

Motywacja

Poszczególne części infrastruktury usługi mogą zmieniać się w różnym tempie. Mogą mieć też inne cechy, które sprawiają, że łatwiej byłoby zarządzać nimi oddzielnie. Na przykład

niektóre metody zarządzania instancjami serwerów wymagają ich częstego niszczenia i odbudowy¹⁷. Pewne usługi używają jednak trwałych danych przechowywanych w bazie danych lub woluminie dysku. Zarządzanie serwerami i danymi w oddzielnych stosach oznacza, że mogą mieć różne cykle życia, przy stosie serwera odbudowywanym znacznie częściej, niż stos danych.

Konsekwencje

Chociaż mniejsze stosy są z natury prostsze, większa liczba zmieniających się elementów zwiększa złożoność. W rozdziale 17 opisuję techniki obsługi integracji wielu stosów.

Implementacja

Dodanie nowego mikrostosu wiąże się z utworzeniem nowego projektu stosu. Trzeba wyznaczyć granice między stosami we właściwych miejscach, aby zapewnić im odpowiedni rozmiar i łatwe zarządzanie. Powiązane wzorce oferują rozwiązania tego problemu. Może być również potrzebna integracja różnych stosów, którą opisuję w rozdziale 17.

Powiązane wzorce

Mikrostosy znajdują się na przeciwnym krańcu widma niż stos monolityczny (patrz „Antywzorzec: stos monolityczny” na stronie 52), w którym pojedynczy stos zawiera całą infrastrukturę systemu.

Podsumowanie

Stosy infrastruktury są podstawowymi cegiełkami zautomatyzowanej infrastruktury. Wzorce opisane w tym rozdziale stanowią punkt wyjścia do myślenia o organizowaniu infrastruktury w stosy.

¹⁷ Na przykład serwery niezmiennialne (patrz „Wzorzec: serwer niezmiennialny” na stronie 189).

Tworzenie środowisk przy użyciu stosów

W rozdziale 5 opisałem stos infrastruktury jako kolekcję zasobów infrastruktury, którą można zarządzać jak pojedynczą jednostką. Środowisko to także kolekcja zasobów infrastruktury. Czy w takim razie stos jest tym samym, co środowisko? Jak pokażę w tym rozdziale, być może tak, a może nie.

Środowisko (*environment*) to kolekcja oprogramowania i zasobów infrastruktury zorganizowanych wokół konkretnego celu, na przykład obsługi fazy testowania lub świadczenia usług w jakimś regionie geograficznym. Stos albo zbiór stosów to sposób definiowania kolekcji zasobów infrastruktury i zarządzania nią. Używamy więc stosu lub wielu stosów do implementacji środowiska. Można zaimplementować środowisko jako pojedynczy stos lub można zestawić środowisko z wielu stosów. Można nawet utworzyć kilka środowisk w jednym stosie, chociaż nie powinno się tego robić.

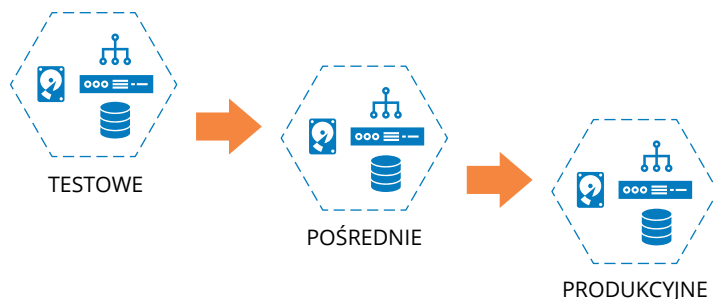
O co chodzi w tych środowiskach

Pojęcie środowiska jest jedną z tych rzeczy, które w IT przyjmujemy za pewnik. Często jednak mamy na myśli nieco inne rzeczy, gdy używamy tego terminu w różnym kontekście. W tej książce terminu *środowisko* używam jako kolekcji operacyjnie powiązanych zasobów infrastruktury. Oznacza to, że zasoby w środowisku obsługują określone działanie, takie jak testowanie lub uruchamianie systemu. Najczęściej istnieje wiele środowisk, z uruchomioną w każdym z nich instancją tego samego systemu.

Istnieją dwa typowe przypadki użycia, w których wiele środowisk ma uruchomione instancje tego samego systemu. Jednym z nich jest obsługa procesu dostarczania, a drugim uruchamianie wielu instancji produkcyjnych systemu.

Środowiska dostarczania

Najbardziej znanym przypadkiem wykorzystywania wielu środowisk jest obsługa progresywnego procesu wydawania oprogramowania – nazywanego czasem ścieżką do produkcji. Kompilacja aplikacji jest wdrażana po kolei w każdym środowisku w celu obsługi różnych działań związanych z programowaniem i testowaniem, aż ostatecznie zostaje wdrożona w środowisku produkcyjnym (rysunek 6-1).



Rysunek 6-1 Środowiska dostarczania ShopSpinner

Będę wykorzystywał ten zbiór środowisk w całym tym rozdziale do ilustrowania wzorców definiowania środowisk jako kodu.

Wiele środowisk produkcyjnych

Można również używać wielu środowisk dla kompletnych, niezależnych kopii systemu używanych w produkcji. Oto niektóre powody:

Odporność na błędy

Jeśli jedno środowisko ulegnie awarii, to inne nadal będą udostępniać usługę. Taka sytuacja może obejmować proces awaryjnego przenoszenia obciążenia z uszkodzonego środowiska na inne. Można również zapewnić odporność na błędy wewnątrz środowiska dysponując wieloma instancjami jakiejś części infrastruktury, jak w przypadku klastra serwerów. Uruchamianie dodatkowych środowisk powoduje powielanie całej infrastruktury i zapewnia wyższy poziom odporności na błędy, chociaż przy wyższych kosztach. O strategiach ciągłości wykorzystujących infrastrukturę jako kod piszę w podrozdziale „Ciągłość” na stronie 367.

Skalowalność

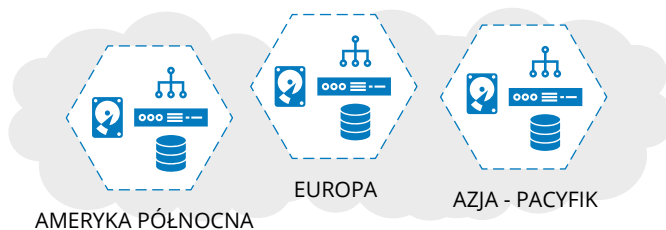
Mając wiele środowisk można wykorzystywać je do rozkładu obciążenia. Często robi się to tworząc oddzielne środowiska dla różnych regionów geograficznych. Wiele środowisk może służyć do uzyskania zarówno skalowalności, jak i odporności na błędy. Jeśli w jakimś regionie następuje awaria, obciążenie jest przenoszone do środowiska innego regionu, dopóki awaria nie zostanie usunięta.

Podział

Można uruchamiać wiele instancji jakiejś aplikacji lub usługi dla różnych baz użytkowników, na przykład różnych klientów. Uruchamianie tych instancji w różnych środowiskach pozwala wzmocnić podział, a silniejszy podział może z kolei pomóc spełnić wymagania prawne lub warunki zgodności i zwiększyć zaufanie klientów.

ShopSpinner uruchamia oddzielny serwer aplikacji dla każdego ze swoich klientów e-commerce. Wraz z rozszerzeniem obsługi na klientów w Ameryce Północnej, Europie

i Południowej Azji, firma postanowiła utworzyć oddzielne środowisko dla każdego z tych regionów (rysunek 6-2).



Rysunek 6-2 Środowiska regionalne firmy ShopSpinner

Stosowanie w pełni rozdzielonych środowisk zamiast jednego, obejmującego wszystkie regiony, pomaga firmie ShopSpinner zapewnić zgodność z różnymi lokalnymi przepisami dotyczącymi przechowywania danych klientów. Ponadto, gdy zachodzi potrzeba dokonania zmian wymagających przestoju, można to robić w każdym regionie w innym czasie. Dzięki temu łatwiej jest dostosować przestoje do różnych stref czasowych.

W pewnym momencie firma ShopSpinner podpisała kontrakt z siecią sklepów farmaceutycznych o nazwie The Medicine Barn. Z powodów prawnych The Medicine Barn nie może przechowywać danych swoich klientów razem z innymi firmami. Dlatego zespół ShopSpinner zaproponował uruchomienie całkowicie oddzielnego środowiska przeznaczonego dla The Medicine Barn, za wyższą cenę niż w przypadku środowiska współdzielonego.

Środowiska, spójność i konfiguracja

Ponieważ wiele środowisk ma służyć do uruchamiania instancji tego samego systemu, infrastruktura tych środowisk powinna być spójna. Spójność środowisk jest jednym z głównych powodów rozwoju infrastruktury jako kodu.

Różnice między środowiskami stwarzają ryzyko niespójnego zachowania. Testowanie oprogramowania w jednym środowisku może nie ujawnić problemów występujących w innym. Może się nawet zdarzyć, że oprogramowanie zostanie wdrożone pomyślnie w jednych środowiskach, a w innych nie.

Z drugiej strony pewne specyficzne różnice między środowiskami są zwykle potrzebne. Środowiska testowe bywają mniejsze od środowisk produkcyjnych. Różne osoby mogą mieć różne uprawnienia w różnych środowiskach. Środowiska dla różnych klientów mogą mieć różne funkcje i cechy. A co najmniej nazwy i identyfikatory mogą być różne (*appserver-test*, *appserver-stage*, *appserver-prod*). Trzeba więc konfigurować przynajmniej niektóre aspekty środowisk.

Kluczowe sprawy w przypadku środowisk to testowanie oraz strategia dostarczania. Gdy ten sam kod infrastruktury jest stosowany do każdego środowiska, testowanie go w jednym środowisku na ogół daje pewność, że będzie działał prawidłowo w pozostałych

środowiskach. Takiej pewności nie ma jednak, jeśli instancje infrastruktury różnią się znacząco między sobą.

Można próbować zwiększyć pewność poprzez testowanie kodu infrastruktury przy użyciu różnych wartości konfiguracyjnych. Jednak tego typu testowanie bywa niepraktyczne. W takich sytuacjach może zachodzić potrzeba dodatkowej kontroli, takiej jak testy po udostępnieniu lub monitorowanie środowisk produkcyjnych. Testowaniem i dostarczaniem zajmę się bardziej szczegółowo w rozdziale 8.

Wzorce budowania środowisk

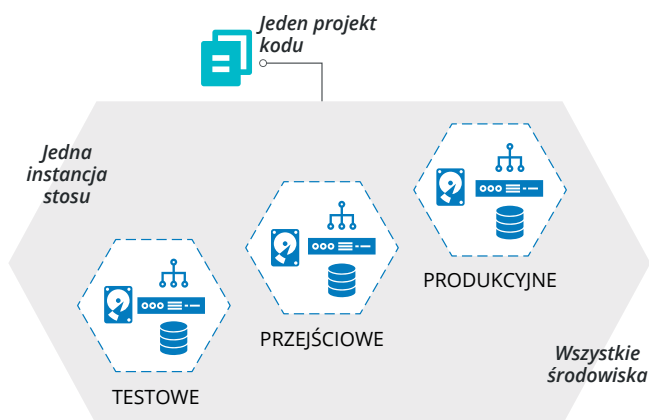
Jak wspomniałem wcześniej, środowisko jest konceptualną kolekcją elementów infrastruktury, a stos jest konkretną kolekcją elementów infrastruktury. Projekt stosu jest kodem źródłowym używanym do tworzenia jednej lub więcej instancji stosu. Jak więc należy wykorzystywać projekty i instancje stosów do implementowania środowisk?

Opiszę dwa antywzorce i jeden wzorec implementowania środowisk przy użyciu stosów infrastruktury. Każdy z tych wzorców opisuje sposób definiowania wielu środowisk przy użyciu stosów infrastruktury. Niektóre systemy składają się z wielu stosów, jak to opisałem w podrozdziale „Wzorce i antywzorce konstruowania stosów” na stronie 52. Jak to wygląda w przypadku wielu środowisk, wyjaśnię w podrozdziale „Tworzenie środowisk z wieloma stosami” na stronie 67.

Antywzorzec: stos wielu środowisk

Stos wielu środowisk definiuje infrastrukturę dla wielu środowisk i zarządza nią jako instancją jednego stosu.

Na przykład, jeśli mamy trzy środowiska do testowania i uruchamiania aplikacji, to projekt jednego stosu zawiera kod dla wszystkich tych trzech środowisk (rysunek 6-3).



Rysunek 6-3 Stos wielu środowisk definiuje infrastrukturę dla wielu środowisk i zarządza nią jako instancją jednego stosu

Motywacje

Wiele osób tworzy tego typu strukturę podczas nauki nowego narzędzia do stosów, ponieważ dodawanie nowych środowisk do już istniejącego projektu wydaje się naturalne.

Konsekwencje

Podczas uruchamiania narzędzia do aktualizacji instancji stosu zasięg potencjalnych zmian obejmuje cały stos. Jeśli w kodzie występuje jakiś błąd lub konflikt, to cała instancja jest podatna na zagrożenie¹.

Jeśli środowisko produkcyjne jest w tej samej instancji stosu, co inne środowisko, zmiana tego drugiego niesie ze sobą ryzyko spowodowania problemów ze środowiskiem produkcyjnym. Błąd kodowania, niespodziewana zależność czy nawet nieprawidłowość w samym narzędziu mogą popsuć środowisko produkcyjne, mimo że celem miała być jedynie zmiana środowiska testowego.

Powiązane wzorce

Można ograniczyć promień wybuchu dla zmian, umieszczając środowiska w oddzielnych stosach. Jednym z oczywistych sposobów na to jest środowisko *kopiuj-wklej*, w którym każde środowisko jest projektem oddzielnego stosu, chociaż takie rozwiązanie jest uznawane za antywzorzec.

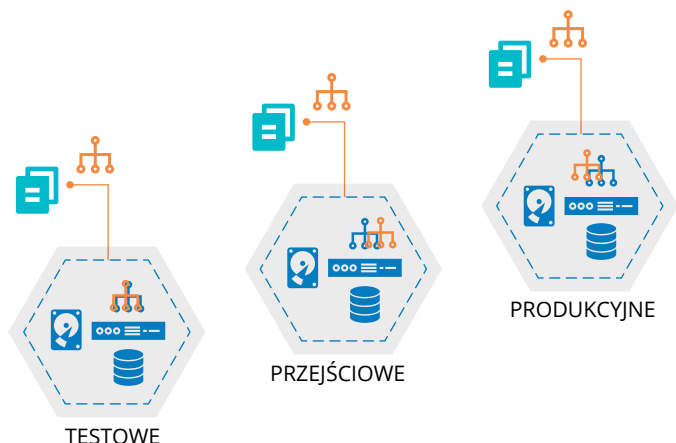
Lepszym podejściem jest wzorzec stosu wielokrotnego użytku (patrz „Wzorzec: stos wielokrotnego użytku” na stronie 65). W tym wzorcu pojedynczy projekt jest używany do zdefiniowania ogólnej struktury dla środowiska, a następnie używany do zarządzania oddzielnymi instancjami stosu w poszczególnych środowiskach. Chociaż oznacza to korzystanie z jednego projektu, to jest on za każdym razem stosowany tylko do jednej instancji środowiska. W efekcie promień wybuchu dla zmian jest ograniczony do tego jednego środowiska.

Antywzorzec: środowiska kopiuj-wklej

Wzorzec środowiska kopiuj-wklej wykorzystuje oddzielny projekt kodu źródłowego stosu dla każdej instancji stosu infrastruktury.

W naszym przykładzie z trzema środowiskami o nazwach *test*, *staging* i *production*, występuje oddzielny projekt infrastruktury dla każdego z tych trzech środowisk (rysunek 6-4). Zmiany są dokonywane poprzez edytowanie kodu w jednym środowisku, a następnie kopiowanie tych zmian po kolei do każdego z pozostałych.

¹ Charity Majors podzieliła się na blogu (<https://oreil.ly/pWONA>) swoimi bolesnymi doświadczeniami z pracy ze stosem wielu środowisk.



Rysunek 6-4 Środowisko kopiuj-wklej wykorzystuje oddzielną kopię projektu kodu źródłowego dla każdej instancji

Motywacja

Środowiska kopiuj-wklej są intuicyjnym sposobem na utrzymanie wielu środowisk. Unikają problemu promienia wybuchu występującego w przypadku antywzorca stosu wielu środowisk. Każdą instancję stosu można też łatwo dostosowywać.

Zastosowanie

Środowiska kopiuj-wklej bywają odpowiednie, gdy chcemy utrzymywać i zmieniać różne instancje i nie martwić się o powielanie lub spójność kodu.

Konsekwencje

Utrzymywanie wielu środowisk kopiuj-wklej może być nie lada wyzwaniem. W przypadku konieczności dokonania zmiany kodu trzeba skopiować ją do każdego projektu. Prawdopodobnie trzeba wtedy też przetestować każdą instancję oddzielnie, ponieważ zmiana może działać prawidłowo w jednej, ale niekoniecznie w pozostałych.

W środowiskach kopiuj-wklej często występuje dryf konfiguracji (patrz „Dryf konfiguracji” na stronie 17). Używanie tych środowisk w przypadku środowisk dostarczania zmniejsza niezawodność procesu wdrażania i poprawność testów na skutek niespójności środowisk.

Środowiska kopiuj-wklej mogą być spójne zaraz po pierwszym skonfigurowaniu, ale z czasem pojawiają się między nimi różnice.

Implementacja

Aby utworzyć środowisko kopiuj-wklej, trzeba skopiować kod projektu z odpowiedniej instancji stosu do nowego projektu. Następnie należy zmienić ten kod, aby dostosować

go do nowej instancji. Każda zmiana dokonana w jednym stosie musi zostać skopiowana do wszystkich pozostałych projektów stosu, z zachowaniem ich indywidualnych ustawień.

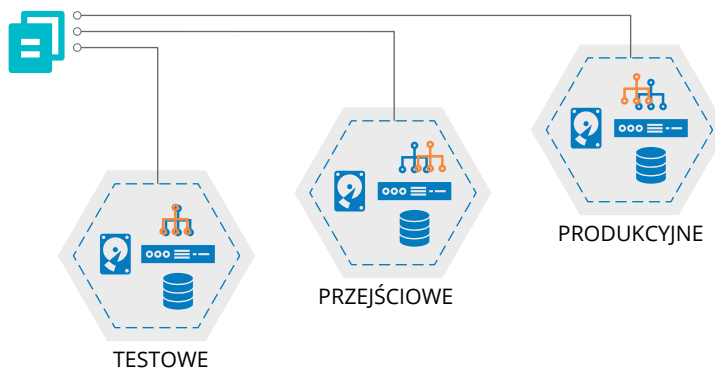
Powiązane wzorce

Gałęzie środowiska (patrz „Dostarczanie kodu z repozytorium kodu źródłowego” na stronie 317) mogą być uważane za odmianę środowisk kopiuj-wklej. Każda gałąź zawiera kopię kodu, a kopiowanie kodu między gałęziami odbywa się poprzez scalanie (*merging*). Ciągłe stosowanie kodu (patrz „Ciągłe stosowanie kodu” na stronie 342) może pomóc uniknąć pułapek wersji kopiuj-wklej, ponieważ gwarantuje brak przenoszenia modyfikacji kodu z jednego środowiska do drugiego. Edytowanie kodu w ramach scalania go z gałęzią środowiska niesie ze sobą zagrożenia antywzorca kopiuj-wklej.

Do środowisk kopiuj-wklej podobny jest również wzorzec stos opakowujący (patrz „Wzorzec: stos opakowujący” na stronie 84). Stos opakowujący wykorzystuje oddzielny projekt stosu dla każdego środowiska w celu ustawienia parametrów konfiguracyjnych. Jednak kod stosu jest implementowany w składnikach stosu, na przykład jako kod modułu wielokrotnego użytku. Taki kod nie jest kopiowany i wklejany do każdego środowiska, ale promowany, podobnie jak stos wielokrotnego użytku. Jeśli jednak będziemy dodawać do projektów stosu opakowującego coś więcej niż podstawowe parametry instancji stosu, może on przerodzić się w antywzorzec typu środowisko kopiuj wklej. W przypadkach, w których instancje stosu mają reprezentować ten sam stos, zwykle bardziej odpowiednim wzorcem jest stos wielokrotnego użytku.

Wzorzec: stos wielokrotnego użytku

Stos wielokrotnego użytku to projekt kodu źródłowego infrastruktury, który jest używany do tworzenia wielu instancji stosu (rysunek 6-5).



Rysunek 6-5 Wiele instancji stosu utworzonych przy użyciu jednego projektu stosu wielokrotnego użytku

Motywacja

Stos wielokrotnego użytku jest stosowany w celu utrzymywania wielu spójnych instancji infrastruktury. Po dokonaniu zmian w kodzie stosu można zastosować je i przetestować w jednej instancji, a następnie wykorzystać tę samą wersję kodu do utworzenia lub aktualizacji wielu dodatkowych instancji. Celem jest wyposażanie nowych instancji stosu minimalnym nakładem pracy, być może nawet automatycznie.

Weźmy jako przykład zespół ShopSpinner, który wyodrębnił wspólny kod z różnych projektów stosu, z których każdy używał serwera aplikacji. Członkowie zespołu umieścili ten wspólny kod w module wykorzystywanym przez wszystkie projekty stosu. Później zdali sobie sprawę, że projekty stosu dla aplikacji ich klientów nadal wyglądały bardzo podobnie. Oprócz wykorzystywania modułu do tworzenia serwera aplikacji, każdy stos zawierał kod do tworzenia baz danych oraz dedykowanych usług rejestrowania i raportowania dla każdego klienta.

Dokonywanie zmian w tym kodzie i ich testowanie dla różnych klientów stawało się kłopotliwe, a zespół ShopSpinner co miesiąc dopisywał nowych klientów. Dlatego postanowiono utworzyć jeden projekt stosu, definiujący stos aplikacji klienta. Projekt ten nadal wykorzystywał współdzielony moduł serwera aplikacji Java, podobnie jak to robi kilka innych aplikacji (Jira i GoCD). Jednak projekt zawierał także kod do konfigurowania pozostałej części infrastruktury dla poszczególnych klientów.

Teraz, gdy pojawia się nowy klient i trzeba utworzyć nową instancję, zespół wykorzystuje wspólny projekt stosu klienta. Gdy trzeba coś naprawić lub ulepszyć w kodzie bazowym projektu, zmiana jest stosowana do instancji testowych, aby upewnić się, czy jest poprawna, a następnie wdrażana po kolei w instancjach klientów.

Zastosowanie

Stos wielokrotnego użytku może być wykorzystywany w przypadku środowisk dostarczania lub wielu środowisk produkcyjnych. Wzorzec jest przydatny tam, gdzie nie ma zbytniego zróżnicowania środowisk. Jest za to mniej użyteczny, gdy środowiska wymagają rozległego dostosowania.

Konsekwencje

Możliwość udostępniania i aktualizacji wielu stosów za pomocą jednego projektu zwiększa skalowalność, niezawodność i wydajność. Pozwala zmniejszyć nakład pracy na zarządzanie wieloma instancjami, ograniczyć ryzyko awarii podczas dokonywania zmian oraz przyspieszyć wdrażanie zmian w większej liczbie systemów.

Zazwyczaj poszczególne instancje wymagają odmiennego skonfigurowania niektórych aspektów stosu, jak choćby nazw. Poświęcę temu osobny rozdział (rozdział 7).

Kod projektu stosu musi zostać dokładnie przetestowany przed zastosowaniem zmian do infrastruktury krytycznej dla działania firmy. Będę zajmował się tym w kilku rozdziałach, szczególnie w rozdziałach 8 i 9.

Implementacja

Stos wielokrotnego użytku tworzymy jako projekt stosu infrastruktury, a następnie uruchamiamy narzędzie do zarządzania stosem za każdym razem, gdy chcemy utworzyć lub zaktualizować instancję tego stosu. Wskazanie konkretnej instancji do utworzenia lub aktualizacji odbywa się za pomocą składni polecenia narzędzia stosu. Na przykład w przypadku Terraform trzeba określić inny obszar roboczy lub plik stanu dla każdej instancji. W przypadku CloudFormation trzeba przekazać unikatowy identyfikator stosu dla każdej instancji.

Poniższy przykład przedstawia fikcyjne polecenie o nazwie `stack`, które udostępnia dwie instancje stosu na podstawie jednego projektu. Polecenie przyjmuje argument `env`, który identyfikuje unikatową instancję:

```
> stack up env=test --source mystack/src
SUCCESS: stack 'test' created
> stack up env=staging --source mystack/src
SUCCESS: stack 'staging' created
```

Z reguły różnice między instancjami stosu powinny być definiowane przy użyciu prostych parametrów – łańcuchów znaków, liczb, a w niektórych przypadkach list. Ponadto infrastruktura utworzona za pomocą stosu wielokrotnego użytku nie powinna się różnić zbytnio między instancjami.

Powiązane wzorce

Stos wielokrotnego użytku jest ulepszoną wersją antywzorca środowiska kopiuj-wklej (patrz „Antywzorec: środowiska kopiuj-wklej” na stronie 63), gdyż ułatwia zachowanie spójności wielu instancji.

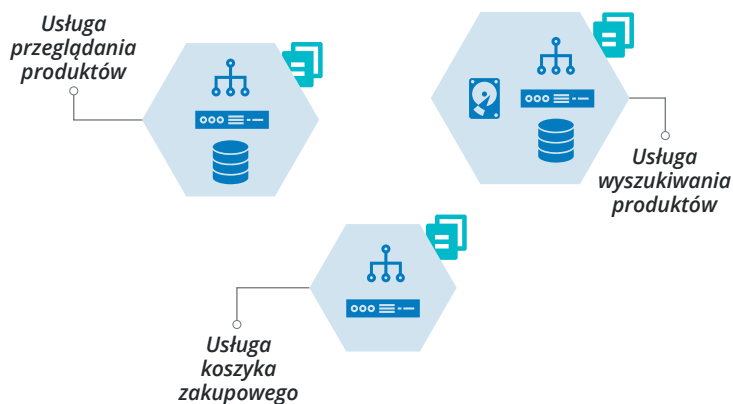
Wzorec stosu opakowującego (patrz „Wzorec: stos opakowujący” na stronie 84) definiuje stos wielokrotnego użytku przy użyciu składników stosu, ale wykorzystuje różne projekty stosu do ustawiania wartości parametrów poszczególnych instancji.

Tworzenie środowisk z wieloma stosami

Wzorec stosu wielokrotnego użytku opisuje podejście do implementacji wielu środowisk. W rozdziale 5 opisałem różne sposoby konstruowania infrastruktury systemu w wielu stosach (patrz „Wzorce i antywzorce konstruowania stosów” na stronie 52). Jest kilka sposobów implementacji stosów, które łączą te dwa wymiary środowisk i struktury systemu.

Prostym przypadkiem jest implementacja całego systemu jako pojedynczego stosu. Udostępniając instancję takiego stosu otrzymujemy kompletne środowisko. Przedstawiłem to na diagramie wzorca stosu wielokrotnego użytku (rysunek 6-5).

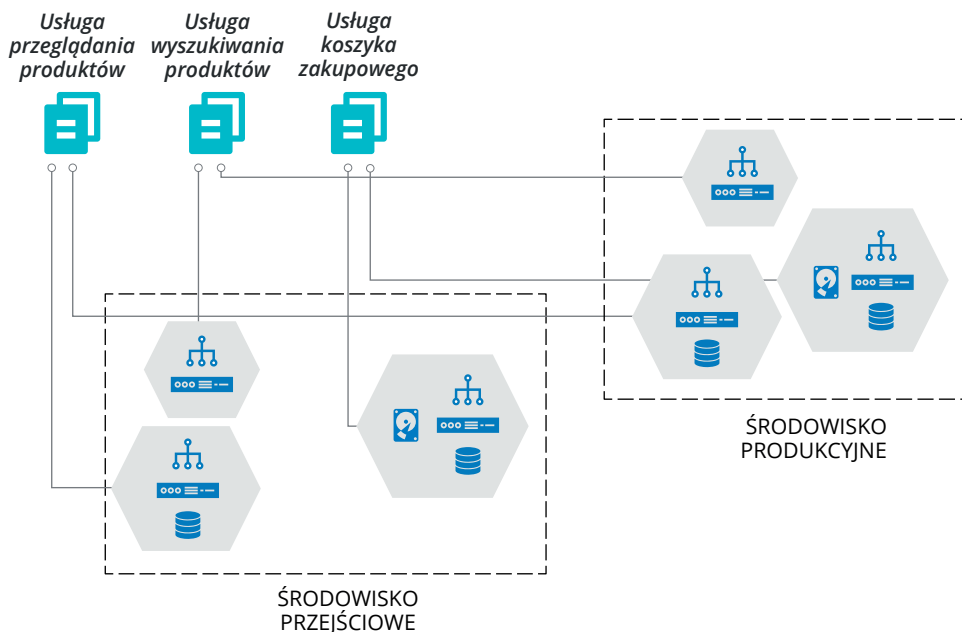
Jednak większe systemy należy rozбивać na wiele stosów. Na przykład, stosując wzorec stosu usług („Wzorec: stos usług” na stronie 56) otrzymujemy oddzielny stos dla każdej usługi, co zostało pokazane na rysunku 6-6.



Rysunek 6-6 Przykład użycia oddzielnego stosu infrastruktury dla każdej usługi

Aby utworzyć wiele stosów, trzeba udostępnić instancję stosu każdej usługi dla każdego środowiska, jak na rysunku 6-7.

Projekty z ponownym wykorzystaniem stosów



Rysunek 6-7 Używanie wielu stosów do budowy poszczególnych środowisk

Do zbudowania pełnego środowiska z wieloma stosami trzeba użyć następujących poleceń:

```
> stack up env=staging --source product_browse_stack/src
SUCCESS: stack 'product_browse-staging' created
```

```
> stack up env=staging --source product_search_stack/src  
SUCCESS: stack 'product_search-staging' created  
> stack up env=staging --source shopping_basket_stack/src  
SUCCESS: stack 'shopping_basket-staging' created
```

W rozdziale 15 opiszę strategie dzielenia systemów na wiele stosów, a w rozdziale 17 omówię sposoby integracji infrastruktury w wielu stosach.

Podsumowanie

Stosy wielokrotnego użytku powinny być podstawowym wzorcem dla większości zespołów, które muszą zarządzać dużymi infrastrukturami. Stos jest wygodną jednostką do testowania i dostarczania zmian. Daje pewność, że każda instancja środowiska jest zdefiniowana i zbudowana z zachowaniem spójności. W odróżnieniu od modułów, stos jest bardziej wszechstronny jako jednostka zmian i zwiększa możliwości ich łatwego dostarczania zarówno szybko, jak i często.

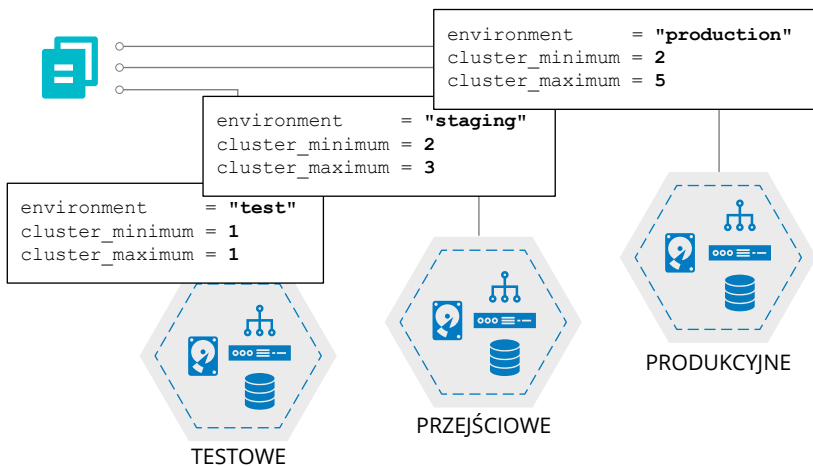
Konfigurowanie instancji stosu

Używanie jednego projektu kodu stosu ułatwia utrzymanie wielu spójnych instancji infrastruktury, jak to zostało opisane w podrozdziale „Wzorec: stos wielokrotnego użytku” na stronie 65. Często jednak zachodzi konieczność indywidualnego dostosowywania różnych instancji stosu. Na przykład klaster może liczyć mniej serwerów w środowisku programistycznym, niż produkcyjnym.

Oto przykład kodu stosu, który definiuje kontener obsługujący klaster z konfigurowaną minimalną i maksymalną liczbą serwerów:

```
container_cluster: web_cluster-$(environment)
  min_size: ${cluster_minimum}
  max_size: ${cluster_maximum}
```

Powyższy kod otrzymuje przez parametry różne wartości dla każdego środowiska, jak to jest pokazane na rysunku 7-1.



Rysunek 7-1 Używanie tego samego kodu z różnymi wartościami parametrów dla poszczególnych środowisk

Narzędzia stosu, takie jak Terraform i CloudFormation, obsługują wiele metod ustawiania wartości parametrów konfiguracyjnych. Do typowych należy przekazywanie wartości w wierszu polecenia, wczytywanie ich z pliku oraz pobieranie ich przez kod infrastruktury z magazynu klucz-wartość.

Zespoły zarządzające infrastrukturą muszą same decydować, jak używać tych funkcji do zarządzania wartościami konfiguracyjnymi i przekazywania ich do narzędzia stosu. Najważniejsze, aby definiowanie wartości i stosowanie ich w każdym środowisku odbywało się w sposób spójny.

Zasada projektowania: stosuj proste parametry

Głównym powodem definiowania infrastruktury jako kodu jest zapewnienie spójności konfigurowanych systemów, jak to zostało opisane w podrozdziale „Zasada: minimalizuj różnicowanie” na stronie 16. Konfigurowalny kod stosu stwarza ryzyko niespójności. Im więcej możliwości konfigurowania oferuje projekt stosu, tym trudniej jest zrozumieć zachowanie jego różnych instancji i zapewnić efektywne testowanie kodu oraz regularne i niezawodne dostarczanie zmian do wszystkich instancji.

Dlatego najlepiej jest stosować proste parametry stosu i używać ich w prosty sposób:

- Preferuj proste typy parametrów, takie jak łańcuchy, liczby oraz ewentualnie listy i mapy klucz-wartość. Unikaj przekazywania bardziej złożonych struktur danych.
- Minimalizuj liczbę parametrów, które można ustawić dla stosu. Unikaj definiowania parametrów, które „mogą się przydać”. Dodawaj parametry tylko wtedy, gdy już są potrzebne. Możesz zawsze dodać parametr później, gdy coś się zmieni.
- Unikaj używania parametrów jako instrukcji warunkowych, gdyż powodują znaczące różnice w wynikowej infrastrukturze. Na przykład parametr logiczny (tak/nie), wskazujący, czy w stosie ma być udostępniana pewna usługa, zwiększa złożoność.

Gdy stosowanie tej zasady staje się trudne, oznacza to prawdopodobnie, że należy przebudować kod stosu, dzieląc go na wiele projektów stosu.

Używanie parametrów stosu do tworzenia unikatowych identyfikatorów

Jeśli tworzymy wiele instancji stosu na podstawie jednego projektu (zgodnie z wzorcem stosu wielokrotnego użytku, opisanym w podrozdziale „Wzorec: stos wielokrotnego użytku” na stronie 65), mogą wystąpić błędy w zasobach infrastruktury wymagających

unikatowych identyfikatorów. Aby zrozumieć, co mam na myśli, popatrzmy na następujący pseudokod, definiujący serwer aplikacji:

```
server:
  id: appserver
  subnet_id: appserver-subnet
```

Fikcyjna platforma chmurowa wymaga, aby wartość `id` była unikatowa, kiedy więc uruchamiam polecenie `stack` w celu utworzenia drugiego stosu, pojawia się błąd:

```
> stack up environment=test --source mystack/src
SUCCESS: stack 'test' created
> stack up environment=staging --source mystack/src
FAILURE: server 'appserver' already exists in another stack
```

Aby uniknąć takich konfliktów, mogę zastosować w moim kodzie stosu parametry. Nowa wersja kodu poniżej zawiera dodany parametr o nazwie `environment`, którego używam do przypisania unikatowego identyfikatora serwera. Ponadto w każdym środowisku dodaję serwer do innej podsieci:

```
server:
  id: appserver-${environment}  subnet_id: appserver-subnet-${environment}"
```

Teraz mogę wykonać moje fikcyjne polecenie `stack` w celu utworzenia wielu instancji stosu i błąd nie wystąpi.

Przykładowe parametry stosu

Posłużę się przykładowym stosem, żeby porównać i skonfrontować różne wzorce konfiguracji stosu. Przykładem tym jest projekt szablonu stosu, który definiuje klaster kontenerów składający się z dynamicznej puli węzłów hostów oraz pewnych konstrukcji sieciowych. Struktura projektu jest pokazana w przykładzie 7-1.

Przykład 7-1 *Struktura przykładowego projektu szablonu stosu definiującego klaster kontenerów*

```
|— src/
|   |— cluster.infra
|   |— networking.infra
|— test/
```

Stos klastra wykorzystuje parametry wymienione w przykładzie 7-2 dla trzech różnych instancji stosu. `environment` jest unikatowym identyfikatorem każdego środowiska, którego można używać do nadawania nazw i tworzenia unikatowych identyfikatorów. `cluster_minimum` i `cluster_maximum` definiują przedział wielkości klastra kontenerów. Kod infrastruktury w pliku `cluster.infra` definiuje klaster na platformie chmurowej, która

skaluje liczbę węzłów odpowiednio do obciążenia. Każde z trzech środowisk, *test*, *staging* i *production*, wykorzystuje inny zbiór wartości.

Przykład 7-2 *Przykładowe wartości parametrów użyte do opisu wzorca*

instancja stosu	środowisko	cluster_minimum	cluster_maximum
cluster_test	test	1	1
cluster_staging	staging	2	3
cluster_production	production	2	6

Wzorce konfigurowania stosów

Zobaczyliśmy, dlaczego należy parametryzować stosy i mieliśmy próbkę sposobu implementacji parametrów przez narzędzia. Teraz opiszę niektóre wzorce i antywzorce zarządzania parametrami i przekazywania ich do narzędzi:

Ręczne parametry stosu

Uruchamianie narzędzia stosu i wpisywanie wartości parametrów w wierszu polecenia.

Parametry skryptowe

Kodowanie wartości parametrów dla każdej instancji w skrypcie uruchamiającym narzędzie stosu.

Pliki konfiguracyjne stosów

Deklarowanie wartości parametrów dla każdej instancji w plikach konfiguracyjnych trzymanyh w projekcie kodu stosu.

Stos opakowujący

Tworzenie oddzielnego projektu stosu infrastruktury dla każdej instancji i importowanie współdzielonego modułu z kodem stosu.

Parametry stosu potokowego

Definiowanie wartości parametrów w konfiguracji etapu potoku dla każdej instancji.

Rejestr parametrów stosu

Wartości parametrów są przechowywane w centralnej lokalizacji.

Antywzorec: ręczne parametry stosu

Najbardziej naturalną metodą podawania wartości dla instancji stosu jest ich ręczne wpisywanie w wierszu polecenia, jak w przykładzie 7-3.

Przykład 7-3 *Przykład ręcznego wpisywania parametrów w wierszu polecenia*

```
> stack up environment=production --source mystck/src  
FAILURE: No such directory 'mystck/src'
```

```
> stack up environment=production --source mystack/src
SUCCESS: new stack 'production' created
> stack destroy environment=production --source mystack/src
SUCCESS: stack 'production' destroyed
> stack up environment=production --source mystack/src
SUCCESS: existing stack 'production' modified
```

Motywacja

Wpisywanie wartości w wierszu polecenia jest trywialnie łatwe, co bywa pomocne podczas nauki korzystania z narzędzia. Wygodnie jest też wpisywać parametry w wierszu polecenia w celu eksperymentowania.

Konsekwencje

Podczas wpisywania wartości w wierszu polecenia łatwo jest popełnić błąd. Trudno też czasem pamiętać, co należy wpisać. W przypadku infrastruktury, na której nam zależy, raczej nie chcemy ryzykować przypadkowego popsucia czegoś ważnego na skutek błędnego wpisania polecenia mającego na celu dokonanie poprawki lub ulepszenia. Gdy wiele osób pracuje nad stosem infrastruktury, jak to ma miejsce w zespole, nie można oczekiwać od wszystkich pamiętania prawidłowych wartości dla każdej instancji.

Ręczne parametry stosu nie nadają się do automatycznego stosowania kodu infrastruktury do środowisk, na przykład przy użyciu CI czy CD.

Implementacja

Wartości parametrów z naszego przykładu (przykład 7-2) należy przekazywać w wierszu polecenia zgodnie ze składnią oczekiwaną przez konkretne narzędzie. Dla mojego fikcyjnego narzędzia stack wygląda to następująco:

```
stack up \
  environment=test \
  cluster_minimum=1 \
  cluster_maximum=1 \
  ssl_cert_passphrase="correct horse battery staple"
```

Każdy, kto uruchamia polecenie, musi znać wpisywane hasła i klucze, które trzeba przekazać w wierszu polecenia dla danego środowiska. Zespół powinien używać narzędzia do zarządzania hasłami w celu ich bezpiecznego przechowywania i udostępniania swoim członkom, a także ich rotowania w momencie odchodzenia osób z zespołu¹.

¹ Przykładowe narzędzia, których zespoły mogą używać do bezpiecznego udostępniania haseł, to: GPG (<https://gnupg.org>), KeePass (<https://keepass.info>), 1Password (<https://1password.com/teams>), Keeper (<https://keepersecurity.com>) i LastPass (<https://www.lastpass.com/>).

Powiązane wzorce

Wzorzec parametrów skryptowych (patrz „Wzorzec: parametry skryptowe” na stronie 78) wykorzystuje polecenie, które należałoby wpisać ręcznie, ale umieszcza je w skrypcie. Wzorzec parametrów stosu potokowego (patrz „Wzorzec: parametry stosu potokowego” na stronie 87) wygląda podobnie, tylko polecenie jest zapisywane w konfiguracji potoku, zamiast w skrypcie.

Wzorzec: zmienne środowiskowe stosu

Wzorzec zmiennych środowiskowych stosu polega na ustawianiu wartości parametrów jako zmiennych środowiskowych do wykorzystania przez narzędzie stosu. Wzorzec ten jest często łączony z innym wzorcem do ustawiania zmiennych środowiskowych.

Zmienne środowiskowe są ustawiane wcześniej, jak to widać w przykładzie 7-4 (więcej informacji w podpunkcie „Implementacja”).

Przykład 7-4 Ustawianie zmiennych środowiskowych

```
export STACK_ENVIRONMENT=test
export STACK_CLUSTER_MINIMUM=1
export STACK_CLUSTER_MAXIMUM=1
export STACK_SSL_CERT_PASSPHRASE="correct horse battery staple"
```

Istnieją różne wersje implementacji, ale najbardziej podstawowa polega na bezpośrednim odwoływaniu się do nich przez kod stosu, jak w przykładzie 7-5.

Przykład 7-5 Kod stosu wykorzystujący zmienne środowiskowe

```
container_cluster: web_cluster-{{ENV("STACK_ENVIRONMENT")}}
min_size: {{ENV("STACK_CLUSTER_MINIMUM")}}
max_size: {{ENV("STACK_CLUSTER_MAXIMUM")}}
```

Motywacja

Zmienne środowiskowe są obsługiwane przez większość platform i narzędzi, więc łatwo z nich korzystać.

Zastosowanie

Jeśli już używamy zmiennych środowiskowych w naszym systemie i mamy odpowiednie mechanizmy do zarządzania nimi, to wykorzystanie ich do parametrów stosu może być wygodne.

Konsekwencje

Do ustawiania wartości zmiennych potrzebny jest dodatkowy wzorzec z tego rozdziału. Powoduje to zwiększenie liczby „ruchomych części” i utrudnia śledzenie wartości konfiguracyjnych dla konkretnej instancji stosu. Ponadto zmiana wartości wymaga więcej pracy.

Stosowanie zmiennych środowiskowych bezpośrednio w kodzie stosu, jak w przykładzie 7-5, prawdopodobnie powoduje zbyt silne sprzężenie kodu stosu ze środowiskiem wykonawczym.

Ustawianie wpisów tajnych za pomocą zmiennych środowiskowych może ujawnić je innym procesom działającym w tym samym systemie.

Implementacja

Jak już zaznaczyłem, trzeba ustawić zmienne środowiskowe, które mają być używane, co oznacza dodanie kolejnego wzorca z tego rozdziału. Na przykład, jeśli zmienne środowiskowe mają być ustawiane przez ludzi w ich lokalnym środowisku, aby móc zastosować kod stosu, w praktyce oznacza to posługiwanie się antywzorcem ręcznych parametrów stosu (patrz „Antywzorec: ręczne parametry stosu” na stronie 74). Można ustawiać je w skrypcie uruchamiającym narzędzie stosu (wzorec parametrów skryptowych opisany w podrozdziale „Wzorec: parametry skryptowe” na stronie 78) albo wykorzystać do tego narzędzie potoku (patrz „Wzorec: parametry stosu potokowego” na stronie 87).

Innym sposobem jest umieszczenie wartości w skrypcie importowanym przez ludzi lub instancje do ich lokalnego środowiska. Jest to odmiana wzorca plików konfiguracyjnych stosu (patrz „Wzorec: pliki konfiguracyjne stosu” na stronie 81). Skrypt ustawiający zmienne wyglądałby identycznie jak w przykładzie 7-4, a dowolne polecenie uruchamiające narzędzie stosu importowałoby go do środowiska:

```
source ./environments/staging.env
stack up --source ./src
```

Alternatywą jest wbudowanie wartości środowiskowych do instancji obliczeniowej uruchamiającej narzędzie stosu. Na przykład, w przypadku zapewnienia oddzielnego węzła agenta CD do wywoływania narzędzia stosu w celu utworzenia i aktualizacji stosów w poszczególnych środowiskach, kod budujący węzeł mógłby ustawiać odpowiednie wartości jako zmienne środowiskowe. Takie zmienne byłyby dostępne dla każdego polecenia uruchamianego w węźle, łącznie z narzędziem stosu.

Ale by to osiągnąć, konieczne jest przekazanie wartości do kodu budującego węzły agentów. W tym celu należy więc wybrać kolejny wzorec z tego rozdziału.

Inną stroną implementacji tego wzorca jest sposób, w jaki narzędzie stosu ma pobierać wartości środowiskowe. W przykładzie 7-5 zobaczyliśmy, jak kod stosu może bezpośrednio odczytywać zmienne środowiskowe.

Zamiast tego można użyć skryptu orkiestracji stosu (patrz „Używanie skryptów do opakowywania narzędzi infrastruktury” na stronie 327) do odczytania zmiennych środowiskowych i przekazania ich do narzędzia stosu w wierszu polecenia. Kod w skrypcie orkiestracji wyglądałby następująco:

```
stack up \
  environment=${STACK_ENVIRONMENT} \
  cluster_minimum=${STACK_CLUSTER_MINIMUM} \
```

```
cluster_maximum=${STACK_CLUSTER_MAXIMUM} \  
ssl_cert_passphrase="${STACK_SSL_CERT_PASSPHRASE}"
```

Takie podejście oddziela kod stosu od środowiska, w którym jest wykonywany.

Powiązane wzorce

Wzorec ten można łączyć z każdym z pozostałych wzorców przedstawionych w tym rozdziale w celu ustawienia wartości środowiskowych.

Wzorec: parametry skryptowe

Wzorec parametrów skryptowych oznacza zakodowanie wartości parametrów w skrypcie uruchamiającym narzędzie stosu. Można napisać oddzielny skrypt dla każdego środowiska albo jeden skrypt zawierający wartości dla wszystkich środowisk:

```
if ${ENV} == "test"  
    stack up cluster_maximum=1 env="test"  
elif ${ENV} == "staging"  
    stack up cluster_maximum=3 env="staging"  
elif ${ENV} == "production"  
    stack up cluster_maximum=5 env="production"  
end
```

Motywacja

Skrypty są prostym sposobem przechwytywania wartości dla poszczególnych instancji i pozwalają uniknąć problemów towarzyszących antywzorcowi ręcznych parametrów stosu (patrz „Antywzorec: ręczne parametry stosu” na stronie 74). Dają pewność spójnego stosowania wartości w każdym środowisku. Poddając skrypt kontroli wersji można uzyskać śledzenie dowolnych zmian wartości konfiguracyjnych.

Zastosowanie

Skrypt wyposażania stosu jest wygodnym sposobem ustawiania parametrów w przypadku ustalonego zbioru środowisk, nie zmieniających się zbyt często. Nie wymaga on dodatkowych ruchomych części w postaci któregoś z pozostałych wzorców w tym rozdziale.

Ponieważ trudno jest kodować w skryptach wpisy tajne, wzorec ten nie nadaje się do takich wpisów. Nie oznacza to, że nie można używać go w przypadku wpisów tajnych, tylko że do zajmowania się nimi potrzebna jest implementacja oddzielnego wzorca (sugestie można znaleźć w podrozdziale „Obsługa wpisów tajnych jako parametrów” na stronie 96).

Konsekwencje

Polecenia używane do uruchamiania narzędzia stosu często stają się z czasem skomplikowane. Przygotowywanie skryptów potrafi zmienić się w zagmatwanego potwora.

W podrozdziale „Używanie skryptów do opakowywania narzędzi infrastruktury” na stronie 327 opisuję, jak używać takich skryptów oraz jak wyglądają pułapki i zalecenia dotyczące zachowania ich w stanie łatwym do utrzymania. Należy pamiętać o testowaniu przygotowywania skryptów, ponieważ mogą być źródłem problemów z systemami, które mają wyposażać.

Implementacja

Istnieją dwie popularne implementacje tego wzorca. Jedna to pojedynczy skrypt, który przyjmuje środowisko jako argument wiersza polecenia i ma zakodowane na stałe wartości parametrów dla poszczególnych środowisk. Prostą wersję tego podejścia widać w przykładzie 7-6.

Przykład 7-6 Przykład skryptu z parametrami dla wielu środowisk

```
#!/bin/sh

case $1 in test)
    CLUSTER_MINIMUM=1
    CLUSTER_MAXIMUM=1
    ;; staging)
    CLUSTER_MINIMUM=2
    CLUSTER_MAXIMUM=3
    ;; production)
    CLUSTER_MINIMUM=2
    CLUSTER_MAXIMUM=6
    ;;
*)
    echo "Unknown environment $1"
    exit 1
    ;;
esac

stack up \
    environment=$1 \
    cluster_minimum=${CLUSTER_MINIMUM} \
    cluster_maximum=${CLUSTER_MAXIMUM}
```

Druga implementacja to oddzielne skrypty dla poszczególnych instancji stosu, co jest pokazane w przykładzie 7-7.

Przykład 7-7 Przykład struktury projektu ze skryptami dla poszczególnych środowisk

```
our-infra-stack/
├─ bin/
│   ├── test.sh
│   ├── staging.sh
│   └─ production.sh
```

```
└─ src/  
└─ test/
```

Wszystkie skrypty są identyczne, tylko mają zakodowane w środku inne wartości parametrów. Sam skrypt jest mniejszy, ponieważ nie zawiera logiki potrzebnej do wyboru odpowiednich wartości parametrów. Z drugiej strony, utrzymanie takiego skryptu wymaga więcej pracy. W przypadku konieczności zmiany polecenia trzeba zrobić to we wszystkich skryptach. Posiadanie oddzielnego skryptu dla każdego środowiska może również stwarzać pokusę dostosowywania poszczególnych środowisk, a to prowadzi do braku spójności.

Skrypt wyposażający powinien podlegać kontroli źródeł. Umieszczenie go w tym samym projekcie, co wyposażany przez niego stos, zapewnia jego synchronizację z kodem stosu. Na przykład, przy dodawaniu nowego parametru trzeba dodać go do kodu źródłowego infrastruktury oraz do wyposażającego skryptu. Dzięki kontroli źródeł zawsze będzie wiadomo, którą wersję skryptu należy uruchamiać dla danej wersji kodu stosu.

Podrozdział „Używanie skryptów do opakowywania narzędzi infrastruktury” na stronie 327 zawiera znacznie bardziej szczegółowe omówienie używania skryptów do uruchamiania narzędzi stosu.

Jak wspomniałem wcześniej, nie należy kodować na stałe wpisów tajnych w skryptach, czyli wymagają one oddzielnego wzorca. Do jego obsługi można użyć skryptu. W przykładzie 7-8 narzędzie wiersza polecenia pobiera wpis tajny od menedżera wpisów tajnych, zgodnie z wzorcem rejestru parametrów (patrz „Wzorzec: rejestr parametrów stosu” na stronie 90).

Przykład 7-8 *Pobieranie klucza od menedżera wpisów tajnych za pomocą skryptu*

```
...  
# (Set environment specific values as in other examples)  
...  
  
SSL_CERT_PASSPHRASE=$(some-tool get-secret id="/ssl_cert_passphrase/${ENV}")  
  
stack up \  
  environment=${ENV} \  
  cluster_minimum=${CLUSTER_MINIMUM} \  
  cluster_maximum=${CLUSTER_MAXIMUM} \  
  ssl_cert_passphrase="${SSL_CERT_PASSPHRASE}"
```

Polecenie `some-tool` łączy się z menedżerem wpisów tajnych i pobiera wpis tajny dla odpowiedniego środowiska na podstawie identyfikatora `/ssl_cert_passphrase/${ENV}`. Ten skrypt zakłada, że sesja ma uprawnienia do używania menedżera wpisów tajnych. Programista infrastruktury może użyć tego narzędzia do rozpoczęcia sesji przed uruchomieniem skryptu. Alternatywnie instancja obliczeniowa, która uruchamia skrypt, może mieć uprawnienia do pobierania wpisów tajnych przy użyciu autoryzacji bez wpisów tajnych (jak to opisuję w podrozdziale „Autoryzacja bez wpisu tajnego” na stronie 97).

Powiązane wzorce

Skrypty wyposażające uruchamiają za nas narzędzie wiersza polecenia, są więc sposobem na pominięcie antywzorca ręcznych parametrów stosu (patrz „Antywzorzec: ręczne parametry stosu” na stronie 74. Wzorzec plików konfiguracyjnych stosu przenosi wartości parametrów ze skryptu do oddzielnych plików.

Wzorzec: pliki konfiguracyjne stosu

We wzorcu plików konfiguracyjnych stosu wartości parametrów dla poszczególnych instancji są trzymane w oddzielnych plikach, zarządzanych razem z kodem stosu za pomocą systemu kontroli wersji. Ilustruje to przykład 7-9.

Przykład 7-9 *Przykład projektu z plikiem parametrów dla każdego środowiska*

```
├─ src/
│   ├── cluster.infra
│   ├── host_servers.infra
│   └── networking.infra
├─ environments/
│   ├── test.properties
│   ├── staging.properties
│   └── production.properties
└─ test/
```

Motywacja

Tworzenie plików konfiguracyjnych dla poszczególnych instancji stosu jest proste i łatwe do zrozumienia. Ponieważ każdy plik jest zapisywany w repozytorium kodu źródłowego, łatwo jest:

- zobaczyć, jakie wartości są używane dla konkretnego środowiska („Jaki jest maksymalny rozmiar klastra dla produkcji?”)
- śledzić historię w celu debugowania („Kiedy nastąpiła zmiana maksymalnego rozmiaru klastra?”)
- dokonywać inspekcji zmian („Kto zmienił maksymalny rozmiar klastra?”)

Wzorzec plików konfiguracyjnych stosu wymusza oddzielenie konfiguracji od kodu stosu.

Zastosowanie

Pliki konfiguracyjne stosu są przydatne w sytuacji, gdy liczba środowisk nie zmienia się często. W przypadku nowej instancji stosu wzorzec wymaga dodania kolejnego pliku do projektu. Wymaga również (i pomaga to zapewnić) spójnej logiki tworzenia i aktualizowania odmiennych instancji, ponieważ same pliki konfiguracyjne nie zawierają logiki.

Konsekwencje

Gdy jest potrzebna nowa instancja stosu, należy dodać kolejny plik konfiguracyjny do projektu stosu. Taka metoda blokuje możliwość automatycznego tworzenia nowych środowisk w locie. W podrozdziale „Wzorzec: efemeryczny stos testowy” na stronie 137 opisuję podejście do zarządzania środowiskami testowymi oparte na automatycznym tworzeniu środowisk. Można obejść ten problem, tworząc na żądanie plik konfiguracyjny dla środowiska efemerycznego.

Pliki z parametrami mogą utrudniać zmiany konfiguracji środowisk podrzędnych w przypadku potoków dostarczania zmian opisanych w podrozdziale „Potoki dostarczania infrastruktury” na stronie 113. Każda zmiana w kodzie projektu stosu musi przejść przez wszystkie etapy potoku, zanim zostanie zastosowana w środowisku produkcyjnym. Wykonanie tego może zająć trochę czasu i nie dodaje żadnej wartości, jeśli zmiana konfiguracji ma zastosowanie tylko do produkcji.

Definiowanie wartości parametrów może być powodem znacznej złożoności skryptów wyposażających. Powiem o tym więcej w podrozdziale „Używanie skryptów do opakowywania narzędzi infrastruktury” na stronie 327, ale tytułem zapowiedzi zauważmy, że zespoły chcą często definiować wartości domyślne dla projektów stosu i dla środowisk, a w związku z tym potrzebują logiki łączenia tych wartości z wartościami specyficznymi dla pożądaných instancji stosu w różnych środowiskach. Modele dziedziczenia dla wartości parametrów bywają bardzo zagmatwane i mylące.

Pliki konfiguracyjne przechowywane w kontroli źródła nie powinny zawierać wpisów tajnych. W przypadku tych wpisów należy wybrać dodatkowy wzorzec z tego rozdziału do ich obsługi albo zaimplementować oddzielny plik konfiguracyjny wpisów tajnych, poza kontrolą źródła.

Implementacja

Wartości parametrów stosu są definiowane w oddzielnym pliku dla każdego środowiska, jak to zostało pokazane wcześniej w przykładzie 7-9.

Zawartość pliku konfiguracyjnego może wyglądać następująco:

```
env = staging
cluster_minimum = 2
cluster_maximum = 3
```

W momencie uruchamiania polecenia stack przekazujemy ścieżkę do odpowiedniego pliku parametrów:

```
stack up --source ./src --config ./environments/staging.properties
```

Jeśli system składa się z wielu stosów, zarządzanie konfiguracją wielu środowisk może stać się skomplikowane. Występują dwa popularne sposoby rozmieszczania plików parametrów w takiej sytuacji. Jeden z nich polega na umieszczeniu pliku konfiguracyjnego każdego środowiska razem z kodem stosu:

```

├─ cluster_stack/
│  └─ src/
│     ├── cluster.infra
│     ├── host_servers.infra
│     └─ networking.infra
│  └─ environments/
│     ├── test.properties
│     ├── staging.properties
│     └─ production.properties
└─ appserver_stack/
   └─ src/
      ├── server.infra
      └─ networking.infra
   └─ environments/
      ├── test.properties
      ├── staging.properties
      └─ production.properties

```

Drugi sposób to umieszczenie plików konfiguracyjnych wszystkich stosów w jednym miejscu:

```

├─ cluster_stack/
│  ├── cluster.infra
│  ├── host_servers.infra
│  └─ networking.infra
├─ appserver_stack/
│  ├── server.infra
│  └─ networking.infra
└─ environments/
   ├── test/
   │  ├── cluster.properties
   │  └─ appserver.properties
   ├── staging/
   │  ├── cluster.properties
   │  └─ appserver.properties
   └─ production/
      ├── cluster.properties
      └─ appserver.properties

```

Każde z tych podejść może na swój sposób prowadzić do bałaganu i zamieszania. Gdy trzeba zmienić wszystko w jakimś środowisku, dokonanie zmian w plikach konfiguracyjnych wielu projektów stosu jest bolesne. Gdy trzeba zmienić konfigurację jednego

stosu w wielu różnych środowiskach, w których występuje, szperanie w drzewie pełnym konfiguracji dla wielu innych stosów również nie jest zabawne.

Jeśli chcemy używać plików konfiguracyjnych do dostarczania wpisów tajnych, zamiast zastosować dla nich oddzielny wzorzec, powinniśmy zarządzać tymi plikami poza kodem projektu podpiętym do kontroli źródła.

W przypadku lokalnych środowisk programowania można wymagać od użytkowników ręcznego utworzenia pliku w określonej lokalizacji. Oto przykład przekazywania takiej lokalizacji do polecenia `stack`:

```
stack up --source ./src \
  --config ./environments/staging.properties \
  --config ../secrets/staging.properties
```

W tym przykładzie narzędzie stosu dostaje dwa argumenty `--config` i z obu wczytuje wartości parametrów. Katalog o nazwie `.secrets` znajduje się poza folderem projektu, czyli nie należy do kontroli źródła.

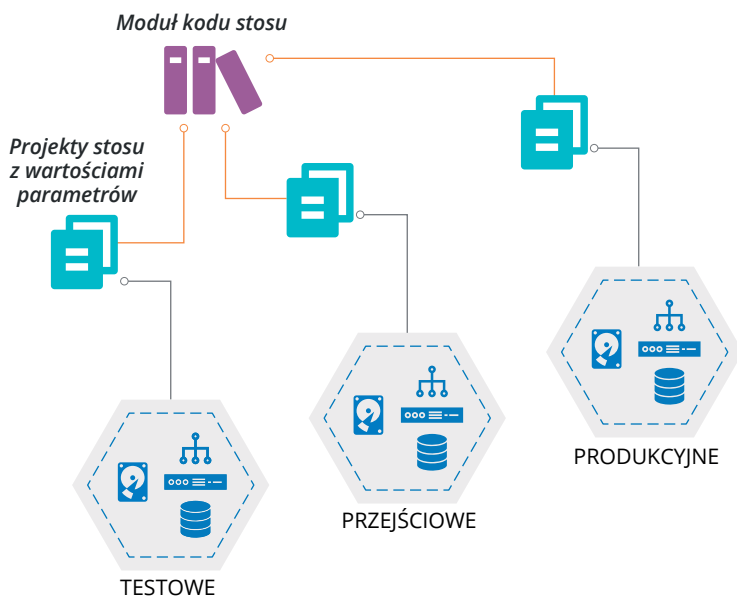
Taka metoda bywa trudniejsza w przypadku automatycznego uruchamiania narzędzia stosu z poziomu instancji obliczeniowej, takiej jak agent potoku CD. Można przygotować podobne pliki właściwości wpisów tajnych dla instancji obliczeniowych, ale grozi to ujawnieniem wpisów innym procesom uruchomionym w ramach tego samego agenta. Ponadto trzeba udostępnić wpisy tajne procesowi, który buduje instancję obliczeniową dla agenta, nadal więc pozostaje problem z ładowaniem początkowym.

Powiązane wzorce

Umieszczanie wartości konfiguracyjnych w plikach upraszcza skrypty wyposażające opisane w podrozdziale „Wzorzec: parametry skryptowe” na stronie 78. Można uniknąć niektórych ograniczeń plików konfiguracyjnych środowiska, stosując zamiast tego wzorzec rejestru parametrów stosu (patrz „Wzorzec: parametry stosu potokowego” na stronie 87). Taka metoda powoduje przeniesienie wartości parametrów poza kod projektu stosu do centralnej lokalizacji, co pozwala używać różnych przepływów pracy dla kodu i konfiguracji.

Wzorzec: stos opakowujący

Wzorzec stosu opakowującego wykorzystuje projekt stosu infrastruktury dla każdej instancji jako opakowanie importu składnika kodu stosu (patrz rozdział 16). Każdy projekt opakowania definiuje wartości parametrów dla jednej instancji stosu. Następnie importuje składnik współdzielony przez wszystkie instancje tego stosu (patrz rysunek 7-2).



Rysunek 7-2 Stos opakowujący wykorzystuje projekt stosu infrastruktury dla każdej instancji jako opakowanie importu modułu stosu

Motywacja

Stos opakowujący wykorzystuje funkcjonalność modułów narzędzia stosu lub obsługę biblioteki do wielokrotnego używania współdzielonego kodu w poszczególnych instancjach stosu. Wykorzystując takie funkcjonalności modułu narzędzia, jak tworzenie wersji, zarządzanie zależnościami i repozytorium artefaktów, można zaimplementować potok dostarczania zmian (patrz „Potoki dostarczania infrastruktury” na stronie 113). W chwili, gdy to piszę, większość narzędzi stosu infrastruktury nie oferuje formatu spakowania projektu, który nadawałby się do implementacji potoków dla kodu stosu. Trzeba więc tworzyć własne, niestandardowe procesy pakowania stosu. Można obejść ten problem, stosując stos opakowujący oraz wersjonowanie i promowanie kodu stosu jako modułu.

Stosy opakowujące pozwalają zapisać logikę udostępniania i konfigurowania stosu w tym samym języku, który jest używany do zdefiniowania infrastruktury, zamiast w oddzielnym języku, jak ma to miejsce w przypadku skryptu wyposażającego (patrz „Wzorzec: parametry skryptowe” na stronie 78).

Konsekwencje

Składniki tworzą dodatkową warstwę złożoności między stosem a kodem zawartym w składniku. Mamy teraz dwa poziomy: projekt stosu, który zawiera projekty opakowujące oraz składnik, który zawiera kod stosu.

Ponieważ każda instancja stosu ma oddzielny projekt kodu, niektórzy mogą ulegać pokusie dodawania niestandardowej logiki dla każdej instancji. Niestandardowy kod instancji powoduje, że baza kodu staje się niespójna i trudna w utrzymaniu.

Ponieważ wartości parametrów są definiowane w projektach opakowujących zarządzanych w kontroli źródła, nie można używać tego wzorca do zarządzania wpisami tajnymi. Trzeba więc dodać kolejny wzorec z tego rozdziału, aby udostępnić stosom te wpisy.

Implementacja

Każda instancja stosu ma oddzielny projekt stosu infrastruktury. Na przykład trzeba mieć oddzielny projekt Terraform dla każdego środowiska. Można zaimplementować to jako środowisko kopiuj-wklej (patrz „Antywzorec: środowiska kopiuj-wklej” na stronie 63), z każdym środowiskiem w oddzielnym repozytorium.

Alternatywą jest zrobienie z każdego projektu środowiska folderu w jednym repozytorium:

```
my_stack/  
├─ test/  
│   └─ stack.infra  
├─ staging/  
│   └─ stack.infra  
└─ production/  
    └─ stack.infra
```

Kod infrastruktury dla stosu należy zdefiniować jako moduł, zgodnie z implementacją używanego narzędzia. Można umieścić kod modułu w tym samym repozytorium, co stosy opakowujące. Takie rozwiązanie uniemożliwi jednak korzystanie z możliwości wersjonowania modułu. To znaczy, że nie będzie można używać różnych wersji kodu infrastruktury w różnych środowiskach, co jest kluczowe w przypadku stopniowego testowania kodu.

Poniższy przykład przedstawia stos opakowujący, który importuje moduł o nazwie `container_cluster_module` i określa wersję modułu oraz parametry konfiguracyjne, które mają zostać przekazane do niego:

```
module:  
  name: container_cluster_module  
  version: 1.23  
  parameters:  
    env: test  
    cluster_minimum: 1  
    cluster_maximum: 1
```

Kod stosu opakowującego dla środowisk *staging* i *production* jest podobny, z wyjątkiem wartości parametrów i być może wersji modułu, która ma być użyta.

Struktura projektu dla tego modułu mogłaby wyglądać następująco:

```

├─ container_cluster_module/
│   └─ cluster.infra
│       └─ networking.infra
└─ test/

```

W przypadku dokonania zmiany w kodzie modułu trzeba go przetestować i przesłać do repozytorium modułu. Sposób działania repozytorium zależy od konkretnego narzędzia stosu infrastruktury. Następnie można zaktualizować testową instancję stosu importując nową wersję modułu i stosując ją w środowisku testowym.

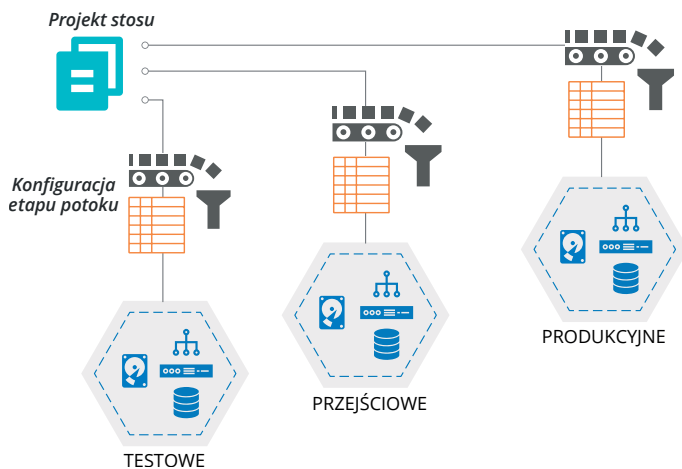
Narzędziem orkiestracji stosu, które implementuje wzorec stosu opakowującego, jest Terragrunt².

Powiązane wzorce

Stos opakowujący jest podobny do wzorca parametrów skryptowych. Główna różnica polega na tym, że wykorzystuje on język posiadanego narzędzia stosu zamiast oddzielnego języka skryptowego oraz że kod infrastruktury stanowi oddzielny składnik.

Wzorec: parametry stosu potokowego

Wzorec parametrów stosu potokowego polega na definiowaniu wartości dla każdej instancji w konfiguracji potoku dostarczania (patrz rysunek 7-3).



Rysunek 7-3 Każdy etap, który stosuje kod stosu, przekazuje odpowiednie wartości konfiguracyjne dla środowiska

O tym, jak używać potoku dostarczania zmian w celu stosowania kodu stosu infrastruktury do środowiska, piszę w podrozdziale „Potoki dostarczania infrastruktury” na

² <https://github.com/gruntwork-io/terragrunt>

stronie 113. Potok można zaimplementować za pomocą takich narzędzi jak Jenkins, GoCD lub ConcourseCI (więcej na temat tych narzędzi w podrozdziale „Usługi i oprogramowanie potoków dostarczania” na stronie 117).

Motywacja

Jeśli używamy narzędzia potoku do uruchamiania narzędzia stosu infrastruktury, zapewnia ono gotowy mechanizm przechowywania i przekazywania wartości parametrów do tego narzędzia. Zakładając, że narzędzie potoku samo jest konfigurowane przez kod, to wartości też są zdefiniowane jako kod i przechowywane w systemie kontroli wersji.

Wartości konfiguracyjne są trzymane oddzielnie od kodu infrastruktury. Można zmieniać wartości konfiguracyjne dla środowisk podrzędnych i stosować je natychmiast, bez konieczności przepuszczania nowej wersji kodu infrastruktury od początku potoku.

Zastosowanie

Zespoły, które już stosują kod infrastruktury do środowisk za pomocą potoku, mogą łatwo wykorzystać to do ustawiania parametrów stosu dla każdego środowiska. Jednak w przypadku stosów wymagających więcej niż kilku wartości parametrów definiowanie ich w konfiguracji potoku ma poważne wady, należy więc tego unikać.

Konsekwencje

Definiując zmienne instancji stosu w konfiguracji potoku wiążemy wartości konfiguracyjne z procesem dostarczania. Istnieje ryzyko, że w efekcie konfiguracja potoku stanie się skomplikowana i trudna w utrzymaniu.

Im więcej wartości konfiguracyjnych jest zdefiniowanych w potoku, tym trudniej jest uruchamiać narzędzie stosu poza potokiem. Taki potok może stać się pojedynczym punktem awarii – możemy nie być w stanie naprawić, odzyskać lub odbudować środowiska w przypadku awarii, zanim nie zostanie odzyskany potok. Ponadto zespołowi może być trudno tworzyć i testować kod stosu poza potokiem.

Ogólnie najlepiej jest, aby konfiguracja potoku do stosowania projektu stosu była możliwie najmniejsza i jak najprostsza. Większość logiki powinna znajdować się w skrypcie wywoływanym przez potok, a nie w konfiguracji potoku.



Serwery CI, potoki i wpisy tajne

Pierwszą rzeczą, której szuka większość atakujących po uzyskaniu dostępu do sieci korporacyjnej, są serwery CI i CD. Są to dobrze znane skarbnice haseł i kluczy, które można wykorzystać do wyrządzenia maksymalnych szkód użytkownikom i klientom.

Większość narzędzi CI i CD, z którymi miałem do czynienia, nie oferuje zbyt solidnego modelu zabezpieczeń. Należy zakładać, że każdy, kto ma dostęp do naszego narzędzia potoku lub może modyfikować kod wykonywany przez to narzędzie (czyli prawdopodobnie każdy programista w naszej

organizacji), ma dostęp do każdego wpisu tajnego przechowywanego przez to narzędzie.

Jest to prawdą nawet wtedy, gdy narzędzie szyfruje wpisy tajne, ponieważ może ono także odszyfrować te wpisy. Jeśli można sprawić, że narzędzie uruchomi jakieś polecenie, to zwykle można też sprawić, że odszyfruje dowolny wpis tajny, który przechowuje. Należy starannie analizować wszystkie narzędzia CI i CD, aby ocenić, na ile dobrze spełniają one wymagania bezpieczeństwa danej organizacji.

Implementacja

Parametry powinny być implementowane przy użyciu konfiguracji „jako kodu” narzędzia potoku. Przykład 7-10 przedstawia konfigurację etapu potoku przy użyciu pseudokodu.

Przykład 7-10 *Przykład konfiguracji etapu potoku*

```
stage: apply-test-stack
input_artifacts: container_cluster_stack
commands:
  unpack ${input_artifacts}
  stack up --source ./src environment=test cluster_minimum=1 cluster_maximum=1
  stack test environment=test
```

W tym przykładzie wartości są przekazywane w wierszu polecenia. Można również ustawić je jako zmienne środowiskowe wykorzystywane przez kod stosu, jak to jest pokazane w przykładzie 7-11 (patrz także „Wzorzec: zmienne środowiskowe stosu” na stronie 76).

Przykład 7-11 *Przykład konfiguracji etapu potoku przy użyciu zmiennych środowiskowych*

```
stage: apply-test-stack
input_artifacts: container_cluster_stack
environment_vars:
  STACK_ENVIRONMENT=test
  STACK_CLUSTER_MINIMUM=1
  STACK_CLUSTER_MAXIMUM=1
commands:
  unpack ${input_artifacts}
  stack up --source ./src
  stack test environment=test
```

W tym przykładzie narzędzie potoku ustawia zmienne środowiskowe przed wywołaniem `commands`.

Wiele narzędzi potoku udostępnia funkcje zarządzania wpisami tajnymi, których można używać do przekazywania wpisów do polecenia `stack`. Można w pewien sposób ustawić wartości wpisów tajnych w narzędziu potoku, a następnie odwołać się do nich w zadaniu potoku, jak to jest pokazane w przykładzie 7-12.

Przykład 7-12 Przykład etapu potoku z wpisem tajnym

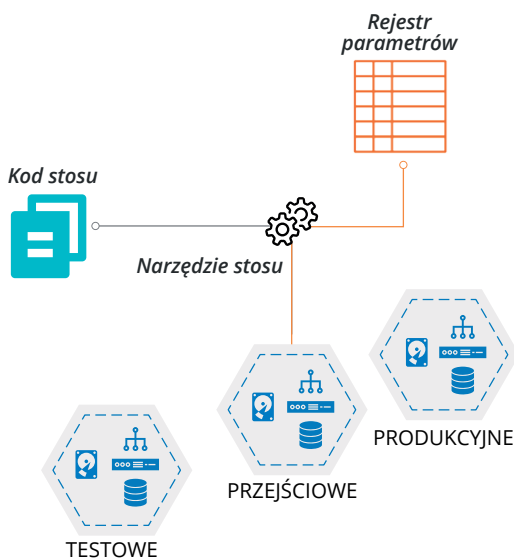
```
stage: apply-test-stack
input_artifacts: container_cluster_stack
commands:
  unpack ${input_artifacts}
  stack up --source ./src environment=test \
    cluster_minimum=1 \
    cluster_maximum=1 \
    ssl_cert_passphrase=${STACK_SSL_CERT_PASSPHRASE}
```

Powiązane wzorce

Definiowanie poleceń i parametrów w celu stosowania kodu stosu dla każdego środowiska w konfiguracji potoku jest podobne do wzorca parametrów skryptowych. Różnica dotyczy tego, gdzie znajduje się skrypt – w konfiguracji potoku czy w plikach skryptów.

Wzorec: rejestr parametrów stosu

Wzorec rejestru parametrów stosu zakłada zarządzanie wartościami parametrów dla instancji stosu w centralnej lokalizacji, a nie razem z kodem stosu. Narzędzie stosu musi pobrać odpowiednie wartości w momencie stosowania kodu stosu do danej instancji (patrz rysunek 7-4).



Rysunek 7-4 Wartości parametrów instancji stosu przechowywane w centralnym rejestrze



Rejestry konfiguracji i rejestry parametrów stosu

Używam terminu *rejestr konfiguracji* na określenie usługi przechowującej wartości konfiguracyjne, które mogą być używane do wielu celów, w tym do odnajdywania usług, integracji stosów albo monitorowania konfiguracji. Zajmę się tym dokładniej w podrozdziale „Rejestr konfiguracji” na stronie 93.

Gdy piszę konkretnie o przechowywaniu wartości konfiguracyjnych dla instancji stosu, używam terminu *rejestr parametrów stosu*. Zatem rejestr parametrów stosu jest szczególnym przypadkiem użycia rejestru konfiguracji.

Motywacja

Przechowywanie parametrów w rejestrze powoduje oddzielenie konfiguracji od implementacji. Parametry te można ustawiać, wykorzystywać i wyświetlać za pomocą różnych narzędzi, używając różnych języków i technologii. Ta elastyczność redukuje sprzężenie między różnymi częściami systemu. Można zmienić dowolne narzędzie korzystające z rejestru nie wpływając na pozostałe narzędzia, które go używają.

Ponieważ rejestry parametrów stosu są niezależne od narzędzi, mogą stanowić źródło prawdy o konfiguracji infrastruktury, a nawet systemu i pełnić rolę bazy danych zarządzania konfiguracją (Configuration Management Database – CMDB). Takie dane konfiguracyjne mogą być przydatne w kontekstach regulowanych, ułatwiając generowanie raportów dla inspekcji.

Zastosowanie

Jeśli rejestr konfiguracji jest używany do innych celów, warto wykorzystać go także jako rejestr parametrów stosu. Na przykład rejestr konfiguracji stanowi użyteczną metodę integracji wielu stosów (patrz „Wykrywanie zależności między stosami” na stronie 281).

Konsekwencje

Rejestr parametrów stosu wymaga rejestru konfiguracji, który stanowi dodatkową ruchomą część w całym systemie. Rejestr jest zależnością dla stosu i potencjalnym punktem awarii. Brak dostępności rejestru może uniemożliwiać ponowne udostępnienie lub aktualizację stosu do momentu, aż rejestr zostanie przywrócony. Taka zależność bywa bolesna w scenariuszach odzyskiwania po awarii i powoduje umieszczenie usługi rejestru na ścieżce krytycznej.

Zarządzanie wartościami parametrów niezależnie od kodu stosu, który ich używa, ma swoje wady i zalety. Z jednej strony można zmieniać konfigurację instancji stosu bez dokonywania zmian w projekcie stosu. Jeśli jeden zespół zajmuje się projektem stosu wielokrotnego użytku, pozostałe zespoły mogą używać go do tworzenia własnych instancji stosu bez konieczności dodawania lub zmieniania plików konfiguracyjnych w samym projekcie stosu.

Z drugiej strony dokonywanie zmian w więcej niż jednym miejscu – projekcie stosu i rejestrze parametrów – zwiększa złożoność i ryzyko błędów.

Implementacja

Dwa sposoby implementacji rejestru parametrów omówię w podrozdziale „Rejestr konfiguracji” na stronie 93. W skrócie, może to być usługa przechowująca pary klucz/wartość albo może to być struktura plików lub katalogów zawierających pary klucz/wartość. W obu przypadkach wartości parametrów są zwykle przechowywane w strukturze hierarchicznej, dzięki czemu można je zapisywać i wyszukiwać na podstawie środowiska i stosu, a być może także innych czynników, takich jak aplikacja, usługa, zespół, lokalizacja lub klient.

Wartości dla klastra kontenerów wykorzystywanego w tym rozdziale mogą wyglądać jak w przykładzie 7-13.

Przykład 7-13 Przykład wpisów w rejestrze konfiguracji

```
└─ env/
   └─ test/
      └─ cluster/
         └─ min = 1
         └─ max = 1
      └─ staging/
         └─ cluster/
            └─ min = 2
            └─ max = 3
      └─ production/
         └─ cluster/
            └─ min = 2
            └─ max = 6
```

Gdy stosujemy kod stosu infrastruktury do instancji, narzędzie stosu pobiera odpowiednią wartość na podstawie klucza. Trzeba przekazać parametr `environment` do narzędzia stosu, aby kod mógł go użyć do wyszukania odpowiedniej lokalizacji w rejestrze:

```
cluster:
  id: container_cluster-${environment}
  minimum: ${get_value("/env/${environment}/cluster/min")}
  maximum: ${get_value("/env/${environment}/cluster/max")}
```

Taka implementacja wiąże kod stosu z rejestrem konfiguracji. Rejestr jest niezbędny do uruchamiania i testowania kodu, co może być zbytnim obciążeniem. Można to obejść pobierając wartości z rejestru za pomocą skryptu, przekazującego je następnie do kodu stosu w postaci zwykłych parametrów. Taka metoda pozwala elastycznie ustawiać parametry na inne sposoby. Jest to szczególnie przydatne w przypadku kodu stosu wielokrotnego użytku, gdyż daje użytkownikom kodu więcej opcji konfigurowania ich instancji stosu.

Usługi zarządzania wpisami tajnymi (patrz „Zasoby pamięci masowej” na stronie 27) są specjalnym rodzajem rejestru parametrów. Używane prawidłowo zapewniają dostęp do wpisów tajnych tylko tym osobom i usługom, które ich potrzebują, bez szerszego ich ujawniania. Niektóre usługi i produkty rejestru konfiguracji mogą być wykorzystywane do przechowywania zarówno wartości tajnych, jak i nietajnych. Ważne jest jednak, aby unikać zapisywania wartości tajnych w rejestrach, które ich nie chronią. W przeciwnym razie takie rejestry stają się łatwym celem dla atakujących.

Powiązane wzorce

Zwykle trzeba przekazać co najmniej jeden parametr do narzędzia stosu, aby wskazać, które parametry instancji stosu mają zostać użyte. W tym celu można użyć wzorca skryptu udostępniającego stos albo wzorca parametrów stosu potokowego.



Wzorce konfiguracji łańcuchowej

Większość narzędzi obsługuje łańcuch opcji konfiguracyjnych z przewidywalną hierarchią pierwszeństwa – wartości plików konfiguracyjnych są przesłane przez zmienne środowiskowe, a te są przesłane przez parametry wiersza polecenia. Większość trybów konfiguracji stosu stosuje podobną hierarchię.

Rejestr konfiguracji

Większe organizacje z dużą liczbą zespołów pracujących nad większymi systemami o wielu ruchomych częściach często uznają rejestr konfiguracji za przydatny. Można go używać do konfigurowania instancji stosów, jak to pisałem w podrozdziale „Wzorec: rejestr parametrów stosu” na stronie 90. Można go także używać do zarządzania zależnościami integracji różnych instancji stosu, aplikacji i innych usług, jak to wyjaśnię w podrozdziale „Wykrywanie zależności między stosami” na stronie 281.

Rejestr może stanowić przydatne źródło informacji o składzie i stanie infrastruktury. Można używać go do tworzenia narzędzi, pulpitów nawigacyjnych i raportów, a także do monitorowania i inspekcji systemów.

Warto więc zapoznać się ze sposobami implementacji i wykorzystania rejestru konfiguracji.

Implementowanie rejestru konfiguracji

Istnieją różne metody tworzenia rejestru konfiguracji. Można użyć gotowego rejestru udostępnianego przez posiadane narzędzie do automatyzacji infrastruktury, ale można też uruchomić produkt do tworzenia rejestru ogólnego przeznaczenia. Większość dostawców chmury pozwala również korzystać z własnych usług rejestru konfiguracji. Jeśli ktoś jest odważny, może samodzielnie stworzyć praktyczny rejestr używając dość prostych elementów.

Rejestry narzędzi do automatyzacji infrastruktury

Wiele łańcuchów narzędzi do automatyzacji infrastruktury zawiera usługę rejestru konfiguracji. Zwykle jest to część scentralizowanej usługi, która może obejmować dodatkowo takie funkcje, jak zarządzanie kodem źródłowym, monitorowanie, pulpity nawigacyjne i orkiestrację poleceń. Oto kilka przykładów:

- Chef Infra Server³
- PuppetDB⁴
- Ansible Tower⁵
- Salt Mine⁶

Bywa też możliwe używanie tych usług z narzędziami spoza oferującego je łańcucha. Większość potrafi eksponować wartości, dzięki czemu można napisać skrypt, który będzie odnajdywał informacje o bieżącym stanie infrastruktury zarządzanej przez narzędzie konfiguracji. Niektóre rejestry narzędzi infrastruktury są rozszerzalne, co pozwala przechowywać w nich dane pochodzące z innych narzędzi.

Powoduje to jednak zależność od łańcucha narzędzi, który zapewnia usługę rejestru. Taka usługa może nie w pełni obsługiwać integrację z narzędziami innych firm. Może nie udostępniać kontraktu ani interfejsu API, gwarantującego przyszłą kompatybilność.

Jeśli więc ktoś rozważa wykorzystanie magazynu danych narzędzia infrastruktury jako rejestru konfiguracji ogólnego przeznaczenia, musi się przyjrzeć, czy dobrze obsługuje ten przypadek użycia i jakie niesie ograniczenia.

Gotowe rejestry konfiguracji ogólnego przeznaczenia

Istnieje wiele dedykowanych produktów zapewniających dostęp do rejestru konfiguracji i bazy danych dla magazynu klucz-wartość poza łańcuchami narzędzi konkretnego narzędzia automatyzacji. Oto kilka przykładów:

- Zookeeper⁷
- etcd⁸
- Consul⁹
- doozerd¹⁰

Produkty te są zasadniczo kompatybilne z różnymi narzędziami, językami i systemami, więc nie powodują ograniczenia do żadnego konkretnego łańcucha narzędzi.

³ <https://oreil.ly/Fe3Ky>

⁴ <https://oreil.ly/tO8Na>

⁵ <https://oreil.ly/XXq8y>

⁶ <https://oreil.ly/YvCOB>

⁷ https://oreil.ly/N_zla

⁸ <https://oreil.ly/Ez96x>

⁹ <https://www.consul.io>

¹⁰ <https://oreil.ly/nfdbT>

Jednak zdefiniowanie sposobu przechowywania danych może wymagać całkiem sporo pracy. Czy klucze powinny mieć strukturę w rodzaju *environment/service/application*, *service/application/environment*, czy całkiem inną? Integracja różnych systemów z takim rejestrem może wymagać napisania i utrzymywania niestandardowego kodu. Ponadto rejestr konfiguracji stanowi dla zespołu kolejną rzecz, którą trzeba wdrożyć i uruchomić.

Usługi rejestru na platformie chmurowej

Większość platform chmurowych udostępnia usługę przechowywania par klucz-wartość, na przykład AWS SSM Parameter Store. Ma ona większość zalet gotowego rejestru konfiguracji ogólnego przeznaczenia, a do tego nie musimy jej samodzielnie instalować i obsługiwać. Z kolei wiąże nas ona z dostawcą chmury. W niektórych przypadkach okazuje się, że usługa rejestru w jednej chmurze jest używana do zarządzania infrastrukturą działającą w innej chmurze!

Rejestry konfiguracji własnej roboty

Zamiast uruchamiać serwer rejestru konfiguracji, niektóre zespoły tworzą niestandardowy, uproszczony rejestr konfiguracji, przechowując pliki konfiguracyjne w centralnej lokalizacji lub używając magazynu rozproszonego. Zazwyczaj wykorzystywana jest istniejąca usługa przechowywania plików, jak magazyn obiektów (np. zasobnik S3 w AWS), system kontroli wersji, sieciowy system plików, a nawet serwer WWW.

Odmianą tego podejścia jest pakowanie ustawień konfiguracyjnych do pakietów systemowych, takich jak plik *.deb* czy *.rpm* (dla dystrybucji Linuksa opartej odpowiednio na wersji Debian lub Red Hat) i wysyłanie ich do wewnętrznego repozytorium APT lub YUM. Można następnie pobierać takie pliki konfiguracyjne do lokalnych serwerów, używając standardowego narzędzia do zarządzania pakietami.

Innym wariantem jest używanie standardowego serwera relacyjnej bazy danych lub magazynu dokumentów.

Wszystkie te podejścia wykorzystują istniejące usługi, dzięki czemu można je szybko zaimplementować w przypadku prostych projektów, zamiast instalować i uruchamiać nowy serwer. Ale w sytuacjach, które nie są już tak trywialne, może się okazać, że tworzymy i utrzymujemy funkcjonalność, która leży gotowa na półce.

Jeden czy wiele rejestrów konfiguracji

Połączenie wszystkich wartości konfiguracyjnych ze wszystkich systemów, usług i narzędzi wygląda na atrakcyjny pomysł. Dzięki temu wszystko byłoby trzymane w jednym miejscu, a nie rozproszone po wielu różnych systemach. „Jeden rejestr, który rządzi wszystkim”. Jednak nie zawsze jest to praktyczne w przypadku większych, bardziej heterogenicznych środowisk.

Wiele narzędzi, takich jak usługi monitorowania i systemy konfiguracji serwerów, ma swój własny rejestr. Często można znaleźć gotowe rejestry i katalogi, które bardzo

dobrze nadają się do konkretnych celów, takich jak zarządzanie licencjami, odnajdywanie usług czy katalogi użytkowników. Naginanie wszystkich tych narzędzi do korzystania z jednego systemu prowadzi do nieustannego przepływu pracy. Każda aktualizacja dowolnego narzędzia wymaga oceny, testowania i potencjalnie więcej pracy w celu podtrzymania integracji.

Lepszym wyjściem może być pobieranie odpowiednich danych z usług, w których są przechowywane. Trzeba się tylko upewnić, który system jest źródłem prawdy dla konkretnych danych lub elementu konfiguracji i mając tę wiedzę, zaprojektować swoje systemy i narzędzia.

Niektóre zespoły wykorzystują systemy wiadomości do udostępniania danych konfiguracyjnych jako zdarzeń. Za każdym razem, gdy system zmienia wartość konfiguracyjną, wysyła zdarzenie. Pozostałe systemy mogą monitorować kolejkę zdarzeń pod kątem zmian elementów konfiguracji, którymi są zainteresowane.

Obsługa wpisów tajnych jako parametrów

Systemy potrzebują różnych wpisów tajnych. Używane przez nas narzędzie stosu może potrzebować hasła lub klucza, aby użyć API platformy w celu utworzenia lub zmiany infrastruktury. Może być także konieczne udostępnianie wpisów tajnych środowiskom, na przykład, aby jakaś aplikacja dysponowała hasłem potrzebnym do połączenia się z bazą danych.

Ważne jest, żeby od samego początku obchodzić się z tego typu wpisami tajnymi w sposób bezpieczny. Zarówno w przypadku korzystania z chmury publicznej, jak i prywatnej ujawnienie hasła może mieć straszne konsekwencje. Dlatego nawet jeśli piszemy kod tylko po to, aby nauczyć się używać nowego narzędzia lub platformy, nigdy nie powinniśmy umieszczać wpisów tajnych w kodzie. Krąży wiele historii o ludziach, którzy umieścili wpis tajny w repozytorium kodu źródłowego, uważając go za prywatny, aby dowiedzieć się wkrótce, że został on odkryty przez hakerów i wykorzystany, powodując ogromne straty finansowe.

Istnieje kilka podejść do obsługi wpisów tajnych wymaganych przez kod infrastruktury bez faktycznego umieszczania ich w kodzie. Należy do nich szyfrowanie wpisów tajnych, autoryzacja bez wpisu tajnego, wstrzykiwanie wpisów tajnych podczas wykonywania oraz wpisy tajne jednorazowe.

Szyfrowanie wpisów tajnych

Wyjątkiem od reguły zabraniającej umieszczania wpisów tajnych w kodzie źródłowym jest szyfrowanie ich w kodzie; git-crypt, blackbox, sops i transcrypt to tylko niektóre narzędzia, które pomagają szyfrować wpisy tajne w repozytorium.

Klucz do odszyfrowania wpisu tajnego nie powinien z kolei znajdować się w repozytorium; w przeciwnym razie byłby dostępny dla atakujących po uzyskaniu przez nich dostępu do kodu. Aby umożliwić odszyfrowywanie, trzeba użyć jednej z pozostałych opisanych tutaj metod.

Autoryzacja bez wpisu tajnego

Wiele usług i systemów udostępnia sposoby autoryzacji działań bez użycia wpisu tajnego. Wiele platform chmurowych może oznaczyć instancję obliczeniową – taką jak maszyna wirtualna czy instancja kontenera – jako uprawnioną do wykonywania uprzywilejowanych działań.

Na przykład instancji AWS EC2 można przypisać profil IAM, który daje procesom w tej instancji prawo wykonywania zbioru poleceń API. Jeśli skonfigurujemy narzędzie do zarządzania stosem tak, aby było wykonywane w takiej instancji, unikniemy konieczności zarządzania wpisem tajnym, który mógłby zostać wykorzystany przez atakujących.

W niektórych przypadkach można użyć autoryzacji bez wpisu tajnego w celu uniknięcia potrzeby dostarczania tego wpisu podczas tworzenia infrastruktury. Weźmy jako przykład serwer aplikacji, który wymaga dostępu do instancji bazy danych. Zamiast dostarczać hasło do serwera aplikacji za pomocą narzędzia konfiguracyjnego serwera, można tak skonfigurować serwer bazy danych, aby autoryzował połączenia z serwera aplikacji, choćby na podstawie jego adresu sieciowego.

Wiązanie uprawnień z instancją obliczeniową lub adresem sieciowym zmienia jedynie wektor potencjalnego ataku. Każdy, kto uzyska dostęp do takiej instancji, będzie mógł wykorzystać jej uprawnienia. Trzeba dołożyć wszelkich starań, aby chronić dostęp do uprzywilejowanych instancji. Z drugiej strony ktoś, kto uzyska dostęp do instancji, może uzyskać dostęp do przechowywanych w niej wpisów tajnych, więc nadawanie uprawnień takiej instancji nie musi być wcale gorsze. Ponadto wpis tajny może zostać potencjalnie wykorzystany z innych lokalizacji, dlatego ogólnie dobrze jest całkowicie zrezygnować z używania wpisów tajnych.

Wstrzykiwanie wpisów tajnych podczas wykonywania

Jeśli nie można uniknąć korzystania z wpisu tajnego na potrzeby stosu lub innego kodu infrastruktury, można rozważyć metody wstrzykiwania go podczas wykonywania. Zazwyczaj implementacja ma postać parametru stosu. W rozdziale tym opisuję szczegółowo obsługi wpisów tajnych jako parametrów dla każdego wzorca i antywzorca.

Należy rozważyć dwie różne sytuacje wykonywania: lokalne środowisko projektowe oraz nienadzorowanych agentów. Osoby pracujące nad kodem infrastruktury często trzymają wpisy tajne w pliku lokalnym, który nie podlega systemowi kontroli wersji¹¹. Narzędzie stosu może czytać taki plik bezpośrednio, co jest właściwe zwłaszcza w przypadku używania wzorca pliku konfiguracyjnego stosu (patrz „Wzorzec: pliki konfiguracyjne stosu” na stronie 81). Plik może być też skryptem ustawiającym wpisy tajne w zmiennych środowiskowych, co dobrze działa z wzorcem zmiennych środowiskowych stosu (patrz „Wzorzec: zmienne środowiskowe stosu” na stronie 76).

¹¹ Pracą lokalną nad kodem stosu zajmę się bardziej szczegółowo w podrozdziale „Prywatne instancje infrastruktury” na stronie 338.

Podejścia te działają również w przypadku agentów nienadzorowanych, takich jak te, które są używane do testowania CI lub potoków dostarczania CD¹². Ale wpisy tajne trzeba wówczas przechowywać w serwerze lub kontenerze, w którym uruchomiono agenta. Alternatywą jest wykorzystywanie funkcji oprogramowania agenta do zarządzania wpisami tajnymi w celu dostarczenia wpisów do polecenia `stack`, jak to ma miejsce w przypadku wzorca parametrów stosu potokowego (patrz „Wzorzec: parametry stosu potokowego” na stronie 87). Inną opcją jest pobieranie wpisów tajnych od usługi zarządzania wpisami tajnymi (typu opisanego w podrozdziale „Zasoby pamięci masowej” na stronie 27), co jest równoważne wzorcowi rejestru parametrów stosu (patrz „Wzorzec: rejestr parametrów stosu” na stronie 90).

Wpisy tajne jednorazowe

Ciekawym rozwiązaniem, które umożliwiają platformy dynamiczne, jest tworzenie wpisów tajnych w locie i używanie ich tylko na zasadzie „ścisłej potrzeby”. W przykładzie z hasłem bazy danych kod wyposażający bazę danych automatycznie generuje hasło i przekazuje je do kodu wyposażającego serwer aplikacji. Ludzie nie muszą nigdy widzieć tego wpisu tajnego, dlatego nie jest on przechowywany nigdzie indziej.

W razie potrzeby można zastosować kod, który zresetuje hasło. Jeśli serwer aplikacji zostanie odbudowany, można ponownie uruchomić kod serwera bazy danych, aby wygenerować dla niego nowe hasło.

Usługi zarządzania wpisami tajnymi, takie jak HashiCorp Vault, mogą również generować i ustawiać w locie hasła w innych systemach i usługach. Takie hasło może zostać następnie udostępnione narzędziu stosu w momencie wyposażania infrastruktury albo bezpośrednio wykorzystującej je usługę, takiej jak serwer aplikacji. Metoda haseł jednorazowych¹³ wykorzystuje to podejście w stopniu maksymalnym, tworząc nowe hasło za każdym razem, gdy dochodzi do uwierzytelnienia.

Podsumowanie

Wielokrotne używanie projektu stosu wymaga możliwości konfigurowania różnych instancji tego stosu. Zakres konfiguracji powinien być jak najmniejszy. Jeśli okazuje się, że są potrzebne dwie instancje stosu znacząco różne od siebie, należy zdefiniować je jako oddzielne stosy.

Projekt stosu powinien definiować kształt stosu spójny we wszystkich instancjach. Tam, gdzie występują dwie na pozór podobne rzeczy – na przykład serwery aplikacji – ale o różnych kształtach, jak najbardziej usprawiedliwione jest zastosowanie dwóch różnych projektów stosu.

¹² Sposób ich użycia jest opisany w podrozdziale „Potoki dostarczania infrastruktury” na stronie 113.

¹³ <https://oreil.ly/KheAb>

Podstawowa praktyka: ciągle testuj i dostarczaj

Ciągle testowanie i dostarczanie jest drugą z trzech podstawowych praktyk Infrastruktury jako kodu, do których należy ponadto definiowanie wszystkiego jako kodu i tworzenie małych elementów. Testowanie jest podstawą inżynierii oprogramowania zwinnego (Agile). Extreme Programming (XP)¹ kładzie nacisk na pisanie najpierw testów za pomocą TDD i częste integrowanie kodu za pomocą CI². CD rozszerza to na testowanie pełnej gotowości produkcyjnej kodu w trakcie opracowywania go przez programistów, zamiast czekania na zakończenie pracy nad jego wydaniem³.

Jeśli mocne skupienie się na testowaniu daje dobre wyniki w przypadku pisania kodu aplikacji, to rozsądnie jest oczekiwać, że będzie to również przydatne w odniesieniu do kodu infrastruktury. W tym rozdziale omówię strategię testowania i dostarczania infrastruktury. W dużym stopniu będę korzystał ze sposobów podejścia do jakości używanych w inżynierii zwinnej, w tym TDD, CI i CD. Wszystkie te praktyki budują jakość systemu poprzez wbudowanie testowania kodu w proces jego pisania, zamiast odkładania tego na później.

W tym rozdziale skoncentruję się na podstawowych podejściach do testowania infrastruktury i towarzyszących temu wyzwaniach. W następnym rozdziale wykorzystam to do przedstawienia konkretnych wskazówek na temat testowania kodu stosu infrastruktury, a w rozdziale 11 omówię testowanie kodu konfiguracji serwera (patrz „Testowanie kodu serwera” na stronie 171).

¹ <https://oreil.ly/dg1SK>

² Patrz „Continuous Integration” (<https://oreil.ly/Gck3D>), autor Martin Fowler.

³ Książka Jeza Humble’a i Davida Farleya *Continuous Delivery* (Addison-Wesley) definiuje zasady i praktyki CD, przechodząc od mglistych sformułowań w Manifeście Agile Manifesto do szeroko rozpowszechnionej praktyki wśród zespołów dostarczających oprogramowanie.

Po co ciągle testować kod infrastruktury?

Testowanie zmian w infrastrukturze to z pewnością dobry pomysł. Ale potrzeba stworzenia i utrzymywania zestawu zautomatyzowanych testów może już nie wydawać się tak oczywista. Często myślimy o tworzeniu infrastruktury jako czynności jednorazowej: utworzyć, przetestować, a potem używać. Po co wkładać wysiłek w opracowanie zestawu zautomatyzowanych testów dla czegoś, co tworzymy tylko raz?

Utworzenie zestawu zautomatyzowanych testów to ciężka praca, zwłaszcza jeśli weźmiemy pod uwagę wysiłek potrzebny do implementacji narzędzi i usług dostarczania i testowania – serwery CI, potoki, moduły uruchamiające testy, rusztowanie testów i różnego typu narzędzia do skanowania i sprawdzania poprawności. Gdy zaczynamy przygodę z infrastrukturą jako kodem, budowanie tych wszystkich rzeczy może wydawać się większym zadaniem niż samo budowanie systemów, które będą uruchamiane przy ich użyciu.

W podrozdziale „Używanie infrastruktury jako kodu do optymalizacji pod kątem zmian” na stronie 6 wyjaśniłem powody implementowania systemów dostarczania zmian infrastruktury. Powtórzę jeszcze raz: ilość zmian wprowadzanych w infrastrukturze po jej utworzeniu jest znacznie większa, niż można oczekiwać. Gdy jakiś nietrywialny system jest już używany, trzeba go co chwila łatać, aktualizować, naprawiać i ulepszać.

Główną zaletą CD jest usunięcie klasycznego rozróżnienia z epoki żelaza pomiędzy fazami „tworzenia” i „uruchamiania” w cyklu życia systemu⁴. Systemy dostarczania, łącznie ze zautomatyzowanym testowaniem i promocją kodu mają być projektowane i implementowane razem z samym systemem. Postępując w ten sposób należy stopniowo rozbudowywać infrastrukturę i ulepszać ją przez jej cały okres funkcjonowania. Bycie „używanym” oznacza niemal dowolne zdarzenie, zmianę tego, kto używa systemu, ale nie tego, jak nim zarządza.

Co oznacza ciągle testowanie

Jednym z kamieni węgielnych inżynierii zwinnej jest testowanie w trakcie pracy – *dbanie o jakość*. Im wcześniej można stwierdzić, że każda linia napisanego kodu jest gotowa do produkcji, tym szybciej można pracować i tym prędyj można dostarczyć rezultat. Szybsze znajdowanie problemów oznacza również poświęcanie mniej czasu na cofanie się w celu zbadania problemów oraz tracenie mniej czasu na naprawianie i przepisywanie kodu. Usuwanie problemów na bieżąco pozwala uniknąć kumulacji długu technicznego.

Większość ludzi rozumie znaczenie szybkiej informacji zwrotnej. Ale tym, co naprawdę wyróżnia bardzo wydajne zespoły, jest to, jak agresywnie dążą one do faktycznie nieustającej informacji zwrotnej.

Tradycyjne podejścia zakładały testowanie dopiero po zaimplementowaniu przez zespół pełnej funkcjonalności systemu. Metodologie ograniczone czasowo idą dalej. Zespół przeprowadza testy okresowo w trakcie tworzenia oprogramowania, na przykład

4 Jak to jest opisane w podrozdziale „Od epoki żelaza do epoki chmury” na stronie 4.

na koniec sprintu. Zespoły stosujące metodę Lean lub Kanban testują każdą historyjkę zaraz po jej ukończeniu⁵.

Naprawdę ciągle testowanie wymaga testowania nawet jeszcze częściej. Ludzie piszą i uruchamiają testy w trakcie tworzenia kodu, jeszcze przed ukończeniem historyjki. Często przekazują swój kod do scentralizowanego, zautomatyzowanego systemu budowy – najlepiej przynajmniej raz dziennie⁶.

Ludzie muszą otrzymywać informację zwrotną jak najszybciej po przekazaniu kodu, aby zareagować na to możliwie jak najmniejszą przerwą w swoim przepływie pracy. Ciasne pętle informacji zwrotnej stanowią istotę ciągłego testowania.

Testowanie natychmiastowe i ostateczne

Innym sposobem potraktowania tego tematu jest klasyfikowanie każdej czynności testowania jako natychmiastowej lub ostatecznej. Testowanie natychmiastowe następuje w momencie przekazywania kodu. Testowanie ostateczne następuje z pewnym opóźnieniem, na przykład po ręcznym sprawdzeniu albo według jakiegoś harmonogramu.

W wersji idealnej testowanie jest naprawdę natychmiastowe i następuje podczas pisania kodu. Są dostępne działania weryfikujące, wykonywane w samym edytorze, takie jak wyróżnianie elementów składni albo uruchamianie testów jednostkowych. Protokół LSP, Language Server Protocol (<https://langserver.org>), definiuje standard integracji kontroli składni z IDE, obsługiwany przez implementacje dla różnych języków.

Ci, którzy wolą wiersz polecenia jako środowisko programowania, mogą używać narzędzia w rodzaju inotifywait (<https://oreil.ly/h47e0>) lub entr (<https://oreil.ly/koche>) do przeprowadzania kontroli z poziomu terminala, gdy nastąpi zmiana kodu.

Innym przykładem weryfikacji natychmiastowej jest programowanie w parach, będące zasadniczo przeglądaniem kodu podczas pracy. Programowanie parami zapewnia znacznie szybszą informację zwrotną niż przeglądanie kodu, które następuje po zakończeniu pracy nad jakąś historyjką lub funkcją, gdy ktoś inny znajdzie czas, aby przejrzeć efekty naszej pracy.

Budowa CI i potok CD powinny być uruchamiane natychmiast po przekazaniu jakiejkolwiek zmiany do bazy kodu. Uruchamianie natychmiastowe po każdej zmianie nie tylko oznacza szybszą informację zwrotną, ale zapewnia też mały zasięg

5 Wyjaśnienie scenariuszy Agile można znaleźć w witrynie Mountain Goat Software (<https://oreil.ly/DEG5M>).

6 Badanie *Accelerate* opublikowane w corocznym raporcie *State of DevOps* (<https://oreil.ly/ysk9n>) pokazuje, że zespoły, w których każdy co najmniej raz dziennie scala swój kod, są zwykle bardziej wydajne od tych, w których dzieje się to rzadziej. W najbardziej wydajnych zespołach, które widziałem, programiści przekazują swój kod kilka razy dziennie, czasami nawet co godzinę.

zmiany przy każdym uruchomieniu. Jeśli potok jest uruchamiany tylko okresowo, może obejmować wiele zmian pochodzących od wielu osób. Jeśli którykolwiek test zakończy się niepowodzeniem, trudniej będzie ustalić, która zmiana spowodowała problem i w efekcie więcej osób będzie musiało poświęcić swój czas na odszukanie błędu i jego usunięcie.

Co należy testować w przypadku infrastruktury?

Istotą CI jest jak najszybsze testowanie każdej dokonanej zmiany. Istotą CD jest maksymalizacja zasięgu tego testowania. Jak mówi Jez Humble, „Osiągamy to wszystko pilnując, aby nasz kod był zawsze gotowy do wdrożenia”⁷.

Zapewnianie jakości (Quality assurance) dotyczy zarządzania ryzykiem stosowania kodu w systemie. Czy kod nie powoduje błędów po zastosowaniu? Czy tworzy właściwą infrastrukturę? Czy infrastruktura działa tak, jak powinna? Czy spełnia operacyjne kryteria wydajności, niezawodności i bezpieczeństwa? Czy jest zgodna z przepisami i regułami nadzoru?

CD dotyczy poszerzania zakresu zagrożeń, które są testowane natychmiast po przekazaniu zmiany do bazy kodu, bez czekania na testowanie ostateczne, które może mieć miejsce dni, tygodnie, a nawet miesiące później. Dlatego przy każdym przekazaniu zmiany potok stosuje kod w realistycznych środowiskach testowych i uruchamia kompleksowe testy. W wersji idealnej po przejściu kodu przez zautomatyzowane etapy potoku zostaje on zatwierdzony jako gotowy do produkcji.

Każdy zespół powinien zidentyfikować zagrożenia związane z wprowadzaniem zmian do swojego kodu infrastruktury i utworzyć powtarzalny proces testowania dowolnej zmiany pod kątem tych zagrożeń. Taki proces ma formę zestawów testów zautomatyzowanych i testów ręcznych. Zestaw testów jest kolekcją testów zautomatyzowanych uruchamianych jako grupa.

Kiedy ludzie myślą o testowaniu zautomatyzowanym, na ogół myślą o testach funkcjonalnych, jak testy jednostkowe i testy podróży oparte na interfejsie użytkownika. Ale zakres zagrożeń obejmuje więcej niż tylko wady funkcjonalne, dlatego zakres weryfikacji jest również szerszy. Ograniczenia i wymagania wykraczające poza samą funkcjonalność są często nazywane wymaganiami niefunkcjonalnymi (Non-Functional Requirements – NFR) lub wymaganiami współzależności funkcjonalnych (Cross-Functional Requirements – CFR)⁸. Oto niektóre przykłady tego, co można weryfikować, zarówno automatycznie, jak i ręcznie:

7 Więcej o wzorcach CD można znaleźć w witrynie Jeza Humble’a (<https://continuousdelivery.com>).

8 Moja koleżanka Sarah Taraporewalla ukuła termin CFR, aby podkreślić, że ludzie nie powinni uważać ich za odrębne od pracy programistycznej, lecz mające zastosowanie do całej pracy. Zajrzyj na jej witrynę (<https://oreil.ly/J75na>).

Jakość kodu

Czy kod jest czytelny i można go utrzymywać? Czy jest zgodny ze standardami zespołu dotyczącymi formatu i struktury kodu? W zależności od używanych narzędzi i języków może być dostępne skanowanie kodu pod kątem błędów syntaktycznych i zgodności z regułami formatowania oraz uruchamianie analizy złożoności. Wiele z tych narzędzi nie jest dostępnych dla języków infrastruktury (czasem nawet żadne!). Zależy to od ich popularności i tego, jak długo już istnieją. Ręczne metody sprawdzania obejmują procesy warunkowego przeglądania kodu, sesje prezentacji kodu oraz programowanie parami.

Funkcjonalność

Czy kod robi to, co powinien? Ostateczne przetestowanie funkcjonalności odbywa się poprzez wdrożenie aplikacji w infrastrukturze i sprawdzanie, czy działa poprawnie. Takie działanie pozwala sprawdzić pośrednio, czy infrastruktura jest prawidłowa, ale często także wychwycić problemy przed wdrożeniem aplikacji. Przykładem tego w przypadku infrastruktury jest routing sieciowy. Czy można nawiązać połączenie HTTPS między publicznym Internetem a serwerami WWW? Tego typu rzeczy można testować, używając podzbioru całej infrastruktury.

Bezpieczeństwo

Bezpieczeństwo można testować na różnych poziomach – od skanowania kodu, przez testowanie jednostkowe, aż po testy integracyjne i monitorowanie środowiska produkcyjnego. Istnieją narzędzia przeznaczone konkretnie do testowania bezpieczeństwa, takie jak skanery luk w zabezpieczeniach. Z reguły dobrze jest również dopisać testy bezpieczeństwa do standardowego zestawu testów. Na przykład testy jednostkowe mogą stosować asercje dotyczące otwartych portów, obsługi kont użytkowników czy uprawnień dostępu.

Zgodność

Systemy mogą wymagać zgodności z prawem, przepisami, normami branżowymi, zobowiązaniami umownymi lub zasadami organizacji. Zapewnianie i udowadnianie zgodności bywa czasochłonnym zajęciem dla zespołów ds. infrastruktury i działania. Niezwykle przydatne do tego może być testowanie automatyczne, pozwalające szybko wykryć naruszenia i dostarczyć dowody dla audytorów. Tak jak w przypadku bezpieczeństwa, można to robić na wielu poziomach walidacji, od poziomu kodu po testowanie wersji produkcyjnej. Szersze omówienie tych działań można znaleźć w podrozdziale „Nadzór w przepływie pracy opartym na potoku” na stronie 343.

Wydażność

Zautomatyzowane narzędzia mogą sprawdzać, jak szybko wykonują się określone działania. Testowanie szybkości połączenia sieciowego od punktu A do punktu B może ujawnić problemy z konfiguracją sieci lub z platformą chmurową, jeśli zostanie wykonane jeszcze przed wdrożeniem aplikacji. Znajdowanie problemów z wydajnością w podzbiorze systemu to kolejny przykład tego, jak można szybciej uzyskać informacje zwrotne.

Skalowalność

Zautomatyzowane testy mogą udowodnić prawidłowe działanie skalowania; na przykład sprawdzając, czy automatycznie skalowany klaster dodaje węzły, gdy potrzeba. Testy mogą również sprawdzać, czy skalowanie daje oczekiwane wyniki. Na przykład może się okazać, że dodawanie węzłów do klastra nie poprawia wydajności z powodu wąskiego gardła w innym miejscu systemu. Częste uruchamianie tych testów pozwala szybko odkryć, że jakaś zmiana infrastruktury zakłóciła skalowanie.

Dostępność

Podobnie, zautomatyzowane testowanie może dowieść, że system pozostanie dostępny nawet w przypadku awarii. Testy mogą niszczyć zasoby, takie jak węzły klastra i sprawdzać czy klaster automatycznie je zastępuje. Można też sprawdzać, czy problemy, które nie są automatycznie rozwiązywane, są przynajmniej bezpiecznie obsługiwane; na przykład poprzez wyświetlenie strony błędu i niedopuszczenie do uszkodzenia danych.

Operatywność

Można automatycznie testować dowolne inne wymagania systemowe potrzebne do działania. Zespoły mogą testować monitorowanie (wywoływać błędy i udowadniać, że monitoring wykrywa je i raportuje), rejestrowanie i zautomatyzowane czynności konserwacyjne.

Validacje wszystkich tych typów mogą być wykonywane na więcej niż jednym poziomie zakresu, od konfiguracji serwera, przez kod stosu aż po w pełni zintegrowany system. Omówię to w podrozdziale „Testowanie progresywne” na stronie 109. Ale najpierw chcę zająć się tym, co sprawia, że infrastruktura jest szczególnie trudna do testowania.

Wyzwania związane z testowaniem kodu infrastruktury

Większość spotykanych przeze mnie zespołów, które pracują z infrastrukturą jako kodem, zmaga się z implementacją zautomatyzowanego testowania i dostarczania swojego kodu infrastruktury na tym samym poziomie, jaki stosują w przypadku kodu aplikacji. A wiele zespołów bez doświadczenia w inżynierii oprogramowania zwinnego ma z tym jeszcze większy problem.

Założeniem infrastruktury jako kodu jest możliwość stosowania do infrastruktury praktyk inżynierii oprogramowania, takich jak testowanie zwinne. Są jednak znaczące różnice między kodem infrastruktury a kodem aplikacji. Dlatego trzeba dostosowywać niektóre techniki i sposoby myślenia rodem z testowania aplikacji, aby miały praktyczne zastosowanie do infrastruktury.

Przedstawię teraz kilka wyzwań, które wynikają z różnic między kodem infrastruktury a kodem aplikacji.

Wyzwanie: testy kodu deklaratywnego mają często małą wartość

Jak wspomniałem w rozdziale 4 („Deklaratywne języki infrastruktury” na stronie 37), wiele narzędzi infrastruktury używa języków deklaratywnych zamiast imperatywnych. Kod deklaratywny zazwyczaj deklaruje pożądany stan jakiejś infrastruktury, jak poniższy kod definiujący podsieć:

Przykład 8-1

```
subnet:
  name: private_A
  address_range: 192.168.0.0/16
```

Test tego kodu może polegać po prostu na jego powtórzeniu:

```
assert:
  subnet("private_A").exists
assert:
  subnet("private_A").address_range is("192.168.0.0/16")
```

Zestaw testów niskiego poziomu kodu deklaratywnego może przybrać postać ćwiczenia z księgowości. Po każdej zmianie kodu infrastruktury trzeba zmienić test, aby pasował do tej zmiany. Jaką wartość mają takie testy? No cóż, testowanie polega na zarządzaniu ryzykiem, zastanówmy się więc, jakiego rodzaju ryzyko powyższy test może odkryć:

- Kod infrastruktury nie został nigdy zastosowany.
- Kod infrastruktury został zastosowany, ale narzędzie nie zrobiło tego poprawnie i nie został zwrócony błąd.
- Ktoś zmienił kod infrastruktury, ale zapomniał dopasować test do tej zmiany.

Pierwsze ryzyko może być realne, ale nie wymaga ono oddzielnego testu dla każdej deklaracji. Zakładając, że kod wykonuje wiele rzeczy w serwerze, wystarczy pojedynczy test, aby ujawnić, że z jakiegoś powodu kod nie został zastosowany. Drugie ryzyko sprowadza się do zabezpieczenia się przed błędem w używanym narzędziu. Twórcy tego narzędzia powinni naprawić błąd albo zespół powinien zmienić narzędzie na bardziej niezawodne. Widziałem zespoły wykonujące tego typu testy w sytuacjach, gdy odkryły konkretny błąd i chciały się przed nim zabezpieczyć. Takie testowanie jest uzasadnione, gdy chcemy poradzić sobie ze znanym błędem, ale poddawanie kodu szczegółowym testom tylko ze względu na ryzyko występowania błędu w narzędziu jest marnotrawstwem.

Ostatnim ryzykiem jest logika cykliczna. Usunięcie testu usunęłoby ryzyko, którego test dotyczy, a do tego usunęłoby zajęcie dla zespołu.



Testy deklaratywne

Do pisania testów przydatny jest format *Given, When, Then* (dla, gdy, wtedy)⁹. Test deklaracyjny pomija część „When”, tworząc format w rodzaju „Given rzecz, Then ma ona następujące cechy.” Tak napisane testy sugerują, że testowany kod nie generuje zmiennych wyników. Testy deklaratywne, podobnie jak kod deklaracyjny, znajdziemy w wielu bazach kodu infrastruktury, ale trzeba mieć świadomość, że wiele narzędzi i praktyk może się nie nadawać do testowania kodu dynamicznego.

Zdarzają się przypadki, w których dobrze jest przetestować kod deklaracyjny. Dwa, które przychodzą mi na myśl, to kod deklaracyjny, który może dawać różne wyniki oraz łączenie wielu deklaracji.

Testowanie zmiennego kodu deklaracyjnego

Poprzedni przykład kodu deklaracyjnego był prosty – wartości były zakodowane na stałe, więc rezultat zastosowania kodu był oczywisty. Zmienne wprowadzają możliwość generowania różnych wyników, przez co mogą powstawać zagrożenia, w wyniku których testowanie staje się bardziej użyteczne. Zmienne nie zawsze tworzą warianty wymagające testowania. Co by się stało, gdybyśmy dodali kilka prostych zmiennych do wcześniejszego przykładu?

```
subnet:
  name: ${MY_APP}-${MY_ENVIRONMENT}
  address_range: ${SUBNET_IP_RANGE}
```

Ten kod nie niesie wielkiego ryzyka, którym by nie zarządzało stosujące go narzędzie. Jeśli ktoś ustawi nieprawidłowe wartości zmiennych, narzędzie nie zadziała pomyślnie i zgłosi błąd.

Kod staje się bardziej ryzykowny w przypadku większej liczby możliwych wyników. Dodajmy do przykładu trochę kodu warunkowego:

```
subnet:
  name: ${MY_APP}-${MY_ENVIRONMENT}
  address_range: get_networking_subrange(
    get_vpc(${MY_ENVIRONMENT}),
    data_centers.howmany,
    data_centers.howmany++
  )
```

Teraz kod zawiera pewną logikę, która może być warta testowania. Wywołuje dwie funkcje, `get_networking_subrange` i `get_vpc`, z których każda może zakończyć się niepowodzeniem albo zwrócić rezultat, który w nieoczekiwany sposób wpłynie na drugą funkcję.

9 Post Perry'ego Fowlera (<https://oreil.ly/HIv5b>) zawiera objaśnienie testów *Given, When, Then*.

Efekt zastosowania tego kodu różni się w zależności od danych wejściowych i kontekstu, co sprawia, że warto jest napisać testy.



Wyobraźmy sobie, że zamiast wywoływać te funkcje, napisalibyśmy, jako część deklaracji naszej podsieci, kod wybierający podzbiór zakresu adresów. Byłby to przykład łączenia kodu deklaratywnego i imperatywnego (opisany w podrozdziale „Oddzielaj kod deklaratywny od imperatywnego” na stronie 43). Testy kodu podsieci musiałyby uwzględniać różne skrajne przypadki kodu imperatywnego. Na przykład co się stanie, jeśli zakres nadrzędny będzie mniejszy od potrzebnego?

Jeśli kod deklaratywny jest na tyle złożony, że wymaga złożonego testowania, jest to sygnał, że należy zabrać część logiki z deklaracji i umieścić ją w bibliotece napisanej w języku proceduralnym. Następnie można napisać oddzielne testy dla takiej funkcji i uprościć test dla deklaracji podsieci.

Testowanie kombinacji kodu deklaratywnego

Inna sytuacja, w której testowanie ma większą wartość, ma miejsce w przypadku wielu deklaracji infrastruktury tworzących razem bardziej skomplikowane struktury. Na przykład może to być kod definiujący wiele struktur sieciowych – blok adresów, moduł równoważenia obciążenia, reguły routingu i bramę. Każdy fragment takiego kodu jest zwykle na tyle prosty, że nie wymaga testowania. Jednak ich kombinacja daje rezultat, który jest już wart testowania – na przykład czy można nawiązać połączenie sieciowe z punktu A do punktu B.

Testowanie, czy narzędzie utworzyło rzeczy zadeklarowane w kodzie, jest zwykle mniej wartościowe, niż sprawdzenie, czy te rzeczy zapewniają oczekiwany rezultat.

Wyzwanie: testowanie kodu infrastruktury jest powolne

Aby przetestować kod infrastruktury, trzeba zastosować go do odpowiedniej infrastruktury. A udostępnianie instancji infrastruktury jest często powolne, zwłaszcza gdy trzeba ją utworzyć na platformie chmurowej. Większość zespołów zmagających się z implementacją zautomatyzowanego testowania infrastruktury stwierdza, że czas potrzebny na utworzenie testowej infrastruktury stanowi przeszkodę dla szybkiej informacji zwrotnej.

Rozwiązaniem jest zwykle stosowanie kombinacji różnych strategii:

Dzielenie infrastruktury na bardziej poręczne fragmenty

Dobrze jest włączyć testowalność jako element projektowania struktury systemu, gdyż jest to jeden z podstawowych sposobów uzyskania systemu łatwego do utrzymania, rozbudowy i ewolucji. Jedną z taktyk jest stosowanie małych fragmentów, gdyż z reguły można je szybciej udostępniać i testować. Łatwiej jest pisać i utrzymywać testy dla małych, luźno powiązanych elementów, ponieważ są one prostsze i mają mniejszy obszar ryzyka. W rozdziale 15 omówię ten temat bardziej szczegółowo.

Precyzowanie, minimalizowanie i izolowanie zależności

W każdym elemencie naszego systemu mogą występować zależności od innych części tego systemu, od usług platformy oraz od usług i systemów znajdujących się poza naszym zespołem, działem czy organizacją. To wszystko ma wpływ na testowanie, zwłaszcza gdy trzeba polegać na kimś innym, kto ma udostępnić instancje potrzebne do przeprowadzenia testów. Mogą być one powolne, drogie, zawodne lub zawierać niespójne dane testowe, zwłaszcza jeśli są współdzielone z innymi użytkownikami. Przydatnym sposobem izolacji komponentu, w celu jego szybkiego przetestowania, są atrapy (*test doubles*). Atrapy mogą być używane jako część strategii testowania progresywnego – najpierw testujemy komponent przy użyciu atrap, a potem testujemy go w wersji zintegrowanej z innymi komponentami i usługami. Więcej informacji o atrapach jest w podrozdziale „Używanie warunków początkowych testu do obsługi zależności” na stronie 132.

Testowanie progresywne

Zwykle dysponujemy wieloma zestawami testów do testowania różnych aspektów systemu. Najpierw można uruchamiać szybkie testy, aby w przypadku niepowodzenia mieć szybciej informację zwrotną, a dopiero potem uruchamiać testy wolniejsze, obejmujące szerszy zakres. Zajmę się tym dokładniej w następnym podrozdziale „Testowanie progresywne”.

Wybieranie instancji efemerycznych lub trwałych

Można tworzyć i niszczyć instancję infrastruktury przy każdym testowaniu (*instancja efemeryczna*), ale można też pozostawiać ją działającą między testami (*instancja trwała*). Używanie instancji efemerycznych może znacząco spowalniać testy, ale są bardziej przejrzyste i dają bardziej spójne wyniki. Stosowanie instancji trwałych skraca czas potrzebny do uruchamiania testów, ale może prowadzić do pozostawiania zmian, a z czasem kumulowania niespójności. Należy wybrać odpowiednią strategię dla danego zestawu testów i zrewidować decyzję po jakimś czasie w zależności od tego, jak się sprawdza. Bardziej konkretne przykłady implementacji instancji efemerycznych i trwałych są przedstawione w podrozdziale „Wzorzec: efemeryczny stos testowy” na stronie 137.

Testy online i offline

Niektóre typy testów trzeba uruchamiać *online*, co wymaga udostępnienia infrastruktury na „prawdziwej” platformie chmurowej. Inne można uruchamiać *offline* na własnym laptopie lub agencji kompilacji. Testy, które można uruchamiać offline, obejmują sprawdzanie składni kodu oraz testy wykonywane z użyciem maszyny wirtualnej lub instancji kontenera. Należy znać charakter swoich testów, żeby wiedzieć w jaki sposób można je uruchamiać. Testowanie offline jest zwykle znacznie szybsze, więc wykonuje się je raczej wcześniej. W przypadku niektórych testów można użyć atrap do emulacji API chmury w trybie offline. Więcej informacji o testowaniu stosów offline i online zawartych jest w podrozdziałach „Etapy testowania offline dla stosów” na stronie 125 i „Etapy testowania online dla stosów” na stronie 128.

W przypadku każdej z tych strategii należy regularnie oceniać jakość działania. Jeśli testy są niewiarygodne, nie działają poprawnie lub zwracają niespójne wyniki, należy poszukać przyczyn i albo je naprawić, albo zastąpić innymi. Jeśli pewne testy rzadko kończą się niepowodzeniem albo te same testy prawie zawsze wspólnie kończą się niepowodzeniem, można spróbować usunąć je, aby uprościć zestaw testów. Jeśli więcej czasu zajmuje wyszukiwanie i usuwanie problemów w samych testach, a nie w testowanym kodzie, trzeba poszukać sposobów na ich uproszczenie i ulepszenie.

Wyzwanie: zależności komplikują testowanie infrastruktury

Czas potrzebny na skonfigurowanie infrastruktury innej niż ta, od której zależy nasz kod, jeszcze bardziej spowalnia testowanie. Użyteczną techniką, która w tym pomaga, jest zastąpienie zależności atrapami (test doubles).

Istnieją różne typy atrapy: atrapy mock, fake i stub. Atrapa zastępuje zależność wymaganą przez komponent, dzięki czemu można testować go w izolacji. Każdy z tych terminów bywa używany w inny sposób przez różnych ludzi. Ja uznałem za użyteczne definicje, które znalazłem w książce Gerarda Meszarosa *xUnit Test Patterns* (Addison-Wesley)¹⁰.

W kontekście infrastruktury występuje rosnąca liczba narzędzi umożliwiających makietowanie API dostawców chmur¹¹. Dzięki nim można stosować kod infrastruktury w lokalnej makiecie chmury w celu przetestowania niektórych jego aspektów. Makietę nie powie nam, czy nasze struktury sieciowe działają poprawnie, ale powinna dać odpowiedź, czy są z grubsza prawidłowe.

Często użycie atrap dla różnych komponentów infrastruktury jest bardziej przydatne, niż dla samej platformy infrastruktury. W rozdziale 9 są podane przykłady używania atrapy i warunków początkowych testu (test fixtures) do testowania stosów infrastruktury (patrz „Używanie warunków początkowych testu do obsługi zależności” na stronie 132). Rozdziały w części IV zawierają opis podziału infrastruktury na mniejsze elementy i ich integracji. Warunki początkowe testu są kluczowym narzędziem do zapewniania luźnego sprzężenia komponentów.

Testowanie progresywne

Większość nietrywialnych systemów potrzebuje wielu zestawów testów do sprawdzania zmian. Poszczególne zestawy pozwalają testować różne aspekty (patrz „Co należy testować w przypadku infrastruktury?” na stronie 102). Jeden zestaw może testować jakiś problem w trybie offline, na przykład szukać luk w zabezpieczeniach, skanując składnię kodu.

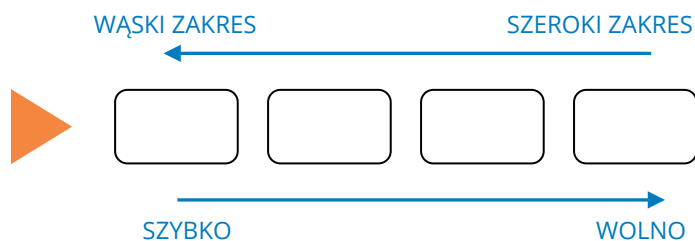
10 Przydatne informacje o atrapach są na blogu bliki Martina Fowlera „Mocks Aren't Stubs” (<https://oreil.ly/SvTx1>).

11 Przykładami narzędzi i bibliotek do makietowania chmur są Localstack (<https://oreil.ly/8Quwj>) i moto (<https://oreil.ly/VEWnf>). Do Better As Code (<https://oreil.ly/LGaYA>) zawiera aktualną listę tego typu narzędzi.

Inny zestaw może sprawdzać ten sam problem w trybie online, na przykład szukać luk w zabezpieczeniach sondując uruchomioną instancję stosu infrastruktury.

Testowanie progresywne polega na uruchamianiu testów w postaci sekwencji. Sekwencja zaczyna się od prostych testów, działających szybko na mniejszych fragmentach kodu, a potem rozrasta się do bardziej kompleksowych testów, obejmujących szerszy zbiór zintegrowanych komponentów i usług. Modele testowania, takie jak piramida testów i ser szwajcarski, pomagają ustalić, jak zorganizować działania weryfikacyjne w różnych zestawach testów.

Naczelną zasadą strategii progresywnej jest zapewnienie szybkiej i dokładnej informacji zwrotnej. Z reguły oznacza to uruchamianie na początku szybkich testów, obejmujących wąwszy zakres i mniej zależności, a następnie stopniowe dodawanie komponentów i punktów integracji (rysunek 8-1). W ten sposób drobne błędy są szybko wychwytywane, a to pozwala natychmiast je naprawiać i ponownie testować.



Rysunek 8-1 Zakres a szybkość testowania progresywnego

Gdy test o szerokim zakresie kończy się niepowodzeniem, trzeba zbadać wiele komponentów i zależności. Należy więc starać się jak najwcześniej znaleźć potencjalny obszar błędu, aby był on jak najmniejszy.

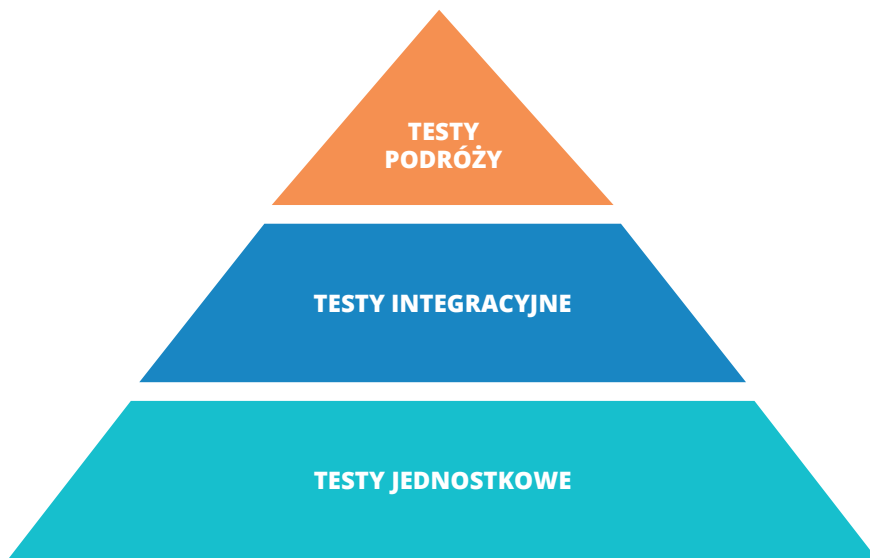
Innym celem strategii testowania jest zapewnienie możliwości zarządzania całym zestawem testów. Należy unikać powielania testów na różnych poziomach. Na przykład można sprawdzać, czy kod konfiguruje serwer aplikacji ustawia prawidłowe uprawnienia katalogów w folderze dziennika. Taki test powinien być uruchamiany na wczesnym etapie, poświęconym jawnemu testowaniu konfiguracji serwera. Nie należy stosować testu sprawdzającego uprawnienia do plików na etapie testowania całego stosu infrastruktury udostępnionego w chmurze.

Piramida testów

Piramida testów jest dobrze znanym modelem testowania oprogramowania¹². Główną ideą piramidy testów jest większa liczba testów w niższych warstwach modelu, stanowiących wczesne etapy testowania progresywnego i mniejsza liczba testów w wyższych warstwach, stanowiących końcowe etapy (patrz rysunek 8-2).

¹² Dokładne omówienie zawiera książka „The Practical Test Pyramid” Hama Vocke’a (<https://oreil.ly/oV266>).

Piramida została opracowana na potrzeby tworzenia oprogramowania. Dolna warstwa piramidy składa się z testów jednostkowych, z których każdy sprawdza mały fragment kodu i działa bardzo szybko¹³. Środkowa warstwa zawiera testy integracyjne, z których każdy obejmuje kolekcję zebranych razem komponentów. W górnej warstwie są testy podróży, oparte na interfejsie użytkownika, testujące aplikację jako całość.



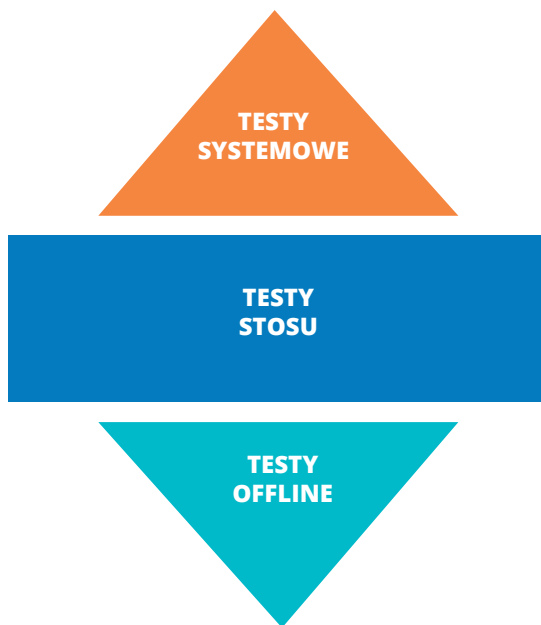
Rysunek 8-2 *Klasyczna piramida testów*

Testy na wyższych poziomach piramidy obejmują zakres objęty już przez testy na niższych poziomach. Oznacza to, że mogą być one mniej kompleksowe – muszą jedynie testować funkcjonalność wynikającą z integracji komponentów, a nie dowodzić prawidłowego działania komponentów niższego poziomu.

Piramida testów ma mniejszą wartość w przypadku deklaracyjnych baz kodu infrastruktury. Większość deklaracyjnego kodu stosu niskiego poziomu (patrz „Języki infrastruktury niskiego poziomu” na stronie 50) napisanego dla takich narzędzi, jak Terraform i CloudFormation, jest za długa do testowania jednostkowego i zależna od platformy infrastruktury. Moduły deklaratywne (patrz „Ponowne używanie kodu deklaracyjnego za pomocą modułów” na stronie 266) są trudne do testowania w sensowny sposób, zarówno z powodu małej wartości testowania kodu deklaracyjnego (patrz „Wyzwanie: testy kodu deklaracyjnego mają często małą wartość” na stronie 105), jak i dlatego, że zazwyczaj niewiele można przetestować bez infrastruktury.

¹³ Zobacz definicję testów jednostkowych ExtremeProgramming.org (<https://oreil.ly/EoSH6>). Definicja UnitTest na blogu bliki Martina Fowlera (<https://oreil.ly/kW14x>) zawiera omówienie kilku sposobów myślenia o testach jednostkowych.

To oznacza, że choć testy infrastruktury niskiego poziomu są stosowane w praktyce niemal zawsze, może nie być ich tak wiele, jak sugeruje model piramidy. Dlatego zestaw testów infrastruktury dla infrastruktury deklaratywnej może wyglądem przypominać bardziej diament, jak to jest pokazane na rysunku 8-3.



Rysunek 8-3 *Diament testów infrastruktury*

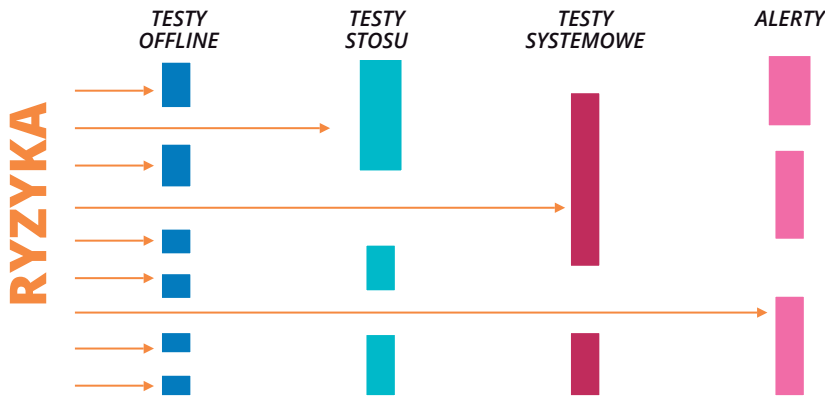
Piramida może być bardziej odpowiednia w przypadku bazy kodu infrastruktury w większym stopniu wykorzystującej biblioteki dynamiczne (patrz „Dynamiczne tworzenie elementów stosu za pomocą bibliotek” na stronie 267) napisane w językach imperatywnych (patrz „Programowalne, imperatywne języki infrastruktury” na stronie 39). Takie bazy kodu mają więcej małych komponentów, które dają zmienne wyniki, co oznacza więcej do testowania.

Model testowania „ser szwajcarski”

Innym sposobem myślenia o tym, jak zorganizować testy progresywne, jest model sera szwajcarskiego¹⁴. Jest to koncepcja zarządzania ryzykiem wywodząca się spoza branży oprogramowania. Pomysł jest taki, że dana warstwa testowania może mieć dziury, jak plaster sera szwajcarskiego, umożliwiające przegapienie jakiegoś defektu lub ryzyka. Ale łącząc wiele warstw otrzymujemy już coś w rodzaju bloku sera szwajcarskiego, w którym żadna dziura nie przechodzi na wylot.

¹⁴ <https://oreil.ly/XExp1>

Celem używania modelu sera szwajcarskiego, gdy myślimy o testowaniu infrastruktury, jest skupienie się na tym, żeby wyłapać dowolne zagrożenie (patrz rysunek 8-4). Nadal chcemy wykrywać problemy w najniższej warstwie, w której można to zrobić, ale ważne jest, aby w ogóle doszło do odpowiedniego testowania gdzieś w całym modelu.



Rysunek 8-4 Model testowania „ser szwajcarski”

Kluczowym wnioskiem jest zasada testowania opartego na ryzyku, a nie na dopasowaniu formuły.

Potoki dostarczania infrastruktury

Potok CD łączy implementację testowania progresywnego z dostarczaniem kodu w różnych środowiskach w drodze do wersji produkcyjnej¹⁵. Rozdział 19 zawiera szczegółowe informacje o tym, jak za pomocą potoków można pakować, integrować i stosować kod do środowisk. W tej części wyjaśnię, jak zaprojektować potok do testowania progresywnego.

Gdy ktoś przekazuje zmianę kodu do repozytorium kodu źródłowego, zespół używa centralnego systemu, gdzie zmiana przechodzi szereg etapów w celu jej przetestowania i dostarczenia. Proces ten jest zautomatyzowany, chociaż wyzwalanie i zatwierdzanie działań może odbywać się przy udziale ludzi.

Potok automatyzuje procesy związane z pakowaniem, promowaniem i stosowaniem kodu oraz testów. Ludzie mogą przeglądać zmiany, a nawet przeprowadzać testowanie eksploracyjne w środowiskach. Nie powinni jednak wykonywać ręcznie poleceń w celu wdrażania i stosowania zmian. Nie powinni również wybierać opcji konfiguracyjnych

¹⁵ Sam Newman opisał koncepcję potoków budowania w kilku postach na blogu począwszy od 2005 roku i podsumował je w 2009 roku w poście „A Brief and Incomplete History of Build Pipelines” (<https://oreil.ly/hm75g>). Potoki spopularyzowała książka Jeza Humble’a i Dave’a Farley’a „Continuous Delivery” (wspominana wcześniej w tym rozdziale). Jez udokumentował wdrożenie wzorca potoku w swojej witrynie (<https://oreil.ly/lGvIn>).

ani podejmować w locie innych decyzji. Działania te powinny być zdefiniowane jako kod i wykonywane przez system. Automatyzacja procesu oznacza jego spójną realizację za każdym razem i na każdym etapie. Takie postępowanie zwiększa niezawodność testów i zapewnia spójność instancji infrastruktury.

Każda zmiana powinna być wypychana od początku potoku. W przypadku wykrycia błędu w „dolnym” (dalszym) etapie potoku, nie należy naprawiać go na tym etapie i kontynuować pozostałej części potoku. Zamiast tego należy naprawić kod w repozytorium i wypchnąć nową zmianę od początku potoku, jak to jest pokazane na rysunku 8-5. Taka praktyka zapewnia pełne przetestowanie każdej zmiany.



Rysunek 8-5 Aby poprawić błąd, należy uruchomić nowy przebieg potoku

Na rysunku jedna zmiana pomyślnie przechodzi przez potok. Druga zmiana kończy się niepowodzeniem w środku potoku. Wykonana poprawka jest wypychana do produkcji przez trzeci przebieg potoku.

Etapy potoku

Każdy etap potoku może wykonywać różne działania i być wyzwalany na różne sposoby. Oto niektóre cechy charakterystyczne pojedynczego etapu potoku:

Wyzwalacz

Zdarzenie powodujące rozpoczęcie wykonywania etapu. Uruchomienie może następować automatycznie na skutek przesłania zmiany do repozytorium kodu albo pomyślnego wykonania poprzedniego etapu potoku. Ktoś może również wyzwolić etap ręcznie, na przykład, gdy osoba testująca lub menedżer wydania zdecyduje o zastosowaniu zmiany kodu do danego środowiska.

Działanie

To, co się dzieje podczas wykonywania etapu. Etap może powodować wykonanie wielu akcji. Na przykład może to być zastosowanie kodu w celu udostępnienia stosu infrastruktury, wykonanie testów, a następnie zniszczenie stosu.

Zatwierdzenie

Oznakowanie etapu – zaliczony lub niezaliczony. System może automatycznie oznaczać etap jako zaliczony (często mówi się „zielony”), gdy polecenia działają bez błędów i wszystkie zautomatyzowane testy przechodzą. Alternatywnie to człowiek musi

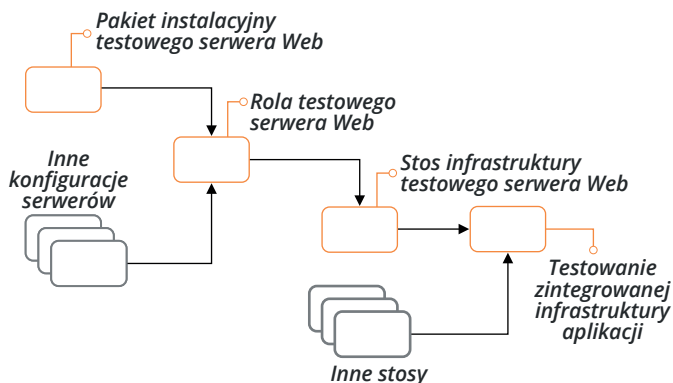
oznaczyć etap jako zatwierdzony. Na przykład osoba testująca może zatwierdzić etap po przeprowadzeniu testów eksploracyjnych dotyczących zmiany. Ręcznego zatwierdzania etapów można używać również do obsługi podpisów nadzoru.

Dane wyjściowe

Artefakty lub inne rzeczy wyprodukowane przez etap. Typowe dane wyjściowe obejmują pakiet kodu infrastruktury albo raport z testów.

Zakres komponentów testowanych w ramach etapu

W strategii testowania progresywnego na wcześniejszych etapach sprawdzane są poszczególne komponenty, a na późniejszych dokonuje się integracji komponentów i testuje je razem. Na rysunku 8-6 widać przykład progresywnego testowania komponentów, które prowadzi do serwera WWW działającego jako część większego stosu.

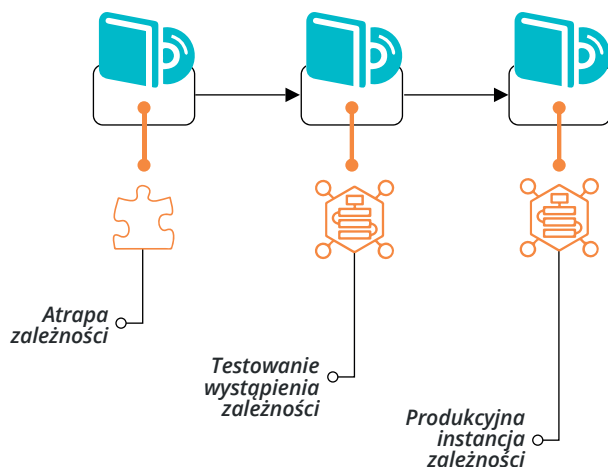


Rysunek 8-6 Stopniowe integrowanie i testowanie komponentów

Jeden etap może obejmować uruchamianie testów wielu komponentów, na przykład zestaw testów jednostkowych. Alternatywnie każdy komponent może mieć oddzielny etap testowania. W rozdziale 17 naszkicuję różne strategie integracji komponentów w kontekście stosów infrastruktury (patrz „Integrowanie projektów” na stronie 317).

Zakres zależności używanych w etapie

Wiele elementów systemu zależy od innych usług. Stos serwera aplikacji może łączyć się z usługą zarządzania tożsamościami w celu przeprowadzenia uwierzytelnienia użytkownika. Aby przetestować to progresywnie, można najpierw uruchomić etap z testami serwera aplikacji bez usługi zarządzania tożsamościami, na przykład używając zamiast niej atrapy. Na dalszym etapie zostałyby wykonane dodatkowe testy serwera aplikacji zintegrowanego z instancją testową usługi zarządzania tożsamościami, a etap produkcyjny byłby zintegrowany z instancją produkcyjną (rysunek 8-7).



Rysunek 8-7 Integracja progresywna z zależnościami



Stosuj tylko takie etapy, które wnoszą jakąś wartość

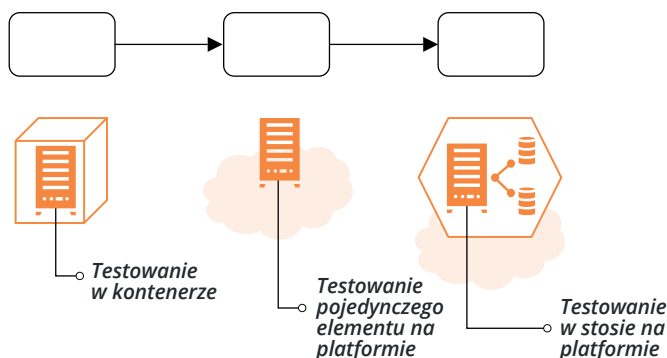
Należy unikać tworzenia niepotrzebnych etapów w potoku, gdyż każdy etap zwiększa czas i koszty procesu dostarczania. Dlatego nie można tworzyć oddzielnego etapu dla każdego komponentu i każdej integracji dla zapewnienia jednolitego obrazu. Taki podział testowania na etapy ma sens tylko wtedy, gdy wnosi wystarczająco dużo wartości, które uzasadniają dodatkowe nakłady. Niektóre powody, które mogą do tego skłonić, to szybkość, niezawodność, koszt i kontrola.

Elementy platformy wymagane przez etap

Usługi platformy są szczególnym rodzajem zależności dla tworzonego systemu. System może na koniec działać na platformie naszej infrastruktury, ale do uruchomienia i przetestowania jego fragmentów może wystarczyć tryb offline.

Na przykład kod, który definiuje struktury sieciowe, musi udostępnić te struktury na platformie chmurowej, żeby móc wykonać sensowne testy. Ale kod instalujący pakiet serwera aplikacji można zwykle przetestować w lokalnej maszynie wirtualnej, a nawet w kontenerze, bez konieczności stawiania maszyny wirtualnej na platformie chmurowej.

Dlatego wcześniejsze etapy testowania mogą być wykonywane dla niektórych komponentów bez użycia pełnej platformy chmurowej (rysunek 8-8).



Rysunek 8-8 Progresywne korzystanie z elementów platformy

Usługi i oprogramowanie potoków dostarczania

Do utworzenia potoku potrzebne jest oprogramowanie lub hostowana usługa. System potoków musi umożliwiać kilka rzeczy:

- Pozwalać skonfigurować etapy potoku.
- Wyzwalać etapy w wyniku różnych działań, łącznie ze zautomatyzowanymi zdarzeniami i ręcznymi wyzwalaczami. Narzędzie powinno obsługiwać bardziej złożone relacje, takie jak *zwijanie* (jeden etap z wieloma etapami wejściowymi) i *rozwijanie* (jeden etap z wieloma etapami wyjściowymi).
- Obsługiwać wszystkie działania potrzebne w etapach, łącznie ze stosowaniem kodu infrastruktury i uruchamianiem testów. Powinna istnieć możliwość tworzenia działań niestandardowych, a nie tylko korzystania z gotowego, zamkniętego zestawu.
- Obsługiwać artefakty i inne dane wyjściowe etapów, w tym także przekazywać je z jednego etapu do następnego.
- Pomagać śledzić i korelować konkretne wersje i instancje kodu, artefaktów, danych wyjściowych oraz infrastruktury.

Dostępnych jest kilka opcji systemu potoków:

Serwer budowy

Wiele zespołów do tworzenia potoków używa serwera budowy, takiego jak Jenkins¹⁶, Team City¹⁷, Bamboo¹⁸ czy GitHub Actions¹⁹. Są to zwykle potoki „zorientowane na zadanie”, a nie „zorientowane na strumień”. Podstawowy projekt z założenia nie koreluje wersji kodu ani artefaktów i wykonuje się w kilku zadaniach. Większość tych produktów dodała obsługę potoków jako nakładkę w interfejsie użytkownika i konfiguracji.

¹⁶ <https://jenkins.io>

¹⁷ <https://oreil.ly/Ez5Zv>

¹⁸ https://oreil.ly/VipT_

¹⁹ <https://oreil.ly/nDqlF>

Oprogramowanie CD

Oprogramowanie CD jest zbudowane wokół koncepcji potoku. Każdy etap jest definiowany jako część potoku, a wersje kodu i artefakty są skojarzone z potokiem, więc można je śledzić do przodu i do tyłu. Narzędzia CD obejmują GoCD²⁰, ConcourseCI²¹ oraz BuildKite²².

Usługi SaaS

Do hostowanych usług CI i CD należą CircleCI²³, TravisCI²⁴, AppVeyor²⁵, Drone²⁶ i BoxFuse²⁷.

Usługi platformy chmurowej

Większość dostawców chmury dysponuje usługami CI i CD, w tym AWS CodeBuild²⁸ (CI) i AWS CodePipeline²⁹ (CD) oraz Azure Pipelines³⁰.

Usługi repozytorium kodu źródłowego

Wiele produktów i wielu dostawców repozytorium kodu źródłowego dodało obsługę CI, umożliwiającą tworzenie potoków. Dwa wiodące przykłady to GitHub Actions³¹ oraz GitLab CI/CD³².

Wszystkie wymienione tutaj produkty zostały zaprojektowane z myślą o aplikacjach. Można używać większości z nich do tworzenia potoków dla infrastruktury, ale może to wymagać dodatkowej pracy.

W chwili, gdy to piszę, zaczynają się pojawiać produkty i usługi zaprojektowane dla infrastruktury jako kodu. Jest to szybko zmieniający się obszar, więc podejrzewam, że to, co mam do powiedzenia na temat tych narzędzi, będzie już nieaktualne, gdy książka trafi do rąk czytelnika i zabraknie w niej nowszych narzędzi. Ale warto przyjrzeć się temu, co istnieje teraz, aby zapewnić kontekst do oceny narzędzi w miarę ich pojawiania się i ewolucji:

- Atlantis³³ to produkt, który pomaga zarządzać żądaniami ściągnięcia w projektach Terraform i uruchamiać polecenia plan i apply dla pojedynczych instancji. Atlantis

20 W celu pełnego ujawnienia informacji, mój pracodawca, firma ThoughtWorks, stworzyła GoCD. Wcześniej był to produkt komercyjny, ale teraz to w pełni open source.

21 <https://concourse-ci.org>

22 <https://buildkite.com>

23 <https://circleci.com>

24 <https://travis-ci.org>

25 <https://www.appveyor.com>

26 <https://drone.io>

27 <https://boxfuse.com>

28 <https://oreil.ly/010rL>

29 <https://oreil.ly/5n2tp>

30 https://oreil.ly/rkJO_

31 <https://oreil.ly/I35Js>

32 <https://oreil.ly/fOrdb>

33 <https://oreil.ly/YMPuO>

nie uruchamia testów, ale może służyć do tworzenia ograniczonych potoków obsługujących przeglądanie kodu i zatwierdzanie zmian infrastruktury.

- Terraform Cloud³⁴ szybko się rozwija. Jest to usługa specyficzna dla Terraform, obejmująca więcej funkcji (takich jak rejestr modułów) niż CI i potoki. Terraform Cloud można używać do tworzenia ograniczonych potoków, planujących i stosujących kod projektu do wielu środowisk. Chmura nie wykonuje jednak testów innych niż walidacje zasad za pomocą produktu Sentinel firmy HashiCorp³⁵.
- Firma WeaveWorks³⁶ tworzy produkty i usługi do zarządzania klastrami Kubernetes. Są wśród nich narzędzia do zarządzania dostarczaniem zmian do konfiguracji klastra, jak również aplikacje wykorzystujące potoki oparte na gałęziach Git, podejście nazwane metodologią GitOps³⁷. Nawet jeśli ktoś nie używa narzędzi WeaveWorks, warto obserwować ten rozwojowy model. Opowiem o tym nieco więcej w ramce „GitOps” na stronie 343.

Testowanie w środowisku produkcyjnym

Testowanie wydań i zmian przed zastosowaniem ich do środowiska produkcyjnego ma wielką wagę w naszej branży. U jednego klienta naliczyłem osiem grup, które musiały przeglądać i zatwierdzać wydania, nie mówiąc o różnych zespołach technicznych, wykonujących zadania związane z instalacją i konfiguracją różnych części systemu³⁸.

W miarę wzrostu złożoności i skali systemów zmniejsza się zakres zagrożeń, które można potencjalnie sprawdzić poza środowiskiem produkcyjnym. Nie oznacza to, że testowanie zmian przed ich zastosowaniem do produkcji jest bezwartościowe. Ale wiara w to, że testowanie poprzedzające wydanie może objąć wszystkie zagrożenia, prowadzi do:

- Nadmiernego inwestowania w testowanie wydań wstępnych, daleko poza punkt, po którym jego wartość zaczyna maleć
- Niedostateczne inwestowanie w testowanie w środowisku produkcyjnym



Więcej informacji o testowaniu w środowisku produkcyjnym

Aby uzyskać więcej informacji o testowaniu w środowisku produkcyjnym, polecam obejrzeć wykład Charity Majors „Yes, I Test in Production (And So Should You)”³⁹, będący kluczowym źródłem moich rozważań na ten temat.

³⁴ <https://oreil.ly/Gqlmf>

³⁵ <https://oreil.ly/s1fkv>

³⁶ <https://www.weave.works>

³⁷ <https://oreil.ly/GlpZY>

³⁸ Były to grupy ds.: zarządzania zmianami, bezpieczeństwem informacji, zarządzania ryzykiem, zarządzania usługami, zarządzania przejściem, testowania integracji systemów, akceptacji użytkowników plus nadzór techniczny.

³⁹ <https://oreil.ly/exX7g>

Czego nie można powielić poza środowiskiem produkcyjnym

Jest kilka cech środowiska produkcyjnego, których nie da się realistycznie odtworzyć poza tym środowiskiem:

Dane

System produkcyjny może zawierać większe ilości danych niż da się wcześniej replikować, a do tego na pewno będą w nim niespodziewane wartości i kombinacje danych (dziękujemy użytkownikom!).

Użytkownicy

Ze względu na samą ich liczbę użytkownicy są dużo bardziej pomysłowi w tworzeniu dziwnych rzeczy, niż zespół testujący.

Ruch

Jeśli system ma do czynienia z nietrywialnym poziomem ruchu, nie da się zreplikować liczby i rodzajów aktywności, które regularnie będą miały miejsce. Tygodniowy test zanurzeniowy ma się nijak do rocznego działania w środowisku produkcyjnym.

Współbieżność

Narzędzia testowe mogą emulować wielu użytkowników korzystających z systemu jednocześnie, ale nie potrafią replikować nietypowych kombinacji działań, które użytkownicy wykonują współbieżnie.

Dwa wyzwania, które wynikają z tych cech, są takie, że generują one zagrożenia, których nie można przewidzieć oraz tworzą warunki, których nie można powielić na tyle dobrze, żeby wykonać testy w innym środowisku niż produkcyjne.

Uruchamiając testy w środowisku produkcyjnym wykorzystujemy warunki, które tam panują – duże ilości naturalnych danych i nieprzewidywalne działania współbieżne.



Po co testować gdziekolwiek indziej niż w środowisku produkcyjnym?

Oczywiście testowanie w środowisku produkcyjnym nie zastępuje testowania zmian przed zastosowaniem ich do środowiska produkcyjnego. Dobrze jest wyjaśnić sobie wcześniej, co można realnie (i należy) przetestować wcześniej:

- Czy to działa?
- Czy mój kod się wykonuje?
- Czy działa błędnie w sposób, który można przewidzieć?
- Czy działa błędnie w sposób, w jaki działał już błędnie wcześniej?

Testowanie zmian przed produkcją dotyczy *znanych niewiadomych*, rzeczy, o których wiemy, że mogą pójść źle. Testowanie zmian w środowisku produkcyjnym dotyczy *nieznanych niewiadomych*, czyli bardziej nieprzewidywalnych zagrożeń.

Zarządzanie ryzykiem testowania w środowisku produkcyjnym

Testowanie w środowisku produkcyjnym stwarza nowe zagrożenia. Jest kilka rzeczy, które pomagają zarządzać tymi zagrożeniami:

Monitorowanie

Efektywne monitorowanie daje pewność, że da się wykryć problemy powodowane przez testy, dzięki czemu będzie je można szybko zatrzymać.

Obserwowalność

Obserwowalność zapewnia wgląd w to, co się dzieje w systemie na poziomie szczegółów, a to pomaga szybko wykrywać i usuwać problemy, jak również poprawia jakość możliwych testów⁴⁰.

Wdrażanie bez przestojów

Możliwość szybkiego wdrażania i cofania zmian pomaga zmniejszyć ryzyko błędów (patrz „Zmiana działającej infrastruktury” na stronie 360).

Wdrażanie progresywne

Jeśli można uruchomić jednocześnie dwie różne wersje komponentów albo mieć różne konfiguracje dla różnych zbiorów użytkowników, to można testować zmiany w warunkach środowiska produkcyjnego przed udostępnieniem ich ogółowi użytkowników (patrz „Zmiana działającej infrastruktury” na stronie 360).

Zarządzanie danymi

Testy produkcyjne nie powinny wprowadzać niewłaściwych zmian w danych ani ujawniać danych wrażliwych. Można utrzymywać rekordy danych testowych, takich jak użytkownicy i numery kart kredytowych, które nie będą wyzwać działań w świecie rzeczywistym.

Inżynieria chaosu

Należy zmniejszać ryzyko w środowiskach produkcyjnych poprzez celowe wstrzykiwanie znanych rodzajów błędów, aby udowodnić, że posiadane systemy ograniczania ryzyka działają prawidłowo (patrz „Inżynieria chaosu” na stronie 371).

⁴⁰ Chociaż często wiąże się ją z monitorowaniem, obserwowalność polega na dawaniu ludziom możliwości zrozumienia tego, co dzieje się w środku systemu. Patrz „Introduction to Observability” firmy Honeycomb (<https://oreil.ly/OVXNF>).



Monitorowanie jako testowanie

Monitorowanie może być postrzegane jako pasywne testowanie w środowisku produkcyjnym. Nie jest to prawdziwe testowanie, ponieważ nie wykonujemy żadnych działań ani nie sprawdzamy wyniku. Zamiast tego obserwujemy naturalną aktywność użytkowników i szukamy niepożądanych rezultatów.

Monitorowanie powinno stanowić część strategii testowania, ponieważ jest częścią działań wykonywanych w celu zarządzania zagrożeniami dla naszego systemu.

Podsumowanie

W tym rozdziale omówiłem ogólne wyzwania i podejścia do testowania infrastruktury. Unikałem zbytniego zagłębiania się w tematykę testowania, jakości i zarządzania zagrożeniami. Jeśli dla kogoś nie są to dziedziny, w których ma duże doświadczenie, ten rozdział mógł stanowić dobry początek. Zachęcam do dalszej lektury, gdyż testowanie i zapewnianie jakości (QA) stanowią podstawę infrastruktury jako kodu.

Testowanie stosów infrastruktury

W tym rozdziale opisuję stosowanie praktyki ciągłego testowania i dostarczania kodu do stosów infrastruktury. Na przykładzie zespołu ShopSpinner pokażę, jak testować projekt stosu. Będzie to obejmować używanie etapów testowania w trybie online i offline oraz wykorzystywanie atrap do oddzielania stosu od zależności.

Przykładowa infrastruktura

Zespół ShopSpinner wykorzystuje projekty stosu wielokrotnego użytku (patrz „Wzorzec: stos wielokrotnego użytku” na stronie 65) do tworzenia spójnych instancji infrastruktury aplikacji dla każdego ze swoich klientów. Może również w ten sam sposób tworzyć testowe instancje infrastruktury w potoku.

Infrastruktura w tych przykładach to standardowy system trójwarstwowy. W skład każdej warstwy wchodzi:

Klaster kontenerów serwerów WWW

Zespół uruchamia jeden klaster kontenerów serwerów WWW dla każdego regionu i w każdym środowisku testowym. Aplikacje w regionie lub środowisku współużytkują ten klaster. Przykłady w tym rozdziale skupiają się na infrastrukturze, która jest specyficzna dla każdego klienta, a nie na infrastrukturze współdzielonej. Dlatego klaster współdzielony jest w tych przykładach zależnością. Szczegółowe informacje o koordynowaniu zmian i testowaniu w ramach tej infrastruktury są podane w rozdziale 17.

Serwer aplikacji

Infrastruktura dla każdej instancji aplikacji obejmuje maszynę wirtualną, wolumin dysku stałego oraz sieć. Sieć obejmuje blok adresów, bramę, trasy do serwera na jego porcie sieciowym oraz reguły dostępu do sieci.

Baza danych

ShopSpinner uruchamia oddzielną instancję bazy danych dla każdej instancji aplikacji klienta, korzystając z DBaaS swojego dostawcy (patrz „Zasoby pamięci masowej” na stronie 27). Kod infrastruktury zespołu ShopSpinner definiuje ponadto blok adresów, routing oraz uwierzytelnianie i reguły dostępu do bazy danych.

Przykładowy stos

Na początek możemy zdefiniować pojedynczy stos wielokrotnego użytku, obejmujący całą infrastrukturę oprócz klastra serwerów WWW. Struktura projektu może wyglądać, jak w przykładzie 9-1.

Przykład 9-1 Projekt stosu dla aplikacji klienta ShopSpinner

```
stack-project/  
└─ src/  
    ├── appserver_vm.infra  
    ├── appserver_networking.infra  
    ├── database.infra  
    └─ database_networking.infra
```

Wewnątrz tego projektu plik *appserver_vm.infra* zawiera kod zbliżony do tego, co jest pokazane w przykładzie 9-2.

Przykład 9-2 Fragment zawartości pliku *appserver_vm.infra*

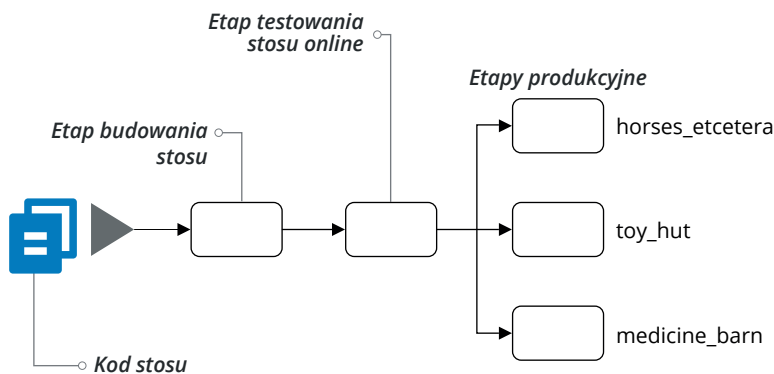
```
virtual_machine:  
  name: appserver-${customer}-${environment}  
  ram: 4GB  
  address_block: ADDRESS_BLOCK.appserver-${customer}-${environment}  
  storage_volume: STORAGE_VOLUME.app-storage-${customer}-${environment}  
  base_image: SERVER_IMAGE.shopspinner_java_server_image  
  provision:  
    tool: servermaker  
    parameters:  
      maker_server: maker.shopspinner.xyz  
      role: appserver  
      customer: ${customer}  
      environment: ${environment}  
  
storage_volume:  
  id: app-storage-${customer}-${environment}  
  size: 80GB  
  format: xfs
```

Członek zespołu lub zautomatyzowany proces może tworzyć albo aktualizować instancję stosu, uruchamiając narzędzie stosu. Przekazywanie wartości do instancji odbywa się przy użyciu jednego z wzorców opisanych w rozdziale 7.

Zgodnie z opisem w rozdziale 8 zespół wykorzystuje wiele etapów testowania („Testowanie progresywne” na stronie 109), zorganizowanych w sekwencyjny potok („Potoki dostarczania infrastruktury” na stronie 113).

Potok dla przykładowego stosu

Prosty potok dla stosu infrastruktury aplikacji ShopSpinner ma dwa etapy testowania¹, po których następuje etap, w którym kod jest stosowany do środowiska produkcyjnego każdego klienta (patrz rysunek 9-1).



Rysunek 9-1 Uproszczony przykładowy potok dla stosu

Pierwszym etapem potoku jest etap budowy stosu. *Etap budowy* w przypadku aplikacji obejmuje zwykle kompilację kodu, uruchomienie testów jednostkowych (opisanych w podrozdziale „Piramida testów” na stronie 110) i budowę artefaktu, który można wdrożyć. Więcej szczegółów na temat typowego etapu budowy dla kodu infrastruktury jest podanych w podrozdziale „Budowanie projektu infrastruktury” na stronie 314. Ponieważ wcześniejsze etapy potoku powinny być wykonywane szybciej, pierwszy etap jest zwykle używany do uruchamiania testów offline. Drugi etap przykładowego potoku obejmuje uruchomienie testów online dla projektu stosu. Każdy z etapów potoku może obejmować uruchomienie więcej niż jednego zestawu testów.

Etapy testowania offline dla stosów

Etap offline jest wykonywany „lokalnie” na agencie węzła usługi uruchamiającej etap (patrz „Usługi i oprogramowanie potoków dostarczania” na stronie 117), bez konieczności udostępniania infrastruktury na naszej platformie infrastruktury. Ścisłe testowanie offline odbywa się całkowicie w ramach lokalnego serwera lub instancji kontenera, bez łączenia się z jakimikolwiek usługami zewnętrznymi, takimi jak baza danych. Łagodniejsze testowanie offline może stosować połączenie z istniejącą instancją usługi, być może nawet

¹ Ten potok jest znacznie prostszy od tego, który byłby używany w rzeczywistości. Prawdopodobnie występowałyby jeszcze co najmniej jeden etap do testowania stosu i aplikacji razem (patrz „Dostarczanie infrastruktury i aplikacji” na stronie 308). Mogłby być także potrzebny etap testowania akceptacji przez klienta przed etapem produkcyjnym dla każdego klienta. Nie ma tu również etapów nadzoru i zatwierdzania, które bywają wymagane przez organizacje.

z API chmury, ale bez wykorzystywania jakiegokolwiek realnej infrastruktury stosu. Etap offline powinien:

- Wykonywać się szybko, dostarczając od razu informację zwrotną w przypadku nieprawidłowości
- Weryfikować poprawność komponentów w izolacji, aby zapewnić zaufanie do każdego z nich i uprościć debugowanie błędów
- Potwierdzać, że komponenty są dokładnie oddzielone

Niektóre testy, które można przeprowadzać na kodzie stosu podczas etapu, to sprawdzanie składni, statyczna analiza kodu w trybie offline, statyczna analiza kodu z użyciem API platformy oraz testowanie z użyciem atrap API.

Sprawdzanie składni

W przypadku większości narzędzi stosu można wykonywać polecenie *dry run*, które analizuje składnię kodu bez stosowania go do infrastruktury. Jeśli w kodzie jest błąd, polecenie kończy się błędem. Taka kontrola pozwala szybko zorientować się, gdzie jest jakaś literówka w kodzie zmiany, ale pomija wiele innych błędów. Przykładami narzędzi skryptowych do sprawdzania składni są `terraform validate` i `aws cloudformation validate-template`. Rezultat zakończonego niepowodzeniem sprawdzania składni może wyglądać następująco:

```
$ stack validate
```

```
Error: Invalid resource type
   on appserver_vm.infra line 1, in resource "virtual_mahcine":
   stack does not support resource type "virtual_mahcine".
```

Statyczna analiza kodu w trybie offline

Niektóre narzędzia potrafią analizować składnię i analizować kod źródłowy stosu pod kątem szerszej klasy problemów niż tylko składniowe, ale nadal bez łączenia się z platformą infrastruktury. Taka analiza nazywana jest często *lintingiem*². Tego rodzaju narzędzie może szukać błędów kodowania, mylącego lub złego stylu kodowania, niestosowania zasad stylu kodowania albo potencjalnych problemów zabezpieczeń. Niektóre narzędzia potrafią nawet modyfikować kod, aby był zgodny z określonym stylem, na przykład polecenie `terraform fmt`. Jednak narzędzi do analizowania kodu infrastruktury nie ma aż tylu, co w przypadku analizowania języków programowania aplikacji. Niektóre

² Termin *lint* (<https://oreil.ly/jul6X>) pochodzi od klasycznego uniksowego narzędzia do analizy kodu źródłowego C.

z nich, to [tflint](https://oreil.ly/ZBAng)³, [CloudFormation Linter](https://oreil.ly/lDuDV)⁴, [cfn_nag](https://oreil.ly/u2Hp1)⁵, [tfsec](https://oreil.ly/isSKE)⁶ oraz [checkov](https://www.checkov.io)⁷. Oto przykład błędu pochodzący z fikcyjnego narzędzia do analizy:

```
$ stacklint
1 issue(s) found:

Notice: Missing 'Name' tag (vms_must_have_standard_tags)
on appserver_vm.infra line 1, in resource "virtual_machine":
```

W tym przykładzie występuje niestandardowa reguła o nazwie `vms_must_have_standard_tags`, która wymaga, aby wszystkie maszyny wirtualne miały zestaw tagów obejmujący tag o nazwie `Name`.

Statyczna analiza kodu z użyciem API

W zależności od narzędzia niektóre elementy statycznej analizy kodu mogą wymagać łączenia się z API platformy chmurowej w celu sprawdzenia czy nie ma konfliktów z tym, co umożliwia platforma. Na przykład `tflint` może sprawdzać kod projektu Terraform, aby upewnić się, czy wszystkie typy instancji (rozmiary maszyn wirtualnych) albo AMI (obrazy serwerów) zdefiniowane w kodzie faktycznie istnieją. W odróżnieniu od podglądu zmian (patrz „Podgląd: patrzenie, jakie zmiany zostaną dokonane” na stronie 128), tego typu walidacja testuje kod w sposób ogólny, a nie pod kątem konkretnej instancji stosu na platformie.

Kolejny przykład przedstawia niepowodzenie, ponieważ kod deklarujący serwer wirtualny podaje obraz serwera, którego nie ma na platformie:

```
$ stacklint
1 issue(s) found:

Notice: base_image 'SERVER_IMAGE.shopspinner_java_server_image' doesn't exist
(validate_server_images)

on appserver_vm.infra line 5, in resource "virtual_machine":
```

Testowanie z użyciem atrapy API

Czasem można zastosować kod stosu do lokalnej atrapy instancji API platformy infrastruktury. Nie ma zbyt wielu narzędzi do makietowania API. Jedynym, o którego istnieniu wiem, gdy to piszę, jest `Localstack`⁸. Niektóre narzędzia potrafią makietować fragmenty

³ <https://oreil.ly/ZBAng>

⁴ <https://oreil.ly/lDuDV>

⁵ <https://oreil.ly/u2Hp1>

⁶ <https://oreil.ly/isSKE>

⁷ <https://www.checkov.io>

⁸ <https://oreil.ly/8Quwj>

platformy, na przykład emulator Azurite⁹, który emuluje magazyn kolejki i obiektów blob Azure.

Stosowanie deklaratywnego kodu stosu do lokalnej atrapy może ujawnić błędy kodowania, niewykryte podczas testowania składni i analizowania kodu. W praktyce testowanie kodu deklaratywnego z użyciem atrapy API platformy infrastruktury nie wnosi wiele wartości, z powodów omówionych w podrozdziale „Wyzwanie: testy kodu deklaratywnego mają często małą wartość” na stronie 105. Takie atrapy mogą być jednak przydatne w przypadku testów jednostkowych kodu imperatywnego (patrz „Programowalne, imperatywne języki infrastruktury” na stronie 39), zwłaszcza bibliotek (patrz „Dynamiczne tworzenie elementów stosu za pomocą bibliotek” na stronie 267).

Etapy testowania online dla stosów

Etap online polega na użyciu platformy infrastruktury do utworzenia instancji stosu i interakcji z nią. Tego typu etap jest wolniejszy, ale pozwala wykonać bardziej znaczące testy niż etap offline. Usługa potoku dostarczania uruchamia zwykle narzędzie stosu na jednym z jej węzłów lub agentów, ale do interakcji z instancją stosu wykorzystuje API platformy. Usługa wymaga uwierzytelnienia w API platformy (bezpieczne metody są opisane w podrozdziale „Obsługa wpisów tajnych jako parametrów” na stronie 96). Chociaż etap testowania online jest zależny od platformy infrastruktury, powinno być możliwe przetestowanie stosu z minimalną liczbą innych zależności. W szczególności należy tak projektować infrastrukturę i testy, aby można było zawsze utworzyć i przetestować instancję stosu bez konieczności jej integracji z instancjami innych stosów.

Na przykład infrastruktura aplikacji klienta ShopSpinner działa z użyciem współdzielonego stosu klastra serwerów WWW. Ale członkowie zespołu ShopSpinner do implementacji infrastruktury i etapów testowania używają technik, które pozwalają im testować kod stosu aplikacji bez instancji klastra serwerów WWW.

Techniki dzielenia stosów i usuwania silnego sprzężenia między nimi omówię w rozdziale 15. Zakładając, że zbudowaliśmy naszą infrastrukturę w taki właśnie sposób, możemy użyć warunków początkowych testu i testować sam stos, jak to jest opisane w podrozdziale „Używanie warunków początkowych testu do obsługi zależności” na stronie 132.

Najpierw zastanówmy się, jak działają różne rodzaje testów stosu online. Testy wykonywane w ramach etapu online obejmują: podgląd zmian, sprawdzanie, czy zmiany zostały zastosowane poprawnie oraz potwierdzanie wyników.

Podgląd: patrzeć, jakie zmiany zostaną dokonane

Niektóre narzędzia stosu potrafią porównać kod stosu z istniejącą instancją i utworzyć listę zmian, które można wprowadzić, ale bez ich faktycznej realizacji. Dobrze znanym przykładem jest polecenie Terraform *plan*.

⁹ https://oreil.ly/FazP_

Najczęściej ludzie używają podglądu zmian w instancjach produkcyjnych jako bezpiecznej metody przejrzania listy zmian w celu upewnienia się, że nie zdarzy się nic nieoczekiwanego. Stosowanie zmian do stosu w ramach potoku może mieć postać procesu dwuetapowego. W pierwszym etapie uruchamiany jest podgląd, a potem ktoś wyzwała drugi etap w celu zastosowania zmian, ale dopiero po przejrzaniu wyników podglądu.

Przeglądanie zmian przez ludzi nie jest wiarygodne. Człowiek może nie zrozumieć albo nie zauważyć problematycznej zmiany. Można napisać zautomatyzowany test do sprawdzania wyników polecenia podglądu. Tego typu test mógłby sprawdzać zmiany pod kątem zasad i kończyć się niepowodzeniem, gdyby na przykład kod miał utworzyć przestarzały typ zasobu. Albo mógłby szukać uciążliwych zmian i kończyć się niepowodzeniem, gdyby kod powodował odbudowę lub zniszczenie instancji bazy danych.

Inny problem polega na tym, że podglądy wykonywane przez narzędzia stosów zazwyczaj nie są zbyt głębokie. Podgląd powie nam, że poniższy kod utworzy nowy serwer:

```
virtual_machine:
  name: myappserver
  base_image: "java_server_image"
```

Jednak powyższy podgląd może nie powiedzieć nam, że obraz "java_server_image" nie istnieje, a więc polecenie apply nie utworzy serwera.

Podgląd zmian stosu przydaje się do sprawdzenia ograniczonego zbioru zagrożeń tuż przed zastosowaniem zmiany kodu do instancji. Jest za to mniej przydatny do testowania kodu, który ma być wielokrotnie używany do wielu instancji, na przykład środowisk testowych na potrzeby dostarczenia wydania. Zespoły wykorzystujące środowiska kopiuj-wklej (patrz „Antywzorzec: środowiska kopiuj-wklej” na stronie 63) często używają etapu podglądu jako minimalnego testu dla każdego środowiska. Natomiast zespoły wykorzystujące stosy wielokrotnego użytku (patrz „Wzorzec: stos wielokrotnego użytku” na stronie 65) mogą dokonywać bardziej znaczącej weryfikacji swojego kodu za pomocą instancji testowych.

Weryfikacja: stosowanie asercji dotyczących zasobów infrastruktury

Mając instancję stosu, można w ramach etapu online użyć testów z asercjami dotyczącymi infrastruktury w stosie. Oto kilka przykładowych środowisk do testowania zasobów infrastruktury:

- Awspec¹⁰
- Clarity¹¹
- Inspec¹²

¹⁰ <https://oreil.ly/V6H8K>

¹¹ <https://oreil.ly/wIJgE>

¹² <https://www.inspec.io>

- Taskcat¹³
- Terratest¹⁴

Zestaw testów dla maszyny wirtualnej z przykładowego kodu stosu zamieszczonego wcześniej w tym rozdziale mógłby wyglądać następująco:

```
given virtual_machine(name: "appserver-testcustomerA-staging") {
  it { exists }
  it { is_running }
  it { passes_healthcheck }
  it { has_attached_storage_volume(name: "app-storage-testcustomerA-staging") }
}
```

Większość narzędzi do testowania stosu udostępnia biblioteki, które ułatwiają pisanie asercji dla typów zasobów infrastruktury opisanych w rozdziale 3. Ten przykładowy test używa zasobu `virtual_machine` do identyfikacji maszyny wirtualnej w instancji stosu dla środowiska przejściowego. Zawiera kilka asercji dotyczących zasobu: czy został utworzony (`exists`), czy działa, a nie został zakończony (`is_running`) i czy platforma infrastruktury uważa go za poprawny (`passes_healthcheck`).

Proste asercje mają często małą wartość (patrz „Wyzwanie: testy kodu deklaratywnego mają często małą wartość” na stronie 105), ponieważ jedynie powtarzają kod testowanej infrastruktury. Kilka podstawowych asercji (takich jak `exists`) pomaga sprawdzić, czy kod został zastosowany pomyślnie. Pozwalają szybko zidentyfikować podstawowe problemy z konfiguracją etapu potoku lub skryptami konfiguracyjnymi testów. Testy, takie jak `is_running` i `passes_healthcheck` powiedzą nam, że narzędzie stosu pomyślnie utworzyło maszynę wirtualną, ale nastąpiła jej awaria lub pojawił się inny zasadniczy problem. Takie proste asercje oszczędzają czas podczas rozwiązywania problemów.

Chociaż można tworzyć asercje odzwierciedlające każdy element konfiguracji maszyny wirtualnej w kodzie stosu, jak ilość pamięci RAM lub przypisany do niej adres sieciowy, mają one małą wartość i powodują dodatkowe obciążenie.

Czwarta asercja w naszym przykładzie, `has_attached_storage_volume()`, jest bardziej interesująca. Sprawdza ona, czy wolumin pamięci masowej zdefiniowany w tym samym stosie jest dołączony do maszyny wirtualnej. Celem tego jest potwierdzenie, że kombinacja wielu deklaracji działa poprawnie (jak to zostało omówione w podrozdziale „Testowanie kombinacji kodu deklaratywnego” na stronie 107). W zależności od platformy i narzędzi kod stosu może zostać pomyślnie zastosowany, ale pozostawić serwer i wolumin niepołączone ze sobą. Albo w kodzie stosu może występować błąd psujący połączenie.

Innym przypadkiem, w którym asercje mogą być przydatne, jest dynamiczny kod stosu. Gdy przekazywanie różnych parametrów do stosu może powodować różne wyniki, warto czasem dodać asercje dotyczące tych wyników. Weźmy jako przykład poniższy kod, który tworzy infrastrukturę dla serwera aplikacji dostępnego publicznie lub wewnętrznie:

¹³ <https://oreil.ly/PLzOJ>

¹⁴ <https://oreil.ly/oes8O>

```

virtual_machine:
  name: appserver-${customer}-${environment}
  address_block:
    if(${network_access} == "public")
      ADDRESS_BLOCK.public-${customer}-${environment}
    else
      ADDRESS_BLOCK.internal-${customer}-${environment}
    end

```

Można mieć etap testowania tworzący instancję każdego typu z asercjami sprawdzającymi w obu przypadkach, czy konfiguracja sieciowa jest prawidłowa. Bardziej skomplikowane wersje należy umieszczać w modułach lub bibliotekach (patrz rozdział 16) i testować te moduły niezależnie od kodu stosu. Takie postępowanie upraszcza testowanie kodu stosu.

Asercje potwierdzające utworzenie zasobów infrastruktury zgodnie z oczekiwaniami są przydatne tylko do pewnego stopnia. Najcenniejsze testowanie to takie, które potwierdza, że zasoby działają tak, jak powinny.

Wyniki: potwierdzanie prawidłowego działania infrastruktury

Najważniejszą częścią testowania oprogramowania aplikacji jest testowanie funkcjonalne. Analogicznie, w przypadku infrastruktury można potwierdzać, że da się używać jej zgodnie z zamierzeniem. Oto przykłady wyników, które można testować dla kodu stosu infrastruktury:

- Czy można utworzyć połączenie sieciowe z segmentu sieciowego serwera WWW do segmentu sieciowego obsługującego aplikację na odpowiednim porcie?
- Czy można uruchomić i wdrożyć aplikację w instancji naszego stosu klastra kontenerów?
- Czy można bezpiecznie podłączyć ponownie wolumin pamięci masowej po odbudowaniu instancji serwera?
- Czy system równoważenia obciążenia prawidłowo obsługuje instancje serwerów podczas ich dodawania i usuwania?

Testowanie wyników jest bardziej skomplikowane niż weryfikowanie, czy coś istnieje. Testy wymagają nie tylko tworzenia i aktualizacji instancji stosu, co jest omówione w podrozdziale „Wzorce cyklu życia dla testowych instancji stosów” na stronie 136, ale czasem również warunków początkowych testu. Warunki początkowe testu są zasobem infrastruktury używanym wyłącznie do obsługi testu (warunki początkowe testu są omówione w podrozdziale „Używanie warunków początkowych testu do obsługi zależności” na stronie 132).

Poniższy test nawiązuje połączenie z serwerem, aby sprawdzić dostępność portu i zwraca oczekiwaną odpowiedź HTTP:


```

given stack_instance(stack: "shopspinner_networking",
                     instance: "online_test") {

  can_connect(ip_address: stack_instance.appserver_ip_address,
              port:443)

  http_request(ip_address: stack_instance.appserver_ip_address,
               port:443,
               url: '/').response.code is('200')
}

```

Środowisko testowania i biblioteki implementują szczegóły weryfikacji w rodzaju `can_connect` i `http_request`. Aby dowiedzieć się, jak pisać rzeczywiste testy, trzeba przeczytać dokumentację swojego narzędzia testowego.

Używanie warunków początkowych testu do obsługi zależności

Wiele projektów stosu zależy od zasobów utworzonych poza stosem, takich jak współdzielona sieć zdefiniowana w innym projekcie stosu. Warunki początkowe testu są zasobem infrastruktury tworzonym specjalnie po to, aby ułatwić udostępnienie i testowanie samej instancji stosu, bez udziału instancji pozostałych stosów. Atrapy wspomniane w poprzednim rozdziale „Wyzwanie: zależności komplikują testowanie infrastruktury” na stronie 109 są rodzajem warunków początkowych testu.

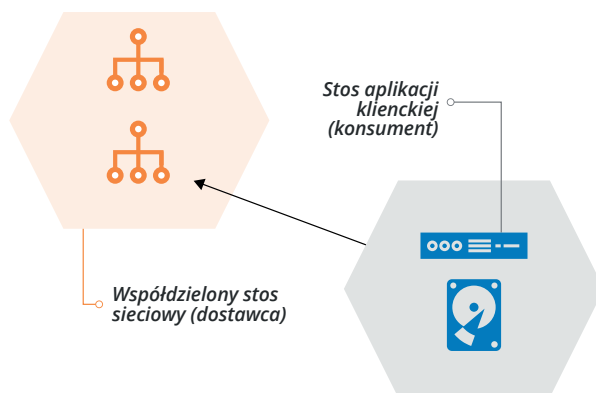
Używanie warunków początkowych testów pozwala znacznie łatwiej zarządzać testami, utrzymywać luźne sprzężenie między stosami i szybciej otrzymywać informacje zwrotne. Bez warunków początkowych testu zachodzi często konieczność utworzenia i utrzymywania skomplikowanych zestawów infrastruktury testowej.

Warunki początkowe testu nie są częścią testowanego stosu. Jest to dodatkowa infrastruktura, tworzona w celu obsługi testów. Warunki początkowe testu są używane do reprezentowania zależności stosu.

Dana zależność jest albo *nadrzędna*, co oznacza, że testowany stos używa zasobów udostępnianych przez inny stos, albo *podrzędna*, czyli taka, gdy inne stosy używają zasobów należących do testowanego stosu. Niektórzy nazywają stos z zależnościami podrzędnymi *dostawcą*, ponieważ dostarcza on zasoby. Stos z zależnościami nadrzędnymi jest czasem nazywany *konsumentem* (patrz rysunek 9-2).

W naszym przykładzie z ShopSpinner występuje stos dostawcy, który definiuje współdzielone struktury sieciowe. Struktury te są używane przez stosy konsumenckie, w tym stos definiujący infrastrukturę aplikacji konsumenta. Stos aplikacji tworzy serwer i przypisuje go do bloku adresów sieciowych¹⁵.

¹⁵ Rozdział 17 zawiera wyjaśnienie, jak łączyć zależności stosu.



Rysunek 9-2 Przykład stosu dostawcy i stosu konsumenta

Dany stos może być zarówno dostawcą, jak i konsumentem, konsumując zasoby z jednego stosu i dostarczając zasoby dla innych stosów. Używając warunków początkowych testu można zastępować zarówno punkty integracji nadrzędnej stosu, jak i podrzędnej.

Atrapy dla zależności nadrzędnych

Gdy trzeba przetestować stos zależny od innego stosu, można utworzyć atrapę (*test double*). W przypadku stosów oznacza to zwykle utworzenie pewnej dodatkowej infrastruktury. W naszym przykładzie współdzielonego stosu sieci i stosu aplikacji, stos aplikacji musi utworzyć swój serwer w bloku adresów sieciowych, zdefiniowanym przez stos sieci. Konfiguracja testu powinna umożliwiać utworzenie bloku adresów jako atrapy do przetestowania samego stosu aplikacji.

Utworzenie bloku adresów jako atrapy bywa lepszym rozwiązaniem niż utworzenie instancji całego stosu sieci. Stos sieci może obejmować dodatkową infrastrukturę, której testowanie nie jest konieczne. Na przykład może definiować zasady sieci, trasy, inspekcję i inne zasoby dla środowiska produkcyjnego, których testowanie jest przesadą.

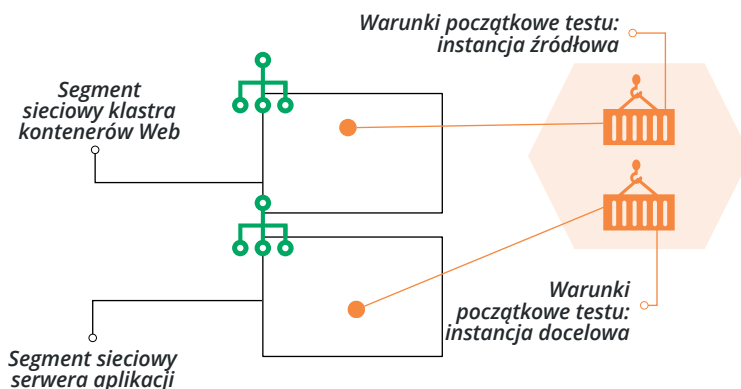
Ponadto utworzenie zależności jako warunków początkowych testu w projekcie stosu konsumentckiego oddziela go od stosu dostawczego. Jeśli ktoś pracuje nad zmianą projektu stosu sieci, nie oznacza to automatycznie pracy nad stosem aplikacji.

Potencjalną zaletą tego typu oddzielania jest uzyskiwanie stosów łatwiejszych do wielokrotnego użytku i składania. Zespół ShopSpinner może chcieć tworzyć różne projekty stosu sieci do różnych celów. Jeden stos może tworzyć ściśle kontrolowaną i objętą inspekcją sieć dla usług o bardziej rygorystycznych wymaganiach dotyczących zgodności, na przykład do przetwarzania płatności zgodnie ze standardem PCI¹⁶ lub regulacjami dotyczącymi ochrony danych konsumentów. Inny stos może tworzyć sieć, która nie musi być zgodna z PCI. Testując stosy aplikacji zespół ułatwia sobie zadanie, mogąc używać tylko jednego z nich.

¹⁶ <https://oreil.ly/MuWAM>

Warunki początkowe testu dla zależności podrzędnych

Warunków początkowych testu można również używać w odwrotnej sytuacji, do testowania stosu, który udostępnia zasoby innym stosom. Pokazana na rysunku 9-3 instancja stosu definiuje struktury sieciowe dla ShopSpinner, w tym segmenty i routing dla klastra kontenerów serwerów WWW i serwerów aplikacji. Stos sieci nie tworzy klastra kontenerów ani serwerów aplikacji, dlatego w celu przetestowania sieci konfiguracja inicjuje warunki początkowe testu dla każdego z tych segmentów.



Rysunek 9-3 Instancja testu stosu sieci ShopSpinner z warunkami początkowymi testu

Warunki początkowe testu w tych przykładach obejmują dwie instancje kontenera, po jednej przypisanej do każdego segmentu sieci w stosie. Do testowania wyników można często używać tych samych narzędzi testowania, co w przypadku testów weryfikacyjnych (patrz „Weryfikacja: stosowanie asercji dotyczących zasobów infrastruktury” na stronie 129). Poniższe przykładowe testy wykorzystują fikcyjny język DSL do testowania stosu:

```
given stack_instance(stack: "shopspinner_networking",
                     instance: "online_test") {

  can_connect(from: $HERE,
              to: get_fixture("web_segment_instance").address,
              port: 443)

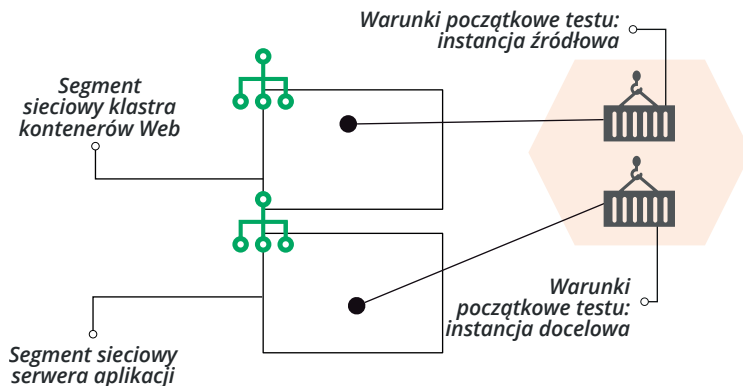
  can_connect(from: get_fixture("web_segment_instance"),
              to: get_fixture("app_segment_instance").address,
              port: 8443)

}
```

Metoda `can_connect` jest wykonywana z poziomu `$HERE`, gdzie `$HERE` może być agentem, na którym kod jest wykonywany, albo z instancji kontenera. Metoda próbuje nawiązać połączenie HTTPS na wskazanym porcie z adresem IP. Metoda `get_fixture()` pobiera szczegóły instancji kontenera utworzonej jako warunek początkowy testu.

Metoda `can_connect` może być udostępniana przez środowisko testowe albo może to być metoda niestandardowa, napisana przez zespół.

Połączenia nawiązane przez kod tego przykładowego testu są pokazane na rysunku 9-4.



Rysunek 9-4 Testowanie łączności w stosie sieci ShopSpinner

Na diagramie widać ścieżki dla obu testów. Pierwszy test łączy się spoza stosu z warunkiem początkowym testu w segmencie WWW. Drugi test łączy się z warunkiem początkowym w segmencie WWW z warunkiem początkowym w segmencie aplikacji.

Refaktoryzacja komponentów w celu umożliwienia ich izolowania

Czasami konkretnego komponentu nie można łatwo wyizolować. Zależności od innych komponentów mogą być zakodowane na stałe albo po prostu zbyt zagmatwane, aby je rozdzielić. Jedną z zalet pisania testów podczas projektowania i budowania systemów, a nie na końcu, jest wymuszanie w ten sposób lepszego projektowania. Komponent trudny do testowania w izolacji jest objawem wad projektu. Dobrze zaprojektowany system powinien zawierać luźno sprzężone komponenty.

Jeśli więc napotykamy komponent, który trudno wyizolować, należy poprawić projekt. Może okazać się konieczne ponowne napisanie całych komponentów albo wymiana bibliotek, narzędzi lub aplikacji. Jak to się mówi, jest to cecha, a nie błąd. Czytelny projekt i luźno sprzężony kod to efekt uboczny dbania o testowalność systemu.

Istnieje kilka strategii restrukturyzacji systemów. Refaktoryzację i inne techniki ulepszania architektury systemu opisuje Martin Fowler¹⁷. Na przykład Strangler Application¹⁸ nadaje priorytet utrzymywaniu w pełni działającego systemu podczas jego restrukturyzacji z biegiem czasu.

Bardziej szczegółowe reguły i przykłady modularyzacji i integracji infrastruktury są przedstawione w części IV tej książki.

¹⁷ <https://oreil.ly/3oSt8>

¹⁸ <https://oreil.ly/Lb7Zw>

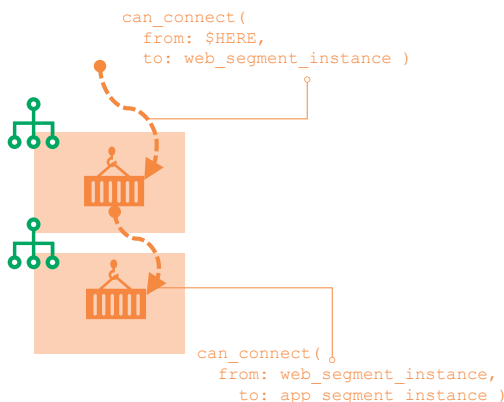
Wzorce cyklu życia dla testowych instancji stosów

Przed pojawieniem się wirtualizacji i chmury każdy utrzymywał statyczne, długotrwałe środowiska testowe. Chociaż wiele zespołów nadal ma takie środowiska, tworzenie i niszczenie środowisk na żądanie ma swoje zalety. W kolejnych przykładach zobaczymy wady i zalety utrzymywania trwałej instancji stosu, tworzenia efemerycznej instancji dla każdego uruchomienia testu oraz łączenia obu tych podejść. Wzorce te można stosować również do środowisk testowych aplikacji i całego systemu, a także do testowania kodu stosu infrastruktury.

Wzorzec: trwały stos testowy

Znany również jako: środowisko statyczne.

Etap testowania może wykorzystywać instancję *trwałego stosu testowego*, czyli stale działającą. Etap stosuje każdą zmianę kodu jako aktualizację istniejącej instancji stosu, uruchamia testy i pozostawia wynikowy, zmodyfikowany stos do następnego razu (patrz rysunek 9-5).



Rysunek 9-5 Instancja trwałego stosu testowego

Motywacja

Zazwyczaj znacznie szybciej można zastosować zmiany do istniejącej instancji stosu niż do takiej, którą trzeba wcześniej utworzyć. Z tego powodu trwały stos testowy zapewnia szybszą informację zwrotną, nie tylko w ramach samego etapu, ale całego potoku.

Zastosowanie

Trwały stos testowy jest przydatny, gdy można bez problemów stosować kod stosu do instancji. Jeśli jednak trzeba tracić czas na naprawianie uszkodzonych instancji w celu ponownego uruchomienia potoku, należy rozważyć inny wzorec z tego rozdziału.

Konsekwencje

Wcale nie tak rzadko dochodzi do „zaklinowania” instancji stosu, czyli sytuacji, w której zmiana kończy się niepowodzeniem i pozostawia stos w stanie, w którym każda następna próba zastosowania kodu stosu również kończy się niepowodzeniem. Często instancja potrafi się zaklinować tak poważnie, że narzędzie stosu nie może jej nawet zniszczyć, aby można ją było ponownie uruchomić. W efekcie zespół musi poświęcić bardzo dużo czasu na ręczne odblokowanie uszkodzonych instancji testowych.

W wielu przypadkach udaje się zmniejszyć częstotliwość klinowania się stosów, poprawiając projekt. Rozbijanie stosów na mniejsze i prostsze stosy oraz upraszczanie zależności między stosami pozwala ograniczyć występowanie zakleszczeń. Więcej na ten temat jest w rozdziale 15.

Implementacja

Wdrożenie trwałego stosu testowego jest proste. Etap potoku uruchamia polecenie narzędzia stosu, aby zaktualizować instancję przy użyciu odpowiedniej wersji kodu stosu, wykonuje testy i po ich zakończeniu pozostawia instancję stosu na jej miejscu.

Stos można całkowicie odbudować w ramach procesu ad hoc, na przykład uruchamiając narzędzie z lokalnego komputera lub używając dodatkowego etapu lub zadania spoza rutynowego przepływu potoku.

Powiązane wzorce

Wzorzec okresowej odbudowy stosu omówiony w podrozdziale „Wzorzec: okresowa odbudowa stosu” na stronie 140 jest prostą modyfikacją tego wzorca, polegającą na niszczeniu instancji pod koniec każdego dnia pracy i budowaniu rano kolejnej.

Wzorzec: efemeryczny stos testowy

Znany również jako: środowisko dynamiczne.

W przypadku wzorca *efemerycznego stosu testowego* etap testowania tworzy i niszczy nową instancję stosu za każdym razem, gdy musi ją uruchomić (patrz rysunek 9-6).

Motywacja

Efemeryczny stos testowy zapewnia czyste środowisko dla każdego uruchamianego testu. Nie ma w nim zagrożeń wynikających z danych, warunków początkowych ani żadnych innych „śmieci” pozostałych po poprzednim uruchomieniu.

Zastosowanie

Instancje efemeryczne nadają się do stosów, które można szybko dostarczyć zaczynając od zera. „Szybko” oznacza szybką informację zwrotną, na której zespołom tak zależy. W przypadku częstych zmian, takich jak zatwierdzenia kodu aplikacji podczas szybkich

faz programowania, czas budowy nowego środowiska bywa prawdopodobnie dłuższy od akceptowanego przez ludzi. Ale mniej częste zmiany, na przykład poprawki do systemu operacyjnego, mogą dopuszczać testowanie z użyciem pełnej odbudowy.

Konsekwencje

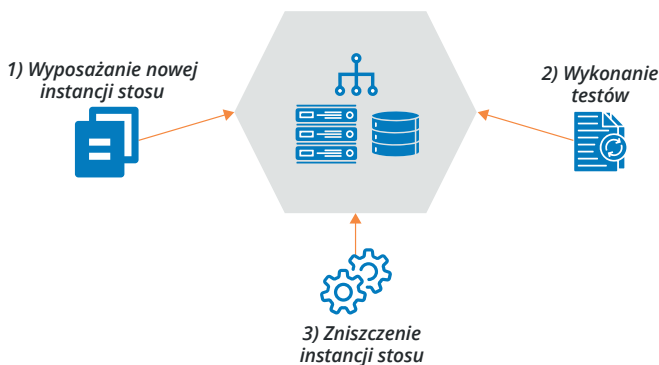
Tworzenie stosów od podstaw zajmuje zwykle dużo czasu. Dlatego etapy wykorzystujące efemeryczne instancje stosu mają wolniejsze pętle informacji zwrotnej i cykle dostarczania.

Implementacja

W celu zaimplementowania efemerycznej instancji testowej, końcowy etap testów powinien uruchomić polecenie niszczące instancję stosu po zakończeniu testowania i raportowania. Można tak skonfigurować etap, aby zatrzymywał się przed zniszczeniem instancji, jeśli testy się nie powiodły, umożliwiając wykonanie debugowania po awarii.

Powiązane wzorce

Podobnie wygląda wzorzec ciągłego resetowania stosu („Wzorzec: ciągłe resetowanie stosu” na stronie 141), ale w nim polecenia utworzenia i destrukcji stosu uruchamiane są poza przebiegiem testowym, więc potrzebny na to czas nie ma wpływu na pętlę informacji zwrotnej.

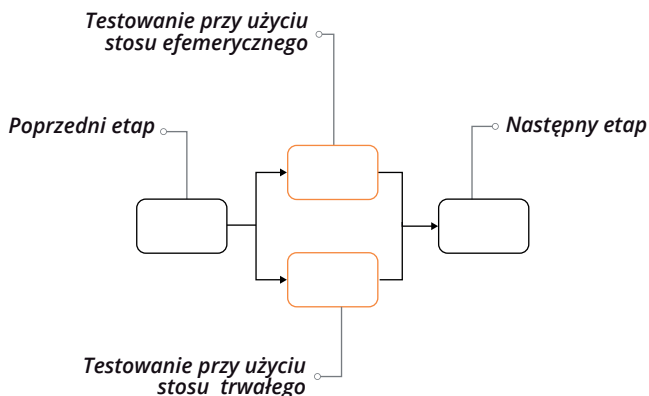


Rysunek 9-6 Efemeryczna instancja stosu testowego

Antywzorzec: podwójny etap stosu – trwały i efemeryczny

Znany również jako: *szybki i brudny plus wolny i czysty*.

W przypadku *trwałych* i *efemerycznych* stosów, potok wysyła każdą zmianę do dwóch różnych wystąpień stosu, jednego, który używa efemerycznej instancji stosu oraz drugiego, który używa trwałej instancji stosu. Stanowi to połączenie wzorca trwałego stosu testowego (patrz „Wzorzec: trwały stos testowy” na stronie 136) i wzorca efemerycznego stosu testowego (patrz „Wzorzec: efemeryczny stos testowy” na stronie 137).



Rysunek 9-7 Podwójny etap stosu – trwały i efemeryczny

Motywacja

Zespoły implementują zwykle ten wzorec, aby uniknąć wad każdego z tych wzorców, które łączy on w sobie. Jeśli wszystko działa poprawnie, etap „szybki i brudny” (wykorzystujący instancję trwałą) zapewnia szybką informację zwrotną. Jeśli etap ten zakończy się niepowodzeniem z powodu zaklinowania środowiska, w końcu otrzymamy informację zwrotną z etapu „wolnego i czystego” (wykorzystującego instancję efemeryczną).

Zastosowanie

Czasem warto zaimplementować oba rodzaje etapu jako rozwiązanie przejściowe, na drodze do bardziej niezawodnej metody.

Konsekwencje

W praktyce używanie obu rodzajów cyklu życia stosu oznacza połączenie wad jednego i drugiego. Jeśli aktualizacja istniejącego stosu bywa zawodna, to zespół nadal będzie poświęcał czas na ręczne naprawianie etapu, gdy coś pójdzie źle. A do tego i tak będzie prawdopodobnie czekał na zakończenie wolniejszego etapu, aby mieć pewność, że zmiana jest poprawna.

Ten antywzorec jest ponadto kosztowny, ponieważ wymaga podwójnych zasobów infrastruktury, przynajmniej podczas wykonywania testów.

Implementacja

Implementacja podwójnych etapów wymaga utworzenia dwóch etapów potoku, obu wyzwalanych przez poprzedni etap potoku dla projektu stosu, jak to jest pokazane na rysunku 9-7. Czasem bywa konieczne ukończenie obu etapów, aby można było promować wersję stosu do następnego etapu, a czasem można to zrobić, gdy chociaż jeden z etapów zakończy się powodzeniem.

Powiązane wzorce

Ten antywzorzec jest kombinacją wzorca trwałego stosu testowego (patrz „Wzorzec: trwały stos testowy” na stronie 136) i wzorca efemerycznego stosu testowego (patrz „Wzorzec: efemeryczny stos testowy” na stronie 137).

Wzorzec: okresowa odbudowa stosu

Znany również jako: nocna odbudowa.

Wzorzec *okresowej odbudowy stosu* używa trwałej instancji stosu testowego (patrz „Wzorzec: trwały stos testowy” na stronie 136) do etapu testowania stosu, a następnie wykorzystuje proces, który jest uruchamiany poza przepływem w celu zniszczenia i odbudowy instancji stosu zgodnie z pewnym harmonogramem, na przykład co noc.

Motywacja

Okresowa odbudowa jest często stosowana w celu redukcji kosztów. Stos jest niszczone pod koniec dnia pracy, a nowy jest udostępniany na początku następnego dnia.

Okresowa odbudowa bywa pomocna w przypadku zawodnych aktualizacji stosu, w zależności od tego, co jest przyczyną tej zawodności. W niektórych przypadkach wykorzystanie zasobów przez instancje rośnie z czasem, na przykład zajętość pamięci i magazynu ulega kumulacji podczas kolejnych testów. Regularne resetowanie pozwala zrobić z tym porządek.

Zastosowanie

Odbudowywanie instancji stosu w celu poradzenia sobie z nadmiernym zużyciem zasobów często maskuje problemy lub wady projektu, które są tego przyczyną. W takiej sytuacji wzorzec ten może być w najlepszym przypadku tymczasowym trikiem, a w najgorszym – sposobem na gromadzenie się problemów aż do nastąpienia katastrofy.

Z drugiej strony, niszczenie instancji stosu w celu redukcji kosztów, gdy nie jest ona używana, ma sens szczególnie w przypadku korzystania z zasobów taryfowych, takich jak udostępniają publiczne platformy chmurowe.

Konsekwencje

Jeśli wzorzec ten jest używany do zwalniania bezczynnych zasobów, to trzeba się zastanowić, skąd można mieć pewność, że nie są one potrzebne. Na przykład osoby pracujące poza godzinami pracy albo w innych strefach czasowych mogą mieć zablokowany dostęp do środowisk testowych.

Implementacja

Większość narzędzi do orkiestracji potoków pozwala łatwo tworzyć zadania uruchamiane zgodnie z harmonogramem w celu niszczenia i odbudowy instancji stosów. Bardziej

wyrafinowane rozwiązanie działa na bazie poziomów aktywności. Na przykład można mieć zadanie, które niszczy instancję, jeśli etap stosu nie został uruchomiony w ciągu ostatniej godziny.

Dostępne są trzy opcje wyzwalania budowy nowej instancji po zniszczeniu poprzedniej. Jedną z nich jest odbudowa instancji natychmiast po zniszczeniu. Takie podejście czyści zasoby, ale nie zmniejsza kosztów.

Drugą opcją jest budowa nowej instancji środowiska w zaplanowanym momencie. Może to jednak uniemożliwić ludziom pracę w elastycznych godzinach.

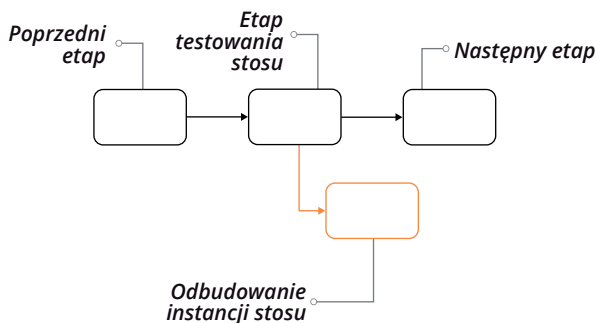
Trzecią opcją jest dostarczenie nowej instancji dla etapu testowania, jeśli aktualnie nie istnieje. Należy utworzyć oddzielne zadanie, które niszczy instancję, albo według harmonogramu, albo po okresie bezczynności. Za każdym razem, gdy jest uruchamiany etap testowania, następuje sprawdzenie, czy instancja już działa. Jeśli nie, najpierw inicjowana jest nowa instancja. Przy takim podejściu trzeba czasem dłużej niż zwykle czekać na uzyskanie wyników testu. Pierwsza osoba wypychająca rano zmianę musi poczekać, aż system zainicjuje stos.

Powiązane wzorce

Ten wzorzec może działać jak wzorzec trwałego stosu testowego (patrz „Wzorzec: trwały stos testowy” na stronie 136) – jeśli aktualizacje stosu są zawodne, ludzie muszą poświęcać czas na naprawianie uszkodzonych instancji.

Wzorzec: ciągłe resetowanie stosu

W przypadku wzorca *ciągłego resetowania stosu* po każdym zakończeniu etapu testowania stosu zostaje wykonane poza przepływem zadanie, które niszczy i odbudowuje instancję stosu (patrz rysunek 9-8).



Rysunek 9-8 Przepływ potoku dla ciągłego resetowania stosu

Motywacja

Niszczenie i odbudowywanie instancji stosu za każdym razem zapewnia czyste konto dla wykonania testu. Wzorzec może automatycznie usuwać uszkodzone instancje, o ile nie są zbyt uszkodzone, aby narzędzie stosu mogło je zniszczyć. Ponadto eliminuje czas potrzebny na tworzenie i niszczenie instancji stosu z poziomu pętli informacji zwrotnej.

Inną zaletą tego wzorca jest możliwość niezawodnego testowania procesu aktualizacji, który miałby miejsce dla danej wersji kodu stosu w środowisku produkcyjnym.

Zastosowanie

Niszczenie instancji w tle potrafi działać prawidłowo, o ile projekt stosu nie ma tendencji do psucia się i nie wymaga ręcznej interwencji w celu naprawy.

Konsekwencje

Ponieważ stos jest niszczony i udostępniany poza przepływem dostarczania w ramach potoku, ewentualne problemy mogą nie być widoczne. Potok może być „zielony”, ale instancja testu może mieć problem za kulisami. Gdy następna zmiana osiągnie etap testowania, może potrwać, zanim zdamy sobie sprawę, że zakończyła się niepowodzeniem z powodu zadania w tle, a nie jej samej.

Implementacja

Gdy etap testowania zakończy się pomyślnie, kod projektu stosu jest promowany do następnego etapu. Następuje również wyzwolenie zadania niszczącego i odbudowującego instancję stosu. Gdy ktoś wypchnie nową zmianę w kodzie, etap testowania zastosuje ją do instancji jako aktualizację.

Odbudowując instancję trzeba zdecydować, która wersja kodu stosu ma zostać użyta. Można użyć tej samej, która właśnie przeszła przez etap testowania. Alternatywą jest pobranie ostatniej wersji kodu stosu zastosowanej do instancji produkcyjnej. W ten sposób każda wersja kodu stosu będzie testowana jako aktualizacja bieżącej wersji produkcyjnej. W zależności od tego, jak wygląda typowy przepływ kodu infrastruktury do wersji produkcyjnej, może to być dokładniejsza reprezentacja procesu uaktualniania.

Powiązane wzorce

W teorii wzorzec ten przypomina wzorzec trwałego stosu testowego (patrz „Wzorzec: trwały stos testowy” na stronie 136), zapewniając informację zwrotną, a jednocześnie oferuje niezawodność wzorca efemerycznego stosu testowego (patrz „Wzorzec: efemeryczny stos testowy” na stronie 137).

Orkiestracja testów

Opisałem wszystkie ruchome części testowania stosów: rodzaje testów i walidacji, które można stosować, używanie warunków początkowych testów do obsługi zależności oraz cykle życia testowych instancji stosów. Ale jak to połączyć, aby można było konfigurować i uruchamiać testy?

Większość zespołów używa skryptów do orkiestracji swoich testów. Często są to te same skrypty, które są wykorzystywane do orkiestracji uruchamiania narzędzi stosu. W podrozdziale „Używanie skryptów do opakowywania narzędzi infrastruktury” na stronie 327 zajmę się szczegółowo tymi skryptami, które mogą obsługiwać konfigurację, koordynowanie działań na wielu stosach oraz inne aktywności, w tym także testowanie. Orkiestracja testów może obejmować:

- Tworzenie warunków początkowych testu
- Ładowanie danych testowych (częściej potrzebne do testowania aplikacji niż do testowania infrastruktury)
- Zarządzanie cyklem życia instancji stosu testowego
- Dostarczania parametrów do narzędzia testu
- Uruchamianie narzędzia testowego
- Konsolidację wyników testu
- Czyszczenie instancji testowych, warunków początkowych i danych

Większość tych zagadnień, jak warunki początkowe testu i cykle życia instancji stosu, zostało omówionych wcześniej w tym rozdziale. Inne, w tym uruchamianie testów i konsolidowanie wyników, zależą od konkretnego narzędzia.

Dwie wskazówki, które należy wziąć pod uwagę w przypadku orkiestracji testów, dotyczą wspomagania lokalnego testowania i unikania silnego sprzężenia z narzędziami potoku.

Wspomaganie lokalnego testowania

Osoby pracujące nad kodem stosu infrastruktury powinny móc same uruchamiać testy przed wypchnięciem kodu do współdzielonego potoku i środowisk. W podrozdziale „Prywatne instancje infrastruktury” na stronie 338 omówię podejścia, które pomagają pracować z osobistymi instancjami stosu na platformie infrastruktury. W ten sposób można tworzyć kod i uruchamiać testy online przed wypchnięciem zmian.

Oprócz możliwości pracy z osobistymi instancjami stosów ludzie potrzebują także narzędzi testujących i innych elementów potrzebnych do uruchamiania testów w ich lokalnym środowisku pracy. Wiele zespołów korzysta ze środowisk programowania opartych na kodzie, które automatycznie instalują i konfiguruje narzędzia. Używając kontenerów

lub maszyn wirtualnych można tworzyć pakiety środowisk programowania i uruchamiać je na różnego typu systemach komputerowych¹⁹. Alternatywnie można używać hostowanych stacji roboczych (najlepiej skonfigurowanych jako kod), chociaż potrafią one powodować opóźnienia, zwłaszcza w przypadku zespołów rozproszonych.

Kluczem do ułatwienia ludziom samodzielnego uruchamiania testów jest używanie tych samych skryptów orkiestracji na wszystkich etapach pracy lokalnej i potoku. Takie postępowanie gwarantuje, że testy będą wszędzie skonfigurowane i wykonywane w sposób spójny.

Unikanie silnego sprzężenia z narzędziami potoku

Wiele narzędzi do orkiestracji potoków oraz CI ma funkcje lub wtyczki do orkiestracji testów, a nawet do ich konfigurowania i uruchamiania. Chociaż funkcje te mogą wydawać się wygodne, nie pozwalają łatwo konfigurować i uruchamiać testów w spójny sposób poza potokiem. Mieszanie konfiguracji potoku i testów może ponadto utrudniać wprowadzanie zmian.

Zamiast tego należy zaimplementować własną orkiestrację testów w oddzielnym skrypcie lub narzędziu. Etap testowania powinien wywoływać to narzędzie, przekazując do niego jak najmniej parametrów konfiguracyjnych. Takie podejście pozwala zachować luźne sprzężenie orkiestracji potoku z orkiestracją testów.

Narzędzia do orkiestracji testów

Wiele zespołów tworzy niestandardowe skrypty do orkiestracji testów. Skrypty te wyglądają podobnie albo mogą być nawet identyczne jak skrypty używane do orkiestracji zarządzania stosem (patrz „Używanie skryptów do opakowywania narzędzi infrastruktury” na stronie 327). Ludzie używają skryptów Bash, plików wsadowych, Ruby, Pythona, Make, Rake i innych, o których nawet nie słyszałem.

Jest kilka narzędzi, które zostały zaprojektowane specjalnie do orkiestracji testów infrastruktury. Dwa, które znam, to Test Kitchen²⁰ i Molecule²¹. Test Kitchen jest produktem open source firmy Chef, który pierwotnie miał na celu testowanie książek kucharskich Chef. Molecule jest narzędziem open source zaprojektowanym do testowania scenariuszy Ansible. Oba tych narzędzi można używać do testowania stosów infrastruktury, na przykład w postaci Kitchen-Terraform²².

Korzystanie z tych narzędzi niesie pewne wyzwanie, wynikające z faktu, że zostały zaprojektowane z myślą o konkretnym przepływie pracy i może być trudno dostosować

19 Wygodnym narzędziem do udostępniania konfiguracji maszyny wirtualnej członkom zespołu jest Vagrant (<https://www.vagrantup.com>).

20 <https://kitchen.ci>

21 https://oreil.ly/_CZtn

22 <https://oreil.ly/BBfoT>

ich konfigurację do przepływu, który jest nam potrzebny. Niektórzy próbują je dostroić i dopasować, podczas gdy inni stwierdzają, że łatwiej jest im napisać własne skrypty.

Podsumowanie

W tym rozdziale podałem przykład tworzenia potoku z wieloma etapami w celu implementacji podstawowej praktyki ciągłego dostarczania i testowania dla kodu na poziomie stosu. Jednym z wyzwań związanych z testowaniem kodu stosu są narzędzia. Chociaż jest dostępnych trochę narzędzi – o wielu z nich wspominam w tym rozdziale – TDD, CI i zautomatyzowane testowanie nie mają jeszcze ugruntowanej pozycji w infrastrukturze w chwili, kiedy to piszę. Trzeba samemu wybrać narzędzia, których chce się używać, a być może, na skutek braku odpowiedniego narzędzia konieczne będzie napisanie własnych skryptów. Miejmy nadzieję, że z czasem się to poprawi.

Praca z serwerami i innymi platformami wykonawczymi aplikacji

Środowiska wykonawcze aplikacji

W podrozdziale „Części systemu infrastruktury” na stronie 21 przedstawiłem środowiska wykonawcze aplikacji jako część modelu organizującego elementy systemu w postaci trzech warstw. W tym modelu zasoby z warstwy infrastruktury są łączone w celu udostępnienia platformy wykonawczej, na której można wdrażać aplikacje.

Środowiska wykonawcze aplikacji (*application runtime*) składają się ze stosów infrastruktury, definiowanych i tworzonych przy użyciu narzędzi do zarządzania infrastrukturą, jak to zostało opisane w rozdziale 5 i jest przedstawione na rysunku 10-1.



Rysunek 10-1 Warstwa aplikacji złożona ze stosów infrastruktury

Punktem wyjścia do projektowania i implementowania infrastruktury środowiska wykonawczego aplikacji jest poznanie tych aplikacji, które będą z niej korzystać. Jakiego języka i jakich stosów wykonania używają? Czy będą spakowane i wdrażane na serwerach, czy w kontenerach, a może w postaci bezserwerowego kodu FaaS? Czy mają to być pojedyncze aplikacje wdrażane w pojedynczych lokalizacjach, czy może wiele usług dystrybuowanych w ramach klastra? Jakie są ich wymagania w zakresie łączności i danych?

Odpowiedzi na te wszystkie pytania prowadzą do poznania zasobów infrastruktury, które warstwa wykonawcza aplikacji będzie musiała wyposażać i zarządzać nimi w celu wykonywania tych aplikacji. Części warstwy wykonawczej aplikacji są mapowane na części platformy infrastruktury opisane w podrozdziale „Zasoby infrastruktury” na stronie 25. Obejmuje to środowisko wykonawcze oparte na zasobach obliczeniowych, zarządzanie danymi zbudowane na bazie zasobów magazynowych oraz łączność złożoną z zasobów sieciowych.

W tym rozdziale streszczę każdą z tych relacji, koncentrując się na sposobach organizowania zasobów infrastruktury w platformy wykonawcze dla aplikacji. Będzie to grunt pod następne rozdziały, w których opowiem bardziej szczegółowo, jak definiować te zasoby i jak zarządzać nimi jako kodem – serwerami jako kodem (rozdział 11) i klastrami jako kodem (rozdział 14).

Infrastruktura natywna dla chmury i oparta na aplikacjach

Oprogramowanie natywne dla chmury jest tak zaprojektowane i zaimplementowane, aby wykorzystywało dynamiczny charakter nowoczesnej infrastruktury. W odróżnieniu od oprogramowania starszej generacji, instancje aplikacji natywnych dla chmury mogą być w transparentny sposób dodawane, usuwane i przenoszone w ramach bazowej infrastruktury. Bazowa platforma dynamicznie alokuje zasoby obliczeniowe i pamięciowe oraz kieruje ruch do i z aplikacji. Aplikacja bezproblemowo integruje się z usługami, takimi jak monitorowanie, rejestrowanie, uwierzytelnianie i szyfrowanie.

Ludzie z Heroku sformułowali 12-czynnikową¹ metodologię budowania aplikacji wykonywanych w infrastrukturze chmury. Zwrot „natywna dla chmury” jest często kojarzony z ekosystemem Kubernetes².

Wiele organizacji ma w swoim portfolio oprogramowanie, które nie jest natywne dla chmury. Chociaż część z tego oprogramowania można przekonwertować lub przepisać na nowo, aby stało się natywne dla chmury, w wielu przypadkach koszty takiego przedsięwzięcia nie równoważą ewentualnych korzyści. Strategia infrastruktury opartej na aplikacjach polega na budowaniu środowisk wykonawczych aplikacji dla aplikacji korzystających z nowoczesnej, dynamicznej infrastruktury.

¹ <https://12factor.net>

² Cloud Native Computing Foundation (<https://www.cncf.io>) stara się ustalić standardy dla tego gatunku.

Zespoły zapewniają również środowiska wykonawcze dla nowych aplikacji, które działają jako kod bezserwerowy albo w kontenerach. Zapewniają też infrastrukturę do obsługi istniejących już aplikacji. Cała infrastruktura jest definiowana, inicjowana i zarządzana jako kod. Infrastruktura oparta na aplikacji może być udostępniana dynamicznie przy użyciu warstwy abstrakcji (patrz „Budowanie warstwy abstrakcji” na stronie 279).

Cele środowiska wykonawczego aplikacji

Implementowanie strategii opartej na aplikacji zaczyna się od analizy wymagań posiadanego portfela aplikacji dotyczących środowiska wykonawczego. Dopiero wtedy można zaprojektować rozwiązania dla środowiska wykonawczego, które będą spełniać te wymagania, implementować stosy wielokrotnego użytku, składniki stosów i inne elementy, których zespoły używają do organizowania środowisk dla określonych aplikacji.

Wdrażalne części aplikacji

Wdrażalne wydanie aplikacji może obejmować różne elementy. Odkładając na bok dokumentację i inne metadane, przykładowa lista składników wdrożenia aplikacji wygląda następująco:

Pliki wykonywalne

Podstawę wydania stanowią pliki wykonywalne (choćby jeden), bez względu na to, czy są to pliki binarne, czy interpretowane skrypty. Za członków tej kategorii można również uważać biblioteki i inne pliki używane przez pliki wykonywalne.

Konfiguracja serwera

Wiele pakietów wdrożenia aplikacji dokonuje zmian w konfiguracji serwera. Mogą one dotyczyć kont użytkowników, w ramach których będą uruchamiane procesy, struktury folderów oraz plików konfiguracyjnych systemu.

Struktury danych

Gdy aplikacja korzysta z bazy danych, jej wdrożenie może powodować utworzenie lub aktualizację schematów. Dana wersja schematu odpowiada zwykle jakiejś wersji pliku wykonywalnego, dlatego najlepiej jest wiązać je i wdrażać razem.

Dane referencyjne

Wdrożenie aplikacji może wypełniać bazę danych lub inny magazyn początkowym zbiorem danych. Mogą to być dane referencyjne, zmieniające się wraz z nowymi wersjami lub dane przykładowe, które pomagają użytkownikom rozpocząć korzystanie z aplikacji zaraz po jej zainstalowaniu.

Łączność

Wdrożenie aplikacji może określać konfigurację sieci, na przykład porty sieciowe. Może również obejmować elementy używane do obsługi łączności, jak certyfikaty lub klucze do szyfrowania lub uwierzytelniania połączeń.

Parametry konfiguracyjne

Pakiet wdrożenia aplikacji może ustawiać parametry konfiguracyjne, bądź to kopiując pliki konfiguracyjne na serwer, bądź wypychając ustawienia do rejestru.

Granica między aplikacją a infrastrukturą może przebiegać w różnych miejscach. Na przykład można dołączyć wymaganą bibliotekę do pakietu wdrożenia aplikacji, a można ją udostępnić jako część infrastruktury.

Na przykład obraz kontenera zawiera zwykle większość systemu operacyjnego, jak również aplikację, która będzie na nim działać. Niezmienialny serwer albo niezmienialny stos idą jeszcze dalej, łącząc aplikację i infrastrukturę w jedną całość. Z drugiej strony, widziałem przykłady kodu infrastruktury wyposażającego biblioteki oraz pliki konfiguracyjne dla konkretnej aplikacji, co sprawiało, że w pakiecie aplikacji pozostawało naprawdę niewiele.

Na organizację bazy kodu ma również wpływ odpowiedź na kolejne pytanie. Czy kod aplikacji jest trzymany razem z kodem infrastruktury, czy oddzielnie? Kwestia granic kodu zostanie omówiona w dalszej części tej książki (patrz rozdział 18), ale jedna z zasad mówi, że zazwyczaj dobrze jest dopasować strukturę bazy kodu do wdrażanych elementów.

Pakiety wdrożenia

Aplikacje są często zorganizowane w postaci pakietów wdrożenia, których format zależy od typu środowiska wykonawczego. Przykłady formatów pakietów wdrożenia i związanych z nimi środowisk wykonawczych są wymienione w tabeli 10-1.

Tabela 10-1 Przykłady docelowych środowisk wykonawczych i formaty pakietów aplikacji

Docelowe środowisko wykonawcze	Przykładowe pakiety
System operacyjny serwera	Pliki RPM (Red Hat), pliki <i>.deb</i> (Debian), pakiety instalatora MSI (Windows)
Aparat wykonawczy języka	Gemy języka Ruby, pakiety pip Pythona, pliki <i>.jar</i> , <i>.war</i> i <i>.ear</i> Javy.
Środowisko wykonawcze kontenera	Obrazy platformy Docker
Klastry aplikacji	Deskryptory wdrażania Kubernetes, schematy Helm
Bezserwerowe usługi FaaS	Pakiet wdrożenia usługi Lambda

Format pakietu wdrożenia jest standardem, umożliwiającym narzędziom wdrażania lub środowiskom wykonawczym wyodrębnianie różnych części aplikacji i umieszczanie ich we właściwych miejscach.

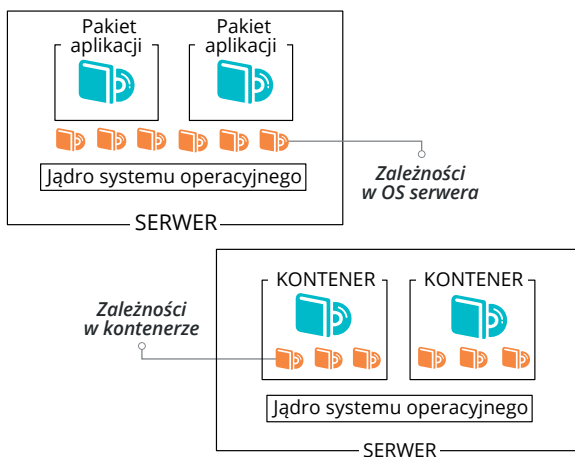
Wdrażanie aplikacji na serwerach

Serwery, zarówno fizyczne, jak i wirtualne, są tradycyjnymi platformami wykonawczymi. Aplikacja jest pakowana z zastosowaniem formatu pakowania systemu operacyjnego, takiego jak RPM, plik *.deb* czy Windows MSI albo jest pakowana w formacie środowiska uruchomieniowego języka, takim jak gem języka Ruby lub plik *.war* języka Java. Od niedawna jako formaty pakowania i wdrażania aplikacji na serwerach są coraz chętniej stosowane obrazy kontenerów, takie jak obrazy Docker.

Definiowanie i udostępnianie serwerów jako kodu jest tematem rozdziału 11. Zagadnienie to pokrywa się z tematem wdrażania aplikacji, jeśli chodzi o potrzebę decydowania, kiedy i jak należy wykonywać polecenia wdrażania (patrz „Konfigurowanie nowej instancji serwera” na stronie 180).

Pakowanie aplikacji do kontenerów

Kontenery pobierają zależności z systemu operacyjnego do pakietu aplikacji, czyli obrazu kontenera, co pokazuje rysunek 10-2³.



Rysunek 10-2 Zależności mogą być zainstalowane w systemie operacyjnym lub ulokowane w kontenerach

Dołączanie zależności do kontenerów powoduje, że są one większe od typowych pakietów w formacie systemu operacyjnego lub środowiska uruchomieniowego języka, ale ma kilka zalet:

- Kontener tworzy bardziej spójne środowisko do uruchamiania aplikacji. Bez kontenera aplikacja musi bazować na bibliotekach, konfiguracji, kontach użytkowników

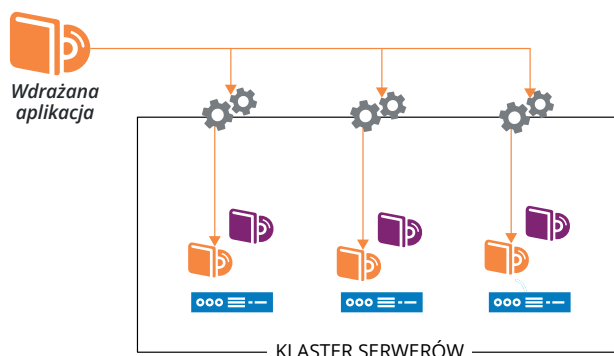
³ Dominującym formatem kontenera jest Docker (<https://oreil.ly/bJt9->). Inne to CoreOS rkt (<https://coreos.com/rkt>) i Windows Containers (<https://oreil.ly/2uxgp>).

i innych elementach, które mogą się różnić w poszczególnych serwerach. Kontener wiąże środowisko wykonawcze z aplikacją i jej zależnościami.

- Skonteneryzowana aplikacja jest w znacznym stopniu odizolowana od serwera, na którym działa, co zapewnia elastyczność przy wyborze miejsca jej uruchomienia.
- Pakowanie kontekstu systemu operacyjnego aplikacji do obrazu kontenera upraszcza i standaryzuje wymagania wobec serwera hosta. Serwer musi mieć zainstalowane narzędzia do uruchamiania kontenerów, ale niewiele ponadto.
- Ograniczenie zmienności środowisk wykonawczych aplikacji pozwala zapewnić większą jakość. Testując instancję kontenera w jednym środowisku można mieć uzasadnioną pewność, że będzie ona zachowywać się tak samo w innych środowiskach.

Wdrażanie aplikacji w klastrach serwerów

Aplikacje są często wdrażane w grupach serwerów, ponieważ przed pojawieniem się aplikacji opartych na kontenerach, to klastry liczyły się najbardziej. W typowym modelu występuje klastrowanie serwerów (zgodnie z opisem w podrozdziale „Zasoby obliczeniowe” na stronie 26) i na każdym serwerze uruchamiany jest identyczny zestaw aplikacji. Aplikacje są pakowane w identyczny sposób, jak dla pojedynczego serwera, a proces wdrażania jest powtarzany dla każdego serwera w puli, na przykład przy użyciu narzędzia do zdalnego wykonywania skryptów z poleceniami, takiego jak Capistrano⁴ lub Fabric⁵. Patrz rysunek 10-3.



Rysunek 10-3 Aplikacje są wdrażane na każdym serwerze klastra

Jeśli aplikacja ma być wdrożona na wielu serwerach, to musimy wybrać sposób orkiestracji wdrożenia. Czy chcemy wdrożyć ją na wszystkich serwerach jednocześnie? Czy na czas tej operacji cała usługa musi przejść w tryb offline? A może będziemy aktualizować po kolei każdy serwer? Można skorzystać z wdrażania przyrostowego na serwerach i zastosować strategię wdrażania progresywnego, taką jak wzorzec wdrażania niebiesko-zielony lub

⁴ <https://capistranorb.com>

⁵ <https://www.fabfile.org>

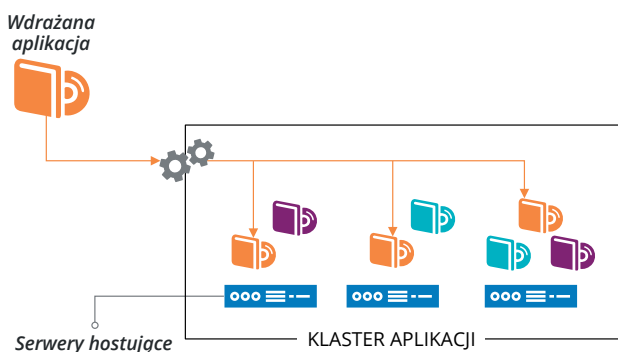
kanarkowy (więcej o tych strategiach jest w podrozdziale „Zmiana działającej infrastruktury” na stronie 360).

Oprócz wdrożenia kodu aplikacji na serwerach konieczne może być wdrożenie innych elementów, na przykład zmian w strukturach danych lub łączności.

Wdrażanie aplikacji w klastrach aplikacji

Jak już zostało to opisane w podrozdziale „Zasoby obliczeniowe” na stronie 26, klaster obsługujący aplikacje jest pulą serwerów wykonujących jedną lub więcej aplikacji. W odróżnieniu od klastra serwerów, w którym każdy serwer wykonuje ten sam zestaw aplikacji, w klastrze aplikacji różne serwery mogą wykonywać różne grupy instancji aplikacji (patrz rysunek 10-4).

Podczas wdrażania jakiejś aplikacji w klastrze planista (scheduler) decyduje, na których serwerach będą uruchamiane instancje tej aplikacji. Planista może zmienić ten rozkład, dodając lub usuwając instancje aplikacji w ramach serwerów, zgodnie z różnymi algorytmami i ustawieniami.



Rysunek 10-4 Aplikacje są wdrażane w klastrze i rozdzielane pomiędzy węzły

W dawnych czasach⁶ najpopularniejsze były klastry aplikacji oparte na Javie (Tomcat, Websphere, Weblogic, JBoss i inne). Kilka lat temu pojawiła się fala systemów zarządzania klastrami, w tym Apache Mesos⁷ i DC/OS⁸, z których wiele było zainspirowanych przez Borg firmy Google⁹. W ostatnich czasach serwery aplikacji i orkiestratory klastrów zostały wyparte przez systemy skupione na orkiestracji instancji kontenerów.

Definiowanie i udostępnianie klastrów jako kodu stanowi temat rozdziału 14. Gdy mamy już klaster, wdrożenie pojedynczej aplikacji może być proste. Wystarczy zapakować

⁶ „Dawne czasy” miały miejsce mniej niż 10 lat temu.

⁷ <https://oreil.ly/46lat>

⁸ <https://dcos.io>

⁹ Borg to wewnętrzny, zastrzeżony system zarządzania klastrami firmy Google, udokumentowany w artykule „Large-Scale Cluster Management at Google with Borg” (<https://oreil.ly/yA8nV>).

ją do kontenera Docker i wypchnąć do klastra. Ale bardziej złożone aplikacje, z większą liczbą ruchomych części, potrafią stawiać większe wymagania.

Pakiety do wdrażania aplikacji w klastrach

Nowoczesne aplikacje składają się często z wielu procesów i składników wdrożonych w złożonej infrastrukturze. Środowisko wykonawcze musi wiedzieć, jak wykonywać te różne elementy:

- Jaka jest minimalna i maksymalna liczba wykonywanych instancji?
- Skąd środowisko wykonawcze ma wiedzieć, kiedy dodawać lub usuwać instancje?
- Skąd środowisko wykonawcze ma wiedzieć, czy dana instancja jest w dobrej kondycji, czy też należy ją uruchomić ponownie?
- Jak należy wyposażyć pamięć masową i dołączyć do każdej instancji?
- Jakie są wymagania dotyczące łączności i bezpieczeństwa?

Różne platformy wykonawcze udostępniają różne funkcjonalności i wiele z nich stosuje własne formaty pakowania i konfiguracji. Platformy te często używają *manifestu wdrożenia*, który odwołuje się do rzeczywistych artefaktów wdrożenia (na przykład obrazu kontenera), a nie pliku archiwum obejmującego wszystkie wdrażalne elementy. Oto niektóre przykłady manifestów wdrożenia aplikacji klastrowych:

- Schematy Helm¹⁰ dla klastrów Kubernetes
- Manifesty wdrożenia Kubernetes dla Weave Cloud¹¹
- Usługi AWS ECS Services¹², które można definiować jako kod za pomocą swoich ulubionych narzędzi do zarządzania stosem
- Plany Azure App Service¹³
- CNAB Cloud Native Application Bundle¹⁴

Różne pakiety i manifesty wdrożenia działają na różnych poziomach. Niektóre są skoncentrowane na jednej jednostce wdrożenia, co oznacza oddzielny manifest dla każdej aplikacji, inne definiują kolekcję usług możliwych do wdrożenia. W zależności od narzędzia, każda usługa w kolekcji może mieć oddzielny manifest, przy czym manifest wyższego poziomu definiuje elementy wspólne i parametry integracji.

Manifest wdrożenia serwera WWW ShopSpinner może mieć postać pseudokodu pokazanego w przykładzie 10-1.

¹⁰ <https://helm.sh>

¹¹ <https://oreil.ly/XbgmJ>

¹² <https://oreil.ly/GKLie>

¹³ <https://oreil.ly/x113B>

¹⁴ <https://cnab.io>

Przykład 10-1 Przykład manifestu wdrożenia klastra aplikacji

```
service:
  name: webservers
  organization: ShopSpinner
  version: 1.0.3 application:
  name: nginx
  container_image:
    repository: containers.shopspinner.xyz
    path: /images/nginx
    tag: 1.0.3
  instance:
    count_min: 3
    count_max: 10
    health_port: 443
    health_url: /alive
  connectivity:
    inbound:
      id: https_inbound
      port: 443
      allow_from: $PUBLIC_INTERNET
      ssl_cert: $SHOPSPINNER_PUBLIC_SSL_CERT
    outbound:
      port: 443
      allow_to: [ $APPSERVER_APPLICATIONS.https_inbound ]
```

W przykładzie tym pokazane jest, gdzie i jak znaleźć obraz kontenera (blok `container_image`), ile instancji można uruchomić i jak sprawdzić ich kondycję. Ponadto zdefiniowane są reguły połączeń przychodzących i wychodzących.

Wdrażanie aplikacji bezserwerowych na platformach FaaS

W rozdziale 3 wymieniłem platformy wykonawcze aplikacji bezserwerowych FaaS jako rodzaj zasobu obliczeniowego (patrz „Zasoby obliczeniowe” na stronie 26). Większość z tych platform stosuje własny format definiowania wymagań wykonawczych aplikacji oraz pakowania ich razem z kodem i powiązanymi elementami w celu wdrożenia w instancji środowiska wykonawczego.

Podczas pisania aplikacji FaaS nie znamy szczegółów dotyczących serwerów lub kontenerów, które posłużą do wykonywania naszego kodu. Jednak kod będzie zapewne potrzebował jakiejś infrastruktury do działania. Na przykład może potrzebować routingu sieciowego dla połączeń przychodzących i wychodzących, pamięci masowej albo kolejek wiadomości. Platforma FaaS może być zintegrowana z platformą bazowej infrastruktury i automatycznie udostępniać potrzebne jej elementy. A może okazać się konieczne zdefiniowanie elementów infrastruktury w oddzielnym narzędziu do definiowania stosu. Wiele

narzędzi stosu, jak Terraform i CloudFormation, pozwala deklarować udostępnianie kodu FaaS w ramach stosu infrastruktury.

Definiowanie i udostępnianie środowisk wykonawczych FaaS w celu uruchamiania kodu jest opisane w rozdziale 14.

Dane aplikacji

Nad danymi zastanawiamy się często dopiero po wdrożeniu i uruchomieniu aplikacji. Udostępniamy bazy danych i woluminy pamięci masowej, ale jak w przypadku wielu części infrastruktury, dopiero wprowadzanie w nich zmian okazuje się trudne. Zmiana danych i struktur jest czasochłonna, skomplikowana i ryzykowna.

Wdrażanie aplikacji obejmuje często tworzenie lub zmianę struktur danych, łącznie z konwersją istniejących danych w przypadku zmiany struktur. Aktualizacja struktur danych powinna być smartwieniem aplikacji i procesu wdrożenia aplikacji, a nie platformy infrastruktury i środowiska wykonawczego. Natomiast infrastruktura i usługi środowiska wykonawczego aplikacji muszą zapewniać utrzymanie danych podczas zmiany lub awarii infrastruktury i innych bazowych zasobów. Różne podejścia do tego wyzwania są opisane w podrozdziale „Ciągłość danych w zmieniającym się systemie” na stronie 373.

Schematy i struktury danych

Niektóre magazyny danych mają ściśle określoną strukturę, jak w przypadku baz danych SQL, podczas gdy w innych nie obowiązuje żaden porządek ani schemat. Ściśle uporządkowana, oparta na schemacie baza danych wymusza strukturę danych, odmawiając przechowywania tych, które nie są prawidłowo sformatowane. Za zarządzanie formatem danych w bazach danych pozbawionych schematu odpowiadają korzystające z nich aplikacje.

Nowe wydanie aplikacji może zawierać zmiany w strukturach danych. W przypadku bazy danych opartej na schemacie wymaga to zmiany definicji struktur danych w tej bazie. Na przykład nowe wydanie może dodawać pole do rekordów danych, czego klasycznym przykładem jest rozbitcie pojedynczego pola „nazwa” na oddzielne pola z pierwszym imieniem, drugim imieniem i nazwiskiem.

Gdy następuje zmiana struktury danych, wszystkie istniejące dane muszą zostać przekonwertowane na nową strukturę, niezależnie od typu bazy danych. W przypadku rozbicia pola „nazwa” potrzebny jest proces, który podzieli nazwy przechowywane w bazie danych i umieści rezultat w oddzielnych polach.

Zmiana struktur danych i konwersja danych nazywane są *migracją schematu*. Jest kilka narzędzi i bibliotek, których aplikacje i narzędzia wdrażania mogą używać do zarządzania tym procesem, w tym Flyway¹⁵, DBDeploy¹⁶, Liquibase¹⁷ oraz db-migrate. Programiści

¹⁵ <https://flywaydb.org>

¹⁶ Narzędzie DBDeploy spopularyzowało ten styl migracji schematu bazy danych, razem z Ruby on Rails. Jednak projekt ten nie jest aktualnie kontynuowany.

¹⁷ <https://www.liquibase.org>

mogą używać tych narzędzi do definiowania przyrostowych zmian bazy danych jako kodu. Zmiany można rejestrować w systemie kontroli wersji i pakować jako część wydania. Takie działanie pomaga zapewnić synchronizację schematów bazy danych z wersją aplikacji wdrożoną w instancji.

W celu bezpiecznego i elastycznego zarządzania częstymi zmianami danych i schematów zespoły mogą używać strategii ewolucji baz danych. Strategie te są zgodne z podejściami zwinnymi inżynierii oprogramowania, w tym CI i CD, a także infrastrukturą jako kodem¹⁸.

Infrastruktura magazynu aplikacji natywna dla chmury

Infrastruktura natywna dla chmury jest dynamicznie przydzielana aplikacjom i usługom na żądanie. Niektóre platformy udostępniają magazyn natywny dla chmury, tak samo jak obliczenia i sieć. Gdy system dodaje instancję jakiejś aplikacji, może automatycznie udostępnić i dołączyć magazyn. Wymagania dotyczące magazynu, w tym sposób formatowania lub dane do załadowania w momencie udostępniania są określone w manifeście wdrażania aplikacji (patrz przykład 10-2).

Przykład 10-2 Przykład manifestu wdrażania aplikacji z magazynem danych

```
application:
  name: db_cluster
  compute_instance:
    memory: 2 GB
    container_image: db_cluster_node_application
  storage_volume:
    size: 50 GB
    volume_image: db_cluster_node_volume
```

Ten prosty przykład zawiera definicję sposobu tworzenia węzłów dla dynamicznie skalowanego klastra bazy danych. Dla każdej instancji węzła platforma utworzy instancję kontenera z oprogramowaniem bazy danych i dołączy wolumin dyskowy sklonowany z obrazu zainicjowanego przy użyciu pustego segmentu bazy danych. Podczas rozruchu instancja połączy się z klastrem i zsynchronizuje dane ze swoim lokalnym woluminem.

Łączność aplikacji

Oprócz zasobów obliczeniowych potrzebnych do wykonywania kodu oraz zasobów magazynowych do przechowywania danych, aplikacje potrzebują sieci do nawiązywania połączeń przychodzących i wychodzących. Pakiet aplikacji oparty na serwerze, na przykład

¹⁸ Więcej informacji o strategiach i technikach ewolucji baz danych można znaleźć w „Evolutionary Database Design” (<https://oreil.ly/HCuqr>) Pramoda Sadalage’a, *Refactoring Databases* (<https://oreil.ly/ta6pb>) Scotta Amblera i Pramoda Sadalage’a (Addison-Wesley Professional) oraz *Database Reliability Engineering* (<https://oreil.ly/2wTKaxq>), Laine Campbell i Charity Majors (O’Reilly).

serwerze WWW, może konfigurować porty sieciowe i dodawać klucze szyfrowania dla połączeń przychodzących. Jednak tradycyjnie wyglądało to tak, że ktoś osobno konfigurował infrastrukturę poza serwerem.

Można zdefiniować (wraz z zarządzaniem) adresowanie, routing, nadawanie nazw, reguły zapory i podobne zagadnienia jako część projektu stosu infrastruktury, a następnie wdrożyć aplikację w wynikowej infrastrukturze. Podejście bardziej natywne dla chmury polega na definiowaniu wymagań sieciowych jako części manifestu wdrożenia aplikacji i pozostawić środowisku wykonawczemu aplikacji dynamiczne alokowanie zasobów. Widać to w pierwszej części przykładu 10-1, zamieszczonej poniżej:

```
application:
  name: nginx
  connectivity:
    inbound:
      id: https_inbound
      port: 443
      allow_from: $PUBLIC_INTERNET
      ssl_cert: $SHOPSPINNER_PUBLIC_SSL_CERT
    outbound:
      port: 443
      allow_to: [ $APPSERVER_APPLICATIONS.https_inbound ]
```

Przykład zawiera definicje połączeń przychodzących i wychodzących, odwołujące się do innych części systemu. Są nimi publiczny Internet, prawdopodobnie brama oraz porty wejściowe HTTPS dla serwerów aplikacji, zdefiniowanych i wdrożonych w tym samym klastrze przy użyciu ich własnych manifestów wdrożenia.

Środowiska wykonawcze aplikacji udostępniają aplikacjom wiele popularnych usług. Wiele z tych usług należy do kategorii odnajdywanie usług.

Odnajdywanie usług

Aplikacje i usługi wykonywane w infrastrukturze muszą często wiedzieć, jak znaleźć inne aplikacje i usługi. Na przykład frontendowa aplikacja sieci Web może wysyłać żądania do backendowej usługi w celu przetworzenia transakcji dla użytkowników.

Takie rozwiązanie nie jest trudne w środowisku statycznym. Aplikacje mogą używać znanych nazw hostów dla innych usług, trzymanych prawdopodobnie w pliku konfiguracyjnym, który w razie potrzeby można aktualizować.

Ale w przypadku infrastruktury dynamicznej, w której lokalizacje serwerów i usług są płynne, potrzebny jest bardziej responsywny sposób szukania usług.

Oto kilka popularnych mechanizmów odnajdywania:

Adresy IP zapisane na stałe

Każda usługa ma przydzielony adres IP. Na przykład serwer monitorowania działa zawsze z adresem 192.168.1.5. Aby zmienić adres lub uruchomić kilka instancji usługi

(na przykład w celu kontrolowanego wdrożenia ważnej aktualizacji), trzeba przebudować i ponownie wdrożyć aplikacje.

Pliki hosts

Przy użyciu konfiguracji serwera na każdym serwerze generowany jest plik `/etc/hosts` (lub jego odpowiednik), który mapuje nazwy usług na ich bieżące adresy. Metoda jest niechlujną alternatywą dla DNS, ale zauważyłem, że jest używana do obejścia starszych implementacji DNS¹⁹.

DNS (Domain Name System)

Wpisy DNS mapują nazwy usług na ich bieżące adresy IP przy użyciu wpisów DNS zarządzanych za pomocą kodu albo DDNS (Dynamic DNS²⁰). DNS stanowi dojrzałe, dobrze obsługiwane rozwiązanie problemu.

Tagi zasobów

Zasoby infrastruktury są otagowane w celu wskazywania usługi, którą udostępniają oraz kontekstu, takiego jak środowisko. Odnajdywanie polega na użyciu API platformy w celu znalezienia zasobów z odpowiednimi tagami. Należy uważać, aby uniknąć silnego sprzężenia kodu aplikacji z platformą infrastruktury.

Rejestr konfiguracji

Instancje aplikacji mogą utrzymywać aktualne szczegóły łączności w scentralizowanym rejestrze (patrz „Rejestr konfiguracji” na stronie 93), aby inne aplikacje mogły je przeglądać. Może to być pomocne, gdy potrzebnych jest więcej informacji niż sam adres; na przykład stan kondycji lub inny wskaźnik.

Przyczepka (sidecar)

Przyczepka to oddzielny proces wykonywany równolegle z każdą instancją aplikacji. Aplikacja może używać przyczepki jako serwera proxy dla połączeń wychodzących, bramy dla połączeń przychodzących lub jako usługi odnajdywania. Przyczepki same potrzebują jakiejś metody do odnajdywania sieci. Tą metodą może być inny mechanizm odnajdywania albo może ona wykorzystywać inny protokół komunikacyjny²¹. Przyczepki są zwykle częścią siatki usług (siatki usług są omówione w podrozdziale „Siatka usług” na stronie 238) i często oferują więcej, niż tylko odnajdywanie usług. Na przykład przyczepka może obsługiwać uwierzytelnianie, szyfrowanie, rejestrowanie oraz monitorowanie.

19 Typowym przykładem „nieskomplikowanej implementacji DNS” są organizacje, które nie pozwalają zespołom zmieniać wpisów DNS. Dzieje się tak zwykle dlatego, że organizacja nie dysponuje nowoczesnymi procesami automatyzacji i nadzoru, które pozwalałyby to robić bezpiecznie.

20 <https://oreil.ly/EhGir>

21 Dokumentacja platformy Consul firmy HashiCorp (<https://oreil.ly/z4fNG>) wyjaśnia, w jaki sposób komunikują się jej przyczepki.

Brama API

Brama API jest scentralizowaną usługą http, która definiuje trasy i punkty końcowe (endpoints). Zazwyczaj oferuje więcej usług; na przykład uwierzytelnianie, szyfrowanie, rejestrowanie i monitorowanie. Innymi słowy, brama API nie różni się od przyczepki, tyle że jest raczej scentralizowana, niż rozproszona²².



Unikanie twardych granic między infrastrukturą, środowiskiem wykonawczym i aplikacjami

Teoretycznie byłoby dobrze udostępniać środowiska wykonawcze aplikacji jako kompletny zestaw usług dla programistów, zasłaniając przed nimi szczegóły bazowej infrastruktury. W praktyce granice są dużo bardziej rozmyte niż pokazane w tym modelu. Różni ludzie i zespoły potrzebują dostępu do zasobów na różnych poziomach abstrakcji, przy różnych poziomach kontroli. Nie należy więc projektować i implementować systemów z bezwzględnymi granicami, lecz definiować elementy, które można na różne sposoby składać i przedstawiać różnym użytkownikom.

Podsumowanie

Celem infrastruktury jest uruchamianie przydatnych aplikacji i usług. Wskazówki i pomysły zawarte w tej książce powinny pomóc czytelnikowi w udostępnianiu zestawów zasobów infrastruktury w potrzebnej formie. Podejście do infrastruktury oparte na aplikacjach koncentruje się na wymaganiach aplikacji dotyczących środowiska wykonawczego, pomagając projektować stosy, serwery, klastry i inne konstrukcje warstwy środkowej dla uruchamianych aplikacji.

22 Moi koledzy wyrazili zaniepokojenie z powodu tendencji do centralizowania logiki biznesowej w bramach API. Więcej szczegółów można znaleźć w artykule *Overambitious API gateways* (<https://oreil.ly/aqA2q>) opublikowanym w raporcie Technology Radar firmy ThoughtWorks.

Budowanie serwerów jako kodu

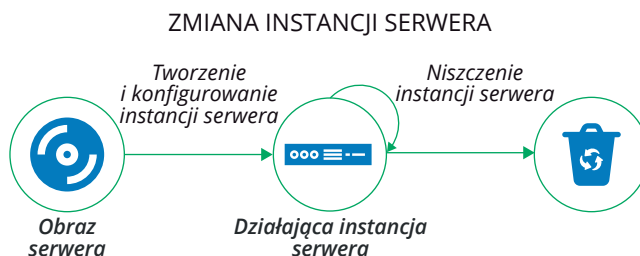
Idea infrastruktury jako kodu zrodziła się jako metoda konfigurowania serwerów. Administratorzy systemów pisali różne skrypty – powłoki, wsadowe czy w Perlu. Pionierem wykorzystywania deklaratywnych, idempotentnych języków DSL do instalowania pakietów i zarządzania plikami konfiguracyjnymi na serwerach był system CFEngine. Po nim pojawiły się Puppet i Chef. Narzędzia te zakładały rozpoczynanie od istniejącego serwera – często fizycznego serwera w szafie rackowej, czasem maszyny wirtualnej wykorzystującej VMware, a z biegiem czasu instancji w chmurze.

Teraz albo skupiamy się na stosach infrastruktury, w których serwery są tylko jedną z części, albo pracujemy z klastrami kontenerów, gdzie serwery są schowanym głębiej szczegółem. Ale serwery nadal stanowią podstawę większości środowisk wykonawczych aplikacji. Większość systemów istniejących na rynku więcej niż kilka lat uruchamia przynajmniej niektóre aplikacje na serwerach. A nawet zespoły wykorzystujące klastry muszą zwykle budować i uruchamiać serwery dla węzłów hostów.

Serwery są bardziej skomplikowane niż inne rodzaje infrastruktury, takie jak sieć czy magazyn. Mają więcej części ruchomych i odmian, dlatego większość zespołów zajmujących się systemami nadal spędza całkiem sporo czasu grzebiąc w swoich systemach operacyjnych, pakietach i plikach konfiguracyjnych.

W tym rozdziale omówię podejścia do budowania konfiguracji serwera i zarządzania nią jako kodem. Zacznę od zawartości serwerów (co trzeba skonfigurować) oraz ich cyklu życia (kiedy mają miejsce działania konfiguracyjne). Następnie przejdę do przeglądu narzędzi i kodu konfiguracyjnego serwerów. Główna część tego rozdziału jest poświęcona różnym sposobom tworzenia instancji serwerów, metodom ich wstępnej budowy w celu umożliwienia tworzenia wielu spójnych instancji oraz podejściom do stosowania konfiguracji serwera w całym cyklu jego życia.

Cykl życia serwera można potraktować jako kilka faz przejściowych, przedstawionych na rysunku 11-1.



Rysunek 11-1 Podstawowy cykl życia serwera

Pokazany tutaj podstawowy cykl życia składa się z trzech faz przejściowych:

1. Utworzenie i skonfigurowanie instancji serwera
2. Zmiana istniejącej instancji serwera
3. Usunięcie instancji serwera

W tym rozdziale opisuję tworzenie i konfigurowanie serwerów. W rozdziale 12 wyjaśnię, jak dokonywać zmian w serwerach, a w rozdziale 13 zajmę się tworzeniem i aktualizacją obrazów serwerów, których można używać do tworzenia instancji serwerów.

Co jest trzymane na serwerze

Dobrze jest mieć świadomość, jakie rzeczy są trzymane na serwerze i skąd pochodzą. Jedno z możliwych podejść do rzeczy trzymanych na serwerze to ich podział na oprogramowanie, konfigurację i dane. Kategorie te, opisane w tabeli 11-1, pomagają zrozumieć, jak narzędzia do zarządzania konfiguracją powinny traktować konkretny plik lub zbiór plików.

Różnica między konfiguracją a danymi polega na tym, że w pierwszym przypadku zawartość pliku jest automatycznie zarządzana przez narzędzia do automatyzacji. Mimo że dziennik systemowy jest niezbędny dla infrastruktury, zautomatyzowane narzędzie do konfiguracji traktuje go jako dane. Podobnie, jeśli jakaś aplikacja trzyma w pliku na serwerze takie rzeczy, jak konta i preferencje swoich użytkowników, narzędzie do konfiguracji serwera traktuje ten plik jako dane.

Tabela 11-1 Rodzaje rzeczy trzymanych na serwerze

Rodzaj rzeczy	Opis	Traktowanie przez narzędzie do zarządzania konfiguracją
Oprogramowanie	Aplikacje, biblioteki i inny kod. Nie muszą to być pliki wykonywalne; mogą to być praktycznie dowolne pliki, które są statyczne i nie zmieniają się w zależności od systemu. Przykładem są pliki z danymi stref czasowych w systemie Linux.	Upewnia się, że wszystko wygląda tak samo na podobnych serwerach; nie interesuje go zawartość.

Rodzaj rzeczy	Opis	Traktowanie przez narzędzie do zarządzania konfiguracją
Konfiguracja	Pliki wykorzystywane do sterowania działaniem systemu i/ lub aplikacji. Zawartość może być inna na różnych serwerach, w zależności od ich roli, środowiska, instancji itd. Pliki są traktowane jako część infrastruktury, a nie konfiguracja, którą zarządzają same aplikacje. Na przykład, jeśli interfejs użytkownika jakiejś aplikacji zarządza profilami użytkowników, to z punktu widzenia infrastruktury pliki danych z tymi profilami nie są traktowane jako część konfiguracji; zamiast tego są dane. W tym znaczeniu plik konfiguracyjny aplikacji, który jest przechowywany w systemie plików i zarządzany przez infrastrukturę, jest uważany za część konfiguracji.	Buduje zawartość pliku na serwerze; zapewnia jego spójność.
Dane	Pliki generowane i aktualizowane przez system i aplikacje. Infrastruktura może częściowo odpowiadać za te dane, na przykład ich dystrybucję, tworzenie kopii zapasowych albo replikację. Ale infrastruktura traktuje zawartość tych plików jak czarną skrzynkę i nie interesuje się ich zawartością. Przykładami danych w tym sensie są pliki danych baz danych i pliki dzienników.	Występują i zmieniają się; mogą wymagać zachowania, ale nie należy próbować zarządzać ich zawartością.

Skąd pochodzą rzeczy trzymane na serwerze

Oprogramowanie, konfiguracja i dane, które składają się na instancję serwera, mogą być dodawane w momencie tworzenia i konfigurowania serwera albo podczas jego zmiany. Jest kilka możliwych źródeł tych elementów (patrz rysunek 11-2):

Podstawowy system operacyjny (OS)

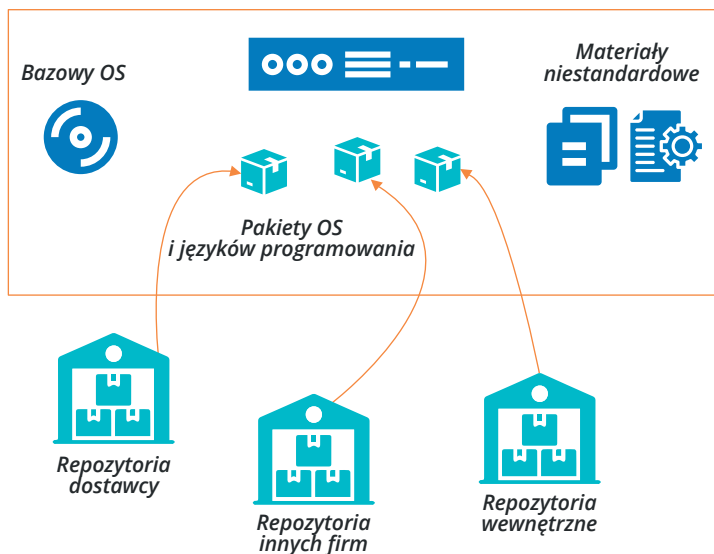
Obraz instalacji systemu operacyjnego może być fizycznym dyskiem, plikiem ISO lub stockowym obrazem serwera. Proces instalacji OS może dokonywać wyboru składników.

Repozytoria pakietów OS

Instalacja OS albo etap konfiguracji po instalacji może uruchamiać narzędzie pobierające i instalujące pakiety z jednego lub więcej repozytoriów (więcej informacji o formatach pakietów OS jest w tabeli 10-1). Dostawcy OS udostępniają zwykle repozytorium z pakietami, które obsługują. Można korzystać z repozytoriów innych firm z pakietami komercyjnymi lub open source. Można też uruchomić wewnętrzne repozytorium dla pakietów opracowanych lub wyselekcjonowanych lokalnie.

Repozytoria języków, platform programistycznych i innych platform

Oprócz pakietów specyficznych dla OS można instalować pakiety dotyczące języków i platform programistycznych, takie jak biblioteki Javy lub gemy Ruby. Podobnie jak w przypadku pakietów OS, można pobierać pakiety z repozytoriów zarządzanych przez dostawców języków, firmy trzecie lub grupy wewnętrzne.



Rysunek 11-2 *Pakiety i inne elementy instancji serwera*

Pakiety niestandardowe

Niektórzy dostawcy i grupy wewnętrzne udostępniają oprogramowanie, które ma własny instalator lub wymaga dużo więcej kroków, niż uruchomienie standardowego narzędzia do zarządzania pakietami.

Oddzielny materiał

Można dodawać i modyfikować pliki poza kontrolą aplikacji lub składnika; przykładem jest dodawanie kont użytkowników lub ustawianie reguł lokalnej zapory ogniowej.

Następne pytanie dotyczy tego, jak dodawać rzeczy do instancji serwerów podczas ich tworzenia lub zmiany.

Kod konfiguracji serwera

Pierwsza generacja narzędzi do kodu jako infrastruktury skupiała się na automatyzacji konfiguracji serwera. Oto kilka tego typu narzędzi:

- Ansible¹
- CFEngine²
- Chef³
- Puppet⁴

1 <https://www.ansible.com>

2 <https://cfengine.com>

3 <https://www.chef.io>

4 <https://puppet.com>

- Saltstack⁵

Wiele z tych narzędzi korzysta z agenta zainstalowanego na każdym serwerze zgodnie z wzorcem pobierania konfiguracji serwera (patrz „Wzorzec: pobieranie konfiguracji serwera” na stronie 193). Udostępniają one agenta, którego można zainstalować jako usługę albo uruchamiać okresowo za pomocą zadania cron. Inne narzędzia przewidują uruchamianie ich z centralnego serwera i łączenie się z każdym zarządzanym serwerem, zgodnie z wzorcem wypychania („Wzorzec: wypychanie konfiguracji serwera” na stronie 192). Większość tych narzędzi pozwala zaimplementować zarówno wzorzec pobierania, jak i wypychania. W przypadku używania narzędzia, takiego jak Ansible, które jest przeznaczone do modelu wypychania, można zainstalować je wstępnie na serwerach i uruchamiać za pomocą crona. Jeśli z kolei używane jest narzędzie typu Chef lub Puppet, udostępniające agenta do wersji z modelem pobierania, można zamiast tego uruchomić polecenie z centralnego serwera, które zaloguje się i uruchomi klienta na każdej maszynie. Tak więc narzędzie nie narzuca wyboru wzorca.

Wielu dostawców narzędzi do konfigurowania serwerów oferuje serwery repozytoriów do przechowywania kodu konfiguracji. Przykłady to Ansible Tower, Chef Server i Puppetmaster. Mogą mieć one dodatkowe funkcje, takie jak rejestr konfiguracji (patrz „Rejestr konfiguracji” na stronie 93), pulpity nawigacji, rejestrowanie czy scentralizowane wykonywanie.

Wybór dostawcy udostępniającego wszechstronny ekosystem narzędzi zapewne upraszcza pracę zespołu zajmującego się infrastrukturą. Przydaje się jednak, jeśli można wymieniać elementy takiego ekosystemu na inne narzędzia, dzięki czemu zespół może wybierać to, co najbardziej odpowiada jego potrzebom. Na przykład, jeśli zespół używa wielu narzędzi integrujących się z rejestrem konfiguracji, może uznać, że autonomiczny rejestr konfiguracji ogólnego przeznaczenia będzie lepszy niż rejestr powiązany z narzędziem do konfiguracji serwera.

Znacząca część działań zespołu zajmującego się nowoczesną infrastrukturą dotyczy bazy kodu. W rozdziale 4 można znaleźć zasady i wskazówki, które mają zastosowanie do kodu serwera, kodu stosu itd.

Moduły kodu konfiguracji serwera

Kod konfiguracji serwera, jak każdy kod, wydaje się mieć skłonność do zmieniania się z czasem w rozległy bałagan. Uniknięcie tego wymaga dyscypliny i dobrych nawyków. Na szczęście narzędzia umożliwiają organizowanie i porządkowanie kodu w postaci oddzielnych jednostek.

Wiele narzędzi do konfiguracji serwerów pozwala organizować kod w moduły. Na przykład Ansible stosuje podręczniki, Chef grupuje przepisy w książkach kucharskich, a Puppet ma moduły zawierające manifesty. Moduły te można organizować, wersjonować i wydawać indywidualnie.

⁵ <https://www.saltstack.com>

Grupę modułów, które mają zostać zastosowane do serwera, można oznaczyć przy użyciu ról, definiujących cel takiej grupy. W odróżnieniu od innych koncepcji, większość dostawców narzędzi wydaje się być zgodna co do używania terminu *rola* w tym kontekście i nie wymyśla własnych określeń⁶.

Wiele z tych narzędzi obsługuje również rozszerzenia ich podstawowego modelu zasobów. Język narzędzia dostarcza model jednostek zasobów serwera, takich jak użytkownicy, pakiety i pliki. Na przykład można napisać własnego dostawcę LWRP (Chef Lightweight Resource Provider) i dodać zasób aplikacji Java, który będzie instalował i konfigurował aplikacje napisane przez nasz zespół programistów. Rozszerzenia te są podobne do modułów stosu (patrz rozdział 16).

Więcej informacji na temat modularyzacji infrastruktury, w tym wskazówki dotyczące dobrego projektowania, można znaleźć w rozdziale 15.

Projektowanie modułów kodu konfiguracji serwera

Każdy moduł kodu konfiguracji serwera (na przykład każda książka kucharska lub podręcznik) powinien być zaprojektowany i napisany pod kątem jednego, spójnego zagadnienia. Rada ta jest zgodna z zasadą projektowania oprogramowania mówiącą o *oddzielaniu zagadnień*⁷. Typowym podejściem jest posiadanie oddzielnego modułu dla każdej aplikacji.

Na przykład można by utworzyć moduł do zarządzania serwerem aplikacji, takim jak Tomcat. Kod w tym module instalowałby oprogramowanie Tomcat, konta użytkowników i grupy, używane do jego uruchamiania oraz foldery z odpowiednimi uprawnieniami dla dzienników i innych plików. Kod budowałby także pliki konfiguracyjne Tomcat, obejmujące konfigurację portów i ustawień. Ponadto kod modułu dokonywałby integracji serwera Tomcat z różnymi usługami, takimi jak agregacja dzienników, monitorowanie i zarządzanie procesami.

Wiele zespołów uznaje za przydatne projektowanie modułów konfiguracji serwera w różny sposób w zależności od ich zastosowania. Na przykład można podzielić swoje moduły na moduły bibliotek i moduły aplikacji. (W przypadku Chefa byłyby to książki kucharskie bibliotek i książki kucharskie aplikacji).

Moduł bibliotek zarządza podstawową koncepcją, która może być używana wielokrotnie przez moduły aplikacji. Wracając do przykładu z modulem Tomcat, można byłoby napisać moduł do pobierania parametrów i użyć go do konfiguracji instancji serwera Tomcat zoptymalizowanej do różnych celów.

Moduł aplikacji, zwany także modulem opakowującym, importuje jeden lub więcej modułów bibliotek i ustawia ich parametry w bardziej konkretnym celu. Zespół ShopSpinner mógłby mieć jeden moduł Servermaker, który instalowałby serwer Tomcat w celu uruchamiania aplikacji katalogu produktów i drugi moduł, który instalowałby serwer Tomcat dla

6 Byłoby zupełnie zrozumiałe, gdyby ludzie z firmy Chef nazwali rolę *czapkę*, dzięki czemu mogliby mówić, że serwery noszą czapkę *kucharską*. Jestem wdzięczny, że tego nie zrobili.

7 <https://oreil.ly/CPXDu>

aplikacji zarządzającej klientami⁸. Oba te moduły używałyby współdzielonej biblioteki, instalującej serwer Tomcat.

Wersjonowanie i promowanie kodu serwera

Podobnie jak w przypadku każdego kodu, należy mieć możliwość przetestowania i promocii kodu konfiguracji serwera do kolejnych środowisk przed zastosowaniem go do systemów produkcyjnych. Niektóre zespoły stosują wersjonowanie i promowanie wszystkich modułów kodu serwera łącznie, jako pojedynczej jednostki, budując je razem (patrz „Wzorzec: integracja projektów podczas budowania” na stronie 319) albo budując i testując je oddzielnie, a łącząc przed użyciem w środowiskach produkcyjnych (patrz „Wzorzec: integracja projektów podczas dostarczania” na stronie 322). Inne zespoły wersjonują i dostarczają każdy moduł niezależnie (patrz „Wzorzec: integracja projektów podczas stosowania” na stronie 324).

Jeśli zarządzamy każdym modulem jako odrębną jednostką, to musimy obsługiwać wszystkie zależności między nimi. Wiele narzędzi do konfiguracji serwerów pozwala określić zależności modułu w deskryptorze. Na przykład w książce kucharskiej Chefa zależności są wymienione w polu `depends` pliku metadanych książki⁹. Narzędzie do konfiguracji serwera używa tej specyfikacji do pobrania i zastosowania zależnych modułów do instancji serwera. Zarządzanie zależnościami modułów konfiguracji serwera działa tak samo, jak w przypadku systemów zarządzania pakietami oprogramowania, takimi jak pakiety RPM albo pip Pythona.

Oprócz tego trzeba przechowywać różne wersje kodu konfiguracji serwera i zarządzać nimi. Często występuje jedna wersja kodu, która jest aktualnie zastosowana do środowisk produkcyjnych oraz inne wersje, pozostające w fazach programowania i testowania.

Organizowanie, pakowanie i dostarczanie kodu infrastruktury jest dokładniej omówione w rozdziałach 18 i 19.

Role serwerów

Jak wspomniałem wcześniej, rola serwera jest sposobem definiowania grupy modułów konfiguracji serwera, które mają zostać zastosowane do serwera. Rola może także ustawiać niektóre parametry domyślne. Oto przykładowa rola `application-server`:

```
role: application-server
server_modules:
  - tomcat
  - monitoring_agent
  - logging_agent
  - network_hardening
```

⁸ W rozdziale 4 napisałem, że Servermaker jest fikcyjnym narzędziem do konfiguracji serwera, podobnym do Ansible, Chef i Puppet.

⁹ https://oreil.ly/_4eEP

```
parameters:  
  - inbound_port: 8443
```

Przypisanie tej roli do serwera powoduje zastosowanie konfiguracji serwera Tomcat, agentów monitorowania i rejestrowania oraz wzmocnienie sieci. Zostają też ustawione parametry portu przychodzącego – prawdopodobnie jest to coś, czego moduł `network_hardening` używa do otwierania tego portu w lokalnej zaporze, blokując jednocześnie całą resztę.

Role mogą się z czasem mieszać, dlatego należy używać ich w sposób konkretny i spójny. Bardziej korzystne może być definiowanie szczegółowych ról i łączenie ich podczas tworzenia określonych serwerów. Pewne serwery wymagają przypisania wielu ról, takich jak `ApplicationServer`, `MonitoredServer` i `PublicFacingServer`. Każda z tych ról obejmuje niewielką liczbę modułów serwera o wąskim przeznaczeniu.

Alternatywą jest tworzenie ról wyższego poziomu, obejmujących więcej modułów. W takim modelu każdy serwer ma wtedy zwykle jedną rolę, która może być dość specyficzna, na przykład `ShoppingServiceServer` lub `JiraServer`.

Typowym podejściem jest stosowanie dziedziczenia ról. Na początku definiuje się rolę podstawową, obejmującą oprogramowanie i konfigurację wspólną dla wszystkich serwerów. Taka rola może zawierać również wzmocnienie sieci, konta użytkowników administracyjnych oraz agentów monitorowania i rejestrowania. Bardziej konkretne role, takie jak serwery aplikacji czy hosty kontenerów, składają się z roli podstawowej i kilku dodatkowych modułów konfiguracji serwera oraz parametrów:

```
role: base-role  
  server_modules:  
    - monitoring_agent  
    - logging_agent  
    - network_hardening  
role: application-server  
  include_roles:  
    - base_role  
  server_modules:  
    - tomcat  
  parameters:  
    - inbound_port: 8443  
role: shopping-service-server  
  include_roles:  
    - application-server  
  server_modules:  
    - shopping-service-application
```

Powyższy kod przykładowy definiuje trzy role w układzie hierarchicznym. Rola `shopping-service-server` dziedziczy wszystko po roli `application-server` i dodaje moduł instalujący konkretną aplikację przewidzianą do wdrożenia. Rola `application-server` dziedziczy po roli `base-role` i dodaje serwer Tomcat oraz konfigurację portu sieciowego. Rola `base-role` definiuje podstawowy zestaw modułów konfiguracyjnych ogólnego przeznaczenia.

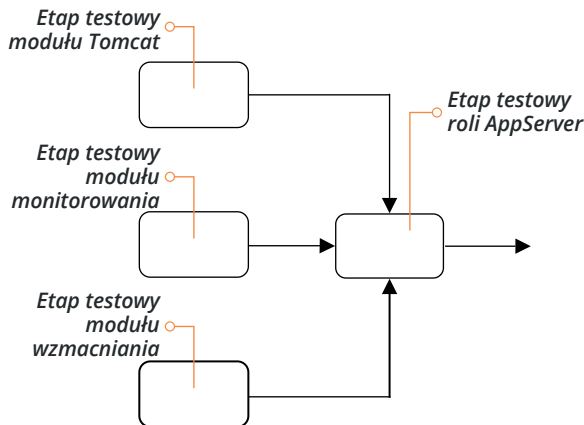
Testowanie kodu serwera

W rozdziale 8 wyjaśniłem, jak wygląda stosowanie zautomatyzowanego testowania i teorii CD do infrastruktury, a w rozdziale 9 opisałem różne podejścia do implementacji tego w przypadku stosów infrastruktury. Wiele koncepcji przedstawionych w obu tych rozdziałach ma zastosowanie do testowania kodu serwera.

Testowanie progresywne kodu serwera

Między testowaniem a projektem kodu zachodzi dynamiczny, samowzmacniający się związek. Łatwiej jest pisać i utrzymywać testy dla bazy kodu o przejrzystej strukturze. Pisanie testów i dbanie o ich wykonywalność zmusza do utrzymywania przejrzystej struktury. Wypychanie każdej zmiany do potoku, który wykonuje testy, pomaga zespołowi utrzymać dyscyplinę podczas nieustannej refaktoryzacji w celu minimalizacji długu technicznego i projektowego.

Opisana wcześniej struktura ról serwera, składających się z modułów konfiguracji serwera, które same mogą być zorganizowane w postaci modułów bibliotek i aplikacji, pasuje doskonale do strategii testowania progresywnego (patrz „Testowanie progresywne” na stronie 109). Seria etapów potoku może testować każdy z nich na kolejnych etapach integracji, jak to jest pokazane na rysunku 11-3.



Rysunek 11-3 Testowanie progresywne modułów kodu serwera

Oddzielny etap testuje każdy moduł kodu w momencie, gdy ktoś dokona w nim zmiany. Rola serwera również ma swój etap testów. Etap ten uruchamia testy, gdy jeden z modułów używanych przez rolę zostanie zmieniony i przejdzie swój własny etap. Ponadto etap testowania roli jest wykonywany, gdy ktoś zmieni kod roli, na przykład dodając lub usuwając jakiś moduł albo zmieniając parametr.

Co testować w przypadku kodu serwera

Trudno jest zdecydować, co należy testować w przypadku kodu serwera. Wracamy tu do pytania, czy testowanie kodu deklaratywnego ma jakąkolwiek wartość (patrz „Wyzwanie: testy kodu deklaratywnego mają często małą wartość” na stronie 105). Moduły kodu serwera, zwłaszcza w dobrze zaprojektowanej bazie kodu, są zwykle małe, proste i nastawione na konkretny cel. Na przykład moduł do instalacji JVM Javy może być pojedynczą instrukcją z kilkoma parametrami:

```
package:
  name: java-${JAVA_DISTRIBUTION}
  version: ${JAVA_VERSION}
```

W praktyce nawet pozornie proste moduły instalacji pakietów mogą zawierać więcej kodu niż w podanym przykładzie. Taki kod może dostosowywać ścieżki plików, pliki konfiguracyjne albo dodawać konto użytkownika. Ale często większość testu polega na przepisaniu samego kodu. Testy powinny koncentrować się na typowych problemach, zmiennych wynikach oraz kombinacjach kodu.

Należy zwracać uwagę na typowe problemy, czyli na to, co w praktyce najczęściej źle działa, i sprawdzać podstawową poprawność. W przypadku prostej instalacji pakietu dobrze jest sprawdzić, czy polecenie jest dostępne w domyślnej ścieżce użytkownika. Taki test może wywoływać polecenie i upewniać się, że zostało wykonane:

```
given command 'java -version' {
  its(exit_status) { should_be 0 } }
```

Jeśli pakiet może dawać radykalnie różne wyniki w zależności od przekazywanych do niego parametrów, potrzebne są testy potwierdzające jego prawidłowe zachowanie w różnych przypadkach. W przypadku pakietu instalacji JVM można by uruchomić test z różnymi dystrybucjami języka Java, aby upewnić się, że polecenie java jest dostępne bez względu na wybraną wersję.

W praktyce wartość testów wzrasta wraz ze zwiększaniem liczby integrowanych elementów. Można więc pisać i uruchamiać więcej testów dla roli serwera aplikacji niż dla modułów należących do tej roli.

Tak jest szczególnie wtedy, gdy ma miejsce integracja i interakcja modułów. Moduł, który instaluje serwer Tomcat, raczej nie koliduje z modułem instalującym agenta monitorowania, ale moduł wzmacniania sieci może już powodować problem. W tym przypadku można mieć test wykonywany dla roli serwera aplikacji, który potwierdzi, że serwer aplikacji działa i akceptuje żądania nawet po zablokowaniu portów.

Jak testować kod serwera

Większość zautomatyzowanych testów kodu serwera polega na uruchamianiu poleceń na serwerze lub w instancji kontenera i sprawdzaniu wyników. Testy te mogą potwierdzać

istnienie i stan zasobów, takich jak pakiety, pliki, konta użytkowników i działające procesy. Testy mogą również sprawdzać rezultaty; na przykład łącząc się z portem sieciowym w celu przekonania się, czy usługa zwraca oczekiwany wynik.

Popularne narzędzia do testowania warunków na serwerze to Inspec¹⁰, Serverspec¹¹ i Terratest¹².

Strategie testowania całych stosów infrastruktury obejmują uruchamianie testów w trybach offline i online (patrz „Etap testowania offline dla stosów” na stronie 125 i „Etap testowania online dla stosów” na stronie 128). Testowanie online oznacza uruchomienie wszystkiego na platformie infrastruktury. Zwykle kod serwera można testować w trybie offline, używając kontenerów lub lokalnych maszyn wirtualnych.

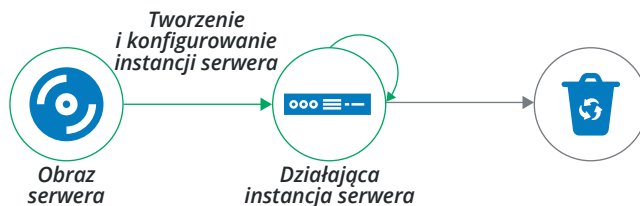
Programista infrastruktury pracujący lokalnie może utworzyć instancję kontenera lub lokalną maszynę wirtualną przy użyciu minimalnej instalacji systemu operacyjnego, zastosować kod konfiguracji serwera, a następnie uruchamiać testy.

Etap potoku testujące kod serwera mogą robić to samo, uruchamiając instancję kontenera lub maszyny wirtualnej na agencie (na przykład węzeł Jenkins wykonujący zadanie). Alternatywą jest utworzenie przez etap autonomicznej instancji kontenera w klastrze kontenerów. Działa to dobrze, gdy sama orkiestracja potoku jest skonteneryzowana.

Można orkiestrować testowanie kodu serwera zgodnie ze wskazówkami dla stosów (patrz „Orkiestracja testów” na stronie 143). Obejmuje to pisanie skryptów konfiguracyjnych wymagania wstępne testowania, takie jak instancje kontenerów, przed uruchomieniem testów i raportowaniem wyników. Podczas testowania lokalnego należy uruchamiać te same skrypty, co w przypadku usługi potoku, aby wyniki były spójne.

Tworzenie nowej instancji serwera

Opisany wcześniej podstawowy cykl życia serwera zaczyna się od utworzenia nowej instancji serwera. Dłuższy cykl życia obejmuje etapy tworzenia i aktualizacji obrazów serwera, na podstawie których tworzone są instancje serwera, ale zostawimy ten temat na inny rozdział (rozdział 13). Na razie cykl życia serwera zaczyna się od utworzenia instancji, jak to jest pokazane na rysunku 11-4.



Rysunek 11-4 Tworzenie instancji serwera

¹⁰ <https://www.inspec.io>

¹¹ <https://serverspec.org>

¹² <https://oreil.ly/W6H5Q>

Pełny proces tworzenia serwera obejmuje wyposażenie jego instancji, aby był w pełni gotowy do używania. Oto niektóre działania związane z tworzeniem i wyposażaniem instancji serwera:

- Alokacja zasobów infrastruktury potrzebnych do utworzenia instancji serwera. W tym celu wybierany jest fizyczny serwer z puli albo serwer hosta do uruchomienia jako maszyna wirtualna. W przypadku maszyny wirtualnej to hiperwizor uruchomiony w hoście alokuje pamięć i pozostałe zasoby. Proces ten może również alokować miejsce na woluminy dysków dla instancji serwera.
- Instalacja systemu operacyjnego i wstępnego oprogramowania. System operacyjny może zostać skopiowany na wolumin dysku, jak podczas rozruchu z obrazu serwera typu AMI. Alternatywnie może zostać wykonany proces instalacji, który wybierze i skopiuje pliki do nowej instancji, na przykład za pomocą instalatora obsługującego skrypty. Przykładami instalatorów systemu operacyjnego obsługujących skrypty są Red Hat Kickstart¹³, Solaris JumpStart¹⁴, Debian Preseed¹⁵ i plik odpowiedzi instalacji systemu Windows¹⁶.
- Zastosowanie dodatkowej konfiguracji do instancji serwera. Na tym etapie proces uruchamia narzędzie do konfiguracji serwera, zgodnie z wzorcem opisanym w podrozdziale „Jak stosować kod konfiguracji serwera” na stronie 191.
- Skonfigurowanie i dołączenie zasad oraz zasobów sieciowych. Proces ten może obejmować przypisanie serwera do bloku adresów sieci, utworzenie tras i dodanie reguł zapory.
- Zarejestrowanie instancji serwera w usługach, na przykład dodanie nowego serwera do systemu monitorowania.

Punkty te nie wykluczają się wzajemnie – proces tworzenia serwera może wykorzystywać jedną lub wiele metod uruchamiania tych działań.

Istnieje kilka mechanizmów, których można używać do wyzwalania utworzenia instancji serwera. Przyjrzymy się teraz każdemu z tych mechanizmów.

Ręczne tworzenie nowej instancji serwera

Platformy infrastruktury udostępniają narzędzia do tworzenia nowych serwerów, zazwyczaj w postaci interfejsu WWW, narzędzia wiersza polecenia, a czasami aplikacji GUI. Podczas tworzenia każdego serwera trzeba wybrać odpowiednie dla siebie opcje, w tym obraz źródłowy, alokowane zasoby i szczegóły sieci:

```
$ mycloud server new \  
  --source-image=stock-linux-1.23 \  
  --source-key=...
```

¹³ <https://oreil.ly/k6d53>

¹⁴ <https://oreil.ly/flDLL>

¹⁵ <https://oreil.ly/tXdcu>

¹⁶ <https://oreil.ly/oOq0C>

```
--memory=2GB \  
--vnet=appservers
```

O ile dobrze jest pobawić się z różnymi interfejsami użytkownika i narzędziami wiersza polecenia, aby poeksperymentować z platformą, nie jest to właściwa metoda tworzenia serwerów. Te same zasady, które zostały omówione wcześniej w odniesieniu do tworzenia i konfigurowania stosów (patrz „Wzorce i antywzorce konstruowania stosów” na stronie 52), mają zastosowanie do serwerów. Ręczne ustawianie opcji (opisane w podrozdziale „Antywzorzec: ręczne parametry stosu” na stronie 74) sprzyja błędom i prowadzi do niespójnie skonfigurowanych serwerów, nieprzewidywalnych systemów i zbyt wielu prac konserwacyjnych.

Tworzenie serwera przy użyciu skryptu

Stosowanie skryptów zapewnia tworzenie spójnych serwerów. Skrypt opakuje narzędzie wiersza polecenia lub wykorzystuje API platformy infrastruktury do utworzenia instancji serwera, a opcje konfiguracji są ustawione w jego kodzie albo w plikach konfiguracyjnych. Skrypt tworzący serwery nadaje się do wielokrotnego użytku, jest spójny i przejrzysty. Jest to ta sama koncepcja, co w przypadku wzorca parametrów skryptowych dla stosów (patrz „Wzorzec: parametry skryptowe” na stronie 78):

```
mycloud server new \  
--source-image=stock-linux-1.23 \  
--memory=2GB \  
--vnet=appservers
```

Powyższy skrypt wygląda tak samo, jak wcześniejszy przykład wiersza polecenia. Ale ponieważ jest to skrypt, inni członkowie zespołu mogą zobaczyć, jak utworzyłem swój serwer. Mogą też sami utworzyć więcej serwerów i mieć pewność, że będą one działały identycznie jak mój.

Zanim pojawił się produkt Terraform i inne narzędzia do zarządzania stosami, większość zespołów, z którymi pracowałem, tworzyło serwery za pomocą napisanych w tym celu skryptów. Zazwyczaj były to skrypty konfigurowane przy użyciu plików konfiguracyjnych, jak we wzorcu konfiguracji stosu (patrz „Wzorzec: pliki konfiguracyjne stosu” na stronie 81). Jednak ulepszanie i naprawianie tych skryptów zajmowało za dużo czasu.

Tworzenie serwera przy użyciu narzędzia zarządzania stosem

Za pomocą narzędzia do zarządzania stosem, omówionego w rozdziale 5, można zdefiniować serwer w kontekście pozostałych zasobów infrastruktury, jak to jest pokazane w przykładzie 11-1. Narzędzie tworzy i aktualizuje instancję serwera przy użyciu API platformy.

Przykład 11-1 Kod stosu definiujący serwer

```
server:
  source_image: stock-linux-1.23
  memory: 2GB
  vnet: ${APPSERVER_VNET}
```

Jest kilka powodów, dla których używanie narzędzia stosu do tworzenia serwerów jest tak wygodne. Jednym z nich jest to, że narzędzie bierze na siebie logikę, którą trzeba samodzielnie implementować w skrypcie, na przykład sprawdzanie błędów. Inną zaletą stosu jest obsługiwanie przez stos integracji z innymi elementami infrastruktury; dotyczy to na przykład dołączenia serwera do struktur sieciowych i pamięci masowej zdefiniowanej w stosie. Kod w przykładzie 11-1 ustawia parametr `vnet` używając zmiennej `${APPSERVER_VNET}`, która prawdopodobnie odwołuje się do struktury sieciowej zdefiniowanej w innej części kodu stosu.

Konfigurowanie automatycznego tworzenia serwerów przez platformę

Większość platform infrastruktury potrafi automatycznie tworzyć nowe instancje serwerów w określonych okolicznościach. Dwa typowe przypadki to *automatyczne skalowanie*, czyli dodawanie serwerów w celu obsługi zwiększonego obciążenia oraz *automatyczne odzyskiwanie*, czyli zastępowanie instancji serwera w przypadku awarii. Można to zwykle zdefiniować za pomocą kodu stosu, jak w przykładzie 11-2.

Przykład 11-2 Kod stosu dla zautomatyzowanego skalowania

```
server_cluster:
  server_instance:
    source_image: stock-linux-1.23
    memory: 2GB
    vnet: ${APPSERVER_VNET}
  scaling_rules:
    min_instances: 2
    max_instances: 5
    scaling_metric: cpu_utilization
    scaling_value_target: 40%
  health_check:
    type: http
    port: 8443
    path: /health
    expected_code: 200
    wait: 90s
```

Kod w tym przykładzie każe platformie utrzymywać co najmniej 2 i co najwyżej 5 działających instancji serwera. Platforma dodaje i usuwa instancje według potrzeb, aby wykorzystanie CPU pozostawało w nich na poziomie zbliżonym do 40%.

Definicja obejmuje również sprawdzenie stanu. Sprawdzenie polega na wysłaniu żądania HTTP do /health na port 8443. Kod odpowiedzi HTTP 200 oznacza, że serwer jest w dobrym stanie. Jeśli serwer nie zwróci spodziewanego kodu w ciągu 90 sekund, platforma przyjmie, że serwer przestał działać, więc zniszczy go i utworzy nową instancję.

Tworzenie serwera przy użyciu sieciowego narzędzia wyposażania

W rozdziale 3 wspomniałem o chmurach bare-metal, dynamicznie udostępniających serwery sprzętowe. Taki proces składa się zwykle z trzech następujących etapów:

1. Wybór nieużywanego serwera fizycznego i uruchomienie go w trybie „instalacji sieciowej”, obsługiwanej przez oprogramowanie układowe serwera (na przykład rozruch PXE¹⁷).
2. Rozruch serwera przez instalator sieciowy przy użyciu prostego obrazu rozruchowego systemu operacyjnego, w celu zainicjowania instalacji systemu.
3. Pobranie obrazu instalacji systemu operacyjnego i skopiowanie go na podstawowy dysk twardy.
4. Ponowny rozruch serwera w celu uruchomienia systemu operacyjnego w trybie konfiguracji, za pomocą nienadzorowanego skryptowego instalatora systemu operacyjnego.

Dostępnych jest wiele narzędzi do zarządzania tym procesem, w tym:

- Crowbar¹⁸
- Cobbler¹⁹
- FAI, czyli Fully Automatic Installation²⁰
- Foreman²¹
- MAAS²²
- Rebar²³
- Tinkerbell²⁴

Zamiast rozruchu z obrazu instalacji systemu operacyjnego można zastosować rozruch z przygotowanego obrazu serwera. Stwarza to możliwość zaimplementowania kilku innych metod przygotowywania serwerów, które zostaną przedstawione w następnym podrozdziale.

¹⁷ <https://oreil.ly/j1lTS>

¹⁸ <http://opencrowbar.github.io>

¹⁹ <http://cobbler.github.io>

²⁰ <http://fai-project.org>

²¹ <http://theforeman.org>

²² <https://maas.io>

²³ <https://rebar.digital>

²⁴ <https://tinkerbell.org>



Zdarzenia FaaS mogą pomóc w udostępnianiu serwera

Bezserwerowy kod FaaS może odgrywać rolę przy wyposażaniu nowego serwera. Platforma może uruchamiać ten kod na różnych etapach procesu, przed, w trakcie i po utworzeniu nowej instancji. Przykładem może być przypisanie zasad bezpieczeństwa do nowego serwera, zarejestrowanie go w usłudze monitorowania albo uruchomienie narzędzia do konfiguracji serwera.

Wstępne budowanie serwerów

Jak już napisałem wcześniej (patrz „Skąd pochodzą rzeczy trzymane na serwerze” na stronie 165), jest kilka źródeł zawartości nowego serwera, w tym instalacja systemu operacyjnego, pakiety pobrane z repozytorium i niestandardowe pliki konfiguracyjne oraz pliki aplikacji skopiowane do serwera.

Chociaż można zebrać to wszystko w momencie tworzenia serwera, istnieje również kilka sposobów wcześniejszego przygotowania zawartości serwera. Metody te pozwalają zoptymalizować proces tworzenia serwera, przyspieszając go i upraszczając, a do tego ułatwiają tworzenie wielu spójnych instancji. Za chwilę przedstawię kilka podejść do tego tematu. W dalszej części przedstawię opcje stosowania konfiguracji przed, w trakcie i po zakończeniu procesu wyposażania.

Klonowanie serwera „na gorąco”

Większość platform infrastruktury pozwala w prosty sposób zduplikować działający serwer. Klonowanie serwera w ten sposób jest szybkie i łatwe, a do tego zapewnia, że serwery są spójne z momentem klonowania.

Klonowanie działającego serwera może się przydać, gdy trzeba zbadać serwer albo rozwiązać towarzyszący mu problem bez zakłócania jego pracy. Jednak niesie to ze sobą pewne ryzyko. Po pierwsze, kopia serwera produkcyjnego utworzona do eksperymentów może wpłynąć na środowisko produkcyjne. Na przykład, jeśli sklonowany serwer aplikacji ma skonfigurowane korzystanie z produkcyjnej bazy danych, to można niechcący zanieczyścić lub uszkodzić dane produkcyjne.

Sklonowane serwery działające w środowisku produkcyjnym zawierają zwykle dane historyczne z oryginalnego serwera, takie jak wpisy w pliku dziennika. Zanieczyszczenie tych danych może wprowadzać w błąd podczas rozwiązywania problemów. Widziałem ludzi spędzających czas na głowieniu się nad komunikatami o błędach, które okazały się pochodzić z innego serwera niż debugowany.

Ponadto sklonowane serwery nie są tak naprawdę powtarzalne. Dwa serwery sklonowane z tego samego rodzica w różnych momentach będą inne i nie można się cofnąć, żeby zbudować trzeci serwer i wiedzieć z całą pewnością, czy i jak różni się on od wszystkich pozostałych. Sklonowane serwery są źródłem dryfu konfiguracji (patrz „Dryf konfiguracji” na stronie 17).

Używanie migawek serwera

Zamiast tworzyć nowe instancje przez bezpośrednie klonowanie działającego serwera, można zrobić jego migawkę i wykorzystać ją do budowy nowych serwerów. I w tym przypadku większość platform infrastruktury udostępnia polecenia i wywołania API umożliwiające łatwe tworzenie statycznych obrazów, będących migawkami serwerów. Taka metoda pozwala utworzyć tyle serwerów, ile tylko się chce, dając pewność, że każdy z nich będzie identyczny jak obraz początkowy.

Tworzenie serwerów na podstawie migawki wykonanej dla działającego serwera ma jednak wiele innych wad klonowania „na gorąco”. Migawka może być zanieczyszczona dziennikami, konfiguracją i innymi danymi oryginalnego serwera. Bardziej efektywne jest utworzenie czystego obrazu serwera od podstaw.

Tworzenie czystego obrazu serwera

Obraz serwera to migawka utworzona na podstawie czystego, znanego źródła, umożliwiająca dzięki temu tworzenie wielu spójnych instancji serwera. Można ją wykonać, korzystając z tych samych funkcji platformy infrastruktury, co w przypadku tworzenia migawek działającego serwera, tyle że oryginalna instancja serwera nie występuje nigdy jako część ogólnego systemu. Działając w ten sposób można mieć pewność, że każdy nowy serwer będzie czysty.

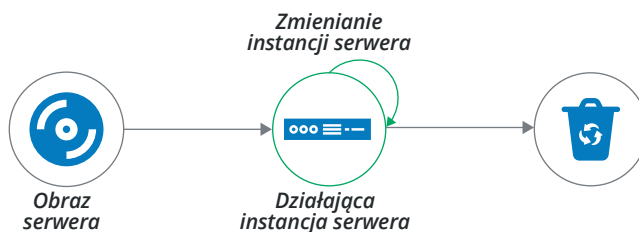
Typowy proces tworzenia obrazu serwera obejmuje:

1. Utworzenie nowej instancji serwera na podstawie znanego źródła, takiego jak obraz instalacyjny dostawcy systemu operacyjnego lub obraz dostarczony w tym celu przez platformę infrastruktury.
2. Zastosowanie kodu konfiguracyjnego serwera lub uruchomienie skryptów na serwerze. Efektem może być instalacja standardowych pakietów i agentów potrzebnych na wszystkich serwerach, wzmocnienie konfiguracji pod kątem bezpieczeństwa oraz zastosowanie wszystkich najnowszych poprawek.
3. Wykonanie migawki serwera w celu uzyskania obrazu nadającego się jako źródło do tworzenia wielu instancji serwera. Może to obejmować działania służące oznaczeniu migawki jako obrazu do tworzenia nowych serwerów. Może też obejmować tagowanie i wersjonowanie pomocne w zarządzaniu biblioteką wielu obrazów serwera.

Obrazy serwera są nazywane czasem *złotymi obrazami*. Chociaż niektóre zespoły tworzą takie obrazy ręcznie, zapewne według zapisanej listy kontrolnej działań, czytelnicy tej książki od razu widzą korzyści płynące z automatyzacji tego procesu i zarządzania obrazami serwerów jako kodem. Tworzenie i dostarczanie obrazów serwerów jest tematem rozdziału 13.

Konfigurowanie nowej instancji serwera

W tym rozdziale do tej pory opisałem, jakie są elementy serwera, skąd pochodzą, jakie są metody tworzenia nowych instancji serwerów oraz jaką wartość mają wstępnie zbudowane obrazy serwerów. Ostatnią częścią procesu tworzenia i wyposażania serwera jest zastosowanie kodu automatycznej konfiguracji serwera do nowych serwerów. Jest kilka momentów, w których można to zrobić, co widać na rysunku 11-5.



Rysunek 11-5 Punkty cyklu życia serwera, w których można zastosować konfigurację

Konfigurację można zastosować podczas tworzenia obrazu serwera, podczas tworzenia instancji serwera z obrazu oraz podczas pracy serwera:

Konfigurowanie obrazu serwera

Konfiguracja jest stosowana podczas budowy obrazu serwera, jeśli ma on służyć do tworzenia wielu instancji serwera. Można traktować to jako jednorazowe wykonanie konfiguracji, która będzie używana wielokrotnie. Często nazywa się to *wypiekaniem* (baking) obrazu serwera.

Konfigurowanie nowej instancji serwera

Konfiguracja jest stosowana podczas tworzenia nowej instancji serwera. Można traktować to jako wielokrotne wykonywanie konfiguracji. Czasem nazywa się to *smażeniem* (frying) instancji serwera.

Konfigurowanie działającej instancji serwera

Konfigurowanie ma miejsce, gdy serwer jest już używany. Typowym powodem takiego działania bywa konieczność wprowadzenia zmian, na przykład zastosowanie poprawek zabezpieczeń. Inną przyczyną może być cofnięcie zmian dokonanych poza zautomatyzowaną konfiguracją w celu wymuszenia spójności. Niektóre zespoły stosują również taką konfigurację, aby przekształcić istniejący już serwer, na przykład zmienić serwer WWW w serwer aplikacji.

Ostatnia z tych opcji, konfigurowanie działającej instancji serwera, jest zwykle stosowana w celu zmiany serwera, co będzie tematem rozdziału 12. Pierwsze dwie opcje odnoszą się do konfiguracji podczas tworzenia nowego serwera, więc zdecydowanie wchodzą w zakres tego rozdziału. Podstawowe pytanie odnosi się do wyboru właściwego momentu zastosowania konfiguracji do nowego serwera – czy należy wybrać smażenie czy pieczenie?

Smażenie instancji serwera

Jak już wyjaśniłem, smażenie serwera oznacza stosowanie konfiguracji podczas tworzenia jego nowej instancji. Można doprowadzić to do skrajności, utrzymując obraz każdego serwera w postaci minimalnej i dodając wszystko, co jest specyficzne dla niego, dopiero w momencie wyposażania. Taka metoda oznacza, że nowe serwery zawierają zawsze najnowsze zmiany, w tym poprawki systemowe, najnowsze wersje pakietów oprogramowania i aktualne opcje konfiguracji. Smażenie serwera jest przykładem integracji na etapie dostarczania (patrz „Wzorzec: integracja projektów podczas dostarczania” na stronie 322).

Takie podejście upraszcza zarządzanie obrazami. Obrazów nie ma dużo, prawdopodobnie tylko po jednym dla każdej kombinacji sprzętu i konkretnej wersji systemu operacyjnego używanej w infrastrukturze – na przykład, jedna dla 64-bitowej wersji Windows 2019 i po jednej dla 32-bitowej i 64-bitowej wersji Ubuntu 18.x. Obrazy nie muszą być aktualizowane zbyt często, ponieważ niewiele się w nich zmienia. A bez względu na wszystko i tak można zastosować najnowsze poprawki podczas wyposażania każdego serwera.

Niektóre zespoły, z którymi pracowałem, skupiały się na smażeniu na wczesnym etapie wdrażania automatyzacji infrastruktury. Konfigurowanie narzędzi i procesów do zarządzania obrazami serwerów wymaga dużo wysiłku. Umieszczaliśmy to zadanie na naszej liście prac do wykonania zaraz po budowie podstawowego zarządzania infrastrukturą.

W innych sytuacjach smażenie ma sens, gdyż serwery są bardzo zmienne. Znana mi firma hostingowa oferuje klientom do wyboru dużą liczbę niestandardowych serwerów. Firma utrzymuje minimalistyczne obrazy bazowe serwerów i wkłada więcej wysiłku w automatyzację, która przeprowadza konfigurację każdego nowego serwera.

Jest kilka potencjalnych problemów z instalowaniem i konfigurowaniem elementów serwera podczas tworzenia każdej instancji. Obejmują one:

Prędkość

Działania wykonywane podczas budowy serwera wydłużają jej czas trwania. Ten dodatkowy czas stanowi szczególny problem w przypadku uruchamiania serwerów w celu obsługi skoków obciążenia lub przywrócenia pracy po awarii.

Wydażność

Konfigurowanie serwera często obejmuje pobieranie przez sieć pakietów z repozytorium. Może to być nieekonomiczne i powolne. Na przykład, jeśli trzeba w krótkim czasie uruchomić 20 serwerów i każdy z nich potrzebuje pobrać te same poprawki i instalatory aplikacji, to jest to nieekonomiczne.

Zależności

Konfigurowanie serwera wymaga zwykle dostępu do repozytoriów artefaktów i innych systemów. Jeśli któreś z nich są w trybie offline lub nieosiągalne, to nie można utworzyć nowego serwera. Może to być szczególnie dotkliwe w sytuacjach awaryjnych, gdy trzeba szybko odbudować dużą liczbę serwerów. W takich przypadkach urządzenia sieciowe lub repozytoria również mogą nie działać, komplikując grafik wyznaczający prawidłową kolejność restartowania lub odbudowy poszczególnych systemów.

Pieczenie obrazów serwera

Na drugim końcu spektrum tworzenia serwerów leży konfigurowanie niemal wszystkiego w obrazie serwera. Tworzenie serwerów odbywa się wówczas bardzo szybko i prosto, ponieważ wystarczy zastosować obraz specyficzny dla danej instancji. Pieczenie obrazu serwera jest przykładem integracji podczas tworzenia (patrz „Wzorzec: integracja projektów podczas budowania” na stronie 319).

Pieczenie ma zalety, których brakuje smażeniu. Wstawianie konfiguracji do obrazu serwera jest szczególnie przydatne w przypadku systemów mających dużą liczbę podobnych instancji oraz wtedy, gdy trzeba często i szybko tworzyć serwery.

Jednym z wyzwań towarzyszących pieczeniu obrazów serwera jest konieczność skonfigurowania narzędzi i zautomatyzowanych procesów ułatwiających aktualizowanie i wdrażanie nowych wersji. Na przykład w przypadku pojawienia się nowej poprawki zabezpieczeń systemu operacyjnego albo kluczowych pakietów „zapieczonych” w obrazie serwera, trzeba mieć możliwość utworzenia nowych obrazów i wdrożenia ich w miejscach istniejących serwerów szybko i przy minimalnych zakłóceniach. O tym, jak można to zrobić, opowiem w rozdziale 13.

Inny problem związany z pieczeniem obrazów serwera dotyczy prędkości. Nawet w przypadku dojrzałego, zautomatyzowanego procesu aktualizacji obrazów, utworzenie i wdrożenie nowego obrazu potrafi zająć wiele minut – zwykle od 10 do 60. Można temu zaradzić, wprowadzając zmiany do działających serwerów (temat rozdziału 12) lub stosując proces będący połączeniem pieczenia i smażenia.

Łączenie pieczenia i smażenia

W praktyce większość zespołów używa kombinacji pieczenia i smażenia do konfigurowania nowych serwerów. Można wypośredkować między elementami, które należy konfigurować w obrazach serwera, a tymi, które należy stosować podczas tworzenia instancji serwera. Niektóre pozycje konfiguracji mogą występować w obu częściach tego procesu.

Głównymi kwestiami wpływającymi na decyzję, kiedy jest ten właściwy moment na zastosowanie konkretnej konfiguracji, są czas, jaki jest do tego potrzebny i częstotliwość zmian. Rzeczy, których stosowanie trwa dłużej i które zmieniają się rzadziej, są wyraźnymi kandydatami do zapieczenia w obrazie serwera. Na przykład można zainstalować oprogramowanie serwera aplikacji w obrazie serwera, co znacznie przyspieszy uruchamianie wielu instancji tego serwera i do tego zredukuje obciążenie sieci podczas tego procesu.

Po drugiej stronie kompromisu są rzeczy, których instalacja trwa krócej i które częściej się zmieniają, więc lepiej jest je smażyć. Przykładem są aplikacje tworzone lokalnie. Jednym z najpopularniejszych przypadków instalowania nowych serwerów na żądanie jest testowanie w ramach procesu wydawania oprogramowania.

Bardzo wydajne zespoły programowania mogą wypuszczać codziennie dziesiątki albo i więcej kompilacji swoich aplikacji, opierając się na procesie CI, który automatycznie wdraża i testuje każdą kompilację. Pieczenie nowego obrazu serwera dla każdej nowej

kompilacji aplikacji jest zbyt powolne dla tego rodzaju tempa pracy, dlatego lepsze jest wdrażanie aplikacji podczas tworzenia serwera testowego.

Innym sposobem łączenia pieczenia i smażenia przez zespół jest zapiekane wszystkiego, co się da w obrazach serwera, a smażenie tylko wersji zaktualizowanych. Zespół może piec zaktualizowane obrazy serwera w wolniejszym tempie, na przykład raz na tydzień lub co miesiąc. Gdy pojawia się potrzeba aktualizacji czegoś, co zwykle jest zapiekane w obrazie, na przykład poprawka zabezpieczeń lub ulepszenie konfiguracji, można umieścić to w procesie tworzenia serwera i usmażyć na upieczonym obrazie.

Gdy nadchodzi czas upieczenia zaktualizowanego obrazu, zespół umieszcza w nim aktualizacje i usuwa z procesu tworzenia. Taka metoda pozwala zespołowi wprowadzać nowe zmiany szybko, mniejszym nakładem pracy. Zespoły często używają tej metody w połączeniu z ciągłym stosowaniem kodu (patrz „Wzorzec: ciągła synchronizacja konfiguracji” na stronie 187) do aktualizacji istniejących serwerów bez konieczności ich odbudowywania.

Stosowanie konfiguracji serwera podczas tworzenia serwera

Większość narzędzi używanych do tworzenia serwerów przy użyciu omówionych już wcześniej metod (patrz „Tworzenie nowej instancji serwera” na stronie 173), takich jak narzędzie wiersza polecenia, API platformy czy narzędzie do zarządzania stosem, umożliwia zastosowanie kodu konfiguracji serwera. Na przykład narzędzie do zarządzania stosem powinno oferować składnię obsługującą popularne narzędzia lub pozwalającą uruchomić dowolne polecenie w nowej instancji serwera, jak w przykładzie 11-3.

Przykład 11-3 *Kod stosu uruchamiający moje fikcyjne narzędzie do konfiguracji serwera*

```
server:
  source_image: stock-linux-1.23
  memory: 2GB
  vnet: ${APPSERVER_VNET}
  configure:
    tool: servermaker
    code_repo: servermaker.shopspinner.xyz
    server_role: appserver
    parameters:
      app_name: catalog_service
      app_version: 1.2.3
```

Powyższy kod uruchamia narzędzie Servermaker i przekazuje do niego nazwę hosta serwera, na którym jest kod konfiguracyjny, rolę, którą należy zastosować do serwera (appserver) oraz pewne parametry, które trzeba przekazać do kodu konfiguracji serwera (app_name i app_version).

Niektóre narzędzia pozwalają nawet osadzić bezpośrednio w kodzie stosu kod konfiguracji serwera albo polecenia powłoki do wykonania. Można się skusić i użyć tej metody

do zaimplementowania logiki konfiguracji serwera. W prostych przypadkach bywa to dobrym pomysłem, ale w większości sytuacji taki kod robi się duży i skomplikowany. Lepiej jest więc wyodrębnić go, aby baza kodu była czysta i łatwa do utrzymania.

Podsumowanie

W tym rozdziale omówiłem różne aspekty tworzenia i wyposażania nowych serwerów. Rzeczy przechowywane na serwerze należą do takich kategorii, jak oprogramowanie, konfiguracja i dane. Są one zwykle pobierane z podstawowej instalacji systemu operacyjnego i pakietów pochodzących z różnych repozytoriów, w tym zarządzanych przez dostawców systemu operacyjnego oraz języków, firmy trzecie i wewnętrzne zespoły. Serwer powstaje zwykle przez wybranie zawartości obrazu serwera i użycie narzędzia do konfiguracji serwera w celu zastosowania dodatkowych pakietów i ustawień.

Aby utworzyć serwer, można uruchomić narzędzie wiersza polecenia lub użyć interfejsu użytkownika, ale najlepiej jest skorzystać z procesu opartego na kodzie. Teraz jest już mniej prawdopodobne, że czytelnik napisze w tym celu niestandardowy skrypt, a bardziej, że użyje narzędzia do zarządzania stosem. Chociaż opisałem różne podejścia do budowania serwerów, zwykle zalecam tworzenie obrazów serwerów.

Zarządzanie zmianami w serwerach

Wiele organizacji i zespołów skupia się na procesach i narzędziach do budowania serwerów i pozostałej infrastruktury, a zaniedbuje zmiany. Gdy jest potrzebna zmiana – w celu usunięcia problemu, zastosowania poprawek zabezpieczeń albo aktualizacji oprogramowania – traktują to jako nietypowe zdarzenie. Jeśli każda zmiana jest czymś wyjątkowym, nie można ich zautomatyzować. To z powodu takiego myślenia wiele organizacji ma nie-spójne, połatanne systemy. To dlatego wielu z nas spędza czas pomiędzy wykonywaniem nudnych codziennych zadań i gaszeniem pożarów.

Jedyną pewną rzeczą, jaką można powiedzieć o systemach informatycznych, jest to, że się zmieniają. Jeśli zdefiniujemy je jako kod, będziemy uruchamiać go na dynamicznych platformach infrastruktury i dostarczać za pomocą potoku zmian, to sprawimy, że dokonywanie zmian będzie łatwe i rutynowe. Jeśli systemy są zawsze tworzone i zmieniane tylko za pomocą kodu i potoków, to możemy zagwarantować ich spójność i mieć pewność, że są zgodne z wszystkimi wymaganymi regułami.

W podrozdziałach „Co jest trzymane na serwerze” na stronie 164 i „Skąd pochodzą rzeczy trzymane na serwerze” na stronie 165 opisałem zawartość serwera i jej pochodzenie. Wszystko, co jest na danym serwerze, pochodzi ze zdefiniowanego źródła, którym może być instalacja systemu operacyjnego, repozytorium pakietów systemowych albo kod konfiguracji serwera. Każda zmiana, która ma dotyczyć serwera, musi dotyczyć jednej z tych rzeczy.

W tym rozdziale opisuję, jak zmieniać rzeczy na serwerze, zmieniając kod, który definiuje, skąd pochodzą te rzeczy i stosując go w ten lub inny sposób. Implementując niezawodny, zautomatyzowany proces zmian dla swoich serwerów zapewniamy możliwość szybkiego i niezawodnego wdrażania zmian w całym środowisku. Wkładając minimalny wysiłek możemy zapewnić obecność najnowszych wersji zatwierdzonych pakietów i konfiguracji we wszystkich serwerach.

Jest kilka wzorców stosowania kodu konfiguracji do serwerów. Należy do nich między innymi stosowanie każdej zmiany na bieżąco, ciągłe synchronizowanie oraz odbudowywanie serwerów w celu ich zmiany. Innym aspektem dokonywania zmian jest sposób uruchamiania narzędzia w celu zastosowania zmian do serwera – czy ma to być wypychanie czy pobieranie konfiguracji. Poza tym występuje jeszcze kilka innych zdarzeń w cyklu życia serwera, od wstrzymywania, przez odbudowę po usunięcie serwera.

Wzorce zarządzania zmianami: kiedy stosować zmiany

Jest jeden antywzorzec i dwa wzorce decydowania, kiedy należy stosować zmiany do instancji serwera.

Antywzorzec: stosowanie każdej zmiany oddzielnie

Znany również jako: automatyzacja ad hoc.

W przypadku antywzorca *stosowanie każdej zmiany oddzielnie*, kod konfiguracji jest stosowany do serwera tylko wtedy, gdy pojawia się określona zmiana do zastosowania.

Jako przykład weźmy zespół, który uruchamia wiele serwerów aplikacji Tomcat. Członkowie zespołu muszą uruchomić podręcznik Ansible, aby zainstalować i skonfigurować serwer Tomcat, kiedy chcą utworzyć jego nową instancję, ale gdy serwer już działa, nie uruchamiają podręcznika Ansible, dopóki nie jest to konieczne. Po wydaniu nowej wersji Tomcat, aktualizują podręcznik i stosują go do swoich serwerów.

W skrajnej wersji tego antywzorca stosujemy kod do serwera tylko wtedy, gdy chcemy coś zmienić.

Nasz przykładowy zespół zauważa w pewnym momencie, że jeden z jego serwerów aplikacji odnotowuje znacznie większy ruch i na skutek obciążenia Tomcat zaczyna być niestabilny. Członkowie zespołu wprowadzają pewne zmiany do podręcznika, aby zoptymalizować konfigurację Tomcat pod kątem większego obciążenia i następnie stosują podręcznik do serwera, który ma problem. Ale nie muszą tego robić dla pozostałych serwerów aplikacji, ponieważ nie wymagają one zmiany.

Motywacja

Administratorzy systemów i sieci tradycyjnie zarządzali serwerami ręcznie. Jeśli trzeba było coś zmienić, to logowali się do odpowiedniego serwera i wprowadzali zmianę. Dlaczego mielibyśmy robić to inaczej? Nawet osoby, które używają skryptów, zazwyczaj piszą i uruchamiają skrypt w celu dokonania konkretnej zmiany. Posługiwanie się antywzorcem i stosowanie każdej zmiany oddzielnie jest rozszerzeniem tego typu pracy, które przypadkiem wykorzystuje narzędzie infrastruktury jako kodu zamiast polecenia ręcznego albo skryptu włącz-wyłącz.

Zastosowanie

Stosowanie kodu tylko wtedy, gdy wymaga tego konkretna zmiana, może się sprawdzać w przypadku pojedynczego serwera tymczasowego. Nie jest to jednak odpowiednia metoda zrównoważonego zarządzania grupą serwerów.

Konsekwencje

Jeśli kod konfiguracji jest stosowany tylko wtedy, gdy trzeba wprowadzić konkretną zmianę, mogą występować długie okresy, w których kod nigdy nie będzie stosowany do danej

instancji serwera. Gdy w końcu dojdzie do zastosowania kodu, może się to nie udać z powodu występowania dodatkowych różnic w serwerze oprócz tych, których miała dotyczyć zmiana.

Rzeczy na serwerach mają skłonność do zmian, jeśli się tego nie pilnuje. Ktoś może dokonać zmiany ręcznie – na przykład w celu szybkiego usunięcia awarii. Ktoś inny może zmienić system, instalując jego nowszą wersję lub nowe pakiety aplikacji. Działania te należą do kategorii szybkich zmian, co do których mamy pewność, że niczego nie zepsują. Ale także do kategorii zmian, o których tydzień później nikt nie pamięta (ponieważ była to niewielka zmiana), dopóki nie spędzimy kilku godzin na debugowaniu ich skutków.

Problem staje się jeszcze poważniejszy, gdy stosujemy zmiany do jednych serwerów, a do innych nie. Rozważmy poprzedni przykład, w którym zespół stosował kod służący do optymalizacji wydajności tylko do jednego serwera. W jakiś czas później ktoś z zespołu musi wprowadzić inną zmianę do serwerów aplikacji.

Gdy to nastąpi, poprzednia zmiana dotycząca optymalizacji wydajności zostanie zastosowana do serwerów, które jej jeszcze nie mają, jako efekt uboczny stosowania kodu dla nowej zmiany. Wcześniejsza zmiana może mieć nieoczekiwany wpływ na pozostałe serwery. Co gorsza, osoba stosująca kod może nie pamiętać o optymalizacji wydajności, więc odkrycie przyczyny problemów, które spowoduje, może potrwać znacznie dłużej.

Implementacja

Osoby przyzwyczajone do ręcznego dokonywania zmian lub używania skryptów typu włącz-wyłącz, zazwyczaj używają kodu konfiguracji serwera w taki sam sposób. Postrzegają to narzędzie – Ansible, Chef, Puppet – jako narzędzie skryptowe o niewygodnej składni. Takie osoby najczęściej uruchamiają narzędzie ręcznie z lokalnego komputera, zamiast pozostawić stosowanie kodu potokowi lub innej usłudze orkiestracji.

Powiązane wzorce

Ludzie mają skłonność do stosowania każdej zmiany oddzielnie w połączeniu ze wzorcem wypychania konfiguracji („Wzorec: wypychanie konfiguracji serwera” na stronie 192) zamiast pobierania. Alternatywą dla tego antywzorca są: ciągła synchronizacja oraz serwery niezmiennialne (patrz „Wzorec: serwer niezmiennialny” na stronie 189).

Wzorec: ciągła synchronizacja konfiguracji

Znany również jako: zaplanowana aktualizacja konfiguracji serwera.

Ciągła synchronizacja konfiguracji oznacza powtarzające się częste stosowanie konfiguracji kodu do serwera, niezależnie od tego, czy kod ten uległ zmianie. Powoduje to wycofanie lub ujawnienie wszystkich niespodziewanych różnic, które mogły się wkraść do serwera lub innych zasobów używanych przez kod konfiguracji.

Motywacja

Chcielibyśmy wierzyć, że konfiguracja serwera jest przewidywalna. Po zastosowaniu kodu do serwera nic nie powinno się w nim zmieniać do następnego zastosowania kodu. I jeśli kod pozostaje taki sam, nie ma potrzeby go stosować.

Jednak serwery i ich kod konfiguracji są podstępne. Czasami serwer zmienia się z oczywistych powodów, takich jak zalogowanie się kogoś lub ręczne wprowadzenie zmiany. Ludzie mają zwyczaj dokonywania w ten sposób drobnych zmian, ponieważ nie sądzą, że spowoduje to jakikolwiek problem. Jednak często się mylą co do tego. Z kolei w innych przypadkach zespoły zarządzają pewnymi aspektami serwera, używając innych niż zwykle narzędzi albo procesów. Na przykład niektóre zespoły wykorzystują specjalistyczne narzędzia do aktualizowania lub poprawiania serwerów, zwłaszcza jeśli chodzi o poprawki zabezpieczeń.

Nawet jeśli serwer się nie zmienił, zastosowanie tego samego kodu konfiguracji serwera wiele razy może spowodować pojawienie się różnic. Na przykład kod może używać parametrów z centralnego rejestru konfiguracji. Jeśli jeden z tych parametrów ulegnie zmianie, kod może zrobić coś całkiem innego przy następnym wykonaniu na serwerze.

Kolejnym zewnętrznym źródłem zmian są pakiety. Jeśli kod konfiguracji instaluje pakiet z repozytorium, to może zaktualizować go do nowszej wersji, gdy stanie się dostępna. Można próbować jawnie określać wersje pakietów, ale powoduje to wejście na jedną z dwóch groźnych ścieżek. Jedna z nich prowadzi do tego, że system zawiera z czasem wiele przestarzałych pakietów, w tym takie, które mają luki w zabezpieczeniach dobrze znane atakującym. Druga natomiast powoduje, że zespół poświęca masę energii na ręczne aktualizowanie numerów wersji pakietów w kodzie serwera.

Poprzez regularne stosowanie kodu konfiguracji serwera, według zautomatyzowanego harmonogramu, mamy pewność, że wszystkie nasze serwery są skonfigurowane w sposób spójny. Ponadto wszelkie zmiany, niezależnie od źródeł, są stosowane raczej wcześniej niż później.

Zastosowanie

Łatwiej jest zaimplementować ciągłą synchronizację niż jej główną alternatywę – serwery niezmiennialne. Większość narzędzi infrastruktury jako kodu – na przykład Ansible, Chef i Puppet – zostało zaprojektowanych zgodnie z tym wzorcem. Szybsze i mniej uciążliwe jest stosowanie zmian za pomocą aktualizacji istniejącej instancji serwera niż poprzez tworzenie jego nowej instancji.

Konsekwencje

Gdy zautomatyzowany proces stosuje konfigurację serwera w całym naszym środowisku, zachodzi ryzyko, że coś się zepsuje. Wszystko, co może się nieoczekiwanie zmienić, jak to opisałem wcześniej w podrozdziale „Motywacja”, może spowodować awarię serwera. Aby temu przeciwdziałać, należy mieć skuteczny system monitorowania, który ostrzega

o problemach oraz dobrą procedurę testowania i dostarczania kodu przed zastosowaniem zmian do systemów produkcyjnych.

Implementacja

Jak wspomniałem wcześniej, większość narzędzi do konfigurowania serwerów jako kodu wykorzystuje wzorzec ciągłego uruchamiania. Używane przez nie mechanizmy są opisane w podrozdziałach „Wzorzec: wypychanie konfiguracji serwera” na stronie 192 oraz „Wzorzec: pobieranie konfiguracji serwera” na stronie 193.

Większość implementacji ciągłej synchronizacji jest uruchamianych według harmonogramu. Zazwyczaj moment wykonania jest inny dla różnych serwerów, tak aby wszystkie serwery nie były wybudzane i nie miały uruchamianej konfiguracji jednocześnie¹. Czasem jednak chcemy zastosować kod szybciej, na przykład w celu usunięcia błędu lub wdrożenia oprogramowania. Różne narzędzia oferują do tego różne rozwiązania.

Powiązane wzorce

Ciągła synchronizacja jest implementowana przy użyciu wzorca wypychania konfiguracji lub jej pobierania. Alternatywą są serwery niezmiennialne.

Wzorzec: serwer niezmiennialny

Serwer niezmiennialny (*immutable*) to instancja serwera, której konfiguracja nigdy się nie zmienia. Dostarczanie zmian odbywa się poprzez utworzenie nowej instancji serwera ze zmienioną konfiguracją i użycie jej do zastąpienia istniejącego serwera².

Motywacja

Serwery niezmiennialne redukują ryzyko towarzyszące wprowadzaniu zmian. Zamiast stosować zmianę do działającej instancji serwera, tworzymy całkiem nową instancję. Mamy okazję przetestować tę nową instancję, a następnie wstawić ją zamiast poprzedniej. Takie działanie pozwala sprawdzić, czy nowa instancja działa prawidłowo jeszcze przed zniszczeniem oryginalnej, a jeśli coś pójdzie nie tak, przywrócić szybko poprzednią.

Zastosowanie

Organizacje, które wymagają ścisłej kontroli i spójności konfiguracji serwerów, mogą uznać serwery niezmiennialne za przydatne. Na przykład firma telekomunikacyjna, która wykorzystuje tysiące obrazów serwerów, może postanowić nie stosować zmian do działających serwerów, woląc zagwarantować stabilność ich konfiguracji.

1 Na przykład opcja `--sp` klienta Chefa (<https://oreil.ly/MH-54>).

2 Mój kolega Peter Gillard-Moss i były kolega Ben Butler-Cole używali serwerów niezmiennych podczas pracy na platformie Mingle SaaS ThoughtWorks (<http://thght.works/1Vw3GY8>).

Konsekwencje

Implementacja serwerów niezmiennalnych wymaga rozbudowanego zautomatyzowanego procesu budowania, testowania i aktualizowania obrazów serwera (opisanego w rozdziale 13). Projekt systemu i aplikacji musi obsługiwać zamianę instancji serwera bez przerywania usług (pomysły można znaleźć w podrozdziale „Zmiana działającej infrastruktury” na stronie 360).

Pomimo nazwy serwery niezmiennalne ulegają zmianom³.

Może pojawić się dryf konfiguracji, zwłaszcza jeśli ludzie mogą logować się do serwerów i ręcznie dokonywać w nich zmian, zamiast budować nowe instancje przy użyciu procesu zmiany konfiguracji. Dlatego zespoły wykorzystujące serwery niezmiennalne powinny być ostrożne i zapewniać aktualność działających instancji. Można jak najbardziej łączyć serwery niezmiennalne z antywzorcem stosowania zmian na bieżąco (patrz „Antywzorec: stosowanie każdej zmiany oddzielnie” na stronie 186), czego efektem mogą być serwery działające przez długi czas bez zmian, pozbawione poprawek i ulepszeń obecnych w serwerach zbudowanych później. Zespoły powinny rozważać wyłączenie dostępu do serwerów albo zezwalać na dostęp i ręczną zmianę usług wyłącznie metodą „zbicia szybki”⁴.

Implementacja

Większość zespołów, które używają serwerów niezmiennalnych, przechowuje całą masę konfiguracji w obrazach serwerów, wybierając częściej pieczenie obrazów (patrz „Pieczenie obrazów serwera” na stronie 182) niż smażenie instancji. Tak więc podstawą serwerów niezmiennalnych są potoki lub zestawy potoków do automatycznego budowania i aktualizowania obrazów serwerów.

Można smażyć konfigurację w instancjach serwerów niezmiennalnych (patrz „Smażenie instancji serwera” na stronie 181), o ile po utworzeniu instancji nie będzie już ona zmieniana. Ale bardziej rygorystyczna forma serwerów niezmiennalnych unika dodawania jakichkolwiek zmian do instancji serwerów. Przy takim podejściu tworzy się i testuje obraz serwera, a następnie promuje go z jednego środowiska do następnego. Ponieważ niewiele lub wręcz nic się nie zmienia w każdej instancji serwera, mniejsze jest ryzyko przedostania się błędów z jednego środowiska do drugiego.

3 Niektórzy ludzie protestują, że infrastruktura niezmiennalna jest niewłaściwym podejściem, ponieważ liczne aspekty serwerów, w tym dzienniki, pamięć i przestrzeń procesów, ulegają zmianom. Argument jest podobny do odrzucania „obliczeń bezserwerowych”, ponieważ w tle i tak są używane serwery. Terminologia jest metaforą. Pomimo niedoskonałości metafory, opisywane przez nią podejście jest dla wielu osób przydatne.

4 Proces „zbicia szybki” (przez analogię do szybki zasłaniającej przycisk alarmu pożarowego) może służyć do chwilowego uzyskania dostępu o podniesionych uprawnieniach w sytuacji awaryjnej. Proces ten powinien być bardzo widoczny, aby zniechęcić do jego rutynowego stosowania. Jeśli ludzie zaczynają używać tego procesu, zespół powinien ocenić, jakie zadania ich do tego zmuszają i poszukać sposobów obsługi tych zadań bez tego procesu. Więcej informacji można znaleźć w webinarium Dereka A. Smitha „Break Glass Theory” (<https://oreil.ly/GkcM2>).

Powiązane wzorce

Ludzie stosują zwykle pieczenie serwerów (patrz „Pieczenie obrazów serwera” na stronie 182), aby obsługiwać serwery niezmiennialne. Ciągła synchronizacja (patrz „Wzorzec: ciągła synchronizacja konfiguracji” na stronie 187) jest podejściem odwrotnym, polegającym na rutynowym stosowaniu zmian do działających instancji serwerów. Serwery niezmiennialne są podzbiorem infrastruktury niezmiennialnej (patrz „Infrastruktura niezmiennialna” na stronie 342).



Poprawianie serwerów

Wiele zespołów jest przyzwyczajonych, że naprawianie serwerów to specjalny, oddzielny proces. Ale jeśli mamy zautomatyzowany potok, który dostarcza zmiany do serwerów i stosujemy ciągłą synchronizację kodu serwera do istniejących serwerów, możemy w ten sam sposób również naprawiać swoje serwery.

Zespoły, z którymi pracowałem, pobierają, testują i dostarczają najnowsze poprawki zabezpieczeń do swoich systemów operacyjnych i innych podstawowych pakietów, robiąc to co tydzień, a czasem nawet codziennie.

Procedura szybkiego i częstego poprawiania zrobiła wrażenie na dyrektorze ds. informatyki u jednego z moich klientów, gdy prasa biznesowa ogłosiła odkrycie poważnej luki w zabezpieczeniach podstawowego pakietu systemu operacyjnego. Dyrektor zażądał, abyśmy wszystko porzucili i opracowali plan usunięcia luki oraz oszacowali, ile czasu to potrwa i jakie koszty pociągnie za sobą. Był mile zaskoczony, gdy powiedzieliśmy mu, że poprawka usuwająca problem została już wprowadzona w ramach porannej rutynowej aktualizacji.

Jak stosować kod konfiguracji serwera

Po opisaniu wzorców dotyczących tego, kiedy stosować kod konfiguracji kodu do serwerów zajmę się teraz wzorcami sposobów stosowania kodu do instancji serwerów.

Wzorce te odnoszą się do zmieniania serwerów, zwłaszcza w ramach procesu ciągłej synchronizacji (patrz „Wzorzec: ciągła synchronizacja konfiguracji” na stronie 187). Ale są również używane podczas budowania nowych instancji serwerów w celu smażenia konfiguracji w instancji (patrz „Smażenie instancji serwera” na stronie 181). Oprócz tego konieczny będzie wybór wzorca stosowania konfiguracji podczas budowania obrazów serwera (rozdział 13).

Dla danej instancji serwera, zarówno tej nowej, właśnie budowanej, jak i tymczasowej, wykorzystywanej do zbudowania obrazu, oraz już istniejącej, są dostępne dwa wzorce uruchamiania narzędzia do konfiguracji serwera w celu zastosowania kodu. Jednym jest *wypychanie*, a drugim *pobieranie*.

Wzorzec: wypychanie konfiguracji serwera

Wzorzec *wypychanie konfiguracji serwera* wykorzystuje proces działający na zewnątrz nowej instancji serwera, który łączy się z serwerem i jest wykonywany, pobierając i stosując kod.

Motywacja

Zespoły używają wypychania, aby uniknąć konieczności wstępnego instalowania narzędzi do konfiguracji serwera w obrazach serwerów.

Zastosowanie

Wzorzec wypychania jest przydatny, gdy potrzebna jest ściślejsza kontrola nad harmonogramem stosowania aktualizacji konfiguracji do istniejących serwerów. Na przykład, jeśli mamy zdarzenia, takie jak wdrożenia oprogramowania, obejmujące sekwencje działań na wielu serwerach, możemy zaimplementować je za pomocą centralnego procesu, który to zorganizuje.

Konsekwencje

Wzorzec wypychania konfiguracji wymaga możliwości łączenia się z instancjami serwerów i wykonywania procesu konfiguracji poprzez sieć. Wymaganie to może stwarzać zagrożenie bezpieczeństwa, ponieważ otwiera wektor, którego atakujący mogą użyć do połączenia się i dokonania nieautoryzowanych zmian w naszych serwerach.

Uruchamianie wypychania konfiguracji dla instancji serwerów, które są tworzone automatycznie przez platformę (patrz „Konfigurowanie automatycznego tworzenia serwerów przez platformę” na stronie 176) może być niewygodne, na przykład w sytuacji automatycznego skalowania lub zautomatyzowanego odzyskiwania. Tym niemniej, da się to zrobić.

Implementacja

Jednym ze sposobów wypychania konfiguracji serwera jest uruchamianie narzędzia do konfiguracji serwera z czyjegoś komputera lokalnego. Jak jednak zobaczymy w podrozdziale „Stosowanie kodu z poziomu scentralizowanej usługi” na stronie 337, lepiej jest uruchamiać narzędzia z centralnego serwera lub usługi, aby zapewnić spójność procesu i kontrolę nad nim.

Niektóre narzędzia do konfiguracji serwera obejmują aplikację serwerową do zarządzania połączeniami z instancjami komputerów, na przykład Ansible Tower. Niektóre firmy oferują usługi SaaS do zdalnego konfigurowania instancji serwerów dla nas, choć wiele organizacji woli nie udostępniać stronom trzecim takiego poziomu kontroli nad swoją infrastrukturą.

W pozostałych przypadkach można samemu zaimplementować usługę centralną do uruchamiania narzędzi do konfiguracji serwera. Najczęściej spotykałem zespoły

używające do tego celu produktów serwerowych CI lub CD. Sprowadza się to do implementacji zadań CI lub etapów potoku, które uruchamiają narzędzie do konfiguracji na konkretnym zestawie serwerów. Zadanie jest wyzwalane na podstawie zdarzeń, takich jak zmiany w kodzie konfiguracji serwera lub utworzenie nowych środowisk.

Narzędzie do konfiguracji serwera musi mieć możliwość łączenia się z instancjami serwerów.

Chociaż niektóre narzędzia używają do tego celu jakiegoś własnego, niestandardowego protokołu sieciowego, większość wykorzystuje SSH. Każda instancja serwera musi akceptować połączenia SSH przychodzące od narzędzia serwera i pozwalać mu działać z uprawnieniami wystarczającymi do zastosowania zmian w jej konfiguracji.

Niezwykle ważne jest używanie silnego uwierzytelniania i zarządzanie wpisami tajnymi dla tych połączeń. W przeciwnym razie system konfiguracji serwera będzie ogromną luką w zabezpieczeniach.

Tworząc i konfigurując nową instancję serwera można dynamicznie wygenerować nowy tajny wpis uwierzytelniania, taki jak klucz SSH. Większość platform infrastruktury udostępnia metodę ustawienia klucza podczas tworzenia nowej instancji. Narzędzie do konfiguracji serwera może następnie użyć tego wpisu tajnego, a potencjalnie wyłączyć i usunąć klucz, gdy nie będzie już potrzebny.

Jeśli musimy stosować zmiany konfiguracji do istniejących instancji serwerów, jak w przypadku ciągłej synchronizacji, to potrzebujemy długoterminowej metody uwierzytelniania połączeń przychodzących od narzędzia do konfiguracji. Najprostszym wyjściem jest instalacja jednego klucza na wszystkich instancjach serwerów. Jednak taki pojedynczy klucz stanowi zagrożenie. W przypadku jego ujawnienia atakujący może uzyskać dostęp do każdego serwera w naszym środowisku.

Alternatywą jest ustawienie unikalnego klucza dla każdej instancji serwera. Ważne jest, aby zarządzać dostępem do tych kluczy tak, aby był on możliwy dla narzędzia konfiguracji serwera, ograniczając jednocześnie ryzyko uzyskania takiego samego dostępu przez atakującego – na przykład poprzez przejęcie kontroli nad serwerem, na którym działa narzędzie. Istotna jest tutaj porada z podrozdziału „Obsługa wpisów tajnych jako parametrów” na stronie 96.

Podejściem stosowanym przez wiele organizacji jest posiadanie wielu serwerów lub usług zarządzających konfiguracją serwerów. Cała struktura jest podzielona na różne strefy bezpieczeństwa i każda instancja usługi konfiguracji serwera ma dostęp tylko do jednej strefy. Taki podział pozwala zmniejszyć zasięg ewentualnego naruszenia zabezpieczeń.

Powiązane wzorce

Alternatywą dla wzorca wypychania jest wzorzec pobierania konfiguracji serwera.

Wzorzec: pobieranie konfiguracji serwera

Wzorzec *pobieranie konfiguracji serwera* wykorzystuje proces działający w samej instancji serwera, który pobiera i stosuje kod konfiguracji. Proces jest wyzwalany w momencie,

gdy zostaje utworzona nowa instancja serwera. W przypadku istniejących już instancji, korzystających z wzorca ciągłej synchronizacji, proces jest zwykle uruchamiany według harmonogramu, budząc się co jakiś czas i stosując bieżącą konfigurację.

Motywacja

Konfiguracja serwera oparta na pobieraniu unika konieczności akceptowania przez instancje serwera połączeń przychodzących z serwera centralnego, więc pomaga ograniczyć powierzchnię ataku. Wzorzec upraszcza konfigurowanie instancji tworzonych automatycznie przez platformę infrastruktury, na przykład w wyniku automatycznego skalowania lub zautomatyzowanego odzyskiwania (patrz „Konfigurowanie automatycznego tworzenia serwerów przez platformę” na stronie 176).

Zastosowanie

Konfigurowanie serwera oparte na pobieraniu można implementować w przypadku budowania lub używania obrazów serwerów, które mają wstępnie zainstalowane narzędzie do konfiguracji serwera.

Implementacja

Do pobierania konfiguracji używany jest obraz serwera z zainstalowanym wstępnie narzędziem do konfiguracji serwera. W przypadku pobierania konfiguracji dla nowych instancji serwerów, obraz serwera musi mieć skonfigurowane uruchamianie narzędzia podczas pierwszego rozruchu.

Powszechnie używanym narzędziem do automatycznego uruchamiania tego typu procesu jest Cloud-init⁵. Używając API platformy infrastruktury można przekazywać parametry do nowej instancji serwera, nawet polecenia do wykonania, a także parametry przekazywane do narzędzia do konfiguracji serwera (patrz przykład 12-1).

Przykład 12-1 *Przykład kodu stosu uruchamiającego skrypt konfiguracji*

```
server:
  source_image: stock-linux-1.23
  memory: 2GB
  vnet: ${APPSERVER_VNET}
  instance_data:
    - server_tool: servermaker
    - parameter: server_role=appserver
    - parameter: code_repo=servermaker.shopspinner.xyz
```

Skrypt musi być tak skonfigurowany, aby pobierał kod konfiguracji z centralnego repozytorium i stosował go przy uruchomieniu. Jeśli działające serwery są aktualizowane metodą ciągłej synchronizacji, proces konfiguracji powinien to ustawić, niezależnie od tego, czy

⁵ <https://oreil.ly/-2y6B>

jest uruchamiany proces w tle dla narzędzia do konfiguracji serwera, czy zadanie crona uruchamia narzędzie według harmonogramu.

Nawet jeśli nie budujemy własnych obrazów serwerów, większość obrazów udostępnianych przez dostawców chmur publicznych ma wstępnie zainstalowaną usługę cloud-init i popularne narzędzia do konfiguracji serwerów.

Kilka innych narzędzi, w szczególności Saltstack, do wyzwalania konfiguracji serwera wykorzystuje podejście oparte na wiadomościach i zdarzeniach. Każda instancja serwera uruchamia agenta, który łączy się z współdzieloną magistralą usług, z której otrzymuje polecenia zastosowania kodu konfiguracji.

Powiązane wzorce

Alternatywą dla wzorca pobierania jest wzorzec wypychania konfiguracji serwera (patrz „Wzorzec: wypychanie konfiguracji serwera” na stronie 192).

Konfiguracja zdecentralizowana

Większość narzędzi do konfiguracji serwera udostępnia centralną usługę, którą uruchamia się na maszynie lub klastrze w celu centralnego sterowania dystrybucją kodu i parametrów konfiguracji oraz zarządzania innymi działaniami. Niektóre zespoły wolą działać bez usługi centralnej.

Głównym powodem, dla którego zespoły decentralizują konfigurację, jest uproszczenie zarządzania infrastrukturą. Serwer konfiguracji to jeszcze jeden zestaw elementów do zarządzania i możliwy pojedynczy punkt awarii. Jeśli serwer konfiguracji nie działa, nie można budować nowych maszyn, co czyni z niego zależność w sytuacji odzyskiwania po awarii. Serwery konfiguracji mogą być również wąskim gardłem i wymagać skalowania w celu obsługi połączeń od (i prawdopodobnie do) setek albo tysięcy instancji serwerów.

Aby zaimplementować zdecentralizowaną konfigurację, należy zainstalować i uruchomić narzędzie do konfiguracji w trybie offline, na przykład używając chef-solo zamiast chef-client. Można uruchomić narzędzie z poziomu skryptu, który sprawdzi centralne repozytorium plików i pobierze najnowszą wersję kodu konfiguracji serwera. Kod jest przechowywany lokalnie w instancjach serwerów, więc narzędzie i tak może działać, jeśli repozytorium plików nie jest dostępne.

Centralne repozytorium plików może być pojedynczym punktem awarii albo wąskim gardłem, podobnie jak serwer konfiguracji. W praktyce jest jednak wiele prostych, bardzo niezawodnych i wydajnych opcji hostowania statycznych plików, takich jak konfiguracje serwerów. Należą do nich serwery sieci Web, sieciowe serwery plików i usługi przechowywania obiektów, takie jak AWS S3.

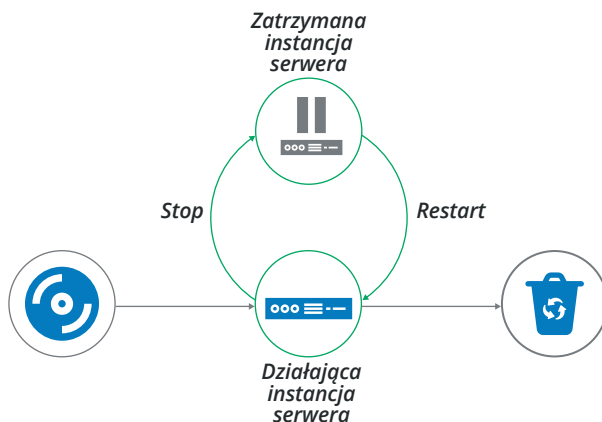
Innym sposobem implementacji wzorca decentralizacji jest dołączenie kodu konfiguracji serwera do pakietu systemowego, takiego jak plik *.rpm* lub *.deb* i hostowanie go w prywatnym repozytorium pakietów. Zwykły proces wykonuje polecenie `yum update` lub `apt-get update`, które instaluje lub aktualizuje pakiet, kopiując kod konfiguracji serwera do lokalnego katalogu.

Pozostałe zdarzenia w cyklu życia serwera

Tworzenie, zmienianie i usuwanie instancji serwera składa się na jego podstawowy cykl życia. Ale występują jeszcze inne fazy w przypadku wydłużonego cyklu życia, w tym zatrzymywanie i restartowanie serwera (rysunek 12-1), zastępowanie go oraz odzyskiwanie po awarii.

Zatrzymywanie i restartowanie instancji serwera

Gdy większość naszych serwerów była fizycznymi urządzeniami podłączonymi do prądu, zwykle wyłączałyśmy je w celu wymiany sprzętu na nowszy albo restartowałyśmy, gdy trzeba było zaktualizować pewne składniki systemu operacyjnego.



Rysunek 12-1 Cykl życia serwera – zatrzymywanie i restartowanie

Ludzie nadal zatrzymują i uruchamiają ponownie serwery wirtualne, czasami z tych samych powodów – aby zmienić konfigurację sprzętu wirtualnego albo uaktualnić jąrdro systemu operacyjnego. Czasami serwery są wyłączane w celu obniżenia kosztów hostingu. Na przykład niektóre zespoły wyłączają serwery do programowania i testowania na noc i w weekendy, jeśli nikt z nich nie korzysta.

Natomiast w przypadku serwerów, które można łatwo odbudować, wiele zespołów po prostu niszczy je, gdy nie są używane i tworzy nowe, gdy pojawi się potrzeba. Wynika

to częściowo z tego, że jest to łatwe i można w ten sposób zaoszczędzić nieco więcej pieniędzy niż stosując wyłączenie.

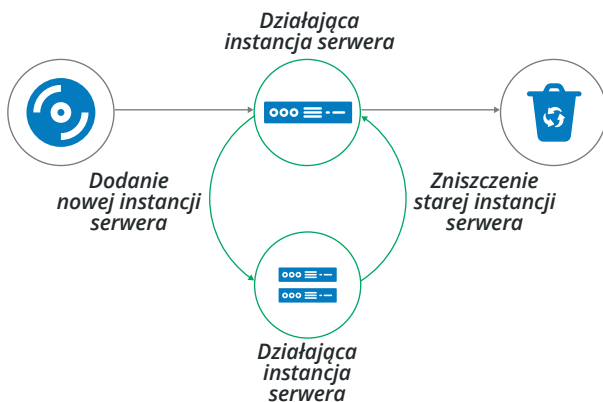
Niszczanie i odbudowywanie serwerów, zamiast zatrzymywania ich i trzymania w pobliżu, jest w zgodzie z filozofią traktowania serwerów jak zwierzęta hodowlane, a nie domowych pupilów (o czym była mowa w notce „Bydło hodowlane, a nie domowi ulubieńcy” na stronie 16). Zespoły mogą zatrzymywać i restartować serwery, zamiast je odbudowywać, ponieważ obawiają się takiej odbudowy. Zachowywanie i przywracanie danych aplikacji jest często wyzwaniem. W podrozdziale „Ciągłość danych w zmieniającym się systemie” na stronie 373 jest podanych kilka sposobów radzenia sobie z tym wyzwaniem.

Tak więc stosowanie zasady „nie zatrzymuj i nie restartuj serwerów” zmusza zespół do implementacji niezawodnych procesów i narzędzi do odbudowywania serwerów i kontynuowania ich pracy.

Zatrzymywanie serwerów może komplikować działania konserwacyjne. Aktualizacje konfiguracji i poprawki do systemu nie są stosowane do zatrzymanych serwerów. W zależności od sposobu zarządzania tego typu aktualizacjami serwery mogą je otrzymywać ponownie podczas najbliższego restartu albo mogą je przegapiać.

Zastępowanie instancji serwera

Jedną z wielu korzyści, jakie przyniosło przejście z serwerów fizycznych na wirtualne, jest łatwość budowania i zastępowania instancji serwerów. Wiele wzorców i praktyk opisanych w tej książce, w tym serwery niezmiennalne (patrz „Wzorzec: serwer niezmiennalny” na stronie 189) i częsta aktualizacja obrazów serwera, opiera się na możliwości zbudowania nowego serwera i zastąpienia nim już działającego (rysunek 12-2).



Rysunek 12-2 Cykl życia serwera – zastępowanie instancji serwera

Podstawowy proces zastępowania instancji serwera polega na utworzeniu nowej instancji, sprawdzeniu jej gotowości, rekonfiguracji pozostałej infrastruktury i systemów w celu

korzystania z nowej instancji, przetestowaniu działania, a następnie zniszczeniu starej instancji.

W zależności od aplikacji i systemów używających instancji serwera, zastąpienie może nie wymagać ich wyłączenia albo co najwyżej minimalnego przestoju. Poradę na ten temat można znaleźć w podrozdziale „Zmiana działającej infrastruktury” na stronie 360.

Niektóre platformy infrastruktury oferują funkcję automatyzacji procesu zastępowania serwera. Na przykład można zastosować zmianę konfiguracji do definicji serwerów w klastrze serwerów z automatycznym skalowaniem, zaznaczając, że powinien on używać nowego obrazu serwera, zawierającego poprawki zabezpieczeń. Platforma automatycznie doda nowe instancje, sprawdzi ich kondycję i usunie stare.

W innych przypadkach bywa konieczne samodzielne zastępowanie serwerów. Jednym ze sposobów zrobienia tego w ramach systemu dostarczania zmian opartego na potoku jest powiększanie i zmniejszanie. Najpierw wypychamy zmianę, która powoduje dodanie nowego serwera, a następnie wypychamy zmianę, która powoduje usunięcie starego serwera. Więcej szczegółów jest podanych w podrozdziale „Powiększanie i zmniejszanie” na stronie 363.

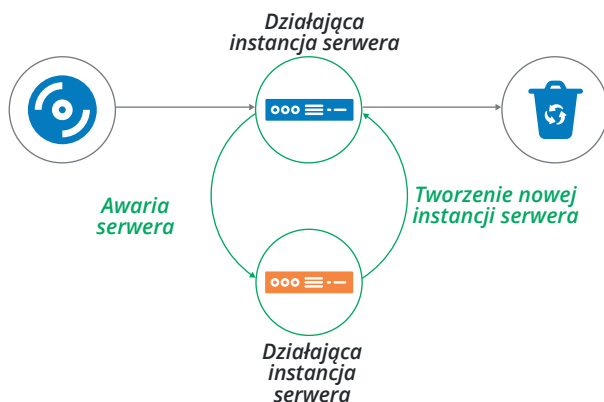
Odzyskiwanie serwera po awarii

Infrastruktura chmury niekoniecznie jest niezawodna. Niektórzy dostawcy, w tym AWS, jawnie informują, że mogą zakończyć działanie instancji serwera bez ostrzeżenia – na przykład, gdy postanowią wymienić bazowy sprzęt⁶. Nawet dostawcy z silniejszą gwarancją dostępności miewają awarie sprzętu, które wpływają na hostowane systemy.

Proces odzyskiwania uszkodzonego serwera jest podobny do procesu zastępowania serwera. Jedną z różnic jest kolejność działań – nowy serwer jest tworzony po zniszczeniu starego, a nie na odwrót (patrz rysunek 12-3). Druga różnica jest taka, że zwykle zastępujemy jakiś serwer celowo, natomiast awaria nie jest raczej zamierzona⁷. Tak jak w przypadku zastępowania instancji serwera, odzyskiwanie może wymagać ręcznego działania. Bywa też możliwe takie skonfigurowanie platformy i innych usług, aby wykrycie awarii i odzyskanie serwera odbywało się automatycznie.

6 Amazon udostępnia dokumentację dotyczącą zasad wycofywania swoich instancji (<https://oreil.ly/vSc0J>).

7 Wyjątkiem jest inżynieria chaosu, będąca praktyką celowego wywoływania awarii w celu wykazania możliwości odzyskania. Patrz „Inżynieria chaosu” na stronie 371.



Rysunek 12-3 Cykl życia serwera – odzyskiwanie instancji serwera po awarii

Podsumowanie

Omówiliśmy kilka wzorców dotyczących tego, kiedy stosować zmiany konfiguracji do serwerów – czy stosować je ciągle do działających serwerów czy tworzyć w tym celu nowe instancje. Przyjrzelśmy się również wzorcom sposobu stosowania zmian – wypychaniu i pobieraniu. Na koniec wspomnieliśmy o kilku innych zdarzeniach z cyklu życia serwerów, takich jak zatrzymywanie, zastępowanie i odzyskiwanie serwerów.

Ten rozdział w połączeniu z poprzednim, dotyczącym tworzenia serwerów, obejmuje podstawowe zdarzenia w życiu serwera. Ale wiele podejść do tworzenia i zmieniania serwerów, o których mówiliśmy, polega na dostosowywaniu obrazów serwerów, używanych następnie do tworzenia lub aktualizacji wielu instancji serwerów. Dlatego w następnym rozdziale zajmiemy się podejściami do definiowania, budowania i aktualizowania obrazów serwerów.

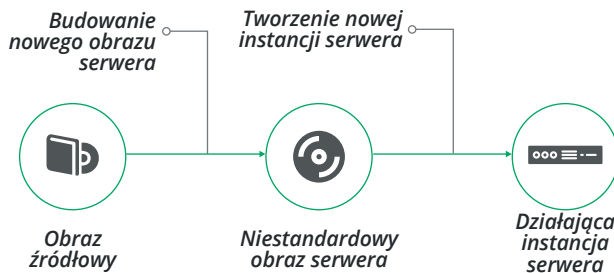
Obrazy serwerów jako kod

W rozdziale 11 omówiłem tworzenie wielu instancji serwera na podstawie jednego źródła („Wstępne budowanie serwerów” na stronie 178) i poleciłem używanie do tego celu czystych obrazów serwera („Tworzenie czystego obrazu serwera” na stronie 179). Jeśli ktoś się zastanawiał, jaki jest najlepszy sposób skorzystania z tej rady, to ten rozdział ma mu służyć pomocą.

Jak w przypadku większości rzeczy w systemie, do budowania i aktualizacji obrazów serwera w powtarzalny sposób należy używać kodu. Tylko wtedy dokonując każdej zmiany w obrazie, na przykład instalacji najnowszych poprawek systemu operacyjnego, można mieć pewność, że budowanie będzie odbywać się w sposób spójny.

Używając zautomatyzowanego, opartego na kodzie procesu budowania, testowania, dostarczania i aktualizowania obrazów serwera, łatwiej jest utrzymywać swoje serwery oraz zapewniać ich zgodność i jakość.

Podstawowy cykl życia obrazu serwera obejmuje budowę niestandardowego obrazu na podstawie oryginalnego źródła (patrz rysunek 13-1).



Rysunek 13-1 Cykl życia obrazu serwera

Budowanie obrazu serwera

Większość platform do zarządzania infrastrukturą ma swój format obrazu serwera, który służy do tworzenia instancji serwerów. Amazon ma AMI, Azure ma obrazy zarządzane, a VMware ma szablony VM (maszyn wirtualnych). Hostowane platformy udostępniają

wstępnie spakowane obrazy stockowe z popularnymi systemami operacyjnymi i dystrybucjami, dzięki czemu można tworzyć serwery bez konieczności samodzielnego budowania ich obrazów.

Po co budować obraz serwera?

Większość zespołów kończy na tym, że buduje niestandardowe obrazy, zamiast korzystać z obrazów stockowych od dostawcy platformy. Oto najczęstsze powody takiej decyzji:

Zgodność z ładem organizacyjnym

Wiele organizacji, zwłaszcza należących do branż regulowanych, musi mieć pewność, że budowane serwery są zgodne z surowymi wytycznymi.

Utwarczanie zabezpieczeń

Stockowe obrazy serwerów mają zwykle zainstalowanych wiele pakietów i narzędzi, aby były przydatne do różnych zastosowań. Budując niestandardowe obrazy można ograniczyć ich zawartość do minimum. Wzmacnianie serwera może obejmować wyłączanie i usuwanie nieużywanych kont użytkowników i usług systemowych, wyłączanie wszystkich portów sieciowych, które nie są niezbędne do działania oraz blokowanie uprawnień do systemów plików i folderów¹.

Optymalizacja pod kątem wydajności

Wiele kroków podejmowanych w celu wzmacniania zabezpieczeń obrazu, takich jak zatrzymywanie i usuwanie niepotrzebnych usług systemowych, ogranicza ponadto zapotrzebowanie serwera na zasoby pamięci i CPU. Minimalizacja wielkości obrazu serwera pozwala dodatkowo szybciej tworzyć instancje serwera, co bywa pomocne w scenariuszach skalowania i odzyskiwania.

Instalowanie popularnych pakietów

W obrazie serwera można zainstalować standardowy zestaw usług, agentów i narzędzi, aby mieć pewność, że będą one dostępne we wszystkich instancjach. Mogą to być na przykład agenty monitorowania, konta użytkowników systemu oraz specyficzne dla danego zespołu narzędzia i skrypty konserwacji.

Budowanie obrazów dla ról serwera

Wiele przykładów przedstawionych w tym rozdziale idzie dalej, niż budowa standardowego obrazu ogólnego przeznaczenia. Można tworzyć obrazy serwerów dostosowane do konkretnego celu, którym może być węzeł klastra kontenerów, serwer aplikacji albo agent serwera CI.

¹ Oto kilka źródeł zawierających więcej informacji o wzmacnianiu zabezpieczeń: „Proactively Hardening Systems Against Intrusion: Configuration Hardening” (https://oreil.ly/_ibdW) w witrynie Tripwire, „25 Hardening Security Tips for Linux Servers” (<https://oreil.ly/XiL0h>) oraz „20 Linux Server Hardening Security Tips” (<https://oreil.ly/yZUVa>).

Jak zbudować obraz serwera

W rozdziale 11 opisałem kilka różnych sposobów tworzenia nowej instancji serwera, w tym klonowanie istniejącej instancji podczas pracy oraz używanie migawki istniejącej instancji (patrz „Wstępne budowanie serwerów” na stronie 178). Ale instancje budowane przy użyciu dobrze zarządzanego obrazu serwera są czystsze i bardziej spójne.

Są dwa podejścia do budowy obrazu serwera. Najpopularniejsze i najprostsze polega na *budowaniu obrazu online*, czyli utworzeniu i skonfigurowaniu nowej instancji serwera, a następnie przekształceniu jej w obraz. Drugie podejście, *budowanie obrazu offline*, polega na zamontowaniu woluminu dysku i skopiowaniu na niego potrzebnych plików, a następnie przekształceniu go w obraz rozruchowy.

Tym, co różni oba te podejścia, są przede wszystkim szybkość i stopień trudności. Łatwiej jest zbudować obraz serwera online, ale rozruch i konfiguracja nowej instancji może wymagać trochę czasu – od kilku do kilkudziesięciu minut. Zamontowanie i przygotowanie dysku offline trwa zwykle znacznie krócej, ale wymaga więcej pracy.

Przed omówieniem sposobu implementacji każdej z tych opcji warto zwrócić uwagę na kilka narzędzi do tworzenia obrazów serwerów.

Narzędzia do budowania obrazów serwerów

Działanie narzędzi i usług do budowania obrazów serwerów polega zwykle na orkiestracji procesu tworzenia instancji serwera lub woluminu dysku, uruchomieniu narzędzi lub skryptów do konfiguracji serwera w celu jego dostosowania, a następnie użyciu API platformy infrastruktury w celu konwersji instancji serwera na obraz. Czytelnik powinien umieć zdefiniować ten proces jako kod, z dobrze już znanych powodów. Pokażę za chwilę przykłady kodu budującego obraz, ilustrujące podejścia online i offline.

Netflix był pionierem intensywnego używania obrazów serwerów i udostępnił w formie open source narzędzie Aminator², które stworzył do tego celu. Narzędzie Aminator jest specyficzne dla sposobu pracy Netfliksa – budowania obrazów CentOS i Red Hat dla AWS EC2³.

Najpopularniejszym narzędziem do budowania obrazów, w chwili gdy to piszę, jest Packer⁴ firmy HashiCorp. Packer obsługuje różne systemy operacyjne i platformy infrastruktury.

Niektóre platformy infrastruktury oferują własne usługi do budowania obrazów serwerów, na przykład AWS Image Builder⁵ i Azure Image Builder⁶.

Pokazane w tym rozdziale przykłady kodu budującego obrazy serwerów wykorzystują prosty, fikcyjny język konstruktora obrazów.

2 <https://oreil.ly/P2L1K>

3 Netflix opisał swoje podejście do budowy i używania AMI w poście na blogu (https://oreil.ly/e0_B3).

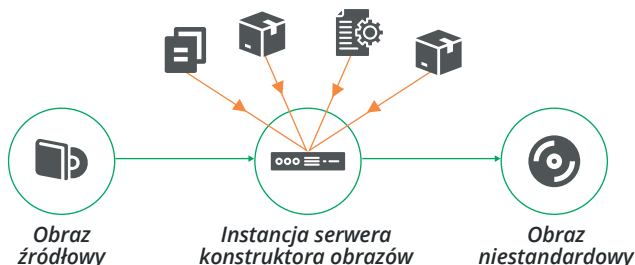
4 <http://packer.io>

5 <https://oreil.ly/hvf5R>

6 <https://oreil.ly/jfy8n>

Proces budowania obrazu online

Metoda budowania obrazów serwerów w trybie online obejmuje rozruch nowej, czystej instancji serwera, skonfigurowanie jej, a następnie przekonwertowanie na format obrazu serwera używany przez platformę infrastruktury, jak to jest pokazane na rysunku 13-2.



Rysunek 13-2 *Proces budowania obrazu serwera online*

Proces zaczyna się od rozruchu nowej instancji serwera z obrazu źródłowego, jak to jest pokazane w przykładzie 13-1. Więcej o opcjach obrazów źródłowych jest w podrozdziale „Zawartość źródłowa obrazu serwera” na stronie 208. Jeśli do budowy obrazu jest używane narzędzie typu Packer, wykorzystuje ono API platformy do utworzenia instancji serwera.

Przykład 13-1 *Przykład kodu konstruktora obrazu uruchamianego z obrazu źródłowego*

```
image:
  name: shopspinner-linux-image
  platform: fictional-cloud-service
  origin: fci-12345678
  region: europe
  instance_size: small
```

Ten przykładowy kod buduje obraz FCS o nazwie `shopspinner-linux-image`, uruchamiając go z istniejącego obrazu FCI o ID `fci-12345678`⁷. Jest to stockowa dystrybucja Linuksa, udostępniana przez dostawcę chmury. Powyższy kod nie określa żadnych działań konfiguracyjnych, więc wynikowy obraz jest dość prostą kopią obrazu źródłowego.

Kod definiuje ponadto kilka cech instancji serwera uruchamianej w celu zbudowania obrazu, w tym region i typ instancji.

⁷ FCS to Fictional Cloud Service (fikcyjna usługa chmurowa), podobna do AWS, Azure i innych wspomnianych w podrozdziale „Platformy infrastruktury” na stronie 22. FSI to Fictional Server Image (fikcyjny obraz serwera), podobny do AWS AMI i Azure Managed Image.

Infrastruktura dla instancji konstruktora

Instancja budująca obraz serwera potrzebuje z reguły pewnych zasobów infrastruktury – na przykład bloku adresów. Trzeba pilnować, aby stosować w tym celu odpowiednio bezpieczny i odizolowany kontekst.

Uruchamianie instancji konstruktora przy użyciu istniejącej infrastruktury bywa wygodne, ale może stwarzać zagrożenie dla procesu budowy obrazu. Najlepiej jest utworzyć jednorazową infrastrukturę do budowy obrazu, a na koniec ją usunąć. Kod w przykładzie 13-2 dodaje ID podsieci i klucz SSH do instancji konstruktora obrazu.

Przykład 13-2 *Dynamicznie przydzielane zasoby infrastruktury*

```
image:
  name: shopspinner-linux-image
  origin: fci-12345678
  region: europe
  size: small
  subnet: ${IMAGE_BUILDER_SUBNET_ID}
  ssh_key: ${IMAGE_BUILDER_SSH_KEY}
```

Te nowe wartości używają parametrów, które można automatycznie generować i przekazywać do narzędzia konstruktora obrazu. Proces budowania obrazu może zaczynać się od utworzenia instancji stosu infrastruktury, definiującego te rzeczy i niszczącego je po uruchomieniu narzędzia konstruktora obrazu. Jest to przykład skryptu orkiestracji narzędzia infrastruktury (więcej na ten temat jest w podrozdziale „Używanie skryptów do opakowywania narzędzi infrastruktury” na stronie 327).

Należy minimalizować zasoby i możliwości dostępu zapewniane instancji serwera konstruktora obrazu. Dostęp przychodzący powinien być dozwolony tylko dla narzędzia wyposażającego serwer i to jedynie w przypadku wypychania konfiguracji (patrz „Wzorzec: wypychanie konfiguracji serwera” na stronie 192). Dostęp wychodzący powinien być dozwolony tylko w zakresie niezbędnym do pobierania pakietów i konfiguracji.

Konfigurowanie instancji konstruktora

Gdy instancja serwera już działa, nasz proces może zastosować do niej konfigurację. Większość narzędzi do budowania obrazu serwera pozwala uruchamiać popularne narzędzia do konfigurowania serwerów (patrz „Kod konfiguracji serwera” na stronie 166), w sposób podobny do narzędzi zarządzania stosem. Kod w przykładzie 13-3 jest rozszerzeniem kodu z przykładu 13-2 i powoduje uruchomienie narzędzia Servermaker w celu konfiguracji instancji jako serwera aplikacji.

Przykład 13-3 *Używanie narzędzia Servermaker do konfigurowania instancji serwera*

```
image:
  name: shopspinner-linux-image
  origin: fci-12345678
```

```
region: europe
instance_size: small
subnet: ${IMAGE_BUILDER_SUBNET_ID}
configure:
  tool: servermaker
  code_repo: servermaker.shopspringer.xyz
  server_role: appserver
```

Powyższy kod stosuje rolę serwera, jak w podrozdziale „Role serwerów” na stronie 169, która może powodować instalację serwera aplikacji i powiązanych elementów.

Budowanie obrazu serwera przy użyciu prostych skryptów

Jeśli do konfigurowania nowych instancji serwerów jest już używane przeznaczone do tego narzędzie, warto wykorzystać to samo narzędzie, a często również ten sam kod konfiguracji serwera, do budowania obrazów serwerów. Jest to zgodne z procesem pieczenia i smażenia opisanym w podrozdziale „Łączenie pieczenia i smażenia” na stronie 182.

Jednak dla zespołów, które pieką bardziej rozbudowane obrazy serwerów (patrz „Pieczenie obrazów serwera” na stronie 182), w pełni rozwinięte narzędzie do konfigurowania serwerów może być przesadą. Takie narzędzia i ich języki są bardziej skomplikowane, aby można było stosować je wielokrotnie do serwerów o nieznanym, zmiennym warunkach początkowych.

Natomiast za każdym razem, gdy budujemy obraz serwera, używamy zazwyczaj znanego, spójnego źródła. Dlatego bardziej odpowiedni może być prosty język skryptowy – na przykład Bash, skrypty wsadowe albo PowerShell.

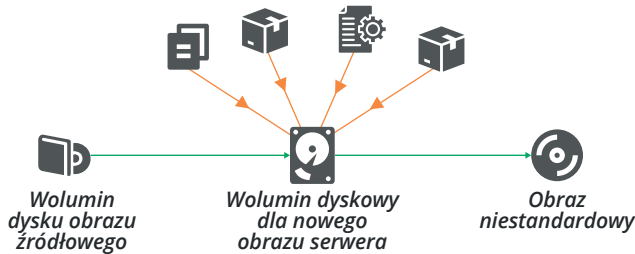
Jeśli ktoś pisze skrypty w celu konfigurowania obrazów serwerów, powinien pisać małe, proste skrypty, każdy skupiony na jednym zadaniu, zamiast dużych, złożonych skryptów z pętlami warunkowymi. Lubię dodawać na początku nazw skryptów liczby, dzięki czemu widać wyraźnie, w jakiej kolejności należy je uruchamiać, jak to jest pokazane w przykładzie 13-4.

Przykład 13-4 Przykład konstruktora obrazu wykorzystującego skrypty powłoki

```
image:
  name: shopspringer-linux-image
  origin: fci-12345678
  configure:
    commands:
      - 10-install-monitoring-agent.sh
      - 20-install-jdk.sh
      - 30-install-tomcat.sh
```

Proces budowania obrazu offline

Proces rozruchu instancji serwera, zastosowania konfiguracji, a następnie wyłączenia instancji w celu utworzenia obrazu serwera może być powolny. Alternatywą jest użycie obrazu źródłowego, który można zamontować jako wolumin dysku – na przykład AMI z dyskiem EBS w AWS. Można zamontować kopię takiego obrazu jako wolumin dysku, a następnie skonfigurować pliki w tym obrazie dysku, jak to jest pokazane na rysunku 13-3. Po zakończeniu konfigurowania obrazu dysku należy odmontować go i przekonwertować na format obrazu serwera używany przez platformę.



Rysunek 13-3 Proces budowania obrazu serwera w trybie offline

Aby zbudować obraz na woluminie dysku, trzeba zamontować ten wolumin w instancji obliczeniowej, w której działa nasze narzędzie do budowania obrazu. Na przykład, jeśli budujemy obrazy przy użyciu potoku, jak to jest opisane w dalszej części tego rozdziału, może to być agent narzędzia CD.

Kolejnym niezbędnym elementem jest narzędzie lub skrypt do prawidłowego skonfigurowania woluminu dysku. To może stanowić problem. Narzędzia do konfigurowania serwerów stosują kod konfiguracji do instancji serwera, na której są uruchomione. Nawet w przypadku prostszych skryptów, takich jak skrypty powłoki, skrypty te często uruchamiają narzędzie typu instalator pakietów, który domyślnie instaluje pliki w systemie plików działającego serwera:

```
yum install java
```

Wiele narzędzi ma opcje wiersza polecenia lub konfiguracji, za pomocą których można ustawić instalowanie plików w innej ścieżce. Pozwala to zmienić miejsce docelowe instalowanych pakietów (na przykład `yum install java --prefix /mnt/image_builder/`):

```
yum install --prefix /mnt/image_builder/ java
```

Jest to jednak trudne i łatwo popełnić błąd. Poza tym nie wszystkie narzędzia udostępniają tego typu opcje.

Innym rozwiązaniem jest użycie polecenia `chroot`, które uruchamia polecenie z innym katalogiem głównym systemu plików:

```
chroot /mnt/image_builder/ yum install java
```

Zaletą stosowania polecenia `chroot` jest to, że działa ono dla każdego narzędzia i polecenia. Popularne narzędzia do budowania obrazów, jak Packer, mają wbudowaną obsługę `chroot`⁸.

Zawartość źródłowa obrazu serwera

Do zbudowania obrazu serwera potrzebnych jest kilka elementów:

Kod konstruktora obrazu

Kod definiuje obraz używany przez narzędzie konstruktora. Przykładem jest plik szablonu narzędzia Packer. Kod ten definiuje pozostałe elementy składające się na obraz.

Obraz źródłowy

W wielu przypadkach budując obraz zaczynamy od innego obrazu. Możemy potrzebować obrazu źródłowego w formacie naszej platformy, na przykład AMI. Albo może to być obraz instalacyjny systemu operacyjnego w postaci fizycznej (DVD) albo danych (obraz ISO). W niektórych przypadkach obraz jest budowany od podstaw, na przykład podczas uruchamiania instalatora systemu operacyjnego. Każdy z tych przypadków jest opisany bardziej szczegółowo w dalszej części.

Kod konfiguracji serwera

Aby dostosować obraz, można użyć narzędzia do konfiguracji serwera lub skryptów, jak to zostało opisane w poprzednim podrozdziale.

Pozostałe elementy serwera

Do budowy obrazu serwera można używać dowolnych, a nawet wszystkich elementów i źródeł tych elementów, opisanych w podrozdziale „Co jest trzymane na serwerze” na stronie 164. Podczas budowania obrazów często korzysta się z pakietów przechowywanych w repozytoriach publicznych lub wewnętrznych.

Ten zestaw elementów występuje, gdy używamy potoku do automatyzacji procesu budowania obrazu, co jest opisane w dalszej części tego rozdziału. Poszczególne rodzaje obrazów źródłowych zasługują na nieco więcej uwagi.

Budowanie przy użyciu stockowego obrazu serwera

Dostawcy platform infrastruktury, dostawcy systemów operacyjnych oraz projekty open source często budują obrazy serwerów i udostępniają je użytkownikom i klientom. Jak już wspomniałem wcześniej, wiele zespołów wykorzystuje stockowe obrazy, zamiast budować własne. Ale jeśli ktoś buduje własne, to obrazy te są zwykle dobrym punktem wyjścia.

Jednym z problemów towarzyszących stockowym obrazom jest to, że wiele z nich zawiera zbyt dużo zasobów. Twórcy instalują w nich często szeroką gamę narzędzi i pakietów,

⁸ Zobacz kreatory `chroot` narzędzia Packera dla AMI AWS (https://oreil.ly/B_FuK) oraz dla obrazów serwerów Azure (<https://oreil.ly/umfU6>).

aby były natychmiast dostępne dla jak największej liczby użytkowników. Należy jednak minimalizować zestaw rzeczy instalowanych na niestandardowych obrazach serwerów, zarówno ze względów bezpieczeństwa, jak i wydajności.

Można albo usuwać niechcianą zawartość z obrazu źródłowego w ramach procesu konfiguracji obrazu serwera, albo szukać mniejszych obrazów źródłowych. Niektórzy dostawcy i organizacje budują obrazy JEOS (Just Enough Operating System). Dodawanie potrzebnych rzeczy do minimalnego obrazu podstawowego jest bardziej niezawodnym podejściem do utrzymywania minimalnych obrazów.

Budowanie obrazu serwera od podstaw

Platforma infrastruktury może nie udostępniać stockowych obrazów, zwłaszcza jeśli zespół używa wewnętrznej chmury lub platformy wirtualizacji. Niektóre zespoły po prostu wolą nie używać takich obrazów. Alternatywą jest budowa niestandardowych obrazów serwerów od podstaw. Obraz instalacyjny systemu operacyjnego zapewnia czysty, spójny punkt wyjścia do budowy szablonu serwera. Proces budowania szablonu zaczyna się od rozruchu instancji serwera z obrazu systemu operacyjnego i uruchomienia procesu instalacji opartego na skryptach⁹.

Pochodzenie obrazu serwera i jego zawartości

Innym problemem związanym z obrazami bazowymi, jak z każdą zawartością od firm trzecich, jest ich pochodzenie i bezpieczeństwo. Trzeba mieć pełną świadomość, kto dostarcza obraz i jakiego używa procesu w celu zapewnienia bezpiecznego tworzenia obrazów. Oto kilka rzeczy, które należy wziąć pod uwagę:

- Czy obrazy i pakiety zawierają oprogramowanie ze znanymi lukami zabezpieczeń?
- Jakie kroki (typu statyczna analiza kodu) podejmuje dostawca w celu skanowania swojego kodu pod kątem potencjalnych problemów?
- Skąd wiadomo, czy jakieś dołączone oprogramowanie lub narzędzie nie zbiera danych w sposób sprzeczny z zasadami naszej organizacji lub lokalnymi przepisami?
- Jakich procesów używa dostawca w celu wykrywania nielegalnego manipulowania w kodzie i jego zapobieganiu, i czy trzeba implementować coś po swojej stronie, na przykład sprawdzać podpisy pakietów?

Nie wolno bezgranicznie ufać zawartości pochodzącej od dostawcy lub firmy trzeciej, dlatego należy implementować własne metody sprawdzania. Podrozdział „Etap testowania obrazu serwera” na stronie 217 zawiera sugestie, jak wbudować do potoku sprawdzanie obrazów serwera.

⁹ Niektóre oparte na skryptach instalatory systemów operacyjnych to Red Hat Kickstart (<https://oreil.ly/AsDhQ>), Solaris JumpStart (<https://oreil.ly/CcETy>), Debian Preseed (https://oreil.ly/3_wlN) oraz plik odpowiedzi instalacji systemu Windows (<https://oreil.ly/UH-5s>).

Zmienianie obrazu serwera

Obrazy serwerów z czasem stają się nieaktualne, w miarę jak pojawiają się nowe wersje pakietów i konfiguracji. Można oczywiście stosować poprawki i aktualizacje przy każdym tworzeniu nowego serwera, ale proces ten wydłuża się wraz z upływem czasu, zmniejszając korzyści płynące z używania obrazu serwera. Aby wszystko działało płynnie, trzeba regularnie odświeżać obrazy.

Oprócz utrzymywania aktualnych obrazów poprzez stosowanie najnowszych poprawek, często trzeba ulepszać podstawową konfigurację, dodawać i usuwać standardowe pakiety oraz dokonywać innych rutynowych zmian w obrazach serwerów. Im więcej jest zawartości upieczonej w niestandardowych obrazach serwerów, a nie usmażonej w instancjach serwerów podczas ich tworzenia, tym częściej trzeba aktualizować obrazy.

W następnym podrozdziale wyjaśnię, jak używać potoku do wprowadzania, testowania i dostarczania zmian do obrazów serwerów. Zanim jednak przejdziemy do potoków, musimy omówić kilka ważnych kwestii. Jedną z nich jest sposób dokonywania aktualizacji – czy chcemy używać istniejącego obrazu serwera jako podstawy dla zmian, czy też chcemy odbudowywać obraz od podstaw. Kolejną kwestią jest wersjonowanie obrazów serwera.

Odrzewanie czy pieczenie świeżego obrazu

Gdy pojawia się konieczność zbudowania nowej wersji obrazu serwera – na przykład w celu dodania najnowszych poprawek systemu operacyjnego – można skorzystać z tych samych narzędzi i procesów, które posłużyły do budowy pierwszej wersji obrazu.

Można użyć poprzedniej wersji obrazu jako obrazu źródłowego i *odgrzać* ten obraz. W ten sposób nowa wersja będzie na pewno spójna z poprzednią wersją, ponieważ jedyne różnice między nimi będą skutkiem zmian wprowadzonych jawnie podczas stosowania konfiguracji serwera do nowego obrazu.

Alternatywą jest *upieczenie świeżego obrazu*, czyli zbudowanie nowej wersji przy użyciu oryginalnych źródeł. Oznacza to, że jeśli pierwsza wersja obrazu została zbudowana od podstaw, każda nowa wersja również musi być budowana od podstaw.

Chociaż odrzewanie obrazów potrafi ograniczyć zmiany między dwiema kolejnymi wersjami, pieczenie świeżego obrazu dla nowej wersji powinno dawać te same rezultaty, zakładając korzystanie z tych samych źródeł. Świeże wersje są zapewne czystsze, a ich odtwarzanie bardziej niezawodne, ponieważ nie zawierają żadnych pozostałości po poprzednich wersjach.

Na przykład jedna konfiguracja serwera może instalować jakiś pakiet, a późniejsza go usuwać. Taki pakiet będzie nadal obecny w nowszych obrazach serwera, jeśli będą tworzone w wyniku odrzewania starszych obrazów. Trzeba dodać kod jawnie usuwający ten pakiet. Taki nadmiarowy kod trzeba utrzymywać, a później wyczyścić. Jeśli zamiast tego będziemy za każdym razem piekli nowy obraz, nowsze obrazy serwera nie będą po prostu zawierały tego pakietu, więc nie trzeba będzie pisać kodu w celu jego usunięcia.

Wersjonowanie obrazu serwera

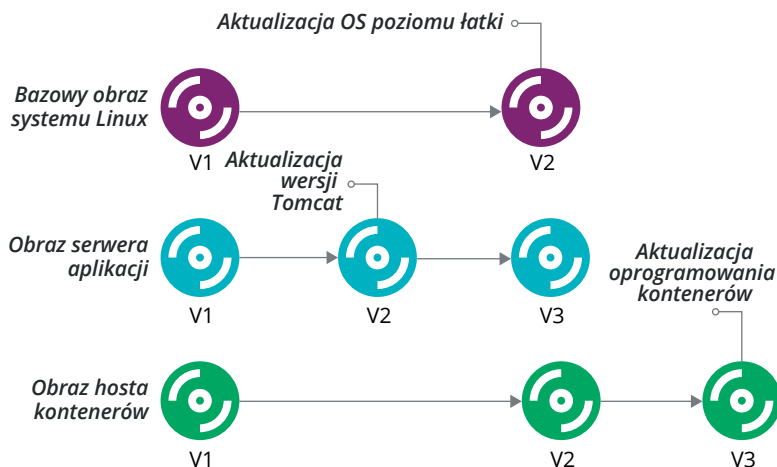
Ważne jest, aby każdy, kto używa obrazów serwera i utworzonych z nich serwerów, mógł śledzić ich wersje. Trzeba wiedzieć, która wersja została użyta do utworzenia danej instancji serwera, która wersja jest najnowsza dla danego obrazu oraz jaka zawartość źródłowa oraz jaki kod konfiguracji zostały użyte do utworzenia obrazu.

Pomocna jest również możliwość wskazania listy wersji obrazów serwera, które zostały wykorzystane do utworzenia aktualnie używanych instancji – na przykład w celu identyfikacji instancji, które wymagają aktualizacji w momencie wykrycia luki w zabezpieczeniach.

Wiele zespołów używa różnych obrazów serwerów. Na przykład można budować oddzielne obrazy dla serwera aplikacji i hosta kontenera, a do tego mieć obraz systemu operacyjnego Linux ogólnego przeznaczenia. W takim przypadku każdy z tych obrazów jest zarządzany jako oddzielny komponent, z własną historią wersji, jak to jest pokazane na rysunku 13-4.

Na tym rysunku zespół najpierw aktualizuje obraz serwera aplikacji, aby uaktualnić zainstalowaną na nim wersję Tomcat. W efekcie powstaje wersja 2 tego obrazu, ale pozostałe obrazy zachowują wersję 1. Następnie zespół aktualizuje wszystkie obrazy stosując aktualizacje systemu operacyjnego Linux. W efekcie obraz serwera aplikacji zmienia swoją wersję na 3, a pozostałe obrazy na 2. Na koniec zespół uaktualnia wersję oprogramowania kontenera w obrazie hosta, zmieniając jego wersję na 3.

Większość platform infrastruktury bezpośrednio nie obsługuje numerowania wersji obrazów serwerów. W wielu przypadkach można osadzać numer wersji w nazwie obrazu serwera. Obrazy z przykładu na rysunku 13-4 mogą mieć nazwy `appserver-3`, `basic-linux-2` i `container-node-3`.



Rysunek 13-4 Przykład historii wersji dla wielu obrazów serwerów

Inną opcją jest umieszczanie numerów wersji i nazw obrazów w tagach, jeśli platforma to umożliwia. Obraz może mieć tag `Name=appserver` oraz drugi `Version=3`.

Wybierając mechanizm trzeba zwracać uwagę, aby pozwalał łatwo wyszukiwać, odkrywać i raportować nazwy i wersje obrazów. Większość zespołów, z którymi pracowałem, używała obu wymienionych metod, ponieważ tagi łatwo jest wyszukiwać, a nazwy bardziej przyciągają ludzki wzrok.

Można stosować różne schematy numerowania wersji. Popularnym podejściem jest wersjonowanie semantyczne¹⁰, wykorzystujące pola trzyczęściowego numeru wersji (np. 1.0.4) w celu wskazania, jak znacząca jest zmiana wersji na kolejną. Lubię dodawać do wersji datę i godzinę (na przykład 1.0.4-20200229_0420), aby od razu wiedzieć, kiedy dana wersja została zbudowana.

Oprócz umieszczania numeru wersji w samym obrazie serwera można również dodawać tagi lub etykiety do instancji serwerów z nazwą i wersją obrazu serwera, który posłużył do ich utworzenia. Pozwala to mapować każdą instancję z powrotem na obraz użyty do jej utworzenia. Jest to przydatne przy wdrażaniu nowych wersji obrazów.

Aktualizowanie instancji serwera po zmianie obrazu

Budując nową wersję obrazu serwera można od razu zastąpić wszystkie instancje tego serwera na podstawie nowego obrazu albo poczekać, aż zostaną one zastąpione z czasem w naturalny sposób.

Zasada ponownego budowania istniejących serwerów w celu zastąpienia starych wersji obrazu może być uciążliwa i czasochłonna. Ale za to zapewnia spójność serwerów i powoduje nieustanne sprawdzanie odporności systemu (patrz także „Ciągłe odzyskiwanie po awarii” na stronie 370). Dobry potok pozwala łatwo zarządzać takim procesem, a zmiany niepowodujące przestojów (patrz „Zmiany bez przestojów” na stronie 366) sprawiają, że jest on mniej uciążliwy. W efekcie zasada ta jest preferowana przez większość zespołów stosujących dojrzałą i powszechną automatyzację.

Łatwiej jest poczekać z wymianą serwerów na nową wersję obrazu, aż będą one odbudowywane z innych powodów. Taka sytuacja miewa miejsce w przypadku wdrażania aktualizacji oprogramowania lub innych zmian za pomocą budowania nowych instancji serwera (patrz „Wzorzec: serwer niezmienny” na stronie 189).

Wadą czekania na aktualizację serwerów na podstawie nowych wersji obrazów jest to, że może to trochę potrwać i pozostawić nas z serwerami utworzonymi z wielu różnych wersji obrazów.

Taka sytuacja powoduje niespójność. W efekcie możemy mieć serwery aplikacji z różnymi wersjami aktualizacji pakietów systemu operacyjnego i konfiguracji, co powoduje niezrozumiały brak spójności w działaniu. W niektórych przypadkach starsze wersje obrazów i zbudowane z nich serwery mogą mieć luki w zabezpieczeniach lub inne problemy.

¹⁰ <https://semver.org>

Jest kilka strategii, których można używać do łagodzenia tych problemów. Jedną z nich jest śledzenie działających instancji i porównywanie ich z wersjami obrazów użytych do ich utworzenia. Może to być wykaz lub raport podobny do tabeli 13-1.

Tabela 13-1 Przykładowy raport na temat instancji i wersji ich obrazów

Obraz	Wersja	Liczba instancji
basic-linux	1	4
basic-linux	2	8
appserver	1	2
appserver	2	11
appserver	3	8
container-node	1	2
container-node	2	15
container-node	3	5

Jeśli taka informacja jest łatwo dostępna i pozwala zagłębić się w listę określonych serwerów, to można zidentyfikować instancje serwerów, które wymagają pilnej odbudowy. Wyobraźmy sobie, że dowiadujemy się o luce w zabezpieczeniach, którą usuwa najnowsza aktualizacja naszej dystrybucji Linuksa. Dołączamy poprawkę do obrazu basic-linux-2, appserver-3 i container-node-2. Na podstawie raportu widzimy, że musimy odbudować 19 instancji serwerów (4 podstawowe serwery linuxowe w wersji 1, 13 serwerów aplikacji w wersjach 1 i 2 oraz 2 węzły kontenera w wersji 1).

Mogą również występować ograniczenia wiekowe. Na przykład może obowiązywać zasada nakazująca zastępowanie wszystkich działających instancji serwerów, zbudowanych z obrazu serwera starszego niż trzy miesiące. Raport lub wykaz powinien wtedy podawać liczbę instancji, które przekroczyły ten limit.

Udostępnianie obrazu serwera do wykorzystania przez wiele zespołów

W wielu organizacjach obrazy serwerów są budowane przez centralny zespół, który następnie udostępnia je innym zespołom. Taka sytuacja stwarza dodatkowe problemy w zarządzaniu aktualizacjami obrazów serwerów.

Zespół wykorzystujący obraz musi mieć pewność, że jest gotowy do używania nowej wersji tego obrazu. Wyobraźmy sobie, że wewnętrzny zespół aplikacji zainstalował w podstawowym obrazie Linuksa, udostępnionym przez zespół obliczeniowy, oprogramowanie śledzące błędy. Zespół obliczeniowy może w pewnym momencie wyprodukować nową wersję podstawowego obrazu Linuksa z aktualizacjami, które będą niezgodne z oprogramowaniem do śledzenia błędów.

W modelu idealnym obrazy serwera są przekazywane do potoku infrastruktury każdego zespołu. Gdy zespół będący właścicielem obrazu wypuszcza jego nową wersję z potoku, to potok infrastruktury każdego zespołu pobiera tę wersję i aktualizuje swoje instancje

tego serwera. Potok wewnętrznego zespołu powinien automatycznie sprawdzać, czy oprogramowanie śledzące błędy działa z nową wersją obrazu, zanim odbuduje serwery potrzebne jego użytkownikom.

Niektóre zespoły mogą stosować bardziej konserwatywne podejście i *przypinać* numer wersji obrazu, której należy używać. Wewnętrzny zespół aplikacji mógłby przypiąć do swojej infrastruktury wersję 1 podstawowego obrazu Linuksa. Z chwilą wypuszczenia przez zespół obliczeniowy wersji 2 tego obrazu, wewnętrzny zespół aplikacji nadal używałby wersji 1, do momentu, w którym byłby gotów przetestować i wdrożyć wersję 2.

Wiele zespołów stosuje metodę przypinania, jeśli ich automatyzacja testowania i dostarczania zmian w infrastrukturze serwera nie jest zbyt dojrzała. Takie zespoły muszą wykonywać więcej działań ręcznie, aby mieć pewność co do zmiany obrazu.

Nawet w przypadku dojrzałej automatyzacji, niektóre zespoły przypinają obrazy (i inne zależności), których zmiany są dość ryzykowne. Zależności mogą wymagać więcej ostrożności, ponieważ oprogramowanie, które zespół wdraża w obrazach, jest bardziej wrażliwe na zmiany w systemie operacyjnym. Powodem może być też zespół dostarczający obraz – być może zewnętrzna grupa – który jest znany z wypuszczania wersji z problemami, więc zaufanie do niego jest niskie.

Obsługa istotnych zmian w obrazie

Niektóre zmiany w obrazach serwerów są bardziej znaczące i przez to wymagają często ręcznego testowania i modyfikowania zależności. Na przykład nowa wersja obrazu serwera aplikacji może obejmować ważne uaktualnienie oprogramowania tego serwera.

Zamiast więc traktować to jako drobną aktualizację obrazu serwera, można użyć wersjonowania semantycznego i wskazać, że jest to bardziej znacząca zmiana albo nawet zbudować całkowicie inny obraz serwera.

W przypadku używania wersjonowania semantycznego, większość zmian powoduje zwiększenie najniższej cyfry wersji, na przykład 1.2.5 na 1.2.6. Ważna zmiana jest wskazywana przez zwiększenie drugiej lub pierwszej cyfry. W przypadku drobnej aktualizacji oprogramowania serwera aplikacji, która nie powinna powodować problemów ze zgodnością aplikacji, można zwiększyć numer wersji obrazu serwera z 1.2.6 na 1.3.0. W przypadku istotnej zmiany, która może skutkować awariami aplikacji, trzeba zwiększyć pierwszą cyfrę, czyli z 1.3.0 przejść na 2.0.0.

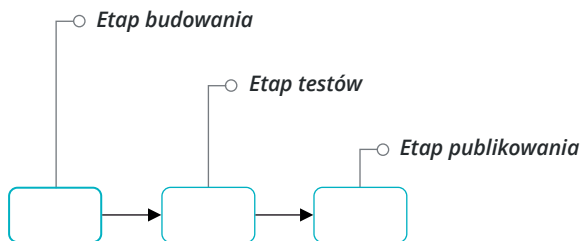
W niektórych przypadkach, zwłaszcza gdy zespoły mają w planie używanie starszej wersji obrazu przez jakiś czas, zamiast zmieniać numer wersji można utworzyć całkiem nowy obraz. Na przykład, jeśli podstawowy obraz Linuksa wykorzystuje Centos 9.x i chcemy rozpocząć testowanie i wdrażanie wersji Centos 10.x, to zamiast zwiększać numer wersji obrazu, można utworzyć nowy obraz o nazwie `basix-linux10-1.0.0`. W ten sposób migracja do nowej wersji systemu operacyjnego będzie bardziej wyraźnym działaniem niż rutynowa aktualizacja obrazu.

Używanie potoku do testowania i dostarczania obrazu serwera

W rozdziale 8 opisałem używanie potoków do testowania i dostarczania zmian kodu infrastruktury („Potoki dostarczania infrastruktury” na stronie 113). Potoki są również doskonałym sposobem budowania obrazów serwerów. Korzystanie z potoku ułatwia budowanie nowych wersji obrazu, zapewniając ich spójność. Mając dojrzały potok zintegrowany z potokami dla innych części systemu można bezpiecznie wdrażać poprawki i aktualizacje systemu operacyjnego w całym swoim środowisku raz na tydzień albo nawet codziennie.

Dobrym pomysłem jest częste budowanie nowych obrazów serwerów, na przykład co tydzień, zamiast robienia dłuższych przerw, rzędu kilku miesięcy. Jest to w zgodzie z podstawową praktyką ciągłego testowania i dostarczania zmian. Im dłużej się czeka z budową nowego obrazu serwera, tym więcej zmian zawiera taki obraz i w efekcie tym więcej pracy trzeba włożyć w testowanie i usuwanie problemów.

Podstawowy potok obrazu serwera składa się z trzech etapów, pokazanych na rysunku 13-5.

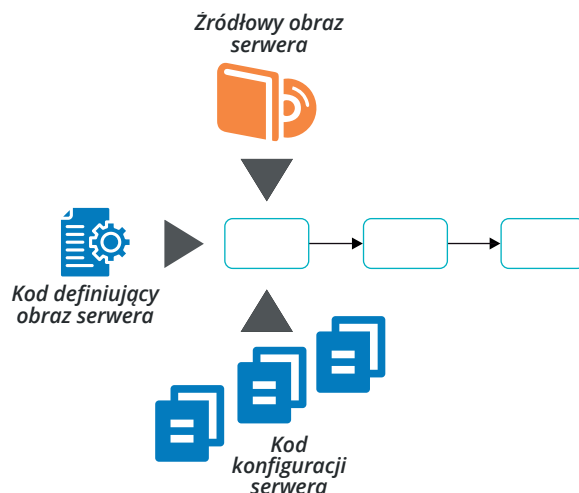


Rysunek 13-5 Prosty potok obrazu serwera

Każdy z tych etapów – budowa, testowanie i publikacja obrazu serwera – zasługuje, aby dokładniej mu się przyjrzeć.

Etap budowy obrazu serwera

Etap budowy obrazu serwera w ramach potoku obejmuje automatyczną implementację procesu budowy obrazu w trybie online („Proces budowania obrazu online” na stronie 204) lub procesu budowy obrazu w trybie offline („Proces budowania obrazu offline” na stronie 207). Efektem tego etapu jest obraz serwera, traktowany w następnych etapach jako kandydat do wydania (patrz rysunek 13-6).



Rysunek 13-6 Etap budowy obrazu serwera

Można tak skonfigurować etap budowy obrazu serwera, aby był uruchamiany automatycznie w wyniku zmiany jakichkolwiek danych wejściowych obrazu wymienionych w podrozdziale „Zawartość źródłowa obrazu serwera” na stronie 208. Oto kilka przykładów:

- Ktoś zatwierdza zmianę w kodzie konstruktora obrazu, na przykład szablon Packera
- Dostawca publikuje nową wersję obrazu źródłowego, na przykład AMI od dostawcy systemu operacyjnego
- Sami dokonujemy zmian w kodzie konfiguracji serwera używanym do obrazu serwera
- Dostawcy pakietów zainstalowanych w obrazie serwera publikują ich nowe wersje

W praktyce wiele zmian elementów źródłowych, takich jak pakiety systemu operacyjnego, trudno jest wykrywać automatycznie i wykorzystywać do wyzwalania potoku. Można też nie chcieć budować nowego obrazu serwera dla każdej aktualizacji pakietu, zwłaszcza jeśli obraz wykorzystuje dziesiątki czy nawet setki pakietów.

Można więc automatycznie wyzwać nowe obrazy tylko dla ważnych zmian, na przykład źródłowego obrazu serwera lub kodu konstruktora obrazu. Następnie można zaimplementować harmonogram budowania – na przykład co tydzień – w celu wdrażania wszystkich aktualizacji drobniejszych elementów źródłowych.

Jak już wspomniałem, wynikiem etapu budowy jest obraz serwera, którego można użyć jako kandydata do wydania. Jeśli przed udostępnieniem obrazu chcemy go przetestować – zalecane w przypadku wszystkich nietrywialnych zastosowań – to w tym momencie nie należy oznaczać obrazu jako gotowego do użycia. Można dołączyć do obrazu tag z numerem wersji, zgodnie z wcześniejszymi wskazówkami. Można również dołączyć tag wskazujący, że jest to nieprzetestowany kandydat do wydania; na przykład `Release_Status=Candidate`.

Etap testowania obrazu serwera

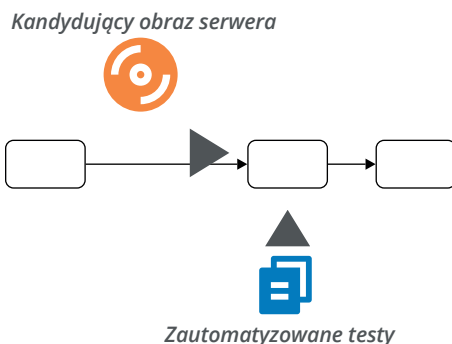
Biorąc pod uwagę nacisk kładziony w tej książce na testowanie, jest chyba zrozumiałe, jaką wartość ma automatyczne testowanie nowych obrazów podczas ich budowania. Zazwyczaj obrazy serwerów są testowane za pomocą tych samych narzędzi do testowania, które są używane do testowania kodu serwera (patrz „Testowanie kodu serwera” na stronie 171).

W przypadku budowania obrazów serwerów online (patrz „Proces budowania obrazu online” na stronie 204), można uruchamiać zautomatyzowane testy instancji serwera po jej skonfigurowaniu, a przed wyłączeniem i konwersją na obraz. Ale są z tym dwa problemy. Po pierwsze, testy mogą zanieczyścić obraz, pozostawiając pliki testowe, wpisy w dzienniku lub inne elementy, jako efekt uboczny testów. Mogą nawet pozostawiać konta użytkowników lub inne formy dostępu do serwerów tworzonych na podstawie obrazu.

Innym problemem związanym z testowaniem instancji serwera przed jej przekształceniem na obraz jest brak pewności wiarygodnej replikacji serwerów, które będą tworzone z obrazu. Proces konwersji instancji na obraz może zmieniać ważne aspekty serwera, na przykład prawa dostępu użytkowników.

Najbardziej wiarygodną metodą testowania obrazu serwera jest posłużenie się nową instancją utworzoną z końcowego obrazu. Wadą tego podejścia jest wydłużony czas otrzymywania informacji zwrotnej z testów.

Tak więc typowy etap testowania obrazu serwera, jak widać na rysunku 13-7, obejmuje pobranie identyfikatora obrazu utworzonego na etapie budowy, wykorzystanie go do utworzenia instancji tymczasowej i uruchomienie dla niej zautomatyzowanych testów. Jeśli testy zakończą się pomyślnie, do obrazu dołączany jest tag wskazujący, że obraz jest gotowy do następnego etapu potoku.

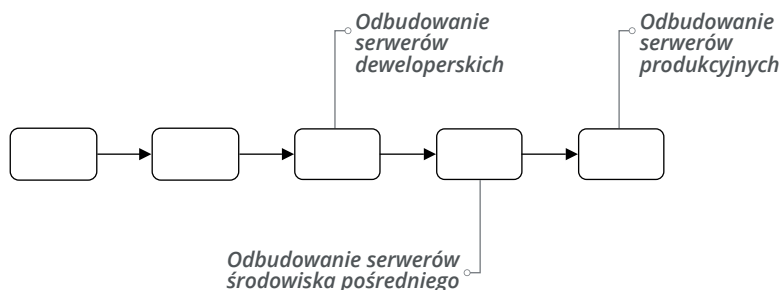


Rysunek 13-7 Etap testowania obrazu serwera

Etap dostarczania obrazu serwera

Potok obrazu może zawierać dodatkowe etapy dla innych działań – na przykład testowania zabezpieczeń. Na koniec wersja obrazu, która przeszła wszystkie etapy testowania, jest oznaczana jako gotowa do użytku.

W niektórych systemach potok, który tworzy obraz, obejmuje etapy, w ramach których następuje odbudowa serwerów z nowego obrazu (patrz rysunek 13-8). Potok może uruchamiać etap dla każdego środowiska dostarczania w drodze do produkcji, zapewne wyzwalając etapy wdrażania i testowania aplikacji.



Rysunek 13-8 Potok dostarczający obrazy do środowisk

W systemach, w których zarządzanie infrastrukturą i środowiskami podlega różnym zespołom, potok obrazu może się kończyć z chwilą oznaczenia obrazu jako gotowego do użycia. Potoki innych zespołów pobierają wówczas nową wersję obrazu jako dane wejściowe, zgodnie z opisem w podrozdziale „Udostępnianie obrazu serwera do wykorzystania przez wiele zespołów” na stronie 213.

Używanie wielu obrazów serwera

Niektóre zespoły utrzymują tylko jeden obraz serwera. Tworzą one różne potrzebne im typy serwerów stosując odpowiednie role serwera (patrz „Role serwerów” na stronie 169) podczas tworzenia instancji serwerów. Natomiast inne zespoły uznają za przydatne lub wręcz konieczne utrzymywanie wielu obrazów serwera.

Trzymanie wielu obrazów może być potrzebne do obsługi wielu platform infrastruktury, wielu systemów operacyjnych oraz wielu architektur sprzętowych. Wiele obrazów może również pomagać w optymalizacji czasu tworzenia serwera, jeśli jest stosowana strategia pieczenia zamiast smażenia („Pieczenie obrazów serwera” na stronie 182).

Przyjrzyjmy się teraz każdemu z tych scenariuszy – obsłudze wielu platform i zapiekaniu ról w obrazach serwerów – a następnie omówimy strategię utrzymywania wielu obrazów serwera.

Obrazy serwera dla różnych platform infrastruktury

Organizacja może używać więcej niż jednej platformy infrastruktury w ramach różnych strategii chmurowych – wielochmurowej, polichmurowej¹¹ oraz hybrydowej. Każda z tych platform wymaga budowy oddzielnych obrazów serwera.

Często daje się używać tego samego kodu konfiguracji dla różnych platform, stosując zapewne jakieś odmiany, którymi można zarządzać przy użyciu parametrów. Można zacząć od różnych obrazów źródłowych dla każdej platformy, chociaż w niektórych przypadkach może to być obraz od dostawcy systemu operacyjnego lub zaufanej firmy trzeciej, który umożliwia spójne budowanie dla wszystkich platform.

Każda platforma infrastruktury powinna mieć oddzielny potok do budowania, testowania i dostarczania nowych wersji obrazu. Potoki te mogą pobierać kod konfiguracji serwera jako materiał wejściowy. Na przykład w momencie aktualizacji kodu konfiguracji serwera należy wypchnąć tę zmianę do potoku każdej platformy, aby dla wszystkich zbudować i przetestować nowy obraz serwera.

Obrazy serwera dla różnych systemów operacyjnych

W przypadku obsługi wielu systemów operacyjnych lub dystrybucji, takich jak Windows, Red Hat Linux i Ubuntu Linux, trzeba utrzymywać dla nich oddzielne obrazy. Ponadto prawdopodobnie potrzebny jest oddzielny obraz dla każdej ważnej wersji systemu operacyjnego używanej przez organizację. Budowa każdego z tych obrazów wymaga oddzielnego obrazu źródłowego. Niektóre z tych obrazów mogą pozwalać na korzystanie z tego samego kodu konfiguracji serwera.

Ale w wielu przypadkach pisanie kodu konfiguracji serwera obsługującego różne systemy operacyjne zwiększa złożoność, co oznacza bardziej skomplikowane testowanie. Potok testowania musi użyć instancji serwera (kontenera) dla każdej wersji. W przypadku niektórych konfiguracji może być łatwiej napisać oddzielny moduł konfiguracji kodu dla każdego systemu operacyjnego lub dystrybucji.

Obrazy serwera dla różnych architektur sprzętowych

Niektóre organizacje uruchamiają instancje serwerów na sprzęcie o różnej architekturze CPU, takiej jak x86 czy ARM. Najczęściej można zbudować niemal identyczne obrazy serwera dla każdej architektury, używając tego samego kodu. Niektóre aplikacje wykorzystują określone funkcje sprzętowe i są bardziej wrażliwe na różnice między nimi. W takich przypadkach potoki powinny dokładniej testować obrazy na różnych architekturach, aby wykryć problemy.

¹¹ <https://oreil.ly/eCayw>

Obrazy serwera dla różnych ról

Wiele zespołów wykorzystuje obraz serwera ogólnego przeznaczenia i podczas tworzenia jego instancji stosuje konfigurację roli (patrz „Role serwerów” na stronie 169). Gdy jednak konfigurowanie roli serwera wymaga instalowania zbyt dużej ilości oprogramowania, jak w przypadku serwera aplikacji, lepszym rozwiązaniem może być upieczenie konfiguracji roli w dedykowanym obrazie.

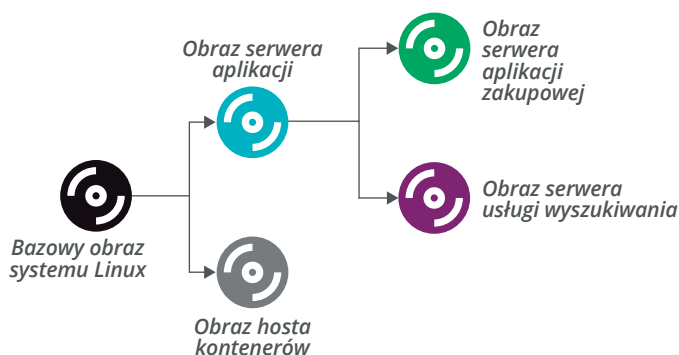
Wcześniejszy przykład w tym rozdziale („Wersjonowanie obrazu serwera” na stronie 211) wykorzystywał niestandardowe obrazy dla serwerów aplikacji i hostów kontenerów oraz obraz systemu Linux ogólnego przeznaczenia. W kolejnym przykładzie widać, że nie jest konieczne trzymanie oddzielnego obrazu dla każdej używanej roli serwera.

Aby zdecydować, czy warto jest utrzymywać oddzielny obraz dla konkretnej roli, należy wziąć pod uwagę czas i koszt utrzymywania potoków, testów i obszaru pamięci, i porównać to z wadami konfigurowania instancji serwerów podczas ich tworzenia – szybkością, wydajnością i zależnościami (patrz „Smażenie instancji serwera” na stronie 181).

Warstwowanie obrazów serwera

W przypadku dużej liczby obrazów serwerów opartych na rolach można rozważyć ułożenie ich w warstwy. Na przykład tworzymy podstawowy obraz systemu operacyjnego z domyślnymi pakietami i konfiguracjami, które chcemy instalować na wszystkich serwerach. Następnie używamy tego obrazu jako obrazu źródłowego do budowy bardziej konkretnych, opartych na rolach, obrazów dla serwerów aplikacji, węzłów kontenerów i tak dalej, jak to jest pokazane na rysunku 13-9.

W tym przykładzie zespół ShopSpinner wykorzystuje podstawowy obraz Linuksa jako obraz źródłowy do budowy obrazu serwera aplikacji i obrazu węzła hostującego kontener. Obraz serwera aplikacji ma wstępnie zainstalowane dwa produkty: Java i Tomcat. Zespół wykorzystuje następnie ten obraz jako obraz źródłowy do budowy obrazów z bardziej konkretnymi, wstępnie zainstalowanymi aplikacjami.



Rysunek 13-9 Warstwowanie obrazów serwera



Nadzór i obrazy serwerów

Tradycyjne podejścia do nadzoru obejmują często ręczny proces przeglądania i zatwierdzania za każdym razem, gdy jest budowany nowy obraz serwera. Jak wyjaśniam w podrozdziale „Nadzór w przepływie pracy opartym na potoku” na stronie 343, definiowanie serwerów jako kodu i używanie potoków do dostarczania zmian daje okazję do bardziej zdecydowanych podejść.

Zamiast sprawdzać nowe obrazy serwerów, osoby odpowiedzialne za nadzór mogą sprawdzać kod, który je tworzy. Co więcej, mogą wraz z zespołem ds. infrastruktury i innymi zaangażowanymi utworzyć zautomatyzowaną kontrolę, która będzie dokonywana w potokach w celu zapewnienia zgodności. Potoki nie tylko mogą przeprowadzać taką kontrolę za każdym razem, gdy jest budowany nowy obraz, ale mogą także przeprowadzać ją dla nowych instancji serwerów, aby upewnić się, czy zastosowanie kodu konfiguracji serwera nie spowodowało jego „rozmiękczenia”.

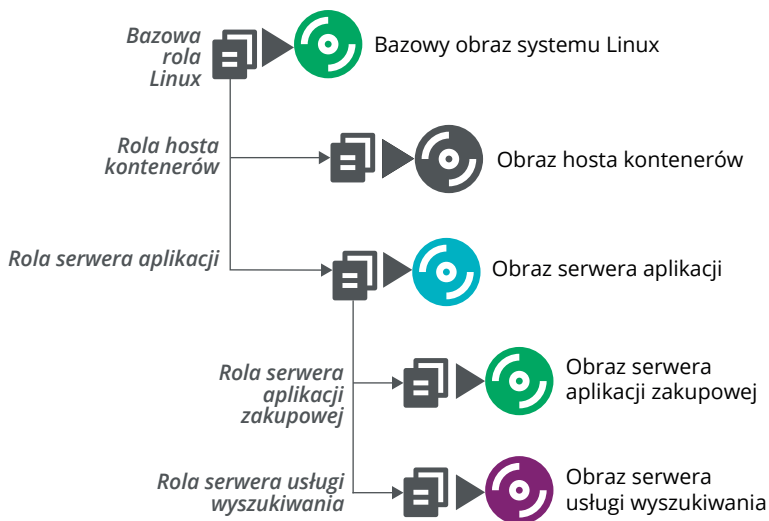
Aplikacja zakupowa i usługa wyszukiwania ShopSpinner wymagają częstego skalowania w górę i w dół, aby obsłużyć natężenie ruchu zmieniające się o różnych porach dnia, dlatego dobrze jest mieć dla nich gotowe, upieczone obrazy. Dla innych aplikacji i usług zespół buduje instancje serwerów przy użyciu bardziej ogólnego obrazu serwera aplikacji, ponieważ nie tworzy takich instancji zbyt często.

Obraz każdego serwera jest budowany przez oddzielny potok, wyzwalany w momencie zmiany obrazu wejściowego. Pętle informacji zwrotnej mogą być bardzo długie. Gdy zespół zmienia podstawowy obraz Linuksa, potok tej zmiany wyzwała na koniec potoki serwera aplikacji i hosta kontenerów. Następnie potok obrazu serwera aplikacji wyzwała potoki budujące na jego podstawie nowe wersje obrazów serwera. Jeśli wykonanie każdego potoku zajmuje 10 – 15 minut, oznacza to, że propagacja zmian do wszystkich obrazów może potrwać do 45 minut.

Alternatywą jest użycie płytszej strategii budowania obrazów.

Współdzielenie kodu przez obrazy serwerów

W przypadku modelu opartego na dziedziczeniu, w którym role serwerów dziedziczą konfigurację po sobie, niekoniecznie trzeba budować obrazy według opisanego przed chwilą modelu warstwowego. Zamiast tego można tworzyć warstwy kodu konfiguracji serwera w definicjach ról serwera i stosować cały kod bezpośrednio do budowy każdego obrazu od podstaw, jak to jest pokazane na rysunku 13-10.



Rysunek 13-10 Warstwy ról w obrazach serwera

Taka strategia implementuje identyczny tryb warstwowy, ale realizuje to łącząc odpowiedni kod konfiguracji dla każdej roli i stosując go jednocześnie. W efekcie do budowy obrazu serwera aplikacji nie jest wykorzystywany obraz podstawowego serwera, lecz zamiast tego stosowany jest cały używany kod, od obrazu podstawowego serwera po obraz serwera aplikacji.

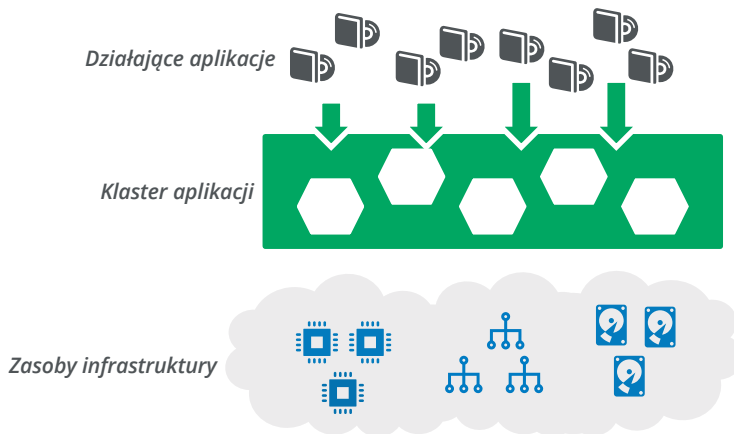
Budowanie obrazów w ten sposób skraca czas budowy obrazów na samym dole hierarchii. Potok, który buduje obraz dla usługi wyszukiwania, jest wykonywany natychmiast po tym, gdy ktoś dokona zmiany w kodzie konfiguracji podstawowego serwera linuksowego.

Podsumowanie

Można budować i utrzymywać serwery bez tworzenia niestandardowych obrazów. Jednak utworzenie i utrzymywanie zautomatyzowanego systemu do budowania obrazów serwerów ma swoje zalety. Zapiękając wcześniej większą część konfiguracji w obrazach serwerów można szybciej tworzyć instancje serwerów, mniej obciążać sieć i ograniczać ilość zależności.

Budowanie klastrów jako kodu

W rozdziale 3 opisałem klaster hostujący aplikację jako usługę dynamicznie wdrażającą i uruchamiającą instancje aplikacji w puli serwerów (patrz „Zasoby obliczeniowe” na stronie 26). Przykładami systemów klastrów aplikacji są Kubernetes, AWS ECS, HashiCorp Nomad, Mesos i Pivotal Diego. Model ten oddziela problemy orkiestracji aplikacji od problemów udostępniania i konfigurowania serwerów, na których działają (patrz rysunek 14-1).



Rysunek 14-1 *Klaster aplikacji tworzy warstwę pośredniczącą między zasobami infrastruktury a działającymi na nich aplikacjami*

Do definiowania klastra aplikacji i zarządzania nim należy oczywiście używać kodu. Jak wyjaśniłem w rozdziale 10, można zbudować klaster aplikacji jako jeden lub więcej stosów aplikacji w formie kodu. W tym rozdziale pokażę na przykładach, jak to może wyglądać. Przedstawię kilka różnych topologii stosu, w tym pojedynczy stos dla całego klastra oraz podział klastrów na kilka stosów (patrz „Topologie stosu dla klastrów aplikacji” na stronie 226). Będą też diagramy potoków dostarczania zmian kodu infrastruktury dla tych topologii.

Ponadto omówię strategię udostępniania klastrów lub ich nieudostępniania innym środowiskom i zespołom (patrz „Strategie współdzielenia dla klastrów aplikacji” na stronie 234). Na koniec rozdziału opowiem o infrastrukturze dla aplikacji bezserwerowych (patrz „Infrastruktura dla bezserwerowej usługi FaaS” na stronie 240).

Kontenery jako kod

Jedną z wielu zalet kontenerów jest to, że są one definiowane jako kod. Obraz kontenera jest tworzony jeden raz z pliku definicji, a następnie wykorzystywany do tworzenia wielu instancji. Aby efektywnie używać kontenerów, należy traktować je jako niezmiennalne i bezstanowe.

Nie wolno dokonywać zmian w zawartości instancji kontenera; zamiast tego trzeba zmienić definicję, utworzyć nowy obraz, a następnie zamienić instancję. Nie należy też zapisywać stanu w kontenerze. Zamiast tego można zapisywać stan i dane przy użyciu innych zasobów infrastruktury (patrz „Ciągłość danych w zmieniającym się systemie” na stronie 373, w rozdziale 21).

Niezmiennalna i bezstanowa natura kontenerów sprawia, że są one idealnie przystosowane do dynamicznego charakteru platform chmurowych, dlatego też są tak silnie związane z natywnymi architekturami chmur (patrz „Infrastruktura natywna dla chmury i oparta na aplikacjach” na stronie 150).

Rozwiązania dla klastrów aplikacji

Istnieją dwa główne podejścia do wdrażania i utrzymywania klastrów aplikacji. Jedno z nich polega na użyciu zarządzanego klastra jako usługi, zazwyczaj udostępnianej jako część platformy infrastruktury. Drugie polega na wdrożeniu spakowanego rozwiązania klastrowego w zasobach infrastruktury niskiego poziomu.

Klaster jako usługa

Większość platform infrastruktury udostępnia zarządzaną usługę klastra. Można definiować, udostępniać i zmieniać klaster jako kod za pomocą narzędzia do konfiguracji stosu. Używanie klastra jako usługi wymaga utworzenia stosu zawierającego klaster i elementy pomocnicze. Wiele takich klastrów jest opartych na rozwiązaniu Kubernetes, w tym EKS, AKS i GKE. Inne wykorzystują zastrzeżoną usługę kontenerową, taką jak ECS.



Zarządzane klastry Kubernetes nie są warstwą abstrakcji chmury

Na pierwszy rzut oka zarządzane przez dostawców klastry Kubernetes wydają się być doskonałym sposobem na programowanie i uruchamianie aplikacji w sposób niewidoczny na różnych platformach chmurowych.

Można dziś budować aplikacje dla Kubernetes i uruchamiać je w dowolnej chmurze!

W praktyce, chociaż klastr aplikacji może być przydatny jako jedna z części warstwy uruchomieniowej aplikacji, utworzenie pełnej platformy uruchomieniowej wymaga dużo więcej pracy. Zaś prawdziwe wyabstrahowanie klastra w bazowej platformie chmurowej nie jest trywialne.

Aplikacje potrzebują także dostępu do innych zasobów udostępnianych przez klastr, oprócz obliczeniowych, w tym do pamięci i sieci. Zasoby te są inaczej udostępniane przez poszczególne platformy, chyba że uda się je wyabstrahować.

Trzeba także zapewnić usługi, takie jak monitorowanie, zarządzanie tożsamościami oraz zarządzanie wpisami tajnymi. I znowu można albo użyć innej usługi na każdej platformie chmurowej, albo zbudować usługę lub warstwę abstrakcji, którą da się wdrożyć i utrzymywać w każdej chmurze.

Tak więc klastr aplikacji jest w rzeczywistości małą częścią całej platformy hostingu aplikacji. I nawet ta część różni się na ogół zależnie od chmury pod względem wersji, implementacji i narzędzi podstawowego systemu Kubernetes.

Spakowana dystrybucja klastra

Wiele zespołów woli samodzielnie instalować klastry aplikacji i zarządzać nimi, niż używać klastrów zarządzanych. Najpopularniejszym podstawowym rozwiązaniem klastrowym jest Kubernetes.

Do wdrożenia Kubernetes w przeznaczonej dla niego infrastrukturze można użyć instalatora, takiego jak kops¹, Kubeadm² i kubespray³.

Istnieją również dostępne spakowane dystrybucje Kubernetes, obejmujące jeszcze inne usługi i funkcje. Są ich dosłownie dziesiątki. Oto kilka z nich⁴:

- HPE Container Platform⁵ (Hewlett Packard)
- OpenShift⁶ (Red Hat)
- Pivotal Container Services (PKS)⁷ (VMware/Pivotal)
- Rancher RKE⁸

Niektóre z tych produktów stosowały na początku własne formaty kontenerów oraz usługi planowania i orkiestracji aplikacji. Z czasem twórcy wielu z nich przerobili swoje produkty

1 <https://oreil.ly/D3xVB>

2 <https://oreil.ly/XeMTa>

3 <https://kubespray.io>

4 Wykaz certyfikowanych dystrybucji Kubernetesa jest dostępny w witrynie (<https://oreil.ly/HR9VS>).

5 <https://oreil.ly/FARcr>

6 <https://www.openshift.com>

7 <https://oreil.ly/sCN-y>

8 <https://oreil.ly/jxlRa>

na wzór cieszących się ogromną popularnością Dockera i Kubernetes, zaprzestając rozwijania własnych rozwiązań.

Jednak kilka produktów oparło się asymilacji z bogiem Kubernetes⁹. Należą do nich HashiCorp Nomad¹⁰ i Apache Mesos¹¹, oba umożliwiające orkiestrację instancji kontenerów, jak również nieskonteneryzowanych aplikacji w różnych zasobach obliczeniowych. Platforma Cloud Foundry Application Runtime¹² (CFAR) oferuje własną orkiestrację kontenerów Diego¹³, choć pozwala również używać Kubernetes¹⁴.

Topologie stosu dla klastrów aplikacji

Klaster aplikacji składa się z różnych ruchomych części. Jeden zestaw części stanowią aplikacje i usługi zarządzające klastrem. Niektóre z tych usług to:

- Dyspozytor (scheduler) – decyduje o liczbie działających instancji każdej aplikacji i miejscu ich uruchamiania
- Monitorowanie – służy do wykrywania problemów z instancjami aplikacji, aby w razie potrzeby można je było restartować lub przenosić
- Rejestr konfiguracji – służy do przechowywania informacji potrzebnych do zarządzania klastrem i konfigurowania aplikacji
- Wykrywanie usług – umożliwia aplikacjom i usługom ustalanie bieżącej lokalizacji instancji aplikacji
- API i UI zarządzania – zapewniają kontakt z klastrem narzędziom i użytkownikom

Wiele wdrożeń klastrów uruchamia usługi zarządzania na dedykowanych serwerach, niezależnych od usług obsługujących instancje aplikacji. Usługi te powinny również działać w klastrze dla zapewnienia większej odporności.

Innymi ważnymi składnikami klastra aplikacji są węzły hostujące aplikacje. Węzły te są pulą serwerów, na których dyspozytor uruchamia instancje aplikacji. Często konfiguruje się je jako klaster serwerów (patrz „Zasoby obliczeniowe” na stronie 26), aby móc zarządzać liczbą i lokalizacją instancji serwerów. Siatka usług (patrz „Siatka usług” na stronie 238) może uruchamiać procesy „przyczepki” w węzłach hostujących instancje aplikacji.

Pokazany na rysunku przykładowy klaster aplikacji (rysunek 14-2) obejmuje serwery do uruchamiania usług zarządzania klastrem, klaster serwerów do uruchamiania instancji aplikacji oraz bloki adresów sieci.

⁹ <https://oreil.ly/tgr7o>

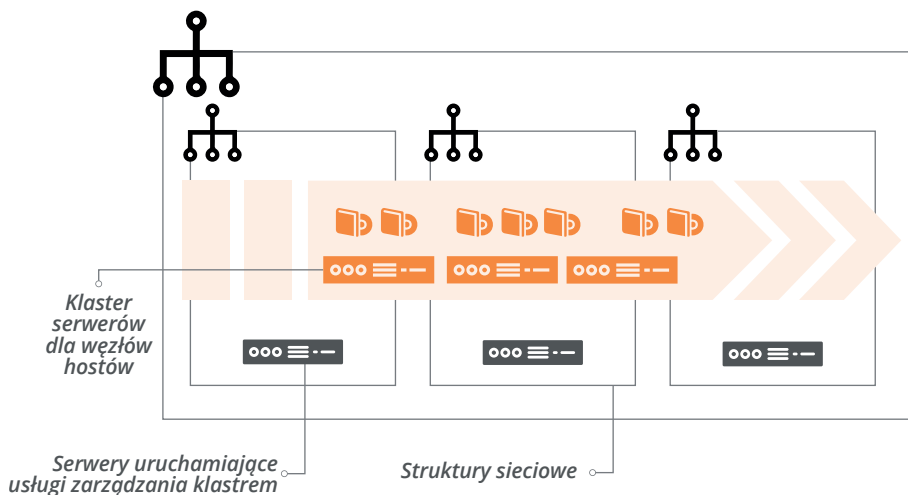
¹⁰ <https://www.nomadproject.io>

¹¹ <http://mesos.apache.org>

¹² <https://oreil.ly/trdM1>

¹³ <https://oreil.ly/vyW86>

¹⁴ Więcej szczegółów można znaleźć w „CF Container Runtime” (<https://oreil.ly/Nptwe>).



Rysunek 14-2 Przykład klastra aplikacji

Struktury sieciowe dla klastra aplikacji mogą być płaskie. Usługi klastra przypisują adresy sieciowe do instancji aplikacji. Powinny również obsługiwać zabezpieczenia sieciowe, w tym szyfrowanie i zarządzanie połączeniami, często przy użyciu siatki usług.

Do udostępniania tej infrastruktury używane są stosy infrastruktury. W rozdziale 5 wyjaśniłem, że rozmiar stosu i zakres jego zawartości mają wpływ na szybkość i ryzyko zmian. W szczególności podrozdział „Wzorce i antywzorce konstruowania stosów” na stronie 52 zawiera listę wzorców rozmieszczania zasobów w stosach. W kolejnych przykładach zobaczymy, jak można stosować te wzorce do infrastruktury klastra.

Stos monolityczny wykorzystujący klaster jako usługę

Najprostszym rozwiązaniem jest zdefiniowanie wszystkich elementów klastra w jednym stosie, zgodnie z antywzorcem stosu monolitycznego (patrz „Antywzorec: stos monolityczny” na stronie 52). Chociaż stos monolityczny jest antywzorcem w przypadku dużego rozmiaru, pojedynczy stos może być przydatny, jeśli zaczynamy od małego, prostego klastra.

W przykładzie 14-1 klaster jest używany jako usługa, podobnie do AWS EKS, AWS ECS, Azure AKS i Google GKE. Zatem kod definiuje klaster, ale nie musi udostępniać serwerów do uruchamiania zarządzania klastrem, ponieważ platforma infrastruktury robi to w tle.

Przykład 14-1 Kod stosu definiujący wszystko dla klastra

```
address_block:
  name: cluster_network
  address_range: 10.1.0.0/16"
```



```

vlangs:
  - vlan_a:
      address_range: 10.1.0.0/8
  - vlan_b:
      address_range: 10.1.1.0/8
  - vlan_c:
      address_range: 10.1.2.0/8

application_cluster:
  name: product_application_cluster
  address_block: $address_block.cluster_network

server_cluster:
  name: "cluster_nodes"
  min_size: 1
  max_size: 3
  vlans: $address_block.cluster_network.vlans
  each_server_node:
    source_image: cluster_node_image
    memory: 8GB

```

W tym przykładzie nie ma wielu rzeczy niezbędnych w przypadku prawdziwego klastra, takich jak trasy sieciowe, zasady bezpieczeństwa czy monitorowanie. Ale widać, że wszystkie istotne elementy dotyczące sieci, definicji klastra i puli serwerów w węzłach hostujących są w tym samym projekcie.

Stos monolityczny dla spakowanego rozwiązania klastrowego

Kod w przykładzie 14-1 wykorzystuje usługę klastra aplikacji udostępnianą przez platformę infrastruktury. Wiele zespołów używa zamiast tego spakowanego rozwiązania klastrowego dla aplikacji (opisanego w podrozdziale „Spakowana dystrybucja klastra” na stronie 225). Takie rozwiązania mają swoje instalatory, które wdrażają na serwerach oprogramowanie do zarządzania klastrem.

W przypadku używania jednego z takich rozwiązań, stos infrastruktury udostępnia infrastrukturę potrzebną instalatorowi do wdrożenia i skonfigurowania klastra. Uruchomienie instalatora powinno być oddzielnym krokiem. W ten sposób można testować stos infrastruktury niezależnie od klastra aplikacji. Konfiguracja klastra powinna być możliwa do zdefiniowania jako kod. Jeśli tak jest, można odpowiednio zarządzać tym kodem, używając testów i potoku, które pomagają w łatwym i bezpiecznym dostarczaniu aktualizacji i zmian.

Do wdrażania na serwerach systemu zarządzania klastrem może być przydatny kod konfiguracji serwera (jak w rozdziale 11). Niektóre spakowane produkty używają standardowych narzędzi do konfiguracji, takich jak Ansible dla OpenShift, więc czasem udaje się włączyć je do procesu budowania stosu. Przykład 14-2 zawiera fragment kodu, który

można dodać do kodu stosu monolitycznego z przykładu 14-1 w celu utworzenia serwera dla aplikacji zarządzającej klastrem.

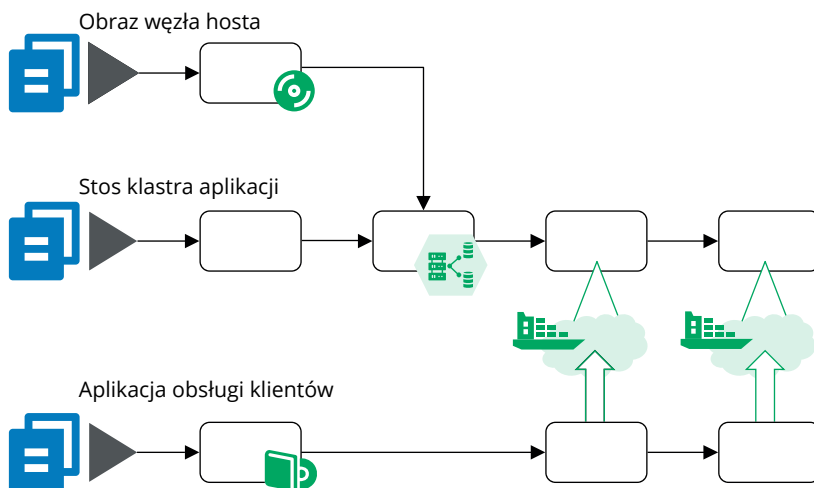
Przykład 14-2 Kod budujący serwera zarządzania klastrem

```
virtual_machine:  
  name: cluster_manager  
  source_image: linux-base  
  memory: 4GB  
  provision:  
    tool: servermaker  
    parameters:  
      maker_server: maker.shoppinner.xyz  
      role: cluster_manager
```

Kod konfiguruje serwer, uruchamiając fikcyjne narzędzie `servermaker` i stosując rolę `cluster_manager`.

Potok dla monolitycznego stosu klastra aplikacji

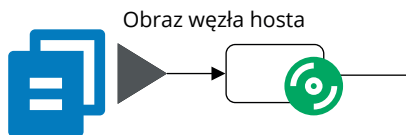
Skoro jest tylko jeden stos, do testowania i dostarczania zmian kodu do instancji klastra aplikacji wystarczy pojedynczy potok. Są jednak potrzebne także potoki dla innych elementów – obrazu serwera dla węzłów hostów i samych aplikacji. Na rysunku 14-3 widać potencjalny projekt tych potoków.



Rysunek 14-3 Przykład potoków dla klastra wykorzystującego stos monolityczny

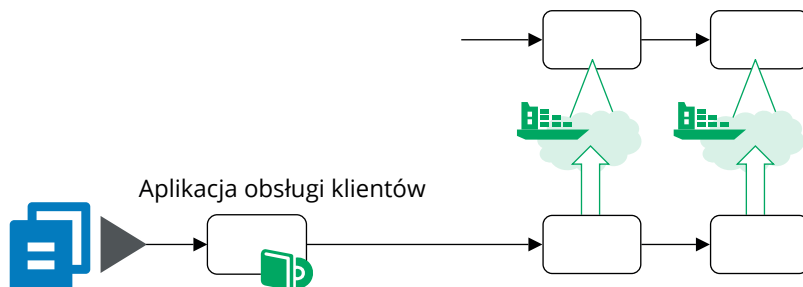
Górnypotok (rysunek 14-4) buduje obraz serwera dla węzłów hostów, zgodnie z opisem w podrzdziale „Używanie potoku do testowania i dostarczania obrazu serwera” na stronie 215.

Rezultatem tego potoku jest obraz serwera przetestowany w izolacji. Testy dla tego obrazu sprawdzają prawdopodobnie, czy jest zainstalowane oprogramowanie do zarządzania kontenerami i czy jest ono zgodne z zasadami bezpieczeństwa.



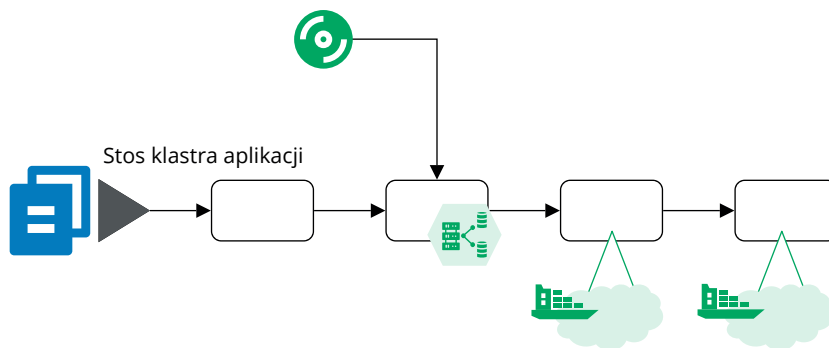
Rysunek 14-4 Potok dla obrazu serwera w węźle hosta

Dolny potok (rysunek 14-5) dotyczy aplikacji wdrażanej w klastrze. W praktyce jest ich kilka, po jednym dla każdej oddzielnie wdrażanej aplikacji. Potok ten obejmuje co najmniej jeden wczesny etap, służący do budowy i testowania samej aplikacji. Następne etapy zajmują się wdrażaniem aplikacji w klastrach w poszczególnych środowiskach. Aplikacja może być testowana, przeglądana i udostępniana do użytku produkcyjnego w każdym z tych środowisk. Potoki dla różnych aplikacji są bardzo luźno sprzężone z potokami dla instancji klastra. Etapy testowania aplikacji mogą być wyzwalane po zaktualizowaniu klastra. Takie rozwiązanie pomaga wykryć wszelkie problemy z aplikacjami, spowodowane dokonaniem zmiany w klastrze.



Rysunek 14-5 Potoki dla dostarczania aplikacji do klastra

Potok dla stosu klastra aplikacji na rysunku 14-6 zaczyna się od etapu offline („Etapy testowania offline dla stosów” na stronie 125), który uruchamia sprawdzanie składni i stosuje kod stosu do lokalnej atrapy platformy infrastruktury („Testowanie z użyciem atrapy API” na stronie 127). Testy te pozwalają wykryć problemy na poziomie kodowania, bez konieczności korzystania z zasobów platformy infrastruktury, więc działają szybko.



Rysunek 14-6 *Potok dla kodu stosu klastra*

Drugi etap tego potoku jest etapem online (patrz „Etapy testowania online dla stosów” na stronie 128), który tworzy instancję stosu na platformie infrastruktury. Instancja może być trwała (patrz „Wzorzec: trwały stos testowy” na stronie 136) albo efemeryczna (patrz „Wzorzec: efemeryczny stos testowy” na stronie 137). Testy na tym etapie mogą sprawdzać, czy usługi zarządzania klastrem są prawidłowo utworzone i dostępne. Można również testować pod kątem problemów z bezpieczeństwem – na przykład upewnając się, że dostęp do punktów końcowych zarządzania klastrem jest zablokowany¹⁵.

Ponieważ ten monolityczny stos klastra zawiera kod do tworzenia serwerów w węzłach hostów, etap testowania online może sprawdzać również ten aspekt. Test może wdrażać przykładową aplikację w klastrze i udowadniać, że działa. Zaletą używania przykładowej aplikacji zamiast prawdziwej jest to, że może być ona uproszczona. Ograniczając do minimum zbiór zależności i konfiguracji można mieć pewność, że jakiegokolwiek niepowodzenie testu będzie spowodowane przez problemy z wyposażaniem klastra i nie ma związku ze złożonością wdrażania rzeczywistej aplikacji.

Zauważmy, że ten etap potoku jest obszerny. Testuje zarówno konfigurację klastra, jak i klastery serwerów w węzłach hostów. Testuje także obraz serwera w kontekście klastra. Jest tu cała masa rzeczy, które mogą spowodować niepowodzenie tego etapu, co komplikuje rozwiązywanie problemów i usuwanie awarii.

Najwięcej czasu podczas tego etapu zajmuje wyposażenie wszystkiego, dużo więcej niż samo wykonanie testów. Te dwa problemy – różnorodność rzeczy testowanych w ramach tego jednego etapu i czas potrzebny na wyposażanie – są głównymi powodami, dla których należy rozbić ten klaster na wiele stosów.

¹⁵ Platforma Kubernetes powodowała kiedyś problemy, umożliwiając korzystanie z API zarządzania bez uwierzytelniania. Postępując zgodnie ze wskazówkami (<https://oreil.ly/Uw4Zh>) można sprawdzić, czy podjęte kroki zabezpieczają klaster i napisać testy wstrzymujące zmiany kodu i konfiguracji, które przypadkowo otwierają wejście dla atakujących.

Przykład wielu stosów dla klastra

Podział kodu infrastruktury dla klastra na wiele stosów może poprawić niezawodność i tempo procesu dokonywania zmian. Każdy stos powinien być tak zaprojektowany, aby umożliwiał udostępnianie i testowanie go w izolacji, bez konieczności tworzenia instancji pozostałych stosów.

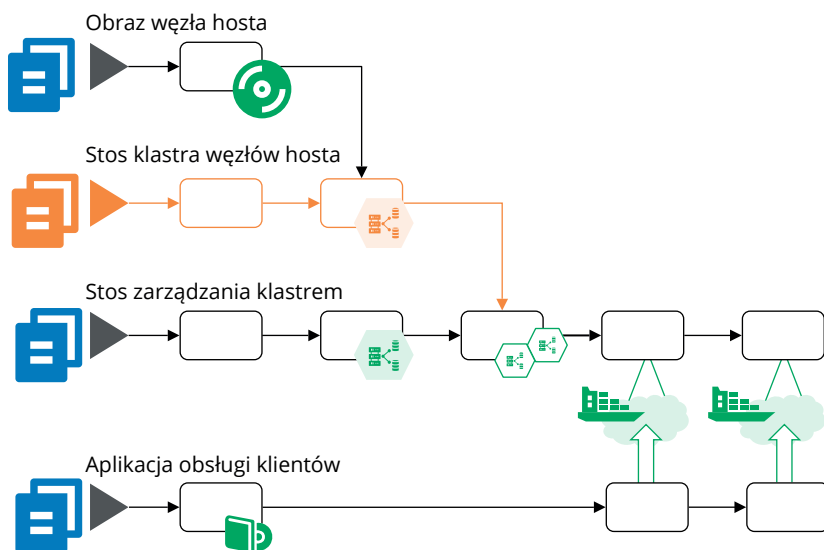
Na początku trzeba pobrać pulę serwerów do osobnego stosu, jak w przykładzie 14-3. Instancję tego stosu można wyposażać i testować bez klastra aplikacji. Testowanie powinno sprawdzać, czy platforma pomyślnie uruchamia serwery z obrazów i czy trasy sieciowe działają poprawnie. Można również testować niezawodność, wywołując problem z jednym z serwerów i sprawdzając, czy platforma automatycznie go zastąpi.

Przykład 14-3 *Kod stosu definiujący pulę serwerów w węzłach hostów*

```
server_cluster:
  name: "cluster_nodes"
  min_size: 1
  max_size: 3
  vlans: $address_block.host_node_network.vlans
  each_server_node:
    source_image: cluster_node_image
    memory: 8GB
address_block:
  name: host_node_network
  address_range: 10.2.0.0/16"
  vlans:
    - vlan_a:
      address_range: 10.2.0.0/8
    - vlan_b:
      address_range: 10.2.1.0/8
    - vlan_c:
      address_range: 10.2.2.0/8
```

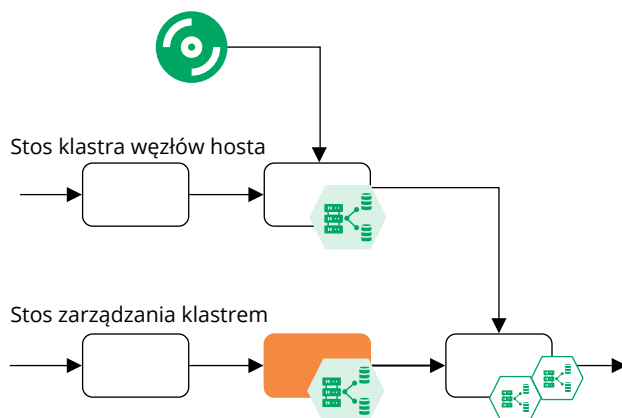
Ten kod dodaje oddzielne sieci VLAN dla węzłów hostów, w odróżnieniu od wcześniejszego kodu dla stosu monolitycznego (patrz przykład 14-1). Dobrą praktyką jest umieszczanie węzłów hostów i zarządzania klastrem w różnych segmentach sieci, co można zrobić w ramach stosu monolitycznego. Prowadzi to do podziału stosu i zapewnia przynajmniej ograniczenie sprzężenia między stosami.

Podział stosu powoduje dodanie nowego potoku dla stosu węzła hosta klastra, jak to jest pokazane na rysunku 14-7.



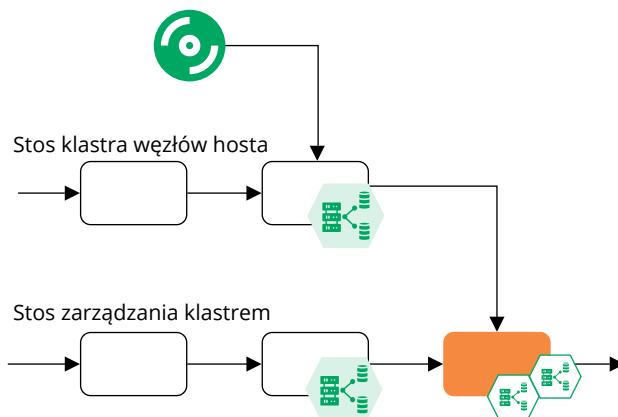
Rysunek 14-7 Dodatkowy potok dla puli węzłów hostów

Chociaż w połączonym potoku jest o kilka etapów więcej, są one lżejsze i szybsze. Etap testowania online dla stosu zarządzania klastrem (wyróżniony na rysunku 14-8) udostępnia jedynie infrastrukturę zarządzania klastrem, co trwa krócej niż etap online w potoku stosu monolitycznego. Ten stos nie zależy już od potoku dla obrazów serwera w węzłach hostów i nie zawiera węzłów serwerów. W efekcie testy na tym etapie mogą koncentrować się na sprawdzaniu, czy zarządzanie klastrem zostało skonfigurowane prawidłowo i bezpiecznie.



Rysunek 14-8 Etapy testowania online dla potoków klastra

Ten poprawiony projekt łączy potok dla stosu serwera węzła hosta z potokiem dla stosu zarządzania klastrem na etapie integracji stosów, jak to jest pokazane na rysunku 14-9.



Rysunek 14-9 Etap testowania integracji stosów dla klastra

To na etapie testowania online następuje udostępnienie i wykonanie testów instancji obu stosów razem. Testy te mogą skupiać się na problemach dotyczących wyłącznie takiej kombinacji, nie muszą więc powielać działań testowych z poprzednich etapów. Na tym etapie można wdrożyć przykładową aplikację i sprawdzić, czy działa prawidłowo w klastrze. Można również przetestować niezawodność i skalowanie, wywołując awarie w aplikacji testowej i tworząc warunki niezbędne do zwiększenia liczby instancji.

Można zdecydować się na utworzenie jeszcze większej liczby stosów, na przykład wydzielając wspólną infrastrukturę sieci ze stosu zarządzania. W rozdziałach 15 i 17 zajmę się bardziej szczegółowo dekompozycją i integracją infrastruktury w stosach.

Strategie współdzielenia dla klastrów aplikacji

Ile klastrów należy uruchomić, jaką powinny mieć wielkość i jak bardzo można je obciążać?

Teoretycznie można mieć jeden klaster z manifestem określającym środowiska i granice aplikacji w instancji tego klastra. Jest jednak wiele powodów, które sprawiają, że pojedynczy klaster może nie być praktycznym rozwiązaniem¹⁶:

Zarządzanie zmianami

Klaster wymaga czasem aktualizacji, poprawek i zmian. Dlatego trzeba mieć możliwość przeprowadzania testów w sposób, który nie będzie zakłócał działania usług. W przypadku zmian przełomowych, czyli wymagających przestoju lub niosących takie ryzyko, planowanie czasu spełniającego wymagania wszystkich zespołów, aplikacji

¹⁶ Rob Hirschfeld opisuje swoje badania kompromisów między wielkością klastra i udostępnianiem w artykule „The Optimal Kubernetes Cluster Size? Let’s Look at the Data” (<https://oreil.ly/8NUDP>).

i regionów może być trudne. Uruchomienie wielu klastrów pozwala łatwiej zaplanować okna na konserwację i ograniczyć skutki nieudanych zmian.

Segregacja

Wiele implementacji klastrów nie zapewnia wystarczająco silnej segregacji między aplikacjami, danymi i konfiguracją. Implementacje klastrów mogą wymagać różnych reżimów nadzoru w zależności od uruchamianych na nich usług. Na przykład usługi obsługujące numery kart kredytowych miewają bardziej restrykcyjne wymagania dotyczące zgodności, dlatego uruchamianie ich w oddzielnym klastrze pozwala uprościć wymagania w innych klastrach.

Konfigurowalność

Niektóre aplikacje, jak i zespoły, mają różne wymagania dotyczące konfiguracji używanych przez siebie klastrów. Udostępnianie im oddzielnych instancji klastrów ogranicza konflikty powodowane przez konfigurację.

Wydajność i skalowalność

Rozwiązania klastrowe mają różną charakterystykę pod względem skalowalności. Wiele z nich nie radzi sobie z większym opóźnieniem, przez co uruchamianie jednego klastra rozciągającego się na różne regiony geograficzne jest niepraktyczne. Aplikacje mogą zderzać się z ograniczonym dostępem do zasobów lub problemami rywalizacji podczas skalowania w górę w jednym klastrze.

Dostępność

Pojedynczy klaster to pojedynczy punkt awarii. Uruchamianie wielu klastrów pomaga poradzić sobie z różnymi scenariuszami awarii.

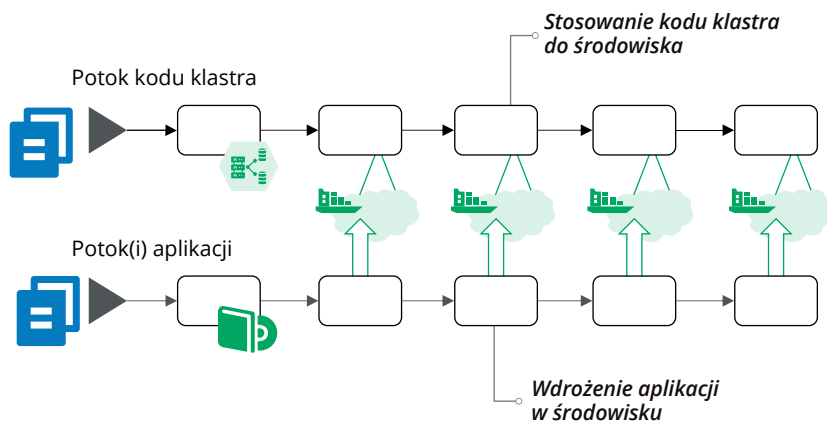
Jest kilka potencjalnych strategii dobierania wielkości instancji klastra i udostępniania. Aby wybrać właściwą strategię dla swojego systemu, należy rozważyć wymagania dotyczące segregacji zmian, konfiguralności, wydajności, skalowania, dystrybucji i dostępności, a następnie przetestować klaster aplikacji pod kątem tych wymagań.

Jeden duży klaster do wszystkiego

Zarządzanie jednym klastrem może być prostsze od zarządzania wieloma klastrami. Prawdopodobnym wyjątkiem jest zarządzanie zmianami. Dlatego należy mieć przynajmniej jedną oddzielną instancję klastra do testowania zmian, wykorzystującą potok do wdrażania i testowania zmian konfiguracji klastra przed zastosowaniem ich do wersji produkcyjnej.

Oddzielne klastry dla etapów dostarczania

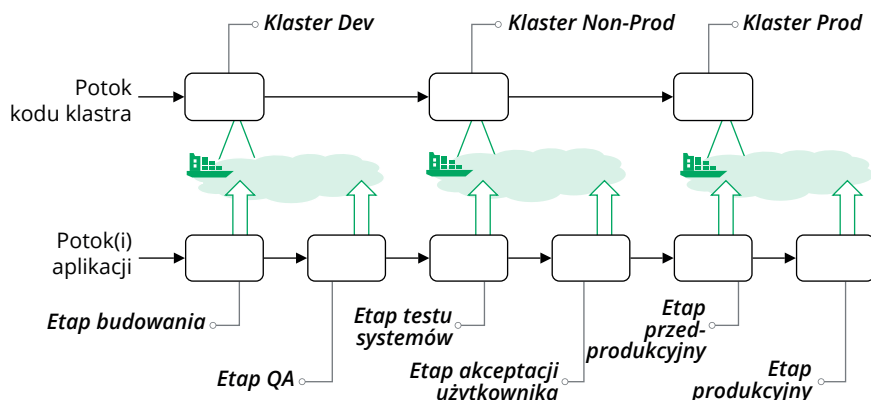
Można uruchamiać różne klastry dla różnych części procesu dostarczania oprogramowania. Może się to sprowadzać do uruchamiania po jednym klastrze dla każdego środowiska (patrz rysunek 14-10).



Rysunek 14-10 *Potoki zarządzające po jednym klastrze dla każdego środowiska wdrażania aplikacji*

Klaster dedykowany dla każdego środowiska pozwala uniknąć niespójności, które mogą występować w przypadku aplikacji pochodzących z wielu środowisk współużytkujących zasoby. Jednak utrzymanie oddzielnej instancji dla każdego etapu dostarczania może być trudne i kosztowne. Na przykład, jeśli proces dostarczania dynamicznie tworzy instancje testowe, może być konieczne dynamiczne tworzenie instancji klastra do ich uruchamiania, a to może być bardzo powolne.

Odmianą rozdzielania klastrów dla etapów dostarczania jest współużytkowanie klastrów na wielu etapach. Na rysunku 14-11 widać trzy klastry. Klaster DEV jest przeznaczony do projektowania i są w nim uruchamiane instancje, w których ludzie tworzą i wykorzystują różne zestawy danych dla bardziej eksploracyjnych scenariuszy testowych. Klaster NON-PROD służy do bardziej rygorystycznych etapów dostarczania, z zarządzanymi zbiorami danych testowych. Klaster PROD hostuje środowiska PREPROD i PROD, z których każde zawiera dane klientów i w efekcie ma jeszcze bardziej rygorystyczne wymagania dotyczące nadzoru.



Rysunek 14-11 Instancje klastrów współdzielone przez wiele środowisk

W przypadku hostingu wielu środowisk we współdzielonym klastrze trzeba dążyć do maksymalnej segregacji wszystkich tych środowisk. Najlepiej, aby aplikacje i usługi operacyjne nie były w stanie widzieć instancji z innych środowisk ani wchodzić z nimi w interakcję. Wiele mechanizmów segregacji aplikacji oferowanych przez rozwiązania klastrów aplikacji jest „miękkich.” Na przykład można czasem oznaczać instancje za pomocą tagów, aby wskazać ich środowisko, ale jest to czysta konwencja. Należy szukać silniejszych metod segregacji aplikacji.

Klastry dla nadzoru

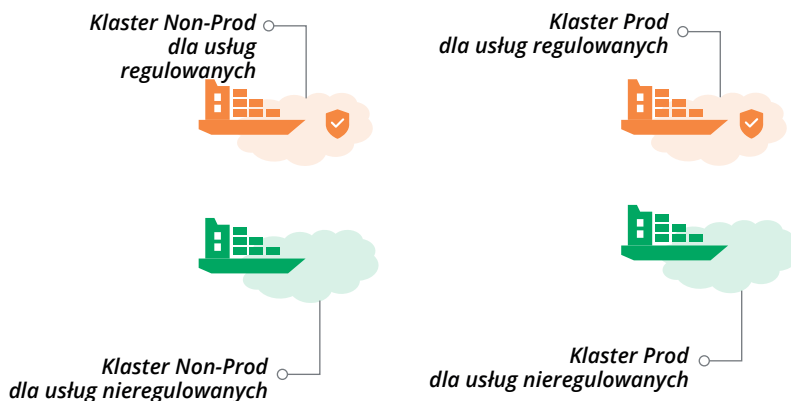
Jedną z zalet posiadania oddzielnych klastrów dla różnych części procesu dostarczania jest możliwość lepszego spełnienia wymagań nadzoru, które zazwyczaj różnią się na poszczególnych etapach tego procesu. Etap produkcyjny ma bardziej rygorystyczne wymagania, ponieważ uruchamianie na tym etapie usługi są najbardziej krytyczne dla firmy, a dane najbardziej wrażliwe.

Dość często różne części systemu mają różne wymagania dotyczące nadzoru i zgodności na poszczególnych etapach dostarczania. Najbardziej typowym przykładem są usługi obsługujące numery kart kredytowych, podlegające określonym standardom¹⁷. Innymi przykładami są usługi dotyczące danych osobowych klientów, które mogą podlegać regulacjom, takim jak RODO.

Hosting usług podlegających bardziej rygorystycznym standardom na dedykowanych klastrach może powodować uproszczenie i wzmocnienie zgodności i inspekcji. Można wprowadzić silniejszą kontrolę tych klastrów, uruchamianych w nich aplikacji i dostarczania zmian do nich. Klastry hostujące usługi o mniej rygorystycznych wymaganiach zgodności pozwalają usprawniać procesy nadzoru i kontroli.

¹⁷ <https://oreil.ly/FUMrX>

Weźmy jako przykład dwa klastry, jeden używany do hostingu projektowania, testowania i produkcji dla usług regulowanych, a drugi dla usług nieregulowanych. Można też podzielić instancje klastra według etapów dostarczania i wymagań regulacyjnych, jak to jest pokazane na rysunku 14-12.



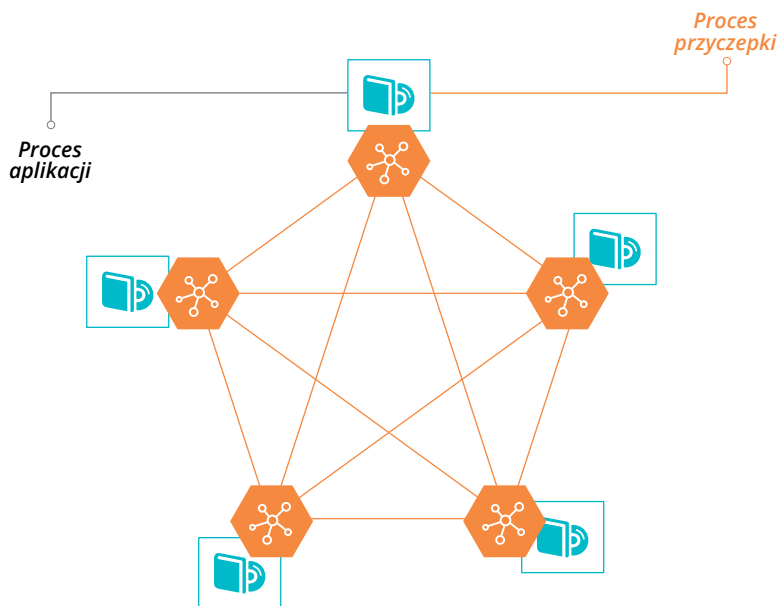
Rysunek 14-12 Oddzielne klastry dla etapów dostarczania i wymagań regulacyjnych

Klastry dla zespołów

Kolejnym powodem organizowania wielu klastrów mogą być zespoły. Często różne zespoły są odpowiedzialne za dostarczanie i uruchamianie różnych rodzajów aplikacji i usług, które mogą mieć różne wymagania dotyczące hostingu. Na przykład zespół zajmujący się usługami dla klientów może stosować inne wymagania dotyczące nadzoru i dostępności niż zespół zajmujący się usługami dla działu wewnętrznego. Klaster przypisany do zespołu może być zoptymalizowany pod kątem wymagań tego zespołu i jego aplikacji.

Siatka usług

Siatka usług to zdecentralizowana sieć usług, która dynamicznie zarządza łącznością między częściami systemu rozproszonego. Siatka przenosi możliwości sieciowe z warstwy infrastruktury do warstwy wykonawczej aplikacji modelu opisanego w podrozdziale „Części systemu infrastruktury” na stronie 21. W typowej implementacji siatki usług, każda instancja aplikacji komunikuje się z innymi instancjami przy użyciu procesu przyczepki (patrz rysunek 14-13).



Rysunek 14-13 Procesy przyczepki umożliwiają komunikację z innymi procesami w siatce usług

Oto niektóre usługi udostępniane aplikacjom przez siatkę usług:

Routing

Kierowanie ruchu do najbardziej odpowiedniej instancji danej aplikacji, niezależnie od tego, gdzie jest aktualnie uruchomiona. Dynamiczny routing za pomocą siatki usług umożliwia zaawansowane scenariusze wdrażania, takie jak niebiesko-zielony i kanarowy, opisane w podrozdziale „Zmiana działającej infrastruktury” na stronie 360.

Dostępność

Wymuszanie reguł ograniczania liczby żądań, na przykład bezpieczników¹⁸.

Bezpieczeństwo

Obsługa szyfrowania, w tym certyfikatów.

Uwierzytelnianie

Wymuszanie reguł określających możliwość nawiązywania połączeń między usługami. Zarządzanie certyfikatami dla uwierzytelniania peer-to-peer.

Obserwowalność, monitorowanie i rozwiązywanie problemów

Rejestrowanie połączeń i innych zdarzeń w celu umożliwienia śledzenia żądań w złożonych systemach rozproszonych.

¹⁸ <https://oreil.ly/thM6m>

Siatka usług działa dobrze w połączeniu z klastrem hostingu aplikacji. Klastery aplikacji dynamicznie udostępnia zasoby obliczeniowe oddzielone od zasobów niższego poziomu. Siatka usług dynamicznie zarządza komunikacją między aplikacjami, oddzieloną od zasobów sieciowych niższego poziomu. Oto zalety tego modelu:

- Upraszcza projektowanie aplikacji, przenosząc wspólne rzeczy poza aplikację, do przyczepki.
- Upraszcza budowanie i ulepszanie wspólnych rzeczy w ramach całej podległej infrastruktury, ponieważ wymaga wdrażania aktualizacji jedynie w przyczepkach, bez potrzeby zmieniania kodu we wszystkich aplikacjach i usługach.
- Obsługuje dynamiczny charakter wdrażania aplikacji, ponieważ ten sam system orkiestracji i planowania, który służy do wdrażania i konfigurowania instancji aplikacji (np. w kontenerach), może służyć do wdrażania i konfigurowania razem z nimi instancji przyczepki.

Przykładami siatek usług są HashiCorp Consul¹⁹, Envoy²⁰, Istio²¹ i Linkerd²².

Siatki usług są najczęściej kojarzone z systemami konteneryzowanymi. Jednak można implementować ten model także w systemach niekonteneryzowanych; na przykład wdrażając procesy przyczepki w maszynach wirtualnych.

Siatka usług zwiększa złożoność. Podobnie jak w przypadku natywnych modeli architektonicznych chmury, takich jak mikrousługi, siatka usług wydaje się atrakcyjna, ponieważ upraszcza tworzenie poszczególnych aplikacji. Jednak złożoność nie znika; zostaje jedynie przeniesiona do infrastruktury. W rezultacie organizacja i tak musi wiedzieć, jak tym zarządzać i być przygotowana na trudny proces nauki.

Bardzo ważne jest zachowywanie wyraźnych granic między siecią zaimplementowaną na poziomie infrastruktury a siecią zaimplementowaną w siatce usług. Bez dobrego projektu i zdyscyplinowanej implementacji można powielić i mieszać różne rzeczy. System będzie przez to trudniejszy do zrozumienia, dokonywanie w nim zmian bardziej ryzykowne, a rozwiązywanie problemów bardziej skomplikowane.

Infrastruktura dla bezserwerowej usługi FaaS

W rozdziale 3 wymieniłem usługę bezserwerową FaaS jako jeden ze sposobów udostępniania aplikacjom zasobów obliczeniowych przez platformę (patrz „Zasoby obliczeniowe” na stronie 26). Zwykły model dla kodu aplikacji zakłada jego ciągłe działanie w kontenerze lub na serwerze. FaaS wykonuje kod aplikacji na żądanie, w odpowiedzi na zdarzenie lub według harmonogramu.

¹⁹ <https://www.consul.io>

²⁰ <https://www.envoyproxy.io>

²¹ <https://istio.io>

²² <https://linkerd.io>

Kod typu FaaS jest przydatny dla dobrze zdefiniowanych, krótkotrwałych działań, które wymagają szybkiego uruchamiania. Typowymi przykładami są obsługa żądań HTTP oraz odpowiadanie na zdarzenia błędów w kolejce komunikatów. W razie potrzeby platforma uruchamia wiele instancji kodu równolegle, na przykład, aby obsłużyć wiele zdarzeń przychodzących jednocześnie.

Usługa FaaS może być bardzo wydajna w przypadku obciążeń, w ramach których zapotrzebowanie mocno się zmienia, skalując działania w górę podczas silnych wzrostów albo wyłączając je całkowicie, gdy nie są potrzebne.

Termin „bezserwerowa” nie jest najbardziej odpowiedni w przypadku tej usługi, ponieważ oczywiście kod jest wykonywany na serwerze. Chodzi po prostu o to, że serwer jest praktycznie niewidoczny dla programisty. To samo jest prawdą w przypadku kontenerów, więc tym, co wyróżnia tak zwaną „bezserwerowość” nie jest oderwanie od serwerów. Prawdziwa różnica polega na tym, że w przypadku bezserwerowości mamy do czynienia z krótkotrwałymi, a nie długotrwałymi procesami.

Z tego powodu wiele osób woli mówić o usłudze FaaS niż o bezserwerowości. To również odróżnia FaaS od innych zastosowań terminu „bezserwerowość”, który może również oznaczać BaaS (*Backend as a Service*), czyli zewnętrznie hostowaną usługę²³.

Środowiska wykonawcze FaaS stosują takie same modele, co klastry aplikacji – środowisko wykonawcze FaaS udostępniane jako usługa przez platformę infrastruktury pakietową usługę FaaS, która wymaga od użytkownika udostępnienia i skonfigurowania infrastruktury oraz narzędzi do zarządzania. Przykładami środowisk wykonawczych FaaS udostępnianych jako usługa są:

- AWS Lambda²⁴
- Azure Functions²⁵
- Google Cloud Functions²⁶

Przykładami pakietowych środowisk wykonawczych FaaS są:

- Fission²⁷
- Kubeless²⁸
- OpenFaaS²⁹
- Apache OpenWhisk³⁰

Do udostępniania infrastruktury dla pakietowych rozwiązań FaaS można używać strategii opisanych w tym rozdziale, takich jak pule serwerów i usługi zarządzania. Trzeba dobrze

²³ Więcej informacji jest w artykule Mike’a Robertsa „Serverless Architectures” (<https://oreil.ly/wHE-5>).

²⁴ <https://oreil.ly/wM1T2>

²⁵ <https://oreil.ly/Yp76V>

²⁶ <https://oreil.ly/HGViq>

²⁷ <https://fission.io>

²⁸ <https://kubeless.io>

²⁹ <https://www.openfaas.com>

³⁰ <https://openwhisk.apache.org>

rozumieć, jak działa rozwiązanie FaaS i zdawać sobie sprawę z możliwości „wyciekania” danych z kodu. Na przykład kod może pozostawiać pliki tymczasowe lub inne pozostałości w lokalizacjach, w których będą one dostępne dla innego kodu FaaS, co może powodować problemy z bezpieczeństwem i zgodnością. To, jak dobrze rozwiązanie FaaS spełnia nasze wymagania dotyczące segregacji danych i czy daje się skalować, powinno wpływać na decyzję, czy uruchamiać wiele instancji środowiska wykonawczego FaaS.

Usługi FaaS oferowane przez dostawców chmur nie dają zwykle aż takich możliwości konfiguracji, co klastry aplikacji. Na przykład z reguły nie musimy określać wielkości ani charakteru serwerów, na których jest wykonywany kod. To drastycznie ogranicza wielkość infrastruktury, którą trzeba zdefiniować i nią zarządzać.

Jednak w większości przypadków FaaS wchodzi w interakcję z innymi usługami i zasobami. Czasem wymaga definiowania sieci dla żądań przychodzących, wyzwających aplikację FaaS oraz dla żądań wychodzących, generowanych przez kod. Kod FaaS często wykonuje operacje zapisu i odczytu danych oraz komunikatów z użyciem urządzeń pamięci, baz danych i kolejek komunikatów. Wszystko to wymaga zdefiniowania i przetestowania zasobów infrastruktury. I oczywiście kod FaaS powinien być dostarczany i testowany przy użyciu potoku, jak w przypadku każdego innego kodu. Dlatego konieczna jest znajomość wszystkich praktyk dotyczących definiowania i promowania kodu infrastruktury oraz integrowania go z procesami testowania i aplikacji.

Podsumowanie

Zadaniem infrastruktury obliczeniowej jest wspomaganie usług. Usługi są zapewniane przez aplikacje, uruchamiane w środowisku wykonawczym. W tym rozdziale opisałem, jak można za pomocą kodu zdefiniować infrastrukturę udostępniającą środowiska wykonawcze dla aplikacji wykorzystywanych przez organizację.

CZĘŚĆ IV

Projektowanie infrastruktury

Podstawowa praktyka: małe, proste elementy

Udany system z upływem czasu zwykle się rozrasta. Coraz więcej osób z niego korzysta, coraz więcej osób nad nim pracuje, coraz więcej rzeczy jest do niego dodawanych. W miarę powiększania się systemu zmiany stają się bardziej ryzykowne i bardziej skomplikowane, a to z kolei prowadzi do wzrostu złożoności i czasochłonności zarządzających nimi procesów. Koszty dokonywania zmian sprawiają, że coraz trudniej jest naprawiać i ulepszać system, przez co rośnie dług techniczny i obniża się jakość systemu.

Jest to negatywna wersja cyklu, w którym szybkość wpływa na poprawę jakości, a lepsza jakość umożliwia szybsze dostarczanie zmian, co zostało opisane w rozdziale 1.

Stosowanie trzech podstawowych praktyk infrastruktury jako kodu – definiowanie wszystkiego jako kodu, ciągłe testowanie i dostarczanie oraz budowanie małych elementów („Trzy podstawowe praktyki dotyczące infrastruktury jako kodu” na stronie 11) – umożliwia pozytywną wersję cyklu.

W tym rozdziale koncentruję się na trzeciej praktyce – tworzeniu systemu z małych elementów w celu zapewnienia szybszego tempa zmian przy jednoczesnej poprawie jakości, pomimo rozrastania się systemu. Większość narzędzi i języków do kodowania infrastruktury ma funkcje do obsługi modułów, bibliotek i innego typu komponentów. Jednak teoria i praktyka projektowania infrastruktury nie osiągnęły jeszcze takiego stopnia dojrzałości, jak projektowanie oprogramowania.

Dlatego w tym rozdziale będziemy wykorzystywać zasady projektowania modularnego wypracowane przez dekady projektowania oprogramowania, analizując je z punktu widzenia infrastruktury sterowanej kodem. Następnie przyjrzymy się różnym typom komponentów systemu infrastruktury, zwracając uwagę na to, jak mogą być używane do zapewnienia lepszej modularności. Na koniec poznamy bazujące na tym różne podejścia do wyznaczania granic między komponentami infrastruktury.

Projektowanie pod kątem modularności

Celem modularności jest umożliwienie szybszego i bezpieczniejszego dokonywania zmian w systemie. Modularność pozwala wspierać to na kilka sposobów. Jednym z nich jest usunięcie powielania implementacji, co redukuje liczbę zmian kodu potrzebnych do wprowadzenia konkretnej modyfikacji. Drugim sposobem jest uproszczenie implementacji poprzez dostarczenie komponentów, które można zestawiać na różne sposoby do różnych zastosowań.

Trzecim sposobem jest takie projektowanie systemu, aby można było zmieniać małe komponenty bez konieczności modyfikowania pozostałych części systemu. Zmienianie małych elementów jest łatwiejsze, bezpieczniejsze i szybsze niż dużych.

Większość zasad projektowania pod kątem modularności ma pewną wadę. Ich nieuważne stosowanie może w rzeczywistości uczynić system bardziej kruchym i trudniejszym do zmiany. Cztery kluczowe wskaźniki z podrozdziału „Cztery kluczowe wskaźniki” na stronie 10 są przydatnym narzędziem do oceny skuteczności modularyzacji systemu.

Cechy dobrze zaprojektowanych komponentów

Projektowanie komponentów to sztuka decydowania o tym, które elementy systemu należy grupować, a które rozdzielać. Prawidłowa decyzja wymaga rozumienia relacji i zależności między elementami. Dwoma ważnymi cechami projektowymi komponentu są sprzężenie i spójność¹. Celem dobrego projektu jest osiągnięcie słabego sprzężenia i silnej spójności.

Sprzężenie (coupling) opisuje, jak często zmiana jednego komponentu wymaga zmiany innego komponentu. Brak sprzężenia nie jest realistycznym celem dla dwóch części systemu. Takie sprzężenie oznacza prawdopodobnie, że są to części należące do różnych systemów. Zamiast tego należy dążyć, aby sprzężenie było słabe, czyli luźne. Obraz serwera i stos są ze sobą sprzężone, ponieważ uaktualnienie oprogramowania na serwerze może wymagać zwiększenia pamięci alokowanej dla instancji serwera w stosie. Jednak uaktualnienie obrazu serwera nie powinno za każdym razem pociągać za sobą konieczności zmiany kodu w stosie. Słabe sprzężenie ułatwia dokonywanie zmian w komponencie przy niewielkim ryzyku popsucia pozostałych części systemu.

Spójność (cohesion) opisuje relację między elementami wewnątrz komponentu. Podobnie jak w przypadku sprzężenia, koncepcja spójności odnosi się do wzorca zmian. Zmiany zdefiniowanego w stosie zasobu o słabej spójności często nie dotyczą innych zasobów w stosie. Stos infrastruktury, który definiuje oddzielne struktury sieciowe dla serwerów wyposażanych przez dwa inne stosy, ma słabą spójność. Komponenty o silnej spójności są łatwiejsze do zmiany, ponieważ są mniejsze, prostsze i czystsze oraz mają mniejszy promień wybuchu (Blast Radius) niż komponenty zawierające siatkę luźno powiązanych elementów.

¹ <https://oreil.ly/Qe3Sh>



Cztery zasady prostego projektu

Kent Beck, twórca XP i TDD, często cytuje cztery zasady² tworzenia prostego projektu komponentu. Zgodnie z nimi prosty kod powinien:

- Przechodzić pomyślnie testy (robić to, co powinien)
- Ujawniać swoje zamiary (być przejrzysty i łatwy do zrozumienia)
- Nie zawierać duplikatów
- Zawierać jak najmniej elementów

Zasady projektowania komponentów

W dziedzinie architektury i projektowania oprogramowania istnieje wiele zasad i wskazówek dotyczących projektowania komponentów o słabym sprzężeniu i silnej spójności.

Unikanie powielania

Zasada DRY (Don't Repeat Yourself), czyli nie powtarzaj się, mówi „Każda porcja wiedzy musi mieć pojedynczą, jednoznaczną, autorytatywną reprezentację w systemie”³. Powielanie zmusza ludzi do wprowadzania zmian w wielu miejscach.

Na przykład wszystkie stopy w ShopSpinner używają konta użytkownika `provisioner` do stosowania konfiguracji do instancji serwerów. Początkowo szczegóły logowania dla tego konta były określone w każdym stosie, podobnie jak kod, który buduje podstawowy obraz serwera. Kiedy ktoś potrzebował zmienić dane logowania dla tego konta, musiał je odnaleźć i zmienić w każdej z tych lokalizacji w bazie kodu. Dlatego zespół postanowił przenieść dane logowania do centralnej lokalizacji i w efekcie każdy stos oraz konstruktor obrazu serwera odwołuje się teraz do tej lokalizacji.



Kiedy powielanie może być przydatne

Zasada DRY zniechęca do powielania implementacji koncepcji, co nie jest tym samym, co dokładne powielanie linii kodu. Używanie wielu komponentów zależnych od współdzielonego kodu może powodować silne sprzężenie, utrudniające dokonywanie zmian.

Widziałem zespoły naciskające na centralizację wszystkich wyglądających podobnie fragmentów kodu. Efektem było na przykład tworzenie wszystkich serwerów wirtualnych przy użyciu jednego modułu. W praktyce serwery tworzone do różnych celów, takie jak serwery aplikacji, serwery WWW i serwery kompilacji, muszą być zazwyczaj definiowane inaczej. Moduł, który ma tworzyć wszystkie typy serwerów, może okazać się zbyt skomplikowany.

Zastanawiając się, czy kod jest powielony i powinien zostać scentralizowany, należy zwrócić uwagę, czy naprawdę reprezentuje on tę samą koncepcję. Czy zmiana jednego wystąpienia tego kodu oznacza zawsze, że inna instancja również musi się zmienić?

² <https://oreil.ly/gelGI>

³ Zasada DRY i inne są opisane w *The Pragmatic Programmer: From Journeyman to Master* Andrew Hunta i Davida Thomasa (Addison-Wesley).

Pojawia się też pytanie, czy dobrym pomysłem jest wspólne zablokowanie obu instancji kodu z tym samym cyklem zmian. Wymuszanie aktualizacji każdego serwera aplikacji w tym samym momencie może być nierealistyczne.

Ponowne użycie kodu zwiększa sprzężenie. Dlatego dobra zasada praktyczna dotycząca wielokrotnego używania kodu to stosowanie go wewnątrz komponentu, ale nie między komponentami.

Zasada kompozycji

Aby utworzyć system komponowalny, jego części składowe muszą być niezależne. Powinno dać się łatwo zastąpić jedną część relacji zależności bez naruszania drugiej⁴.

Zespół ShopSpinner zaczyna od jednego obrazu linuksowego serwera aplikacji, udostępnianego przez różne stosy. Później dodaje obraz windowsowego serwera aplikacji. Zespół projektuje kod udostępniający obraz serwera z dowolnego stosu, dzięki czemu może łatwo przełączać się między tymi dwoma obrazami serwera, w zależności od potrzeb danej aplikacji.

Zasada jednej odpowiedzialności

Zasada pojedynczej odpowiedzialności (single responsibility principle – SRP) mówi, że każdy komponent powinien odpowiadać za jedną rzecz. Chodzi o to, aby każdy komponent był na czymś skupiony i w efekcie jego zawartość była spójna⁵.

Komponent infrastruktury, bez względu na to, czy jest to serwer, biblioteka konfiguracji, komponent stosu czy stos, powinien być zorganizowany wokół jednego celu. Ten cel może mieć charakter warstwowy. Udostępnianie infrastruktury dla jakiejś aplikacji jest pojedynczym celem, który można zrealizować za pomocą stosu infrastruktury. Można podzielić ten cel na: bezpieczny routing ruchu dla aplikacji, zaimplementowany w bibliotece stosu; serwer aplikacji, zaimplementowany za pomocą obrazu serwera; oraz instancję bazy danych, zaimplementowaną w postaci modułu stosu. Każdy komponent, na każdym poziomie, ma jeden, łatwo zrozumiały cel.

Projektowanie komponentów opartych na pojęciach dziedzinowych, a nie technicznych

Budowanie komponentów opartych na pojęciach technicznych jest często bardzo kuszące. Na przykład może wydawać się dobrym pomysłem utworzenie komponentu definiującego serwer i wykorzystywanie go we wszystkich stosach potrzebujących serwera. W praktyce współdzielony komponent jest sprzężony z każdym kodem, który go używa.

4 Zasada kompozycji jest jedną z podstawowych zasad filozofii Uniksa (<https://oreil.ly/2szfb>).

5 Mój kolega James Lewis stosuje SRP do odpowiedzi na pytanie, jak duża powinna być mikrousluga (<https://oreil.ly/tPf8f>). Jego rada jest taka, że komponent powinien być do ogarnięcia przez jedną osobę, przynajmniej koncepcyjnie, co odnosi się zarówno do komponentów infrastruktury, jak i oprogramowania.

Lepszym podejściem jest tworzenie komponentów opartych na pojęciach dziedzinowych. Serwer aplikacji jest pojęciem dziedzinowym i może być używany w wielu aplikacjach. Serwer kompilacji to również pojęcie dziedzinowe i w praktyce może być przydatne tworzenie jego oddzielnych instancji dla poszczególnych zespołów. Takie komponenty są lepsze niż zwykłe serwery, używane najczęściej w różny sposób w różnych miejscach.

Prawo Demeter

Prawo Demeter, zwane również *zasadą minimalnej wiedzy*⁶, mówi, że komponent nie powinien mieć wiedzy o sposobach implementacji innych komponentów. Taka reguła wymusza stosowanie jasnych i prostych interfejsów między komponentami.

Zespół ShopSpinner początkowo naruszał tę zasadę, wykorzystując stos definiujący klaster serwerów aplikacji oraz współdzielony stos sieci definiujący moduł równoważenia obciążenia i reguły zapory dla tego klastra. Współdzielony stos sieci miał zbyt szczegółową wiedzę o stosie serwerów aplikacji.



Dostawcy i konsumenci

W relacji zależności między komponentami *dostawca* tworzy lub definiuje zasób wykorzystywany przez *konsumenta*.

Współdzielony stos sieci może być dostawcą, tworzącym bloki adresów sieci, takie jak podsieci. Stos infrastruktury aplikacji może być konsumentem współdzielonego stosu sieci, udostępniającym serwery i moduły równoważenia obciążenia w ramach podsieci zarządzanych przez dostawcę.

Głównym tematem tego rozdziału jest definiowanie i implementowanie interfejsów między komponentami infrastruktury.

Brak zależności cyklicznych

Relacje, w których występuje komponent udostępniający zasoby konsumentom, nie powinny nigdy zawierać pętli (czyli cyklu). Innymi słowy, komponent będący dostawcą nie powinien nigdy używać zasobów pochodzących od kogoś ze swoich bezpośrednich lub pośrednich konsumentów.

Przykład ShopSpinner ze współdzielonym stosem sieci zawiera zależność cykliczną. Stos serwerów aplikacji przypisuje serwery w klastrze do struktur sieciowych we współdzielonym stosie sieci. Współdzielony stos sieci tworzy moduł równoważenia obciążenia i reguły zapory dla określonych klastrów serwerów w stosie serwerów aplikacji.

Zespół ShopSpinner może rozwiązać problem zależności cyklicznych i ograniczyć wiedzę stosu sieci o pozostałych komponentach, przenosząc elementy sieci specyficzne dla stosu serwerów aplikacji do tego stosu. Spowoduje to również poprawę spójności i sprzężenia, ponieważ stos sieci nie będzie już zawierał elementów, które są związane głównie z elementami innego stosu.

⁶ <https://oreil.ly/6x0aa>

Używanie testowania do podejmowania decyzji projektowych

W rozdziale 8 i 9 opisałem zasady ciągłego testowania kodu infrastruktury podczas pracy nad zmianami. Tak silny nacisk na testowanie sprawia, że testowalność staje się istotnym zagadnieniem przy projektowaniu komponentów infrastruktury.

System dostarczania zmian musi umożliwiać tworzenie i testowanie kodu infrastruktury na każdym poziomie, od modułu konfiguracji serwera instalującego agenta monitorowania, aż po stos kodu budujący klaster kontenerów. Etapy potoków muszą pozwalać szybko tworzyć instancje poszczególnych komponentów w izolacji. Taki poziom testowania nie jest możliwy w przypadku bazy kodu spaghetti z poplątanymi zależnościami albo dużych komponentów, których udostępnienie zajmuje pół godziny.

Powyższe wyzwania utrudniają często wprowadzenie efektywnego, zautomatyzowanego systemu testowania kodu infrastruktury. Ciężko jest pisać i uruchamiać zautomatyzowane testy dla marnie zaprojektowanego systemu. I tu ujawnia się ukryta korzyść z automatycznego testowania: zmusza do lepszego projektowania. Jedynym sposobem na ciągłe testowanie i dostarczanie kodu jest zaimplementowanie i utrzymywanie jasnych projektów systemów – słabo sprzężonych i silnie spójnych.

Łatwiej jest implementować zautomatyzowane testowanie luźno sprzężonych modułów konfiguracji serwerów. Łatwiej jest budować i wykorzystywać atrapy modułów z jasnymi, prostymi interfejsami (patrz „Używanie warunków początkowych testu do obsługi zależności” na stronie 132). Mały, dobrze zdefiniowany stos można szybciej udostępnić i testować w potoku.

Modularyzacja infrastruktury

System infrastruktury obejmuje różnego rodzaju komponenty, jak napisałem w rozdziale 3, a każdy z nich może być zbudowany z różnych części. Instancja serwera może być utworzona z obrazu, przy użyciu roli konfiguracji serwera, odwołującej się do zbioru modułów konfiguracji serwera, które z kolei mogą importować biblioteki kodu. Stos infrastruktury może składać się z instancji serwerów i wykorzystywać moduły lub biblioteki kodu stosu. Do tego stosy można łączyć, tworząc większe środowiska.

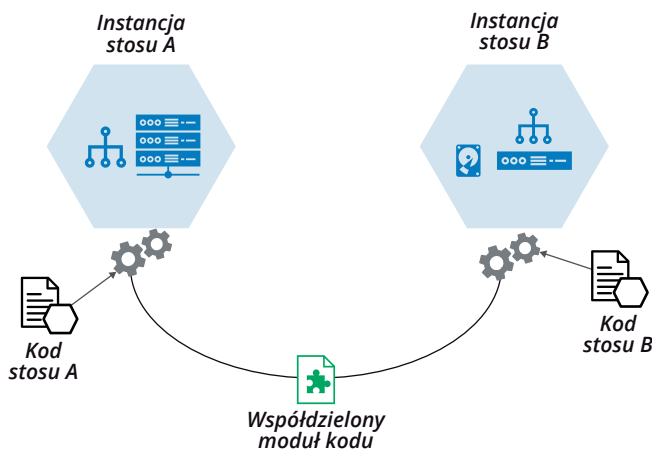
Komponenty stosów a stosy jako komponenty

Stos infrastruktury, zgodnie z definicją z rozdziału 5, jest podstawową jednostką wdrożeniową infrastruktury. Stos jest przykładem *architektonicznego kwantu*, który Ford, Parsons i Kua definiują jako „niezależnie wdrażany komponent o silnej spójności funkcjonalnej, zawierający wszystkie elementy strukturalne wymagane przez system do prawidłowego działania”⁷. Innymi słowy, stos jest komponentem, który można niezależnie wypychać do produkcji.

7 Patrz rozdział 4 w *Building Evolutionary Architectures*, autorzy Neal Ford, Rebecca Parsons i Patrick Kua (O'Reilly).

Jak już wspomniałem wcześniej, stos może zawierać komponenty i sam może być komponentem. Jednymi z potencjalnych komponentów stosu mogą być serwery, na tyle ważne, że zajmiemy się nimi dokładniej w dalszej części tego rozdziału. Większość narzędzi do zarządzania stosem obsługuje również wstawianie kodu stosu do modułów albo wykorzystywanie bibliotek do generowania elementów stosu.

Na rysunku 15-1 widać dwa stosy, nazwane StackA i StackB, które wykorzystują wspólny, dzielony moduł kodu definiujący strukturę sieci.



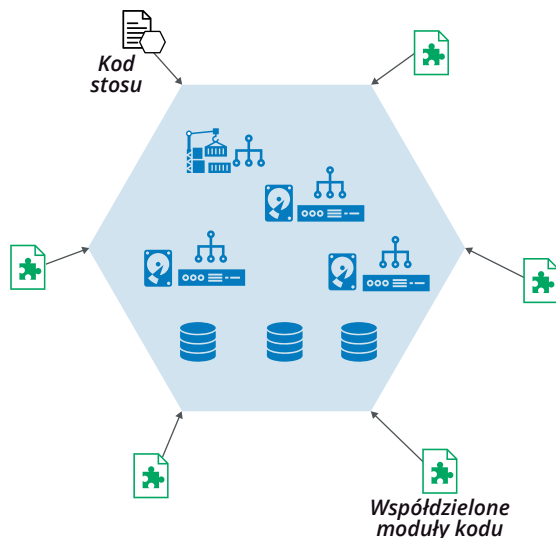
Rysunek 15-1 Współdzielony moduł kodu wykorzystywany przez dwa stosy

W rozdziale 16 są opisane niektóre wzorce i antywzorce używania bibliotek i modułów kodu stosu. Biblioteki i moduły stosów przydają się do wielokrotnego używania kodu. Są natomiast mniej pomocne, jeśli chodzi o ułatwianie dokonywania zmian w stosach. Widziałem zespoły usiłujące poprawić stos monolityczny (patrz „Antywzorec: stos monolityczny” na stronie 52), rozbijając kod na moduły. Chociaż dzięki modułom łatwiej było śledzić kod, to każda instancja stosu pozostawała nadal tak samo duża i złożona, jak wcześniej.

Na rysunku 15-2 widać, jak kod podzielony na oddzielne moduły jest łączony w instancji stosu.

Moduł używany przez inne stosy nie tylko wymaga dodawania go do elementów w ramach każdej instancji stosu, ale tworzy także sprzężenie między tymi stosami. Zmiana modułu w celu spełnienia wymagań jednego stosu może wpłynąć na pozostałe stosy wykorzystujące ten moduł. Takie sprzężenie może utrudniać dokonywanie zmian.

Lepszym podejściem do ułatwiania zarządzania dużym stosem jest podzielenie go na wiele stosów, z których każdy może być udostępniany, zarządzany i zmieniany niezależnie od pozostałych. W podrozdziale „Wzorce i antywzorce konstruowania stosów” na stronie 52 jest wymienionych kilka wzorców ustalania wielkości i zawartości stosu. Bardziej szczegółowe informacje o zarządzaniu zależnościami między stosami są podane w rozdziale 17.

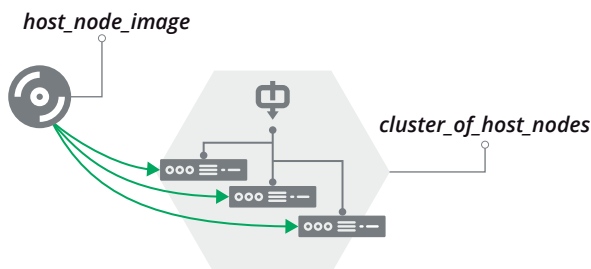


Rysunek 15-2 Moduły stosu zwiększają złożoność instancji stosu

Używanie serwera w stosie

Serwery są typowym komponentem stosu. W rozdziale 11 opisałem różne komponenty samego serwera i jego cykl życia. Kod stosu zazwyczaj łączy serwery jako kombinację obrazów serwera (patrz rozdział 13) oraz modułów konfiguracji serwera (patrz „Kod konfiguracji serwera” na stronie 166), często za pomocą wskazania roli (patrz „Role serwerów” na stronie 169).

Baza kodu zespołu ShopSpinner zawiera przykład użycia obrazu serwera jako komponentu stosu. Chodzi o stos noszący nazwę `cluster_of_host_nodes`, który buduje klastery serwerów, pełniących rolę węzłów hostujących kontenery, jak to jest pokazane na rysunku 15-3.



Rysunek 15-3 Obraz serwera jako dostawca dla stosu

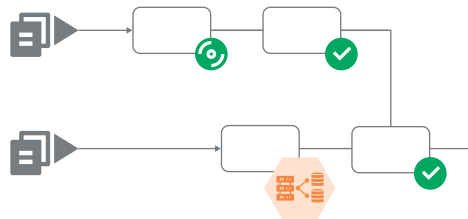
Kod definiujący klastery serwerów określa nazwę obrazu serwera `host_node_image`:

```

server_cluster:
  name: "cluster_of_host_nodes"
  min_size: 1
  max_size: 3
  each_server_node:
    source_image: host_node_image
    memory: 8GB

```

Zespół wykorzystuje potok do budowy i testowania zmian w obrazie serwera. Inny potok testuje zmiany w `cluster_of_host_nodes`, integrując ten klaster z najnowszą wersją `host_node_image`, która pomyślnie przeszła swoje testy (patrz rysunek 15-4).



Potok dla stosu `cluster_of_host_nodes`

Rysunek 15-4 Potok do integracji obrazu serwera i stosu będącego jego konsumentem

W podrozdziale „Potoki dostarczania infrastruktury” na stronie 113 jest wyjaśnione, jak wygląda działanie potoków dla infrastruktury.

Ten przykład kryje jednak drobny problem. Pierwszy etap potoku dla stosu `cluster_of_host_nodes` nie używa obrazu `host_node_image`. Natomiast przykład kodu stosu zawiera nazwę tego obrazu, więc nie może być uruchamiany jako etap testowania online („Etap testowania online dla stosów” na stronie 128). Testowanie kodu stosu bez obrazu może być przydatne, ponieważ zespół może wykryć problemy z kodem stosu bez konieczności wyposażania pełnych serwerów w węzłach hostujących, co wymaga czasu.

Zespół ShopSpinner radzi sobie z tym problemem, zastępując w kodzie stosu zakodowany na stałe obraz `host_node_image` parametrem (rozdział 7). Taki kod jest wygodniejszy do testowania:

```

server_cluster:
  name: "cluster_of_host_nodes"
  min_size: 1
  max_size: 3
  each_server_node:
    source_image: ${HOST_NODE_SERVER_IMAGE}
    memory: 8GB

```

Etap testowania online dla stosu `cluster_of_host_nodes` może ustawiać parametr `HOST_NODE_SERVER_IMAGE` na ID uproszczonego obrazu serwera. Zespół może uruchamiać testy na tym etapie w celu sprawdzenia, czy klaster serwerów działa poprawnie, skalując

go w górę i w dół oraz odzyskując popsute instancje. Okrojony obraz serwera jest przykładem atrapy (patrz „Używanie warunków początkowych testu do obsługi zależności” na stronie 132).

Prosta zmiana polegająca na zastąpieniu parametrem zakodowanego na stałe odwołania do obrazu serwera ogranicza sprzężenie. Ponadto jest zgodna z zasadą kompozycji (patrz „Zasada kompozycji” na stronie 248). Zespół może łatwo tworzyć instancje `cluster_of_host_nodes` używając innego obrazu serwera, co może się przydać, gdy ktoś z zespołu zechce testować i stopniowo wdrażać inny system operacyjny w swoich klastrach.

Infrastruktura bez współdzielenia

W przypadku obliczeń rozproszonych *architektura bez współdzielenia* (<https://oreil.ly/4MFP8>) umożliwia skalowanie, w którym dodanie nowego węzła do systemu nie powoduje wystąpienia rywalizacji o jakiekolwiek zasoby poza tym węzłem.

Typowym kontrprzykładem jest architektura systemu, w której procesory współdzielą jeden dysk. Rywalizacja o ten dysk ogranicza skalowalność w postaci zwiększania liczby procesorów. Usunięcie współdzielonego dysku z projektu spowoduje, że skalowanie systemu na skutek dodawania procesorów stanie się bliższe liniowemu.

Projekt „bez współdzielenia” w przypadku kodu infrastruktury przenosi zasoby ze współdzielonego stosu do każdego stosu, który ich potrzebuje, usuwając relację dostawca-konsument. Na przykład zespół ShopSpinner może połączyć stos infrastruktury aplikacji i współdzielony stos sieci w jeden stos.

Każda instancja infrastruktury aplikacji ma swój własny zbiór struktur sieciowych. Powoduje to powielenie struktur sieciowych, ale zachowuje każdą instancję aplikacji niezależną od innych. Podobnie jak w przypadku architektury systemu rozproszonego, efektem jest usunięcie ograniczeń skalowania. Na przykład zespół ShopSpinner może dodać tyle instancji infrastruktury aplikacji, ile potrzebuje, bez używania przestrzeni adresów alokowanej za pomocą jednego współdzielonego stosu sieci.

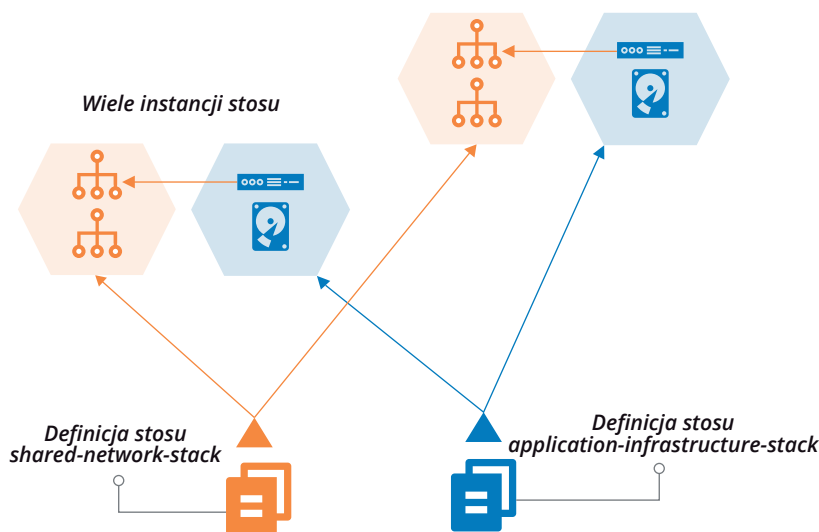
Jednak bardziej typowym powodem używania kodu infrastruktury bez współdzielenia jest ułatwienie modyfikowania, odbudowywania i odzyskiwania zasobów sieciowych dla stosu aplikacji. Projekt współdzielonego stosu sieci zwiększa promień wybuchu i koszty zarządzania towarzyszące pracy z siecią.

Infrastruktura bez współdzielenia obsługuje również model zabezpieczeń zerowego zaufania (patrz „Model bezpieczeństwa zerowego zaufania z SDN” na stronie 30), ponieważ każdy stos można zabezpieczyć oddzielnie.

Projekt „bez współdzielenia” nie wymaga umieszczania wszystkiego w jednej instancji stosu. W przypadku zespołu ShopSpinner alternatywą dla łączenia infrastruktury sieci i aplikacji w jednym stosie jest zdefiniowanie infrastruktury sieci

i aplikacji w różnych stosach, jak wcześniej, za to utworzenie oddzielnej instancji stosu sieci dla każdej instancji stosu aplikacji (patrz rysunek 15-5).

Przy takim podejściu instancje stosu aplikacji nie współdzielą sieci z innymi stosami i obie części mogą być zarządzane niezależnie. Wadą tego rozwiązania jest to, że wszystkie instancje stosu sieci pozostają nadal zdefiniowane przez ten sam kod, więc dowolna zmiana tego kodu wymaga dodatkowej pracy w celu upewnienia się, czy każda instancja działa poprawnie.



Rysunek 15-5 Wiele stosów z modelem wdrażania bez współdzielenia

Wytyczanie granic między komponentami

Aby podzielić infrastrukturę, jak w przypadku każdego systemu, należy poszukać szwów. Szew (*seam*) jest miejscem, w którym można zmienić zachowanie się systemu bez stosowania edycji⁸. Pomyśl polega na szukaniu naturalnych miejsc do wytyczenia granic między częściami systemu, w których można utworzyć proste, przejrzyste punkty integracji.

Każda z poniższych strategii grupuje elementy infrastruktury na podstawie określonych aspektów: wzorców zmian, struktur organizacyjnych, bezpieczeństwa i nadzoru oraz odporności i skalowania. Strategie te, jak większość zasad i praktyk architektonicznych, sprowadzają się do optymalizacji pod kątem zmian. Jest to poszukiwanie metody projektowania komponentów umożliwiającej łatwiejsze, bezpieczniejsze i szybsze dokonywanie zmian w systemie.

⁸ Michael Feathers wprowadził termin „szew” w swojej książce *Working Effectively with Legacy Code* (Addison-Wesley). Więcej informacji jest także dostępnych online (<https://oreil.ly/Y4EwT>).

Dopasowywanie granic do naturalnych wzorców zmian

Najbardziej podstawowe podejście do optymalizacji granic komponentów polega na zrozumieniu ich naturalnych wzorców zmian. To jest idea kryjąca się za szukaniem szwów – szew jest naturalną granicą.

W przypadku istniejącego systemu można dowiedzieć się, które rzeczy zmieniają się zwykle razem badając historię zmian. Najlepszym źródłem są bardziej drobiazgowo zmiany, takie jak zatwierdzenia kodu. Najbardziej skuteczne zespoły stosują optymalizację pod kątem częstych zmian, w pełni integrując i testując każdą z nich. Dowiadując się, które komponenty zmieniają się razem w ramach jednego zatwierdzenia albo luźno związanych zatwierdzeń w wielu komponentach, można znaleźć wzorce sugerujące sposób refaktoryzacji kodu w celu uzyskania silniejszej spójności i słabszego sprzężenia.

Badanie elementów na wyższych poziomach pracy, takich jak bilety, scenariusze czy projekty, może pomóc zrozumieć, które części systemu biorą często udział w zmianach. Ale optymalizacja powinna być wykonywana pod kątem małych, częstych zmian. Dlatego należy schodzić na niższy poziom i sprawdzać, które zmiany mogą być dokonywane niezależnie od innych, aby umożliwić zmiany przyrostowe w przypadku inicjatyw obejmujących większe modyfikacje.

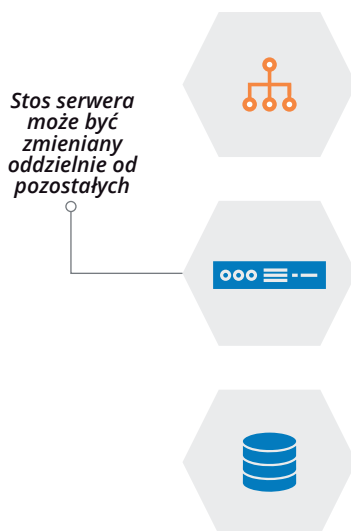
Dopasowywanie granic do cykli życia komponentów

Różne części infrastruktury mogą mieć różne cykle życia. Na przykład serwery w klastrze (patrz „Zasoby obliczeniowe” na stronie 26) są tworzone i niszczone dynamicznie, może nawet wiele razy dziennie. Wolumin magazynu bazy danych zmienia się rzadziej.

Organizowanie zasobów infrastruktury w nadające się do wdrażania komponenty, zwłaszcza stosy infrastruktury, zgodnie z ich cyklem życia może uprościć zarządzanie. Rozważmy stos infrastruktury serwera aplikacji ShopSpinner składający się z tras sieciowych, klastra serwerów i instancji bazy danych.

Serwery w tym stosie są aktualizowane co najmniej raz w tygodniu, a do ich odbudowy są używane nowe obrazy serwera z ostatnimi poprawkami systemu operacyjnego (jak była mowa w rozdziale 13). Urządzenie magazynujące bazy danych rzadko się zmienia, chociaż mogą powstawać nowe instancje w celu odzyskania lub replikacji instancji aplikacji. Zespół zmienia czasem sieć w innych stosach, co wymaga aktualizacji routingu specyficznego dla aplikacji w tym stosie.

Definiowanie tych elementów w jednym stosie niesie ze sobą pewne ryzyko. Aktualizacja obrazu serwera aplikacji może się nie powieść. Usunięcie problemu może oznaczać konieczność przebudowy całego stosu, w tym urządzenia magazynującego bazy danych, co z kolei wymaga utworzenia kopii zapasowej danych w celu przywrócenia ich w nowej instancji (patrz „Ciągłość danych w zmieniającym się systemie” na stronie 373). Chociaż można zarządzać tym w ramach jednego stosu, byłoby prościej, gdyby urządzenie magazynujące bazy danych było zdefiniowane w oddzielnym stosie, jak to jest pokazane na rysunku 15-6.



Rysunek 15-6 Różne stosy mają różne cykle życia

Zmiany są dokonywane w tych mikrostosach (patrz „Wzorzec: mikrostos” na stronie 57) bez wpływania bezpośredniego na pozostałe. Takie podejście umożliwia włączenie zdarzeń zarządzania specyficznych dla stosu. Na przykład jakakolwiek zmiana w stosie urządzenia magazynującego bazy danych może wyzwolić archiwizację danych, co prawdopodobnie byłoby zbyt kosztowne w przypadku wyzwalania przez każdą zmianę innych elementów pierwszego, połączonego stosu.

Optymalizacja granic stosów pod kątem cykli życia jest szczególnie przydatna w przypadku zautomatyzowanego testowania w potokach. Etapy potoków są często uruchamiane wiele razy dziennie, gdy ludzie pracują nad zmianami infrastruktury, dlatego muszą być zoptymalizowane, aby dostarczać szybko informacje zwrotne i utrzymywać dobre tempo pracy. Organizowanie elementów infrastruktury w oddzielne stosy na podstawie ich cyklu życia może skrócić czas potrzebny do wprowadzania zmian w testowaniu.

Na przykład podczas pracy nad kodem infrastruktury dla serwerów aplikacji niektóre etapy potoku mogą za każdym razem odbudowywać stos (patrz „Wzorzec: efemeryczny stos testowy” na stronie 137). Odbudowywanie struktur sieciowych lub dużych urządzeń magazynujących dane może być powolne i nie musi być konieczne dla wielu zmian podczas pracy. W takim przypadku pokazany wcześniej projekt mikrostosu (rysunek 15-6) może usprawnić proces testowania i dostarczania.

Trzecim przypadkiem użycia stosów rozdzielonych na podstawie cyklu życia jest zarządzanie kosztami. Wyłączanie lub niszczenie i odbudowywanie infrastruktury, która nie jest potrzebna w okresach spokoju, to powszechny sposób zarządzania kosztami w chmurach publicznych. Ale odbudowa niektórych elementów, na przykład magazynu danych, może nie być taka łatwa. Takie elementy można wydzielić do oddzielnych stosów i pozostawiać je działające w momencie, gdy inne stosy będą niszczone w celu redukcji kosztów.

Dopasowywanie granic do struktur organizacyjnych

Prawo Conway'a⁹ mówi, że systemy odzwierciedlają zwykle strukturę organizacji, która je tworzy¹⁰. Zespołowi jest zwykle łatwiej integrować oprogramowanie i infrastrukturę, które w całości należą do niego i w naturalny sposób będzie tworzył dla nich solidniejsze granice z częściami systemu należącymi do innych zespołów.

Prawo Conway'a ma dwie zasadnicze konsekwencje dla projektowania systemów obejmujących infrastrukturę. Jedną z nich jest unikanie projektowania komponentów, które muszą być zmieniane przez wiele zespołów. Drugą jest rozważenie organizacji zespołów w sposób odzwierciedlający pożądane granice architektoniczne, zgodnie z „Odwrotnym manewrem Conwaya”¹¹.



Starsze silosy tworzą infrastrukturę rozłączną

Starsze struktury organizacyjne, które powierzają budowanie i uruchamianie oddzielnym zespołom tworzą często niespójną infrastrukturę po drodze do produkcji, zwiększając czas, koszt i ryzyko dostarczania zmian. Współpracowałem kiedyś z organizacją, w której zespół programowania aplikacji i zespół systemów operacyjnych używały różnych narzędzi konfiguracyjnych do budowania serwerów. Przy każdym wydaniu oprogramowania marnowali kilka tygodni na wdrożenie aplikacji i uruchomienie ich w środowiskach produkcyjnych.

W przypadku infrastruktury, w szczególności, warto zastanowić się, jak dopasować projekt do struktury zespołów używających tej infrastruktury. W większości organizacji są to linie produktów lub usług i aplikacji. Nawet jeśli jest to infrastruktura używana przez wiele zespołów, jak usługa DBaaS (patrz „Zasoby pamięci masowej” na stronie 27), można tak zaprojektować infrastrukturę, aby zarządzać oddzielnymi instancjami dla każdego zespołu.

Dopasowywanie instancji infrastruktury do wykorzystujących je zespołów sprawia, że zmiany są mniej uciążliwe. Zamiast negocjować jedno okno zmian z wszystkimi zespołami używającymi współdzielonej instancji, można negocjować oddzielne okna dla każdego zespołu.

Wytyczanie granic wspierających odporność

Gdy coś źle działa w systemie, można odbudować niezależnie wdrażany komponent, taki jak stos infrastruktury. Elementy wewnątrz stosu można naprawić lub odbudować ręcznie, przeprowadzając *chirurgię infrastruktury*. Operowanie infrastrukturą wymaga ostrożnej interwencji ze strony kogoś, kto ją dobrze zna. Prosty błąd może znacznie bardziej pogorszyć sytuację.

⁹ <https://oreil.ly/Sgb2w>

¹⁰ Pełna definicja ma postać „Każda organizacja, która projektuje jakiś system (szeroko rozumiany) stworzy projekt o strukturze będącej kopią struktury komunikacyjnej tej organizacji”.

¹¹ https://oreil.ly/_dI92

Niektóre osoby są dumne z wykonywania operacji na infrastrukturze, ale jest to metoda awaryjna, kompensująca braki w systemie zarządzania infrastrukturą.

Alternatywą dla operowania infrastrukturą jest odbudowa komponentów przy użyciu dobrze zdefiniowanych procesów i narzędzi. Powinno być możliwe odbudowanie dowolnej instancji stosu poprzez wyzwolenie tego samego zautomatyzowanego procesu, który jest używany do stosowania zmian i aktualizacji. Jeśli tak jest, to nie trzeba budzić najbardziej błyskotliwego chirurga od infrastruktury, gdy coś się popsuje w środku nocy. W wielu przypadkach można automatycznie wyzwolić odzyskiwanie.

Komponenty infrastruktury należy tak projektować, aby można je było szybko odbudować i odzyskać. Jeśli zasoby są organizowane w komponenty na podstawie ich cyklu życia („Dopasowywanie granic do cyklów życia komponentów” na stronie 256), to można również wziąć pod uwagę przypadki użycia odbudowy i odzyskiwania.

Tak jest w przypadku wcześniejszego przykładu podziału infrastruktury zawierającej trwałe dane (rysunek 15-6). Proces odbudowy magazynu danych obejmuje działania automatycznie zapisujące i ładujące dane, co jest przydatne w scenariuszu odzyskiwania po awarii. Więcej na ten temat znajdziemy w podrozdziale „Ciągłość danych w zmieniającym się systemie” na stronie 373.

Podział infrastruktury na komponenty na podstawie procesu ich odbudowy pomaga uprościć i zoptymalizować odzyskiwanie. Innym podejściem do zagadnienia odporności jest uruchamianie wielu instancji części infrastruktury. Strategie redundancji mogą również pomagać w skalowaniu.

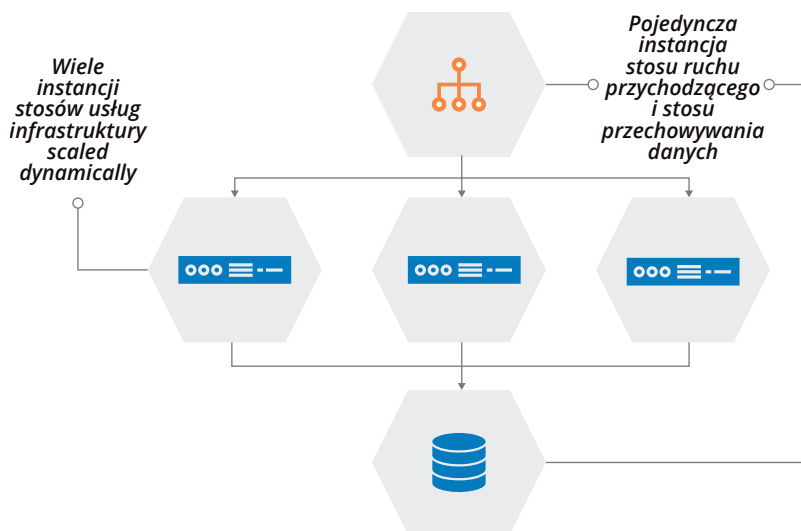
Tworzenie granic wspierających skalowanie

Typową strategią skalowania systemów jest tworzenie dodatkowych instancji niektórych ich komponentów. Takie instancje można dodawać w okresach zwiększonego zapotrzebowania. Można też rozważać wdrażanie ich w różnych regionach geograficznych.

Większość platform chmurowych potrafi automatycznie skalować klastry serwerów (patrz „Zasoby obliczeniowe” na stronie 26) w górę i w dół, w zależności od zmiany obciążenia. Główną zaletą bezserwerowej usługi FaaS (patrz „Infrastruktura dla bezserwerowej usługi FaaS” na stronie 240) jest to, że wykonuje ona instancje kodu tylko wtedy, gdy zachodzi taka potrzeba.

Jednak pozostałe elementy infrastruktury, takie jak bazy danych, kolejki komunikatów i urządzenia magazynujące, mogą stawać się wąskimi gardłami w przypadku wzrostu obliczeń. A różne części oprogramowania mogą stawać się wąskimi gardłami nawet niezależnie od infrastruktury.

Zespół ShopSpinner może na przykład wdrażać wiele instancji stosu usługi przeglądania produktów, aby radzić sobie z większym obciążeniem, ponieważ w godzinach szczytu większość ruchu przychodzącego od użytkowników trafia do tej właśnie części systemu. Zespół utrzymuje jedną instancję frontendowego stosu routingu ruchu i jedną instancję stosu bazy danych, z którą łączy się instancje serwera aplikacji (patrz rysunek 15-7).



Rysunek 15-7 Skalowanie liczby instancji różnych stosów

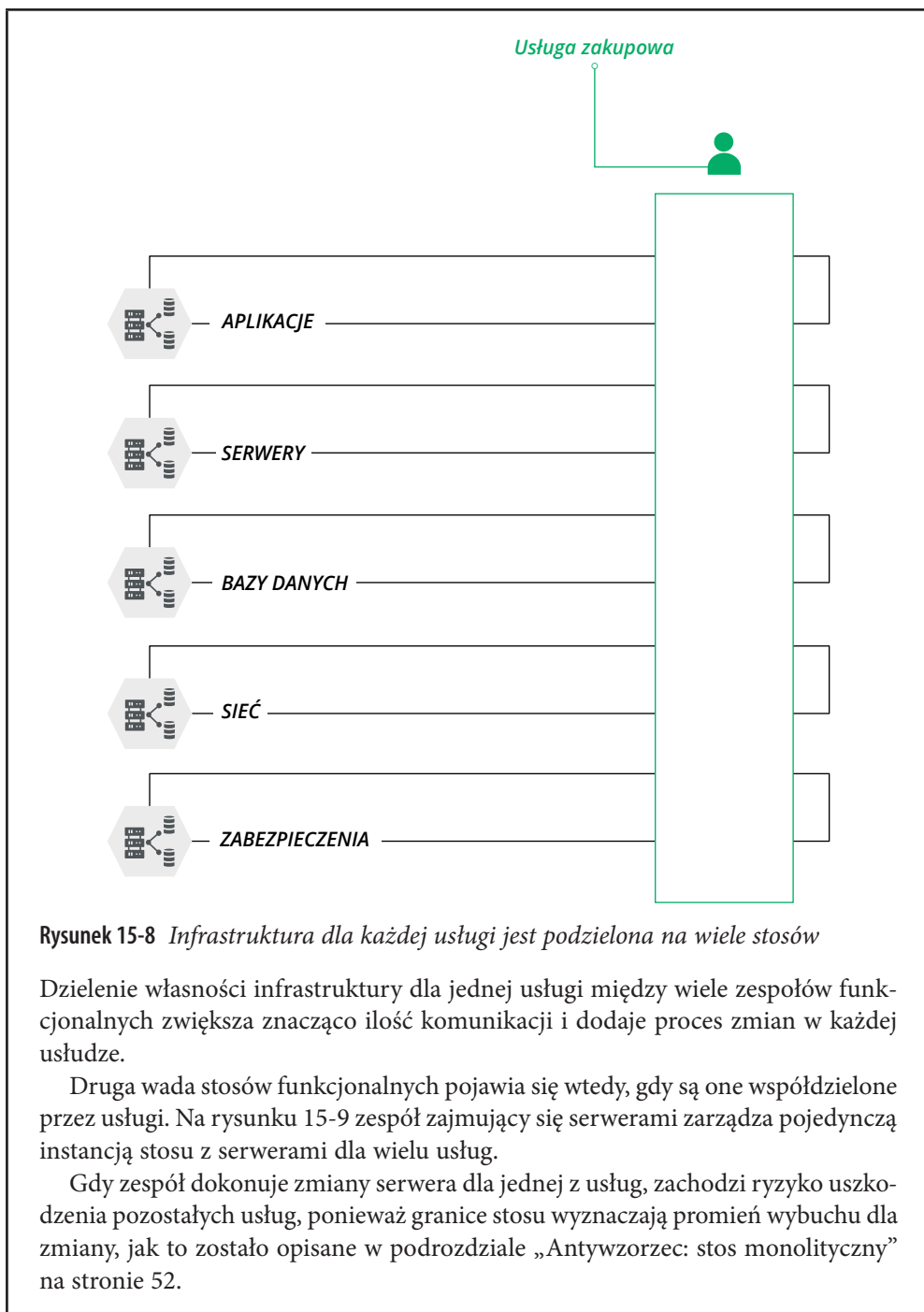
Inne części systemu, takie jak realizacja zamówień usługi zarządzania profilami klientów, prawdopodobnie nie wymagają skalowania razem z usługą przeglądania produktów. Rozbicie tych usług na różne stosy pomaga zespołowi szybciej je skalować. Zmniejsza też marnotrawstwo, które miałyby miejsce w przypadku replikacji wszystkiego.

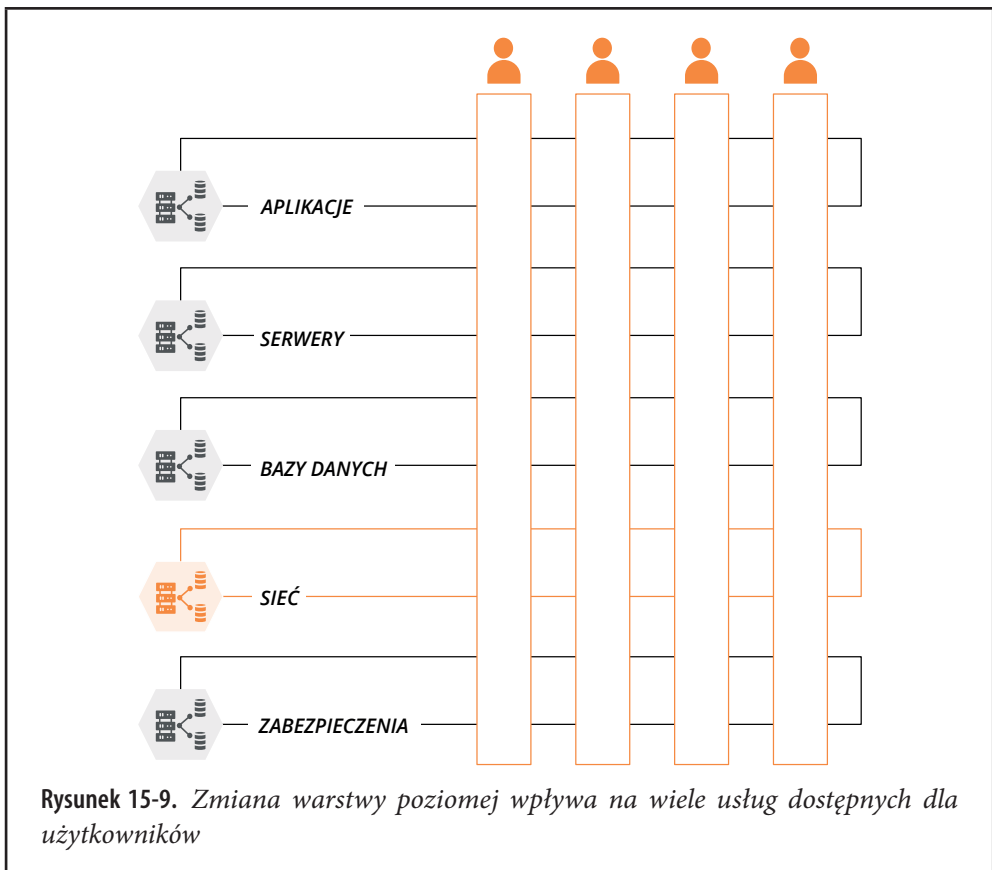
Preferowanie grupowania pionowego a nie poziomego

Tradycyjnie wielu architektów organizuje systemy pod kątem funkcjonalności. Elementy sieciowe mieszkają razem, elementy bazy danych mieszkają razem i elementy systemu operacyjnego mieszkają razem. Jest to często rezultat projektu, zgodnie z przewidywaniami prawa Conway’a – gdy zespoły są organizowane na podstawie technicznej funkcjonalności, to będą dzielić infrastrukturę według tego, czym zarządzają.

W takim podejściu kryje się pułapka, ponieważ usługa udostępniana użytkownikom wykorzystuje wiele funkcjonalności. Jest to często przedstawiane jako pionowa usługa przecinająca poziome warstwy funkcjonalne, jak na rysunku 15-8.

Organizowanie elementów systemu w pionowe, funkcjonalne stosy infrastruktury ma dwie wady. Jedną z nich jest to, że zmiana infrastruktury dla jednej usługi może wymagać dokonania zmian w wielu stosach. Zmiany te muszą być starannie zorkiestrowane, aby nie dopuścić do powstania zależności w stosie konsumencie, zanim pojawi się ona w stosie dostawcy (patrz „Dostawcy i konsumenci”).





Dopasowywanie granic do zasad bezpieczeństwa i nadzoru

Zabezpieczenia, zgodność i nadzór chronią dane, transakcje i dostępność usług. Różne części systemu podlegają różnym regułom. Na przykład standard bezpieczeństwa PCI¹² nakłada wymagania na części systemu obsługujące numery kart kredytowych i przetwarzanie płatności. Również dane osobowe klientów i pracowników muszą być często obsługiwane bardziej rygorystycznie.

Wiele organizacji dzieli swoją infrastrukturę według przepisów i zasad dotyczących hostowanych przez siebie usług i danych. Takie postępowanie zapewnia przejrzystość oceny, jakie środki należy podjąć w przypadku danego komponentu infrastruktury. Proces dostarczania zmian można dopasować do wymagań nadzoru. Na przykład proces ten może wymuszać i rejestrować przeglądy i zatwierdzenia oraz generować raporty o zmianach upraszczające inspekcję.

¹² <https://oreil.ly/JDkPl>



Granice sieci nie są granicami stosu infrastruktury

Ludzie często dzielą infrastrukturę na strefy bezpieczeństwa sieci. Systemy działające w strefie frontendowej są dostępne bezpośrednio z publicznego Internetu, chronione przez zapory i inne mechanizmy. Inne strefy, na przykład dla baz danych i hostingu aplikacji, są dostępne jedynie z innych wybranych stref, z dodatkowymi warstwami zabezpieczeń.

Chociaż te granice są ważne dla ochrony przed atakami sieciowymi, zazwyczaj nie są odpowiednie do organizowania kodu infrastruktury w jednostki nadające się do wdrożenia. Umieszczanie kodu serwerów sieci Web i modułów równoważenia obciążenia we „frontendowym” stosie nie tworzy warstwy chroniącej przed złośliwymi zmianami kodu dotyczącego serwerów aplikacji lub baz danych. Model zagrożeń dla ataków na narzędzia i kod infrastruktury różni się od modelu zagrożeń dla ataków na sieć.

Jak najbardziej należy używać kodu infrastruktury do tworzenia warstwowych granic sieci¹³. Ale nie można zakładać, że dobrym pomysłem będzie wykorzystanie modelu bezpieczeństwa sieci do strukturyzacji kodu infrastruktury.

Podsumowanie

Na początku tego rozdziału opisałem, jak zarządzać większą, bardziej złożoną infrastrukturą zdefiniowaną jako kod, dzieląc ją na mniejsze fragmenty. Przydaje się do tego znajomość poprzednich rozdziałów tej książki, w których omówiłem między innymi jednostki organizacyjne infrastruktury (stosy i serwery) oraz podstawowe praktyki – definiowanie rzeczy jako kodu i ciągle testowanie.

13 Chociaż można rozważać model zerowego zaufania zamiast modeli zabezpieczeń opartych wyłącznie na obwodzie, o czym wspominałem w podrozdziale „Model bezpieczeństwa zerowego zaufania z SDN” na stronie 30.

Budowanie stosów z komponentów

W rozdziale 15 wyjaśniłem, jak dobrze zaprojektowane komponenty mogą sprawić, że dokonywanie zmian w systemie infrastruktury będzie szybsze i bezpieczniejsze. Łączy się to z motywem przewodnim tej książki, czyli używaniem szybkich zmian do ciągłego poprawiania jakości systemu i wykorzystywaniem wysokiej jakości do umożliwiania szybszych zmian.

W tym rozdziale skoncentruję się na modularyzacji stosów infrastruktury, czyli dzieleniu ich na mniejsze fragmenty kodu. Jest kilka powodów, dla których warto rozważać modularyzację stosu:

Ponowne wykorzystanie

Umieszczenie wiedzy o implementacji konkretnej konstrukcji w komponencie pozwala wykorzystywać ją w różnych stosach.

Kompozycja

Zapewnienie możliwości zmieniania implementacji koncepcji zwiększa elastyczność konstrukcji stosu.

Testowalność

Dzielenie stosu na fragmenty, które można niezależnie testować przed ich integracją, zwiększa szybkość i specyficzność testów. Jeśli komponent jest komponowalny, można dodatkowo zastosować warunki początkowe testu („Używanie warunków początkowych testu do obsługi zależności” na stronie 132), aby jeszcze bardziej poprawić izolację i szybkość testowania.

Współdzielenie

Dobrze przetestowane, nadające się do składania (komponowalne) komponenty wielokrotnego użytku mogą być udostępniane innym zespołom, które dzięki temu mogą budować lepsze systemy i robić to szybciej.

Jak już wspomniałem w podrozdziale „Komponenty stosów a stosy jako komponenty” na stronie 250, podział stosu na moduły i biblioteki pozwala uprościć kod, ale w żaden sposób nie zmniejsza ani nie upraszcza instancji stosu. Komponenty stosu mogą potencjalnie

jeszcze pogorszyć sytuację, ukrywając liczbę i złożoność zasobów infrastruktury dodawanych przez siebie do instancji stosu.

Dlatego należy dobrze rozumieć, co się kryje za używanymi abstrakcjami, bibliotekami i platformami. Takie rzeczy są wygodne, ponieważ pozwalają skupić się na zadaniach wyższego poziomu. Jednak nie powinny zastępować pełnego zrozumienia sposobu implementacji systemu.

Języki infrastruktury dla komponentów stosu

W rozdziale 4 opisałem różne rodzaje języków kodu infrastruktury. Dwa główne typy języków do definiowania stosów to deklaratywny (patrz „Deklaratywne języki infrastruktury” na stronie 37) i imperatywny (patrz „Programowalne, imperatywne języki infrastruktury” na stronie 39). Wspomniałem tam również, że różne rodzaje języków są odpowiednie dla różnych rodzajów kodu (patrz „Języki deklaratywne czy imperatywne do infrastruktury” na stronie 40).

Różnice te są często źródłem problemów pojawiających się, gdy ludzie piszą komponenty stosu używając złego języka. Używanie niewłaściwego typu języka prowadzi zwykle do powstania mieszanki kodu deklaratywnego i imperatywnego, co, jak wyjaśniłem wcześniej, jest Złą Rzeczą (patrz „Oddzielaj kod deklaratywny od imperatywnego” na stronie 43).

Decyzja, którego języka użyć, zależy zwykle od narzędzia wykorzystywanego do zarządzania stosem infrastruktury i obsługiwanych przez nie języków¹.

Wzorce zdefiniowane w dalszej części tego rozdziału powinny zachęcić do zastanawiania się nad celem, jaki ma być osiągnięty za pomocą konkretnego stosu i jego komponentów. Aby wykorzystać to do wyboru rodzaju języka i potencjalnie rodzaju narzędzia stosu, które ma zostać użyte, rozważymy dwie klasy komponentów stosu oparte na rodzaju języka.

Ponowne używanie kodu deklaratywnego za pomocą modułów

Większość narzędzi do zarządzania stosem za pomocą języków deklaratywnych pozwala pisać współdzielone komponenty przy użyciu tych samych języków. CloudFormation ma zagnieżdżone stosy², a Terraform ma moduły³. Można przekazywać parametry do tych modułów, a języki umożliwiają przynajmniej pewną programowalność (jak podjęzyk wyrażen HCL dla Terraform). Ale języki są zasadniczo deklaratywne, więc niemal każda zapisana w nich złożona logika wygląda prymitywnie.

1 Gdy to piszę, w połowie 2020 roku, dostawcy narzędzi szybko rozwijają swoje strategie związane z typami języków. Mam nadzieję, że w ciągu najbliższych kilku lat doczekamy się dojrzałej komponentyzacji stosu.

2 <https://oreil.ly/hdRmW>

3 <https://oreil.ly/V4Iyw>

Dlatego moduły zawierające kod deklaracyjny nadają się najlepiej do definiowania komponentów infrastruktury, które nie zmieniają się za bardzo. Moduł deklaracyjny sprawdza się dobrze jako moduł fasady (patrz „Wzorzec: moduł fasady” na stronie 268), który opakowuje i upraszcza zasób udostępniany przez platformę infrastruktury. Takie moduły robią się nieprzyjemne, gdy są używane w bardziej złożonych przypadkach, zmieniając się w moduły spaghetti (patrz „Antywzorzec: moduł spaghetti” na stronie 274).

Jak wspomniałem w podrozdziale „Wyzwanie: testy kodu deklaracyjnego mają często małą wartość” na stronie 105, testowanie modułu deklaracyjnego powinno być stosunkowo proste. Rezultaty stosowania modułu deklaracyjnego są dość podobne do siebie, nie trzeba więc przeprowadzać kompleksowych testów. Nie oznacza to, że w ogóle nie należy testować tych modułów. Jeśli moduł zawiera wiele deklaracji tworzących bardziej złożony obiekt, trzeba sprawdzić, czy spełnia on wszystkie wymagania.

Dynamiczne tworzenie elementów stosu za pomocą bibliotek

Niektóre narzędzia do zarządzania stosem, jak Pulumi i AWS CDK, używają języków imperatywnych ogólnego przeznaczenia. Posługując się tymi językami można tworzyć biblioteki wielokrotnego użytku i odwoływać się do nich w kodzie projektu stosu. Biblioteka może zawierać bardziej złożoną logikę, dynamicznie udostępniającą zasoby infrastruktury w zależności od sposobu użycia. Na przykład infrastruktura zespołu ShopSpinner obejmuje różne stosy infrastruktury serwerów aplikacji. Każdy z tych stosów udostępnia serwer aplikacji i struktury sieciowe dla aplikacji. Niektóre z aplikacji mają być dostępne publicznie, a inne tylko do użytku wewnętrznego.

W obu przypadkach stos infrastruktury musi przypisać do serwera adres IP i nazwę DNS oraz utworzyć trasę sieciową z odpowiedniej bramy. Adres IP i nazwa DNS będą się różnić w przypadku aplikacji publicznych i wewnętrznych. Ponadto aplikacje publiczne potrzebują reguły zapory zezwalającej na połączenia.

Stos `checkout_service` obsługuje aplikację publiczną:

```
application_networking = new ApplicationServerNetwork(PUBLIC_FACING, "checkout")

virtual_machine:
  name: appserver-checkout
  vlan: $(application_networking.address_block)
  ip_address: $(application_networking.private_ip_address)
```

Kod stosu tworzy obiekt `ApplicationServerNetwork` z biblioteki `application_networking`, który udostępnia lub odwołuje się do potrzebnych elementów infrastruktury:

```
class ApplicationServerNetwork {
  def vlan;
  def public_ip_address;
  def private_ip_address;
  def gateway;
```



```

def dns_hostname;
public ApplicationServerNetwork(application_access_type, hostname) {
    if (application_access_type == PUBLIC_FACING) {
        vlan = get_public_vlan()
        public_ip_address = allocate_public_ip()
        dns_hostname = PublicDNS.set_host_record(
            "${hostname}.shopspinnners.xyz",
            this.public_ip_address
        )
    } else {
        // Similar stuff but for a private VLAN
    }

    private_ip_address = allocate_ip_from(this.vlan)
    gateway = get_gateway(this.vlan)
    create_route(gateway, this.private_ip_address)
    if (application_access_type == PUBLIC_FACING) {
        create_firewall_rule(ALLOW, '0.0.0.0', this.private_ip_address, 443)
    }
}
}

```

Ten pseudokod przypisuje serwer do już istniejącej publicznej sieci VLAN i ustawia jego prywatny adres IP z przedziału adresów VLAN. Ustawia również publiczny wpis DNS dla serwera, który w naszym przykładzie ma mieć postać *checkout.shopspinnners.xyz*. Biblioteka wyszukuje bramę na podstawie użytej sieci VLAN, czyli w przypadku aplikacji wewnętrznej będzie się to różnić.

Wzorce dla komponentów stosu

Poniższy zestaw wzorców i antywzorców prezentuje różne podejścia do projektowania komponentów stosu oraz oceny już istniejących komponentów. Nie jest to pełna lista sposobów, jak należy i jak nie należy budować modułów i bibliotek. Jest to raczej punkt wyjścia do rozważań na ten temat.

Wzorzec: moduł fasady

Znany również jako: moduł opakowujący.

Moduł fasady tworzy uproszczony interfejs zasobu należącego do języka narzędzia stosu lub platformy infrastruktury. Moduł udostępnia kilka parametrów, których może użyć wywołujący kod (patrz przykład 16-1).

Przykład 16-1 Przykładowy kod wykorzystujący moduł fasady

```
use module: shopspinner-server
  name: checkout-appserver
  memory: 8GB
```

Moduł wykorzystuje te parametry do wywoływania zasobu, który opakowuje, a pozostałe parametry wymagane przez zasób koduje na stałe (przykład 16-2).

Przykład 16-2 Kod przykładowego modułu fasady

```
declare module: shopspinner-server
  virtual_machine:
    name: ${name}
    source_image: hardened-linux-base
    memory: ${memory}
    provision:
      tool: servermaker
      maker_server: maker.shopspinner.xyz
      role: application_server
  network:
    vlan: application_zone_vlan
```

Wywołanie tego przykładowego modułu pozwala utworzyć serwer wirtualny o podanej nazwie i ilości pamięci. Każdy serwer utworzony za pomocą tego modułu wykorzystuje obraz źródłowy, rolę i sieć zdefiniowane przez moduł.

Motywacja

Moduł fasady upraszcza i standaryzuje typowy przypadek użycia danego zasobu infrastruktury. Kod stosu wykorzystujący moduł fasady powinien być dzięki temu prostszy i bardziej czytelny. Poprawa jakości kodu modułu staje się natychmiast dostępna we wszystkich stosach, które go używają.

Zastosowanie

Moduł fasady nadaje się najlepiej do prostych przypadków użycia, zazwyczaj obejmujących podstawowy zasób infrastruktury.

Konsekwencje

Moduł fasady pozwala ograniczyć sposób wykorzystania danego zasobu infrastruktury. Może to być przydatne, gdyż redukuje warianty i standaryzuje implementacje, przez co stają się one lepsze i bezpieczniejsze. Ale zmniejsza jednocześnie elastyczność, więc nie nadaje się do każdego przypadku użycia.

Moduł stanowi dodatkową warstwę kodu między kodem stosu i kodem, który bezpośrednio określa zasób infrastruktury. Ta dodatkowa warstwa powoduje zwiększenie

nakładów na utrzymanie, debugowanie i ulepszanie kodu. Może także utrudniać zrozumienie kodu stosu.

Implementacja

Implementowanie modułu fasady polega zwykle na określeniu zasobu infrastruktury za pomocą wielu wartości zakodowanych na stałe i małej liczby wartości przekazywanych z poziomu kodu wykorzystującego moduł. Odpowiednim językiem dla modułu fasady jest język deklaracyjny infrastruktury.

Powiązane wzorce

Moduł zaciemniający jest modulem fasady, który niewiele ukrywa, za to zwiększa złożoność i nie wnosi wiele wartości. Moduł pakietowy („Wzorzec: moduł pakietowy” na stronie 273) deklaruje wiele powiązanych zasobów infrastruktury, czyli jest jak moduł fasady z większą liczbą części.

Antywzorzec: moduł zaciemniający

Moduł zaciemniający opakowuje kod dla elementu infrastruktury zdefiniowanego przez język stosu lub platformę infrastruktury, ale nie upraszcza go ani nie wnosi żadnej konkretnej wartości. W najgorszych przypadkach moduł ten dodatkowo komplikuje kod (patrz przykład 16-3).

Przykład 16-3 Przykładowy kod wykorzystujący moduł zaciemniający

```
use module: any_server
  server_name: checkout-appserver
  ram: 8GB
  source_image: base_linux_image
  provisioning_tool: servermaker
  server_role: application_server
  vlan: application_zone_vlan
```

Moduł sam przekazuje parametry bezpośrednio do kodu narzędzia do zarządzania stosem, jak to jest pokazane w przykładzie 16-4.

Przykład 16-4 Kod przykładowego modułu zaciemniającego

```
declare module: any_server
  virtual_machine:
    name: ${server_name}
    source_image: ${origin_server_image}
    memory: ${ram}
  provision:
    tool: ${provisioning_tool}
    role: ${server_role}
```

```
network:  
  vlan: ${server_vlan}
```

Motywacja

Moduł zaciemniający może być modulem fasady (patrz „Wzorzec: moduł fasady” na stronie 268), w którym coś poszło źle. Czasami ludzie piszą tego rodzaju moduł, chcąc trzymać się zasady DRY (patrz „Unikanie powielania” na stronie 247). Widzą, że kod, który definiuje typowy element infrastruktury, taki jak serwer wirtualny, moduł równoważenia obciążenia czy grupę zabezpieczeń, jest używany w wielu miejscach w bazie kodu. Tworzą więc moduł, który raz deklaruje ten typ elementu i używają go wszędzie. Ale ponieważ elementy są używane inaczej w różnych częściach kodu, muszą udostępniać dużą liczbę parametrów w swoim module.

Inni ludzie tworzą moduły zaciemniające, dążąc do zaprojektowania własnego języka do odwoływania się do elementów infrastruktury, „ulepszając” język udostępniony przez używane narzędzie stosu.

Zastosowanie

Nikt celowo nie tworzy modułu zaciemniającego. Można dyskutować, czy dany moduł jest modulem zaciemniającym, czy fasadowym i takie rozważania są użyteczne. Należy się przyjrzeć, czy moduł wnosi jakąś prawdziwą wartość, i jeśli nie, to zmienić go na kod bezpośrednio używający języka stosu.

Konsekwencje

Pisanie, używanie i utrzymywanie kodu modułu zamiast bezpośredniego wykorzystywania konstrukcji udostępnianych przez narzędzie stosu oznacza dodatkowe obciążenie. W efekcie jest więcej kodu do utrzymania, więcej rzeczy do nauki i więcej ruchomych części w procesie budowy i dostarczania. Komponent powinien dawać korzyści uzasadniające ponoszenie dodatkowych kosztów.

Implementacja

Jeśli moduł ani nie upraszcza zasobów, które definiuje, ani nie wnosi dodatkowej wartości w stosunku do stojącego za nim kodu języka stosu, należy rozważyć jego zamianę na bezpośrednie użycie języka stosu.

Powiązane wzorce

Moduł zaciemniający jest podobny do modułu fasady (patrz „Wzorzec: moduł fasady” na stronie 268), ale nie upraszcza znacząco kryjącego się za nim kodu.

Antywzorzec: moduł niewspółdzielony

Moduł niewspółdzielony jest używany tylko raz w bazie kodu, czyli nie jest wykorzystywany przez wiele stosów.

Motywacja

Ludzie zwykle tworzą moduły niewspółdzielone w celu uporządkowania kodu w projekcie stosu.

Zastosowanie

W miarę rozrastania się kodu projektu w pewnym momencie staje się kuszące podzielenie go na moduły. Jeśli kod zostanie tak podzielony, że pozwoli testować każdy moduł oddzielnie, to może to ułatwić pracę. W przeciwnym razie lepiej jest poszukać innej metody usprawnienia bazy kodu.

Konsekwencje

Zorganizowanie kodu pojedynczego stosu w moduły zwiększa nakład pracy przy bazie kodu, prawdopodobnie dodając wersjonowanie i inne ruchome części. Budowanie modułów wielokrotnego użytku, gdy praktycznie będą one używane jednokrotnie, to przykład paradygmatu *YAGNI* („You Aren’t Gonna Need It” – nie będziesz tego potrzebować)⁴, czyli wkładania teraz wysiłku w coś, co może, ale nie musi, przydać się w przyszłości.

Implementacja

Gdy projekt stosu staje się zbyt duży, jest kilka innych rozwiązań poza umieszczaniem jego kodu w modułach. Często lepiej jest rozbić stos na kilka stosów, używając odpowiedniego wzorca stosu strukturalnego (patrz rozdział 17). Jeśli stos jest dość spójny (patrz „Cechy dobrze zaprojektowanych komponentów” na stronie 246), można po prostu розміścić kod w oddzielnych plikach i, jeśli trzeba, w oddzielnych folderach. Takie działanie może ułatwić poruszanie się po kodzie i zrozumienie go bez nakładu pracy towarzyszącego innym opcjom.

Reguła trzech w przypadku ponownego wykorzystania oprogramowania sugeruje, aby przekształcać coś w komponent wielokrotnego użytku, jeśli można znaleźć trzy miejsca, w których trzeba go użyć⁵.

Powiązane wzorce

Moduł niewspółdzielony może dokładnie odwzorowywać albo elementy infrastruktury niskiego poziomu, jak moduł fasady („Wzorzec: moduł fasady” na stronie 268), albo

⁴ <https://oreil.ly/nQBTm>

⁵ Reguła trzech została zdefiniowana w książce Roberta Glassa *Facts and Fallacies of Software Engineering* (Addison-Wesley). Ponadto Jeff Atwood skomentował tę regułę w swoim poście dotyczącym iluzji ponownego wykorzystania (<https://oreil.ly/TBkQC>).

jednostki wyższego poziomu, jak jednostka domeny infrastruktury („Wzorzec: jednostka domeny infrastruktury” na stronie 277).

Wzorzec: moduł pakietowy

Moduł pakietowy deklaruje kolekcję powiązanych zasobów infrastruktury z uproszczonym interfejsem. Kod stosu wykorzystuje taki moduł do definiowania tego, co musi udostępnić:

```
use module: application_server
  service_name: checkout_service
  application_name: checkout_application
  application_version: 1.23
  min_cluster: 1
  max_cluster: 3
  ram_required: 4GB
```

Kod modułu deklaruje wiele zasobów infrastruktury, zazwyczaj skoncentrowanych wokół głównego zasobu. W przykładzie 16-5 takim zasobem jest klastery serwerów, ale oprócz niego jest jeszcze moduł równoważenia obciążenia oraz wpis DNS.

Przykład 16-5 Kod modułu dla serwera aplikacji

```
declare module: application_server

  server_cluster:
    id: "${service_name}-cluster"
    min_size: ${min_cluster}
    max_size: ${max_cluster}
    each_server_node:
      source_image: base_linux
      memory: ${ram_required}
      provision:
        tool: servermaker
        role: appserver
        parameters:
          app_package: "${checkout_application}-${application_version}.war"
          app_repository: "repository.shopspinner.xyz"

  load_balancer:
    protocol: https
    target:
      type: server_cluster
      target_id: "${service_name}-cluster"

  dns_entry:
    id: "${service_name}-hostname"
    record_type: "A"
```

```
hostname: "${service_name}.shopspinner.xyz"  
ip_address: {$load_balancer.ip_address}
```

Motywacja

Moduł pakietowy jest wygodny do zdefiniowania spójnej kolekcji zasobów infrastruktury. Pozwala uniknąć rozwlekłego, redundantnego kodu. Tego typu moduły są przydatne do gromadzenia wiedzy o różnych potrzebnych elementach i sposobach ich powiązania w celu wspólnego wykorzystania.

Zastosowanie

Moduł pakietowy jest odpowiedni w przypadku wykorzystywania deklaratywnego języka stosu, gdy zaangażowane zasoby nie zmieniają się zbyt często w zależności od przypadku użycia. Jeśli jednak potrzebny jest moduł do tworzenia różnych zasobów lub konfigurowania ich w odmienny sposób w zależności od przeznaczenia, to należy albo utworzyć oddzielne moduły, albo przejść na język imperatywny i utworzyć jednostkę domeny infrastruktury (patrz „Wzorzec: jednostka domeny infrastruktury” na stronie 277).

Konsekwencje

W niektórych sytuacjach moduł pakietowy potrafi udostępniać więcej zasobów niż potrzeba. Użytkownicy modułu powinni wiedzieć, co udostępnia i starać się nie korzystać z niego, jeśli w ich przypadku jest to przesadą.

Implementacja

Moduł należy definiować w sposób deklaratywny, dołączając elementy infrastruktury blisko związane z zadeklarowanym przeznaczeniem.

Powiązane wzorce

Moduł fasady („Wzorzec: moduł fasady” na stronie 268) opakowuje pojedynczy zasób infrastruktury, natomiast moduł pakietowy obejmuje wiele zasobów, chociaż oba mają charakter deklaratywny. Jednostka domeny infrastruktury („Wzorzec: jednostka domeny infrastruktury” na stronie 277) jest podobna do modułu pakietowego, ale dynamicznie generuje zasoby infrastruktury. Moduł spaghetti to moduł pakietowy, który chciałby być jednostką domeny, ale wpada w obłęd na skutek ograniczeń języka deklaratywnego.

Antywzorzec: moduł spaghetti

Moduł spaghetti jest konfigurowalny do tego stopnia, że pozwala dawać znacząco różne wyniki w zależności od dostarczonych parametrów. Implementacja tego modułu jest zagniatwana i trudna do zrozumienia, ponieważ jest w nim zbyt wiele ruchomych części (patrz przykład 16-6).

Przykład 16-6 *Przykład modułu spaghetti*

```
declare module: application-server-infrastructure
variable: network_segment = {
  if ${parameter.network_access} = "public"
    id: public_subnet
  else if ${parameter.network_access} = "customer"
    id: customer_subnet
  else
    id: internal_subnet
  end
}

switch ${parameter.application_type}:
  "java":
    virtual_machine:
      origin_image: base_tomcat
      network_segment: ${variable.network_segment}
      server_configuration:
        if ${parameter.database} != "none"
          database_connection: ${database_instance.my_database.connection_string}
        end
      ...
  "NET":
    virtual_machine:
      origin_image: windows_server
      network_segment: ${variable.network_segment}
      server_configuration:
        if ${parameter.database} != "none"
          database_connection: ${database_instance.my_database.connection_string}
        end
      ...
  "php":
    container_group:
      cluster_id: ${parameter.container_cluster}
      container_image: nginx_php_image
      network_segment: ${variable.network_segment}
      server_configuration:
        if ${parameter.database} != "none"
          database_connection: ${database_instance.my_database.connection_string}
        end
      ...
end

switch ${parameter.database}:
  "mysql":
```



```
database_instance: my_database
  type: mysql
  ...
...
```

Ten przykładowy kod tworzy serwer i przypisuje go do jednego z trzech różnych segmentów sieci oraz opcjonalnie tworzy klaster bazy danych i przekazuje parametry połączenia do konfiguracji serwera. Natomiast w niektórych przypadkach tworzy grupę instancji kontenerów zamiast serwera wirtualnego. Taki moduł jest trochę jak potwór.

Motywacja

Podobnie jak w przypadku innych antywzorców, moduł spaghetti powstaje przez przypadek, często po pewnym czasie. Można utworzyć moduł fasady („Wzorzec: moduł fasady” na stronie 268) lub moduł pakietowy („Wzorzec: moduł pakietowy” na stronie 273), który z czasem staje się coraz bardziej złożony, żeby mógł obsługiwać bardzo odmienne przypadki, z pozoru wyglądające na podobne.

Moduły spaghetti są często wynikiem próby implementacji jednostki domeny infrastruktury („Wzorzec: jednostka domeny infrastruktury” na stronie 277) przy użyciu języka deklaratywnego.

Konsekwencje

Moduł, który robi zbyt wiele rzeczy, jest trudniejszy w utrzymaniu niż moduł o węższym zakresie działania. Im więcej rzeczy wykonuje w infrastrukturze i im bardziej mogą być one zróżnicowane, tym trudniej jest go zmienić nie psując niczego. Takie moduły są też trudniejsze do testowania. Jak wyjaśniłem w rozdziale 8, lepiej zaprojektowany kod jest łatwiejszy do testowania. Jeśli więc są problemy z napisaniem zautomatyzowanych testów i zbudowaniem potoków do testowania modułu w izolacji, jest to znak, że mamy do czynienia z modułem spaghetti.

Implementacja

Kod modułu spaghetti często zawiera konstrukcje warunkowe, które powodują stosowanie różnych specyfikacji w różnych sytuacjach. Na przykład moduł klastra baz danych może mieć parametr wskazujący, którą bazę danych ma udostępnić.

Po wykryciu modułu spaghetti należy go refaktoryzować.

Często można go podzielić na różne moduły, każdy o bardziej konkretnym przeznaczeniu. Na przykład, można dokonać dekompozycji modułu infrastruktury jednej aplikacji na różne moduły dla poszczególnych części infrastruktury tej aplikacji. Stos wykorzystujący moduły będące efektem takiej dekompozycji, zamiast modułu spaghetti z przykładu 16-6, może wyglądać tak, jak w przykładzie 16-7.

Przykład 16-7 *Przykład wykorzystania modułów utworzonych w wyniku dekompozycji pojedynczego modułu spaghetti*

```
use module: java-application-servers
  name: checkout_appserver
  application: "shopping_app"
  application_version: "4.20"
  network_segment: customer_subnet
  server_configuration:
    database_connection: ${module.mysql-database.outputs.connection_string}

use module: mysql-database
  cluster_minimum: 1
  cluster_maximum: 3
  allow_connections_from: customer_subnet
```

Każdy z tych modułów jest mniejszy, prostszy i dużo łatwiejszy do utrzymania i testowania niż oryginalny moduł spaghetti.

Powiązane wzorce

Moduł spaghetti to często efekt próby zbudowania jednostki domeny infrastruktury przy użyciu kodu deklaratywnego. W założeniu mógł to być także moduł fasady („Wzorzec: moduł fasady” na stronie 268) lub moduł pakietowy („Wzorzec: moduł pakietowy” na stronie 273), który ktoś próbował następnie rozbudować, aby obsługiwał różne przypadki użycia.

Wzorzec: jednostka domeny infrastruktury

Jednostka domeny infrastruktury implementuje komponent stosu wyższego poziomu, łącząc wiele zasobów infrastruktury niższego poziomu. Przykładem koncepcji wyższego poziomu jest infrastruktura potrzebna do uruchomienia aplikacji.

Oto przykład pokazujący, jak można w kodzie projektu stosu używać biblioteki implementującej instancję infrastruktury aplikacji Java:

```
use module: application_server
  service_name: checkout_service
  application_name: checkout_application
  application_version: 1.23
  traffic_level: medium
```

Powyższy kod definiuje aplikację i wersję do wdrożenia, a także poziom natężenia ruchu. Kod biblioteki jednostki domeny mógłby wyglądać podobnie jak przykładowy moduł pakietowy (przykład 16-5), ale zawiera kod dynamiczny, który udostępnia zasoby zgodnie z parametrem `traffic_level`:

```

...
switch (${traffic_level}) {
  case ("high") {
    $appserver_cluster.min_size = 3
    $appserver_cluster.max_size = 9
  } case ("medium") {
    $appserver_cluster.min_size = 2
    $appserver_cluster.max_size = 5
  } case ("low") {
    $appserver_cluster.min_size = 1
    $appserver_cluster.max_size = 2
  }
} ...

```

Motywacja

Jednostka domeny stanowi często część warstwy abstrakcji (patrz „Budowanie warstwy abstrakcji” na stronie 279), której można używać do definiowania i budowania infrastruktury opartej na wymaganiach wyższego poziomu. Zespół platformy infrastruktury buduje w ten sposób komponenty, których inne zespoły mogą używać do składania własnych stosów.

Zastosowanie

Ponieważ jednostka domeny infrastruktury dynamicznie udostępnia zasoby infrastruktury, powinna być napisana raczej w języku imperatywnym niż deklaratywnym. Więcej na ten temat jest w podrozdziale „Języki deklaratywne czy imperatywne do infrastruktury” na stronie 40.

Implementacja

Na konkretnym poziomie zaimplementowanie jednostki domeny infrastruktury jest kwestią napisania kodu. Ale najlepszym sposobem utworzenia wysokiej jakości bazy kodu, łatwego do zrozumienia i utrzymania, jest zastosowanie podejścia opartego na projekcie.

Zalecam korzystanie z wiedzy wyniesionej z lekcji projektowania i architektury oprogramowania. Wzorzec jednostki domeny infrastruktury wywodzi się z metody projektowania DDD (*Domain Driven Design*), która polega na tworzeniu modelu koncepcyjnego dla domeny biznesowej systemu oprogramowania i wykorzystywaniu go do projektowania samego systemu⁶. Infrastruktura, zwłaszcza zaprojektowana i zbudowana jako oprogramowanie, powinna być postrzegana jako domena sama w sobie. Domena ta buduje, dostarcza i uruchamia oprogramowanie.

⁶ Patrz *Domain-Driven Design: Tackling Complexity in the Heart of Software*, autor Eric Evans (Addison-Wesley).

Szczególnie skutecznym podejściem jest wykorzystywanie przez organizację metody DDD do projektowania architektury dla oprogramowania biznesowego, a następnie rozszerzanie domeny w celu włączenia do niej systemów i usług używanych do budowania i uruchamiania tego oprogramowania.

Powiązane wzorce

Moduł pakietowy (patrz „Wzorzec: moduł pakietowy” na stronie 273) jest podobny do jednostki domeny, ponieważ tworzy spójną kolekcję zasobów infrastruktury. Jednak moduł pakietowy tworzy zazwyczaj dość statyczny zbiór zasobów, bez większego zróżnicowania. Moduł pakietowy prezentuje zwykle podejście wstępujące, zaczynające się od zasobów infrastruktury, które mają zostać utworzone. Jednostka domeny jest podejściem zstępującym, zaczynającym się od wymagań przypadku użycia.

Większość modułów spaghetti (patrz „Antywzorzec: moduł spaghetti” na stronie 274) dla stosów infrastruktury jest rezultatem wykorzystywania kodu deklaratywnego do implementacji logiki dynamicznej. Z czasem jednostka domeny infrastruktury robi się nadmiernie skomplikowana. Jednostka domeny o słabej spójności staje się modulem spaghetti.

Budowanie warstwy abstrakcji

Warstwa abstrakcji dostarcza uproszczony interfejs zasobów niskiego poziomu. Zbiór komponentów stosu, umożliwiających składanie i wielokrotne użycie, może pełnić rolę warstwy abstrakcji zasobów infrastruktury. Komponenty mogą implementować wiedzę o tym, jak składać zasoby niskiego poziomu udostępniane przez platformę infrastruktury w jednostki przydatne dla osób skoncentrowanych na zadaniach wyższego poziomu.

Na przykład zespół aplikacji może potrzebować środowiska obejmującego serwer aplikacji, instancję bazy danych oraz dostęp do kolejek komunikatów. W tym celu zespół może użyć komponentów, które wyodrębnią szczegóły reguł składania dla routingu i uprawnień do zasobów infrastruktury.

Komponenty mogą być przydatne nawet dla zespołów dysponujących umiejętnościami i doświadczeniem w implementowaniu zasobów niskiego poziomu. Warstwa abstrakcji pomaga odseparować różne sprawy, dzięki czemu pozwala skupić się na problemie na konkretnym poziomie szczegółowości. Powinna także umożliwiać dogłębną analizę i ewentualne ulepszanie lub rozszerzanie stojących za nią komponentów, gdy zajdzie taka potrzeba.

W przypadku niektórych systemów bywa możliwe zaimplementowanie warstwy abstrakcji przy użyciu raczej statycznych elementów, takich jak moduły fasadowe (patrz „Wzorzec: moduł fasady” na stronie 268) lub moduły pakietowe (patrz „Wzorzec: moduł pakietowy” na stronie 273). Ale częściej potrzebne są bardziej elastyczne komponenty warstwy, dlatego dynamiczne komponenty, takie jak jednostki domeny infrastruktury (patrz „Wzorzec: jednostka domeny infrastruktury” na stronie 277) są bardziej przydatne.

Warstwa abstrakcji może powstać samoczynnie na skutek tworzenia przez ludzi bibliotek i innych komponentów. Ale dobrze jest mieć projekt i standardy wyższego poziomu, tak aby komponenty warstwy dobrze współpracowały ze sobą i pasowały do spójnego obrazu systemu.

Komponenty warstwy abstrakcji są zwykle budowane przy użyciu języka infrastruktury niskiego poziomu („Języki infrastruktury niskiego poziomu” na stronie 50). Wiele zespołów uważa za wygodne wykorzystanie warstwy abstrakcji do budowy języka wysokiego poziomu („Języki infrastruktury wysokiego poziomu” na stronie 51) do definiowania stosów. Efektem jest często język deklaracyjny wyższego poziomu określający wymagania części środowiska wykonawczego aplikacji, który odwołuje się do komponentów napisanych w języku imperatywnym niskiego poziomu.



Modele abstrakcji aplikacji

Model OAM, Open Application Model⁷, jest przykładem próby zdefiniowania standardowej architektury rozdzielającej aplikację, środowisko wykonawcze i infrastrukturę.

Podsumowanie

Budowanie stosów z komponentów może być przydatne, gdy nad infrastrukturą pracuje i używa jej wiele osób i zespołów. Trzeba mieć jednak świadomość złożoności, jaką niosą ze sobą warstwy abstrakcji i biblioteki komponentów, i pilnować, aby użycie tych konstrukcji odpowiadało wielkości i złożoności systemu.

⁷ <https://oam.dev>

Używanie stosów jako komponentów

W systemie infrastruktury stos jest zwykle komponentem najwyższego poziomu. Jest to największa jednostka, jaką można niezależnie definiować, wyposażać i zmieniać. Wzorzec stosu wielokrotnego użytku (patrz „Wzorzec: stos wielokrotnego użytku” na stronie 65) zachęca do traktowania stosu jako głównej jednostki służącej do udostępniania i wielokrotnego używania infrastruktury.

Infrastruktury złożone z małych stosów są bardziej funkcjonalne niż duże stosy złożone z modułów i bibliotek. Mały stos można zmieniać szybciej, łatwiej i bezpieczniej niż duży stos. Taka strategia wspiera pozytywny cykl wykorzystywania szybkości zmian do poprawy jakości i wysokiej jakości do umożliwienia szybkich zmian. Budowanie systemów z wielu małych stosów wymaga pilnowania, aby każdy stos miał odpowiednią wielkość i był dobrze zaprojektowany, spójny i luźno sprzężony. Rada z rozdziału 15 ma również zastosowanie do stosów, a także innych rodzajów komponentów infrastruktury. Szczególnym wyzwaniem związanym ze stosami jest implementacja ich integracji bez tworzenia ścisłego sprzężenia.

Integracja stosów obejmuje zwykle jeden stos, który zarządza jakimś zasobem, oraz drugi stos, który z niego korzysta. Jest wiele popularnych technik implementacji wykrywania i integracji zasobów przez stosy, ale ich efektem jest często ścisłe sprzężenie, które utrudnia dokonywanie zmian. Dlatego w tym rozdziale przedstawię różne podejścia pod kątem ich wpływu na sprzężenie.

Wykrywanie zależności między stosami

System ShopSpinner zawiera stos konsumenta, `application-infrastructurestack`, zintegrowany z elementami sieci zarządzanymi przez inny stos, `sharednetwork-stack`. Stos sieci deklaruje sieć VLAN:

```
vlan:  
  name: "appserver_vlan"  
  address_range: 10.2.0.0/8
```

Stos aplikacji definiuje serwer aplikacji, przypisany do sieci VLAN:

```
virtual_machine:  
  name: "appserver-${ENVIRONMENT_NAME}"  
  vlan: "appserver_vlan"
```

W tym przykładzie zależność między stosami jest zakodowana na stałe, tworząc bardzo ściśle sprzężenie. Do testowania zmian w kodzie stosu aplikacji będzie zawsze potrzebna instancja stosu sieci. Z kolei zmiany stosu sieci będą ograniczone przez zależność drugiego stosu od nazwy sieci VLAN.

Jeśli ktoś zdecyduje się dodać więcej sieci VLAN w celu zapewnienia większej odporności, zmusi konsumentów do zmiany implementacji kodu ich stosu wraz ze zmianą stosu sieci¹. W przeciwnym razie, jeśli pozostanie oryginalna nazwa, zrobi się bałagan, który utrudni zrozumienie i utrzymywanie infrastruktury:

```
vlans:  
  - name: "appserver_vlan"  
    address_range: 10.2.0.0/8  
  - name: "appserver_vlan_2"  
    address_range: 10.2.1.0/8  
  - name: "appserver_vlan_3"  
    address_range: 10.2.2.0/8
```

Kodowanie na stałe punktów integracji potrafi utrudnić utrzymywanie wielu instancji infrastruktury, na przykład w różnych środowiskach. Może się sprawdzać, w zależności od API platformy infrastruktury dla konkretnych zasobów. Na przykład, jeśli tworzymy instancję stosu infrastruktury dla każdego środowiska na innym koncie w chmurze, to możemy w każdym z nich używać tej samej nazwy VLAN. Ale częściej zachodzi konieczność integracji z wykorzystaniem różnych nazw zasobów w poszczególnych środowiskach.

Dlatego należy unikać kodowania zależności na stałe. Zamiast tego lepiej jest rozważyć użycie jednego z poniższych wzorców odkrywania zależności.

Wzorzec: dopasowywanie zasobów

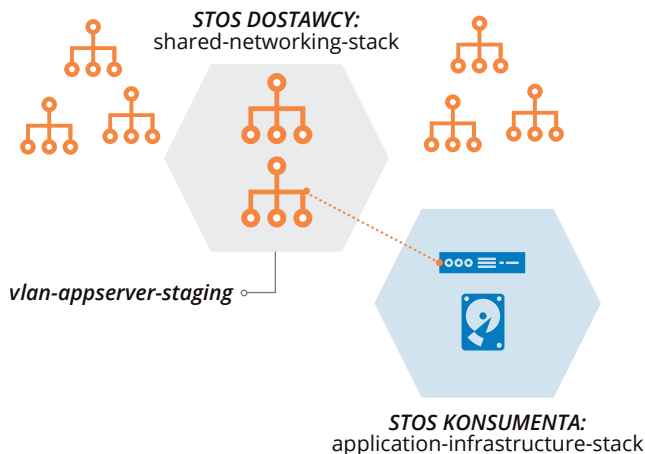
Stos konsument wykorzystuje *dopasowywanie zasobów* od wykrywania zależności, szukając zasobów infrastruktury pasujących do nazw, tagów lub innych cech identyfikujących. Na przykład stos dostawcy może nazywać sieci VLAN według należących do nich rodzajów zasobów i ich środowiska (patrz rysunek 17-1)

W tym przykładzie `vlan-appserver-staging` jest nazwą przeznaczoną dla serwerów aplikacji w środowisku przejściowym. Kod stosu `application-infrastructure-stack` odnajduje ten zasób dopasowując wzorzec nazewnictwa:

1 W podrozdziale „Zmiana działającej infrastruktury” na stronie 360 są podane mniej uciążliwe sposoby zarządzania tego rodzaju zmianą.

```
virtual_machine:
  name: "appserver-#{ENVIRONMENT_NAME}"
  vlan: "vlan-appserver-#{ENVIRONMENT_NAME}"
```

Wartość `ENVIRONMENT_NAME` jest przekazywana do narzędzia do zarządzania stosem podczas stosowania kodu, zgodnie z opisem w rozdziale 7.



Rysunek 17-1 Dopasowywanie zasobów w celu wykrycia zależności

Motywacja

Dopasowywanie zasobów jest proste do zaimplementowania za pomocą większości języków i narzędzi do zarządzania stosem. Wzorec ten eliminuje większość zakodowanych na stałe zależności, redukując sprzężenie.

Dopasowywanie zasobów pozwala również uniknąć sprzężenia narzędzi. Infrastruktura dostawcy i stos konsumenta mogą być zaimplementowane przy użyciu różnych narzędzi.

Zastosowanie

Dopasowywanie zasobów w celu wykrywania zależności może być stosowane, gdy zespoły zarządzające kodem dostawcy i konsumenta mają dobre rozeznanie, które zasoby powinny być używane jako zależności. W przypadku problemów z podziałem zależności między zespoły należy rozważyć przejście na inny, alternatywny wzorec.

Dopasowywanie zasobów przydaje się w większych organizacjach lub w grupach organizacji, w których różne zespoły używają różnych narzędzi do zarządzania swoją infrastrukturą, ale nadal potrzebują integracji na poziomie infrastruktury. Nawet tam, gdzie w danej chwili wszyscy używają jednego narzędzia, dopasowywanie zasobów zmniejsza uzależnienie od tego narzędzia, stwarzając możliwość wykorzystania nowych narzędzi do innych części systemu.

Konsekwencje

Zaraz po zaimplementowaniu przez stos konsumenta dopasowywania zasobów w celu wykrywania zasobu z innego stosu wzorzec dopasowywania staje się kontraktem. Jeśli ktoś zmieni wzorzec nazewnictwa sieci VLAN we współdzielonym stosie sieci, nastąpi przerwanie zależności konsumenta.

Dlatego zespół konsumenta powinien wykrywać zależności za pomocą dopasowywania zasobów tylko w taki sposób, jaki zespół dostawcy jawnie obsługuje. Zespoły dostawców powinny wyraźnie komunikować, jakie wzorce dopasowywania zasobów obsługują i dbać o utrzymanie integralności tych wzorców jako kontraktu.

Implementacja

Istnieje kilka sposobów wykrywania zasobów infrastruktury przez dopasowywanie. Najprostszą metodą jest używanie zmiennych w nazwie zasobu, jak to już zostało pokazane we wcześniejszym przykładzie:

```
virtual_machine:
  name: "appserver-${ENVIRONMENT_NAME}"
  vlan: "vlan-appserver-${ENVIRONMENT_NAME}"
```

Ciąg `vlan-appserver-${ENVIRONMENT_NAME}` będzie pasował do odpowiedniej sieci VLAN dla danego środowiska.

Większość języków stosu dysponuje funkcjami dopasowywania innych atrybutów poza nazwą zasobu. W Terraform może to być źródło danych², a w CDK AWS importowanie zasobu³.

W poniższym przykładzie (wykorzystującym pseudokod) dostawca przypisuje tagi do swoich sieci VLAN:

```
vlan:
- appserver_vlan
  address_range: 10.2.0.0/8
  tags:
    network_tier: "application_servers"
    environment: ${ENVIRONMENT_NAME}
```

Kod konsumenta wykrywa potrzebną sieć VLAN używając tych tagów:

```
external_resource:
  id: appserver_vlan
  match:
    tag: name == "network_tier" && value == "application_servers"
    tag: name == "environment" && value == ${ENVIRONMENT_NAME} virtual_machine:
```

² <https://oreil.ly/4tkdu>

³ <https://oreil.ly/ZLEFv>

```
name: "appserver-${ENVIRONMENT_NAME}"
vlan: external_resource.appserver_vlan
```

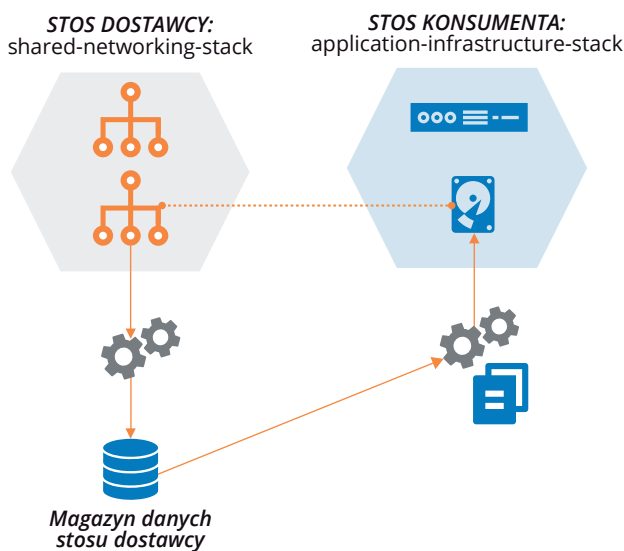
Powiązane wzorce

Wzorzec dopasowywania zasobów jest podobny do wzorca wyszukiwania danych stosu. Główna różnica polega na tym, że dopasowywanie zasobów nie zależy od implementacji tego samego narzędzia stosu u dostawcy i konsumenta.

Wzorzec: wyszukiwanie danych w stosie

Znany również jako: zdalne wyszukiwanie w pliku stanu, wyszukiwanie odwołań w stosie lub wyszukiwanie zasobów w stosie.

Wyszukiwanie danych w stosie polega na znajdowaniu zasobów dostawcy przy użyciu struktur danych utrzymywanych przez narzędzie zarządzające stosem dostawcy (rysunek 17-2).



Rysunek 17-2 Wyszukiwanie danych w stosie w celu wykrycia zależności

Wiele narzędzi do zarządzania stosem utrzymuje struktury danych dla każdej instancji stosu, które obejmują wartości eksportowane przez kod stosu. Przykładem tego są zdalne pliki stanu Terraform i Pulumi.

Motywacja

Dostawcy narzędzi do zarządzania stosem pozwalają w prosty sposób używać ich funkcji wyszukiwania danych w stosie do integracji różnych projektów. Większość implementacji udostępniania danych między stosami wymaga od stosu dostawcy jawnego

zadeklarowania, które zasoby mają zostać opublikowane, aby mogły być używane przez inne stosy. Takie postępowanie zniechęca stosy konsumentów do tworzenia zależności od zasobów bez wiedzy dostawcy.

Zastosowanie

Wykorzystywanie funkcji wyszukiwania danych w stosie do wykrywania zależności między stosami działa, gdy cała infrastruktura w systemie jest zarządzana przy użyciu tego samego narzędzia.

Konsekwencje

Wyszukiwanie danych w stosie powoduje zwykle ograniczenie się do jednego narzędzia do zarządzania stosem. Jest możliwe zastosowanie wzorca z różnymi narzędziami, jak to jest opisane w implementacji tego wzorca, ale powoduje to większą złożoność implementacji.

Wzorzec ten nie działa czasem w różnych wersjach tego samego narzędzia stosu. Aktualizacja narzędzia może bowiem spowodować zmiany struktur danych stosu. To bywa źródłem problemów podczas uaktualniania stosu dostawcy do nowszej wersji narzędzia. Dopóki nie zostanie również uaktualniony stos konsument do nowej wersji narzędzia, starsza jego wersja może nie być w stanie wyodrębnić wartości zasobu z uaktualnionego stosu dostawcy. Uniemożliwia to stopniowe wdrażanie aktualizacji narzędzia stosu w stosach, potencjalnie wymuszając uciążliwą, skoordynowaną aktualizację w całym podległym systemie.

Implementacja

Implementacja wyszukiwania danych w stosie wykorzystuje funkcję narzędzia do zarządzania stosem i język jego definicji.

Terraform przechowuje wartości wyjściowe⁴ w zdalnym pliku stanu. Pulumi również przechowuje szczegóły zasobów w pliku stanu, do którego można się odwoływać w stosie konsumencie za pomocą StackReference⁵. CloudFormation pozwala eksportować i importować wartości wyjściowe stosu⁶ między stosami, a dostęp do nich ma także AWS CDK⁷.

Dostawca zazwyczaj jawnie deklaruje zasoby udostępniane konsumentom:

```
stack:
  name: shared_network_stack
  environment: ${ENVIRONMENT_NAME}
vlans:
  - appserver_vlan
```

⁴ https://oreil.ly/9_tiM

⁵ <https://oreil.ly/5ITLw>

⁶ <https://oreil.ly/75KHJ>

⁷ Patrz AWS CDK Developer Guide (<https://oreil.ly/jWSJG>).

```
    address_range: 10.2.0.0/8
export:
  - appserver_vlan_id: appserver_vlan.id
```

Konsument deklaruje odwołanie do stosu dostawcy i używa tego odwołania w identyfikatorze sieci VLAN eksportowanym przez ten stos:

```
external_stack:
  name: shared_network_stack
  environment: ${ENVIRONMENT_NAME}

virtual_machine:
  name: "appserver-${ENVIRONMENT_NAME}"
  vlan: external_stack.shared_network_stack.appserver_vlan.id
```

W powyższym przykładzie kod stosu zawiera osadzone odwołanie do zewnętrznego stosu. Inną opcją jest użycie wstrzykiwania zależności (patrz „Wstrzykiwanie zależności” na stronie 290), dzięki któremu kod stosu jest mniej sprzężony z wykrywaniem zależności. Skrypt orkiestracji wyszukuje wartość wyjściową w stosie dostawcy i przekazuje ją do kodu stosu jako parametr.

Chociaż wyszukiwanie danych w stosie jest związane z narzędziem, które zarządza stosem dostawcą, zazwyczaj można pobrać te wartości za pomocą skryptu w celu wykozystania przez inne narzędzia, jak to jest pokazane w przykładzie 17-1.

Przykład 17-1 *Użycie narzędzia stosu do wykrycia identyfikatora zasobu w strukturze danych instancji stosu*

```
#!/usr/bin/env bash

VLAN_ID=$(
  stack value \
    --stack_instance shared_network-staging \
    --export_name appserver_vlan_id
)
```

Powyższy kod uruchamia fikcyjne polecenie stack, określające przeszukiwaną instancję stosu (shared_network-staging) oraz eksportowaną zmienną do odczytu i druku (appserver_vlan_id). Polecenie powłoki zapamiętuje wynik polecenia, którym jest identyfikator sieci VLAN, w zmiennej powłoki o nazwie VLAN_ID. Skrypt może następnie używać tej zmiennej na różne sposoby.

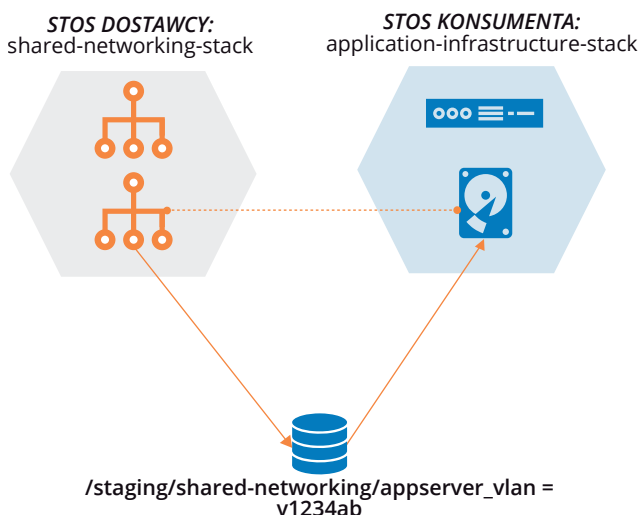
Powiązane wzorce

Główne alternatywne wzorce to dopasowywanie zasobów („Wzorzec: dopasowywanie zasobów” na stronie 282) oraz wyszukiwanie w rejestrze.

Wzorzec: wyszukiwanie w rejestrze integracji

Znany również jako: rejestr integracji.

Używając wyszukiwania w rejestrze integracji stos konsumenta może wykryć zasób opublikowany przez stos dostawcy (rysunek 17-3). Oba stosy odwołują się do rejestru, wykorzystując jego znaną lokalizację do zapisywania i odczytywania wartości.



Rysunek 17-3 Wyszukiwanie w rejestrze integracji w celu wykrycia zależności

Wiele narzędzi stosu obsługuje zapisywanie i pobieranie wartości przy użyciu różnego typu rejestrów w kodzie definicji. Kod `shared-networking-stack` ustawia wartość:

```
vlans:
  - appserver_vlan
    address_range: 10.2.0.0/8

registry:
  host: registry.shopspinner.xyz
  set:
    /${ENVIRONMENT_NAME}/shared-networking/appserver_vlan: appserver_vlan.id
```

Następnie kod `application-infrastructure-stack` pobiera i wykorzystuje tę wartość:

```
registry:
  id: stack_registry
  host: registry.shopspinner.xyz
  values:
    appserver_vlan_id: /${ENVIRONMENT_NAME}/shared-networking/appserver_vlan
```

```
virtual_machine:  
  name: "appserver-${ENVIRONMENT_NAME}"  
  vlan: stack_registry.appserver_vlan_id
```

Motywacja

Stosowanie rejestru konfiguracji rozdziela narzędzia do zarządzania stosem używane do różnych stosów infrastruktury. Różne zespoły mogą używać różnych narzędzi, o ile zgadzają się korzystać z tej samej usługi rejestru konfiguracji i identycznej konwencji nazewnictwa do przechowywania wartości w tym rejestrze. Taki rozdział ułatwia ponadto stopniowe uaktualnianie i zmienianie narzędzi, po jednym stosie na raz.

Zastosowanie rejestru konfiguracji sprawia, że punkty integracji stosów stają się jawne. Stos konsument może używać tylko wartości jawnie opublikowanych przez stos dostawcy, więc zespół dostawcy może swobodnie zmieniać sposób implementacji swoich zasobów.

Zastosowanie

Wzorzec wyszukiwania w rejestrze konfiguracji jest przydatny dla większych organizacji, w których różne zespoły mogą używać różnych technologii. Może być także użyteczny dla organizacji zaniepokojonych ograniczeniem do jednego narzędzia.

Jeśli system wykorzystuje już rejestr konfiguracji („Rejestr konfiguracji” na stronie 93), na przykład do przekazywania wartości konfiguracji do instancji stosów zgodnie z wzorcem rejestru parametrów stosu („Wzorzec: rejestr parametrów stosu” na stronie 90), to może mieć sens użycie tego samego rejestru do integracji stosów.

Konsekwencje

W momencie zastosowania wzorca wyszukiwania w rejestrze konfiguracji rejestr ten staje się usługą krytyczną. Można nie mieć możliwości udostępniania lub odzyskiwania zasobów, gdy rejestr przestanie być dostępny.

Implementacja

Warunkiem wstępnym używania tego wzorca jest rejestr konfiguracji, omówiony w podrozdziale „Rejestr konfiguracji” (rozdział 7). Niektórzy dostawcy narzędzi do infrastruktury udostępniają serwery rejestrów, o czym można przeczytać w podrozdziale „Rejestry narzędzi do automatyzacji infrastruktury” na stronie 94. W przypadku każdego oferowanego rejestru trzeba się upewnić, że jest on dobrze obsługiwany przez narzędzia, których używamy i które możemy brać pod uwagę w przyszłości.

Należy koniecznie ustalić jasne zasady nazewnictwa parametrów, zwłaszcza w przypadku używania rejestru do integracji infrastruktury przez wiele zespołów. Liczne organizacje wykorzystują hierarchiczną przestrzeń nazw, podobną do struktury katalogów lub folderów, nawet jeśli rejestr ma zaimplementowany prosty mechanizm klucz/wartość. Taka struktura zawiera zwykle komponenty przeznaczone dla jednostek architektonicznych (takich jak usługi, aplikacje czy produkty), środowisk, miejsc geograficznych lub zespołów.

Na przykład system ShopSpinner mógłby używać ścieżki hierarchicznej opartej na regionach geograficznych:

```
/infrastructure/  
├─ au/  
│   ├── shared-networking/  
│   │   └─ appserver_vlan=  
│   └─ application-infrastructure/  
│       └─ appserver_ip_address=  
└─ eu/  
    ├── shared-networking/  
    │   └─ appserver_vlan=  
    └─ application-infrastructure/  
        └─ appserver_ip_address=
```

W tym przykładzie adres IP serwera aplikacji dla rejonu Europy jest w lokalizacji */infrastructure/eu/application-infrastructure/appserver_ip_address*.

Powiązane wzorce

Wzorec ten jest podobny do wzorca wyszukiwania danych w stosie (patrz „Wzorec: wyszukiwanie danych w stosie” na stronie 285), który również używa rejestru do zapisywania i pobierania wartości. Jednak ten drugi wzorec wykorzystuje struktury danych specyficzne dla narzędzia do zarządzania stosem, podczas gdy wzorec wyszukiwania w rejestrze integracji używa implementacji rejestru ogólnego przeznaczenia. Zasadniczo identyczny jest jeszcze inny wzorec, wzorec rejestru parametrów (patrz „Wzorec: rejestr parametrów stosu” na stronie 90), w którym jeden stos pobiera wartości z rejestru, aby użyć ich w danej instancji stosu. Jedyna różnica polega na tym, że w przypadku tego wzorca wartość pochodzi z innego stosu i jest jawnie używana do integracji zasobów infrastruktury między stosami.

Wstrzykiwanie zależności

Przedstawione wzorce opisują strategię wykrywania przez konsumenta zasobów zarządzanych przez dostawcę. Większość narzędzi do zarządzania stosem pozwala bezpośrednio używać tych wzorców w kodzie definicji stosu. Jest jednak argument przemawiający za rozdzieleniem kodu definiującego zasoby stosu od kodu wykrywającego zasoby, z którymi ma nastąpić integracja.

Rozważmy poniższy fragment z wcześniejszego przykładu implementacji wzorca dopasowywania zależności (patrz „Wzorec: dopasowywanie zasobów” na stronie 282):

```
external_resource:  
  id: appserver_vlan  
  match:
```

```
tag: name == "network_tier" && value == "application_servers"
tag: name == "environment" && value == ${ENVIRONMENT_NAME}

virtual_machine:
  name: "appserver-${ENVIRONMENT_NAME}"
  vlan: external_resource.appserver_vlan
```

Zasadniczą część tego kodu stanowi deklaracja maszyny wirtualnej. Cała reszta to elementy zewnętrzne, szczegóły implementacji służące zgromadzeniu wartości konfiguracyjnych dla maszyny wirtualnej.

Problemy z mieszaniem kodu definicji i zależności

Łączenie wykrywania zależności z kodem definicji stosu stanowi dodatkowe obciążenie poznawcze podczas czytania tego kodu lub pracy nad nim. Chociaż nie stanowi to przeszkody nie do pokonania, powoduje lekkie utrudnienie.

Można usunąć ten narzut poznawczy dzieląc kod na oddzielne pliki w projekcie stosu. Ale innym problemem związanym z włączaniem kodu wykrywania do projektu definicji stosu jest tworzenie sprzężenia stosu z mechanizmem zależności.

Rodzaj i stopień sprzężenia z mechanizmem zależności i wpływ, jaki ma to sprzężenie, różnią się w zależności od mechanizmów i ich implementacji. Należy unikać lub przynajmniej minimalizować sprzężenie ze stosem dostawcą oraz z usługami, takimi jak rejestr konfiguracji.

Sprzężenie definicji i zarządzania zależnościami może utrudniać tworzenie i testowanie instancji stosu. Wiele podejść do testowania wykorzystuje praktyki, takie jak instancje efemeryczne („Wzorzec: efemeryczny stos testowy” na stronie 137) lub atrapy („Używanie warunków początkowych testu do obsługi zależności” na stronie 132) do umożliwienia szybkiego i częstego testowania. Może to stanowić wyzwanie, jeśli skonfigurowanie zależności wymaga zbyt wiele pracy i czasu.

Zakodowanie określonych założeń dotyczących zależności na stałe w kodzie stosu może utrudnić jego wielokrotne używanie. Na przykład, jeśli jakiś zespół tworzy stos infrastruktury podstawowego serwera, z którego mają korzystać inne zespoły, mogą one chcieć używać różnych metod do konfigurowania swoich zależności i zarządzania nimi. Niektóre z nich mogą nawet chcieć wymienić stos dostawcy. W przypadku stosu sieci będzie to inna wersja dla aplikacji publicznych, a inna dla aplikacji wewnętrznych.

Oddzielanie zależności od ich wykrywania

Wstrzykiwanie zależności (Dependency injection – DI) to technika, w której komponent otrzymuje zależności, a nie sam je wykrywa. Projekt stosu infrastruktury deklaruje zasoby, od których zależy, jako parametry, identyczne jak parametry konfiguracji instancji opisane w rozdziale 7. Skrypt lub inne narzędzie, które orkiestruje narzędzie do zarządzania stosem („Używanie skryptów do opakowywania narzędzi infrastruktury” na stronie 327), odpowiada za wykrywanie zależności i przekazywanie ich do stosu.

Zobaczmy, jak przykład `application-infrastructure-stack` używany wcześniej do zilustrowania wzorców wykrywania zależności wyglądałby w wersji z DI:

```
parameters:
  - ENVIRONMENT_NAME
  - VLAN

virtual_machine:
  name: "appserver-${ENVIRONMENT_NAME}"
  vlan: ${VLAN}
```

Powyższy kod deklaruje dwa parametry, które muszą być ustawione w momencie stosowania kodu do instancji. Parametr `ENVIRONMENT_NAME` jest prostym parametrem stosu, używanym do nazwania maszyny wirtualnej serwera aplikacji. Parametr `VLAN` jest identyfikatorem sieci VLAN, do której należy przypisać maszynę wirtualną.

Aby zarządzać instancją tego stosu, trzeba odkryć i udostępnić wartość parametru `VLAN`. Może to zrobić skrypt orkiestracji, używając dowolnego wzorca opisanego w tym rozdziale. Może ustawić parametr na podstawie tagów, znajdując dane wyjściowe projektu stosu dostawcy za pomocą narzędzia do zarządzania stosem albo wyszukać wartość w rejestrze.

Przykładowy skrypt wykorzystujący wyszukiwanie danych w stosie (patrz „Wzorec: wyszukiwanie danych w stosie” na stronie 285) mógłby użyć narzędzia stosu do pobrania identyfikatora sieci VLAN z instancji stosu dostawcy, jak w przykładzie 17-1, a następnie przekazać tę wartość do polecenia `stack` dla instancji stosu konsumenta:

```
#!/usr/bin/env bash

ENVIRONMENT_NAME=$1

VLAN_ID=$(
  stack value \
    --stack_instance shared_network-${ENVIRONMENT_NAME} \
    --export_name appserver_vlan_id
)

stack apply \
  --stack_instance application_infrastructure-${ENVIRONMENT_NAME} \
  --parameter application_server_vlan=${VLAN_ID}
```

Pierwsze polecenie pobiera wartość `appserver_vlan_id` z instancji stosu dostawcy o nazwie `shared_network-${ENVIRONMENT_NAME}`, a następnie przekazuje ją jako parametr do stosu konsumenta `application_infrastructure-${ENVIRONMENT_NAME}`.

Zaletą tego podejścia jest prostszy kod definicji stosu i możliwość używania go w różnym kontekście. Pracując nad zmianami kodu stosu na swoim laptopie można przekazać dowolną wartość VLAN. Można zastosować kod używając lokalnej atrapy API (patrz „Testowanie z użyciem atrapy API” na stronie 127) albo do osobistej instancji na platformie infrastruktury (patrz „Prywatne instancje infrastruktury” na stronie 338). Sieci VLAN udostępniane w takich sytuacjach mogą być bardzo proste.

W środowiskach bardziej zbliżonych do produkcyjnych sieć VLAN może być częścią bardziej złożonego stosu sieci. Możliwość podmiany różnych implementacji dostawcy ułatwia implementację testowania progresywnego (patrz „Testowanie progresywne” na stronie 109), w którym wcześniejsze etapy potoku wykonują się szybko i testują komponent konsumenta w izolacji, a etapy późniejsze testują zintegrowany system w sposób bardziej kompleksowy.



Pochodzenie wstrzykiwania zależności

DI wywodzi się ze świata projektowania oprogramowania obiektowego (OO) w pierwszych latach XXI wieku. Zwolennicy XP odkryli, że testy jednostkowe i TDD były znacznie łatwiejsze w przypadku oprogramowania napisanego pod kątem DI. Pionierami DI były struktury Java, takie jak PicoContainer⁸ i Spring⁹. Martin Fowler w swoim artykule „Inversion of Control Containers and the Dependency Injection Pattern” z 2004 roku¹⁰ podaje uzasadnienie takiego podejścia do projektowania oprogramowania obiektowego.

Podsumowanie

Składanie infrastruktury z dobrze zaprojektowanych, odpowiedniej wielkości luźno sprzężonych stosów pozwala łatwiej, szybciej i bezpieczniej wprowadzać zmiany w systemie. Takie podejście wymaga postępowania zgodnie z ogólnymi zasadami projektowania dotyczącymi modularyzacji infrastruktury (patrz rozdział 15). Wymaga także upewnienia się, że stosy nie stają się zbyt silnie ze sobą sprzężone na skutek udostępniania i dostarczania zasobów.

⁸ <https://oreil.ly/jF07b>

⁹ <https://oreil.ly/qGt9g>

¹⁰ <https://oreil.ly/1eMFQ>

CZĘŚĆ V

Dostarczanie infrastruktury

Organizowanie kodu infrastruktury

Baza kodu infrastruktury może zawierać różne rodzaje kodu, w tym definicje stosów, konfiguracje serwerów, moduły, biblioteki, testy, konfigurację i narzędzia.

Jak powinien być zorganizowany ten kod dla wielu projektów i każdego z nich z osobna? Jak powinny być zorganizowane projekty w ramach repozytoriów? Czy kod infrastruktury i kod aplikacji powinny być trzymane razem, czy oddzielnie? Jak powinien być zorganizowany kod dla środowiska podzielonego na wiele części?

Organizowanie projektów i repozytoriów

W tym kontekście *projekt* jest kolekcją kodu używanego do budowania dyskretnych komponentów systemu. Nie ma sztywnej reguły, jak duży może być jeden projekt lub jego komponent. W podrozdziale „Wzorce i antywzorce konstruowania stosów” na stronie 52 opisane są różne przykłady zasięgu stosu infrastruktury.

Projekt może być zależny od innych projektów w bazie kodu. W wersji idealnej zależności te i granice między projektami są dobrze zdefiniowane i wyraźnie odzwierciedlone w sposobie organizacji kodu projektu.

Prawo Conwaya (patrz „Dopasowywanie granic do struktur organizacyjnych” na stronie 258) mówi, że istnieje bezpośredni związek między strukturą organizacji i budowanymi przez nią systemami. Złe dopasowanie struktur zespołów i praw własności systemów oraz kodu definiującego te systemy, powoduje zgrzyty i brak wydajności.

Odwrotną stroną wytyczania granic między projektami jest integrowanie projektów, między którymi występują zależności, jak to zostało opisane dla stosów w rozdziale 17.

O tym, jak i kiedy można integrować z projektem różne zależności, można przeczytać w podrozdziale „Integrowanie projektów” na stronie 317.

Problem organizacji kodu ma dwa wymiary. Jednym z nich jest miejsce przechowywania różnych rodzajów kodu – kodu stosów, konfiguracji serwerów, obrazów serwerów, konfiguracji, testów, narzędzi dostarczania i aplikacji. Drugim jest sposób organizacji projektów w ramach repozytoriów kodu źródłowego. To drugie zagadnienie jest nieco prostsze, więc zajmijmy się nim w pierwszej kolejności.

Jedno repozytorium czy wiele?

Zakładając, że mamy wiele projektów kodu, pojawia się pytanie, czy należy trzymać je wszystkie w jednym repozytorium w systemie kontroli źródła, czy rozdzielić je na kilka? Jeśli już zdecydujemy się na więcej niż jedno repozytorium, to czy każdy projekt powinien mieć własne repozytorium, czy niektóre projekty należy pogrupować i trzymać w repozytoriach współdzielonych? I jeśli chcemy trzymać wiele projektów w wielu repozytoriach, to na jakiej podstawie mamy decydować, które projekty należy pogrupować, a które rozdzielić?

Należy wziąć pod uwagę kilka czynników za i przeciw:

- Trzymanie projektów w różnych repozytoriach ułatwia utrzymywanie granic na poziomie kodu.
- Zmuszanie wielu zespołów do pracy nad kodem w jednym repozytorium może tworzyć narzut i rodzić konflikty.
- Umieszczanie kodu w wielu repozytoriach może komplikować pracę nad zmianami wykraczającymi poza jedno repozytorium.
- Kod trzymany razem w jednym repozytorium jest wersjonowany i może się rozgałęziać, co upraszcza integrację niektórych projektów i strategię dostarczania.
- Różne systemy zarządzania kodem źródłowym (takie jak Git, Perforce i Mercurial) mają inną wydajność i skalowalność oraz funkcje obsługujące złożone scenariusze.

Przyjrzyjmy się głównym wariantom organizacji projektów w ramach repozytoriów w świetle tych czynników.

Jedno repozytorium na wszystko

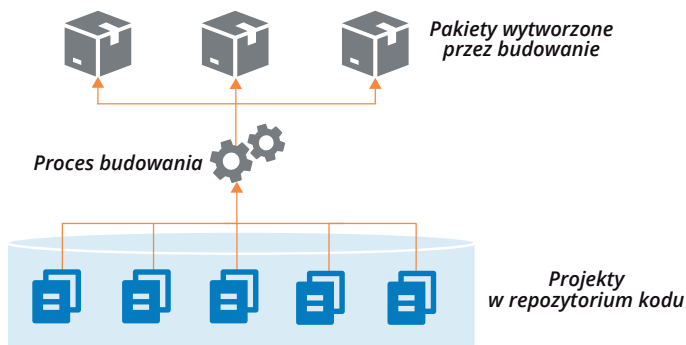
Niektóre zespoły, a nawet niektóre większe organizacje, utrzymują jedno repozytorium dla całego swojego kodu. Wymaga to oprogramowania systemu kontroli źródła umożliwiającego skalowanie do wymaganego poziomu użycia. Niektóre systemy mają problem z obsługą bazy kodu w miarę powiększania się jej rozmiaru, historii, liczby użytkowników i poziomu aktywności¹. Dlatego dzielenie repozytorium staje się kwestią zarządzania wydajnością.

Pojedyncze repozytorium może być łatwiejsze w użyciu. Ludzie mogą wyewidencjonować dowolny projekt, nad którym muszą pracować, gwarantując spójność każdej wersji. Niektóre programy do kontroli wersji oferują funkcje, takie jak sparse-checkout, które pozwalają użytkownikowi pracować z podzbiorem repozytorium.

1 Facebook, Google i Microsoft używają bardzo dużych repozytoriów. Każda z tych firm dokonała niestandardowych zmian w używanym oprogramowaniu do kontroli wersji albo stworzyła własne. Więcej informacji można znaleźć w „Scaling version control software” (<https://oreil.ly/2KBk8>). Aby poznać historię podejścia Google’a, można także zajrzeć do „Scaled trunk-based development” (<https://oreil.ly/Dc21t>) Paula Hammanta.

Monorepo – jedno repozytorium, jedna budowa

Pojedyncze repozytorium dobrze się sprawdza w przypadku integracji podczas budowania (patrz „Wzorzec: integracja projektów podczas budowania” na stronie 319). Strategia monorepo wykorzystuje wzorzec integracji podczas budowania do projektów trzymanych w jednym repozytorium. Uproszczona wersja monorepo polega na budowie wszystkich projektów w repozytorium, jak pokazano na rysunku 18-1.

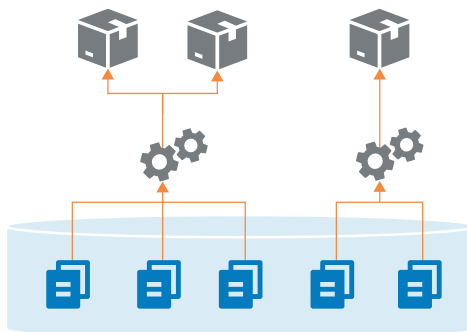


Rysunek 18-1 Budowanie wszystkich projektów w repozytorium razem

Chociaż projekty są budowane razem, mogą tworzyć wiele artefaktów, takich jak pakiety aplikacji, stopy infrastruktury czy obrazy serwerów.

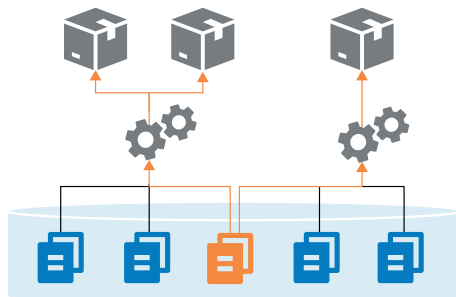
Jedno repozytorium, wiele budów

Większość organizacji, które trzymają wszystkie swoje projekty w jednym repozytorium, niekoniecznie uruchamia jedną budowę dla nich wszystkich. Często stosują oddzielne budowy w celu utworzenia różnych podzbiorów swojego systemu (patrz rysunek 18-2).



Rysunek 18-2 Budowa różnych kombinacji projektów z jednego repozytorium

Takie budowy często współdzielią pewne projekty. Na przykład dwie różne budowy mogą używać tej samej biblioteki współdzielonej (rysunek 18-3).

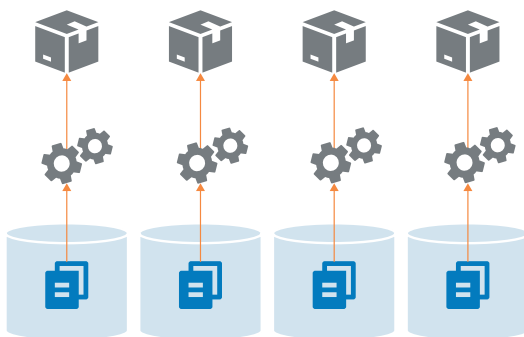


Rysunek 18-3 *Współdzielenie komponentu przez różne budowy w jednym repozytorium*

Jedną z pułapek zarządzania wieloma projektami jest możliwość zatarcia granic między nimi. W kodzie pisanym dla jednego projektu, mogą pojawić się bezpośrednie odwołania do plików innego projektu w tym samym repozytorium. Taka sytuacja prowadzi do ściślejszego sprzężenia i słabszej widoczności zależności. Z czasem projekty stają się splątane i trudne do utrzymania, ponieważ zmiana w pliku jednego projektu może wywołać nieoczekiwane konflikty w innych projektach.

Oddzielne repozytorium dla każdego projektu (mikrorepo)

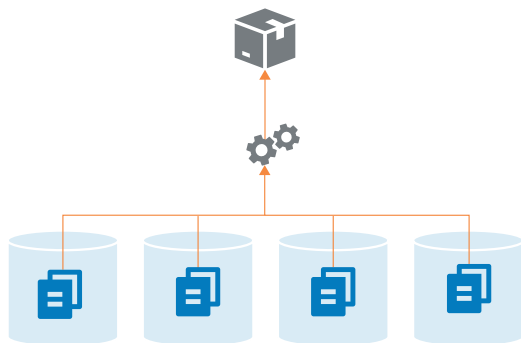
Stosowanie oddzielnego repozytorium dla każdego projektu to druga skrajność (rysunek 18-4).



Rysunek 18-4 *Każdy projekt w oddzielnym repozytorium*

Taka strategia gwarantuje wyraźne rozdzielenie projektów, zwłaszcza w przypadku potoku budującego i testującego każdy projekt oddzielnie przed ich integracją. Jeśli ktoś wywendycjonuje dwa projekty i dokona zmiany plików w obrębie tych projektów, potok zakończy się niepowodzeniem, ujawniając problem.

Podchodząc czysto technicznie, można użyć integracji podczas budowania wielu projektów zarządzanych w oddzielnych repozytoriach, dokonując najpierw wywidencjonowania wszystkich budów (patrz rysunek 18-5).



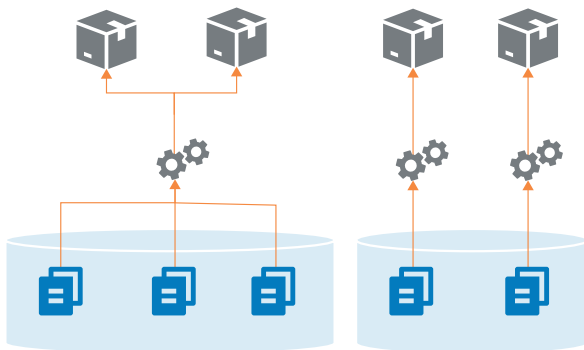
Rysunek 18-5 Jedna budowa w obrębie wielu repozytoriów

W praktyce lepszym wyjściem jest budowanie z użyciem wielu projektów w jednym repozytorium, ponieważ ich kod jest wtedy wersjonowany wspólnie. Wypychanie zmian dla jednej budowy do wielu repozytoriów komplikuje proces dostarczania. Etap dostarczania musiałby w jakiś sposób wiedzieć, które wersje zaangażowanych repozytoriów należy wywidencjonować, aby uzyskać spójną budowę.

Jednoprojektowe repozytoria sprawdzają się najlepiej w przypadku integracji na etapie dostarczania lub stosowania. Zmiana w dowolnym repozytorium uruchamia proces dostarczania jego projektu, łącząc go z innymi projektami w dalszej części przepływu.

Wiele repozytoriów z wieloma projektami

Podczas gdy niektóre organizacje wybierają jedną lub drugą skrajność – jedno repozytorium na wszystko lub oddzielne repozytorium na każdy projekt – większość utrzymuje wiele repozytoriów z więcej niż jednym projektem (patrz rysunek 18-6).



Rysunek 18-6 Wiele repozytoriów z wieloma projektami

Często grupowanie projektów w repozytoriach następuje w sposób organiczny, a nie w wyniku zastosowania strategii typu monorepo czy mikrorepo. Jest jednak kilka czynników, które wpływają na to, jak płynnie wszystko działa.

Jednym czynnikiem, jak wynika z dyskusji nad innymi strategiami repozytorium, jest dopasowanie grupowania projektów do strategii budowy i dostarczania. Do jednego repozytorium powinny trafiać projekty, które są blisko powiązane, zwłaszcza gdy są integrowane podczas budowania. Jeśli ścieżki dostarczania projektów nie są ściśle zintegrowane, można rozważyć trzymanie tych projektów w oddzielnych repozytoriach.

Innym czynnikiem jest posiadanie prawa własności przez zespół. Chociaż nad różnymi projektami w tym samym repozytorium może pracować wiele osób i zespołów, bywa to kłopotliwe. Dzienniki zmian zawierają w efekcie poprzepłatane historie zatwierdzeń od różnych zespołów na skutek niepowiązanych strumieni pracy. Niektóre organizacje ograniczają dostęp do kodu. Kontrola dostępu do systemów kontroli źródła jest często zarządzana przez repozytorium, co stanowi kolejny czynnik decydujący o tym, gdzie trzymać poszczególne projekty.

Jak wspomniałem przy okazji pojedynczych repozytoriów, projekty w ramach repozytorium łatwiej ulegają splątaniu na skutek zależności plików. Dlatego zespoły mogą rozdzielać projekty między repozytoria na podstawie tego, gdzie są potrzebne silniejsze granice z architektonicznego i projektowego punktu widzenia.

Organizowanie różnych rodzajów kodu

Różne projekty w bazie kodu infrastruktury definiują różnego typu elementy systemu, takie jak aplikacje, stosy infrastruktury, moduły konfiguracji serwerów czy biblioteki. Te projekty mogą zawierać różne rodzaje kodu, w tym deklaracje, kod imperatywny, wartości konfiguracyjne, testy i skrypty narzędziowe. Dysponowanie strategią organizacji tych rzeczy pomaga zachować bazę kodu w stanie ułatwiającym jej utrzymanie.

Pliki pomocnicze projektu

Ogólnie rzecz biorąc każdy kod pomocniczy danego projektu powinien być trzymany razem z kodem tego projektu. Typowy układ projektu dla stosu może wyglądać tak, jak w przykładzie 18-1.

Przykład 18-1 *Przykładowy układ folderów dla projektu*

```
|— build.sh
|— src/
|— test/
|— environments/
|— pipeline/
```

Struktura folderów dla tego projektu obejmuje:

`src/`

Kod stosu infrastruktury, stanowiący serce projektu.

`test/`

Kod testów. Folder ten można podzielić na podfoldery dla testów uruchamianych w różnych fazach, na przykład offline i online. Testy wykorzystujące różne narzędzia, jak analiza statyczna, testy wydajnościowe i testy funkcjonalne, również mogą mieć swoje podfoldery.

`environments/`

Konfiguracja. Folder ten zawiera oddzielny plik z wartościami konfiguracyjnymi dla każdej instancji stosu.

`pipeline/`

Konfiguracja dostarczania. Folder zawiera pliki konfiguracyjne do tworzenia etapów dostarczania w narzędziu potoku dostarczania (patrz „Usługi i oprogramowanie potoków dostarczania” na stronie 117).

`build.sh/`

Skrypt do implementacji działań budowy. Tego typu skrypty są omówione w podrozdziale „Używanie skryptów do opakowywania narzędzi infrastruktury” na stronie 327.

Oczywiście to tylko przykład. Ludzie różnie organizują swoje projekty i uwzględniają wiele innych rzeczy, których tutaj nie pokazałem.

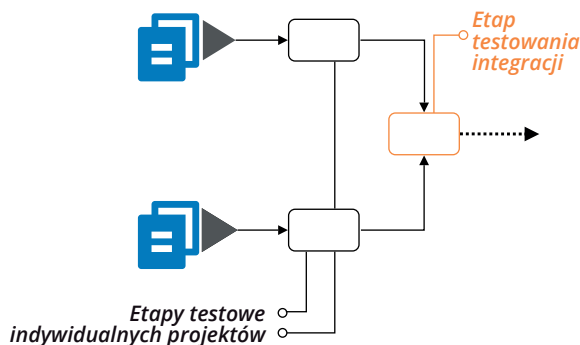
Kluczowym wnioskiem jest zalecenie, aby pliki specyficzne dla projektu były trzymane razem z projektem. W ten sposób, gdy ktoś wyewidencjonuje jakąś wersję projektu, będzie wiedział, że kod infrastruktury, testy i dostarczanie mają tę samą wersję, więc powinny ze sobą współpracować. Jeśli testy są przechowywane w oddzielnym projekcie, łatwo się z nimi rozminąć, uruchamiając ich złą wersję dla testowanego kodu.

Jednak niektóre testy, konfiguracja lub inne pliki mogą nie być specyficzne dla jednego tylko projektu. Jak sobie z tym radzić?

Testy wielu projektów

Testowanie progresywne (patrz „Testowanie progresywne” na stronie 109) polega na testowaniu każdego projektu oddzielnie przed przetestowaniem go po zintegrowaniu z innymi projektami, jak to jest pokazane na rysunku 18-7.

Kod testu do uruchomienia na poszczególnych etapach dla każdego projektu można dla wygody umieścić razem z tym projektem. Ale co z kodem testu dla etapu integracji? Można umieścić ten kod w jednym z projektów albo utworzyć oddzielny projekt dla testów integracji.



Rysunek 18-7 Testowanie projektów oddzielnie, a następnie razem

Trzymanie testów integracji w ramach projektu

W wielu przypadkach, w których ma miejsce integracja wielu projektów, jeden projekt jest oczywistym punktem wejścia dla określonych rodzajów testów. Na przykład wiele testów funkcjonalnych łączy się z usługą frontendową, aby potwierdzić działanie całego systemu. Jeśli komponent backendowy, taki jak baza danych, nie jest skonfigurowany poprawnie, usługa frontendowa nie może się z nim połączyć, więc test kończy się niepowodzeniem.

W takich przypadkach kod testu integracji może być trzymany dla wygody z projektem udostępniającym usługę frontendową. Najprawdopodobniej kod testu jest sprzężony z tą usługą, na przykład musi znać nazwę hosta oraz port dla tej usługi.

Należy oddzielać te testy od testów uruchamianych na wcześniejszych etapach dostarczania – na przykład podczas testowania z użyciem atrap. Każdy zestaw testów można trzymać w oddzielnym podfolderze projektu.

Testy integracji nadają się bardziej do projektów będących konsumentami innych projektów (patrz „Używanie warunków początkowych testu do obsługi zależności” na stronie 132), niż do projektu dostawcy. W przykładzie ShopSpinner występuje jeden projekt stosu, który definiuje instancje infrastruktury aplikacji, współdzieląc struktury sieciowe zdefiniowane w innym stosie.

Umieszczenie testów integracji w projekcie współdzielonego stosu sieci byłoby sprzeczne z kierunkiem zależności. Projekt sieci musi znać określone szczegóły stosu aplikacji i wszystkich innych stosów, które go używają, aby sprawdzić, czy integracja działa poprawnie. Stos infrastruktury aplikacji wie już o współdzielonym stosie sieci, więc trzymanie testów integracji z kodem stosu aplikacji pozwala uniknąć zapętlenia zależności między projektami.

Dedykowane projekty testów integracji

Alternatywnym podejściem jest utworzenie oddzielnego projektu do testów integracji, być może po jednym dla każdego etapu integracji. Takie podejście jest powszechne, gdy inny zespół jest właścicielem testów integracji, co przewiduje prawo Conwaya. Inne zespoły postępują tak w sytuacji, gdy nie jest oczywiste, który projekt pasuje do testów integracji.

Wersjonowanie może być wyzwaniem w sytuacji, gdy trzeba zarządzać zestawami testów integracji niezależnie od testowanego przez nie kodu. Ludziom może się mylić, którą wersję testów integracji mają uruchamiać dla danej wersji kodu systemu. Aby sobie z tym poradzić, trzeba koniecznie pisać i zmieniać testy razem z kodem, zamiast wykonywać te działania oddzielnie. Do tego trzeba zaimplementować sposób korelowania wersji projektu, na przykład używając podejścia fan-in, opisanego przy okazji wzorca integracji podczas dostarczania (patrz „Wzorzec: integracja projektów podczas dostarczania” na stronie 322).

Organizowanie kodu według koncepcji domeny

Kod pojedynczego projektu może składać się z wielu części. Projekt infrastruktury aplikacji w przykładzie ShopSpinner definiuje klaster serwerów i instancję bazy danych oraz struktury sieciowe i zasady bezpieczeństwa dla nich. Wiele zespołów definiuje struktury sieciowe i zasady bezpieczeństwa w ich własnych plikach, jak w przykładzie 18-2.

Przykład 18-2 Pliki źródłowe zorganizowane według technologii

```
└─ src/
   ├── cluster.infra
   ├── database.infra
   ├── load_balancer.infra
   ├── routing.infra
   ├── firewall_rules.infra
   └─ policies.infra
```

Plik *firewall_rules.infra* zawiera reguły zapory dla maszyn wirtualnych utworzonych w *cluster.infra*, jak również reguły dla instancji bazy danych zdefiniowanej w *database.infra*.

Taki sposób organizacji kodu koncentruje się na elementach funkcjonalnych, a nie sposobie ich wykorzystania. Często łatwiej jest zrozumieć, pisać, zmieniać i utrzymywać kod powiązanych elementów, gdy są one w tym samym pliku. Wyobraźmy sobie z jednej strony plik z trzydziestoma regułami dostępu do ośmiu różnych usług, a z drugiej plik definiujący jedną usługę i dotyczące jej trzy reguły zapory.

Taka koncepcja jest zgodna z zasadą projektowania mówiącą o bazowaniu na pojęciach dziedzinowych, a nie technicznych (patrz „Projektowanie komponentów opartych na pojęciach dziedzinowych, a nie technicznych” na stronie 248).

Organizowanie plików z wartościami konfiguracyjnymi

W rozdziale 7 opisałem wzorzec plików konfiguracyjnych do zarządzania wartościami parametrów dla różnych instancji projektu stosu (patrz „Wzorzec: pliki konfiguracyjne stosu” na stronie 81). Opis sugerował dwa różne sposoby organizacji plików konfiguracyjnych dla poszczególnych środowisk w wielu projektach. Jeden sposób polega na przechowywaniu ich razem z odpowiednim projektem:

```
├─ application_infra_stack/  
│   ├── src/  
│   └─ environments/  
│       ├── test.properties  
│       ├── staging.properties  
│       └─ production.properties  
└─ shared_network_stack/  
    ├── src/  
    └─ environments/  
        ├── test.properties  
        ├── staging.properties  
        └─ production.properties
```

Drugi sposób polega na utworzeniu oddzielnego projektu z konfiguracją wszystkich stosów, zorganizowanego według środowisk:

```
├─ application_infra_stack/  
│   └─ src/  
│       └─ shared_network_stack/  
│           └─ src/  
└─ configuration/  
    ├── test/  
    │   ├── application_infra.properties  
    │   └─ shared_network.properties  
    ├── staging/  
    │   ├── application_infra.properties  
    │   └─ shared_network.properties  
    └─ production/  
        ├── application_infra.properties  
        └─ shared_network.properties
```

Przechowywanie wartości konfiguracyjnych z kodem projektu oznacza połączenie uogólnionego kodu wielokrotnego użytku (zakładając, że stos jest wielokrotnego użytku, co zostało opisane w podrozdziale „Wzorzec: stos wielokrotnego użytku” na stronie 65) ze szczegółami konkretnych instancji. W wersji idealnej, zmiana konfiguracji środowiska nie powinna wymagać modyfikacji projektu stosu.

Z drugiej strony zapewne łatwiej jest prześledzić i zrozumieć wartości konfiguracyjne, gdy znajdują się blisko projektów, z którymi są związane, niż w przypadku ulokowania ich w monolitycznym projekcie konfiguracji. Jak zwykle istotnym czynnikiem jest dopasowanie i przypisanie prawa własności do zespołu. Oddzielenie kodu infrastruktury od jego konfiguracji może zniechęcać do przejścia prawa własności do jednego i drugiego i wzięcia za nie odpowiedzialności.

Zarządzanie kodem infrastruktury i aplikacji

Czy kod aplikacji i kod infrastruktury powinny być trzymane w oddzielnych repozytoriach, czy razem? Różne osoby mogą uznać każdy wariant za prawidłowy. Właściwa odpowiedź zależy od struktury danej organizacji i prawa własności.

Zarządzanie kodem infrastruktury i kodem aplikacji w oddzielnych repozytoriach jest odpowiednie dla modelu operacyjnego, w którym budowę infrastruktury i aplikacji oraz zarządzaniem nimi zajmują się odrębne zespoły. Ale jeśli zespoły zajmujące się aplikacjami odpowiadają za infrastrukturę, zwłaszcza infrastrukturę specyficzną dla ich aplikacji, pojawia się już problem.

Rozdzielenie kodu tworzy barierę poznawczą, nawet jeśli członkowie zespołu aplikacji odpowiadają za elementy infrastruktury związane z ich aplikacją.

Jeśli ten kod jest w innym repozytorium niż to, którego najczęściej używają w swojej pracy, nie będą mieli takiego samego komfortu zagłębiając się w nim. Tak jest szczególnie wtedy, gdy jest to kod, który słabiej znają i dodatkowo jest on zmieszany z kodem infrastruktury dla innych części systemu.

Kod infrastruktury ulokowany w obszarze bazy kodu należącym do zespołu budzi mniejsze obawy. Członkowie zespołu nie mają uczucia, że jakaś zmiana może popsuć czyjąś aplikację czy nawet podstawowe części infrastruktury.



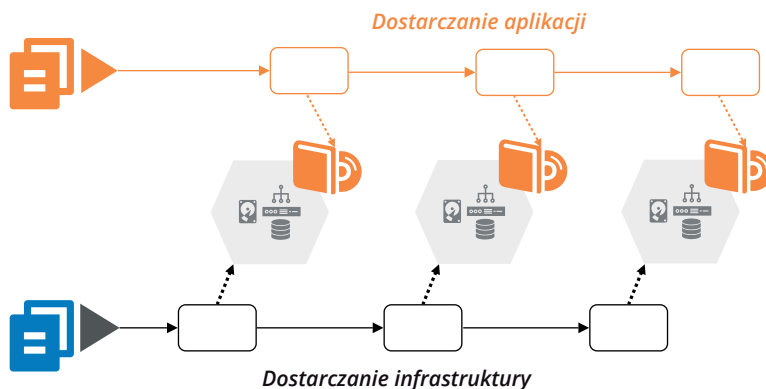
DevOps i struktury zespołów

Ruch DevOps zachęca organizacje do eksperymentowania z alternatywami dla tradycyjnego podziału na programowanie i eksploatację. Więcej szczegółowych przemyśleń na temat strukturyzacji zespołów aplikacji i infrastruktury można znaleźć w pracy Matthew Skeltona i Manuela Paisa na stronie Team Topologies².

² <https://teamtopologies.com>

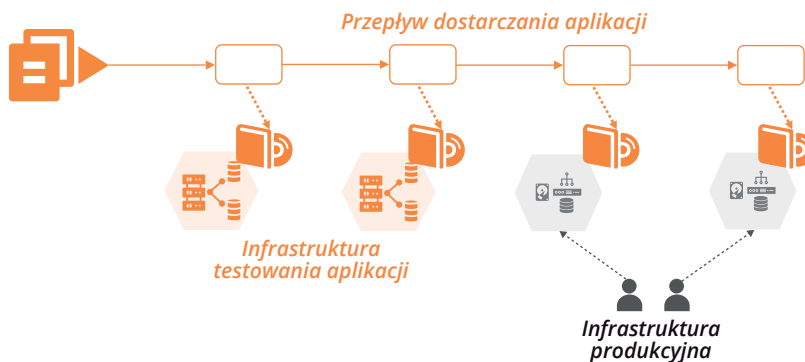
Dostarczanie infrastruktury i aplikacji

Bez względu na to, czy kod aplikacji i kod infrastruktury są zarządzane razem czy nie, na koniec i tak są wdrażane do tego samego systemu³. Zmiany kodu infrastruktury powinny być integrowane i testowane z aplikacjami w trakcie całego procesu dostarczania aplikacji (rysunek 18-8).



Rysunek 18-8 Dostarczanie aplikacji do środowisk zarządzanych przez kod infrastruktury

W przeciwieństwie do tego wiele organizacji ma starodawny pogląd na infrastrukturę produkcyjną jako oddzielny silos. Dość często jeden zespół jest właścicielem infrastruktury produkcyjnej, w tym środowiska przejściowego i przedprodukcyjnego, ale nie odpowiada za środowisko deweloperskie ani testowe (rysunek 18-9).



Rysunek 18-9 Oddzielna własność infrastruktury produkcyjnej

³ Pytania – i wzorce – dotyczące tego, kiedy integrować projekty, odnoszą się do integracji aplikacji z infrastrukturą. Więcej informacji na ten temat jest w podrozdziale „Integrowanie projektów” na stronie 317.

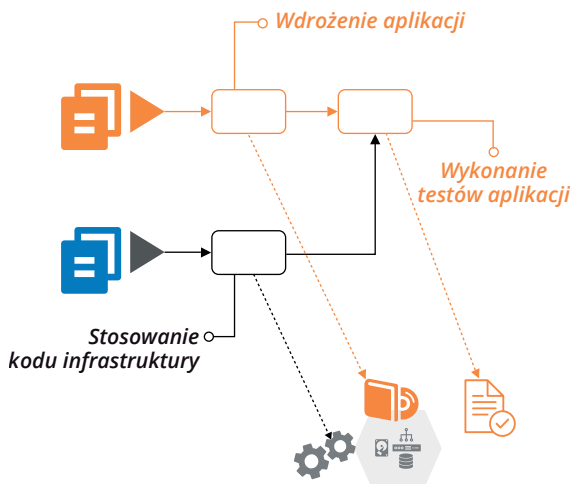
Taka separacja utrudnia dostarczanie aplikacji, a także dokonywanie zmian infrastruktury. Zespoły nie wykrywają konfliktów albo rozbieżności między dwoma częściami systemu aż do późnego etapu procesu dostarczania. Jak zostało wyjaśnione w rozdziale 8, ciągła integracja i testowanie wszystkich części systemu podczas pracy nad zmianami jest najskuteczniejszym sposobem zapewnienia wysokiej jakości i niezawodnego dostarczania.

Dlatego strategia dostarczania powinna zapewniać wprowadzanie zmian w kodzie infrastruktury w obrębie wszystkich środowisk. Jest kilka opcji przepływu zmian infrastruktury.

Testowanie aplikacji razem z infrastrukturą

Jeśli zmiany infrastruktury są dostarczane w ramach ścieżki dostarczania aplikacji, można wykorzystać zautomatyzowane testy aplikacji. Na każdym etapie, po zastosowaniu zmiany infrastruktury, należy wyzwolić etap testowania aplikacji (patrz rysunek 18-10).

Metoda testowania progresywnego („Testowanie progresywne” na stronie 109) wykorzystuje testy aplikacji do testowania integracji. Wersje aplikacji i infrastruktury mogą być powiązane ze sobą i przechodzić przez resztę przepływu dostarczania zgodnie z wzorcem integracji podczas dostarczania (patrz „Wzorzec: integracja projektów podczas dostarczania” na stronie 322). Można też wypychać zmiany infrastruktury do dalszych środowisk bez integracji z jakimikolwiek przeprowadzonymi zmianami aplikacji, przy użyciu integracji podczas stosowania (patrz „Wzorzec: integracja projektów podczas stosowania” na stronie 324).



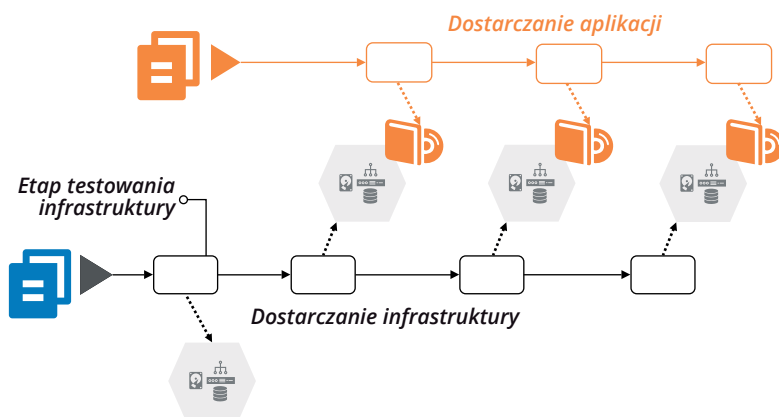
Rysunek 18-10 Wykonywanie testów aplikacji w momencie zmiany infrastruktury

Wypychanie zmian aplikacji i infrastruktury tak szybko, jak to tylko możliwe, to rozwiązanie idealne. Ale w praktyce nie zawsze można zapobiec problemom towarzyszącym każdemu rodzajowi zmian w organizacji. Na przykład, jeśli interesariusze wymagają

głębszego przeglądu zmian w aplikacji widocznej dla użytkownika, może być konieczne szybsze wypychanie rutynowych zmian infrastruktury. W przeciwnym razie proces wydawania aplikacji może zostać uwikłany w procedurę pilnych zmian, takich jak poprawki zabezpieczeń czy drobne zmiany dotyczące aktualizacji konfiguracji.

Testowanie infrastruktury przed integracją

Ryzyko stosowania kodu infrastruktury do współdzielonych środowisk programowania i testowania polega na tym, że uszkodzenie tych środowisk ma wpływ także na inne zespoły. Dlatego dobrze jest mieć oddzielnie etapy i środowiska dostarczania do testowania kodu infrastruktury, przed ich promocją do środowisk współdzielonych (zobacz rysunek 18-11).



Rysunek 18-11 Etap testowania infrastruktury

Taki pomysł jest specyficzną implementacją testowania progresywnego („Testowanie progresywne” na stronie 109) i integracji projektów podczas dostarczania („Wzorzec: integracja projektów podczas dostarczania” na stronie 322).

Używanie kodu infrastruktury do wdrażania aplikacji

Kod infrastruktury definiuje rzeczy, które mają trafić do serwerów. Wdrożenie aplikacji oznacza wstawienie tych rzeczy do serwerów. Z tego względu może wydawać się sensowne napisanie kodu infrastruktury automatyzującego proces wdrażania aplikacji. W praktyce mieszanie kwestii wdrożenia aplikacji i konfiguracji infrastruktury powoduje bałagan. Interfejs między aplikacją i infrastrukturą powinien być prosty i przejrzysty.

Systemy pakowania systemów operacyjnych, takie jak RPM, pliki `.deb` i pliki `.msi` mają dobrze zdefiniowany interfejs pakowania i wdrażania aplikacji. Kod infrastruktury może wskazywać plik pakietu do wdrożenia, a następnie pozwalać narzędziu do wdrożenia na kontynuację.

Problemy się pojawiają, gdy wdrożenie aplikacji obejmuje wiele czynności, a zwłaszcza gdy obejmuje wiele ruchomych części. Na przykład napisałem kiedyś książkę kucharską Chefa do wdrażania w linuxowych maszynach wirtualnych aplikacji Java napisanych przez mój zespół przy użyciu środowiska Dropwizard⁴. Książka kucharska musiała:

1. Pobrać nową wersję aplikacji i rozpakować ją do nowego folderu.
2. Zatrzymać proces poprzedniej wersji aplikacji, jeśli był wykonywany.
3. Zaktualizować pliki konfiguracyjne, jeśli trzeba.
4. Zaktualizować łącze symboliczne, aby wskazywało aktualną wersję aplikacji w nowym folderze.
5. Uruchomić skrypty migracji schematu bazy danych⁵.
6. Uruchomić proces dla nowej wersji aplikacji.
7. Sprawdzić, czy nowy proces działa poprawnie.

Powyższa książka kucharska sprawiała problemy zespołowi, ponieważ czasami nie potrafiła wykryć, że poprzedni proces nie został zakończony albo że nowy proces uległ awarii minutę po jego uruchomieniu.

Zasadniczo był to skrypt proceduralny w ramach deklaratywnej bazy kodu infrastruktury. Sytuacja poprawiła się po podjęciu decyzji o pakowaniu naszych aplikacji jako RPM, co oznaczało, że mogliśmy używać narzędzi i skryptów przeznaczonych specjalnie do wdrażania i uaktualniania aplikacji. Napisałem testy dla naszego procesu tworzenia pakietów RPM, które nie korzystały z reszty naszej bazy kodu Chefa, więc mogliśmy zgłębiać konkretne problemy, z powodu których wdrożenie bywało zawodne.

Inny problem związany z używaniem kodu infrastruktury do wdrażania aplikacji dotyczy sytuacji, w której proces wdrożenia wymaga orkiestracji wielu elementów. Proces mojego zespołu działał dobrze, gdy aplikacje Dropwizard były wdrażane na jednym serwerze. Zmieniło się to, gdy zaczęliśmy używać wielu serwerów w celu równoważenia obciążenia aplikacji.

Nawet po przejściu na pakiety RPM książki kucharskie nie zarządzały kolejnością wdrażania na wielu serwerach. W efekcie klaster mógł wykonywać różne wersje podczas operacji wdrażania. Ponadto skrypt migracji bazy danych powinien być wykonywany tylko raz, więc musieliśmy zaimplementować mechanizm blokowania, aby zapewnić, że zostanie uruchomiony tylko przez proces wdrożenia na pierwszym serwerze.

Nasze rozwiązanie polegało na przeniesieniu operacji wdrożenia poza kod konfiguracji serwera, do skryptu wypychającego aplikacje do serwerów z centralnego miejsca wdrażania – naszego serwera budowy. Skrypt ten zarządzał kolejnością wdrażania na serwerach

⁴ <https://www.dropwizard.io>

⁵ Patrz „Using Migration Scripts in Database Deployments” (https://oreil.ly/S_jbC) od Red Gate.

i migracjami schematu bazy danych, implementując wdrażanie bez przestojów poprzez modyfikowanie konfiguracji równoważenia obciążenia dla toczącej się aktualizacji⁶.

Rozproszone, natywne aplikacje chmurowe zwiększają problem orkiestracji wdrożeń aplikacji. Orkiestrowanie zmian w dziesiątkach, setkach, a nawet tysiącach instancji aplikacji może być naprawdę skomplikowane. Zespoły używają narzędzi do wdrażania, takich jak Helm⁷ i Octopus Deploy⁸, aby zdefiniować wdrożenie grup aplikacji. Narzędzia te wymuszają rozdzielenie zagadnień, koncentrując się na wdrożeniu zestawu aplikacji i pozostawiając udostępnienie bazowego klastra innym częściom bazy kodu.

Jednak najbardziej niezawodną strategią wdrażania aplikacji jest dbanie o luźne powiązanie każdego elementu. Im łatwiej i bezpieczniej można wdrożyć jakąś zmianę niezależnie od pozostałych zmian, tym bardziej niezawodny jest cały system.

Podsumowanie

Infrastruktura jako kod, jak sama nazwa sugeruje, oznacza sterowanie architekturą, jakością i możliwością zarządzania infrastrukturą systemu z poziomu bazy kodu. Dlatego baza kodu musi być zorganizowana i zarządzana zgodnie z wymaganiami biznesowymi i architekturą systemu. Musi trzymać się zasad i praktyk inżynierskich, zapewniających efektywność zespołu.

6 W celu wprowadzenia zmian schematu bazy danych bez przestoju użyliśmy wzorca powiększania i zmniejszania (<https://oreil.ly/RUu61>).

7 <https://helm.sh>

8 <https://octopus.com>

Dostarczanie kodu infrastruktury

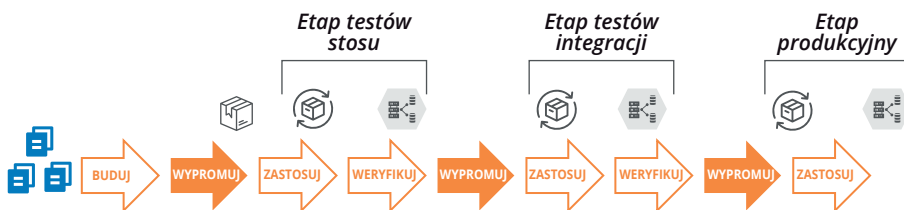
Cykl życia dostarczania oprogramowania jest ważnym pojęciem w naszej branży. Proces dostarczania infrastruktury wyglądał inaczej w epoce żelaza, kiedy mógł być mniej rygorystyczny. Często zmiana infrastruktury produkcyjnej odbywała się bez wcześniejszego przetestowania; jak na przykład wymiana sprzętu.

Ale używanie kodu do definiowania infrastruktury stwarza okazję zarządzania zmianami za pomocą bardziej kompleksowego procesu. Odtwarzanie w środowisku programowania modyfikacji ręcznie zbudowanego systemu, na przykład zmiany wielkości pamięci RAM w serwerze, może wydawać się mało sensowne. Ale zaimplementowanie zmiany w kodzie pozwala łatwo dostarczyć ją po ścieżce do produkcji za pomocą potoku (patrz „Potoki dostarczania infrastruktury” na stronie 113). Takie rozwiązanie umożliwia nie tylko wychwycenie problemu dotyczącego samej zmiany, co jest raczej niezbyt istotne (zwiększenie pamięci RAM niesie małe ryzyko popsucia czegośkolwiek), ale pozwala dodatkowo wykryć problemy z samym procesem wprowadzania zmian. Ponadto gwarantuje, że wszystkie środowiska na ścieżce do produkcji są skonfigurowane w sposób spójny.

Dostarczanie kodu infrastruktury

Metafora potoku opisuje przepływ zmiany w kodzie infrastruktury od osoby wprowadzającej tę zmianę do instancji produkcyjnych. Działania wymagane przez ten proces dostawy wpływają na sposób organizacji bazy kodu.

Potok dostarczania kolejnych wersji kodu obejmuje wiele rodzajów działań, w tym budowanie, promowanie, stosowanie i sprawdzanie poprawności. Każdy etap potoku sam może składać się z wielu działań, jak to jest pokazane na rysunku 19-1.



Rysunek 19-1 Fazy dostarczania projektu kodu infrastruktury

Budowanie

Obejmuje przygotowanie wersji kodu do użycia i udostępnienie jej innym fazom. Budowanie jest realizowane w potoku zwykle raz i powtarzane przy każdej zmianie kodu źródłowego.

Promowanie

Obejmuje przenoszenie wersji kodu między etapami dostarczania, co zostało opisane w podrozdziale „Testowanie progresywne” na stronie 109. Na przykład po pomyślnym przejściu wersji projektu stosu przez etap testowania można promować tę wersję na znak, że jest gotowa do etapu testów integracji systemu.

Stosowanie

Obejmuje uruchomienie odpowiednich narzędzi do zastosowania kodu w odpowiedniej instancji infrastruktury. Instancją tą może być środowisko dostarczania używane do wykonywania testów lub instancja produkcyjna.

Podrozdział „Potoki dostarczania infrastruktury” na stronie 113 zawiera więcej szczegółów na temat projektowania potoku. Tutaj skoncentrujemy się na budowaniu i promowaniu kodu.

Budowanie projektu infrastruktury

Budowanie projektu infrastruktury polega na przygotowaniu kodu do użycia. Oto niektóre działania, które mogą być wykonywane w ramach tego etapu:

- Pobieranie zależności etapu budowania, takich jak biblioteki, w tym również pochodzących z innych projektów w bazie kodu oraz z bibliotek zewnętrznych
- Rozwiązywanie konfiguracji etapu budowania, na przykład pobieranie wartości konfiguracyjnych współdzielonych przez wiele projektów
- Kompilowanie i przekształcanie kodu, takie jak generowanie plików konfiguracyjnych z szablonów
- Uruchamianie testów, w tym testów offline i online („Etapy testowania offline dla stosów” na stronie 125 oraz „Etapy testowania online dla stosów” na stronie 128)
- Przygotowywanie kodu do użycia, nadawanie mu formatu używanego przez odpowiednie narzędzie do infrastruktury do jego zastosowania
- Udostępnianie kodu do użycia

Istnieje kilka różnych sposobów przygotowania kodu infrastruktury i udostępnienia go do użycia. Niektóre narzędzia bezpośrednio obsługują konkretne sposoby, takie jak standardowy format pakietu artefaktów lub repozytorium. Inne pozostawiają to zespołom, które mogą zaimplementować własny sposób dostarczania kodu.

Pakowanie kodu infrastruktury jako artefaktu

W przypadku niektórych narzędzi „przygotowanie kodu do użycia” obejmuje połączenie plików w pakiet o określonym formacie, czyli artefakt. Taki proces jest typowy dla języków programowania ogólnego przeznaczenia, jak Ruby (gemy), JavaScript (NPM) i Python (pakiety Pythona używane z instalatorem pip). Inne formaty pakietów do instalowania plików i aplikacji dla konkretnych systemów operacyjnych to *.rpm*, *.deb*, *.msi* i NuGet (Windows).

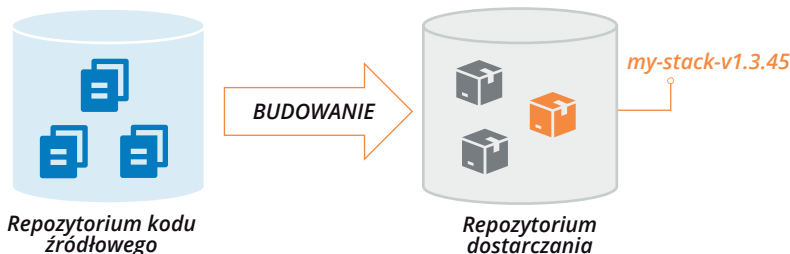
Niewiele narzędzi infrastruktury stosuje format pakietu dla swoich projektów kodu. Ale niektóre zespoły budują w takiej sytuacji własne artefakty, grupując kod stosu lub kod serwera w plikach ZIP lub „tarballach” (archiwach tar skompresowanych za pomocą gzip). Niektóre zespoły wykorzystują formaty pakowania systemu operacyjnego, tworząc pliki RPM, rozpakowujące na przykład pliki Chef Cookbook na serwerze. Jeszcze inne zespoły tworzą obrazy Docker, zawierające kod projektu stosu razem z wersją wykonywalną narzędzia stosu.

Pozostałe zespoły nie pakują kodu infrastruktury do artefaktu, zwłaszcza w przypadku narzędzi, które nie mają swojego natywnego formatu pakowania. Ostateczna decyzja zależy od przyjętego sposobu publikacji, udostępniania i używania kodu, na co ma wpływ rodzaj używanego repozytorium.

Używanie repozytorium do dostarczania kodu infrastruktury

Zespoły wykorzystują repozytorium kodu źródłowego do przechowywania kodu źródłowego swojej infrastruktury i zarządzania jego zmianami. Często mają oddzielne repozytorium do przechowywania kodu, który jest już gotowy do dostarczenia do środowisk i instancji. Ale niektóre zespoły, jak zobaczymy, używają tego samego repozytorium do obu celów.

Koncepcyjnie etap budowania rozdziela te dwa rodzaje repozytorium. Najpierw pobiera kod z repozytorium kodu źródłowego, potem składa go, a następnie publikuje w repozytorium dostarczania (patrz rysunek 19-2).



Rysunek 19-2 Etap budowania obejmuje publikację kodu w repozytorium dostarczania

Repozytorium dostarczania zawiera zwykle wiele wersji kodu danego projektu. Opisane w skrócie fazy promocji oznaczają odpowiednio wersję kodu projektu, aby wskazać etap,

na którym się już znajduje; na przykład czy jest to wersja gotowa do testów integracji, czy do produkcji.

Działanie „zastosuj” pobiera wersję kodu projektu z repozytorium dostarczania i stosuje ją do konkretnej instancji, takiej jak środowisko SIT lub środowisko PROD.

Istnieje kilka różnych sposobów implementowania repozytorium dostarczania. Dany system może wykorzystywać różne implementacje repozytorium dla różnych typów projektów. Na przykład może używać repozytorium konkretnego narzędzia, takiego jak Chef Server, dla projektów wykorzystujących to narzędzie. Ten sam system może używać ogólnej usługi przechowywania plików, na przykład zasobnika S3, dla projektów wykorzystujących takie narzędzie, jak Packer, które nie dysponuje formatem pakietu ani wyspecjalizowanym repozytorium.

Istnieje kilka rodzajów implementacji repozytorium dostarczania kodu.

Wyspecjalizowane repozytorium artefaktów

Większość formatów pakietów omówionych w poprzednim podrozdziale oferuje gotowy produkt lub usługę repozytorium pakietów albo standard możliwy do zaimplementowania przez różne produkty i usługi. Istnieje wiele gotowych repozytoriów w postaci produktów lub hostowanych usług dla plików *.rpm*, *.deb*, *.gem* i *.npm*.

Niektóre produkty z repozytoriami, takie jak Artifactory¹ i Nexus², obsługują wiele formatów pakietów. Zespoły w organizacjach, które je stosują, przechowują w nich czasem swoje artefakty, takie jak pliki ZIP i tarballe. Wiele platform chmurowych obejmuje wyspecjalizowane repozytoria artefaktów, takie jak magazyny obrazów serwerów.

Projekt ORAS (OCI Registry As Storage)³ oferuje sposób wykorzystania repozytoriów artefaktów zaprojektowanych pierwotnie dla obrazów Dockera jako repozytoriów dla dowolnego typu artefaktów.

Jeśli repozytorium artefaktów obsługuje tagi lub etykiety, można wykorzystywać je do promocji. Na przykład, aby promować artefakt projektu stosu do etapu testów integracji systemu, można dodać do niego tag `SIT_STAGE=true` lub `Stage=SIT`.

Innym rozwiązaniem jest utworzenie wielu repozytoriów w ramach serwera repozytoriów, z jednym repozytorium dla każdego etapu dostarczania. W celu promocji jakiegoś artefaktu należy wówczas skopiować lub przenieść ten artefakt do odpowiedniego repozytorium.

Repozytorium specyficzne dla narzędzia

Wiele narzędzi infrastruktury dysponuje wyspecjalizowanym repozytorium, które nie wykorzystuje spakowanych artefaktów. Zamiast tego trzeba uruchamiać narzędzie, które przesyła kod projektu na serwer, przypisując do niego wersję. Działa to niemal identycznie jak wyspecjalizowane repozytorium artefaktów, ale bez pliku pakietu.

¹ <https://oreil.ly/k3xmX>

² https://oreil.ly/mSh_i

³ <https://oreil.ly/CGFsM>

Przykłady to Chef Server⁴ (obsługa samodzielna), Chef Community Cookbooks⁵ (wersja publiczna) i Terraform Registry⁶ (moduły publiczne).

Repozytorium w postaci ogólnego magazynu plików

Wiele zespołów, zwłaszcza tych, które używają własnych formatów do zapisywania projektów kodu infrastruktury do dostarczania, przechowuje je w magazynie plików ogólnego przeznaczenia, wykorzystując do tego jakiś produkt lub usługę. Może to być serwer plików, serwer WWW albo usługa przechowywania obiektów.

Takie repozytoria nie zapewniają specjalnych funkcji do obsługi artefaktów, takich jak numerowanie wersji wydań. Trzeba więc samodzielnie przypisywać numery wersji, umieszczając je choćby w nazwie pliku (na przykład *my-stack-v1.3.45.tgz*). W celu promocji artefaktu można skopiować go lub dowiązać do folderu dla odpowiedniego etapu dostarczania.

Dostarczanie kodu z repozytorium kodu źródłowego

W sytuacji, gdy kod źródłowy jest już w repozytorium kodu źródłowego, a wiele narzędzi kodu infrastruktury nie dysponuje formatem pakietu i łańcuchem narzędzi do traktowania kodu jako wydania, wiele zespołów po prostu stosuje ten kod do środowisk bezpośrednio z repozytorium kodu źródłowego.

Stosowanie kodu z głównej gałęzi (konaru) do wszystkich środowisk utrudniałoby zarządzanie różnymi wersjami tego kodu. Dlatego większość zespołów, które tak działają, używa gałęzi, często utrzymując oddzielną gałąź dla każdego środowiska. Promowanie wersji kodu do jakiegoś środowiska odbywa się przez scalenie jej z odpowiednią gałęzią. GitOps łączy tę praktykę z ciągłą synchronizacją (więcej szczegółów można znaleźć w podrozdziałach „Ciągłe stosowanie kodu” na stronie 342 i „GitOps” na stronie 343).

Używanie gałęzi do promowania kodu może zacierać różnicę między edytowaniem kodu i jego dostarczaniem. Podstawowa zasada CD mówi, aby nigdy nie zmieniać kodu po etapie budowania⁷. Chociaż zespół może podjąć decyzję o zakazie edytowania kodu w gałęziach, często trudno jest utrzymać dyscyplinę.

Integrowanie projektów

Jak wspomniałem w podrozdziale „Organizowanie projektów i repozytoriów” na stronie 297, między projektami w bazie kodu często występują zależności. Pojawia się więc kolejne pytanie, kiedy i jak łączyć różne wersje projektów zależnych od siebie.

Jako przykład rozważmy kilka projektów w bazie kodu zespołu ShopSpinner. Są w nim dwa projekty stosu. Jeden z nich, *application_infrastructure-stack*, definiuje

⁴ <https://oreil.ly/HWYph>

⁵ <https://oreil.ly/lTyBp>

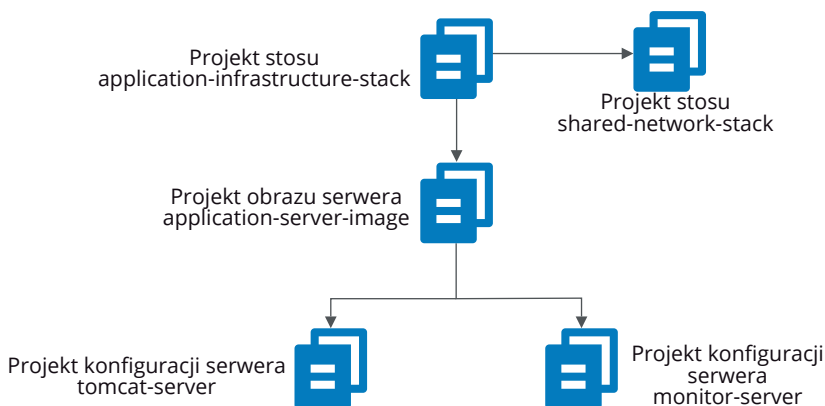
⁶ <https://oreil.ly/J85R1>

⁷ „Only build packages once”, wzorce CD Jeza Humble’a (<https://oreil.ly/yXQ9j>).

infrastrukturę specyficzną dla aplikacji, w tym pulę maszyn wirtualnych i reguły równoważenia obciążenia dla ruchu aplikacji. Drugi projekt stosu, `shared_network_stack`, definiuje typową sieć współdzieloną przez wiele instancji `application_infrastructure-stack`, w tym bloki adresów (VPC i podsieci) oraz reguły zapory zezwalające na ruch do serwerów aplikacji.

Zespół ma również dwa projekty konfiguracji serwerów, `tomcat-server`, który konfiguruje i instaluje oprogramowanie serwera aplikacji oraz `monitor-server`, który robi to samo dla agenta monitorowania.

Piąty projekt infrastruktury, `application-server-image`, buduje obraz serwera używając modułów konfiguracji `tomcat-server` i `monitor-server` (rysunek 19-3).



Rysunek 19-3 Przykład zależności między projektami infrastruktury

Projekt `application_infrastructure-stack` tworzy swoją infrastrukturę w ramach struktur sieciowych utworzonych przez `shared_network_stack`. Ponadto wykorzystuje obrazy serwera zbudowane przez projekt `application-server-image` do utworzenia serwerów w swoim klastrze serwerów aplikacji. Z kolei `application-server-image` buduje obrazy serwera stosując definicje konfiguracji serwera w `tomcat-server` i `monitor-server`.

Gdy ktoś dokonuje zmiany w jednym z tych projektów kodu infrastruktury, powstaje nowa wersja kodu tego projektu. Ta wersja musi być zintegrowana z wersją każdego z pozostałych projektów. Wersje projektów można integrować podczas budowania, dostarczania lub stosowania.

Różne projekty danego systemu można integrować w różnych momentach, co zobaczymy na przykładzie ShopSpinner w opisach kolejnych wzorców.



Konsolidowanie i integrowanie

Składanie elementów programu komputerowego jest określane mianem *konsolidowania*⁸. Istnieją podobieństwa między konsolidowaniem oprogramowania a opisywanym w tym rozdziale integrowaniem projektów infrastruktury.

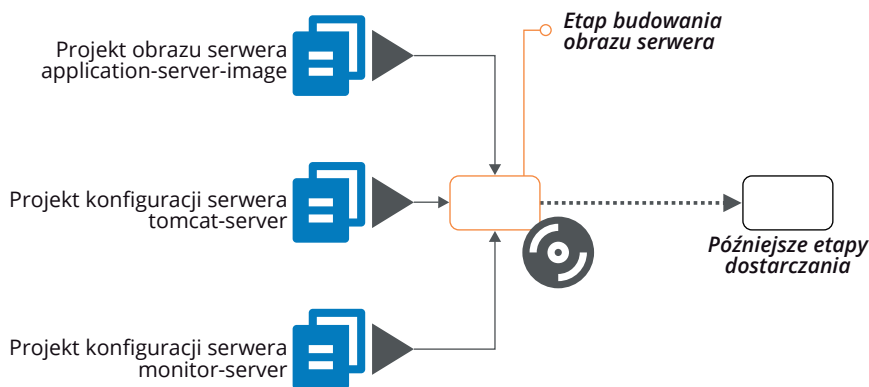
Biblioteki są *konsolidowane statycznie* podczas budowania pliku wykonywalnego, podobnie do integracji projektów podczas budowania. Biblioteki zainstalowane w maszynie są *konsolidowane dynamicznie* za każdym razem, gdy w pliku wykonywalnym dochodzi do wywołania, które ich dotyczy, co przypomina nieco integrację projektów podczas stosowania. W obu przypadkach w środowisku wykonawczym może nastąpić zmiana wersji dostawcy lub konsumenta.

Analogia nie jest idealna, ponieważ zmiana programu wpływa na *zachowanie się* wersji wykonywalnej, podczas gdy zmiana infrastruktury (mówiąc ogólnie) wpływa na *stan zasobów*.

Wzorzec: integracja projektów podczas budowania

Wzorzec *integracji projektów podczas budowania* dotyczy czynności budowania obejmującego wiele projektów. Wymaga to integracji zależności między tymi projektami i ustalenia wersji ich kodu.

Proces budowy obejmuje często budowanie i testowanie każdego składowego projektu przed zbudowaniem i przetestowaniem ich razem (patrz rysunek 19-4). Tym, co odróżnia ten wzorzec od innych, alternatywnych wzorców, jest utworzenie na koniec jednego artefaktu dla wszystkich projektów albo zestawu artefaktów wersjonowanych, promowanych i stosowanych jako grupa.



Rysunek 19-4 Przykład integracji projektów podczas budowania

⁸ <https://oreil.ly/sGoGY>

W tym przykładzie jeden etap budowy tworzy obraz serwera przy użyciu wielu projektów konfiguracji serwera.

Etap budowy może składać się z wielu kroków, takich jak budowanie i testowanie poszczególnych modułów konfiguracji serwera. Ale efekt końcowy – obraz serwera – składa się z kodu wszystkich projektów składowych.

Motywacja

Budowanie projektów razem umożliwia wczesne rozwiązywanie wszystkich problemów z zależnościami. Takie działanie zapewnia szybką informację zwrotną o konfliktach i pozwala uzyskać wysoki poziom spójności bazy kodu, od procesu dostarczania po produkcję. Kod projektu integrowany podczas budowania jest spójny przez cały cykl dostarczania. Ta sama wersja kodu jest stosowana na każdym etapie tego procesu aż do produkcji.

Zastosowanie

Użycie tego wzorca zamiast innych, alternatywnych, jest głównie kwestią upodobań. Zależy od tego, który zbiór argumentów za i przeciw bardziej do nas przemawia i na ile nasz zespół potrafi zarządzać złożonym budowaniem wielu projektów jednocześnie.

Konsekwencje

Integrowanie wielu projektów podczas budowania jest skomplikowane, zwłaszcza przy bardzo dużej ich liczbie. W zależności od implementacji budowy, może to prowadzić do wolniejszego uzyskiwania informacji zwrotnej.

Stosowanie integracji projektów podczas budowania na dużą skalę wymaga zaawansowanych narzędzi do orkiestracji budowy. Większe organizacje, które używają tego wzorca do dużych baz kodu, takie jak Google i Facebook, mają zespoły przeznaczone do utrzymywania własnych narzędzi.

Niektóre z dostępnych narzędzi pozwalają budować bardzo dużą liczbę projektów oprogramowania jednocześnie, o czym będzie mowa przy okazji implementacji. Ale takie podejście nie jest aż tak powszechne w branży, jak budowanie projektów oddzielnie, dlatego nie ma zbyt wielu narzędzi i materiałów pomocniczych.

Ponieważ projekty są budowane razem, granice między nimi są mniej wyraźne niż w przy pozostałych wzorcach. Może to prowadzić do silniejszego sprzężenia między projektami. Gdy ma to miejsce, może być trudno dokonać nawet małej zmiany bez wpływania na pozostałe części bazy kodu, a to wydłuża czas i zwiększa ryzyko zmian.

Implementacja

Przechowywanie wszystkich projektów do budowy oprogramowania w jednym repozytorium, nazywane często monorepo⁹, upraszcza budowanie ich jednocześnie, dzięki

⁹ <https://oreil.ly/6yVmG>

integracji wersjonowania ich kodu (patrz „Monorepo – jedno repozytorium, jedna budowa” na stronie 299).

Większość narzędzi do budowania oprogramowania, takich jak Gradle, Make, Maven, MSBuild i Rake, jest wykorzystywanych do orkiestracji budowy umiarkowanej liczby projektów jednocześnie. Wykonywanie budowy i testów bardzo dużej liczby projektów może wymagać długiego czasu.

Można przyspieszyć ten proces stosując równoległość wykonania, czyli budując i testując wiele projektów w oddzielnych wątkach, procesach, a nawet w sieci obliczeniowej. Ale do tego potrzebne są większe zasoby obliczeniowe.

Lepszym sposobem optymalizacji budowania na wielką skalę jest używanie grafu skierowanego do ograniczenia budowania i testowania tylko do tych części bazy kodu, które uległy zmianie. Jeśli zrobimy to dobrze, powinniśmy skrócić czas potrzebny do budowy i testów po zatwierdzeniu zmiany, tak że będzie to trwało niewiele dłużej niż budowanie projektów oddzielnie.

Istnieje kilka wyspecjalizowanych narzędzi do budowania zaprojektowanych z myślą o obsłudze bardzo dużych budów obejmujących wiele projektów. Większość z tych narzędzi została zainspirowana wewnętrznymi narzędziami, stworzonymi przez Google i Facebook. Należą do nich między innymi Bazel¹⁰, Buck¹¹, Pants¹² i Please¹³.

Powiązane wzorce

Alternatywą dla integracji wersji projektów podczas budowania jest robienie tego podczas dostarczania (patrz „Wzorzec: integracja projektów podczas dostarczania” na stronie 322) lub podczas stosowania (patrz „Wzorzec: integracja projektów podczas stosowania” na stronie 324).

Strategia zarządzania wieloma projektami w jednym repozytorium („Monorepo – jedno repozytorium, jedna budowa” na stronie 299), chociaż nie jest wzorcem, wspomaga ten wzorzec. Przykład użyty dla tego wzorca (rysunek 19-4) przedstawia zastosowanie kodu konfiguracji serwera podczas tworzenia obrazu serwera (patrz „Pieczenie obrazów serwera” na stronie 182). Wzorzec serwera niezmiennego („Wzorzec: serwer niezmienny” na stronie 189) to kolejny przykład w przeważającym stopniu integracji podczas budowania a nie dostarczania.

Chociaż metoda ta nie występuje w tej książce jako wzorzec, w wielu przypadkach zależności od bibliotek innych firm są rozwiązywane podczas budowania, poprzez pobranie ich i dołączenie do elementu dostarczanego. Różnica polega na tym, że takie zależności nie są budowane i testowane razem z używającym ich projektem. Jeśli zależności te pochodzą z innych projektów w ramach tej samej organizacji, to jest to przykład integracji projektów podczas dostarczania.

¹⁰ <https://bazel.build>

¹¹ <https://buck.build>

¹² <https://www.pantsbuild.org>

¹³ <https://please.build>



Czy to jest integracja projektów podczas budowania czy monorepo?

Większość opisów strategii monorepo¹⁴ dotyczącej organizacji bazy kodu dotyczy budowania wspólnie wszystkich projektów znajdujących się w repozytorium – to, co nazwałem *integracją projektów podczas budowania*. Postanowiłem nie nazywać tego wzorca *monorepo*, ponieważ nazwa ta oznacza używanie jednego repozytorium kodu, co jest jedną z opcji implementacji.

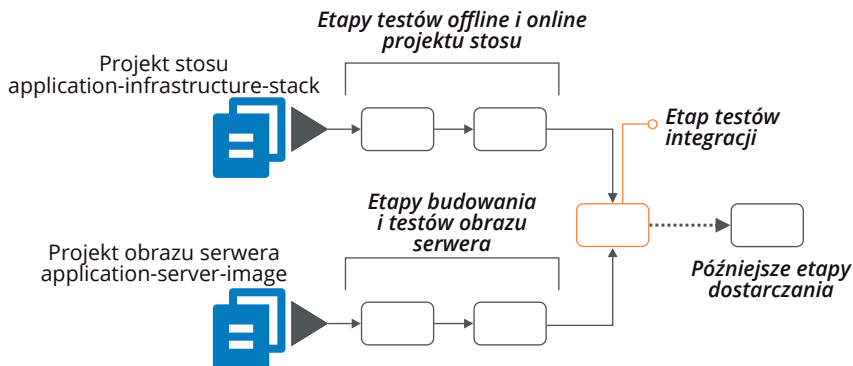
Znam zespoły, które zarządzają wieloma projektami w jednym repozytorium i nie budują ich razem. Choć zespoły te często nazywają swoją bazę kodu monorepo, nie stosują opisanego tutaj wzorca.

Z drugiej strony technicznie jest możliwe wywidencjonowanie projektów kodu z oddzielnych repozytoriów i budowanie ich razem. Pasuje to do opisanego tutaj wzorca, ponieważ oznacza integrację wersji projektów podczas budowania. Jednak takie działanie komplikuje korelowanie i śledzenie wersji kodów źródłowych używanych do budowy, na przykład w momencie debugowania problemów w fazie produkcji.

Wzorzec: integracja projektów podczas dostarczania

W przypadku wielu projektów z zależnościami między nimi *integracja projektów podczas dostarczania* polega na budowaniu i testowaniu każdego projektu oddzielnie, przed ich połączeniem. Takie podejście powoduje integrowanie wersji kodów później, niż ma to miejsce we wzorcu integracji podczas budowania. Po zbudowaniu i przetestowaniu projektów ich kod już wspólnie przechodzi przez resztę cyklu dostarczania.

Na przykład projekt `application-infrastructure-stack` w ShopSpinner definiuje klastr maszyn wirtualnych używając obrazu serwera zdefiniowanego w projekcie `application-server-image` (patrz rysunek 19-5).



Rysunek 19-5 Przykład integracji projektów podczas dostarczania

¹⁴ <https://oreil.ly/6yVmG>

Po wprowadzeniu zmiany w kodzie stosu infrastruktury potok dostarczania samodzielnie buduje i testuje projekt stosu, jak to zostało opisane w rozdziale 9.

Jeśli nowa wersja projektu stosu przejdzie pomyślnie te testy, przechodzi do fazy testów integracji, które polegają na sprawdzeniu integracji stosu z ostatnim obrazem serwera, który przeszedł pomyślnie własne testy. Ten etap jest punktem integracji dwóch projektów. Wersje tych projektów przechodzą następnie razem do dalszych etapów potoku.

Motywacja

Indywidualne budowanie i testowanie projektów przed ich integracją to jeden ze sposobów wymuszenia wyraźnych granic i słabego sprzężenia między projektami.

Wyobraźmy sobie na przykład, że członek zespołu ShopSpinner implementuje regułę zapory w `application-infrastructure-stack`, która otwiera port TCP zdefiniowany w pliku konfiguracyjnym `application-server-image`. Zespół pisze kod, który odczytuje numer portu bezpośrednio z pliku konfiguracyjnego. Ale po wypchnięciu tego kodu etap testów tego stosu kończy się niepowodzeniem, ponieważ plik konfiguracyjny z innego projektu nie jest dostępny dla agenta budowy.

To niepowodzenie jest pożyteczne. Ujawnia sprzężenie między dwoma projektami. Członek zespołu może zmienić swój kod i używać wartości parametru dla numeru portu do otwarcia, ustawiając tę wartość później (przy użyciu jednego z wzorców opisanych w rozdziale 7). Taki kod będzie łatwiejszy w utrzymaniu niż baza kodu z bezpośrednimi odwołaniami do plików w różnych projektach.

Zastosowanie

Integracja podczas dostarczania jest przydatna, gdy potrzebujemy wyraźnych granic między projektami w bazie kodu, ale nadal chcemy testować i dostarczać wersje poszczególnych projektów razem. Wzorzec jest trudny do skalowania dla dużej liczby projektów.

Konsekwencje

Integracja podczas dostarczania umieszcza złożone rozwiązywanie i koordynowanie różnych wersji różnych projektów w procesie dostarczania. Wymaga to wyrafinowanej implementacji dostarczania, takiej jak potok (zobacz „Usługi i oprogramowanie potoków dostarczania” na stronie 117).

Implementacja

Potoki dostarczania integrują różne projekty przy użyciu projektu potoku `fan-in`¹⁵. Etap, na którym dochodzi do połączenia różnych projektów, nazywany jest etapem `fan-in` lub etapem integracji projektów¹⁶.

¹⁵ <https://oreil.ly/ShOFm>

¹⁶ Wzorzec `fan-in` jest podobny do (a może jest to tylko inna nazwa) wzorca potoku `Aggregate Artifact` (<https://oreil.ly/uezro>).

Sposób integracji różnych projektów w ramach etapu zależy od typu tych projektów. W przykładzie projektu stosu wykorzystującego obraz serwera można byłoby zastosować kod stosu i przekazać odwołanie do odpowiedniej wersji obrazu. Zależności infrastruktury są pobierane z repozytorium dostarczania kodu (patrz „Używanie repozytorium do dostarczania kodu infrastruktury” na stronie 315).

Ten sam zbiór połączonych wersji projektów musi być stosowany na dalszych etapach procesu dostarczania. Są dwa typowe podejścia do realizacji tego zadania.

Jedną z metod jest zebranie kodu wszystkich projektów w jeden artefakt i użycie go na dalszych etapach. Na przykład, gdy są do zintegrowania i przetestowania dwa różne projekty stosu, etap integracji może spakować kod obu projektów do jednego archiwum i wypromować je do dalszych etapów potoku. Przepływ GitOps wykonałby scalenie projektów z gałęzią etapu integracji, a następnie scaliłby je z tej gałęzi z następnymi gałęziami przepływu.

Innym podejściem jest utworzenie pliku deskryptora z numerami wersji poszczególnych projektów. Na przykład:

```
descriptor-version: 1.9.1

stack-project:
  name: application-infrastructure-stack
  version: 1.2.34

server-image-project:
  name: application-server-image
  version: 1.1.1
```

Proces dostarczania traktuje plik deskryptora jak artefakt. Każdy etap, na którym stosowany jest kod infrastruktury, pobiera artefakt indywidualnego projektu z repozytorium dostarczania.

Trzecie podejście polega na oznaczaniu odpowiednich zasobów przy użyciu zagregowanego numeru wersji.

Powiązane wzorce

Wzorzec integracji projektów podczas budowania (patrz „Wzorzec: integracja projektów podczas budowania” na stronie 319) polega na integracji projektów na samym początku, a nie po wykonaniu już pewnych operacji dostarczania dla poszczególnych projektów. Wzorzec integracji projektów podczas stosowania powoduje integrację projektów na każdym etapie dostarczania, który wykorzystuje je razem, ale bez „blokady” wersji.

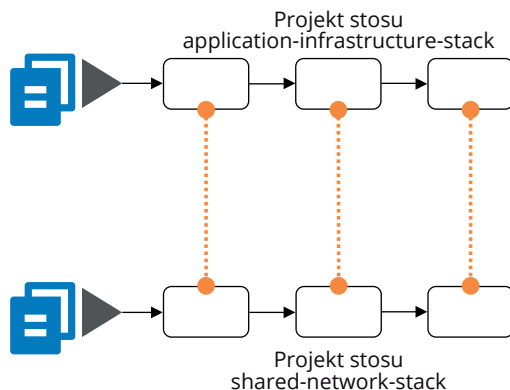
Wzorzec: integracja projektów podczas stosowania

Znany również jako: dostarczanie bez sprzężeń lub potoki bez sprzężeń.

Integracja projektów podczas stosowania polega na oddzielnym wypychaniu wielu projektów przez etapy dostarczania. W momencie dokonania zmiany kodu projektu potok

stosuje zaktualizowany kod do każdego środowiska na ścieżce dostarczania tego projektu. Wersja kodu projektu może być integrowana z wersjami różnych nadrzędnych i podrzędnych projektów w każdym z tych środowisk.

W przykładzie ShopSpinner projekt `application-infrastructure-stack` zależy od struktur sieciowych utworzonych przez projekt `shared-network-stack`. Każdy projekt ma własne etapy dostarczania, jak to jest pokazane na rysunku 19-6.



Rysunek 19-6 Przykład integracji projektów podczas ich stosowania

Integracja projektów odbywa się przez zastosowanie kodu `application-infrastructure-stack` do środowiska. Działanie to tworzy lub zmienia klastery serwerów, wykorzystujące struktury sieciowe (na przykład podsieci) należące do współdzielonej sieci.

Powyższa integracja następuje niezależnie od tego, która wersja stosu współdzielonej sieci jest aktualnie w danym środowisku. Tak więc integracja konkretnych wersji odbywa się oddzielnie za każdym razem, gdy dochodzi do zastosowania kodu.

Motywacja

Integrowanie projektów podczas stosowania minimalizuje sprzężenie między projektami. Różne zespoły mogą wypychać zmiany do swoich systemów do produkcji bez konieczności koordynowania ich i bez blokowania przez problemy związane ze zmianą projektów innych zespołów.

Zastosowanie

Taki poziom braku sprzężenia odpowiada organizacjom z autonomiczną strukturą zespołów. Pomaga również w przypadku wielkoskalowych systemów, gdzie koordynowanie wydań i dostarczanie ich zgodnie przez setki lub tysiące inżynierów jest niepraktyczne.

Konsekwencje

Wzorzec ten przenosi ryzyko zerwania zależności między projektami do etapu stosowania. Nie zapewnia spójności w całym potoku. Jeśli ktoś wypchnie zmianę jednego projektu

przez potok szybciej niż zmiany innych projektów, w środowisku produkcyjnym zostaną zintegrowane inne wersje niż miało to miejsce w środowiskach testowych.

Trzeba ostrożnie zarządzać interfejsami między projektami, aby zapewnić maksymalną kompatybilność różnych wersji po każdej stronie dowolnej zależności. Dlatego wzorzec ten wymaga bardziej złożonego projektowania, utrzymywania i testowania zależności oraz interfejsów.

Implementacja

Pod pewnymi względami projektowanie i implementowanie niesprzężonych budów i potoków z użyciem integracji podczas stosowania jest prostsze od innych, alternatywnych wzorców. Każdy potok buduje, testuje i dostarcza pojedynczy projekt.

W rozdziale 17 omówiłem integrowanie różnych stosów infrastruktury. Na przykład podczas stosowania projektu `application-infrastructure-stack`, musi nastąpić odwołanie do struktur sieciowych utworzonych przez `shared-network-stack`. Jak już wiemy, istnieją pewne techniki udostępniania identyfikatorów między stosami infrastruktury. Ponieważ nie ma gwarancji, która wersja kodu innego projektu została użyta w danym środowisku, zespoły muszą wyraźnie identyfikować zależności między projektami i traktować je jako kontrakty. Projekt `shared-network-stack` ujawnia identyfikatory struktur sieciowych, których mogą używać inne projekty. Musi ujawniać je w sposób ustandaryzowany, wykorzystując jeden z wzorców opisanych w rozdziale 17.

Jak wyjaśniłem w podrozdziale „Używanie warunków początkowych testu do obsługi zależności” na stronie 132, za pomocą warunków początkowych testu można testować każdy stos osobno. W przykładzie z `ShopSpinner` zespół chciałby testować projekt `application-infrastructure-stack` bez użycia instancji `shared-network-stack`. Stos sieci definiuje redundantną i złożoną infrastrukturę, która nie jest potrzebna do obsługi przypadków testowych. Dlatego konfiguracja testów zespołu może tworzyć okrojony zestaw sieciowy. Pozwala to również ograniczyć ryzyko ewolucji stosu aplikacji w kierunku przejścia szczegółów implementacji stosu sieci.

Zespół będący właścicielem projektu, od którego zależą inne projekty, może zaimplementować testy kontraktowe udowadniające, że jego kod spełnia oczekiwania. Projekt `shared-network-stack` może weryfikować, czy struktury sieciowe – podsieci – są tworzone i czy ich identyfikatory są ujawniane za pomocą mechanizmu używanego przez inne projekty do ich wykorzystania.

Należy pilnować, aby testy kontraktowe były wyraźnie oznaczone. Jeśli ktoś dokonania zmiany kodu, która spowoduje niepowodzenie testu, powinien rozumieć, że mógł popsuć inne projekty, a nie uznać, że musi jedynie zaktualizować test, aby dopasować go do zmiany.

Wiele organizacji uznaje za przydatne testowanie CDC (consumer-driven contract), czyli testowanie kontraktów sterowanych przez konsumenta¹⁷. W tym modelu zespół pracujący nad projektem *konsumenta*, zależnym od zasobów utworzonych w projekcie

¹⁷ <https://oreil.ly/cgPTj>

dostawcy, pisze testy wykonywane w potoku projektu dostawcy. Pomaga to zespołowi projektu dostawcy zrozumieć oczekiwania zespołu konsumenta.

Powiązane wzorce

Na przeciwnym biegunie do tego wzorca jest wzorec integracji projektów podczas budowania („Wzorec: integracja projektów podczas budowania” na stronie 319). Wzorec ten integruje projekty tylko raz, na początku cyklu dostarczania, a nie za każdym razem. Wzorec integracji projektów podczas dostarczania („Wzorec: integracja projektów podczas dostarczania” na stronie 322) również integruje projekty raz, ale w pewnym momencie cyklu dostarczania, a nie na jego początku.

Podrozdział „Smażenie instancji serwera” na stronie 181 ilustruje wykorzystanie wzorca integracji podczas stosowania do udostępniania serwera. Zależności, takie jak moduły konfiguracji serwera, są stosowane za każdym razem, gdy jest tworzona nowa instancja serwera, zazwyczaj z użyciem najnowszej wersji modułu serwera, która została promowana do odpowiedniego etapu.

Używanie skryptów do opakowywania narzędzi infrastruktury

Większość zespołów zarządzających kodem infrastruktury tworzy niestandardowe skrypty do orkiestracji i uruchamiania swoich narzędzi infrastruktury. Niektóre wykorzystują narzędzia do budowy oprogramowania, takie jak Make, Rake lub Gradle. Inne piszą skrypty w językach Bash, Python lub PowerShell. W wielu przypadkach ten pomocniczy kod staje się co najmniej tak skomplikowany, jak kod definiujący infrastrukturę, co powoduje, że zespoły spędzają większość czasu na jego debugowaniu i utrzymywaniu.

Zespoły mogą uruchamiać skrypty podczas etapu budowania, dostarczania lub stosowania. Często skrypty obsługują więcej niż jedną z tych faz projektu. Skrypty mogą realizować różne zadania, między innymi takie jak:

Konfigurowanie

Zbieranie wartości parametrów konfiguracyjnych, z ewentualnym ustaleniem ich hierarchii. Więcej na ten temat wkrótce.

Rozwiązywanie zależności

Rozwiązywanie i pobieranie bibliotek, dostawców i innego kodu.

Pakowanie

Przygotowywanie kodu do dostarczenia albo przez upakowanie go w postaci artefaktu, albo poprzez utworzenie gałęzi lub scalenie z nią.

Promowanie

Przenoszenie kodu z jednego etapu do drugiego – poprzez zastosowanie tagów, przeniesienie artefaktu, utworzenie gałęzi lub scalenie z nią.

Orkiestracja

Stosowanie różnych stosów i innych elementów infrastruktury w prawidłowej kolejności, na podstawie ich zależności.

Wykonywanie

Uruchamianie odpowiednich narzędzi infrastruktury, zbieranie argumentów wiersza polecenia i plików konfiguracyjnych zgodnie z instancją, do której jest stosowany kod.

Testowanie

Konfigurowanie i uruchamianie testów, łącznie z udostępnianiem warunków początkowych testu i danych, oraz zbieranie i publikowanie wyników.

Zbieranie wartości konfiguracyjnych

Zestawianie i rozwiązywanie wartości konfiguracyjnych może być jednym z bardziej złożonych zadań skryptu opakowującego. Rozważmy system, taki jak na przykład ShopSpinner, który obejmuje wiele środowisk dostarczania, wiele instancji produkcyjnych klienta i wiele komponentów infrastruktury.

Prosty, jednopoziomowy zbiór wartości konfiguracyjnych, z jednym plikiem dla każdej kombinacji komponentu, środowiska i klienta, wymaga całkiem sporej liczby plików. I wiele wartości potrafi się w nich powtarzać.

Wyobraźmy sobie wartość `store_name` dla każdego klienta, którą należy ustawić dla każdej instancji każdego komponentu. Zespół szybko postanawia ustawić tę wartość w jednym miejscu, razem z współdzielonymi wartościami i dodać do skryptu opakowującego kod odczytujący wartości z konfiguracji współdzielonej oraz z indywidualnych konfiguracji poszczególnych komponentów.

Wkrótce okazuje się, że potrzebne są jeszcze pewne współdzielone wartości we wszystkich instancjach w danym środowisku, co oznacza utworzenie trzeciego zbioru konfiguracyjnego. Gdy jakiś element konfiguracji ma inne wartości w różnych plikach konfiguracyjnych, skrypt musi rozwiązać ten problem zgodnie z regułą pierwszeństwa.

Tego typu hierarchia parametrów jest nieco kłopotliwa do zakodowania. Ludziom trudno jest zorientować się w niej podczas wprowadzania nowych parametrów, konfigurowania prawidłowych wartości oraz śledzenia i debugowania wartości używanych w danej instancji.

Stosowanie rejestru konfiguracji nadaje złożoności inne zabarwienie. Zamiast poszukiwać wartości parametrów w tablicy plików, trzeba szukać ich w poddrzewach rejestru. Skrypt opakowujący może obsługiwać rozwiązywanie wartości z różnych części rejestru, podobnie jak w przypadku plików konfiguracyjnych. A można też użyć skryptu do wcześniejszego ustawienia wartości rejestru, tak aby znał logikę potrzebną do rozwiązywania hierarchii wartości domyślnych i mógł ustalić wartość końcową dla poszczególnych instancji. Każda z tych metod powoduje ból głowy przy ustawianiu i śledzeniu pochodzenia wartości danego parametru.

Upraszczenie skryptów opakowujących

Widziałem zespoły, które poświęcają więcej czasu na usuwanie błędów w swoich skryptach opakowujących niż na ulepszanie kodu infrastruktury. Taka sytuacja jest wynikiem mieszania i sprzęgania zagadnień, które powinny być rozdzielone. Oto kilka pomysłów, które warto rozważyć:

Podział cyklu życia projektu

Pojedynczy skrypt nie powinien obsługiwać zadań wykonywanych podczas różnych faz cyklu życia projektu – budowy, promowania i stosowania. Należy pisać i wykorzystywać oddzielne skrypty do każdego z tych działań. Trzeba dobrze rozumieć, w których miejscach mają być przekazywane informacje z każdej z tych faz do następnej. Fazy powinny mieć wyraźnie zaimplementowane granice, jak w przypadku każdego interfejsu API czy kontraktu.

Separacja zadań

Należy rozdzielać różne zadania związane z zarządzaniem infrastrukturą, takie jak zestawianie wartości konfiguracyjnych, pakowanie kodu i uruchamianie narzędzi infrastruktury. I w tym przypadku ważne jest definiowanie punktów integracji między tymi zadaniami i dbanie o ich luźne sprzężenie.

Brak sprzężenia między projektami

Skrypt, który orkiestruje działania obejmujące wiele projektów, powinien być odseparowany od skryptów wykonujących zadania w ramach poszczególnych projektów. I powinna być możliwość niezależnego wykonywania tych zadań dla dowolnego projektu.

Niewiedza kodu opakowującego

Skrypty nie powinny nic wiedzieć o obsługiwanych przez siebie projektach. Należy unikać kodowania na stałe czynności, które zależą od tego, co kod infrastruktury robi w ramach skryptów opakowujących. Idealne skrypty opakowujące są ogólne i można ich używać do dowolnego projektu infrastruktury o określonym kształcie (np. do każdego projektu wykorzystującego dane narzędzie stosu).

W tym wszystkim pomaga traktowanie skryptów opakowujących jak „prawdziwego” kodu. Skrypty można testować i weryfikować za pomocą takich narzędzi, jak `shellcheck`¹⁸. Należy stosować do nich zasady projektowania dobrego oprogramowania, takie jak regułę kompozycji, zasadę pojedynczej odpowiedzialności i projektowanie oparte na koncepcjach domeny. Więcej na ten temat można znaleźć w rozdziale 15 i odsyłaczach do innych źródeł informacji na temat projektowania dobrego oprogramowania.

¹⁸ <https://oreil.ly/TTriI>

Podsumowanie

Stworzenie solidnego, niezawodnego procesu dostarczania kodu infrastruktury jest podstawowym warunkiem osiągnięcia dobrej wydajności mierzonej czterema kluczowymi wskaźnikami (patrz „Cztery kluczowe wskaźniki” na stronie 10). System dostarczania jest formalną implementacją szybkiego i niezawodnego dostarczania zmian w systemie.

Przepływy pracy zespołowej

Używanie kodu do budowania i zmieniania infrastruktury jest podejściem radykalnie innym od tradycyjnych sposobów pracy. Zmiany w serwerach wirtualnych i konfiguracjach sieci są dokonywane pośrednio, zamiast dotychczasowego wpisywania komend w wierszu polecenia lub bezpośredniego edytowania konfiguracji na żywo. Pisanie kodu, a następnie wypychanie go w celu zastosowania przez zautomatyzowane systemy jest większą zmianą niż opanowanie nowego narzędzia lub umiejętności.

Infrastruktura jako kod zmienia sposób, w jaki każdy, kto jest zaangażowany w projektowanie i budowanie infrastruktury oraz zarządzanie nią, pracuje samodzielnie lub w grupie. Ten rozdział ma na celu wyjaśnienie, jak różne osoby pracują nad kodem infrastruktury. Procesy pracy nad infrastrukturą obejmują projektowanie, definiowanie i stosowanie kodu.

Oto niektóre cechy charakterystyczne efektywnych procesów używanych przez zespoły zarządzające infrastrukturą jako kodem:

- Zautomatyzowany proces jest najłatwiejszym, najbardziej naturalnym sposobem dokonywania zmian przez członków zespołu.
- Ludzie mają jasny sposób zapewniania jakości, funkcjonalności i zgodności z przyjętymi zasadami.
- Zespół zapewnia aktualność swojego systemu przy niewielkim wysiłku. Tam, gdzie trzeba, rzeczy są spójne, a tam, gdzie są wymagane warianty, są one czytelne i dobrze zarządzane.
- Wiedza zespołu o systemie jest zawarta w kodzie, a sposoby pracy zespołu są wyrażone w automatyzacji.
- Błędy są natychmiast widoczne i łatwe do poprawienia.
- Można łatwo i bezpiecznie zmieniać kod definiujący system, a także automatyzację testującą i dostarczającą ten kod.

Ogólnie rzecz biorąc, dobry zautomatyzowany przepływ pracy jest wystarczająco szybki, aby w sytuacji awaryjnej sprawnie zastosować poprawkę do systemu i żeby nikt nie miał ochoty zabierać się do tego ręcznie. Jest też na tyle niezawodny, że ludzie mają do niego większe zaufanie niż do własnego, ręcznego manipulowania przy konfiguracji działającego systemu.

Ten rozdział i następny są poświęcone omówieniu elementów pracy zespołowej nad kodem infrastruktury. W tym rozdziale skoncentrujemy się na tym, co ludzie robią w swoich przepływach pracy, a w kolejnym przyjrzymy się sposobom organizacji baz kodu infrastruktury i zarządzania nimi.



Mierzenie efektywności przepływu pracy

Cztery kluczowe wskaźniki z badania Accelerate, wspomnianego w podrzdziale „Cztery kluczowe wskaźniki” na stronie 10, są dobrą podstawą podejmowania decyzji, jak mierzyć efektywność naszego zespołu. Dowody wskazują, że organizacje dobrze sobie radzące według tych wskaźników zazwyczaj dobrze realizują swoje zasadnicze cele organizacyjne, takie jak rentowność i cena akcji.

Zespół może użyć tych wskaźników do ustalenia wskaźników mierzących poziom usług (SLI – Service Level Indicator), docelowych poziomów usług (SLO – Service Level Objective), używanych na potrzeby zespołu oraz gwarantowanych poziomów usług (SLA – Service Level Agreement), będących zobowiązaniami wobec innych stron¹. To, co konkretnie powinniśmy mierzyć, zależy od kontekstu naszego zespołu i sposobów, w jakie próbujemy poprawić wyniki na wyższym poziomie.

Ludzie

Niezawodny, zautomatyzowany system IT jest jak Soylent Green – jego tajnym składnikiem są ludzie². O ile człowiek nie powinien być potrzebny do dostarczania zmian kodu do systemów produkcyjnych, może poza przeglądaniem wyników testów i klikaniem przycisków, to ludzie są potrzebni do ciągłego budowania, naprawiania, adaptowania i ulepszania systemu.

Istnieje kilka ról związanych z większością systemów infrastruktury, zarówno tych zautomatyzowanych, jak i innych. Role te nie przekładają się często na osoby na zasadzie „jeden do jednego” – niektóre osoby mają przypisanych więcej ról niż jedną, a do tego niektóre role mogą być współdzielone przez wiele osób:

Użytkownicy

Kto bezpośrednio używa infrastruktury? W wielu organizacjach robią to zespoły zajmujące się aplikacjami. Zespoły te mogą programować własne aplikacje albo konfigurować aplikacje innych firm i zarządzać nimi.

1 Więcej na temat SLO, SLA i SLI można znaleźć w „SRE Fundamentals” Google’a (<https://oreil.ly/-0NXt>).

2 Soylent Green (<https://oreil.ly/kuwDD>) to produkt spożywczy z klasycznego dystopijnego filmu science fiction o tej samej nazwie (polski tytuł „Zielona pożywka”). Spoiler: „Soylent Green to ludzie”! Choć moi prawnicy radzą mi zwrócić uwagę, że w przypadku niezawodnego zautomatyzowanego systemu IT tajnym składnikiem są żywi ludzie.

Specjaliści nadzoru

Wiele osób ustawia zasady dotyczące środowiska pod kątem różnych aspektów, takich jak bezpieczeństwo, zgodność z prawem, architektura, wydajność, kontrola kosztów i poprawności.

Projektanci

Ludzie, którzy projektują infrastrukturę. W niektórych organizacjach są to architekci, być może podzieleni na różne domeny, takie jak sieć lub magazyn.

Twórcy narzędzi

Ludzie, którzy zapewniają usługi, narzędzia i komponenty, używane przez inne zespoły do budowania i uruchamiania środowisk. Przykładem jest zespół monitorujący lub programiści, którzy tworzą biblioteki kodu infrastruktury wielokrotnego użytku.

Budowniczy

Ludzie, którzy budują i zmieniają infrastrukturę. Mogą to robić ręcznie przy użyciu konsol lub innych interfejsów, uruchamiając skrypty lub narzędzia stosujące kod infrastruktury.

Testerzy

Ludzie, którzy sprawdzają infrastrukturę. Ta rola nie ogranicza się do analityków jakości (QA – quality analysts). Obejmuje także ludzi testujących i oceniających systemy pod kątem domeny nadzoru, takiej jak bezpieczeństwo lub wydajność.

Pomoc techniczna

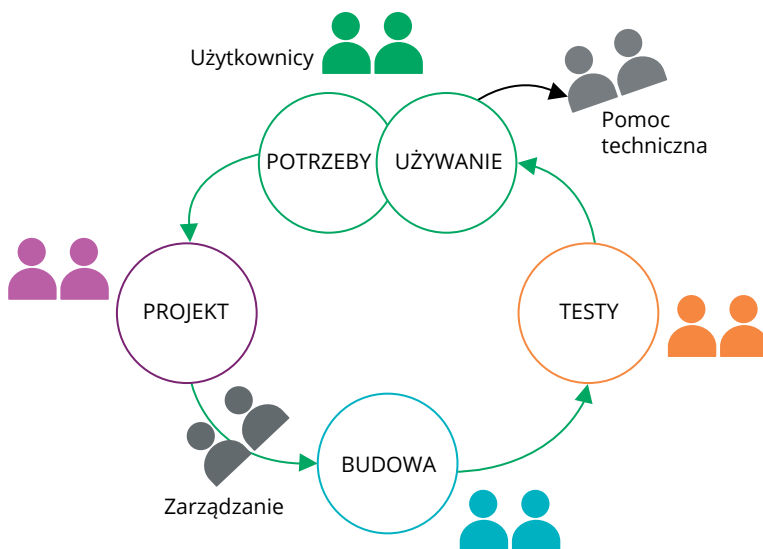
Ludzie, którzy pilnują, aby system działał poprawnie i naprawiają go w razie potrzeby.

Na rysunku 20-1 pokazana jest klasyczna struktura z oddzielnym zespołem dla każdej części przepływu pracy dotyczącego zmiany systemu.

Wiele ról można podzielić według różnych dziedzin infrastruktury, takich jak sieć, pamięć masowa lub serwery. Potencjalnie można je również podzielić według dziedzin nadzoru, takich jak bezpieczeństwo, zgodność, architektura i wydajność. Wiele dużych organizacji tworzy barokowe struktury organizacyjne składające się z mikrospecjalności³.

Ale zdarza się również, i to często, że jakaś osoba lub zespół wykracza poza różne role w swojej pracy. Na przykład zespół do spraw bezpieczeństwa informacji (infosec) może ustanawiać standardy, dostarczać narzędzia do skanowania i przeprowadzać inspekcję zabezpieczeń. W dalszej części tego rozdziału przyjrzymy się sposobom reorganizacji obowiązków (patrz „Reorganizowanie obowiązków” na stronie 344).

3 Pracowałem w międzynarodowym banku razem z grupą, która miała cztery różne środowiska testowania wydania, po jednym na każdy etap procesu wydawania. Dla każdego z tych środowisk jeden zespół konfigurował infrastrukturę, inny zespół wdrażał i konfigurował aplikację, a trzeci zespół testował ją. Niektóre z tych dwunastu zespołów nie były świadome istnienia ich odpowiedników. Efektem tego była słaba wymiana wiedzy i brak spójności na przestrzeni całego procesu wydawania.



Rysunek 20-1 Klasyczne przypisanie wyspecjalizowanego zespołu do każdej części przepływu pracy

Kto pisze kod infrastruktury?

Oto kilka różnych wersji odpowiedzi udzielanych przez organizacje na pytanie, kto pisze i edytuje kod infrastruktury:

Budowniczowie piszą kod

Niektóre organizacje próbują zachować tradycyjne procesy i strukturę zespołów. W efekcie zespół, który buduje (i prawdopodobnie obsługuje) infrastrukturę, wykorzystuje narzędzia infrastruktury jako kodu do optymalizacji swojej pracy. Użytkownicy proszą o jakieś środowisko i zespół budowniczych buduje je dla nich używając swoich narzędzi i skryptów. Ramka „Używanie mapowania strumienia wartości do usprawniania przepływów pracy” zawiera przykład tego, jak optymalizacja procesu zespołu budującego nie poprawia całłościowych działań ani pod względem szybkości, ani jakości.

Użytkownicy piszą kod

Wiele organizacji pozwala zespołom ds. aplikacji definiować infrastrukturę używaną przez te aplikacje. W ten sposób potrzeby użytkownika są dostosowywane do rozwiązania. Wymaga to jednak od każdego zespołu, aby były w nim osoby z dużym doświadczeniem w dziedzinie infrastruktury lub narzędzi upraszczających definiowanie infrastruktury. Wyzwaniem związanym z tymi narzędziami jest dopilnowanie, aby spełniały one potrzeby zespołów ds. aplikacji, zamiast je ograniczać.

Twórcy narzędzi piszą kod

Zespoły specjalistów mogą tworzyć platformy, biblioteki i narzędzia pozwalające użytkownikom definiować potrzebną im infrastrukturę. W takich przypadkach użytkownicy zajmują się zwykle więcej pisaniem konfiguracji niż kodu. Różnica między pisaniem kodu przez twórców narzędzi i budowniczych polega na samowystarczalności. Zespół budowniczych pisze i wykorzystuje kod w odpowiedzi na prośbę użytkowników o utworzenie lub zmianę środowiska. Zespół twórców narzędzi pisze kod, który użytkownicy wykorzystują do tworzenia lub zmieniania swoich własnych środowisk. Podrozdział „Budowanie warstwy abstrakcji” na stronie 279 zawiera przykład tego, co twórcy narzędzi mogą zbudować.

Członkowie nadzoru i testerzy piszą kod

Osoby, które ustanawiają zasady i standardy oraz ci, którzy muszą zapewniać zmiany, mogą tworzyć narzędzia pomagające innym ludziom weryfikowanie ich własnego kodu. Osoby te mogą zostać twórcami narzędzi lub blisko współpracować z takimi twórcami.

Używanie mapowania strumienia wartości do usprawniania przepływów pracy

Mapowanie strumienia wartości jest przydatnym sposobem analizowania czasu realizacji, umożliwiającym zorientowanie się, na co jest poświęcany ten czas⁴.

Mierząc czas poświęcany na różne działania, łącznie z czekaniem, można skoncentrować się na usprawnieniu obszarów, które są najistotniejsze. Zbyt często optymalizujemy części procesu, które wydają się w oczywisty sposób być nieefektywne, ale mają niewielki wpływ na ogólny czas realizacji. Na przykład widziałem zespoły implementujące automatyzację w celu skrócenia czasu wyposażania serwera z ośmiu godzin do dziesięciu minut. Oznacza to imponującą redukcję aż o 98%. Jeśli jednak użytkownicy czekają zwykle dziesięć dni, aby dostać nowy serwer, to całkowity zysk jest znacznie mniej ekscytujący i wynosi 10%. Jeśli prośba o serwer czeka w kolejce przeciętnie osiem dni, to na tym przede wszystkim należy skoncentrować swoje wysiłki.

Mapowanie strumienia wartości pokazuje czas potrzebny na wykonanie działania, co pozwala znaleźć najlepsze możliwości usprawnienia. Dokonując usprawnień należy kontynuować mierzenie czasu realizacji od końca do końca, a także innych wskaźników, takich jak na przykład wskaźniki niepowodzeń. Pomaga to uniknąć optymalizacji jednej części procesu, która mogłaby spowodować pogorszenie całego przepływu.

⁴ Dobrym źródłem informacji jest książka *Value Stream Mapping* Karena Martina i Mike’a Osterlinga (McGraw-Hill Education).

Stosowanie kodu do infrastruktury

Typowy przepływ pracy w przypadku dokonywania zmian infrastruktury zaczyna się od kodu we współdzielonym repozytorium źródła. Członek zespołu pobiera najnowszą wersję kodu do swojego środowiska pracy i edytuje ją. Gdy nowa wersja kodu jest już gotowa, zostaje wypchnięta do repozytorium źródła, a następnie jest stosowana do różnych środowisk.

Przystępując do automatyzacji infrastruktury wiele osób uruchamia narzędzia używając wiersza polecenia w swoim środowisku. Ale za takimi działaniami kryją się pułapki.

Stosowanie kodu z poziomu lokalnej stacji roboczej

Stosowanie kodu infrastruktury z poziomu wiersza polecenia może być przydatne w przypadku testowej instancji infrastruktury, której nikt inny nie używa. Ale jeśli mamy do czynienia z współdzieloną instancją infrastruktury, bez względu na to, czy jest to środowisko produkcyjne, czy dostarczania (patrz „Środowiska dostarczania” na stronie 59), uruchamianie narzędzia z poziomu naszego lokalnego środowiska pracy już stwarza problemy.

Ktoś może wprowadzić zmiany do swojej lokalnej wersji kodu przed jej zastosowaniem. Jeśli zastosuje ten kod przed wypchnięciem zmian do współdzielonego repozytorium, to nikt inny nie będzie miał dostępu do tej wersji kodu. Może to spowodować problemy, gdy ktoś będzie musiał debugować infrastrukturę.

Jeśli osoba, która zastosowała lokalną wersję kodu, nie wypchnie natychmiast swoich zmian, ktoś inny może pobrać i edytować starszą wersję kodu. Zastosowanie kodu przez tę drugą osobę spowoduje cofnięcie zmian wprowadzonych przez pierwszą. Taka sytuacja szybko staje się myląca i trudna do rozwikłania (patrz rysunek 20-2).



Rysunek 20-2 Lokalne edytowanie i stosowanie kodu prowadzi do konfliktów

Należy zwrócić uwagę, że rozwiązania blokujące, takie jak blokowanie stanu przez Terraform⁵, nie zapobiegają tej sytuacji. Blokowanie uniemożliwia dwóm osobom zasto-

5 https://oreil.ly/f_QGZ

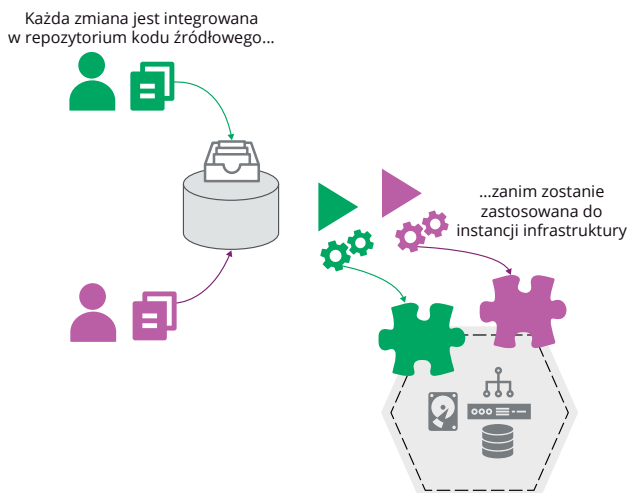
sowanie ich kodu do tej samej instancji jednocześnie. Natomiast w chwili, gdy to piszę, rozwiązania blokujące nie uniemożliwiają ludziom stosowania rozbieżnych wersji kodu, o ile tylko każdy zaczeka na swoją kolej.

Wniosek jest więc taki, że do dowolnej instancji infrastruktury kod powinien być zawsze stosowany z tej samej lokalizacji. Można wyznaczyć konkretną osobę odpowiedzialną za każdą instancję. Ale za tym również kryją się liczne pułapki, w tym ryzyko uzależnienia od jednej osoby i jej stacji roboczej. Lepszym rozwiązaniem jest centralny system obsługujący współdzielone instancje infrastruktury.

Stosowanie kodu z poziomu scentralizowanej usługi

Można wykorzystać scentralizowaną usługę do stosowania kodu infrastruktury do instancji, niezależnie od tego, czy jest to aplikacja hostowana samodzielnie, czy usługa innej firmy. Scentralizowana usługa pobiera kod z repozytorium kodu źródłowego lub repozytorium artefaktów (patrz „Pakowanie kodu infrastruktury jako artefaktu” na stronie 315) i stosuje go do infrastruktury, wymuszając klarowny, kontrolowany proces zarządzania wersją kodu, która ma być stosowana.

Jeśli dwie osoby pobierają i edytują kod, muszą rozwiązać wszystkie różnice w swoim kodzie w momencie jego integracji z gałęzią używaną przez narzędzie. Gdy pojawia się problem, można łatwo zobaczyć, która wersja kodu została zastosowana i poprawić ją (patrz rysunek 20-3).



Rysunek 20-3 Centralne integrowanie i stosowanie kodu

Centralna usługa zapewnia również konsekwentne uruchamianie narzędzia infrastruktury, bez zakładania, że nikt się nie pomylił ani nie „usprawnił” przepływu pracy (nie wyłączył się z niego). Za każdym razem używa tych samych wersji narzędzia, skryptów i narzędzi pomocniczych.

Wykorzystywanie centralnej usługi dobrze pasuje do modelu potoku dostarczania kodu infrastruktury do różnych środowisk (patrz „Potoki dostarczania infrastruktury” na stronie 113). Dowolne narzędzie lub usługa, których używamy do orkiestracji zmian w całym potoku, przejmuje odpowiedzialność za uruchamianie naszego narzędzia infrastruktury w celu zastosowania najnowszej wersji kodu w każdym środowisku.

Inną korzyścią z wykonywania kodu infrastruktury przez centralną usługę jest wymuszanie przez nią na wszystkich członkach zespołu automatyzacji całego procesu. W przypadku uruchamiania narzędzia z własnej stacji roboczej łatwo jest zostawić kilka rzeczy niedokończonych, które trzeba wykonać ręcznie przed lub po uruchomieniu narzędzia. Centralna usługa nie pozostawia innego wyjścia niż całkowite zautomatyzowanie zadania.



Narzędzia i usługi, które uruchamiają narzędzie infrastruktury za nas

Jest kilka sposobów stosowania kodu infrastruktury przez scentralizowaną usługę. Jeśli używamy serwera budowy, takiego jak Jenkins albo narzędzia CD, takiego jak GoCD lub ConcourseCI, możemy zaimplementować zadania lub etapy do uruchamiania naszego narzędzia infrastruktury. Tego typu narzędzia pozwalają zarządzać różnymi wersjami kodu z repozytorium źródła i potrafią promować kod między etapami. Te wielofunkcyjne narzędzia ułatwiają ponadto integrowanie przepływów pracy dla aplikacji, infrastruktury i innych części systemu. Można używać samodzielnie hostowanych instancji tych usług albo korzystać z ofert zewnętrznych. Więcej informacji o serwerach i oprogramowaniu potoków można znaleźć w podrozdziale „Usługi i oprogramowanie potoków dostarczania” na stronie 117.

Kilku dostawców oferuje produkty lub usługi przeznaczone specjalnie do uruchamiania narzędzi infrastruktury. Przykłady to Terraform Cloud⁶, Atlantis⁷ i Pulumi for Teams⁸. WeaveWorks oferuje platformę Weave Cloud⁹, która stosuje kod infrastruktury do klastrów Kubernetes.

Prywatne instancje infrastruktury

Większość przepływów pracy omawianych w tej książce obejmuje pobranie kodu, jego edycję i wypchnięcie do współdzielonego repozytorium kodu¹⁰. Następnie proces potoku dostarczania stosuje kod do odpowiedniego środowiska.

Najlepiej jest móc przetestować swoje zmiany kodu przed wypchnięciem ich do współdzielonego repozytorium. Takie działanie pozwala upewnić się, że wprowadzona zmiana działa zgodnie z oczekiwaniami i jest szybsze niż czekanie na przepchnięcie kodu przez potok aż do etapu testów online (patrz „Etapy testowania online dla stosów” na stronie 128). Pomaga również uniknąć przerwania budowy w przypadku niepowodzenia

6 <https://oreil.ly/67w1M>

7 <https://oreil.ly/YJps9>

8 <https://oreil.ly/-qhGe>

9 <https://oreil.ly/HrAI9>

10 Zwłaszcza w rozdziałach 8 i 9.

któregoś z etapów potoku na skutek naszej zmiany i tym samym przeszkodzenia innym w pracy nad bazą kodu.

Istnieje kilka rzeczy, które zespół może zrobić, aby ułatwić testowanie zmian kodu przed jego wypchnięciem.

Po pierwsze, należy upewnić się, że każda osoba pracująca nad kodem infrastruktury może utworzyć własną instancję infrastruktury. Istnieją ograniczenia na to, co ludzie mogą testować lokalnie, bez korzystania z platformy chmurowej, co opisałem w podrzdziale „Etapy testowania offline dla stosów” na stronie 125. Niektórych może kusić uruchomienie współdzielonych „deweloperskich” instancji infrastruktury. Ale jak już wyjaśniłem wcześniej, pozwolenie wielu osobom na stosowanie lokalnie edytowanego kodu do współdzielonej instancji prowadzi do zamieszania. Dlatego należy znaleźć sposób, aby ludzie mogli stawiać własne instancje infrastruktury i niszczyć je, gdy już nie będą ich aktywnie używać.

Po drugie, należy dbać, aby elementy infrastruktury były małe. Jest to oczywiście jedna z trzech podstawowych praktyk powtarzanych w tej książce (patrz rozdział 15). Członek zespołu powinien móc samodzielnie postawić instancję dowolnego komponentu swojego systemu, być może używając warunków początkowych testu w celu obsługi zależności (patrz „Używanie warunków początkowych testu do obsługi zależności” na stronie 132). Ludziom trudno jest pracować nad prywatnymi instancjami, jeśli wymaga to postawienia całego systemu, chyba że jest to wyjątkowo mały system.

Po trzecie, ludzie powinni używać tych samych narzędzi i skryptów do stosowania i testowania swoich instancji infrastruktury, co w przypadku instancji współdzielonych, na przykład w potoku. Pomaga w tym tworzenie pakietów z narzędziami i skryptami, aby każdy mógł wykorzystywać je w swojej lokalizacji¹¹.



Centralne zarządzanie prywatnymi instancjami

Stosowanie kodu do prywatnych instancji na poziomie stacji roboczych członków zespołu jest bezpieczniejsze niż stosowanie go do instancji współdzielonych. Ale czasem warto jest użyć scentralizowanej usługi dla tych instancji. Widziałem zespół, który zmagał się ze zniszczeniem prywatnej instancji, którą ktoś pozostawił działającą wyjeżdżając na wakacje. Osoba ta utworzyła instancję używając lokalnej wersji kodu infrastruktury, której nie wypchnęła do repozytorium, przez co trudno było ją zniszczyć.

Z tego powodu niektóre zespoły wprowadzają zasadę, że każda osoba musi wypchnąć zmianę do prywatnej gałęzi, którą centralna usługa stosuje do prywatnej instancji infrastruktury tej osoby, aby można było ją testować. W takim układzie prywatna gałąź emuluje lokalny kod, więc ludzie nie uznają zmiany za zatwierdzoną, dopóki nie zostanie scalona z gałęzią współdzieloną lub pniem. Natomiast kod jest centralnie dostępny i inne osoby mogą go oglądać i używać w razie nieobecności twórcy.

¹¹ batest (<https://batest.dev>) i Dojo (<https://oreil.ly/v2V7j>) to przykłady narzędzi, które budują powtarzalny, umożliwiający współdzielenie kontener do tworzenia aplikacji i infrastruktury.

Gałęzie kodu źródłowego w przepływach pracy

Gałęzie są zaawansowaną funkcją współdzielonych repozytoriów źródła, która ułatwia zespołom dokonywanie zmian w różnych kopiach bazy kodu – *gałęziach* (*branch*) – a następnie integrowanie wykonanej pracy. Jest wiele strategii i wzorców używania gałęzi jako części przepływu pracy zespołu. Zamiast omawiać je tutaj, odsyłam do artykułu Martina Fowlera „Patterns for Managing Source Code Branches”¹².

Warto podkreślić kilka różnic w ramach strategii rozgałęziania w kontekście infrastruktury jako kodu. Jedną z nich jest różnica między wzorcami ścieżki do produkcji a wzorcami integracji w przypadku rozgałęziania. Drugą jest znaczenie częstotliwości integracji.

Zespoły używają wzorców rozgałęziania *ścieżki do produkcji* w celu ustalania, które wersje kodu mają być stosowane do środowisk¹³. Typowe wzorce ścieżki do produkcji obejmują gałęzie wydań i gałęzie środowisk (gałęzie środowisk są omówione w podrozdziale „Dostarczanie kodu z repozytorium kodu źródłowego” na stronie 317).

Wzorce integracji w przypadku rozgałęziania opisują, w jaki sposób osoby pracujące nad bazą kodu mogą decydować, kiedy i jak ma być integrowana ich praca¹⁴. Większość zespołów używa wzorca integracji magistrali (*mainline*), albo z rozgałęzieniem funkcji, albo z ciągłą integracją.

Konkretny wzorec lub strategia mają mniejsze znaczenie niż sposób ich użycia. Najważniejszym czynnikiem wpływającym na efektywność wykorzystania gałęzi przez zespół jest *częstotliwość integracji*, czyli jak często każdy scala cały swój kod z tą samą (główną) gałęzią centralnego repozytorium¹⁵. Badanie DORA Accelerate wykazało, że częstsza integracja całego kodu w ramach zespołu jest skorelowana z lepszymi wynikami handlowymi. Oznacza to, że wszyscy w zespole powinni integrować cały swój kod – na przykład z główną gałęzią lub pniem – co najmniej raz dziennie.



Scalanie to nie integracja

Ludzie czasem myślą serwer budowania, automatycznie uruchamiający testy gałęzi, z ciągłą integracją. Praktyka ciągłej integracji (i korelacja z większą wydajnością zespołu) opiera się na pełnej integracji wszystkich zmian, nad którymi cały zespół pracuje w bazie kodu.

Chociaż osoba używająca wzorca gałęzi funkcji potrafi często scalać aktualną główną gałąź z własną gałęzią, to zazwyczaj nie integruje efektów swojej pracy z powrotem z główną gałęzią, dopóki nie dokończy swojej funkcji. A jeśli pozostałe osoby pracują w ten sam sposób nad swoimi

¹² <https://oreil.ly/Ozgav>

¹³ Zobacz podrozdział o ścieżce do produkcji w artykule Fowlera „Patterns for Managing Source Code Branches” (<https://oreil.ly/xxBYF>).

¹⁴ Fowler opisuje w artykule również wzorce integracji (<https://oreil.ly/DRZ2U>).

¹⁵ Szczegółowe omówienie można znaleźć w podrozdziale na temat częstotliwości integracji (<https://oreil.ly/1I8V3>).

gałęziami funkcji, kod nie będzie w pełni zintegrowany, dopóki wszyscy nie dokończą swoich funkcji i nie scalą zmian z główną gałęzią.

Integracja obejmuje scalanie w obu kierunkach – scalanie przez poszczególne osoby ich zmian z główną gałęzią, jak również scalanie odwrotne, czyli scalanie głównej gałęzi z ich własną gałęzią lub lokalną kopią kodu. Ciągła integracja oznacza zaś, że każdy robi to podczas swojej pracy, co najmniej raz dziennie.

Zapobieganie dryfowi konfiguracji

W rozdziale 2 zostały opisane niebezpieczeństwa, jakie niesie dryf konfiguracji (patrz „Dryf konfiguracji” na stronie 17), czyli utrata spójności elementów infrastruktury z upływem czasu. Dryf konfiguracji jest często wynikiem sytuacji, w której zespoły wykorzystują narzędzia do kodowania infrastruktury do automatyzacji fragmentów działań starszego typu, zamiast w pełni przyjąć swoje metody pracy.

Jest kilka rzeczy, które można zrobić w swoich przepływach pracy, aby uniknąć dryfu konfiguracji.

Minimalizacja opóźnienia automatyzacji

Opóźnienie automatyzacji to odstęp między kolejnymi uruchomieniami zautomatyzowanego procesu, takiego jak zastosowanie kodu infrastruktury. Im więcej czasu mija od ostatniego uruchomienia procesu, tym większe jest prawdopodobieństwo jego niepowodzenia¹⁶. Rzeczy zmieniają się z czasem, nawet jeśli nikt świadomie nie dokonuje zmian.

Nawet jeśli kod się nie zmienił, to zastosowanie go po dłuższej przerwie może się nie powieść i to z różnych przyczyn:

- Ktoś zmienił inną część systemu, na przykład zależność, w taki sposób, że awaria pojawia się tylko wtedy, gdy ponownie zastosujemy nasz kod.
- Uaktualnienie lub zmiana konfiguracji narzędzia lub usługi używanej do stosowania kodu może być niekompatybilna z naszym kodem.
- Zastosowanie niezmienionego kodu może mimo wszystko spowodować aktualizację zależności przechodnich, na przykład pakietów systemu operacyjnego.
- Ktoś mógł dokonać poprawki lub ulepszenia, ale zapomniał o zapisaniu tego w kodzie. Ponowne zastosowanie naszego kodu cofa tę zmianę.

¹⁶ Opóźnienie automatyzacji dotyczy również innych rodzajów automatyzacji. Na przykład, jeśli uruchomimy zautomatyzowany zestaw testów aplikacji dopiero na koniec długiego cyklu wydania, to spędzimy dni lub tygodnie na aktualizowaniu tego zestawu testów, aby dopasować go do zmian kodu. Jeśli uruchamiamy testy za każdym razem po zatwierdzeniu zmiany kodu, to wystarczy kilka zmian w celu dopasowania tych testów do kodu, więc ich pełny zestaw jest zawsze gotowy do użycia.

Wniosek z opóźnienia automatyzacji jest taki, że im częściej stosujemy kod infrastruktury, tym mniejsze jest prawdopodobieństwo niepowodzenia. Gdy pojawiają się problemy, można szybciej wykryć przyczynę, ponieważ mniej się zmieniło od ostatniego pomyślnego uruchomienia.

Unikanie stosowania ad hoc

Niektóre zespoły mają nawyk kontynuowania metody pracy pochodzącej z epoki żelaza, polegającej na stosowaniu kodu wyłącznie w celu dokonania specyficznej zmiany. Mogą używać kodu infrastruktury do udostępnienia nowej infrastruktury, ale nie w celu dokonania zmian w już istniejących systemach. Albo mogą napisać i zastosować kod infrastruktury do wprowadzenia zmian ad hoc w określonych częściach swojego systemu. Na przykład potrafią zakodować jednorazową zmianę konfiguracji dla jednego ze swoich serwerów aplikacji. Nawet jeśli jakieś zespoły używają kodu do wprowadzania zmian i stosują ten kod do wszystkich instancji, to zdarza się, że robią to tylko wtedy, gdy dokonują zmiany w kodzie. Takie nawyki mogą powodować dryf konfiguracji lub opóźnienie automatyzacji.

Ciągłe stosowanie kodu

Podstawową strategią eliminacji dryfu konfiguracji jest ciągłe stosowanie kodu infrastruktury do wszystkich instancji, nawet wtedy, gdy kod nie uległ zmianie. Wiele narzędzi do konfiguracji serwerów, w tym Chef i Puppet, jest tak zaprojektowanych, aby powtarzały stosowanie konfiguracji według harmonogramu, zwykle co godzinę¹⁷.

Metodologia GitOps (patrz „GitOps” na stronie 343) obejmuje ciągłe stosowanie kodu z gałęzi kodu źródłowego do każdego środowiska. Należy mieć możliwość używania centralnej usługi stosowania kodu (jak to jest opisane w podrozdziale „Stosowanie kodu z poziomu scentralizowanej usługi” na stronie 337) w celu ciągłego ponawiania stosowania kodu do każdej instancji.

Infrastruktura niezmiennalna

Infrastruktura niezmiennalna rozwiązuje problem dryfu konfiguracji w inny sposób. Zamiast częstego stosowania kodu konfiguracji do instancji infrastruktury, stosuje się go tylko raz, podczas tworzenia instancji. Z chwilą dokonania zmiany kodu zostaje utworzona nowa instancja, która zastępuje starą.

Wprowadzanie zmian przez tworzenie nowych instancji wymaga zaawansowanych technik w celu uniknięcia przestoju („Zmiany bez przestojów” na stronie 366) i może nie nadawać się do wszystkich przypadków użycia. Opóźnienie automatyzacji nadal stanowi

¹⁷ Wczesne przedstawienie tej koncepcji można znaleźć w artykule ConfigurationSynchronization (<https://oreil.ly/3KZL4>).

potencjalny problem, dlatego zespoły wykorzystujące infrastrukturę niezmienną zazwyczaj często odbudowują instancje, jak w przypadku serwerów feniksów¹⁸.

GitOps

GitOps jest odmianą infrastruktury jako kodu, która polega na ciągłej synchronizacji kodu, od gałęzi kodu źródłowego do środowisk. GitOps kładzie nacisk na definiowanie systemów jako kodu (patrz „Podstawowa praktyka: definiowanie wszystkiego jako kodu” na stronie 11).

Metodologia GitOps nie określa podejścia do testowania i dostarczania kodu infrastruktury, ale jest kompatybilna z używaniem potoku do dostarczania kodu („Potoki dostarczania infrastruktury” na stronie 113). GitOps odradza jednak wykorzystywanie artefaktów dostarczania („Pakowanie kodu infrastruktury jako artefaktu” na stronie 315), proponując zamiast tego promowanie zmian kodu poprzez scalanie ich z gałęziami kodu źródłowego (patrz „Dostarczanie kodu z repozytorium kodu źródłowego” na stronie 317).

Innym kluczowym elementem GitOps jest ciągła synchronizacja kodu z systemami („Ciągłe stosowanie kodu” na stronie 342). Zamiast używać zadania serwera budowy lub etapu potoku do stosowania kodu w momencie jego zmiany („Unikanie stosowania ad hoc” na stronie 342), GitOps wykorzystuje usługę, która ciągle porównuje kod z systemem, zmniejszając dryf konfiguracji (patrz „Dryf konfiguracji” na stronie 17).

Niektóre zespoły opisują swoje procesy jako GitOps, ale w praktyce implementują jedynie gałęzie dla środowisk, bez ciągłej synchronizacji kodu ze środowiskami. W ten sposób można łatwo doprowadzić do procesu zmian ad hoc i złych nawyków kopiowania, wklejania i edytowania zmian kodu dla każdego środowiska (patrz „Antywzorzec: środowiska kopiuj-wklej” na stronie 63).

Nadzór w przepływie pracy opartym na potoku

Nadzór jest zmartwieniem większości organizacji, zwłaszcza dużych i tych, które działają w branżach regulowanych, takich jak finanse i opieka zdrowotna. Niektórzy ludzie postrzegają nadzór jako przekleństwo, które oznacza stawianie dodatkowej przeszkody na drodze do wykonania pożytecznej pracy. Jednak oznacza po prostu dbanie o to, żeby wszystko było robione odpowiedzialnie, zgodnie z zasadami organizacji.

W rozdziale 1 wyjaśniłem, że jakość – nadzór jest aspektem jakości – może zapewnić szybkie dostarczanie, a możliwość szybkiego dostarczania zmian może poprawić

18 Serwer feniks (<https://oreil.ly/ngW7M>) to serwer, który jest często odbudowywany w celu zapewnienia, że proces wyposażania jest powtarzalny. Można to osiągnąć za pomocą innych konstrukcji infrastruktury, w tym stosów infrastruktury.

jakość (patrz „Używanie infrastruktury jako kodu do optymalizacji pod kątem zmian” na stronie 6). Zgodność jako kod¹⁹ wykorzystuje automatyzację i praktyki bardziej zespołowej pracy, aby to pozytywne sprzężenie przynosiło efekty.

Reorganizowanie obowiązków

Definiowanie systemów jako kodu stwarza możliwości reorganizacji obowiązków ludzi biorących udział w pracy nad infrastrukturą (osób wymienionych w podrozdziale „Ludzie” na stronie 332) oraz sposobu, w jaki angażują się oni w swoją pracę. Oto niektóre czynniki składające się na te możliwości:

Wielokrotne używanie

Kod infrastruktury może być przeprojektowywany, sprawdzany i używany ponownie do wielu środowisk i systemów. Zastosowanie kodu do każdego nowego serwera lub środowiska nie wymaga długiego przeprojektowywania, sprawdzania i zatwierdzania, jeśli jest to kod, który przeszedł już przez cały ten proces.

Kod roboczy

Ponieważ pisanie kodu odbywa się szybko, ludzie mogą podejmować decyzje na podstawie przeglądania kodu roboczego i przykładowej infrastruktury. Zapewnia to szybsze i dokładniejsze pętle informacji zwrotnej niż korzystanie z diagramów i specyfikacji.

Spójność

Kod tworzy środowiska w sposób znacznie bardziej konsekwentny niż ludzie posługujący się listami kontrolnymi. W efekcie testowanie i przeglądanie infrastruktury we wcześniejszych środowiskach daje szybszą i lepszą informację zwrotną niż robienie tego na dalszym etapie procesu.

Zautomatyzowane testowanie

Zautomatyzowane testowanie, w tym dotyczące różnych kwestii nadzoru, takich jak bezpieczeństwo i zgodność, daje ludziom pracującym nad kodem infrastruktury szybką informację zwrotną. Mogą oni usunąć wiele problemów w trakcie pracy, bez konieczności angażowania do tego specjalistów.

Demokratyzacja jakości

Ludzie, którzy nie są specjalistami, mogą dokonywać zmian w kodzie dotyczącym potencjalnie wrażliwych obszarów infrastruktury, takich jak sieć i zasady bezpieczeństwa. Mogą używać narzędzi i testów stworzonych przez specjalistów w celu sprawdzania poprawności swoich zmian. Zaś specjaliści nadal mogą przeglądać i zatwierdzać zmiany przed ich zastosowaniem do systemów produkcyjnych. Takie przeglądanie jest bardziej wydajne, ponieważ specjalista może bezpośrednio oglądać kod, raporty testowania i działające instancje testowe.

19 Artykuły na temat zgodności jako kodu można znaleźć na stronie O'Reilly (https://oreil.ly/G_cYR).

Kanały nadzoru

Baza kodu infrastruktury i potoki używane do dostarczania zmian do instancji produkcyjnych mogą być zorganizowane na podstawie wymagań dotyczących nadzoru. W ten sposób np. zmiana zasad bezpieczeństwa będzie przechodzić przez etapy przeglądania i zatwierdzania niekoniecznie wymagane w przypadku zmian dotyczących mniej wrażliwych obszarów.

Wiele sposobów, jakimi możemy zmienić to, jak ludzie zarządzają systemami, obejmuje przesunięcie ich obowiązków „w lewo”, czyli na wcześniejsze etapy procesu.

Przesunięcie w lewo

W rozdziale 8 opisałem zasady i praktyki implementowania zautomatyzowanych testów i potoków służących do dostarczania zmian kodu do środowisk. Termin *przesunięcie w lewo* odnosi się do lokalizacji tych działań w ramach przepływów pracy i praktyk dostarczania.

Kod jest rygorystycznie testowany podczas implementacji, na „lewym” końcu przepływu prezentowanego na większości diagramów procesów. Dzięki temu organizacje mogą poświęcać mniej czasu na bardzo złożone procesy występujące na „prawym” końcu przepływu, tuż przed zastosowaniem kodu do środowiska produkcyjnego.

Ludzie zajmujący się nadzorem i testowaniem skupiają się na tym, co dzieje się podczas implementacji – pracy z zespołami, udostępnianiu narzędzi i umożliwianiu praktyk wczesnego i częstego testowania.

Przykładowy proces w przypadku infrastruktury jako kodu podlegającej nadzorowi

ShopSpinner wykorzystuje stos wielokrotnego użytku („Wzorzec: stos wielokrotnego użytku” na stronie 65), za pomocą którego może tworzyć infrastrukturę dla serwera aplikacji do hostowania instancji swojej usługi dla klienta. Gdy ktoś zmienia kod dla tego stosu, może to mieć wpływ na wszystkich klientów.

Grupa liderów technicznych, która odpowiada za decyzje architektoniczne, definiuje wymagania CFR (opisane w podrozdziale „Co należy testować w przypadku infrastruktury?” na stronie 102), które infrastruktura serwera aplikacji musi spełniać. Wymagania te obejmują liczbę i częstotliwość zamówień, które użytkownicy mogą składać w instancji klienta, czasy reakcji interfejsu i czasy odzyskiwania po awarii serwera.

Zespół ds. infrastruktury i zespół ds. aplikacji do spółki z inżynierami niezawodności witryny (Site Reliability Engineers)²⁰ i zapewniania jakości (QA) implementują pewne zautomatyzowane testy, aby sprawdzić wydajność stosu serwera aplikacji w stosunku do wymagań CFR. Testy te muszą być wbudowane w kilka etapów potoku w celu stopniowego testowania różnych komponentów stosu (zgodnie z treścią podrozdziału

²⁰ <https://oreil.ly/eC5yw>

„Testowanie progresywne” na stronie 109). Po umieszczeniu tych testów we właściwych miejscach ludzie nie muszą już przekazywać zmian infrastruktury w celu przejrzenia ich przez grupę liderów technicznych, SRE czy kogokolwiek innego. Gdy jakiś inżynier zmieni na przykład konfigurację sieci, potok automatycznie sprawdzi, czy wynikowa infrastruktura będzie nadal spełniać wymagania CFR, zanim będzie można zastosować ją do instancji produkcyjnych klientów. Jeśli inżynier popełni błąd powodujący niezgodność z CFR, zostanie on wykryty w ciągu kilku minut, gdy etap potoku zmieni kolor na czerwony i będzie można go natychmiast poprawić.

W niektórych przypadkach zmiana może powodować problem z instancją klienta niewykrywalny dla zautomatyzowanych testów. Grupa może przeprowadzić drobiazgową „sekcję”²¹, aby sprawdzić co się stało. Może się okazać, że problem nie został przewidziany przez żadne z wymagań CFR, więc trzeba zmienić któreś wymaganie CFR lub dodać nowe do listy. A może w testach jest luka, która przyczyniła się do przeoczenia problemu i w takim przypadku należy poprawić zestaw testów.



Normalizacja procesów poprawek awaryjnych

Wiele zespołów stosuje oddzielny proces dla zmian awaryjnych, aby umożliwić szybkie dostarczenie poprawek. Potrzeba oddzielnego procesu dla szybkich poprawek jest sygnałem, że zwykły proces zmian można ulepszyć.

Proces zmiany awaryjnej pozwala przyspieszyć działanie na jeden z dwóch sposobów. Jednym jest pominięcie kroków niepotrzebnych. Drugim jest pominięcie kroków niezbędnych. Jeśli można bezpiecznie pominąć jakiś krok w sytuacji awaryjnej, gdy jest silna presja, a stawka jest wysoka, to prawdopodobnie można go także pominąć w zwykłym procesie. Jeśli pominięcie kroku niesie ze sobą niedopuszczalne ryzyko, należy znaleźć rozwiązanie bardziej skuteczne i stosować je za każdym razem²².

Podsumowanie

Gdy organizacja zdefiniuje swoją infrastrukturę jako kod, jej ludzie powinni poświęcać mniej czasu na wykonywanie rutynowych działań i odgrywanie roli strażnika. Zamiast tego powinni poświęcać więcej czasu na ciągłe doskonalenie swoich umiejętności w zakresie ulepszania samego systemu. Ich wysiłki będą widoczne w czterech wskaźnikach dotyczących dostarczania oprogramowania i wydajności operacyjnej.

²¹ <https://oreil.ly/fqPKj>

²² Steve Smith definiuje to jako antywzorzec *podwójne strumienie wartości* (<https://oreil.ly/FI7ng>).

Bezpieczne zmienianie infrastruktury

Temat częstego i szybkiego dokonywania zmian przewija się przez całą tę książkę. Jak wspomniałem na samym początku („Zarzut: musimy wybierać między szybkością i jakością” na stronie 8), szybkość nie tylko nie przyczynia się do niestabilności systemów, ale wręcz zapewnia ich stabilność i na odwrót. Mantra nie brzmi „działaj szybko i psuj”, ale raczej „działaj szybko i usprawniaj”.

Jednak stabilność i jakość nie wynikają z poprawy samej szybkości. Badanie cytowane w pierwszym rozdziale pokazuje, że próba optymalizacji pod kątem tylko szybkości lub tylko jakości nic nie daje. Sprawą kluczową jest uwzględnianie obu tych aspektów. Należy skupiać się na możliwości dokonywania zmian często, szybko i bezpiecznie oraz na szybkim wykrywaniu błędów i ich usuwaniu.

Wszystko, co jest zalecane w tej książce – od konsekwentnego wykorzystywania kodu do budowania infrastruktury, przez robienie z testowania stałego fragmentu pracy, aż po rozbijanie systemów na mniejsze części – umożliwia szybkie, częste i bezpieczne wprowadzanie zmian.

Ale częste dokonywanie zmian w infrastrukturze stanowi wyzwanie dla nieprzerwanej dostępności usług. W tym rozdziale przyjrzymy się temu wyzwaniu i technikom radzenia sobie z nim. U podłoża tych technik leży nastawienie, aby nie postrzegać zmian jako zagrożenia dla stabilności i ciągłości, ale jako okazję do wykorzystania dynamicznego charakteru nowoczesnej infrastruktury. Stosując zasady, praktyki i techniki opisane w całej tej książce można minimalizować zakłócenia powodowane przez zmiany.

Ograniczanie zasięgu zmiany

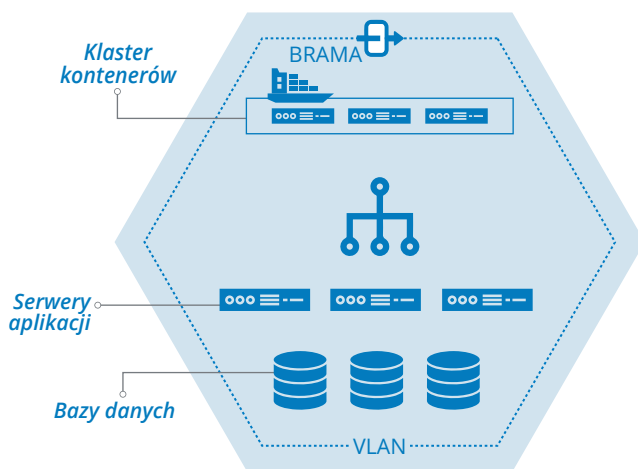
Agile, XP, Lean i inne podobne podejścia optymalizują dostarczanie pod kątem szybkości i niezawodności poprzez dokonywanie zmian małymi porcjami. Łatwiej jest planować, implementować, testować i debugować małe zmiany niż duże, dlatego trzeba się starać ograniczać wielkość porcji¹. Oczywiście często zachodzi konieczność dokonania

1 Donald G. Reinertsen opisuje koncepcję zmniejszenia rozmiaru porcji w swojej książce *The Principles of Product Development Flow* (Celeritas Publishing).

znaczących zmian w systemie, ale można sobie z tym radzić rozbijając je na mniejsze porcje, nadające się do wprowadzenia jednorazowo.

Weźmy jako przykład zespół ShopSpinner, który zbudował na początku infrastrukturę z pojedynczym stosem infrastruktury. Stos ten zawierał klaster serwerów WWW i serwer aplikacji. Z czasem członkowie zespołu dodali więcej serwerów aplikacji i z niektórych zrobili klastry. Potem zdali sobie sprawę, że uruchamianie klastra serwerów WWW i wszystkich serwerów aplikacji w jednej sieci VLAN było złym pomysłem, więc usprawnili projekt swojej sieci przenosząc te elementy do różnych sieci VLAN. Postanowili również skorzystać z rady zawartej w tej książce i podzielili infrastrukturę na wiele stosów, aby umożliwić zmienianie ich indywidualnie.

Oryginalna implementacja zespołu ShopSpinner obejmowała jeden stos i jedną sieć VLAN (patrz rysunek 21-1).

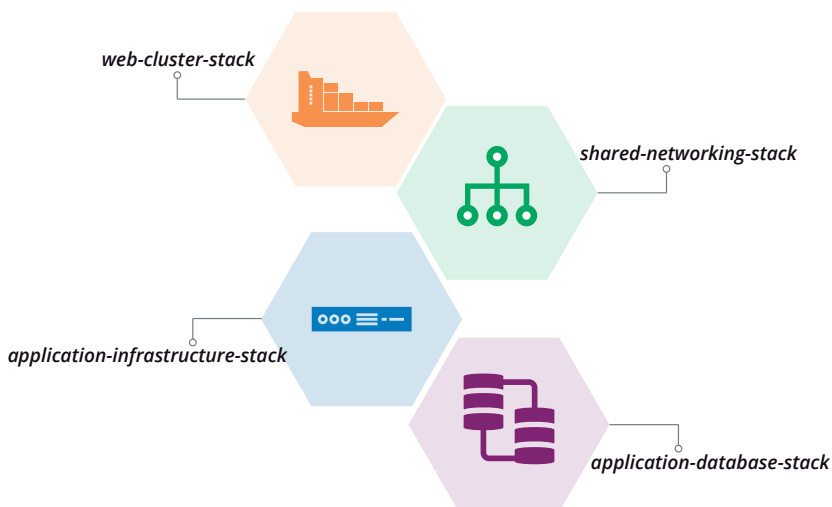


Rysunek 21-1 Przykładowa implementacja początkowa – jeden stos i jedna sieć VLAN

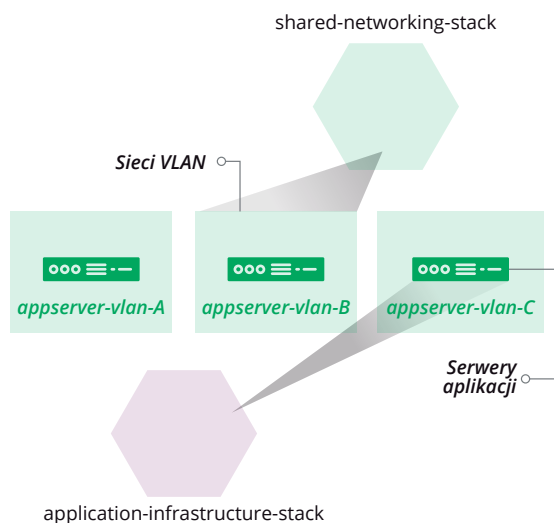
Zespół planuje podzielić swój stos na wiele stosów. Będą one obejmowały współdzielony stos sieci (networking-stack) i stos infrastruktury aplikacji (application-infrastructure-stack), znane z przykładów w poprzednich rozdziałach tej książki. Przewidywany jest również stos web-cluster-stack do zarządzania klastrem kontenerów dla frontendowych serwerów WWW oraz stos applicationdatabase-stack do zarządzania instancją bazy danych dla każdej aplikacji (rysunek 21-2).

Zespół chce także podzielić pojedynczą sieć VLAN na wiele sieci VLAN. Serwery aplikacji będą rozrzucone po tych sieciach VLAN dla zapewnienia redundancji (rysunek 21-3).

W rozdziale 17 opisałem różne opcje wyboru przy projektowaniu i niektóre wzorce implementacji podziału tych przykładowych stosów. Teraz możemy zbadać sposoby przechodzenia od jednej implementacji do drugiej w systemie produkcyjnym.



Rysunek 21-2 Plan podziału na wiele stosów



Rysunek 21-3 Plan utworzenia wielu sieci VLAN

Małe zmiany

Największy bałagan, jaki zdarzyło mi się zrobić w kodzie, miał miejsce, gdy zbudowałem za dużo rzeczy lokalnie przed ich wypchnięciem. Skupienie się na wykonaniu od razu całej zamierzonej pracy jest kuszące. Trudniej jest zrobić drobną zmianę, która tylko nieznacznie przybliży nas do pełnego celu. Implementowanie dużych modyfikacji w postaci serii małych zmian wymaga nowego sposobu myślenia i nowych nawyków.

Na szczęście branża zajmująca się rozwojem oprogramowania wskazała drogę. Przedstawiłem już w tej książce wiele technik obsługujących budowanie systemów porcjami, takich jak TDD (Test-Driven Development), CI (Continuous Integration) i CD (Continuous Delivery). Testowanie progresywne i dostarczanie zmian kodu przy użyciu potoku, opisane w rozdziale 8 i przywoływane w każdym rozdziale, to metody, które to umożliwiają. Pozwalają wprowadzać małe zmiany w kodzie, wypychać je, dowiadywać się, czy działają i umieszczać w środowisku produkcyjnym.

Zespoły, które skutecznie używają tych metod, bardzo często wypychają zmiany. Jeden inżynier może wypychać zmiany nawet co godzinę i każda z tych zmian jest zintegrowana z główną bazą kodu i testowana w pełni zintegrowanym systemie pod kątem zastosowania do środowiska produkcyjnego.

Ludzie używają różnych wariantów dokonywania znaczących modyfikacji w postaci serii drobnych zmian:

Zmiany przyrostowe

Zmiana przyrostowa to taka zmiana, która powoduje dodanie jednego fragmentu zaplanowanej implementacji. Przyrostowa budowa przykładowego systemu ShopSpinner polegałaby na implementowaniu za każdym razem jednego stosu. Najpierw zostałby utworzony współdzielony stos sieci, potem stos klastra serwerów WWW, a na koniec stos infrastruktury aplikacji.

Zmiany iteracyjne

Zmiany iteracyjne oznaczają stopniowe ulepszanie systemu. Na początku budowy systemu ShopSpinner zostałaby utworzona bazowa wersja wszystkich trzech stosów. Potem nastąpiłaby seria zmian, z których każda rozszerzałaby możliwości każdego z tych stosów.

Kroczący szkielet

Kroczący szkielet² to podstawowa implementacja głównych części nowego systemu, która ma pomóc zweryfikować ogólny projekt i strukturę³. Ludzie często tworzą kroczący szkielet dla projektu infrastruktury, a do tego podobne, wstępne implementacje aplikacji, które będą jej używać, aby zespoły mogły się przekonać, jak będzie wyglądało dostarczanie, wdrażanie i działanie. Wstępna implementacja oraz wybór narzędzi i usług dla szkieletu często nie pokrywają się z tym, co jest zaplanowane w dłuższej perspektywie. Na przykład można planować wykorzystanie w pełni funkcjonalnego rozwiązania do monitorowania, a zbudować kroczący szkielet, który będzie używał podstawowych, gotowych usług oferowanych przez dostawcę chmury.

² <https://oreil.ly/1I7bC>

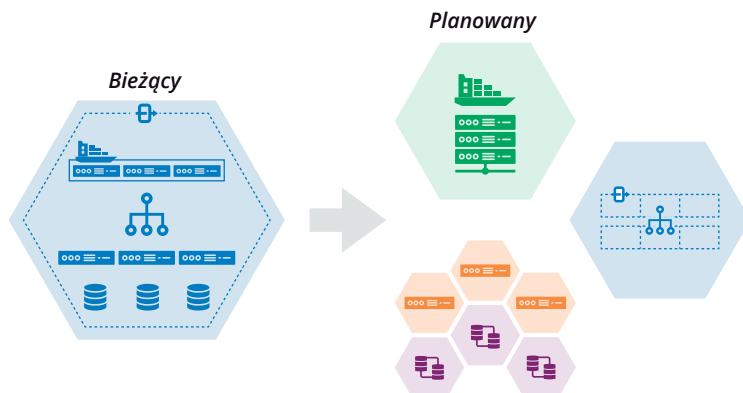
³ Książka *Growing Object-Oriented Software, Guided by Tests* Steve'a Freemana i Nata Pryce'a (Addison-Wesley) zawiera rozdział poświęcony kroczącemu szkieletom.

Refaktoryzacja

Refaktoryzacja⁴ oznacza zmianę projektu systemu lub tylko jego komponentu, bez zmiany jego zachowania. Refaktoryzacja jest często stosowana w celu utorowania drogi dla modyfikacji zmieniających zachowanie⁵. Refaktoryzacja może poprawiać przejrzystość kodu w celu ułatwienia dokonywania w nim zmian albo reorganizować kod w celu uzyskania zgodności z planowanymi zmianami.

Przykład refaktoryzacji

Zespół ShopSpinner postanowił dokonać dekompozycji swojego aktualnego stosu na wiele stosów i instancji stosów. Zaplanowana implementacja obejmuje jedną instancję stosu dla klastra kontenerów hostującego serwery WWW i drugą dla struktur współdzielonej sieci. Zespół będzie miał również po dwa stosy dla każdej usługi, jeden dla serwera aplikacji i skojarzonej z nim sieci, a drugi dla instancji bazy danych dla tej usługi (patrz rysunek 21-4).



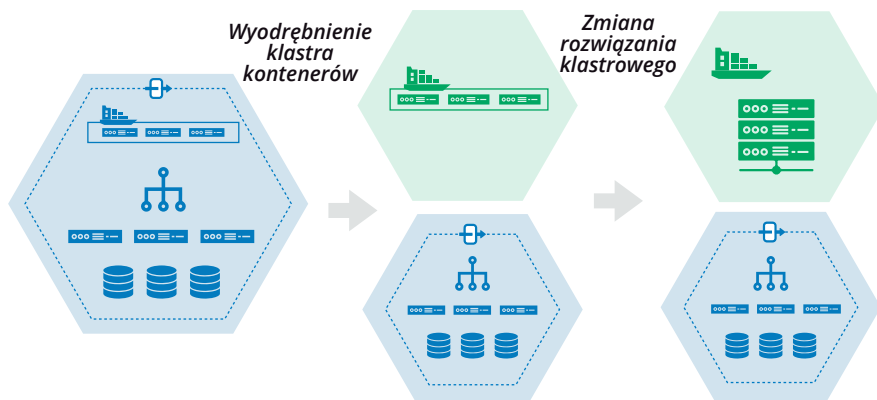
Rysunek 21-4 Plan dekompozycji stosu

Członkowie zespołu chcą również wymienić swój klastę kontenerów, przechodząc z klastra Kubernetes, który wdrażają sami na maszynach wirtualnych, na usługę Containers as a Service, oferowaną przez ich dostawcę chmury (patrz „Rozwiązania dla klastrów aplikacji” na stronie 224).

Zespół postanowił zaimplementować dekomponowaną architekturę metodą przyrostową. Pierwszym krokiem ma być wyodrębnienie klastra kontenerów do jego własnego stosu, a następnym wymiana rozwiązania kontenerowego wewnątrz stosu (rysunek 21-5).

⁴ <https://refactoring.com>

⁵ Kent Beck, w swoim artykule „SB Changes” (<https://oreil.ly/NWj5T>), opisuje przepływy pracy dla wprowadzania „dużych zmian w małych, bezpiecznych krokach”. Obejmują one wykonanie szeregu zmian, z których część porządkuje kod w celu przygotowania go na zmianę zachowania, a pozostałe dokonują zmiany zachowania. Sprawą kluczową jest to, że każda zmiana powoduje jedno albo drugie, nigdy to i to.



Rysunek 21-5 Plan wyodrębnienia i wymiany klastra kontenerów

Ten plan jest przykładem użycia refaktoryzacji w celu umożliwienia dokonania zmiany. Modyfikacja klastra kontenera będzie łatwiejsza w przypadku umieszczenia go w oddzielnym stosie niż wtedy, gdy stanowi część dużego stosu obejmującego inne elementy infrastruktury. Po wyodrębnieniu klastra do jego własnego stosu członkowie zespołu mogą zdefiniować jego punkty integracji dla reszty systemu. Mogą też napisać testy i procedury weryfikacji zapewniające czystość separacji i integracji. Pozwoli to zespołowi uzyskać pewność, że będzie mógł bezpiecznie zmienić zawartość stosu klastra.



Budowanie nowej wersji

Zamiast zmieniać stopniowo istniejący system produkcyjny, można zbudować niezależnie jego nową wersję i na koniec przełączyć na nią użytkowników. Zbudowanie nowej wersji może być łatwiejsze w przypadku drastycznej zmiany projektu i implementacji systemu. Ale nawet w takim przypadku dobrze jest doprowadzić nowy system do wersji produkcyjnej tak szybko, jak to tylko możliwe. Wyodrębnianie i przebudowywanie systemu fragmentami jest mniej ryzykowne niż budowanie wszystkiego jednocześnie. Ponadto pozwala szybciej testować i uzyskiwać korzyści z ulepszeń. Dlatego nawet gruntowna przebudowa może odbywać się metodą przyrostową.

Wypychanie niekompletnych zmian do produkcji

Jak dokonać znacznej zmiany systemu produkcyjnego za pomocą serii małych zmian przyrostowych nie przerywając działania usługi? Niektóre z tych małych zmian mogą same w sobie nie być przydatne. A usunięcie dotychczasowej funkcjonalności przed ukończeniem wszystkich zmian może nie być praktyczne.

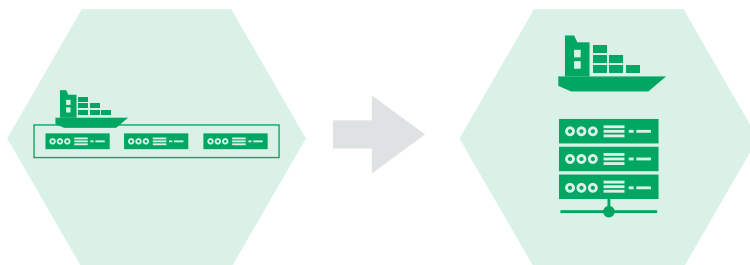
W podrozdziale „Przykład refaktoryzacji” na stronie 351 zostały przedstawione dwie zmiany przyrostowe – wyodrębnienie klastra kontenerów z jednego stosu do oddzielnego, a następnie zamianę rozwiązania klastrowego w nowym stosie. Każdy z tych kroków

stanowi sporą zmianę, więc jego implementacja wymaga prawdopodobnie serii wypchnięć mniejszych porcji kodu.

Wiele z tych małych zmian uniemożliwia jednak korzystanie z klastra. Trzeba więc znaleźć taki sposób ich wprowadzenia, aby zachować istniejący kod i jego funkcjonalność. W zależności od sytuacji można w tym celu stosować różne techniki.

Instancje równoległe

Drugi etap przykładu zamiany klastra zaczyna się w momencie, gdy oryginalne rozwiązanie ma swój własny stos, a kończy, gdy stare rozwiązanie klastrowe zostaje zastąpione nowym (patrz rysunek 21-6).



Rysunek 21-6 *Zamiana rozwiązania klastrowego*

Obecne rozwiązanie jest pakietową dystrybucją Kubernetes o nazwie KubeCan⁶. Zespół chce się przełączyć na FKS, zarządzaną usługę klastrową udostępnianą przez platformę chmurową⁷. Więcej informacji o klastrach jako usłudze i pakietowych dystrybucjach klastrów można znaleźć w podrozdziale „Rozwiązania dla klastrów aplikacji” na stronie 224.

Przekształcanie małymi krokami klastra KubeCan na FKS nie jest praktyczne. Ale zespół może uruchomić te dwa klastry równolegle. Istnieje kilka metod równoległego uruchamiania dwóch różnych stosów kontenerów z jedną instancją oryginalnego stosu.

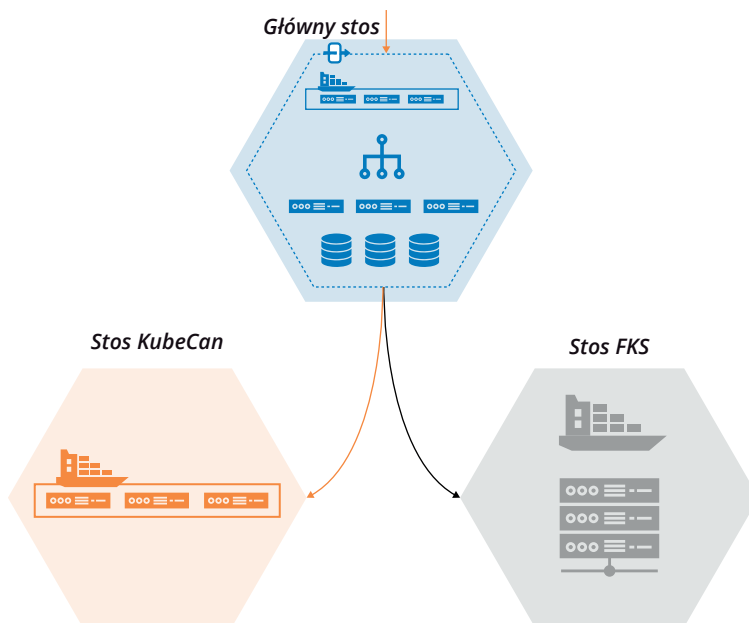
Jedna z opcji polega na zastosowaniu parametru w głównym stosie, służącego do wskazania stosu klastra, z którym ma nastąpić integracja (patrz rysunek 21-7).

Przy użyciu tej opcji jeden ze stosów jest włączony i będzie wykorzystywany do pracy na bieżąco. Drugi stos jest wyłączony, ale nadal istnieje. Zespół może testować drugi stos w pełnym środowisku operacyjnym, sprawdzać i opracowywać jego potok dostarczania oraz zestaw testów, a także integrować z pozostałymi częściami infrastruktury w każdym środowisku.

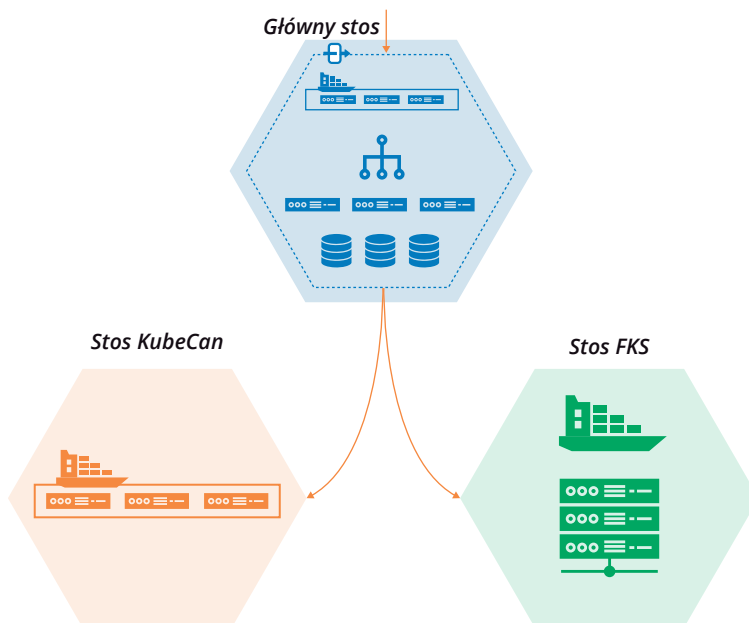
Inną opcją jest integracja obu stosów ze stosem głównym (patrz rysunek 21-8).

⁶ KubeCan to jeszcze jeden z fikcyjnych produktów preferowanych przez fikcyjny zespół ShopSpinner.

⁷ FKS to skrót od *Fictional Kubernetes Service* (fikcyjna usługa Kubernetes).



Rysunek 21-7 Jeden stos jest włączony, a drugi wyłączony



Rysunek 21-8 Implementacja każdego klastra działa we własnej instancji stosu



Po co w ogóle wyodrębniać stare rozwiązanie kontenerowe?

Zważywszy, że na koniec powstał samodzielny stos z nowym rozwiązaniem kontenerowym, prawdopodobnie można było pominąć etap wyodrębniania starego rozwiązania do jego własnego stosu. Wystarczyło po prostu od razu utworzyć nowy stos z nowym rozwiązaniem.

Wyodrębniając stare rozwiązanie łatwiej jest zapewnić zgodność nowego rozwiązania ze starym. Wyodrębniony stos wyraźnie definiuje sposób integracji klastra z pozostałą infrastrukturą. Używając wyodrębnionego stosu do produkcji gwarantujemy poprawność punktów integracji.

Dodawanie zautomatyzowanych testów i nowego potoku dla wyodrębnionego stosu zapewnia też szybkie wykrywanie sytuacji, w których wprowadzane zmiany powodują jakieś błędy.

W przypadku pozostawienia starego rozwiązania klastrowego w oryginalnym stosie i budowania nowego oddzielnie, ich zamiana byłaby uciążliwa. Do samego końca nie wiedzielibyśmy, czy nie podjęliśmy niewłaściwych decyzji dotyczących projektu i implementacji. W efekcie integracja nowego stosu z pozostałymi częściami infrastruktury oraz testowanie, debugowanie i usuwanie błędów zajęłoby trochę czasu.

Przy takim układzie można kierować obciążenie do obu stosów klastrów, dzieląc je na różne sposoby:

Procent obciążenia

Każdy stos przejmuje część obciążenia. Zazwyczaj starszy stos obsługuje na początku większość obciążenia, a nowy tylko niewielki procent, aby ocenić, jak działa. W miarę docierania się nowego stosu można stopniowo zwiększać jego obciążenie. Gdy nowy stos będzie już pomyślnie obsługiwał 100% obciążenia i wszyscy będą przygotowani, można usunąć stary stos. Ta opcja zakłada, że nowy stos ma te same możliwości co stary i nie występują żadne problemy z rozdzielaniem danych lub komunikatów między oba stosy.

Migracja usług

Migracja usług jedna po drugiej do nowego klastra. Obciążenia w stosie głównym, takie jak połączenia sieciowe lub komunikaty, są kierowane do dowolnej instancji stosu, na której działa odpowiednia usługa. Opcja ta jest szczególnie przydatna, gdy trzeba zmodyfikować aplikacje usługowe przenosząc je do nowego stosu. Wymaga to często bardziej złożonej integracji, może nawet między starym i nowym stosem klastra. Taka złożoność bywa uzasadniona w przypadku migracji złożonego portfela usług⁸.

8 Tego typu złożony scenariusz migracji z aplikacjami zintegrowanymi w dwóch hostujących klastrach jest powszechny w przypadku migracji dużej liczby aplikacji serwerowych hostowanych w centrum danych do chmurowej platformy hostingowej.

Partycjonowanie użytkowników

W niektórych przypadkach różne grupy użytkowników są kierowane do różnych implementacji stosów. Testerzy i użytkownicy wewnętrzni stanowią często pierwszą grupę. Przeprowadzają testy eksploracyjne i sprawdzają nowy system przed zaryzykowaniem dopuszczenia „prawdziwych” klientów. Czasami można pójść tą drogą i dać dostęp klientom chętnym do wykonania testów alfa i zapoznania się z usługami. Takie przypadki mają większy sens, gdy usługi uruchamiane z użyciem nowego stosu zostały zmienione w sposób zauważalny dla użytkowników.

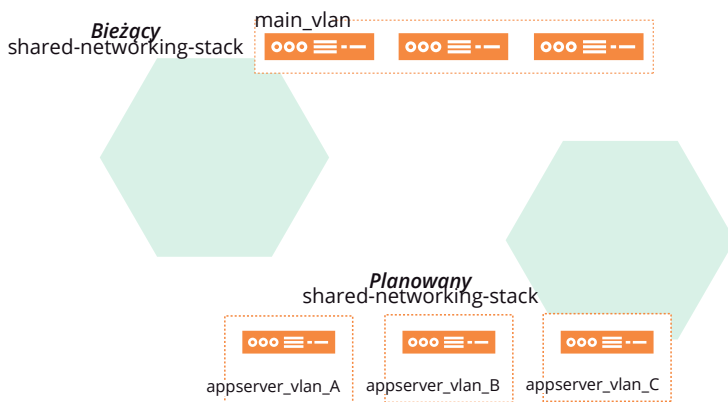
Warunkowe lub równoległe uruchamianie nowych i starych części systemu jest typem rozgałęzienia na podstawie abstrakcji. Stopniowe przenoszenie porcji obciążenia do nowych części systemu jest tzw. wydaniem kanarkowym, ang. *canary release*. *Dark launching*, czyli uruchamianie po ciemku, oznacza wstawianie nowej możliwości systemu do wersji produkcyjnej i uruchamianie jej w celu przetestowania przez zespół, ale bez poddawania faktycznemu obciążeniu.

Transformacje kompatybilne wstecz

Chociaż niektóre zmiany mogą wymagać budowy nowego komponentu i uruchamiania go obok starego aż do momentu ukończenia, jest wiele zmian, które można wprowadzać bezpośrednio w komponencie bez wpływania na użytkowników i komponenty konsumentów.

Nawet w przypadku dodawania lub zmieniania kodu dostarczanego do komponentów konsumentów, często można dodawać nowe punkty integracji zachowując w stanie niezmienionym dotychczasowe punkty integracji. Komponenty będące konsumentami mogą przełączyć się na korzystanie z nowych punktów integracji według własnego harmonogramu.

Na przykład zespół ShopSpinner planuje zmienić stos sieci współdzielonej, *shared-networking-stack*, aby zamienić pojedynczą sieć VLAN na trzy sieci VLAN (rysunek 21-9).



Rysunek 21-9 Zmiana pojedynczej sieci VLAN na trzy sieci VLAN

Stosy konsumentów, w tym `application-infrastructure-stack`, są integrowane z pojedynczą siecią VLAN zarządzaną przez stos sieci, przy użyciu jednej z metod odkrywania opisanej w podrozdziale „Wykrywanie zależności między stosami” na stronie 281. Kod `shared-networking-stack` eksportuje identyfikator VLAN, aby umożliwić jego odkrycie przez stosy konsumentów:

```
vlangs:
  - main_vlan
    address_range: 10.2.0.0/8

export:
  - main_vlan: main_vlan.id
```

Nowa wersja `shared-networking-stack` tworzy trzy sieci VLAN i eksportuje ich identyfikatory pod nowymi nazwami. Ponadto eksportuje jeden z identyfikatorów VLAN przy użyciu starego identyfikatora:

```
vlangs:
  - appserver_vlan_A
    address_range: 10.1.0.0/16
  - appserver_vlan_B
    address_range: 10.2.0.0/16
  - appserver_vlan_C
    address_range: 10.3.0.0/16

export:
  - appserver_vlan_A: appserver_vlan_A.id
  - appserver_vlan_B: appserver_vlan_B.id
  - appserver_vlan_C: appserver_vlan_C.id
# Deprecated
  - main_vlan: appserver_vlan_A.id
```

Dzięki zachowaniu starego identyfikatora zmodyfikowany stos sieci może nadal działać dla stosu infrastruktury konsumenta. Kod konsumenta należy tak zmodyfikować, aby używał nowych identyfikatorów, a z chwilą pozbycia się wszystkich zależności od starego identyfikatora, można usunąć go z kodu stosu sieci.

Przełączniki funkcji

Dokonując zmiany w komponencie często nadal trzeba używać istniejącej implementacji aż do zakończenia wprowadzania tej zmiany. Niektórzy rozgałęziają kod w kontroli źródła, pracując nad nową wersją w jednej gałęzi, a starą gałąź wykorzystując w produkcji. Takie podejście niesie ze sobą pewne problemy:

- Dbanie o to, aby zmiany pozostałych części komponentu były scalane z obydwoma gałęziami wymaga dodatkowej pracy, na przykład podwójnego poprawiania błędów.

- Ciągłe testowanie i wdrażanie obu gałęzi wymaga dodatkowego wysiłku i zasobów. Alternatywą jest mniej rygorystyczne testowanie jednej gałęzi, co jednak zwiększa ryzyko błędów i w efekcie przysparza dodatkowej pracy.
- Gdy zmiana jest ukończona, podmiana instancji produkcyjnych jest operacją przypominającą „wielki wybuch”, o dużym ryzyku niepowodzenia.

Bardziej efektywna jest praca nad zmianami w głównej bazie kodu, bez rozgałęziania. Wykorzystując przełączniki funkcji można przełączać implementację kodu w zależności od środowiska. W środowiskach używanych do testowania przełącznik powoduje wybranie nowego kodu, a w środowisku produkcyjnym kodu dotychczasowego. Do wskazania kodu, który ma być zastosowany do danego środowiska, można używać parametru konfiguracyjnego stosu (zgodnie z opisem w rozdziale 7).

Po zakończeniu dodawania sieci VLAN do stosu `shared-networking-stack`, co zostało już opisane, zespół ShopSpinner musi zmienić stos `application-infrastructure-stack`, aby zaczął wykorzystywać nowe sieci VLAN. Okazuje się, że nie jest to tak prosta zmiana, jak się początkowo wydawało.

Stos aplikacji definiuje trasy sieciowe specyficzne dla aplikacji, wirtualne adresy IP (VIP) modułów równoważnia obciążenia i reguły zapory. Jest to bardziej skomplikowane, gdy serwery aplikacji są hostowane w różnych sieciach VLAN, zamiast w jednej.

Implementacja kodu i testów dla tej zmiany zajmie członkom zespołu kilka dni. Nie jest to tak długo, aby decydować się na utworzenie oddzielnego stosu, jak zostało opisane w podrozdziale „Instancje równoległe” na stronie 353. Ale zespół dba o wypychanie zmian przyrostowych do repozytorium w trakcie pracy, aby móc otrzymywać ciągle informację zwrotną z testów, w tym także testów integracji systemu.

Zespół postanowił dodać parametr konfiguracyjny do stosu `application-infrastructure-stack`, umożliwiający wybór różnych części kodu stosu, w zależności od tego, czy ma zostać użyta jedna sieć VLAN, czy wiele.

Poniższy fragment kodu stosu wykorzystuje trzy zmienne – `app_server_A_vlan`, `appserver_B_vlan` i `appserver_C_vlan` – do wskazania, która sieć VLAN ma zostać przypisana do każdego serwera aplikacji. Wartość każdej z tych zmiennych jest ustawiana inaczej w zależności od wartości parametru przełącznika funkcji `toggle_use_multi ple_vlans`:

```
input_parameters:
  name: toggle_use_multiple_vlans
  default: false

variables:
  - name: appserver_A_vlan
    value:
      $IF(${toggle_use_multiple_vlans} appserver_vlan_A ELSE main_vlan)
  - name: appserver_B_vlan
    value:
      $IF(${toggle_use_multiple_vlans} appserver_vlan_B ELSE main_vlan)
  - name: appserver_C_vlan
```

```

value:
    $IF(${toggle_use_multiple_vlans} appserver_vlan_C ELSE main_vlan)

virtual_machine:
    name: appserver-${SERVICE}-A
    memory: 4GB
    address_block: ${appserver_A_vlan}

virtual_machine:
    name: appserver-${SERVICE}-B
    memory: 4GB
    address_block: ${appserver_B_vlan}

virtual_machine:
    name: appserver-${SERVICE}-C
    memory: 4GB
    address_block: ${appserver_C_vlan}

```

Jeśli wartość `toggle_use_multiple_vlans` jest ustawiona na `false`, to wszystkie parametry `appserver_X_vlan` są ustawiane na identyfikator starej sieci VLAN, `main_vlan`. W przypadku wartości `true` każda zmienna jest ustawiana na jeden z identyfikatorów nowych sieci VLAN.

Ten sam parametr przełącznik jest używany w pozostałych częściach kodu stosu, w których zespół pracuje nad konfiguracją routingu i innych złożonych elementów.



Porady dotyczące przełączników funkcji

Porady dotyczące używania przełączników funkcji i przykłady kodu można znaleźć w artykule „Feature Toggles (*aka* Feature Flags)” Pete’a Hodgsona (<https://oreil.ly/sIUNA>). Od siebie muszę dodać kilka wskazówek.

Po pierwsze, należy minimalizować liczbę używanych przełączników. Przełączniki funkcji i konstrukcje warunkowe wprowadzają bałagan, który utrudnia zrozumienie kodu, jego utrzymanie i debugowanie. Powinny być stosowane jak najkrócej. Trzeba usuwać je jak najszybciej, razem z kodem warunkowym i zależnościami od starej implementacji. Jedyne przełączniki funkcji, który może pozostać po upływie paru tygodni, to prawdopodobnie parametr konfiguracyjny.

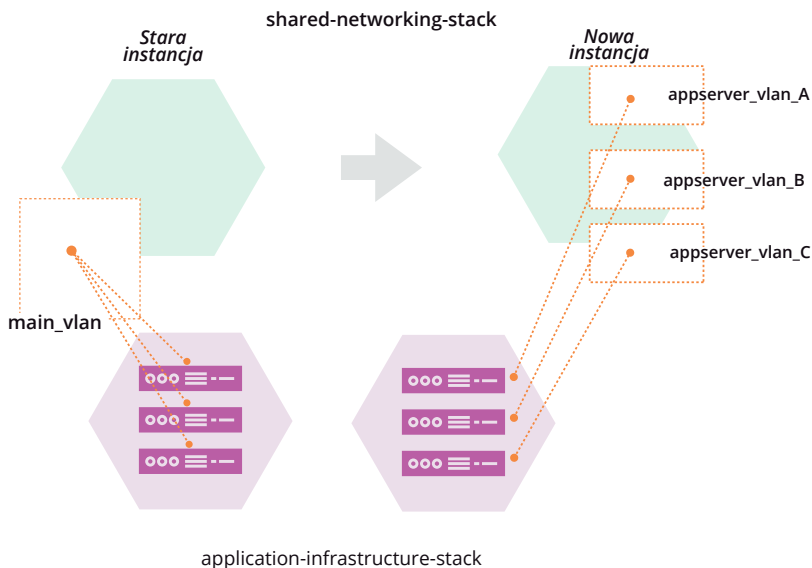
Przełączniki funkcji powinny mieć nazwy odpowiadające swojemu działaniu. Należy unikać nazw ogólnikowych, takich jak `kod_nowej_sieci`. Nazwa przełącznika we wcześniejszym przykładzie, `toggle_use_multiple_vlans`, wskazuje, że jest to przełącznik funkcji, a nie parametr konfiguracyjny i informuje czytelnika, że chodzi o używanie wielu sieci VLAN.

Nazwa powinna wyraźnie wskazywać, jak działa przełącznik. Patrząc na nazwę w rodzaju `toggle_multiple_vlans` albo jeszcze gorzej `toggle_vlans`, nie można mieć pewności, czy taki przełącznik włącza, czy też wyłącza kod wielu sieci VLAN. A to prowadzi do błędów, wynikających z nieprawidłowego używania kodu warunkowego przez niektóre osoby.

Zmiana działającej infrastruktury

Opisane techniki i przykłady wyjaśniają, jak zmieniać kod infrastruktury. Zmienianie działających instancji infrastruktury może być bardziej skomplikowane, zwłaszcza w przypadku zmieniania zasobów, których konsumentem jest inna infrastruktura.

Na przykład, jeśli zespół ShopSpinner zastosuje zmianę kodu `shared-networking-stack` powodującą zastąpienie pojedynczej sieci VLAN trzema sieciami VLAN, jak to zostało opisane w podrozdziale „Transformacje kompatybilne wstecz” na stronie 356, to co się stanie z zasobami z innych stosów, przypisanymi do pierwszej sieci VLAN (patrz rysunek 21-10)?



Rysunek 21-10 *Zmiana używanych struktur sieciowych*

Zastosowanie kodu sieci usuwa `main_vlan` razem z trzema instancjami serwerów. W działającym środowisku zniszczenie tych serwerów lub odłączenie ich od sieci spowoduje przerwanie wszystkich świadczonych przez nie usług.

Większość platform infrastruktury odmówi zniszczenia struktury sieciowej z dołączonymi do niej instancjami serwerów, czyli taka operacja się nie powiedzie. Jeśli stosowany kod zmiany powoduje również usunięcie lub zmiany innych zasobów, operacja może implementować zmiany w instancji pozostawiając środowisko w stanie pośrednim między starą i nową wersją kodu stosu. Taka sytuacja niemal zawsze jest niedobra.

Istnieje kilka sposobów realizacji tego typu zmiany działającej infrastruktury. Jednym jest zachowanie starej sieci VLAN, `main_vlan` i dodanie dwóch nowych sieci VLAN, `appserver_vlan_B` i `appserver_vlan_C`.

W ten sposób będziemy mieli trzy sieci VLAN, jak chcieliśmy, tylko nazwa jednej z nich będzie odbiegać od pozostałych. Zachowanie istniejącej sieci VLAN prawdopodobnie spowoduje również pozostawienie bez zmian innych jej aspektów, takich jak zakres adresów IP. I w tym przypadku można jednak zdecydować się na kompromis utrzymując oryginalną sieć VLAN mniejszą od pozostałych.

Tego typu kompromisy nie są dobre, ponieważ prowadzą do niespójnych systemów oraz powodują, że utrzymanie i debugowanie kodu staje się niewygodne.

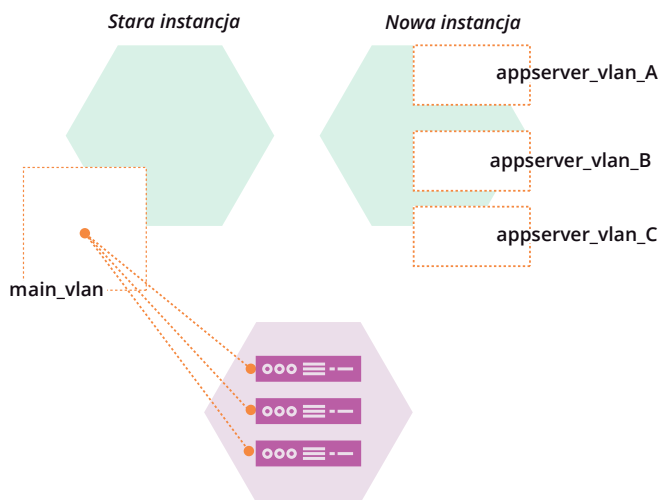
Można używać innych technik do zmiany działających systemów, pozostawiających je w czystym, spójnym stanie. Jedną z nich jest edycja zasobów infrastruktury przy użyciu operacji na infrastrukturze. Inną jest powiększanie i zmniejszanie zasobów infrastruktury.

Chirurgia infrastruktury

Niektóre narzędzia do zarządzania stosem, jak Terraform, umożliwiają dostęp do struktur danych mapujących zasoby infrastruktury na kod. Są to te same struktury danych, które są używane we wzorcu wyszukiwania danych w stosie przeznaczonym do wykrywania zależności (patrz „Wzorzec: wyszukiwanie danych w stosie” na stronie 285).

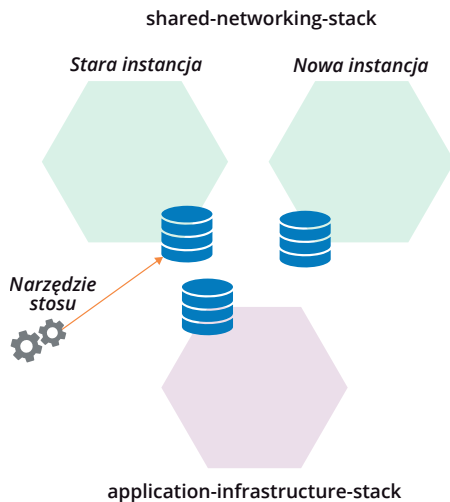
Niektóre narzędzia stosu (ale nie wszystkie) udostępniają opcję edycji swoich struktur danych. Można ją wykorzystywać do wprowadzania zmian w działającej infrastrukturze.

Zespół ShopSpinner chce wykorzystać fikcyjne narzędzie stack do edycji struktur danych swojego stosu. Członkowie zespołu dokonają tą metodą zmiany swojego środowiska produkcyjnego tak, aby używało ono trzech nowych sieci VLAN. Najpierw utworzą drugą instancję stosu `shared-networking-stack` z nową wersją kodu (patrz rysunek 21-11).



Rysunek 21-11 Równoległe instancje produkcyjnego stosu sieci

Każda z tych trzech instancji stosów – instancja `application-infrastructure-stack` oraz stara i nowa instancja `shared-networking-stack` – zawiera strukturę danych wskazującą, które zasoby w platformie infrastruktury należą do tego stosu (patrz rysunek 21-12).



Rysunek 21-12 Każda instancja stosu ma własną strukturę danych stosu

Zespół ShopSpinner przeniesie `main_vlan` ze struktur danych starej instancji stosu do struktur danych nowej instancji stosu, a następnie użyje jej do zastąpienia `appserver_vlan_A`.

Sieć VLAN w platformie infrastruktury nie zmieni się w żaden sposób, a instancje serwerów pozostaną całkowicie nietknięte. Zmiany te będą w całości ćwiczeniem z księgowości w strukturach danych narzędzia stosu.

Aby przenieść `main_vlan` ze starej do nowej instancji stosu, zespół musi uruchomić polecenie stosu `stack`:

```
$ stack datafile move-resource \  
    source-instance=shared-networking-stack-production-old \  
    source-resource=main_vlan \  
    destination-instance=shared-networking-stack-production-new Success: Resource  
moved
```

Następnym krokiem jest usunięcie `appserver_vlan_A`. Sposób wykonania tego różni się w zależności od faktycznie używanego narzędzia do zarządzania stosem. Fikcyjne polecenie `stack` pozwala wykonać tę operację niesłychanie prosto. Uruchomienie poniższego polecenia niszczy sieć VLAN w platformie infrastruktury i usuwa ją z pliku struktur danych:

```
$ stack datafile destroy-resource \  
    instance=shared-networking-stack-production-new \  
    resource=appserver_vlan_A  
Success: Resource destroyed and removed from the datafile
```

Zauważmy, że członkowie zespołu nie usunęli `appserver_vlan_A` z kodu źródłowego stosu, jeśli więc teraz zastosują ten kod do instancji, utworzą tę sieć ponownie. Ale tego nie zamierzają. Zamiast tego uruchomią polecenie zmieniające nazwę zasobu `main_vlan`, przeniesionego ze starej instancji stosu:

```
$ stack datafile rename-resource \  
    instance=shared-networking-stack-production-new \  
    from=main_vlan \  
    to=appserver_vlan_A Success: Resource renamed in the datafile
```

Po zastosowaniu przez zespół kodu `shared-networking-stack` do nowej instancji nic nie powinno się zmienić. Jeśli o to chodzi, wszystko, co jest w kodzie, jest również w instancji.

Należy pamiętać, że możliwość edycji i przenoszenia zasobów między stosami zależy całkowicie od narzędzia do zarządzania stosem. Większość narzędzi oferowanych przez dostawców chmur, przynajmniej gdy to piszę, nie daje możliwości edycji struktur danych stosu⁹.

Łatwo jest popełnić błąd podczas edycji struktur danych stosu, dlatego ryzyko spowodowania przestoju jest duże. Można napisać skrypt w celu implementacji poleceń i przetestować go we wcześniejszych środowiskach potoku. Ale tego typu edycje nie są idempotentne. Zakładają konkretny stan początkowy i jeśli coś się zmieni, to uruchomienie skryptu może dać nieprzewidywany efekt.

Przeglądanie struktur danych stosu może być przydatne do debugowania, ale należy unikać ich edycji. Taka edycja bywa konieczna w przypadku szukania przyczyny przestoju. Ciśnienie towarzyszące takim sytuacjom sprawia jednak, że często jeszcze łatwiej jest popełnić błąd. Nie należy wpadać w nawyk edytowania struktur danych stosu. Za każdym razem, gdy dojdzie do takiej edycji, zespół powinien przeprowadzić drobiazgową autopsję (*blameless postmortem*)¹⁰ w celu zrozumienia, jak nie dopuścić do powtórzenia się tej sytuacji.

Bezpieczniejszym sposobem dokonywania zmian w działającej infrastrukturze jest użycie wzorca powiększania i zmniejszania.

Powiększanie i zmniejszanie

Zespoły zajmujące się infrastrukturą wykorzystują wzorzec powiększania i zmniejszania (zwany również Zmianą równoległą¹¹) do zmieniania interfejsów bez uszkodzania konsumentów. Chodzi o to, że zmiana interfejsu dostawcy wymaga dwóch działań: zmiany dostawcy, a następnie zmiany konsumentów. Wzorzec powiększania i zmniejszania polega na rozdzieleniu tych działań.

9 Zainteresowanych przykładami narzędzi, które obsługują edytowanie struktur danych stosu, odsyłam do dwóch narzędzi: polecenia `terraform mv` (<https://oreil.ly/JoJZN>) i polecenia `pulumi state` (<https://oreil.ly/U-o->).

10 <https://oreil.ly/qbx7x>

11 <https://oreil.ly/InnHG>

Istotą wzorca jest dodanie najpierw nowego zasobu, przy zachowaniu już istniejącego, a następnie zmienianie kolejnych konsumentów pod kątem nowego zasobu, a na koniec usunięcie starego, niepotrzebnego zasobu. Każda z tych zmian jest dostarczana przy użyciu potoku (patrz „Potoki dostarczania infrastruktury” na stronie 113), dzięki czemu może być dokładnie przetestowana.

Dokonywanie zmiany metodą powiększania i zmniejszania jest podobne do transformacji kompatybilnej wstecz (patrz „Transformacje kompatybilne wstecz” na stronie 356). Tamta metoda polegała na zamianie starego zasobu i przekierowaniu starego interfejsu na jeden z nowych zasobów. Ale zastosowanie nowego kodu do działającej instancji spowodowałoby próbę zniszczenia starego zasobu, co z kolei mogłoby zakłócić działanie dowolnego dołączonego do niego konsumenta albo po prostu zakończyć się niepowodzeniem. Dlatego koniecznych jest kilka dodatkowych działań.

Pierwszym krokiem, który musi wykonać zespół ShopSpinner w celu użycia wzorca powiększania i zmniejszania do zmiany sieci VLAN, jest dodanie nowych sieci VLAN do stosu `shared-networking-stack`, przy zachowaniu nadal `main_vlan`:

```
vlans:
  - main_vlan
    address_range: 10.2.0.0/8
  - appserver_vlan_A
    address_range: 10.1.0.0/16
  - appserver_vlan_B
    address_range: 10.2.0.0/16
  - appserver_vlan_C
    address_range: 10.3.0.0/16

export:
  - main_vlan: main_vlan.id
  - appserver_vlan_A: appserver_vlan_A.id
  - appserver_vlan_B: appserver_vlan_B.id
  - appserver_vlan_C: appserver_vlan_C.id
```

W odróżnieniu od techniki instancji równoległych („Instancje równoległe” na stronie 353) i operacji na infrastrukturze („Chirurgia infrastruktury” na stronie 361), zespół ShopSpinner nie dodaje drugiej instancji stosu, a jedynie zmienia istniejącą.

Po zastosowaniu tego kodu istniejące instancje konsumentów pozostają niezmienione – nadal są dołączone do `main_vlan`. Zespół może dodawać nowe zasoby do nowych sieci VLAN i także przełączać się na nie wprowadzając zmiany w konsumentach.

Sposób, w jaki można przełączyć konsumenta na nowe zasoby, zależy od konkretnej infrastruktury i platformy. W niektórych przypadkach można uaktualnić definicję zasobu, aby dołączyć go do nowego interfejsu dostawcy. W innych może być konieczne zniszczenie i odbudowanie zasobu.

Zespół ShopSpinner nie może ponownie przypisać istniejących instancji serwerów wirtualnych do nowych sieci VLAN. Może natomiast wykorzystać wzorzec powiększania

i zmniejszania do wymiany tych serwerów. Kod `application-infrastructure-stack` definiuje każdy serwer przy użyciu statycznego adresu IP, który kieruje ruch do serwera:

```
virtual_machine:
  name: appserver-${SERVICE}-A
  memory: 4GB
  vlan: external_stack.shared_network_stack.main_vlan

static_ip:
  name: address-${SERVICE}-A
  attach: virtual_machine.appserver-${SERVICE}-A
```

Pierwszym krokiem zespołu jest dodanie nowej instancji serwera podłączonej do nowej sieci VLAN

```
virtual_machine:
  name: appserver-${SERVICE}-A2
  memory: 4GB
  vlan: external_stack.shared_network_stack.appserver_vlan_A

virtual_machine:
  name: appserver-${SERVICE}-A
  memory: 4GB
  vlan: external_stack.shared_network_stack.main_vlan

static_ip:
  name: address-${SERVICE}-A
  attach: virtual_machine.appserver-${SERVICE}-A
```

Pierwsza instrukcja `virtual_machine` w tym kodzie tworzy nową instancję serwera o nazwie `appserver-${SERVICE}-A2`. Potok zespołu dostarcza tę zmianę do każdego środowiska. Nowa instancja serwera nie jest używana w tym momencie, ale zespół może dodać kilka zautomatyzowanych testów, żeby potwierdzić jej poprawne działanie.

Następnym krokiem zespołu jest przełączenie ruchu użytkowników na nową instancję serwera. Zespół wprowadza kolejną zmianę w kodzie, modyfikując instrukcję `static_ip`:

```
virtual_machine:
  name: appserver-${SERVICE}-A2
  memory: 4GB
  vlan: external_stack.shared_network_stack.appserver_vlan_A

virtual_machine:
  name: appserver-${SERVICE}-A
  memory: 4GB
  vlan: external_stack.shared_network_stack.main_vlan

static_ip:
  name: address-${SERVICE}-A
  attach: virtual_machine.appserver-${SERVICE}-A2
```

Wypchnięcie tej zmiany przez potok powoduje uaktywnienie nowego serwera i zatrzymanie ruchu do starego serwera. Zespół może sprawdzić, czy wszystko działa poprawnie i w razie problemów łatwo cofnąć tę zmianę, aby przywrócić stary serwer.

Gdy nowy serwer działa prawidłowo, zespół może usunąć stary serwer z kodu stosu:

```
virtual_machine:
  name: appserver-${SERVICE}-A2
  memory: 4GB
  vlan: external_stack.shared_network_stack.appserver_vlan_A

static_ip:
  name: address-${SERVICE}-A
  attach: virtual_machine.appserver-${SERVICE}-A2
```

Z chwilą wypchnięcia tej zmiany przez potok i zastosowania jej do wszystkich środowisk `stos application-infrastructure-stack` przestaje być zależny od sieci `main_vlan` w `shared-networking-stack`. Po zmianie infrastruktury wszystkich konsumentów zespół ShopSpinner może usunąć `main_vlan` z kodu stosu dostawcy:

```
vlan:
  - appserver_vlan_A
    address_range: 10.1.0.0/16
  - appserver_vlan_B
    address_range: 10.2.0.0/16
  - appserver_vlan_C
    address_range: 10.3.0.0/16

export:
  - appserver_vlan_A: appserver_vlan_A.id
  - appserver_vlan_B: appserver_vlan_B.id
  - appserver_vlan_C: appserver_vlan_C.id
```

W ten sposób zmiana sieci VLAN została zakończona i ostatnie pozostałości po `main_vlan` zostały usunięte¹².

Zmiany bez przestoju

Wiele technik opisanych w tym rozdziale wyjaśnia, jak stopniowo implementować zmiany. Najlepiej by było, abyśmy mogli wprowadzać zmiany do istniejącej infrastruktury bez zakłócania udostępnianych przez nią usług. Niektóre zmiany wymagają jednak niszczenia zasobów, a przynajmniej ich zmiany w taki sposób, który może przerwać dostępność usług. Jest kilka popularnych technik radzenia sobie z takimi sytuacjami.

12 Nie martw się, strach będzie utrzymywał lokalne systemy w ryzach (<https://oreil.ly/C-DDT>).

Zmiany niebiesko-zielone

Zmiana niebiesko-zielona obejmuje utworzenie nowej instancji, przełączenie użytkownika na nową instancję, a następnie usunięcie starej instancji. Konceptyjnie jest to podobne do wzorca powiększania i zmniejszania („Powiększanie i zmniejszanie” na stronie 363), który polega na dodawaniu i usuwaniu zasobów w ramach instancji komponentu, takiego jak stos. Jest to kluczowa technika implementowania infrastruktury niezmiennalnej (patrz „Infrastruktura niezmiennalna” na stronie 342).

Zmiany niebiesko-zielone wymagają mechanizmu obsługi przełączania obciążenia z jednej instancji na drugą, takiego jak moduł równoważenia obciążenia w przypadku ruchu w sieci. Wyrafinowane implementacje pozwalają „odcedzać” obciążenie, poprzez alokowanie nowych zadań do nowej instancji i czekanie na zakończenie wszystkich już alokowanych zadań przez starą instancję, a potem jej zniszczenie. Na przykład niektóre zautomatyzowane rozwiązania klastrowe serwerów i aplikacji udostępniają to jako funkcję, umożliwiając „aktualizacje kroczące” instancji w klastrze.

W przypadku infrastruktury statycznej zmiana niebiesko-zielona jest implementowana za pomocą dwóch środowisk. Jedno środowisko działa bez przerwy, a drugie czeka w gotowości na przyjęcie następnej wersji. Kolory *niebieski* i *zielony* podkreślają, że są to dwa równoważne środowiska, które działają po kolei, a nie środowisko podstawowe i środowisko pomocnicze.

Współpracowałem z organizacją, która zaimplementowała niebiesko-zielone centra danych. Proces wydania nowej wersji obejmował przełączenie obciążeń całego systemu z jednego centrum danych na drugie. Skala tego zadania okazała się tak trudna do ogarnięcia, że musieliśmy pomóc zaimplementować wdrożenie na mniejszą skalę, tak aby zmiana niebiesko-zielona dotyczyła tylko tej usługi, która była aktualizowana.

Ciągłość

W rozdziale 1 omówiłem podstawowe różnice między tradycyjnym podejściem do zarządzania infrastrukturą, z „epoki żelaza”, a podejściem nowoczesnym, z „epoki chmury” (patrz „Od epoki żelaza do epoki chmury” na stronie 4). W czasach, w których mieliśmy więcej do czynienia z urządzeniami fizycznymi i zarządzaliśmy nimi ręcznie, koszt dokonywania zmian był wysoki.

Wysoki był również koszt popełnienia błędu. Jeśli nie zdołałem wyposażyć nowego serwera w dostateczną ilość pamięci, zajmowało mi tydzień albo więcej zamówienie dodatkowego RAM-u, zanieśenie go do centrum danych, wyłączenie serwera i wyciągnięcie go z szafy rackowej, otwarcie pokrywy, włożenie kości, a następnie zamontowanie serwera z powrotem w szafie rackowej i uruchomienie go.

Koszt wprowadzenia zmiany przy użyciu metod z epoki chmury jest znacznie niższy, podobnie jak koszt i czas potrzebny do poprawienia błędu. Jeśli nie zapewnię serwerowi dostatecznej ilości pamięci, będę potrzebował kilku minut, aby to naprawić, edytując odpowiedni plik i stosując go do mojego serwera wirtualnego.

Podejścia z epoki żelaza do zagadnienia ciągłości kładą nacisk na zapobieganie. Stosują optymalizację pod kątem MTBF (*Mean Time Between Failure*), czyli średniego czasu między awariami, poświęcając szybkość i częstotliwość zmian. Podejścia z epoki chmury stosują optymalizację pod kątem MTTR (*Mean Time to Recover*), czyli *średniego czasu do odzyskania*. Niektórzy entuzjaści nowoczesnych metod wpadają w pułapkę uważając, że skupienie się na MTTR oznacza poświęcenie MTBF, ale jest to nieprawdą, jak wyjaśniłem w podrozdziale „Zarzut: musimy wybierać między szybkością i jakością” na stronie 8. Zespoły skupiające się na czterech kluczowych wskaźnikach (szybkości i częstotliwości zmian, MTTR oraz współczynniku niepowodzenia zmian, opisanych w podrozdziale „Cztery kluczowe wskaźniki” na stronie 10) uzyskują solidną wartość MTBF jako efekt uboczny. Nie chodzi o to, aby „poruszać się szybko i psuć”, ale raczej „poruszać się szybko i naprawiać”.

Jest kilka elementów służących zapewnieniu ciągłości w przypadku nowoczesnej infrastruktury. Prewencja, na której koncentrowały się praktyki zarządzania zmianami w epoce żelaza, jest istotna, ale automatyzacja i infrastruktura chmury pozwalają używać bardziej efektywnych, zwinnych praktyk inżynierskich (Agile) do ograniczania błędów. Ponadto możemy wykorzystywać nowe technologie i praktyki odzyskiwania i odbudowywania systemów do zapewnienia wyższego poziomu ciągłości, niż można było sobie wcześniej wyobrazić. A poprzez ciągłe uruchamianie mechanizmów dostarczania zmian i odzyskiwania systemów możemy zapewnić niezawodność i gotowość tych systemów na różnego rodzaju awarie.

Ciągłość poprzez zapobieganie błędom

Jak już wspomniałem, podejście z epoki żelaza do zarządzania zmianami było głównie prewencyjne. Ponieważ koszt naprawy błędu był wysoki, organizacje inwestowały mocno w zapobieganie błędom. Ze względu na dokonywanie zmian przede wszystkim ręcznie, zapobieganie polegało między innymi na zawężaniu grona osób, które mogły to robić. Jedni ludzie szczegółowo planowali i projektowali zmiany, a inni przeglądali je dokładnie i omawiali. Panowało przekonanie, że większa liczba ludzi poświęcających więcej czasu na wcześniejsze analizowanie zmian pozwoli zawczasu wychwycić błędy.

Jednym z problemów związanych z tym podejściem jest rozdzźwięk między dokumentami projektowymi a implementacją. Coś, co na diagramie wygląda na proste, w rzeczywistości może być skomplikowane. Ludzie popełniają błędy, zwłaszcza gdy wykonują obszerne, a przy tym nieczęste aktualizacje. Efekt jest taki, że tradycyjne, sporadyczne, pracownice zaplanowane zmiany dużych partii mają wysoki poziom awaryjności i często długi czas odzyskiwania.

Praktyki i wzorce opisywane w całej tej książce mają na celu zapobieganie błędom, ale nie kosztem częstotliwości i szybkości zmian. Zmiany zdefiniowane jako kod odzwierciedlają ich implementację lepiej, niż może to zrobić jakikolwiek diagram czy dokument projektowy. Ciągłe integrowanie, stosowanie i testowanie zmian podczas pracy jest dowodem ich przygotowania do produkcji. Wykorzystywanie potoku do testowania

i dostarczania zmian zapewnia, że żaden krok nie zostanie pominięty i wymusza spójność różnych środowisk. Efektem jest zmniejszenie prawdopodobieństwa wystąpienia awarii w środowisku produkcyjnym.

Najważniejszym spostrzeżeniem zwinnego tworzenia oprogramowania i infrastruktury jako kodu jest odwrócenie podejścia do zmian. Zamiast bać się zmian i unikać ich, jeśli to tylko możliwe, można zapobiegać błędom poprzez częste dokonywanie zmian. Jedynym sposobem lepszego wprowadzania zmian jest robienie tego często, nieustannie ulepszając swoje systemy i procesy.

Inne kluczowe spostrzeżenie mówi, że w miarę, jak systemy stają się coraz bardziej złożone, nasze możliwości replikacji i dokładnego testowania zachowania się kodu w środowisku produkcyjnym kurczą się. Musimy mieć świadomość, co możemy i czego nie możemy przetestować przed wdrożeniem do produkcji i jak ograniczać ryzyko poprzez lepszy wgląd w systemy produkcyjne (patrz „Testowanie w środowisku produkcyjnym” na stronie 119).

Ciągłość poprzez szybkie odzyskiwanie

Praktyki opisane do tej pory w tym rozdziale pomagają skrócić przestój. Ograniczanie wielkości zmian, dokonywanie ich stopniowo oraz testowanie przed wdrożeniem do produkcji, wszystko to pomaga obniżyć współczynnik niepowodzenia zmian. Ale nierozsądnie byłoby zakładać, że możemy całkowicie zapobiec błędom, dlatego musimy mieć również możliwość szybkiego i łatwego odzyskiwania po awarii.

Praktyki zalecane w całej tej książce ułatwiają odbudowę dowolnej części systemu. System składa się z luźno powiązanych komponentów, z których każdy jest zdefiniowany jako idempotentny kod. Można łatwo naprawić albo zniszczyć i odbudować dowolną instancję komponentu stosując ponownie jego kod. W przypadku odbudowy komponentu trzeba zapewnić ciągłość danych obsługiwanych przez komponent, co jest omówione w podrozdziale „Ciągłość danych w zmieniającym się systemie” na stronie 373.

W niektórych przypadkach platforma lub usługa potrafi automatycznie odbudować uszkodzoną infrastrukturę. Platforma infrastruktury lub środowisko wykonawcze aplikacji niszczy i odbudowuje poszczególne komponenty, które nie przejdą kontroli stanu. Ciągłe stosowanie kodu do instancji („Ciągłe stosowanie kodu” na stronie 342) powoduje automatyczne cofnięcie wszelkich odchyień od kodu. Można też ręcznie wyzwolić etap potoku („Potoki dostarczania infrastruktury” na stronie 113) w celu ponownego zastosowania kodu do uszkodzonego komponentu.

W innych scenariuszach awarii systemy te mogą nie potrafić rozwiązać problemu automatycznie. Instancja obliczeniowa może źle działać, ale w taki sposób, że będzie pozytywnie przechodzić kontrolę stanu. Element infrastruktury może przestać działać prawidłowo, zachowując jednocześnie zgodność z definicją kodu, więc ponowne zastosowanie kodu nic nie da.

Takie scenariusze wymagają dodatkowego działania w celu zastąpienia wadliwych komponentów. Można ustawić flagę dla komponentu, aby zautomatyzowany system uznał

go za uszkodzonego i w efekcie zniszczył, a potem zastąpił nowym. Albo, jeśli proces odzyskiwania wykorzystuje system, który ponownie stosuje kod, można samemu zniszczyć komponent i pozwolić systemowi zbudować nową instancję.

W przypadku każdego scenariusza sytuacji awaryjnej, wymagającego wykonania działania przez jakąś osobę, musimy upewnić się, że dysponujemy narzędziami, skryptami lub innymi mechanizmami prostymi w użyciu. Ludzie nie powinni wykonywać całej sekwencji działań; na przykład, tworzyć kopii zapasowej danych przed zniszczeniem instancji. Zamiast tego powinni uruchamiać zadanie, które za nich wykona wszystkie niezbędne kroki. Chodzi o to, aby w przypadku awarii nie trzeba było zastanawiać się, jak można prawidłowo odzyskać system.

Ciągłe odzyskiwanie po awarii

Podejścia do zarządzania infrastrukturą z epoki żelaza postrzegały *odzyskiwanie po awarii* jako zdarzenie wyjątkowe. Usunięcie awarii statycznego sprzętu wymaga przeniesienia obciążenia na oddzielny zestaw sprzętowy utrzymywany w stanie gotowości.

Wiele organizacji rzadko testuje operację odzyskiwania – najwyżej co kilka miesięcy, a w niektórych przypadkach raz w roku. Widziałem mnóstwo organizacji sporadycznie testujących proces przełączania awaryjnego. Założenie jest takie, że jeśli zajdzie taka potrzeba, zespół znajdzie sposób uruchomienia zapasowego systemu, nawet jeśli potrwa to kilka dni.

Ciągłe odzyskiwanie po awarii wykorzystuje te same procesy i narzędzia, które są używane do wyposażania i zmiany infrastruktury. Jak już zostało to opisane, można stosować kod infrastruktury w celu jej odbudowy po awarii, być może z jakąś dodatkową automatyzacją w celu uniknięcia utraty danych.

Jedna z zasad infrastruktury w epoce chmury mówi, aby zakładać, że systemy są zawodne („Zasada: zakładaj, że systemy są zawodne” na stronie 13). Nie można zainstalować oprogramowania na maszynie wirtualnej i oczekiwać, że będzie ono działać tak długo, jak chcemy. Nasz dostawca chmury może przenieść, zniszczyć albo wymienić maszynę lub system hosta w celu konserwacji, zastosowania poprawek zabezpieczeń albo aktualizacji. Trzeba więc być przygotowanym na wymianę serwera, gdy zajdzie taka potrzeba¹³.

Traktowanie odzyskiwania po awarii jak przedłużenia normalnych działań czyni tę operację bardziej niezawodną niż w przypadku uznawania jej za wyjątek. Zespół sprawdza narzędzia i proces odzyskiwania wiele razy dziennie w ramach swojej pracy nad zmianami kodu infrastruktury i aktualizacjami systemu. Jeśli ktoś dokona zmiany w skrypcie lub innym kodzie, która przerwie wyposażanie lub podczas aktualizacji dojdzie do utraty

13 Instancje obliczeniowe stały się bardziej niezawodne od pierwszych dni przetwarzania w chmurze. Pierwotnie nie można było wyłączyć instancji AWS EC2 i uruchomić jej później – po jej zatrzymaniu znikala na zawsze. Efemeryczny charakter obliczeń zmusił użytkowników chmury do zastosowania nowych praktyk uruchamiania niezawodnych usług w zawodnej infrastrukturze. To był początek infrastruktury jako kodu, inżynierii chaosu i innych praktyk infrastruktury z epoki chmury.

danych, zazwyczaj spowoduje to niepowodzenie potoku na etapie testowania, więc będzie można szybko zastosować poprawkę.

Inżynieria chaosu

Pionierem ciągłego odzyskiwania po awarii i zarządzania infrastrukturą w epoce chmury był Netflix¹⁴. Jego narzędzia Chaos Monkey i Simian Army¹⁵ posunęły koncepcję ciągłego odzyskiwania po awarii o krok dalej, udowadniając skuteczność mechanizmów ciągłości systemów firmy poprzez wstrzykiwanie błędów do systemów produkcyjnych. Rozwinęło się to w dziedzinę zwaną inżynierią chaosu, „dyscyplinę eksperymentowania z systemem w celu budowy zaufania do jego możliwości”¹⁶.

Aby było jasne, inżynieria chaosu nie polega na nieodpowiedzialnym powodowaniu przerw w usługach produkcyjnych. Chodzi o eksperymentowanie z określonymi scenariuszami awarii, jakie mogą przydarzyć się systemowi. Są to podstawowe testy produkcyjne, udowadniające prawidłowe działanie mechanizmów wykrywania i odzyskiwania. Celem jest uzyskanie szybkiej informacji zwrotnej w przypadku, gdy zmiana systemu wywoła skutek uboczny w postaci zakłócenia działania tych mechanizmów.

Planowanie na wypadek awarii

Awarie są nieuniknione. O ile można i należy podejmować kroki w celu zmniejszenia ich prawdopodobieństwa, trzeba także stosować środki, dzięki którym ich wystąpienia będą mniej szkodliwe i łatwiejsze w obsłudze.

Zespół powinien organizować warsztaty mapowania scenariuszy awarii, aby w wyniku burzy mózgów ustalać niepowodzenia, które mogą wystąpić i następnie planować środki zaradcze¹⁷. Można utworzyć mapę prawdopodobieństwa i skutków każdego z tych scenariuszy, opracować listę działań do podjęcia w przypadku tych scenariuszy, a następnie nadać im odpowiednie priorytety w wykazie prac zespołu.

Każdy scenariusz awarii wymaga zbadania kilku aspektów:

Przyczyny i zapobieganie

Jakie sytuacje mogą prowadzić do tej awarii i co można zrobić, aby zmniejszyć ryzyko ich wystąpienia? Na przykład serwerowi może zabraknąć miejsca na dysku w przypadku gwałtownego wzrostu zapotrzebowania. Można sobie z tym poradzić analizując wzorce użycia dysku i zwiększając jego wielkość, aby była wystarczająca w momencie wyższego poziomu użycia. Można również zaimplementować zautomatyzowane mechanizmy

14 Wczesne lekcje budowania bardzo niezawodnych usług na dużą skalę w chmurze publicznej można znaleźć w artykule „5 Lessons We’ve Learned Using AWS” (https://oreil.ly/_JjnV), napisanym w 2010 roku.

15 <https://oreil.ly/7aik3>

16 Ta definicja pochodzi z witryny internetowej poświęconej zasadom inżynierii chaosu (<https://principlesofchaos.org>).

17 Zobacz także „Failure mode and effects analysis” (<https://oreil.ly/Rf1pS>).

ciągłej analizy poziomów użycia i prognozowania, aby można było prewencyjnie zwiększać ilość miejsca na dysku w przypadku wahań wzorców. Kolejnym krokiem byłoby automatyczne dostosowywanie pojemności dysku do rosnącego wykorzystania.

Tryb niepowodzenia

Co się dzieje w momencie wystąpienia awarii? Co można zrobić, aby ograniczyć skutki bez udziału człowieka? Na przykład, jeśli danemu serwerowi brakuje miejsca na dysku, uruchomiona na nim aplikacja może nadal akceptować transakcję, ale ich nie zapisywać. To może być bardzo groźne, dlatego warto zmodyfikować taką aplikację, aby nie akceptowała żadnej transakcji, jeśli nie może jej zapisać na dysku. W wielu przypadkach zespoły nie wiedzą, co się stanie tak naprawdę, gdy wystąpi dany błąd. Byłoby idealnie, gdyby tryb awaryjny zachowywał pełną funkcjonalność systemu. Na przykład, gdy aplikacja przestaje odpowiadać, moduł równoważenia obciążenia może przestać kierować do niej ruch.

Wykrywanie

Jak wykryć wystąpienie awarii? Co można zrobić, aby wykryć ją szybciej, może nawet zanim wystąpi? Można stwierdzić brak miejsca na dysku w momencie awarii aplikacji i telefonu skarżącego się klienta do dyrektora. Lepiej jest otrzymać powiadomienie od systemu o awarii aplikacji. A jeszcze lepiej jest zostać poinformowanym o małej ilości miejsca na dysku, zanim go zupełnie zabraknie.

Naprawa

Jakie kroki należy podjąć, aby usunąć awarię? W niektórych scenariuszach, jak zostało to opisane wcześniej, systemy potrafią automatycznie rozwiązać problem, na przykład niszcząc i odbudowując nieodpowiadającą instancję aplikacji. W innych przypadkach konieczne jest wykonanie kilku działań w celu naprawy i restartu usługi.

Jeśli system automatycznie obsługuje scenariusz awarii, na przykład restartując nieodpowiadającą instancję obliczeniową, należy rozważyć głębszy scenariusz awarii. Dlaczego instancja przestała odpowiadać za pierwszym razem? Jak wykrywać i naprawiać kryjące się za tym problemy? W ciągu nie dłużej niż kilku dni można zwykle zauważyć, że instancje aplikacji są odtwarzane co kilka minut.

Planowanie na wypadek awarii jest procesem ciągłym. Za każdym razem, gdy w systemie wydarzy się jakiś incydent, również w środowisku programowania lub testowania, zespół powinien zastanowić się, czy nie pojawił się nowy scenariusz awarii, który wymaga zdefiniowania i opracowania planu.

Należy implementować testy potwierdzające scenariusze awarii. Na przykład, jeśli uważamy, że w sytuacji, gdy serwerowi brakuje miejsca na dysku, aplikacja przestaje akceptować transakcje, należy automatycznie dodać nowe instancje serwerów i ostrzec zespół, że jest potrzebny zautomatyzowany test sprawdzający ten scenariusz. Można to sprawdzić w ramach etapu potoku (testowanie dostępności w podrozdziale „Co należy testować w przypadku infrastruktury?” na stronie 102) lub przy użyciu eksperymentu chaosu.



Stopniowe ulepszanie ciągłości

Łatwo jest zdefiniować ambitne środki odzyskiwania, umożliwiające systemowi elegancką obsługę każdej możliwej awarii bez przerywania usługi. Nigdy nie spotkałem zespołu, który by miał czas i zasoby, aby zbudować nawet połowę tego, co by chciał.

Mapując scenariusze awarii i środki zaradcze można zdefiniować przyrostowy zestaw metod do zaimplementowania. Należy podzielić je na oddzielne wątki implementacji i ustalić ich priorytety na liście zadań, opierając się na prawdopodobieństwie każdego scenariusza, potencjalnych szkodach i koszcie implementacji. Na przykład, chociaż pięknie byłoby automatycznie zwiększać miejsce na dysku dla aplikacji, gdy jest go mało, otrzymywanie ostrzeżenia przed jego całkowitym wyczerpaniem jest już pierwszym cennym krokiem.

Ciągłość danych w zmieniającym się systemie

Wiele praktyk i technik epoki chmury dotyczących wdrażania oprogramowania i zarządzania infrastrukturą radośnie zaleca niszczenie co jakiś czas i powiększanie zasobów, jedynie napominając o problemie danych. Trzeba wybaczyć, jeśli ktoś uważa, że dla hipsterów DevOps cała idea danych to powrót do epoki żelaza – w końcu prawidłowa aplikacja dwunastoczynnikowa¹⁸ jest bezstanowa. Większość systemów w realnym świecie wykorzystuje dane i ludzie mogą być do nich mocno przywiązani.

Dane mogą stanowić wyzwanie w przypadku przyrostowych zmian systemu, jak to zostało opisane w podrozdziale „Wypytywanie niekompletnych zmian do produkcji” na stronie 352. Uruchamianie równoległych instancji infrastruktury magazynu może doprowadzić do niespójności, a nawet do uszkodzenia danych. Wiele podejść do przyrostowego wdrażania zmian bazuje na możliwości ich cofnięcia, co może nie być realne w przypadku zmian schematów danych.

Dynamiczne dodawanie, usuwanie i odbudowywanie zasobów infrastruktury obsługujących dane jest wyjątkowo trudne. Są jednak na to sposoby, zależne od sytuacji. Niektóre z nich to blokowanie, segregowanie, replikacja i ponowne ładowanie.

Blokowanie

Niektóre platformy infrastruktury i narzędzia do zarządzania stosem pozwalają blokować określone zasoby, aby nie zostały usunięte przez polecenia, które chciałyby je zniszczyć. W przypadku zastosowania tego ustawienia do elementu magazynu, narzędzie odmówi zastosowania zmiany tego elementu, pozostawiając członkom zespołu możliwość dokonania zmiany ręcznie.

¹⁸ <https://12factor.net/>

Wiąże się z tym jednak kilka problemów. W niektórych przypadkach, jeśli stosujemy zmianę do zasobu chronionego, narzędzie może pozostawić stos w stanie częściowo zmodyfikowanym i w efekcie spowodować przestój usług.

Ale najistotniejszym problemem jest to, że ochrona niektórych zasobów przed automatyzowanymi zmianami zachęca do ręcznego dokonywania zmian, a to z kolei pociąga za sobą towarzyszące temu błędy. Dużo lepiej jest znaleźć sposób automatyzacji procesu, który będzie bezpiecznie zmieniał infrastrukturę.

Segregowanie

Segregacja danych może polegać na oddzieleniu obsługujących je zasobów od innych części systemu; chociażby poprzez utworzenie dla nich osobnego stosu (przykład tego jest podany w podrozdziale „Wzorzec: mikrostos” na stronie 57). Można bezkarnie niszczyć i odbudowywać instancję obliczeniową jedynie odłączając i przyłączając ponownie jej wolumin dyskowy.

Trzymanie danych w bazie danych zapewnia jeszcze większą elastyczność, potencjalnie umożliwiając dodawanie wielu instancji obliczeniowych. Nadal potrzebna jest strategia utrzymania ciągłości danych dla stosu hostującego dane, ale zakres problemu jest już węższy. Można też całkowicie uwolnić się od ciągłości danych używając hostowanej usługi DBaaS.

Replikacja

W zależności od danych i sposobu zarządzania nimi, można je replikować między wieloma instancjami infrastruktury. Klasycznym przykładem jest klastr rozproszony bazy danych, w którym dane są replikowane między węzłami.

Przy zastosowaniu odpowiedniej strategii replikacji dane są ładowane ponownie do świeżo odbudowanego węzła z innych węzłów klastra. Taka strategia zawodzi w przypadku utraty zbyt wielu węzłów, co może mieć miejsce w efekcie poważnej awarii hostingu. Dlatego takie podejście sprawdza się jako pierwsza linia obrony, natomiast trudniejsze scenariusze awarii wymagają zastosowania innego mechanizmu.

Ponowne ładowanie

Najbardziej znanym rozwiązaniem zapewniającym ciągłość danych jest tworzenie kopii zapasowych i odzyskiwanie danych przy użyciu bardziej niezawodnej infrastruktury pamięci masowej. Przed odbudową infrastruktury hostującej dane trzeba najpierw wykonać ich kopię zapasową, a następnie załadować je do nowej instancji, gdy zostanie już utworzona. Można również wykonywać okresowe kopie zapasowe, wykorzystywane w scenariuszach odzyskiwania, aczkolwiek prowadzi to do utraty wszystkich danych zmienionych pomiędzy wykonaniem kopii zapasowej a odzyskaniem. Można zminimalizować ten efekt i być może nawet wyeliminować całkowicie za pomocą strumieniowania zmian

danych do kopii zapasowej, na przykład poprzez zapisywanie ich w dzienniku transakcji bazy danych.

Platformy chmurowe udostępniają różne usługi pamięci masowej, co zostało opisane w podrozdziale „Zasoby pamięci masowej” na stronie 27, o różnych poziomach niezawodności. Na przykład usługi obiektowej pamięci masowej, jak AWS S3, dają zwykle większą gwarancję trwałości danych niż usługi blokowej pamięci masowej, jak AWS EBS. Można więc implementować kopie zapasowe przy użyciu kopiowania lub strumieniowania danych do obiektowego woluminu pamięci masowej.

Należy automatyzować nie tylko proces wykonywania kopii zapasowej danych, ale także proces ich odzyskiwania. Wykorzystywana platforma infrastruktury może pozwalać łatwo to robić i to na różne sposoby. Na przykład można automatycznie wykonać migawkę woluminu magazynu danych na dysku przed dokonaniem w nim zmian.

Migawki woluminów dysku mogą być wykorzystywane do optymalizacji procesu dodawania węzłów do systemu, takiego jak klastery bazy danych. Zamiast tworzyć nowy węzeł bazy danych z pustym woluminem magazynu danych, można dołączyć go do kłona dysku innego węzła, przyspieszając w ten sposób synchronizację i przełączenie węzła w tryb online.

„Nieprzetestowane kopie zapasowe to to samo, co brak kopii zapasowych” – tak brzmi popularne powiedzenie w naszej branży. Jeśli ktoś postępuje zgodnie z praktykami infrastruktury jako kodu, to znaczy, że stosuje automatyczne testowanie różnych aspektów swojego systemu. W takim razie może robić to samo z kopiami zapasowymi. Warto przećwiczyć proces wykonywania i przywracania kopii zapasowej w ramach potoku lub jako eksperyment chaosu, niezależnie od tego, czy jest on stosowany w środowisku produkcyjnym.

Mieszane podejścia do ciągłości danych

Często najlepszym rozwiązaniem jest kombinacja segregacji, replikacji i ponownego ładowania. Segregowanie danych umożliwia bardziej elastyczne zarządzanie pozostałymi częściami systemu. Replikacja zapewnia dostępność danych przez większość czasu. A ponowne ładowanie danych jest zabezpieczeniem na wypadek bardziej ekstremalnych sytuacji.

Podsumowanie

Zagadnienie ciągłości jest często zbywane przez zwolenników nowoczesnych praktyk zarządzania infrastrukturą w epoce chmury. Najbardziej znane podejścia stosowane w celu zapewnienia niezawodnego działania systemów opierają się na założeniu z epoki żelaza, według którego dokonywanie zmian jest kosztowne i ryzykowne. Podejścia te zazwyczaj podważają zalety chmury, metod zwinnych i innych podejść skoncentrowanych na szybkim tempie zmian.

Mam nadzieję, że wyjaśniłem w tym rozdziale, jak myśleć w kategoriach epoki chmury tworzyć bardziej niezawodne systemy, i to nie pomimo szybkiego tempa zmian, lecz

właśnie dzięki niemu. Wykorzystując dynamiczny charakter nowoczesnych platform infrastruktury można stosować implementację kładącą nacisk na testowanie i spójność, wynikającą z praktyk inżynierii zwinnej. Efektem tego jest wysoki poziom przekonania, że można w sposób ciągły dostarczać ulepszenia do swojego systemu i traktować każde niepowodzenie jako okazję do nauki i doskonalenia.

A

„A Brief and Incomplete History of Build Pipelines”, post na blogu [113](#)

Accelerate State of DevOps Report [xvii](#), [6](#), [101](#)

ad hoc, stosowanie [342](#)

ADR (architecture decision record) [44](#)

AKS (Azure Kubernetes Service) [27](#), [224](#), [228](#)

Amazon [198](#)

Amazon DynamoDB [27](#)

Amazon Elastic Container Service for Kubernetes (EKS) [27](#), [224](#), [228](#)

Amazon Elastic Container Services (ECS) [27](#), [224](#), [228](#)

Amazon, S3 [28](#)

Ambler, Scott, Refactoring Databases [159](#)

Aminator (witryna internetowa) [203](#)

Andrew File System [28](#)

Ansible [34](#), [41](#), [166](#)

Ansible Cloud Modules (witryna internetowa) [48](#)

Ansible for OpenShift (witryna internetowa) [229](#)

Ansible Tower [93](#), [167](#)

antywzorzec

definicja [xviii](#)

dla stosów infrastruktury [52-58](#)

moduł niewspółdzielony [272](#)

moduł spaghetti [274-277](#)

moduł zaciemniający [270](#)

podwójne etapy stosu – trwałe
i efemeryczne [138](#)

ręczne ustawianie parametrów
stosu [74-76](#)

stos monolityczny [52-54](#)

stos wielu środowisk [62](#)

stosowanie dla zmiany [186](#)

środowiska kopiuj-wklej [63](#)

Apache Mesos (witryna internetowa) [155](#), [226](#)

Apache OpenWhisk (witryna internetowa) [241](#)

API

statyczna analiza kodu z użyciem [126](#)

testowanie z użyciem atrap [127](#)

API, wrota [29](#), [162](#)

aplikacja, dane [158-159](#)

aplikacja, klastrer hostingu [27](#)

aplikacja, pakiet [32](#)

aplikacje

części wdrażalne [151](#)

dostarczanie [308](#)

jako komponenty systemów
infrastruktury [21](#)

łączność z [159-162](#)

pakiety do wdrażania

w klastrach [156-157](#)

pakowanie do kontenerów [153](#)

testowanie z użyciem infrastruktury [309](#)

wdrażanie na serwerach [153](#)

wdrażanie przy użyciu kodu
infrastruktury [310](#)

wdrażanie usługi bezserwerowej
FaaS [157](#)

aplikacje (*cd.*)

- wdrażanie w klastrach aplikacji 155
- wdrażanie w klastrach serwerów 154
- aplikacje, klastry 223-242
 - dla nadzoru 237
 - dla zespołów 238
 - oddzielne dla etapów dostarczania 236
 - omówienie 223
 - rozwiązania dla 224-226
 - strategie dla 234-240
 - topologie stosów dla 226-234
 - wdrażanie aplikacji 155
- aplikacje, wzorzec stosu grupy 52, 54
- AppVeyor (witryna internetowa) 118
- Architectural Quantum 250
- architecture decision record (ADR) 44
- artefakty 315
- Artifactory (witryna internetowa) 316
- ASG (Auto Scaling Group) 26
- asynchroniczne komunikaty 29
- Atlantis 118, 338
- atrapy 107, 134
- automatyczne odzyskiwanie 177
- automatyczne skalowanie 177
- automatyzacja
 - opóźnienie 341
 - spirala strachu przed 19
- autopsja 346, 363
- awaria, odzyskiwanie po 371
- awaria, planowanie na wypadek 371
- awaryjne poprawki, proces 346
- awaryjny tryb 371
- AWS CodeBuild (witryna internetowa) 118
- AWS CodePipeline (witryna internetowa) 118
- AWS ECS Services (witryna internetowa) 156
- AWS Elastic BeanStalk 23, 28
- AWS Image Builder (witryna internetowa) 204
- AWS Lambda 27, 241
- AWS SageMaker 27
- AWS, podsieci 29
- Awspec (witryna internetowa) 129

- Azure App Service Plans (witryna internetowa) 156
- Azure Block Blobs 28
- Azure Cosmos DB 27
- Azure Devops 23
- Azure Functions 27, 241
- Azure Image Builder (witryna internetowa) 204
- Azure Kubernetes Service (AKS) 27, 224, 228
- Azure ML Services 27
- Azure Page Blobs 28
- Azure Pipelines (witryna internetowa) 118
- Azure Resource Manager (witryna internetowa) 48
- Azurite (witryna internetowa) 127

B

- BaaS (Backend as a Service) 241
- Bamboo (witryna internetowa) 117
- bare metal 26
- batect (witryna internetowa) 338
- baza danych 124
- baza kodu 35, 43, 49, 66, 86, 101, 111, 152, 167, 313, 340, 345
- Bazel (witryna internetowa) 321
- Beck, Kent 247, 351
- Behr, Kevin, The Visible Ops Handbook 6
- bez przestojów wdrażanie, jako ryzyko testowania w środowisku produkcyjnym 122
- bez przestojów, zmiany 366
- bez wpisów tajnych, autoryzacja 97
- bez współdzielenia, architektura 254
- bezpieczeństwo
 - jako usługa siatki usług 239
 - testowania 103
- bezpieczeństwo bez obwodu 28
- bezpośrednie połączenie 29
- bezserwerowość, koncepcja 27, 241
- biblioteki, tworzenie elementów stosu przy użyciu 267
- blameless postmortem* (autopsja) 346, 363
- blokowa pamięć masowa (wirtualne woluminy dyskowe) 28

- blokowanie danych 374
- Borg 155
- Bosh (witryna internetowa) 48
- BoxFuse (witryna internetowa) 118
- Buck (witryna internetowa) 321
- budowa, etap 125
- budowa, serwer 117
- BuildKite (witryna internetowa) 118
- Butler-Cole, Ben 189
- C**
- CaaS (Containers as a Service) 26, 352
- Campbell, Laine, Database Reliability Engineering 159
- can_connect, metoda 135
- Capistrano (witryna internetowa) 154
- CD (Continuous Delivery)
 - omówienie 99
 - potoki dostarczania
 - infrastruktury 113-120
 - powody 100-105
 - testowanie progresywne 109-113
 - testowanie w środowisku produkcyjnym 120
 - towarzyszące wyzwania 105-109
- CDC (consumer-driven contract),
 - testowanie 327
- CDK (Cloud Development Kit) 35, 39, 284, 286
- CDN (Content Distribute Network) 30
- Centos 215
- CFAR (Cloud Foundry Application Runtime) 226
- CFEngine 34, 166
- cf_nag (witryna internetowa) 126
- CFR (Cross-Functional Requirement) 102
- CFRs (Cross-Functional Requirements) 102
- Chaos Monkey 371
- checkov (witryna internetowa) 126
- Chef Community Cookbooks (witryna internetowa) 317
- Chef Infra Server (witryna internetowa) 93
- Chef Provisioning (witryna internetowa) 48
- Chef Server 167, 316-317
- Chef 34, 41, 43, 166, 168
- chmurowo agnostyczne 25
- chroot, polecenie 208
- CI 54
- CI, serwery 89
- ciągła synchronizacja konfiguracji,
 - wzorzec 187, 191
- ciągłe resetowanie stosu, wzorzec 141
- ciągłość
 - ciągłe odzyskiwanie po awarii 371
 - inżynieria chaosu 371
 - omówienie 367
 - planowanie na wypadek awarii 371
 - przez szybkie odzyskiwanie 369
 - przez zapobieganie błędom 368
 - przyrostowe ulepszanie 372
 - w zmieniających się systemach 374-375
- CircleCI (witryna internetowa) 118
- Clarity (witryna internetowa) 129
- Clay-Shafer, Andrew xvii
- Cloud Development Kit (CDK). *Zobacz* CDK (Cloud Development Kit)
- Cloud Foundry Application Runtime (CFAR) 226
- Cloud Native Computing Foundation 150
- CloudFormation 34, 41, 48-49, 66, 157, 266, 286
- CloudFormation Linter (witryna internetowa) 126
- CloudFoundry, manifesty aplikacji 41
- cloud-init (witryna internetowa) 193
- CMDB (Configuration Management Database) 91
- CNAB (Cloud Native Application Bundle) (witryna internetowa) 156
- Cobbler (witryna internetowa) 178
- ConcourseCI 88, 118
- Configuration Management Database (CMDB) 91
- Consul (witryna internetowa) 93, 161, 240
- Containers as a Service (CaaS) 26, 352
- Content Distribute Network (CDN) 30
- Continuous Delivery (CD). *Zobacz* CD (Continuous Delivery)

Continuous Delivery (Humble i Farley) 99,
113

Conwaya prawo 258, 297

CoreOS rkt (witryna internetowa) 153

Cron 167

Crowbar (witryna internetowa) 178

częstotliwość wdrażania, jako kluczowy
wskaźnik wydajności operacyjnej
i dostarczania oprogramowania 11

D

dane

testowanie 121

zarządzanie, jako ryzyko testowania
w produkcji 122

dane wyjściowe, jako etap potoku 115

dane, centrum 29, 39, 367

dane, schematy 158

dane, struktury 151, 158

dane, zbiory 121, 237

dark launching 356

Database Reliability Engineering (Campbell
i Majors) 159

DBaaS (Database as a Service) 28

DBDeploy (witryna internetowa) 158

dbmigrate (witryna internetowa) 158

DDD (Domain Driven Design) 279

DDNS (Dynamic DNS) 161

.deb pliki 310, 315

DebianPreseed (witryna internet.) 174, 209

Debois, Patrick xvii

definicja 4

deklaratywne języki infrastruktury 37

deklaratywne testy 106

deklaratywny kod

omówienie 35, 40, 43

testowanie kombinacji 107

testowanie zmiennego 107

wielokrotne używanie za pomocą
modułów 266

demokratyzacja jakości, w przepływie pracy
opartym na potoku 345

DevOps xvii, 4, 307

DI (dependency injection) 290-293

DIY, rejestry konfiguracji 94

dług techniczny 5

DNS (Domain Name System) 161

Do Better As Code (witryna
internetowa) 109

Docker (witryna internetowa) 153

Dojo (witryna internetowa) 338

dokumentacja, kod jako 43

Domain Driven Design (DDD) 279

Domain Name System (DNS) 161

domenowe pojęcia 249, 306

doozerd (witryna internetowa) 93

dopasowywanie zasobów, wzorzec 282-285

DORA 11

dostarczanie 313-330

budowanie infrastruktury, projekty 314

czas realizacji, jako kluczowy wskaźnik
wydajności operacyjnej i dostarczania
oprogramowania 11

integrowanie projektów 317-327

konfiguracja i skrypty usług 32

omówienie 313

pakowanie kodu infrastruktury jako
artefaktów 315

środowiska dla 60

usługi i oprogramowanie
potoków 117-120

używanie repozytoriów do dostarczania
kodu infrastruktury 315-317

dostawcy 133, 249

dostępność

jako usługa siatki usług 239

klastrów aplikacji 236

testowania 103

droga do produkcji 60

Drone (witryna internetowa) 118

Dropwizard (witryna internetowa) 310

DRY (Don't Repeat Yourself), zasada 247

DSL (domain-specific language) 40

dwunastoczynnikowa metodologia (witryna
internetowa) 150, 374

Dynamic DNS (DDNS) 161

działanie, jako etap potoku 114

dzienniki zmian 301

E

ECS (Amazon Elastic Container Services) 27, 224, 228
 EDGE, model 17
 EDGE: Value-Driven Digital Transformation (Highsmith, Luu i Robinson) 17
 efemeryczne instancje 109
 efemeryczny stos testowy, wzorzec 137
 EKS (Amazon Elastic Container Service for Kubernetes) 27, 224, 228
 entr, narzędzie 101
 Envoy (witryna internetowa) 240
 epoka chmury 13
 epoka żelaza 4
 etcd (witryna internetowa) 93
 „Evolutionary Database Design” (Sadalage) 159
 Extreme Programming (XP) 99

F

FaaS (Function as a Service)
 bezserwerowe środowiska wykonawcze 27
 infrastruktura dla bezserwerowych 240-242
 omówienie 178
 wdrażanie aplikacji bezserwerowych 157
 Fabric (witryna internetowa) 154
 Facebook 299
 Facts and Fallacies of Software Engineering (Glass) 273
 „fan-in”, projekt potoku 324
 Farley, David, Continuous Delivery 99, 113
 fasadowy moduł, wzorzec 268-270, 272, 274, 277
 FCS (Fictional Cloud Service) xviii, 205
 Feathers, Michael, Working Effectively with Legacy Code 256
 feniks, serwer 342
 Fission (witryna internetowa) 241
 5 Lessons We’ve Learned Using AWS (witryna internetowa) 371
 fizyczne serwery 26

FKS (Fictional Kubernetes Service) xviii, 354
 flagi funkcji 357
 Flyway (witryna internetowa) 158
 Foreman (witryna internetowa) 178
 Fowler, Martin 40, 110, 135
 „Patterns for Managing Source Code Branches” 340
 Freeman, Steve, Growing Object-Oriented Software, Guided by Tests 349
 FSI (Fictional Server Image) xviii, 205
 Function as a Service (FaaS). Zobacz FaaS (Function as a Service)
 funkcja, rozgałęzianie 54
 funkcjonalność, testowania 103

G

GCE Persistent Disk 28
 GDPR (witryna internetowa) 237
 get_fixture(), metoda 135
 get_networking_subrange, funkcja 107
 get_vpc, funkcja 107
 Gillard-Moss, Peter 189
 git-crypt (witryna internetowa) 96
 GitHub 117-118
 GitLab 118
 GitOps (witryna internetowa) 120
 GitOps, metodologia 342
 Given, When, Then, format 106
 GKE (Google Kubernetes Engine) 27, 224, 228
 Glass, Robert, Facts and Fallacies of Software Engineering 273
 GoCD 88, 118
 Google 11, 299
 Google Cloud Deployment Manager (witryna internetowa) 48
 Google Cloud Functions 27, 241
 Google Cloud Storage 28
 Google Kubernetes Engine (GKE) 27, 224, 228
 Google ML Engine 27
 gotowość kodu do działania 34
 GPG (witryna internetowa) 76

granice

- dopasowywanie do cykli życia komponentów 256-257
- dopasowywanie do naturalnych wzorców zmian 256
- dopasowywanie do struktur organizacyjnych 258
- dopasowywanie do zasad bezpieczeństwa i nadzoru 262
- sieci 263
- twarde 162
- tworzenie wspierających odporność 258
- tworzenie wspierających skalowanie 259
- wytaczanie między komponentami 256-263
- Growing Object-Oriented Software, Guided by Tests (Freeman i Pryce) 349

H

- HashiCorp 94
- HCL, język konfiguracyjny 39
- Helm (witryna internetowa) 156, 311
- Helm, wykresy 41
- Heroku 23
- Highsmith, Jim, EDGE: Value-Driven Digital Transformation 17
- Hirschfield, Rob 234
- Hodgson, Pete 360
- Honeycomb 122
- hosts, pliki 161
- HPE Container Platform (witryna internetowa) 225
- Humble, Jez, Continuous Delivery 99, 113
- Hunt, Craig, TCP/IP Network Administration 30
- hybrydowa chmura 25

I

- IaaS (Infrastructure as a Service) 22
- idempotencja 38
- identyfikowalność, kodu 33
- imperatywny kod 35, 40, 43
- infrastruktura
 - bezpieczne zmienianie 347-375

- dla bezserwerowej usługi FaaS 240-242
- dla instancji konstruktorów 205
- dostarczanie 308
- dzielenie na poręczne fragmenty 107-109
- języki kodowania. *Zobacz* kodowanie, języki
- komponenty systemowe 21
- modularyzacja 250-254
- natywna dla chmury 150, 159
- niezmienialna 342
- oparta na aplikacjach 150
- platformy. *Zobacz* platformy
- potoki dostarczania 113-120
- projekty budowania 314
- rejestry narzędzi do automatyzacji 93
- testowanie przed integracją 310
- używanie skryptów do opakowywania narzędzi 327-330
- zasoby dla 25-30
- zmiana działającej 360-375
- infrastruktura jako kod
 - definiowanie 31
 - historia xvii
 - korzyści 6
 - omówienie 3
 - podstawowe praktyki 11
 - używanie w celu optymalizacji pod kątem zmian 6
 - zasady implementacji w przypadku definiowania 43
- infrastruktura natywna chmury 150, 159
- infrastruktura oparta na aplikacjach 150
- infrastruktura, operacje na 258, 361-363
- infrastruktura, skrypty 35
- infrastruktura, stosy 71-98, 123-145
 - budowanie środowisk z wieloma 67
 - instancje stosów 49
 - języki infrastruktury niskiego poziomu 50
 - języki infrastruktury wysokiego poziomu 51
 - kod źródłowy 49
 - konfigurowanie serwerów w 50
 - omówienie 32, 47

podgląd zmian 128
 przykład 123
 testowanie offline 125
 testowanie online 128-132
 wzorce cykli życia dla instancji
 testowych 136-143
 wzorce i antywzorce 52-58
 inotifywait, narzędzie 101
 Inspec (witryna internetowa) 129, 172
 integracja projektów podczas stosowania,
 wzorzec 324-327
 integracja projektu podczas budowania,
 wzorzec 319-322
 integracja projektu podczas dostarczania,
 wzorzec 322-324
 integracja, częstotliwość 341
 integracja, testy 305
 integracja, wzorce 340
 „Introduction to Observability” 122
 „Inverse Conway Maneuver” 258
 inżynieria chaosu 14, 122, 198, 371
 IP, adresy kodowane na stałe 160
 Istio (witryna internetowa) 240
 IT z Dzikiego zachodu 4
 iteracyjne zmiany 349

J

Jacob, Adam xvii
 jakość
 kodu 102
 ważniejsza od szybkości 9
 JavaScript 43
 jeden rejestr konfiguracji 96
 jednorazowe hasła 98
 jednorazowe wpisy tajne 98
 jednostka domeny infrastruktury,
 wzorzec 274, 277
 jednostkowe testy 110
 Jenkins 88, 117
 JEOS (Just Enough Operating System) 209
 języki. *Zobacz* kodowanie, języki
 języki, ich repozytoria jako źródła
 elementów serwera 165
 JSON 41

K

kanarkowe wdrażanie, wzorzec 356
 Kanies, Luke xvii
 KeePass (witryna internetowa) 76
 Keeper (witryna internetowa) 76
 Kim, Gene, The Visible Ops Handbook 6
 Kitchen-Terraform (witryna
 internetowa) 144
 klaster 224
Zobacz także aplikacje, klastry; serwery,
 klastry
 klaster jako usługa 224, 227
 klaster kontenerów serwerów WWW 123
 klucz-wartość, magazyn 28, 71, 93
 klucz-wartość, para 92
 kod
 ciągle stosowanie 342
 jakość 102
 w przepływie pracy opartym na
 potoku 344
 zarządzanie w systemie kontroli wersji
 (VCS) 33
 kod źródłowy, repozytorium 118, 317
 kod źródłowy, rozgałęzienia w przepływach
 pracy 340
 kodowane na stałe adresy IP 160
 kodowanie, języki
 dla komponentów stosu 266-268
 języki infrastruktury, deklaratywne 37
 języki infrastruktury,
 imperatywne 39-40
 języki infrastruktury niskiego
 poziomu 50
 języki infrastruktury specyficzne dla
 domeny (DSL) 40
 języki infrastruktury wysokiego
 poziomu 51
 języki ogólnego przeznaczenia 43
 języki proceduralne 35, 43, 107
 omówienie 34
 programowalne 39-40
 komponenty
 refaktoryzacja 135
 zakres testowany w ramach etapów 115

kompozycja, jako powód modularyzacji
stosów 265

konfiguracja

jako zadanie skryptowe 328

klastry aplikacji i 234

wybieranie narzędzi dla

eksternalizowanej 32

konfiguracja łańcuchowa, wzorce 93

konfiguracja, dryf 18, 341-342

konfiguracja, parametry 152

konfiguracja, rejestry 90, 93-96, 161

konfiguracja, wartości 306, 328

konfiguracyjne pliki 82

konkretne narzędzie, repozytorium 317

konstruktor, instancje 205

konstruktorzy

jako twórcy kodu 334

w przepływie pracy zespołu 333

konsumенты 133, 249

kontenery 26

jako kod 224

metody wyodrębniania 355

pakowanie aplikacji do 153

kopii wklej, środowiska, wzorzec 63

kops (witryna internetowa) 225

korelacja, kodu 34

kroczący szkielec 349

Kubeadm (witryna internetowa) 225

KubeCan 354

Kubeless (witryna internetowa) 241

Kubernetes 225, 231

Kubernetes borg (witryna internetowa) 226

Kubernetes Clusters 225

kubespary (witryna internetowa) 225

L

Language Server Protocol (LSP) 101

LastPass (witryna internetowa) 76

Lekkie nadzorowanie 17

LeRoy, Jonny 17

Lewis, James 248

Lightweight Resource Provider (LWRP) 168

Linkerd (witryna internetowa) 240

linting 126

Liquibase (witryna internetowa) 158

Localstack (witryna internetowa) 109, 127

logika klista, ryzyko 106

lokalna stacja robocza, używanie do

stosowania kodu 336

lokalne testowanie 143

LOM (lights-out management) 178

LSP (Language Server Protocol) 101

Luu, Linda, EDGE: Value-Driven Digital
Transformation 17

LWRP (Lightweight Resource
Provider) 168

Ł

ład organizacyjny, zgodność z 202

łączenie 319

łączność 152

M

MAAS (witryna internetowa) 178

Majors, Charity 63, 120

Database Reliability Engineering 159

Martin, Karen, Value Stream Mapping 335

McCance, Gavin 15

Mean Time Between Failure (MTBF) 367

Mean Time to Recover (MTTR) 367

Mean Time to Restore (MTTR) 11

Meszaros, Gerard, xUnit Test Patterns 109

metodologie ograniczone czasowo 100

MGs (Google Managed Instance

Groups) 26

Microsoft 299

mikrorepo 300

mikrostos, wzorzec 52, 57

Minimalizuj różnicowanie, zasada 17-19

modularność

infrastruktura a 250-254

projektowanie dla 246-250

moduł pakietowy, wzorzec 270, 274, 277

Molecule (witryna internetowa) 144

monitorowanie

jako ryzyko testowania w środowisku
produkcyjnym 122

jako usługa siatki usług 239

monolityczny stos
 dla pakietowych rozwiązań
 klastrowych 228
 potoki dla 229-232
 używanie klastra jako usługi 227
 monolityczny stos, antywzorzec 52-54
 monorepo 299, 321-322
 moto (witryna internetowa) 109
 Mountain Goat Software (witryna internetowa) 101
 możliwość wielokrotnego wykorzystania
 jako przyczyna modularyzacji
 stosów 265
 kodu 12
 w przepływie pracy opartym na
 potoku 344
 .msi pliki 310, 315
 MTBF (Mean Time Between Failure) 367
 MTTR (Mean Time to Recover) 367
 MTTR (Mean Time to Restore) 11

N

na gorąco 179
 nadmierne inwestowanie 120
 nadrzędne zależności 133
 nadzór
 kanały, w przepływie pracy opartym na
 potoku 345
 klastry aplikacji 237
 obrazy serwerów i 221
 w przepływie pracy opartym na
 potoku 344-346
 najlepsze praktyki xviii
 najpierw chmura, strategia xiii
 narzędzia
 do budowania obrazów serwerów 203
 do orkiestracji testów 144
 wybieranie dla eksternalizacji
 konfiguracji 32
 National Institute of Standards and
 Technology (NIST) 22
 natychmiastowe testowanie 101
 Netflix 203, 371
 Network File System 28

Newman, Sam 113
 Nexus (witryna internetowa) 316
 NFRs (Non-Functional Requirements) 102
 niebiesko-zielone wdrażanie, wzorzec 366
 niedoinwestowanie 120
 niestandardowe pakiety, jako źródło dla
 serwerów 166
 niewspółdzielony moduł, antywzorzec 272
 niezmiennalna infrastruktura 342
 niezmiennalny serwer, wzorzec 187,
 189-191, 321
 nieznane niewiadome 121
 niskiego poziomu języki infrastruktury 50
 NIST (National Institute of Standards and
 Technology) 22
 Nomad (witryna internetowa) 226
 Normalizacja dewiacji 9
 NuGet 315

O

1Password (witryna internetowa) 76
 O'Reilly (witryna internetowa) 344
 obciążenie, procent 356
 obiektów magazyn 28
 obliczenia w chmurze 22
 obliczeniowe instancje 371
 obliczeniowe zasoby 26
 obserwowalność
 jako ryzyko testowania w środowisku
 produkcyjnym 122
 jako usługa siatki usług 239
 OCI Registry As Storage (ORAS) 316
 Octopus Deploy (witryna internetowa) 311
 oddzielanie zagadnień 168
 oddzielanie zależności 291
 oddzielny materiał, jako źródło dla
 serwerów 166
 odgrzewanie obrazów serwerów 210
 odporność na błędy, w przypadku wielu
 środowisk produkcyjnych 60
 offline, budowanie obrazów 203, 207
 offline, testowanie 109, 125
 ogólnego przeznaczenia, języki
 programowania 43

ogólnego przeznaczenia rejestr konfiguracji, produkty 93
ogólny magazyn plików, repozytorium 317
okresowe odbudowywanie stosu, wzorzec 140
omówienie 13
online, budowanie obrazów 203-206
online, testowanie 109, 128-132
opakowujące skrypty, upraszczanie 329
opakowujący stos, wzorzec 64, 67, 74, 84-87
Open Application Model (witryna internetowa) 280
OpenFaaS (witryna internetowa) 241
OpenShift (witryna internetowa) 225
OpenStack Cinder 28
OpenStack Heat (witryna internetowa) 48
OpenStack Swift 28
operacyjne usługi 32
operatywność testowania 105
oprogramowanie CD 118
oprogramowanie, potok dostarczania 117-120
ORAS (OCI Registry As Storage) 316
organizacja
 kodu według pojęć domeny 306
 omówienie 297
 pliki wartości konfiguracyjnych 306
 projektów 297-302
 repozytoriów 297-302
 różne rodzaje kodu 302-307
 zarządzanie kodem infrastruktury i aplikacji 307-312
orkiestracja, jako zadanie skryptowe 328
ostateczne testowanie 101
Osterling, Mike, Value Stream Mapping 335

P

PaaS (Platform as a Service) 23, 27
Packer (witryna internetowa) 203
pakiety
 instalowanie popularnych 202
 jako zadanie skryptowe 328
pamięć masowa, zasoby 27
pamięć podręczna 30

Pants (witryna internetowa) 321
parametry stosu potoku, wzorzec 74, 76, 87-90
parametry, obsługa wpisów tajnych jako 96-98
parametry, pliki z 82
Parsons, Rebecca 40
partycjonowanie użytkowników 356
„Patterns for Managing Source Code Branches” (Fowler) 340
PCI, standard 134, 237, 262
pieczenie obrazów serwerów 182, 210
pieczenie serwerów 191
Pilnuj, abyś mógł powtórzyć każdy proces, zasada 18
pionowe grupowanie 260
piramida testów 110-112
PKS (Pivotal Container Services) (witryna internetowa) 225
platforma, elementy 116
platformy 21-30
 dynamiczne 22
 jako komponenty systemów infrastruktury 22
 komponenty systemów infrastruktury 21
 obrazy serwerów dla różnych 219
 omówienie 21
Please (witryna internetowa) 321
pliki
 obsługa projektu 302
 wartość konfiguracyjna 306
Plumi for Teams (witryna internetowa) 338
pobieranie konfiguracji serwera, wzorzec 167, 189, 193
podrzędne zależności 133-134
podstawowy system operacyjny, jako źródło serwerów 165
podwójne etapy stosu – trwałe i efemeryczne, antywzorzec 138
polichmura 25
pomoc techniczna, w przepływie pracy zespołu 333
ponowne ładowanie danych 374
poprawianie 372

poprawianie serwerów 191
 potoki
 dla monolitycznych stosów klastrów aplikacji 229-232
 etapy 114
 nadzór w przepływie pracy opartym na potoku 344-346
 narzędzia 144
 obrazy serwerów a 215
 omówienie 89
 powielanie, unikanie 247
 powiększanie i pomniejszanie, wzorec 363-366
 poziome grupowania 260
 pożar opon 3
 praktyka xiv
 Prawo Demeter 249
 procent nieudanych zmian, jako kluczowy wskaźnik dostarczania oprogramowania i wydajności operacyjnej 11
 produkcja, wypychanie niekompletnych zmian 353-360
 programowalne, imperatywne języki infrastruktury 39
 programowanie w parach 101
 progresywne testowanie 109-113, 309
 progresywne wdrażanie, jako ryzyko testowania w produkcji 122
 projekt, pliki pomocnicze 302
 projekt, zapach 43
 projektanci, w przepływie pracy zespołu 333
 projekty
 integrowanie 317-327
 organizacja 297-302
 wiele 301
 promień wybuchu 54
 promocja, jako zadanie skryptowe 328
 proxy 29
 Pryce, Nat, Growing Object-Oriented Software, Guided by Tests 349
 prymitywy 25
 prywatne instancje infrastruktury 338

przełączniki funkcji 357
 przesunięcie w lewo 345
 przeznaczone do testów integracji, testy 305
 przeźroczystość, kodu 12
 przyczepka 161
 przyczyny i zapobieganie 371
 przyrostowe zmiany 349
 Pulumi 35, 39, 48, 286, 363
 Puppet Cloud Management (witryna internetowa) 48
 Puppet 34, 41, 166
 PuppetDB (witryna internetowa) 93
 Puppetmaster 167
 Python 43

R

Rancher RKS (witryna internetowa) 225
 Rebar (witryna internetowa) 178
 Red Hat Kickstart (witryna internetowa) 174, 209
 Refactoring Databases (Ambler and Sadalage) 159
 refaktoryzacja 135, 351
 referencyjne dane 151
 reguła kompozycji 248
 reguła trzech 273
 Reinerstein, Donald G., The Principles of Product Development Flow 348
 rejestr platformy, usługi 94
 rejestry
 konfiguracji 90, 93-96
 konfiguracji, jeden 96
 konfiguracji, wiele 96
 narzędzie automatyzacji infrastruktury 93
 parametrów stosu 90
 replikowanie danych 374
 repozytoria
 organizacja 297-302
 repozytorium kodu źródłowego 317
 repozytorium specyficzne dla narzędzia 317
 repozytorium w postaci ogólnego magazynu plików 317

repozytoria(cd.)

- używanie repozytoriów do dostarczania kodu infrastruktury 315-317
- wiele 301
- wyspecjalizowane repozytorium artefaktów 316
- repozytorium platformy, źródło dla serwerów 165
- repozytorium środowisk, jako źródło dla serwerów 165
- ręczne parametry stosu, antywzorzec 74-76
- Robert, Mike, „Serverless Architecture” 241
- Robinson, David, EDGE: Value-Driven Digital Transformation 17
- role
 - obrazy serwerów dla różnych 220
 - serwery 169
- routing, jako usługa siatki usług 239
- rozgałęzianie na podstawie abstrakcji 356
- rozgałęzianie ścieżki do produkcji, wzorce 340
- rozwiązywanie problemów, jako usługa siatki usług 239
- Rób tak, aby wszystko było odtwarzalne, zasada 14
- równoległe instancje 353
- równoważenie obciążenia, reguły 29
- .rpm pliki 310, 315
- Ruby 43
- ruch, testowanie a 121
- ryzyka, zarządzanie w przypadku testowania w środowisku produkcyjnym 121

S

- SaaS, usługi 118
- Sadalage, Pramod
 - „Evolutionary Database Design” 159
 - Refactoring Databases 159
- Salt Cloud (witryna internetowa) 48
- Salt Mine (witryna internetowa) 93
- Saltstack 34, 166, 194
- SDK (Software Development Kit) 35
- SDN (Software Defined Networking) 28

segregacja

- danych 374
- klastry aplikacji a 234
- wielu środowisk produkcyjnych 61
- semantyczne wersjonowanie 212
- Sentinel (witryna internetowa) 118
- ser szwajcarski, model testowania 112
- Server Message Block 28
- „Serverless Architecture” (Robert) 241
- Servermaker xix, 18, 35, 50, 168, 184, 206, 229
- Serverspec (witryna internetowa) 172
- Service Level Agreements (SLAs) 332
- Service Level Indicators (SLIs) 332
- Service Level Objectives (SLOs) 332
- serwer aplikacji 123
- serwer, kod
 - omówienie 151, 167
 - projektowanie 168
 - promowanie 169
 - stosowanie 192-196
 - testowanie 171-173
 - wersjonowanie 169
- serwery 163-185, 199
 - budowanie przy użyciu sieciowych narzędzi wyposażania 177
 - elementy konfiguracji 32
 - klonowanie „na gorąco” 179
 - kod konfiguracji 166
 - konfigurowanie platform w celu automatycznego tworzenia 177
 - konfigurowanie w stosach 50
 - migawki 179
 - naprawianie 191
 - odzyskiwanie po awarii 198
 - omówienie 163, 185
 - role 32, 169, 203
 - rzeczy trzymane na nich 164
 - stosowanie konfiguracji podczas budowania 184
 - tworzenie przy użyciu narzędzi do zarządzania stosem 175
 - tworzenie przy użyciu skryptów 175
 - używanie w stosach 252-253

- wdrażanie aplikacji 153
- wstępne budowanie 178
- zarządzanie zmianami, wzorce 186-191
- zdarzenia w cyklu życia 196-199
- źródło 165
- serwery, efektywności i 182
- serwery, instancje
 - aktualizowanie 212
 - konfigurowanie nowych 180-184
 - ręczne budowanie nowych 174
 - smażenie 181
 - tworzenie nowych 173-178
 - zastępowanie 197
 - zatrzymywanie i uruchamianie 196
- serwery, klastry 26, 154
- serwery, obrazy
 - budowanie 202-210
 - budowanie od podstaw 209
 - dla różnych platform infrastruktury 219
 - dla różnych ról 220
 - dla różnych systemów operacyjnych 219
 - dla sprzętu o różnej architekturze 220
 - etap budowania 215-217
 - etap dostarczania 218
 - etap testów 217
 - konfigurowanie 181
 - nadzór 221
 - narzędzia do budowania 203
 - obsługiwanie dużych zmian 214
 - odgrzewanie 210
 - omówienie 201
 - pieczenie 182, 210
 - pochodzenie 209
 - postać źródłowa 208-210
 - stos 209
 - testowanie 215
 - tworzenie czystych 180
 - używanie skryptów do budowy 206
 - używanie wielu 219-222
 - warstwowanie 220
 - wersjonowanie 211
 - wielu zespołów 214
 - współdzielenie kodu 222
 - zmienianie 210-215
- shellcheck (witryna internetowa) 330
- ShopSpinner 247-256, 259, 267, 281, 290, 306, 317, 322-327, 345, 348-349, 356-360
- sieciowe systemy plików (współdzielone woluminy sieciowe) 28
- sieciowe zasoby 28
- sieć, bloki adresów 23, 29
- sieć, granice 263
- sieć, reguły dostępu (reguły zapory) 29
- Simian Army 371
- skalowalność
 - klastry aplikacji a 234
 - testowania 103
 - w przypadku wielu środowisk produkcyjnych 60
- składnia, sprawdzanie 126
- skryptowe parametry, wzorzec 74, 76, 78-81
- skrypty
 - do budowania obrazów serwerów 206
 - infrastruktury 35
 - opakowujące 329
 - używanie do opakowywania narzędzi infrastruktury 327-330
 - używanie do tworzenia serwerów 175
- SLAs (Service Level Agreements) 332
- SLIs (Service Level Indicators) 332
- SLOs (Service Level Objectives) 332
- smażenie instancji serwerów 181
- Smith, Derek A. 190
- Software Defined Networking (SDN) 28
- Software Development Kit (SDK) 35
- Solaris JumpStart (witryna internetowa) 174, 209
- some-tool, polecenie 81
- sops (witryna internetowa) 96
- Soylent Green 332
- Spafford, George, The Visible Ops Handbook 6
- spaghetti moduł, antywzorec 274-277
- spakowana dystrybucja klastra 225
- specjaliści ds nadzoru
 - jako twórcy kodu 335
 - w przepływie pracy zespołów 333

spójność (kohezja)

spójność (kohezja) 246

spójność

kodu 12

w przepływie pracy opartym na
potoku 344

sprzęt, różne architektury, obrazy serwerów
dla 220

sprężenie 246

SRE (Site Reliability Engineer) 345

SREs (Site Reliability Engineers) 345

SRP (single responsibility principle) 248

Stackmaker *xix*

StackReference (witryna internetowa) 286

starsze silosy 258

State of DevOps Report (see Accelerate
State of DevOps Report)

statyczna analiza kodu

offline 126

za pomocą API 127

stos, instancje

omówienie 49, 71

przykład parametrów stosu 73

utwardzanie wpisów tajnych jako
parametrów 96-98

używanie parametrów stosu do tworzenia
unikatowych identyfikatorów 73

wzorce konfigurowania stosów 74-93

stos, narzędzie 90

stos, rejestry parametrów 90

stos, skrypt orkiestracji 78

stos, używanie narzędzi do zarządzania do
tworzenia serwerów 175

stos, używanie parametrów do tworzenia
unikatowych identyfikatorów 73

stos, wzorzec plików konfiguracyjnych 74,
81

stos, wzorzec rejestru parametrów 84, 90-
93, 290

stos, wzorzec wyszukiwania danych 285-
287, 290

stos, wzorzec zmiennych

środowiskowych 76-78

stosowanie każdej zmiany oddzielnie,
antywzorzec 186

stosuj proste parametry, zasada
projektowania 72

stosy, ich komponenty 265-281

języki infrastruktury 266-268

omówienie 265, 281

stosy jako komponenty 250-251

używanie serwerów 252-253

wzorce 268-279

zależności między 281-293

Strangler Application (witryna
internetowa) 135

strukturalne dane, magazyn 28

strumień wartości, mapowanie 335

Subnet 23

system operacyjny, repozytoria pakietów
jako źródło dla serwerów 165

systemy operacyjne, obrazy serwerów dla
różnych 219

szew 256

szybkość

serwery a 182

stawianie wyżej od jakości 9

szyfrowanie wpisów tajnych 96

Ś

ściśle sprzężenie 144

śnieżynka, system 15

środowiska 59-69

budowanie z użyciem wielu stosów 67

dostarczanie 60

konfigurowanie 61

omówienie 59

spójność 61

wiele produkcyjnych 60

wzorce budowania 62-67

środowiska uruchomieniowe aplikacji

cele dla 151

dane aplikacji 158-159

infrastruktura natywna dla chmury 150

infrastruktura oparta na aplikacjach 150

jako komponenty systemów

infrastruktury 22

łączność aplikacji 159-162

omówienie 149

pakiety wdrażania 152
 wdrażanie aplikacji na serwerach 153
 wstrzykiwanie wpisów tajnych 97
 środowisko, rozgałęzianie 64

T

Taraporewalls, Sarah 102
 Taskcat (witryna internetowa) 129
 TCP/IP Network Administration (Hunt) 30
 TDD (Test Driven Development) 99
 Team City (witryna internetowa) 117
 Terra 266, 286
 Terraform 34, 39, 41, 48-49, 66, 94, 284, 361
 Terraform Cloud 118, 338
 terraform fmt, polecenie 126
 terraform mv, polecenie 363
 Terraform Registry (witryna internetowa) 317
 Terraform, blokowanie stanu 337
 Terragrunt (witryna internetowa) 87
 Terratest (witryna internetowa) 129, 172
 Test Driven Development (TDD) 99
 Test Kitchen (witryna internetowa) 144
 testerzy
 jako twórcy kodu 335
 w przepływie pracy zespołu 333
 testowalność, jako powód modularyzacji stosów 265
 testowanie
 aplikacji z infrastrukturą 309
 infrastruktury przed integracją 310
 integracji 305
 jako zadanie skryptowe 328
 kodu serwera 171-173
 kombinacji kodu deklaratywnego 107
 lokalne 143
 natychmiastowe 101
 obrazów serwera 215
 offline 109
 online 109
 orkiestrowanie 143-145
 ostateczne 101
 progresywne 109-113, 309

projektowanie i 250
 przy użyciu atrap API 127
 używanie warunków początkowych testu do obsługi zależności 132-135
 wielu projektów 303
 wyniki 132
 zależności podrzędnych 134
 zmiennego kodu deklaratywnego 107
 testowanie kodu infrastruktury jest powolne, wyzwanie 107-109
 testy kodu deklaratywnego mają często małą wartość, wyzwanie 105-107
 tfllint (witryna internetowa) 126
 „The Goldilocks Zone of Lightweight Architectural Governance” 17
 The Principles of Product Development Flow (Reinerstein) 348
 The Visible Ops Handbook (Kim, Spafford and Behr) 6
 ThoughtWorks 118
 Tinkerbell (witryna internetowa) 178
 Tomcat 168, 186
 transcrypt (witryna internetowa) 96
 transformacja kompatybilna wstecz 356
 trasy 29
 TravisCI (witryna internetowa) 118
 trwałe instancje 109
 trwałe stos testowy, wzorzec 136
 Turing, języki kompletne w sensie 38
 twórcy narzędzi
 jako twórcy kodu 335
 w przepływie pracy zespołu 333
 Twórz rzeczy jednorazowe, zasada 15
 TypeScript 43

U

usług stos, wzorzec 52, 56-57
 usługa scentralizowana, stosowanie kodu z 337
 usługi platformy chmurowej 118
 usługi, migracja 356
 usługi, odkrywanie 160
 usługi, siatka 30, 238-240
 uwierzelnianie, bez wpisów tajnych 97

uwierzytelnianie, jako usługa siatki usług 239

użytkownicy
jak twórcy kodu 334
testowanie a 121
w przepływie pracy zespołu 333

V

Vagrant (witryna internetowa) 144

Value Stream Mapping (Martin and Osterling) 335

Vaughan, Diane 9

VCS (version control system) 33

Virtual Private Cloud (VPC) 23, 29

virtual private network (VPN) 29

Visual Basic for Applications 33

VLAN, sieci 23, 39, 281

VM (virtual machine) 15, 26

VMware 15, 21, 163, 202

Vocke, Ham 110

VPC (Virtual Private Cloud) 23, 29

VPN (virtual private network) 29

W

walidacja, reguły 32

warstwowanie obrazów serwerów 220

warstwy abstrakcji 280

wdrażanie, manifest 156

wdrażanie, pakiety 152

Weave Cloud (witryna internetowa) 156, 338

WeaveWorks (witryna internetowa) 120, 338

wersjonowanie

kodu serwera 169

obrazów serwera 211

widoczność, kodu 34

wiele rejestrów konfiguracji 96

wiele środowisk produkcyjnych 60

wielochmurowa strategia 25

wielokrotnego użytku stos, wzorzec 65-67

wieloprojektowe testy 303

wielu środowisk stos, antywzorzec 62

Willis, John xvii

Windows Containers (witryna internetowa) 153

Windows, plik odpowiedzi instalacji (witryna internetowa) 174, 209

wirtualizacja, technologia 23

Working Effectively with Legacy Code (Feathers) 256

wpisy tajne

jednorazowe 98

obsługiwanie jako parametrów 96-98

omówienie 89

szyfrowanie 96

usługa zarządzania 28

wstrzykiwanie podczas wykonywania zarządzanie 93

wrota 29

wskaźniki, wydajności operacyjnej i dostarczania oprogramowania 11

współbieżność, testowanie i 121

współdzielenie, jako powód modularyzacji stosów 265

wstrzykiwanie zależności (DI). Zobacz DI (dependency injection)

wycofywanie, kodu 34

wydajność

klastry aplikacji a 234

optymalizowanie 202

testowania 103

wykonanie, jako zadanie skryptowe 328

wykonawcze 151

wykrywanie 372

wyniki, testowanie 132

wyposażające skrypty 79

wypychanie konfiguracji serwera, wzorzec 167, 187, 189, 192

wysokiego poziomu języki infrastruktury 51

wyspecjalizowane repozytorium artefaktów 316

wyszukiwanie w rejestrze integracji, wzorzec 288-290

wyzwalacz, jako etap potoku 114

wzorze

budowania środowisk 62-67

ciągła synchronizacja konfiguracji 187, 191

ciągle resetowanie stosu 141

definicja *xviii*

dopasowywanie zasobów 282-285

efemeryczny stos testowy 137

integracja 340

integracja projektu podczas budowania 319-322

integracja projektu podczas dostarczania 322-324

integracja projektu podczas stosowania 324-327

jednostka domeny infrastruktury 274, 277

komponentów stosu 268-279

konfigurowania stosów 74-93

mikrostos 52, 57

moduł fasadowy 268-270, 272, 274, 277

moduł pakietowy 270, 274, 277

okresowa odbudowa stosu 140

parametry skryptowe 74, 76, 78-81

parametry stosu potoku 74, 76, 87-90

pliki konfiguracyjne stosów 74, 81-84

pobieranie konfiguracji serwera 167, 189, 193

powiększanie i pomniejszanie 363-366

rejestr parametrów stosu 84, 90-93, 290

rozgałęzianie ścieżki do produkcji 340

serwer *niezmienialny* 187, 189-191, 321

stos grupy aplikacji 52, 54

stos opakowujący 64, 67, 74, 84-87

stos usług 52, 56-57

stos wielokrotnego użytku 65-67

stosów infrastruktury 52-58

trwały stos testowy 136

wdrażanie kanarkowe 356

wdrażanie niebiesko-zielone 366

wypychanie konfiguracji serwera 167, 187, 192

wyszukiwanie danych w stosie 285-287, 290

wyszukiwanie w rejestrze integracji 288-290

zmienne środowiskowe stosu 76-78

X

XP (Extreme Programming) 99, 110

xUnit Test Patterns (Meszaros) 109

Y

YAGNI („You Aren't Gonna Need It”) 273

YAML 39, 41

Z

zabezpieczenia, utwardzanie 202

zaciemniający moduł, antywzorzec 270

zagnieżdżone stosy 266

zaklinowane stosy 137

Zakładaj, że systemy są zawodne, zasada 14

zakres

komponentów testowanych

w etapach 115

zależności używanych podczas

etapów 115

zmiany, ograniczanie 347-352

zależności

izolowanie 107

jako zadanie skryptowe 328

kołowe 250

między stosami komponentów 281-293

minimalizowanie 107

zależności (*cd*)

minimalizowanie za pomocą kodu

definicji 291

oddzielanie 291

precyzowanie 107

serwery i 182

używanie warunków początkowych testu

do obsługi 132-135

zakres używany w etapach 115

Zależności komplikują testowanie

infrastruktury, wyzwanie 109

zapach 43

zasady *xviii*, 13-20

zasada minimalnej wiedzy. *Zobacz* prawo

Demeter

zasada najmniejszych uprawnień 28

zasobów, tagi

zasobów, tagi [161](#)

zasoby

obliczeniowe [26](#)

omówienie [25](#)

pamięci masowej [27](#)

sieciowe [28](#)

zatwierdzenie, jako etap potoku [114](#)

zautomatyzowane testowanie, w przepływie

pracy opartym na potoku [344](#)

„zbicie szybki”, proces [190](#)

zdecentralizowana konfiguracja [194](#)

zero zaufania, zabezpieczenia [28](#), [263](#)

zespół, przepływ pracy [331-346](#)

gałęzie kodu źródłowego [340](#)

ludzie [332-335](#)

mierzenie efektywności [332](#)

nadzór w przepływie pracy opartym na
potoku [344-346](#)

omówienie [331](#)

stosowanie kodu do infrastruktury [336](#)

zapobieganie dryfowi konfiguracji [341](#)

zgodność testowania [103](#)

złote obrazy [180](#)

zmiany, zarządzanie

klastry aplikacji i [234](#)

wzorce [186-191](#)

Zookeeper (witryna internetowa) [93](#)

0 autorze

Kief Morris jest globalnym dyrektorem ds. inżynierii chmury w ThoughtWorks. Prowadzi dyskusje na temat ról, regionów i branż w różnych firmach – od globalnych przedsiębiorstw po rozwijające się dopiero startupy. Lubi pracować i rozmawiać z ludźmi, poszukując lepszych praktyk inżynieryjnych, zasad projektowania architektury i praktyk dostarczania przeznaczonych do tworzenia systemów w chmurze.

Na początku lat 90 na Florydzie Kief uruchomił swój pierwszy system online, BBS (bulletin board system). Później zapisał się na studia magisterskie z informatyki na Uniwersytecie Tennessee, ponieważ wydawało mu się to najprostszym sposobem nawiązania prawdziwego kontaktu z Internetem. Przystąpienie do zespołu zajmującego się administracją systemu Wydziału Informatyki dało mu okazję do zarządzania setkami komputerów używających różnych odmian Uniksa.

Gdy bańka internetowa zaczęła się nadymać, Kief przeniósł się do Londynu, przyciągnięty przez wielokulturową mieszankę branż i ludzi. Nadal tam mieszka razem ze swoją żoną, synem i kotem.

Większość firm, dla których Kief pracował przed ThoughtWorks, były to poststartupy, zajmujące się tworzeniem i skalowaniem. Niektóre tytuły, które mu nadawano lub sam sobie wybierał, to programista, administrator systemów, zastępca dyrektora ds. technicznych, menedżer ds. badań i rozwoju, menedżer hostingu, kierownik techniczny, architekt techniczny, konsultant i dyrektor ds. inżynierii chmurowej.

Zwierzę na okładce *Infrastruktury jako kodu* to sęp Rüppella (*Gyps rueppellii*), pochodzący z Sahelu w Afryce (strefy geograficznej stanowiącej przejście między Saharą a sawanną). Gatunek został nazwany na cześć XIX-wiecznego niemieckiego odkrywcy i zoologa Eduarda Rüppella.

Jest to duży ptak (o rozpiętości skrzydeł 230–250 cm i wadze 6,8–9 kg) o cętkowanych brązowych piórach i żółto-białej szyi i głowie. Podobnie jak wszystkie sępy, jest mięsożerny i żywi się niemal wyłącznie padliną. Przedstawiciele tego gatunku używają swoich ostrych szponów i dziobów do wrywania kawałków mięsa ze zwłok i mają na języku skierowane do tyłu kolce, umożliwiające dokładne oskrobywanie kości. Są to ptaki bardzo społeczne, które zazwyczaj milczą, ale w miejscach gniazdowania kolonii lub podczas walki o pożywienie wydają głośne piskliwe dźwięki.

Sęp Rüppella jest monogamiczny i łączy się w pary na całe życie, które może trwać 40–50 lat. Pary lęgowe zakładają gniazda w pobliżu klifów (i często używają ich przez wiele lat). Budują je z patyków i wykładają trawą i liśćmi. Co roku składane jest tylko jedno jajo – piskląta stają się niezależne w momencie rozpoczęcia następnego sezonu lęgowego. Sępy należące do tego gatunku nie latają zbyt szybko (około 35 km/godz.), ale zapuszczają się do 150 km od gniazda w poszukiwaniu pożywienia.

Sępy Rüppella to najwyżej latające ptaki, jakie zaobserwowano; istnieje dowód na to, że potrafią wznieść się na 11 km nad poziom morza, czyli pułap samolotów pasażerskich. Mają specjalną hemoglobinę we krwi, która pozwala im efektywniej wchłaniać tlen na dużych wysokościach.

Liczebność populacji tego gatunku spada i jest on uważany za zagrożony. Chociaż jednym z czynników jest utrata siedlisk, najpoważniejszym zagrożeniem jest trucie. Sęp nie jest nawet jego głównym celem: farmerzy często umieszczają truciznę w zwłokach zwierząt gospodarskich, aby zemścić się na drapieżnikach, takich jak lwy i hieny. Ponieważ sępy szukają pożywienia kierując się wzrokiem i gdy je znajdą, gromadzą się wokół niego całymi stadami, setki tych ptaków potrafią zginąć za jednym razem.

Wiele zwierząt przedstawianych na okładkach książek O'Reilly jest zagrożonych wyginięciem; każde z nich jest ważne dla świata.

Kolorowa ilustracja na okładce autorstwa Karena Montgomery jest oparta na czarno-białej rycinie z Historii naturalnej Cassella. Czcionki użyte na okładce to Gilroy Semibold i Guardian Sans. Czcionka tekstu to Minion Pro firmy Adobe; czcionka nagłówków to Myriad Condensed firmy Adobe; a czcionka kodu to Ubuntu Mono studia Dalton Maag.

Polecamy także:

TypeScript. Skuteczne programowanie

62 sposoby ulepszania kodu TypeScript

ISBN: 978-83-7541-416-5

284 strony

TypeScript jest typowanym nadzbiorem języka JavaScript, stanowiącym potencjalne rozwiązanie wielu słynnych bolączek, z którymi borykają się programiści JavaScript. Niniejsza praktyczna książka, wykorzystująca metody spopularyzowane w książkach *Skuteczny nowoczesny C++* i *Java. Efektywne programowanie* zawiera omówienie 62 zagadnień wraz z konkretnymi zaleceniami, jak należy korzystać z języka i czego należy unikać. **Dan Vanderkam** jest głównym programistą w Sidewalk Labs. Od dawna uczestniczy w projektach typu Open Source. Wcześniej pracował na uniwersytecie Icahn School of Medicine at Mount Sinai, a także uczestniczył w rozwijaniu funkcji wyszukiwania używanych przez miliardy użytkowników Google.



Angular: Instalacja i działanie

Nauka krok po kroku

ISBN: 978-83-7541-361-8

318 stron

Ten praktyczny przewodnik szybko pozwoli na przyspieszenie działania struktury Angular w celu tworzenia wydajnych aplikacji na komputery osobiste i urządzenia mobilne. Wersja ta, wstępnie znana jako Angular 2, jest napisaną od nowa platformą autorstwa tego samego zespołu, który tworzył AngularJS. Programiści znający poprzednią wersję także ocenią tę książkę jako wartościowe źródło informacji.

Shyam Sehadi jest dyrektorem technicznym (CTO) w firmie ReStok Ordering Solutions. Wcześniej był inżynierem programowania w Amazon i Google i kierował zespołem inżynierskim w firmie Hopscotch z siedzibą w Bombaju. Shyam jest autorem dwóch wcześniejszych książek na temat Angular.

Więcej informacji i ofert: www.ksiazki.promise.pl

Polecamy także:



Python w uczeniu maszynowym

Podejście sterowane testami

ISBN: 978-83-7541-357-1

242 strony

Ten praktyczny przewodnik pozwoli osiągnąć biegłość w stosowaniu uczenia maszynowego w codziennej pracy. Autor bez akademickich rozważań pokazuje, jak integrować i testować algorytmy uczenia maszynowego w swoim kodzie. Przedstawia wykorzystanie testów z użyciem bibliotek NumPy, Pandas, Scikit-Learn oraz SciPy, ilustrując je licznymi wykresami oraz przykładami kodu.

Matthew Kirk jest konsultantem, autorem i międzynarodowym prelegentem, specjalizującym się w uczeniu maszynowym i analizie danych z wykorzystaniem języków Ruby i Python. Lubi pomagać innym programistom w integrowaniu analizy danych ze stosowanymi przez nich technologiami.

Progresywne aplikacje webowe

Potęga aplikacji natywnych w przeglądarce

ISBN: 978-83-7541-348-9

298 stron

Aplikacje natywne, ustąpcie miejsca. Nowe, progresywne aplikacje webowe mają możliwości, które wkrótce sprawią, że będziecie przestarzałe. Dzięki temu nauczysz się, jak tworzyć aplikacje webowe wykorzystujące funkcje do tej pory dostępne wyłącznie w aplikacjach natywnych, takie jak szybki czas ładowania, powiadomienia z serwera, dostęp offline, skróty na ekranie startowym i środowisko przypominające działanie aplikacji.

Tal Ater jest programistą, konsultantem i przedsiębiorcą z ponad 20 letnim stażem. Miliony osób codziennie używają jego rozwiązań, w tym popularnych bibliotek service worker i rozpoznawania mowy.



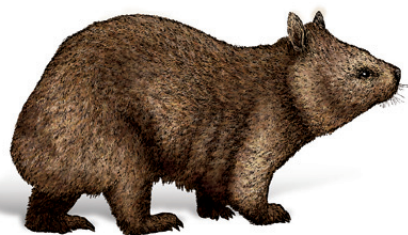
Więcej informacji i ofert: www.ksiazki.promise.pl

Polecamy także:

O'REILLY®

Praktyczne uczenie nienadzorowane przy użyciu języka Python

Jak budować użytkowe rozwiązania uczenia
maszynowego na podstawie
nieoznakowanych danych.



Ankur A. Patel

Praktyczne uczenie nienadzorowane przy użyciu języka Python

ISBN: 978-83-7541-426-4

362 str

Wielu ekspertów branżowych uważa uczenie nienadzorowane za kolejną granicę w dziedzinie sztucznej inteligencji. Ponieważ większość danych na świecie jest nieoznakowana, nie można zastosować konwencjonalnego uczenia nadzorowanego. Uczenie nienadzorowane może pozwolić odnaleźć istotne wzorce ukryte głęboko w tych danych, które inaczej są niemal niemożliwe do odkrycia.

Ankur A. Patel jest wiceprezesem ds. informatyki analitycznej w firmie 7Park Data. Ankur i jego zespół analizy danych wykorzystują dane alternatywne do opracowywania produktów związanych z danymi dla funduszy hedgingowych i korporacji oraz rozwijają usługi uczenia maszynowego dla klientów firmowych.

Wydajne programowanie w R

Praktyczny przewodnik po lepszym programowaniu

ISBN: 978-83-7541-352-6

242 strony

Istnieje wiele znakomitych materiałów dotyczących wizualizacji, analizy danych i tworzenia pakietów w języku R. Setki rozproszonych winiet, stron internetowych i forów wyjaśnia, jak używać R. Niewiele jednak napisano o tym, jak zapewnić *efektywne działanie* języka R – aż do teraz. Ten podręcznik uczy, jak tworzyć wydajny kod w tym języku.

Colin Gillespie jest wykładowcą na Uniwersytecie w Newcastle w Wielkiej Brytanii, języka R uczy od 2005 roku. Jego zainteresowania badawcze obejmują obliczenia o wysokiej wydajności oraz statystykę bayesowską.

Robin Lovelace z uniwersytetu w Leeds w Wielkiej Brytanii od lat wykorzystuje R i uczy tego języka na wszystkich poziomach.

O'REILLY®



Wydajne programowanie w R

PRAKTYCZNY PRZEWODNIK PO LEPSZYM PROGRAMOWANIU

Colin Gillespie i Robin Lovelace

Więcej informacji i ofert: www.ksiazki.promise.pl

Polecamy także:

O'REILLY®

Wzorce projektowe uczenia maszynowego

Rozwiązania typowych problemów dotyczących przygotowania danych, konstruowania modeli i MLOps



Valliappa Lakshmanan,
Sara Robinson, Michael Munn

Wzorce projektowe uczenia maszynowego

Rozwiązania typowych problemów dotyczących przygotowania danych, konstruowania modeli i MLOps

ISBN: 978-83-7541-441-7

Opisane w tej książce wzorce obejmują najlepsze praktyki i rozwiązania powtarzalnych problemów w uczeniu maszynowym. Autorzy, inżynierowie Google, skatalogowali sprawdzone metody, aby pomóc sprostać typowym problemom występującym w całym procesie uczenia maszynowego. **Valliappa (Lak) Lakshmanan** jest globalnym kierownikiem działu analizy danych i rozwiązań sztucznej inteligencji w Google Cloud. **Sara Robinson** jest rzeczniczką deweloperów w zespole Google Cloud, skupiającą się na uczeniu maszynowym. **Michael Munn** jest inżynierem rozwiązań uczenia maszynowego w Google, gdzie pomaga klientom projektować, implementować i wdrażać modele uczenia maszynowego.

Mikrouslugi Budowa i działanie

Przewodnik po budowaniu architektury mikrouslug

ISBN: 978-83-7541-433-1

Architektury mikrouslug oferują większą prędkość wprowadzania zmian, lepszą skalowalność oraz czystsze, łatwiejsze do rozwijania projekty systemów. Jednak implementowanie architektury mikrouslug nie jest łatwe. Jak dokonywać niezliczonych wyborów, przeszkolić zespół pod kątem tych wszystkich szczegółów technicznych i poprowadzić organizację w stronę udanego wdrożenia, aby zmaksymalizować szanse powodzenia?

Ronnie Mitra jest autorem, strategiem i konsultantem z przeszło 25-letnim doświadczeniem w pracy z technologiami łączności i sieci WWW. **Irakli Nadareishvili** jest wiceprezesem do spraw innowacji w firmie Capital One, kierując zespołami odpowiedzialnymi za budowanie platformy bankowości opartymi na mikrouslugach.

O'REILLY®

Mikrouslugi Budowa i działanie

Przewodnik po budowaniu architektury mikrouslug



Ronnie Mitra,
Irakli Nadareishvili

Więcej informacji i ofert: www.ksiazki.promise.pl