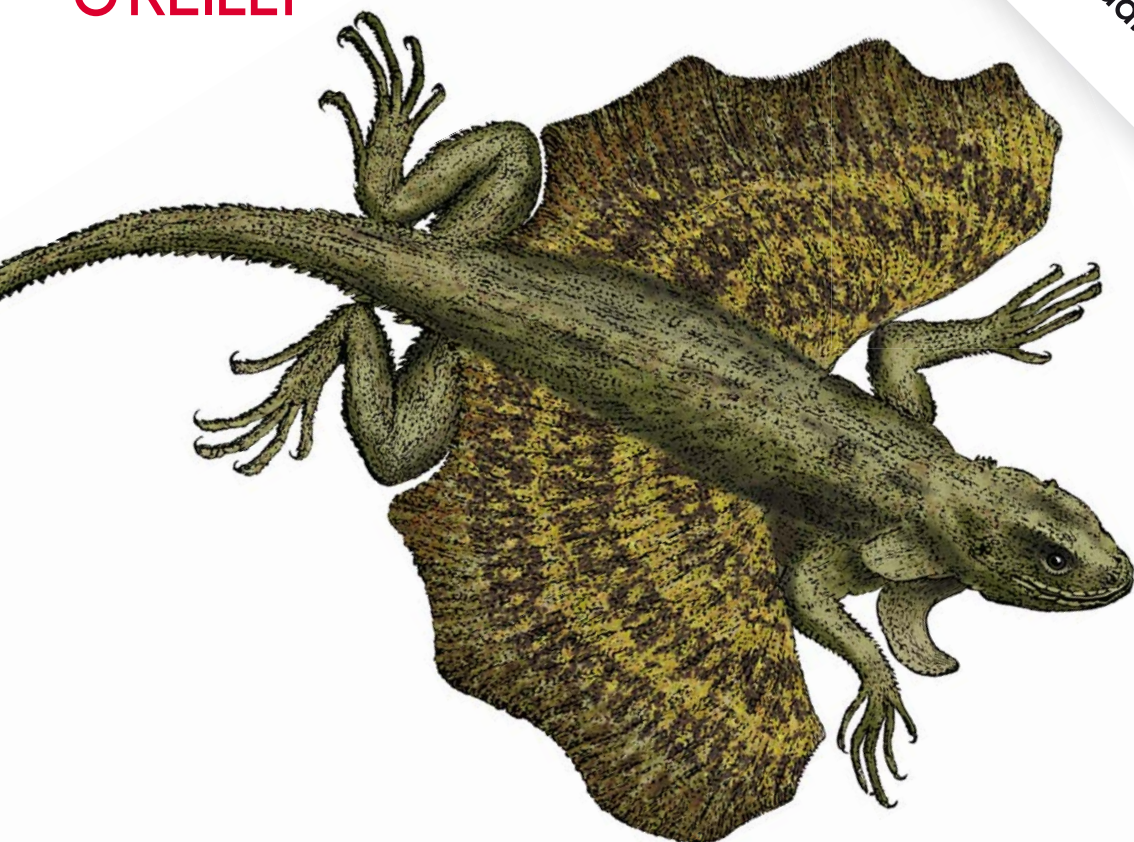


O'REILLY®

Wydanie II



Terraform

Krótkie wprowadzenie

Tworzenie infrastruktury za pomocą kodu

Helion 

Yevgeniy Brikman

Tytuł oryginału: Terraform: Up & Running: Writing Infrastructure as Code, 2nd Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-6650-3

© 2020 Helion SA

Authorized Polish translation of the English edition of *Terraform: Up & Running, 2nd Edition*

ISBN 9781492043225 © 2019 Yevgeniy Brikman

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

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

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

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

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

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/terra2.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie/terra2_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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

Dla Mamy, Taty, Lyalyi i Molly

Wprowadzenie	9
1. Dlaczego Terraform?	21
Powstanie ruchu DevOps	21
Infrastruktura jako kod	23
Skrypty tymczasowe	24
Narzędzia zarządzania konfiguracją	25
Narzędzia szablonów serwera	27
Narzędzia instrumentacji	31
Narzędzia provisioningu	33
Korzyści płynące z infrastruktury jako kodu	35
Jak działa Terraform?	37
Porównanie Terraform z innymi narzędziami IaC	39
Zarządzanie konfiguracją kontra provisioning	39
Infrastruktura niemodyfikowalna kontra modyfikowalna	40
Język proceduralny kontra deklaratywny	41
Serwer główny kontra jego brak	44
Agent kontra jego brak	45
Duża społeczność kontra mała	46
Rozwiązanie dojrzałe kontra najnowsze	50
Używanie razem wielu narzędzi	50
Podsumowanie	53
2. Rozpoczęcie pracy z Terraform	55
Utworzenie konta AWS	56
Instalacja Terraform	59
Wdrożenie pojedynczego serwera	60
Wdrożenie pojedynczego serwera WWW	67
Wdrażanie konfigurowalnego serwera WWW	74
Wdrażanie klastra serwerów WWW	79

Wdrożenie mechanizmu równoważenia obciążenia	82
Porządkowanie	90
Podsumowanie	91
3. Zarządzanie informacjami o stanie Terraform	93
Czym są informacje o stanie Terraform?	93
Współdzielony magazyn danych dla plików informacji o stanie	95
Ograniczenia backendu Terraform	102
Izolowanie plików informacji o stanie	104
Izolacja za pomocą przestrzeni roboczych	106
Izolacja za pomocą układu plików	110
Źródło danych terraform_remote_state	115
Podsumowanie	124
4. Zastosowanie modułów do tworzenia infrastruktury Terraform wielokrotnego użycia	125
Podstawy modułów	128
Dane wejściowe modułu	130
Wartości lokalne modułu	134
Dane wyjściowe modułu	136
Problemy z modułami	138
Ścieżki dostępu do pliku	138
Osadzony blok kodu	139
Wersjonowanie modułu	141
Podsumowanie	146
5. Sztuczki i podpowiedzi dotyczące Terraform — pętle, konstrukcje if, wdrażanie i problemy	149
Pętle	150
Pętla za pomocą parametru count	150
Pętla za pomocą wyrażenia for_each	156
Pętla za pomocą wyrażenia for	161
Pętla za pomocą dyrektywy for ciągu tekstowego	164
Wyrażenie warunkowe	165
Wyrażenie warunkowe z użyciem parametru count	166
Definiowanie warunku za pomocą for_each i wyrażen	175
Wyrażenia warunkowe wraz z dyrektywą if ciągu tekstowego	176
Wdrożenie bez przestoju	177
Problemy związane z Terraform	188
Ograniczenia parametru count i wyrażenia for_each	188
Ograniczenia wdrożenia bez przestoju	190

Awaryjne poprawki planów	191
Trudności podczas refaktoryzacji	192
Osiągnięcie ostatecznej spójności może wymagać nieco czasu	195
Podsumowanie	196
6. Produkcyjny kod Terraform	197
Dlaczego przygotowanie infrastruktury o jakości produkcyjnej trwa tak długo?	199
Lista rzeczy do zrobienia podczas tworzenia infrastruktury o jakości produkcyjnej	201
Moduły infrastruktury o jakości produkcyjnej	203
Małe moduły	203
Moduły łączone z innymi	208
Moduły możliwe do testowania	216
Moduły możliwe do wydania	219
Moduły wykraczające poza Terraform	223
Podsumowanie	229
7. Testowanie kodu Terraform	231
Testy ręczne	232
Podstawy ręcznego przeprowadzania testów	233
Uporządkowanie środowiska po zakończeniu testów	237
Testy zautomatyzowane	238
Testy jednostkowe	239
Testy integracji	265
Testy typu E2E	279
Inne podejścia w zakresie testów	284
Podsumowanie	286
8. Używanie Terraform w zespołach	289
Adaptacja infrastruktury jako kodu przez zespół	289
Przekonanie szefa do pomysłu	290
Stopniowe wprowadzanie zmian	292
Zapewnienie zespołowi czasu na naukę	294
Sposób pracy podczas wdrażania kodu aplikacji	295
Użycie systemu kontroli wersji	296
Lokalne uruchomienie kodu	296
Wprowadzenie zmian w kodzie	297
Przekazanie zmian do zatwierdzenia	298
Uruchomienie testów zautomatyzowanych	299
Połączenie kodu istniejącego z nowym i wydanie produktu	299
Wdrożenie	300

Sposób pracy podczas wdrażania kodu infrastruktury	305
Użycie systemu kontroli wersji	305
Lokalne uruchomienie kodu	309
Wprowadzenie zmian w kodzie	310
Przekazanie zmian do zatwierdzenia	311
Uruchomienie testów zautomatyzowanych	314
Połączenie kodu istniejącego z nowym i wydanie produktu	315
Wdrożenie	315
Zebranie wszystkiego w całość	324
Podsumowanie	326
A Polecane zasoby	329

Wprowadzenie

Dawno temu w odległym centrum danych grupa wiekowych istot ludzkich o potężnych możliwościach, określana mianem administratorów systemu, zajmowała się ręcznym przygotowywaniem infrastruktury. Każdy serwer, każda baza danych, mechanizm równoważenia obciążenia, a nawet najmniejszy element konfiguracji sieci były tworzone i zarządzane ręcznie. To były mroczne czasy, pełne najróżniejszych obaw: dotyczących przestoju, przypadkowego użycia błędnej konfiguracji, przeprowadzenia wolnego i zawodnego wdrożenia oraz tego, co się stanie, gdy administrator systemu przejdzie na ciemną stronę (np. pojedzie na wakacje). Dobrą wiadomością jest to, że dzięki ruchowi DevOps pojawił się znacznie lepszy sposób na wykonywanie wymienionych wcześniej zadań: *Terraform*.

Terraform (<https://www.terraform.io/>) to stworzone przez HashiCorp narzędzie typu open source, pozwalające na zdefiniowanie infrastruktury jako kodu przy użyciu prostego, deklaratywnego języka. Tę infrastrukturę można wdrażać i później zarządzać nią za pomocą wielu dostępnych publicznie dostawców chmury (np. Amazon Web Services, Microsoft Azure, Google Cloud Platform, Digital Ocean) oraz prywatnych platform chmury i wirtualizacji (np. OpenStack i VMWare), używając do tego zaledwie kilku poleceń. Przykładowo, zamiast ręcznie klikać na stronie internetowej lub wydawać dziesiątki poleceń, w przedstawionym tutaj przykładzie znajdziesz cały kod wymagany do skonfigurowania serwera w usłudze AWS:

```
provider "aws" {  
  region = "us-east-2"  
}  
  
resource "aws_instance" "example" {  
  ami           = "ami-0c55b159cbfafe1f0"  
  instance_type = "t2.micro"  
}
```

Aby wdrożyć ten kod, należy wydać następujące polecenia:

```
$ terraform init  
$ terraform apply
```

Dzięki prostocie i potężnym możliwościom Terraform stał się kluczowym graczem w świecie DevOps. Pozwala, aby żmudne, zawodne i ręczne zarządzanie infrastrukturą zostało zastąpione przez rozwiązanie zautomatyzowane, na bazie którego można opracować wszystkie praktyki DevOps (np. automatyczne testowanie, ciągłą integrację, ciągłe dostarczanie oprogramowania) i narzędzia (np. Docker, Chef, Puppet).

Dzięki tej książce będziesz mógł bardzo szybko rozpocząć pracę z Terraform i wykorzystać ten framework we własnych projektach.

Rozpoczniesz od wdrożenia za pomocą Terraform najprostszego przykładu w postaci aplikacji typu Witaj, świecie (właściwie już zobaczyłeś ten przykład), a skończysz na przygotowaniu pełnego stosu (klastr serwerów, mechanizm równoważenia obciążenia, baza danych), który potrafi obsługiwać ogromne ilości ruchu sieciowego i duże zespoły programistów — wszystkie informacje niezbędne do opracowania takiego rozwiązania zdobędziesz w trakcie lektury zaledwie kilku rozdziałów. Materiał przedstawiony w książce nie tylko dotyczy reguł związanych z DevOps i infrastrukturą jako kodem (ang. *infrastructure as code*, IaC), ale również zawiera dziesiątki przykładów gotowych do natychmiastowego wypróbowania — upewnij się więc, że masz pod ręką komputer.

Zanim skończysz lekturę książki, będziesz gotowy do wykorzystania Terraform w rzeczywistych projektach.

Dla kogo jest przeznaczona ta książka?

Ta książka jest przeznaczona dla każdego, kto odpowiada za kod po jego utworzeniu: administratorów systemów, inżynierów operacji, inżynierów wydania, inżynierów niezawodności działania witryny internetowej, inżynierów DevOps, programistów infrastruktury, programistów pełnego stosu, menedżerów inżynierów i CTO. Niezależnie od nazwy stanowiska — np. zarządzasz infrastrukturą, wdrażaniem kodu, konfigurowaniem serwerów, skalowaniem klastrów, tworzeniem kopii zapasowej danych, monitorowaniem aplikacji i reagowaniem o trzeciej nad ranem na zdarzenia — ta książka jest właśnie dla Ciebie.

Wszystkie zadania wymienione wcześniej są razem określane mianem operacji. W przeszłości bardzo często zdarzało się, że programiści pracujący nad projektem potrafili tworzyć kod, ale nie rozumieli wspomnianych operacji. Równie powszechną sytuacją było, że administratorzy systemów rozumieli wagę operacji, ale nie wiedzieli, jak tworzyć kod. Takie rozróżnienie mogło istnieć w przeszłości, natomiast w nowoczesnym świecie, w którym powszechne stało się przetwarzanie w chmurze i stosowanie podejścia DevOps, każdy programista musi mieć umiejętności w zakresie operacji, każdy administrator systemu zaś powinien potrafić tworzyć kod.

W książce nie przyjąłem założenia, że jesteś ekspertem w tworzeniu kodu lub administrowaniu systemem — wystarczające są podstawowe umiejętności w zakresie programowania, pracy w powłocie i wykorzystania oprogramowania serwerowego (np. związanego z obsługą witryn internetowych). Wszystkie inne umiejętności, które będą potrzebne podczas lektury, możesz zdobywać po drodze. Zanim dotrzesz do końca książki, będziesz dysponować solidną wiedzą dotyczącą jednego z aspektów o znaczeniu krytycznym podczas nowoczesnego programowania i przeprowadzania operacji: zarządzania infrastrukturą jako kodem.

Dowiesz się nie tylko, jak za pomocą Terraform zarządzać infrastrukturą jako kodem, ale także jak te zadania mieszczą się w ogólnym świecie DevOps. Oto wybrane pytania, na które będziesz mógł odpowiedzieć po lekturze książki:

- Dlaczego w ogóle miałbym stosować IaC?
- Jakie są różnice między zarządzaniem konfiguracją, zgraniem całości, provisioningiem i stosowaniem szablonów w serwerze?

- Dlaczego powinienem używać narzędzi takich jak Terraform, Chef, Ansible, Puppet, Salt, Cloud-Formation, Docker, Packer lub Kubernetes?
- Na czym polega działanie narzędzia Terraform i jak je można wykorzystać do zarządzania infrastrukturą?
- Jak tworzyć moduły Terraform wielokrotnego użycia?
- Jak utworzyć kod Terraform, który będzie wystarczająco niezawodny do zastosowania w produkcji?
- Jak przetestować kod Terraform?
- Jak można dodać Terraform do automatycznego procesu wdrażania?
- Jakie są najlepsze praktyki podczas używania Terraform w zespole programistów?

Jedynym potrzebnym narzędziem jest komputer (Terraform działa w większości systemów operacyjnych) połączony z internetem, nie bez znaczenia jest też chęć do nauki.

Dlaczego napisałem tę książkę?

Terraform to narzędzie o potężnych możliwościach i działa ze wszystkimi rozwiązaniami od popularnych dostawców chmury. Wykorzystuje prosty i przejrzysty język, zapewnia solidne możliwości w zakresie wielokrotnego użycia kodu, testowania i wersjonowania. To oprogramowanie typu open source, dla którego istnieje przyjazna i aktywna społeczność. Jednak mimo tych zalet istnieje jeden obszar, który można uznać za słabą stronę Terraform: brak dojrzałości.

Terraform to stosunkowo nowa technologia. W chwili gdy piszę te słowa (maj 2019), nie osiągnęła jeszcze wersji 1.0.0, a pomimo coraz większej popularności wciąż trudno jest znaleźć książki, blogi lub ekspertów pomagających w opanowaniu arkanów tego narzędzia. Oficjalna dokumentacja Terraform sprawdza się doskonale w zakresie dostarczenia informacji o podstawowej składni i funkcjach, ale jednocześnie oferuje niewiele informacji o wzorcach, najlepszych praktykach, testowaniu, wielokrotnym użyciu kodu lub wykorzystaniu Terraform w zespołach programistów. Tę sytuację można porównać do następującej: próbujesz płynnie opanować np. język francuski, ucząc się jedynie słownictwa, a zupełnie pomijając gramatykę i idiomy.

Tę książkę napisałem, aby pomóc programistom w nabyciu biegłości w pracy z Terraform. Korzystam z Terraform przez cztery z pięciu lat jego istnienia, przede wszystkim w mojej firmie, Gruntwork (<https://www.gruntwork.io/>). W firmie Grunt Terraform to jedno z podstawowych narzędzi użytych do przygotowania biblioteki liczącej ponad 300 000 wierszy przetestowanego w boju i możliwego do wielokrotnego użycia kodu infrastruktury, który jest wykorzystywany przez setki firm. Utworzenie i zarządzanie na przestrzeni wielu lat tak wielką ilością kodu infrastruktury oraz wykorzystanie go przez dużą liczbę firm pozwoliło na zebranie wielu trudnych doświadczeń. W tym czasie otrzymałem też niejedną bolesną nauczkę. Moim celem jest podzielenie się zdobytym doświadczeniem, aby długi proces poznawania Terraform można było zredukować do znacznie krótszego okresu liczonego w dniach.

Oczywiście nie nabierzesz biegłości w pracy z Terraform, jeśli ograniczysz się jedynie do przeczytania książki. Jeżeli chcesz biegle posługiwać się np. językiem francuskim, musisz rozmawiać z Francuzami,

oglądać francuską telewizję i słuchać francuskiej muzyki. Analogicznie, aby biegle posługiwać się Terraform, trzeba tworzyć rzeczywisty kod Terraform, zastosować go do zarządzania rzeczywistym oprogramowaniem oraz wdrażać oprogramowanie w rzeczywistych serwerach. Dlatego też przygotuj się na czytanie, tworzenie i uruchamianie ogromnej ilości kodu.

Co znajduje się w książce?

Oto krótkie omówienie materiału zamieszczonego w poszczególnych rozdziałach.

Rozdział 1., „Dlaczego Terraform?”

Jak podejście DevOps zmienia sposób uruchamiania oprogramowania; ogólne omówienie narzędzi infrastruktury jako kodu wraz z uwzględnieniem zarządzania konfiguracją, szablonami w serwerze, zgrywaniem całości i stosowaniem narzędzi provisioningu; zalety infrastruktury jako kodu; porównanie narzędzi: Terraform, Chef, Puppet, Ansible, SaltStack, OpenStack Heat i CloudFormation; sposoby na połączenie narzędzi takich jak Terraform, Packer, Docker, Ansible i Kubernetes.

Rozdział 2., „Rozpoczęcie pracy z Terraform”

Instalowanie Terraform; ogólne omówienie składni i narzędzi powłoki Terraform; wdrażanie pojedynczego serwera; wdrażanie serwera WWW; wdrażanie klastra serwerów WWW; wdrażanie mechanizmu równoważenia obciążenia; uporządkowanie utworzonych zasobów.

Rozdział 3., „Zarządzanie informacjami o stanie Terraform”

Czym są informacje o stanie Terraform; jak przechowywać informacje o stanie, aby były dostępne dla wielu członków zespołu; jak blokować pliki informacji o stanie, aby uniknąć stanu wyścigu; jak zarządzać danymi poufnymi za pomocą Terraform; jak odizolować pliki informacji o stanie, aby ograniczyć szkody powstałe na skutek błędów; jak wykorzystać przestrzeń roboczą Terraform; jakie są najlepsze praktyki w zakresie tworzenia układu plików i katalogów dla projektu Terraform; jak używać informacji o stanie tylko do odczytu.

Rozdział 4., „Zastosowanie modułów do tworzenia infrastruktury Terraform wielokrotnego użycia”

Czym jest moduł Terraform; w jaki sposób można utworzyć prosty moduł; jak zapewnić możliwość konfigurowania modułu za pomocą danych wejściowych i wyjściowych; czym są wartości lokalne; na czym polega wersjonowanie modułów; jakie problemy napotyka się podczas pracy z modułami; jak można wykorzystać moduły w celu definiowania wielokrotnego użycia i konfigurowalnych fragmentów infrastruktury.

Rozdział 5., „Sztuczki i podpowiedzi dotyczące Terraform — pętle, konstrukcje if, wdrażanie i problemy”

Stosowanie pętli opartych na parametrze count, wyrażeniach for_each i for oraz dyrektywie ciągu tekstowego for; konstrukcje warunkowe oparte na parametrze count, wyrażeniach for_each i for oraz dyrektywie ciągu tekstowego if; funkcje wbudowane; wdrażanie bez przestoju; najczęstsze problemy i pułapki podczas pracy z Terraform: ograniczenia parametru count i wyrażenia for_each, wpadki związane z wdrożeniem bez przestoju, awarie, wydawałoby się, prawidłowych planów, problemy z refaktoryzacją i brak zachowania spójności.

Rozdział 6., „Produkcyjny kod Terraform”

Dlaczego projekt wykorzystujący podejście DevOps zawsze zabiera więcej czasu, niż przewidywano; lista rzeczy do sprawdzenia dla kodu produkcyjnego; przygotowywanie modułów Terraform dla środowiska produkcyjnego; małe moduły; moduły złożone; moduły możliwe do testowania; moduły możliwe do wydania; rejestr Terraform; luki w zabezpieczeniach Terraform.

Rozdział 7., „Testowanie kodu Terraform”

Ręczne testowanie kodu Terraform; środowisko produkcyjne i porządkowanie go; zautomatyzowane testy kodu Terraform; testy jednostkowe kodu Terraform; testy typu E2E; mechanizm wstrzykiwania zależności; jednoczesne przeprowadzanie testów; etapy przeprowadzania testów; ponowne wykonywanie testów; piramida testów; analiza statyczna; sprawdzanie właściwości.

Rozdział 8., „Używanie Terraform w zespołach”

Zaadaptowanie Terraform przez zespół; jak przekonać szefa do stosowania Terraform; sposoby na wdrażanie kodu aplikacji; sposoby na wdrażanie kodu infrastruktury; system kontroli wersji; złota reguła Terraform; przegląd kodu; reguły dotyczące tworzenia kodu; styl Terraform; stosowanie ciągłej integracji i ciągłego wdrażania w Terraform; proces wdrażania.

Książkę możesz przeczytać od początku do końca lub też przechodzić między rozdziałami, które najbardziej Cię interesują. Warto w tym miejscu dodać, że przykłady w poszczególnych rozdziałach odwołują się do przykładów w poprzednich częściach książki i zostały zbudowane na ich podstawie. Dlatego też, jeśli pominiesz rozdział, skorzystaj z archiwum materiałów przygotowanych dla tej publikacji (więcej informacji na ten temat znajdziesz nieco dalej). Na końcu książki znajduje się dodatek A, w którym zamieściłem listę proponowanych zasobów dostarczających więcej informacji na temat Terraform, operacji infrastruktury jako kodu i podejścia DevOps.

Co nowego w drugim wydaniu?

Pierwsze wydanie książki pojawiło się w 2017 roku, teraz mamy maj 2019 i przygotowuję wydanie drugie. Na przestrzeni tego czasu zmieniło się naprawdę wiele. Drugie wydanie ma wielkość niemal dwukrotnie większą od pierwszego (około 160 stron więcej), zawiera dwa całkowicie nowe rozdziały oraz poważne zmiany we wszystkich pozostałych rozdziałach i fragmentach kodu.

Jeżeli czytałeś pierwsze wydanie książki i chciałbyś się dowiedzieć, co nowego jest w drugim wydaniu, lub po prostu jesteś ciekaw, jak wyglądała ewolucja Terraform, zapoznaj się z przedstawionymi tutaj punktami:

Cztery wydania główne Terraform

Gdy pierwsze wydanie książki trafiło do rąk czytelników, narzędzie Terraform było dostępne w wersji 0.8. Między pierwszą i drugą edycją pojawiły się cztery wydania główne Terraform i obecnie mamy wersję 0.12. W tym wydaniu znalazła się niesamowita nowa funkcjonalność, o której wspomnę w książce, choć dla użytkowników to oznacza znacznie więcej pracy podczas uaktualniania oprogramowania¹.

¹ Zapoznaj się z przewodnikami uaktualniania Terraform, które znajdziesz na stronie <https://www.terraform.io/upgrade-guides/index.html>.

Usprawnienia w zakresie testów zautomatyzowanych

Narzędzia i praktyki w zakresie tworzenia testów zautomatyzowanych dla kodu Terraform uległy dużym zmianom. Całkowicie nowy rozdział 7. jest poświęcony testowaniu i przedstawia zagadnienia takie jak testy jednostkowe, testy integracji, testy typu E2E, mechanizm wstrzykiwania zależności, równoległe wykonywanie testów, analiza statyczna itd.

Usprawnienia dotyczące modułów

Narzędzia i praktyki w zakresie tworzenia modułów dla kodu Terraform uległy dużym zmianom. W całkowicie nowym rozdziale 6. znajdziesz informacje o tworzeniu przetestowanych w boju modułów produkcyjnych Terraform wielokrotnego użycia — to są moduły, na których prawidłowym działaniu Twoja firma może polegać.

Usprawnienia w sposobie działania

Rozdział 8. został napisany całkowicie od początku, aby odzwierciedlić zmiany w sposobach, na jakie zespoły włączają Terraform do swojego działania. Znajdziesz tutaj dokładne omówienie zajmowania się kodem aplikacji i kodem infrastruktury — od jego tworzenia, poprzez testowanie, aż po wdrożenie w produkcji.

HCL2

W wydaniu Terraform 0.12 nastąpiło przejście z języka HCL na HCL2. Nowa wersja zawiera m.in. doskonałą obsługę wyrażeń (nie trzeba więc opakowywać wszystkiego w `${...}!`), rozbudowany system ograniczeń, wyrażenia warunkowe obliczane z opóźnieniem, obsługę wyrażeń `null`, `for_each` i `for`, dynamiczne bloki kodu wewnętrznego itd. Wszystkie przykłady przedstawione w książce zostały uaktualnione do HCL2, a dokładne omówienie funkcjonalności oferowanej przez nowy język znajdziesz w rozdziałach 5. i 6.

Reorganizacja obsługi informacji o stanie w Terraform

Wydanie 0.9 zawierało backend przeznaczony do przechowywania i współdzielenia informacji o stanie Terraform łącznie z obsługą nakładania blokad. W wydaniu 0.9 pojawiły się również środowiska stanów jak mechanizm pozwalający na zarządzanie wdrożeniami w wielu środowiskach. Natomiast w wydaniu 0.10 te środowiska stanu zostały zastąpione przez przestrzenie robocze Terraform. Wszystkie te tematy zostaną omówione w rozdziale 3.

Podział dostawców Terraform

W wydaniu 0.10 podstawowy kod Terraform został oddzielony od kodu obsługującego poszczególnych wszystkich dostawców (kod dla AWS, kod dla GCP, Azure itd.). To pozwoliło na opracowywanie dostawców w oddzielnych repozytoriach, w różnym tempie oraz z własnym wersjonowaniem. To jednak oznacza konieczność wydania polecenia `terraform init` w celu pobrania kodu dostawcy — to polecenie trzeba wydać za każdym razem, gdy zaczyna się pracę z nowym modułem. Więcej informacji na ten temat przedstawię w rozdziałach 2. i 7.

Ogromny wzrost liczby dostawców

Od 2016 roku zwiększyła się lista dostawców chmur obsługiwanych przez Terraform, od (jedynie) AWS, GCP i Azure do ponad 100 oficjalnych dostawców i wielu opracowanych przez spo-

łeczność². To oznacza możliwość wykorzystania Terraform do zarządzania nie tylko wieloma innymi typami chmury (teraz mamy dostawców dla Alicloud, Oracle Cloud Infrastructure, VMware vSphere i innych), ale również wieloma innymi aspektami związanymi z kodem, czyli m.in. zarządzania wersją (dostawcy dla GitHub, GitLab i BitBucket), magazynami danych (dostawcy dla MySQL, PostgreSQL i InfluxDB), systemami monitorowania i ostrzegania (dostawcy dla DataDog, New Relic i Grafana), narzędziami obsługi platform (dostawcy dla Kubernetes, Helm, Heroku, Rundeck i Rightscale) itd. Co więcej, każdy z dostawców jest obecnie lepiej obsługiwany: dostawca AWS zapewnia dostęp do większości ważnych usług AWS, a nowe usługi są dodawane bardzo często, nawet jeszcze przed ich pojawieniem się w CloudFormation.

Rejestr Terraform

Firma HashiCorp udostępniła w 2017 roku repozytorium modułów Terraform (nazwane *Terraform Registry* i dostępne pod adresem <https://registry.terraform.io/>). To jest interfejs użytkownika ułatwiający przeglądanie i wykorzystywanie modułów Terraform wielokrotnego użycia, udostępnionych jako oprogramowanie typu open source opracowywane przez społeczność. W 2018 roku firma HashiCorp udostępniła możliwość uruchamiania prywatnej wersji Terraform Registry we własnej organizacji. Z kolei wydanie 0.11 zaoferowało doskonałą obsługę składni przeznaczonej do pracy z modułami pochodzącymi z Terraform Registry. Więcej informacji na temat wielokrotnego użycia modułów znajdziesz w rozdziale 6.

Lepsza obsługa błędów

Wydanie 0.9 wprowadziło uaktualnioną obsługę błędów stanu. Jeżeli podczas zapisu w zdalnym backendzie informacji o stanie pojawił się błąd, to dane były zapisywane lokalnie w pliku o nazwie *errored.tfstate*. Wydanie 0.12 zawiera całkowicie przeprojektowaną obsługę błędów, która teraz przechwytyuje błędy na znacznie wcześniejszym etapie, wyświetla zdecydowanie czytelniejsze komunikaty błędów oraz zawiera ścieżkę dostępu do pliku, numer wiersza i fragment kodu zawierający błąd.

Wiele innych, drobniejszych zmian

Między wersjami od 0.8 do 0.12 wprowadzono jeszcze wiele innych, drobniejszych zmian, m.in. pojawienie się wartości lokalnych (rozdział 4.), nowe sposoby na współpracę Terraform ze światem zewnętrznym za pomocą skryptów (rozdział 6.), wykonywanie polecenia `terraform plan` jako części polecenia `terraform apply` (rozdział 2.), poprawki dotyczące problemów związanych z wywołaniem `create_before_destroy`, znaczne usprawnienia w zakresie parametru `count`, który teraz może zawierać odwołania do źródeł danych i zasobów (rozdział 3.), dziesiątki nowych funkcji wbudowanych, dziedziczenie bloku `provider` i wiele innych.

Czego nie znajdziesz w książce?

Ta książka nie jest rozbudowanym podręcznikiem użytkownika Terraform. Nie omówiłem w niej wszystkich dostawców chmury, a także wszystkich zasobów obsługiwanych przez dostawców chmury i każdego dostępnego polecenia Terraform. Związane z tym szczegóły bez problemu znajdziesz w dokumentacji Terraform zamieszczonej na stronie <https://www.terraform.io/docs/>.

² Lista dostawców Terraform znajduje się na stronie <https://www.terraform.io/docs/providers/>.

Wprawdzie wspomniana dokumentacja zawiera również wiele użytecznych odpowiedzi, ale jeśli dopiero zaczynasz pracę z Terraform, infrastrukturą jako kodem oraz operacjami, możesz nawet nie wiedzieć, jakie pytania zadawać. Dlatego też w tej książce skoncentrowałem się na zagadnieniach *nieporuszonych* w dokumentacji, czyli przede wszystkim na tym, jak wyjść poza podstawowe przykłady i wykorzystać Terraform w rzeczywistych projektach. Moim celem jest umożliwienie Ci jak najszybszego rozpoczęcia pracy, więc przedstawiłem to, co przede wszystkim powinieneś wiedzieć o Terraform, a także informacje o tym, jak zastosować go w swoim sposobie pracy. Ponadto wyjaśniłem, jakie wzorce sprawdzają się najlepiej podczas pracy z kodem Terraform.

W celu przedstawienia wspomnianych wzorców w książce znalazło się wiele przykładowych fragmentów kodu. Postarałem się, aby były one jak najłatwiejsze go samodzielnego wypróbowania, stąd maksymalne ograniczenie zależności od podmiotów zewnętrznych. Niemal wszystkie przykłady wykorzystują tylko jednego dostawcę chmury, AWS, więc musisz założyć konto dla tylko jednej usługi. (AWS oferuje dość sporo zasobów dla użytkowników kont bezpłatnych i uruchomienie przedstawionych przykładów nie powinno Cię nic kosztować). W książce nie znajdziesz zatem omówienia płatnych usług HashiCorp: Terraform Pro i Terraform Enterprise, także zaprezentowane fragmenty kodu nie wymagają tych usług. Wszystkie przykładowe fragmenty kodu udostępniłem jako oprogramowanie typu open source.

Przykładowe fragmenty kodu udostępnione jako open source

Wszystkie oryginalne przykładowe fragmenty kodu zamieszczone w książce znajdziesz w repozytorium pod adresem:

<https://github.com/brikis98/terraform-up-and-running-code>

Pobierz to repozytorium jeszcze przed przystąpieniem do lektury książki, co pozwoli na wypróbowanie przykładów w komputerze lokalnym:

```
$ git clone https://github.com/brikis98/terraform-up-and-running-code.git
```

Przykłady w repozytorium zostały podzielone na poszczególne rozdziały. Warto w tym miejscu dodać, że większość przykładów zawiera kod w postaci, w jakiej on znajduje się na *końcu* rozdziału. Jeżeli chcesz nauczyć się jak najwięcej, lepiej wpisuj przykłady samodzielnie, zupełnie od początku.

Tworzenie kodu rozpocznie się w rozdziale 2., z którego dowiesz się, jak używać Terraform do wdrożenia podstawowego klastra serwerów WWW zupełnie od początku. Dzięki wykonaniu poleceń zawartych w kolejnych rozdziałach zorientujesz się, jak opracować i usprawnić ten przykład klastra serwerów WWW.

Wprowadzaj przedstawiane zmiany i staraj się wpisywać kod samodzielnie, a podane wcześniej repozytorium GitHub potraktuj jedynie jako ostatnią deskę ratunku w sytuacji, gdy utkniesz i naprawdę nie będziesz potrafił sobie poradzić.



Wszystkie przykłady przedstawione w książce zostały przetestowane wraz z wydaniem Terraform 0.12.x, które było najnowsze w chwili powstawania książki. Skoro Terraform to względnie nowe narzędzie i jeszcze nie osiągnęło wersji 1.0.0, istnieje prawdopodobieństwo, że w przyszłych wydaniach zostaną wprowadzone zmiany niezgodne z wcześniejszymi wydaniem, pewne najlepsze praktyki zaś ulegną zmianie i będą ewoluować na przestrzeni czasu.

Spróbuję wprowadzać uaktualnienia w miarę swoich możliwości, ale projekt Terraform zmienia się na tyle szybko, że być może będziesz musiał samodzielnie poradzić sobie z pewnymi kwestiami. Najnowsze informacje, posty na blogu, a także zapowiedzi konferencji dotyczących Terraform i podejścia DevOps znajdziesz na stronie internetowej książki oraz w newsletterze (zachęcam Cię do zapisania się do niego).

Użycie przykładowych kodów

Książka ta ma na celu pomóc Ci w pracy. Ogólnie rzecz biorąc, można wykorzystywać przykłady z niej w swoich programach i w dokumentacji. Nie trzeba kontaktować się z nami w celu uzyskania zezwolenia, dopóki nie powieła się znaczących ilości kodu. Przykładowo pisanie programu, w którym znajdzie się kilka fragmentów kodu z tej książki, nie wymaga zezwolenia, jednak sprzedawanie lub rozpowszechnianie płyty CD-ROM zawierającej przykłady z książki wydawnictwa O'Reilly już tak. Odpowiedź na pytanie przez cytowanie tej książki lub przykładowego kodu nie wymaga zezwolenia, ale włączenie wielu przykładowych kodów z tej książki do dokumentacji produktu czytelnika już tak.

Jestem wdzięczny za umieszczanie przypisów, ale nie wymagam tego. Przypis zwykle zawiera tytuł, autora, wydawcę i ISBN. Na przykład: Yevgeniy Brikman, *Terraform. Krótkie wprowadzenie. Tworzenie infrastruktury za pomocą kodu*. Wydanie II, ISBN 978-83-283-6649-7, Helion, Gliwice 2020.

Konwencje zastosowane w książce

W tej książce zastosowano następujące konwencje typograficzne:

Kursywa

Wskazuje na nowe pojęcia, adresy URL i e-mail, nazwy plików, rozszerzenia plików itd.

Czcionka o stałej szerokości

Użyta w przykładowych fragmentach kodu, a także w samym tekście, aby odwołać się do pewnych poleceń lub innych elementów programistycznych, takich jak nazwy zmiennych lub funkcji, bazy danych, typy danych, zmienne środowiskowe, polecenia i słowa kluczowe.

Pogrubiona czcionka o stałej szerokości

Użyta w celu wyeksponowania poleceń lub innego tekstu, który powinien być wprowadzony przez czytelnika.



Taka ikona oznacza ogólną uwagę.



Taka ikona oznacza ostrzeżenie.

Podziękowania

Josh Padnick

Ta książka nie powstałaby bez Ciebie. Pokazałeś mi Terraform, nauczyłeś podstaw i pomogłeś w opanowaniu zagadnień zaawansowanych. Dziękuję za okazaną pomoc i wiedzę, którą później mogłem zamieścić w tej książce. Dziękuję Ci za to, że jesteś fantastycznym współnikiem — to pozwala nam prowadzić razem biznes i nadal cieszyć się życiem. Przede wszystkim dziękuję Ci za to, że jesteś dobrym przyjacielem i dobrym człowiekiem.

O'Reilly Media

Dziękuję za opublikowanie kolejnej mojej książki. Czytanie i pisanie całkowicie zmieniło moje życie i cieszę się, że pomogliście mi podzielić się wiedzą z innymi. Specjalne podziękowania kieruję do Briana Andersona, który pomógł mi podczas pracy nad pierwszym wydaniem książki, i Virgini Wilson, która pomogła mi przy drugim wydaniu.

Pracownicy firmy Gruntwork

Nie jestem w stanie podziękować Wam wystarczająco (a) za dołączenie do niewielkiego startupu, (b) za tworzenie wspaniałego oprogramowania, (c) za cierpliwość, gdy byłem zajęty przygotowaniem wydania drugiego książki, (d) za to, że jesteście wspaniałymi kolegami i przyjaciółmi.

Klienci firmy Gruntwork

Dziękuję, że daliście szansę małej i nieznannej wcześniej firmie, a tym samym staliście się królikami doświadczalnymi podczas naszych eksperymentów z Terraform. Celem firmy Gruntwork jest maksymalne ułatwienie procesu poznawania, tworzenia i wdrażania oprogramowania. Nie zawsze udało nam się zrealizować te cele (wiele naszych błędów uwzględniłem w tej książce) i dlatego jestem Wam wdzięczny za cierpliwość i chęć pomocy w usprawnieniu świata oprogramowania.

HashiCorp

Dziękuję za przygotowanie kolekcji wspaniałych narzędzi stosowanych w podejściu DevOps, m.in. Terraform, Packer, Consul i Vault. Usprawniliśmy świat DevOps dzięki tym narzędziom, które stały się źródłem utrzymania dla milionów programistów.

Kief Morris, Seth Vargo, Mattias Gees, Ricardo Ferreira, Akash Mahajan, Moritz Heiber

Dziękuję za przeczytanie wstępnych wersji książki i przekazanie wielu szczegółowych, konstruktywnych uwag. Wasze sugestie pomogły w znacznym poprawieniu tej książki.

Czytelnicy pierwszego wydania

Dzięki czytelnikom pierwszego wydania książki możliwe było przygotowanie drugiej edycji. Dziękuję wam za to. Wasze uwagi, pytania, zgłoszenia i nieustanne zgłaszanie uaktualnień zmotywowały mnie do napisania około 160 nowych stron. Mam nadzieję, że ten nowy materiał uznacie za użyteczny, i nadal czekam na Wasze opinie.

Mama, Tata, Larisa, Molly

Przypadkowo napisałem kolejną książkę. To prawdopodobnie oznacza, że nie spędziłem z Wami tyle czasu, ile bym chciał. Dziękuję za okazane wsparcie. Kocham Was.

Dlaczego Terraform?

Oprogramowanie nie jest uznawane za gotowe, gdy kod działa w komputerze programisty. Nie jest również gotowe po zaliczeniu wszystkich testów lub gdy ktoś stwierdzi: „Można wydać tę aplikację”. Oprogramowanie nie może być uznane za gotowe aż do chwili jego *dostarczenia* użytkownikowi.

Dostarczanie oprogramowania oznacza wykonanie pracy niezbędnej w celu udostępnienia kodu klientowi, np. uruchomienie tego kodu w serwerach produkcyjnych, utworzenie kodu w sposób odporny na przestój lub maksymalne obciążenie sieci, a także zapewnienie ochrony kodu przed atakami. Zanim zagłębisz się w szczegóły związane z Terraform, warto wykonać krok wstecz i spojrzeć z szerszej perspektywy na to, jak Terraform wpasowuje się w proces dostarczania oprogramowania.

W rozdziale zostaną poruszone zagadnienia:

- powstanie ruchu DevOps,
- infrastruktura jako kod,
- korzyści z infrastruktury jako kodu,
- sposób działania Terraform,
- porównanie Terraform z innymi narzędziami infrastruktury jako kodu.

Powstanie ruchu DevOps

W nie tak odległej przeszłości, jeśli chciało się zbudować firmę zajmującą się tworzeniem oprogramowania, trzeba było zajmować się również zarządzaniem mnóstwem sprzętu komputerowego. Konieczne było przygotowanie szafek i umieszczenie w nich serwerów w obudowach typu rack, wykonanie niezbędnych połączeń między poszczególnymi urządzeniami, przygotowanie chłodzenia, utworzenie awaryjnego systemu zasilania itd. Sensowne wydawało się posiadanie jednego zespołu, zwykle nazywanego programistami (ang. *developers*), odpowiedzialnego za tworzenie oprogramowania, i drugiego zespołu, zwykle określanego operacyjnym (ang. *operations*), odpowiedzialnego za zarządzanie dostępnym sprzętem komputerowym.

Zespół programistów tworzył aplikację, a następnie przekazywał ją zespołowi operacyjnemu, którego zadaniem było ustalenie, jak ją wdrożyć i uruchomić. Większość zadań była wykonywana ręcznie. Po części było to nieuniknione, ponieważ większość pracy wiązała się fizycznie z urządzeniami (np. układanie serwerów, łączenie urządzeń kablami itd.). Jednak nawet związane z oprogramowa-

niem zadania w zespole operacyjnym, takie jak instalowanie aplikacji i jej zależności, bardzo często były wykonywane ręcznie przez wydawanie poleceń w serwerze.

Wprawdzie na początku takie rozwiązanie się sprawdza, ale wraz z rozwojem i ze wzrostem firmy pojawiają się problemy. Najczęściej spotykamy się z następującą sytuacją: ponieważ wydania są realizowane ręcznie, wraz ze wzrostem liczby serwerów wydania stają się wolne, bolesne i nieprzewidywalne. Zespół operacyjny czasami popełnia błędy, czego efektem są *minimalne różnice* w konfiguracji poszczególnych serwerów (ten problem jest często określany mianem *zmiany konfiguracji*). To z kolei przekłada się na wzrost liczby błędów. Programiści bronią się twierdzeniem „to działa w moim komputerze”, a przestoje pojawiają się znacznie częściej.

Pracownicy działu operacyjnego, zmęczeni telefonami o trzeciej w nocy po każdym nowym wydaniu oprogramowania, zmniejszają częstotliwość tych wydań do jednego tygodniowo. Następnie do jednego miesięcznie, a później do jednego co pół roku. Na tygodnie przed wydaniem oprogramowania w danym półroczu zespoły próbują ujednolicić projekty, co prowadzi do ogromnego bałaganu i rodzi konflikty. Nikt nie potrafi ustabilizować gałęzi zawierającej wersję oprogramowania przeznaczoną do wydania. Zespoły zaczynają nawzajem zrzucać na siebie odpowiedzialność. Sytuacja staje się trudna i wydaje się, że firma wkrótce stanie.

Obecnie jesteśmy świadkami ogromnej zmiany w tym zakresie. Zamiast zarządzać własnymi centrami danych, wiele firm korzysta z chmury i czerpie korzyści z dostępności usług takich jak Amazon Web Services (AWS), Microsoft Azure i Google Cloud Platform (GCP). Zamiast inwestycji ogromnych środków w sprzęt wiele zespołów operacyjnych zajmuje się pracą nad oprogramowaniem, wykorzystując do tego narzędzia takie jak Chef, Puppet, Terraform i Docker. Zamiast zmagać się z ustawianiem serwerów i łączeniem przewodów sieciowych, wielu administratorów systemów zajmuje się tworzeniem kodu.

W efekcie zespoły programistyczny i operacyjny poświęcają większość czasu na pracę nad oprogramowaniem, a granica między nimi powoli się zaciera. Nadal rozsądne jest posiadanie oddzielnego zespołu programistów odpowiedzialnych za obsługę kodu aplikacji i zespołu operacyjnego odpowiedzialnego za obsługę kodu operacyjnego, choć nie ulega wątpliwości, że obie te grupy muszą ściślej ze sobą współpracować. W taki sposób dotarliśmy do *ruchu DevOps*.

DevOps nie jest nazwą zespołu, stanowiska lub konkretnej technologii. To raczej zbiór procesów, idei i technik. Każdy ma nieco inną definicję *DevOps*, ale na potrzeby materiału przedstawionego w książce będę wykorzystywał następującą:

Celem DevOps jest znacznie efektywniejsze dostarczanie oprogramowania.

Zamiast wielodniowych, kosztownych operacji łączenia projektów kod jest integrowany nieustannie i zawsze pozostaje w stanie pozwalającym na jego wdrożenie. Zamiast raz w miesiącu wdrożenia kodu mogą być przeprowadzane wielokrotnie w ciągu dnia, a nawet wraz z każdą operacją przekazania kodu do repozytorium. Ponadto zamiast stałych przestojów tworzy się odporny i samonaprawiający się system, a rozwiązania z zakresu monitorowania i ostrzegania wykorzystuje do wychwytywania problemów, które nie mogą być usunięte automatycznie.

Wyniki firm, które zdecydowały się na zastosowanie podejścia DevOps, są zdumiewające. Przykładowo firma Nordstorm przekonała się, że zastosowanie praktyk DevOps w organizacji pozwoliło na zwiększenie o 100% liczby funkcji dostarczanych każdego miesiąca, skrócenie liczby usterek o połowę, skrócenie o 60% czasu realizacji (ang. *lead time*) — w tym kontekście to opóźnienie między pojawieniem się pomysłu i uruchomienie kodu w środowisku produkcyjnym — oraz zmniejszenie liczby incydentów produkcyjnych o 60 – 90%. Gdy dział LaserJet Firmware w HP zaczął stosować praktyki DevOps, czas poświęcany przez programistów na tworzenie kodu wzrósł z 5% do 40%, a ogólny koszt prac programistycznych spadł o 40%. Z kolei firma Etsy wykorzystała praktyki DevOps w celu przejścia od stresujących i rzadkich wdrożeń powodujących przestoje i awarie do wielokrotnych wdrożeń w ciągu dnia (od 25 do 50) wraz ze znacznie niższą liczbą przestojów¹.

Mamy cztery podstawowe wartości w ruchu DevOps — są to: kultura, automatyzacja, pomiar i współdzielenia, co czasami jest określane akronimem CAMS (ang. *culture, automation, measurement, sharing*). Ta książka nie jest wyczerpującym przewodnikiem po ruchu DevOps (materiały na ten temat, z którymi warto się zapoznać, wymienilem w dodatku A), więc zamierzam skoncentrować się tylko na jednej z wymienionych wartości: automatyzacji.

Celem jest jak największa automatyzacja procesu dostarczania oprogramowania. To oznacza zarządzanie infrastrukturą nie przez klikanie na stronie internetowej lub ręczne wydawanie poleceń w powłoce, ale za pomocą kodu. Ta koncepcja jest zwykle określana mianem *infrastruktura jako kod*.

Infrastruktura jako kod

Idea stojąca za infrastrukturą jako kodem (ang. *infrastructure as code*, IaC) polega na tworzeniu i wykonywaniu kodu w celu zdefiniowania, uaktualnienia i usunięcia infrastruktury. To pokazuje ważną zmianę w nastawieniu, polegającą na tym, że wszystkie aspekty operacji są traktowane jako oprogramowanie — nawet te związane z przestawieniem sprzętu (np. fizyczne umieszczenie serwera w pewnym miejscu). Przy czym kluczowe znaczenie w praktykach DevOps ma to, że niemalże *wszystkim* można zarządzać w kodzie: serwerami, bazami danych, sieciami, plikami dzienników zdarzeń, konfiguracją aplikacji, dokumentacją, testami zautomatyzowanymi, procesami wdrażania itd.

Istnieje pięć szerokich kategorii narzędzi IaC:

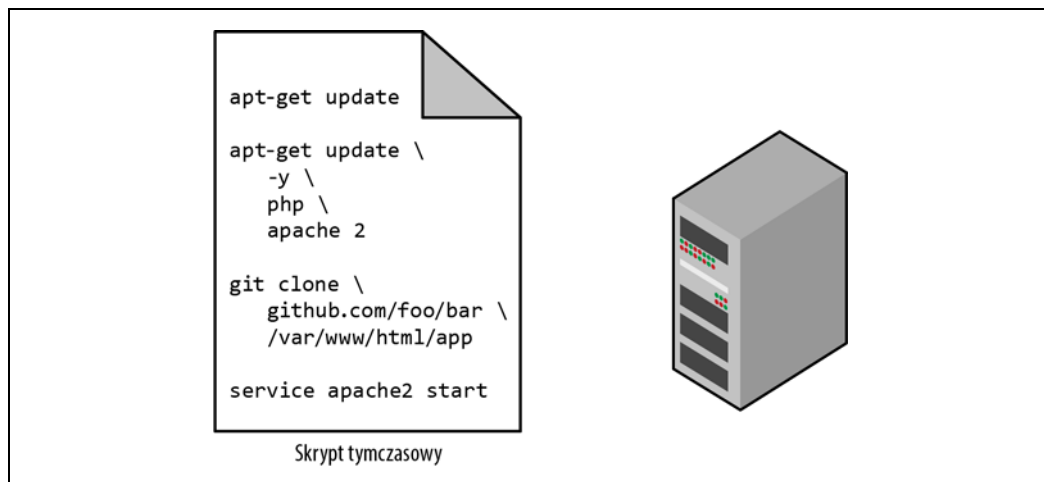
- skrypty tymczasowe,
- narzędzia zarządzania konfiguracją,
- narzędzia szablonów serwera,
- narzędzia instrumentacji,
- narzędzia provisioningu.

Dalej po kolei przedstawię te kategorie.

¹ Te informacje pochodzą z książki *DevOps. Światowej klasy zwinność, niezawodność i bezpieczeństwo w Twojej organizacji* (Helion), której autorami są Gene Kim, Patrick Debois, John Willis, Jez Humble i John Allspaw.

Skrypty tymczasowe

Najprostsze podejście w zakresie automatyzacji czegokolwiek polega na utworzeniu *skryptu tymczasowego*. Zadanie przeznaczone do ręcznego wykonania dzielisz na kolejne kroki, a następnie używasz ulubionego języka skryptowego (np. Bash, Ruby, Python) do zdefiniowania poszczególnych kroków w kodzie i wykonujesz skrypt w serwerze, jak pokazałem na rysunku 1.1.



Rysunek 1.1. Uruchamianie skryptu tymczasowego w serwerze

Dla przykładu spójrz na przedstawiony tutaj skrypt Bash o nazwie *setup-webserver.sh* przeprowadzający konfigurację serwera przez zainstalowanie zależności, pobranie kodu z repozytorium Git i uruchomienie serwera WWW Apache:

```
# Uaktualnienie bufora narzędzia apt-get.
```

```
sudo apt-get update
```

```
# Instalacja PHP i Apache.
```

```
sudo apt-get install -y php apache2
```

```
# Pobranie kodu z repozytorium.
```

```
sudo git clone https://github.com/briki98/php-app.git /var/www/html/app
```

```
# Uruchomienie serwera Apache.
```

```
sudo service apache2 start
```

Ogromną zaletą i jednocześnie największą wadą skryptów tymczasowych jest możliwość użycia popularnych języków programowania ogólnego przeznaczenia i utworzenie kodu w dowolny sposób.

Podczas gdy narzędzia opracowane specjalnie z myślą o IaC dostarczają spójne API przeznaczone do wykonywania skomplikowanych zadań, to jeśli używasz języka programowania ogólnego przeznaczenia, musisz tworzyć niestandardowy kod dla każdego zadania. Co więcej, narzędzia zaprojektowane dla IaC zwykle wymuszają stosowanie określonej struktury kodu, w przypadku języków programowania ogólnego przeznaczenia zaś każdy programista ma własny styl i inaczej wykonuje pewne zadania. Żadna z wymienionych kwestii nie stanowi poważnego problemu w ośmiowierszowym skrypcie instalującym serwer Apache, ale sytuacja szybko wymknie się spod kontroli, gdy skrypty tymczasowe

będą używane do zarządzania dziesiątkami serwerów, baz danych, mechanizmów równoważenia obciążenia, konfiguracji sieciowych itd.

Jeżeli kiedykolwiek musiałeś obsługiwać ogromne repozytorium skryptów Bash, doskonale wiesz, że praktycznie zawsze prowadzi to do powstania niemożliwego w zarządzaniu tzw. *kodu spaghetti*. Skrypty tymczasowe doskonale sprawdzają się podczas wykonywania jednorazowych zadań. Jeżeli zamierzasz zarządzać całą infrastrukturą jako kodem, powinieneś zdecydować się na dedykowane IaC narzędzie opracowane do wykonywania konkretnych zadań.

Narzędzia zarządzania konfiguracją

Chef, Puppet, Ansible i SaltStack to przykłady *narzędzi zarządzania konfiguracją*, co oznacza, że zostały zaprojektowane do instalowania oprogramowania w istniejących serwerach oraz zarządzania nim. Dla przykładu w kolejnym fragmencie kodu przedstawiłem rolę *Ansible* o nazwie *web-server.yml* odpowiedzialną za taką samą konfigurację serwera WWW Apache, jaka wcześniej była przeprowadzana w skrypcie *setup-webserver.sh*.

```
- name: Uaktualnienie bufora narzędzia apt-get.
  apt:
    update_cache: yes

- name: Instalacja PHP.
  apt:
    name: php

- name: Instalacja Apache.
  apt:
    name: apache2

- name: Pobranie kodu z repozytorium.
  git: repo=https://github.com/brikis98/php-app.git dest=/var/www/html/app

- name: Uruchomienie serwera Apache.
  service: name=apache2 state=started enabled=yes
```

Ten kod jest podobny do użytego w skrypcie Bash, ale wykorzystanie narzędzia takiego jak Ansible ma wiele zalet, z których tutaj wymieniałem tylko kilka:

Konwencje tworzenia kodu

Ansible wymusza spójność, przewidywalną strukturę, dołączanie dokumentacji, stosowanie pewnego układu plików, czytelne nazwy parametrów, zarządzanie informacjami niejawnymi itd. Podczas gdy każdy programista tworzy skrypty tymczasowe w odmienny sposób, większość narzędzi zarządzania konfiguracją jest dostarczana wraz z zestawem konwencji ułatwiających poruszanie się po kodzie.

Powtarzalność

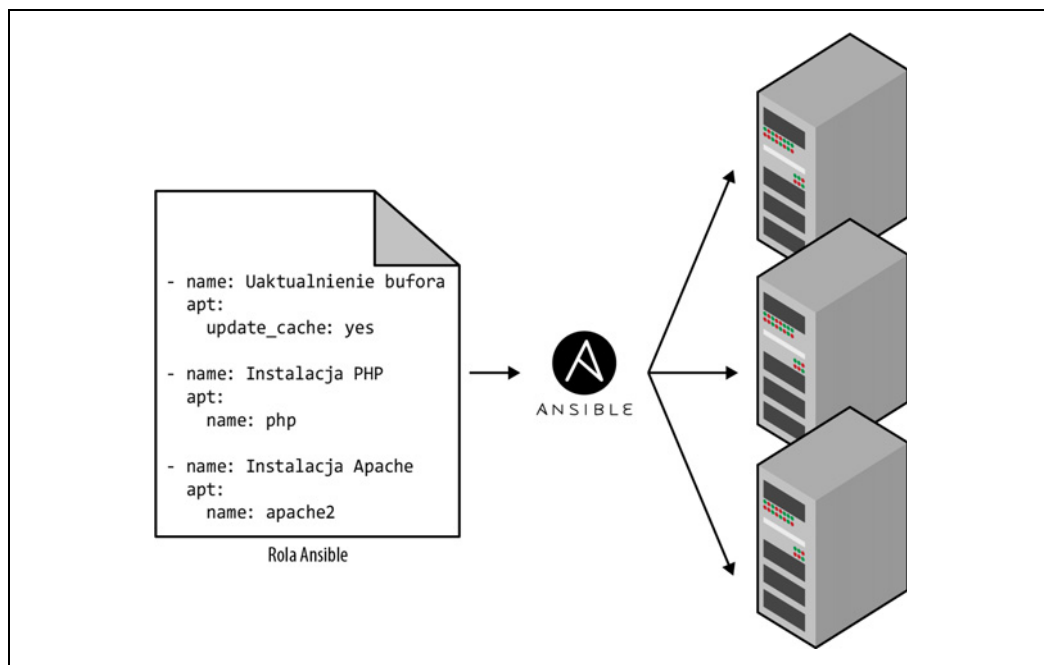
Utworzenie jednorazowo wykonywanego skryptu tymczasowego nie należy do zbyt trudnych zadań. Z kolei opracowanie skryptu tymczasowego, który będzie działał prawidłowo nawet wtedy, gdy jest w ciągłym użyciu, to znacznie trudniejsze zadanie. Za każdym razem, gdy będziesz tworzyć katalog za pomocą kodu w skrypcie, musisz pamiętać o sprawdzeniu, czy ten katalog

istnieje. Za każdym razem, gdy dodasz wiersz konfiguracyjny do pliku, musisz sprawdzić, czy taki wiersz jeszcze nie istnieje. Jeżeli chcesz uruchomić aplikację, musisz sprawdzić, czy nie została uruchomiona już wcześniej.

Kod działający poprawnie niezależnie od liczby jego uruchomień jest nazywany *kodelem powtarzalnym*. Aby zagwarantować powtarzalność przedstawionego wcześniej skryptu Bash, musiałbyś dodać wiele wierszy kodu zawierających dużo konstrukcji `if`. Z kolei większość funkcji Ansible domyślnie zapewnia powtarzalność. Przykładowo kod w pliku `web-server.yml` zainstaluje oprogramowanie Apache tylko, jeśli nie jest ono zainstalowane, spróbuje uruchomić serwer WWW Apache tylko, jeśli nie został uruchomiony wcześniej.

Dystrybucja

Skrypty tymczasowe są przeznaczone do działania w pojedynczym komputerze lokalnym. Ansible i inne narzędzia służące do zarządzania konfiguracją zostały zaprojektowane specjalnie do zarządzania ogromną liczbą zdalnych serwerów, jak możesz zobaczyć na rysunku 1.2.



Rysunek 1.2. Narzędzie zarządzania konfiguracją, takie jak Ansible, może wykonywać kod w ogromnej liczbie serwerów

Przykładowo, aby zastosować rolę `web-server.yml` w pięciu serwerach, trzeba zacząć od utworzenia pliku o nazwie `hosts` zawierającego adresy IP tych serwerów.

```
[webservers]
11.11.11.11
11.11.11.12
11.11.11.13
11.11.11.14
11.11.11.15
```

Teraz można zdefiniować następujący tzw. *playbook Ansible*:

```
- hosts: webservers
  roles:
  - webserver
```

Na końcu można za pomocą przedstawionego polecenia wykonać zdefiniowany kod:

```
ansible-playbook playbook.yml
```

To nakazuje Ansible równoczesne skonfigurowanie wszystkich pięciu serwerów. Ewentualnie za pomocą polecenia o nazwie `serial` umieszczonego we wspomnianym playbooku Ansible można zdefiniować wdrożenie określane mianem *rolling deployment*, które będzie seriami uaktualniało serwery. Dlatego też przypisanie parametrowi `serial` wartości 2 oznacza, że Ansible będzie jednocześnie uaktualniać dwa serwery, a sama operacja zostanie wielokrotnie powtórzona, aż do chwili skonfigurowania wszystkich serwerów (w omawianym przykładzie jest to pięć serwerów). Powielenie tej logiki w skrypcie tymczasowym może zabrać dziesiątki lub nawet setki wierszy kodu.

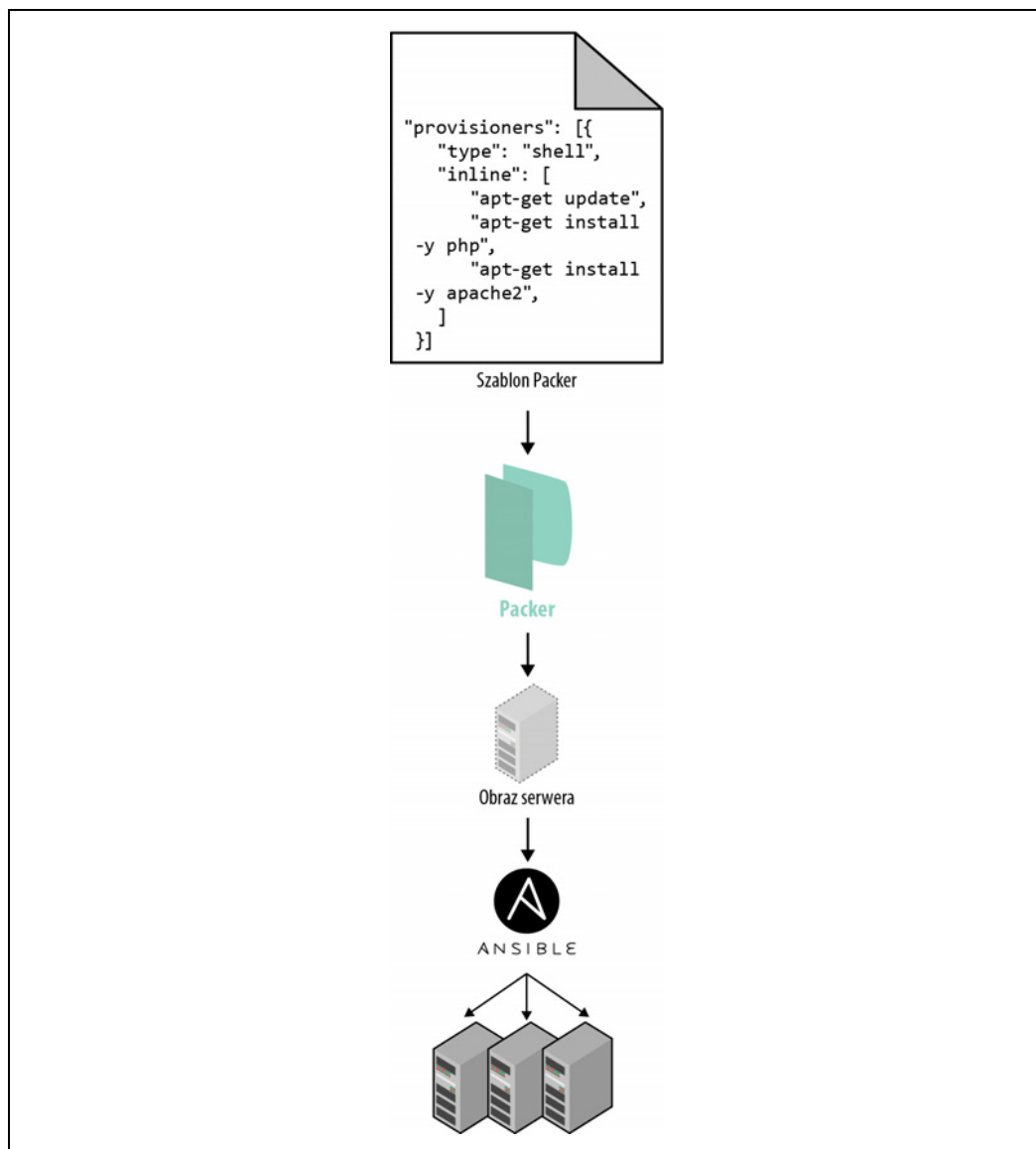
Narzędzia szablonów serwera

Zyskującym ostatnio popularność rozwiązaniem alternatywnym dla zarządzania konfiguracją jest wykorzystanie *narzędzi szablonów serwera*, takich jak Docker, Packer i Vagrant. Zamiast na uruchamianiu ogromnej liczby serwerów i konfigurowaniu ich przez wykonywanie tego samego kodu w każdym z nich idea stojąca za narzędziami szablonów serwera polega na utworzeniu *obrazu* serwera zawierającego pełną „migawkę” systemu operacyjnego (OS), oprogramowania, plików i wszelkich innych ważnych elementów. Następnie za pomocą narzędzia typu IaC można ten obraz zainstalować we wszystkich serwerach, jak pokazałem na rysunku 1.3.

Jak widać na rysunku 1.4, istnieją dwie szerokie kategorie narzędzi przeznaczonych do pracy z obrazami.

Maszyny wirtualne

Maszyna wirtualna (ang. *virtual machine*, VM) emuluje cały system komputerowy, wraz ze sprzętem. Uruchamiasz program tzw. *hipernadzorcę* (ang. *hypervisor*) — taki jak VMware, VirtualBox, Parallels itd. — w celu wirtualizacji (czyli symulowania) procesora, pamięci, dysku twardego i sieci. Zaletą takiego rozwiązania jest to, że każdy *obraz maszyny wirtualnej* uruchamiany przez hipernadzorcę może mieć dostęp jedynie do wirtualizowanego sprzętu, więc tym samym pozostaje w pełni odizolowany od komputera gospodarza i pozostałych obrazów VM. Ponadto będzie działał w dokładnie taki sam sposób we wszystkich środowiskach (w Twoim komputerze, w serwerze działu QA, w serwerze produkcyjnym itd.). Natomiast wadą wirtualizacji jest to, że emulacja całego niezbędnego sprzętu i uruchamianie oddzielnego systemu operacyjnego dla każdej maszyny wirtualnej powoduje duże obciążenie w kategoriach poziomu użycia procesora, pamięci i czasu uruchamiania. Do zdefiniowania obrazów VM jako kodu możesz wykorzystać takie narzędzia jak Packer i Vagrant.

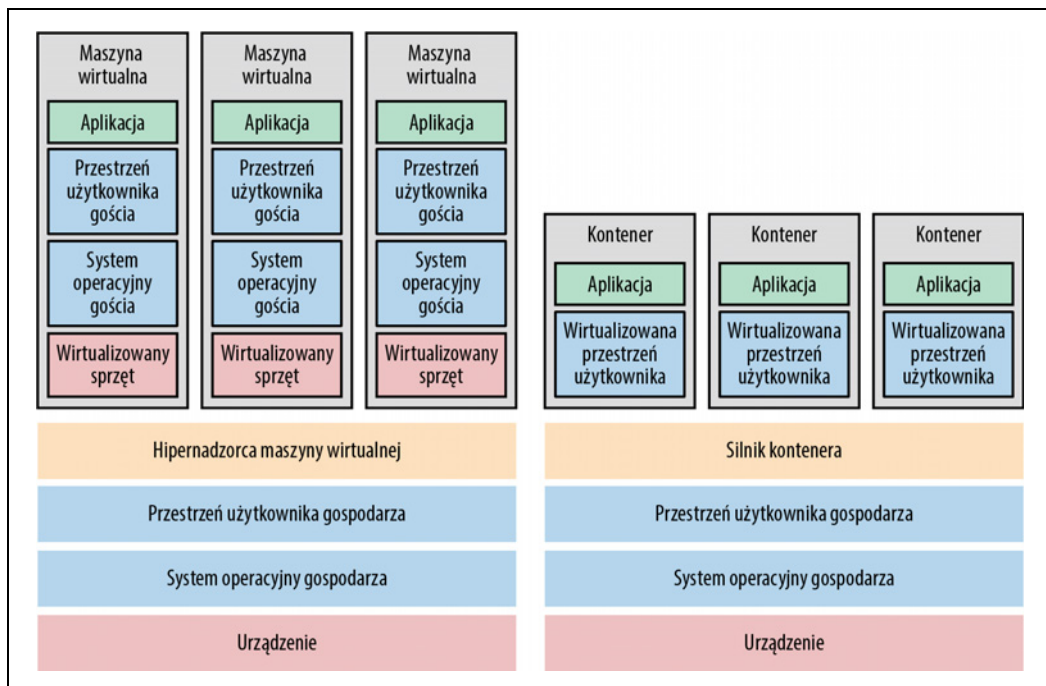


Rysunek 1.3. Narzędzie szablonu serwera, takie jak Packer, pozwala na utworzenie obrazu serwera. Następnie można wykorzystać inne narzędzia, takie jak Ansible, do zainstalowania tego obrazu we wszystkich serwerach

Kontenery

Kontener emuluje przestrzeń użytkownika systemu operacyjnego². Uruchamiasz tzw. *silnik kontenera*, taki jak Docker, CoreOS rkt, cri-o, aby w ten sposób utworzyć odizolowane procesy, obszar pamięci, punkty montowania i sieć. Zaletą takiego podejścia jest to, że każdy kontener

² W większości nowoczesnych systemów operacyjnych kod działa w dwóch „przestrzeniach”: *jądra* i *użytkownika*. Kod uruchomiony w przestrzeni jądra ma bezpośredni i niczym nieograniczony dostęp do całego urządzenia.



Rysunek 1.4. Dwa podstawowe rodzaje obrazów: maszyny wirtualne (po lewej) i kontenery (po prawej). Maszyna wirtualna przeprowadza wirtualizację sprzętu, natomiast kontener jedynie przestrzeni użytkownika

uruchomiony przez silnik kontenera ma dostęp jedynie do własnej przestrzeni użytkownika, więc pozostaje odizolowany od komputera gospodarza oraz pozostałych kontenerów. Ponadto kontener działa w dokładnie taki sam sposób we wszystkich środowiskach (w Twoim komputerze, w serwerze działu QA, w serwerze produkcyjnym itd.). Natomiast wadą tego podejścia jest to, że wszystkie kontenery działające w pojedynczym serwerze współdzielą sprzęt i jądro systemu operacyjnego, więc znacznie trudniej jest osiągnąć poziom izolacji i bezpieczeństwa oferowany przez maszyny wirtualne³. Jednak ze względu na współdzielenie jądra i zasobów sprzętowych uruchomienie kontenera może zabrać jedynie kilka milisekund, a sam kontener praktycznie

Nie zostały nałożone żadne ograniczenia w zakresie zabezpieczeń (tzn. można wykonać każdą instrukcję procesora, uzyskać dostęp do każdego miejsca na dysku twardym, zapisać dane w każdej komórce pamięci) i bezpieczeństwa (awaria w przestrzeni jądra najczęściej prowadzi do awarii całego komputera). Dlatego też przestrzeń jądra jest zwykle zarezerwowana dla działających na niskim poziomie, najbardziej zaufanych funkcji systemu operacyjnego (zazwyczaj nazywanych jądrem). Natomiast kod działający w przestrzeni użytkownika nie ma żadnego bezpośredniego dostępu do urządzenia i musi korzystać z API udostępnionego przez jądro systemu operacyjnego. Wspomniane API może wymuszać pewne ograniczenia zabezpieczeń (np. uprawnienia użytkownika) i bezpieczeństwa (np. awaria w przestrzeni użytkownika zwykle wpływa jedynie na daną aplikację) i dlatego praktycznie cały kod aplikacji jest uruchamiany w przestrzeni użytkownika.

³ Ogólnie rzecz biorąc, kontenery zapewniają poziom izolacji wystarczający do uruchamiania własnego kodu. Jeżeli chcesz uruchamiać kod opracowany przez podmioty zewnętrzne (np. tworzysz własnego dostawcę chmury), który może aktywnie podejmować podejrzone działania, lepiej jest skorzystać z oferowanych przez maszyny wirtualne zalet większej izolacji.

nie stanowi żadnego obciążenia dla procesora i pamięci. Obrazy kontenerów jako kodu można zdefiniować za pomocą takich narzędzi jak Docker i CoreOS rkt.

Dla przykładu spójrz na szablon narzędzia Packer o nazwie *web-server.json*, tworzący tzw. obraz maszyny Amazon (ang. *amazon machine image*, AMI), czyli obraz maszyny wirtualnej możliwy do uruchomienia w chmurze AWS:

```
{
  "builders": [{
    "ami_name": "packer-example",
    "instance_type": "t2.micro",
    "region": "us-east-2",
    "type": "amazon-ebs",
    "source_ami": "ami-0c55b159cbfaffe1f0",
    "ssh_username": "ubuntu"
  }],
  "provisioners": [{
    "type": "shell",
    "inline": [
      "sudo apt-get update",
      "sudo apt-get install -y php apache2",
      "sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app"
    ],
    "environment_vars": [
      "DEBIAN_FRONTEND=noninteractive"
    ]
  }]
}
```

Ten szablon narzędzia Packer przeprowadza tę samą konfigurację serwera WWW Apache, którą wcześniej widziałeś w przykładzie *setup-webserver.sh*, używając tego samego kodu Bash⁴. Jedyna różnica między tym i poprzednim przykładem polega na tym, że szablon Packer nie uruchamia serwera WWW Apache (za pomocą wywołania `sudo service apache2 start`). To wynika z faktu stosowania szablonów zwykle do instalowania oprogramowania w obrazach, więc to oprogramowanie powinno działać właściwie tylko po uruchomieniu danego obrazu, np. przez wdrożenie go w serwerze.

Utworzenie obrazu AMI na podstawie tego szablonu odbywa się po wydaniu polecenia `packer build web-server.json`. Po zakończeniu procesu tworzenia obrazu można go umieścić we wszystkich swoich serwerach AWS, skonfigurować każdy z nich do uruchomienia serwera WWW Apache po uruchomieniu serwera AWS (przykład takiego rozwiązania przedstawię w dalszej części rozdziału), a będą one działały w dokładnie taki sam sposób.

Warto w tym miejscu wspomnieć, że poszczególne narzędzia szablonów serwera mają nieco odmienne przeznaczenie. Packer jest zwykle używany do tworzenia obrazów działających na bazie serwerów produkcyjnych — przykładem może być pokazane tutaj utworzenie obrazu AMI dla konta produkcyjnego AWS. Narzędzie Vagrant jest zwykle używane do tworzenia obrazów działających w komputerach programistów, podobnie jak wykorzystujesz aplikację VirtualBox do tworzenia obrazów uruchamianych w swoim komputerze lokalnym działającym pod kontrolą systemu Linux, macOS lub Windows. Docker jest zwykle wykorzystywany do tworzenia obrazów poszczególnych aplikacji.

⁴ Jako alternatywę dla Bash'a narzędzie Packer pozwala również na skonfigurowanie obrazów za pomocą narzędzia zarządzania konfiguracją, takiego jak Ansible lub Chef.

Kontenery Dockera mogą działać w komputerach produkcyjnych lub programistycznych, o ile inne narzędzie skonfigurowało ten komputer do pracy z Docker Engine. Przykładowo powszechnie stosowanym wzorcem jest utworzenie obrazu AMI wraz z zainstalowanym silnikiem Dockera, wdrożenie tego obrazu AMI w klastrze serwerów w ramach swojego konta AWS, a następnie wdrożenie poszczególnych kontenerów Dockera w klastrze, aby w ten sposób móc uruchamiać opracowane aplikacje.

Szablony serwerów to komponenty o znaczeniu kluczowym podczas przejścia do *infrastruktury niemodyfikowalnej*. Ta idea powstała na skutek zaczerpnięcia inspiracji z programowania funkcyjnego, w którym wartość zmiennej po jej zdefiniowaniu nigdy nie ulega zmianie. Jeżeli chcesz cokolwiek uaktualnić, tworzysz nową zmienną. Skoro zmienne nigdy się nie zmieniają, znacznie łatwiej jest uzasadnić potrzebę utworzenia danego fragmentu kodu.

Idea stojąca za infrastrukturą niezmienną jest podobna: po wdrożeniu serwera nigdy nie wprowadzasz w nim zmian. Jeżeli zachodzi potrzeba uaktualnienia czegokolwiek, np. wdrożenia nowej wersji kodu, tworzysz nowy obraz na podstawie szablonu serwera i wdrażasz go w nowym serwerze. Skoro serwer nigdy się nie zmienia, znacznie łatwiej jest uzasadnić potrzebę jego wdrożenia.

Narzędzia instrumentacji

Wprawdzie narzędzia szablonów serwera sprawdzają się doskonale podczas tworzenia maszyn wirtualnych i kontenerów, ale jak faktycznie można nimi zarządzać? W większości przypadków konieczne jest wykonanie przedstawionych tutaj zadań:

- Wdrażanie maszyn wirtualnych i kontenerów, co pozwala na efektywne wykorzystanie sprzętu.
- Przygotowanie uaktualnień dla istniejącej floty maszyn wirtualnych i kontenerów z wykorzystaniem strategii takich jak stałe wdrożenia, wdrożenia typu niebieski-zielony, a także tzw. *wdrożenie kanarkowe* (ang. *canary deployment*).
- Monitorowanie stanu maszyn wirtualnych i kontenerów oraz automatyczne zastępowanie uszkodzonych (*automatyczna naprawa*).
- Skalowanie liczby maszyn wirtualnych i kontenerów w górę lub w dół w zależności od obciążenia (automatyczne skalowanie).
- Rozkład ruchu między maszynami wirtualnymi i kontenerami (*mechanizm równoważenia obciążenia*).
- Umożliwienie maszynom wirtualnym i kontenerom wyszukiwania się w sieci i komunikowania poprzez nią (*usługa odkrywania*).

Obsługa tych zadań jest domeną *narzędzi instrumentacji*, takich jak Kubernetes, Marathon/Mesos, Amazon Elastic Container Service (Amazon ECS), Docker Swarm i Nomad. Przykładowo Kubernetes pozwala na zdefiniowanie sposobu zarządzania kontenerami Dockera jako kodem. Najpierw wdrażasz *klaster Kubernetes*, czyli grupę serwerów zarządzanych przez Kubernetes, a następnie używasz jej do uruchamiania kontenerów Dockera. Większość dostawców

chmury oferuje natywną obsługę w zakresie wdrażania zarządzanych klastrów Kubernetes, np. Amazon Elastic Container Service for Kubernetes (Amazon EKS), Google Kubernetes Engine (GKE) i Azure Kubernetes Service (AKS).

Jeśli masz działający klaster, w pliku YAML możesz zdefiniować sposób uruchamiania kontenera Dockera jako kodu.

```
apiVersion: apps/v1
# Użycie obiektu Deployment do wdrożenia wielu replik kontenerów
# Dockera oraz do deklaratywnego przekazywania im uaktualnień.
kind: Deployment

# Metadane dotyczące tego obiektu Deployment, m.in. nazwa.
metadata:
  name: example-app

# Specyfikacja konfigurująca dany obiekt Deployment.
spec:
  # Określenie sposobu wyszukiwania kontenerów przez ten obiekt Deployment.
  selector:
    matchLabels:
      app: example-app
  # Nakazanie obiektowi Deployment uruchomienie
  # trzech replik kontenerów Dockera.
  replicas: 3

  # Określenie sposobu uaktualniania obiektu Deployment. W omawianym przykładzie
  # zostało skonfigurowane nieustanne przekazywanie uaktualnień.
  strategy:
    rollingUpdate:
      maxSurge: 3
      maxUnavailable: 0
    type: RollingUpdate

# To jest szablon dla wdrażanych kontenerów.
template:

  # To są metadane dla kontenerów, m.in. etykiety.
  metadata:
    labels:
      app: example-app

  # Specyfikacja kontenera.
  spec:
    containers:

      # Uruchomienie serwera WWW Apache nasłuchującego na porcie 80.
      - name: example-app
        image: httpd:2.4.39
        ports:
          - containerPort: 80
```

Ten plik nakazuje Kubernetesowi utworzenie tzw. *obiektu Deployment*, czyli deklaratywnego rozwiązania pozwalającego na zdefiniowanie następujących kwestii:

- Jeden lub więcej kontenerów Dockera, które będą razem uruchamiane. Taka grupa kontenerów jest w Kubernetesie określana mianem *pod*. Zdefiniowany w tym fragmencie kodu pod zawiera jeden kontener Dockera, w którym jest uruchamiany serwer WWW Apache.
- Ustawienia dla każdego kontenera Dockera w podzie. W omawianym przykładzie pod konfiguruje serwer WWW Apache w taki sposób, aby nasłuchiwał na porcie 80.
- Liczba kopii (tzw. *replik*) poda uruchomionych w klastrze. W tym przykładzie zostały skonfigurowane trzy repliki. Kubernetes automatycznie ustala, gdzie w klastrze mają zostać wdrożone poszczególne pody, i wykorzystuje przy tym algorytm ustalający optymalny serwer w kategoriach wysokiej dostępności (np. chodzi o uruchamianie podów w oddzielnych serwerach, aby awaria jednego z nich nie doprowadziła do awarii całej aplikacji), zasobów (wybór serwera posiadającego dostępne porty, zasoby procesora, wolną pamięć lub inne zasoby wymagane przez kontener), wydajności (np. próba wybrania serwera o najmniejszym obciążeniu i najmniejszej liczbie kontenerów) itd. Kubernetes nieustannie monitoruje klastę, aby zagwarantować, że w każdej chwili są uruchomione trzy repliki, i automatycznie zastępować nowymi wszelkie pody, które uległy uszkodzeniu lub przestały udzielać odpowiedzi na zapytania.
- Sposób wdrażania uaktualnień. Po wdrożeniu nowej wersji kontenera Dockera przedstawiony wcześniej kod będzie tworzył trzy nowe repliki, sprawdzi poprawność ich działania, a następnie usunie trzy stare repliki.

Ta niewielka liczba wierszy pliku YAML oferuje dość potężne możliwości! Wydanie polecenia `kubectl apply -f example-app.yml` nakazuje Kubernetesowi wdrożenie aplikacji. Później możesz wprowadzić zmiany w pliku YAML i ponownie wydać polecenie `kubectl apply`, aby zastosować uaktualnienie.

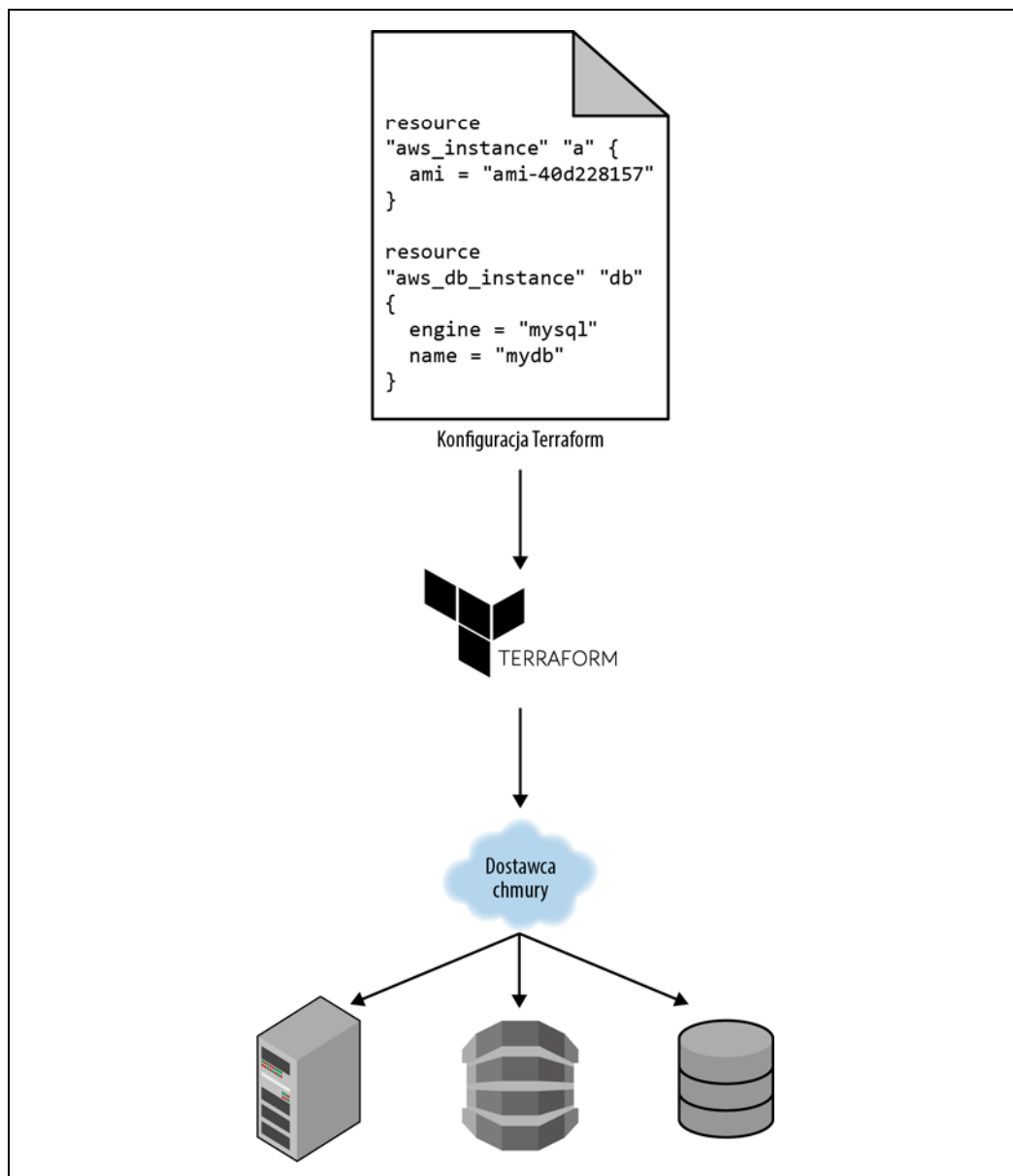
Narzędzia provisioningu

Podczas gdy zarządzanie konfiguracją, szablony serwera i narzędzia instrumentacji definiują kod przeznaczony do uruchomienia w każdym serwerze, *narzędzia provisioningu*, takie jak Terraform, CloudFormation i OpenStack Heat, są odpowiedzialne za utworzenie wspomnianych serwerów. Przy czym narzędzia provisioningu mogą nie tylko tworzyć serwery, ale również bazy danych, bufory, mechanizmy równoważenia obciążenia, kolejki, systemy monitorowania, konfiguracje sieci, ustawienia zapory sieciowej, reguły routingu, certyfikaty SSL (ang. *secure socket layer*) i praktycznie każdy inny aspekt infrastruktury, jak pokazałem na rysunku 1.5.

Przedstawiony tutaj fragment kodu powoduje wdrożenie serwera WWW za pomocą Terraform.

```
resource "aws_instance" "app" {
  instance_type    = "t2.micro"
  availability_zone = "us-east-2a"
  ami              = "ami-0c55b159cbfafa1f0"

  user_data = <<-EOF
    #!/bin/bash
    sudo service apache2 start
  EOF
}
```



Rysunek 1.5. Narzędzia provisioningu wraz z dostawcą chmury pozwalają na tworzenie serwerów, baz danych, mechanizmów równoważenia obciążenia oraz wielu innych komponentów infrastruktury

Nie przejmuj się, jeśli nie znasz użytej składni. W tym momencie skoncentruj się na dwóch parametrach.

ami

Określa identyfikator obrazu AMI przeznaczonego do wdrożenia na serwerze. Temu parametrowi można przypisać wartość identyfikatora obrazu AMI utworzonego w poprzedniej

sekcji za pomocą narzędzia Packer i jego szablonu *web-server.json*, który zawierał kod dotyczący PHP, Apache i aplikacji.

`user_data`

To jest skrypt Bash wykonywany podczas uruchamiania serwera WWW. Omawiany przykład wykorzystuje ten kod do uruchomienia Apache.

Innymi słowy, w omawianym przykładzie pokazałem, jak narzędzia provisioningu i szablony serwera mogą ze sobą współdziałać, co jest często stosowanym wzorcem infrastruktury niemodyfikowalnej.

Korzyści płynące z infrastruktury jako kodu

Skoro poznałeś różne odmiany IaC, być może zastanawiasz się, po co to wszystko. Dlaczego miałbyś uczyć się nowych języków i narzędzi oraz tworzyć kolejny kod, którym będziesz musiał zarządzać?

Odpowiedzią jest to, że masz do czynienia z kodem o potężnych możliwościach. W zamian za inwestycję w postaci zmiany ręcznie wykonywanych zadań na kod bardzo zwiększają się Twoje możliwości w zakresie dostarczania oprogramowania. Zgodnie z 2016 State of DevOps Report (<https://puppet.com/resources/report/2016-state-devops-report/>) organizacje stosujące praktyki DevOps, np. podejście typu IaC, przeprowadzają wdrożenia 200-krotnie częściej, podnoszą się po awarii 24-krotnie szybciej, a czas realizacji w ich przypadku jest 2555-krotnie krótszy niż organizacji niestosujących praktyk DevOps.

Gdy infrastruktura jest zdefiniowana jako kod, można wykorzystać szeroką gamę praktyk tworzenia oprogramowania znacznie usprawniających proces jego dostarczania. Do tego procesu zaliczane są m.in.:

Samoobsługa

Wiele zespołów zajmujących się ręcznym wdrażaniem kodu ma małą liczbę administratorów systemu (często tylko jednego), którzy są jedynymi osobami znającymi magiczne zaklęcia pozwalające na zadziałanie wdrożenia i jako jedyni mają dostęp do środowiska produkcyjnego. To staje się poważnym wąskim gardłem wraz z rozwojem firmy. Jeżeli infrastruktura została zdefiniowana w kodzie, cały proces wdrożenia można zautomatyzować i pozwolić programistom na samodzielne wdrożenia, gdy będą do tego gotowi.

Szybkość i bezpieczeństwo

Po zautomatyzowaniu procesu wdrożenia stanie się on znacznie krótszy, ponieważ komputer jest w stanie wykonywać kroki wdrożenia zdecydowanie szybciej niż człowiek. Ponadto jest to bezpieczniejsze rozwiązanie, jeśli wziąć pod uwagę, że mamy do czynienia z zautomatyzowanym procesem, który jest spójniejszy, powtarzalny i niepodatny na błędy powstające na skutek ręcznego wykonywania zadań.

Dokumentacja

Zamiast zamknąć informacje o stanie infrastruktury w głowie jednego administratora systemu, stan infrastruktury można umieścić w plikach kodu źródłowego, które są czytelne dla każdego. Innymi słowy, podejście IaC działa w charakterze dokumentacji pozwalającej każdemu pracownikowi organizacji na poznanie sposobu działania procesu wdrożenia, nawet jeśli administrator systemu uda się na wakacje.

System kontroli wersji

Pliki kodu źródłowego w podejściu IaC można przechowywać w systemie kontroli wersji. W takim przypadku cała historia infrastruktury jest przechwycona w zapisie zdarzenia operacji przekazania danych do repozytorium (tzw. zatwierdzenia). To staje się narzędziem o potężnych możliwościach podczas debugowania, ponieważ po wystąpieniu problemu możesz zajrzeć do dziennika zdarzeń zatwierdzenia i ustalić, co zmieniło się w infrastrukturze. Drugim krokiem może być rozwiązanie problemu przez zwykłe przywrócenie kodu IaC do wcześniejszej wersji, o której wiadomo, że działa prawidłowo.

Sprawdzanie poprawności

Jeżeli stan infrastruktury został zdefiniowany w kodzie, podczas każdej zmiany można przeprowadzić analizę kodu, wykonać zestaw zautomatyzowanych testów, a także przekazać kod do narzędzi analizy statycznej — wszystkie te praktyki pozwalają na znaczne zmniejszenie ryzyka usterek kodu.

Wielokrotne użycie

Infrastrukturę można umieścić w modułach wielokrotnego użycia, więc zamiast przeprowadzać zupełnie od początku każde wdrożenie każdego produktu w każdym środowisku, możesz opierać się na doskonale znanych, udokumentowanych i przetestowanych w boju komponentach⁵.

Szczyście

Jest jeszcze jeden bardzo ważny i zarazem często niedoceniany powód, dla którego powinieneś stosować podejście IaC: szczęście. Ręczne wdrażanie kodu i zarządzanie infrastrukturą jest nudne i żmudne. Programiści i administratorzy systemów nie lubią tego rodzaju zadań, ponieważ nie wiążą się one z kreatywnością, wyzwaniem, uznaniem itd. Możesz wdrożyć kod działający miesiącami bez zastrzeżeń i nikt tego nie dostrzeże — aż do dnia, w którym nabroisz. To tworzy stresogenne i nieprzyjazne środowisko. Podejście IaC oferuje lepszą alternatywę pozwalającą komputerowi na wykonywanie zadań, w których sprawdza się najlepiej (automatyzacja), a programistom również na robienie tego, w czym są najlepsi (tworzenie kodu).

Skoro dowiedziałeś się, skąd takie duże znaczenie podejścia IaC, kolejnym pytaniem może być, czy Terraform jest najlepszym dla Ciebie narzędziem IaC. Aby móc na nie odpowiedzieć, najpierw musisz zapoznać się z naprawdę krótkim wprowadzeniem do sposobu działania Terraform. Następnie przedstawię porównanie Terraform z innymi popularnymi rozwiązaniami w zakresie stosowania praktyk IaC, czyli Chef, Puppet i Ansible.

⁵ Zajrzyj do przygotowanej przez Gruntwork biblioteki kodu stosującego podejście IaC (<https://gruntwork.io/infrastructure-as-code-library/>).

Jak działa Terraform?

Oto bardzo uogólniony opis sposobu działania Terraform: to narzędzie typu open source utworzone w języku Go przez HashiCorp. Kod Go jest kompilowany na postać pojedynczego pliku binarnego (zamiast po jednym pliku binarnym dla każdego obsługiwanego systemu operacyjnego) o nazwie, która nie powinna być zaskoczeniem: *terraform*.

Ten plik binarny można wykorzystać do wdrożenia infrastruktury z poziomu laptopa lub też utworzyć serwer czy inny komputer — i nie przejmować się przy tym żadną dodatkową infrastrukturą, która pozwoli na tę operację. To jest możliwe, ponieważ w tle plik binarny *terraform* wykonuje wywołania API w imieniu jednego dostawcy lub większej grupy *dostawców*, takich jak AWS, Azure, Google Cloud, DigitalOcean, OpenStack i wielu innych. To oznacza, że Terraform może wykorzystać infrastrukturę tych dostawców do obsługi własnych API serwerów, a także mechanizmów uwierzytelniania stosowanych wraz z tymi dostawcami (np. klucze API, które masz już dla dostawcy AWS).

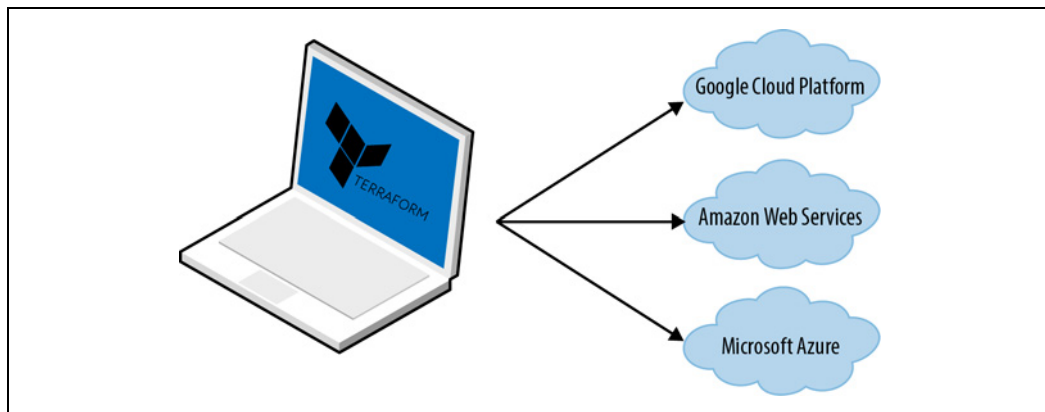
Skąd Terraform wie, które wywołanie API ma zostać wykonane. Odpowiedź kryje się w tworzonych *konfiguracjach Terraform*, które są plikami tekstowymi określającymi infrastrukturę przeznaczoną do utworzenia. Wspomniane konfiguracje to „kod” w wyrażeniu „infrastruktura jako kod”. Spójrz na przykładową konfigurację Terraform.

```
resource "aws_instance" "example" {
  ami      = "ami-0c55b159cbfafa1f0"
  instance_type = "t2.micro"
}

resource "google_dns_record_set" "a" {
  name      = "demo.google-example.com"
  managed_zone = "example-zone"
  type      = "A"
  ttl       = 300
  rrdatas   = [aws_instance.example.public_ip]
}
```

Nawet jeśli nigdy wcześniej nie widziałeś kodu Terraform, nie powinieneś mieć zbyt wielu problemów z ustaleniem sposobu jego działania. Ten fragment kodu nakazuje Terraform wykonanie wywołań API do AWS w celu wdrożenia serwera, a następnie wywołań API do Google Cloud w celu utworzenia wpisu DNS prowadzącego do adresu IP serwera w AWS. Mamy tutaj do czynienia z pojedynczą, prostą składnią (poznasz ją w rozdziale 2.) pozwalającą Terraform na wdrożenie powiązanych ze sobą zasobów między wieloma dostawcami chmury.

Całą infrastrukturę — serwery, bazy danych, mechanizm równoważenia obciążenia, topologię sieci itd. — możesz zdefiniować w plikach konfiguracyjnych Terraform, które następnie trafią do systemu kontroli wersji. Później, wydając polecenia takie jak `terraform apply`, przeprowadzasz wdrożenie tej infrastruktury. Plik binarny *terraform* przetwarza Twój kod, konwertuje go na serię wywołań API do dostawców chmury wymienionych w kodzie, a następnie w jak najefektywniejszy sposób wywołuje te API w Twoim imieniu, jak pokazałem na rysunku 1.6.



Rysunek 1.6. Terraform to plik binarny konwertujący zawartość plików konfiguracyjnych na wywołania API do dostawców chmury

Gdy ktokolwiek w zespole wprowadza zmiany w infrastrukturze, zamiast uaktualniać ją ręcznie i bezpośrednio w serwerze, zmiany nanosi w plikach konfiguracyjnych Terraform, weryfikuje je za pomocą zautomatyzowanych testów i analizy kodu, przekazuje uaktualniony kod do systemu kontroli wersji, a następnie wydaje polecenie `terraform apply` w celu zlecenia Terraform wykonania niezbędnych wywołań API i wdrożenia zmian.



Pełna przenośność między dostawcami chmury

Skoro Terraform obsługuje wielu różnych dostawców chmury, często pojawia się kwestia obsługi *pełnej przenośności* między nimi. Przykładowo, jeśli Terraform wykorzystałeś do zdefiniowania wielu serwerów, baz danych, mechanizmów równoważenia obciążenia i innych komponentów infrastruktury w AWS, czy możesz Terraform wydać polecenie wdrożenia tej samej infrastruktury u innego dostawcy chmury, np. Azure lub Google Cloud, za pomocą kilku kliknięć myszą?

To pytanie okazuje się być fałszywym tropem. Rzeczywistość jest taka, że nie można wdrożyć „dokładnie tej samej infrastruktury” u innego dostawcy chmury, ponieważ poszczególni dostawcy nie oferują dokładnie tych samych typów infrastruktury! Serwery, mechanizmy równoważenia obciążenia i bazy danych oferowane przez AWS różnią się od tych w Azure i Google Cloud pod względem funkcjonalności, konfiguracji, zarządzania, bezpieczeństwa, skalowalności, dostępności, możliwości monitorowania itd. Nie ma łatwego sposobu na „pełne” zniwelowanie tych różnic, zwłaszcza jeśli funkcjonalność dostępna u jednego dostawcy chmury często w ogóle nie istnieje u innych dostawców.

Podejście stosowane przez Terraform pozwala na tworzenie kodu przeznaczonego dla konkretnego dostawcy i wykorzystanie pełni oferowanych przez niego unikatowych możliwości, ale z użyciem tego samego języka, zestawu narzędzi i tych samych praktyk IaC, niezależnie od tego, dla którego dostawcy jest przeznaczony kod.

Porównanie Terraform z innymi narzędziami IaC

Infrastruktura jako kod jest wspaniała, ale proces wyboru narzędzia IaC już niekoniecznie. Funkcjonalność wielu narzędzi IaC nakłada się na siebie. Wiele z nich jest dostępnych jako oprogramowanie typu open source, choć równie dużo to produkty komercyjne. O ile wcześniej nie używałeś takiego narzędzia, prawdopodobnie nie wiesz, jakie kryteria zastosować przy wyborze narzędzia IaC.

Tę sytuację jeszcze bardziej utrudnia fakt, że większość dostępnych porównań narzędzi IaC ogranicza się do zaledwie przedstawienia listy ogólnych właściwości poszczególnych programów. Na jej podstawie można odnieść wrażenie, że każde narzędzie będzie dobre. Wprawdzie z technicznego punktu widzenia to prawda, ale te informacje nie okazują się zbyt pomocne. Można to porównać do stwierdzenia, że początkujący programista osiągnie sukces po utworzeniu witryny internetowej za pomocą języka PHP, C lub asemblera — pod względem technicznym to prawda, mimo to brakuje tutaj wielu informacji, które mają znaczenie podczas dokonywania wyboru.

W kolejnych sekcjach przedstawię dość dokładne porównanie najpopularniejszych narzędzi provisioningu i narzędzi przeznaczonych do zarządzania konfiguracją: Terraform, Chef, Puppet, Ansible, SaltStack, CloudFormation i OpenStack Heat. Moim celem jest umożliwienie Ci określenia, czy Terraform to dobry wybór. Mam zamiar to zrobić poprzez wyjaśnienie, dlaczego moja firma Gruntwork (<https://www.gruntwork.io/>) wybrała Terraform jako narzędzie IaC oraz, w pewnym sensie, dlaczego skłoniło mnie to do napisania tej książki⁶. Podobnie jak w przypadku wszelkich decyzji związanych z technologią, pod uwagę trzeba wziąć kompromisy i priorytety. Nawet jeśli Twoje priorytety będą inne niż moje, mam nadzieję, że omówieniem mojego procesu wyboru pomogę Ci w podjęciu decyzji.

Oto najważniejsze kompromisy, o których trzeba pamiętać:

- zarządzanie konfiguracją kontra provisioning,
- infrastruktura niemodyfikowalna kontra modyfikowalna,
- język proceduralny kontra deklaracyjny,
- serwer główny kontra jego brak,
- agent kontra jego brak,
- duża społeczność kontra mała,
- rozwiązanie dojrzałe kontra najnowsze,
- używanie razem wielu narzędzi.

Zarządzanie konfiguracją kontra provisioning

Jak miałeś okazję zobaczyć wcześniej, Chef, Puppet, Ansible i SaltStack to narzędzia zarządzania konfiguracją, podczas gdy CloudFormation, Terraform i OpenStack Heat to narzędzia provisioningu. Wprawdzie granica między tymi typami narzędzi nie jest do końca wyraźnie ustalona, ale

⁶ Docker, Packer i Kubernetes nie zostały uwzględnione w porównaniu, ponieważ te rozwiązania mogą być stosowane w połączeniu z dowolnymi narzędziami provisioningu i zarządzania konfiguracją.

biorąc pod uwagę to, że zwykle narzędzia konfiguracji mogą do pewnego stopnia zajmować się provisioningiem (np. Ansible pozwala na wdrożenie serwera), narzędzia provisioningu zaś pozwalają na przeprowadzanie konfiguracji (np. masz możliwość wykonywania skryptów konfiguracyjnych w serwerach przygotowanych przez Terraform), najczęściej wybierasz narzędzie najlepiej dopasowane do danej sytuacji⁷.

W szczególności jeśli korzystasz z narzędzia szablonów serwera, np. Dockera lub Packera, odpada większość potrzeb związanych z zarządzaniem konfiguracją. Po utworzeniu obrazu na podstawie pliku *Dockerfile* lub szablonu Packer pozostało już tylko przygotowanie infrastruktury przeznaczonej do uruchamiania tych obrazów. Jeżeli chodzi o provisioning, najlepszym rozwiązaniem jest narzędzie provisioningu.

Zatem, jeśli nie używałeś narzędzi szablonów serwerów, dobrą alternatywą jest wspólne zastosowanie narzędzia konfiguracji i narzędzia provisioningu. Przykładowo Terraform możesz wykorzystać do przygotowania serwerów, które następnie skonfigurujesz za pomocą narzędzia Chef.

Infrastruktura niemodyfikowalna kontra modyfikowalna

Narzędzia konfiguracji takie jak Chef, Puppet, Ansible i SaltStack zwykle domyślnie stosują paradygmat infrastruktury niemodyfikowalnej. Przykładowo, jeśli nakażesz narzędziu Chef zainstalowanie nowej wersji OpenSSL, nastąpi uruchomienie procesu aktualizacji oprogramowania w istniejących serwerach i zmiany zostaną w nich wprowadzone. Wraz z upływem czasu i kolejnymi aktualizacjami każdy serwer ma unikatową historię zmian. W efekcie poszczególne serwery nieco się różnią od siebie, co prowadzi do powstawania drobnych błędów konfiguracji, które są trudne do zdiagnozowania i reprodukcji (to jest dokładnie ten sam problem związany ze zmianą konfiguracji jak w przypadku ręcznego zarządzania serwerami, choć zdecydowanie mniej kłopotliwy, gdy stosowane jest narzędzie zarządzania konfiguracją). Nawet po przeprowadzeniu zautomatyzowanych testów te błędy są trudne do wychwycenia — zmiana konfiguracji może działać świetnie w serwerze testowym, a nieco odmiennie w serwerze produkcyjnym, który ma zakumulowane miesiące uaktualnień nieodzwierciedlone w środowisku testowym.

Jeżeli narzędzia provisioningu, takiego jak Terraform, używasz do wdrażania obrazów utworzonych przez Dockera lub Packera, większość „zmian” to rzeczywiste wdrożenia zupełnie nowego serwera. Przykładowo, aby wdrożyć nową wersję OpenSSL, narzędzie Packer musisz wykorzystać do zbudowania obrazu wraz z nową wersją OpenSSL, wdrożyć ten obraz w nowych serwerach, a następnie zakończyć działanie starych serwerów. Skoro każde wdrożenie korzysta z niemodyfikowalnych obrazów nowych serwerów, takie podejście znacznie ogranicza ryzyko wprowadzenia błędów związanych ze zmianą konfiguracji, ponieważ dokładnie wiesz, jakie oprogramowanie działa w poszczególnych serwerach. Ponadto w każdej chwili bardzo łatwo możesz wdrożyć dowolną wcześniejszą wersję oprogramowania (tzn. dowolny z wcześniej utworzonych obrazów). Dzięki temu zautomatyzowane testy są znacznie efektywniejsze, ponieważ niemodyfikowalne obrazy zaliczające testy

⁷ Obecnie granica między narzędziami konfiguracji i narzędziami provisioningu zaciera się jeszcze bardziej, ponieważ część najważniejszych narzędzi konfiguracji została znacznie usprawniona w zakresie obsługi provisioningu, przykładem mogą być tutaj Chef Provisioning (<https://web.archive.org/web/20190930010054/https://docs.chef.io/provisioning.html>) i Puppet AWS Module (<https://github.com/puppetlabs/puppetlabs-aws>).

w środowisku testowym niemal na pewno będą zachowywały się w dokładnie taki sam sposób w środowisku produkcyjnym.

Oczywiście istnieje możliwość wymuszenia na narzędziu zarządzania konfiguracją przeprowadzania niemodyfikowalnych wdrożeń. To jednak nie jest typowe podejście w takich narzędziach, natomiast jest naturalnym sposobem działania narzędzi provisioningu. Warto w tym miejscu wspomnieć, że podejście niemodyfikowalne również ma pewne wady. Przykładowo ponowne tworzenie obrazu na podstawie szablonu serwera i ponowne wdrażanie tego obrazu we wszystkich serwerach z powodu wprowadzenia drobnej zmiany może być niezwykle czasochłonne. Co więcej, niezmiennosc będzie trwała tylko do chwili faktycznego uruchomienia obrazu. Po przygotowaniu i uruchomieniu serwera rozpocznie on wprowadzanie zmian na dysku twardym, czego skutkiem będzie rozpoczęcie wprowadzania drobnych zmian konfiguracji (choć to można przezwyciężyć w przypadku częstych wdrożeń).

Język proceduralny kontra deklaratywny

Chef i Ansible zachęcają do stosowania stylu *proceduralnego*, w którym tworzysz kod określający krok po kroku, jak ma zostać osiągnięty oczekiwany stan końcowy. Terraform, CloudFormation, Salt-Stack, Puppet i OpenStack Heat zachęcają do stosowania stylu bardziej *deklaratywnego*, w którym tworzysz kod określający żądany stan końcowy, narzędzie pozwalające na stosowanie praktyk IaC zaś jest odpowiedzialne za znalezienie sposobu na przejście do tego stanu.

Aby pokazać różnicę między tymi podejściami, posłużę się przykładem. Wyobraź sobie, że chcesz wdrożyć 10 serwerów (*egzemplarze EC2* w AWS) przeznaczonych do uruchomienia obrazu AMI wraz z identyfikatorem `ami-0c55b159cbfafa1f0` (Ubuntu 18.04). Spójrz na uproszczony przykład szablonu Ansible pokazujący, jak osiągnąć żądany efekt za pomocą podejścia proceduralnego.

```
- ec2:
  count: 10
  image: ami-0c55b159cbfafa1f0
  instance_type: t2.micro
```

Oto uproszczony przykład konfiguracji Terraform wykonującej to samo zadanie, ale z zastosowaniem podejścia deklaratywnego:

```
resource "aws_instance" "example" {
  count      = 10
  ami       = "ami-0c55b159cbfafa1f0"
  instance_type = "t2.micro"
}
```

Na pierwszy rzut oka oba podejścia wyglądają podobnie, a po ich zastosowaniu za pomocą Ansible lub Terraform otrzymujemy podobny efekt. Interesujące jest to, co się stanie, gdy zachodzi potrzeba wprowadzenia zmiany.

Przykładowo przyjmujemy założenie o zwiększeniu się poziomu ruchu sieciowego, więc zachodzi potrzeba zwiększenia liczby serwerów do 15. W przypadku Ansible utworzony wcześniej kod proceduralny nie jest dłużej użyteczny — jeżeli zmienisz liczbę serwerów na 15 i ponownie wykonasz ten kod, nastąpi wdrożenie 15 nowych (kolejnych) serwerów, co razem daje 25 serwerów. Dlatego też musisz dokładnie wiedzieć, co zostało wcześniej wdrożone, i na tej podstawie utworzyć zupełnie nowy skrypt proceduralny, który będzie odpowiadał za dodanie pięciu nowych serwerów.

```
- ec2:
  count: 5
  image: ami-0c55b159cbfafef1f0
  instance_type: t2.micro
```

Natomiast w stylu deklaratywnym, skoro określasz oczekiwany stan końcowy, a Terraform szuka sposobu na jego otrzymanie, to Terraform odpowiada za ustalenie, jak zapewnić otrzymanie oczekiwanego stanu końcowego. Jeżeli chcesz wdrożyć 5 kolejnych serwerów, musisz jedynie powrócić do tej samej konfiguracji Terraform i zmienić liczbę serwerów z obecnych 10 na oczekiwane 15.

```
resource "aws_instance" "example" {
  count      = 15
  ami       = "ami-0c55b159cbfafef1f0"
  instance_type = "t2.micro"
}
```

Po zastosowaniu tej konfiguracji Terraform ustala, że wcześniej zostało utworzonych 10 serwerów, więc teraz trzeba utworzyć jedynie 5 nowych. Przed zastosowaniem nowej konfiguracji można skorzystać z polecenia `terraform plan` Terraform i sprawdzić, jakie zmiany zostaną wprowadzone.

\$ terraform plan

```
# aws_instance.example[11] will be created
+ resource "aws_instance" "example" {
+   ami           = "ami-0c55b159cbfafef1f0"
+   instance_type = "t2.micro"
+   (...)
+ }

# aws_instance.example[12] will be created
+ resource "aws_instance" "example" {
+   ami           = "ami-0c55b159cbfafef1f0"
+   instance_type = "t2.micro"
+   (...)
+ }

# aws_instance.example[13] will be created
+ resource "aws_instance" "example" {
+   ami           = "ami-0c55b159cbfafef1f0"
+   instance_type = "t2.micro"
+   (...)
+ }

# aws_instance.example[14] will be created
+ resource "aws_instance" "example" {
+   ami           = "ami-0c55b159cbfafef1f0"
+   instance_type = "t2.micro"
+   (...)
+ }
```

Plan: 5 to add, 0 to change, 0 to destroy.

Co się stanie, gdy zajdzie potrzeba wdrożenia innej wersji aplikacji, np. obrazu AMI o identyfikatorze `ami-02bcbb802e03574ba`? W przypadku podejścia proceduralnego oba wcześniejsze szablony Ansible ponownie będą bezużyteczne, więc trzeba będzie przygotować kolejny szablon przeznaczony do wysłania 10 wdrożonych wcześniej serwerów (a może to było 15 serwerów?) i ostrożnie

uaktualnić każdy z nich. Natomiast w przypadku podejścia deklaratywnego wracasz do dokładnego tego samego pliku konfiguracyjnego i zmieniasz parametr `ami` na `ami-02bcbb802e03574ba`:

```
resource "aws_instance" "example" {
  count      = 15
  ami       = "ami-02bcbb802e03574ba"
  instance_type = "t2.micro"
}
```

Oczywiście omawiane tutaj przykłady są bardzo uproszczone. Ansible pozwala na używanie tagów podczas wyszukiwania istniejących egzemplarzy EC2 przed wdrożeniem nowych (np. za pomocą parametrów `instance_tags` i `count_tag`). Jednak konieczność samodzielnego zajęcia się tego rodzaju logiką dla każdego zasobu zarządzanego przez Ansible, na podstawie wcześniejszej historii zasobu, może być zaskakująco skomplikowana — istniejące egzemplarze trzeba wyszukiwać nie tylko po tagach, ale również po wersji obrazu. To pokazuje dwa poważne problemy związane z proceduralnymi narzędziami stosującymi podejście IaC:

Kod proceduralny nie pozwala na pełne przechwycenie stanu infrastruktury

W omawianym przykładzie zapoznanie się z trzema utworzonymi wcześniej szablonami Ansible jest niewystarczające do ustalenia, co zostało wdrożone. Trzeba również znać *kolejność* wykonywania tych skryptów. Jeżeli zastosujesz je w innej kolejności, możesz otrzymać odmienną infrastrukturę i to jest coś, co nie będzie odzwierciedlone przez bazę kodu. Innymi słowy, musisz znać pełną historię każdej wcześniej wprowadzonej zmiany.

Kod proceduralny ogranicza możliwość jego wielokrotnego używania

Możliwość wielokrotnego użycia kodu proceduralnego jest znacznie ograniczona ze względu na konieczność uwzględnienia aktualnego stanu infrastruktury. Skoro ten stan nieustannie się zmienia, kod utworzony tydzień temu może być nieużyteczny, ponieważ został opracowany do zmiany już nieistniejącego stanu infrastruktury. W efekcie proceduralne bazy kodu mają tendencję do rozrastania się i zwiększania poziomu swojego skomplikowania wraz z upływem czasu.

Dzięki deklaratywnemu podejściu Terraform kod zawsze przedstawia aktualny stan infrastruktury. Na podstawie kodu od razu można ustalić, co aktualnie jest wdrożone, jak zostało skonfigurowane, i nie trzeba się przy tym przejmować np. historią tych wdrożeń. To niezwykle ułatwia tworzenie kodu wielokrotnego użycia, ponieważ nie trzeba ręcznie zajmować się uwzględnieniem aktualnego stanu. Zamiast tego można się skoncentrować na opisaniużądanego stanu, a Terraform ustali, jak automatycznie przejść z jednego stanu do drugiego. W efekcie baza kodu Terraform zwykle pozostaje mała i łatwa do zrozumienia.

Oczywiście języki deklaratywne również mają swoje wady. Bez dostępu do pełnego języka programowania możliwości w zakresie wyrażania potrzeb są ograniczone. Przykładowo niektóre rodzaje zmian infrastruktury, takie jak wdrożenie bez przestoju, są trudne do wyrażenia w kategoriach czysto deklaratywnych (choć wcale nie niemożliwe, o czym przekonasz się podczas lektury rozdziału 5.). Podobnie ograniczone są możliwości w zakresie definiowania „logiki” (np. konstrukcje warunkowe typu `if`, pętle), tworzenie generycznego kodu wielokrotnego użycia może być trudne. Na szczęście Terraform oferuje pewną liczbę potężnych komponentów — takich jak zmienne danych wejściowych i wyjściowych, moduły, polecenia `create_before_destroy`, `count`, składnia trójargumentowa

i funkcje wbudowane — pozwalających na tworzenie przejrzystego, konfigurowalnego i modułowego kodu, nawet w języku deklaratywnym. Do tego tematu jeszcze wrócę w rozdziałach 4. i 5.

Serwer główny kontra jego brak

Domyślnie Chef, Puppet i SaltStack wymagają działania tzw. *serwera głównego* (ang. *master server*) przeznaczonego do przechowywania informacji o stanie infrastruktury i do przekazywania uaktualnień. Za każdym razem, gdy chcesz coś uaktualnić w infrastrukturze, używasz klienta (np. narzędzia działającego w powłocie) w celu wydania nowych poleceń do serwera głównego, który z kolei przekazuje uaktualnienia do wszystkich pozostałych serwerów — lub też te serwery regularnie pobierają uaktualnienia z serwera głównego.

Zastosowanie serwera głównego niesie wiele korzyści. Przede wszystkim to jest pojedyncze, centralne miejsce przeznaczone do analizy i zarządzania stanem infrastruktury. Wiele narzędzi zarządzania konfiguracją dostarcza nawet dla serwera głównego interfejs oparty na przeglądarce WWW (przykładami są tutaj Chef Console i Puppet Enterprise Console), ułatwiający sprawdzenie tego, co się dzieje we wdrożeniu. Ponadto część serwerów głównych nieustannie działa w tle i wymusza stosowanie danej konfiguracji. Dzięki temu, jeśli w serwerze zostanie ręcznie wprowadzona zmiana, serwer główny może ją wycofać i tym samym pomaga w uniknięciu wprowadzania zmian w konfiguracji.

Jednak wykorzystanie serwera głównego ma pewne poważne wady:

Dodatkowa infrastruktura

Konieczność wdrożenia dodatkowego serwera lub nawet klastra takich serwerów (w celu zapewnienia wysokiej dostępności i skalowalności), aby móc uruchomić serwer główny.

Konieczność obsługi

Trzeba pamiętać o obsłudze, uaktualnianiu, tworzeniu kopii zapasowej, monitorowaniu i skalowaniu serwera głównego.

Bezpieczeństwo

Konieczne jest zapewnienie klientowi możliwości komunikacji z serwerem głównym, który z kolei musi mieć możliwości komunikowania się z pozostałymi serwerami. To najczęściej oznacza otworenie dodatkowych portów i konfigurację dodatkowych systemów uwierzytelniania, co znowu zwiększa obszar dla potencjalnych ataków.

Chef, Puppet i SaltStack w różnym stopniu zapewniają obsługę węzłów pozbawionych serwera głównego, gdy oprogramowanie agenta wymienionych narzędzi działa w każdym serwerze, zwykle uruchamiane w ramach pewnego harmonogramu (np. zadanie cron wykonywane co pięć minut) używanego do pobierania najnowszych uaktualnień z systemu kontroli wersji (zamiast z serwera głównego). To znacznie zmniejsza liczbę elementów ruchomych, choć, jak dowiesz się z dalszej części rozdziału, jednocześnie zwiększa liczbę pytań pozostawionych bez odpowiedzi, zwłaszcza w zakresie przygotowywania serwerów i instalowania w nich oprogramowania agenta.

Ansible, CloudFormation, OpenStack Heat i Terraform domyślnie nie używają serwera głównego. Z technicznego punktu widzenia mogą opierać działanie na serwerze głównym, ale jest on częścią używanej infrastruktury, a nie oddzielnym serwerem wymagającym zarządzania. Przykładowo Terra-

form komunikuje się z dostawcami chmury za pomocą API dostawców chmury, więc w pewnym sensie te serwery API są serwerami głównymi, z wyjątkiem tego, że nie wymagają dodatkowej infrastruktury i mechanizmów uwierzytelniania (np. można wykorzystać własne klucze SSH). Ansible działa poprzez nawiązanie bezpośredniego połączenia z każdym serwerem poprzez SSH, więc nie wymaga żadnej dodatkowej infrastruktury lub mechanizmów uwierzytelniania (można wykorzystać własne klucze SSH).

Agent kontra jego brak

Chef, Puppet i SaltStack wymagają zainstalowania *oprogramowania agenta* (np. Chef Client, Puppet Agent, Salt Minion) w każdym serwerze, który ma zostać skonfigurowany. Agent zwykle działa w tle w każdym serwerze i jest odpowiedzialny za instalację najnowszych uaktualnień zarządzania konfiguracją.

Takie rozwiązanie również ma pewne wady:

Bootstrapping

W jaki sposób przygotować serwery i zainstalować w nich oprogramowanie agenta? Pewne narzędzia zarządzania konfiguracją mogą pomóc przy założeniu, że procesy dodatkowe zajmą się tym dla wspomnianych narzędzi (np. najpierw użyjesz Terraform do wdrożenia serwerów wraz z obrazem AMI zawierającym zainstalowanego agenta). Z kolei inne narzędzia konfiguracji mają specjalne procesy wymagające jednorazowego wydania poleceń w celu przygotowania serwerów z wykorzystaniem API dostawcy chmury i poprzez SSH zainstalowania w tych serwerach oprogramowania agenta.

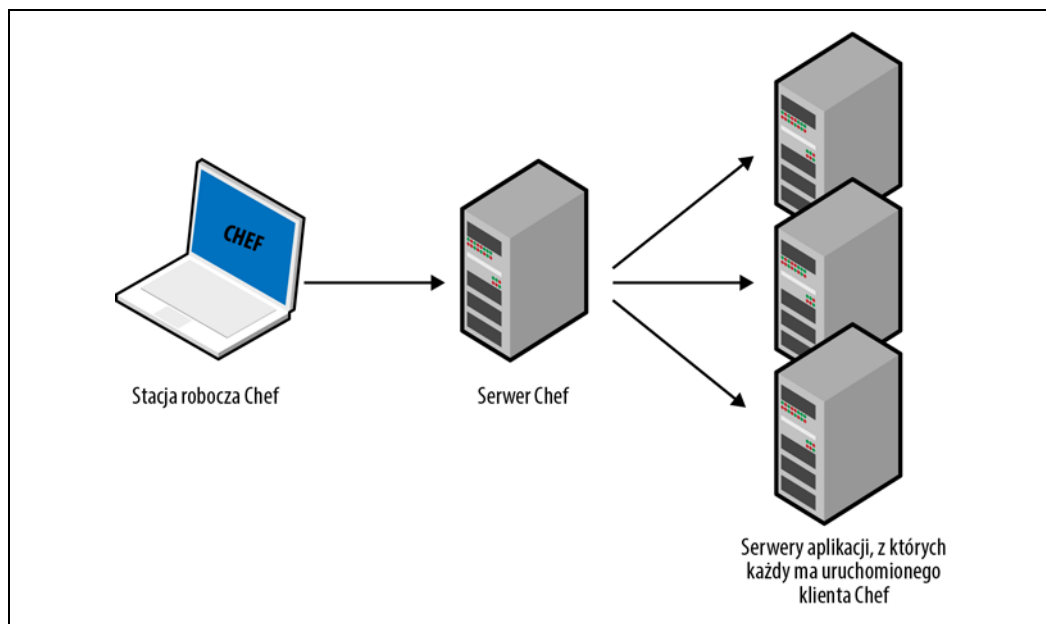
Obsługa

Oprogramowanie agenta trzeba regularnie i ostrożnie uaktualniać i zwracać uwagę na zachowanie zgodności z serwerem głównym, o ile taki jest stosowany. Ponadto konieczne jest monitorowanie oprogramowania agenta i jego ponowne uruchamianie, jeśli ulegnie awarii.

Bezpieczeństwo

Jeżeli oprogramowanie agenta pobiera konfigurację z serwera głównego (lub innego serwera w przypadku nieużywania serwera głównego), konieczne jest otworenie portów dla ruchu wychodzącego w każdym serwerze. Jeśli natomiast serwer główny przekazuje konfigurację do agenta, w każdym serwerze konieczne jest otworenie portów dla ruchu przychodzącego. W obu przypadkach należy określić sposób uwierzytelnienia agenta w serwerze, z którym prowadzi komunikację, co z kolei zwiększa obszar dla potencjalnych ataków.

Także i w tym zakresie Chef, Puppet i SaltStack różnią się poziomem obsługi dla trybów pracy bez agenta (np. salt-ssh), ale należy mieć świadomość, że taki tryb nie zapewnia dostępu do pełnych możliwości narzędzia zarządzania konfiguracją. Dlatego też w rzeczywistych wdrożeniach domyślna konfiguracja dla Chef, Puppet i SaltStack niemal zawsze zawiera agenta i najczęściej również serwer główny, jak pokazałem na rysunku 1.7.



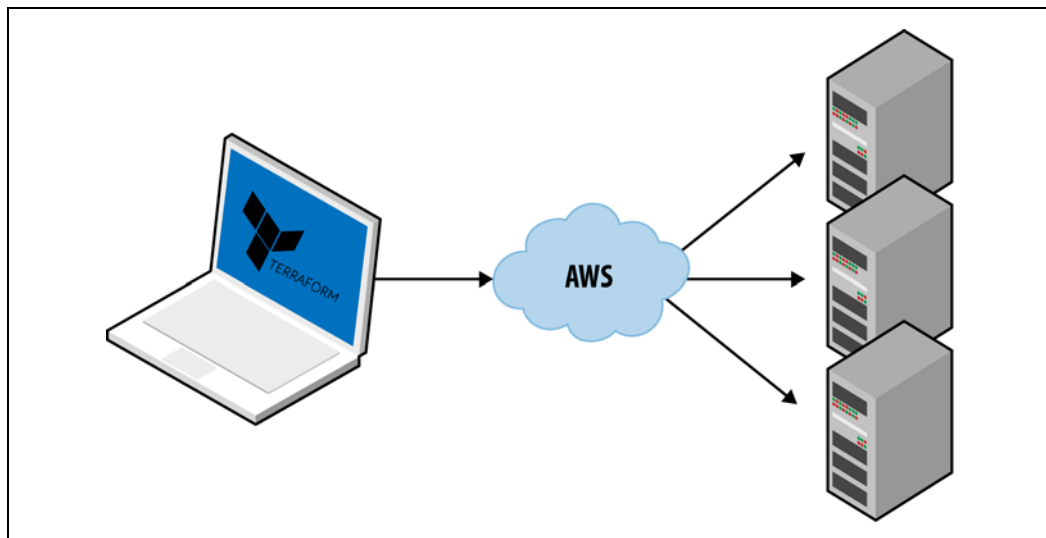
Rysunek 1.7. Typowa architektura Chef, Puppet i SaltStack opiera się na wielu ruchomych elementach. Przykładowo domyślna konfiguracja Chef oznacza uruchomienie w komputerze klienta Chef komunikującego się z serwerem głównym, który z kolei wdraża zmiany przez komunikowanie się z klientami Chef uruchomionymi we wszystkich pozostałych serwerach

Wszystkie te ruchome elementy wprowadzają do infrastruktury ogromną liczbę nowych trybów awarii. Za każdym razem, gdy o trzeciej nad ranem otrzymasz zgłoszenie błędu, musisz ustalić, czy to jest wynik błędu w kodzie aplikacji, czy w kodzie IaC, czy w kliencie zarządzania konfiguracją, czy w serwerze głównym, czy w trakcie komunikacji z serwerem głównym, czy w trakcie komunikacji innych serwerów z serwerem głównym, czy...

Ansible, CloudFormation, OpenStack Heat i Terraform nie wymagają instalacji żadnych dodatkowych agentów. A dokładnie to część z nich wymaga agentów, przy czym to oprogramowanie jest zwykle instalowane jako część używanej infrastruktury. Przykładowo AWS, Azure, Google Cloud i inni dostawcy chmury biorą na siebie instalację, zarządzanie i uwierzytelnianie oprogramowania agenta w każdym fizycznym serwerze. Jako użytkownik Terraform nie musisz się tym zajmować: wydajesz polecenia, a agent dostawcy chmury wykonuje je we wszystkich Twoich serwerach, jak możesz zobaczyć na rysunku 1.8. W przypadku Ansible konieczne jest uruchomienie demona SSH, który i tak działa w większości serwerów.

Duża społeczność kontra mała

Niezależnie od wybranej technologii wybierasz także społeczność. W wielu przypadkach ekosystem zbudowany wokół projektu może mieć ogromny wpływ na ocenę danej technologii, nawet większy niż jakość samej technologii. Społeczność określa liczbę osób pracujących nad projektem, liczbę dostępnych



Rysunek 1.8. Terraform wykorzystuje architekturę opartą na agencie i pozbawioną serwera głównego. Potrzebujesz jedynie klienta Terraform, który zajmie się resztą, wykorzystując do tego API dostawcy chmury takiego jak AWS

wtyczek, możliwość integracji z rozwiązaniami oraz dostępność rozszerzeń, pomocy technicznej (np. blogi, pytania zadane w serwisach takich jak StackOverflow) i łatwość zatrudnienia osoby, która będzie mogła pomóc w rozwiązaniu problemu (np. pracownika, konsultanta, komercyjnej pomocy technicznej).

Bardzo trudno jest przeprowadzić dokładne porównanie społeczności, choć w internecie można znaleźć informacje o pewnych trendach. W tabeli 1.1 wymieniałem popularne i stosujące praktyki IaC narzędzia wraz z danymi, które zebrałem w maju 2019 roku. Te dane to m.in. rodzaj projektu (typu open source lub zamknięty kod źródłowy), obsługiwani dostawcy chmury, całkowita liczba osób pracujących nad projektem i gwiazdek zebranych przez projekt w serwisie GitHub, liczba operacji przekazania do repozytorium w ciągu ostatnich 30 dni, liczba aktywnych zgłoszeń w ciągu ostatnich 30 dni w okresie od połowy kwietnia do połowy maja, liczba bibliotek typu open source dostępnych dla narzędzia, liczba dotyczących danego narzędzia pytań zadanych w serwisie StackOverflow oraz liczba zamieszczonych w serwisie <https://pl.indeed.com/> ofert pracy, w których treści został wspomniany dany projekt⁸.

⁸ Większość tych danych — m.in. liczba osób pracujących nad projektem, gwiazdek, zmian i zgłoszeń — pochodzi z repozytoriów typu open source i narzędzi zgłaszania błędów (przede wszystkim GitHub) dla danego projektu. Ponieważ CloudFormation to rozwiązanie oparte na zamkniętym kodzie źródłowym, część tych informacji jest niedostępna.

Tabela 1.1. Porównanie społeczności wybranych narzędzi IaC

	Kod źródłowy	Chmura	Liczba pracujących nad projektem	Liczba gwiazdek	Liczba zatwierdzeń (ostatnie 30 dni)	Liczba błędów	Liczba bibliotek	Liczba pytań w serwisie Stack Overflow	Liczba ofert pracy
Chief	otwarty	wszystkie	562	5794	435	86	3832 ^a	5982	4378 ^b
Puppet	otwarty	wszystkie	515	5299	94	314 ^c	6110 ^d	3585	4200 ^e
Ansible	otwarty	wszystkie	4386	37 161	506	523	20 677 ^f	11 746	8787
SaltStack	otwarty	wszystkie	2237	9901	608	441	318 ^g	1062	1622
Cloud Formation	zamknięty	AWS	?	?	?	?	377 ^h	3315	2318
Heat	otwarty	wszystkie	361	349	12	600 ⁱ	0 ^j	88	2201 ^k
Terraform	otwarty	wszystkie	1261	16 827	173	204	1462 ^l	2730	3641

^a Liczba receptur dostępnych w Chef Supermarket (<https://supermarket.chef.io/cookbooks>).

^b Aby uniknąć błędnych wyników dla wyrażenia *chef*, przeprowadziłem wyszukiwanie dla *chef devops*.

^c Wartość na podstawie konta Puppet Labs JIRA (<https://tickets.puppetlabs.com/secure/Dashboard.jspa>).

^d Liczba modułów w Puppet Forge (<https://forge.puppet.com/>).

^e Aby uniknąć błędnych wyników dla wyrażenia *puppet*, przeprowadziłem wyszukiwanie dla *puppet devops*.

^f Liczba wielokrotnego użycia ról w Ansible Galaxy (<https://galaxy.ansible.com/>).

^g Liczba wzorów udostępnionych w koncie GitHub Salt Stack Formulas (<https://github.com/saltstack-formulas>).

^h Liczba szablonów udostępnionych w koncie GitHub awslabs (<https://github.com/awslabs>).

ⁱ Wartość na podstawie narzędzia zgłaszania błędów OpenStack (<https://bugs.launchpad.net/openstack>).

^j Nie byłem w stanie znaleźć żadnych kolekcji szablonów OpenStack Heat opracowanych przez społeczność.

^k Aby uniknąć błędnych wyników dla wyrażenia *heat*, przeprowadziłem wyszukiwanie dla *openstack*.

^l Liczba modułów w repozytorium Terraform Registry (<https://registry.terraform.io/>).

Oczywiście to nie jest doskonałe porównanie typu jeden do jednego. Przykładowo dla części narzędzi istnieje więcej niż tylko jedno repozytorium, a część korzysta z innych metod zgłaszania błędów i sugestii. Wyszukiwanie ofert pracy w języku angielskim, zawierających słowa *chef* i *puppet*, nie należy do łatwych zadań. Dla Terraform od 2017 roku istnieją oddzielne repozytoria dla kodu dostawców, więc pomiar aktywności na bazie jedynie repozytorium podstawowego daje znacznie zaniżony wynik (mniej więcej 10-krotnie) itd.

Mając to na względzie, warto zwrócić uwagę na kilka oczywistych trendów. Po pierwsze, poza CloudFormation wszystkie wymienione w tabeli narzędzia stosujące praktyki IaC są dostępne jako oprogramowanie typu open source i działają z wieloma dostawcami chmury. Po drugie, Ansible prowadzi w kategorii popularności, przy czym Salt i Terraform znajdują się tuż za nim.

Innym interesującym trendem, na który należy zwrócić uwagę, jest zmiana wartości wymienionych w tabeli względem tych, które przedstawiłem w pierwszym wydaniu książki. W tabeli 1.2 zaprezentowałem wyrażoną w procentach zmianę każdej wartości wobec tych, które zostały zebrane we wrześniu 2016 roku.

Tabela 1.2. Zmiana wartości dotyczących społeczności wybranych narzędzi IaC dla danych zebranych między wrześniem 2016 roku i majem 2019 roku

	Kod źródłowy	Chmura	Liczba pracujących nad projektem	Liczba gwiazdek	Liczba zatwierdzeń (ostatnie 30 dni)	Liczba błędów	Liczba bibliotek	Liczba pytań w serwisie Stack Overflow	Liczba ofert pracy
Chief	otwarty	wszystkie	+18%	+31%	+139%	+48%	+26%	+43%	-22%
Puppet	otwarty	wszystkie	+19%	+27%	+19%	+42%	+38%	+36%	-19%
Ansible	otwarty	wszystkie	+195%	+97%	+49%	+66%	+157%	+223%	+125%
SaltStack	otwarty	wszystkie	+40%	+44%	+79%	+27%	+33%	+73%	+257%
Cloud Formation	zamknięty	AWS	?	?	?	?	+57%	+441%	+249%
Heat	otwarty	wszystkie	+28%	+23%	-85%	+1566%	0	+69%	+2 957%
Terraform	otwarty	wszystkie	+93%	+194%	-61%	-58%	+3555%	+1984%	+8 288%

Dane zamieszczone w tabeli 1.2 również nie są doskonałe, ale wystarczające do tego, aby dostrzec trend: Terraform i Ansible zyskują ogromną popularność. Wzrost liczby osób pracujących nad tymi projektami, otrzymanych gwiazdek, istniejących dla nich bibliotek typu open source, pytań zadanych w serwisie StackOverflow oraz ofert pracy jest po prostu oszałamiający⁹. Oba wymienione narzędzia mogą się pochwalić ogromnymi, aktywnymi społecznościami i na podstawie tego trendu można przyjąć założenie, że w przyszłości staną się one jeszcze większe.

Rozwiązanie dojrzałe kontra najnowsze

Kolejnym kluczowym czynnikiem przy wyborze technologii jest jej dojrzałość.

W tabeli 1.3 wymienię daty wydania i obecne numery wersji poszczególnych narzędzi stosujących praktyki IaC, analizowanych w tym podrozdziale (stan na maj 2019 roku).

To również nie jest dokładne porównanie, ponieważ poszczególne narzędzia stosują odmienne schematy wersjonowania. Mimo to trendy powinny być wyraźnie widoczne. Jak dotąd Terraform jest najmłodszym narzędziem IaC w tym porównaniu. Ponieważ nadal znajduje się w wersji wcześniejszej niż 1.0.0, nie ma gwarancji stabilności i wstecznej zgodności API, a błędy pojawiają się dość często (na szczęście większość z nich to drobiazgi). To jest największa wada Terraform: wprowadzie w krótkim okresie stało się niezwykle popularne, ale ceną, którą trzeba zapłacić za korzystanie z nowego narzędzia, jest to, że nie zostało jeszcze tak dopracowane jak inne opcje IaC.

⁹ Spadek liczby zatwierdzeń w systemie kontroli wersji i liczby zgłoszeń wynika wyłącznie z tego, że podana wartość dotyczy jedynie podstawowego repozytorium Terraform. W 2017 roku kod przeznaczony do obsługi dostawców został przeniesiony do oddzielnych repozytoriów, więc ogromna aktywność w tych ponad 100 repozytoriach nie została uwzględniona w tabeli.

Tabela 1.3. Porównanie dojrzałości wybranych narzędzi IaC w maju 2019 roku

	Pierwsze wydanie	Obecne wydanie
Puppet	2005	6.0.9
Chef	2009	12.19
CloudFormation	2011	???
SaltStack	2011	2019.2.0
Ansible	2012	2.5.5
Heat	2012	12.0.0
Terraform	2014	0.12.0

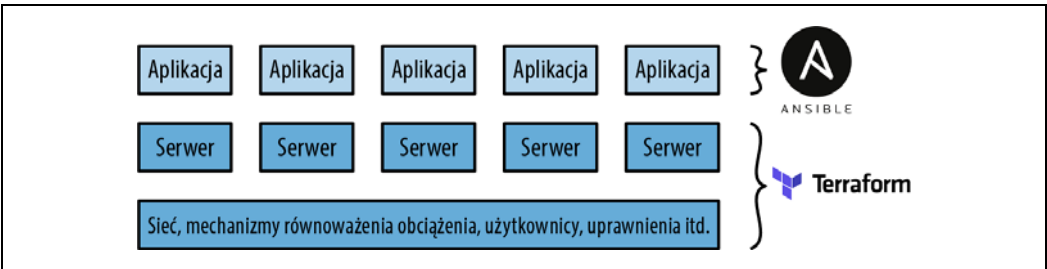
Używanie razem wielu narzędzi

Wprawdzie w rozdziale porównywałem narzędzia IaC, ale rzeczywistość jest taka, że prawdopodobnie będziesz korzystać z wielu narzędzi podczas tworzenia infrastruktury. Każde z nich ma zalety i wady, więc do Ciebie należy wybór odpowiedniego narzędzia do wykonania konkretnego zadania.

W tej sekcji przedstawiam trzy często spotykane połączenia, z którymi zetknąłem się podczas pracy dla różnych firm.

Provisioning plus zarządzanie konfiguracją

Przykład: Terraform i Ansible. Terraform wykorzystujesz do wdrażania całej infrastruktury łącznie z topologią sieci (wirtualna prywatna chmura (ang. *virtual private cloud*, VPC), podmaski, tabele routingu), magazynami danych (np. MySQL, Redis), mechanizmami równoważenia obciążenia oraz serwerami. Następnie używasz Ansible do wdrożenia aplikacji w tych serwerach, jak pokazałem na rysunku 1.9.

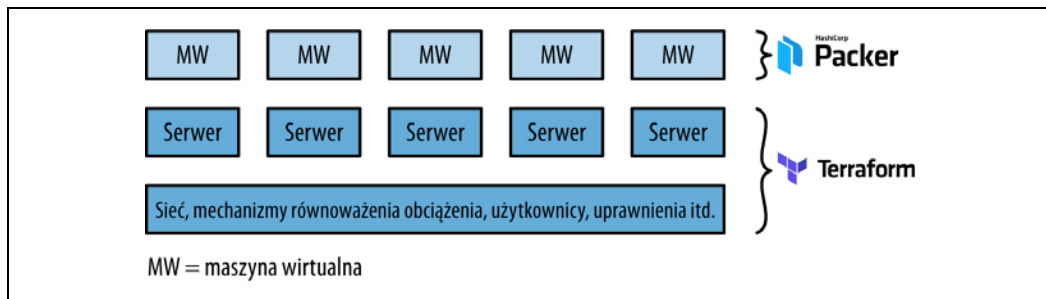


Rysunek 1.9. Używanie razem narzędzi Terraform i Ansible

To jest łatwe rozwiązanie na początek, ponieważ nie wymaga dodatkowej infrastruktury (Terraform i Ansible to aplikacje działające jedynie po stronie klienta) i istnieje wiele sposobów na zapewnienie współpracy między Ansible i Terraform (np. Terraform dodaje specjalne znaczniki do serwerów, które z kolei Ansible wykorzystuje do odnalezienia danego serwera i jego skonfigurowania). Wadą tego połączenia są tworzenie dużej ilości kodu proceduralnego i modyfikowalne serwery, więc wraz ze wzrostem bazy kodu, infrastruktury i zespołu obsługa całości staje się coraz trudniejsza.

Provisioning plus szablony serwerów

Przykład: Terraform i Packer. Narzędzia Packer używasz do przygotowania aplikacji w postaci obrazu maszyny wirtualnej. Następnie wykorzystujesz Terraform do wdrożenia (a) serwerów za pomocą wspomnianych obrazów maszyn wirtualnych i (b) pozostałej części infrastruktury, łącznie z topologią sieci (VPC, podmaski, tabele routingu), magazynami danych (np. MySQL, Redis) oraz mechanizmami równoważenia obciążenia, jak pokazałem na rysunku 1.10.



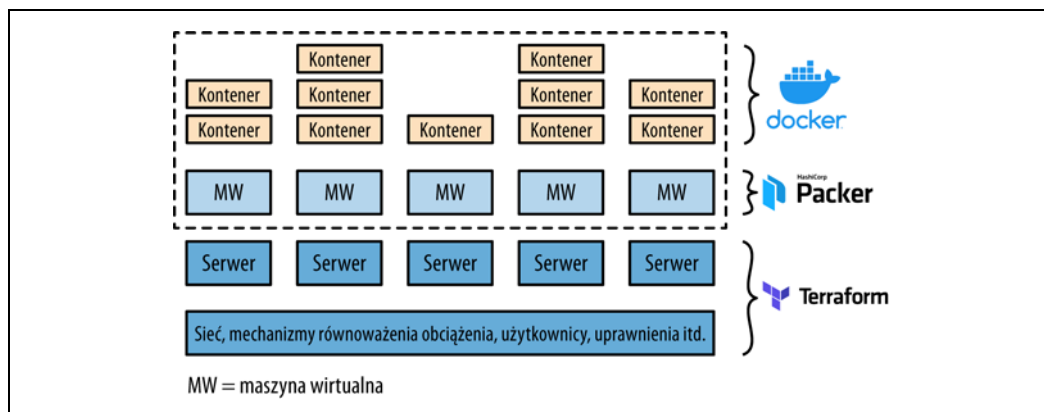
Rysunek 1.10. Używanie razem narzędzi Terraform i Packer

To również jest łatwe rozwiązanie na początek, ponieważ nie wymaga dodatkowej infrastruktury (Terraform i Packer to aplikacje działające jedynie po stronie klienta) i w dalszej części książki nabędziesz dużej wprawy we wdrażaniu obrazów maszyn wirtualnych za pomocą Terraform. Co więcej, to jest podejście infrastruktury niemodyfikowalnej, co znacznie ułatwia późniejszą obsługę. Jednak i to połączenie ma pewne wady. Pierwsza polega na tym, że przygotowanie i wdrożenie maszyny wirtualnej może trwać bardzo długo, co zmniejsza liczbę przeprowadzanych wdrożeń. Druga, jak się dowiesz z dalszych rozdziałów książki, jest taka, że strategie wdrażania możliwe do implementacji za pomocą Terraform są ograniczone (nie możesz natywnie zastosować wdrożenia typu niebieski-zielony). Dlatego skutkiem będzie tworzenie ogromnej liczby skomplikowanych skryptów wdrożenia lub zwrócenie się ku narzędziom instrumentacji, co przedstawię w następnym punkcie.

Provisioning plus szablony serwerów plus instrumentacja

Przykład: Terraform, Packer, Docker i Kubernetes. Narzędzia Packer używasz do przygotowania obrazu maszyny wirtualnej zawierającej zainstalowane narzędzia Docker i Kubernetes. Następnie wykorzystujesz Terraform do wdrożenia (a) klastra serwerów, z których każdy będzie uruchamiał wspomniany obraz maszyny wirtualnej, i (b) pozostałej części infrastruktury, łącznie z topologią sieci (VPC, podmaski, tabele routingu), magazynami danych (np. MySQL, Redis) oraz mechanizmami równoważenia obciążenia. Na końcu, po uruchomieniu klastra serwerów, nastąpi przygotowanie klastra Kubernetes, który będzie można wykorzystać do uruchamiania i zarządzania aplikacjami Dockera, jak pokazałem na rysunku 1.11.

Zaletą takiego podejścia jest to, że obrazy Dockera są tworzone dość szybko, można je uruchamiać i testować w komputerze lokalnym oraz wykorzystać wszystkie zalety wbudowanej funkcjonalności Kubernetes, m.in. stosowanie różnych strategii wdrażania, automatyczną naprawę, automatyczne skalowanie itd. Wadą tego połączenia jest większy poziom skomplikowania, zarówno w kategoriach



Rysunek 1.11. Używanie razem narzędzi Terraform, Packer, Docker i Kubernetes

dotychczasowej infrastruktury przeznaczonej do uruchomienia rozwiązania (klastry Kubernetes są kosztowne i trudne do wdrożenia i działania, choć większość najważniejszych dostawców chmury oferuje teraz zarządzane usługi Kubernetes, co może odciążyć Cię od pewnych zadań), jak i w kategoriach poznawania, zarządzania i debugowania rozwiązania.

Podsumowanie

Po połączeniu wszystkiego w całość w tabeli 1.4 wymieniałem najpopularniejsze narzędzia stosujące praktyki IaC. Zwróć uwagę, że ta tabela pokazuje *domyślny* lub *najczęściej stosowany* sposób, w jaki są używane te narzędzia IaC. Jak wspomniałem we wcześniejszej części rozdziału, te narzędzia IaC są na tyle elastyczne, że mogą być używane także w innych konfiguracjach (np. Chef bez serwera głównego, Salt do przygotowania infrastruktury niemodyfikowalnej itd.).

W firmie Gruntwork chcieliśmy zastosować rozwiązanie typu open source, niezależne od chmury narzędzie provisioningu zapewniające obsługę infrastruktury niemodyfikowalnej, język deklaratywny, architekturę pozbawioną serwera głównego i agenta, a także charakteryzującą się dużą społecznością i dojrzałą bazą kodu. Z tabeli 1.4 wynika, że choć Terraform nie jest perfekcyjnym rozwiązaniem, to najlepiej spełnia postawione przez nas wymagania.

Czy Terraform spełnia również Twoje kryteria? Jeśli tak, przejdź do rozdziału 2., z którego dowiesz się, jak można korzystać z Terraform.

Tabela 1.4. Porównanie najczęściej stosowanego sposobu użycia popularnych narzędzi IaC

	Kod źródłowy	Chmura	Typ	Infrastruktura	Język	Agent	Server główny	Spoleczność	Dojrzałość
Chef	otwarty	wszystkie	konfiguracja zarządzania	zmienna	proceduralny	tak	tak	duża	wysoka
Puppet	otwarty	wszystkie	konfiguracja zarządzania	zmienna	deklaratywny	tak	tak	duża	wysoka
Ansible	otwarty	wszystkie	konfiguracja zarządzania	zmienna	proceduralny	nie	nie	olbrzymia	średnia
SaltStack	otwarty	wszystkie	konfiguracja zarządzania	zmienna	deklaratywny	tak	tak	duża	średnia
CloudFormation	zamknięty	AWS	provisioning	niezmienna	deklaratywny	nie	nie	mała	średnia
Heat	otwarty	wszystkie	provisioning	niezmienna	deklaratywny	nie	nie	mała	niska
Terraform	otwarty	wszystkie	provisioning	niezmienna	deklaratywny	nie	nie	olbrzymia	niska

Rozpoczęcie pracy z Terraform

W rozdziale przedstawiam podstawy dotyczące pracy z Terraform. Praca z tym narzędziem jest bardzo łatwa, więc na przestrzeni kilkudziesięciu stron rozdziału przejdziesz od wydania pierwszych prostych poleceń Terraform do poleceń umożliwiających wdrożenie klastra serwerów wraz z mechanizmem równoważenia obciążenia, który będzie pozwalał na rozkład ruchu sieciowego między poszczególne serwery. Taka infrastruktura to dobry punkt wyjścia dla uruchamiania skalowalnych i charakteryzujących się wysoką dostępnością usług sieciowych. W kolejnych rozdziałach ten przykład zostanie jeszcze bardziej rozbudowany.

Terraform ma możliwość provisioningu infrastruktury w publicznych dostawcach chmury, takich jak Amazon Web Services (AWS), Azure, Google Cloud i DigitalOcean, a także w prywatnych chmurach i platformach, takich jak OpenStack i VMware. W praktycznie wszystkich przykładowych fragmentach kodu w tym rozdziale oraz w pozostałej części książki będziesz korzystać z AWS. Z wymienionych tutaj powodów AWS to dobry wybór podczas poznawania Terraform:

- Jak dotychczas AWS to najpopularniejszy dostawca infrastruktury chmury. Jego udział w rynku infrastruktury chmury wynosi około 45%, czyli znacznie więcej niż trzech kolejnych konkurentów (Microsoft, Google i IBM) razem wziętych (<https://www.geekwire.com/2016/study-aws-45-share-public-cloud-infrastructure-market-microsoft-google-ibm-combined/>).
- AWS oferuje szeroką gamę niezawodnych i skalowalnych usług hostingu w chmurze, m.in. Amazon Elastic Compute Cloud (Amazon EC2), którą można wykorzystać do wdrażania serwerów wirtualnych, automatycznie skalowaną grupę (ang. *auto scaling group*, ASG) ułatwiającą zarządzanie klastrem serwerów wirtualnych i mechanizm równoważenia obciążenia (ang. *elastic load balancer*, ELB) pozwalający na rozkładanie ruchu sieciowego między klastrami serwerów wirtualnych¹.
- AWS oferuje hojnie wyposażone (<https://aws.amazon.com/free/>) konto (bezpłatne przez pierwszy rok), które powinno być wystarczające do wypróbowania wszystkich przykładów przedstawionych w książce. Jeżeli już wykorzystałeś konto próbne, to przykłady z książki nie powinny kosztować Cię więcej niż kilkanaście złotych.

¹ Jeżeli terminologia stosowana przez AWS jest dezorientująca, zapoznaj się z jej wyjaśnieniem opublikowanym na stronie <https://expeditedsecurity.com/aws-in-plain-english/>.

Jeżeli nigdy wcześniej nie korzystałeś z AWS lub Terraform, nie przejmuj się, ponieważ ten przewodnik jest przeznaczony dla początkujących użytkowników obu technologii. Omówię tutaj następujące etapy:

- utworzenie konta AWS,
- instalacja Terraform,
- wdrożenie pojedynczego serwera,
- wdrożenie pojedynczego serwera WWW,
- wdrożenie konfigurowalnego serwera WWW,
- wdrożenie klastra serwerów WWW,
- wdrożenie mechanizmu równoważenia obciążenia,
- uporządkowanie po wdrożeniach.



Przykładowe fragmenty kodu

Przypominam, że wszystkie przykładowe fragmenty kodu znajdują się w repozytorium GitHub pod adresem <https://github.com/brikis98/terraform-up-and-running-code>.

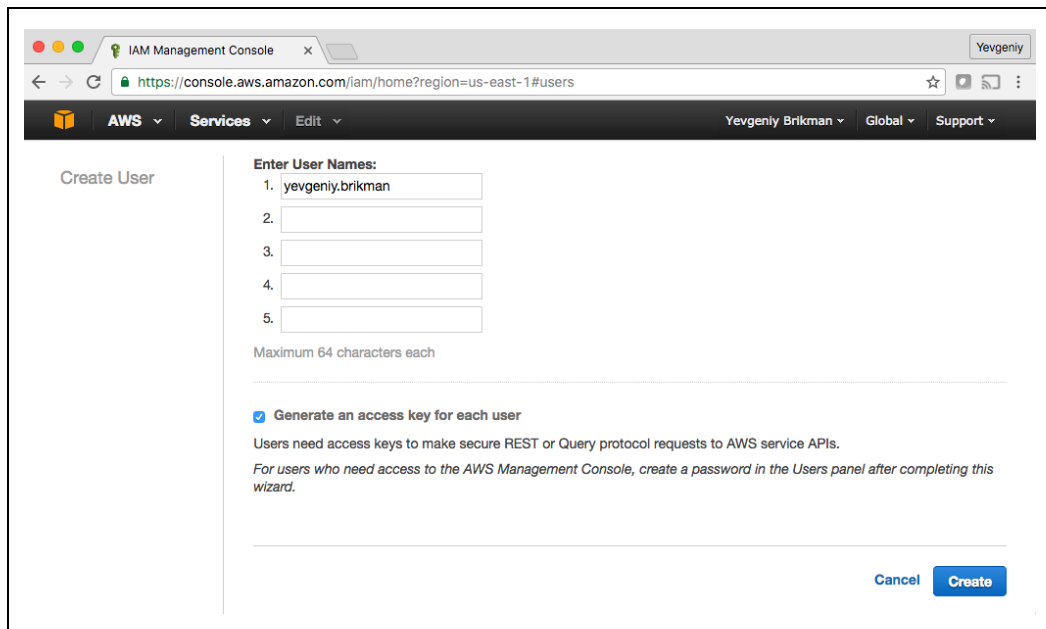
Utworzenie konta AWS

Jeżeli jeszcze nie masz konta AWS, przejdź na stronę <https://aws.amazon.com/> i je utwórz. Podczas rejestracji konta w AWS początkowo będziesz zalogowany jako *użytkownik root*. Ten użytkownik ma pełnię uprawnień i może zrobić absolutnie wszystko z kontem AWS. Dlatego też z perspektywy zapewnienia bezpieczeństwa używanie tego konta do wykonywania codziennych zadań jest niewskazane. Konto użytkownika *root* powinieneś wykorzystać tylko do tworzenia innych kont użytkowników o znacznie bardziej ograniczonych uprawnieniach, a następnie natychmiast przejść do jednego z tych kont².

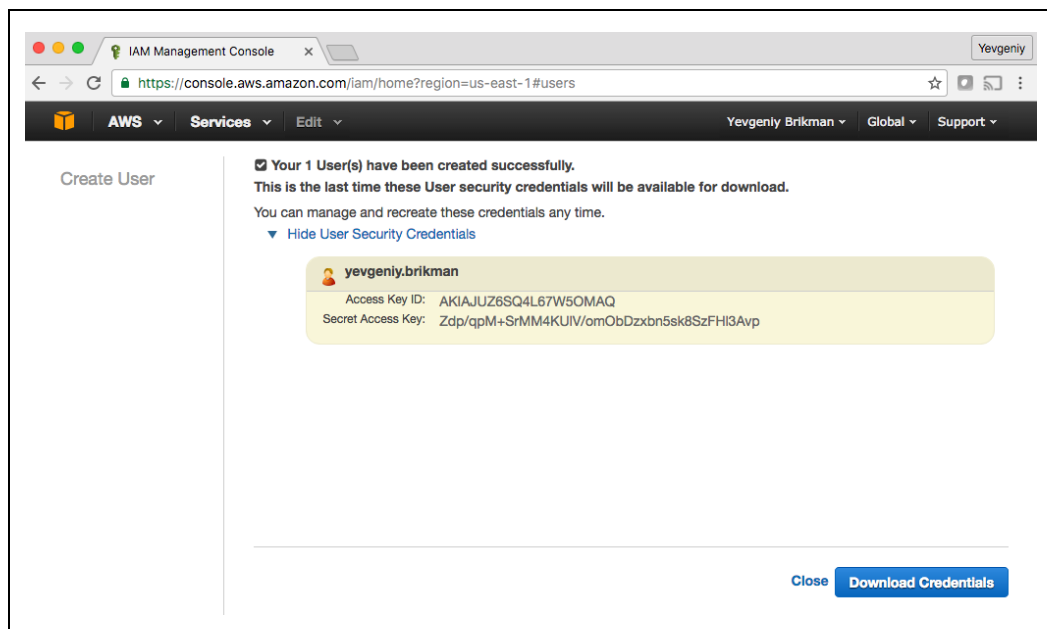
Aby utworzyć znacznie bardziej ograniczone konto użytkownika, będziesz musiał skorzystać z usługi *Identity and Access Management* (IAM). Pozwala ona na zarządzanie kontami użytkowników i ich uprawnieniami. Aby utworzyć nowego *użytkownika IAM*, przejdź do konsoli IAM (<https://console.aws.amazon.com/iam/home>), kliknij *Users*, a następnie kliknij przycisk *Create New Users*. Podaj nazwę użytkownika i upewnij się, że zaznaczona jest opcja *Generate an access key for each user*, jak pokazałem na rysunku 2.1 (warto pamiętać, że AWS może wprowadzać zmiany w projekcie konsoli, więc strona IAM może wyglądać już inaczej w chwili, gdy czytasz te słowa).

Kliknij przycisk *Create*. AWS wyświetli dane uwierzytelniające dla tego użytkownika składające się z wartości *Access Key ID* i *Secret Access Key*, jak możesz zobaczyć na rysunku 2.2. Musisz je natychmiast zapisać, ponieważ nigdy więcej nie zostaną wyświetlone, a będą potrzebne w dalszej części rozdziału. Nie zapominaj, że te dane uwierzytelniające zapewniają dostęp do konta AWS,

² Więcej szczegółowych informacji na temat najlepszych praktyk w zakresie zarządzania kontami użytkowników w AWS znajdziesz na stronie <https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html>.



Rysunek 2.1. Tworzenie nowego użytkownika za pomocą konsoli IAM



Rysunek 2.2. Dane uwierzytelniające AWS powinieneś przechowywać w bezpiecznym miejscu. Nigdy nikomu ich nie udostępniaj (nie przejmuj się, te pokazane na rysunku zostały spreparowane)

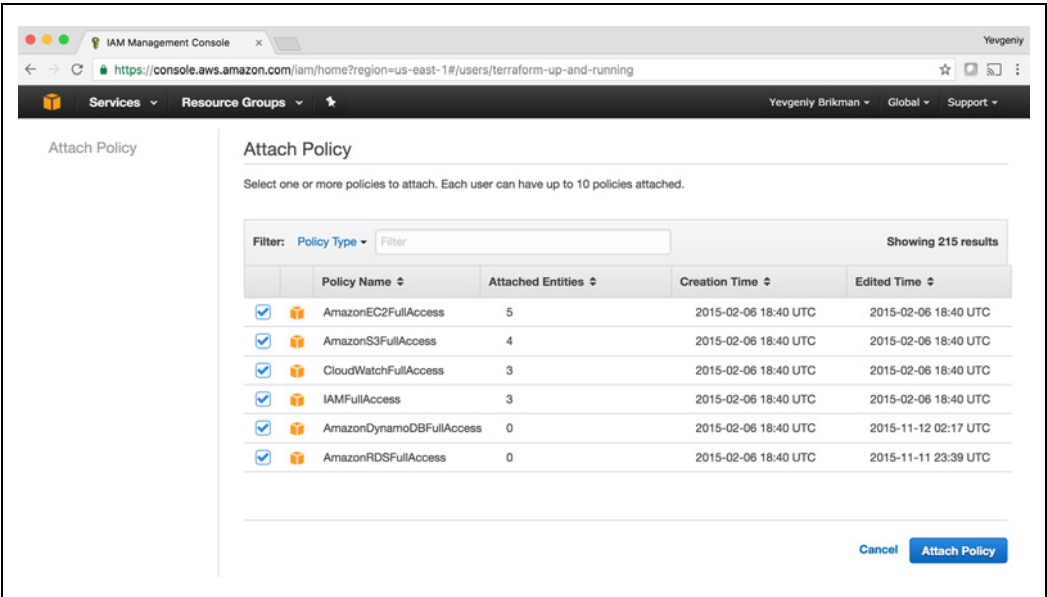
więc przechowuj je bezpiecznie (np. w menedżerze haseł, takim jak 1Password, LastPass, lub w tzw. pęku kluczy systemu macOS) i nigdy nikomu nie udostępniaj.

Po zapisaniu danych uwierzytelniających możesz kliknąć przycisk *Close*. To spowoduje przejście na listę użytkowników IAM. Kliknij utworzone przed chwilą konto użytkownika i przejdź na kartę *Permissions*. Domyślnie nowe konto użytkownika nie ma żadnych uprawnień i tym samym nie pozwala na zrobienie cokolwiek.

Aby zdefiniować uprawnienia użytkownikowi IAM do wykonania jakiegokolwiek zadania, konto trzeba powiązać ze zdefiniowaną polityką IAM. Ta *polityka IAM* to dokument JSON definiujący, co użytkownik może, a czego nie może zrobić. Masz możliwość samodzielnego zdefiniowania polityki IAM lub wykorzystania jednej z predefiniowanych, które są określane jako *polityki zarządzane*³ (ang. *managed policies*).

Do uruchomienia przykładów przedstawionych w książce konieczne jest dodanie do konta użytkownika IAM wymienionych tutaj polityk zarządzanych (jak pokazałem na rysunku 2.3):

- AmazonEC2FullAccess — ta polityka będzie wymagana w bieżącym rozdziale.
- AmazonS3FullAccess — ta polityka będzie wymagana w rozdziale 3.
- AmazonDynamoDBFullAccess — ta polityka będzie wymagana w rozdziale 3.
- AmazonRDSFullAccess — ta polityka będzie wymagana w rozdziale 3.
- CloudWatchFullAccess — ta polityka będzie wymagana w rozdziale 5.
- IAMFullAccess — ta polityka będzie wymagana w rozdziale 5.



Rysunek 2.3. Dodawanie wielu polityk zarządzanych IAM do nowego konta użytkownika IAM

³ Więcej informacji na temat polityk IAM znajdziesz na stronie https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_managed-vs-inline.html.



Uwaga dotycząca domyślnej wirtualnej chmury prywatnej

Jeżeli korzystasz z istniejącego konta AWS, musi ono mieć tzw. *default VPC*, czyli domyślną wirtualną chmurę prywatną (ang. *virtual private cloud*). VPC to odizolowany obszar konta AWS z własną siecią wirtualną i przestrzenią adresu IP. Praktycznie każdy zasób AWS jest wdrażany do VPC. Jeżeli wyraźnie nie wskażesz VPC, zasób zostanie wdrożony w domyślnej chmurze VPC, która jest częścią każdego konta AWS. Wszystkie przykłady zamieszczone w książce opierają się na tej domyślnej chmurze VPC. Dlatego też, jeśli z jakiegokolwiek powodu usunąłeś ją w koncie AWS, musisz skorzystać z innego regionu (każdy region ma własną domyślną chmurę VPC) lub utworzyć nową domyślną chmurę za pomocą konsoli AWS (<https://console.aws.amazon.com/iam/home>). W przeciwnym razie będziesz musiał uaktualnić niemalże wszystkie przykłady, aby zawierały parametr `vpc_id` lub `subnet_id` określający niestandardową chmurę VPC.

Instalacja Terraform

Terraform możesz pobrać ze strony domowej projektu znajdującej się pod adresem <https://www.terraform.io/>. Kliknij łącze pobierania, wybierz odpowiedni pakiet dla używanego systemu operacyjnego, pobierz archiwum ZIP i rozpakuj je w katalogu, w którym chcesz mieć zainstalowane oprogramowanie Terraform. Archiwum zawiera pojedynczy plik binarny o nazwie *terraform*, którego położenie powinienś dodać do zmiennej systemowej PATH. Ewentualnie Terraform znajdziesz za pomocą menedżera pakietów w używanym systemie operacyjnym, np. w przypadku macOS należy wydać polecenie `brew install terraform`.

Aby sprawdzić poprawność instalacji, w powłoce wydaj polecenie `terraform`, które powinno spowodować wygenerowanie danych wyjściowych podobnych do tutaj przedstawionych.

```
$ terraform
```

```
Usage: terraform [-version] [-help] <command> [args]
```

```
Common commands:
```

<code>apply</code>	Builds or changes infrastructure
<code>console</code>	Interactive console for Terraform interpolations
<code>destroy</code>	Destroy Terraform-managed infrastructure
<code>env</code>	Workspace management
<code>fmt</code>	Rewrites config files to canonical format
<code>(...)</code>	

Aby narzędzie Terraform mogło wprowadzać zmiany w koncie AWS, konieczne jest zdefiniowanie danych uwierzytelniających dla utworzonego wcześniej użytkownika IAM w postaci zmiennych środowiskowych `AWS_ACCESS_KEY_ID` i `AWS_SECRET_ACCESS_KEY`. Oto jak można to zrobić w powłoce systemu UNIX, Linux lub macOS:

```
$ export AWS_ACCESS_KEY_ID=(wygenerowana przez AWS wartość Access Key ID)
```

```
$ export AWS_SECRET_ACCESS_KEY=(wygenerowana przez AWS wartość Secret Access Key)
```

Warto w tym miejscu dodać, że te zmienne pozostaną aktywne jedynie w bieżącej powłoce. Jeżeli więc ponownie uruchomisz komputer lub przejdiesz do nowego okna powłoki, będziesz musiał jeszcze raz je wyeksportować.



Opcje uwierzytelniania

Poza zmiennymi środowiskowymi Terraform obsługuje te same mechanizmy uwierzytelniania co w przypadku AWS CLI i narzędzi SDK. Dlatego też ma możliwość wykorzystania danych uwierzytelniających znajdujących się w katalogu `$HOME/.aws/credentials`, które są generowane automatycznie po wydaniu polecenia `configure` w AWS CLI lub rolach IAM dodawanych do praktycznie dowolnego zasobu w AWS. Więcej informacji na ten temat znajdziesz w artykule *A Comprehensive Guide to Authenticating to AWS on the Command Line* opublikowanym na stronie <https://blog.gruntwork.io/a-comprehensive-guide-to-authenticating-to-aws-on-the-command-line-63656a686799>.

Wdrożenie pojedynczego serwera

Kod Terraform jest tworzony w języku HCL (ang. *hashicorp configuration language*) i umieszczany w plikach wraz z rozszerzeniem `.tf`⁴. To język deklaratywny, więc Twoim zadaniem będzie przedstawienie oczekiwanej infrastruktury, Terraform zaś ustali, jak ją utworzyć. Terraform może tworzyć infrastrukturę na wielu różnych platformach — w terminologii Terraform platforma docelowa jest określana mianem *dostawcy*; do obsługiwanych dostawców zaliczają się AWS, Azure, Google Cloud, DigitalOcean itd.

Kod Terraform można tworzyć za pomocą dowolnego edytora tekstu. Jeżeli dobrze poszukasz, znajdziesz wtyczki podświetlania składni Terraform dostępne dla większości najważniejszych edytorów programistycznych (konieczne będzie wpisanie w ulubionej wyszukiwarce internetowej słowa *HCL* zamiast *Terraform*), czyli m.in. dla vim, emacs, Sublime Text, Atom, Visual Studio Code i IntelliJ (ten ostatni zapewnia również obsługę refaktoryzacji, wyszukiwania użycia danego komponentu i przejścia do jego deklaracji).

Pierwszym krokiem podczas pracy z Terraform jest zwykle skonfigurowanie *dostawcy*, który ma być używany. Utwórz pusty katalog i umieść w nim plik o nazwie `main.tf` wraz z pokazaną tutaj zawartością:

```
provider "aws" {  
  region = "us-east-2"  
}
```

W ten sposób informujesz Terraform, że będziesz używać AWS jako dostawcy i chcesz wdrożyć infrastrukturę w regionie `us-east-2`. AWS ma centra danych na całym świecie, pogrupowane w regiony. Ten *region* AWS to oddzielny obszar geograficzny, taki jak `us-east-2` (Ohio), `eu-west-1` (Irlandia) i `ap-southeast-2` (Sydney).

⁴ Kod Terraform można również tworzyć w czystym formacie JSON i umieszczać w plikach wraz z rozszerzeniem `.tf.json`. Więcej informacji na temat składni HCL i JSON znajdziesz w witrynie Terraform pod adresem <https://www.terraform.io/docs/configuration/syntax.html>.

W ramach każdego regionu znajduje się wiele odizolowanych centrów danych określanych mianem *stref dostępności* (ang. *availability zones*, AZ), takich jak east-2a, us-east-2b itd.⁵

Dla każdego typu dostawcy istnieje wiele różnych *zasobów* możliwych do utworzenia, takich jak serwery, bazy danych i mechanizmy równoważenia obciążenia. Ogólna składnia utworzenia zasobu w Terraform przedstawia się następująco:

```
resource "<DOSTAWCA>" "<TYP>" "<NAZWA>" {  
  [KONFIGURACJA ...]  
}
```

gdzie DOSTAWCA to nazwa dostawcy (np. aws), TYP określa typ zasobu tworzonego w tym dostawcy (np. instance), NAZWA to identyfikator używany w kodzie Terraform w celu odwołania się do tego zasobu (np. my_instance), a KONFIGURACJA składa się z jednego lub więcej *argumentów* charakterystycznych dla tego zasobu.

Przykładowo, jeśli chcesz wdrożyć pojedynczy (wirtualny) serwer w AWS, nazywany *egzemplarzem EC2*, skorzystaj z zasobu aws_instance w pliku *main.tf*:

```
resource "aws_instance" "example" {  
  ami          = "ami-0c55b159cbf1f0"  
  instance_type = "t2.micro"  
}
```

Wprawdzie zasób aws_instance obsługuje wiele różnych argumentów, ale w tym momencie wymagane jest podanie tylko dwóch:

ami

To jest obraz AMI (ang. *amazon machine image*) przeznaczony do uruchomienia w danym egzemplarzu EC2. Zarówno bezpłatne, jak i płatne obrazy AMI znajdziesz w sekcji *AWS Marketplace* (<https://aws.amazon.com/marketplace>). Ewentualnie możesz je tworzyć samodzielnie za pomocą narzędzi takich jak Packer (więcej informacji na ten temat przedstawiłem w rozdziale 1.). W omawianym przykładzie wartością parametru ami jest identyfikator obrazu AMI Ubuntu 18.04 w regionie us-east-2. Ten obraz jest bezpłatny.

instance_type

To jest typ egzemplarza EC2 do uruchomienia. Poszczególne egzemplarze EC2 różnią się procesorem, ilością pamięci i miejsca na dysku oraz przepustowością sieci. Listę wszystkich dostępnych typów egzemplarzy EC2 znajdziesz na stronie <https://aws.amazon.com/ec2/instance-types/>. W omawianym przykładzie wykorzystałem t2.micro, który charakteryzuje się jednym wirtualnym procesorem, ma do dyspozycji 1 GB pamięci RAM i jest dostępny w ramach bezpłatnego konta AWS.

⁵ Więcej informacji na temat regionów AWS i stref AZ znajdziesz na stronie <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html>.



Korzystaj z dokumentacji

Terraform zapewnia obsługę dziesiątek dostawców, z których każdy ma dziesiątki zasobów oferujących dziesiątki argumentów. Nie ma możliwości zapamiętania tych wszystkich danych. Gdy tworzysz kod Terraform, powinieneś regularnie zaglądać do dokumentacji Terraform, w której znajdziesz informacje o dostępnych zasobach i sposobach ich używania. Przykładowo dokumentacja dla zasobu `aws_instance` znajduje się na stronie <https://www.terraform.io/docs/providers/aws/r/instance.html>. Wprawdzie używam Terraform od lat, ale mimo to wielokrotnie w ciągu dnia zaglądam do dokumentacji.

W powłoce przejdź do katalogu zawierającego utworzony wcześniej plik `main.tf`, a następnie wydaj polecenie `terraform init`:

```
$ terraform init
```

```
Initializing the backend...
```

```
Initializing provider plugins...
```

```
- Checking for available provider plugins...
```

```
- Downloading plugin for provider "aws" (terraform-providers/aws) 2.10.0...
```

```
The following providers do not have any version constraints in configuration,  
so the latest version was installed.
```

```
To prevent automatic upgrades to new major versions that may contain breaking  
changes, it is recommended to add version = "..." constraints to the  
corresponding provider blocks in configuration, with the constraint strings  
suggested below.
```

```
* provider.aws: version = "~> 2.10"
```

```
Terraform has been successfully initialized!
```

Plik binarny `terraform` zawiera podstawową funkcjonalność narzędzia Terraform, ale nie jest dostarczany wraz z kodem dla jakiegokolwiek dostawcy (np. AWS, Azure, GCP itd.). Dlatego też podczas pierwszego uruchomienia tego narzędzia konieczne jest wydanie polecenia `terraform init` nakazującego Terraform analizę kodu, ustalenie użytego dostawcy i pobranie dla niego kodu. Domyślnie pobrany kod dostawcy zostanie umieszczony w katalogu `.terraform`, który jest katalogiem roboczym dla narzędzia Terraform, więc można go dodać do pliku `.gitignore`. W późniejszych rozdziałach będziesz miał okazję poznać jeszcze kilka innych sposobów wykorzystania polecenia `terraform init` i katalogu `.terraform`. Teraz wystarczy pamiętać o konieczności wydania tego polecenia za każdym razem, gdy rozpoczynasz pracę nad nowym kodem Terraform. Wielokrotne wydanie tego polecenia jest bezpieczne (jest ono powtarzalne).

Po pobraniu kodu dostawcy można już wydać polecenie `terraform plan`:

```
$ terraform plan
```

```
(...)
```

```
Terraform will perform the following actions:
```

```
# aws_instance.example will be created
+ resource "aws_instance" "example" {
  + ami                    = "ami-0c55b159cbfafelf0"
  + arn                   = (known after apply)
  + associate_public_ip_address = (known after apply)
  + availability_zone      = (known after apply)
  + cpu_core_count         = (known after apply)
  + cpu_threads_per_core   = (known after apply)
  + get_password_data      = false
  + host_id                = (known after apply)
  + id                    = (known after apply)
  + instance_state         = (known after apply)
  + instance_type          = "t2.micro"
  + ipv6_address_count     = (known after apply)
  + ipv6_addresses        = (known after apply)
  + key_name               = (known after apply)
  (...)
}
```

Plan: 1 to add, 0 to change, 0 to destroy.

Polecenie terraform plan pozwala sprawdzić, co zostanie zrobione, jeszcze zanim będą wprowadzone jakiegokolwiek zmiany. To doskonały sposób na sprawdzenie kodu przed jego wydaniem. Dane wyjściowe wygenerowane przez to polecenie są podobne do otrzymanych po wydaniu polecenia diff w systemie UNIX lub Linux albo po wydaniu polecenia git: wszystko oznaczone znakiem + będzie utworzone, oznaczone znakiem - zostanie usunięte, a oznaczone tyldą, ~, zostanie zmodyfikowane. W omawianym przykładzie widzimy, że Terraform planuje jedynie utworzenie pojedynczego egzemplarza EC2, co jest dokładnie tym, czego w tym miejscu oczekujemy.

Rzeczywiste utworzenie egzemplarza następuje po wydaniu polecenia terraform apply:

```
$ terraform apply
```

```
(...)
```

Terraform will perform the following actions:

```
# aws_instance.example will be created
+ resource "aws_instance" "example" {
  + ami                    = "ami-0c55b159cbfafelf0"
  + arn                   = (known after apply)
  + associate_public_ip_address = (known after apply)
  + availability_zone      = (known after apply)
  + cpu_core_count         = (known after apply)
  + cpu_threads_per_core   = (known after apply)
  + get_password_data      = false
  + host_id                = (known after apply)
  + id                    = (known after apply)
  + instance_state         = (known after apply)
  + instance_type          = "t2.micro"
  + ipv6_address_count     = (known after apply)
  + ipv6_addresses        = (known after apply)
  + key_name               = (known after apply)
  (...)
}
```

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value:

Zwróć uwagę na to, że polecenie `terraform apply` wyświetla dokładnie te same dane wyjściowe, które już wcześniej wygenerowało polecenie `terraform plan`, oraz prosi o potwierdzenie operacji. Tak więc plan operacji jest wprawdzie dostępny w postaci oddzielnego polecenia, ale przydaje się raczej do sprawdzenia operacji podczas analizy kodu (do tego tematu powrócę w rozdziale 8.). W większości przypadków będziesz od razu wydawać polecenie `terraform apply` i przeglądać dane wyjściowe planu przed potwierdzeniem operacji.

Wpisz **yes** i naciśnij klawisz *Enter*, aby wdrożyć egzemplarz EC2:

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: **yes**

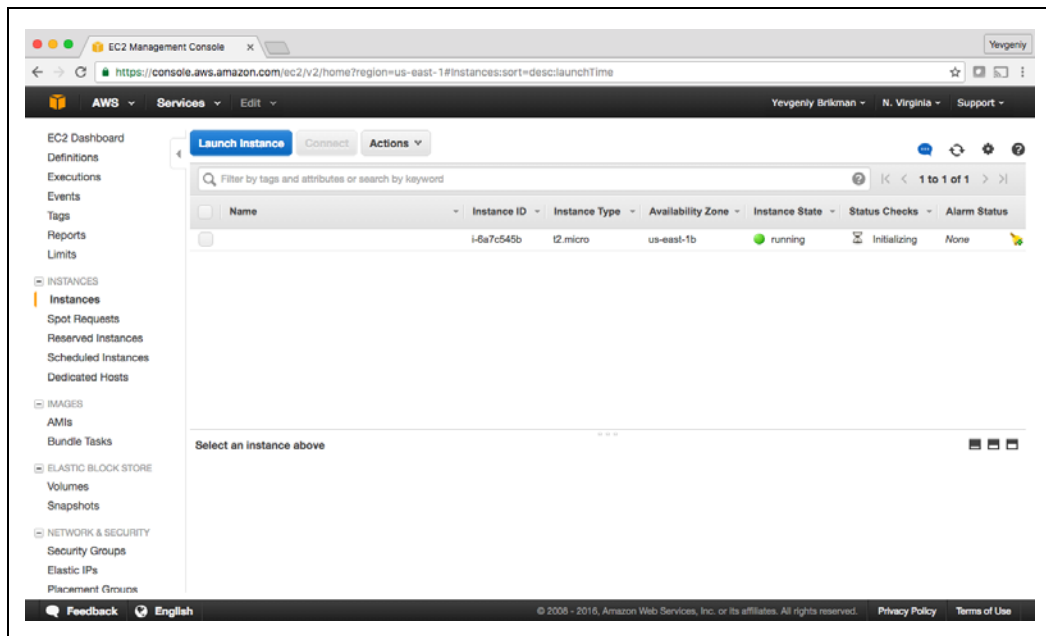
```
aws_instance.example: Creating...  
aws_instance.example: Still creating... [10s elapsed]  
aws_instance.example: Still creating... [20s elapsed]  
aws_instance.example: Still creating... [30s elapsed]  
aws_instance.example: Creation complete after 38s [id=i-07e2a3e006d785906]
```

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Moje gratulacje, w ten sposób za pomocą Terraform wdrożyłeś egzemplarz EC2 w ramach utworzonego wcześniej konta AWS. Aby to potwierdzić, przejdź do konsoli EC2 (<https://console.aws.amazon.com/iam/home>) — powinieneś zobaczyć dane podobne do pokazanych na rysunku 2.4.

Oczywiście ten egzemplarz istnieje, choć trzeba przyznać, że to nie jest najbardziej ekscytujący przykład na świecie. Zróbmy coś znacznie bardziej interesującego. Przede wszystkim zwróć uwagę na to, że utworzony egzemplarz EC2 nie ma nazwy. Można ją dodać za pomocą sekcji `tags` w zasobie `aws_instance`:

```
resource "aws_instance" "example" {  
  ami           = "ami-0c55b159cbfaffe1f0"  
  instance_type = "t2.micro"  
  
  tags = {  
    Name = "terraform-example"  
  }  
}
```

Rysunek 2.4. Pojedynczy egzemplarz EC2

Ponownie wydaj polecenie `terraform apply` i zobacz, jaki będzie efekt jego wykonania.

```
$ terraform apply
```

```
aws_instance.example: Refreshing state...
(...)
```

Terraform will perform the following actions:

```
# Ten egzemplarz aws_instance.example zostanie uaktualniony.
~ resource "aws_instance" "example" {
    ami                = "ami-0c55b159cbfafa1f0"
    availability_zone   = "us-east-2b"
    instance_state      = "running"
    (...)
+ tags                = {
    + "Name" = "terraform-example"
  }
  (...)
}
```

Plan: 0 to add, 1 to change, 0 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.

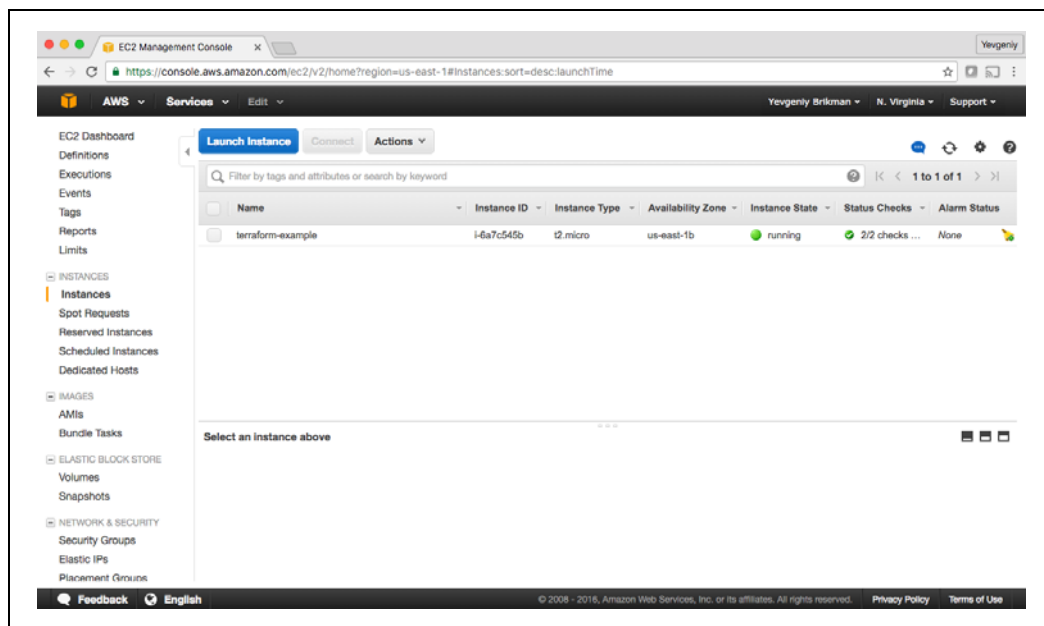
Only 'yes' will be accepted to approve.

Enter a value:

Terraform monitoruje wszystkie zasoby utworzone dla danego zbioru plików konfiguracyjnych, więc doskonale wie o istnieniu egzemplarza EC2 (zwróć uwagę na komunikat `Refreshing state...`).

podczas wykonywania polecenia `terraform apply`). Dlatego też pokazuje różnice między aktualnie wdrożonym egzemplarzem i tym zdefiniowanym w kodzie Terraform (to jest jedna z zalet używania języka deklaratywnego zamiast proceduralnego, o czym wspominałem w rozdziale 1.). Zgodnie w wygenerowanych danych wyjściowych Terraform chce utworzyć pojedynczy tag o nazwie `Name`, co jest dokładnie tym, czego oczekujemy. Wpisz więc słowo **yes** i naciśnij klawisz `Enter`.

Po odświeżeniu konsoli EC2 zobaczysz dane podobne do pokazanych na rysunku 2.5.



Rysunek 2.5. Egzemplarz EC2 ma teraz nadaną nazwę

Skoro utworzyłeś już pewien kod Terraform, być może będziesz chciał go umieścić w systemie kontroli wersji. To pozwala na współdzielenie kodu wraz z innymi członkami zespołu, śledzenie historii zmian infrastruktury, a także używanie dziennika zdarzeń operacji zatwierdzenia do usuwania błędów. Oto polecenia tworzące lokalne repozytorium Git przeznaczone do przechowywania pliku konfiguracyjnego Terraform:

```
$ git init
$ git add main.tf
$ git commit -m "Pierwsze przekazanie pliku do repozytorium"
```

Powinieneś utworzyć również plik o nazwie `.gitignore` wskazujące systemowi kontroli wersji Git pliki i katalogi, które mają być ignorowane i nie trafić do repozytorium. Oto zawartość wymienionego pliku:

```
.terraform
*.tfstate
*.tfstate.backup
```

W omawianym przykładzie plik `.gitignore` nakazuje Gitowi zignorowanie katalogu `.terraform` używanego przez narzędzie Terraform jako katalog tymczasowy. Zignorowane mają być również pliki `*.tfstate`, które Terraform wykorzystuje do przechowywania informacji o stanie (z rozdziału 3.

dowiesz się, dlaczego tych plików nie należy umieszczać w systemie kontroli wersji). Plik `.gitignore` również trzeba umieścić w repozytorium:

```
$ git add .gitignore
$ git commit -m "Dodanie pliku .gitignore"
```

Aby współdzielić plik ze współpracownikami, musisz utworzyć współdzielone repozytorium Git, do którego będą mieli dostęp. Jednym z możliwych rozwiązań jest wykorzystanie serwisu GitHub. Przejdź do witryny <https://github.com/>, utwórz konto (o ile jeszcze go nie masz) i nowe repozytorium. Lokalne repozytorium Git skonfiguruj do użycia nowego repozytorium GitHub jako zdalnego punktu końcowego o nazwie `origin`, co wymaga wydania następującego polecenia:

```
$ git remote add origin git@github.com:<NAZWA_UŻYTKOWNIKA>/<NAZWA_REPOZYTORIUM>.git
```

Gdy będziesz chciał udostępnić kod współpracownikom, powinieneś go *przekazać* do zdalnego repozytorium za pomocą następującego polecenia:

```
$ git push origin master
```

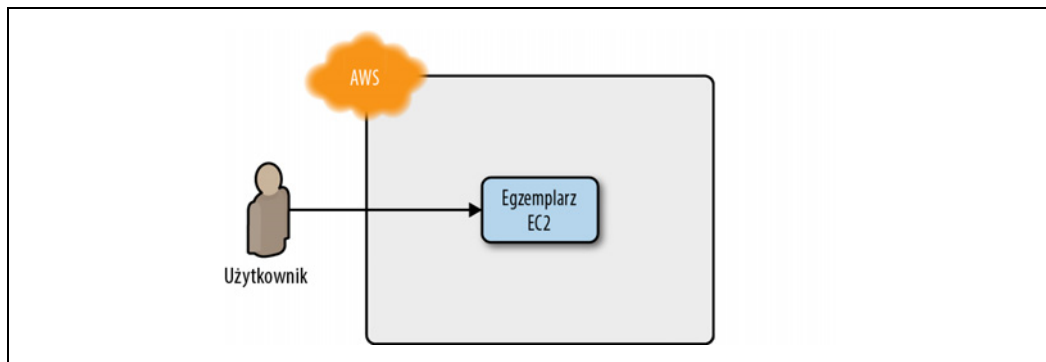
Natomiast w celu zobaczenia zmian wprowadzonych przez współpracowników musisz *pobrać* zmiany ze zdalnego repozytorium za pomocą następującego polecenia:

```
$ git pull origin master
```

Podczas lektury pozostałej części książki, i ogólnie w trakcie pracy z Terraform, upewnij się co do częstego wydawania poleceń `git commit` i `git push`, aby wprowadzone zmiany przekazywać do repozytorium. Dzięki temu nie tylko możesz współpracować z innymi osobami nad danym kodem, ale także zmiany infrastruktury są zapisywane w komunikatach operacji przekazania kodu do repozytorium, co okazuje się niezwykle użyteczne podczas procesu usuwania błędów z kodu. Więcej informacji na temat używania Terraform w zespole znajdziesz w rozdziale 8.

Wdrożenie pojedynczego serwera WWW

Następnym krokiem jest uruchomienie serwera WWW w przygotowanym egzemplarzu. Moim celem jest tutaj wdrożenie najprostszej możliwej architektury sieciowej: pojedynczy serwer WWW udzielający odpowiedzi na żądania HTTP (zobacz rysunek 2.6).



Rysunek 2.6. Rozpocznij od najprostszej architektury — pojedynczy serwer WWW uruchomiony w egzemplarzu AWS i udzielający odpowiedzi na żądania HTTP

W rzeczywistym projekcie serwer WWW utworzysz prawdopodobnie za pomocą frameworka takiego jak Ruby on Rails lub Django. Jednak w celu zachowania prostoty omawianego przykładu zdecydowałem się na przygotowanie najprostszego serwera WWW, który zawsze będzie zwracał komunikat *Witaj, świecie*⁶.

```
#!/bin/bash
echo "Witaj, świecie" > index.html
nohup busybox httpd -f -p 8080 &
```

To jest skrypt Bash umieszczający tekst *Witaj, świecie* w pliku o nazwie *index.html* i uruchamiający narzędzie busybox (<https://busybox.net/>), które jest domyślnie zainstalowane w systemie Ubuntu w celu uruchomienia serwera WWW nasłuchującego na porcie numer 8080 i udostępniającego ten plik. Polecenie busybox zostało opakowane wywołaniem nohup i &, aby ten serwer na stałe działał w tle po zakończeniu wykonywania samego skryptu Bash.



Numery portów

Powodem użycia w omawianym przykładzie portu numer 8080 zamiast domyślnego portu HTTP 80 jest to, że nasłuchiwanie na porcie o numerze niższym niż 1024 wymaga uprawnień użytkownika root. To jest ryzykowne, ponieważ jeżeli atakujący włamie się do serwera, uzyska pełnię uprawnień, które ma w systemie użytkownik root.

Dlatego też najlepszym rozwiązaniem jest uruchamianie serwera WWW w ramach konta użytkownika innego niż root i dysponującego jedynie ograniczonymi uprawnieniami. To oznacza konieczność nasłuchiwania na portach o znacznie większych numerach. Jak dowiesz się z dalszej części rozdziału, można skonfigurować mechanizm równoważenia obciążenia nasłuchujący na porcie 80, a następnie przekazujący ruch sieciowy do portów o wyższych numerach w serwerze WWW.

Jak można uruchomić ten skrypt za pomocą egzemplarza EC2? Dokładnie w ten sposób, który przedstawiłem w rozdziale 1.: wystarczy za pomocą narzędzia Packer przygotować własny obraz AMI wraz z zainstalowanym serwerem WWW. Skoro w omawianym przykładzie serwer WWW to zaledwie jeden wiersz kodu wykonywany przez busybox, wystarczy wykorzystanie zwykłego obrazu AMI zawierającego dystrybucję Ubuntu 18.04 i uruchomienie tego skryptu jako części konfiguracji *danych użytkownika* egzemplarza EC2. Po uruchomieniu egzemplarza EC2 masz możliwość przekazania do danych użytkownika skryptu powłoki lub dyrektywy inicjalizującej chmurę, a egzemplarz EC2 wykona je podczas uruchamiania egzemplarza. W celu przekazania skryptu powłoki do danych użytkownika należy skorzystać z argumentu `user_data` w kodzie Terraform:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafa1f0"
  instance_type = "t2.micro"

  user_data = <<-EOF
    #!/bin/bash
    echo "Witaj, świecie" > index.html
    nohup busybox httpd -f -p 8080 &
  EOF
}
```

⁶ Użyteczną listę serwerów HTTP tworzonych za pomocą tylko jednego wiersza kodu znajdziesz na stronie <https://gist.github.com/willurd/5720255>.

```
tags = {
  Name = "terraform-example"
}
```

Znaczniki `<<-EOF` i `EOF` to składnia typu *heredoc* w Terraform pozwalająca na tworzenie wielowierszowych ciągów tekstowych bez konieczności wstawiania znaków nowego wiersza.

Zanim ten serwer WWW zadziała, konieczne jest wykonanie jeszcze jednego kroku. Domyślnie AWS nie pozwala na jakikolwiek ruch przychodzący lub wychodzący z egzemplarza EC2. Aby zezwolić egzemplarzowi EC2 na otrzymywanie ruchu sieciowego poprzez port 8080, konieczne jest utworzenie grupy bezpieczeństwa.

```
resource "aws_security_group" "instance" {
  name = "terraform-example-instance"

  ingress {
    from_port = 8080
    to_port   = 8080
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Ten kod powoduje utworzenie nowego zasobu o nazwie `aws_security_group` (zwróć uwagę na to, że nazwy wszystkich zasobów dla dostawcy AWS rozpoczynają się prefiksem `aws_`) zezwalającego na obsługę żądań TCP kierowanych do portu 8080 z adresów IP określonych przez blok CIDR `0.0.0.0/0`. Blok *CIDR* pozwala w zwięzły sposób określić zakres adresów IP. Przykładowo blok `10.0.0.0/24` przedstawia wszystkie adresy IP z przedziału od `10.0.0.0` do `10.0.0.255`. Natomiast blok `0.0.0.0/0` przedstawia każdy możliwy adres IP, więc zdefiniowana tutaj grupa bezpieczeństwa zezwala na obsługę żądań do portu 8080 przychodzących z dowolnego adresu IP⁷.

Ograniczenie się do zaledwie utworzenia grupy bezpieczeństwa jest niewystarczające. Konieczne jest jeszcze nakazanie egzemplarzowi EC2 jej użycia, co odbywa się przez przekazanie identyfikatora grupy jako argumentu `vpc_security_group_ids` zasobu `aws_instance`. Aby to zrobić, musisz dowiedzieć się nieco o *wyrażeniach* Terraform.

Wyrażenie w Terraform to cokolwiek, co zwraca wartość. Miałeś już okazję poznać najprostszy typ wyrażenia, czyli *literal*, w postaci ciągu tekstowego (np. `"ami-0c55b159cbfafe1f0"`) i liczby (np. `5`). Terraform obsługuje jeszcze wiele innych typów wyrażeń, przedstawię je w tej książce.

Szczególnie użytecznym typem wyrażenia jest *odwołanie*, które pozwala na uzyskanie dostępu do wartości zdefiniowanych w innych fragmentach kodu. Jeżeli chcesz otrzymać dostęp do identyfikatora zasobu grupy bezpieczeństwa, będziesz musiał skorzystać z *odwołania atrybutu zasobu*, którego składnia przedstawia się następująco:

```
<DOSTAWCA>.<TYP>.<NAZWA>.<ATRYBUT>
```

⁷ Jeżeli chcesz dowiedzieć się więcej na temat sposobu działania bloku CIDR, zajrzyj do artykułu w Wikipedii na stronie https://pl.wikipedia.org/wiki/Classless_Inter-Domain_Routing. Przydatny kalkulator pozwalający na konwersję zakresów adresów IP i notacji CIDR jest dostępny na stronie <https://cidr.xyz/>, a także w powłoce po zainstalowaniu polecenia `ipcalc`.

gdzie DOSTAWCA to nazwa dostawcy (np. aws), TYP określa typ zasobu tworzonego w tym dostawcy (np. security_group), NAZWA to nazwa zasobu (np. grupa bezpieczeństwa ma nazwę instance), a ATRYBUT składa się z jednego lub więcej atrybutów *wyeksportowanych* przez dany zasób (listę dostępnych atrybutów znajdziesz w dokumentacji poszczególnych zasobów). Grupa bezpieczeństwa eksportuje atrybut o nazwie id, więc odwołujące się do niego wyrażenie ma następującą postać:

```
aws_security_group.instance.id
```

Identyfikator grupy bezpieczeństwa można stosować w argumentcie vpc_security_group_ids zasobu aws_instance.

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafa1f0"
  instance_type = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]

  user_data = <<-EOF
    #!/bin/bash
    echo "Witaj, świecie" > index.html
    nohup busybox httpd -f -p 8080 &
  EOF

  tags = {
    Name = "terraform-example"
  }
}
```

Po dodaniu odwołania z jednego zasobu do innego powstaje *niejawna zależność*. Terraform przetwarza te zależności, tworzy ich wykres i wykorzystuje go do automatycznego określania kolejności, w jakiej powinny być tworzone zasoby. Przykładowo, jeśli kod jest wdrażany zupełnie od zera, Terraform wie o konieczności utworzenia grupy bezpieczeństwa przez egzemplarzem EC2, ponieważ egzemplarz EC2 odwołuje się do identyfikatora grupy bezpieczeństwa. Istnieje nawet możliwość wyświetlenia przez Terraform wykresu zależności, co wymaga wydania polecenia terraform graph.

```
$ terraform graph
```

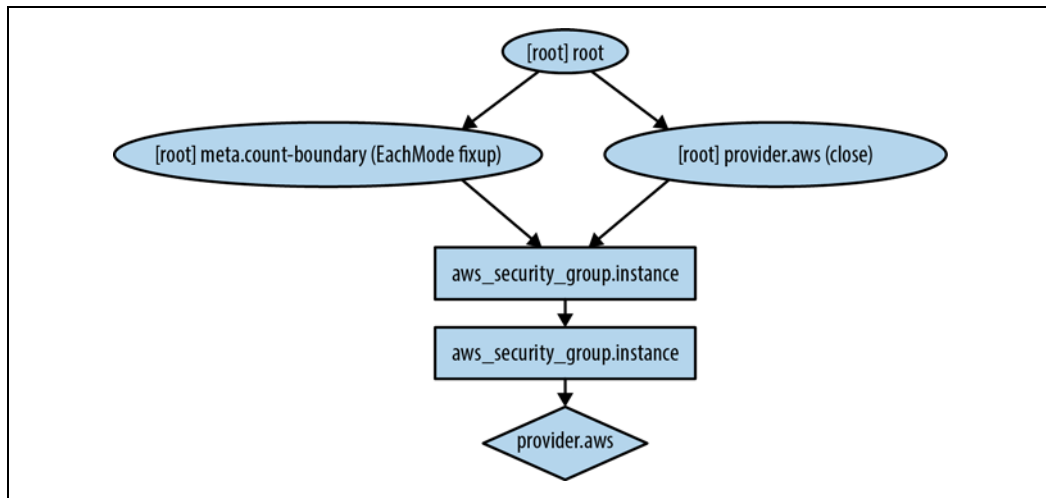
```
digraph {
  compound = "true"
  newrank = "true"
  subgraph "root" {
    "[root] aws_instance.example"
    [label = "aws_instance.example", shape = "box"]
    "[root] aws_security_group.instance"
    [label = "aws_security_group.instance", shape = "box"]
    "[root] provider.aws"
    [label = "provider.aws", shape = "diamond"]
    "[root] aws_instance.example" ->
      "[root] aws_security_group.instance"
    "[root] aws_security_group.instance" ->
      "[root] provider.aws"
    "[root] meta.count-boundary (EachMode fixup)" ->
      "[root] aws_instance.example"
    "[root] provider.aws (close)" ->
      "[root] aws_instance.example"
    "[root] root" ->
      "[root] meta.count-boundary (EachMode fixup)"
  }
}
```

```

    "[root] root" ->
      "[root] provider.aws (close)"
  }
}

```

Dane wyjściowe tego polecenia to wykres opisany w języku o nazwie DOT i możliwy do konwersji na postać obrazka, podobnego do wykresu zależności pokazanego na rysunku 2.7. To wymaga użycia dowolnej aplikacji typu Graphviz lub jej wersji online — GraphvizOnline (<http://dreampuf.github.io/GraphvizOnline/>).



Rysunek 2.7. Wykres zależności dla egzemplarza EC2 i jego grupy bezpieczeństwa

Gdy Terraform analizuje drzewo zależności, zasoby tworzy jednocześnie, na ile to możliwe. To oznacza, że zmiany są wprowadzane dość efektywnie. To jest piękno języka deklaratywnego — wskazujesz tylko cel i pozwalasz Terraform na znalezienie najefektywniejszego sposobu na jego osiągnięcie.

Po wydaniu polecenia `terraform apply` zobaczysz, że Terraform chce utworzyć grupę bezpieczeństwa i zastąpić egzemplarz EC2 nowym, który zawiera nowe dane użytkownika.

\$ terraform apply

(...)

Terraform will perform the following actions:

```

# Egzemplarz aws_instance.example musi być zastąpiony nowym.
-/+ resource "aws_instance" "example" {
  ami                = "ami-0c55b159cbfafelf0"
  ~ availability_zone = "us-east-2c" -> (known after apply)
  ~ instance_state   = "running" -> (known after apply)
  instance_type      = "t2.micro"
  (...)
  + user_data         = "c765373..." # forces replacement
  ~ volume_tags       = {} -> (known after apply)
  ~ vpc_security_group_ids = [
    - "sg-871fa9ec",

```

```

] -> (known after apply)
(...)
}
# Utworzenie grupy bezpieczeństwa aws_security_group.instance.
+ resource "aws_security_group" "instance" {
+   arn                = (known after apply)
+   description        = "Managed by Terraform"
+   egress              = (known after apply)
+   id                  = (known after apply)
+   ingress              = [
+     + {
+       + cidr_blocks    = [
+         + "0.0.0.0/0",
+       ]
+       + description    = ""
+       + from_port      = 8080
+       + ipv6_cidr_blocks = []
+       + prefix_list_ids = []
+       + protocol        = "tcp"
+       + security_groups = []
+       + self             = false
+       + to_port         = 8080
+     },
+   ]
+   name                = "terraform-example-instance"
+   owner_id             = (known after apply)
+   revoke_rules_on_delete = false
+   vpc_id               = (known after apply)
}

```

Plan: 2 to add, 0 to change, 1 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

Enter a value:

Znaki + i - w danych wyjściowych wygenerowanych przez polecenie terraform plan oznaczają „zastąpienie”. Poszukaj wyrażenia *forces replacement* (z ang. wymuszone zastąpienie) w danych wyjściowych, a zobaczysz, co zostanie zastąpione. W przypadku wielu argumentów zasobu `aws_instance` ich zmiana wymusza zastąpienie zasobu, co oznacza zakończenie działania pierwotnego egzemplarza EC2 i utworzenie zupełnie nowego. To jest przykład omówionego w rozdziale 1. paradygmatu infrastruktury niemodyfikowalnej. Warto w tym miejscu dodać, że pomimo zastąpienia serwera WWW nowym żaden z użytkowników serwera nie doświadczy przestoju — w rozdziale 5. przedstawię więcej informacji na temat wdrożenia bez przestoju w Terraform.

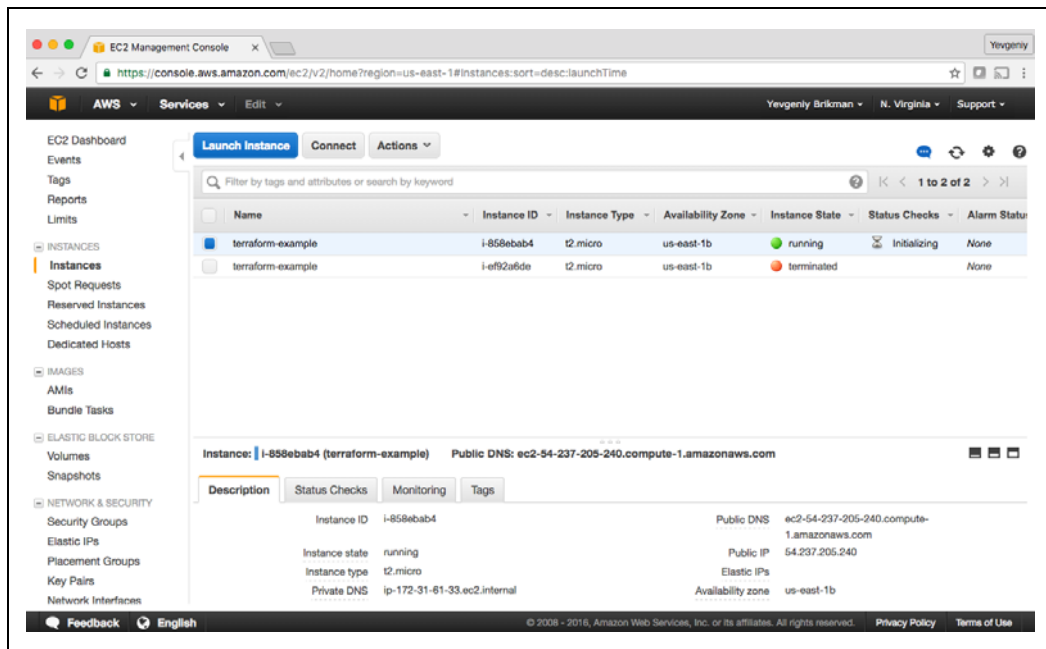
Ponieważ plan przedstawia się dobrze, wpisz **yes** i naciśnij klawisz *Enter*. Zobaczysz, że nowy egzemplarz EC2 został wdrożony (rysunek 2.8).

Po kliknięciu nowego egzemplarza jego publiczny adres IP znajdziesz w panelu *Description* w dolnej części ekranu. Oczekaj minutę lub dwie na pełne uruchomienie egzemplarza, a następnie za pomocą przeglądarki WWW lub polecenia typu `curl` wykonaj żądanie HTTP pod ten adres IP i port numer 8080:

```

$ curl http://<PUBLICZNY_ADRES_IP_EGZEMPLARZA_EC2>:8080
Witaj, świecie

```

Rysunek 2.8. Nowy egzemplarz EC2 wraz z kodem serwera WWW zastępuje poprzedni egzemplarz

Świetnie! Masz uruchomiony serwer WWW w chmurze AWS.



Bezpieczeństwo sieci

W celu zachowania prostoty wszystkich przykładów przedstawionych w książce są one wdrażane nie tylko w domyślnej prywatnej wirtualnej chmurze (o czym wspominałem już wcześniej), ale również w domyślnych podsieciach danej sieci VPC. Standardowo sieć VPC jest podzielona na jedną lub więcej podsieci, z których każda ma własny adres IP. Podsieci w sieci domyślnej VPC są *podsieciami publicznymi*, co oznacza, że mają adresy IP dostępne z poziomu publicznego internetu. Dlatego też egzemplarz EC2 możesz przetestować z poziomu komputera domowego.

Uruchomienie serwera w podsieci publicznej jest dobrym rozwiązaniem podczas krótkich eksperymentów, natomiast w rzeczywistych projektach wiąże się z ryzykiem. Hakerzy na całym świecie *nieustannie* losowo skanują adresy IP w poszukiwaniu jakichkolwiek słabych ogniów. Jeżeli Twoje serwery są udostępnione publicznie, wystarczy pozostawienie jednego niechronionego portu lub uruchomienie przestarzałego kodu zawierającego znane luki w zabezpieczeniach, a ktoś może się do nich włamać.

Dlatego też w systemach produkcyjnych wszystkie serwery i zdecydowanie wszystkie magazyny danych powinny być wdrażane w *podsieciach prywatnych*, których adresy IP są dostępne jedynie wewnątrz danego VPC, a nie z zewnątrz (np. z publicznego

internetu). Jedyne serwery, które powinny działać w podsieciach publicznych, to mała liczba odwrotnych proxy i mechanizmów równoważenia obciążenia, aby projekt był maksymalnie zamknięty i chroniony (przykład wdrożenia mechanizmu równoważenia obciążenia przedstawię w dalszej części rozdziału).

Wdrażanie konfigurowalnego serwera WWW

Być może zauważyłeś, że w kodzie serwera WWW numer portu 8080 został powielony i pojawia się zarówno w grupie bezpieczeństwa, jak i konfiguracji danych użytkownika. To oznacza złamanie reguły *nie powtarzaj się* (ang. *don't repeat yourself*, DRY): każdy fragment wiedzy musi mieć pojedynczą, jednoznaczną i kategorię reprezentację w systemie⁸. Jeżeli numer portu został podany w dwóch miejscach, bardzo łatwo jest uaktualnić go w jednym i zapomnieć o tym w drugim miejscu.

Aby pozostać w zgodzie z regułą DRY i zapewnić większą konfigurowalność, Terraform pozwala na zdefiniowanie zmiennych *danych wejściowych*. Oto składnia przeznaczona do zadeklarowania takiej zmiennej:

```
variable "NAZWA" {  
  [KONFIGURACJA ...]  
}
```

Część główna deklaracji zmiennej może zawierać trzy parametry, wszystkie są opcjonalne:

description

Używanie tego parametru w celu przygotowania dokumentacji o sposobie stosowania zmiennej zawsze będzie dobrym pomysłem. Twój współpracownik będzie miał dostęp do tego opisu nie tylko podczas odczytywania kodu źródłowego, ale również po wydaniu poleceń `terraform plan` lub `terraform apply` (przykłady przedstawię wkrótce).

default

Mamy kilka sposobów na dostarczenie wartości zmiennej, m.in. przekazanie w powłocie (za pomocą opcji `-var`), poprzez plik (za pomocą opcji `-var-file`) i poprzez zmienną środowiskową. (Terraform wyszukuje zmienne środowiskowe o nazwach `TF_VAR_<nazwa_zmiennej>`). Jeżeli wartość nie zostanie przekazana, zmienna wykorzysta wartość domyślną. Natomiast w przypadku braku wartości domyślnej Terraform interaktywnie poprosi o jej podanie.

type

Ten parametr pozwala na wymuszenie *ograniczenia typu* w zmiennych przekazywanych przez użytkownika. Terraform obsługuje wiele różnych ograniczeń typu m.in. `string`, `number`, `bool`, `list`, `map`, `set`, `object`, `tuple` i `any`. Jeżeli nie podasz typu, przyjęte będzie założenie, że jest nim `any`.

Oto przykład zmiennej danych wejściowych sprawdzającej, czy przekazaną wartością jest liczba:

```
variable "number_example" {  
  description = "Przykład zmiennej typu number w Terraform"  
  type        = number  
}
```

⁸ Zob. Andy Hunt, Dave Thomas, *Pragmatyczny programista. Od czeladnika do mistrza*, Helion.

```
    default    = 42
  }
```

To natomiast przykład zmiennej sprawdzającej, czy wartość jest liczbą:

```
variable "list_example" {
  description = "Przykład zmiennej typu list Terraform"
  type        = list
  default     = ["a", "b", "c"]
}
```

Istnieje możliwość łączenia ograniczeń typu. W tym przykładzie zmienna danych wejściowych wymaga, aby wszystkie elementy listy były liczbami.

```
variable "list_numeric_example" {
  description = "Przykład zmiennej w postaci listy liczb w Terraform"
  type        = list(number)
  default     = [1, 2, 3]
}
```

Kolejna zmienna wymaga, aby wszystkie wartości na mapie były ciągami tekstowymi.

```
variable "map_example" {
  description = "Przykład mapowania w Terraform"
  type        = map(string)
  default = {
    key1 = "wartość1"
    key2 = "wartość2"
    key3 = "wartość3"
  }
}
```

Istnieje również możliwość tworzenia bardziej skomplikowanych *typów strukturalnych* za pomocą ograniczeń typu `object` i `tuple`.

```
variable "object_example" {
  description = "Przykład typu strukturalnego w Terraform"
  type        = object({
    name      = string
    age       = number
    tags      = list(string)
    enabled   = bool
  })

  default = {
    name      = "value1"
    age       = 42
    tags      = ["a", "b", "c"]
    enabled   = true
  }
}
```

W tym przykładzie została utworzona zmienna danych wejściowych wymagająca wartości w postaci obiektu wraz z kluczami `name` (to musi być ciąg tekstowy), `age` (to musi być liczba), `tags` (to musi być lista ciągów tekstowych) i `enabled` (to musi być wartość boolowska). Jeżeli spróbujesz przypisać tej zmiennej wartość niedopasowaną do tego typu, Terraform natychmiast wygeneruje komunikat błędu. W kolejnym przykładzie pokazałem wynik próby przypisania kluczowi `enabled` ciągu tekstowego zamiast wartości boolowskiej.

```
variable "object_example_with_error" {
  description = "Przykład typu strukturalnego w Terraform generujący błąd"
  type       = object({
    name     = string
    age      = number
    tags     = list(string)
    enabled  = bool
  })

  default = {
    name     = "value1"
    age      = 42
    tags     = ["a", "b", "c"]
    enabled  = "invalid"
  }
}
```

Skutkiem będzie wygenerowanie błędu:

```
$ terraform apply
```

```
Error: Invalid default value for variable
```

```
on variables.tf line 78, in variable "object_example_with_error":
78:   default = {
79:     name   = "value1"
80:     age    = 42
81:     tags   = ["a", "b", "c"]
82:     enabled = "invalid"
83:   }
```

This default value is not compatible with the variable's type constraint: a bool is required.

W przykładzie serwera WWW potrzebna jest zmienna przechowująca numer portu:

```
variable "server_port" {
  description = "Numer portu używany przez serwer dla żądań HTTP"
  type       = number
}
```

Zwróć uwagę na to, że zmienna `server_port` nie zawiera parametru `default`, więc jeśli teraz wydasz polecenie `terraform apply`, Terraform interaktywnie poprosi o podanie wartości dla `server_port` i wyświetli wartość parametru `description` zmiennej:

```
$ terraform apply
```

```
var.server_port
  Numer portu używany przez serwer dla żądań HTTP
```

```
Enter a value:
```

Jeżeli nie chcesz mieć do czynienia z interaktywnym podawaniem wartości, możesz ją dostarczyć za pomocą opcji `-var` powłoki.

```
$ terraform plan -var "server_port=8080"
```

Zmienną można zdefiniować także za pomocą zmiennej środowiskowej o nazwie `TF_VAR_<nazwa>`, gdzie `<nazwa>` wskazuje nazwę zmiennej, której wartość jest przypisywana.

```
$ export TF_VAR_server_port=8080
$ terraform plan
```

Jeśli natomiast nie chcesz zapamiętywać dodatkowych argumentów powłoki w trakcie każdego wydawania poleceń `terraform plan` i `terraform apply`, możesz określić wartość parametru `default`.

```
variable "server_port" {
  description = "Numer portu używany przez serwer dla żądań HTTP"
  type        = number
  default     = 8080
}
```

W celu użycia w kodzie Terraform wartości zmiennej danych wejściowych możesz skorzystać z nowego typu wyrażenia o nazwie *odwołanie do zmiennej*, którego składnia przedstawia się następująco:

```
var.<NAZWA_ZMIENNEJ>
```

Spójrz na przykład pokazujący przypisanie parametrom `from_port` i `to_port` grupy bezpieczeństwa wartości zmiennej `server_port`.

```
resource "aws_security_group" "instance" {
  name = "terraform-example-instance"

  ingress {
    from_port = var.server_port
    to_port   = var.server_port
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Dobrym rozwiązaniem jest używanie tej samej zmiennej podczas definiowania portu w skrypcie danych użytkownika. W celu użycia odwołania wewnątrz literału ciągu tekstowego konieczne jest wykorzystanie nowego typu wyrażenia o nazwie *interpolacja*, którego składnia przedstawia się następująco:

```
"${...}"
```

W nawiasie klamrowym można umieścić dowolne, prawidłowe odwołanie, a Terraform skonwertuje je na postać ciągu tekstowego. Dla przykładu spójrz na sposób użycia `var.server_port` w ciągu tekstowym danych użytkownika:

```
user_data = <<-EOF
#!/bin/bash
echo "Witaj, świecie" > index.html
nohup busybox httpd -f -p ${var.server_port} &
EOF
```

Poza zmiennymi danych wejściowych Terraform pozwala na definiowanie *zmiennych danych wyjściowych* za pomocą następującej składni:

```
output "<NAZWA>" {
  value = <WARTOŚĆ>
  [KONFIGURACJA ...]
}
```

Tutaj `NAZWA` to nazwa zmiennej danych wyjściowych, `WARTOŚĆ` zaś może być dowolnym wyrażeniem Terraform, które ma zostać wyświetlone. Z kolei `KONFIGURACJA` może zawierać dwa opcjonalne parametry dodatkowe:

description

Używanie tego parametru w celu przygotowania dokumentacji o typie danych przechowywanych przez zmienną zawsze jest dobrym pomysłem.

sensitive

Przypisanie temu parametrowi wartości true nakazuje Terraform, aby nie wyświetlać wartości tego parametru na końcu wykonywania polecenia `terraform apply`. To jest użyteczna możliwość, gdy zmienna danych wyjściowych zawiera informacje wrażliwe, takie jak hasło lub klucz prywatny.

Przykładowo, zamiast korzystać z konsoli EC2 i samodzielnie wyszukiwać adres IP serwera, odpowiedni adres IP można dostarczyć w postaci zmiennej danych wyjściowych.

```
output "public_ip" {
  value       = aws_instance.example.public_ip
  description = "Publiczny adres IP serwera WWW"
}
```

Ten kod ponownie korzysta z odwołania do atrybutu, tym razem jest to atrybut `public_ip` zasobu `aws_instance`. Jeżeli teraz wydasz polecenie `terraform apply`, Terraform nie wprowadzi żadnych zmian (ponieważ nie został zmodyfikowany żaden zasób), ale na końcu wyświetli nowe dane wyjściowe.

```
$ terraform apply
```

```
(...)
```

```
aws_security_group.instance: Refreshing state... [id=sg-078ccb4f9533d2c1a]
aws_instance.example: Refreshing state... [id=i-028cad2d4e6bddec6]
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
public_ip = 54.174.13.5
```

Jak możesz zobaczyć, wartość zmiennej danych wyjściowych jest wyświetlana w konsoli po wydaniu polecenia `terraform apply`. Użytkownicy zawierających zmienne danych wyjściowych mogą to uznać za użyteczne (teraz po wdrożeniu serwera WWW od razu znasz adres IP, który powinien być przetestowany). Polecenie `terraform output` pozwala na wyświetlenie listy wszystkich zmiennych danych wyjściowych bez wprowadzania jakichkolwiek zmian.

```
$ terraform output
```

```
public_ip = 54.174.13.5
```

Natomiast wydanie polecenia `terraform output <NAZWA>` spowoduje wyświetlenie wartości podanej zmiennej danych wyjściowych:

```
$ terraform output public_ip
```

```
54.174.13.5
```

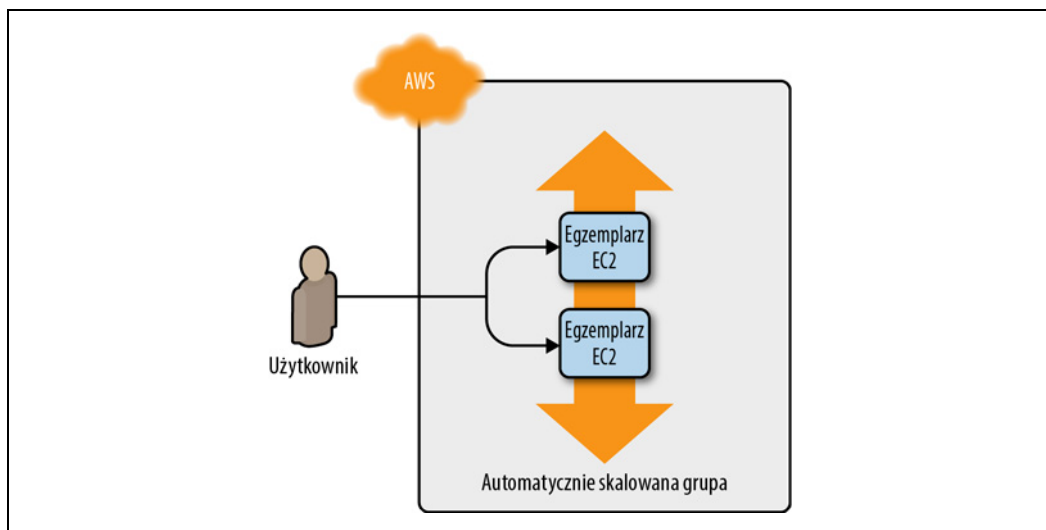
To przydaje się szczególnie w skryptach. Przykładowo można utworzyć skrypt wdrożenia wykonujący polecenie `terraform apply` w celu wdrożenia serwera WWW, `terraform output public_ip` w celu ustalenia jego publicznego adresu IP i `curl` wraz z tym adresem IP, aby potwierdzić poprawność wdrożenia.

Zmienne danych wejściowych i wyjściowych mają również znaczenie podczas tworzenia konfigurowalnej infrastruktury wielokrotnego użycia jako kodu; więcej informacji na ten temat znajdziesz w rozdziale 4.

Wdrażanie klastra serwerów WWW

Uruchomienie pojedynczego serwera to dobry początek, choć w rzeczywistości pojedynczy serwer jest jednocześnie jednym punktem awarii. Jeżeli ten serwer ulegnie awarii lub zostanie przeciążony zbyt dużym ruchem sieciowym, użytkownicy nie będą mieli dostępu do udostępnianej przez ten serwer witryny internetowej. Rozwiązaniem jest uruchomienie klastra serwerów, routing ruchu sieciowego serwerów, które uległy awarii, oraz dostosowywanie wielkości klastra (w górę lub w dół) na podstawie ruchu sieciowego⁹.

Ręczne zarządzanie takim klastrem oznacza dużo pracy. Na szczęście można to zlecić AWS przez wykorzystanie automatycznie skalowanej grupy (ang. *auto scaling group*, ASG), jak pokazałem na rysunku 2.9. ASG zajmuje się wieloma zadaniami, m.in. uruchamianiem klastra egzemplarzy EC2, monitorowaniem stanu poszczególnych egzemplarzy, zastępowaniem egzemplarzy, które uległy awarii, oraz dostosowywaniem wielkości klastra na podstawie jego obciążenia.



Rysunek 2.9. Zamiast pojedynczego serwera uruchom klastery serwerów WWW za pomocą automatycznie skalowanej grupy

Pierwszym krokiem podczas tworzenia ASG jest przygotowanie *konfiguracji startowej* określającej sposób skonfigurowania poszczególnych egzemplarzy EC2 w ASG. Zasób `aws_launch_configuration` używa praktycznie tych samych parametrów znanych z zasobu `aws_instance` (dwa z tych parametrów

⁹ Szczegółowe informacje na temat tworzenia wysoko dostępnych i skalowanych systemów w AWS znajdziesz w artykule opublikowanym na stronie <https://www.airpair.com/aws/posts/building-a-scalable-web-app-on-amazon-web-services-p1>.

mają inne nazwy — zamiast `ami` mamy `image_id` i zamiast `vpc_security_group` mamy `security_groups`), więc wystarczy zamienić poprzednią nazwę parametru na nową:

```
resource "aws_launch_configuration" "example" {
  image_id      = "ami-0c55b159cbfaffe1f0"
  instance_type = "t2.micro"
  security_groups = [aws_security_group.instance.id]

  user_data = <<-EOF
    #!/bin/bash
    echo "Witaj, świecie" > index.html
    nohup busybox httpd -f -p ${var.server_port} &
  EOF
}
```

Teraz można już utworzyć ASG za pomocą zasobu `aws_autoscaling_group`.

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name

  min_size = 2
  max_size = 10

  tag {
    key      = "Name"
    value    = "terraform-asg-example"
    propagate_at_launch = true
  }
}
```

W omawianym przykładzie ASG będzie się składać z od 2 do 10 egzemplarzy EC2 (domyślnie 2 po uruchomieniu), z których każdy jest oznaczony tagiem `terraform-asg-example`. Zwróć uwagę na użycie przez ASG odwołania do nazwy konfiguracji startowej. To prowadzi do pewnego problemu: konfiguracja startowa jest niemodyfikowana, więc zmiana jakiegokolwiek jej parametru spowoduje, że Terraform spróbuje ją zastąpić nową. Standardowo podczas zastępowania zasobu Terraform najpierw usuwa start zasób, a następnie tworzy jego zamiennik. Jednak obecnie to ASG ma odwołanie do starego zasobu, dlatego Terraform nie może go usunąć.

Rozwiązaniem tego problemu jest ustawienie *lifecycle*. Każdy zasób Terraform obsługuje to ustawienie określające sposób tworzenia, uaktualniania i (lub) usuwania zasobu. Szczególnie użyteczną wartością ustawienia *lifecycle* jest `create_before_destroy`. Jeżeli przypiszesz wartość `true` właściwości `create_before_destroy`, Terraform odwróci kolejność zastępowania zasobów, czyli najpierw utworzy zamiennik (uaktualni przy tym odwołania, aby zamiast do starego prowadziły do jego zamiennika), a dopiero później usunie stary zasób. Do `aws_launch_configuration` dodaj blok *lifecycle*, jak pokazałem w kolejnym fragmencie kodu.

```
resource "aws_launch_configuration" "example" {
  image_id      = "ami-0c55b159cbfaffe1f0"
  instance_type = "t2.micro"
  security_groups = [aws_security_group.instance.id]

  user_data = <<-EOF
    #!/bin/bash
    echo "Witaj, świecie" > index.html
    nohup busybox httpd -f -p ${var.server_port} &
  EOF
}
```



```
# Wymagane podczas używania konfiguracji startowej wraz z automatycznie skalowaną grupą.
# https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
lifecycle {
  create_before_destroy = true
}
}
```

Jest jeszcze jeden parametr, który koniecznie należy dodać do ASG, aby przedstawione rozwiązanie mogło działać: `subnet_ids`. Ten parametr określa, do której podsieci VPC powinny zostać wdrożone egzemplarze EC2 (więcej informacji na temat podsieci znajdziesz we wcześniejszej części rozdziału, a dokładnie w uwadze „Bezpieczeństwo sieci”). Każda podsieć istnieje w odizolowanej strefie dostępności AWS (tzn. odizolowanym centrum danych), więc wdrożenie egzemplarzy w wielu podsięciach gwarantuje działanie usługi nawet w przypadku awarii któregoś z centrum danych. Wprawdzie można przygotować na stałe zdefiniowaną listę podsieci, ale takie rozwiązanie jest trudne w obsłudze i nieprzenośne. Dlatego też lepsze podejście polega na wykorzystaniu *źródeł danych* w celu pobrania listy podsieci z konta AWS.

Źródło danych przedstawia fragment informacji tylko do odczytu pobieranych od dostawcy (tutaj AWS) w trakcie każdego uruchomienia Terraform. Dodanie źródła danych do konfiguracji Terraform nie powoduje utworzenia niczego nowego, to jest sposób na wykonanie zapytania do API dostawcy i pobrania danych, które następnie zostają udostępnione pozostałemu kodowi Terraform. Każdy dostawca Terraform udostępnia wiele różnych źródeł danych. Przykładowo dostawca AWS oferuje źródła danych pozwalające na pobieranie informacji z danych VPC, podsieci, identyfikatorów obrazów AMI, zakresów adresów IP, tożsamości bieżącego użytkownika itd.

Składnia używania źródła danych jest bardzo podobna do składni zasobu:

```
data "<DOSTAWCA>_<TYP>" "<NAZWA>" {
  [KONFIGURACJA ...]
}
```

gdzie DOSTAWCA to nazwa dostawcy (np. `aws`), TYP określa typ źródła danych przeznaczonego do użycia (np. `vpc`), NAZWA to identyfikator używany w kodzie Terraform w celu odwołania się do tego źródła danych, a KONFIGURACJA składa się z jednego lub więcej argumentów charakterystycznych dla tego źródła danych. Dla przykładu spójrz na sposób użycia źródła danych `aws_vpc` w celu wyszukania danych dla domyślnego VPC (więcej informacji na temat domyślnego VPC znajdziesz we wcześniejszej części rozdziału):

```
data "aws_vpc" "default" {
  default = true
}
```

Dzięki źródłom danych przekazywane argumenty zwykle mają postać filtrów wskazujących źródłu danych, jakie informacje są szukane. W przypadku `aws_vpc` jedynym filtrem jest `default = true`, który nakazuje Terraform wyszukanie domyślnego VPC w Twoim koncie AWS.

Aby pobrać dane ze źródła danych, konieczne jest wykorzystanie następującej składni odwołania atrybutu:

```
data.<DOSTAWCA>_<TYP>.<NAZWA>.<ATRYBUT>
```

Przykładowo w celu pobrania identyfikatora VPC ze źródła danych `aws_vpc` należy użyć przedstawionego tutaj polecenia:

```
data.aws_vpc.default.id
```

Istnieje możliwość połączenia tego polecenia z innym źródłem danych, np. `aws_subnet_ids`, i wyszukania podsieci w danym VPC:

```
data "aws_subnet_ids" "default" {
  vpc_id = data.aws_vpc.default.id
}
```

Na końcu można pobrać identyfikatory podsieci ze źródła danych `aws_subnet_ids` i nakazać ASG użycie tych podsieci za pomocą (dość dziwnie nazwanego) argumentu `vpc_zone_identifier`.

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids

  min_size = 2
  max_size = 10

  tag {
    key           = "Name"
    value         = "terraform-asg-example"
    propagate_at_launch = true
  }
}
```

Wdrożenie mechanizmu równoważenia obciążenia

Na tym etapie można wdrożyć ASG, choć powstanie wówczas mały problem: istnieje kilka serwerów, każdy z własnym adresem IP, a użytkownikowi końcowemu zwykle chcemy przekazać tylko pojedynczy adres IP. Jednym ze sposobów rozwiązania tego problemu jest wdrożenie *mechanizmu równoważenia obciążenia* w celu rozłożenia ruchu sieciowego między serwery i przekazanie wszystkim użytkownikom adresu IP (w rzeczywistości to nazwa DNS) mechanizmu równoważenia obciążenia. Utworzenie takiego mechanizmu, który będzie skalowalny, to dość dużo pracy. Także w tym przypadku można to zlecić AWS, wykorzystując usługę ELB (ang. *elastic load balancer*), jak pokazałem na rysunku 2.10.

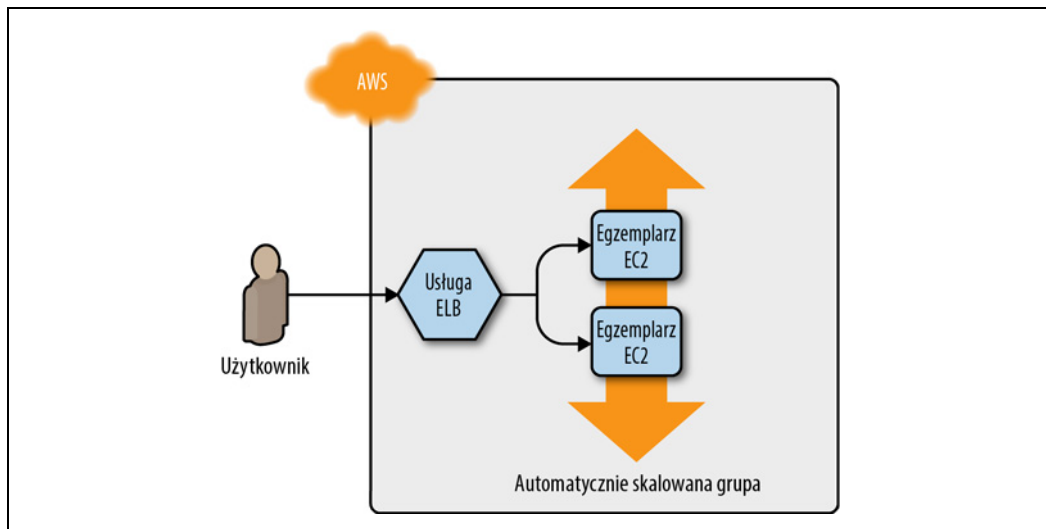
Dostawca AWS oferuje trzy odmienne rodzaje mechanizmu równoważenia obciążenia:

ALB (ang. *application load balancer*)

Ten typ jest najlepiej dopasowany do równoważenia ruchu sieciowego HTTP i HTTPS. Działa na warstwie aplikacji (warstwa 7) modelu OSI.

NLB (ang. *network load balancer*)

Ten typ jest najlepiej dopasowany do równoważenia ruchu sieciowego TCP, UDP i TLS. Może być znacznie szybciej niż ALB skalowany w górę i w dół w reakcji na obciążenie (typ NLB został zaprojektowany do obsługi maksymalnie dziesiątek milionów żądań na sekundę). Działa na warstwie transportowej (warstwa 4) modelu OSI.



Rysunek 2.10. Użycie usługi ELB do rozłożenia ruchu sieciowego w ASG

CLB (ang. *classic load balancer*)

Ten typ jest uznawany za przestarzały i był dostępny przed wprowadzeniem ALB i NLB. Potrafi obsługiwać ruch sieciowy HTTP, HTTPS, TCP i TLS, choć oferuje mniej funkcji niż ALB lub NLB. Działa na warstwach aplikacji (warstwa 7) i transportowej (warstwa 4) modelu OSI.

Obecnie większość aplikacji powinna używać ALB lub NLB. Skoro w omawianym przykładzie prosty serwer WWW działa wraz ruchem sieciowym HTTP i nie wymaga wyjątkowej wydajności, najlepszym typem będzie ALB.

Jak możesz zobaczyć na rysunku 2.11, typ ALB składa się z kilku portów:

Porty nasłuchujące

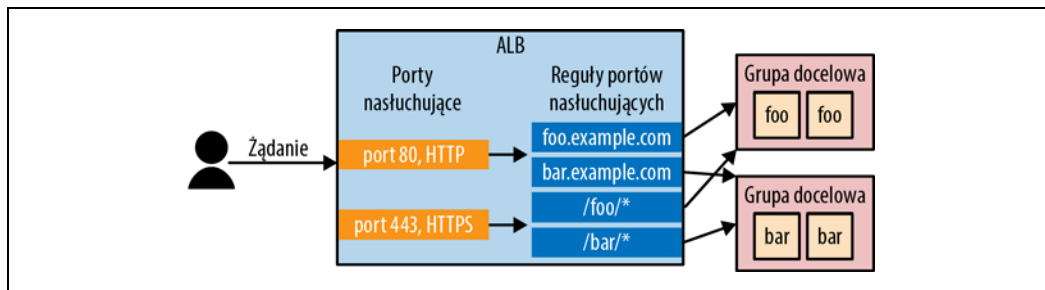
Nasłuchuje na określonym porcie (np. 80) i protokole (np. HTTP).

Reguły portów nasłuchujących

Pobiera żądanie kierowane do portu nasłuchującego i to — o dopasowanej ścieżce dostępu (np. `/foo` lub `/bar`) albo nazwie hosta (np. `foo.example.com` lub `bar.example.com`) — przekazuje do określonej grupy docelowej.

Grupa docelowa

Jeden lub więcej serwerów otrzymujących żądania z mechanizmu równoważenia obciążenia. Grupa docelowa sprawdza również stan serwera i przekazuje żądanie do wyłącznie prawidłowo działającego serwera.



Rysunek 2.11. Ogólny schemat działania mechanizmu równoważenia obciążenia typu ALB

Pierwszym krokiem jest utworzenie samego typu ALB za pomocą zasobu `aws_lb`.

```
resource "aws_lb" "example" {
  name           = "terraform-asg-example"
  load_balancer_type = "application"
  subnets       = data.aws_subnet_ids.default.ids
}
```

Zwróć uwagę na to, że parametr `subnets` konfiguruje mechanizm równoważenia obciążenia do użycia wszystkich podsieci w domyślnym VPC, co odbywa się za pomocą źródła danych `aws_subnet_ids`¹⁰. Mechanizm równoważenia obciążenia w AWS nie składa się z jednego, ale z kilku serwerów, które mogą działać w różnych podsieciach (a tym samym w różnych centrach danych). AWS automatycznie skaluje liczbę serwerów mechanizmu równoważenia obciążenia na podstawie wielkości ruchu sieciowego i stanu serwerów, więc standardowo masz zapewnioną skalowalność i wysoką dostępność.

Następnym krokiem jest zdefiniowanie za pomocą zasobu `aws_lb_listener` komponentu nasłuchującego ALB.

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = 80
  protocol          = "HTTP"
  # Domyślnie zwracana jest prosta strona błędu 404.
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: nie znaleziono strony"
      status_code  = 404
    }
  }
}
```

Ten komponent konfiguruje ALB nasłuchujący na domyślnym porcie HTTP, czyli 80, używający protokołu HTTP i zwracający prostą stronę błędu 404 jako odpowiedź domyślną dla żądań niedopasowanych do żadnej reguły portu nasłuchującego.

¹⁰ W celu zachowania prostoty omawianego przykładu egzemplarze EC2 i ALB będą uruchomione w tych samym podsieciach. W środowisku produkcyjnym prawdopodobnie będziesz je uruchamiać w oddzielnych podsieciach — egzemplarze EC2 w prywatnych (aby nie były dostępne bezpośrednio z internetu publicznego), a ALB w publicznych (aby użytkownicy mieli do nich bezpośredni dostęp).

Warto w tym miejscu dodać, że domyślnie wszystkie zasoby AWS, co dotyczy również typu ALB, nie zezwalają na ruch przychodzący i wychodzący, więc konieczne jest utworzenie nowej grupy bezpieczeństwa przeznaczonej specjalnie dla ALB. Ta grupa bezpieczeństwa powinna zezwalać na żądania przychodzące na porcie 80, aby poprzez HTTP zapewnić dostęp do mechanizmu równoważenia obciążenia, oraz na żądania wychodzące na wszystkich portach, aby mechanizm równoważenia obciążenia mógł sprawdzać stan serwerów.

```
resource "aws_security_group" "alb" {
  name = "terraform-example-alb"

  # Zezwolenie na przychodzące żądania HTTP.
  ingress {
    from_port = 80
    to_port   = 80
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Zezwolenie na wszystkie żądania wychodzące.
  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Konieczne jest nakazanie zasobowi `aws_lb` użycie tej nowej grupy bezpieczeństwa. W tym celu można wykorzystać argument `security_groups`.

```
resource "aws_lb" "example" {
  name                = "terraform-asg-example"
  load_balancer_type = "application"
  subnets            = data.aws_subnet_ids.default.ids
  security_groups     = [aws_security_group.alb.id]
}
```

Kolejnym krokiem jest utworzenie grupy docelowej dla ASG przy użyciu do tego zasobu `aws_lb_target_group`.

```
resource "aws_lb_target_group" "asg" {
  name     = "terraform-asg-example"
  port     = var.server_port
  protocol = "HTTP"
  vpc_id   = data.aws_vpc.default.id

  health_check {
    path           = "/"
    protocol       = "HTTP"
    matcher        = "200"
    interval       = 15
    timeout        = 3
    healthy_threshold = 2
    unhealthy_threshold = 2
  }
}
```

Zauważ, że ta grupa docelowa będzie sprawdzać stan egzemplarzy EC2 poprzez okresowe wykonywanie żądania HTTP do każdego egzemplarza. Dany egzemplarz zostanie uznany za „sprawny” tylko, jeśli udzieli odpowiedzi dopasowanej do skonfigurowanej wartości określonej mianem *matcher* (to może być np. kod stanu 200 OK). Jeżeli egzemplarz nie udzieli oczekiwanej odpowiedzi, np. na skutek przeciążenia lub awarii, zostanie oznaczony jako „niesprawny” i grupa docelowa automatycznie przestanie przekazywać do niego ruch sieciowy, aby w ten sposób minimalizować negatywny wpływ tego serwera na użytkowników.

Skąd grupa docelowa wie, do których egzemplarzy EC2 mają być wykonywane żądania? Za pomocą zasobu `aws_lb_target_group_attachment` możliwe jest dołączenie statycznej listy egzemplarzy EC2 do grupy docelowej. Jednak w przypadku ASG egzemplarze mogą być uruchamiane i zatrzymywane w każdej chwili, więc taka lista statyczna się nie sprawdzi. Zamiast tego należy skorzystać z zalet doskonałej integracji ASG z ALB. Wróć do zasobu `aws_autoscaling_group` i jego argumentowi `target_group_arns` przypisz wartość wskazującą nową grupę docelową.

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids

  target_group_arns = [aws_lb_target_group.asg.arn]
  health_check_type = "ELB"

  min_size = 2
  max_size = 10

  tag {
    key      = "Name"
    value    = "terraform-asg-example"
    propagate_at_launch = true
  }
}
```

Powinieneś również uaktualnić atrybut `health_check_type` przez przypisanie mu wartości `ELB`. Wartością domyślną tego atrybutu jest `EC2`, która powoduje ograniczenie do jedynie minimalnego sprawdzenia stanu egzemplarza — egzemplarz zostanie uznany za niesprawny tylko wtedy, gdy nadzorca AWS poinformuje o pełnym zamknięciu lub niedostępności maszyny wirtualnej. Wartość `ELB` zapewnia znacznie solidniejsze sprawdzenie, ponieważ nakazuje ASG wykorzystanie wyniku operacji sprawdzenia grupy docelowej do ustalenia, czy egzemplarz jest sprawny. Jeżeli grupa docelowa zgłosi niesprawność egzemplarza, zostanie on automatycznie zastąpiony nowym. W ten sposób egzemplarze będą zastępowane nie tylko po ich całkowitym wyłączeniu, ale również np. po zaprzestaniu udzielania odpowiedzi na żądania z powodu braku wolnej pamięci lub po awarii procesu o znaczeniu krytycznym.

Teraz można już połączyć ze sobą wszystkie elementy układanki poprzez utworzenie za pomocą zasobu `aws_lb_listener_rule` reguł portów nasłuchujących.

```
resource "aws_lb_listener_rule" "asg" {
  listener_arn = aws_lb_listener.http.arn
  priority     = 100

  condition {
    field = "path-pattern"
```

```

    values = ["*"]
  }

  action {
    type = "forward"
    target_group_arn = aws_lb_target_group.asg.arn
  }
}

```

W tym kodzie została dodana reguła, która powoduje przekazanie dopasowanych do dowolnej ścieżki dostępu żądań do grupy docelowej zawierającej ASG.

Ostatnim zadaniem przed wdrożeniem mechanizmu równoważenia obciążenia jest zastąpienie starych danych wyjściowych zmiennej `public_ip` pojedynczego egzemplarza EC2, który był używany wcześniej, danymi wyjściowymi zawierającymi nazwę DNS dla typu ALB.

```

output "alb_dns_name" {
  value = aws_lb.example.dns_name
  description = "Nazwa domeny mechanizmu równoważenia obciążenia"
}

```

Wyдай polecenie `terraform apply` i dokładnie zapoznaj się z wygenerowanymi danymi wyjściowymi. Powinieneś zobaczyć, że pojedynczy egzemplarz EC2 został usunięty, natomiast w jego miejsce Terraform utworzy konfigurację startową, ASG, ALB i grupę bezpieczeństwa. Jeżeli plan wygląda dobrze, wpisz **yes** i naciśnij klawisz *Enter*. Po zakończeniu działania polecenia `terraform apply` powinieneś zobaczyć następujące dane wyjściowe dla `alb_dns_name`:

```

Outputs:
alb_dns_name = terraform-asg-example-123.us-east-2.elb.amazonaws.com

```

Skopiuj ten adres URL. Uruchomienie egzemplarzy i określenie ich jako sprawnych przez ALB zajmie kilka minut. W tym czasie możesz sprawdzić wdrożenie. Przejdź do sekcji ASG w konsoli EC2 (<https://console.aws.amazon.com>) — powinieneś zobaczyć utworzoną grupę ASG, jak pokazałem na rysunku 2.12.

Po przejściu do karty *Instances* zobaczysz uruchamianie dwóch egzemplarzy EC2, jak pokazałem na rysunku 2.13.

Po przejściu do karty *Load Balancers* będziesz mógł podejrzeć typ ALB mechanizmu równoważenia obciążenia, jak pokazałem na rysunku 2.14.

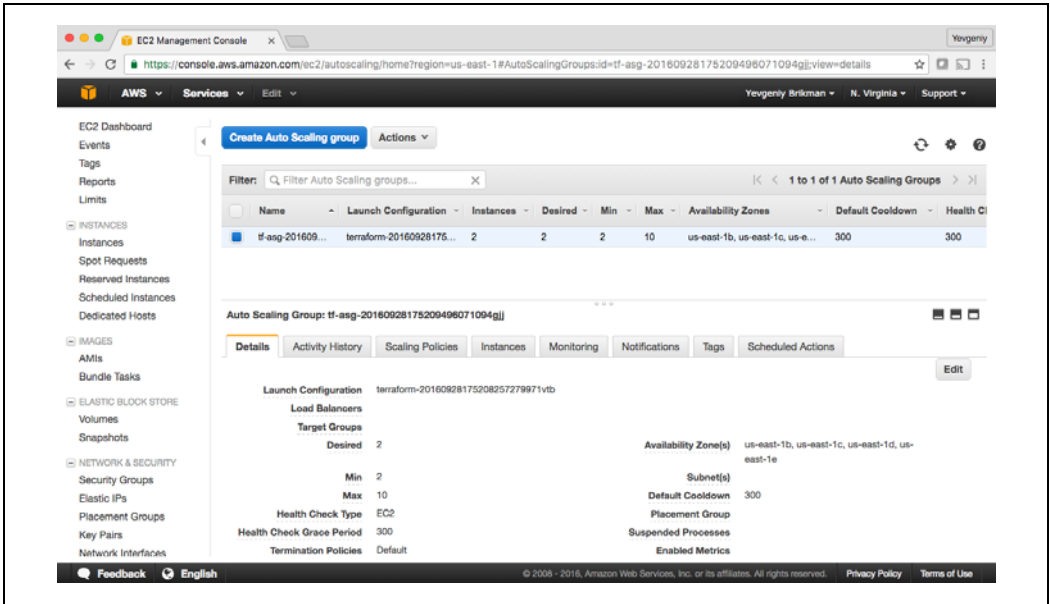
Wreszcie, po przejściu do karty *Target Groups*, otrzymasz informacje o grupie docelowej, jak pokazałem na rysunku 2.15.

Kliknięcie grupy docelowej, a następnie przejście do karty *Targets* w dolnej części ekranu powoduje wyświetlenie informacji o rejestracji egzemplarzy w grupie docelowej i przeprowadzanych operacjach sprawdzania ich stanu. Poczekał chwilę, aż dla obu w sekcji *Status* zostanie wyświetlony komunikat *healthy* (z ang. sprawny). To zwykle wymaga 1 – 2 minut. Gdy widzisz te komunikaty, możesz sprawdzić skopiowane wcześniej dane wyjściowe `alb_dns_name`.

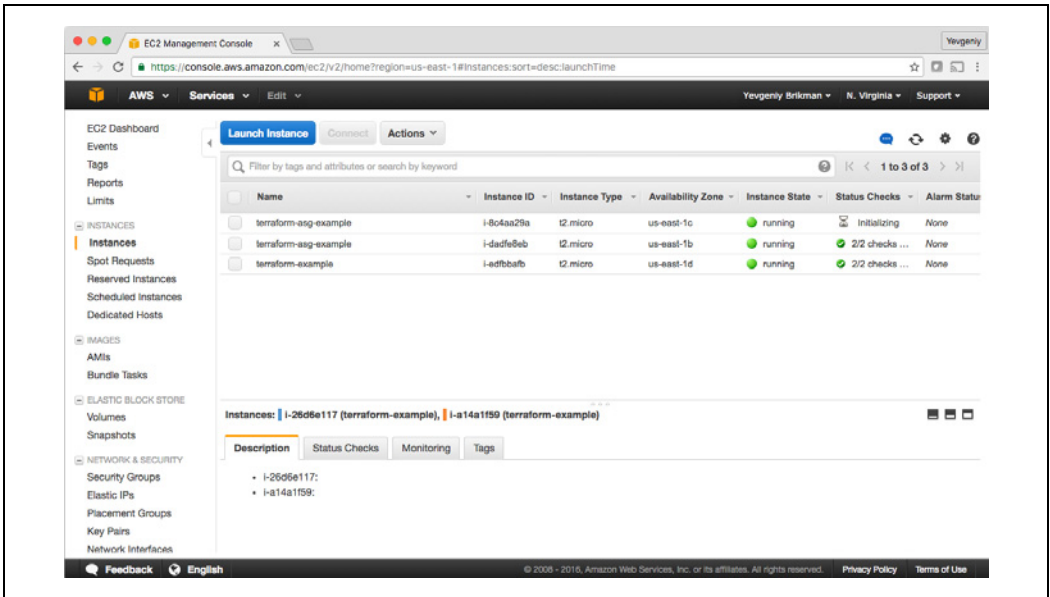
```

$ curl http://<alb_dns_name>
Witaj, świecie

```

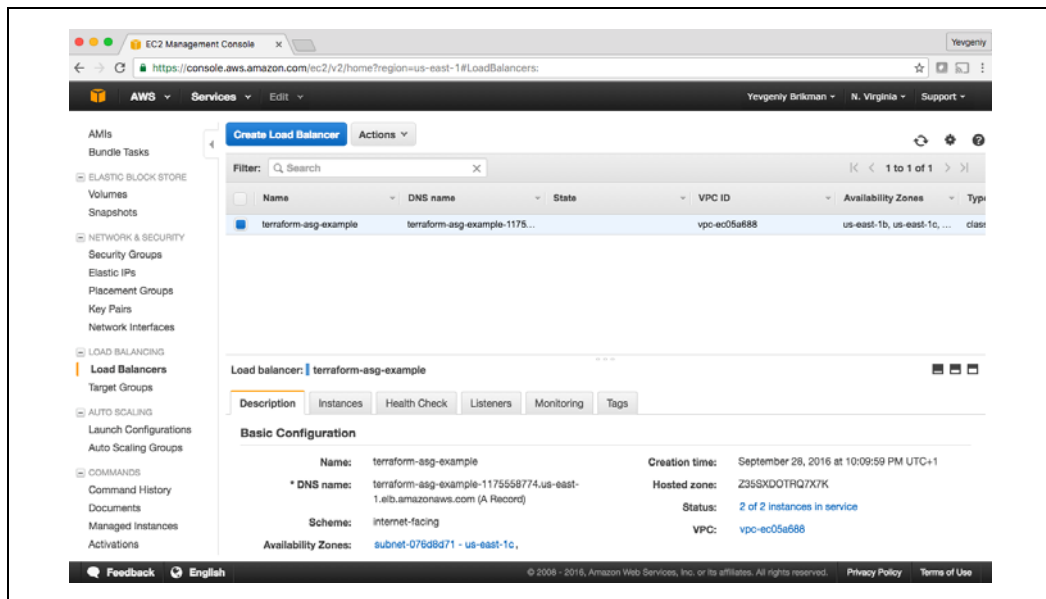


Rysunek 2.12. Grupa ASG w konsoli AWS EC2

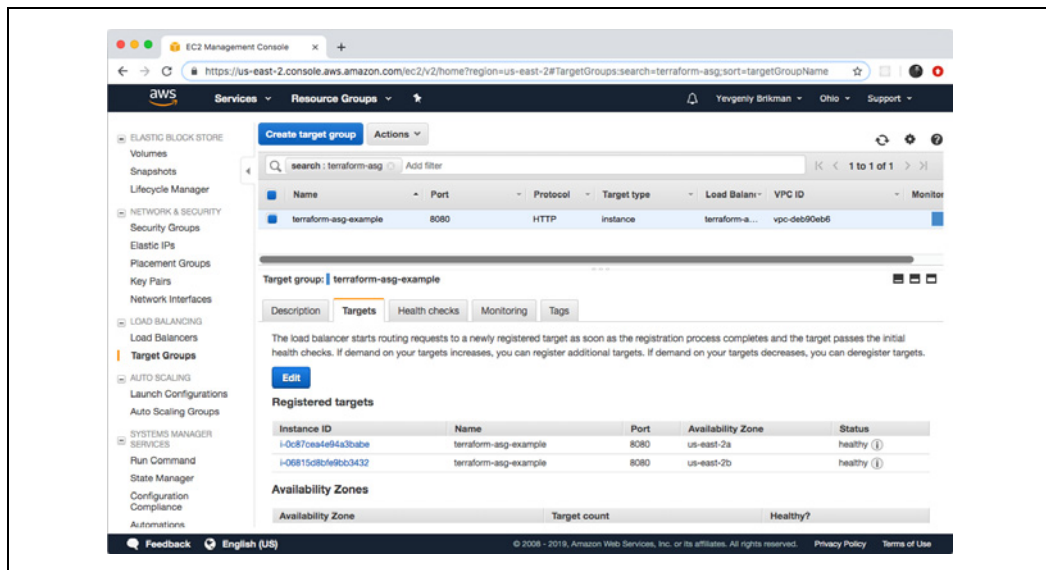


Rysunek 2.13. Uruchamianie egzemplarzy EC2 w grupie ASG

Sukces! Mechanizm równoważenia obciążenia przekazuje ruch do egzemplarzy EC2. Każde żądanie pod podany adres powoduje wybór innego egzemplarza przeznaczonego do obsługi danego żądania. W ten sposób masz wdrożony w pełni działający klaster serwerów WWW.



Rysunek 2.14. Mechanizm równoważenia obciążenia w przykładowej aplikacji



Rysunek 2.15. Informacje o grupie docelowej

Na tym etapie można zobaczyć, jak kłaster reaguje przez uruchamianie nowych i zatrzymywanie starych egzemplarzy. Dla przykładu przejdź do karty *Instances* i zamknij jeden z egzemplarzy przez jego zaznaczenie, kliknięcie przycisku *Actions* na górze, a następnie ustawienie stanu (*Instance State*) jako *Terminate*. Kontynuuj testowanie adresu URL ALB, a zobaczysz, że wciąż otrzymujesz odpowiedź 200 OK dla każdego żądania, nawet po zatrzymaniu egzemplarza. To jest możliwe, ponieważ ALB automatycznie wykrywa zatrzymanie egzemplarza i przestaje kierować do niego ruch sieciowy.

Grupa ASG automatycznie wykrywa dostępność mniej niż dwóch uruchomionych egzemplarzy i automatycznie uruchamia nowy, aby zastąpić zatrzymany (samodzielna naprawa). Zmianę wielkości grupy ASG możesz obserwować przez dodanie parametru `desired_capacity` do kodu Terraform i ponowne wydanie polecenia `terraform apply`.

Porządkowanie

Po zakończeniu eksperymentów z Terraform, na końcu tego rozdziału lub kolejnych, dobrym rozwiązaniem jest usunięcie wszystkich utworzonych wcześniej zasobów, aby za nie nie płacić. Ponieważ Terraform śledzi utworzone zasoby, operacja porządkowania jest prosta i sprowadza się do wydania polecenia `terraform destroy`.

```
$ terraform destroy
```

```
(...)
```

Terraform will perform the following actions:

```
# Zasób aws_autoscaling_group.example zostanie usunięty.
- resource "aws_autoscaling_group" "example" {
  (...)
}

# Zasób aws_launch_configuration.example zostanie usunięty.
- resource "aws_launch_configuration" "example" {
  (...)
}

# Zasób aws_lb.example zostanie usunięty.
- resource "aws_lb" "example" {
  (...)
}

(...)
```

Plan: 0 to add, 0 to change, 8 to destroy.

Do you really want to destroy all resources?

Terraform will destroy all your managed infrastructure, as shown above.

There is no undo. Only 'yes' will be accepted to confirm.

Enter a value:

Operacja jest przeprowadzana bez żadnego ostrzeżenia, że z polecenia `terraform destroy` powinienś korzystać rzadko (o ile w ogóle) w środowisku produkcyjnym. Nie ma czegoś takiego jak cofnięcie wymienionego polecenia, więc Terraform daje ostatnią możliwość na przejrzenie wprowadzonych zmian — wyświetla listę zasobów przeznaczonych do usunięcia i prosi o potwierdzenie tej operacji. Jeżeli wszystko wygląda dobrze, wpisz **yes** i naciśnij klawisz *Enter*. Terraform wygeneruje drzewo zależności i usunie wszystkie zasoby we właściwej kolejności, w maksymalnym stopniu korzystając ze współbieżności podczas tej operacji. Po 1 – 2 minutach Twoje konto w AWS ponownie będzie czyste.

Dodam, że w kolejnych rozdziałach będziemy kontynuować pracę nad tym przykładem, więc nie usuwaj utworzonego tutaj kodu Terraform! Zdecydowanie możesz wydać polecenie `terraform destroy` dla faktycznie wdrożonych zasobów. W końcu piękno infrastruktury jako kodu polega na tym, że wszystkie informacje o zasobach znajdują się w kodzie, więc można je odtworzyć w dowolnej chwili za pomocą polecenia `terraform apply`. Zachęcam, aby ostatnio wprowadzone zmiany przekazać do repozytorium Git, co pozwoli na śledzenie historii całej infrastruktury.

Podsumowanie

W ten sposób zdobyłeś podstawową wiedzę z zakresu pracy z Terraform. Język deklaracyjny pozwala na bardzo łatwe i dokładne opisanie infrastruktury przeznaczonej do utworzenia. Polecenie `terraform plan` daje możliwość weryfikacji wprowadzanych zmian i wychwycenia błędów jeszcze przed wdrożeniem rozwiązania. Zmienne, odwołania i zależności umożliwiają pozbycie się powielonego kodu i zapewniają wysoką konfigurowalność.

Jednak dotychczas poznałeś zaledwie załóżek wiedzy o Terraform. Z rozdziału 3. dowiesz się, jak Terraform śledzi tworzoną infrastrukturę, co ma duży wpływ na sposób tworzenia kodu Terraform. Z kolei w rozdziale 4. zobaczysz, jak za pomocą modułów Terraform można przygotować infrastrukturę wielokrotnego użycia.

Zarządzanie informacjami o stanie Terraform

Gdy w rozdziale 2. używałeś Terraform do tworzenia i uaktualniania zasobów, być może zauważyłeś, że podczas każdego wykonania poleceń `terraform plan` i `terraform apply` Terraform potrafi odnaleźć wcześniej utworzone zasoby i odpowiednio je uaktualnić. Skąd Terraform wie, którymi zasobami ma zarządzać? W ramach konta AWS można tworzyć różnego rodzaju infrastrukturę, wdrażać odmienne mechanizmy (część ręcznie, część za pomocą Terraform, inne zaś poprzez CLI), więc skąd Terraform wie, którą infrastrukturą ma zarządzać?

W tym rozdziale zobaczysz, jak Terraform monitoruje stan infrastruktury, oraz poznasz jej wpływ na układ, izolację i blokowanie projektu w Terraform. Oto lista tematów, które zostaną poruszone:

- Czym są informacje o stanie Terraform?
- Współdzielony magazyn danych dla plików informacji o stanie.
- Ograniczenia backendu Terraform.
- Izolowanie plików stanu:
 - izolowanie za pomocą przestrzeni roboczych,
 - izolowanie za pomocą układu plików.
- Źródło danych `terraform_remote_state`.



Przykładowe fragmenty kodu

Przypominam, że wszystkie przykładowe fragmenty kodu znajdują się w repozytorium GitHub pod adresem <https://github.com/brikis98/terraform-up-and-running-code>.

Czym są informacje o stanie Terraform?

Za każdym razem, gdy uruchamiasz Terraform, to narzędzie rejestruje w pliku *informacji o stanie Terraform* informacje o utworzonej infrastrukturze. Domyślnie w przypadku uruchomienia Terraform w katalogu `/foo/bar` Terraform utworzy plik `/foo/bar/terraform.tfstate`. Ten plik zawiera dane w niestandardowym formacie JSON przedstawiające mapowanie zasobów Terraform w plikach konfiguracyjnych na reprezentację tych zasobów w rzeczywistym projekcie. Przykładowo przyjmujemy założenie o istnieniu w konfiguracji Terraform następującego fragmentu kodu:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafa1f0"
  instance_type = "t2.micro"
}
```

Po wykonaniu polecenia `terraform apply` plik *terraform.tfstate* będzie zawierał m.in. przedstawione tutaj dane (w celu zachowania zwięzłości przedstawiłem tylko fragment tego pliku).

```
{
  "version": 4,
  "terraform_version": "0.12.0",
  "serial": 1,
  "lineage": "1f2087f9-4b3c-1b66-65db-8b78faafc6fb",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "example",
      "provider": "provider.aws",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0c55b159cbfafa1f0",
            "availability_zone": "us-east-2c",
            "id": "i-00d689a0acc43af0f",
            "instance_state": "running",
            "instance_type": "t2.micro",
            "(...)": "(truncated)"
          }
        }
      ]
    }
  ]
}
```

W omawianym przykładzie w wyniku użycia formatu JSON Terraform wie, że zasób o typie `aws_instance` i nazwie `example` odpowiada egzemplarzowi EC2 w Twoim koncie AWS o identyfikatorze `i-00d689a0acc43af0f`. Każde uruchomienie Terraform powoduje pobranie z AWS najnowszych informacji o stanie egzemplarza EC2 i porównanie ich z konfiguracją Terraform w celu określenia zmian koniecznych do wprowadzenia. Innymi słowy, dane wyjściowe polecenia `terraform plan` przedstawiają różnice między kodem istniejącym w komputerze i infrastrukturą wdrożoną w rzeczywistości odkryte na podstawie identyfikatorów pochodzących z pliku stanu.

Jeżeli używasz Terraform w projekcie prywatnym, przechowywanie informacji o stanie w pojedynczym pliku *terraform.tfstate* znajdującym się w katalogu lokalnym sprawdza się doskonale. Jeśli natomiast chcesz używać Terraform w zespole pracującym nad rzeczywistym produktem, pojawi się kilka problemów:



Plik stanu to prywatne API

Format pliku stanu to prywatne API zmieniające się wraz z każdym wydaniem i przeznaczone jedynie do wewnętrznego użycia w Terraform. Nigdy nie powinieneś ręcznie edytować plików stanu Terraform ani tworzyć kodu bezpośrednio odczytującego zawartość tych plików.

Jeżeli z jakiegokolwiek powodu wystąpi konieczność przeprowadzenia operacji na plikach stanu — co powinno zdarzać się wyjątkowo rzadko — skorzystaj z poleceń `terraform import` i `terraform state` (ich przykłady przedstawię w rozdziale 5.).

Współdzielony magazyn danych dla plików informacji o stanie

Aby zachować możliwość wykorzystania Terraform do uaktualnienia infrastruktury, każdy członek zespołu musi mieć dostęp do tych samych plików informacji o stanie Terraform. To oznacza konieczność przechowywania tych plików we współdzielonym magazynie danych.

Nakładanie blokad na pliki informacji o stanie

Gdy dane będą współdzielone, natychmiast pojawi się nowy problem: nakładanie blokad. Bez mechanizmu nakładania blokad, jeśli dwóch członków zespołu będzie jednocześnie wykonywać polecenie Terraform, może dojść do stanu wyścigu, ponieważ wiele procesów Terraform będzie przeprowadzało uaktualnienia plików informacji o stanie, co doprowadzi do konfliktów, utraty danych i uszkodzenia pliku informacji o stanie.

Izolowanie plików informacji o stanie

Podczas wprowadzania zmian w infrastrukturze najlepszą praktyką jest izolowanie poszczególnych środowisk. Przykładowo podczas wprowadzania zmian w środowisku testowym lub roboczym chcesz mieć pewność, że nie ma możliwości przypadkowego uszkodzenia środowiska produkcyjnego. W jaki sposób można odizolować wprowadzane zmiany, skoro cała infrastruktura jest zdefiniowana w tym samym pliku stanu Terraform?

W kolejnych sekcjach omówię poszczególne problemy i pokażę, jak można je rozwiązać.

Współdzielony magazyn danych dla plików informacji o stanie

Najczęściej stosowana technika pozwalająca wielu członkom zespołu na uzyskanie dostępu do zbioru wspólnych plików polega na umieszczeniu ich w systemie kontroli wersji (np. Git). Wprowadzić kod Terraform zdecydowanie powinieneś umieszczać w systemie kontroli wersji, ale przechowywanie w nim plików informacji o stanie Terraform to naprawdę *zły pomysł* z wymienionych tutaj powodów:

Ręcznie popełniony błąd

Bardzo łatwo zapomnieć o pobraniu najnowszych wersji plików z systemu kontroli wersji przed rozpoczęciem pracy z Terraform lub o przekazaniu najnowszych zmian po ich wprowadzeniu. To tylko kwestia czasu, gdy ktoś z członków zespołu wykona polecenie wraz z nieaktualnymi plikami informacji o stanie, co w efekcie doprowadzi do przypadkowego wycofania nowszych zmian lub powielenia poprzedniego wdrożenia.

Nakładanie blokad

Większość systemów kontroli wersji nie oferuje żadnego mechanizmu nakładania blokad, który uniemożliwiłby dwóm członkom zespołu jednoczesne wydanie polecenia `terraform apply` wraz z tym samym plikiem stanu.

Informacje wrażliwe

Wszystkie informacje o stanie Terraform są przechowywane w plikach zwykłego tekstu. To stanowi problem, ponieważ niektóre zasoby Terraform muszą przechowywać dane wrażliwe. Przykładowo, jeśli korzystasz z zasobu `aws_db_instance` do utworzenia bazy danych, nazwa użytkownika i hasło do tej bazy danych zostaną zapisane w pliku informacji o stanie, będącym w formacie zwykłego tekstu. Przechowywanie *gdzieś* poza komputerem, np. w systemie kontroli wersji, plików zawierających takie dane jest złym pomysłem. W chwili gdy piszę te słowa (maj 2019), w społeczności Terraform istnieje otwarte zgłoszenie (<https://github.com/hashicorp/terraform/issues/516>), choć dostępne są również pewne sensowne rozwiązania, które wkrótce przedstawię.

Zamiast systemu kontroli wersji najlepszym sposobem na zarządzanie współdzielonym magazynem danych dla plików informacji o stanie jest wykorzystanie wbudowanej w Terraform obsługi zdalnych backendów. Wspomniany *backend* Terraform określa, jak Terraform wczytuje i przechowuje informacje o stanie. Domyślny backend, używany przez cały czas, to *backend lokalny* przechowujący na dysku lokalnym plik informacji o stanie. Z kolei *backend zdalny* pozwala na przechowywanie w zdalnym, współdzielonym dysku pliku informacji o stanie. Obsługiwanych jest wiele backendów zdalnych, m.in. Amazon S3, Azure Storage, Google Cloud Storage, HashiCorp Terraform Cloud, Terraform Pro i Terraform Enterprise.

Zdalny backend pozwala na rozwiązanie wszystkich trzech wymienionych wcześniej błędów:

Ręcznie popełniony błąd

Po skonfigurowaniu zdalnego backendu Terraform będzie automatycznie wczytywać z niego plik informacji o stanie po każdym wydaniu polecenia `terraform plan` i `terraform apply`, a także po wykonaniu tego drugiego polecenia będzie automatycznie umieszczać w tym backendzie plik informacji o stanie, więc nie ma możliwości popełnienia błędu.

Nakładanie blokad

Większość zdalnych backendów natywnie zapewnia obsługę nakładania blokad. Aby wykonać polecenie `terraform apply`, Terraform automatycznie nałoży blokadę. Jeżeli ktokolwiek inny wyda polecenie `terraform apply`, blokada będzie już nałożona i trzeba będzie poczekać na jej zwolnienie. Istnieje możliwość wydania polecenia `terraform apply` wraz z parametrem `-lock-timeout=<CZAS>` w celu wskazania Terraform, że ma zwolnić blokadę po upływie podanego czasu (np. `-lock-timeout=10m` oznacza zwolnienie blokady po 10 minutach).

Informacje wrażliwe

Większość zdalnych backendów natywnie obsługuje szyfrowanie podczas przekazywania i przechowywania pliku informacji o stanie. Co więcej, te backendy zwykle zapewniają możliwość konfigurowania uprawnień dostępu (np. za pomocą polityki IAM w Amazon S3), więc zachowu-

jesz kontrolę nad tym, kto będzie miał dostęp do plików informacji o stanie, a także jakie dane wrażliwe mogą się w nim znajdować. Wprawdzie lepiej byłoby, gdyby narzędzie Terraform natywnie zapewniało obsługę szyfrowania danych w pliku informacji o stanie, ale wspomniane backendy i tak eliminują większość obaw związanych z zachowaniem bezpieczeństwa, uwzględniając to, że plik informacji o stanie nie jest przechowywany w postaci zwykłego tekstu gdzieś na dysku.

Jeżeli używasz Terraform wraz z AWS, Amazon S3 (Simple Storage Service) — czyli zarządzany przez Amazona magazyn danych — jest zwykle najlepszym rozwiązaniem do użycia w charakterze zdalnego backendu. Istnieje ku temu kilka powodów:

- To jest usługa zarządzana, więc w celu jej użycia nie musisz wdrażać dodatkowej infrastruktury ani nią zarządzać.
- Została zaprojektowana z myślą o 99,999999999% trwałości i 99,99% dostępności, więc nie musisz się przejmować niebezpieczeństwem utraty danych lub przestoju¹.
- Zapewnia obsługę szyfrowania, które zmniejsza obawy związane z przechowywaniem danych wrażliwych w plikach informacji o stanie. Każdy członek zespołu posiadający dostęp do tzw. kubełka S3 (ang. *bucket*) będzie mógł zobaczyć pliki informacji o stanie w postaci niezaszyfrowanej, więc to nadal jest rozwiązanie częściowe. Jednak dane są przynajmniej przechowywane w postaci zaszyfrowanej (Amazon S3 obsługuje szyfrowanie po stronie serwera za pomocą AES-256) oraz w trakcie transportu (Terraform używa SSL do odczytywania i zapisywania danych w Amazon S3).
- Nakładanie blokad jest obsługiwane dzięki DynamoDB (więcej informacji na ten temat przedstawię za chwilę).
- Obsługiwane jest *wersjonowanie*, więc każda wersja pliku informacji o stanie jest przechowywana i można przywrócić starszą wersję, gdy coś pójdzie źle.
- Rozwiązanie jest niedrogie, w większości przypadków wystarczające są możliwości oferowane bezpłatnie przez Amazon S3².

Aby włączyć obsługę zdalnego magazynu danych Amazon S3, w pierwszym kroku utwórz wspomniany wcześniej kubełek S3. W nowym katalogu utwórz plik o nazwie *main.tf* (to powinien być inny katalog niż ten używany do przechowywania konfiguracji w rozdziale 2.) i na jego początku zdefiniuj AWS jako dostawcę.

```
provider "aws" {  
  region = "us-east-2"  
}
```

Teraz utwórz kubełek S3 za pomocą zasobu `aws_s3_bucket`.

```
resource "aws_s3_bucket" "terraform_state" {  
  bucket = "terraform-up-and-running-state"
```

¹ Więcej informacji na temat gwarancji udzielanych przez Amazon S3 znajdziesz na stronie <https://aws.amazon.com/s3/features/>.

² Cennik usługi Amazon S3 znajduje się na stronie <https://aws.amazon.com/s3/pricing/>.

```

# Uniemożliwienie przypadkowego usunięcia tego kubelka S3.
lifecycle {
  prevent_destroy = true
}

# Włączenie wersjonowania, co pozwala na zachowanie pełnej
# historii informacji o stanie.
versioning {
  enabled = true
}

# Domyślne włączenie szyfrowania po stronie serwera.
server_side_encryption_configuration {
  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}
}

```

Ten fragment kodu definiuje cztery argumenty:

`bucket`

To jest nazwa kubelka S3. Zwróć uwagę na to, że nazwy kubelków S3 muszą być *globalnie* unikatowe wśród wszystkich klientów AWS. Dlatego też musisz zmienić wartość parametru `bucket` z `terraform-up-and-running-state` (została wykorzystana przeze mnie) na inną³. Upewnij się o zapisaniu tej nazwy i używanego regionu AWS, ponieważ te dane będą nam potrzebne nieco później.

`prevent_destroy`

To jest drugie ustawienie cyklu życiowego, z którym się spotykasz (pierwsze to poznane w rozdziale 2. `create_before_destroy`). Gdy w zasobie przypiszesz wartość `true` ustawieniu `prevent_destroy`, każda próba usunięcia tego zasobu (np. za pomocą polecenia `terraform destroy`) spowoduje zakończenie działania Terraform wraz z błędem. To jest dobre rozwiązanie uniemożliwiające przypadkowe usunięcie ważnego zasobu, takiego jak kubelek S3, przechowującego wszystkie informacje o stanie Terraform. Oczywiście, jeśli faktycznie będziesz chciał usunąć dany zasób, wystarczy, że umieszysz znak komentarza na początku omawianego ustawienia.

`versioning`

Ten blok pozwala na włączenie wersjonowania w kubelku S3, aby każde uaktualnienie pliku w kubelku powodowało utworzenie nowej wersji tego pliku. To pozwala na sprawdzanie starszych wersji pliku i przywracanie ich w dowolnym momencie.

`server_side_encryption_configuration`

Ten blok włącza szyfrowanie po stronie serwera domyślnie dla wszystkich danych zapisywanych we wskazanym kubelku S3. Dzięki temu masz pewność, że pliki informacji o stanie i wszelkie

³ Więcej informacji o kubekach S3 znajdziesz na stronie <https://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html>.

znajdujące się w nich dane wrażliwe zawsze będą zaszyfrowane na dysku podczas ich przechowywania w S3.

Kolejnym krokiem jest utworzenie tabeli DynamoDB przeznaczonej do użycia podczas nakładania blokad. DynamoDB to opracowany przez Amazon magazyn danych typu klucz-wartość. Obsługuje ściśle spójne operacje odczytu i zapis warunkowy, czyli komponenty niezbędne do opracowania rozproszonego systemu nakładania blokad. Co więcej, to rozwiązanie jest całkowicie zarządzane, więc nie potrzebuje żadnej dodatkowej infrastruktury do działania, a ponadto jest niedrogie — w większości przypadków wystarczające jest korzystanie z bezpłatnego planu Terraform⁴.

Aby skorzystać z DynamoDB w trakcie nakładania blokad podczas pracy z Terraform, konieczne jest utworzenie tabeli DynamoDB wraz z kluczem podstawowym LockID (nazwa musi być zapisana *dokładnie* w podanej postaci). Taką tabelę można utworzyć za pomocą zasobu `aws_dynamodb_table`.

```
resource "aws_dynamodb_table" "terraform_locks" {
  name             = "terraform-up-and-running-locks"
  billing_mode     = "PAY_PER_REQUEST"
  hash_key         = "LockID"
  attribute {
    name = "LockID"
    type = "S"
  }
}
```

Wyдай polecenie `terraform init` w celu pobrania kodu dostawcy, a następnie `terraform apply` w celu wdrożenia projektu. (Uwaga: wdrożenie tego kodu wymaga, aby użytkownik IAM miał uprawnienia do tworzenia kubełków S3 i tabel DynamoDB, co dokładnie przedstawiłem w rozdziale 2.). Wprawdzie po przeprowadzeniu wdrożenia otrzymasz kubełek S3 i tabelę DynamoDB, ale informacje o stanie Terraform nadal będą przechowywane lokalnie. W celu skonfigurowania Terraform do umieszczenia w kubełku S3 informacji o stanie (wraz z zastosowaniem szyfrowania i nakładaniem blokad) konieczne jest dodanie konfiguracji backendu do kodu Terraform. To jest konfiguracja przeznaczona dla samego narzędzia Terraform i dlatego znajduje się w bloku `terraform` o następującej składni:

```
terraform {
  backend "<NAZWA_BACKENDU>" {
    [KONFIGURACJA...]
  }
}
```

gdzie `NAZWA_BACKENDU` to nazwa backendu przeznaczonego do użycia (np. `s3`), a `KONFIGURACJA` składa się z jednego lub więcej argumentów związanych z tym backendem (np. nazwa używanego kubełka S3). Spójrz na przykładową konfigurację backendu dla kubełka S3.

```
terraform {
  backend "s3" {
    # Zmień tę nazwę na nazwę Twojego kubełka!
    bucket = "terraform-up-and-running-state"
    key    = "global/s3/terraform.tfstate"
    region = "us-east-2"
  }
}
```

⁴ Cennik usługi DynamoDB znajduje się na stronie <https://aws.amazon.com/dynamodb/pricing/>.

```

# Zmień tę nazwę na nazwę Twojej tabeli DynamoDB!
dynamodb_table = "terraform-up-and-running-locks"
encrypt         = true
}
}

```

Przeanalizujemy poszczególne ustawienia:

bucket

Nazwa kubełka S3 przeznaczonego do użycia. Upewnij się o zastąpieniu jej nazwą utworzonego wcześniej kubełka S3.

key

To jest ścieżka dostępu do pliku kubełka S3, który ma zostać zapisany i zawierać informacje o stanie Terraform. Z dalszej części rozdziału dowiesz się, dlaczego w omawianym fragmencie kodu została użyta wartość `global/s3/terraform.tfstate`.

region

To jest region AWS, w którym zostanie umieszczony kubełek. Upewnij się o umieszczeniu w tym miejscu wartości przedstawiającej region, w którym wcześniej utworzyłeś kubełek.

dynamodb_table

Tabela DynamoDB używana do obsługi nakładania blokad. Upewnij się o podaniu w tym miejscu nazwy utworzonej wcześniej tabeli DynamoDB.

encrypt

Przypisanie temu ustawieniu wartości `true` gwarantuje, że informacje o stanie Terraform będą przechowywane na dysku w postaci zaszyfrowanej. Wcześniej zostało włączone domyślne szyfrowanie samego kubełka S3, więc to jest druga warstwa zapewniająca, że dane zawsze pozostaną zaszyfrowane.

Aby nakazać Terraform przechowywanie w kubełku S3 pliku informacji o stanie, należy ponownie skorzystać z polecenia `terraform init`. To polecenie nie tylko pobiera kod dostawcy, ale również konfiguruje backend Terraform (inny przykład użycia przedstawię w dalszej części książki). Warto w tym miejscu przypomnieć, że polecenie `terraform init` jest powtarzalne, więc jego wielokrotne wykonywanie jest bezpieczne.

```
$ terraform init
```

```

Initializing the backend...
Acquiring state lock. This may take a few moments...
Do you want to copy existing state to the new backend?
Pre-existing state was found while migrating the previous "local" backend
to the newly configured "s3" backend. No existing state was found in the
newly configured "s3" backend. Do you want to copy this state to the new
"s3" backend? Enter "yes" to copy and "no" to start with an empty state.

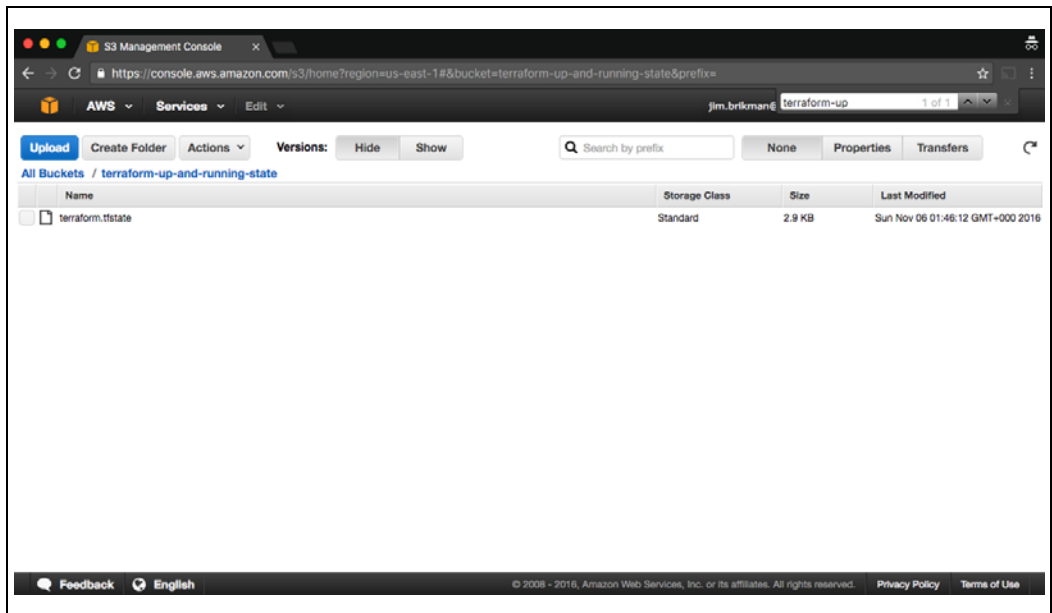
```

```
Enter a value:
```

Terraform automatycznie wykryje lokalne przechowywanie pliku zawierającego informacje o stanie i zapyta, czy skopiować go do nowego kubełka S3. Jeżeli wpiszesz **yes** i naciśniesz klawisz *Enter*, powinieneś zobaczyć następujące dane wyjściowe:

Successfully configured the backend "s3"! Terraform will automatically use this backend unless the backend configuration changes.

Po wykonaniu tego polecenia informacje o stanie Terraform będą przechowywane w utworzonym wcześniej kubelku S3. Możesz to sprawdzić po przejściu w przeglądarce WWW do konsoli zarządzania S3 (<https://aws.amazon.com/console/>) i kliknięciu kubłka. Otrzymasz wynik podobny do pokazanego na rysunku 3.1.



Rysunek 3.1. Informacje o stanie Terraform są przechowywane w kubelku S3

Po włączeniu tego backendu, przed wykonaniem polecenia Terraform automatycznie pobierze z utworzonego kubłka S3 najnowsze informacje o stanie, natomiast po wykonaniu polecenia umieści w kubelku S3 najnowsze informacje o stanie Terraform. Jeżeli chcesz to zobaczyć w akcji, dodaj do kodu przedstawione tutaj zmienne danych wyjściowych.

```
output "s3_bucket_arn" {
  value     = aws_s3_bucket.terraform_state.arn
  description = "Wartość ARN kubełka S3"
}

output "dynamodb_table_name" {
  value     = aws_dynamodb_table.terraform_locks.name
  description = "Nazwa tabeli DynamoDB"
}
```

Te zmienne spowodują wyświetlenie wartości ARN (ang. *amazon resource name*) kubłka S3 i nazwy tabeli DynamoDB. Wydad polecenie `terraform apply`, aby się o tym przekonać.

```
$ terraform apply
```

```
Acquiring state lock. This may take a few moments...
```

```
aws_dynamodb_table.terraform_locks: Refreshing state...
```

```
aws_s3_bucket.terraform_state: Refreshing state...
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

```
Releasing state lock. This may take a few moments...
```

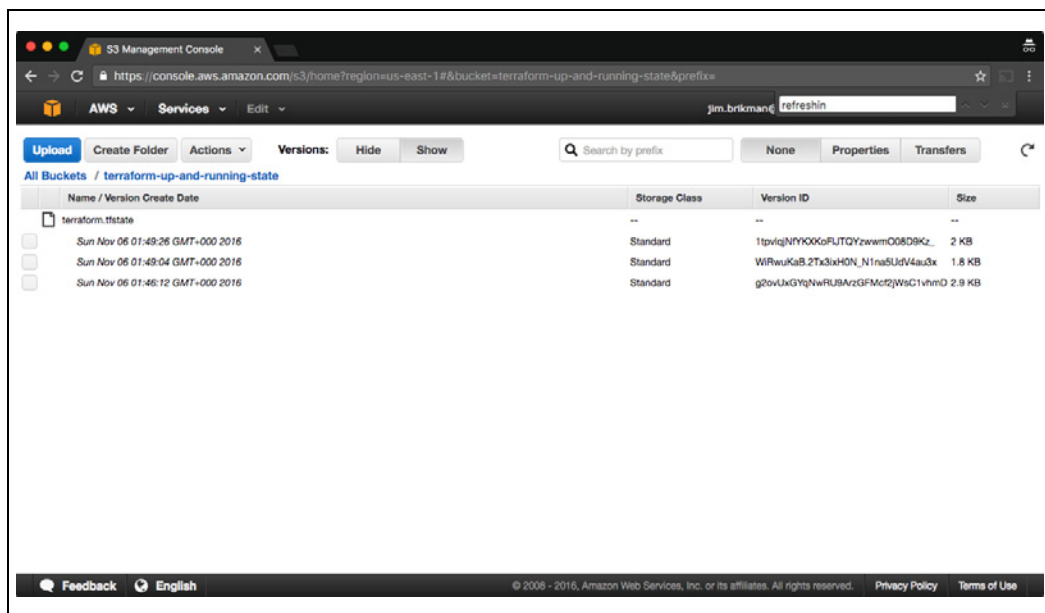
```
Outputs:
```

```
dynamodb_table_name = terraform-up-and-running-locks
```

```
s3_bucket_arn = arn:aws:s3:::terraform-up-and-running-state
```

(Zwróć uwagę, jak Terraform nakłada blokadę przed wykonaniem polecenia `terraform apply` i jak ją zwalnia po zakończeniu jego wykonywania).

Teraz ponownie przejdź do konsoli S3 (<https://aws.amazon.com/console/>), odśwież stronę i kliknij przycisk *Show* wyświetlony w sekcji *Versions*. Powinieneś zobaczyć, że w kubelku S3 jest kilka wersji pliku `terraform.tfstate`, jak pokazałem na rysunku 3.2.



Rysunek 3.2. Wiele wersji przechowywanego w kubelku S3 pliku informacji o stanie Terraform

Masz potwierdzenie, że Terraform automatycznie pobiera i przekazuje dane do kubelka S3. Kubłek S3 przechowuje każdą awersję pliku informacji o stanie Terraform, co może okazać się użyteczne podczas procesu usuwania błędów, a także w trakcie przywracania wcześniejszej wersji pliku, gdy coś pójdzie źle.

Ograniczenia backendu Terraform

Backend Terraform ma pewne ograniczenia i wady, których istnienia trzeba być świadomym. Pierwszym ograniczeniem jest tzw. problem jajka i kury — konieczność użycia Terraform do utworzenia

kubelka S3 przeznaczonego do przechowywania informacji o stanie Terraform. Aby takie rozwiązanie działało, konieczne jest skorzystanie z procesu składającego się z dwóch etapów:

1. Przygotuj kod Terraform przeznaczony do utworzenia kubelka S3 i tabeli DynamoDB, a następnie przeprowadź wdrożenie tego kodu w backendzie lokalnym.
2. Powróć do kodu Terraform, dodaj konfigurację backendu (blok backend) zdalnego pozwalającą na wykorzystanie utworzonego w poprzednim kroku kubelka S3 i tabeli DynamoDB, a następnie wydaj polecenie `terraform init`, aby przechowywane lokalnie informacje o stanie skopiować do S3.

Jeżeli kiedykolwiek będziesz chciał usunąć kubek S3 i tabelę DynamoDB, skorzystaj z odwrotnego procesu, który również składa się z dwóch etapów:

1. Przejdź do kodu Terraform, usuń konfigurację backendu (blok backend) i ponownie wydaj polecenie `terraform init`, aby informacje o stanie Terraform z powrotem skopiować na dysk lokalny.
2. Wydaj polecenie `terraform destroy` w celu usunięcia kubelka S3 i tabeli DynamoDB.

Ten dwuetapowy proces jest nieco dziwny, ale dobrą wiadomością jest możliwość współdzielenia pojedynczego kubelka S3 i tabeli DynamoDB w całym kodzie Terraform. Dlatego też prawdopodobnie omawiany proces trzeba będzie przeprowadzić tylko jednokrotnie (lub po jednym razie dla każdego konta AWS, o ile masz ich więcej). Gdy kubek S3 już istnieje, w pozostałej części kodu Terraform można odwoływać się do konfiguracji backendu bez konieczności podejmowania jakichkolwiek kroków dodatkowych.

Drugie ograniczenie jest znacznie bardziej bolesne: blok backend w Terraform nie pozwala na stosowanie jakichkolwiek zmiennych lub odwołań, więc przedstawiony tutaj fragment kodu nie działa.

```
# Ten kod NIE działa. Niedozwolone jest używanie zmiennych w konfiguracji backendu.
terraform {
  backend "s3" {
    bucket      = var.bucket
    region      = var.region
    dynamodb_table = var.dynamodb_table
    key         = "example/terraform.tfstate"
    encrypt     = true
  }
}
```

To oznacza konieczność ręcznego skopiowania i wklejenia nazwy kubelka S3, regionu, tabeli DynamoDB itd. do każdego modułu Terraform. (Więcej informacji na temat modułów Terraform znajdziesz w rozdziałach 4. i 6., natomiast teraz wystarczy wiedzieć, że moduły pozwalają na organizowanie i wielokrotne używanie kodu Terraform. W rzeczywistych projektach kod Terraform zwykle składa się z wielu małych modułów). Co gorsza, należy zachować ostrożność, aby *nie* skopiować i nie wkleić wartości `key`, a upewnić się o unikatowości wartości `key` dla każdego wdrażanego modułu Terraform. W przeciwnym razie może dojść do przypadkowego nadpisania informacji o stanie innego modułu. Konieczność wykonania wielu operacji kopiowania i wklejania *oraz* ręcznego przeprowadzania licznych zmian wiąże się z podatnością na błędy, zwłaszcza jeśli trzeba wdrażać i zarządzać wieloma modułami Terraform w różnych środowiskach.

W chwili powstawania książki (maj 2019) jedynym dostępnym rozwiązaniem jest wykorzystanie zalet *konfiguracji częściowej*, w której w kodzie Terraform pomija się określone parametry z konfiguracji backendu i zamiast tego przekazuje się je za pomocą argumentów powłoki `-backend-config` w trakcie wywołania `terraform init`. Przykładowo istnieje możliwość wyodrębnienia powtarzających się argumentów *backendu*, takich jak `bucket` i `region`, a następnie umieszczenia ich w oddzielnym pliku o nazwie np. *backend.hcl*.

```
# backend.hcl
bucket      = "terraform-up-and-running-state"
region      = "us-east-2"
dynamodb_table = "terraform-up-and-running-locks"
encrypt      = true
```

W kodzie Terraform pozostał jedynie parametr `key`, ponieważ dla każdego modułu musi być definiowana inna wartość `key`.

```
# To jest konfiguracja częściowa. Pozostałe ustawienia (np. bucket, region) zostaną przekazane
# z pliku za pomocą argumentów -backend-config dla polecenia 'terraform init'.
terraform {
  backend "s3" {
    key = "example/terraform.tfstate"
  }
}
```

Aby zastosować wszystkie ustawienia w przypadku konfiguracji częściowej, należy wydać polecenie `terraform init` wraz z argumentem `-backend-config`.

```
$ terraform init -backend-config=backend.hcl
```

Terraform złączy konfigurację częściową w *backend.hcl* wraz z konfiguracją częściową w kodzie Terraform, aby w ten sposób przygotować pełną konfigurację używaną przez moduł.

Inną możliwością jest wykorzystanie Terragrunt, czyli narzędzia typu open source próbującego wypełnić kilka luk istniejących w Terraform. To narzędzie może pomóc w przygotowaniu konfiguracji backendu z zachowaniem reguły DRY — co odbywa się przez zdefiniowanie w jednym pliku wszystkich podstawowych ustawień backendu (nazwa kubelka, regionu, tabeli DynamoDB), a następnie automatyczne przypisanie argumentowi `key` względnej ścieżki dostępu do katalogu modułu. Przykłady zastosowania Terragrunt przedstawię w rozdziale 8.

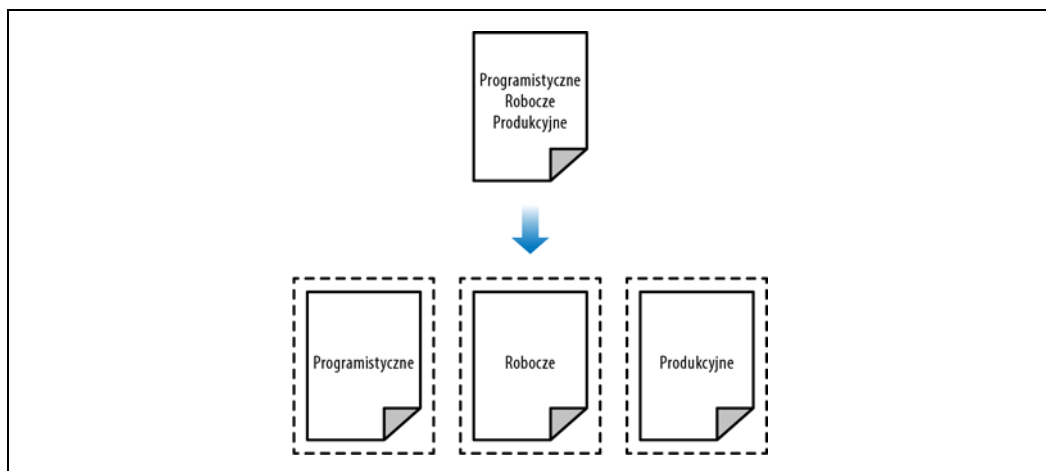
Izolowanie plików informacji o stanie

Po zaimplementowaniu zdalnego backendu i nakładania blokad współpraca między wieloma programistami nie stanowi dłużej problemu. Jednak pozostała jeszcze jedna kwestia do rozwiązania: izolacja. Gdy zaczniesz korzystać z Terraform, być może za kuszące uznasz definiowanie całej infrastruktury w pojedynczym pliku Terraform lub pojedynczym zestawie plików Terraform w katalogu. Problem z takim podejściem polega na tym, że wówczas wszystkie informacje o stanie Terraform są również przechowywane w pojedynczym pliku i błąd może wszystko zniszczyć.

Przykładowo, jeśli próbujesz wdrożyć nową wersję aplikacji, która znajduje się w środowisku roboczym, możesz uszkodzić aplikację znajdującą się w środowisku produkcyjnym. Jeszcze gorszą sytuacją

będzie uszkodzenie pliku zawierającego wszystkie informacje o stanie, co może nastąpić ze względu na brak nałożonej blokady lub ze względu na rzadki błąd Terraform. W takim przypadku cała infrastruktura we wszystkich środowiskach jest uszkodzona⁵.

Największą zaletą stosowania oddzielnych środowisk jest to, że są one od siebie odizolowane. Dlatego też, jeśli wszystkimi środowiskami zarządzasz za pomocą pojedynczego zestawu konfiguracji Terraform, łamiesz tę izolację. Podobnie jak statek zawiera grodzie chroniące poszczególne jego fragmenty przed zalaniem w przypadku przecieku w jednym z przedziałów, tak samo wspomniane „grodzie” zostały wbudowane w narzędzie Terraform, jak możesz zobaczyć na rysunku 3.3.



Rysunek 3.3. Tzw. grodzie zastosowane w projekcie narzędzia Terraform

Jak to zostało pokazane na rysunku 3.3, zamiast zdefiniowania wszystkich środowisk w pojedynczym zbiorze konfiguracji Terraform (na górze rysunku) każde środowisko zostało zdefiniowane oddzielnie (na dole rysunku), więc problem w jednym środowisku pozostaje całkowicie odizolowany od pozostałych środowisk. Mamy dwa sposoby na izolowanie plików informacji o stanie:

Izolacja za pomocą przestrzeni roboczych

To rozwiązanie jest użyteczne w przypadku szybkich, odizolowanych testów w tej samej konfiguracji.

Izolacja za pomocą układu plików

To rozwiązanie jest użyteczne w przypadku produkcji, gdy konieczne jest zachowanie ścisłej separacji między środowiskami.

Oba sposoby omówię dokładniej w kolejnych punktach.

⁵ Na stronie <https://charity.wtf/2016/03/30/terraform-vpc-and-why-you-want-a-tfstate-file-per-env/> znajdziesz przykład wyjaśniający, co może się stać, jeśli nie izolujesz informacji o stanie Terraform.

Isolacja za pomocą przestrzeni roboczych

Przestrzenie robocze Terraform pozwalają na przechowywanie informacji o stanie wielu oddzielnych i nazwanych przestrzeni roboczych. Na początku istnieje tylko jedna przestrzeń robocza o nazwie *default* i jeśli nigdy wyraźnie nie wskazałeś innej, ta domyślna jest używana przez cały czas. W celu utworzenia nowej przestrzeni roboczej lub przełączania się między nimi należy skorzystać z polecenia `terraform workspace`. Poeksperymentujemy teraz z przestrzeniami roboczymi podczas pracy nad kodem Terraform odpowiedzialnym za wdrożenie pojedynczego egzemplarza EC2.

```
resource "aws_instance" "example" {
  ami          = "ami-0c55b159cbf1f0"
  instance_type = "t2.micro"
}
```

Skonfiguruj backend dla tego egzemplarza za pomocą utworzonego wcześniej w rozdziale kubelka S3 i tabeli DynamoDB, przy czym jako wartość atrybutu `key` podaj `workspaces-example/terraform.tfstate`.

```
terraform {
  backend "s3" {
    # Tę wartość zastąp nazwą używanego kubelka!
    bucket      = "terraform-up-and-running-state"
    key         = "workspaces-example/terraform.tfstate"
    region      = "us-east-2"
    # Tę wartość zastąp nazwą używanej tabeli DynamoDB!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt      = true
  }
}
```

Wydadź polecenia `terraform init` i `terraform apply`, aby wdrożyć przygotowany kod.

```
$ terraform init
```

```
Initializing the backend...
```

```
Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
```

```
Initializing provider plugins...
```

```
(...)
```

```
Terraform has been successfully initialized!
```

```
$ terraform apply
```

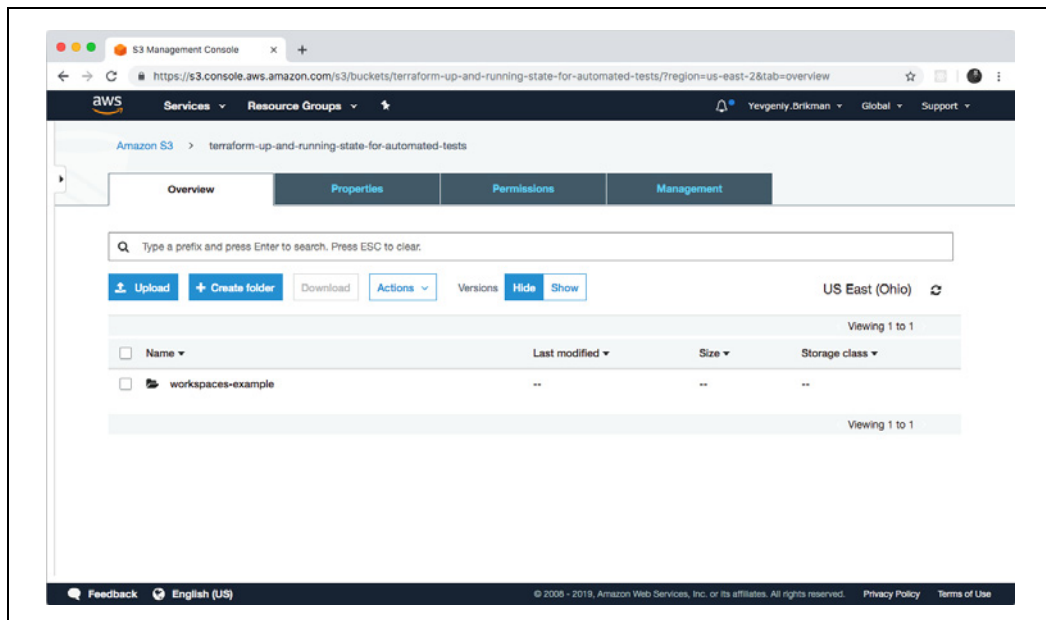
```
(...)
```

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

W przypadku tego wdrożenia informacje o stanie są przechowywane w domyślnej przestrzeni roboczej. Można to potwierdzić przez wydanie polecenia `terraform workspace show`, którego dane wyjściowe wyświetlają aktualnie używaną przestrzeń roboczą.

```
$ terraform workspace show
default
```

Domyślna przestrzeń robocza przechowuje informacje o stanie w położeniu wskazanym za pomocą atrybutu `key` w konfiguracji. Jak pokazałem na rysunku 3.4, jeśli zajrzysz do kubelka S3, plik `terraform.tfstate` znajdziesz w katalogu `workspaces-example`.



Rysunek 3.4. Kubek S3 po umieszczeniu informacji o stanie w domyślnej przestrzeni roboczej

Przystępujemy do utworzenia nowej przestrzeni roboczej o nazwie `example1` za pomocą polecenia `terraform workspace new`.

```
$ terraform workspace new example1
```

```
Created and switched to workspace "example1"!
```

You're now on a new, empty workspace. Workspaces isolate their state, so if you run `"terraform plan"` Terraform will not see any existing state for this configuration.

Zwróć uwagę na to, co się stanie po wykonaniu polecenia `terraform plan`:

```
$ terraform plan
```

Terraform will perform the following actions:

```
# Nastąpi utworzenia zasobu aws_instance.example.
+ resource "aws_instance" "example" {
+   ami               = "ami-0c55b159cbf1f0"
+   instance_type     = "t2.micro"
+   (...)
}
```

Plan: 1 to add, 0 to change, 0 to destroy.

Terraform chce utworzyć zupełnie od początku nowy egzemplarz EC2. To wynika z odizolowania plików informacji o stanie w poszczególnych przestrzeniach roboczych. Skoro teraz znajdujesz się w przestrzeni roboczej `example1`, Terraform nie używa informacji o stanie przechowywanych w domyślnej przestrzeni roboczej i dlatego nie widzi, że egzemplarz EC2 został już utworzony.

Spróbuj wydać polecenie `terraform apply` w celu wdrożenia drugiego egzemplarza EC2 dla nowej przestrzeni roboczej.

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Powtórzmy to ćwiczenie i utwórzmy kolejną przestrzeń roboczą o nazwie `example2`.

```
$ terraform workspace new example2
```

```
Created and switched to workspace "example2"!
```

```
You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

Ponownie wydaj polecenie `terraform apply`, aby wdrożyć trzeci egzemplarz EC3.

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

W ten sposób masz trzy dostępne przestrzenie robocze, co możesz potwierdzić przez wydanie polecenia `terraform workspace list`:

```
$ terraform workspace list
```

```
default
```

```
example1
```

```
* example2
```

Do przechodzenia między poszczególnymi przestrzeniami roboczymi służy polecenie `terraform workspace select`.

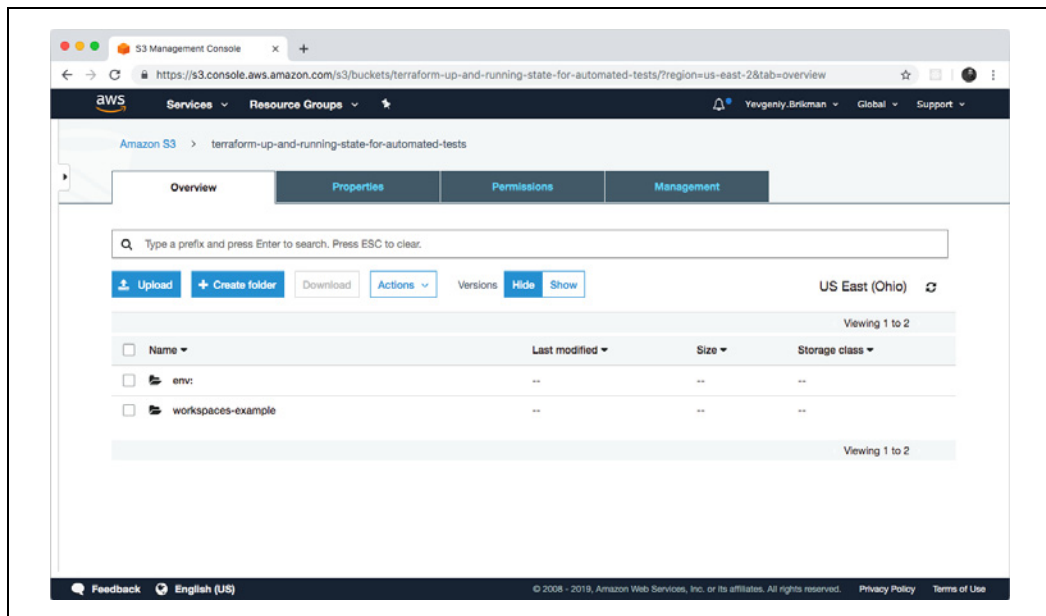
```
$ terraform workspace select example1
```

```
Switched to workspace "example1".
```

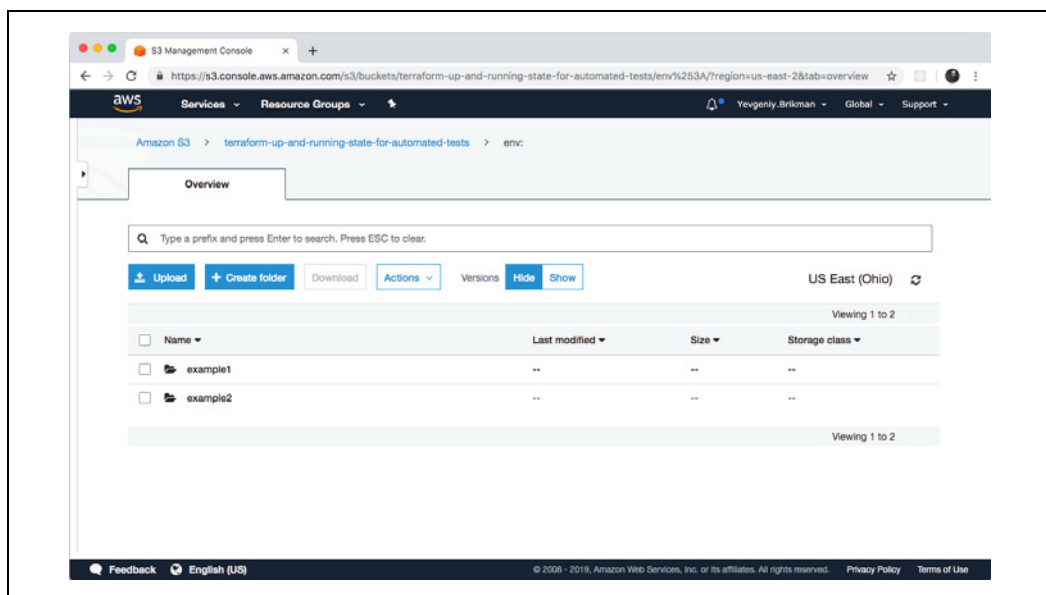
Aby dokładnie zrozumieć sposób działania przestrzeni roboczej, ponownie spójrz na kubełek S3. Powinieneś zobaczyć nowy katalog o nazwie `env`, jak pokazałem na rysunku 3.5.

Wewnątrz katalogu `env` znajduje się po jednym katalogu dla poszczególnych przestrzeni roboczych, co możesz zobaczyć na rysunku 3.6.

Wewnątrz tych przestrzeni roboczych Terraform używa wartości `key` podanej w konfiguracji backendu, więc powinieneś znaleźć pliki `example1/workspaces-example/terraform.tfstate` i `example2/workspaces-example/terraform.tfstate`. Innymi słowy, przejście do innej przestrzeni roboczej jest odpowiednikiem zmiany ścieżki dostępu wskazującej przechowywany plik informacji o stanie.



Rysunek 3.5. Kubełek S3 po rozpoczęciu korzystania z przestrzeni roboczych



Rysunek 3.6. Terraform tworzy po jednym katalogu dla każdej przestrzeni roboczej

To jest użyteczne, gdy masz już wdrożony moduł Terraform i chcesz przeprowadzić z nim pewne eksperymenty (próba refaktoryzacji kodu), ale jednocześnie nie chcesz, aby wpłynęły one na stan wdrożonej infrastruktury. Przestrzeń robocza Terraform pozwala na wydanie polecenia `terraform workspace new` i wdrożenie nowej kopii dokładnie tej samej infrastruktury, ale przechowującej w oddzielnym pliku informacje o stanie.

Istnieje nawet możliwość zmiany sposobu zachowania modułu na podstawie aktualnej przestrzeni roboczej poprzez odczyt jej nazwy za pomocą wyrażenia `terraform.workspace`. Dla przykładu — oto sposób na zdefiniowanie egzemplarza typu `t2.medium` w domyślnej przestrzeni roboczej i `t2.micro` we wszystkich pozostałych przestrzeniach roboczych (w celu zaoszczędzenia pieniędzy podczas eksperymentów):

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfaffe1f0"
  instance_type = terraform.workspace == "default" ? "t2.medium" : "t2.micro"
}
```

W tym fragmencie kodu została użyta *składnia trójargumentowa* do warunkowego przypisania `instance_type` wartości `t2.medium` lub `t2.micro` w zależności od wartości `terraform.workspace`. Szczegółowe omówienie składni trójargumentowej i logiki warunkowej Terraform znajdziesz w rozdziale 5.

Przestrzenie robocze Terraform mogą być doskonałym sposobem na szybkie wypróbowanie różnych wersji kodu, choć jednocześnie mają pewne wady:

- Pliki wszystkich przestrzeni roboczych są przechowywane w tym samym backendzie (np. w tym samym kubku S3). To oznacza wykorzystanie tych samych mechanizmów uwierzytelnienia i kontroli dostępu do wszystkich przestrzeni roboczych, co jest najważniejszym powodem, dla którego ten mechanizm jest nieodpowiedni do izolacji środowiska (np. izolacji środowisk roboczych i produkcyjnego).
- Przestrzenie robocze są niewidoczne w kodzie oraz w terminalu, o ile nie będą wydane polecenia `terraform workspace`. Podczas przeglądania kodu moduł wdrożony w jednej przestrzeni roboczej wygląda dokładnie tak samo jak moduł wdrożony w 10 przestrzeniach roboczych. To powoduje, że obsługa jest znacznie trudniejsza, ponieważ nie widać zbyt dobrze używanej infrastruktury.
- Dwa poprzednie punkty oznaczają, że przestrzenie robocze mogą być podatne na popełnianie błędów. Brak wyraźnej widoczności powoduje, że łatwo zapomnieć, która przestrzeń robocza jest aktualnie używana, i przypadkowo wprowadzić zmiany w nieprawidłowej (np. wydając polecenie `terraform destroy` przez pomyłkę w produkcyjnej przestrzeni roboczej zamiast w roboczej). Skoro używany jest ten sam mechanizm uwierzytelnienia dla wszystkich przestrzeni roboczych, nie ma żadnej linii obrony przed takimi błędami.

Aby zapewnić prawidłową izolację między środowiskami, zamiast przestrzeni roboczych należy skorzystać z układu plików, który jest tematem następnej sekcji. Jednak zanim przejdziesz dalej, upewnij się o usunięciu trzech wdrożonych przed chwilą egzemplarzy EC2, co wymaga wydania w każdej z nich poleceń `terraform workspace select <nazwa>` i `terraform destroy`.

Izolacja za pomocą układu plików

Zapewnienie pełnej izolacji między środowiskami wymaga wykonania następujących działań:

- Umieszczenie w oddzielnych katalogach plików konfiguracyjnych Terraform dla poszczególnych środowisk. Przykładowo cała konfiguracja dla środowiska roboczego może znaleźć się w katalogu o nazwie *stage*, a konfiguracja środowiska produkcyjnego w katalogu o nazwie *prod*.

- Należy skonfigurować oddzielne backendy dla poszczególnych środowisk i użyć do tego oddzielnych mechanizmów uwierzytelniania i kontroli dostępu (np. każde środowisko może znajdować się w oddzielnym koncie AWS wraz z oddzielnym kubełkiem S3 jako backendem).

Dzięki takiemu podejściu użycie oddzielnych katalogów znacznie wyraźniej wskazuje środowisko, w którym jest przeprowadzane wdrożenie. Zastosowanie oddzielnych plików stanu wraz z oddzielnymi mechanizmami uwierzytelniania oznacza znacznie mniejsze niebezpieczeństwo, że uszkodzenie jednego środowiska będzie miało wpływ na inne.

Po prawdzie koncepcja izolacji może wykroczyć poza środowisko i zejść na poziom „komponentu”, na którym komponent to spójny zestaw zasobów zwykle wdrażanych razem. Przykładowo po wdrożeniu podstawowej topologii infrastruktury — w AWS to VPC (ang. *virtual private cloud*) oraz wszystkie powiązane z nim podsieci, reguły routingu, VPN i sieciowe ACL — jej zmianę będziesz przeprowadzać co najwyżej raz na kilka miesięcy. Z drugiej strony nowa wersja serwera WWW może być wdrażana wielokrotnie w ciągu dnia. Jeżeli infrastrukturą komponentu VPC i serwera WWW zarządzasz w tym samym zestawie konfiguracji Terraform, wielokrotnie w ciągu dnia niepotrzebnie narażasz całą topologię sieci na ryzyko uszkodzenia (np. na skutek błędnego błędu w kodzie lub przypadkowego wykonania nieprawidłowego polecenia).

Dlatego też zalecam używanie oddzielnych katalogów Terraform (a tym samym oddzielnych plików stanu) dla poszczególnych środowisk (roboczego, produkcyjnego itd.) i komponentów (VPC, usług, baz danych). Aby zobaczyć, jak to się przedstawia w praktyce, warto zapoznać się z zalecanym układem plików w projektach Terraform.

Na rysunku 3.7 pokazałem układ plików dla typowego projektu Terraform.

Na najwyższym poziomie znajdują się oddzielne katalogi dla poszczególnych „środowisk”. Wprawdzie dokładne środowiska różnią się w poszczególnych projektach, ale zwykle stosują następujące:

stage

To jest środowisko dla obciążenia przedprodukcyjnego (np. do celów testowych).

prod

To jest środowisko dla obciążenia produkcyjnego (np. dla aplikacji przeznaczonych dla użytkowników).

mgmt

To jest środowisko dla narzędzi DevOps (np. bastion host, Jenkins).

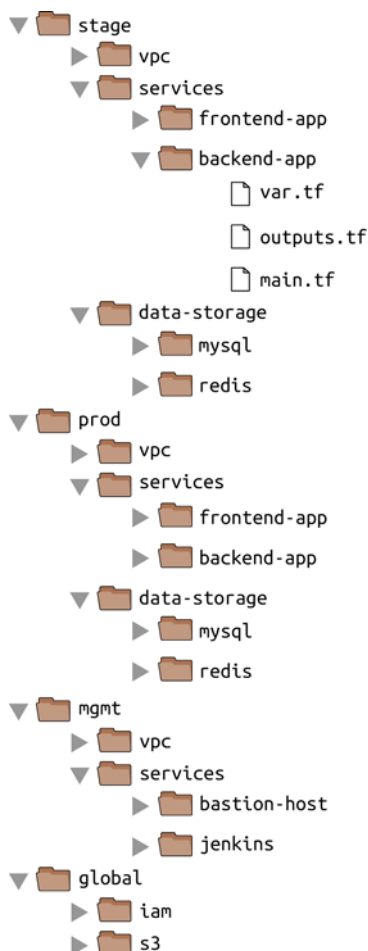
global

To jest miejsce do umieszczenia zasobów wykorzystywanych we wszystkich środowiskach (np. S3, IAM).

W poszczególnych środowiskach znajdują się katalogi przeznaczone dla każdego „komponentu”. Wprawdzie komponenty różnią się w projektach, ale zwykle stosują następujące:

vpc

Topologia sieci dla danego środowiska.



Rysunek 3.7. Typowy układ plików w projekcie Terraform

services

Aplikacje lub mikrousługi przeznaczone do uruchomienia w tym środowisku np. backendu Scali lub Ruby on Rails. Każda aplikacja może znajdować się w oddzielnym katalogu, aby została odizolowana od wszystkich pozostałych aplikacji.

data-storage

Magazyny danych przeznaczone do uruchomienia w danym środowisku, np. MySQL lub Redis. Każdy magazyn danych może być nawet przechowywany w oddzielnym katalogu, aby zapewnić izolację między nimi.

W poszczególnych komponentach znajdują się rzeczywiste pliki konfiguracyjne Terraform zorganizowane zgodnie z przedstawionymi tutaj konwencjami nazw:

variables.tf

Zmienne danych wejściowych.

outputs.tf

Zmienne danych wyjściowych.

main.tf

Zasoby.

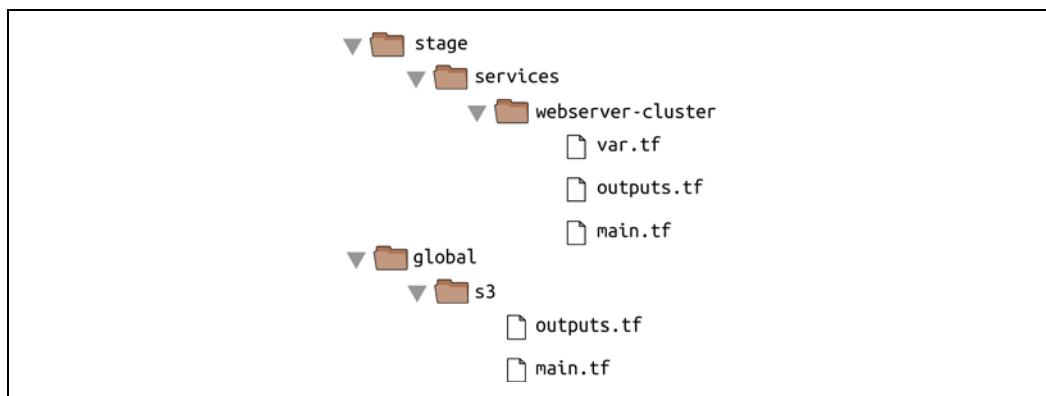
Gdy uruchomisz Terraform, narzędzie szuka w katalogu bieżącym plików wraz z rozszerzeniem *.tf*, więc można stosować dowolnie wybrane nazwy. Wprawdzie nazwy plików nie mają znaczenia dla Terraform, natomiast mogą mieć dla Twoich współpracowników. Stosowanie spójnej i przewidywalnej konwencji nazw znacznie ułatwia poruszanie się po kodzie — zawsze będzie wiadomo, gdzie szukać zmiennej, danych wyjściowych lub zasobu. Jeżeli plik Terraform stanie się ogromny, mam tutaj na myśli przede wszystkim plik *main.tf*, można go podzielić i poszczególne funkcjonalności umieścić w oddzielnych plikach (np. *iam.tf*, *s3.tf*, *database.tf*). To może być również sygnał wskazujący na potrzebę podzielenia kodu na mniejsze moduły, co jest tematem, w który zagłębię się w rozdziale 4.



Unikaj kopiowania i wklejania

Przedstawiony w tej sekcji układ plików zawiera wiele duplikatów. Przykładowo te same katalogi *frontend-app* i *backend-app* znajdują się w katalogach *stage* i *prod*. Nie przejmuj się, nie musisz kopiować ani wklejać tego kodu w całości. W rozdziale 4. zobaczysz, jak można wykorzystać moduły Terraform w celu stosowania reguły DRY.

Wykorzystamy teraz utworzony w rozdziale 2. kod klastra serwera WWW plus utworzony w tym rozdziale kod kubełka Amazon S3 i tabeli DynamoDB, a następnie przeorganizujemy go przy użyciu struktury katalogów pokazanej na rysunku 3.8.



Rysunek 3.8. Układ plików dla kodu klastra serwera WWW

Kubełek S3 utworzony w tym rozdziale powinien zostać przeniesiony do katalogu *global/s3*. Zmienne danych wyjściowych (*s3_bucket_arn* i *dynamodb_table_name*) przenieś do pliku *outputs.tf*.

Podczas przenoszenia katalogu upewnij się, że w trakcie kopiowania plików nie pominiesz (ukrytego) katalogu `.terraform`, aby w ten sposób uniknąć konieczności ponownej inicjalizacji wszystkiego.

Utworzony w rozdziale 2. klaster serwera WWW powinien zostać przeniesiony do `stage/services/webservice-cluster` (potraktuj to jako „testową” lub „roboczą” wersję klastra serwera WWW, wersję „produkcyjną” dodasz w następnym rozdziale). Także tym razem upewnij się o skopiowaniu katalogu `.terraform`, przenieś zmienne danych wejściowych do pliku `variables.tf`, a zmienne danych wyjściowych do `outputs.tf`.

Klaster serwera WWW powinien zostać uaktualniony do użycia S3 jako backendu. Konfigurację backendu można skopiować z `global/s3/main.tf` i wkleić, wprowadzając w niej mniejsze lub większe zmiany, ale koniecznie trzeba upewnić się o zmianie wartości `key` na wskazującą tę samą ścieżkę dostępu katalogu, w którym znajduje się kod Terraform serwera WWW: `stage/services/webserver-cluster/terraform.tfstate`. W ten sposób otrzymujesz mapowanie 1:1 między układem kodu Terraform w systemie kontroli wersji i przechowywanymi w S3 plikami informacji o stanie, więc nie ma wątpliwości o istnieniu połączenia między nimi. Moduł `s3` definiuje wspomnianą wartość `key` zgodnie z wymienioną konwencją.

Taki układ plików znacznie ułatwia przeglądanie kodu i dokładne ustalenie komponentów wdrożonych w poszczególnych środowiskach. Zapewnia też dobrą izolację między środowiskami i komponentami w środowiskach, co gwarantuje, że w przypadku problemów ewentualne szkody będą się ograniczały jedynie do niewielkiego fragmentu całej infrastruktury.

Oczywiście ta sama właściwość może pod pewnymi względami okazać się wadą. Podział komponentów na oddzielne katalogi z jednej strony uniemożliwia przypadkowe uszkodzenie całej infrastruktury przez wydanie jednego nieprawidłowego polecenia, z drugiej — nie pozwala na utworzenie całej infrastruktury przez wydanie tylko jednego polecenia. Gdy wszystkie komponenty dla środowiska są zdefiniowane w pojedynczej konfiguracji Terraform, całe środowisko może powstać po wydaniu polecenia `terraform apply`. Jeśli natomiast wszystkie komponenty są w oddzielnych katalogach, polecenie `terraform apply` trzeba wydać w każdym z nich (jeżeli używasz Terragrunt, ten proces można zautomatyzować za pomocą polecenia `apply-all`⁶).

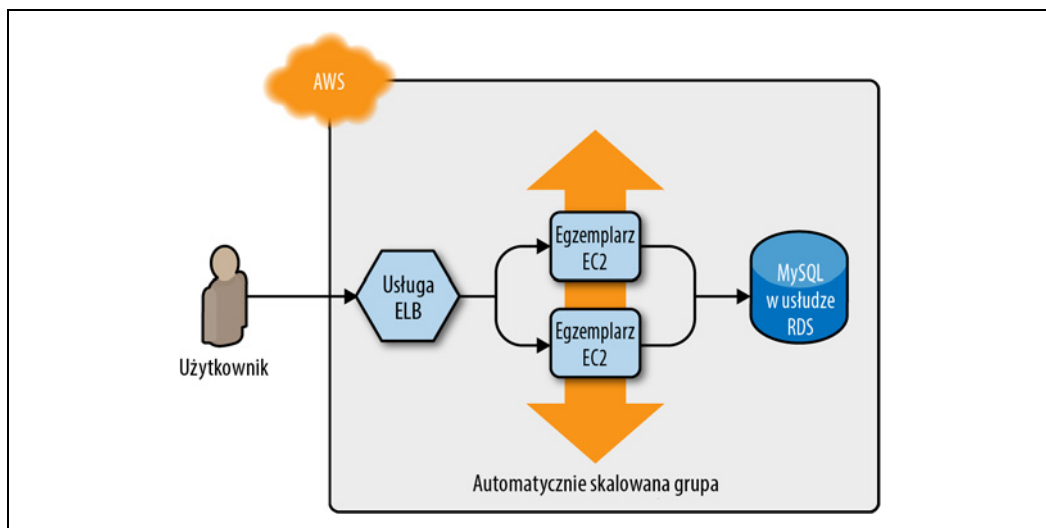
Pojawia się jeszcze inny problem z takim układem plików: znacznie trudniej jest wykorzystać zależności zasobu. Jeżeli kod aplikacji został zdefiniowany w tych samych plikach konfiguracyjnych, w których znajduje się kod bazy danych, to aplikacja będzie miała bezpośredni dostęp do atrybutów bazy danych za pomocą odwołania do atrybutów (np. dostęp do adresu bazy danych za pomocą `aws_db_instance.foo.address`). Nie ma natomiast takiej możliwości, jeśli kod aplikacji i kod bazy danych zgodnie z wcześniejszymi zaleceniami znajdują się w oddzielnych katalogach. Na szczęście także w tym zakresie Terraform oferuje rozwiązanie: źródło danych `terraform_remote_state`.

⁶ Więcej informacji na ten temat znajdziesz w dokumentacji Terragrunt na stronie <https://github.com/gruntwork-io/terragrunt>.

Źródło danych terraform_remote_state

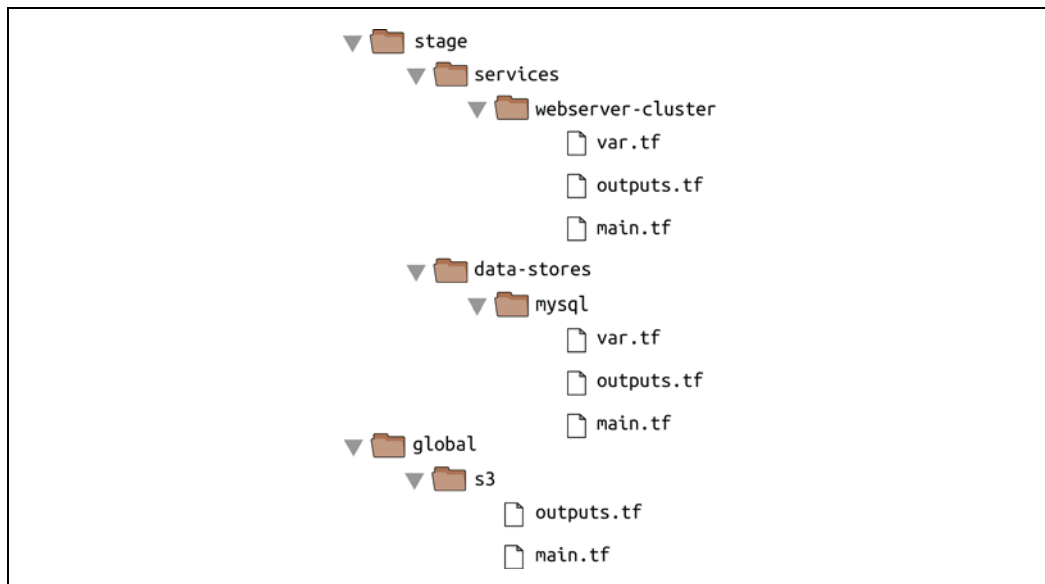
W rozdziale 2. źródło danych wykorzystałeś do pobrania z AWS informacji tylko do odczytu, takich jak źródło danych `aws_subnet_ids`, które zwraca listę podsieci w VPC. Istnieje jeszcze inne źródło danych szczególnie użyteczne podczas pracy z informacjami o stanie: `terraform_remote_state`. To źródło danych można wykorzystać w celu pobrania przechowywanego w innej konfiguracji Terraform pliku informacji o stanie, całkowicie w trybie tylko do odczytu.

Przedstawię to teraz na przykładzie. Wyobraź sobie, że klaster serwera WWW wymaga komunikacji z bazą danych MySQL. Przygotowanie bazy danych działającej w sposób skalowalny, bezpieczny, trwały i zapewniający wysoką dostępność wymaga dość dużo pracy. Zadanie to można pozostawić AWS, tym razem dzięki usłudze RDS (ang. *relational database service*), jak pokazałem na rysunku 3.9. Usługa RDS zapewnia obsługę wielu różnych baz danych, m.in. MySQL, PostgreSQL, SQL Server i Oracle.



Rysunek 3.9. Klaster serwera WWW komunikuje się z bazą danych MySQL wdrożoną za pomocą oferowanej przez Amazon usługi RDS

Bazy danych MySQL możesz nie chcieć definiować w tym samym zestawie plików konfiguracyjnych, w którym został skonfigurowany klaster serwera WWW, ponieważ uaktualnienia klastra serwera WWW będą wdrażane znacznie częściej i nie chcesz ryzykować przypadkowego uszkodzenia bazy danych podczas tej operacji. Dlatego też pierwszym krokiem powinno być utworzenie nowego katalogu `stage/data-stores/mysql` wraz z podstawowymi plikami Terraform (`main.tf`, `variables.tf` i `outputs.tf`), jak pokazałem na rysunku 3.10.



Rysunek 3.10. Kod bazy danych należy umieścić w katalogu `stage/data-stores`

Kolejnym krokiem jest utworzenie zasobów bazy danych w pliku `stage/data-stores/mysql/main.tf`:

```
provider "aws" {
  region = "us-east-2"
}

resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine           = "mysql"
  allocated_storage = 10
  instance_class   = "db.t2.micro"
  name             = "example_database"
  username         = "admin"
  # Jak należy zdefiniować hasło?
  password         = "???"
}
```

Na początku pliku zwykle umieszcza się zasób `provider`, tuż za nim znajduje się nowy zasób: `aws_db_instance`. Ten nowy zasób powoduje utworzenie bazy danych w RDS. Ustawienia w tym kodzie przeprowadzają konfigurację RDS do uruchomienia MySQL wraz z 10 GB pamięci masowej w egzemplarzu `db.t2.micro`, który ma jeden wirtualny procesor, 1 GB pamięci RAM i jest dostępny w ramach bezpłatnego planu AWS.

Zwróć uwagę na to, że jednym z parametrów koniecznych do przekazania zasobowi `aws_db_instance` jest hasło główne do bazy danych. Ponieważ to są dane wrażliwe, nie powinny być umieszczane bezpośrednio w kodzie zdefiniowanym w pliku zwykłego tekstu! Zamiast tego mamy dwie znacznie lepsze możliwości w zakresie przekazywania danych wrażliwych do zasobów Terraform.

Pierwszą możliwością jest wykorzystanie źródła danych Terraform do odczytania danych wrażliwych z przechowyującego je magazynu. Przykładowo dane takie jak hasło do bazy danych można

przechowywać w AWS Secrets Manager, czyli w zarządzanej usłudze AWS przeznaczonej specjalnie do przechowywania danych wrażliwych. Interfejs użytkownika AWS Secrets Manager można wykorzystać do przechowywania i odczytywania danych wrażliwych przez kod Terraform za pomocą źródła danych `aws_secretsmanager_secret_version`.

```
provider "aws" {
  region = "us-east-2"
}

resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine           = "mysql"
  allocated_storage = 10
  instance_class   = "db.t2.micro"
  name             = "example_database"
  username         = "admin"

  password =
    data.aws_secretsmanager_secret_version.db_password.secret_string
}

data "aws_secretsmanager_secret_version" "db_password" {
  secret_id = "mysql-master-password-stage"
}
```

Oto kilka innych obsługiwanych połączeń magazynów danych wrażliwych i źródeł danych, na które warto zwrócić uwagę:

- AWS Secrets Manager i źródło danych `aws_secretsmanager_secret_version` (to połączenie zostało wykorzystane w poprzednim przykładzie).
- AWS Systems Manager Parameter Store i źródło danych `aws_ssm_parameter`.
- AWS Key Management Service (AWS KMS) i źródło danych `aws_kms_secrets`.
- Google Cloud KMS i źródło danych `google_kms_secret`.
- Azure Key Vault i źródło danych `azurerm_key_vault_secret`.
- HashiCorp Vault i źródło danych `vault_generic_secret`.

Drugą możliwością w zakresie obsługi danych wrażliwych jest zarządzanie nimi zupełnie poza Terraform (np. wykorzystanie menedżera haseł, takiego jak 1Password, LastPass lub Dostęp do pęku kluczy w systemie macOS), i przekazywanie ich do Terraform za pomocą zmiennej środowiskowej. W takim przypadku należy w pliku `stage/data-stores/mysql/variables.tf` zdefiniować zmienną o nazwie `db_password`.

```
variable "db_password" {
  description = "Hasło do bazy danych"
  type        = string
}
```

Zwróć uwagę, że ta zmienna nie ma wartości default. To zamierzone działanie. W pliku zwykłego tekstu nie należy przechowywać haseł ani żadnych innych danych wrażliwych. Zamiast tego należy wykorzystać zmienną środowiskową.

Przypominam, że dla każdej zmiennej danych wejściowych `foo` zdefiniowanej w konfiguracjach Terraform odpowiednią wartość można dostarczyć do Terraform za pomocą zmiennej środowiskowej `TF_VAR_foo`. W przypadku zdefiniowanej wcześniej zmiennej danych wejściowych `db_password` utworzenie zmiennej środowiskowej `TF_VAR_foo` w systemach Linux, UNIX i macOS odbywa się za pomocą wymienionego tutaj polecenia:

```
$ export TF_VAR_db_password="(HASŁO_DO_BAZY_DANYCH)"
$ terraform apply
```

(...)

Zauważ spację umieszczoną celowo przez poleceniem `export`, aby uniknąć umieszczenia danych wrażliwych w przechowywanej na dysku historii poleceń Bash⁷. Jeszcze lepszym rozwiązaniem pozwalającym uniknąć przypadkowego umieszczenia danych wrażliwych w pliku zwykłego tekstu na dysku jest przechowywanie ich w przyjaznym powłocie magazynie danych wrażliwych (np. `pass`, który znajdziesz pod adresem <https://www.passwordstore.org/>) i wykorzystanie podpowłoki do bezpiecznego odczytania danych z menedżera `pass`, a następnie umieszczenia ich w zmiennej środowiskowej.

```
$ export TF_VAR_db_password=$(pass database-password)
$ terraform apply
```

(...)



Dane wrażliwe zawsze są przechowywane w pliku informacji o stanie

Odczytywanie danych wrażliwych z ich magazynu lub ze zmiennych środowiskowych jest dobrą praktyką, gwarantującą, że nie zostaną one umieszczone w pliku zwykłego tekstu na dysku. Jednak trzeba w tym miejscu wspomnieć, że niezależnie od sposobu ich odczytywania, jeśli są przekazywane w charakterze argumentu do zasobu Terraform takiego jak `aws_db_instance`, te dane wrażliwe znajdują się również w pliku informacji o stanie Terraform, który ma postać pliku zwykłego tekstu.

To jest znana wada Terraform, dla której nie istnieją efektywne rozwiązania. Dlatego też dobrze jest zachować ostrożność pod względem sposobu przechowywania plików informacji o stanie (np. zawsze włączać ich szyfrowanie) i tego, kto ma do nich dostęp (np. za pomocą uprawnień IAM zablokować do nich dostęp w kubelku S3).

Po skonfigurowaniu hasła następnym krokiem jest skonfigurowanie w kubelku S3 modułu przeznaczonego do przechowywania informacji o stanie w utworzonym wcześniej pliku `stage/data-stores/mysql/terraform.tfstate`.

```
terraform {
  backend "s3" {
    # Tę wartość zastąp nazwą używanego kubelka!
    bucket      = "terraform-up-and-running-state"
    key         = "stage/data-stores/mysql/terraform.tfstate"
```

⁷ W większości powłok systemów Linux, UNIX i macOS każde wydane polecenie jest przechowywane w pliku historii, np. `~/.bash_history`. Jeżeli polecenie będzie rozpoczynało się od spacji, większość powłok nie umieści go w pliku historii. Trzeba w tym miejscu wspomnieć o konieczności przypisania zmiennej środowiskowej `HISTCONTROL` wartości `ignoreboth` włączającej wspomnianą funkcjonalność, o ile nie została włączona domyślnie.

```

region          = "us-east-2"

# Tę wartość zastąp nazwą używanej tabeli DynamoDB!
dynamodb_table = "terraform-up-and-running-locks"
encrypt        = true
}
}

```

Wyдай polecenia `terraform init` i `terraform apply` w celu utworzenia bazy danych. Pamiętaj, że usługa Amazon RDS może wymagać około 10 minut na przygotowanie nawet małej bazy danych, więc bądź cierpliwy.

Skoro masz gotową bazę danych, to jak można przekazać jej adres i numer portu do klastra serwera WWW? Pierwszym krokiem jest dodanie dwóch zmiennych danych wyjściowych do pliku `stage/data-stores/mysql/outputs.tf`.

```

output "address" {
  value      = aws_db_instance.example.address
  description = "Nawiązanie połączenia z bazą danych w tym punkcie końcowym"
}

output "port" {
  value      = aws_db_instance.example.port
  description = "Numer portu, na którym nasłuchuje baza danych"
}

```

Ponownie wydaj polecenie `terraform apply` — w powłoce powinieneś zobaczyć dane wyjściowe podobne do przedstawionych tutaj:

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
address = tf-2016111123.cowu6mts6srx.us-east-2.rds.amazonaws.com
port    = 3306
```

Te dane wyjściowe są teraz przechowywane w dotyczących bazy danych informacjach o stanie Terraform, czyli w umieszczonym w kubelku S3 pliku `stage/data-stores/mysql/terraform.tfstate`. Klastrer serwera WWW będzie mógł odczytać dane z tego pliku po dodaniu źródła danych `terraform_remote_state` do kodu w pliku źródła danych `stage/services/webserver-cluster/main.tf`.

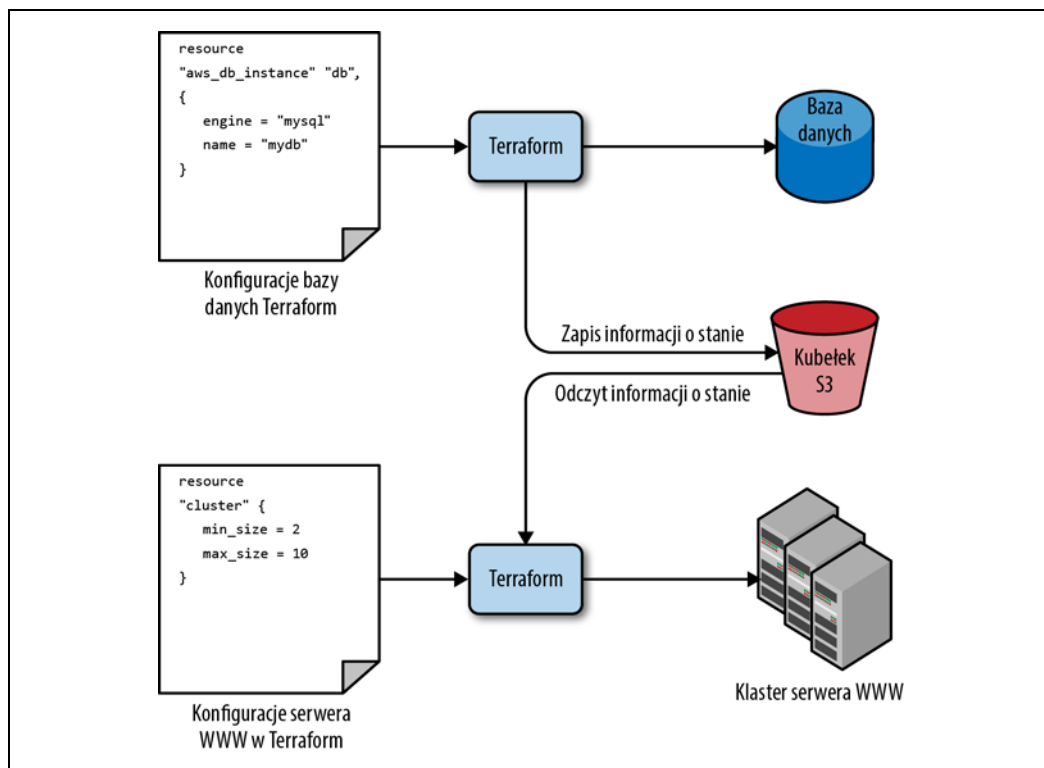
```

data "terraform_remote_state" "db" {
  backend = "s3"

  config = {
    bucket = "(NAZWA_KUBEŁKA)"
    key    = "stage/data-stores/mysql/terraform.tfstate"
    region = "us-east-2"
  }
}

```

Źródło danych `terraform_remote_state` konfiguruje kod klastra serwera WWW w celu odczytania pliku z kubełka S3 i katalogu, w którym baza danych przechowuje informacje o stanie, jak pokazałem na rysunku 3.11.



Rysunek 3.11. Baza danych zapisuje informacje o stanie w kubełku S3 (na górze rysunku), a klaster serwera WWW odczytuje te informacje z tego samego kubełka

Trzeba koniecznie zrozumieć, że podobnie jak w przypadku wszystkich źródeł danych Terraform, dane zwracane przez `terraform_remote_state` są tylko do odczytu. W kodzie Terraform klastra serwera WWW nie umieszczasz niczego, co mogłoby zmodyfikować stan, więc informacje o stanie można pobrać z bazy danych bez obaw o spowodowanie jakichkolwiek problemów związanych z samą bazą danych.

Wszystkie zmienne danych wyjściowych bazy danych są przechowywane w pliku informacji o stanie i mogą być odczytane za pomocą źródła danych `terraform_remote_state`, przy użyciu do tego odwołania do atrybutu w następującej postaci:

```
data.terraform_remote_state.<NAZWA>.outputs.<ATRYBUT>
```

Dla przykładu zobacz, jak można uaktualnić dane użytkownika egzemplarza klastra serwera WWW, aby z `terraform_remote_state` pobrać adres i numer portu serwera bazy danych i umieścić te informacje w udzielanej odpowiedzi HTTP:

```
user_data = <<EOF
#!/bin/bash
echo "Witaj, świecie" >> index.html
```



```
echo "${data.terraform_remote_state.db.outputs.address}" >> index.html
echo "${data.terraform_remote_state.db.outputs.port}" >> index.html
nohup busybox httpd -f -p ${var.server_port} &
EOF
```

Wraz ze zwiększaniem się skryptu danych użytkownika definiowanie go w postaci osadzonej staje się coraz trudniejsze. Ogólnie rzecz biorąc, osadzanie kodu jednego języka programowania (np. Bash) w kodzie utworzonym w innym języku (np. Terraform) znacznie utrudnia obsługę takiego projektu. Dlatego też zatrzymamy się na chwilę i wyodrębnimy kod Basha z kodu Terraform. W tym celu można użyć funkcji wbudowanej `file()` i źródła danych `template_file`. Spójrz na przykład przeprowadzenia takiej operacji.

Terraform oferuje wiele tzw. *funkcji wbudowanych*, które można wywoływać za pomocą wyrażenia w następującej postaci:

```
nazwa_funkcji(...)
```

Dla przykładu rozważ funkcję `format()`.

```
format(<FMT>, <ARGS>, ...)
```

Działanie tej funkcji polega na sformatowaniu argumentów w `ARGS` zgodnie ze składnią `sprintf` w ciągu tekstowym `FMT`⁸. Doskonałym sposobem na eksperymentowanie z funkcjami wbudowanymi jest wydanie polecenia `terraform console` w celu przejścia do konsoli interaktywnej pozwalającej na wypróbowanie składni Terraform, wydawanie poleceń sprawdzających stan infrastruktury i natychmiastowe otrzymywanie wyniku.

```
$ terraform console
```

```
> format("%.3f", 3.14159265359)
3.142
```

Pamiętaj, że konsola Terraform działa w trybie tylko do odczytu, więc nie musisz się martwić o przypadkową zmianę infrastruktury lub stanu.

Istnieje jeszcze wiele innych funkcji wbudowanych przeznaczonych do przeprowadzania operacji na ciągach tekstowych, liczbach, listach i mapowaniach⁹. Jedną z nich jest funkcja `file()`.

```
file(<ŚCIEŻKA_DOSTĘPU>)
```

Ta funkcja odczytuje plik znajdujący się w podanej ścieżce dostępu i zwraca jego zawartość w postaci ciągu tekstowego. Przykładowo skrypt danych użytkownika można umieścić w pliku `tage/services/webserver-cluster/user-data.sh` i następnie wczytywać go za pomocą polecenia:

```
file("user-data.sh")
```

W omawianym przykładzie problemem jest to, że skrypt danych użytkownika dla klastra serwera WWW potrzebuje pewnych danych dynamicznych z Terraform, m.in. portu serwera, adresu i numeru portu bazy danych. Gdy skrypt był osadzony w kodzie Terraform, pobranie wymaganych wartości było możliwe za pomocą odwołań i interpolacji Terraform. Takie rozwiązanie nie działa w przypadku

⁸ Dokumentacja składni `sprintf` znajduje się na stronie <https://golang.org/pkg/fmt/>.

⁹ Pełna lista funkcji wbudowanych znajduje się na stronie <https://www.terraform.io/docs/configuration/functions.html>.

używania funkcji `file()`. Jednak wymienione rozwiązanie zadziała, jeżeli zostanie użyte źródło danych `template_file`.

Źródło danych `template_file` ma dwa argumenty: `template` i `vars`. Pierwszy przedstawia ciąg tekstowy przeznaczony do wygenerowania, drugi mapuje zmienne, aby były dostępne podczas operacji generowania. To źródło danych ma jeden atrybut danych wyjściowych, `rendered`, zawierający wynik wygenerowania zawartości argumentu `template`. Aby zobaczyć to źródło danych w akcji, do pliku `stage/services/webserver-cluster/main.tf` dodaj następujący kod:

```
data "template_file" "user_data" {
  template = file("user-data.sh")

  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  }
}
```

Działanie tego fragmentu kodu polega na umieszczeniu w parametrze `template` zawartości `user-data.sh`, parametr `vars` zaś ma trzy zmienne wymagane przez skrypt danych użytkownika: numer portu serwera, adres i numer portu bazy danych. W celu użycia tych zmiennych należy w pokazany tutaj sposób uaktualnić skrypt `stage/services/webserver-cluster/user-data.sh`.

```
#!/bin/bash

cat > index.html <<EOF
<h1>Witaj, świecie</h1>
<p>Adres bazy danych: ${db_address}</p>
<p>Numer portu bazy danych: ${db_port}</p>
EOF
nohup busybox httpd -f -p ${server_port} &
```

Ten skrypt został nieco zmodyfikowany względem pierwotnego:

- Do wyszukania zmiennych została użyta standardowa w Terraform składnia interpolacji, ale dostępne są jedynie zmienne wymienione w mapowaniu `vars` źródła danych `template_file`. Zwróć uwagę na brak konieczności stosowania jakichkolwiek prefiksów, aby uzyskać dostęp do zmiennych: np. używana jest `server_port` zamiast `var.server_port`.
- Skrypt zawiera teraz pewną składnię HTML, np. `<h1>`, aby dane wyjściowe były nieco czytelniejsze w przeglądarce WWW.

Ostatnim krokiem jest uaktualnienie parametru `user_data` zasobu `aws_launch_configuration`, aby prowadził do atrybutu danych wyjściowych `rendered` źródła danych `template_file`.

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfaffe1f0"
  instance_type   = "t2.micro"
  security_groups = [aws_security_group.instance.id]
  user_data       = data.template_file.user_data.rendered

  # Wymagane podczas używania konfiguracji startowej w automatycznie skalowanej grupie.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```



Informacje dotyczące plików zewnętrznych

Jedną z zalet wyodrębnienia skryptu danych użytkownika do oddzielnego pliku jest możliwość przygotowania dla niego testów jednostkowych. Kod testów może nawet wypełniać interpolowane zmienne, wykorzystując do tego zmienne środowiskowe, ponieważ składnia Basha przeznaczona do wyszukiwania zmiennych jest dokładnie taka sama jak składnia interpolacji Terraform. Przykładowo istnieje możliwość utworzenia zautomatyzowanych testów dla *user-data.sh* w postaci przedstawionego tutaj fragmentu kodu:

```
export db_address=12.34.56.78
export db_port=5555
export server_port=8888

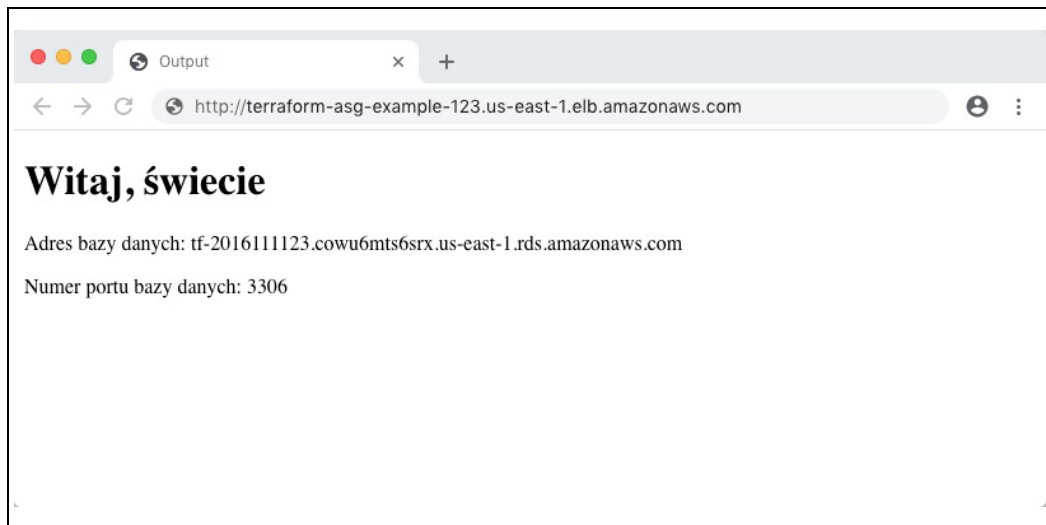
./user-data.sh

output=$(curl "http://localhost:$server_port")

if [[ $output == *"Witaj, świecie"* ]]; then
    echo "Sukces! Z serwera otrzymano oczekiwaną odpowiedź."
else
    echo "Błąd. Z serwera nie otrzymano oczekiwanego komunikatu 'Witaj,
    świecie'."
fi
```

Ten kod jest znacznie czytelniejszy niż w przypadku zawierającego osadzony skrypt Basha.

Jeżeli ten klaster wdrożysz za pomocą polecenia `terraform apply`, poczekaj na zarejestrowanie egzemplarzy w ALB, następnie w przeglądarce WWW przejdź pod adres URL udostępniony przez mechanizm równoważenia obciążenia, a zobaczysz dane podobne do pokazanych na rysunku 3.12.



Rysunek 3.12. Klaster serwera WWW może mieć programistyczny dostęp do adresu i numeru portu bazy danych

Klaster serwera WWW ma teraz poprzez Terraform programistyczny dostęp do adresu i numeru portu bazy danych. Jeżeli korzystasz z rzeczywistego frameworka internetowego (np. Ruby on Rails), możesz umieścić adres i numer portu w zmiennych środowiskowych lub też zapisać w pliku konfiguracyjnym, aby mogły być używane przez bibliotekę obsługi bazy danych (np. ActiveRecord) podczas komunikacji z bazą danych.

Podsumowanie

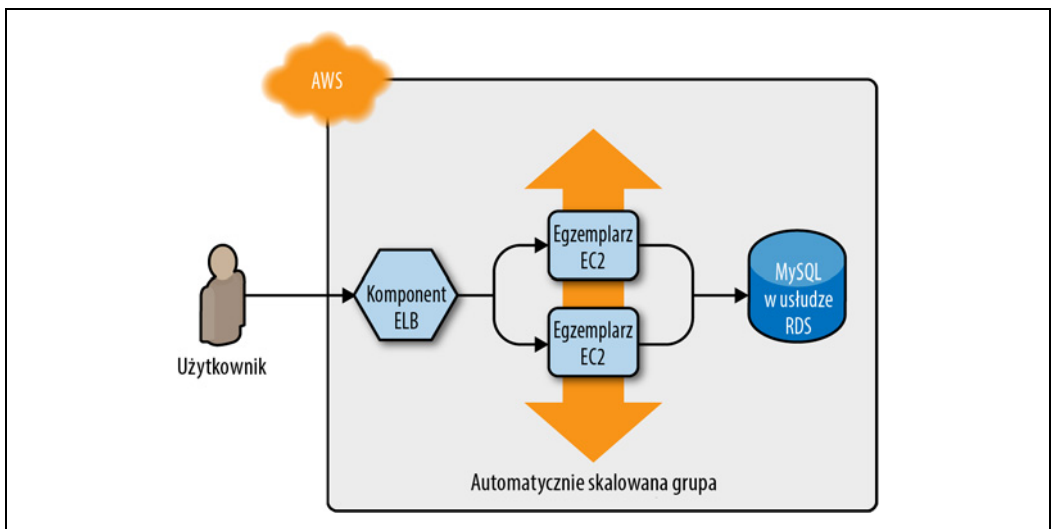
Powodem, dla którego tak wiele wysiłku wkłada się w izolację, nakładanie blokad i obsługę informacji o stanie, jest to, że infrastruktura wyrażona w postaci kodu (IaC) wiąże się z innymi kompromisami niż w przypadku typowego programowania. Gdy stworzysz kod typowej aplikacji, większość błędów jest stosunkowo niewielka i prowadzi do uszkodzenia jedynie fragmentu tej aplikacji. Natomiast gdy stworzysz kod kontrolujący infrastrukturę, błędy są zwykle znacznie poważniejsze i mogą uszkodzić wszystkie aplikacje — a także wszystkie magazyny danych, całą topologię sieci i praktycznie wszystko inne. Dlatego też podczas pracy nad kodem IaC zachęcam do stosowania większej liczby „mechanizmów bezpieczeństwa” niż w przypadku pracy nad typowym kodem¹⁰.

Dość powszechną obawą związaną z użyciem zalecanego układu plików jest to, że prowadzi on do powielonego kodu. Jeżeli chcesz uruchomić klaster serwera WWW w środowiskach roboczym i produkcyjnym, jak wówczas można uniknąć konieczności kopiowania i wklejania dużej ilości kodu między *stage/services/webserver-cluster* i *prod/services/webserver-cluster*? Odpowiedzią jest stosowanie modułów Terraform, które są tematem rozdziału 4.

¹⁰ Więcej informacji na temat mechanizmów bezpieczeństwa znajdziesz na stronie <https://www.ybrikman.com/writing/2016/02/14/agility-requires-safety/>.

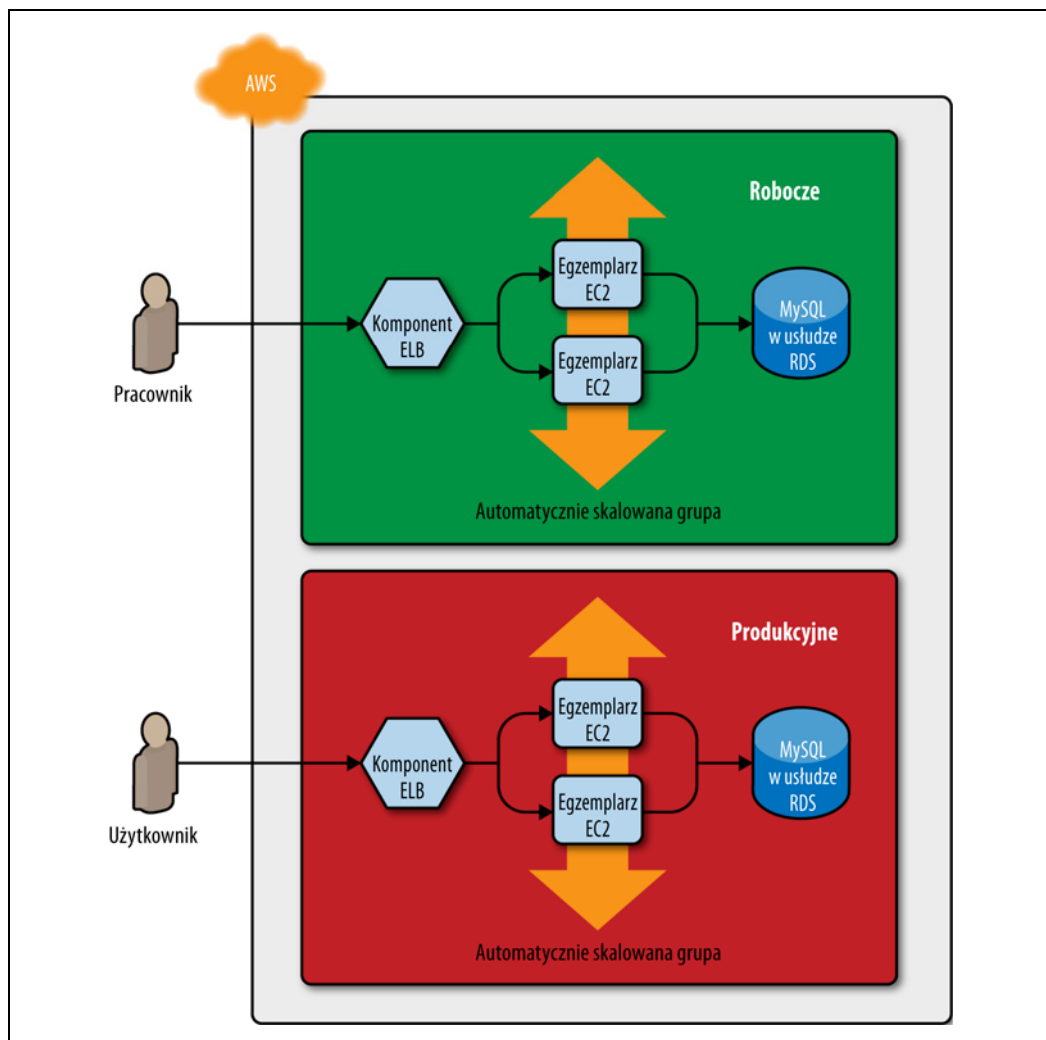
Zastosowanie modułów do tworzenia infrastruktury Terraform wielokrotnego użycia

Na końcu rozdziału 3. wdrożyłeś architekturę pokazaną na rysunku 4.1.



Rysunek 4.1. Architektura wdrożenia oparta na mechanizmie równoważenia obciążenia, klastrze serwera WWW i bazie danych

Wprawdzie taka architektura sprawdza się jako pierwsze środowisko, ale zwykle potrzebne są przynajmniej dwa środowiska: jedno przeznaczone do wewnętrznych testów w zespole („robocze”) i drugie dostępne dla rzeczywistych użytkowników („produkcyjne”), co możesz zobaczyć na rysunku 4.2. W idealnej sytuacji oba te środowiska będą praktycznie identyczne, choć w celu zaoszczędzenia pieniędzy środowisko testowe może mieć mniejszą liczbę mniejszych serwerów.



Rysunek 4.2. Dwa środowiska, z których każde stosuje architekturę opartą na mechanizmie równoważenia obciążenia, klastrze serwera WWW i bazie danych

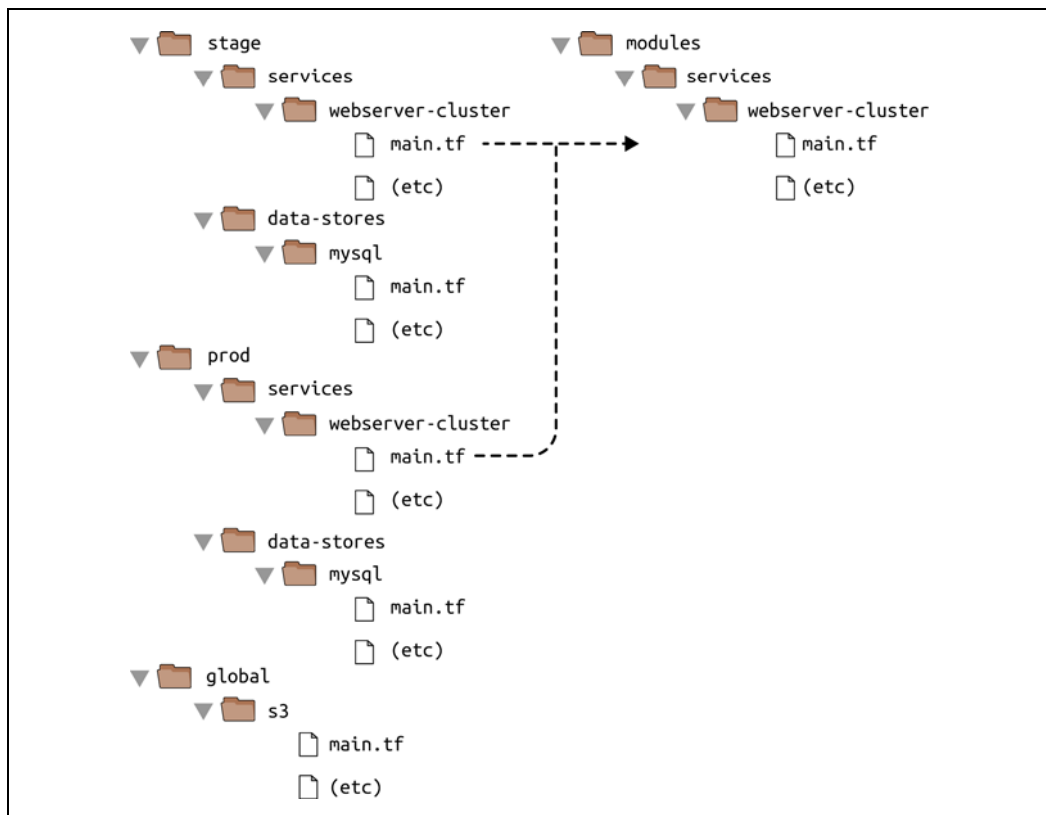
Jak można dodać to środowisko produkcyjne bez konieczności kopiowania i wklejania całego kodu ze środowiska roboczego? Czy istnieje np. możliwość uniknięcia kopiowania całego kodu z katalogu `stage/services/webserver-cluster` do `prod/services/webserver-cluster` oraz z katalogu `stage/data-stores/mysql` do `prod/data-stores/mysql`?

W językach programowania ogólnego przeznaczenia, np. Ruby, jeśli masz pewien kod przeznaczony do użycia w różnych miejscach, możesz go umieścić w funkcji, którą następnie wywołujesz według potrzeb.

```
def example_function()
  puts "Witaj, świecie"
end
```

```
# W innych fragmentach kodu wywołujesz tę funkcję.  
example_function()
```

W przypadku Terraform kod można umieścić w tzw. *module Terraform*, którego następnie można wielokrotnie używać w różnych miejscach kodu. Zamiast kopiować i wklejać ten sam kod w środowiskach roboczym i produkcyjnym, lepiej jest wykorzystać w nich ten sam moduł zawierający kod przeznaczony do wielokrotnego użycia, jak pokazałem na rysunku 4.3.



Rysunek 4.3. Umieszczenie kodu w module pozwala na wielokrotne używanie danego kodu w wielu środowiskach

To jednak wcale nie jest takie łatwe, jak można by sądzić. Moduł to ważny składnik podczas tworzenia łatwego w obsłudze i możliwego do przetestowania kodu Terraform wielokrotnego użycia. Gdy zaczniesz korzystać z modułów, nie będzie już odwrotu. Wszystko to, co będziesz tworzyć, zaczniesz umieszczać w modułach, opracujesz bibliotekę modułów i podzielisz się nią w firmie, zaczniesz używać modułów znalezionych w internecie i nauczysz się traktować całą infrastrukturę jako kolekcję modułów wielokrotnego użycia.

Z tego rozdziału dowiesz się, jak tworzyć i stosować moduły Terraform, na przykładzie następujących zagadnień:

- podstawy modułów,
- dane wejściowe modułu,

- wartości lokalne modułu,
- dane wyjściowe modułu,
- wady modułów,
- wersjonowanie modułów.



Przykładowe fragmenty kodu

Przypominam, że wszystkie przykładowe fragmenty kodu znajdują się w repozytorium GitHub pod adresem <https://github.com/brikis98/terraform-up-and-running-code>.

Podstawy modułów

Moduł Terraform jest bardzo prosty: to zestaw plików konfiguracyjnych Terraform umieszczonych w katalogu. Z technicznego punktu widzenia wszystkie utworzone dotąd konfiguracje były modułami, choć nieszczególnie interesującymi, ponieważ zostały wdrożone bezpośrednio (moduł w katalogu bieżącym jest nazywany *modułem głównym*). Aby poznać prawdziwe możliwości modułów, konieczne jest przygotowanie projektu wykorzystującego moduł z poziomu innego modułu.

Przykładowo kod zdefiniowany w katalogu *stage/services/webserver-cluster* — obejmujący automatycznie skalowaną grupę (ASG), mechanizm równoważenia obciążenia (ALB), grupy bezpieczeństwa i inne zasoby — zamienimy na moduł wielokrotnego użycia.

Pierwszym krokiem jest wydanie w katalogu *stage/services/webserver-cluster* polecenia `terraform destroy` w celu usunięcia wszelkich utworzonych wcześniej zasobów. Następnie należy utworzyć nowy katalog najwyższego poziomu o nazwie *modules* i wszystkie pliki z katalogu *stage/services/webserver-cluster* przenieść do *modules/services/webserver-cluster*. W efekcie powinna powstać struktura plików podobna do pokazanej na rysunku 4.4.

Otwórz plik *main.tf* w katalogu *modules/services/webserver-cluster* i usuń definicję `provider`. Dostawca powinien zostać skonfigurowany przez użytkownika modułu, a nie w samym module.

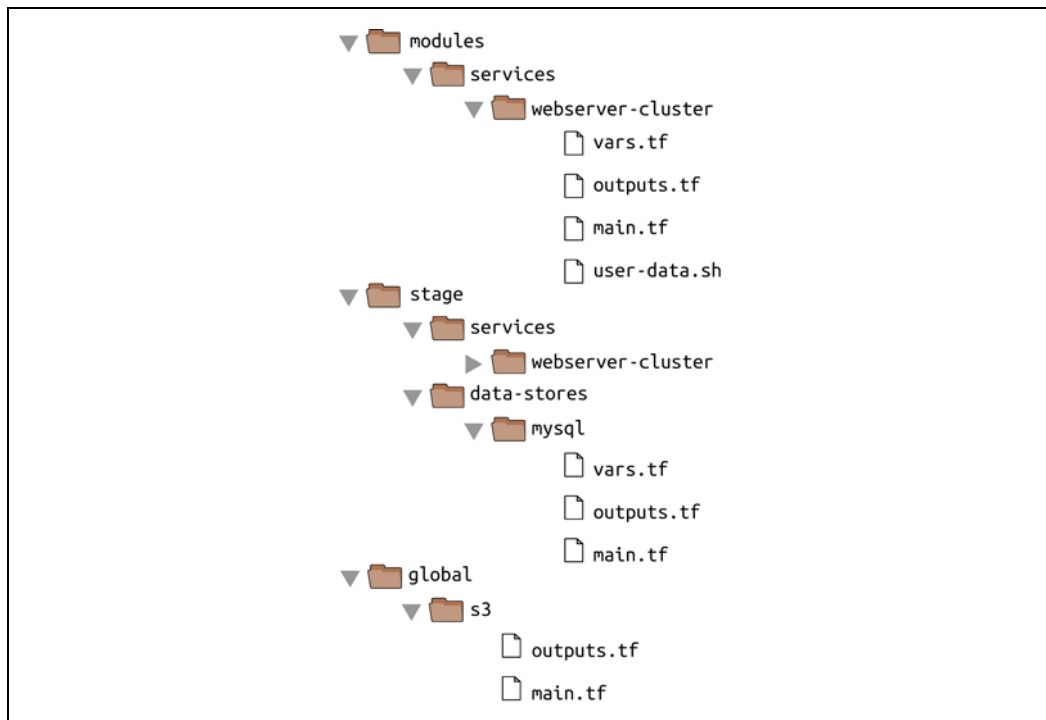
Teraz możesz wprowadzić zmiany pozwalające na wykorzystanie tego modułu w środowisku roboczym. Spójrz na składnię użycia modułu:

```
module "<NAZWA>" {
  source = "<ŹRÓDŁO>"

  [KONFIGURACJA ...]
}
```

gdzie NAZWA to nazwa używana w kodzie Terraform w celu odwołania się do danego modułu (np. *web-service*), ŹRÓDŁO to ścieżka dostępu wskazująca położenie modułu (np. *modules/services/webserver-cluster*), a KONFIGURACJA to jeden lub więcej argumentów charakterystycznych dla tego modułu.

Dlatego też możesz utworzyć nowy plik *stage/services/webserver-cluster/main.tf*, i wykorzystać w nim moduł *webserver-cluster*, jak pokazałem w kolejnym fragmencie kodu.



Rysunek 4.4. Struktura plików wraz z modulem i ze środowiskiem roboczym

```
provider "aws" {
  region = "us-east-2"
}

module "webserver_cluster" {
  source = "../../modules/services/webserver-cluster"
}
```

Dokładnie ten sam moduł można wykorzystać w środowisku produkcyjnym przez utworzenie nowego pliku *prod/services/webserver-cluster/main.tf* z następującą zawartością:

```
provider "aws" {
  region = "us-east-2"
}

module "webserver_cluster" {
  source = "../../modules/services/webserver-cluster"
}
```

W ten sposób przygotowałeś kod przeznaczony do wielokrotnego użycia w różnych środowiskach. W trakcie tej operacji nie trzeba było kopiować i wklejać zbyt dużej ilości kodu. Zwróć uwagę na to, że gdy trzeba dodać moduł do konfiguracji Terraform lub zmodyfikować parametr *source* modułu, konieczne jest wykonanie polecenia *terraform init* przed wydaniem polecenia *terraform plan* lub *terraform apply*.

```
$ terraform init
Initializing modules...
```

```
- webserver_cluster in ../../../../modules/services/webserver-cluster
```

```
Initializing the backend...
```

```
Initializing provider plugins...
```

```
Terraform has been successfully initialized!
```

W ten sposób poznałeś wszystkie asy w rękawie związane z poleceniem `terraform init`. Pobiera ono kod dostawców i modułów, a także konfiguruje backend — to wszystko w ramach jednego, wygodnego polecenia.

Zanim wydasz polecenie `terraform apply` dla tego kodu, musisz wiedzieć o problemie związanym z modulem `webserver-cluster`: wszystkie nazwy są na stałe zdefiniowane. To dotyczy nazw grup bezpieczeństwa, mechanizmu równoważenia obciążenia oraz wszelkich pozostałych zasobów. Dlatego też w przypadku użycia tego modułu więcej niż tylko raz nastąpi wygenerowanie błędu związanego z konfliktem nazw. Nawet szczegóły związane z bazą danych zostały na stałe zdefiniowane, ponieważ plik `main.tf` skopiowany do katalogu `modules/services/webserver-cluster` używa źródła danych `terraform_remote_data` w celu ustalenia adresu i numeru portu bazy danych, wymienione źródło danych zaś ma na stałe zdefiniowaną operację wyszukiwania tych danych w środowisku roboczym.

Aby rozwiązać ten problem, konieczne jest dodanie konfigurowanych danych wejściowych do modułu `webserver-cluster`, aby działał prawidłowo w różnych środowiskach.

Dane wejściowe modułu

Aby w języku programowania ogólnego przeznaczenia, takim jak Ruby, funkcja była konfigurowalna, można dodać do niej parametry.

```
def example_function(param1, param2)
  puts "Witaj, #{param1} #{param2}"
end

# W innych miejscach w kodzie można wywołać funkcję example_function().
example_function("foo", "bar")
```

W Terraform moduł również może mieć parametry danych wejściowych. Aby je zdefiniować, skorzystaj ze znanego Ci już mechanizmu, czyli zmiennych danych wejściowych. Otwórz plik `modules/services/webserver-cluster/variables.tf` i umieść w nim trzy nowe zmienne danych wejściowych.

```
variable "cluster_name" {
  description = "Nazwa używana we wszystkich zasobach klastra"
  type       = string
}

variable "db_remote_state_bucket" {
  description = "Nazwa kubełka S3 dla zdalnych informacji o stanie bazy danych"
  type       = string
}

variable "db_remote_state_key" {
  description = "Ścieżka dostępu do zdalnych informacji o stanie bazy danych w S3"
  type       = string
}
```

Teraz przeanalizuj plik *modules/services/webserver-cluster/main.tf* i wykorzystaj zmienną `var.cluster_name` zamiast na stałe zdefiniowanych nazw (np. zamiast `terraform-asg-example`). Zobacz, jak tę zmianę można wprowadzić w przypadku grupy bezpieczeństwa ALB:

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  ingress {
    from_port = 80
    to_port   = 80
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port = 0
    to_port   = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Zwróć uwagę na parametr `name`, którego wartością jest `"${var.cluster_name}-alb"`. Podobną zmianę trzeba wprowadzić w drugim zasobie `aws_security_group` (np. użyj nazwy `"${var.cluster_name}-instance"`), w zasobie `aws_alb` oraz w sekcji `tag` zasobu `aws_autoscaling_group`.

Powinieneś uaktualnić także źródło danych `terraform_remote_state` w celu użycia `db_remote_state_bucket` i `db_remote_state_key` jako — odpowiednio — wartości `bucket` i `key`. W ten sposób zyskasz pewność co do odczytywania pliku informacji o stanie właściwego środowiska.

```
data "terraform_remote_state" "db" {
  backend = "s3"
  config = {
    bucket = var.db_remote_state_bucket
    key    = var.db_remote_state_key
    region = "us-east-2"
  }
}
```

W środowisku roboczym, w pliku *stage/services/webserver-cluster/main.tf*, możesz odpowiednio wykorzystać nowe zmienne danych wejściowych:

```
module "webserver_cluster" {
  source = "../../modules/services/webserver-cluster"

  cluster_name      = "webserver-cluster"
  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"
}
```

To samo należy powtórzyć w pliku *stage/services/webserver-cluster/main.tf* w środowisku produkcyjnym:

```
module "webserver_cluster" {
  source = "../../modules/services/webserver-cluster"
```

```

cluster_name      = "webservers-prod"
db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"
}

```



Produkcyjna baza danych jeszcze nie istnieje. Jako ćwiczenie pozostawiam Ci dodanie produkcyjnej bazy danych w podobny sposób, jaki zastosowałeś podczas dodawania bazy danych w środowisku roboczym.

Jak możesz zobaczyć, zmienne danych wejściowych dla modułu zostały zdefiniowane za pomocą takiej samej składni, jaka jest stosowana podczas definiowania argumentów dla zasobu. Zmienne danych wejściowych to API modułu, kontrolują one sposób jego zachowania w różnych środowiskach. W omawianym przykładzie zostały użyte różne nazwy w odmiennych środowiskach, ale równie dobrze można zapewnić konfigurowalność pozostałych parametrów. Przykładowo w środowisku roboczym może być uruchamiany mały klaster serwera WWW (w celu obniżenia kosztów), natomiast w środowisku produkcyjnym — większy klaster, który ma możliwość obsłużenia znacznie większego ruchu sieciowego. Aby zastosować takie rozwiązanie, trzeba dodać trzy kolejne zmienne danych wejściowych do pliku *modules/services/webserver-cluster/variables.tf*:

```

variable "instance_type" {
  description = "Typ egzemplarza EC2 do uruchomienia (np. t2.micro)"
  type        = string
}

variable "min_size" {
  description = "Minimalna liczba egzemplarzy EC2 w ASG"
  type        = number
}

variable "max_size" {
  description = "Maksymalna liczba egzemplarzy EC2 w ASG"
  type        = number
}

```

Następnym krokiem jest uaktualnienie konfiguracji startowej w pliku *modules/services/webserver-cluster/main.tf* w celu zdefiniowania parametru `instance_type` dla nowej zmiennej danych wejściowych `var.instance_type`.

```

resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfaffe1f0"
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]
  user_data       = data.template_file.user_data.rendered

  # Wymagane podczas używania konfiguracji startowej wraz z automatycznie skalowaną grupą.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}

```

Powinieneś również uaktualnić definicję ASG w tym samym pliku, aby zdefiniować jego parametry `min_size` i `max_size` dla nowych zmiennych danych wejściowych — odpowiednio: `var.min_size` i `var.max_size`.

```

resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"

  min_size = var.min_size
  max_size = var.max_size

  tag {
    key           = "Name"
    value         = var.cluster_name
    propagate_at_launch = true
  }
}

```

Teraz w środowisku roboczym (*stage/services/webserver-cluster/main.tf*) można tworzyć mały i niedrogi klastery dzięki przypisaniu zmiennej `instance_type` wartości `t2.micro` oraz zmiennym `min_size` i `max_size` wartości 2.

```

module "webserver_cluster" {
  source = "../../modules/services/webserver-cluster"

  cluster_name      = "webservers-stage"
  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"

  instance_type = "t2.micro"
  min_size      = 2
  max_size      = 2
}

```

Z drugiej strony w środowisku produkcyjnym jako wartość `instance_type` można podać większy egzemplarz EC2 charakteryzujący się potężniejszym procesorem i większą ilością pamięci RAM, taki jak `m4.large` (ten egzemplarz nie jest dostępny w bezpłatnym planie AWS, więc jeśli jedynie poznajesz Terraform i nie chcesz być obciążony kosztami przez AWS, pozostań przy wartości `t2.micro` dla `instance_type`). Zmiennej `max_size` można też przekazać wartość 10, aby zezwolić na zwiększanie i zmniejszanie się klastra w zależności od obciążenia (nie przejmuj się, ponieważ początkowo klastery jest uruchamiany tylko z dwoma egzemplarzami).

```

module "webserver_cluster" {
  source = "../../modules/services/webserver-cluster"

  cluster_name      = "webservers-prod"
  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"

  instance_type = "m4.large"
  min_size      = 2
  max_size      = 10
}

```

Wartości lokalne modułu

Używanie zmiennych danych wejściowych do zdefiniowania danych wejściowych modułu jest doskonałym rozwiązaniem. Jednak co można zrobić w sytuacji, gdy zachodzi potrzeba zdefiniowania w module zmiennej, która ma przeprowadzać pewne obliczenia pośrednie? Lub jeśli chcesz stosować w kodzie regułę DRY i jednocześnie nie chcesz udostępniać tej zmiennej jako konfigurowalnych danych wejściowych? Przykładowo zdefiniowany w pliku `modules/services/webserver-cluster/main.tf` mechanizm równoważenia obciążenia w module `webserver-cluster` nasłuchuje na porcie 80, czyli domyślnym porcie dla HTTP. Numer tego portu jest aktualnie skopiowany i wklejony w wielu miejscach, m.in. w konfiguracji mechanizmu równoważenia obciążenia.

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = 80
  protocol          = "HTTP"
  # Domyślnie zwracana jest prosta strona błędu 404.
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: nie znaleziono strony"
      status_code  = 404
    }
  }
}
```

Pojawia się również w grupie bezpieczeństwa mechanizmu równoważenia obciążenia.

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  ingress {
    from_port = 80
    to_port   = 80
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Wartości w grupie bezpieczeństwa — m.in. blok CIDR `0.0.0.0/0` wskazujący na „wszystkie adresy IP”, wartość `0` określająca „dowolny port” i wartość `-1` określająca „dowolny protokół” — również zostały skopiowane i wklejone w wielu miejscach modułu. Gdy takie wartości magiczne są na stałe zdefiniowane w wielu miejscach kodu, jego odczyt i późniejsza konserwacja są znacznie trudniejsze. Wprawdzie te wartości można wyodrębnić i umieścić w zmiennych, ale wówczas użytkownik modułu

mógłby (przypadkowo) je nadpisać, a tego nie chcemy. Zamiast tego, wykorzystując zmienne danych wejściowych, można zdefiniować je jako tzw. *wartości lokalne* w bloku `locals`.

```
locals {
  http_port    = 80
  any_port     = 0
  any_protocol = "-1"
  tcp_protocol = "tcp"
  all_ips      = ["0.0.0.0/0"]
}
```

Wartości lokalne pozwalają na przypisanie nazwy dowolnemu wyrażeniu Terraform oraz wykorzystanie tej nazwy w module. Wspomniane nazwy pozostaną dostępne tylko w danym module, więc nie mają wpływu na pozostałe moduły i nie można ich nadpisywać z zewnątrz. Aby odczytać wartość zmiennej lokalnej, konieczne jest użycie *odwołania do zmiennej lokalnej*, którego składnia przedstawia się następująco:

```
local.<NAZWA>
```

Wykorzystując tę składnię, można uaktualnić mechanizm równoważenia obciążenia.

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = local.http_port
  protocol          = "HTTP"

  # Domyślnie zwracana jest prosta strona błędu 404.
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: nie znaleziono strony"
      status_code  = 404
    }
  }
}
```

Kolejnym krokiem jest uaktualnienie wszystkich grup bezpieczeństwa w module, w tym zdefiniowanej w mechanizmie równoważenia obciążenia.

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  ingress {
    from_port = local.http_port
    to_port   = local.http_port
    protocol  = local.tcp_protocol
    cidr_blocks = local.all_ips
  }

  egress {
    from_port = local.any_port
    to_port   = local.any_port
    protocol  = local.any_protocol
    cidr_blocks = local.all_ips
  }
}
```

Zmienne wartości lokalne niezwykle ułatwiają odczytywanie i konserwację kodu źródłowego, powinieneś więc często z nich korzystać.

Dane wyjściowe modułu

Potężną funkcjonalnością ASG jest możliwość skonfigurowania grupy w taki sposób, aby liczba uruchomionych serwerów była zwiększana lub zmniejszana w odpowiedzi na wielkość obciążenia. Jednym z rozwiązań jest wykorzystanie *akcji harmonogramu*, która potrafi zmienić wielkość klastra na podstawie harmonogramu w ciągu dnia. Przykładowo, jeśli poziom ruchu sieciowego do klastra jest większy w ciągu standardowych godzin pracy, można zdefiniować akcję harmonogramu zwiększającą liczbę serwerów o godzinie 9 i zmniejszającą tę liczbę o godzinie 17.

Jeżeli definiujesz akcję harmonogramu w module `webserver-cluster`, będzie ona zastosowana w środowiskach roboczym i produkcyjnym. Ponieważ wspomniane wcześniej skalowanie jest niepotrzebne w środowisku roboczym, harmonogram automatycznego skalowania można zdefiniować bezpośrednio w konfiguracji środowiska produkcyjnego. (Z rozdziału 5. dowiesz się, jak warunkowo definiować zasoby, co pozwoli na przeniesienie akcji harmonogramu do modułu `webserver-cluster`).

Aby zdefiniować akcję harmonogramu, należy do kodu w pliku `prod/services/webserver-cluster/main.tf` dodać dwa następujące zasoby `aws_autoscaling_schedule`:

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
  scheduled_action_name = "scale-out-during-business-hours"
  min_size              = 2
  max_size              = 10
  desired_capacity       = 10
  recurrence             = "0 9 * * *"
}

resource "aws_autoscaling_schedule" "scale_in_at_night" {
  scheduled_action_name = "scale-in-at-night"
  min_size              = 2
  max_size              = 10
  desired_capacity       = 2
  recurrence             = "0 17 * * *"
}
```

Ten kod używa jednego zasobu `aws_autoscaling_schedule` w celu zwiększenia do 10 liczby serwerów w trakcie godzin pracy (parametr `recurrence` stosuje składnię mechanizmu cron, więc `"0 9 * * *"` oznacza „godzina 9 codziennie”) i drugiego zasobu `aws_autoscaling_schedule` w celu zmniejszenia liczby serwerów w nocy (`"0 17 * * *"` oznacza „godzina 17 codziennie”). Jednak w obu przykładach użycia `aws_autoscaling_schedule` brakuje wymaganego parametru, `autoscaling_group_name`, określającego nazwę ASG. Sama grupa ASG została zdefiniowana w module `webserver-cluster`, więc być może zastanawiasz się, jak uzyskać dostęp do jej nazwy. W językach programowania ogólnego przeznaczenia, np. Ruby, funkcje mogą zwracać wartości.

```
def example_function(param1, param2)
  return "Witaj, #{param1} #{param2}"
end

# W innych miejscach kodu można użyć funkcji example_function().
return_value = example_function("foo", "bar")
```


W Terraform moduł również może zwracać wartość. Także w tym przypadku zastosowanie znajduje mechanizm, który już znasz: zmienne danych wyjściowych. Nazwę grupy ASG można dodać jako zmienną danych wyjściowych w kodzie zdefiniowanym w pliku `/modules/services/webserver-cluster/outputs.tf`.

```
output "asg_name" {
  value       = aws_autoscaling_group.example.name
  description = "Nazwa automatycznie skalowanej grupy"
}
```

Dostęp do zmiennej danych wyjściowych modułu odbywa się za pomocą następującej składni:

```
module.<NAZWA_MODUŁU>.<NAZWA_ZMIENNEJ>
```

Przykładowo:

```
module.frontend.asg_name
```

W pliku `prod/services/webserver-cluster/main.tf` można wykorzystać tę składnię w celu zdefiniowania parametru `autoscaling_group_name` w poszczególnych zasobach `aws_autoscaling_schedule`.

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
  scheduled_action_name = "scale-out-during-business-hours"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 10
  recurrence             = "0 9 * * *"
}
```

```
  autoscaling_group_name = module.webserver_cluster.asg_name
```

```
}
```

```
resource "aws_autoscaling_schedule" "scale_in_at_night" {
  scheduled_action_name = "scale-in-at-night"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 2
  recurrence             = "0 17 * * *"
}
```

```
  autoscaling_group_name = module.webserver_cluster.asg_name
```

```
}
```

Być może będziesz chciał udostępnić jeszcze inną zmienną danych wyjściowych w module `webserver-cluster`: nazwę DNS grupy ASG, aby poznać adres URL, który można przetestować po wdrożeniu klastra. W tym celu dodaj następującą zmienną danych wyjściowych do pliku `/modules/services/webserver-cluster/outputs.tf`:

```
output "alb_dns_name" {
  value       = aws_lb.example.dns_name
  description = "Nazwa domeny mechanizmu równoważenia obciążenia"
}
```

Następnie te dane wyjściowe można „przekazać” w plikach `stage/services/webserver-cluster/outputs.tf` i `prod/services/webserver-cluster/outputs.tf`:

```
output "alb_dns_name" {
  value       = module.webserver_cluster.alb_dns_name
  description = "Nazwa domeny mechanizmu równoważenia obciążenia"
}
```

Klaster serwera WWW jest niemalże gotowy do wdrożenia. Pozostało jeszcze uwzględnienie kilku drobnych kwestii związanych z modułami.

Problemy z modułami

Podczas tworzenia modułów trzeba zwracać uwagę na następujące kwestie:

- ścieżki dostępu do pliku,
- osadzone bloki kodu.

Ścieżki dostępu do pliku

W rozdziale 3. przenieś skrypt danych użytkownika dla klastra serwera WWW do pliku zewnętrznego, *user-data.sh*, i wykorzystaj funkcję wbudowaną `file()` w celu odczytania z dysku zawartości tego pliku. Problem z funkcją `file()` polega na tym, że używana ścieżka dostępu do pliku musi być względna (ponieważ narzędzie Terraform może być uruchamiane w różnych komputerach). Jednak względna dla którego komponentu?

Domyślnie Terraform interpretuje ścieżkę dostępu względem katalogu bieżącego. Takie rozwiązanie jest stosowane podczas użycia funkcji `file()` w pliku konfiguracyjnym Terraform, znajdującym się w tym samym katalogu, w którym zostało wydane polecenie `terraform apply` (o ile funkcja `file()` jest używana w module głównym), ale nie działa, gdy wywołanie `file()` pojawia się w module zdefiniowanym w oddzielnym katalogu.

Aby rozwiązać ten problem, można wykorzystać wyrażenie znane jako *odwołanie ścieżki dostępu*, które ma postać `path.<TYP>`. Terraform obsługuje następujące typy odwołań ścieżek dostępu:

`path.module`

Zwraca ścieżkę dostępu systemu plików modułu, w którym zostało zdefiniowane wyrażenie.

`path.root`

Zwraca ścieżkę dostępu systemu plików modułu głównego.

`path.cwd`

Zwraca ścieżkę dostępu systemu plików aktualnego katalogu roboczego. W trakcie normalnego użycia Terraform jest ona taka sama jak `path.root`, ale w niektórych bardziej zaawansowanych przypadkach użycia Terraform działa w innym katalogu niż katalog modułu głównego i wówczas wspomniane ścieżki dostępu będą odmienne.

W przypadku skryptu danych użytkownika potrzebna jest ścieżka dostępu względem samego modułu, więc w źródle danych `template_file` zdefiniowanym w pliku *modules/services/webserver-cluster/main.tf* powinieneś wykorzystać `path.module`.

```
data "template_file" "user_data" {
  template = file("${path.module}/user-data.sh")
}

vars = {
  server_port = var.server_port
}
```

```

    db_address = data.terraform_remote_state.db.outputs.address
    db_port    = data.terraform_remote_state.db.outputs.port
  }
}

```

Osadzony blok kodu

Konfiguracja dla niektórych zasobów Terraform może być zdefiniowana w postaci osadzonych bloków lub zasobów zewnętrznych. Podczas tworzenia modułu zawsze powinienś preferować wykorzystanie zewnętrznego zasobu.

Przykładowo zasób `aws_security_group` pozwala na zdefiniowanie reguł wejścia i wyjścia za pomocą bloków osadzonych, jak mogłeś zobaczyć w module `webserver-cluster` (*modules/services/webserver-cluster/main.tf*).

```

resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  ingress {
    from_port = local.http_port
    to_port   = local.http_port
    protocol  = local.tcp_protocol
    cidr_blocks = local.all_ips
  }

  egress {
    from_port = local.any_port
    to_port   = local.any_port
    protocol  = local.any_protocol
    cidr_blocks = local.all_ips
  }
}

```

Powinieneś zmienić ten moduł w taki sposób, aby zdefiniować dokładnie te same reguły wejścia i wyjścia za pomocą oddzielnych zasobów `aws_security_group_rule` (upewnij się, że zmodyfikowane są obie grupy bezpieczeństwa w tym module).

```

resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"
}

resource "aws_security_group_rule" "allow_http_inbound" {
  type           = "ingress"
  security_group_id = aws_security_group.alb.id

  from_port = local.http_port
  to_port   = local.http_port
  protocol  = local.tcp_protocol
  cidr_blocks = local.all_ips
}

resource "aws_security_group_rule" "allow_all_outbound" {
  type           = "egress"
  security_group_id = aws_security_group.alb.id
}

```

```

from_port    = local.any_port
to_port      = local.any_port
protocol     = local.any_protocol
cidr_blocks  = local.all_ips
}

```

Jeżeli spróbujesz wykorzystać połączenie *zarówno* bloków osadzonych, jak i oddzielnych zasobów, otrzymasz błędy w przypadku wystąpienia konfliktu reguł routingu i nadpisywania jednej przez drugą. Dlatego też trzeba się zdecydować na blok osadzony lub oddzielny zasób. Z powodu tego ograniczenia podczas tworzenia modułu zawsze należy próbować wykorzystać oddzielny zasób zamiast bloku osadzonego. W przeciwnym razie moduł będzie mniej elastyczny i konfigurowalny.

Przykładowo, jeśli wszystkie reguły wejścia i wyjścia w module `webserver-cluster` zostały zdefiniowane jako oddzielne zasoby `aws_security_group_rule`, moduł może być na tyle elastyczny, aby umożliwić użytkownikom dodawanie własnych reguł z zewnątrz. To wymaga wyeksportowania identyfikatora `aws_security_group` jako zmiennej danych wyjściowych w pliku `modules/services/webserver-cluster/outputs.tf`:

```

output "alb_security_group_id" {
  value     = aws_security_group.alb.id
  description = "Identyfikator grupy bezpieczeństwa dołączonej do mechanizmu równoważenia obciążenia"
}

```

Teraz wyobraź sobie, że w środowisku roboczym konieczne jest udostępnienie portu dodatkowego w celach testowych. To można zrobić bardzo łatwo, przez dodanie zasobu `aws_security_group_rule` w pliku `stage/services/webserver-cluster/main.tf`:

```

resource "aws_security_group_rule" "allow_testing_inbound" {
  type      = "ingress"
  security_group_id = module.webserver_cluster.alb_security_group_id

  from_port    = 12345
  to_port      = 12345
  protocol     = "tcp"
  cidr_blocks  = ["0.0.0.0/0"]
}

```

Jeżeli zdefiniowałeś choć jedną regułę wejścia lub wyjścia w bloku osadzonym, ten fragment kodu nie będzie działał. Warto w tym miejscu dodać, że ten sam typ problemu dotyczy pewnej liczby zasobów Terraform, m.in.:

- `aws_security_group` i `aws_security_group_rule`,
- `aws_route_table` i `aws_route`,
- `aws_network_acl` i `aws_network_acl_rule`.

Wreszcie jesteś gotowy do wdrożenia klastra serwera WWW w środowiskach roboczym i produkcyjnym. Jak zwykle wydaj polecenie `terraform apply` i ciesz się możliwością używania dwóch oddzielnych kopii infrastruktury.



Izolacja sieci

Przykłady w tym rozdziale prowadzą do utworzenia dwóch środowisk odizolowanych w kodzie Terraform, a także odizolowanych w kategoriach użycia oddzielnych mechanizmów równoważenia obciążenia, serwerów i baz danych, choć jednocześnie nieodizolowanych na poziomie sieci. Aby zachować prostotę przykładów przedstawionych w książce, wszystkie zasoby są wdrażane w tej samej chmurze VPC. To oznacza, że serwer znajdujący się w środowisku testowym ma możliwość komunikacji z serwerem w środowisku produkcyjnym i na odwrót.

W rzeczywistych projektach zdefiniowanie obu środowisk w jednym VPC powoduje dwa duże niebezpieczeństwa. Pierwsze: błąd popełniony w jednym środowisku może mieć wpływ na to drugie. Przykładowo, jeśli podczas wprowadzania zmian w środowisku testowym przypadkowo zepsujesz konfigurację tabel routingu, będzie to miało wpływ również na routing w środowisku produkcyjnym. Drugie: jeśli atakujący uzyska dostęp do jednego środowiska, będzie miał również dostęp do drugiego. W przypadku szybkiego wprowadzania zmian w środowisku roboczym i przypadkowego pozostawienia otwartego portu haker, który włamie się do środowiska roboczego, będzie miał również dostęp do środowiska produkcyjnego.

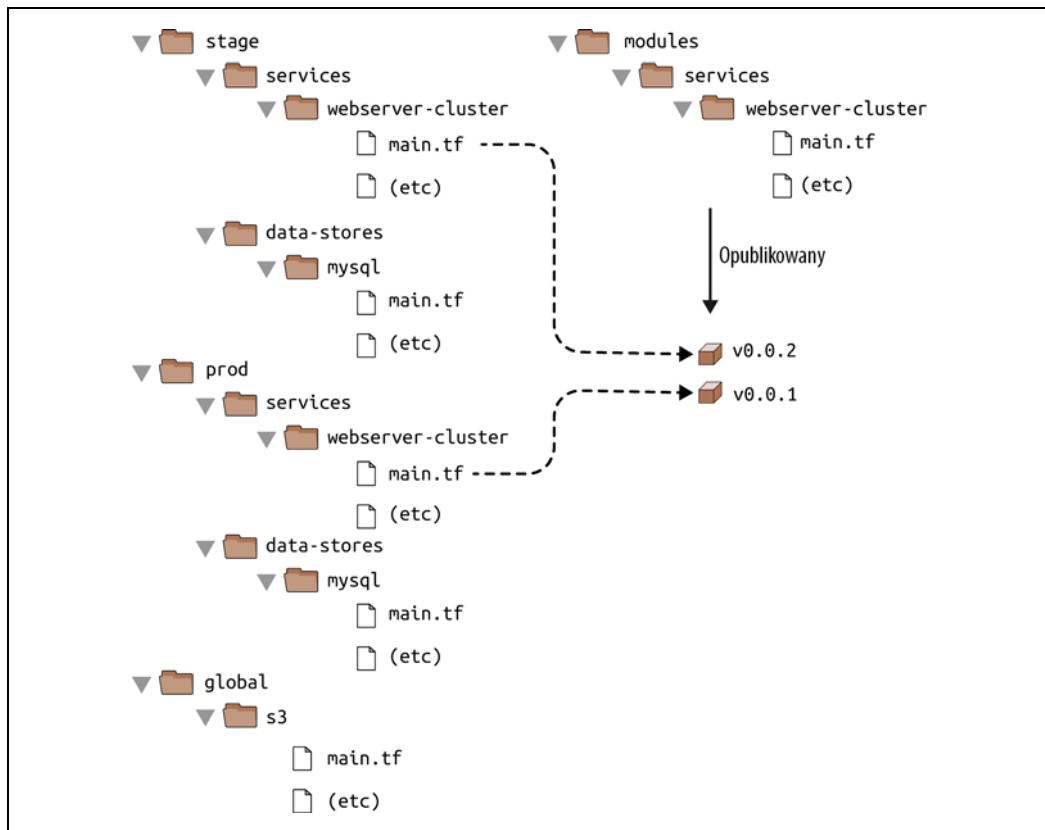
Dlatego też — z wyjątkiem prostych przykładów — w pozostałych projektach poszczególne środowiska powinny działać w oddzielnych VPC. Po prawdzie, jeśli chcesz mieć znacznie większy poziom bezpieczeństwa, te środowiska powinny działać w całkowicie oddzielnych kontach AWS.

Wersjonowanie modułu

Jeżeli środowiska robocze i produkcyjne prowadzą do tego samego katalogu modułu, zmiana wprowadzona w tym katalogu będzie miała wpływ na oba środowiska podczas następnego wdrożenia. Takie połączenie znacznie utrudnia przetestowanie zmiany w środowisku roboczym bez niebezpieczeństwa jej wpływu na środowisko produkcyjne. Zdecydowanie lepsze podejście polega na utworzeniu *modułów wersjonowanych*, co pozwoli na wykorzystanie jednej wersji w środowisku roboczym (np. v0.0.2) i innej w środowisku produkcyjnym (np. v0.0.1), jak pokazałem na rysunku 4.5.

We wszystkich przedstawionych dotychczas przykładach modułu, gdy był on używany, jego parametr `source` otrzymywał wartość w postaci ścieżki dostępu do pliku lokalnego. Poza ścieżkami dostępu do pliku Terraform obsługuje jeszcze inne rodzaje źródeł modułu, takie jak adresy URL Git, adresy URL Mercurial oraz dowolne adresy URL HTTP¹. Najłatwiejszym sposobem na utworzenie modułu wersjonowanego jest umieszczenie kodu modułu w oddzielnym repozytorium Git i przypisanie parametrowi `source` wartości w postaci adresu URL tego repozytorium. To oznacza podzielenie kodu Terraform między (przynajmniej) dwa repozytoria:

¹ Więcej informacji na temat obsługiwanych adresów URL znajdziesz na stronie <https://www.terraform.io/docs/modules/sources.html>.



Rysunek 4.5. Używanie różnych wersji modułu w odmiennych środowiskach

modules

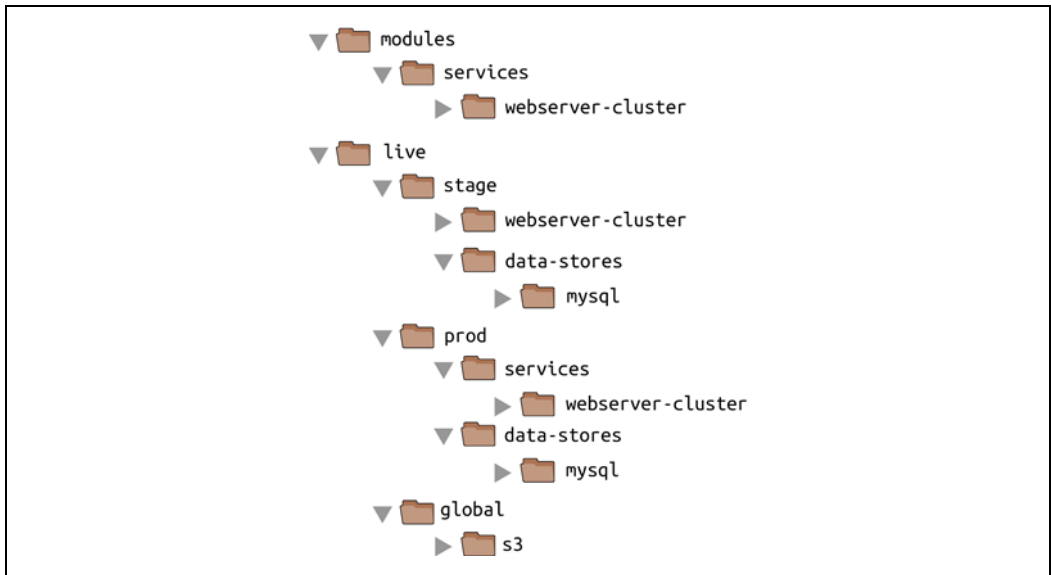
To repozytorium definiuje moduły wielokrotnego użycia. Każdy moduł potraktuj jako „matrycę” pozwalającą na zdefiniowanie określonego fragmentu infrastruktury.

live

To repozytorium definiuje działającą infrastrukturę, która jest wykorzystywana w poszczególnych środowiskach (roboczym, produkcyjnym, zarządzającym itd.). Potraktuj je jako „domy” zbudowane na podstawie „planów” znajdujących się w repozytorium *modules*.

Uaktualniona struktura katalogu kodu Terraform przedstawia się teraz podobnie do pokazanej na rysunku 4.6.

Aby przygotować tę strukturę katalogów, należy zacząć od przeniesienia katalogów *stage*, *prod* i *global* do katalogu o nazwie *live*. Następnym krokiem jest skonfigurowanie katalogów *live* i *modules* jako oddzielnych repozytoriów Git. Oto przykład procedury do przeprowadzenia na katalogu *modules*:



Rysunek 4.6. Układ plików wraz z wieloma repozytoriami

```
$ cd modules
$ git init
$ git add .
$ git commit -m "Pierwsze przekazanie plików do repozytorium modules"
$ git remote add origin "(ADRES URL ZDALNEGO REPOZYTORIUM)"
$ git push origin master
```

Do repozytorium *modules* można dodać również tag, który będzie użyty jako numer wersji. Jeżeli używasz serwisu GitHub do utworzenia wydania, możesz wykorzystać także interfejs użytkownika GitHub (<https://help.github.com/en/github/administering-a-repository/managing-releases-in-a-repository>), co spowoduje utworzenie tagu w tle. Jeśli natomiast nie używasz serwisu GitHub, możesz wykorzystać Git CLI.

```
$ git tag -a "v0.0.1" -m "Pierwsze wydanie modułu webserver-cluster"
$ git push --follow-tags
```

Teraz ten moduł wersjonowany może zostać użyty w środowiskach roboczym i produkcyjnym przez podanie adresu URL Git w parametrze *source*. Zobacz, jak można to zrobić dla *live/stage/services/webserver-cluster/main.tf*, gdy repozytorium *modules* znajduje się w repozytorium serwisu GitHub jako *git hub.com/foo/modules* (zwróć uwagę na konieczność użycia podwójnego ukośnika w adresie URL repozytorium Git).

```
module "webserver_cluster" {
  source = "github.com/foo/modules//webserver-cluster?ref=v0.0.1"

  cluster_name      = "webservers-stage"
  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"

  instance_type = "t2.micro"
  min_size      = 2
  max_size      = 2
}
```

Jeżeli chcesz wypróbować moduły wersjonowane bez konieczności mieszania w repozytoriach Git, zawsze możesz skorzystać z modułu znajdującego się w repozytorium GitHub przygotowanym dla tej książki (musałem podzielić ten adres URL, aby zmieścił się na stronie książki, ale w kodzie powinien się on znajdować w jednym wierszu).

```
source = "github.com/brikis98/terraform-up-and-running-code//  
code/terraform/04-terraform-module/module-example/modules/  
services/webserver-cluster?ref=v0.1.0"
```

Parametr `ref` pozwala na wskazanie konkretnej operacji przekazania plików do repozytorium za pomocą wartości `sha1` hash, nazwy gałęzi lub (jak w omawianym przykładzie) za pomocą określonego tagu Git. Ogólnie rzecz biorąc, zachęcam do używania tagów Git jako numerów wersji modułów. Nazwy gałęzi są niestabilne, ponieważ zawsze jest wykorzystywane ostatnie zatwierdzenie (ang. *commit*) w gałęzi, które może się zmieniać po każdym wydaniu polecenia `init`. Z kolei wartość `sha1` hash nie jest zbyt przyjazna dla człowieka. Natomiast tagi Git są stabilne jak zatwierdzenia (tag to właściwie wskaźnik prowadzący do operacji zatwierdzenia) i pozwalają na stosowanie przyjaznych, czytelnych nazw.

Szczególnie użyteczny schemat nazw dla tagów to tzw. *wersjonowanie semantyczne* (<https://semver.org/>). To jest schemat wersjonowania w formacie `WERSJA_GŁÓWNA.WERSJA_MNIEJSZA.WERSJA_POPRAWKI` (np. 1.0.4) wraz z konkretnymi regułami określającymi, kiedy należy zmienić numer wersji. Oto ogólne reguły inkrementacji numeru wersji:

- `WERSJA_GŁÓWNA` po wprowadzeniu zmian w API niezgodnych z poprzednią wersją,
- `WERSJA_MNIEJSZA` po dodaniu nowej funkcjonalności z zachowaniem zgodności z poprzednią wersją,
- `WERSJA_POPRAWKI` po usunięciu błędu z zachowaniem zgodności z poprzednią wersją.

Wersjonowanie semantyczne wskazuje użytkownikom modułu, jakiego rodzaju zmiany zostały wprowadzone i jakie implikacje może nieść uaktualnienie modułu.

Skoro uaktualniłeś kod Terraform do użycia wersjonowanego adresu URL modułu, musisz nakazać Terraform pobranie kodu modułu, co wymaga ponownego wydania polecenia `terraform init`:

```
$ terraform init  
Initializing modules...  
Downloading git@github.com:brikis98/terraform-up-and-running-code.git?ref=v0.1.0  
for webserver_cluster...
```

(...)

Tym razem wyraźnie widać, że Terraform pobiera kod modułu z repozytorium Git zamiast z lokalnego systemu plików. Po pobraniu kodu modułu można w zwykły sposób wydać polecenie `terraform apply`.

Skoro korzystasz z modułów wersjonowanych, zapoznaj się teraz z procesem wprowadzania zmian. Przyjmuję założenie o wprowadzeniu pewnych zmian w module `webserver-cluster`, które chcesz przetestować poza środowiskiem roboczym. Przede wszystkim trzeba te zmiany przekazać do repozytorium *modules*:



Prywatne repozytoria Git

Jeżeli kod modułu Terraform znajduje się w prywatnym repozytorium Git, użycie tego repozytorium jako źródła (source) modułu wymaga udzielenia Terraform możliwości uwierzytelnienia się w tym repozytorium Git. Zachęcam do wykorzystania SSH, tak unikniesz konieczności umieszczenia danych uwierzytelniających na stałe w kodzie repozytorium. Dzięki uwierzytelnianiu SSH każdy programista może utworzyć klucz SSH, powiązać go z użytkownikiem Git i następnie dodać do ssh-agent. Terraform automatycznie użyje tego klucza do uwierzytelniania, o ile korzystasz z SSH w adresie URL źródła².

Adres URL źródła (source) powinien być w następującej postaci:

```
git@github.com:<WŁAŚCICIEL>/<REPOZYTORIUM>.git//<ŚCIEŻKA>?ref=<WERSJA>
```

Przykładowo:

```
git@github.com:acme/modules.git//example?ref=v0.1.2
```

Aby upewnić się o prawidłowym sformatowaniu adresu URL, spróbuj w powłoce wykonać polecenie `git clone` wraz z bazowym adresem URL.

```
$ git clone git@github.com:<WŁAŚCICIEL>/<REPOZYTORIUM>.git
```

Jeżeli wykonanie tych poleceń zakończy się sukcesem, Terraform będzie mieć możliwość wykorzystania także prywatnego repozytorium Git.

```
$ cd modules
$ git add .
$ git commit -m "Wprowadzono pewne zmiany w module webserver-cluster"
$ git push origin master
```

Następnym krokiem jest utworzenie nowego tagu w repozytorium *modules*:

```
$ git tag -a "v0.0.2" -m "Drugie wydanie modułu webserver-cluster"
$ git push --follow-tags
```

Teraz, po uaktualnieniu *tylko* adresu URL źródła w środowisku roboczym (*live/stage/services/webserver-cluster/main.tf*), nowa wersja zostanie użyta.

```
module "webserver_cluster" {
  source = git@github.com:foo/modules.git//webserver-cluster?ref=v0.0.2

  cluster_name      = "webservers-stage"
  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"

  instance_type = "t2.micro"
  min_size      = 2
  max_size      = 2
}
```

W środowisku produkcyjnym (*live/prod/services/webserver-cluster/main.tf*) można kontynuować używanie niezmodyfikowanej wersji 0.0.1.

² Więcej informacji na temat pracy z kluczami SSH podczas uzyskiwania dostępu do repozytorium znajdziesz na stronie <https://help.github.com/en/github/authenticating-to-github/connecting-to-github-with-ssh>.

```
module "webserver_cluster" {
  source = git@github.com:foo/modules.git//webserver-cluster?ref=v0.0.1

  cluster_name      = "webservers-prod"
  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"

  instance_type = "m4.large"
  min_size      = 2
  max_size      = 10
}
```

Po dokładnym przetestowaniu wersji 0.0.2 i potwierdzeniu jej stabilności można uaktualnić także środowisko produkcyjne. Jeżeli okaże się, że wersja 0.0.2 zawiera błąd, nie stanowi to problemu, ponieważ ten błąd nie ma wpływu na użytkowników środowiska produkcyjnego. Usuń błąd, wydaj nową wersję i powtarzaj cały proces aż do chwili, gdy otrzymasz wydanie wystarczająco stabilne do użycia w środowisku produkcyjnym.



Opracowywanie modułów

Moduły wersjonowane są doskonałym rozwiązaniem, gdy opracowujesz je dla środowiska współdzielonego (np. roboczego lub produkcyjnego). Jednak podczas testowania modułu we własnym komputerze będziesz korzystać ze ścieżek dostępu do plików lokalnych. To pozwala na szybszą iterację, ponieważ można wprowadzać zmiany w katalogach modułu, a następnie wydawać polecenie `terraform plan` lub `terraform apply` i natychmiast sprawdzić wynik działania kodu — nie potrzeba operacji zatwierdzenia kodu i publikowania jego nowej wersji za każdym razem.

Skoro celem tej książki jest pomoc w poznawaniu i eksperymentowaniu z Terraform w możliwie jak najszybszy sposób, w pozostałych przykładach dla modułów będą wykorzystywane ścieżki dostępu do plików lokalnych.

Podsumowanie

Dzięki zdefiniowaniu infrastruktury jako kodu za pomocą modułów w infrastrukturze można stosować różne najlepsze praktyki z zakresu tworzenia oprogramowania. Każda zmiana w module może być przeanalizowana za pomocą operacji sprawdzenia kodu oraz przez testy zautomatyzowane. Otrzymujesz możliwość semantycznego tworzenia wersjonowanych wydań modułów oraz bezpiecznego wypróbowywania modułu w odmiennych środowiskach, a po napotkaniu problemu masz możliwość bezpiecznego przywrócenia poprzedniej wersji modułu.

To wszystko znacznie zwiększa Twoje możliwości w zakresie szybkiego tworzenia infrastruktury i jednocześnie poprawia niezawodność rozwiązania, ponieważ programiści zyskują możliwość wielokrotnego używania dokładnie sprawdzonej, przetestowanej i udokumentowanej infrastruktury. Przykładowo można przygotować moduł kanoniczny definiujący sposób wdrożenia pojedynczej mikrousługi — wraz z określonym sposobem uruchomienia klastra, skalowania klastra w reakcji na obciążenie, rozkładania ruchu sieciowego w klastrze itd. — a każdy członek zespołu będzie

mógł wykorzystać ten moduł do zarządzania własnymi mikrousługami. To będzie wymagało utworzenia zaledwie kilku wierszy kodu.

Aby taki moduł sprawdzał się w przypadku wielu zespołów, kod Terraform w tym module musi być elastyczny i konfigurowalny. Przykładowo jeden zespół korzystający z modułu może wdrażać mikrousługi tylko w pojedynczym egzemplarzu bez mechanizmu równoważenia obciążenia, podczas gdy inny zespół będzie miał wiele egzemplarzy dla mikrousług i mechanizm równoważenia obciążenia rozprowadzający ruch sieciowy między tymi egzemplarzami. Jak można tworzyć konstrukcje warunkowe w Terraform? Czy istnieje sposób na zdefiniowanie pętli w Terraform? Czy można go użyć do wprowadzenia zmian w mikrousługach bez powodowania przestoju usługi? Te zaawansowane aspekty składni Terraform będą tematem rozdziału 5.

Sztuczki i podpowiedzi dotyczące Terraform

— pętle, konstrukcje if, wdrażanie i problemy

Terraform to język deklaracyjny. Jak się dowiedziałeś z rozdziału 1., stosowanie praktyk IaC w języku deklaracyjnym z reguły oznacza przedstawienie znacznie dokładniejszych informacji o tym, co faktycznie zostało wdrożone. Otrzymane informacje są dużo bardziej prawidłowe niż w przypadku języka proceduralnego, więc łatwiej jest uzasadnić stosowanie języka deklaracyjnego. Dodatkową korzyścią płynącą z jego użycia jest mniejsza baza kodu. Jednak pewnego typu zadania są znacznie trudniejsze niż w języku proceduralnym.

Przykładowo język deklaracyjny zwykle nie ma pętli `for`, więc być może zastanawiasz się, jak można powtarzać pewne fragmenty logiki — jak tworzenie wielu podobnych zasobów — bez konieczności kopiowania i wklejania kodu. Podobnie, skoro język deklaracyjny nie obsługuje konstrukcji `if`, jak można warunkowo konfigurować zasoby, np. przygotowanie modułu Terraform pozwalającego na tworzenie pewnych zasobów dla wybranych użytkowników modułu, ale nie dla pozostałych? Jak w języku deklaracyjnym wyrazić ideę z natury proceduralną, np. wdrożenie bez przestoju?

Na szczęście Terraform oferuje konstrukcje — m.in. metaparametr `count`, wyrażenia `for_each` i `for`, blok cyklu życiowego o nazwie `create_before_destroy`, operator trójargumentowy oraz ogromną liczbę funkcji — pozwalające na definiowanie określonych typów pętli, konstrukcji `if` i przeprowadzanie wdrożenia bez przestoju. Oto krótka lista tematów poruszonych w rozdziale:

- pętle,
- konstrukcje warunkowe,
- wdrożenie bez przestoju,
- problemy związane z Terraform.



Przykładowe fragmenty kodu

Przypominam, że wszystkie przykładowe fragmenty kodu znajdują się w repozytorium GitHub pod adresem <https://github.com/brikis98/terraform-up-and-running-code>.

Pętle

Terraform oferuje kilka różnych konstrukcji przeznaczonych do definiowania pętli, a każda z nich jest do zastosowania w nieco odmiennych sytuacjach:

- parametr `count` do przeprowadzania iteracji przez zasoby,
- wyrażenia `for_each` do przeprowadzania iteracji przez zasoby i osadzone bloki kodu w zasobie,
- wyrażenia `for` do przeprowadzania iteracji przez listy i mapowania,
- dyrektywa ciągu tekstowego `for` do przeprowadzania iteracji przez listy i mapowania w ciągu tekstowym.

Przeanalizujemy je po kolei.

Pętla za pomocą parametru `count`

W rozdziale 2. utworzyłeś użytkownika IAM (*AWS Identity and Access Management*) za pomocą graficznego interfejsu użytkownika w konsoli AWS. Mając tego użytkownika, możesz z poziomu kodu Terraform tworzyć kolejnych użytkowników IAM i nimi zarządzać. Spójrz na przedstawiony tutaj fragment kodu Terraform, który powinien znajdować się w pliku `live/global/iam/main.tf`:

```
provider "aws" {
  region = "us-east-2"
}

resource "aws_iam_user" "example" {
  name = "neo"
}
```

Ten kod używa zasobu `aws_iam_user` do utworzenia nowego pojedynczego użytkownika IAM. Co w sytuacji, gdy będziesz chciał utworzyć trzech użytkowników IAM? W języku programowania ogólnego przeznaczenia prawdopodobnie skorzystasz z pętli `for`.

```
# To jest tylko pseudokod i nie będzie działał w Terraform.
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = "neo"
  }
}
```

Terraform nie ma wbudowanej obsługi pętli `for` oraz innej tradycyjnej logiki proceduralnej, więc przedstawiona w tym fragmencie kodu składnia po prostu nie działa. Jednak każdy zasób Terraform ma metaparametr o nazwie `count`. To jest najstarsza, najprostsza i najbardziej ograniczona konstrukcja w Terraform — pozwala zdefiniować liczbę kopii zasobu do utworzenia. Spójrz na przykład użycia tej konstrukcji do utworzenia trzech użytkowników IAM.

```
resource "aws_iam_user" "example" {
  count = 3
  name = "neo"
}
```

Problem związany z tym kodem polega na tym, że wszystkich trzech użytkowników IAM będzie miało tę samą nazwę, co spowoduje wygenerowanie błędu, ponieważ nazwy użytkowników muszą być

unikatowe. Jeżeli miałbyś dostęp do standardowej pętli `for`, mógłbyś użyć indeksu pętli, np. `i`, i nadać każdemu użytkownikowi unikatową nazwę.

```
# To jest tylko pseudokod i nie będzie działał w Terraform.
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = "neo.${i}"
  }
}
```

To samo zadanie w Terraform można wykonać za pomocą `count.index`, aby w ten sposób pobrać indeks każdej „iteracji” w „pętli”.

```
resource "aws_iam_user" "example" {
  count = 3
  name  = "neo.${count.index}"
}
```

Jeżeli wydasz polecenie `terraform plan` dla poprzedniego fragmentu kodu, zobaczysz, że Terraform chce utworzyć trzech użytkowników IAM, każdego z inną nazwą (tutaj "neo", "morpheus", "trinity").

Terraform will perform the following actions:

```
# Utworzony będzie zasób aws_iam_user.example[0].
+ resource "aws_iam_user" "example" {
+   arn                = (known after apply)
+   force_destroy      = false
+   id                 = (known after apply)
+   name               = "neo.0"
+   path               = "/"
+   unique_id          = (known after apply)
+ }

# Utworzony będzie zasób aws_iam_user.example[1].
+ resource "aws_iam_user" "example" {
+   arn                = (known after apply)
+   force_destroy      = false
+   id                 = (known after apply)
+   name               = "neo.1"
+   path               = "/"
+   unique_id          = (known after apply)
+ }

# Utworzony będzie zasób aws_iam_user.example[2].
+ resource "aws_iam_user" "example" {
+   arn                = (known after apply)
+   force_destroy      = false
+   id                 = (known after apply)
+   name               = "neo.2"
+   path               = "/"
+   unique_id          = (known after apply)
+ }
```

Plan: 3 to add, 0 to change, 0 to destroy.

Oczywiście nazwa użytkownika w postaci `neo.0` nie należy do szczególnie użytecznych. Jeżeli połączysz `count.index` z wbudowanymi funkcjami Terraform, uzyskasz możliwość dalszego dostosowywania każdej „iteracji pętli” do własnych potrzeb.

Przykładowo możesz zdefiniować nazwy wszystkich użytkowników IAM w zmiennej danych wejściowych w pliku *live/global/iam/variables.tf*.

```
variable "user_names" {
  description = "Utworzenie użytkowników IAM o podanych nazwach"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

Jeżeli korzystasz z języka programowania ogólnego przeznaczenia oferującego pętle i tablice, to poszczególnych użytkowników IAM możesz skonfigurować przez wyszukanie indeksu *i* w tablicy *var.user_names*.

```
# To jest tylko pseudokod i nie będzie działał w Terraform.
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = vars.user_names[i]
  }
}
```

Jeżeli to samo zadanie chcesz wykonać w Terraform, musisz skorzystać z parametru *count* wraz z:

Składnią wyszukiwania w tablicy

Składnia przeznaczona do wyszukiwania elementów tablicy jest w Terraform podobna do stosowanej w większości innych języków programowania:

```
TABLICA[<INDEKS>]
```

Przykładowo to jest polecenie przeznaczone do pobrania z *var.user_names* elementu o indeksie 1.

```
var.user_names[1]
```

Funkcję length()

Terraform ma funkcję wbudowaną o nazwie *length()*, do której wywołania jest stosowana następująca składnia:

```
length(<TABLICA>)
```

Jak prawdopodobnie się domyśliłeś, ta funkcja zwraca liczbę elementów znajdujących się w podanej tablicy. Funkcja *length()* działa wraz z ciągami tekstowymi i mapowaniami.

Po połączeniu wszystkiego otrzymujesz następujący fragment kodu:

```
resource "aws_iam_user" "example" {
  count = length(var.user_names)
  name  = var.user_names[count.index]
}
```

Teraz po wydaniu polecenia *terraform plan* zobaczysz, że Terraform chce utworzyć trzech użytkowników IAM o unikatowych nazwach.

Terraform will perform the following actions:

```
# Utworzony będzie zasób aws_iam_user.example[0].
+ resource "aws_iam_user" "example" {
+   arn          = (known after apply)
+   force_destroy = false
+   id           = (known after apply)
```



```
+ name          = "neo"
+ path          = "/"
+ unique_id     = (known after apply)
}
```

Utworzony będzie zasób *aws_iam_user.example[1]*.

```
+ resource "aws_iam_user" "example" {
+   arn          = (known after apply)
+   force_destroy = false
+   id          = (known after apply)
+   name        = "trinity"
+   path        = "/"
+   unique_id   = (known after apply)
}
```

Utworzony będzie zasób *aws_iam_user.example[2]*.

```
+ resource "aws_iam_user" "example" {
+   arn          = (known after apply)
+   force_destroy = false
+   id          = (known after apply)
+   name        = "morpheus"
+   path        = "/"
+   unique_id   = (known after apply)
}
```

Plan: 3 to add, 0 to change, 0 to destroy.

Zauważ, że po użyciu `count` wraz z zasobem mamy do czynienia z tablicą zasobów, a nie tylko z pojedynczym zasobem. Skoro `aws_iam_user.example` to teraz tablica użytkowników IAM, to zamiast standardowej składni odczytu atrybutu z zasobu (`<DOSTAWCA>_<TYP>.<NAZWA>.<ATRYBUT>`) musisz wskazać interesującego Cię użytkownika IAM przez określenie jego indeksu w tablicy za pomocą tej samej składni wyszukiwania w tablicy.

`<DOSTAWCA>_<TYP>.<NAZWA>[INDEKS].ATRYBUT`

Przykładowo, jeśli chcesz dostarczyć ARN (ang. *amazon resource name*) jednego z użytkowników IAM jako zmienną danych wejściowych, musisz zapisać to w pokazany tutaj sposób:

```
output "neo_arn" {
  value     = aws_iam_user.example[0].arn
  description = "Wartość ARN dla użytkownika Neo"
}
```

Jeśli natomiast chcesz dostarczyć wartości ARN dla *wszystkich* użytkowników IAM, musisz zamiast z indeksu skorzystać z tzw. *wyrażenia splat*, `*`, w Terraform:

```
output "all_arns" {
  value     = aws_iam_user.example[*].arn
  description = "Wartości ARNs dla wszystkich użytkowników"
}
```

Po wydaniu polecenia `terraform apply` dane wyjściowe `neo_arn` będą zawierały wartość ARN dla użytkownika Neo, natomiast dane wyjściowe `all_arns` — listę wszystkich wartości ARN:

```
$ terraform apply
```

```
(...)
```

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

```
neo_arn = arn:aws:iam::123456789012:user/neo
all_arns = [
  "arn:aws:iam::123456789012:user/neo",
  "arn:aws:iam::123456789012:user/trinity",
  "arn:aws:iam::123456789012:user/morpheus",
]
```

Niestety, parametr `count` ma dwa ograniczenia znacznie zmniejszające jego użyteczność. Pierwsze: pomimo możliwości użycia `count` do iteracji przez cały zasób tego parametru nie można wykorzystać do iteracji przez bloki osadzone. *Blok osadzony* to definiowany w zasobie argument o następującym formacie:

```
resource "xxx" "yyy" {
  <NAZWA> {
    [KONFIGURACJA...]
  }
}
```

gdzie `NAZWA` jest nazwą bloku osadzonego (np. `tag`), a `KONFIGURACJA` to jeden lub więcej argumentów charakterystycznych dla tego bloku (np. `key` i `value`). Dla przykładu spójrz na sposób definiowania tagów w zasobie `aws_autoscaling_group`:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier   = data.aws_subnet_ids.default.ids
  target_group_arns     = [aws_lb_target_group.asg.arn]
  health_check_type     = "ELB"

  min_size = var.min_size
  max_size = var.max_size

  tag {
    key           = "Name"
    value         = var.cluster_name
    propagate_at_launch = true
  }
}
```

Każdy tag wymaga utworzenia nowego bloku osadzonego wraz z wartościami `key`, `value` i `propagate_at_launch`. Przedstawiony tutaj fragment kodu miał na stałe zdefiniowane dane dla pojedynczego tagu, ale być może chciałbyś pozwolić użytkownikom na przekazywanie własnych tagów. Być może za kuszące uznasz wykorzystanie parametru `count` do iteracji przez te tagi i dynamicznego generowania osadzonych bloków tag. Jednak używanie parametru `count` w blokach osadzonych jest nieobsługiwane.

Drugie ograniczenie `count` wiąże się z tym, co się dzieje podczas próby wprowadzenia zmiany. Spójrz na utworzoną wcześniej listę użytkowników IAM.

```
variable "user_names" {
  description = "Utworzenie użytkowników IAM o podanych nazwach"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

Przyjmujemy założenie, że z listy usunięto "trinity". Co się stanie po wydaniu polecenia terraform plan?

```
$ terraform plan
```

```
(...)
```

Terraform will perform the following actions:

```
# Zasób aws_iam_user.example[1] zostanie uaktualniony w miejscu.
~ resource "aws_iam_user" "example" {
  id      = "trinity"
  ~ name   = "trinity" -> "morpheus"
}

# Zasób aws_iam_user.example[2] zostanie usunięty.
- resource "aws_iam_user" "example" {
  - id      = "morpheus" -> null
  - name    = "morpheus" -> null
}
```

Plan: 0 to add, 1 to change, 1 to destroy.

Chwila! To prawdopodobnie nie jest efekt, który chciałeś otrzymać. Zamiast po prostu usunąć użytkownika IAM "trinity", dane wyjściowe wskazują na zmianę nazwy tego użytkownika IAM na "morpheus" i usunięcie pierwotnego użytkownika "morpheus". Co się tutaj dzieje?

Gdy używasz parametru count wraz z zasobem, staje się on listą, czyli tablicą zasobów. Poszczególne zasoby w tablicy są przez Terraform wskazywane za pomocą ich położenia (indeksu) w tej tablicy. Dlatego też po wykonaniu polecenia terraform apply za pierwszym razem wraz z trzema nazwami użytkowników wewnętrzna reprezentacja tych użytkowników IAM w Terraform przedstawia się następująco:

```
aws_iam_user.example[0]: neo
aws_iam_user.example[1]: trinity
aws_iam_user.example[2]: morpheus
```

Po usunięciu elementu z środka tablicy wszystkie elementy znajdujące się po nim są przesunięte do przodu o jedno położenie. Dlatego też po wykonaniu polecenia terraform plan wraz z dwiema nazwami kubełków wewnętrzna reprezentacja tych użytkowników IAM w Terraform będzie wyglądała następująco:

```
aws_iam_user.example[0]: neo
aws_iam_user.example[1]: morpheus
```

Zauważ przesunięcie użytkownika morpheus z położenia o indeksie 2 do położenia o indeksie 1. Skoro indeks jest traktowany jako identyfikator zasobu, dla Terraform to oznacza mniej więcej „nazwę kubełka o indeksie 1 zmień na morpheus i usuń kubełek znajdujący się w indeksie 2”. Innymi słowy, za każdym razem, gdy parametr count jest używany do utworzenia listy zasobów, jeśli usuniesz element z środka listy, Terraform usunie wszystkie znajdujące się po nim zasoby, a następnie odtworzy je zupełnie od początku. Au! Ostateczny wynik jest oczywiście dokładnie taki, jakiego oczekiwałeś (dwóch użytkowników IAM o nazwach morpheus i neo), natomiast usunięcie i modyfikacja zasobu to prawdopodobnie to, czego tutaj nie chciałeś.

W celu usunięcia dwóch wspomnianych wcześniej ograniczeń w Terraform 0.12 wprowadzono wyrażenia `for_each`.

Pętla za pomocą wyrażenia `for_each`

Wyrażenie `for_each` pozwala na iterację przez listę, zbiór i mapowanie w celu utworzenia (a) wielu kopii całego zasobu lub (b) wielu kopii bloku osadzonego w zasobie. Najpierw przedstawię użycie `for_each` do utworzenia wielu kopii zasobu. Składnia wygląda następująco:

```
resource "<DOSTAWCA> <TYP>" "<NAZWA>" {
  for_each = <KOLEKCJA>

  [KONFIGURACJA ...]
}
```

gdzie `DOSTAWCA` to nazwa dostawcy (np. `aws`), `TYP` to nazwa zasobu przeznaczanego do utworzenia w tym dostawcy (np. `instance`), `NAZWA` to identyfikator używany w kodzie Terraform w celu odwołania się do tego zasobu (np. `my_instance`), `KOLEKCJA` to iterowany zbiór lub mapowanie (listy nie są obsługiwane podczas stosowania `for_each` w zasobie), a `KONFIGURACJA` składa się z jednego lub więcej argumentów charakterystycznych dla tego zasobu. W elemencie `KONFIGURACJA` można używać `each.key` i `each.value` w celu uzyskania dostępu do klucza i wartości aktualnego elementu `KOLEKCJA`.

Dla przykładu spójrz na sposób utworzenia tych samych trzech użytkowników IAM, ale za pomocą wyrażenia `for_each`:

```
resource "aws_iam_user" "example" {
  for_each = toset(var.user_names)
  name     = each.value
}
```

Zwróć uwagę na użycie `toset` w celu konwersji listy `var.user_list` na zbiór, ponieważ gdy jest stosowane w zasobie, wyrażenie `for_each` obsługuje jedynie zbiory i mapowania. Podczas iteracji wyrażenia `for_each` przez ten zbiór każda nazwa użytkownika zostaje udostępniona w `each.value`. Nazwa użytkownika będzie również dostępna w `each.key`, choć `each.key` najczęściej używa się w mapowaniach zawierających pary klucz-wartość.

Po użyciu `for_each` w zasobie staje się on mapowaniem zasobów, a nie tylko jednym zasobem (lub tablicą zasobów w przypadku użycia parametru `count`). Aby zobaczyć, co to oznacza, usuń pierwotne zmienne danych wyjściowych `all_arns` i `neo_arn`, a następnie dodaj nową zmienną danych wyjściowych o nazwie `all_users`.

```
output "all_users" {
  value = aws_iam_user.example
}
```

Oto co się stanie po wydaniu polecenia `terraform apply`:

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```

all_users = {
  "morpheus" = {
    "arn" = "arn:aws:iam::123456789012:user/morpheus"
    "force_destroy" = false
    "id" = "morpheus"
    "name" = "morpheus"
    "path" = "/"
    "tags" = {}
  }
  "neo" = {
    "arn" = "arn:aws:iam::123456789012:user/neo"
    "force_destroy" = false
    "id" = "neo"
    "name" = "neo"
    "path" = "/"
    "tags" = {}
  }
  "trinity" = {
    "arn" = "arn:aws:iam::123456789012:user/trinity"
    "force_destroy" = false
    "id" = "trinity"
    "name" = "trinity"
    "path" = "/"
    "tags" = {}
  }
}

```

Masz potwierdzenie utworzenia przez Terraform trzech użytkowników IAM. Zmienna danych wyjściowych `all_users` zawiera mapowanie, którego klucze odpowiadają kluczom `for_each` (w omawianym przykładzie to nazwy użytkowników), natomiast wartościami są wszystkie dane wyjściowe dla zasobu. Jeżeli chcesz przywrócić zmienną danych wyjściowych `all_arns`, musisz wykonać trochę więcej dodatkowej pracy i wyodrębnić te wartości ARN za pomocą funkcji wbudowanej `values()` — która zwraca jedynie wartości z mapowania — i wyrażenia `splat`.

```

output "all_arns" {
  value = values(aws_iam_user.example)[*].arn
}

```

Dzięki temu otrzymujesz oczekiwane dane wyjściowe:

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```

all_arns = [
  "arn:aws:iam::123456789012:user/morpheus",
  "arn:aws:iam::123456789012:user/neo",
  "arn:aws:iam::123456789012:user/trinity",
]

```

Otrzymanie dzięki wyrażeniu `for_each` mapowania zasobów zamiast tablicy zasobów, jak ma to miejsce w przypadku użycia parametru `count`, ma duże znaczenie, ponieważ pozwala to na bezpieczne usuwanie elementów z środka kolekcji. Przykładowo, jeśli ponownie usuniesz użytkownika "trinity"

z środka listy `var.user_names` i wydasz polecenie `terraform plan`, otrzymasz następujące dane wyjściowe:

```
$ terraform plan
```

```
Terraform will perform the following actions:
```

```
# Zasób aws_iam_user.example["trinity"] zostanie usunięty.
- resource "aws_iam_user" "example" {
  - arn      = "arn:aws:iam::123456789012:user/trinity" -> null
  - name    = "trinity" -> null
}
```

```
Plan: 0 to add, 0 to change, 1 to destroy.
```

O to właśnie chodziło! Teraz usuwasz dokładnie ten zasób, którego chciałeś się pozbyć, pozostałe zaś nie będą przesunięte. Dlatego też przy tworzeniu wielu kopii zasobów praktycznie zawsze należy preferować wyrażenie `for_each` zamiast parametru `count`.

Zwróć uwagę na inną zaletę wyrażenia `for_each`: możliwość utworzenia wielu bloków osadzonych w zasobie. Przykładowo za pomocą wyrażenia `for_each` można dynamicznie generować osadzone bloki tag dla grupy ASG w module `webserver-cluster`. Trzeba zacząć od umożliwienia użytkownikom zdefiniowania własnych tagów. W tym celu dodaj do `modules/services/webserver-cluster/variables.tf` nową zmienną danych wejściowych mapowania o nazwie `custom_tags`.

```
variable "custom_tags" {
  description = "Własne tagi przeznaczone do użycia w egzemplarzach ASG"
  type        = map(string)
  default     = {}
}
```

Następnym krokiem jest zdefiniowanie tagów w środowisku produkcyjnym, `live/prod/services/webserver-cluster/main.tf`, w przedstawiony tutaj sposób:

```
module "webserver_cluster" {
  source = "../../../../../modules/services/webserver-cluster"

  cluster_name      = "webservers-prod"
  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"

  instance_type      = "m4.large"
  min_size           = 2
  max_size           = 10
  enable_autoscaling = true

  custom_tags = {
    Owner       = "team-foo"
    DeployedBy = "terraform"
  }
}
```

W tym kodzie zostały zdefiniowane dwa użyteczne tagi: `Owner` określa zespół będący właścicielem tej grupy ASG, natomiast `DeployedBy` określa, że dana infrastruktura została wdrożona za pomocą Terraform. (To wskazuje, że ta infrastruktura nie powinna być modyfikowana ręcznie, do czego jeszcze wrócę w dalszej części rozdziału). Z reguły dobrym rozwiązaniem jest przygotowanie standardu

stosowania tagów w zespole i utworzenie modułów Terraform wymagających trzymania się tego standardu.

Po zdefiniowaniu tagów pozostało już tylko ich rzeczywiste zastosowanie w zasobie `aws_autoscaling_group`. Jak można to zrobić? Konieczne jest wykorzystanie pętli typu `for` do iteracji przez `var.custom_tags`, podobnie jak w przedstawionym tutaj pseudokodzie:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"

  min_size = var.min_size
  max_size = var.max_size

  tag {
    key           = "Name"
    value         = var.cluster_name
    propagate_at_launch = true
  }

  # To jest tylko pseudokod i nie będzie działał w Terraform.
  for (tag in var.custom_tags) {
    tag {
      key           = tag.key
      value         = tag.value
      propagate_at_launch = true
    }
  }
}
```

Ten pseudokod nie działa, ale wyrażenie `for_each` spełni swoje zadanie. Składnia pozwalająca na zastosowanie `for_each` do dynamicznego generowania bloków osadzonych przedstawia się następująco:

```
dynamic "<NAZWA_ZMIENNEJ>" {
  for_each = <KOLEKCJA>

  content {
    [KONFIGURACJA...]
  }
}
```

gdzie `NAZWA_ZMIENNEJ` to nazwa zmiennej przechowującej wartość każdej „iteracji” (zamiast `each`), `KOLEKCJA` to iterowana lista lub mapowanie, natomiast `content` to blok generowany w trakcie poszczególnych iteracji. W bloku `content` istnieje możliwość użycia `<NAZWA_ZMIENNEJ>.key` i `<NAZWA_ZMIENNEJ>.value` w celu uzyskania dostępu do odpowiednio klucza i wartości bieżącego elementu `KOLEKCJA`. Gdy używasz `for_each` wraz z listą, `key` będzie indeksem, natomiast `value` elementem listy znajdującym się w położeniu o podanym indeksie. Z kolei w przypadku używania `for_each` z mapowaniem `key` i `value` to jedna para klucz-wartość w mapowaniu.

Po połączeniu wszystkiego kolejny fragment kodu pokazuje, jak można dynamicznie generować bloki `tag` za pomocą wyrażenia `for_each` w zasobie `aws_autoscaling_group`:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
```

```

vpc_zone_identifier = data.aws_subnet_ids.default.ids
target_group_arns   = [aws_lb_target_group.asg.arn]
health_check_type   = "ELB"

min_size = var.min_size
max_size = var.max_size

tag {
  key          = "Name"
  value        = var.cluster_name
  propagate_at_launch = true
}

dynamic "tag" {
  for_each = var.custom_tags

  content {
    key          = tag.key
    value        = tag.value
    propagate_at_launch = true
  }
}

```

Jeżeli teraz wydasz polecenie `terraform apply`, powinieneś otrzymać plan podobny do następującego:

\$ terraform apply

Terraform will perform the following actions:

Zasób `aws_autoscaling_group.example` zostanie uaktualniony w miejscu.

```

~ resource "aws_autoscaling_group" "example" {
  (...)

    tag {
      key          = "Name"
      propagate_at_launch = true
      value        = "webserver-prod"
    }
+ tag {
+   key          = "Owner"
+   propagate_at_launch = true
+   value        = "team-foo"
+ }
+ tag {
+   key          = "DeployedBy"
+   propagate_at_launch = true
+   value        = "terraform"
+ }
}

```

Plan: 0 to add, 1 to change, 0 to destroy.

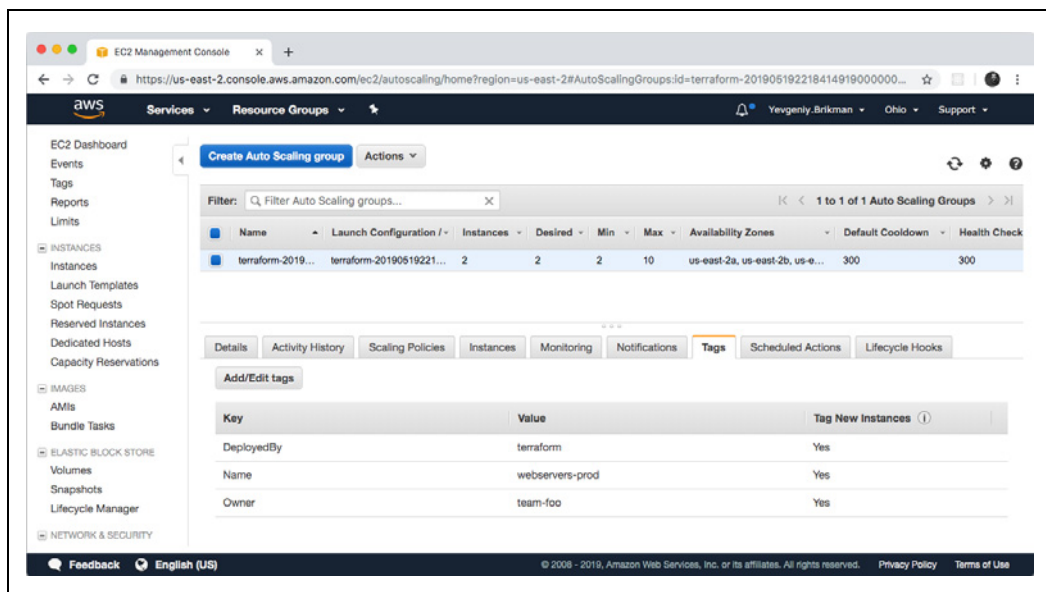
Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

Enter a value:

Wpisz **yes**, aby wdrożyć zmiany. Nowe tagi powinieneś zobaczyć w konsoli EC2, jak pokazałem na rysunku 5.1.



Rysunek 5.1. Dynamiczne tagi w automatycznie skalowanej grupie

Pętla za pomocą wyrażenia for

Dotychczas dowiedziałeś się, jak przeprowadzać iteracje przez zasoby i bloki osadzone. Być może zastanawiasz się, co jest potrzebne do wygenerowania pojedynczej wartości. Przejdźmy na chwilę do przykładów niezwiązanych z klastrem serwera WWW. Wyobraź sobie, że utworzyłeś kod Terraform pobierający listę nazw użytkowników.

```
variable "names" {
  description = "Lista nazw użytkowników"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

W jaki sposób można skonwertować je na zapisane wielkimi literami? W językach programowania ogólnego przeznaczenia, np. w Pythonie, można utworzyć następującą pętlę for:

```
names = ["neo", "trinity", "morpheus"]

upper_case_names = []
for name in names:
    upper_case_names.append(name.upper())

print upper_case_names

# Dane wyjściowe: ['NEO', 'TRINITY', 'MORPHEUS'].
```

Za pomocą składni nazywanej *listą składaną* Python oferuje jeszcze inny sposób na utworzenie kodu działającego w dokładnie ten sam sposób, ale mieszczącego się w jednym wierszu:

```
names = ["neo", "trinity", "morpheus"]

upper_case_names = [name.upper() for name in names]

print upper_case_names

# Dane wyjściowe: ['NEO', 'TRINITY', 'MORPHEUS'].
```

Python umożliwia również filtrowanie listy wynikowej przez zdefiniowanie warunku:

```
names = ["neo", "trinity", "morpheus"]

short_upper_case_names = [name.upper() for name in names if len(name) < 5]

print short_upper_case_names

# Dane wyjściowe: ['NEO'].
```

Terraform oferuje podobną funkcjonalność w postaci wyrażenia `for` (nie należy go mylić z wyrażeniem `for_each`, które poznałeś we wcześniejszej części rozdziału). Podstawowa składnia wyrażenia `for` przedstawia się następująco:

```
[for <ELEMENT> in <LISTA> : <DANE_WYJŚCIOWE>]
```

gdzie `LISTA` to iterowana lista, `ELEMENT` to nazwa zmiennej lokalnej przypisywanej każdemu elementowi `LISTA`, a `DANE_WYJŚCIOWE` to wyrażenie przekształcające `ELEMENT` w pewien sposób. Dla przykładu spójrz na kod Terraform przeznaczony do konwersji listy nazw użytkowników `var.names` na zapisane wielkimi literami:

```
variable "names" {
  description = "Lista nazw użytkowników"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}

output "upper_names" {
  value = [for name in var.names : upper(name)]
}
```

Po wydaniu polecenia `terraform apply` otrzymasz następujące dane wyjściowe:

```
$ terraform apply

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

upper_names = [
  "NEO",
  "TRINITY",
  "MORPHEUS",
]
```

Podobnie jak w przypadku listy składanej w Pythonie, także w Terraform można filtrować wyniki przez zdefiniowanie warunku:

```
variable "names" {
  description = "Lista nazw użytkowników"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
output "short_upper_names" {
  value = [for name in var.names : upper(name) if length(name) < 5]
}
```

Po wydaniu polecenia `terraform apply` otrzymasz następujące dane wyjściowe:

```
short_upper_names = [
  "NEO",
]
```

Wyrażenie `for` w Terraform pozwala również na iterację przez mapowanie, co wymaga użycia przedstawionej tutaj składni:

```
[for <KLUCZ>, <WARTOŚĆ> in <MAPOWANIE> : <DANE_WYJŚCIOWE>]
```

W tym przypadku `MAPOWANIE` wskazuje iterowane mapowanie, `KLUCZ` i `WARTOŚĆ` to nazwy zmiennych lokalnych przypisywane w każdej parze klucz-wartość elementu `MAPOWANIE`, natomiast `DANE_WYJŚCIOWE` to wyrażenie, które w pewien sposób przekształca `KLUCZ` i `WARTOŚĆ`. Spójrz na przykład:

```
variable "hero_thousand_faces" {
  description = "map"
  type        = map(string)
  default     = {
    neo      = "bohaterem"
    trinity  = "zakochana"
    morpheus = "mentorem"
  }
}

output "bios" {
  value = [for name, role in var.hero_thousand_faces : "${name} jest ${role}"]
}
```

Po wydaniu polecenia `terraform apply` otrzymasz następujące dane wyjściowe:

```
map_example = [
  "morpheus jest mentorem",
  "neo jest bohaterem",
  "trinity jest zakochana",
]
```

Wyrażenie `for` może być używane również do wyświetlenia danych wyjściowych mapowania zamiast listy, co wymaga zastosowania przedstawionej tutaj składni:

```
# Iteracja przez listę.
{for <ELEMENT> in <MAPOWANIE> : <KLUCZ_DANYCH_WYJŚCIOWYCH> => <WARTOŚĆ_DANYCH_WYJŚCIOWYCH>}

# Iteracja przez mapowanie.
{for <KLUCZ>, <WARTOŚĆ> in <MAPOWANIE> : <KLUCZ_DANYCH_WYJŚCIOWYCH> =>
<WARTOŚĆ_DANYCH_WYJŚCIOWYCH>}
```

Jedynie różnice polegają na (a) umieszczeniu wyrażenia w nawiasie klamrowym zamiast w kwadrato-wym, (b) wyświetleniu rozdzielonych strzałką klucza i wartości zamiast wyświetlenia tylko pojedynczej

wartości. Dla przykładu zobacz, jak można przekształcić mapowanie w taki sposób, aby wszystkie klucze i wartości były zapisane wielkimi literami:

```
variable "hero_thousand_faces" {
  description = "map"
  type        = map(string)
  default     = {
    neo      = "bohater"
    trinity  = "zakochana"
    morpheus = "mentor"
  }
}

output "upper_roles" {
  value = {for name, role in var.hero_thousand_faces : upper(name) => upper(role)}
}
```

Oto wynik wykonania tego fragmentu kodu:

```
upper_roles = {
  "MORPHEUS" = "MENTOR"
  "NEO"      = "BOHATER"
  "TRINITY"  = "ZAKOCHANA"
}
```

Pętla za pomocą dyrektywy for ciągu tekstowego

Z wcześniejszej części książki dowiedziałeś się o istnieniu interpolacji ciągu tekstowego, co pozwala na odwoływanie się do kodu Terraform z poziomu ciągu tekstowego:

```
"Witaj, ${var.name}"
```

Dyrektywa ciągu tekstowego pozwala na wykorzystanie poleceń kontrolnych (np. pętli for i konstrukcji if) w ciągu tekstowym za pomocą składni podobnej do interpolacji ciągu tekstowego. Jednak zamiast znaku dolara i nawiasu klamrowego, `${...}`, używany jest znak procenta i nawias klamrowy, `%{...}`.

Terraform obsługuje dwa rodzaje dyrektyw ciągu tekstowego: pętlę for i wyrażenia warunkowe. W tej sekcji zamierzam zająć się pętlą for, natomiast do wyrażen warunkowych powrócę w dalszej części rozdziału. Dyrektywa ciągu tekstowego for stosuje następującą składnię:

```
%{ for <ELEMENT> in <KOLEKCJA> }<TREŚĆ>%{ endfor }
```

gdzie KOLEKCJA to iterowana lista lub mapowanie, ELEMENT to nazwa zmiennej lokalnej do przypisania każdemu elementowi KOLEKCJI, a TREŚĆ to treść generowana podczas każdej iteracji (może się odwoływać do ELEMENTU). Spójrz na przedstawiony tutaj przykład.

```
variable "names" {
  description = "Imiona do wygenerowania"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}

output "for_directive" {
  value = <<EOF
%{ for name in var.names }
  ${name}
```

```
%{ endfor }
EOF
}
```

Po wydaniu polecenia `terraform apply` otrzymasz następujące dane wyjściowe:

```
$ terraform apply
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
for_directive =
  neo

  trinity

  morpheus
```

Zwróć uwagę na wszystkie dodatkowe znaki nowego wiersza. W dyrektywie ciągu tekstowego możesz skorzystać ze *znacznika usuwania białych znaków* (~), który spowoduje usunięcie wszystkich białych znaków (spacji i nowego wiersza) przed dyrektywą (jeśli znacznik znajduje się na początku dyrektywy ciągu tekstowego) lub po niej (jeśli znacznik znajduje się na końcu dyrektywy ciągu tekstowego):

```
output "for_directive_strip_marker" {
  value = <<EOF
%{~ for name in var.names }
  ${name}
%{~ endfor }
EOF
}
```

Uaktualniona wersja kodu powoduje wygenerowanie następujących danych wyjściowych:

```
for_directive_strip_marker =
  neo
  trinity
  morpheus
```

Wyrażenie warunkowe

Podobnie jak Terraform oferuje wiele różnych sposobów na przeprowadzanie pętli, tak samo istnieją różne sposoby na stosowanie wyrażen warunkowych, a każdy z nich jest przeznaczony do użycia w odmiennych sytuacjach:

Parametr count

Wykorzystywany w przypadku zasobów warunkowych.

Wyrażenia for_each i for

Wykorzystywane w przypadku zasobów warunkowych i bloków osadzonych w zasobie.

Dyrektywa ciągu tekstowego i f

Wykorzystywana w przypadku wyrażen warunkowych w ciągu tekstowym.

Przeanalizujemy je po kolei.

Wyrażenie warunkowe z użyciem parametru count

Poznany już wcześniej parametr count pozwala na przygotowanie prostej pętli. Jeśli jesteś sprytny, ten sam mechanizm możesz wykorzystać także do przygotowania prostej konstrukcji warunkowej. Rozpocznę od przedstawienia konstrukcji if w następnym punkcie, natomiast później przejdę do konstrukcji if-else.

Konstrukcja if utworzona za pomocą parametr count

W rozdziale 4. opracowałeś moduł Terraform, który można wykorzystać jako „matrycę” podczas wdrażania klastra serwerów WWW. Ten moduł tworzył automatycznie skalowaną grupę (ASG), mechanizm równoważenia obciążenia (ALB), grupy bezpieczeństwa oraz kilka innych zasobów.

Natomiast ten moduł *nie* tworzył harmonogramu akcji. Skoro klastr ma być skalowany jedynie w środowisku produkcyjnym, zasób `aws_autoscaling_schedule` został zdefiniowany bezpośrednio w konfiguracji produkcyjnej w pliku `live/prod/services/webserver-cluster/main.tf`. Czy istnieje sposób na zdefiniowanie zasobów `aws_autoscaling_schedule` w module `webserver-cluster` i ich warunkowego tworzenia dla wybranych użytkowników modułu oraz nietworzenia dla pozostałych użytkowników?

Przekonajmy się. Pierwszym krokiem jest dodanie do pliku `live/prod/services/webserver-cluster/main.tf` zmiennej danych wejściowych typu boolowskiego, która będzie używana do określenia, czy moduł powinien włączać automatyczne skalowanie.

```
variable "enable_autoscaling" {  
  description = "Wartość true oznacza włączenie automatycznego skalowania"  
  type        = bool  
}
```

Jeżeli korzystasz z języka programowania ogólnego przeznaczenia, tej zmiennej danych wejściowych możesz użyć w konstrukcji if.

```
# To jest tylko pseudokod i nie będzie działał w Terraform.  
if var.enable_autoscaling {  
  resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {  
    scheduled_action_name = "${var.cluster_name}-scale-out-during-business-hours"  
    min_size              = 2  
    max_size              = 10  
    desired_capacity       = 10  
    recurrence             = "0 9 * * *"  
    autoscaling_group_name = aws_autoscaling_group.example.name  
  }  
  
  resource "aws_autoscaling_schedule" "scale_in_at_night" {  
    scheduled_action_name = "${var.cluster_name}-scale-in-at-night"  
    min_size              = 2  
    max_size              = 10  
    desired_capacity       = 2  
    recurrence             = "0 17 * * *"  
    autoscaling_group_name = aws_autoscaling_group.example.name  
  }  
}
```

Ponieważ Terraform nie obsługuje konstrukcji `if`, ten kod nie działa. Jednak to samo zadanie można wykonać za pomocą parametru `count` i jego dwóch właściwości:

- Jeżeli w zasobie wartość parametru `count` wynosi 1, to masz jedną kopię danego zasobu. Natomiast wartość 0 parametru `count` oznacza, że zasób w ogóle nie zostanie utworzony.
- Terraform obsługuje wyrażenia warunkowe w formacie `<WARUNEK> ? <WARTOŚĆ_PRAWDY> : <WARTOŚĆ_FAŁSZU>`. To jest składnia trójargumentowa, którą możesz znać z innych języków programowania. Jej działanie polega na sprawdzeniu wartości boolowskiej `WARUNKU`; jeśli wynikiem jest `true`, będzie zwrócona `WARTOŚĆ_PRAWDY`, natomiast w przypadku wyniku `false` będzie zwrócona `WARTOŚĆ_FAŁSZU`.

Połączenie ze sobą tych dwóch właściwości pozwala na uaktualnienie modułu `webserver-cluster` w przedstawiony tutaj sposób:

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
  count = var.enable_autoscaling ? 1 : 0

  scheduled_action_name = "${var.cluster_name}-scale-out-during-business-hours"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 10
  recurrence             = "0 9 * * *"
  autoscaling_group_name = aws_autoscaling_group.example.name
}

resource "aws_autoscaling_schedule" "scale_in_at_night" {
  count = var.enable_autoscaling ? 1 : 0

  scheduled_action_name = "${var.cluster_name}-scale-in-at-night"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 2
  recurrence             = "0 17 * * *"
  autoscaling_group_name = aws_autoscaling_group.example.name
}
```

Jeżeli wartością `var.enable_autoscaling` jest `true`, parametr `count` dla każdego zasobu `aws_autoscaling_schedule` będzie miał wartość 1, więc nastąpi utworzenie po jednej kopii każdego z tych zasobów. Jeśli natomiast wartością `var.enable_autoscaling` jest `false`, parametr `count` dla każdego zasobu `aws_autoscaling_schedule` będzie miał wartość 0, więc zasób w ogóle nie będzie utworzony. To jest dokładnie taka logika warunkowa, jakiej potrzebujemy w omawianym module.

Sposób wykorzystania omawianego modułu można zmodyfikować w środowisku roboczym (`live/stage/services/webserver-cluster/main.tf`) i wyłączyć automatyczne skalowanie przez przypisanie wartości `false` zasobowi `enable_autoscaling`.

```
module "webserver_cluster" {
  source = "../../../../../modules/services/webserver-cluster"

  cluster_name      = "webservers-stage"
  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"

  instance_type      = "t2.micro"
```

```

min_size      = 2
max_size      = 2
enable_autoscaling = false
}

```

Podobnie można ten moduł uaktualnić w środowisku produkcyjnym (*live/prod/services/webserver-cluster/main.tf*) i włączyć automatyczne skalowanie przez przypisanie wartości `true` zasobowi `enable_autoscaling` (upewnij się o usunięciu zasobów `aws_autoscaling_schedule`, które pozostały w środowisku produkcyjnym po wykonywaniu przykładów przedstawionych w rozdziale 4.).

```

module "webserver_cluster" {
  source = "../../../../../modules/services/webserver-cluster"

  cluster_name      = "webservers-prod"
  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"

  instance_type      = "m4.large"
  min_size           = 2
  max_size           = 10
  enable_autoscaling = true

  custom_tags = {
    Owner      = "team-foo"
    DeployedBy = "terraform"
  }
}

```

Takie rozwiązanie sprawdza się doskonale, gdy użytkownik jawnie przekazuje wartość boolowską do modułu. Co można zrobić w sytuacji, gdy wartość boolowska jest wynikiem znacznie bardziej skomplikowanej operacji porównania, np. ciągu tekstowego? Przeanalizujemy teraz taki dużo bardziej złożony przykład.

Przyjmuję założenie, że jako część modułu `webserver-cluster` chcesz utworzyć zbiór *powiadomień CloudWatch*. Takie powiadomienie możesz skonfigurować do przekazywania go za pomocą wielu różnych mechanizmów (np. poczty e-mail, SMS-a itd.), gdy określony wskaźnik osiągnie zdefiniowaną wartość graniczną. W omawianym przykładzie zasób `aws_cloudwatch_metric_alarm` zostanie wykorzystany w *modules/services/webserver-cluster/main.tf* w celu utworzenia powiadomienia, gdy średni poziom użycia procesora w klastrze będzie wynosił ponad 90% w ciągu 5 minut.

```

resource "aws_cloudwatch_metric_alarm" "high_cpu_utilization" {
  alarm_name = "${var.cluster_name}-high-cpu-utilization"
  namespace = "AWS/EC2"
  metric_name = "CPUUtilization"

  dimensions = {
    AutoScalingGroupName = aws_autoscaling_group.example.name
  }

  comparison_operator = "GreaterThanThreshold"
  evaluation_periods   = 1
  period              = 300
  statistic            = "Average"
  threshold            = 90
  unit                 = "Percent"
}

```


To rozwiązanie sprawdza się świetnie w przypadku powiadomienia o poziomie użycia procesora. Co powinieneś zrobić w sytuacji, gdy chcesz dodać kolejne powiadomienie, ale tym razem o małej liczbie pozostałych jednostek czasu pracy procesora¹? Spójrz na kod powiadomienia CloudWatch, które zostanie wyemitowane, gdy wykorzystasz prawie całą dostępną ilość czasu pracy procesora:

```
resource "aws_cloudwatch_metric_alarm" "low_cpu_credit_balance" {
  alarm_name = "${var.cluster_name}-low-cpu-credit-balance"
  namespace  = "AWS/EC2"
  metric_name = "CPUCreditBalance"

  dimensions = {
    AutoScalingGroupName = aws_autoscaling_group.example.name
  }

  comparison_operator = "LessThanThreshold"
  evaluation_periods   = 1
  period               = 300
  statistic             = "Minimum"
  threshold            = 10
  unit                 = "Count"
}
```

Problem polega na tym, że dostępne jednostki czasu pracy procesora mają zastosowanie tylko dla egzemplarzy tXXX (np. t2.micro, t2.medium itd.). W przypadku większych egzemplarzy (np. m4.large) takie jednostki nie są stosowane, więc nie jest przekazywana wartość CPUCreditBalance. Jeżeli utworzysz dla większych egzemplarzy przedstawione tutaj powiadomienie, ono zawsze będzie w stanie INSUFFICIENT_DATA. Czy istnieje jakikolwiek sposób na zdefiniowanie powiadomienia tylko wtedy, gdy nazwa wartości var.instance_type rozpoczyna się od litery t?

Wprawdzie można dodać nową zmienną danych wejściowych typu boolowskiego o nazwie var.is_t2_instance, ale to oznacza nadmiarowość względem var.instance_type i prawdopodobnie będziesz zapominać o jej uaktualnianiu. Znacznie lepszym rozwiązaniem jest wykorzystanie warunku.

```
resource "aws_cloudwatch_metric_alarm" "low_cpu_credit_balance" {
  count = format("%.1s", var.instance_type) == "t" ? 1 : 0

  alarm_name = "${var.cluster_name}-low-cpu-credit-balance"
  namespace  = "AWS/EC2"
  metric_name = "CPUCreditBalance"

  dimensions = {
    AutoScalingGroupName = aws_autoscaling_group.example.name
  }

  comparison_operator = "LessThanThreshold"
  evaluation_periods   = 1
  period               = 300
  statistic             = "Minimum"
  threshold            = 10
  unit                 = "Count"
}
```

¹ Więcej informacji na temat wykorzystania dostępnych jednostek czasu pracy procesora znajdziesz na stronie <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html>.

Kod powiadomienia pozostał praktycznie bez zmian, a jedyną różnicą jest dość skomplikowana postać parametru `count`:

```
count = format("%.1s", var.instance_type) == "t" ? 1 : 0
```

Ten kod wykorzystuje funkcję `format()` do pobrania pierwszego znaku z wartości zmiennej `var.instance_type`. Jeżeli tym znakiem jest `t` (np. wartością jest `t2.micro`), parametr `count` ma przypisywaną wartość 1. W przeciwnym razie wartością parametru `count` jest 0. W ten sposób powiadomienie zostanie utworzone tylko dla typów egzemplarzy udostępniających wartość `CPUCreditsBalance`.

Konstrukcja `if-else` za pomocą parametru `count`

Skoro wiesz, jak można przygotować konstrukcję `if`, być może zastanawiasz się, jak można zdefiniować konstrukcję `if-else`.

We wcześniejszej części rozdziału utworzyłeś kilku użytkowników IAM, którzy mają do EC2 dostęp w trybie tylko do odczytu. Przyjmuję założenie, że jednemu z tych użytkowników, `neo`, chcesz zapewnić również dostęp do CloudWatch. Jednocześnie chcesz, aby osoba stosująca konfigurację Terraform mogła zdecydować, czy użytkownik `neo` będzie miał dostęp w trybie tylko do odczytu, czy też w trybie odczytu i zapisu. To jest nieco sztuczny przykład, ale pozwala na łatwe przedstawienie prostej konstrukcji `if-else`, w której ważne jest to, która z gałęzi zostanie wykonana, a nie jaki kod Terraform został w niej zdefiniowany.

Spójrz na przykład polityki IAM pozwalającej na uzyskanie dostępu do powiadomienia CloudWatch w trybie tylko do odczytu:

```
resource "aws_iam_policy" "cloudwatch_read_only" {
  name = "cloudwatch-read-only"
  policy = data.aws_iam_policy_document.cloudwatch_read_only.json
}

data "aws_iam_policy_document" "cloudwatch_read_only" {
  statement {
    effect = "Allow"
    actions = [
      "cloudwatch:Describe*",
      "cloudwatch:Get*",
      "cloudwatch:List*"
    ]
    resources = ["*"]
  }
}
```

Natomiast w kolejnym fragmencie kodu przedstawiłem przykład polityki IAM pozwalającej na uzyskanie pełnego dostępu do powiadomienia CloudWatch (w trybie odczytu i zapisu):

```
resource "aws_iam_policy" "cloudwatch_full_access" {
  name = "cloudwatch-full-access"
  policy = data.aws_iam_policy_document.cloudwatch_full_access.json
}

data "aws_iam_policy_document" "cloudwatch_full_access" {
  statement {
    effect = "Allow"
```

```

    actions    = ["cloudwatch:*"]
    resources  = ["*"]
  }
}

```

Celem jest dołączenie jednej z tych polityk IAM do użytkownika neo na podstawie wartości nowej zmiennej danych wejściowych o nazwie `give_neo_cloudwatch_full_access`:

```

variable "give_neo_cloudwatch_full_access" {
  description = "Wartość true oznacza pełny dostęp użytkownika neo do CloudWatch"
  type        = bool
}

```

Jeżeli użyjesz języka programowania ogólnego przeznaczenia, to możesz tworzyć konstrukcje `if-else` w następującej postaci:

```

# To jest tylko pseudokod i nie będzie działał w Terraform.
if var.give_neo_cloudwatch_full_access {
  resource "aws_iam_user_policy_attachment" "neo_cloudwatch_full_access" {
    user      = aws_iam_user.example[0].name
    policy_arn = aws_iam_policy.cloudwatch_full_access.arn
  }
} else {
  resource "aws_iam_user_policy_attachment" "neo_cloudwatch_read_only" {
    user      = aws_iam_user.example[0].name
    policy_arn = aws_iam_policy.cloudwatch_read_only.arn
  }
}

```

Aby ten sam efekt osiągnąć w Terraform, należy użyć parametru `count` i wyrażenia warunkowego dla każdego z zasobów.

```

resource "aws_iam_user_policy_attachment" "neo_cloudwatch_full_access" {
  count = var.give_neo_cloudwatch_full_access ? 1 : 0

  user      = aws_iam_user.example[0].name
  policy_arn = aws_iam_policy.cloudwatch_full_access.arn
}

resource "aws_iam_user_policy_attachment" "neo_cloudwatch_read_only" {
  count = var.give_neo_cloudwatch_full_access ? 0 : 1

  user      = aws_iam_user.example[0].name
  policy_arn = aws_iam_policy.cloudwatch_read_only.arn
}

```

Ten kod zawiera dwa zasoby `aws_iam_user_policy_attachment`. Pierwszy powoduje udzielenie pełnych uprawnień dostępu do CloudWatch i ma wyrażenie warunkowe, które przyjmuje wartość 1, gdy wartością `var.give_neo_cloudwatch_full_access` jest `true`, lub 0, gdy wartością `var.give_neo_cloudwatch_full_access` jest `false` (to jest klauzula `if`). Natomiast drugi zasób przydzielający uprawnienia tylko do odczytu CloudWatch ma wyrażenie warunkowe działające w dokładnie odwrotny sposób: przyjmuje wartość 0, gdy wartością `var.give_neo_cloudwatch_full_access` jest `true`, lub 1, gdy wartością `var.give_neo_cloudwatch_full_access` jest `false` (to jest klauzula `else`).

Takie rozwiązanie sprawdza się dobrze, jeśli kod Terraform nie musi mieć żadnych informacji o tym, która z klauzul faktycznie została wykonana. Co możesz zrobić w sytuacji, gdy potrzebujesz dostępu do pewnych atrybutów danych wyjściowych zasobu pochodzących z klauzuli `if` lub `else`? Jakie masz możliwości, gdy chcesz np. zaoferować dwa odmienne skrypty danych użytkownika w module `webserver-cluster` i pozwolić użytkownikowi na wybór wykonywanego? Aktualnie moduł `webserver-cluster` pobiera skrypt `user-data.sh` za pomocą źródła danych `template_file`.

```
data "template_file" "user_data" {
  template = file("${path.module}/user-data.sh")

  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  }
}
```

Obecnie zawartość skryptu `user-data.sh` przedstawia się następująco:

```
#!/bin/bash

cat > index.html <<EOF
<h1>Witaj, świecie</h1>
<p>Adres bazy danych: ${db_address}</p>
<p>Numer portu bazy danych: ${db_port}</p>
EOF

nohup busybox httpd -f -p ${server_port} &
```

Wyobraź sobie teraz, że chcesz pozwolić klastrom serwera WWW na użycie krótszego, alternatywnego skryptu o nazwie `user-data-new.sh`.

```
#!/bin/bash

echo "Witaj, świecie, v2" > index.html
nohup busybox httpd -f -p ${server_port} &
```

Aby użyć tego skryptu, konieczne jest utworzenie nowego źródła danych `template_file`:

```
data "template_file" "user_data_new" {
  template = file("${path.module}/user-data-new.sh")

  vars = {
    server_port = var.server_port
  }
}
```

Pytanie brzmi: jak pozwolić użytkownikowi modułu `webserver-cluster` na wybór jednego z tych skryptów danych użytkownika? Pierwszą myślą może być dodanie do pliku `modules/services/webserver-cluster/variables.tf` nowej zmiennej danych wejściowych typu boolowskiego.

```
variable "enable_new_user_data" {
  description = "Wartość true oznacza użycie nowego skryptu danych użytkownika"
  type        = bool
}
```

Jeżeli korzystasz z języka programowania ogólnego przeznaczenia, to do konfiguracji startowej możesz dodać konstrukcję `if-else` pozwalającą na wybór między dwoma skryptami danych użytkownika oferowanymi przez `template_file`.

```
# To jest tylko pseudokod i nie będzie działał w Terraform.
resource "aws_launch_configuration" "example" {
  image_id      = "ami-0c55b159cbfaffe1f0"
  instance_type = var.instance_type
  security_groups = [aws_security_group.instance.id]

  if var.enable_new_user_data {
    user_data = data.template_file.user_data_new.rendered
  } else {
    user_data = data.template_file.user_data.rendered
  }
}
```

Aby takie rozwiązanie działało w kodzie Terraform, konieczne jest najpierw zastosowanie poznanej wcześniej sztuczki związanej z konstrukcją `if` i tym samym zagwarantowanie, że w rzeczywistości będzie utworzone tylko jedno źródło danych `template_file`.

```
data "template_file" "user_data" {
  count = var.enable_new_user_data ? 0 : 1

  template = file("${path.module}/user-data.sh")

  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  }
}

data "template_file" "user_data_new" {
  count = var.enable_new_user_data ? 1 : 0

  template = file("${path.module}/user-data-new.sh")

  vars = {
    server_port = var.server_port
  }
}
```

Jeżeli wartością `var.enable_new_user_data` jest `true`, nastąpi utworzenie zasobu `data.template_file.user_data_new`, natomiast w przypadku wartości `false` ten zasób nie będzie utworzony. Musisz jedynie wiedzieć, jak przypisać wartość parametru `user_data` zasobu `aws_launch_configuration` do faktycznie istniejącego `template_file`. W tym celu możesz wykorzystać przedstawione tutaj wyrażenie warunkowe.

```
resource "aws_launch_configuration" "example" {
  image_id      = "ami-0c55b159cbfaffe1f0"
  instance_type = var.instance_type
  security_groups = [aws_security_group.instance.id]
```

```

user_data = (
    length(data.template_file.user_data[*]) > 0
    ? data.template_file.user_data[0].rendered
    : data.template_file.user_data_new[0].rendered
)

# Wymagane tylko w przypadku konfiguracji startowej z automatycznie skalowaną grupą.
# https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
lifecycle {
    create_before_destroy = true
}
}

```

Spróbujmy podzielić większą wartość dla parametru `user_data`. Przede wszystkim zwróć uwagę na sprawdzany warunek boolowski.

```
length(data.template_file.user_data[*]) > 0
```

Zauważ, że oba źródła danych `template_file` są tablicami i używają parametru `count`, więc konieczne jest wykorzystanie składni tablicy. Skoro jedna z tych tablic będzie miała wielkość 1, druga zaś wielkość 0, to nie można bezpośrednio uzyskać dostępu do określonego indeksu (np. `data.template_file.user_data[0]`), ponieważ tablica może być pusta. Rozwiązaniem jest zastosowanie wyrażenia `splat`, które zawsze zwraca tablicę (choć istnieje prawdopodobieństwo, że będzie ona pusta) i sprawdza jej wielkość.

Wykorzystując wielkość tablicy, możesz wybrać jedno z wymienionych tutaj wyrażeń:

```

? data.template_file.user_data[0].rendered
: data.template_file.user_data_new[0].rendered

```

W przypadku wyniku warunkowego Terraform przeprowadza obliczenie z opóźnieniem, więc wartość `true` będzie wynikiem, gdy wyrażenie przyjmie wartość `true`, a wynik `false` otrzymamy po przyjęciu wartości `false` przez wyrażenie. To oznacza możliwość bezpiecznego sprawdzenia indeksu 0 w `user_data` i `user_data_new`, ponieważ wiadomo, że w rzeczywistości zostanie wykorzystana tylko niepusta tablica.

Nowy skrypt danych wyjściowych można wykorzystać w środowisku roboczym przez przypisanie wartości `true` parametrowi `enable_new_user_data` w `live/stage/services/webserver-cluster/main.tf`.

```

module "webserver_cluster" {
    source = "../../../../../modules/services/webserver-cluster"

    cluster_name           = "webservers-stage"
    db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
    db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"

    instance_type          = "t2.micro"
    min_size               = 2
    max_size               = 2
    enable_autoscaling     = false
    enable_new_user_data   = true
}

```

W środowisku produkcyjnym można pozostać przy starszej wersji skryptu przez przypisanie wartości `false` parametrowi `enable_new_user_data` w pliku `live/prod/services/webserver-cluster/main.tf`.

```
module "webserver_cluster" {
  source = "../../../../../modules/services/webserver-cluster"

  cluster_name      = "webservers-prod"
  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"
  instance_type      = "m4.large"
  min_size           = 2
  max_size           = 10
  enable_autoscaling  = true
  enable_new_user_data = false

  custom_tags = {
    Owner      = "team-foo"
    DeployedBy = "terraform"
  }
}
```

Wprawdzie używanie parametru `count` i funkcji wbudowanych w celu symulowania konstrukcji `if-else` oznacza konieczność kombinowania, ale przygotowane rozwiązanie działa dość dobrze. Jak możesz zobaczyć w przykładzie omówionego kodu, pozwala na ukrycie skomplikowanych szczegółów przed użytkownikiem, który dzięki temu ma możliwość pracy z przejrzystym i prostym API.

Definiowanie warunku za pomocą `for_each` i wyrażeń

Skoro dowiedziałeś się, jak można definiować logikę warunkową za pomocą zasobów i parametru `count`, prawdopodobnie domyślasz się, że podobną strategię można wykorzystać do przygotowania logiki warunkowej wraz z wyrażeniem `for_each`. Jeżeli przekażesz wyrażeniu `for_each` pustą kolekcję, wygenerowane zostanie zero zasobów lub zero bloków osadzonych. Natomiast skutkiem przekazania niepustej kolekcji jest utworzenie jednego lub więcej zasobów albo bloków osadzonych. Jedyne pytanie dotyczy sposobu warunkowego ustalenia, czy kolekcja powinna być pusta, czy nie.

Odpowiedzią jest połączenie wyrażeń `for_each` i `for`. Dla przykładu przypomnij sobie sposób, w jaki moduł `webserver-cluster` w pliku `modules/services/webserver-cluster/main.tf` definiuje tagi:

```
dynamic "tag" {
  for_each = var.custom_tags

  content {
    key          = tag.key
    value        = tag.value
    propagate_at_launch = true
  }
}
```

Jeżeli kolekcja `var.custom_tags` jest pusta, to wyrażenie `for_each` nie musi przeprowadzać iteracji, więc nie zostaną zdefiniowane żadne tagi. Innymi słowy, masz już pewną logikę warunkową. Jednak można pójść o krok dalej i połączyć wyrażenia `for_each` i `for` w pokazany tutaj sposób:

```
dynamic "tag" {
  for_each = {
    for key, value in var.custom_tags:
      key => upper(value)
      if key != "Name"
    }

  content {
    key          = tag.key
    value        = tag.value
    propagate_at_launch = true
  }
}
```

Zagnieżdżone wyrażenie `for` przeprowadza iterację przez `var.custom_tags` i konwertuje każdą wartość na zapisaną wielkimi literami (prawdopodobnie w celu zachowania spójności) oraz wykorzystuje warunek w wyrażeniu `for` do odfiltrowania wszelkich elementów `key` o wartości `Name`, ponieważ moduł już zdefiniował własny tag `Name`. Dzięki filtrowaniu wartości wyrażenia `for` można zaimplementować dowolną logikę warunkową.

Zwróć uwagę, że choć podczas tworzenia wielu kopii zasobu niemal zawsze powinieneś preferować `for_each` zamiast `count`, to w przypadku logiki warunkowej przypisanie parametrowi `count` wartości 0 lub 1 jest prostsze niż przypisanie `for_each` pustej lub niepustej kolekcji. Dlatego parametru `count` będziesz używać do warunkowego tworzenia zasobów, a wyrażenia `for_each` we wszystkich pozostałych typach pętli i wyrażen warunkowych.

Wyrażenia warunkowe wraz z dyrektywą `if` ciągu tekstowego

Wcześniej w rozdziale dyrektywę `for` ciągu tekstowego wykorzystałeś do obsługi pętli w ciągu tekstowym. Spójrz teraz na drugi typ dyrektywy ciągu tekstowego, która ma następującą postać:

```
%{ if <WARUNEK> }<WARTOŚĆ_PRAWDY>{%{ endif }
```

gdzie `WARUNEK` to dowolne wyrażenie przyjmujące wartość boolowską, natomiast `WARTOŚĆ_PRAWDY` to wyrażenie do wygenerowania, gdy `WARUNEK` zostanie spełniony (jest prawdziwy). Opcjonalnie można wykorzystać klauzulę `else`:

```
%{ if <WARUNEK> }<WARTOŚĆ_PRAWDY>{%{ else }<WARTOŚĆ_FAŁSZU>{%{ endif }
```

gdzie `WARTOŚĆ_FAŁSZU` to wyrażenie do wygenerowania, gdy `WARUNEK` nie zostanie spełniony (jest fałszywy). Spójrz na przedstawiony tutaj przykład:

```
variable "name" {
  description = "Imię do wyświetlenia"
  type        = string
}

output "if_else_directive" {
  value = "Witaj, %{ if var.name != "" }${var.name}%{ else }(nieznajomy)%{ endif }"
}
```


Jeżeli teraz wydasz polecenie `terraform apply` i przypiszesz zmiennej `name` wartość `świecie`, otrzymasz następujące dane wyjściowe:

```
$ terraform apply -var name="świecie"
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
if_else_directive = Witaj, świecie
```

Natomiast po wydaniu polecenia `terraform apply` i przypisaniu `name` pustego ciągu tekstowego otrzymasz nieco inne dane wyjściowe:

```
$ terraform apply -var name=""
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
if_else_directive = Hello, (nieznajomy)
```

Wdrożenie bez przestoju

Gdy moduł ma przejrzyste i proste API przeznaczone do wdrożenia klastra serwera WWW, ważne pytanie brzmi, jak przeprowadzać uaktualnienie tego klastra. Innymi słowy: jak po wprowadzeniu zmian w kodzie można wdrożyć nowy obraz AMI w klastrze? Kolejną kwestią jest to, jak przeprowadzić wdrożenie bez przestoju, czyli bez odcinania użytkowników od usługi.

Pierwszym krokiem jest udostępnienie AMI jako zmiennej danych wejściowych w pliku `modules/services/webserver-cluster/main.tf`. W rzeczywistych przykładach to wszystko, czego będziesz potrzebować, ponieważ kod serwera WWW może być zdefiniowany w AMI. Jednak w uproszczonych przykładach przedstawionych w książce cały kod serwera WWW został zdefiniowany w skrypcie danych użytkownika, a AMI to tylko niezmodyfikowany obraz systemu Ubuntu. Zmiana wersji Ubuntu nie jest zbyt efektywna, więc poza dodaniem nowej zmiennej danych wyjściowych konieczne jest dodanie kolejnej zmiennej danych wejściowych przeznaczonej do kontrolowania skryptu danych użytkownika zwracanego przez nasz jednowierszowy serwer HTTP.

```
variable "ami" {
  description = "Obraz AMI do uruchomienia w klastrze"
  default    = "ami-0c55b159cbfafelf0"
  type       = string
}

variable "server_text" {
  description = "Ciąg tekstowy zwracany przez serwer"
  default     = "Witaj, świecie"
  type        = string
}
```

Podczas eksperymentowania z konstrukcją `if-else` we wcześniejszej części rozdziału utworzyłeś dwa skrypty danych wejściowych użytkownika. Skonsolidujemy je w jeden, aby zachować prostotę przykładu. Po pierwsze, w pliku `modules/services/webserver-cluster/variables.tf` należy usunąć zmienną

danych wejściowych `enable_new_user_data`. Po drugie, w `modules/services/webserver-cluster/main.tf` należy usunąć zasób `template_file` o nazwie `user_data_new`. Po trzecie, w tym samym pliku należy uaktualnić pozostały zasób `template_file` o nazwie `user_data`, aby dłużej nie korzystał ze zmiennej danych wejściowych `enable_new_user_data`, i dodać zmienną danych wejściowych `server_text` do bloku `vars`.

```
data "template_file" "user_data" {
  template = file("${path.module}/user-data.sh")

  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
    server_text = var.server_text
  }
}
```

Kolejnym krokiem jest uaktualnienie skryptu Bash `modules/services/webserver-cluster/user-data.sh`, aby używał zmiennej `server_text` w zwracanym znaczniku `<h1>`:

```
#!/bin/bash

cat > index.html <<EOF
<h1>${server_text}</h1>
<p>Adres bazy danych: ${db_address}</p>
<p>Numer portu bazy danych: ${db_port}</p>
EOF

nohup busybox httpd -f -p ${server_port} &
```

Odszukaj konfigurację startową w `modules/services/webserver-cluster/main.tf`, jej parametrowi `user_data` przypisz pozostały zasób `template_file` (ten o nazwie `user_data`) i przypisz parametrowi `ami` nową zmienną danych wejściowych `ami`.

```
resource "aws_launch_configuration" "example" {
  image_id      = var.ami
  instance_type = var.instance_type
  security_groups = [aws_security_group.instance.id]

  user_data = data.template_file.user_data.rendered

  # Wymagane tylko w przypadku konfiguracji startowej z automatycznie skalowaną grupą.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

Teraz w środowisku roboczym (`live/stage/services/webserver-cluster/main.tf`) można przypisać nowe parametry `ami` i `server_text` oraz usunąć parametr `enable_new_user_data`:

```
module "webserver_cluster" {
  source = "../../../../../modules/services/webserver-cluster"
```

```

ami          = "ami-0c55b159cbfafef0"
server_text = "Komunikat nowego serwera"

cluster_name      = "webservers-stage"
db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"

instance_type    = "t2.micro"
min_size         = 2
max_size         = 2
enable_autoscaling = false
}

```

Ten kod używa tego samego obrazu Ubuntu AMI, ale ma inną wartość parametru `server_text`. Po wykonaniu polecenia `plan` powinien otrzymać następujące dane wyjściowe:

Terraform will perform the following actions:

```

# Zasób module.webserver_cluster.aws_autoscaling_group.ex zostanie uaktualniony w miejscu.
~ resource "aws_autoscaling_group" "example" {
  id              = "webservers-stage-terraform-20190516"
  ~ launch_configuration = "terraform-20190516" -> (known after apply)
  (...)
}

# Zasób module.webserver_cluster.aws_launch_configuration.ex musi być zastąpiony nowym.
+/- resource "aws_launch_configuration" "example" {
  ~ id              = "terraform-20190516" -> (known after apply)
  image_id          = "ami-0c55b159cbfafef0"
  instance_type     = "t2.micro"
  ~ name            = "terraform-20190516" -> (known after apply)
  ~ user_data       = "bd7c0a6" -> "4919a13" # Wymuszone zastąpienie.
  (...)
}

```

Plan: 1 to add, 1 to change, 1 to destroy.

Jak możesz zobaczyć, Terraform chce przeprowadzić dwie zmiany. Pierwsza to zastąpienie starej konfiguracji startowej nową wraz z uaktualnioną zawartością `user_data`. Druga to modyfikacja w miejscu automatycznie skalowanej grupy, aby odwoływała się do nowej konfiguracji startowej. Problem polega na tym, że w przypadku jedynie odwołania się do nowej konfiguracji startowej nie będzie miało ono żadnego efektu aż do chwili uruchomienia przez grupę ASG nowych egzemplarzy EC2. Jak można więc nakazać grupie ASG wdrożenie nowych egzemplarzy?

Jedną z możliwości jest usunięcie grupy ASG (przez wydanie polecenia `terraform destroy`), a następnie jej ponowne utworzenie (poprzez wydanie polecenia `terraform apply`). Jednak w takim podejściu problem polega na tym, że po usunięciu starej grupy ASG użytkownicy doświadczą przestoju aż do chwili uruchomienia nowej grupy ASG. Zamiast tego nas interesuje *wdrożenie bez przestoju*. Należy więc najpierw utworzyć zamiennik grupy ASG, a dopiero później usunąć tę starą. Okazuje się, że ustawienie cyklu życiowego `create_before_destroy`, z którym po raz pierwszy zetknąłeś się w rozdziale 2., działa dokładnie w taki właśnie sposób.

Spójrz, jak można wykorzystać zalety tego ustawienia, aby przeprowadzić wdrożenie bez przestoju²:

1. Skonfiguruj parametr name grupy ASG w taki sposób, aby był bezpośrednio zależny od nazwy konfiguracji startowej. Każda zmiana konfiguracji startowej (po uaktualnieniu obrazu AMI lub danych użytkownika) powoduje, że jej nazwa ulega zmianie i tym samym nazwa grupy ASG również się zmienia, co wymusi na Terraform zastąpienie grupy ASG.
2. Parametrowi create_before_destroy grupy ASG przypisz wartość true, więc za każdym razem, gdy Terraform będzie próbować ją zastąpić nową, najpierw utworzy zamiennik grupy ASG, a dopiero później usunie tę starą.
3. Parametrowi min_elb_capacity grupy ASG przypisz wartość min_size klastra, aby Terraform zaczekał z usunięciem starej grupy ASG przynajmniej do chwili, gdy podana liczba serwerów nowej grupy ASG zostanie uznana za w pełni sprawną w mechanizmie równoważenia obciążenia.

Oto uaktualniona zawartość zasobu aws_autoscaling_group w pliku `modules/services/webserver-cluster/main.tf`:

```
resource "aws_autoscaling_group" "example" {  
  # Wyrażna zależność od nazwy konfiguracji startowej, aby każda jej zmiana  
  # powodowała również zastąpienie nową tej grupy ASG.  
  name = "${var.cluster_name}-${aws_launch_configuration.example.name}"  
  
  launch_configuration = aws_launch_configuration.example.name  
  vpc_zone_identifier   = data.aws_subnet_ids.default.ids  
  target_group_arns     = [aws_lb_target_group.asg.arn]  
  health_check_type     = "ELB"  
  
  min_size = var.min_size  
  max_size = var.max_size  
  
  # Zanim wdrożenie grupy ASG zostanie uznane za zakończone, należy poczekać do  
  # chwili, aż podana liczba egzemplarzy zostanie uznana za w pełni sprawną.  
  min_elb_capacity = var.min_size  
  
  # Podczas zastępowania tej grupy ASG najpierw ma zostać utworzona nowa,  
  # a dopiero później może być usunięta stara.  
  lifecycle {  
    create_before_destroy = true  
  }  
  
  tag {  
    key          = "Name"  
    value        = var.cluster_name  
    propagate_at_launch = true  
  }  
  
  dynamic "tag" {  
    for_each = {  
      for key, value in var.custom_tags:
```

² Podziękowania za opracowanie tej techniki należą się Paulowi Hinze'owi (<https://groups.google.com/forum/#!msg/terraform-tool/7GdhvIOAc80/iNQ93riiLwAJ>).

```

        key => upper(value)
        if key != "Name"
    }

    content {
        key          = tag.key
        value         = tag.value
        propagate_at_launch = true
    }
}
}

```

Po ponownym wykonaniu polecenia terraform plan otrzymasz dane wyjściowe podobne do tutaj przedstawionych:

Terraform will perform the following actions:

```

# Zasób module.webserver_cluster.aws_autoscaling_group.example musi być zastąpiony.
+/- resource "aws_autoscaling_group" "example" {
    ~ id          = "example-2019" -> (known after apply)
    ~ name        = "example-2019" -> (known after apply) # Wymuszone zastąpienie.
    (...)
}

# Zasób module.webserver_cluster.aws_launch_configuration.example musi być zastąpiony.
+/- resource "aws_launch_configuration" "example" {
    ~ id          = "terraform-2019" -> (known after apply)
      image_id     = "ami-0c55b159cbfafa1f0"
      instance_type = "t2.micro"
    ~ name        = "terraform-2019" -> (known after apply)
    ~ user_data    = "bd7c0a" -> "4919a" # Wymuszone zastąpienie.
    (...)
}

(...)

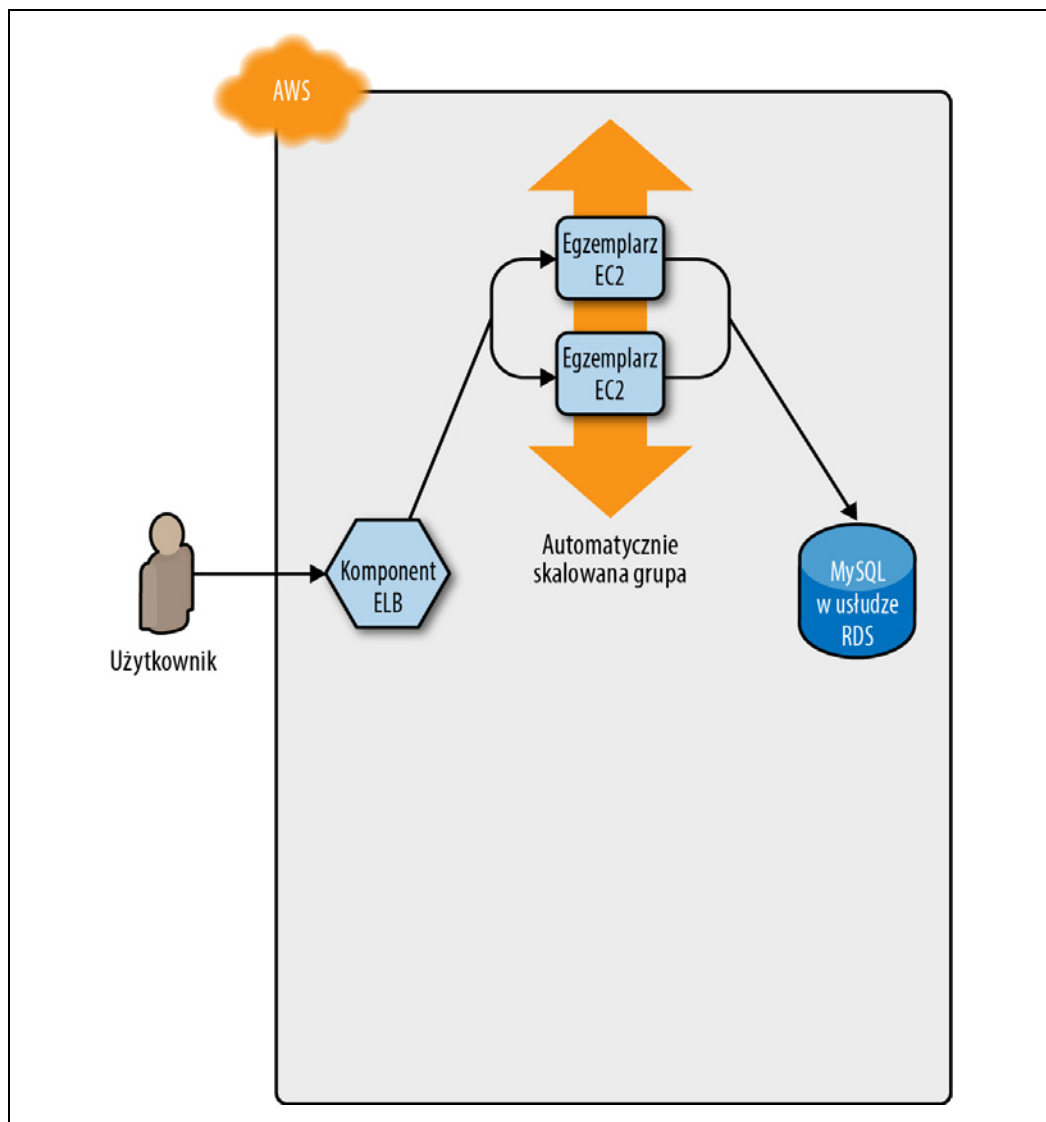
```

Plan: 2 to add, 2 to change, 2 to destroy.

Kluczową kwestią, na którą należy zwrócić uwagę, jest to, że zasób `aws_autoscaling_group` ma obok nazwy parametru umieszczony komunikat `forces replacement` oznaczający zastąpienie go nową grupą ASG działającą wraz z nowym obrazem AMI lub danymi użytkownika. Wyдай polecenie `terraform apply`, aby rozpocząć wdrożenie. W trakcie jego trwania zapoznaj się z przedstawionym tutaj przebiegiem tego procesu.

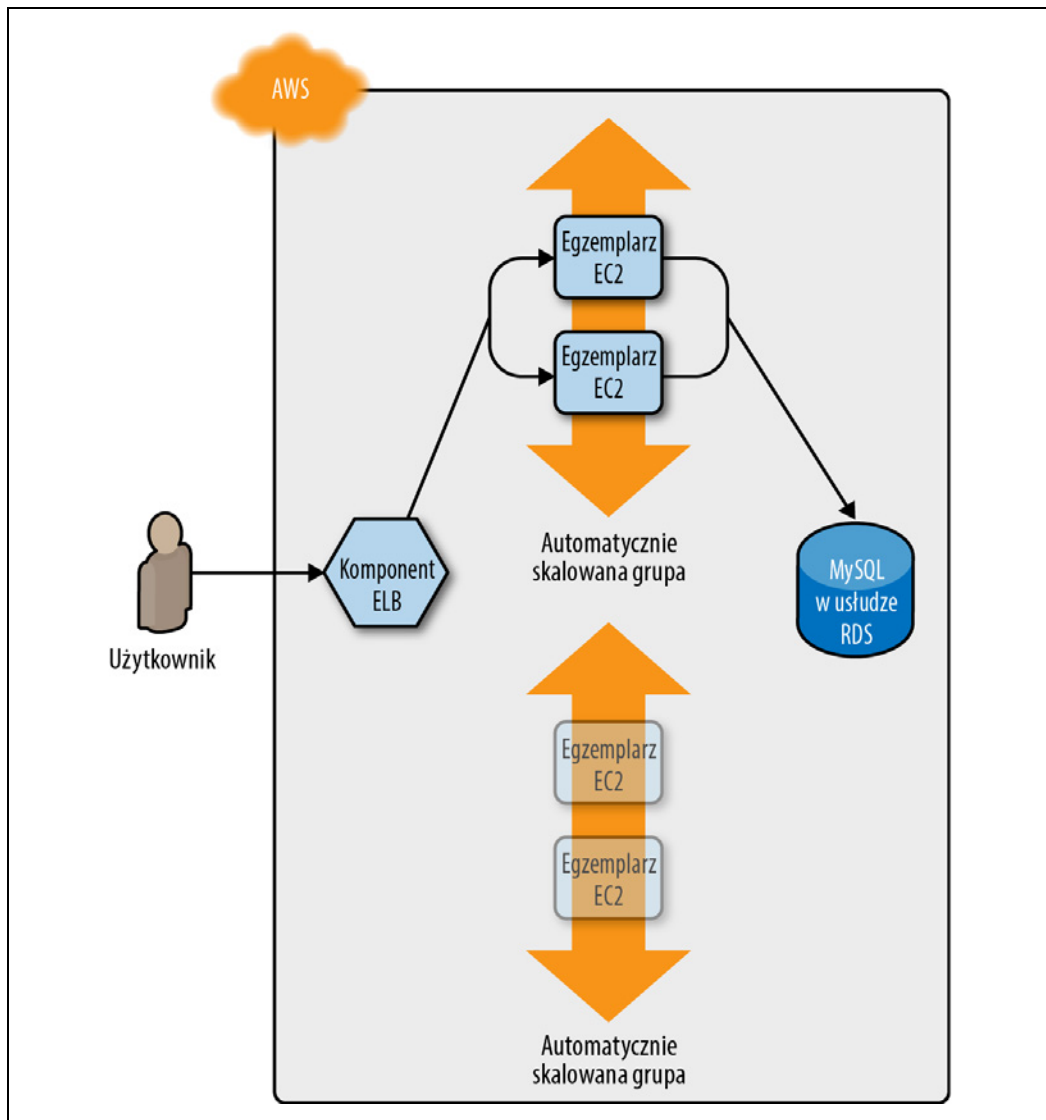
Na początku masz uruchomioną pierwotną grupę ASG wykonującą kod w wersji v1 (zobacz rysunek 5.2).

Wprowadzasz uaktualnienie pewnego aspektu konfiguracji startowej, np. przejście do obrazu AMI zawierającego wersję v2 kodu źródłowego, a następnie wydajesz polecenie `terraform apply`. To wymusza na Terraform rozpoczęcie wdrażania nowej grupy ASG wraz z wersją v2 kodu źródłowego (zobacz rysunek 5.3).



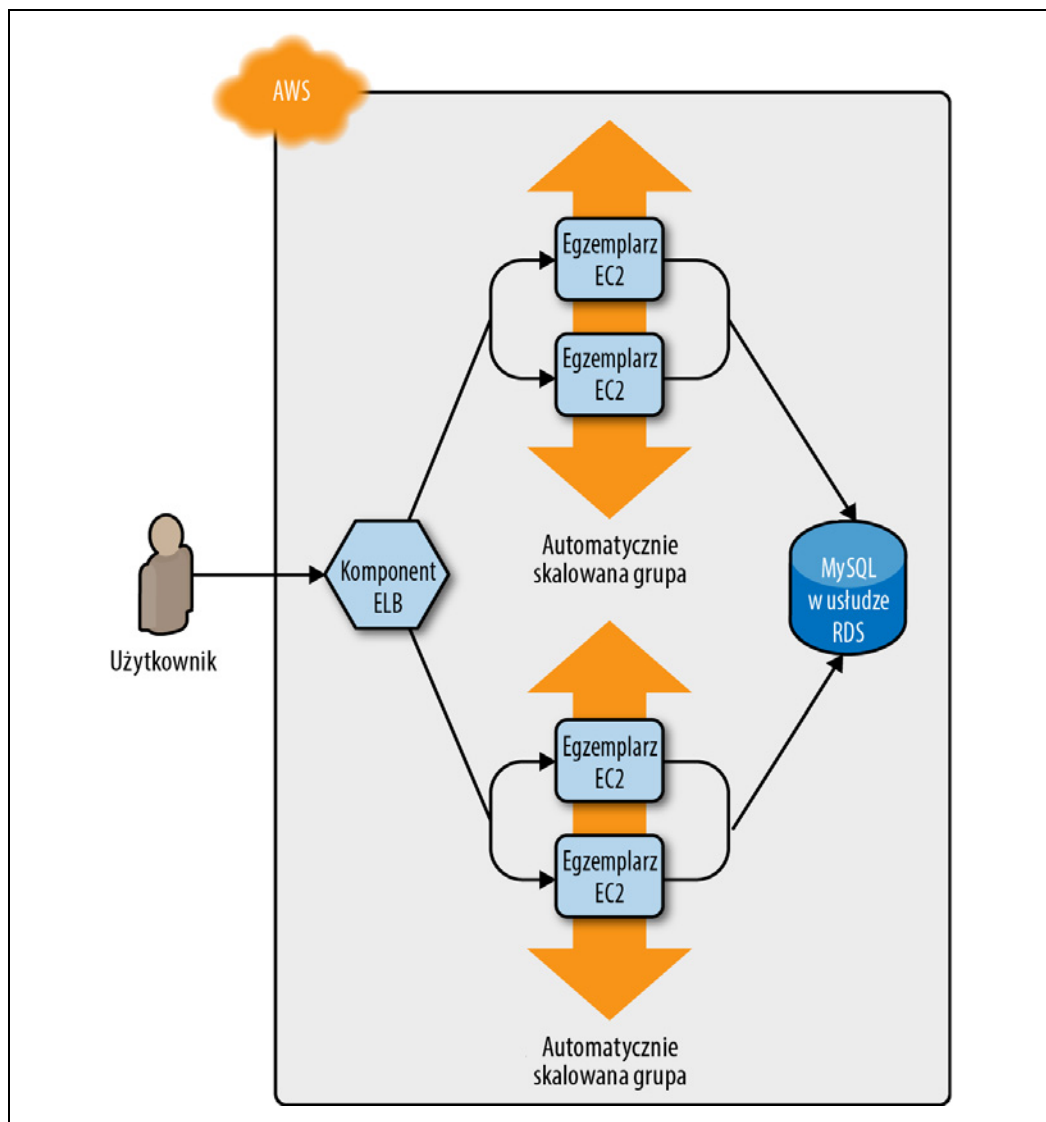
Rysunek 5.2. Początkowo masz pierwotną grupę ASG wraz z uruchomionym kodem w wersji v1

Po 1 – 2 minutach serwery nowej grupy ASG są uruchomione, nawiązały połączenie z bazą danych, zostały zarejestrowane w mechanizmie równoważenia obciążenia oraz rozpoczęły procedurę sprawdzenia swojego stanu. Na tym etapie jednocześnie działają wersje v1 i v2 aplikacji, a wersja używana przez użytkownika zależy od tego, do której został on przekierowany przez mechanizm równoważenia obciążenia (zobacz rysunek 5.4).



Rysunek 5.3. Terraform rozpoczyna wdrażanie nowej grupy ASG wraz z kodem w wersji v2

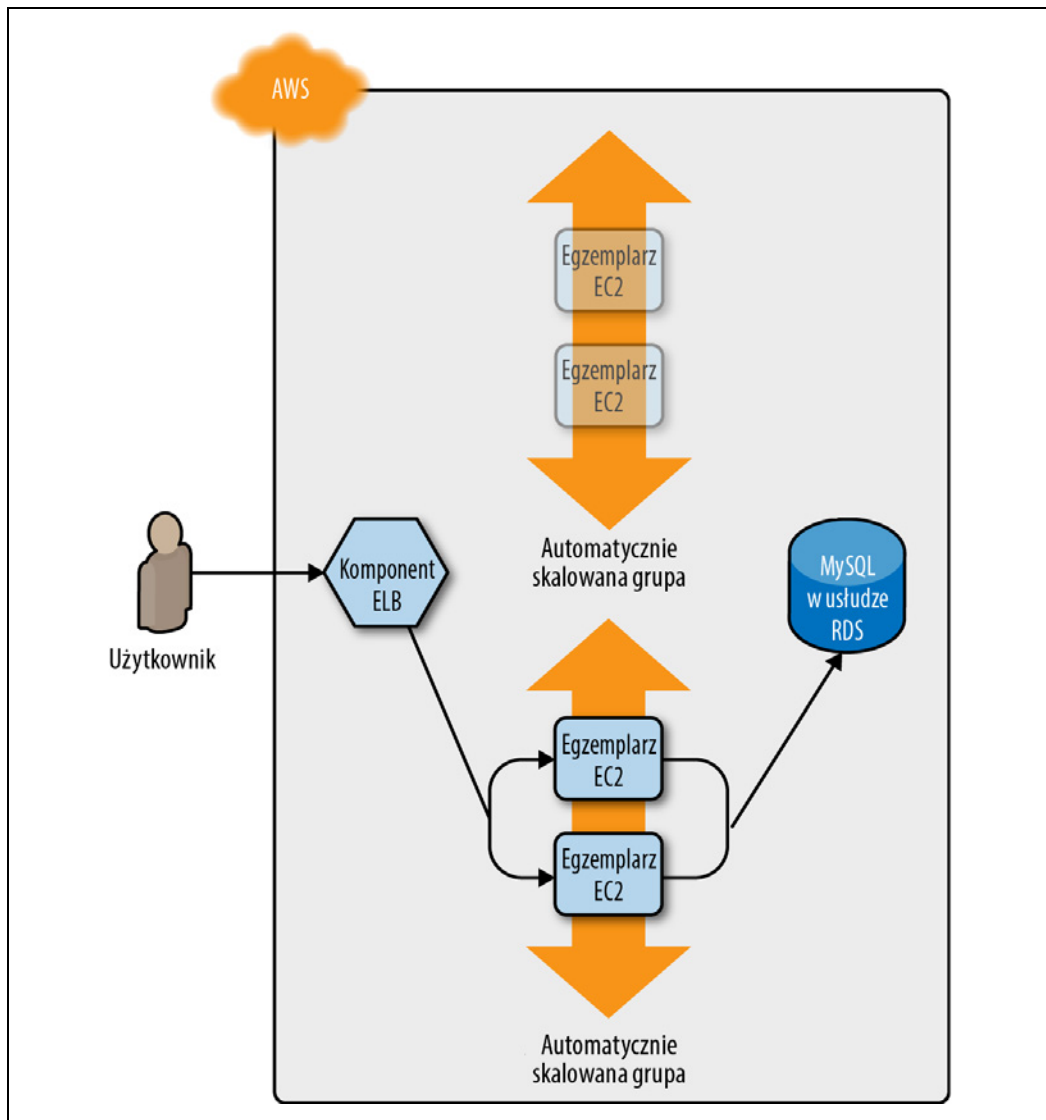
Gdy mechanizm równoważenia obciążenia w nowej grupie ASG klastra kodu w wersji v2 ma przynajmniej `min_elb_capacity` poprawnie działających serwerów, Terraform rozpoczyna usuwanie starej grupy ASG. To oznacza wyrejestrowanie serwerów tej grupy ASG z mechanizmu równoważenia obciążenia, a następnie ich zamknięcie (zobacz rysunek 5.5).



Rysunek 5.4. Serwery w nowej grupie ASG są uruchomione, nawiązały połączenie z bazą danych, zostały zarejestrowane przez mechanizm równoważenia obciążenia i rozpoczęły obsługę ruchu sieciowego

Po 1 – 2 minutach stara grupa ASG została usunięta i tym samym pozostała już tylko aplikacja wraz z kodem w wersji v2, działająca w nowej grupie ASG (zobacz rysunek 5.6).

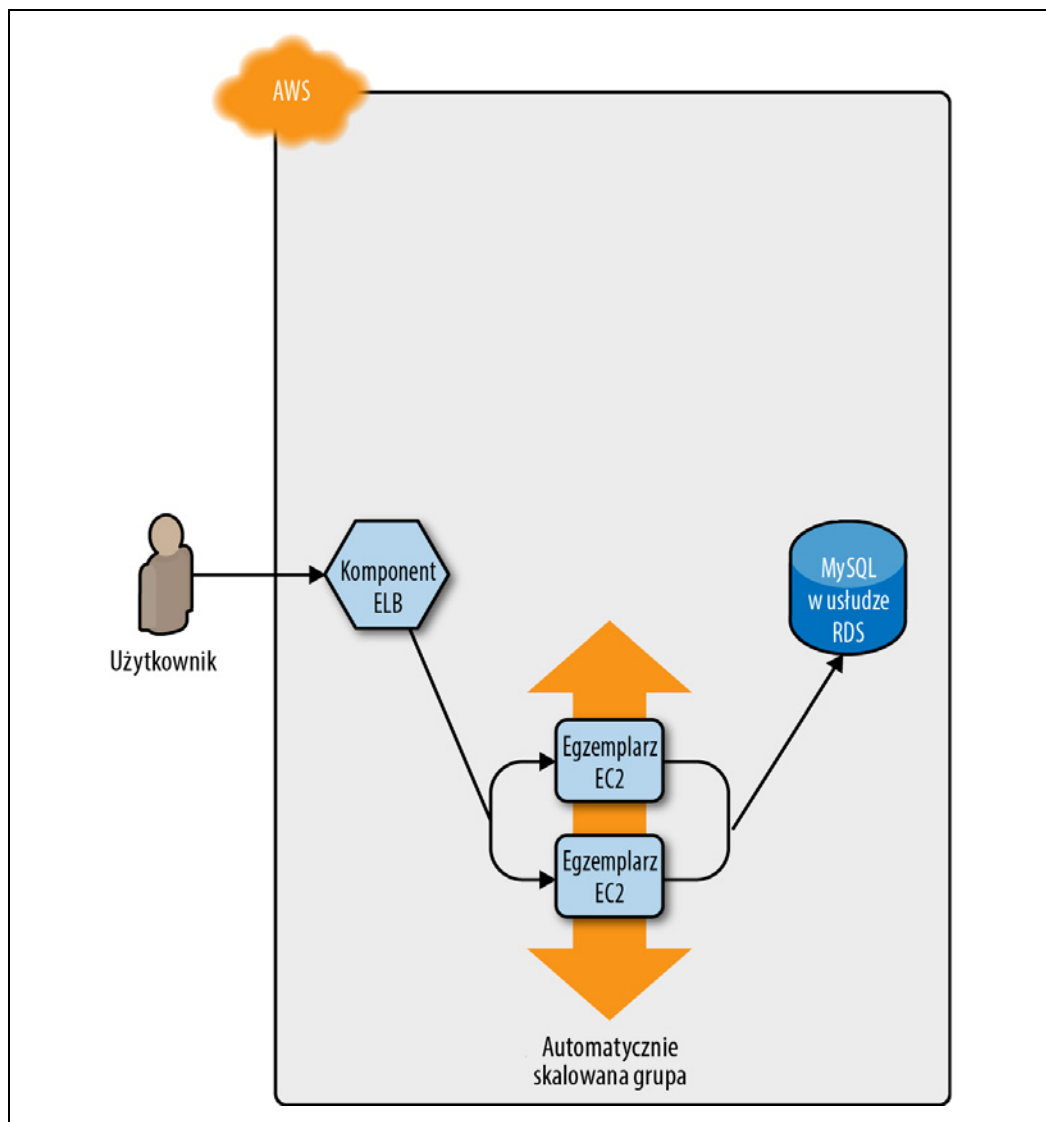
W trakcie tego procesu zawsze istnieją działające serwery, które obsługują żądania przychodzące z mechanizmu równoważenia obciążenia, więc nie ma żadnego przestoju. W przeglądarce WWW przejdź pod adres ALB URL, a otrzymasz dane wyjściowe podobne do pokazanych na rysunku 5.7.



Rysunek 5.5. Serwery w starej grupie ASG zaczynają być wyłączane

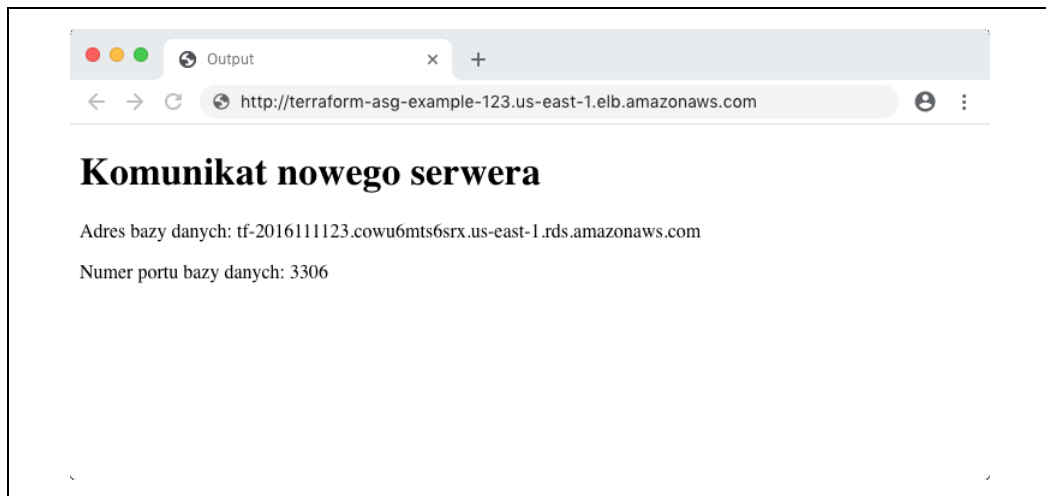
Sukces! Nowy serwer został wdrożony. Jeżeli chcesz poeksperymentować, wprowadź inną zmianę w parametrze `server_text`, np. uaktualnij komunikat do postaci `foo bar` — i ponownie wykonaj polecenie `terraform apply`. Jeżeli używasz systemu Linux, UNIX lub macOS, na oddzielnej karcie narzędzia Terminal możesz wykorzystać jednowierszowe polecenie `curl` w pętli, którego działanie polega na wykonywaniu co sekundę żądania do mechanizmu równoważenia obciążenia, co pozwoli na zobaczenie w akcji żądania bez przestoju.

```
$ while true; do curl http://<adres_url_usługi_elb>; sleep 1; done
```



Rysunek 5.6. W tym momencie pozostała tylko nowa grupa ASG, w której działa kod w wersji v2

Przez mniej więcej pierwszą minutę powinieneś widzieć tę samą odpowiedź w postaci komunikatu Komunikat nowego serwera. Następnie pojawi się komunikat foo bar oznaczający, że nowe egzemplarze zostały zarejestrowane przez mechanizm równoważenia obciążenia i uznane za gotowe do obsługi ruchu sieciowego. Po mniej więcej kolejnej minucie komunikaty Komunikat nowego serwera przestaną się pojawiać i pozostają jedynie foo bar, co oznacza, że stara grupa ASG została wyłączona. Wygenerowane dane wyjściowe będą podobne do tutaj przedstawionych (dla zachowania przejrzystości pokazałem jedynie zawartość znaczników <h1>):



Rysunek 5.7. Nowa wersja kodu została wdrożona

```
Komunikat nowego serwera
Komunikat nowego serwera
Komunikat nowego serwera
Komunikat nowego serwera
Komunikat nowego serwera
Komunikat nowego serwera
foo bar
Komunikat nowego serwera
foo bar
Komunikat nowego serwera
foo bar
Komunikat nowego serwera
foo bar
Komunikat nowego serwera
foo bar
Komunikat nowego serwera
foo bar
foo bar
foo bar
foo bar
foo bar
foo bar
```

Jeżeli w trakcie wdrożenia pojawi się jakikolwiek problem, Terraform automatycznie wycofa operację. Przykładowo, jeśli w kodzie aplikacji w wersji v2 znajduje się błąd uniemożliwiający jej uruchomienie, egzemplarze nowej grupy ASG nie zostaną zarejestrowane przez mechanizm równoważenia obciążenia. Terraform wstrzyma się z rejestracją w mechanizmie równoważenia obciążenia przez czas zdefiniowany przez `wait_for_capacity_timeout` (domyślnie 10 minut) dla `min_elb_capacity` serwerów nowej grupy, a następnie usunie nową grupę i zakończy działanie wraz z błędem (w tym czasie aplikacja w wersji v1 nadal będzie działała bez problemu w starej grupie ASG).

Problemy związane z Terraform

Po zapoznaniu się z tymi wszystkimi podpowiedziami i sztuczkami warto zrobić krok wstecz i poznać kilka problemów, w tym związanych z pętlami, konstrukcjami `if` oraz technikami wdrażania, a także z ogólnymi kwestiami wpływającymi na Terraform jako całość:

- ograniczenia parametru `count` i wyrażenia `for_each`,
- ograniczenia wdrożenia bez przestoju,
- awarie poprawnych planów,
- trudności podczas refaktoryzacji,
- osiągnięcie ostatecznej spójności może wymagać nieco czasu.

Ograniczenia parametru `count` i wyrażenia `for_each`

W przykładach przedstawionych w rozdziale dość często korzystałeś z parametru `count` i wyrażen `for_each` w konstrukcjach `if`. Wprawdzie takie rozwiązanie sprawdza się dobrze, ale jednocześnie ma dwa poważne ograniczenia, o których trzeba wiedzieć:

- W `count` i `for_each` nie można odwoływać się do danych wyjściowych żadnego zasobu.
- Nie można używać `count` i `for_each` w bloku konfiguracyjnym `module`.

Przeanalizujmy dokładniej te ograniczenia.

W `count` i `for_each` nie można odwoływać się do danych wyjściowych żadnego zasobu

Przyjmuję założenie o konieczności wdrożenia wielu egzemplarzy EC2, przy czym z pewnego powodu nie chcesz używać grupy ASG. Kod może przedstawiać się następująco:

```
resource "aws_instance" "example_1" {
  count      = 3
  ami       = "ami-0c55b159cbf1f0"
  instance_type = "t2.micro"
}
```

Skoro parametr `count` ma na stałe zdefiniowaną wartość, ten kod będzie działał bez problemów, po wydaniu polecenia `terraform apply` zaś nastąpi utworzenie trzech egzemplarzy EC2. Co możesz zrobić w sytuacji, gdy będziesz chciał wdrożyć po jednym egzemplarzu EC2 na poszczególnych strefach dostępności (AZ) w bieżącym regionie AWS? Mógłbyś uaktualnić kod w celu pobierania listy stref AZ przy użyciu źródła danych `aws_availability_zones`, a następnie wykorzystać parametr `count` i operację wyszukiwania tablicy do „iteracji” przez te strefy AZ i tworzenia w nich egzemplarza EC2.

```
resource "aws_instance" "example_2" {
  count            = length(data.aws_availability_zones.all.names)
  availability_zone = data.aws_availability_zones.all.names[count.index]
  ami             = "ami-0c55b159cbf1f0"
  instance_type   = "t2.micro"
}

data "aws_availability_zones" "all" {}
```

Ten kod działa prawidłowo, ponieważ parametr `count` może bez problemów odwołać się do źródeł danych. Jednak co się stanie, jeśli liczba egzemplarzy do utworzenia będzie zależała od danych wyjściowych pewnego źródła? Najłatwiejszym sposobem na przeprowadzenie eksperymentów jest wykorzystanie zasobu `random_integer`, który, jak prawdopodobnie domyśliłeś się na podstawie jego nazwy, zwraca losowo wybraną liczbę całkowitą.

```
resource "random_integer" "num_instances" {
  min = 1
  max = 3
}
```

Przedstawiony kod powoduje wygenerowanie losowo wybranej liczby całkowitej z przedziału od 1 do 3. Zobaczmy, co się stanie, jeśli dane wyjściowe tego zasobu będziesz chciał wykorzystać w parametrze `count` egzemplarza `aws_instance`.

```
resource "aws_instance" "example_3" {
  count          = random_integer.num_instances.result
  ami            = "ami-0c55b159cbfafa1f0"
  instance_type = "t2.micro"
}
```

Po wydaniu polecenia `terraform apply` zostanie wygenerowany komunikat błędu:

```
Error: Invalid count argument
  on main.tf line 30, in resource "aws_instance" "example_3":
  30:   count          = random_integer.num_instances.result
```

The "count" value depends on resource attributes that cannot be determined until apply, so Terraform cannot predict how many instances will be created. To work around this, use the `-target` argument to first apply only the resources that the count depends on.

Terraform wymaga możliwości obliczenia wartości `count` i `for_each` podczas fazy planowania, jeszcze *przed* utworzeniem lub zmodyfikowaniem jakiegokolwiek zasobu. To oznacza, że parametr `count` i wyrażenie `for_each` mogą odwoływać się do zdefiniowanych na stałe wartości, zmiennych, źródeł danych, a nawet list zasobów (o ile długość listy będzie mogła być ustalona na etapie planowania), ale nie na podstawie danych wyjściowych zasobu.

Nie można używać `count` i `for_each` w bloku konfiguracyjnym `module`

Być może będziesz chciał spróbować użyć parametru `count` w bloku konfiguracyjnym `module`, jak pokazałem w kolejnym fragmencie kodu:

```
module "count_example" {
  source = "../../../../../modules/services/webserver-cluster"

  count = 3

  cluster_name = "terraform-up-and-running-example"
  server_port  = 8080
  instance_type = "t2.micro"
}
```

Ten kod próbuje wykorzystać parametr `count` w bloku konfiguracji `module` w celu utworzenia trzech kopii zasobu `webserver-cluster`. Ewentualnie czasami będziesz chciał przypisać wartość 0 w bloku `module`, aby w ten sposób zapewnić jego opcjonalne dołączanie na podstawie wyniku pewnego warunku boolowskiego. Wprawdzie ten fragment kodu wydaje się być całkowicie prawidłowy, ale po wydaniu polecenia `terraform apply` wygenerowany zostanie komunikat błędu:

```
Error: Reserved argument name in module block
```

```
on main.tf line 13, in module "count_example":
13:   count = 3
```

The name "count" is reserved for use in a future version of Terraform.

Niestety, w aktualnej (0.12.6) wersji Terraform użycie parametru `count` lub wyrażenia `for_each` w bloku `module` jest nieobsługiwane. Zgodnie z informacjami dotyczącymi wydania Terraform 0.12 (<https://www.hashicorp.com/blog/hashicorp-terraform-0-12-preview-for-and-for-each/>) to ma się zmienić w przyszłości. Dlatego też, gdy czytasz te słowa, być może sytuacja jest już zupełnie odmienna. Możesz to sprawdzić w dokumencie *CHANGELOG*, który znajdziesz pod adresem <https://github.com/hashicorp/terraform/blob/master/CHANGELOG.md>.

Ograniczenia wdrożenia bez przestoju

Wykorzystanie ustawienia `create_before_destroy` w grupie ASG to doskonała technika wdrożenia bez przestoju, choć niestety niepozbawiona wad: dużym ograniczeniem jest to, że nie działa wraz z politykami automatycznego skalowania. Problem polega na wyzerowaniu wielkości grupy ASG do wartości `min_size` po każdym wdrożeniu, co może stanowić problem, jeśli polityki automatycznego skalowania wykorzystujesz do zwiększenia liczby działających serwerów.

Przykładowo moduł `webserver-cluster` zawiera kilka zasobów `aws_autoscaling_schedule` zwiększających o godzinie 9 liczbę serwerów klastra z 2 do 10. Jeżeli wdrożenie zostanie przeprowadzone np. o godzinie 11, nowa grupa ASG będzie składała się tylko z 2 serwerów zamiast z 10 i pozostanie w takiej postaci aż do godziny 9 następnego dnia.

Mamy parę potencjalnych rozwiązań tego problemu:

- Zmiana parametru `recurrence` w zasobie `aws_autoscaling_schedule` z wartości `0 9 * * *` oznaczającej „uruchomienie o godzinie 9” na wartość w postaci `0-59 9-17 * * *` oznaczającą „uruchomienie co minutę w godzinach od 9 do 17”. Jeżeli grupa ASG składa się aktualnie z 10 serwerów, ta polityka automatycznego skalowania nie przyniesie żadnego efektu, co akurat jest dobre. Jeśli natomiast grupa ASG została dopiero wdrożona, zastosowanie tej polityki gwarantuje, że grupa ASG będzie przez co najwyżej minutę składała się z mniejszej niż 10 liczby serwerów. Takie podejście przypomina sztuczkę, a nagły duży spadek liczby serwerów z 10 do 2, a następnie nagły wzrost liczby serwerów ponownie do 10 nadal może powodować problemy u użytkowników.
- Utworzenie własnego skryptu używającego API AWS do ustalenia liczby serwerów uruchomionych w grupie ASG, wywołanie tego skryptu za pomocą źródła danych `external` (więcej informacji na ten temat znajdziesz w następnym rozdziale), a następnie przypisanie parametrowi `desired_capacity` grupy ASG wartości zwróconej przez ten skrypt. W ten sposób niezależnie

od tego, czy zostanie utworzona nowa grupa ASG, zawsze będzie miała tę samą liczbę uruchomionych serwerów, równą liczbie serwerów działających w zastępowanej grupie ASG. Wadą stosowania własnych skryptów jest mniejsza przenośność kodu Terraform i nieco trudniejsza jego konserwacja.

W idealnej sytuacji Terraform będzie oferować doskonałą obsługę wdrożenia bez przestoju, ale w chwili gdy piszę te słowa (maj 2019), firma HashiCorp nie ma krótkoterminowych planów związanych z wprowadzeniem tej funkcjonalności (więcej informacji na ten temat znajdziesz na stronie <https://github.com/hashicorp/terraform/issues/1552>).

Awarie poprawnych planów

Czasami po wydaniu polecenia `terraform plan` otrzymujesz informacje o przygotowaniu całkowicie prawidłowego kodu, a wynikiem wykonania polecenia `terraform apply` jest błąd. Dla przykładu spróbuj dodać zasób `aws_iam_user` z dokładnie tą samą nazwą użytkownika, jakiej użyłeś w rozdziale 2. podczas ręcznego tworzenia użytkownika IAM.

```
resource "aws_iam_user" "existing_user" {  
  # Tę nazwę użytkownika powinieneś zmienić na dokładnie tę, której wcześniej użyłeś do  
  # utworzenia użytkownika IAM, co pozwoli na eksperymenty z poleceniem terraform import.  
  name = "yevgeniy.brikman"  
}
```

Po wydaniu polecenia `terraform plan` zostaniesz poinformowany, że kod przedstawia się prawidłowo i plan jest do zaakceptowania.

Terraform will perform the following actions:

```
# Nastąpi utworzenie zasobu aws_iam_user.existing_user.  
+ resource "aws_iam_user" "existing_user" {  
  + arn              = (known after apply)  
  + force_destroy    = false  
  + id               = (known after apply)  
  + name             = "yevgeniy.brikman"  
  + path             = "/"  
  + unique_id        = (known after apply)  
}
```

Plan: 1 to add, 0 to change, 0 to destroy.

Natomiast po wydaniu polecenia `terraform apply` otrzymasz następujący komunikat błędu:

```
Error: Error creating IAM User yevgeniy.brikman: EntityAlreadyExists:  
User with name yevgeniy.brikman already exists.
```

```
on main.tf line 10, in resource "aws_iam_user" "existing_user":  
10: resource "aws_iam_user" "existing_user" {
```

Problem polega na tym, że istnieje już użytkownik IAM o podanej nazwie. Taka sytuacja może się zdarzyć w przypadku nie tylko użytkowników IAM, ale także praktycznie każdego innego zasobu. Być może ktoś inny utworzył dany zasób ręcznie lub za pomocą poleceń CLI — w efekcie identyfikator jest taki sam jak już istniejącego zasobu, co prowadzi do konfliktu. Mamy wiele odmian tego błędu i początkujący często się przed nim nie chronią.

Kluczowe znaczenie ma tutaj to, że polecenie `terraform plan` szuka zasobów jedynie w pliku informacji o stanie Terraform. Jeżeli zasób jest tworzony w sposób *niepowodujący* umieszczenia informacji o nim w pliku informacji o stanie Terraform — np. za pomocą konsoli AWS — Terraform nie uwzględni ich podczas wykonywania polecenia `terraform plan`. W efekcie próba wykonania planu wyglądającego na prawidłowy zakończy się niepowodzeniem.

Z tej lekcji należy wyciągnąć dwa wnioski:

Gdy rozpoczniesz używanie Terraform, powinieneś stosować tylko Terraform.

Gdy pewna część infrastruktury jest zarządzana przez Terraform, nigdy nie powinieneś przeprowadzać w niej zmian w inny sposób. W przeciwnym razie nie tylko ryzykujesz powstanie dziwnych błędów Terraform, ale też tracisz wiele korzyści wynikających ze stosowania infrastruktury jako kodu, ponieważ nie będzie on dłużej wiernie odzwierciedlał danej infrastruktury.

Jeżeli masz istniejącą strukturę, skorzystaj z polecenia `terraform import`.

Jeżeli infrastrukturę utworzyłeś przed rozpoczęciem pracy z Terraform, zawsze możesz skorzystać z polecenia `terraform import` w celu dodania tej infrastruktury do pliku informacji o stanie Terraform, co pozwoli Terraform rozpocząć zarządzanie tą infrastrukturą. Polecenie `terraform import` pobiera dwa argumenty. Pierwszy to „adres” zasobu w plikach konfiguracyjnych Terraform. Wykorzystywana jest przy tym taka sama składnia jak w przypadku odwołania do zasobu — `<DOSTAWCA>_<TYP>.<NAZWA>`, np. `aws_iam_user.existing_user`. Drugi to identyfikator zasobu wskazujący zasób przeznaczony do zaimportowania. Przykładowo identyfikator zasobu `aws_iam_user` jest nazwą użytkownika (np. `yevgeniy.brikman`), natomiast identyfikator `aws_instance` to identyfikator egzemplarza EC2 (np. `i-190e22e5`). Dokumentacja zamieszczona na dole strony każdego zasobu zwykle zawiera informacje o tym, jak można go zaimportować.

Dla przykładu — oto polecenie `terraform import` możliwe do użycia w celu zsynchronizowania dodanego przed chwilą w konfiguracji Terraform zasobu `aws_iam_user` z użytkownikiem IAM utworzonym w rozdziale 2. (oczywiście nazwę `yevgeniy.brikman` powinieneś zastąpić własną nazwą).

```
$ terraform import aws_iam_user.existing_user yevgeniy.brikman
```

Terraform użyje API AWS do odszukania użytkownika IAM i utworzy w pliku informacji o stanie wpis łączący użytkownika i zasób `aws_iam_user.existing_user` w konfiguracji Terraform. Od tego momentu po wydaniu polecenia `terraform plan` Terraform wie o istnieniu użytkownika IAM i dlatego nie spróbuje utworzyć go ponownie.

Zwróć uwagę, że jeśli masz dużo istniejących zasobów, które mają być zaimportowane do Terraform, to utworzenie dla nich kodu Terraform zupełnie od początku i jednocześnie importowanie może być naprawdę uciążliwe. W takich przypadkach warto zainteresować się narzędziem takim jak *Terraforming* (<http://terraforming.dtan4.net/>), które potrafi automatycznie zaimportować z konta AWS kod i informacje o stanie.

Trudności podczas refaktoryzacji

Często stosowaną praktyką programistyczną jest *refaktoryzacja* oznaczająca zmianę wewnętrznych szczegółów istniejącego fragmentu kodu bez modyfikowania jej zewnętrznego sposobu działania.

Celem refaktoryzacji jest poprawienie czytelności, ułatwienie konserwacji i ogólne usprawnienie kodu źródłowego. Refaktoryzacja ma kluczowe znaczenie podczas tworzenia kodu i należy stosować ją regularnie. Jednak w przypadku Terraform, dowolnej infrastruktury jako kodu, trzeba zachować dużą ostrożność w zakresie definicji „zachowania zewnętrznego” fragmentu kodu, w przeciwnym razie możesz mieć spore problemy.

Dość często stosowaną praktyką podczas refaktoryzacji jest np. zmiana nazwy zmiennej lub funkcji na znacznie czytelniejszą. Wiele środowisk IDE oferuje nawet wbudowaną obsługę refaktoryzacji i potrafi wyręczyć programistę podczas takiej zmiany w całej bazie kodu. Wprawdzie zmiana nazwy to operacja, którą w językach programowania ogólnego przeznaczenia można przeprowadzić bez większego zastanawiania się, w przypadku Terraform trzeba zachować ogromną ostrożność, ponieważ skutkiem może być poważna awaria.

Przykładowo moduł `webserver-cluster` ma zmienną danych wejściowych o nazwie `cluster_name`:

```
variable "cluster_name" {
  description = "Nazwa używana dla wszystkich zasobów klastra"
  type        = string
}
```

Być może zaczniesz używać tego modułu do wdrażania mikrousług i, początkowo, nazwę mikrousługi zdefiniujesz jako `foo`, aby później postanowić o jej zmianie na `bar`. Wprawdzie to może wydawać się prostą operacją, ale może spowodować problemy.

Moduł `webserver-cluster` wykorzystuje zmienną o nazwie `cluster_name` w wielu zasobach, w tym w parametrach `name` dwóch grup bezpieczeństwa oraz w mechanizmie równoważenia obciążenia:

```
resource "aws_lb" "example" {
  name             = var.cluster_name
  load_balancer_type = "application"
  subnets         = data.aws_subnet_ids.default.ids
  security_groups  = [aws_security_group.alb.id]
}
```

Jeżeli zmienisz wartość parametru `name` pewnych zasobów, Terraform usunie starszą wersję zasobu i utworzy zastępującą ją nową wersję. Jeżeli usuwanym zasobem jest mechanizm równoważenia obciążenia, zabraknie komponentu pozwalającego na przekazywanie ruchu sieciowego do klastra serwera WWW, przynajmniej do chwili uruchomienia nowego egzemplarza mechanizmu równoważenia obciążenia. Podobnie, jeśli usuwanym zasobem jest grupa bezpieczeństwa, serwer będzie odrzucał cały ruch sieciowy aż do chwili utworzenia nowej grupy.

Inną operacją refaktoryzacji, którą możesz rozważyć, jest zmiana identyfikatora Terraform. Dla przykładu spójrz na zasób grupy `aws_security_group` w module `webserver-cluster`:

```
resource "aws_security_group" "instance" {
  # (...)
}
```

Identyfikator tego zasobu nosi nazwę `instance`. Podczas refaktoryzacji możesz uznać, że lepszą nazwą dla tego zasobu będzie `cluster_instance`:

```
resource "aws_security_group" "cluster_instance" {
  # (...)
}
```

Jaki będzie wynik takiej zmiany? Masz rację: przestój.

Terraform powiązuje każdy identyfikator zasobu z identyfikatorem dostawcy chmury — mamy więc powiązanie zasobu `iam_user` z identyfikatorem użytkownika IAM w AWS oraz powiązanie zasobu `aws_instance` z identyfikatorem egzemplarza AWS EC2. Jeżeli zmienisz identyfikator zasobu, np. identyfikator zasobu `aws_security_group` z `instance` na `cluster_instance`, z perspektywy Terraform nastąpi usunięcie starego zasobu i dodanie zupełnie nowego. W efekcie po wydaniu polecenia `terraform apply` Terraform usunie starą grupę bezpieczeństwa i utworzy nową, natomiast między tymi operacjami serwery będą odrzucały cały ruch sieciowy.

Z tej lekcji powinieneś wyciągnąć cztery wnioski:

Zawsze używaj polecenia `terraform plan`.

Wszystkie te problemy można wychwycić po wydaniu polecenia `terraform plan`, dokładnym przeanalizowaniu wygenerowanych danych wyjściowych tego polecenia i zwróceniu uwagi na zasoby, które Terraform chce usunąć, choć tego prawdopodobnie nie chcesz.

Stosuj ustawienie `create_before_destroy`.

Jeżeli chcesz zastąpić zasób, zastanów się dobrze nad utworzeniem jego zamiennika przed usunięciem pierwotnego zasobu. Jeśli się na to zdecydujesz, możesz skorzystać z `create_before_destroy`. Ewentualnie ten sam efekt można uzyskać przez samodzielne wykonanie dwóch kroków. Pierwszy: dodanie nowego zasobu do konfiguracji i wydanie polecenia `terraform apply`. Drugi: usunięcie starego zasobu z konfiguracji i ponowne wydanie polecenia `terraform apply`.

Zmiana identyfikatora wymaga zmiany stanu.

Jeżeli chcesz zmienić identyfikator powiązany z zasobem (np. identyfikator zasobu `aws_security_group` z `instance` na `cluster_instance`) bez przypadkowego usunięcia i odtworzenia zasobu, musisz odpowiednio uaktualnić informacje o stanie Terraform. Plik zawierający informacje o stanie nigdy nie powinien być uaktualniany ręcznie — zamiast tego należy skorzystać z polecenia `terraform state`. W szczególności podczas zmiany identyfikatorów należy wykonać polecenie `terraform state mv`, którego składnia przedstawia się następująco:

```
terraform state mv <ODWOŁANIE_PIERWOTNE> <ODWOŁANIE_NOWE>
```

gdzie `ODWOŁANIE_PIERWOTNE` to wyrażenie odwołujące się do obecnego zasobu, natomiast `ODWOŁANIE_NOWE` to nowe położenie dla tego zasobu. Przykładowo, jeśli zmieniasz nazwę identyfikatora zasobu `aws_security_group` z `instance` na `cluster_instance`, musisz wydać następujące polecenie:

```
$ terraform state mv \
  aws_security_group.instance \
  aws_security_group.cluster_instance
```

To wskazuje Terraform, że stan użyty do powiązania z `aws_security_group.instance` powinien być teraz powiązany z `aws_security_group.cluster_instance`. Jeżeli zdecydujesz się na zmianę identyfikatora i wydasz to polecenie, o prawidłowym przeprowadzeniu operacji będzie świadczyło to, że kolejne wydanie polecenia `terraform plan` nie wskaże żadnych zmian.

Część parametrów jest niemodyfikowalna.

Parametry wielu zasobów są niemodyfikowalne, więc jeśli je zmienisz, Terraform usunie stary zasób, a następnie w jego miejsce utworzy nowy. Dokumentacja poszczególnych zasobów często dokładnie wskazuje, co się stanie po zmianie parametru. Dlatego też dobrze jest do niej zaglądać. Warto w tym miejscu ponownie przypomnieć, aby zawsze wykonywać polecenie `terraform plan` i rozważyć użycie strategii `create_before_destroy`.

Osiągnięcie ostatecznej spójności może wymagać nieco czasu

API wybranych dostawców chmury, np. AWS, jest asynchroniczne i spójne. W tym kontekście *asynchroniczność* oznacza, że API może udzielić odpowiedzi natychmiast, bez oczekiwania na zakończenie działania żądanej akcji. *Ostateczna spójność* oznacza, że propagowanie zmiany w całym systemie może zająć nieco czasu, więc w pewnym okresie może pojawić się niespójność w zależności od tego, która replika magazynu danych udzieliła odpowiedzi na wywołanie API.

Dla przykładu przyjmując założenie o wykonywaniu wywołania API do AWS w celu utworzenia egzemplarza EC2. API zwraca mniej więcej od razu komunikat informujący o „sukcesie” (np. kod stanu 201 oznaczający utworzenie zasobu), bez oczekiwania na zakończenie operacji tworzenia egzemplarza EC2. Jeżeli natychmiast spróbujesz nawiązać połączenie z tym egzemplarzem, wynikiem prawdopodobnie będzie niepowodzenie, ponieważ AWS nadal przygotowuje egzemplarz lub jego uruchomienie jeszcze się nie zakończyło. Co więcej, po wykonaniu kolejnego wywołania API w celu pobrania informacji o tym egzemplarzu EC2 wynikiem będzie komunikat błędu (np. kod stanu 404 oznaczający nieznaalezienie zasobu). Tak się dzieje, ponieważ informacje o egzemplarzu EC2 wciąż mogą być przekazywane w AWS i potrzeba trochę czasu, zanim te egzemplarze staną się ogólnie dostępne.

W skrócie: gdy używasz asynchronicznego i ostatecznie spójnego API, musisz chwilę poczekać na zakończenie tej operacji i propagowanie zmian. Niestety, SDK AWS nie oferuje dobrych narzędzi w tym zakresie, a narzędzie Terraform zostało zalane wieloma błędami podobnymi do zgłoszonego pod numerem 6813 (<https://github.com/hashicorp/terraform/issues/6813>):

```
$ terraform apply
```

```
aws_subnet.private-persistence.2: InvalidSubnetID.NotFound:
The subnet ID 'subnet-xxxxxxx' does not exist
```

W omawianym przykładzie tworzysz zasób (np. podsieć), następnie próbujesz wyszukać pewne informacje o tym zasobie (np. identyfikator nowo utworzonej podsieci), a Terraform nie potrafi ich znaleźć. Wprawdzie większość takich błędów (w tym wspomniany 6813) została usunięta, ale podobne pojawiają się od czasu do czasu, zwłaszcza gdy do Terraform zostanie dodana obsługa nowego typu zasobu. Takie błędy są irytujące, choć na szczęście większość z nich pozostaje nieszkodliwa. Po ponownym wydaniu polecenia `terraform apply` wszystko powinno działać zgodnie z oczekiwaniami, ponieważ zanim zdążysz ponownie wydać polecenie, niezbędne informacje zostały propagowane w systemie.

Podsumowanie

Wprawdzie Terraform to język deklaracyjny, ale mimo to zawiera wiele narzędzi, takich jak poznane w rozdziale 4. zmienne i moduły oraz poznane w tym rozdziale konstrukcje: `count`, `for_each`, `for`, `create_before_destroy` i funkcje wbudowane, które zapewniają językowi niezwykłą elastyczność i ekspresję. W rozdziale przedstawiłem wiele rodzajów sztuczek związanych z konstrukcją `if`, więc poświęć trochę czasu na przejrzanie dokumentacji funkcji (<https://www.terraform.io/docs/configuration/functions.html>) i popuść wodze fantazji hakera. OK, może nie aż tak bardzo, ponieważ ktoś później będzie musiał nadal zajmować się obsługą Twojego kodu. Możesz zaszaleć na tyle, na ile pozwala tworzenie przejrzystego i eleganckiego API dla modułów.

Przechodzimy teraz do rozdziału 6., z którego dowiesz się, jak tworzyć nie tylko przejrzyste i eleganckie moduły, ale także moduły jakości produkcyjnej — takie, na których firma może opierać swoje działania.

Produkcyjny kod Terraform

Tworzenie infrastruktury o jakości produkcyjnej jest zadaniem trudnym, stresującym i czasochłonnym. Gdy używam określenia *infrastruktura o jakości produkcyjnej*, mam na myśli serwery, magazyny danych, mechanizmy równoważenia obciążenia, funkcjonalność związaną z zapewnieniem bezpieczeństwa, monitorowanie infrastruktury i generowanie powiadomień, tworzenie potoków oraz wszelkie inne zadania niezbędne do prowadzenia biznesu. Twoja firma liczy na Ciebie: ufa, że infrastruktura nie zawiedzie po zwiększeniu się poziomu ruchu sieciowego, dane nie zostaną utracone w przypadku awarii serwera, nie będą zagrożone w razie próby włamania się do systemu przez hakerów itd. — jeżeli jest inaczej, firma może wypaść z gry. To mam na myśli, gdy w rozdziale odwołuję się do infrastruktury o jakości produkcyjnej.

Miałem okazję pracować z setkami firm i na podstawie zebranych doświadczeń jestem w stanie określić, ile czasu potrzeba na przygotowanie infrastruktury o jakości produkcyjnej:

- Jeżeli chcesz wdrożyć usługę w pełni zarządzaną przez podmiot zewnętrzny, np. bazę danych MySQL działającą w usłudze AWS Relational Database Service (RDS), możesz oczekiwać, że przygotowanie jej do produkcji zajmie od tygodnia do dwóch.
- Jeżeli chcesz uruchomić własną, bezstanową aplikację rozproszoną, np. klaster aplikacji Node.js nieprzechowujących lokalnie danych (czyli przechowujących wszystkie dane w usłudze RDS), działający na bazie grupy AWS ASG, to czas potrzebny na przygotowanie infrastruktury o jakości produkcyjnej wydłuży się dwukrotnie: od dwóch do czterech tygodni.
- Jeżeli chcesz uruchomić własną aplikację rozproszoną zawierającą informacje o stanie, np. klaster Amazon Elasticsearch (Amazon ES) działający na bazie grupy ASG i przechowujący dane na dyskach lokalnych), to czas potrzebny na przygotowanie infrastruktury produkcyjnej ponownie się wydłuży, do 2 – 4 miesięcy.
- Jeżeli chcesz zbudować całą architekturę łącznie z wszystkimi aplikacjami, magazynami danych, mechanizmami równoważenia obciążenia, systemami monitorowania, powiadamiania i zapewnienia bezpieczeństwa itd., czas znów się wydłuży, do 6 – 36 miesięcy — w przypadku mniejszych firm to będzie bliżej 6 miesięcy, natomiast w przypadku większych może zająć lata.

Podsumowanie tych danych zamieściłem w tabeli 6.1.

Tabela 6.1. Czas potrzebny na przygotowanie od zera infrastruktury o jakości produkcyjnej

Typ infrastruktury	Przykład	Przewidywany czas
Usługa zarządzana	Amazon RDS	1 – 2 tygodnie
Samozarządzany system rozproszony (bezstanowy)	Klaster aplikacji Node.js	2 – 4 tygodnie
Samozarządzany system rozproszony (stanowy)	Amazon ES	2 – 4 miesiące
Cała architektura	Aplikacje, magazyny danych, mechanizmy równoważenia obciążenia, monitorowanie itd.	6 – 36 miesięcy

Jeżeli nigdy wcześniej nie zajmowałeś się procesem tworzenia infrastruktury o jakości produkcyjnej, możesz być zaskoczony tym, ile czasu potrzeba na jej przygotowanie. Bardzo często spotykałem się z następującymi reakcjami:

- „Jak to możliwe, że potrzeba aż tak długiego czasu?”
- „Serwer w <nazwa chmury> mogę wdrożyć w ciągu kilku minut. Pozostała część zadania nie może wymagać miesięcy pracy!”
- I bardzo często od wielu pewnych siebie inżynierów — „Jestem przekonany, że te liczby dotyczą innych osób, ja jestem w stanie to zrobić w ciągu kilku dni”.

Każdy, kto przeszedł przez poważną migrację chmury lub zajmował się przygotowaniem od zera zupełnie nowej infrastruktury, wie, że podany czas jest optymistyczny i dotyczy sytuacji, gdy nie pojawiają się żadne nieprzewidywane trudności. Jeżeli w zespole nie masz osób z dużym doświadczeniem w tworzeniu infrastruktury o jakości produkcyjnej lub jeśli zespół podąża w dziesiątkach różnych kierunków i nie znajdujesz czasu na to, aby się skoncentrować, proces będzie trwał znacznie dłużej.

W rozdziale zamierzam wyjaśnić, dlaczego przygotowanie infrastruktury o jakości produkcyjnej zabiera tak dużo czasu, co właściwie oznacza jakość produkcyjna, jakie rozwiązania sprawdzają się najlepiej podczas tworzenia wielokrotnego użycia modułów o jakości produkcyjnej:

- Dlaczego przygotowanie infrastruktury o jakości produkcyjnej zabiera tak dużo czasu?
- Lista rzeczy do zrobienia podczas tworzenia infrastruktury o jakości produkcyjnej.
- Moduły infrastruktury o jakości produkcyjnej:
 - moduły małe,
 - moduły złożone,
 - moduły możliwe do testowania,
 - moduły możliwe do wydania,
 - moduły poza Terraform.



Przykładowe fragmenty kodu

Przypominam, że wszystkie przykładowe fragmenty kodu znajdują się w repozytorium GitHub pod adresem <https://github.com/brikis98/terraform-up-and-running-code>.

Dlaczego przygotowanie infrastruktury o jakości produkcyjnej trwa tak długo?

Czas potrzebny na realizację projektów oprogramowania notorycznie jest szacowany nieprawidłowo. To dotyczy również projektów DevOps. Szybka zmiana, która miała zająć zaledwie 5 minut, zabrała jednak cały dzień. Mniejsza funkcjonalność, której przygotowanie miało zakończyć się w ciągu dnia, wymagała dwóch tygodni pracy. Aplikacja, która miała być w produkcji po dwóch tygodniach, po upływie sześciu miesięcy nadal nie jest gotowa. Infrastruktura i projekty DevOps, a prawdopodobnie także wiele innych typów oprogramowania, są doskonałymi przykładami prawa Hofstadtera¹:

Prawo Hofstadtera: wszystko zajmuje więcej czasu, niż zakładasz, nawet jeśli weźmiesz pod uwagę prawo Hofstadtera.

— Douglas R. Hofstadter

Myszę, że to jest spowodowane trzema czynnikami. Pierwszy: ruch DevOps, podobnie jak cała dziedzina, nadal znajduje się w epoce kamienia łupanego. Nie mam tutaj zamiaru nikogo obrażać, raczej chcę zwrócić uwagę na to, że cała dziedzina wciąż znajduje się na wczesnym etapie rozwoju. Terminy „przetwarzanie w chmurze”, „infrastruktura jako kod” i „DevOps” pojawiły się zaledwie dekadę temu, a narzędzia takie jak Terraform, Docker, Packer i Kubernetes istnieją od kilku lat. Wszystkie te narzędzia i techniki są względnie nowe i bardzo szybko się zmieniają. To oznacza, że nie zaliczają się do szczególnie dopracowanych i niewielka grupa osób ma doświadczenie w pracy z nimi, więc nie powinno być zaskoczeniem, że realizacja projektów zajmuje więcej czasu, niż przewidywano.

Drugi: uczestnicy ruchu DevOps są dość łatwo podatni na tzw. *yak shaving*. Jeżeli jeszcze nie spotkałeś się z tym wyrażeniem, jestem pewien, że będzie to pojęcie, które pokochasz (i znienawidzisz). Najlepsza definicja *yak shaving*, z jaką się dotąd spotkałem, pochodzi z posta opublikowanego na blogu przez Setha Godina²:

„Mam zamiar dzisiaj nawoskować samochód”.

„Ups, ten wąż jest zepsuty po zimie. Muszę pojechać do sklepu i kupić nowy”.

„Jednak ten sklep znajduje się po drugiej stronie mostu i dostanie się do niego bez mojej przepustki będzie kosztowne ze względu na opłatę za przejazd mostem”.

„Chwila, moment! Mogłbym pożyczyć przepustkę od sąsiada...”

„Nie, on nie pożyczy mi przepustki, dopóki nie zwrócę poduszki Moshi Moshi, którą pożyczył od niego mój syn”.

¹ Douglas R. Hofstadter, *Go del, Escher, Bach: An Eternal Golden Braid*. 20th Anniversary edition, Basic Books, New York 1999.

² Seth Godin, *Don't Shave That Yak!*, https://seths.blog/2005/03/dont_shave_that/, 5 marca 2005.

„Nie oddaliśmy jeszcze tej poduszki, ponieważ część jej wypełnienia wypadła i musimy kupić nowe”.

Następnie zdajesz sobie sprawę, że znajdujesz się w zoo i cały czas mówisz o konieczności nawoskowania samochodu.

— Seth Godin

Pojęcie *yak shaving* oznacza te wszystkie drobne, pozornie niepowiązane ze sobą zadania wykonywane, zanim będzie można zrobić zadanie, którym pierwotnie chciałeś się zająć. Jeżeli zajmujesz się tworzeniem oprogramowania, a szczególnie gdy pracujesz w branży stosującej praktyki DevOps, prawdopodobnie już setki razy spotykałeś się z taką sytuacją. Zamierzasz wdrożyć naprawę drobną poprawkę, a wprowadzasz błąd w konfiguracji aplikacji. Wdrażasz więc poprawkę dla konfiguracji aplikacji, a tutaj okazuje się, że masz błąd związany z certyfikatem TLS. Po spędzeniu kilku godzin na lekturze serwisu StackOverflow wreszcie udaje się rozwiązać problem z certyfikatem TLS. Próbujesz ponownie wdrożyć aplikację, ale tym razem niepowodzenie jest skutkiem pewnego błędu w systemie wdrożenia. Poświęcasz godziny na rozwiązanie problemu tylko po to, aby przekonać się, że problem wynika z nieaktualnej wersji systemu Linux. Kolejnym zadaniem jest więc uaktualnienie systemu operacyjnego w całej flocie serwerów, aby można było wdrożyć „szybką” poprawkę polegającą na zmianie jednego znaku.

Programiści DevOps wydają się być szczególnie podatni na takie incydenty. Po części wynika to z konsekwencji niedopracowania technologii DevOps i projektu nowoczesnych systemów, co często prowadzi do konieczności ścisłego powiązania i powielania infrastruktury. Każda zmiana wprowadzana w świecie DevOps przypomina operację wyciągnięcia jednego przewodu USB z pudełka zawierającego splątane ze sobą przewody — w efekcie z pudełka wyciąga się praktycznie wszystkie pozostałe przewody. Po części może to wiązać się z tym, że termin „DevOps” ma naprawdę szerokie znaczenie: wszystko, od tworzenia, przez wdrażanie, aż po zapewnienie bezpieczeństwa.

W ten sposób docieramy do trzeciego czynnika decydującego o długości zadań wykonywanych w ruchu DevOps. Dwa pierwsze mogą zostać sklasyfikowane jako *złożoność przypadkowa*. To pojęcie odwołuje się do problemu powodowanego przez określone narzędzia i wybrany proces, w przeciwieństwie do *złożoności zasadniczej*, która odwołuje się do problemów nieodłącznie związanych z tym, nad czym pracujesz³. Przykładowo, jeśli do utworzenia algorytmów związanych z transakcjami giełdowymi wykorzystasz język C++, zmaganie się z błędami alokacji pamięci jest złożonością przypadkową — gdybyś wybrał inny język programowania, zapewniający automatyczne zarządzanie pamięcią, w ogóle nie miałbyś tego problemu. Natomiast opracowanie algorytmu pozwalającego na podejmowanie trafnych decyzji giełdowych zalicza się do *złożoności zasadniczej* — ten problem musisz rozwiązać niezależnie od wybranego języka programowania.

Trzeci czynnik powodujący wydłużenie zadań wykonywanych w ramach DevOps — *złożoność zasadnicza* danego problemu — wiąże się z długą listą zadań koniecznych do wykonania w celu przygotowania infrastruktury gotowej do zastosowania w środowisku produkcyjnym. Największy problem polega na tym, że większość programistów nie zna większości zadań znajdujących się na tej liście. Dlatego też podczas szacowania czasu potrzebnego na wykonanie projektu zapominają o ogromnej

³ Frederick P. Brooks jr., *The Mythical Man-Month: Essays on Software Engineering. Anniversary edition*, Reading, MA: Addison-Wesley Professional, 1995.

liczbie ważnych — i czasochłonnych — szczegółów. Wspomniana wcześniej lista jest tematem kolejnego podrozdziału.

Lista rzeczy do zrobienia podczas tworzenia infrastruktury o jakości produkcyjnej

Oto przykład ciekawego eksperymentu: zapytaj pracowników firmy, jakie są kroki konieczne do podjęcia przed przejściem do środowiska produkcyjnego. W większości firm, gdy zadasz to pytanie pięciu osobom, otrzymasz pięć odmiennych odpowiedzi. Jedna osoba wspomni o wskaźnikach i powiadomieniach, następna o planowaniu i zapewnieniu wysokiej dostępności, kolejna będzie mówiła o testach zautomatyzowanych i analizie kodu, a jeszcze inna o szyfrowaniu, uwierzytelnianiu i zabezpieczeniu serwera. Jeżeli będziesz mieć szczęście, ktoś może poruszyć temat tworzenia kopii zapasowej i agregacji dziennika zdarzeń. Większość firm nie ma jasnej definicji wymagań koniecznych do spełnienia przed przejściem do produkcji. To oznacza, że poszczególne fragmenty infrastruktury są wdrażane nieco odmiennie i mogą być pozbawione niezbędnej funkcjonalności.

Aby pomóc w poprawieniu sytuacji w tym zakresie, chciałbym podzielić się listą rzeczy do sprawdzenia podczas przygotowywania infrastruktury przeznaczonej do wdrożenia w środowisku produkcyjnym (tabela 6.2). Ta lista zawiera większość elementów o znaczeniu kluczowym, które trzeba wziąć pod uwagę podczas przygotowywania wspomnianej infrastruktury.

Tabela 6.2. Lista rzeczy do sprawdzenia podczas przygotowywania infrastruktury przeznaczonej do wdrożenia w środowisku produkcyjnym

Zadanie	Opis	Przykładowe narzędzia
Instalowanie	Instalacja plików binarnych oprogramowania i wszystkich zależności	Bash, Chef, Ansible, Puppet
Konfigurowanie	Konfigurowanie oprogramowania w środowisku uruchomieniowym. Obejmuje m.in. ustawienia portu, certyfikaty TLS, wykrywanie usług, serwery główne i podrzędne, replikację itd.	Bash, Chef, Ansible, Puppet
Provisioning	Przygotowanie infrastruktury, czyli m.in. serwery, mechanizm równoważenia obciążenia, konfiguracja sieci, ustawienia zapory sieciowej, uprawnienia IAM itd.	Terraform, CloudFormation
Wdrażanie	Wdrażanie usługi w przygotowanej infrastrukturze. Wprowadzanie uaktualnień bez przestoju. To obejmuje m.in. wdrożenia typu niebieski-zielony, ciągłe i kanarkowe	Terraform, CloudFormation, Kubernetes, ECS
Wysoka dostępność	Odporność na awarie procesów, serwerów, usług, centrów danych itd.	Wykorzystanie wielu centrów danych, replikacji w wielu regionach, automatyczne skalowanie, stosowanie mechanizmu równoważenia obciążenia

Tabela 6.2. Lista rzeczy do sprawdzenia podczas przygotowywania infrastruktury przeznaczonej do wdrożenia w środowisku produkcyjnym (ciąg dalszy)

Zadanie	Opis	Przykładowe narzędzia
Skalowanie	Skalowanie w górę lub w dół w odpowiedzi na aktualne obciążenie. Skalowanie poziome (więcej serwerów) i (lub) pionowe (większe serwery)	Automatyczne skalowanie, sharding replikacji, buforowanie, stosowanie zasady dziel i rządź
Zapewnienie wydajności	Optymalizacja użycia procesora, pamięci, dysku, sieci i procesora graficznego. To obejmuje m.in. dostrajanie zapytań, wykonywanie testów wydajności, sprawdzanie obciążenia i profilowanie	Dynatrace, valgrind, VisualWM, ab, Jmeter
Obsługa sieci	Konfigurowanie statycznych i dynamicznych adresów IP, portów, wykrywania usług, zapór sieciowych, DNS, dostępu SSH i VPN	VPC, zapory sieciowe, routery, serwery DNS, OpenVPN
Zapewnienie bezpieczeństwa	Szyfrowanie transmisji (TLS) i danych na dysku, uwierzytelnianie, autoryzacja, zarządzanie danymi wrażliwymi, zabezpieczanie serwera	ACM, Let's Encrypt, KMS, Cognito, Vault, CIS
Obsługa wskaźników	Dostępność wskaźników m.in. biznesowych, aplikacji i serwera, obsługa zdarzeń, monitorowania, śledzenia i powiadamiania	CloudWatch, DataDog, New Relic, Honeycomb
Obsługa dzienników zdarzeń	Rotacja dzienników zdarzeń na dysku. Agregowanie w pewnym położeniu centralnym danych dzienników zdarzeń	CloudWatch Logs, ELK, Sumo Logic, Papertrail
Tworzenie kopii zapasowej i przywracanie z niej danych	Tworzenie kopii zapasowej baz danych, obsługa buforów oraz innych danych w regularnych odstępach czasu. Replikacja danych do innego regionu lub konta	RDS, ElastiCache, replikacja
Optymalizacja kosztów	Wybór odpowiednich egzemplarzy, używanie egzemplarzy typu spot zarezerwowanych, automatyczne skalowanie i pozbywanie się nieużywanych zasobów	Automatyczne skalowanie, egzemplarze typu spot i zarezerwowane
Obsługa dokumentacji	Dokumentowanie kodu, architektury i praktyk. Tworzenie tzw. playbooków w celu reagowania na incydenty	Pliki README, wiki, Slack
Przeprowadzanie testów	Tworzenie testów zautomatyzowanych dla infrastruktury jako kodu. Wykonywanie testów po wprowadzeniu każdej zmiany oraz każdej nocy	Terratest, inspec, serverspec, kitchen-terraform

Większość programistów ma świadomość istnienia kilku pierwszych zadań: instalowania, konfigurowania, provisioningu i wdrożenia. Pozostałe zadania okazują się dla nich zaskoczeniem. Dla przykładu — czy zastanawiałeś się nad odpornością usługi na awarie oraz nad tym, co się stanie w przypadku wyłączenia serwera? Lub po awarii mechanizmu równoważenia obciążenia? Co będzie, jeśli centrum danych zostanie odcięte od sieci? Zadania związane z konfiguracją sieci są zwykle trudne: konfiguracja VPC, VPN, wykrywania usług i dostępu SSH to przykłady podstawowych zadań, które mogą zabrać miesiące oraz często nie są uwzględniane w planach projektów i ramach czasowych. Zadania związane z zapewnieniem bezpieczeństwa, takie jak szyfrowanie danych za pomocą TLS w transporcie, uwierzytelnianie i określanie sposobu na przechowywanie danych wrażliwych, są często pomijane i pozostawiane na sam koniec.

Do przedstawionej tutaj listy powracaj za każdym razem, gdy pracujesz nad nowym komponentem infrastruktury. Nie każdy komponent będzie wymagał wszystkich elementów tej listy, choć powinieneś spójnie i wyraźnie dokumentować to, co zostało zaimplementowane, oraz to, co postanowiłeś pominąć (w takim przypadku wyjaśnij powody tej decyzji).

Moduły infrastruktury o jakości produkcyjnej

Skoro poznałeś listę rzeczy do zrobienia podczas tworzenia poszczególnych komponentów infrastruktury, możemy przejść do najlepszych praktyk związanych z budowaniem wielokrotnego użycia modułów implementujących te zadania. Oto lista tematów, które zostaną poruszone:

- małe moduły,
- moduły łączone z innymi,
- moduły możliwe do przetestowania,
- moduły możliwe do wydania,
- moduły wykraczające poza Terraform.

Małe moduły

Programiści dopiero rozpoczynający pracę z Terraform i ogólnie z praktykami IaC bardzo często definiują całą infrastrukturę dla wszystkich środowisk (programistyczne, robocze, produkcyjne itd.) w pojedynczym pliku lub module. Jak już wspomniałem w rozdziale 3., to jest zły pomysł. W rzeczywistości można pójść o krok dalej i przyjąć następujące założenie: ogromne moduły — czyli zawierające więcej niż kilkaset wierszy kodu lub wdrażające więcej niż tylko kilka ściśle powiązanych ze sobą elementów infrastruktury — powinny być uznawane za szkodliwe.

Oto zaledwie kilka wad ogromnych modułów:

Ogromne moduły są wolne.

Jeżeli cała infrastruktura zostanie zdefiniowana w jednym module Terraform, wykonanie każdego polecenia będzie zabierać mnóstwo czasu. Spotkałem się już z na tyle ogromnymi modułami, że wykonanie polecenia `terraform plan` zajmowało 5 – 6 minut!

Ogromne moduły są niebezpieczne.

Jeżeli cała infrastruktura będzie zarządzana za pomocą pojedynczego modułu, to w celu zmiany czegokolwiek będą potrzebne uprawnienia dostępu do wszystkiego. To oznacza, że praktycznie każdy użytkownik musi być administratorem, co jest sprzeczne z *zasadą najmniejszych uprawnień*.

Ogromne moduły są ryzykowne.

Jeżeli wszystko znajduje się w jednym komponencie, wprowadzenie w nim błędu może zniszczyć całą infrastrukturę. Możesz wprowadzać tylko drobną zmianę w aplikacji środowiska roboczego, ale ze względu na literówkę lub nieprawidłowe polecenie spowodujesz usunięcie produkcyjnej bazy danych.

Ogromne moduły są trudne do zrozumienia.

Im więcej kodu w jednym miejscu, tym trudniej jest jednej osobie go zrozumieć. Jeżeli nie rozumiesz infrastruktury, z którą pracujesz, będziesz nieustannie popełniać błędy.

Ogromne moduły są trudne do przeanalizowania.

Analiza modułu zawierającego kilkadziesiąt wierszy kodu jest łatwa, natomiast analiza modułu składającego się z tysięcy wierszy jest praktycznie niemożliwa. Co więcej, wykonanie polecenia `terraform plan` będzie trwało długo, a jeśli wygenerowane dane wyjściowe będą zawierały kilka tysięcy wierszy, nikt nie zada sobie trudu zapoznania się z tymi danymi. W efekcie nikt nie zauważy małego komunikatu w kolorze czerwonym informującego o usunięciu bazy danych.

Ogromne moduły są trudne do przetestowania.

Testowanie kodu infrastruktury jest trudne, testowanie ogromnej ilości kodu infrastruktury jest niemalże niemożliwe. Do tego tematu powrócę w rozdziale 7.

Ujmując rzecz najkrócej, kod powinien składać się z małych modułów, z których każdy wykonuje po jednym zadaniu. To nie jest podejście nowe lub kontrowersyjne. Prawdopodobnie słyszałeś to już setki razy, choć w nieco odmiennych kontekstach. Oto wersja pochodząca z książki *Czysty kod. Podręcznik dobrego programisty*⁴:

Pierwszą regułą funkcji jest to, że powinna być mała. Drugą regułą funkcji jest to, że powinna być jeszcze mniejsza.

— Robert C. Martin

Wyobraź sobie, że używasz języka programowania ogólnego przeznaczenia, takiego jak Java, Python lub Ruby, i utworzyłeś ogromną funkcję składającą się z około 20 000 wierszy kodu:

```
def huge_function(data_set)
  x_pca = PCA(n_components=2).fit_transform(X_train)
  clusters = clf.fit_predict(X_train)
  ax = plt.subplots(1, 2, figsize=(4))
  ay = plt.subplots(0, 2, figsize=(2))
  fig = plt.subplots(3, 4, figsize=(5))
  fig.subplots_adjust(top=0.85)

  predicted = svc_model.predict(X_test)
  images_and_predictions = list(zip(images_test, predicted))
  for x in 0..xlimit
    ax[0].scatter(X_pca[x], X_pca[1], c=clusters)
    ax[0].set_title('Predicted Training Labels')
    ax[1].scatter(X_pca[x], X_pca[1], c=y_train)
    ax[1].set_title('Actual Training Labels')
    ax[2].scatter(X_pca[x], X_pca[1], c=clusters)
  end

  for y in 0..ylimit
    ay[0].scatter(X_pca[y], X_pca[1], c=clusters)
    ay[0].set_title('Predicted Training Labels')
```

⁴ Robert C. Martin, *Czysty kod. Podręcznik dobrego programisty*, Helion, 2010.

```

    ay[1].scatter(X_pca[y], X_pca[1], c=y_train)
    ay[1].set_title('Actual Training Labels')
    ay[2].scatter(X_pca[y], X_pca[1], c=clusters)
end

#
# Pozostałe 20 000 wierszy kodu.
#
end

```

Od razu wiesz, że ten kod śmierdzi i znacznie lepszym rozwiązaniem będzie przeprowadzenie jego refaktoryzacji na postać mniejszych, oddzielnych funkcji, z których każda wykonuje jedno zadanie.

```

def calculate_images_and_predictions(images_test, predicted)
  x_pca = PCA(n_components=2).fit_transform(X_train)
  clusters = clf.fit_predict(X_train)
  ax = plt.subplots(1, 2, figsize=(4))
  fig = plt.subplots(3, 4, figsize=(5))
  fig.subplots_adjust(top=0.85)

  predicted = svc_model.predict(X_test)
  return list(zip(images_test, predicted))
end

def process_x_coords(ax)
  for x in 0..xlimit
    ax[0].scatter(X_pca[x], X_pca[1], c=clusters)
    ax[0].set_title('Predicted Training Labels')
    ax[1].scatter(X_pca[x], X_pca[1], c=y_train)
    ax[1].set_title('Actual Training Labels')
    ax[2].scatter(X_pca[x], X_pca[1], c=clusters)
  end

  return ax
end

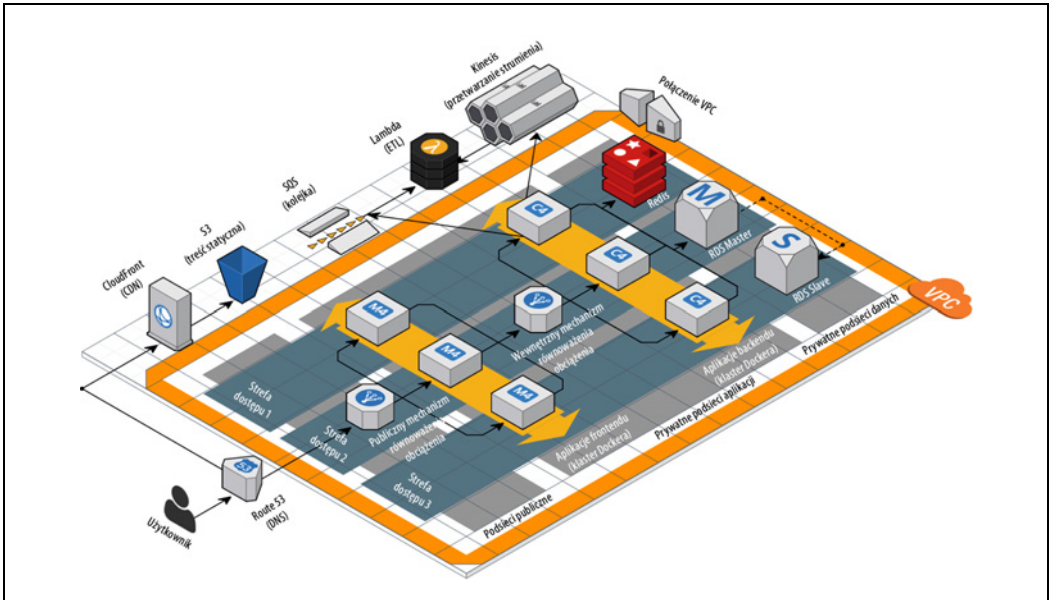
def process_y_coords(ax)
  for y in 0..ylimit
    ay[0].scatter(X_pca[y], X_pca[1], c=clusters)
    ay[0].set_title('Predicted Training Labels')
    ay[1].scatter(X_pca[y], X_pca[1], c=y_train)
    ay[1].set_title('Actual Training Labels')
    ay[2].scatter(X_pca[y], X_pca[1], c=clusters)
  end

  return ay
end

#
# Wiele mniejszych funkcji.
#

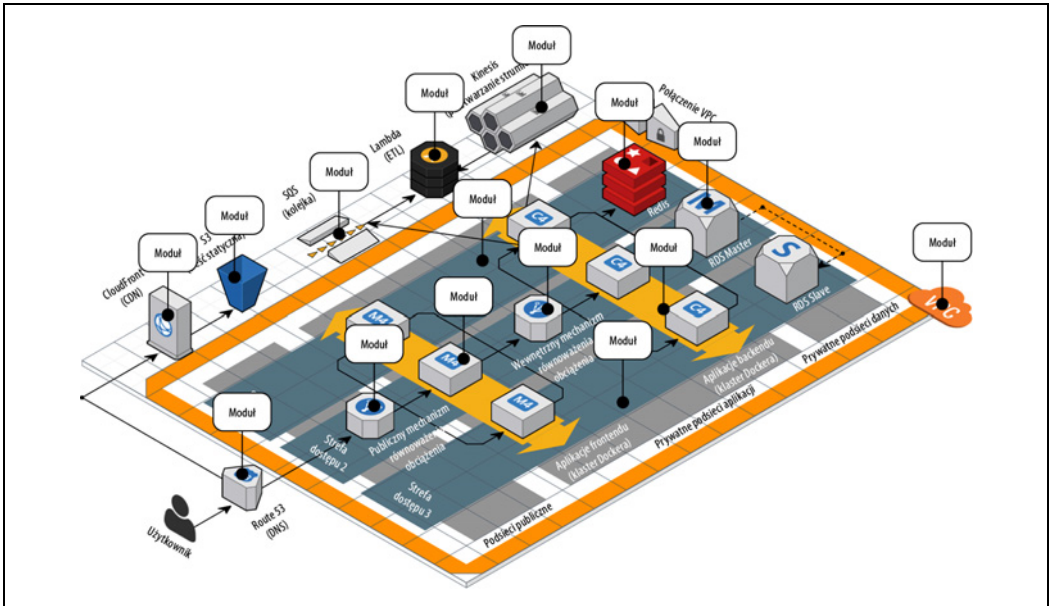
```

Tę samą strategię należy zastosować w Terraform. Wyobraź sobie, że masz do czynienia z architekturą pokazaną na rysunku 6.1.



Rysunek 6.1. Przykład względnie skomplikowanej architektury AWS

Jeżeli ta architektura zostanie zdefiniowana w postaci pojedynczego, ogromnego modułu Terraform, który składa się z 20 000 wierszy, od razu powinniśmy wiedzieć, że to rozwiązanie jest niedobre. Lepsze podejście polega na refaktoryzacji tego kodu na większą liczbę mniejszych, oddzielnych modułów, z których każdy wykonuje jedno zadanie, jak pokazałem na rysunku 6.2.



Rysunek 6.2. Przykład względnie skomplikowanej architektury AWS podzielonej na wiele mniejszych modułów

Zbudowany dotychczas moduł `webserver-cluster` zaczyna się robić zbyt duży i obsługuje trzy niepowiązane ze sobą zadania:

Automatycznie skalowana grupa (ASG)

Moduł `webserver-cluster` stosuje ASG, aby można było przeprowadzać wdrożenia bez przestoju.

Mechanizm równoważenia obciążenia (ALB)

Moduł `webserver-cluster` wdraża mechanizm równoważenia obciążenia.

Aplikacja typu Witaj, świecie

Moduł `webserver-cluster` wdraża również prostą aplikację typu *Witaj, świecie*.

Przeprowadzimy teraz refaktoryzację kodu na trzy mniejsze moduły:

`modules/cluster/asg-rolling-deploy`

Ogólny, wielokrotnego użycia i oddzielny moduł przeznaczony do wdrożenia grupy ASG pozwalającej na przeprowadzanie wdrożenia bez przestoju.

`modules/networking/alb`

Ogólny, wielokrotnego użycia i oddzielny moduł przeznaczony do wdrożenia mechanizmu równoważenia obciążenia.

`modules/services/hello-world-app`

Moduł specjalnie przeznaczony do wdrożenia aplikacji typu *Witaj, świecie*.

Zanim przystąpisz do pracy, upewnij się o wykonaniu poleceń `terraform destroy` we wszystkich wdrożeniach pozostałych z poprzednich rozdziałów. Teraz możesz już przystąpić do wdrażania oddzielnych modułów. Utwórz nowy katalog `modules/cluster/asg-rolling-deploy` i przenieś (np. wytnij i wklej) następujące zasoby z pliku `module/services/webserver-cluster/main.tf` do pliku `modules/cluster/asg-rolling-deploy/main.tf`:

- `aws_launch_configuration`,
- `aws_autoscaling_group`,
- `aws_autoscaling_schedule` (wszystko),
- `aws_security_group` (dla egzemplarzy, ale nie dla ALB),
- `aws_security_group_rule` (reguły dotyczące egzemplarzy, ale nie ALB),
- `aws_cloudwatch_metric_alarm` (wszystko).

Następnie przenieś wymienione tutaj zmienne z pliku `module/services/webserver-cluster/variables.tf` do pliku `modules/cluster/asg-rolling-deploy/variables.tf`:

- `cluster_name`,
- `ami`,
- `instance_type`,
- `min_size`,
- `max_size`,

- `enable_autoscaling`,
- `custom_tags`,
- `server_port`.

Przechodzimy teraz do modułu ALB. Utwórz nowy katalog `modules/networking/alb` i przenieś do niego wymienione tutaj zasoby — z pliku `module/services/webserver-cluster/main.tf` do pliku `modules/networking/alb/main.tf`:

- `aws_lb`,
- `aws_lb_listener`,
- `aws_security_group` (te dotyczące ALB, ale nie dla egzemplarzy),
- `aws_security_group_rule` (reguły dotyczące ALB, ale nie egzemplarzy).

Utwórz plik `modules/networking/alb/variables.tf` i zdefiniuj w nim pojedynczą zmienną.

```
variable "alb_name" {
  description = "Nazwa do użycia w tym module ALB"
  type       = string
}
```

Tę nową zmienną wykorzystaj jak argument `name` zasobu `aws_lb`.

```
resource "aws_lb" "example" {
  name           = var.alb_name
  load_balancer_type = "application"
  subnets       = data.aws_subnet_ids.default.ids
  security_groups = [aws_security_group.alb.id]
}
```

Nową zmienną wykorzystaj również jak argument `name` zasobu `aws_security_group`.

```
resource "aws_security_group" "alb" {
  name = var.alb_name
}
```

To jest dość sporo kodu do przeniesienia, więc możesz skorzystać z kodu, który przygotowałem dla tego rozdziału — znajdziesz go w repozytorium na stronie <https://github.com/brikis98/terraform-up-and-running-code>.

Moduły łączone z innymi

W tym momencie masz dwa małe moduły — odpowiedzialne za wdrożenie grupy ASG i mechanizmu równoważenia obciążenia — dobrze wykonujące swoje zadania. W jaki sposób można je zmusić do współpracy? Jak zbudować moduł, który będzie mógł być wielokrotnego użycia i być łączony z innymi? To pytanie nie jest unikatowe dla Terraform i część programistów zastanawia się nad tym od dekad. Pozwolę sobie w tym miejscu zacytować Douga McIlroya⁵, twórcę potoków UNIX i wielu innych narzędzi tego systemu, `m.in.`, `diff`, `sort`, `join` i `tr`:

⁵ Peter H. Salus, *A Quarter-Century of Unix*, Addison-Wesley Professional, New York 1994.

To jest filozofia systemu UNIX: twórz programy dobrze wykonujące jedno zadanie. Programy buduj w taki sposób, aby ze sobą współdziałały.

— Doug McIlroy

Jednym z rozwiązań jest zastosowanie *kompozycji funkcji*, co oznacza możliwość przekazania danych wyjściowych jednej funkcji jako danych wejściowych innej funkcji. Dla przykładu przyjmuję założenie o istnieniu przedstawionych tutaj małych funkcji w języku Ruby:

```
# Prosta funkcja przeprowadzająca operację dodawania.
def add(x, y)
  return x + y
end

# Prosta funkcja przeprowadzająca operację odejmowania.
def sub(x, y)
  return x - y
end

# Prosta funkcja przeprowadzająca operację mnożenia.
def multiply(x, y)
  return x * y
end
```

Kompozycję funkcji można wykorzystać do ich połączenia przez pobranie danych wyjściowych funkcji `add()` i `sub()`, a następnie przekazania ich jako danych wyjściowych funkcji `multiply()`.

```
# Funkcja złożona składająca się z kilku prostszych funkcji.
def do_calculation(x, y)
  return multiply(add(x, y), sub(x, y))
end
```

Jednym z podstawowych sposobów pozwalających na łączenie funkcji ze sobą jest minimalizacja *efektów ubocznych*: o ile możliwe jest unikanie stanu pochodzącego ze świata zewnętrznego i zamiast tego przekazywanie informacji o stanie za pomocą parametrów danych wejściowych oraz unikanie przekazywania na zewnątrz informacji o stanie i zamiast tego zwracanie wyniku obliczeń za pomocą parametrów danych wyjściowych. Minimalizacja efektów ubocznych to jedno z założeń programowania funkcyjnego, ponieważ wtedy łatwiej uzasadnić potrzebę tworzenia danego fragmentu kodu, łatwiej można go przetestować, a także ponownie wykorzystać. Wielokrotne użycie kodu jest szczególnie kuszące, ponieważ kompozycja funkcji pozwala na stopniowe tworzenie coraz bardziej skomplikowanych funkcji przez łączenie tych prostszych.

Wprawdzie nie można uniknąć efektów ubocznych podczas pracy z kodem infrastruktury, ale wciąż warto stosować te same podstawowe zasady przy tworzeniu modułów Terraform: przekazywanie wszystkiego za pomocą zmiennych danych wejściowych, zwracanie wszystkiego za pomocą zmiennych danych wyjściowych oraz tworzenie skomplikowanych modułów poprzez łączenie prostszych modułów.

Otwórz plik `modules/cluster/arg-rolling-deploy/variables.tf` i umieść w nim cztery nowe zmienne danych wejściowych.

```
variable "subnet_ids" {
  description = "Identyfikatory podsieci do wdrożenia"
  type        = list(string)
```

```

}

variable "target_group_arns" {
  description = "Grupy docelowe wartości ARN w mechanizmie ELB, w którym zostaną
  ↪zarejestrowane egzemplarze"
  type        = list(string)
  default     = []
}

variable "health_check_type" {
  description = "Typ przeprowadzanego sprawdzenia stanu. To musi być EC2 lub ELB"
  type        = string
  default     = "EC2"
}

variable "user_data" {
  description = "Skrypt danych użytkownika przeznaczony do wykonania w każdym egzemplarzu
  ↪podczas jego uruchamiania"
  type        = string
  default     = ""
}

```

Pierwsza zmienna, `subnet_ids`, powoduje przekierowanie modułu `asg-rolling-deploy` do podsieci, w których będzie wdrażana. Wprawdzie zostało na stałe zdefiniowane wdrożenie modułu `webserver-cluster` w domyślnej sieci VPC i podsieciach, ale przez udostępnienie zmiennej `subnet_ids` można ten moduł wykorzystać także z innymi VPC i podsieciami. Dwie kolejne zmienne, `target_group_arns` i `health_check_type`, odpowiadają za konfigurację integracji ASG z mechanizmem równoważenia obciążenia. Wprawdzie moduł `webserver-cluster` ma wbudowany mechanizm równoważenia obciążenia (ALB), ale moduł `asg-rolling-deploy` ma być ogólny, więc udostępnienie ustawień mechanizmu równoważenia obciążenia w postaci zmiennych danych wejściowych pozwala na wykorzystanie ASG w wielu sytuacjach, jak brak mechanizmu równoważenia obciążenia, tylko jeden komponent ALB, wiele komponentów NLB itd.

Te trzy zmienne danych wejściowych przekaż do zasobu `aws_autoscaling_group` zdefiniowanego w pliku `modules/cluster/asg-rolling-deploy/main.tf`, zastąpisz tym samym poprzednie, zdefiniowane na stałe ustawienia odwołujące się do zasobów (ALB) i źródeł danych (np. `aws_subnet_ids`), które nie zostały skopiowane do modułu `asg-rolling-deploy`.

```

resource "aws_autoscaling_group" "example" {
  # Jawna zależność od nazwy konfiguracji startowej, więc każde jej
  # zastąpienie oznacza również konieczność zastąpienia tej grupy ASG.
  name = "${var.cluster_name}-${aws_launch_configuration.example.name}"

  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier   = var.subnet_ids

  # Konfiguracja integracji z mechanizmem równoważenia obciążenia.
  target_group_arns    = var.target_group_arns
  health_check_type    = var.health_check_type

  min_size = var.min_size
  max_size = var.max_size

  # Zanim wdrożenie ASG zostanie uznane za zakończone, należy
  # poczekać na przeprowadzenie podanej tutaj liczby sprawdzeń.

```

```

min_elb_capacity = var.min_size

# (...)
}

```

Czwarta zmienna jest przekazywana w skrypcie danych użytkownika. Podczas gdy moduł `webserver-cluster` miał na stałe zdefiniowany skrypt danych użytkownika i mógł przeprowadzić wdrożenie tylko aplikacji wyświetlającej komunikat typu *Witaj, świecie*, to jeśli skrypt danych użytkownika jest pobierany za pomocą zmiennej danych wejściowych, moduł `asg-rolling-deploy` można wykorzystać do wdrożenia dowolnej aplikacji w grupie ASG. Przekaż zmienną `user_data` do zasobu `aws_launch_configuration` (i tym samym zastąp odwołanie do źródła danych `template_file` nieskopiowanego w module `asg-rolling-deploy`).

```

resource "aws_launch_configuration" "example" {
  image_id      = var.ami
  instance_type = var.instance_type
  security_groups = [aws_security_group.instance.id]
  user_data     = var.user_data

  # Wymagane podczas używania konfiguracji startowej wraz z automatycznie skalowaną grupą.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}

```

Do pliku `modules/cluster/asg-rolling-deploy/outputs.tf` dodaj jeszcze dwie użyteczne zmienne danych wejściowych:

```

output "asg_name" {
  value     = aws_autoscaling_group.example.name
  description = "Nazwa automatycznie skalowanej grupy"
}

output "instance_security_group_id" {
  value     = aws_security_group.instance.id
  description = "Identyfikator grupy bezpieczeństwa egzemplarza EC2"
}

```

Wygenerowanie danych spowoduje, że moduł `asg-rolling-deploy` będzie charakteryzował się jeszcze większymi możliwościami w zakresie wielokrotnego użycia, ponieważ jego użytkownicy będą mogli dodawać nowe sposoby zachowania, np. nowej reguły do grupy bezpieczeństwa.

Z podobnych powodów należy dodać kilka zmiennych danych wyjściowych do pliku `modules/networking/alb/outputs.tf`:

```

output "alb_dns_name" {
  value     = aws_lb.example.dns_name
  description = "Nazwa domeny mechanizmu równoważenia obciążenia"
}

output "alb_http_listener_arn" {
  value     = aws_lb_listener.http.arn
  description = "Wartość ARN komponentu nasłuchującego żądań HTTP"
}

output "alb_security_group_id" {

```

```

    value      = aws_security_group.alb.id
    description = "Identyfikator grupy bezpieczeństwa mechanizmu równoważenia obciążenia"
}

```

Wkrótce będziesz mieć możliwość wykorzystania tych zmiennych.

Ostatnim krokiem jest konwersja modułu `webserver-cluster` na moduł `hello-world-app` przeznaczony do wdrożenia aplikacji wyświetlającej komunikat typu *Witaj, świecie* za pomocą modułów `asg-rolling-deploy` i `alb`. W tym celu zacznij od zmiany nazwy katalogu `module/services/webserver-cluster` na `module/services/hello-world-app`. Po wprowadzeniu przedstawionych wcześniej zmian w pliku `module/services/hello-world-app/main.tf` powinny pozostać jedynie wymienione tutaj zasoby:

- `template_file` (dla danych użytkownika),
- `aws_lb_target_group`,
- `aws_lb_listener_rule`,
- `terraform_remote_state` (dla baz danych),
- `aws_vpc`,
- `aws_subnet_ids`.

Do pliku `modules/services/hello-world-app/variables.tf` dodaj przedstawioną tutaj zmienną:

```

variable "environment" {
  description = "Nazwa środowiska, w którym odbędzie się wdrożenie."
  type        = string
}

```

Kolejnym krokiem jest dodanie utworzonego wcześniej modułu `asg-rolling-deploy` do modułu `hello-world-app` w celu wdrożenia grupy ASG.

```

module "asg" {
  source = "../../cluster/asg-rolling-deploy"

  cluster_name = "hello-world-${var.environment}"
  ami          = var.ami
  user_data    = data.template_file.user_data.rendered
  instance_type = var.instance_type

  min_size      = var.min_size
  max_size      = var.max_size
  enable_autoscaling = var.enable_autoscaling

  subnet_ids      = data.aws_subnet_ids.default.ids
  target_group_arns = [aws_lb_target_group.asg.arn]
  health_check_type = "ELB"

  custom_tags = var.custom_tags
}

```

Do `hello-world-app` dodaj jeszcze moduł `alb`, również utworzony wcześniej, w celu wdrożenia mechanizmu równoważenia obciążenia.

```

module "alb" {
  source = "../../networking/alb"
}

```

```

alb_name = "hello-world-${var.environment}"
subnet_ids = data.aws_subnet_ids.default.ids
}

```

Zwróć uwagę na użycie zmiennej danych wejściowych `environment` jako sposobu na wymuszenie stosowania pewnej konwencji nazw, aby wszystkie zasoby miały przestrzeń nazw oparte na nazwie środowiska (np. `hello-world-stage`, `hello-world-prod`). Ten kod powoduje przypisanie odpowiednich wartości dodanym wcześniej nowym zmiennym `subnet_ids`, `target_group_arns`, `health_check_type` i `user_data`.

Kolejnym krokiem jest konfiguracja komponentu ALB grupy docelowej i reguły komponentu nasłuchującego dla tej aplikacji. Uaktualnij zasób `aws_lb_target_group` w pliku `modules/services/hello-world-app/main.tf` w celu użycia wartości `environment` w atrybucie `name`.

```

resource "aws_lb_target_group" "asg" {
  name      = "hello-world-${var.environment}"
  port      = var.server_port
  protocol  = "HTTP"
  vpc_id    = data.aws_vpc.default.id

  health_check {
    path            = "/"
    protocol        = "HTTP"
    matcher         = "200"
    interval        = 15
    timeout         = 3
    healthy_threshold = 2
    unhealthy_threshold = 2
  }
}

```

Teraz uaktualnij parametr `listener_arn` zasobu `aws_lb_listener_rule`, aby prowadził do danych wyjściowych `alb_http_listener_arn` modułu mechanizmu równoważenia obciążenia.

```

resource "aws_lb_listener_rule" "asg" {
  listener_arn = module.alb.alb_http_listener_arn
  priority     = 100

  condition {
    field = "path-pattern"
    values = ["*"]
  }

  action {
    type = "forward"
    target_group_arn = aws_lb_target_group.asg.arn
  }
}

```

Pozostało już tylko przekazywanie ważnych danych wyjściowych z modułów `asg-rolling-deploy` i `alb` jako danych wyjściowych modułu `hello-world`.

```

output "alb_dns_name" {
  value      = module.alb.alb_dns_name
  description = "Nazwa domeny mechanizmu równoważenia obciążenia"
}

```

```

output "asg_name" {
  value       = module.asg.asg_name
  description = "Nazwa automatycznie skalowanej grupy"
}

output "instance_security_group_id" {
  value       = module.asg.instance_security_group_id
  description = "Identyfikator grupy bezpieczeństwa egzemplarza EC2"
}

```

To jest przykład kompozycji funkcji w akcji: tworzysz bardziej skomplikowane rozwiązanie (aplikacja wyświetlająca komunikat typu *Witaj, świecie*) na podstawie prostszych komponentów (moduły ASG i ALB). Dość często spotykany wzorec podczas pracy z Terraform to wykorzystanie przynajmniej dwóch typów modułów:

Moduły generyczne

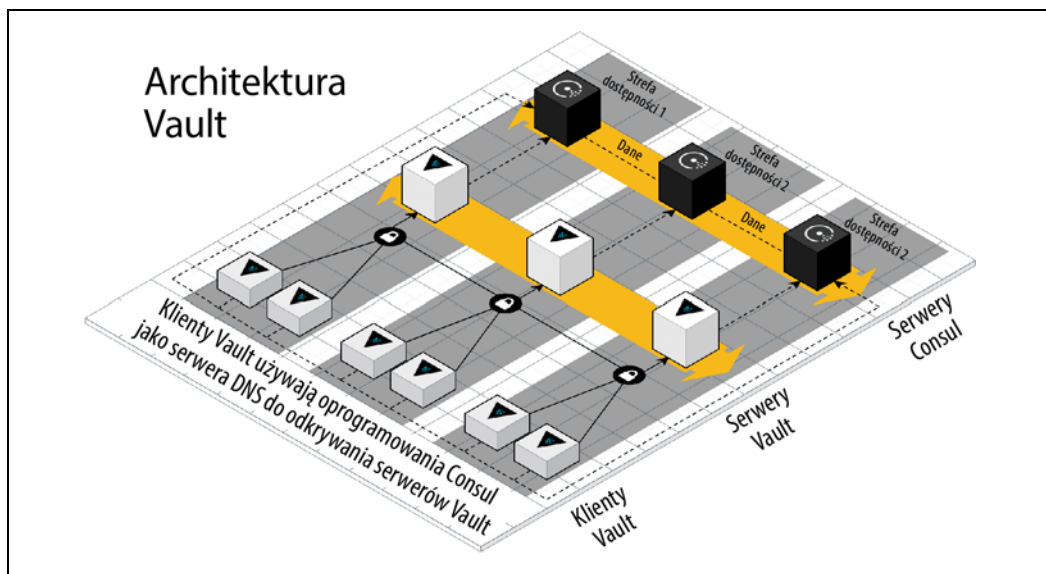
Moduły takie jak `asg-rolling-deploy` i `alb` są podstawowymi elementami konstrukcyjnymi kodu i znajdują wielokrotne zastosowanie w różnych sytuacjach. Miałeś już okazję zobaczyć ich zastosowanie do wdrożenia aplikacji wyświetlającej komunikat typu *Witaj, świecie*. Dokładnie te same moduły można wykorzystać we wdrożeniu np. grupy ASG przeznaczonej do uruchomienia klastra Kafka lub całkowicie samodzielnego komponentu ALB odpowiedzialnego za rozkład obciążenia między wiele działających aplikacji (przygotowanie tylko jednego mechanizmu równoważenia obciążenia dla wszystkim aplikacji jest tańsze niż po jednym komponencie ALB dla każdej aplikacji).

Moduły przeznaczone dla danej sytuacji

Moduły takie jak `hello-world-app` łączą wiele modułów generycznych w celu przygotowania do obsługi jednej konkretnej sytuacji, np. wdrożenia aplikacji wyświetlającej komunikat typu *Witaj, świecie*.

W rzeczywistości może wystąpić konieczność dalszego podziału modułów, aby lepiej przystosować je do łączenia i wielokrotnego używania. Przykładowo zestaw wielokrotnego użycia modułów typu open source można wykorzystać do uruchomienia HashiCorp w repozytorium `terraform-aws-consul` (<https://github.com/hashicorp/terraform-aws-consul>). Consul to rozprowadzany jako open source rozproszony magazyn typu klucz-wartość, który czasami musi nasłuchiwać żądań sieciowych na ogromnej liczbie różnych portów (serwer RPC, CLI RPC, Serf WAN, HTTP API, DNS itd.), co oznacza zdefiniowanie około 20 reguł w grupie bezpieczeństwa. W repozytorium `terraform-aws-consul` te reguły są zdefiniowane w oddzielnym module `consul-security-group-rules` (<https://github.com/hashicorp/terraform-aws-consul/tree/master/modules/consul-security-group-rules>).

Aby się dowiedzieć, dlaczego tak jest, trzeba poznać sposób wdrożenia aplikacji Consul. Jednym z często stosowanych rozwiązań jest wdrożenie jej jako magazynu danych dla HashiCorp Vault, czyli dostępnego jako oprogramowanie typu open source rozproszonego magazynu dla danych wrażliwych, który można wykorzystać do bezpiecznego przechowywania haseł, kluczy API, certyfikatów TLS itd. Zestaw modułów typu open source wielokrotnego użycia przeznaczonych do uruchomienia Vault znajdziesz w repozytorium `terraform-aws-vault` (<https://github.com/hashicorp/terraform-aws-vault/>). Zawiera ono również diagram pokazany na rysunku 6.3, przedstawiający typową architekturę produkcyjną dla Vault.



Rysunek 6.3. Architektura HashiCorp Vault i Consul

W środowisku produkcyjnym Vault zwykle uruchamia się w grupie ASG składającej się z od trzech do pięciu serwerów (wdrożonych za pomocą modułu `vault-cluster`, <https://github.com/hashicorp/terraform-aws-vault/tree/master/modules/vault-cluster>) oraz oprogramowania Consul w oddzielnej grupie ASG składającej się z od trzech do pięciu serwerów (wdrożonych za pomocą modułu `consul-cluster`, <https://github.com/hashicorp/terraform-aws-consul/tree/master/modules/consul-cluster>), więc każdy z nich może być oddzielnie skalowany i zabezpieczany. Jednak w środowisku roboczym uruchamianie takiej liczby serwerów jest zbędne i można zaoszczędzić nieco pieniędzy dzięki uruchomieniu Vault i Consul w tej samej grupie ASG o wielkości prawdopodobnie jednego serwera i wdrożonej za pomocą modułu `vault-cluster`. Jeżeli wszystkie reguły grupy bezpieczeństwa zostały zdefiniowane w module `consul-cluster`, nie możesz ich wykorzystać (bez wcześniejszego skopiowania i wklejenia), o ile do wdrożenia Consul został użyty moduł `vault-cluster`. Skoro reguły zostały zdefiniowane w oddzielnym modelu `consul-security-group-rules`, można je dołączyć do `vault-cluster` lub praktycznie do innego dowolnego typu klastra.

Taki podział okazuje się przesadą w przypadku prostej aplikacji wyświetlającej komunikat typu *Witaj, świecie*. Natomiast dla skomplikowanej, rzeczywistej infrastruktury wydzielenie oddzielnych modułów zawierających reguły grupy bezpieczeństwa, polityki IAM i inne zadania ma ważne znaczenie w zapewnieniu możliwości obsługi różnych wzorców wdrażania. Ten wzorec został użyty nie tylko w przypadku Consul i Vault, ale również wraz ze stosem ELK (uruchamianie Elasticsearch, Logstash i Kibana w oddzielnych klastrach w środowisku produkcyjnym, ale w jednym klastrze w środowisku programistycznym), na platformie Confluent (uruchamianie Kafka, ZooKeeper, REST Proxy i Schema Registry w oddzielnych klastrach w środowisku produkcyjnym, ale w jednym klastrze w środowisku programistycznym), na stosie TICK (uruchamianie Telegraf, InfluxDB, Chronograf i Kapacitor w oddzielnych klastrach w środowisku produkcyjnym, ale w jednym klastrze w środowisku programistycznym) itd.

Moduły możliwe do testowania

Na tym etapie utworzyłeś całkiem sporo kodu w postaci trzech modułów: `asg-rolling-deploy`, `alb` i `hello-world-app`. Następnym krokiem jest sprawdzenie, czy ten kod faktycznie działa.

Utworzone tutaj moduły nie są modułami głównymi przeznaczonymi do bezpośredniego wdrożenia. Aby je wdrożyć, konieczne jest przygotowanie kodu Terraform odpowiedzialnego za przekazywanie niezbędnych argumentów, skonfigurowanie dostawcy i backendu itd. Doskonałym sposobem na zrobienie tego jest utworzenie katalogu o nazwie *example* (z ang. *przykład*), który zgodnie ze swoją nazwą będzie pokazywał sposób użycia modułów. Spróbujmy zastosować właśnie takie rozwiązanie.

Utwórz plik `examples/asg/main.tf` wraz z przedstawionym tutaj kodem:

```
provider "aws" {
  region = "us-east-2"
}

module "asg" {
  source = "../../modules/cluster/asg-rolling-deploy"

  cluster_name = var.cluster_name
  ami          = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  min_size      = 1
  max_size      = 1
  enable_autoscaling = false

  subnet_ids = data.aws_subnet_ids.default.ids
}

data "aws_vpc" "default" {
  default = true
}

data "aws_subnet_ids" "default" {
  vpc_id = data.aws_vpc.default.id
}
```

Pewna część kodu używa modułu `asg-rolling-deploy` w celu wdrożenia grupy ASG o wielkości 1. Spróbuj wydać polecenia `terraform init` i `terraform apply`, a następnie sprawdź, czy zostały wykonane bezbłędnie i faktycznie doprowadziły do utworzenia grupy ASG. Kolejnym krokiem jest dodanie pliku *README.md* wraz z informacjami — i nagle ten mały przykład zyskuje całkiem potężne możliwości. Za pomocą zaledwie kilku plików i wierszy kodu można wykonać wiele zadań:

Ręczne przeprowadzanie testów

Podany fragment kodu można wykorzystać podczas pracy nad modułem `asg-rolling-deploy` w celu wielokrotnego wdrażania i przeprowadzać ręczne wdrażanie za pomocą poleceń `terraform apply` i `terraform destroy`, aby sprawdzić, czy wszystko działa zgodnie z oczekiwaniami.

Zautomatyzowane przeprowadzanie testów

Jak zobaczysz w rozdziale 7., ten przykładowy fragment kodu przedstawia również sposób tworzenia zautomatyzowanych testów dla modułów. Zalecam, aby testy były umieszczane w katalogu *test*.

Wykonywalna dokumentacja

Jeżeli ten przykład (wraz z plikiem *README.me*) przekażesz do systemu kontroli wersji, pozostali członkowie zespołu będą mogli go znaleźć, wykorzystać do zrozumienia sposobu działania modułu i wypróbować go bez konieczności tworzenia jakiegokolwiek kodu. To jest sposób na jednocześnie przekazanie wiedzy pozostałym członkom zespołu i, jeśli dodałeś testy zautomatyzowane, na zagwarantowanie, że te informacje będą prawidłowe.

Każdemu modułowi Terraform w katalogu *modules* powinien odpowiadać przykład umieszczony w katalogu *examples*. Z kolei każdy przykład w katalogu *examples* powinien mieć odpowiedni test umieszczony w katalogu *test*. W rzeczywistości możesz mieć wiele przykładów (i tym samym testów) dla poszczególnych modułów, a każdy z przykładów będzie przedstawiał inne konfiguracje i permutacje sposobu użycia modułu. Być może dodasz kolejny przykład dla modułu *asg-rolling-deploy* pokazujący, jak można go używać wraz z polityką automatycznego skalowania, jak połączyć go z mechanizmem równoważenia obciążenia, jak zdefiniować niestandardowe znaczniki itd.

Ogólna struktura katalogu dla typowego repozytorium *modules* będzie przedstawiała się podobnie do tej:

```
modules
  examples
    alb
    asg-rolling-deploy
      one-instance
      auto-scaling
      with-load-balancer
      custom-tags
    hello-world-app
    mysql
  modules
    alb
    asg-rolling-deploy
    hello-world-app
    mysql
  test
    alb
    asg-rolling-deploy
    hello-world-app
    mysql
```

Jako ćwiczenie do wykonania pozostawiam Ci zmianę utworzonego wcześniej kodu RDS na moduł MySQL oraz dodanie wielu przykładów dla modułów *alb*, *asg-rolling-deploy*, *mysql* i *hello-world-app*.

Doskonałą praktyką podczas opracowywania nowego modułu jest *najpierw* utworzenie przykładowego fragmentu kodu, a dopiero później przystąpienie do tworzenia kodu modułu. Jeżeli rozpoczniesz od implementacji, bardzo łatwo możesz ugrząźć w szczegółach implementacji — zanim wprowadzisz niezbędne modyfikacje i powrócisz do API, otrzymasz moduł nieintuicyjny i trudny w użyciu.

Z drugiej strony, jeśli pracę rozpocznesz od kodu przykładu, możesz zastanowić się nad idealnym sposobem używania modułu, a skutkiem będzie przejrzyste API modułu i możliwość przystąpienia do pracy nad implementacją. Skoro kod przykładu to podstawowy sposób na przetestowanie modułu, mamy tutaj czystą postać podejścia TDD (ang. *test-driven development*), które dokładnie przedstawię w rozdziale 7.

Istnieje jeszcze inna praktyka, której zalety docenisz, gdy zaczniesz regularnie testować tworzone moduły — *przypisanie wersji*. Wszystkim modułom Terraform powinieneś przypisywać określoną wersję za pomocą argumentu `required_version`. Absolutnym minimum powinno być podanie wymaganej wersji głównej Terraform.

```
terraform {  
  # Wymagana dowolna wersja 0.12.x Terraform.  
  required_version = ">= 0.12, < 0.13"  
}
```

Zgodnie z przedstawionym kodem w module dozwolone jest użycie tylko wersji 0.12.x Terraform, ale już nie 0.11.x lub 0.13.x. To ma znaczenie krytyczne, ponieważ każda wersja główna jest niezgodna wstecz. Dlatego też uaktualnienie z wersji 0.11.x do 0.12.x wymaga zmian w kodzie, więc nie chcesz, aby taka zmiana została przeprowadzana przypadkowo. Dzięki dodaniu atrybutu `required_version`, jeśli spróbujesz wydać polecenie `terraform apply` w innej wersji Terraform, natychmiast otrzymasz komunikat błędu.

```
$ terraform apply  
Error: Unsupported Terraform Core version
```

```
This configuration does not support Terraform version 0.11.11. To proceed,  
either choose another supported Terraform version or update the root module's  
version constraint. Version constraints are normally set for good reason, so  
updating the constraint may lead to other errors or unexpected behavior.
```

W przypadku kodu o jakości produkcyjnej zaleca się jeszcze ściślej deklarowanie wersji:

```
terraform {  
  # Wymagana jest dokładnie wersja 0.12.0 Terraform.  
  required_version = "= 0.12.0"  
}
```

Nawet wersja poprawki (np. przejście z 0.12.0 do 0.12.1) może spowodować problemy. Od czasu do czasu poprawki wprowadzają błędy, bywa również, że są niezgodne wstecz (obecnie coraz rzadziej tak się dzieje). Jednak najważniejszym powodem jest to, że jeśli plik informacji o stanie Terraform zostanie zapisany za pomocą nowszej wersji Terraform, nie będzie można go używać w żadnej z wcześniejszych wersji. Dla przykładu przyjmuję założenie o wdrożeniu całego kodu za pomocą Terraform 0.12.0. Pewnego dnia pojawia się programista, który zainstalował wersję 0.12.1 i wydaje polecenie `terraform apply` dla kilku z Twoich modułów. Teraz pliki informacji o stanie stają się niezgodne z wersją 0.12.0, więc jesteś zmuszony do uaktualnienia wszystkich komputerów programistycznych i serwerów ciągłej integracji do wersji 0.12.1!

Ta sytuacja poprawi się po wydaniu Terraform w wersji 1.0.0, gdy zacznie być wymagane zachowanie wstecznej zgodności. Jednak zanim to nastąpi, zachęcam do *ściśłego* podawania wymaganej wersji Terraform. W ten sposób unikniesz przypadkowych uaktualnień. Zamiast tego możesz zdecydować,

kiedy staniesz się gotowy do uaktualnienia, i wtedy przeprowadzić aktualizację jednocześnie we wszystkich komputerach programistycznych i serwerach ciągłej integracji.

Zachęcam również do przypisywania wersji dostawcy.

```
provider "aws" {
  region = "us-east-2"

  # Zgoda na użycie dowolnej wersji 2.x dostawcy AWS.
  version = "~> 2.0"
}
```

Ten kod przypisuje kod dostawcy AWS do dowolnej wersji 2.x (zapis `~> 2.0` jest odpowiednikiem dla `>= 2.0`, `< 3.0`). Warto przypomnieć ponownie, że minimum jest podanie wersji głównej, aby uniknąć przypadkowych zmian niezapewniających wstecznej zgodności. To, czy trzeba będzie przypisać konkretną wersję, zależy od dostawcy. Przykładowo AWS oferuje dość częste uaktualnienia i jednocześnie zapewnia wsteczną zgodność, więc zwykle wystarczy przypisanie jedynie wersji głównej i umożliwienie automatycznego wprowadzania poprawek, aby uzyskać dostęp do nowszych funkcjonalności. Jednak poszczególni dostawcy różnią się od siebie. Dokładnie sprawdzaj, jak sobie radzą z zachowaniem wstecznej zgodności, i odpowiednio przypisuj numer wersji.

Moduły możliwe do wydania

Po utworzeniu i przetestowaniu modułów następnym krokiem jest ich wydanie. Jak mogłeś zobaczyć we wcześniejszej części książki, masz możliwość zastosowania tagów Git wraz z wersjonowaniem semantycznym, co pokazałem w kolejnym fragmencie kodu:

```
$ git tag -a "v0.0.5" -m "Utworzenie nowego modułu hello-world-app"
$ git push --follow-tags
```

Przykładowo w celu wdrożenia wersji v0.0.5 modułu hello-world-app w środowisku roboczym przedstawiony tutaj kod powinien być umieszczony w pliku `live/stage/services/hello-world-app/main.tf`:

```
provider "aws" {
  region = "us-east-2"

  # Zgoda na użycie dowolnej wersji 2.x dostawcy AWS.
  version = "~> 2.0"
}

module "hello_world_app" {
  # TODO: zastąp kolejny wiersz kodu adresem URL i wersją swojego modułu!
  source = git@github.com:foo/modules.git//services/hello-world-app?ref=v0.0.5

  server_text      = "New server text"
  environment      = "stage"
  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"

  instance_type    = "t2.micro"
  min_size         = 2
  max_size         = 2
  enable_autoscaling = false
}
```

Następnie przekaż nazwę domeny mechanizmu ALB jako dane wyjściowe w pliku *live/stage/services/hello-world-app/outputs.tf*:

```
output "alb_dns_name" {
  value      = module.hello_world_app.alb_dns_name
  description = "Nazwa domeny mechanizmu równoważenia obciążenia"
}
```

Teraz można już wdrożyć wersjonowany moduł przez wydanie poleceń `terraform init` i `terraform apply`:

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 13 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

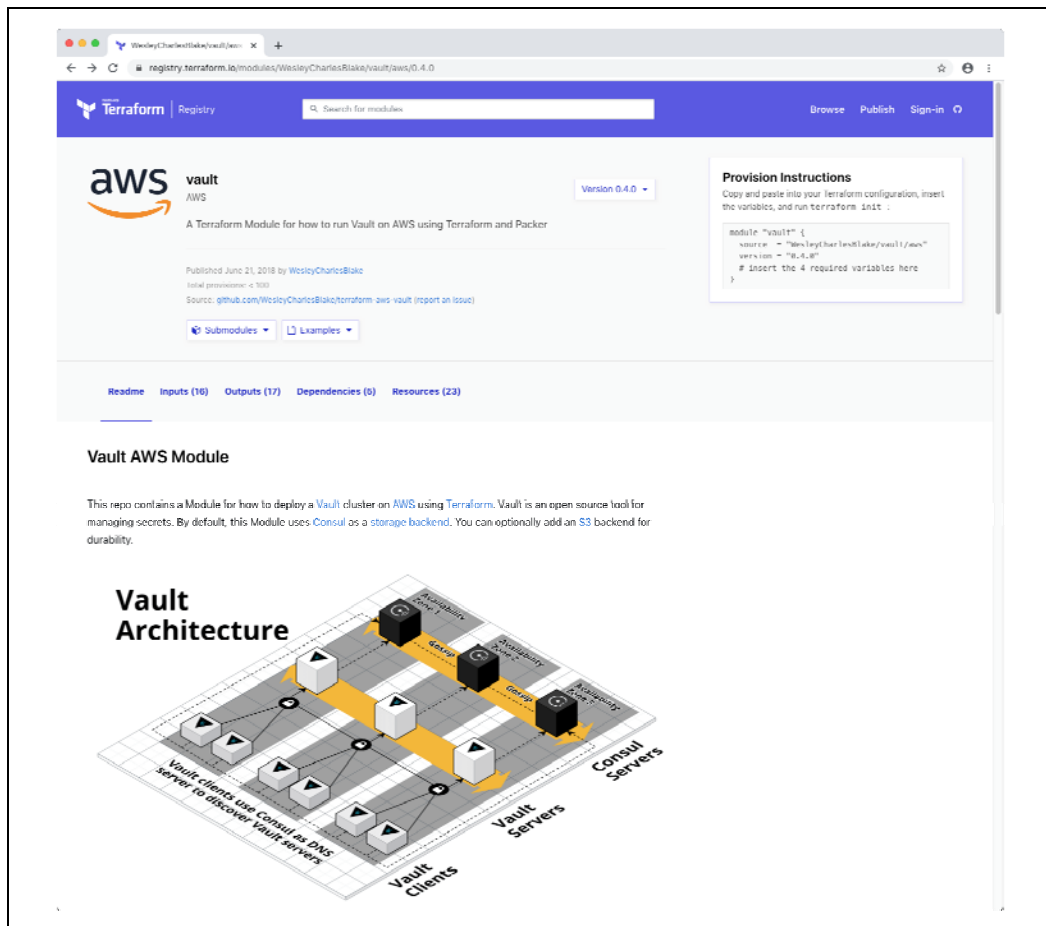
Jeżeli to rozwiązanie działa, dokładnie tę samą wersję — więc dokładnie ten sam kod — można wdrożyć także w innych środowiskach, w tym w produkcyjnym. Jeżeli kiedykolwiek napotkasz problem, wersjonowanie pozwala na wycofanie wdrożenia i powrót do jego starszej wersji.

Innym podejściem podczas wydawania modułów jest ich publikowanie we wspomnianym już wcześniej w książce repozytorium Terraform Registry. W tym dostępnym pod adresem <https://registry.terraform.io/> repozytorium znajdziesz setki wielokrotnego użycia modułów — opracowane przez społeczność i udostępnione jako oprogramowanie typu open source. To są moduły dla AWS, Google Cloud, Azure i wielu innych dostawców chmury. Opublikowanie modułu w oficjalnym repozytorium Terraform Registry wymaga spełnienia kilku wymagań⁶:

- Moduł musi znajdować się w publicznym repozytorium GitHub.
- Nazwa repozytorium musi mieć postać `terraform-<DOSTAWCA>-<NAZWA>`, gdzie `DOSTAWCA` to nazwa dostawcy, dla którego jest przeznaczony moduł (np. `aws`), a `NAZWA` to nazwa modułu (np. `vault`).
- Moduł musi być zgodny z określoną strukturą plików, m.in. umieszczać kod Terraform w katalogu głównym repozytorium, dostarczać plik *README.md* oraz stosować konwencję nazw *main.tf*, *variables.tf* i *outputs.tf*.
- Repozytorium musi stosować tagi Git i wersjonowanie semantyczne (`x.y.z`) dla wydań.

Jeżeli moduł spełnia te wymagania, można się nim podzielić ze światem przez zalogowanie w Terraform Registry za pomocą konta w serwisie GitHub i wykorzystanie działającego w przeglądarce WWW interfejsu użytkownika do opublikowania modułu. Gdy moduł znajdzie się w rejestrze, uzyskasz elegancki interfejs użytkownika pozwalający na przeglądanie zasobów modułu, jak pokazałem na rysunku 6.4.

⁶ Szczegółowe informacje związane z publikowaniem modułów w repozytorium Terraform Registry znajdziesz na stronie <https://www.terraform.io/docs/registry/modules/publish.html>.

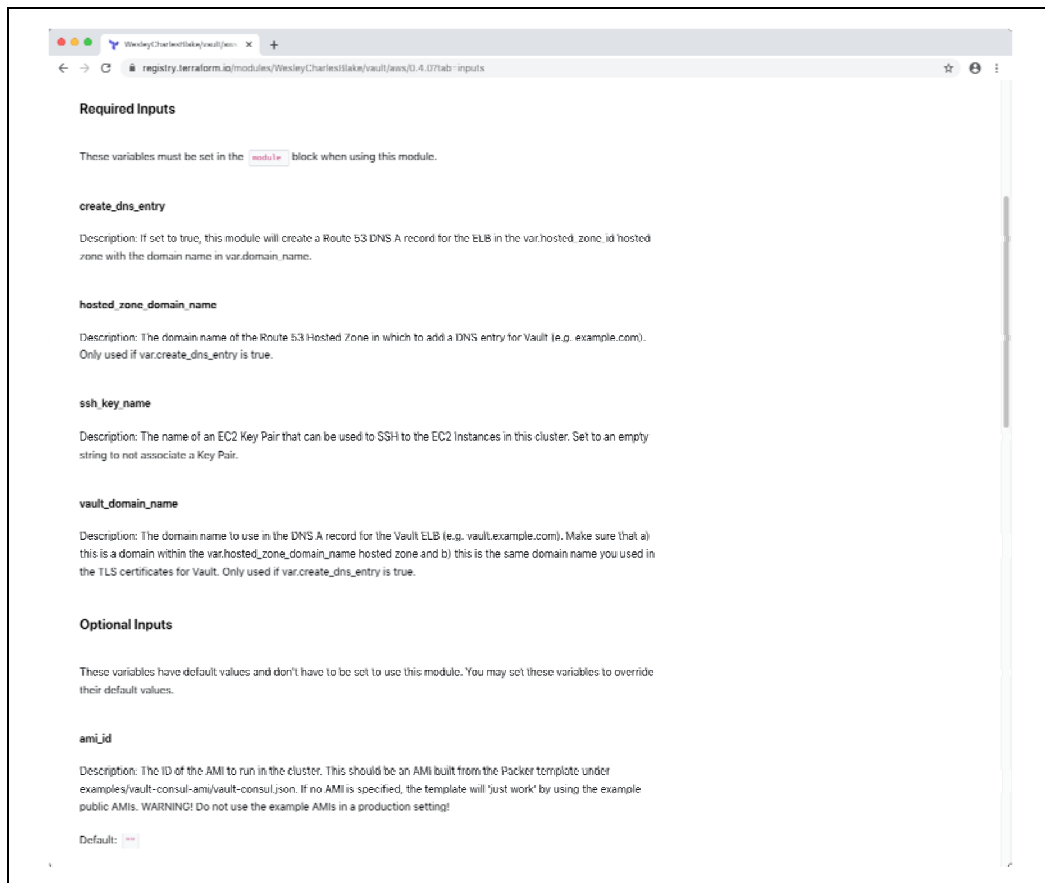


Rysunek 6.4. Moduł HashiCorp Vault w Terraform Registry

Terraform Registry potrafi przetwarzać dane wejściowe i wyjściowe modułu, więc są one również uwzględnione w interfejsie użytkownika, łącznie z polami `type` i `description`, jak pokazałem na rysunku 6.5.

Terraform obsługuje nawet specjalną składnię przeznaczoną do wykorzystania modułów pochodzących z Terraform Registry. Zamiast z długich adresów URL repozytoriów Git wraz z trudnymi do wychwycenia parametrami `ref` można skorzystać z krótszego adresu URL w argumencie `source` i określić wersję za pomocą oddzielnego argumentu `version` i przedstawionej tutaj składni:

```
module "<NAZWA>" {
  source = "<WŁAŚCICIEL>/<REPOZYTORIUM>/<DOSTAWCA>"
  version = "<WERSJA>"
  # (...)
}
```



Rysunek 6.5. Terraform Registry automatycznie przetwarza i wyświetla dane wejściowe i wyjściowe modułu

gdzie NAZWA to identyfikator używany dla modułu w kodzie Terraform, WŁAŚCICIEL to właściciel repozytorium GitHub (np. dla `github.com/foo/bar` właścicielem jest `foo`), REPOZYTORIUM to nazwa repozytorium w serwisie GitHub (np. dla `github.com/foo/bar` repozytorium to `bar`), DOSTAWCA to nazwa dostawcy docelowego (np. `aws`), a WERSJA to używana wersja modułu. Oto przykład użycia modułu Vault z Terraform Registry:

```
module "vault" {
  source = "hashicorp/vault/aws"
  version = "0.12.2"

  # (...)
}
```

Jeżeli jesteś klientem HashiCorp Terraform Enterprise, w taki sam sposób możesz używać Private Terraform Registry, czyli rejestru prywatnych repozytoriów Git dostępnych jedynie dla Twojego zespołu. To jest doskonały sposób na współdzielenie modułów w firmie.

Moduły wykraczające poza Terraform

Wprawdzie to jest książka o Terraform, ale do zbudowania całej infrastruktury o jakości produkcyjnej będą potrzebne jeszcze inne narzędzia, takie jak Docker, Packer, Chef, Puppet oraz oczywiście taśma klejąca, klej i wół roboczy w świecie DevOps, czyli stary, dobry skrypt powłoki Bash. Większość tego kodu można umieścić bezpośrednio w katalogu *modules* wraz z kodem Terraform. Przykładowo w repozytorium HashiCorp Vault, które przedstawiłem we wcześniejszej części rozdziału, katalog *modules* zawierał nie tylko kod Terraform, taki jak wspomniany wcześniej moduł `vault-cluster`, ale również skrypty powłoki Bash, np. `run-vault` (<https://github.com/hashicorp/terraform-aws-vault/tree/master/modules/run-vault>) wykonywany w serwerze Linux podczas jego uruchomienia (np. za pomocą danych użytkownika) oraz przeznaczony do skonfigurowania i uruchomienia Vault.

Możesz jednak pójść o krok dalej i uruchamiać kod inny niż Terraform (np. skrypt) bezpośrednio z poziomu modułu Terraform. Czasami pozwala na to integracja Terraform z innym systemem (zajmowałeś się już wykorzystaniem Terraform do konfiguracji skryptów danych użytkownika do wykonywania w egzemplarzach EC2). W innych przypadkach to rodzaj obejścia ograniczeń Terraform, np. brakującego API dostawcy lub brak możliwości implementacji skomplikowanej logiki ze względu na deklaracyjną naturę Terraform. Jeżeli się rozejrzysz, znajdziesz kilka „luk bezpieczeństwa” w Terraform, dzięki którym to jest możliwe:

- blok `provisioner`,
- blok `provisioner` wraz z `null_resource`,
- zewnętrzne źródła danych.

W kolejnych punktach przeanalizuję wymienione luki.

Blok `provisioner`

Blok `provisioner` w Terraform jest używany do wykonywania skryptów w komputerze lokalnym lub zdalnym podczas pracy z Terraform, zwykle odpowiada za zadania typu przygotowanie zasobów, zarządzanie konfiguracją i operacje porządkowe. Istnieje kilka rodzajów takich bloków, m.in. `local-exec` (wykonanie skryptu w komputerze lokalnym), `remote-exec` (wykonanie skryptu w zdalnym zasobie), `chef` (uruchomienie klienta Chef w zdalnym zasobie) i `file` (skopiowanie plików do zdalnego zasobu)⁷.

Blok `provisioner` można dodać do zasobu za pomocą bloku rozpoczynającego się od słowa kluczowego `provisioner`. Spójrz na przykład użycia bloku `provisioner` typu `local-exec` w celu wykonania skryptu w komputerze lokalnym:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo \"Witaj, świecie z ${uname -smp}\""
  }
}
```

⁷ Pełną listę dostępnych bloków `provisioner` znajdziesz na stronie <https://www.terraform.io/docs/provisioners/index.html>.

Po wydaniu polecenia `terraform apply` zostanie wyświetlony komunikat *Witaj, świecie z*, a następnie lokalny system operacyjny wyświetli pewne szczegóły o systemie otrzymane dzięki wykonaniu polecenia `uname`.

```
$ terraform apply
```

```
(...)
```

```
aws_instance.example (local-exec): Witaj, świecie z Darwin x86_64 i386
```

```
(...)
```

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Wypróbowanie bloku `provisioner` typu `remote-exec` jest nieco bardziej skomplikowane. Aby wykonać kod w zdalnym zasobie, takim jak egzemplarz EC2, klient Terraform musi mieć wymienione tutaj możliwości:

Komunikacja z egzemplarzem EC2 poprzez sieć

Wiesz już doskonale, jak można na to zezwolić za pomocą grupy bezpieczeństwa.

Uwierzytelnianie w egzemplarzu EC2

Blok `provisioner` typu `remote-exec` obsługuje połączenia SSH i WinRM. Skoro zostanie uruchomiony egzemplarz Linux EC2 (Ubuntu), wykorzystasz uwierzytelnianie SSH. To oznacza konieczność skonfigurowania kluczy SSH.

Rozpoczynamy od utworzenia grupy bezpieczeństwa pozwalającej na połączenia przychodzące do portu 22, czyli domyślnego portu dla SSH.

```
resource "aws_security_group" "instance" {
  ingress {
    from_port = 22
    to_port   = 22
    protocol = "tcp"

    # Aby ułatwić wypróbowanie przykładu, zezwalamy na wszystkie połączenia SSH.
    # W rzeczywistym projekcie należy je ograniczyć jedynie do zaufanych adresów IP.
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

W przypadku kluczy SSH standardowy proces oznacza wygenerowanie pary kluczy SSH w komputerze, przekazanie klucza publicznego do AWS i przechowywanie klucza prywatnego w bezpiecznym miejscu, do którego kod Terraform może mieć dostęp. Jednak w celu ułatwienia wypróbowania tego kodu można wykorzystać zasób o nazwie `tls_private_key` do automatycznego wygenerowania klucza prywatnego.

```
# Aby ułatwić wypróbowanie przykładu, klucz prywatny zostanie wygenerowany w Terraform.
# W rzeczywistym projekcie kluczami SSH należy zarządzać poza Terraform.
resource "tls_private_key" "example" {
  algorithm = "RSA"
  rsa_bits  = 4096
}
```


Ten klucz prywatny jest przechowywany w informacjach o stanie Terraform, co nie jest dobrym rozwiązaniem w środowisku produkcyjnym, ale wystarczającym w tym ćwiczeniu. Następnym krokiem jest przekazanie klucza publicznego do AWS za pomocą zasobu `aws_key_pair`.

```
resource "aws_key_pair" "generated_key" {
  public_key = tls_private_key.example.public_key_openssh
}
```

Teraz można przystąpić do utworzenia kodu dla egzemplarza EC2.

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfaffe1f0"
  instance_type = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name      = aws_key_pair.generated_key.key_name
}
```

Kilka pierwszych wierszy kodu powinno być Ci znanych: wdrożenie obrazu AMI z systemem Ubuntu w egzemplarzu typu `t2.micro` i powiązanie utworzonej wcześniej grupy bezpieczeństwa z tym egzemplarzem EC2. Jedynym nowym elementem jest użycie atrybutu `key_name` w celu nakazania AWS powiązania Twojego klucza publicznego z danym egzemplarzem EC2. Usługa AWS spowoduje dołączenie tego klucza publicznego do pliku *authorized_keys* serwera, co pozwoli na nawiązanie połączenia SSH z serwerem za pomocą odpowiedniego klucza prywatnego.

Następnym krokiem do wykonania jest dodanie bloku `provisioner` typu `remote-exec` do naszego egzemplarza EC2.

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfaffe1f0"
  instance_type = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name      = aws_key_pair.generated_key.key_name

  provisioner "remote-exec" {
    inline = ["echo \"Witaj, świecie z $(uname -smp)\""]
  }
}
```

Ten kod przedstawia się niemal identycznie jak w przypadku typu `local-exec` z wyjątkiem tego, że zamiast pojedynczego argumentu `command` został użyty argument `inline` w celu przekazania listy poleceń do wykonania. Gdy w użyciu jest blok `provisioner` typu `remote-exec`, konieczne jest jeszcze skonfigurowanie Terraform do użycia SSH podczas połączenia z egzemplarzem EC2. Odpowiednią konfigurację należy umieścić w bloku `connection`.

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfaffe1f0"
  instance_type = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name      = aws_key_pair.generated_key.key_name

  provisioner "remote-exec" {
    inline = ["echo \"Witaj, świecie z $(uname -smp)\""]
  }

  connection {
    type = "ssh"
  }
}
```

```

    host      = self.public_ip
    user      = "ubuntu"
    private_key = tls_private_key.example.private_key_pem
  }
}

```

Ten blok `connection` nakazuje Terraform nawiązanie połączenia z publicznym adresem IP egzemplarza EC2 przy użyciu SSH wraz z nazwą użytkownika `ubuntu` (to jest nazwa domyślna użytkownika `root` w systemie Ubuntu w obrazie AMI) i automatycznie wygenerowanym kluczem prywatnym. Zwróć uwagę na użycie słowa kluczowego `self` do ustawienia parametru `host`. Wyrażenie `self` stosuje następującą składnię:

```
self.<ATRYBUT>
```

Tę składnię specjalną można wykorzystać tylko w blokach `connection` i `provisioner` w celu odwołania się do `ATRYBUTU` zasobu. Jeżeli spróbujesz użyć standardowej składni `aws_instance.example.<ATRYBUT>`, spowodujesz błąd zależności cyklicznej, ponieważ zasób nie może mieć odwołania do siebie samego, więc wyrażenie `self` jest obejściem przeznaczonym specjalnie dla dostawców.

Po wydaniu polecenia `terraform apply` dla tego kodu otrzymasz następujące dane wyjściowe:

```
$ terraform apply
```

```
(...)
```

```

aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Provisioning with 'remote-exec'...
aws_instance.example (remote-exec): Connecting to remote host via SSH...
aws_instance.example (remote-exec): Connecting to remote host via SSH...
aws_instance.example (remote-exec): Connecting to remote host via SSH...

```

```
(... repeats a few more times ...)
```

```

aws_instance.example (remote-exec): Connecting to remote host via SSH...
aws_instance.example (remote-exec): Connected!
aws_instance.example (remote-exec): Witaj, świecie z Linux x86_64 x86_64

```

```
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
```

Kod w bloku `provisioner` typu `remote-exec` nie ma informacji o tym, kiedy dokładnie egzemplarz EC2 zostanie uruchomiony i będzie gotowy na przyjmowanie połączeń. Dlatego też próba nawiązania połączenia SSH będzie ponawiana wielokrotnie, aż do chwili, gdy się uda, lub do upływu zdefiniowanego czasu (domyślnie to 5 minut, choć tę wartość można zmienić). Ostatecznie nawiązanie połączenia zakończy się sukcesem i otrzymasz z serwera komunikat typu *Witaj, świecie*.

Warto zwrócić uwagę, że w trakcie definiowania bloku `provisioner` domyślnie jest to *blok provisioner oparty na czasie*, co oznacza, że działa (1) podczas wykonywania polecenia `terraform apply` i (b) jedynie podczas początkowej operacji tworzenia zasobu. Blok `provisioner` *nie* zostanie wykonany podczas kolejnych wywołań `terraform apply`, więc jest użyteczny przede wszystkim do uruchomienia kodu początkowo przygotowującego zasób. Jeżeli argumentem bloku `provisioner` będzie `when = "destroy"`, stanie się *blokiem provisioner w trakcie usuwania zasobu*, co oznacza, że będzie wykonany (a) po wydaniu polecenia `terraform destroy` i (b) tuż przed usunięciem zasobu.

W tym samym zasobie można zdefiniować wiele bloków `provisioner`, a Terraform wykona je pojedynczo w kolejności ich zdefiniowania. Istnieje możliwość użycia argumentu `on_failure` w celu wskazania Terraform sposobu obsługi błędów generowanych przez blok `provisioner`: wartość `continue` oznacza zignorowanie błędu i kontynuowanie operacji tworzenia lub usunięcia zasobu, natomiast wartość `abort` oznacza przerwanie operacji tworzenia lub usuwania zasobu.

Blok `provisioner` kontra dane użytkownika

Poznałeś dwa sposoby wykonywania skryptów w serwerze za pomocą Terraform: wykorzystanie bloku `provisioner` typu `remote-exec` oraz użycie skryptu danych użytkownika. Według mnie druga z wymienionych możliwości jest znacznie bardziej użytecznym rozwiązaniem:

- Blok `remote-exec` wymaga nawiązania połączenia SSH lub dostępu WinRM do serwerów, co jest znacznie bardziej skomplikowane w zarządzaniu (jak mogłeś zobaczyć wcześniej na przykładzie grupy bezpieczeństwa i generowania kluczy SSH) i dużo mniej bezpieczne niż dane użytkownika, które wymagają jedynie dostępu API AWS (ten dostęp i tak musisz mieć, jeśli chcesz używać Terraform).
- Skryptów danych użytkownika można używać w grupie ASG, co gwarantuje wykonanie skryptu przez wszystkie serwery podczas uruchamiania grupy. To dotyczy także nowych serwerów uruchamianych ze względu na automatyczne skalowanie grupy lub po awarii istniejącego. Blok `provisioner` jest wykonywany jedynie podczas działania Terraform i w ogóle nie współdziała z grupą ASG.
- Skrypt danych użytkownika można zobaczyć w konsoli EC2 (wybierz egzemplarz, kliknij opcję menu *Actions/Instance Settings/View or Change User Data*), dziennik zdarzeń z jego wykonania zaś znajduje się w samym egzemplarzu EC2 (zwykle w pliku `/var/log/cloud-init*.log`). Obie wymienione możliwości okazują się użyteczne podczas procesu debugowania i żadna z tych możliwości nie jest oferowana przez blok `provisioner`.

Za jedyne zalety bloku `provisioner` można uznać:

- Dowolną wielkość skryptu bloku `provisioner`, podczas gdy skrypt danych użytkownika jest ograniczony do 16 KB.
- Bloki `provisioner` przygotowane dla oprogramowania Chef, Puppet i Salt pozwalają na zainstalowanie, skonfigurowanie i uruchomienie w serwerze klientów narzędzi — odpowiednio — Chef, Puppet i Salt. To oznacza łatwiejsze użycie narzędzi zarządzania konfiguracją zamiast skryptów tymczasowych do konfigurowania serwerów.

Blok `provisioner` wraz z `null_resource`

Blok `provisioner` może być zdefiniowany tylko w zasobie, choć czasami zachodzi potrzeba jego wykonania bez powiązania z określonym zasobem. W takim przypadku można wykorzystać `null_resource`, czyli komponent działający podobnie jak zwykły zasób Terraform, ale niczego nie tworzący. Dzięki zdefiniowaniu bloku `provisioner` wraz z `null_resource` skrypty można uruchamiać jako część cyklu życiowego Terraform, ale bez powiązania z jakimkolwiek „rzeczywistym” zasobem.

```
resource "null_resource" "example" {
  provisioner "local-exec" {
    command = "echo \"Witaj, świecie z $(uname -smp)\""
  }
}
```

Komponent `null_resource` oferuje użyteczny argument o nazwie `triggers` pobierający mapę kluczy i wartości. Gdy wartość ulega zmianie, `null_resource` zostanie ponownie utworzony, co z kolei wymusi ponowne wykonanie bloku `provisioner`. Przykładowo, jeśli chcesz wykonać blok `provisioner` wraz z `null_resource` w trakcie każdego wydania polecenia `terraform apply`, możesz skorzystać z funkcji wbudowanej `uuid()`, która zwraca losowo wygenerowaną wartość UUID podczas każdego wywołania w argumentcie `triggers`.

```
resource "null_resource" "example" {  
  # Użyj wartości UUID do wymuszenia, aby ten komponent null_resource  
  # został odtworzony w trakcie każdego wywołania 'terraform apply'.  
  triggers = {  
    uuid = uuid()  
  }  
  
  provisioner "local-exec" {  
    command = "echo \"Witaj, świecie z $(uname -smp)\""  
  }  
}
```

Teraz podczas każdego wywołania `terraform apply` nastąpi wykonanie bloku `provisioner` typu `local-exec`.

```
$ terraform apply
```

```
(...)
```

```
null_resource.example (local-exec): Witaj, świecie z Darwin x86_64 i386
```

```
$ terraform apply
```

```
null_resource.example (local-exec): Witaj, świecie z Darwin x86_64 i386
```

Zewnętrzne źródło danych

Blok `provisioner` zwykle będzie rozwiązaniem pozwalającym na wykonywanie skryptów z poziomu Terraform, choć nie zawsze to najlepsze podejście. Czasami szukasz po prostu możliwości wykonania skryptu w celu pobrania pewnych danych i ich udostępnienia w samym kodzie Terraform. W takim przypadku można wykorzystać źródło danych `external` pozwalające na użycie w charakterze źródła danych polecenia zewnętrznego implementującego określony protokół.

Protokół przedstawia się następująco:

- Dane można przekazać z Terraform do programu zewnętrznego, używając argumentu query źródła danych `external`. Program zewnętrzny może odczytywać te argumenty w postaci danych JSON pochodzących ze standardowego wejścia.
- Program zewnętrzny może przekazać dane z powrotem do Terraform przez zapis danych JSON w standardowym wyjściu. Następnie pozostała część kodu Terraform może pobrać dane z JSON, wykorzystując do tego atrybut danych wyjściowych `result` zewnętrznego źródła danych.

Spójrz na przykład:

```
data "external" "echo" {  
  program = ["bash", "-c", "cat /dev/stdin"]  
}
```

```

    query = {
      foo = "bar"
    }
  }

  output "echo" {
    value = data.external.echo.result
  }

  output "echo_foo" {
    value = data.external.echo.result.foo
  }

```

Ten przykład używa źródła danych `external` do wykonania skryptu Basha przekazującego do standardowego wyjścia wszelkie dane otrzymane poprzez standardowe wejście. Dlatego też wszelkie dane przekazane za pomocą argumentu `query` powinny zostać wyświetlone za pomocą argumentu danych wyjściowych `result`. Oto wynik wykonania polecenia `terraform apply` dla omawianego fragmentu kodu:

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```

echo = {
  "foo" = "bar"
}
echo_foo = bar

```

Możesz zobaczyć, że `data.external.<NAME>.result` zawiera dane JSON zwrócone przez program zewnętrzny. Do poruszania się po tych danych JSON można wykorzystać składnię `data.external.<NAME>.result.<PATH>` (`np. data.external.echo.result.foo`).

Źródło danych `external` to świetna luka bezpieczeństwa, jeśli potrzebujesz dostępu do danych w kodzie Terraform i nie istnieje źródło danych pozwalające na ich pobranie. Jednak powinieneś dość ostrożnie podchodzić do używania źródeł danych `external` oraz wszelkich pozostałych „luk bezpieczeństwa” w Terraform, ponieważ zmniejszają one przenośność i niezawodność działania kodu. Przykładowo zaprezentowany tutaj kod wykorzystujący źródło danych opiera swoje działanie na powłoce Bash, co oznacza, że moduł Terraform zawierający ten kod nie będzie mógł być wdrożony w systemie Windows.

Podsumowanie

Skoro poznałeś wszystkie składniki pozwalające na tworzenie kodu Terraform o jakości produkcyjnej, warto połączyć ze sobą wszystkie fragmenty układanki. Gdy następnym razem rozpoczniesz pracę nad nowym modulem, skorzystaj z przedstawionego tutaj procesu:

1. Powróć do zamieszczonej w tabeli 6.2 listy rzeczy do sprawdzenia dla infrastruktury o jakości produkcyjnej i ustal, które elementy będziesz implementować, a które zostaną pominięte.

Dane otrzymane po sprawdzeniu tej listy oraz listy w tabeli 6.1 powinny pomóc w oszacowaniu czasu potrzebnego na wykonanie zadania.

2. Utwórz katalog *examples* i najpierw zajmij się zdefiniowaniem kodu przykładu. Postaraj się opracować najlepsze z punktu widzenia użytkownika rozwiązanie i najbardziej przejrzyste API, jakie tylko jesteś w stanie zdefiniować. Przygotuj przykład dla każdej ważnej permutacji modułu oraz dodaj na tyle obszerną dokumentację i zdefiniuj rozsądne wartości domyślne, aby jak najbardziej ułatwić wdrożenie przykładu.
 3. Utwórz katalog *modules* i zaimplementuj API w postaci małych modułów wielokrotnego użycia, które można ze sobą łączyć. Do ich implementacji skorzystaj z połączenia narzędzi Terraform i innych, takich jak Docker, Packer i Bash. Upewnij się o przypisaniu numerów wersji Terraform i dostawców.
 4. Utwórz katalog *test* i umieść w nim zautomatyzowane testy dla każdego przykładu.
- Teraz powinieneś poznać sposoby tworzenia zautomatyzowanych testów dla kodu infrastruktury. To będzie tematem rozdziału 7.

Testowanie kodu Terraform

Świat DevOps jest pełen różnych obaw: przed przestojem, utratą danych, złamaniem zabezpieczeń itd. Za każdym razem, gdy wprowadzasz zmianę, zastanawiasz się, czy będzie ona miała jakieś negatywne skutki. Czy będzie można ją przeprowadzić w taki sam sposób w każdym środowisku? Czy spowoduje jakiegolwiek problemy? Jeżeli wystąpią problemy, ile czasu zajmie ich usunięcie? Firma — wraz z rozwojem — ma więcej do stracenia, co powoduje, że proces wdrożenia staje się coraz bardziej przerażający i podatny na błędy. Wiele firm próbuje to złagodzić przez rzadsze wdrożenia, ale wskutek tego są one dużo większe i w rzeczywistości okazują się znacznie podatniejsze na błędy i awarie.

Jeżeli zarządzasz infrastrukturą w postaci kodu, masz lepsze możliwości w zakresie zmniejszenia ryzyka: testy. Celem testów jest dostarczenie pewności, że zmiany nie spowodują problemów. Kluczowym słowem jest tutaj *pewność*: żadna forma testów nie daje gwarancji, że kod jest pozbawiony błędów, więc mamy do czynienia raczej z prawdopodobieństwem. Jeżeli całą infrastrukturę i proces wdrożenia możesz zdefiniować w kodzie, masz możliwość jego przetestowania w środowisku przedprodukcyjnym. Jeżeli ten kod działa, istnieje wysokie prawdopodobieństwo, że będzie działał także w środowisku produkcyjnym. W świecie wielu obaw i niepewności wysokie prawdopodobieństwo i wysoka pewność dość długo idą w parze.

W tym rozdziale zamierzam przedstawić proces testowania kodu przedstawiającego infrastrukturę. Wykorzystam testy ręczne i zautomatyzowane, przy czym większość rozdziału będzie poświęcona temu drugiemu rodzajowi testów.

- Testy ręczne:
 - podstawy ręcznego przeprowadzania testów,
 - porządkowanie po zakończeniu testów.
- Testy zautomatyzowane:
 - testy jednostkowe,
 - testy integracji,
 - testy typu E2E,
 - inne podejścia w zakresie testów.



Przykładowe fragmenty kodu

Przypominam, że wszystkie przykładowe fragmenty kodu znajdują się w repozytorium GitHub pod adresem <https://github.com/brikis98/terraform-up-and-running-code>.

Testy ręczne

Jeśli zastanawiasz się nad sposobem przetestowania kodu Terraform, dobrze jest pomyśleć, jak byłby przetestowany kod, gdyby został utworzony w języku programowania ogólnego przeznaczenia, takim jak Ruby. Przyjmuję założenie, że w pliku *webserver.rb* utworzyłeś w języku Ruby prosty serwer WWW.

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    case request.path
    when "/"
      response.status = 200
      response['Content-Type'] = 'text/plain'
      response.body = 'Witaj, świecie'
    else
      response.status = 404
      response['Content-Type'] = 'text/plain'
      response.body = 'Nie znaleziono'
    end
  end
end
```

Ten fragment kodu spowoduje udzielenie odpowiedzi wraz z kodem stanu 200 OK i komunikatem Witaj, świecie dla żądań do adresu URL / oraz z kodem stanu 404 dla wszystkich pozostałych adresów URL. Jak można ręcznie przeprowadzić testy tego fragmentu kodu? Typową odpowiedzią jest dodanie kolejnego kodu przeznaczonego do uruchomienia serwera WWW w komputerze lokalnym.

```
# Ten kod będzie działał tylko po uruchomieniu skryptu bezpośrednio z poziomu CLI,
# ale nie w sytuacji, gdy będzie żądany z poziomu innego pliku
if __FILE__ == $0
  # Uruchomienie serwera w komputerze lokalnym i nasłuchującego na porcie 8000.
  server = WEBrick::HTTPServer.new :Port => 8000
  server.mount '/', WebServer

  # Zakończenie działania serwera następuje po naciśnięciu klawiszy Ctrl+C.
  trap 'INT' do server.shutdown end

  # Uruchomienie serwera.
  server.start
end
```

Jeżeli ten plik uruchomisz bezpośrednio w powłoce (CLI), nastąpi uruchomienie serwera WWW nasłuchującego na porcie 8000.

```
$ ruby web-server.rb
[2019-05-25 14:11:52] INFO WEBrick 1.3.1
[2019-05-25 14:11:52] INFO ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]
[2019-05-25 14:11:52] INFO WEBrick::HTTPServer#start: pid=19767 port=8000
```


Do przetestowania działania serwera można wykorzystać przeglądarkę WWW lub narzędzie curl.

```
$ curl localhost:8000/  
Witaj, świecie  
  
$ curl localhost:8000/invalid-path  
Nie znaleziono
```

Przyjmuję założenie o wprowadzeniu w kodzie zmiany polegającej na dodaniu punktu końcowego `/api`, który udziela odpowiedzi wraz z kodem stanu 201 Created i danymi JSON.

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet  
  def do_GET(request, response)  
    case request.path  
    when "/"  
      response.status = 200  
      response['Content-Type'] = 'text/plain'  
      response.body = 'Witaj, świecie'  
    when "/api"  
      response.status = 201  
      response['Content-Type'] = 'application/json'  
      response.body = '{"foo":"bar"}'  
    else  
      response.status = 404  
      response['Content-Type'] = 'text/plain'  
      response.body = 'Nie znaleziono'  
    end  
  end  
end
```

Aby ręcznie przetestować uaktualniony kod, należy nacisnąć klawisze `Ctrl+C` i ponownie uruchomić skrypt, a tym samym serwer WWW.

```
$ ruby web-server.rb  
[2019-05-25 14:11:52] INFO WEBrick 1.3.1  
[2019-05-25 14:11:52] INFO ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]  
[2019-05-25 14:11:52] INFO WEBrick::HTTPServer#start: pid=19767 port=8000  
^C  
[2019-05-25 14:15:54] INFO going to shutdown ...  
[2019-05-25 14:15:54] INFO WEBrick::HTTPServer#start done.  
  
$ ruby web-server.rb  
[2019-05-25 14:11:52] INFO WEBrick 1.3.1  
[2019-05-25 14:11:52] INFO ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]  
[2019-05-25 14:11:52] INFO WEBrick::HTTPServer#start: pid=19767 port=8000
```

Do przetestowania nowej wersji ponownie będzie użyte narzędzie curl.

```
$ curl localhost:8000/api  
{"foo":"bar"}
```

Podstawy ręcznego przeprowadzania testów

Jak w Terraform przedstawia się odpowiednik takiego ręcznego testowania kodu? Przykładowo po pracy na podstawie poprzednich rozdziałów masz opracowany kod Terraform przeznaczony do wdrożenia mechanizmu równoważenia obciążenia, ALB. Spójrz na kod znajdujący się w pliku `modules/networking/alb/main.tf`:

```

resource "aws_lb" "example" {
  name           = var.alb_name
  load_balancer_type = "application"
  subnets       = var.subnet_ids
  security_groups = [aws_security_group.alb.id]
}

resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = local.http_port
  protocol          = "HTTP"

  # Domyślnie zwracana jest prosta strona błędu 404.
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: nie znaleziono strony"
      status_code  = 404
    }
  }
}

resource "aws_security_group" "alb" {
  name = var.alb_name
}

# (...)

```

Jeżeli porównasz ten kod z kodem w języku Ruby, różnica powinna być oczywista: w swoim komputerze nie możesz wdrożyć mechanizmu równoważenia obciążenia AWS, grup docelowych, komponentów nasłuchujących, grup bezpieczeństwa oraz pozostałych elementów infrastruktury.

W ten sposób dochodzimy do *pierwszej reguły związanej z testowaniem*: podczas testowania kodu Terraform nie istnieje tzw. komputer lokalny.

Ten wniosek ma zastosowanie do większości wykorzystujących podejście IaC narzędzi, a nie tylko do Terraform. Praktycznym sposobem na ręczne przetestowanie kodu Terraform jest jego wdrożenie w rzeczywistym środowisku (np. w AWS). Innymi słowy, ręczne wydawanie poleceń `terraform apply` i `terraform destroy` w trakcie lektury książki pokazało, jak przeprowadzać ręczne testowanie kodu Terraform.

To jest jeden z ważnych powodów, dla których w katalogu *examples* każdego modułu należy przygotować łatwe do wdrożenia przykłady, jak to omówiłem w rozdziale 6. Najłatwiejszy sposób na ręczne przetestowanie modułu `alb` polega na użyciu przykładowego kodu utworzonego w katalogu *examples/alb*.

```

provider "aws" {
  region = "us-east-2"

  # Zezwolenie na użycie dowolnej wersji 2.x dostawcy AWS.
  version = "~> 2.0"
}

```

```
module "alb" {
  source = "../../modules/networking/alb"

  alb_name   = "terraform-up-and-running"
  subnet_ids = data.aws_subnet_ids.default.ids
}
```

Podobnie jak to już wielokrotnie widziałeś w książce, wdrożenie przykładu odbywa się po wydaniu polecenia `terraform apply`.

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

Po zakończeniu wdrożenia narzędzie typu `curl` można wykorzystać do przetestowania, czy akcja domyślna ALB zwraca odpowiedź wraz z kodem stanu 404.

```
$ curl \
-s \
-o /dev/null \
-w "%{http_code}" \
hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

```
404
```



Weryfikacja infrastruktury

Przykłady przedstawione w tym rozdziale używają żądań HTTP i narzędzia `curl` do sprawdzenia poprawności działania infrastruktury, ponieważ testowana infrastruktura zawiera mechanizm równoważenia obciążenia udzielający odpowiedzi na żądania HTTP. W przypadku pozostałych typów infrastruktury konieczne jest zastąpienie żądań HTTP i narzędzia `curl` inną formą sprawdzenia poprawności. Przykładowo, jeśli kod infrastruktury przeprowadza wdrożenie bazy danych MySQL, weryfikację trzeba wykonać za pomocą klienta MySQL. Jeśli natomiast kod infrastruktury wdraża serwer VPN, weryfikację trzeba przeprowadzić za pomocą klienta VPN. Z kolei, jeżeli kod infrastruktury wdraża serwer w ogóle nienasłuchujący żądań, konieczne może być nawiązanie połączenia SSH z serwerem i lokalne wykonanie pewnych poleceń w celu jego przetestowania itd. Tak więc opisaną w rozdziale tę samą podstawową strukturę testu wprawdzie można wykorzystać wraz z dowolnym typem infrastruktury, ale procedura weryfikacji będzie się zmieniała w zależności od przedmiotu testu.

Przypominam, że ALB zwraca kod stanu 404, ponieważ nie zostały skonfigurowane żadne reguły komponentów nasłuchujących, a domyślną akcją modułu `alb` jest udzielenie odpowiedzi wraz z kodem stanu 404.

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = local.http_port
}
```

```

protocol      = "HTTP"

# Domyślnie zwracana jest prosta strona błędu 404.
default_action {
  type = "fixed-response"

  fixed_response {
    content_type = "text/plain"
    message_body = "404: nie znaleziono strony"
    status_code  = 404
  }
}
}

```

W ten sposób masz możliwość uruchomienia i przetestowania kodu, więc możesz rozpocząć wprowadzanie zmian. Po każdej modyfikacji — np. po zmianie akcji domyślnej, aby zwracała kod stanu 401 — można ponownie wydać polecenie `terraform apply` i wprowadzić kolejne zmiany.

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

```
Outputs:
```

```
alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

Kolejnym krokiem jest ponowne użycie polecenia `curl` w celu przetestowania nowej wersji.

```

$ curl \
-s \
-o /dev/null \
-w "%{http_code}" \
hello-world-stage-477699288.us-east-2.elb.amazonaws.com

```

```
401
```

Po zakończeniu eksperymentów należy wydać polecenie `terraform destroy`, aby przeprowadzić operacje porządkujące środowisko.

```
$ terraform destroy
```

```
(...)
```

```
Apply complete! Resources: 0 added, 0 changed, 5 destroyed.
```

Innymi słowy, podczas pracy z Terraform każdy programista potrzebuje dobrego przykładu do przetestowania i rzeczywistego środowiska wdrożenia (np. konta AWS) w celu jego wykorzystania jako odpowiednika komputera lokalnego dla tych testów. W procesie ręcznego przeprowadzania testów będziesz tworzyć i usuwać infrastrukturę, po drodze prawdopodobnie popełnisz wiele błędów i dlatego środowisko testowe powinno być całkowicie odizolowane od pozostałych, znacznie stabilniejszych środowisk, takich jak robocze i przede wszystkim produkcyjne.

Zachęcam więc, aby każdy zespół przygotował *odizolowane środowisko*, w którym programiści mogą eksperymentować z infrastrukturą bez obaw o wpływ tych działań na inne środowiska. W rze-

czywistości, aby zminimalizować niebezpieczeństwo wystąpienia konfliktów między programistami (np. gdy dwóch próbuje utworzyć mechanizm równoważenia obciążenia o takiej samej nazwie), regułą jest tworzenie oddzielnego i całkowicie odizolowanego środowiska dla każdego programisty. Przykładowo, jeśli używasz Terraform w AWS, to każdy programista powinien mieć własne konto AWS przeznaczone do testowania dowolnego kodu¹.

Uporządkowanie środowiska po zakończeniu testów

Wprawdzie posiadanie wielu odizolowanych środowisk ma duże znaczenie dla produktywności programisty, ale jeśli nie zachowasz ostrożności, infrastruktura zostanie uruchomiona wszędzie, zaśmieci wszystkie dostępne środowiska i będzie słono kosztować.

Aby trzymać koszty pod kontrolą, należy pamiętać o *drugiej regule związanej z testowaniem*: regularnym porządkowaniu środowisk.

Absolutnym minimum jest wykształcenie nawyku polegającego na tym, że po zakończeniu pracy programista za pomocą polecenia `terraform destroy` usuwa wszystko to, co wcześniej wdrożył. W zależności od środowiska wdrożenia być może otrzymasz narzędzia uruchamiane według określonego harmonogramu (np. zadanie mechanizmu `cron`) i automatycznie przeprowadzające operacje usunięcia nieużywanych lub starych zasobów. Oto kilka przykładów:

`cloud-nuke` (<https://github.com/gruntwork-io/cloud-nuke>)

To narzędzie typu open source może usunąć wszystkie zasoby we wdrożeniu chmury. Aktualnie obsługuje wiele zasobów AWS (egzemplarze Amazon EC2, ASG, ELB itd.), natomiast w przyszłości ma zapewnić obsługę innych zasobów i chmur (np. Google Cloud, Azure). Kluczową funkcjonalnością jest możliwość usunięcia wszystkich zasobów starszych niż podana ilość czasu. Przykładowo często stosowany wzorzec polega na uruchamianiu `cloud-nuke` jako zadania mechanizmu `cron` raz dziennie w każdym środowisku w celu usunięcia wszystkich zasobów starszych niż dwa dni. Zastosowanie ma tutaj założenie, że każda infrastruktura utworzona przez programistę na potrzeby ręcznych testów nie jest przydatna po upływie kilku dni.

```
$ cloud-nuke aws --older-than 48h
```

Janitor Monkey (<https://github.com/Netflix/SimianArmy/wiki/Janitor-Home>)

To narzędzie typu open source, przeprowadzające operacje porządkowe zasobów AWS według ustalonego harmonogramu (domyślnie raz na tydzień). Pozwala na tworzenie elastycznych reguł określających, kiedy zasób ma zostać usunięty, a nawet umożliwia wysłanie powiadomienia przed usunięciem zasobu. To narzędzie jest częścią projektu Netflix Simian Army obejmującego również Chaos Monkey, czyli narzędzie służące do testowania odporności aplikacji. Warto w tym miejscu dodać, że wymieniony projekt nie jest dłużej aktywnie rozwijany, ale jego różne fragmenty trafiły do nowych projektów, przykładem jest Janitor Monkey zastąpiony przez Swabbie (<https://github.com/spinnaker/swabbie>).

¹ AWS nie wymaga żadnych dodatkowych opłat za kolejne konto. Jeśli korzystasz z usługi AWS Organizations, możesz stworzyć wiele kont „potomnych” połączonych (także pod względem rachunków) z jednym kontem „głównym”.

aws-nuke (<https://github.com/rebuy-de/aws-nuke>)

Narzędzie typu open source, przeznaczone do usunięcia wszystkiego w koncie AWS. Konto i zasoby przeznaczone do usunięcia podaje się za pomocą pliku konfiguracyjnego w formacie YAML:

```
# Regiony do usunięcia.
regions:
- us-east-2

# Konta zawierające zasoby do usunięcia.
accounts:
  "111111111111": {}

# Usunięte mają zostać tylko wymienione zasoby.
resource-types:
  targets:
  - S3Object
  - S3Bucket
  - IAMRole
```

Uruchomienie omawianego narzędzia następuje za pomocą polecenia:

```
$ aws-nuke -c config.yml
```

Testy zautomatyzowane



Przed nami ogromna ilość kodu

Tworzenie testów zautomatyzowanych przeznaczonych dla kodu infrastruktury nie jest zadaniem dla każdego. Ta część książki jest bez wątpienia najbardziej skomplikowana i nie będzie łatwą lekturą. Jeżeli tylko przeglądasz książkę, spokojnie możesz pominąć tę część. Z drugiej strony, jeśli naprawdę chcesz się dowiedzieć, jak przetestować kod infrastruktury, powinieneś zakasać rękawy i zabrać się do tworzenia kodu. Nie ma konieczności uruchamiania żadnego kodu w języku Ruby (użyłem go wcześniej tylko w celu wyjaśnienia zasady działania ręcznych testów), za to będziesz tworzyć i uruchamiać duże ilości kodu w języku Go.

Idea stojąca za testami zautomatyzowanymi polega na tworzeniu kodu sprawdzającego, czy rzeczywisty kod działa zgodnie z oczekiwaniami. Jak zobaczysz w rozdziale 8., masz możliwość skonfigurowania serwera ciągłej integracji (ang. *continuous integration*, CI) przeznaczonego do wykonywania tych testów po każdej operacji zatwierdzenia, a następnie natychmiast wycofującego lub poprawiającego te operacje zatwierdzenia, które doprowadziły do niezaliczenia testów. W ten sposób masz gwarancję, że kod zawsze będzie znajdował się w stanie zapewniającym jego prawidłowe działanie.

Ogólnie rzecz biorąc, mamy trzy rodzaje testów zautomatyzowanych:

Testy jednostkowe

Test jednostkowy jest przeznaczony do sprawdzenia funkcjonalności pojedynczego, małego fragmentu kodu. Definicja *jednostki* może być różna, ale w językach programowania ogólnego przeznaczenia przyjęło się, że to jest pojedyncza funkcja lub klasa. Zwykle wszelkie zależności zewnętrzne — np. bazy danych, usługi sieciowe, a nawet systemy plików — są zastępowane

twz. *imitacjami* lub *makietami* (ang. *mock*) pozwalającymi na zachowanie szczegółowej kontroli nad tymi zależnościami (np. przez zwrot konkretnej odpowiedzi przez imitację bazy danych) i sprawdzenie, czy kod działa prawidłowo w różnych sytuacjach.

Testy integracji

Test integracji sprawdza, czy wiele jednostek prawidłowo ze sobą współpracuje. W językach programowania ogólnego przeznaczenia test integracji składa się z kodu sprawdzającego prawidłowość działania wielu funkcji lub klas. Test integracji zwykle stosuje połączenie rzeczywistych zależności i makiet: np. jeśli sprawdzany jest fragment aplikacji odpowiedzialny za komunikację z bazą danych, test może obejmować rzeczywistą bazę danych, natomiast imitowane są inne zależności, np. system uwierzytelniania.

Testy typu E2E

Test typu E2E (ang. *end-to-end*) w trakcie działania wykorzystuje całą architekturę — np. aplikację, magazyny danych, mechanizmy równoważenia obciążenia itd. — i sprawdza system jako całość. Zwykle te testy są przeprowadzane z perspektywy użytkownika. Przykładem może być tutaj wykorzystanie oprogramowania typu Selenium do automatyzacji pracy z produktem za pomocą interfejsu przeglądarki WWW. Test typu E2E zwykle wszędzie używa rzeczywistych systemów, bez żadnych imitacji oraz w architekturze odzwierciedlającej środowisko produkcyjne (choć z mniejszą liczbą słabszych serwerów, aby zaoszczędzić pieniądze).

Każdy typ testu służy do innych celów i może przechwytywać odmienne rodzaje błędów, więc prawdopodobnie będziesz używać połączenia wszystkich trzech typów. Celem testu jednostkowego jest jego szybkie uruchomienie, aby można było jak najszybciej otrzymać informacje odnośnie do wprowadzonej zmiany i potwierdzić poprawność działania różnych permutacji, co z kolei daje pewność o działaniu zgodnie z oczekiwaniami podstawowych elementów konstrukcyjnych aplikacji (poszczególnych jednostek). Jednak prawidłowe działanie jednostek w izolacji nie oznacza, że będą one poprawnie ze sobą współdziałały. Dlatego też potrzebne są testy integracji potwierdzające prawidłową współpracę komponentów aplikacji. Z kolei poprawne działanie różnych części systemu automatycznie nie gwarantuje jego poprawności po wdrożeniu. Stąd potrzeba stosowania testów typu E2E potwierdzających działanie kodu zgodnie z oczekiwaniami w środowisku podobnym do produkcyjnego.

Przechodzimy teraz do utworzenia poszczególnych rodzajów testów w Terraform.

Testy jednostkowe

Aby zrozumieć sposób tworzenia testów jednostkowych dla kodu Terraform, pomocne będzie poznanie sposobu ich tworzenia w języku programowania ogólnego przeznaczenia, takim jak Ruby. Raz jeszcze spojrz na kod serwera WWW utworzonego w języku Ruby:

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    case request.path
    when "/"
      response.status = 200
      response['Content-Type'] = 'text/plain'
      response.body = 'Witaj, świecie'
    when "/api"
      response.status = 201
    end
  end
end
```

```

    response['Content-Type'] = 'application/json'
    response.body = '{"foo":"bar"}'
  else
    response.status = 404
    response['Content-Type'] = 'text/plain'
    response.body = 'Nie znaleziono'
  end
end
end

```

Utworzenie testu jednostkowego dla tego kodu nie będzie takie łatwe, ponieważ pod uwagę trzeba wziąć wymienione tutaj kwestie:

1. Utworzenie egzemplarza klasy `WebServer`. To jest znacznie trudniejsze, niż się wydaje, ponieważ konstruktor klasy `WebServer` rozszerzającej `AbstractServlet` wymaga przekazania pełnej klasy `WEBrick::HTTPServer`. Wprawdzie można utworzyć jej imitację, ale to oznacza naprawdę dużo pracy.
2. Utworzenie obiektu request będącego typu `HTTPRequest`. Nie ma łatwego sposobu na utworzenie egzemplarza tej klasy, a przygotowanie imitacji oznacza naprawdę dużo pracy.
3. Utworzenie obiektu response będącego typu `HTTPResponse`. To również nie należy do łatwych zadań, przygotowanie imitacji zaś oznacza naprawdę dużo pracy.

Gdy utworzenie testu jednostkowego okazuje się trudne, często wskazuje to na niepoprawność kodu i konieczność jego refaktoryzacji. Jednym ze sposobów na refaktoryzację omawianego kodu w języku Ruby, aby ułatwić przygotowanie dla niego testów jednostkowych, jest wyodrębnienie „procedur obsługi” — tzn. kodu obsługującego żądania do adresów URL `/`, `/api` i nieznalezionych — i umieszczenie ich w oddzielnej klasie, np. o nazwie `Handlers`.

```

class Handlers
  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Witaj, świecie!']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    else
      [404, 'text/plain', 'Nie znaleziono']
    end
  end
end

```

Ta nowa klasa ma dwie cechy, na które należy zwrócić uwagę:

Dane wejściowe w postaci prostych wartości

Klasa `Handlers` nie ma zależności od klas `HTTPServer`, `HTTPRequest` i `HTTPResponse`. Zamiast tego wszystkie jej dane wejściowe mają postać prostych parametrów, np. `path` to ścieżka dostępu adresu URL w postaci ciągu tekstowego.

Dane wyjściowe w postaci prostych wartości

Zamiast przypisywania wartości modyfikowalnemu obiektowi `HTTPResponse` (efekt uboczny) metody w klasie `Handlers` zwracają odpowiedź HTTP w postaci prostej wartości — tablica zawierająca kod stanu HTTP, typ treści i samą treść.

Kod pobierający dane wejściowe w postaci prostych wartości i zwracający dane wyjściowe w postaci prostych wartości jest zwykle łatwiejszy do zrozumienia, uaktualnienia i przetestowania. Pracę trzeba zacząć od uaktualnienia klasy `WebServer`, aby podczas udzielania odpowiedzi na żądania używała nowej klasy `Handlers`.

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    handlers = Handlers.new
    status_code, content_type, body = handlers.handle(request.path)

    response.status = status_code
    response['Content-Type'] = content_type
    response.body = body
  end
end
```

Ten kod wywołuje metodę `handle()` klasy `Handlers` i udziela jako odpowiedź HTTP dane zwrócone przez metodę, zawierające kod stanu, typ treści i samą treść. Jak możesz zobaczyć, użycie klasy `Handlers` jest łatwe i proste. Dzięki tym cechom testowanie tej klasy również będzie łatwym zadaniem. Spójrz na test jednostkowy przygotowany dla punktu końcowego `/`:

```
class TestWebServer < Test::Unit::TestCase
  def initialize(test_method_name)
    super(test_method_name)
    @handlers = Handlers.new
  end

  def test_unit_hello
    status_code, content_type, body = @handlers.handle("/")
    assert_equal(200, status_code)
    assert_equal('text/plain', content_type)
    assert_equal('Witaj, świecie', body)
  end
end
```

Kod testu wywołuje tę samą metodę `handle()` klasy `Handlers` i wykorzystuje kilka metod `assert` do sprawdzenia odpowiedzi udzielonej na żądanie do punktu końcowego `/`. Oto jak zostaje wykonany test jednostkowy:

```
$ ruby web-server-test.rb
Loaded suite web-server-test
Finished in 0.000287 seconds.

-----
1 tests, 3 assertions, 0 failures, 0 errors
100% passed
-----
```

Wygląda na to, że test został zaliczony. Spójrz teraz na testy jednostkowe dla punktów końcowych `/api` i błędu 404:

```
def test_unit_api
  status_code, content_type, body = @handlers.handle("/api")
  assert_equal(201, status_code)
  assert_equal('application/json', content_type)
  assert_equal('{"foo":"bar"}', body)
end
```

```

def test_unit_404
  status_code, content_type, body = @handlers.handle("/invalid-path")
  assert_equal(404, status_code)
  assert_equal('text/plain', content_type)
  assert_equal('Nie znaleziono', body)
end

```

Ponownie uruchom testy:

```

$ ruby web-server-test.rb
Loaded suite web-server-test
Finished in 0.000572 seconds.

-----
3 tests, 9 assertions, 0 failures, 0 errors
100% passed
-----

```

W ciągu zaledwie 0,0005272 sekundy możesz się dowiedzieć, czy kod serwera WWW działa zgodnie z oczekiwaniami. To jest prawdziwa potęga testu jednostkowego: szybkie wykonanie, dzięki któremu możesz mieć pewność o prawidłowym działaniu kodu. Jeżeli popełnisz błąd w kodzie — przypadkowo zmienisz odpowiedź udzielaną na żądanie do punktu końcowego */api* — dowiesz się o tym praktycznie natychmiast.

```

$ ruby web-server-test.rb
Loaded suite web-server-test
=====
Failure: test_unit_api(TestWebServer)
web-server-test.rb:25:in `test_unit_api'
    22:   status_code, content_type, body = Handlers.new.handle("/api")
    23:   assert_equal(201, status_code)
    24:   assert_equal('application/json', content_type)
=> 25:   assert_equal('{\"foo\":\"bar\"}', body)
    26:   end
    27:
    28:   def test_unit_404
<\"{\\\"foo\\\":\\\"bar\\\"}\"> expected but was
<\"{\\\"foo\\\":\\\"whoops\\\"}\">
diff:
? {\"foo\":\"bar  \"}
?      whoops
=====
Finished in 0.007904 seconds.

-----
3 tests, 9 assertions, 1 failures, 0 errors
66.6667% passed
-----

```

Podstawy testu jednostkowego

Jak w Terraform przedstawia się odpowiednik testu jednostkowego pokazanego dla kodu w języku Ruby? Pierwszym krokiem jest ustalenie „jednostki” w świecie Terraform. Najbliższym odpowiednikiem pojedynczej funkcji lub klasy w Terraform jest pojedynczy moduł generyczny (określenia „generyczny” użyłem tutaj w znaczeniu zdefiniowanym w rozdziale 6.), np. moduł `alb` utworzony w poprzednim rozdziale. Jak można przetestować ten moduł?

W przypadku języka Ruby, aby można było utworzyć test jednostkowy, należało przeprowadzić refaktoryzację kodu, co pozwoliło na przygotowanie testu jednostkowego bez skomplikowanych zależności w postaci klas `HTTPServer`, `HTTPRequest` i `HTTPResponse`. Jeżeli zastanowisz się nad tym, na czym polega działanie kodu Terraform — wykonywanie wywołań API do AWS w celu utworzenia mechanizmu równoważenia obciążenia, komponentów nasłuchujących, grup docelowych itd. — zdasz sobie sprawę, że 99% tego kodu prowadzi komunikację wraz ze skomplikowanymi zależnościami. Nie ma praktycznego sposobu na zmniejszenie do zera liczby zewnętrznych zależności, a nawet jeśli byś to zrobił, właściwie nie będziesz miał kodu do testowania².

W ten sposób docieramy do *trzeciej reguły związanej z testowaniem*: nie można tworzyć czystych testów jednostkowych dla kodu Terraform.

Nie rozpaczaj. Nadal możesz mieć pewność co do działania kodu Terraform zgodnie z oczekiwaniami, o ile przygotujesz zautomatyzowane testy wykorzystujące ten kod w celu wdrożenia rzeczywistej infrastruktury w rzeczywistym środowisku (np. z użyciem rzeczywistego konta AWS). Innymi słowy, testy jednostkowe dla Terraform to właściwie testy integracji. Mimo to preferuję nazywanie ich testami jednostkowymi, aby podkreślić, że celem jest przeprowadzenie testu pojedynczej jednostki (np. pojedynczego, ogólnego modułu), aby jak najszybciej otrzymać informacje dotyczące sposobu jego działania.

Dlatego też podstawowa strategia w zakresie tworzenia testów jednostkowych dla Terraform przedstawia się następująco:

1. Utworzenie ogólnego, samodzielnego modułu.
2. Utworzenie łatwego do wdrożenia przykładu dla tego modułu.
3. Wydanie polecenia `terraform apply` w celu wdrożenia przykładu w rzeczywistym środowisku.
4. Sprawdzenie, czy przeprowadzone wdrożenie działa zgodnie z oczekiwaniami. Ten krok jest ściśle związany z typem testowanej infrastruktury. Przykładowo w przypadku mechanizmu równoważenia obciążenia, ALB, weryfikacja odbywa się przez wykonanie żądania HTTP i sprawdzenie, czy otrzymana odpowiedź jest zgodna z oczekiwaniami.
5. Wydanie na końcu testu polecenia `terraform destroy`, aby przeprowadzić operacje porządkowe.

Innymi słowy, wykonywane są *dokładnie* te same kroki jak w przypadku ręcznego przeprowadzania testów, choć poszczególne kroki zostały zapisane w postaci kodu. Szczercze mówiąc, to jest dobry model, jeśli chodzi o tworzenie zautomatyzowanych testów dla kodu Terraform — zadaj sobie pytanie, jak możesz to przetestować ręcznie, aby mieć pewność o prawidłowym działaniu kodu, a następnie zaimplementuj ten test w kodzie.

² W rzadkich przypadkach istnieje możliwość nadpisania punktów końcowych używanych przez Terraform do komunikacji z dostawcami. Przykładowo istnieje możliwość nadpisania punktu końcowego używanego przez Terraform do komunikacji z Amazon S3 i zastąpienia go imitacją implementującą API S3. Takie rozwiązanie sprawdza się doskonale w przypadku małej liczby punktów końcowych, ale większość kodu Terraform wykonuje *setki* różnych wywołań API do dostawców i imitowanie ich wszystkich byłoby niepraktyczne. Co więcej, nawet jeśli przygotujesz te imitacje, wciąż nie będzie jasne, czy wynik testu jednostkowego może dać wystarczającą pewność co do prawidłowego działania kodu. Jeżeli utworzysz imitację punktu końcowego dla grupy ASG i mechanizmu równoważenia obciążenia ALB, wykonanie polecenia `terraform apply` może zakończyć się powodzeniem, ale to nie dostarcza żadnych użytecznych informacji o tym, czy kod faktycznie wdroży działającą aplikację na bazie tej infrastruktury.

Do tworzenia kodu testowego można użyć dowolnego języka programowania. W książce wszystkie testy zostały przygotowane w języku programowania Go, co pozwala na wykorzystanie zalet biblioteki typu open source o nazwie Terratest (<https://github.com/gruntwork-io/terratest>), która zapewnia obsługę testowania z użyciem różnych środowisk (takich jak AWS, Google Cloud, Kubernetes). Terraform to wszechstronne narzędzie wraz z setkami innych narzędzi, które znacznie ułatwiają testowanie kodu infrastruktury — m.in. doskonała obsługa dla przedstawionej strategii testowania, w której wydajesz polecenie `terraform apply` dla pewnego kodu, weryfikujesz działanie kodu, a następnie wydajesz na końcu polecenie `terraform destroy` w celu przeprowadzenia operacji porządkowych.

Aby używać Terratest, należy zastosować się do przedstawionej tutaj procedury:

1. Instalacja języka Go: <https://golang.org/doc/install>.
2. Konfiguracja zmiennej środowiskowej `GOPATH`: <https://golang.org/doc/code.html#GOPATH>.
3. Dodanie `$GOPATH/bin` do zmiennej środowiskowej `PATH`.
4. Instalacja Dep, czyli menedżera zależności dla Go: <https://golang.github.io/dep/docs/installation.html#>.
5. Utworzenie w katalogu wskazywanym przez `GOPATH` podkatalogu przeznaczonego dla kodu testowego. Domyślnie `GOPATH` prowadzi do katalogu `$HOME/go`, więc powinieneś utworzyć katalog `$HOME/go/src/terraform-up-and-running`.
6. Wydanie polecenia `dep init` w utworzonym przed chwilą katalogu. To powinno spowodować utworzenie plików `Gopkg.toml` i `Gopkg.lock`, a także pustego katalogu `vendors`.

Można szybko sprawdzić poprawność konfiguracji środowiska pracy poprzez utworzenie w nowym katalogu pliku `go_sanity_test.go` zawierającego następujący fragment kodu:

```
package test

import (
    "fmt"
    "testing"
)

func TestGoIsWorking(t *testing.T) {
    fmt.Println()
    fmt.Println("Jeżeli widzisz ten komunikat, wszystko działa prawidłowo!")
    fmt.Println()
}
```

Uruchom ten test za pomocą polecenia `go test` i upewnij się, że wygenerowane zostały następujące dane wyjściowe:

```
$ go test -v
```

```
Jeżeli widzisz ten komunikat, wszystko działa prawidłowo!
```

³ Przyszłe wersje będą prawdopodobnie używały `go mod` do zarządzania zależnościami. W chwili powstawania książki dostępna była jedynie podstawowa obsługa `go mod`, ale po wydaniu Go 1.13, obsługa `go mod` powinna być włączona domyślnie. Dlatego też prawdopodobnie stanie się standardowym narzędziem przeznaczonym do zarządzania zależnościami w Go — dodatkową korzyścią będzie brak konieczności stosowania zmiennej środowiskowej `GOPATH`. Więcej informacji na ten temat znajdziesz na stronie <https://blog.golang.org/using-go-modules>.

```
PASS
ok  terraform-up-and-running 0.004s
```

(Opcja `-v` powoduje wyświetlenie szczegółowych danych wyjściowych, co gwarantuje, że zawsze będą wyświetlone pełne dane wyjściowe).

Jeżeli pierwszy test został zaliczony, możesz usunąć plik `go_sanitry_test.go` i przystąpić do tworzenia testów jednostkowych dla modułu `alb`. W katalogu `test` utwórz plik o nazwie `alb_example_test.go` wraz z przedstawionym tutaj szkieletem testu jednostkowego.

```
package test

import (
    "testing"
)

func TestAlbExample(t *testing.T) {
}
```

Pierwszym krokiem jest skierowanie Terratest do katalogu zawierającego kod Terraform. Do tego celu należy skorzystać z typu `terraform.Options`.

```
package test

import (
    "github.com/gruntwork-io/terratest/modules/terraform"
    "testing"
)

func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // Powinieneś uaktualnić tę względną ścieżkę dostępu,
        // aby prowadziła do katalogu examples modułu alb!
        TerraformDir: "../examples/alb",
    }
}
```

Zwróć uwagę na to, że w celu przetestowania modułu `alb` testowany będzie przykładowy kod umieszczony w katalogu `examples` (powinieneś uaktualnić względną ścieżkę dostępu w `TerraformDir`, aby prowadziła do katalogu, w którym został utworzony wspomniany przykład). To oznacza, że przykładowy kod służy teraz w trzech rolach: jako wykonywalna dokumentacja, rozwiązanie pozwalające na ręczne testowanie modułów oraz rozwiązanie umożliwiające przeprowadzanie zautomatyzowanych testów modułów.

Zauważ także umieszczone na początku pliku polecenie odpowiedzialne za zaimportowanie biblioteki Terratest. Aby pobrać te zależności do komputera, należy wydać polecenie `dep ensure`.

```
$ dep ensure
```

Polecenie `dep ensure` przeskanuje kod Go, odzyska nowe polecenia importujące biblioteki, automatycznie je pobierze do katalogu `vendors` wraz z wszystkimi zależnościami oraz dołączy je do pliku `Gopkg.lock`. Jeżeli masz wrażenie, że to działa jak magia, możesz skorzystać z polecenia `dep ensure -add` w celu jawnego dodawania żądanych zależności.

```
$ dep ensure -add github.com/gruntwork-io/terratest/modules/terraform
```

Następnym krokiem podczas przeprowadzania zautomatyzowanych testów jest wydanie poleceń `terraform init` i `terraform apply`, aby faktycznie wdrożyć kod. Terratest ma użyteczne metody pomocnicze, które służą do tego celu:

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // Powinieneś uaktualnić tę względną ścieżkę dostępu,
        // aby prowadziła do katalogu examples modułu alb!
        TerraformDir: "../examples/alb",
    }

    terraform.Init(t, opts)
    terraform.Apply(t, opts)
}
```

Szczerze mówiąc, wydawanie poleceń `terraform init` i `terraform apply` jest na tyle często wykonywaną operacją w Terratest, że istnieje wygodna metoda pomocnicza, która zadania obu poleceń wykonuje po wydaniu tylko jednego polecenia.

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // Powinieneś uaktualnić tę względną ścieżkę dostępu,
        // aby prowadziła do katalogu examples modułu alb!
        TerraformDir: "../examples/alb",
    }

    // Wdrożenie przykładu.
    terraform.InitAndApply(t, opts)
}
```

Przedstawiony tutaj fragment kodu to całkiem użyteczny test jednostkowy, ponieważ powoduje wydanie poleceń `terraform init` i `terraform apply` oraz kończy się niepowodzeniem, jeśli wykonanie któregośkolwiek z tych poleceń nie zakończy się sukcesem (np. ze względu na problem z kodem Terraform). Można jednak pójść o krok dalej poprzez wykonanie żądań HTTP do wdrożonego mechanizmu równoważenia obciążenia i sprawdzenie, czy udzielona odpowiedź na żądanie zawiera oczekiwane dane. W tym celu trzeba zapewnić sposób na pobranie nazwy domeny wdrożonego mechanizmu równoważenia obciążenia. Na szczęście ta nazwa jest dostępna jako wartość zmiennej danych wyjściowych w przykładzie `alb`:

```
output "alb_dns_name" {
    value     = module.alb.alb_dns_name
    description = "Nazwa domeny mechanizmu równoważenia obciążenia"
}
```

Terratest ma wbudowane metody pomocnicze przeznaczone do odczytywania danych wyjściowych generowanych przez kod Terraform.

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // Powinieneś uaktualnić tę względną ścieżkę dostępu,
        // aby prowadziła do katalogu examples modułu alb!
        TerraformDir: "../examples/alb",
    }

    // Wdrożenie przykładu.
    terraform.InitAndApply(t, opts)
```

```

// Pobranie adresu URL mechanizmu równoważenia obciążenia.
albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
url := fmt.Sprintf("http://%s", albDnsName)
}

```

Funkcja `OutputRequired()` zwraca dane wyjściowe dla danej nazwy. Jeżeli te dane nie istnieją lub mają postać pustego ciągu tekstowego, test kończy się niepowodzeniem. Działanie tego fragmentu kodu polega na utworzeniu adresu URL na podstawie otrzymanych danych wyjściowych — odbywa się to za pomocą funkcji `fmt.Sprintf()` wbudowanej w język Go (nie zapomnij o zaimportowaniu pakietu `fmt`). Następnym krokiem jest wykonanie pewnych żądań HTTP do ustalonego adresu URL.

```

package test

import (
    "fmt"
    "github.com/gruntwork-io/terratest/modules/http-helper"
    "github.com/gruntwork-io/terratest/modules/terraform"
    "testing"
)

func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // Powinieneś uaktualnić tę względną ścieżkę dostępu,
        // aby prowadziła do katalogu examples modułu alb!
        TerraformDir: "../examples/alb",
    }

    // Wdrożenie przykładu.
    terraform.InitAndApply(t, opts)

    // Pobranie adresu URL mechanizmu równoważenia obciążenia.
    albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
    url := fmt.Sprintf("http://%s", albDnsName)

    // Sprawdzenie, czy domyślna akcja ALB działa prawidłowo i zwraca kod stanu 404.

    expectedStatus := 404
    expectedBody := "404: nie znaleziono strony"

    http_helper.HttpGetWithValidation(t, url, expectedStatus, expectedBody)
}

```

Ten kod używa nowego polecenia importującego z Terratest pakiet `http_helper`, co oznacza konieczność ponownego wydania polecenia `dep ensure`, aby pobrać niezbędny pakiet. Metoda `http_helper.HttpGetWithValidation()` spowoduje wykonanie żądania HTTP GET do przekazanego adresu URL i zakończy test niepowodzeniem, jeśli udzielona odpowiedź nie zawiera wskazanego kodu stanu i treści.

Istnieje pewien problem z tym kodem: krótki czas między zakończeniem wykonywania polecenia `terraform apply` i rozpoczęciem działania nazwy DNS mechanizmu równoważenia obciążenia (tzw. rozpropagowaniem tej nazwy). Jeżeli natychmiast zostanie wywołana metoda `http_helper.HttpGetWithValidation()`, istnieje duże niebezpieczeństwo, że test zakończy się niepowodzeniem. Pół minuty lub minutę później mechanizm równoważenia obciążenia powinien już działać prawidłowo. Jak wspomniałem w rozdziale 6., taki rodzaj asynchroniczności i ostatecznej spójności jest

zupełnie normalny w AWS — jest normą w większości systemów rozproszonych — a rozwiązanie polega na ponowieniu próby. Terratest ma do tego celu odpowiednią metodę pomocniczą.

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // Powinieneś uaktualnić tę względną ścieżkę dostępu,
        // aby prowadziła do katalogu examples modułu alb!
        TerraformDir: "../examples/alb",
    }

    // Wdrożenie przykładu.
    terraform.InitAndApply(t, opts)

    // Pobranie adresu URL mechanizmu równoważenia obciążenia.
    albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
    url := fmt.Sprintf("http://%s", albDnsName)

    // Sprawdzenie, czy domyślna akcja ALB działa prawidłowo i zwraca kod stanu 404.

    expectedStatus := 404
    expectedBody := "404: nie znaleziono strony"

    maxRetries := 10
    timeBetweenRetries := 10 * time.Second

    http_helper.HttpGetWithRetry(
        t,
        url,
        expectedStatus,
        expectedBody,
        maxRetries,
        timeBetweenRetries,
    )
}
```

Metoda `http_helper.HttpGetWithRetry()` jest niemalże identyczna jak metoda `http_helper.↪HttpGetWithValidation()` z tą różnicą, że jeśli nie otrzyma oczekiwanego kodu stanu lub treści, spróbuje ponownie wykonać żądania — wywołanie zostanie powtórzone maksymalnie podaną liczbę razy (10) z zachowaniem podanej przerwy między próbami (10 sekund). Jeżeli ostatecznie metoda otrzyma oczekiwaną odpowiedź, test zostanie zaliczony. Natomiast wykonanie maksymalnej liczby prób bez oczekiwanego wyniku doprowadzi do niezaliczenia testu.

Ostatnim zadaniem jest wydanie polecenia `terraform destroy` na końcu testu, aby przeprowadzić operacje porządkowe. Jak pewnie się domyślasz, do tego celu jest dostępna metoda pomocnicza Terratest: `terraform.Destroy()`. Jeżeli jednak wywołasz tę metodę na samym końcu testu, to — o ile jakkolwiek kod wcześniej spowoduje niezaliczenie testu (np. metoda `http_helper.HttpGetWith↪Retry()` zakończy działanie niepowodzeniem ze względu na błędną konfigurację ALB) — kod testu zakończy działanie bez wywołania `terraform.Destroy()` i infrastruktura wdrożona na potrzeby testu nigdy nie zostanie usunięta.

Dlatego też trzeba mieć pewność, że polecenie `terraform.Destroy()` zawsze będzie wykonywane, nawet jeśli test kończy się niepowodzeniem. W wielu językach programowania oznacza to użycie konstrukcji `try-finally` lub `try-ensure`, natomiast w Go używane jest polecenie `defer`, które gwarantuje

wykonanie przekazanego mu kodu po zakończeniu działania funkcji (niezależnie od wyniku jej wywołania).

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // Powinieneś uaktualnić tę względną ścieżkę dostępu,
        // aby prowadziła do katalogu examples modułu alb!
        TerraformDir: "../examples/alb",
    }

    // Usunięcie całej infrastruktury po zakończeniu testu.
    defer terraform.Destroy(t, opts)

    // Wdrożenie przykładu.
    terraform.InitAndApply(t, opts)

    // Pobranie adresu URL mechanizmu równoważenia obciążenia.
    albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
    url := fmt.Sprintf("http://%s", albDnsName)

    // Sprawdzenie, czy domyślna akcja ALB działa prawidłowo i zwraca kod stanu 404.

    expectedStatus := 404
    expectedBody := "404: nie znaleziono strony"

    maxRetries := 10
    timeBetweenRetries := 10 * time.Second

    http_helper.HttpGetWithRetry(
        t,
        url,
        expectedStatus,
        expectedBody,
        maxRetries,
        timeBetweenRetries,
    )
}
```

Zwróć uwagę na dodanie polecenia `defer` dość wcześnie w kodzie, jeszcze przed wywołaniem `terraform.InitAndApply()`, aby zagwarantować, że nic nie spowoduje niezaliczenia testu przed wywołaniem polecenia `defer`, co z kolei uniemożliwiłoby wywołanie `terraform.Destroy()`.

W porządku, test jednostkowy w końcu jest gotowy do wykonania. Skoro powoduje on wdrożenie infrastruktury w AWS, przed wykonaniem testu konieczne jest uwierzytelnienie konta AWS w zwykły sposób (zapoznaj się z opcjami uwierzytelnienia). Z wcześniejszej części rozdziału dowiedziałeś się, że ręczne testy powinny być przeprowadzane za pomocą odizolowanego konta. W przypadku testów zautomatyzowanych ma to jeszcze większe znaczenie, więc zalecam uwierzytelnienie w całkowicie oddzielnym koncie. Wraz ze wzrostem liczby testów zautomatyzowanych każdy zestaw testów może oznaczać tworzenie setek lub tysięcy zasobów, więc odizolowanie testów od wszystkiego pozostałego odgrywa istotną rolę.

Zwykle zachęcam zespoły do przygotowania zupełnie oddzielnego środowiska (np. całkiem oddzielnego konta AWS) przeznaczonego dla testów zautomatyzowanych — odizolowanego nawet od środowiska przeznaczonego dla ręcznego przeprowadzania testów. Dzięki temu będzie można

bezpiecznie usunąć w środowisku testowym wszystkie zasoby starsze niż kilka godzin, opierając się na założeniu, że żaden z testów nie będzie trwał tak długo.

Po uwierzytelnieniu konta AWS można bezpiecznie przystąpić do wykonania testów przez wydanie następującego polecenia:

```
$ go test -v -timeout 30m
```

```
TestAlbExample 2019-05-26T13:29:32+01:00 command.go:53:  
Running command terraform with args [init -upgrade=false]
```

```
(...)
```

```
TestAlbExample 2019-05-26T13:29:33+01:00 command.go:53:  
Running command terraform with args [apply -input=false -lock=false]
```

```
(...)
```

```
TestAlbExample 2019-05-26T13:32:06+01:00 command.go:121:  
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

```
(...)
```

```
TestAlbExample 2019-05-26T13:32:06+01:00 command.go:53:  
Running command terraform with args [output -no-color alb_dns_name]
```

```
(...)
```

```
TestAlbExample 2019-05-26T13:38:32+01:00 http_helper.go:27:  
Making an HTTP GET call to URL  
http://terraform-up-and-running-1892693519.us-east-2.elb.amazonaws.com  
(...)
```

```
TestAlbExample 2019-05-26T13:38:32+01:00 command.go:53:  
Running command terraform with args  
[destroy -auto-approve -input=false -lock=false]  
(...)
```

```
TestAlbExample 2019-05-26T13:39:16+01:00 command.go:121:  
Destroy complete! Resources: 5 destroyed.
```

```
(...)
```

```
PASS  
ok   terraform-up-and-running 229.492s
```

Zwróć uwagę na użycie argumentu `-timeout 30m` wraz z `go test`. Domyślnie język Go nakłada 10-minutowy limit czasu dla testów, po którego upływie następuje zakończenie wykonywania testów. To nie tylko powoduje niezaliczenie testów, ale też uniemożliwia działanie kodu przeprowadzającego operacje porządkowe (np. `terraform destroy`). Wprawdzie wykonanie przedstawionego tutaj testu ALB powinno trwać około 5 minut, ale gdy test wymaga wdrożenia rzeczywistej infrastruktury, bezpieczniejszym rozwiązaniem jest wydłużenie tego czasu. Dzięki temu można uniknąć nagłego zakończenia testów i pozostawienia uruchomionych komponentów infrastruktury.

Ten test spowoduje wygenerowanie dużej ilości danych wyjściowych i jeśli uważnie się z nimi zapoznasz, będziesz w stanie wychwycić wszystkie kluczowe fragmenty testu:

1. Wydanie polecenia `terraform init`.
2. Wydanie polecenia `terraform apply`.
3. Odczyt zmiennych danych wyjściowych za pomocą `terraform output`.
4. Powtarzające się żądania HTTP do ALB.
5. Wydanie polecenia `terraform destroy`.

Wydajność działania nawet nie zbliża się do wydajności testów jednostkowych w języku Ruby, ale w czasie poniżej 5 minut możesz automatycznie sprawdzić, czy moduł alb działa zgodnie z oczekiwaniami. To jest szybkość, z jaką możesz otrzymać informacje o działaniu infrastruktury w AWS i powinieneś zyskać pewność co do prawidłowego działania kodu. Jeżeli popełnisz jakikolwiek błąd w kodzie — np. przypadkowo zmienisz kod stanu na 401 w akcji domyślnej — dość szybko się o tym dowiesz.

```
$ go test -v -timeout 30m
```

```
(...)
```

```
Validation failed for URL
```

```
http://terraform-up-and-running-931760451.us-east-2.elb.amazonaws.com.
```

```
Response status: 401. Response body: 404: nie znaleziono strony.
```

```
(...)
```

```
Sleeping for 10s and will try again.
```

```
(...)
```

```
Validation failed for URL
```

```
http://terraform-up-and-running-h2ezYz-931760451.us-east-2.elb.amazonaws.com.
```

```
Response status: 401. Response body: 404: nie znaleziono strony.
```

```
(...)
```

```
Sleeping for 10s and will try again.
```

```
(...)
```

```
--- FAIL: TestAlbExample (310.19s)
```

```
http_helper.go:94:
```

```
HTTP GET to URL
```

```
http://terraform-up-and-running-931760451.us-east-2.elb.amazonaws.com
```

```
unsuccessful after 10 retries
```

```
FAIL terraform-up-and-running 310.204s
```

Wstrzykiwanie zależności

Zobaczysz teraz, jak można dodać test jednostkowy dla nieco bardziej skomplikowanego kodu. Raz jeszcze powrócimy do przykładu serwera WWW w języku Ruby i sprawdzimy, co się stanie, gdy trzeba będzie dodać nowy punkt końcowy `/web-service` wykonujący wywołania HTTP do zewnętrznej zależności.

```

class Handlers
  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Witaj, świecie']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    when "/web-service"
      # Nowy punkt końcowy, który wywołuje usługę sieciową.
      uri = URI("http://www.example.org")
      response = Net::HTTP.get_response(uri)
      [response.code.to_i, response['Content-Type'], response.body]
    else
      [404, 'text/plain', 'Nie znaleziono']
    end
  end
end
end

```

Uaktualniona klasa obsługuje teraz adres URL `/web-service` przez wywołania nowej metody o nazwie `web_service`, która wykonuje żądanie HTTP GET do *example.org* i zapewnia proxy dla udzielonej odpowiedzi. Po użyciu narzędzia `curl` w celu wykonania żądania do tego punktu końcowego w odpowiedzi zostanie przekazany następujący dokument:

```
$ curl localhost:8000/web-service
```

```

<!doctype html>
<html>
<head>
  <title>Example Domain</title>
  <!-- (...) -->
</head>
<body>
<div>
  <h1>Example Domain</h1>
  <p>
    This domain is established to be used for illustrative
    examples in documents. You may use this domain in
    examples without prior coordination or asking for permission.
  </p>
  <!-- (...) -->
</div>
</body>
</html>

```

W jaki sposób można dodać test jednostkowy dla nowej metody? Jeżeli spróbujesz przetestować kod w jego obecnej postaci, test jednostkowy będzie musiał zmierzyć się z zachowaniem zewnętrznej zależności (w omawianym przykładzie *example.org*). Takie rozwiązanie ma pewne wady:

- Jeżeli zależność będzie niedostępna, test zakończy się niepowodzeniem, nawet jeśli kod nie zawiera żadnego błędu.
- Jeżeli sposób działania tej zależności będzie ulegał okresowym zmianom (np. zwrot innej treści odpowiedzi), test od czasu do czasu będzie niezaliczony. To oznacza konieczność nieustannego uaktualniania kodu testu, nawet jeśli nie ma żadnych problemów z implementacją.

- Jeżeli wydajność działania zależności jest mała, to wydajność działania testów również będzie niska, co niweluje jedną z podstawowych korzyści testów jednostkowych, czyli szybkie dostarczanie informacji o poprawności implementacji.
- Jeżeli test będzie miał sprawdzić zachowanie kodu w różnych przypadkach skrajnych na podstawie sposobu działania zależności (np. obsługa przekierowań przez kod), nie będzie można tego zrobić bez kontroli nad zewnętrzną zależnością.

Wprawdzie praca z rzeczywistymi zależnościami może mieć sens w przypadku testów integracji lub typu E2E, ale w testach jednostkowych, o ile to możliwe, należy dążyć do minimalizacji zewnętrznych zależności. Typowa strategia polega na zastosowaniu *wstrzykiwania zależności*, co pozwala na przekazanie (inaczej: wstrzyknięcie) zewnętrznej zależności dla kodu zamiast zdefiniowania jej na stałe w danym kodzie.

Przykładowo klasa `Handlers` nie powinna być teraz zmuszona do zajmowania się wszystkimi szczegółami dotyczącymi sposobu wywołania usługi sieciowej. Zamiast tego tę logikę należy umieścić w oddzielnej klasie `WebService`.

```
class WebService
  def initialize(url)
    @uri = URI(url)
  end
  def proxy
    response = Net::HTTP.get_response(@uri)
    [response.code.to_i, response['Content-Type'], response.body]
  end
end
```

Ta klasa pobiera dane wejściowe w postaci adresu URL i udostępnia metodę `proxy()` zapewniającą proxy dla odpowiedzi HTTP GET udzielonej po wykonaniu żądania do tego adresu URL. Teraz można uaktualnić klasę `Handlers` w taki sposób, aby wykorzystała egzemplarz `WebService` jako dane wejściowe i zastosowała go w metodzie `web_service`.

```
class Handlers
  def initialize(web_service)
    @web_service = web_service
  end
  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Witaj, świecie']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    when "/web-service"
      # Nowy punkt końcowy, który wywołuje usługę sieciową.
      @web_service.proxy
    else
      [404, 'text/plain', 'Nie znaleziono']
    end
  end
end
```

Następnie w kodzie implementacji można wstrzyknąć egzemplarz `WebService` odpowiedzialny za wykonywanie żądań HTTP do *example.org*.

```

class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    web_service = WebService.new("http://www.example.org")
    handlers = Handlers.new(web_service)

    status_code, content_type, body = handlers.handle(request.path)

    response.status = status_code
    response['Content-Type'] = content_type
    response.body = body
  end
end

```

W kodzie testu można utworzyć imitację klasy `WebService` pozwalającą na wskazanie imitacji udzielanej odpowiedzi.

```

class MockWebService
  def initialize(response)
    @response = response
  end
  def proxy
    @response
  end
end

```

Teraz można utworzyć egzemplarz klasy `MockWebService` i wstrzyknąć ją do klasy `Handlers` w testach jednostkowych.

```

def test_unit_web_service
  expected_status = 200
  expected_content_type = 'text/html'
  expected_body = 'mock example.org'
  mock_response = [expected_status, expected_content_type, expected_body]

  mock_web_service = MockWebService.new(mock_response)
  handlers = Handlers.new(mock_web_service)

  status_code, content_type, body = handlers.handle("/web-service")
  assert_equal(expected_status, status_code)
  assert_equal(expected_content_type, content_type)
  assert_equal(expected_body, body)
end

```

Ponownie wykonaj testy, aby się upewnić, że wszystko nadal działa zgodnie z oczekiwaniami.

```

$ ruby web-server-test.rb
Loaded suite web-server-test
Started
...
Finished in 0.000645 seconds.
-----
4 tests, 12 assertions, 0 failures, 0 errors
100% passed
-----

```

Fantastycznie. Użycie mechanizmu wstrzykiwania zależności w celu zminimalizowania zależności zewnętrznych pozwoliło na utworzenie szybkich i niezawodnych testów, a także na sprawdzenie

różnych przypadków skrajnych. Skoro trzy dodane wcześniej testy wciąż są zaliczone, masz pewność, że przeprowadzona refaktoryzacja niczego nie uszkodziła.

Powracamy teraz do kodu Terraform, aby zobaczyć, jak mechanizm wstrzykiwania zależności działa wraz z modułami Terraform. Na pierwszy ogień pójdzie moduł `hello-world-app`. Jeżeli jeszcze tego nie zrobiłeś, pierwszym krokiem jest utworzenie łatwego do wdrożenia przykładu, który należy umieścić w katalogu *examples*.

```
provider "aws" {
  region = "us-east-2"

  # Zezwolenie na użycie dowolnej wersji 2.x dostawcy AWS.
  version = "~> 2.0"
}

module "hello_world_app" {
  source = ".././../modules/services/hello-world-app"

  server_text = "Witaj, świecie"
  environment = "example"

  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key    = "examples/terraform.tfstate"

  instance_type    = "t2.micro"
  min_size         = 2
  max_size         = 2
  enable_autoscaling = false
}
```

Problem związany z zależnościami od razu staje się widoczny: w module `hello-world-app` zostało przyjęte założenie o wdrożeniu modułu `mysql`. Ponadto wymagane jest przekazanie szczegółowych informacji o kubełku S3, w którym moduł `mysql` przechowuje informacje o stanie za pomocą argumentów `db_remote_state_bucket` i `db_remote_state_key`. Celem jest tutaj utworzenie testu jednostkowego dla modułu `hello-world-app` i choć przygotowanie czystego testu jednostkowego bez żadnych zależności zewnętrznych nie jest w Terraform możliwe, wciąż dobrym podejściem jest minimalizacja zależności zewnętrznych, gdy tylko to możliwe.

Jednym z pierwszych kroków podczas minimalizacji zależności jest wyraźne zdefiniowanie zależności w module. Oparta na nazwach plików konwencja, którą warto zastosować, polega na przeniesieniu do oddzielnego pliku *dependencies.tf* nazw wszystkich źródeł danych i zasobów przedstawiających zależności zewnętrzne. Spójrz na przykładowy kod w pliku *modules/services/hello-world-app/dependencies.tf*:

```
data "terraform_remote_state" "db" {
  backend = "s3"

  config = {
    bucket = var.db_remote_state_bucket
    key    = var.db_remote_state_key
    region = "us-east-2"
  }
}

data "aws_vpc" "default" {
  default = true
}
```

```

}

data "aws_subnet_ids" "default" {
  vpc_id = data.aws_vpc.default.id
}

```

Ta konwencja ułatwia użytkownikom kodu szybkie ustalenie, jakie zależności zewnętrzne są wymagane do działania danego kodu. W przypadku modułu `hello-world-app` wyraźnie widać, że tymi zależnościami są baza danych, VPC i podsieci. Powstaje więc pytanie, w jaki sposób można wstrzyknąć te zależności z zewnątrz, aby można było je zastąpić w trakcie testu. Odpowiedź już doskonale znasz: za pomocą zmiennych danych wejściowych.

Dla każdej zależności trzeba w pliku `modules/services/hello-world-app/variables.tf` dodać nową zmienną danych wejściowych.

```

variable "vpc_id" {
  description = "Identyfikator grupy VPC, w której będzie przeprowadzone wdrożenie"
  type        = string
  default     = null
}

variable "subnet_ids" {
  description = "Identyfikatory podsieci, w których będzie przeprowadzone wdrożenie"
  type        = list(string)
  default     = null
}

variable "mysql_config" {
  description = "Konfiguracja bazy danych MySQL"
  type        = object({
    address = string
    port    = number
  })
  default    = null
}

```

W tym momencie mamy zmienne danych wejściowych dla identyfikatorów sieci VPC, podsieci i konfiguracji bazy danych MySQL. Każda z tych zmiennych zawiera atrybut `default`, więc to są *zmienne opcjonalne*, którym użytkownik może przypisać pewną wartość lub które może zupełnie pominąć i tym samym wykorzystać wartość domyślną (`default`). Z użytą tutaj wartością domyślną, `null`, jeszcze się nie spotkałeś. Jeżeli wartością argumentu `default` będzie *pusta wartość*, np. w postaci pustego ciągu tekstowego dla `vpc_id` lub pustej listy dla `subnet_ids`, to nie będzie możliwości odróżnienia między (a) pustą wartością zdefiniowaną jako domyślna i (b) sytuacją, w której użytkownik modułu celowo przekazał pustą wartość przeznaczoną do użycia. W takich przypadkach użyteczne jest stosowanie wartości `null`, ponieważ została ona zaprojektowana specjalnie do wskazania, że zmiennej nie została przypisana żadna wartość i użytkownik chce wykorzystać rozwiązanie awaryjne w postaci zachowania domyślnego.

Zwróć uwagę na to, że `mysql_config` używa konstruktora typu `object` w celu utworzenia typu zagnieżdżonego wraz z kluczami `address` i `port`. Ten typ jest specjalnie przeznaczony do dopasowania typów danych wyjściowych modułu `mysql`.

```

output "address" {
  value = aws_db_instance.example.address
}

```



```

    description = "Nawiązanie połączenia z bazą danych w tym punkcie końcowym"
  }

  output "port" {
    value      = aws_db_instance.example.port
    description = "Numer portu, na którym nasłuchuje baza danych"
  }

```

Jedną z zalet takiego rozwiązania jest to, że tuż po zakończeniu refaktoryzacji kodu będzie można w następujący sposób używać modułów `hello-world-app` i `mysql`:

```

module "hello_world_app" {
  source = "../../modules/services/hello-world-app"

  server_text      = "Witaj, świecie"
  environment      = "example"

  # Bezpośrednie przekazanie wszystkich danych wyjściowych modułu mysql!
  mysql_config = module.mysql

  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
}

module "mysql" {
  source = "../../modules/data-stores/mysql"

  db_name      = var.db_name
  db_username  = var.db_username
  db_password  = var.db_password
}

```

Skoro `type of mysql_config` dopasowuje typ danych wyjściowych modułu `mysql`, można je wszystkie przekazać w jednym wierszu. Jeżeli typ zostanie kiedykolwiek zmieniony i nie będzie dłużej dopasowywany, Terraform od razu wygeneruje komunikat błędu i będzie wiadomo o konieczności uaktualnienia typu. To pokazuje nie tylko łączenie funkcji, ale również zapewnienie bezpieczeństwa funkcji.

Zanim rozwiązanie będzie mogło funkcjonować, najpierw trzeba dokończyć refaktoryzację kodu. Ponieważ konfiguracja MySQL może być przekazana jako dane wejściowe, to oznacza, że zmienne `db_remote_state_bucket` i `db_remote_state_key` powinny być już w użyciu, więc ich wartości domyślne należy zdefiniować jako `null`.

```

variable "db_remote_state_bucket" {
  description = "Nazwa kubejka S3 dla bazy danych informacji o stanie Terraform"
  type        = string
  default     = null
}

variable "db_remote_state_key" {
  description = "Ścieżka dostępu w kubejku S3 dla bazy danych informacji o stanie Terraform"
  type        = string
  default     = null
}

```

Kolejnym krokiem jest użycie parametru `count` w celu opcjonalnego utworzenia trzech źródeł danych w pliku `modules/services/hello-world-app/dependencies.tf` na podstawie tego, czy odpowiednia zmienna danych wejściowych ma przypisaną wartość `null`.

```
data "terraform_remote_state" "db" {
  count = var.mysql_config == null ? 1 : 0

  backend = "s3"

  config = {
    bucket = var.db_remote_state_bucket
    key    = var.db_remote_state_key
    region = "us-east-2"
  }
}

data "aws_vpc" "default" {
  count = var.vpc_id == null ? 1 : 0
  default = true
}

data "aws_subnet_ids" "default" {
  count = var.subnet_ids == null ? 1 : 0
  vpc_id = data.aws_vpc.default.id
}
```

Teraz można uaktualnić wszelkie odwołania do tych źródeł danych w celu warunkowego użycia zmiennej danych wejściowych lub źródła danych. Przechylimy je w postaci wartości lokalnych.

```
locals {
  mysql_config = (
    var.mysql_config == null
      ? data.terraform_remote_state.db[0].outputs
      : var.mysql_config
  )

  vpc_id = (
    var.vpc_id == null
      ? data.aws_vpc.default[0].id
      : var.vpc_id
  )

  subnet_ids = (
    var.subnet_ids == null
      ? data.aws_subnet_ids.default[0].ids
      : var.subnet_ids
  )
}
```

Skoro źródła danych używają parametrów `count`, to mają teraz postać tablic. Dlatego też w trakcie odwoływania się do nich trzeba skorzystać ze składni wyszukiwania w tablicy, np. `[0]`. Przeanalizuj kod i każde znalezione odwołania do jednego ze wspomnianych źródeł danych zastąp odwołaniem do odpowiedniej wartości lokalnej. Rozpocznij od uaktualnienia źródła danych `aws_subnet_ids` i użycia `local.vpc_id`.

```
data "aws_subnet_ids" "default" {
  count = var.subnet_ids == null ? 1 : 0
```

```

    vpc_id = local.vpc_id
}

```

Następnie zmodyfikuj w taki sposób, aby parametry `subnet_ids` w modułach `asg` i `alb` korzystały z `local.subnet_ids`.

```

module "asg" {
    source = "../cluster/asg-rolling-deploy"

    cluster_name = "hello-world-${var.environment}"
    ami          = var.ami
    user_data    = data.template_file.user_data.rendered
    instance_type = var.instance_type

    min_size      = var.min_size
    max_size      = var.max_size
    enable_autoscaling = var.enable_autoscaling

    subnet_ids    = local.subnet_ids
    target_group_arns = [aws_lb_target_group.asg.arn]
    health_check_type = "ELB"

    custom_tags = var.custom_tags
}

module "alb" {
    source = "../networking/alb"

    alb_name = "hello-world-${var.environment}"
    subnet_ids = local.subnet_ids
}

```

Uaktualnij zmienne `db_address` i `db_port` w zasobie `user_data`, aby używały `local.mysql_config`:

```

data "template_file" "user_data" {
    template = file("${path.module}/user-data.sh")

    vars = {
        server_port = var.server_port
        db_address  = local.mysql_config.address
        db_port     = local.mysql_config.port
        server_text = var.server_text
    }
}

```

Teraz można uaktualnić parametr `vpc_id` grupy `aws_lb_target_group`, aby używał wartości `local.vpc_id`.

```

resource "aws_lb_target_group" "asg" {
    name     = "hello-world-${var.environment}"
    port     = var.server_port
    protocol = "HTTP"
    vpc_id   = local.vpc_id

    health_check {
        path          = "/"
        protocol      = "HTTP"
        matcher       = "200"
        interval      = 15
    }
}

```

```

        timeout          = 3
        healthy_threshold = 2
        unhealthy_threshold = 2
    }
}

```

Po przeprowadzeniu tych uaktualnień można wstrzyknąć identyfikatory sieci VPC, podsieci i (lub) parametry konfiguracyjne MySQL do modułu `hello-world-app` lub też całkowicie pominąć te parametry, aby moduł wykorzystał odpowiednie źródła danych w celu samodzielnego pobrania wartości. Uaktualnimy teraz przykładową aplikację wyświetlającą komunikat typu *Witaj, świecie* i pozwolimy na wstrzyknięcie konfiguracji MySQL, choć pominiemy parametry identyfikatorów sieci VPC i podsieci, ponieważ ich wartości domyślne są wystarczająco dobre podczas testów. Do pliku *examples/hello-world-app/variables.tf* dodaj nową zmienną danych wejściowych.

```

variable "mysql_config" {
    description = "Konfiguracja bazy danych MySQL"

    type = object({
        address = string
        port    = number
    })

    default = {
        address = "mock-mysql-address"
        port    = 12345
    }
}

```

Przekaż tę zmienną do modułu `hello-world-app` w pliku *examples/hello-world-app/main.tf*:

```

module "hello_world_app" {
    source = "../../modules/services/hello-world-app"

    server_text = "Witaj, świecie"
    environment = "example"

    mysql_config = var.mysql_config

    instance_type    = "t2.micro"
    min_size         = 2
    max_size         = 2
    enable_autoscaling = false
}

```

Teraz zmiennej `mysql_config` w teście jednostkowym można przypisać dowolną wartość. W pliku *test/hello_world_app_example_test.go* utwórz test jednostkowy składający się z następującego kodu:

```

func TestHelloWorldAppExample(t *testing.T) {
    opts := &terraform.Options{
        // Powinieneś uaktualnić tę względną ścieżkę dostępu,
        // aby prowadziła do katalogu examples modułu hello-world-app!
        TerraformDir: "../examples/hello-world-app/standalone",
    }

    // Usunięcie całej infrastruktury po zakończeniu testu.
    defer terraform.Destroy(t, opts)
    terraform.InitAndApply(t, opts)
}

```

```

albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
url := fmt.Sprintf("http://%s", albDnsName)

expectedStatus := 200
expectedBody := "Witaj, świecie"

maxRetries := 10
timeBetweenRetries := 10 * time.Second

http_helper.HttpGetWithRetry(
    t,
    url,
    expectedStatus,
    expectedBody,
    maxRetries,
    timeBetweenRetries,
)
}

```

Ten kod jest niemalże identyczny z testem jednostkowym dla przykładu modułu alb. Jedyna różnica polega na tym, że ustawienie `TerraformDir` wskazuje przykład `hello-world-app` (upewnij się o podaniu odpowiedniej ścieżki dostępu dla Twojego systemu plików), a oczekiwaną odpowiedzią w przypadku ALB jest 200 OK wraz z treścią w postaci komunikatu *Witaj, świecie*. Tylko jeden nowy element trzeba dodać do omawianego testu — zmienną `mysql_config`.

```

opts := &terraform.Options{
    // Powinieneś uaktualnić tę względną ścieżkę dostępu,
    // aby prowadziła do katalogu examples modułu hello-world-app!
    TerraformDir: "../examples/hello-world-app/standalone",

    Vars: map[string]interface{}{
        "mysql_config": map[string]interface{}{
            "address": "mock-value-for-test",
            "port": 3306,
        },
    },
}

```

Parametr `Vars` w `terraform.Options` pozwala na definiowanie zmiennych w kodzie Terraform. Ten kod przekazuje pewne dane imitacji dla zmiennej `mysql_config`. Ewentualnie zmiennej można przypisać dowolną wartość, np. podczas testu uruchomić małą i działającą w pamięci bazę danych i przypisać argumentowi `address` adres IP tej bazy danych.

Uruchom nowy test za pomocą polecenia `go test` i użyj argumentu `-run` do uruchomienia tylko *tego* testu (w przeciwnym razie domyślne zachowanie Go polega na wykonaniu wszystkich testów zdefiniowanych w katalogu bieżącym, czyli także utworzonych wcześniej testów dla przykładu ALB).

```

$ go test -v -timeout 30m -run TestHelloWorldAppExample

(...)

PASS
ok   terraform-up-and-running 204.113s

```

Jeżeli wszystko przebiegnie bez problemów, test spowoduje wydanie polecenia `terraform apply`, powtarzające się wykonywanie żądań HTTP do mechanizmu równoważenia obciążenia, a po otrzymaniu oczekiwanej odpowiedzi nastąpi wydanie polecenia `terraform destroy` i przeprowadzenie operacji porządkowych. Wykonanie testu powinno zabrać jedynie kilka minut. W ten sposób przygotowałeś rozsądny test jednostkowy dla aplikacji wyświetlającej komunikat typu *Witaj, świecie*.

Jednoczesne wykonywanie testów

W poprzednim punkcie użyłeś argumentu `-run` do wykonania pojedynczego testu po wydaniu polecenia `go test`. Jeżeli pominiesz wymieniony argument, Go wykona wszystkie testy, sekwencyjnie. Wprawdzie 4 – 5 minut na wykonanie pojedynczego testu podczas sprawdzania kodu infrastruktury nie jest złym wynikiem, ale jeśli masz dziesiątki testów wykonywanych sekwencyjnie, cała operacja może zabrać długie godziny. Aby skrócić czas oczekiwania na wynik testów, można spróbować jednocześnie uruchomić jak największą liczbę testów.

W celu nakazania Go jednoczesnego wykonywania wielu testów jedyną zmianą konieczną do wprowadzenia jest dodanie wywołania `t.Parallel()` na początku każdego testu. Spójrz na zmodyfikowany plik `test/hello_world_app_example_test.go`:

```
func TestHelloWorldAppExample(t *testing.T) {
    t.Parallel()

    opts := &terraform.Options{
        // Powinieneś uaktualnić tę względną ścieżkę dostępu,
        // aby prowadziła do katalogu examples modułu hello-world-app!
        TerraformDir: "../examples/hello-world-app/standalone",
        Vars: map[string]interface{}{
            "mysql_config": map[string]interface{}{
                "address": "mock-value-for-test",
                "port": 3306,
            },
        },
    }
    // (...)
}
```

Zmodyfikowany plik `test/alb_example_test.go` przedstawia się następująco:

```
func TestAlbExample(t *testing.T) {
    t.Parallel()

    opts := &terraform.Options{
        // Powinieneś uaktualnić tę względną ścieżkę dostępu,
        // aby prowadziła do katalogu examples modułu alb!
        TerraformDir: "../examples/alb",
    }
    // (...)
}
```

Jeżeli teraz wydasz polecenie `go test`, oba zestawy testów zostaną uruchomione jednocześnie. Takie rozwiązanie ma jednak pewną wadę: niektóre zasoby tworzone przez te testy — np. grupa ASG, grupa bezpieczeństwa i ALB — używają tych samych nazw, co spowoduje niepowodzenie testów ze względu na konflikty nazw. Nawet jeśli nie zdecydujesz się na wywołanie `t.Parallel()` w testach,

a inni członkowie zespołu w tym samym momencie uruchomią testy lub jeśli testy są wykonywane za pomocą serwera CI, to konflikty nazw są praktycznie nieuniknione.

To prowadzi nas do *czwartej reguły związanej z testowaniem*: konieczne jest stosowanie przestrzeni nazw dla wszystkich zasobów.

Moduły i przykłady powinny być opracowywane w taki sposób, aby każdy zasób był (opcjonalnie) konfigurowalny. W przykładzie `alb` to może oznaczać zapewnienie możliwości skonfigurowania nazwy ALB. Do pliku `examples/alb/variables.tf` dodaj nową zmienną danych wejściowych wraz z rozsądną wartością domyślną.

```
variable "alb_name" {
  description = "Nazwa modułu ALB i jego wszystkich zasobów"
  type        = string
  default     = "terraform-up-and-running"
}
```

Teraz tę wartość można przekazać do modułu `alb` w pliku `examples/alb/main.tf`:

```
module "alb" {
  source = "../modules/networking/alb"

  alb_name = var.alb_name
  subnet_ids = data.aws_subnet_ids.default.ids
}
```

Kolejnym krokiem jest przypisanie unikatowej wartości zmiennej w pliku `test/alb_example_test.go`:

```
package test

import (
    "fmt"
    "github.com/gruntwork-io/terratest/modules/http-helper"
    "github.com/gruntwork-io/terratest/modules/random"
    "github.com/gruntwork-io/terratest/modules/terraform"
    "testing"
    "time"
)

func TestAlbExample(t *testing.T) {
    t.Parallel()

    opts := &terraform.Options{
        // Powinieneś uaktualnić tę względną ścieżkę dostępu,
        // aby prowadziła do katalogu examples modułu alb!
        TerraformDir: "../examples/alb",

        Vars: map[string]interface{}{
            "alb_name": fmt.Sprintf("test-%s", random.UniqueId()),
        },
    }
    // (...)
}
```

(Zwróć uwagę na użycie nowej metody pomocniczej z pakietu `random` `Terratest`, to oznacza konieczność ponownego wydania polecenia `dep ensure`).

Ten fragment kodu powoduje przypisanie zmiennej `alb_name` wartości `test-<LOSOWY_ID>`, gdzie `LOSOWY_ID` to losowo wybrany unikatowy identyfikator wygenerowany przez metodę pomocniczą `random.UniqueId()` w Terratest. Działanie tej metody pomocniczej polega na zwróceniu losowo wybranego sześciornakowego ciągu tekstowego typu *base-62*. Idea polega na zastosowaniu krótkiego identyfikatora, który będzie można dodawać do nazw większości zasobów bez wywoływania problemów związanych z ograniczeniem długości nazwy. Ta wartość jest na tyle losowa, aby praktycznie wyeliminować niebezpieczeństwo powstawania konfliktów (62^6 daje w przybliżeniu ponad 56 miliardów kombinacji). W ten sposób masz gwarancję, że po jednoczesnym uruchomieniu ogromnej liczby testów ALB nie musisz się martwić o niebezpieczeństwo wystąpienia konfliktów nazw.

Podobną zmianę trzeba wprowadzić w przykładzie aplikacji typu Witaj, świecie. Zacznij od dodania nowej zmiennej danych wejściowych w pliku *examples/hello-world-app/variables.tf*:

```
variable "environment" {
  description = "Nazwa środowiska, w którym będzie przeprowadzone wdrożenie"
  type        = string
  default     = "example"
}
```

Następnie przekaz tę zmienną do modułu `hello-world-app`.

```
module "hello_world_app" {
  source = "../../modules/services/hello-world-app"

  server_text = "Witaj, świecie"

  environment = var.environment

  mysql_config = var.mysql_config

  instance_type    = "t2.micro"
  min_size         = 2
  max_size         = 2
  enable_autoscaling = false
}
```

Teraz w pliku *hello_world_app_example_test.go* można przypisać zmiennej `environment` wartość zawierającą wartość wygenerowaną przez `random.UniqueId()`.

```
func TestHelloWorldAppExample(t *testing.T) {
  t.Parallel()

  opts := &terraform.Options{
    // Powinienesz uaktualnić tę względną ścieżkę dostępu,
    // aby prowadziła do katalogu examples modułu hello-world-app!
    TerraformDir: "../examples/hello-world-app/standalone",

    Vars: map[string]interface{}{
      "mysql_config": map[string]interface{}{
        "address": "mock-value-for-test",
        "port": 3306,
      },
      "environment": fmt.Sprintf("test-%s", random.UniqueId()),
    },
  },
  // (...)
}
```


Po wprowadzeniu przedstawionych zmian można już bezpiecznie uruchomić jednocześnie wszystkie testy.

```
$ go test -v -timeout 30m
```

```
TestAlbExample 2019-05-26T17:57:21+01:00 (...)  
TestHelloWorldAppExample 2019-05-26T17:57:21+01:00 (...)  
TestAlbExample 2019-05-26T17:57:21+01:00 (...)  
TestHelloWorldAppExample 2019-05-26T17:57:21+01:00 (...)  
TestHelloWorldAppExample 2019-05-26T17:57:21+01:00 (...)
```

```
(...)
```

```
PASS
```

```
ok  terraform-up-and-running 216.090s
```

Zobaczysz oba testy uruchomione w tym samym czasie, więc wykonanie całego zestawu zabierze mniej więcej tyle czasu, ile potrzeba na przeprowadzenie najwolniejszego z testów, a nie ile wynosi łączny czas na wykonanie wszystkich testów.



Jednoczesne uruchamianie testów w tym samym katalogu

Oto inny typ równoległości, który należy wziąć pod uwagę: co się stanie, jeśli spróbujesz jednoczesnego uruchomienia wielu testów zautomatyzowanych w tym samym katalogu Terraform. Być może będziesz chciał przeprowadzać kilka różnych testów dla *examples/hello-world-app*, przy czym każdy test będzie sprawdzał odmienne wartości zmiennych danych wejściowych przed wydaniem polecenia `terraform apply`. Jeżeli spróbujesz takiego rozwiązania, napotkasz problem: testy spowodują konflikt, ponieważ będą wydawały polecenia `terraform init`, co z kolei doprowadzi do nadpisywania katalogu *.terraform* i plików stanu Terraform.

Jeżeli chcesz jednocześnie wykonać wiele testów zdefiniowanych w tym samym katalogu, najłatwiejszym rozwiązaniem jest skopiowanie poszczególnych testów do unikalnych katalogów tymczasowych i wykonanie testów w tych katalogach, aby uniknąć potencjalnych konfliktów. Terratest oczywiście ma wbudowaną metodę pomocniczą, która może to zrobić za Ciebie. Ta metoda gwarantuje także poprawność działania względnych ścieżek dostępu w modułach Terraform: zapoznaj się z metodą `test_structure.CopyTerraformFolderToTemp()` i jej dokumentacją.

Testy integracji

Po przygotowaniu testów jednostkowych można przejść do testów integracji. Także w tym przypadku dobrym pomysłem jest rozpoczęcie pracy od przykładu serwera WWW w języku Ruby, aby przygotować pewne rozwiązanie, które później będzie można przenieść do Terraform. Aby przeprowadzić testy integracji w kodzie serwera WWW utworzonego w języku Ruby, konieczne jest wykonanie następujących zadań:

1. Uruchomienie serwera WWW w komputerze lokalnym, aby nasłuchiwał na danym porcie.
2. Wykonywanie żądań HTTP do serwera WWW.
3. Sprawdzenie, czy otrzymane odpowiedzi są zgodne z oczekiwaniami.

Przystępujemy do utworzenia w pliku *web-server-test.rb* metody pomocniczej implementującej wymienione wcześniej kroki.

```
def do_integration_test(path, check_response)
  port = 8000
  server = WEBrick::HTTPServer.new :Port => port
  server.mount '/', WebServer

  begin
    # Uruchomienie serwera WWW w oddzielnym wątku,
    # aby nie zablokował testu.
    thread = Thread.new do
      server.start
    end

    # Wykonywanie żądania HTTP do serwera WWW,
    # który działa pod podanym adresem.
    uri = URI("http://localhost:#{port}#{path}")
    response = Net::HTTP.get_response(uri)

    # Użycie podanej funkcji lambda check_response()
    # do sprawdzenia udzielonej odpowiedzi.
    check_response.call(response)
  ensure
    # Na zakończenie testu następuje
    # zamknięcie serwera i wątku.
    server.shutdown
    thread.join
  end
end
```

Metoda `do_integration_test()` konfiguruje serwer WWW do nasłuchiwania na porcie 8000, uruchamia go w wątku działającym w tle (aby serwer WWW nie blokował wykonywania testów), wykonuje żądania HTTP GET do podanego adresu, przekazuje do funkcji `check_response()` otrzymaną odpowiedź HTTP w celu jej sprawdzenia, a na końcu testu kończy działanie serwera WWW. Spójrz, jak można wykorzystać tę metodę do utworzenia testu integracji dla punktu końcowego/serwera WWW:

```
def test_integration_hello
  do_integration_test('/', lambda { |response|
    assert_equal(200, response.code.to_i)
    assert_equal('text/plain', response['Content-Type'])
    assert_equal('Witaj, świecie', response.body)
  })
end
```

Ta metoda wywołuje metodę `do_integration_test()` wraz ze ścieżką dostępu `/` i przekazuje jej funkcję lambda (czyli praktycznie funkcję jednowierszową), która sprawdza, czy udzielona odpowiedź zawiera kod stanu 200 OK i treść Witaj, świecie. Testy integracji dla innych punktów końcowych są tworzone analogicznie, choć test dla punktu `/web-service` przeprowadza mniej dokładne sprawdzenie (np. zawiera wywołanie `assert_include()` zamiast `assert_equal()`), aby zminimalizować potencjalne zakłócenia na skutek zmian wprowadzonych w *example.org*.

```
def test_integration_api
  do_integration_test('/api', lambda { |response|
    assert_equal(201, response.code.to_i)
    assert_equal('application/json', response['Content-Type'])
  })
end
```

```

    assert_equal('{"foo":"bar"}', response.body)
  })
end

def test_integration_404
  do_integration_test('/invalid-path', lambda { |response|
    assert_equal(404, response.code.to_i)
    assert_equal('text/plain', response['Content-Type'])
    assert_equal('Nie znaleziono', response.body)
  })
end

def test_integration_web_service
  do_integration_test('/web-service', lambda { |response|
    assert_equal(200, response.code.to_i)
    assert_include(response['Content-Type'], 'text/html')
    assert_include(response.body, 'Example Domain')
  })
end

```

Oto wynik uruchomienia wszystkich testów:

```
$ ruby web-server-test.rb
```

```
(...)
```

```
Finished in 0.221561 seconds.
```

```
-----
8 tests, 24 assertions, 0 failures, 0 errors
100% passed
-----
```

Zwróć uwagę na to, że wcześniej wykonanie zestawu zawierającego jedynie testy jednostkowe zabrało 0,000572 sekundy, natomiast po dodaniu testów integracji ten czas wydłużył się do 0,221561 sekundy, co oznacza 387 razy wolniej. Oczywiście ułamek sekundy to nadal bardzo szybko, ale ten wynik jest możliwy do osiągnięcia tylko dlatego, że kod serwera WWW w języku Ruby celowo jest minimalny i nie wykonuje zbyt wielu zadań. Najważniejsze nie są tutaj konkretne wartości, ale ogólny trend: testy integracji są zwykle znacznie wolniejsze niż testy jednostkowe. Do tego zagadnienia jeszcze powrócę w dalszej części rozdziału.

Przechodzimy teraz do testów integracji dla kodu Terraform. Jeżeli za „jednostkę” w Terraform można uznać pojedynczy moduł, test integracji powinien sprawdzać, jak wiele jednostek ze sobą współpracuje, gdy zachodzi potrzeba wdrożenia wielu modułów, oraz czy działają one prawidłowo. Wcześniej wdrożyłeś przykładową aplikację wyświetlającą komunikat typu *Witaj, świecie* wraz z imitacją danych zamiast z prawdziwą bazą danych MySQL. W przypadku testu integracji zostanie faktycznie wdrożony moduł MySQL, aby mieć pewność o prawidłowej integracji z aplikacją typu *Witaj, świecie*. Niezbędny kod powinieneś mieć już zdefiniowany w plikach *live/stage/data-stores/mysql* i *live/stage/services/hello-world-app*.

Oczywiście, jak już wspomniałem w rozdziale, wszystkie testy zautomatyzowane powinny działać w odizolowanym koncie AWS. Dlatego też podczas testowania kodu przeznaczanego dla środowiska roboczego powinieneś uwierzytelnić odizolowane środowisko testowe, a następnie w nim przeprowadzić te testy. Jeżeli moduły mają zdefiniowane na stałe jakiekolwiek dane przeznaczone dla środowiska

roboczego, to jest odpowiednia chwila na zapewnienie konfiguracji tych wartości, aby umożliwić wstrzykiwanie wartości przyjaznych testom. W szczególności nowa zmienna danych wejściowych `db_name` w `live/stage/data-stores/mysql/variables.tf` udostępnia nazwę bazy danych.

```
variable "db_name" {
  description = "Nazwa do użycia dla bazy danych"
  type        = string
  default     = "example_database_stage"
}
```

Przekaż tę wartość do modułu `mysql` w `live/stage/data-stores/mysql/main.tf`:

```
module "mysql" {
  source = "../../../../../modules/data-stores/mysql"

  db_name      = var.db_name
  db_username  = var.db_username
  db_password  = var.db_password
}
```

Przystępujemy do utworzenia szkieletu testu integracji w pliku `test/hello_world_integration_test.go`, a szczególży implementacji umieścimy w nim później.

```
// Te wartości zastap odpowiednimi dla Twoich modułów.
const dbDirStage = "../live/stage/data-stores/mysql"
const appDirStage = "../live/stage/services/hello-world-app"

func TestHelloWorldAppStage(t *testing.T) {
  t.Parallel()

  // Wdrożenie bazy danych MySQL.
  dbOpts := createDbOpts(t, dbDirStage)
  defer terraform.Destroy(t, dbOpts)
  terraform.InitAndApply(t, dbOpts)

  // Wdrożenie modułu hello-world-app.
  helloOpts := createHelloOpts(dbOpts, appDirStage)
  defer terraform.Destroy(t, helloOpts)
  terraform.InitAndApply(t, helloOpts)

  // Sprawdzenie poprawności działania modułu hello-world-app.
  validateHelloApp(t, helloOpts)
}
```

Struktura testu przedstawia się następująco: wdrożenie `mysql`, wdrożenie `hello-world-app`, sprawdzenie poprawności działania aplikacji, usunięcie `hello-world-app` (ta operacja będzie wykonana dopiero na koniec dzięki wykorzystaniu polecenia `defer`) oraz usunięcie `mysql` (ta operacja również będzie wykonana dopiero na koniec dzięki wykorzystaniu polecenia `defer`). Metody `createDbOpts()`, `createHelloOpts()` i `validateHelloApp()` jeszcze nie istnieją, więc zaimplementujemy je pojedynczo, począwszy od `createDbOpts()`.

```
func createDbOpts(t *testing.T, terraformDir string) *terraform.Options {
  uniqueId := random.UniqueId()

  return &terraform.Options{
    TerraformDir: terraformDir,
```

```

Vars: map[string]interface{}{
    "db_name": fmt.Sprintf("test%s", uniqueId),
    "db_password": "password",
},
}

```

Jak dotąd niezbyt wiele się dzieje: kod wskazuje `terraform.Options` w przekazanym katalogu i przypisuje wartości zmiennych `db_name` i `db_password`.

Następnym krokiem jest zajęcie się kwestią związaną z miejscem przechowywania informacji o stanie przez moduł `mysql`. Dotychczas konfiguracja backend korzystała ze zdefiniowanych na stałe wartości.

```

terraform {
  backend "s3" {
    # Te wartości zastąp dotyczącymi używanego kubelka!
    bucket      = "terraform-up-and-running-state"
    key         = "stage/data-stores/mysql/terraform.tfstate"
    region      = "us-east-2"

    # Te wartości zastąp dotyczącymi używanej nazwy tabeli DynamoDB!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt         = true
  }
}

```

Zdefiniowane na stałe wartości stanowią poważny problem podczas testowania, ponieważ jeśli nie zostaną zmienione, skutkiem będzie nadpisanie pliku informacji o stanie rzeczywistego środowiska roboczego. Jedną z możliwości jest wykorzystanie przestrzeni roboczych Terraform (więcej informacji na ich temat przedstawiłem w rozdziale 3.), choć to wciąż będzie wymagało dostępu do kubelka S3 w koncie roboczym, podczas gdy testy powinny być przeprowadzane w zupełnie oddzielnym koncie AWS. Lepszym rozwiązaniem będzie użycie konfiguracji częściowej, również omówionej w rozdziale 3. Całą konfigurację backend przenieś do pliku zewnętrznego, np. *backend.hcl*.

```

bucket      = "terraform-up-and-running-state"
key         = "stage/data-stores/mysql/terraform.tfstate"
region      = "us-east-2"
dynamodb_table = "terraform-up-and-running-locks"
encrypt      = true

```

Natomiast w pliku *live/stage/data-stores/mysql/main.tf* pozostaw pusty blok konfiguracji backend:

```

terraform {
  backend "s3" {
  }
}

```

Po wdrożeniu modułu `mysql` w rzeczywistym środowisku roboczym za pomocą argumentu `-backend-config` nakażesz Terraform użycie konfiguracji backend zdefiniowanej w pliku *backend.hcl*.

```
$ terraform init -backend-config=backend.hcl
```

Po uruchomieniu testów w module `mysql` można nakazać Terraform przekazanie przyjaznych testom wartości za pomocą parametru `BackendConfig` w `terraform.Options`:

```

func createDbOpts(t *testing.T, terraformDir string) *terraform.Options {
    uniqueId := random.UniqueId()

    bucketForTesting := "NAZWA_KUBEŁKA_S3_DLA_TESTÓW"
    bucketRegionForTesting := "NAZWA_KUBEŁKA_S3_DLA_TESTÓW"
    dbStateKey := fmt.Sprintf("%s/%s/terraform.tfstate", t.Name(), uniqueId)

    return &terraform.Options{
        TerraformDir: terraformDir,

        Vars: map[string]interface{}{
            "db_name": fmt.Sprintf("test%s", uniqueId),
            "db_password": "password",
        },

        BackendConfig: map[string]interface{}{
            "bucket":      bucketForTesting,
            "region":      bucketRegionForTesting,
            "key":         dbStateKey,
            "encrypt":     true,
        },
    }
}

```

Konieczne jest uaktualnienie zmiennych `bucketForTesting` i `bucketRegionForTesting` wartościami w Twoim środowisku. Można utworzyć pojedynczy kubełek S3 w ramach konta AWS, aby wykorzystać go jako backend, ponieważ konfiguracja key (ścieżka dostępu w kubełku) zawiera wartość `uniqueId`, która powinna być wystarczająco unikatowa w poszczególnych testach.

Następnym krokiem jest wprowadzenie zmian w module `hello-world-app` w środowisku roboczym. Otwórz plik `live/stage/services/hello-world-app/variables.tf` i udostępnij zmienne dla `db_remote_state_bucket`, `db_remote_state_key` i `environment`:

```

variable "db_remote_state_bucket" {
    description = "Nazwa kubełka S3 dla bazy danych informacji o zdalnym stanie"
    type        = string
}

variable "db_remote_state_key" {
    description = "Ścieżka dostępu do bazy danych informacji o zdalnym stanie"
    type        = string
}

variable "environment" {
    description = "Nazwa środowiska, w którym będzie przeprowadzone wdrożenie"
    type        = string
    default     = "stage"
}

```

Przełącz te wartości do modułu `hello-world-app` w pliku `live/stage/services/hello-world-app/main.tf`:

```

module "hello_world_app" {
    source = "../../../../../modules/services/hello-world-app"

    server_text      = "Witaj, świecie"

    environment      = var.environment
    db_remote_state_bucket = var.db_remote_state_bucket
}

```

```

db_remote_state_key    = var.db_remote_state_key

instance_type          = "t2.micro"
min_size               = 2
max_size               = 2
enable_autoscaling     = false
}

```

Teraz można przystąpić do implementacji metody `createHelloOpts()`.

```

func createHelloOpts(
    dbOpts *terraform.Options,
    terraformDir string) *terraform.Options {

    return &terraform.Options{
        TerraformDir: terraformDir,

        Vars: map[string]interface{}{
            "db_remote_state_bucket": dbOpts.BackendConfig["bucket"],
            "db_remote_state_key": dbOpts.BackendConfig["key"],
            "environment": dbOpts.Vars["db_name"],
        },
    }
}

```

Zwróć uwagę na to, że zmienne `db_remote_state_bucket` i `db_remote_state_key` mają przypisane wartości użyte w konfiguracji `BackendConfig` dla modułu, co ma zagwarantować, że moduł `hello-world-app` odczytuje dane z dokładnie tych samych informacji o stanie, do których są one zapisywane przez moduł `mysql`. Zmienna `environment` otrzymuje wartość `db_name`, aby wszystkie zasoby otrzymywały nazwy w dokładnie ten sam sposób.

Teraz można przystąpić do implementacji metody `validateHelloApp()`.

```

func validateHelloApp(t *testing.T, helloOpts *terraform.Options) {
    albDnsName := terraform.OutputRequired(t, helloOpts, "alb_dns_name")
    url := fmt.Sprintf("http://%s", albDnsName)

    maxRetries := 10
    timeBetweenRetries := 10 * time.Second

    http_helper.HttpGetWithRetryWithCustomValidation(
        t,
        url,
        maxRetries,
        timeBetweenRetries,
        func(status int, body string) bool {
            return status == 200 &&
                strings.Contains(body, "Witaj, świecie")
        },
    )
}

```

Ta metoda używa pakietu `http_helper`, podobnie jak testy jednostkowe, przy czym tym razem jest wykorzystywana metoda `http_helper.HttpGetWithRetryWithCustomValidation()` pozwalająca na zdefiniowanie własnych reguł weryfikacji kodu stanu HTTP i treści odpowiedzi. Takie rozwiązanie jest konieczne w celu sprawdzenia, czy odpowiedź HTTP zawiera ciąg tekstowy *Witaj, świecie*, a nie ma

postać dokładnie podanego ciągu tekstowego — skrypt danych użytkownika w module `hello-world-app` zwraca również odpowiedź HTML wraz z innym ciągiem tekstowym.

W porządku, uruchom test integracji i sprawdź, czy działa zgodnie z oczekiwaniami.

```
$ go test -v -timeout 30m -v "TestHelloWorldAppStage"
```

```
(...)
```

```
PASS
```

```
ok      terraform-up-and-running 795.63s
```

Doskonale. W ten sposób przygotowałeś test integracji pozwalający na sprawdzenie poprawności współpracy ze sobą wielu modułów. Test integracji jest znacznie bardziej skomplikowany niż test jednostkowy, a jego wykonanie zabiera dwukrotnie więcej czasu (10 – 15 minut zamiast 4 – 5 minut). Ogólnie rzecz biorąc, niewiele można zrobić, aby *skrócić* ten czas — wąskie gardło tutaj to wydajność operacji AWS, takich jak wdrożenie i usunięcie RDS, ASG, ALB itd. — choć w pewnych sytuacjach istnieje możliwość przygotowania kodu testu wykonującego *mniej* dzięki użyciu *etapów testu*.

Etapy wykonywania testu

Jeżeli spojrzysz na kod testu integracji, to zauważysz, że zawiera on pięć oddzielnych „etapów” wykonywania:

1. Wydanie polecenia `terraform apply` w module `mysql`.
2. Wydanie polecenia `terraform apply` w module `hello-world-app`.
3. Uruchomienie procedur sprawdzających, czy wszystko działa.
4. Wydanie polecenia `terraform destroy` w module `hello-world-app`.
5. Wydanie polecenia `terraform destroy` w module `mysql`.

Gdy te testy są przeprowadzane w środowisku ciągłej integracji, wówczas będziesz chciał, aby zostały wykonane wszystkie, od początku do końca. Jednak w przypadku uruchamiania tych testów w lokalnym środowisku programistycznym podczas interaktywnego wprowadzania zmian w kodzie wykonanie wszystkich etapów okazuje się niepotrzebne. Przykładowo, jeśli wprowadzasz zmiany jedynie w module `hello-world-app`, ponowne uruchomienie wszystkich testów po każdej zmianie oznacza spory koszt związany z wdrożeniem i usunięciem modułu `mysql`, nawet jeśli żadna zmiana nie ma z nim związku. To oznacza dodanie od 5 do 10 minut do każdego uruchomienia testów.

W idealnej sytuacji sposób działania byłby podobny do następującego:

1. Wydanie polecenia `terraform apply` w module `mysql`.
2. Wydanie polecenia `terraform apply` w module `hello-world-app`.
3. Rozpoczęcie operacji iteratywnego programowania:
 - a. Wprowadzenie zmiany w module `hello-world-app`.
 - b. Ponowne wykonanie polecenia `terraform apply` w module `hello-world-app`, aby zastosować wprowadzone zmiany.

- c. Uruchomienie procedur sprawdzających, czy wszystko działa.
 - d. Jeżeli wszystko działa, następuje przejście do następnego kroku. W przeciwnym wypadku następuje powrót do kroku 3a.
4. Wydanie polecenia `terraform destroy` w module `hello-world-app`.
 5. Wydanie polecenia `terraform destroy` w module `mysql`.

Możliwość szybkiego wykonania pętli wewnętrznej w trzech krokach ma kluczowe znaczenie dla prowadzenia szybkiego, iteratywnego programowania w Terraform. Aby mieć taką możliwość, konieczne jest podzielenie kodu testu na *etapy*, co następnie pozwoli na wybieranie etapów do wykonywania i tych do pominięcia.

Terratest natywnie obsługuje etapy dzięki pakietowi `test_structure` (pamiętaj o wydaniu polecenia `dep ensure`). Idea polega na opakowaniu każdego etapu testu funkcją wraz z nazwą, co pozwala później nakazać Terratest pominięcie niektórych z tych nazw za pomocą zmiennych środowiskowych. Każdy etap testu przechowuje dane na dysku, więc mogą być odczytywane z dysku w trakcie kolejnych uruchomień. Takie rozwiązanie wypróbujesz teraz wraz z `test/hello_world_integration_test.go`, najpierw przygotujesz szkielet testu, a dopiero później dodasz niezbędne metody.

```
func TestHelloWorldAppStageWithStages(t *testing.T) {
    t.Parallel()

    // Przechowywanie funkcji w krótkich nazwach funkcji ma pozwolić,
    // aby przykładowe fragmenty kodu lepiej zmieściły się w książce.
    stage := test_structure.RunTestStage

    // Wdrożenie bazy danych MySQL.
    defer stage(t, "teardown_db", func() { teardownDb(t, dbDirStage) })
    stage(t, "deploy_db", func() { deployDb(t, dbDirStage) })

    // Wdrożenie modułu hello-world-app.
    defer stage(t, "teardown_app", func() { teardownApp(t, appDirStage) })
    stage(t, "deploy_app", func() { deployApp(t, dbDirStage, appDirStage) })

    // Sprawdzenie poprawności działania modułu hello-world-app.
    stage(t, "validate_app", func() { validateApp(t, appDirStage) })
}
```

Struktura testu przedstawia się tak samo jak wcześniej: wdrożenie `mysql`, wdrożenie `hello-world-app`, sprawdzenie poprawności działania aplikacji, usunięcie `hello-world-app` (ta operacja będzie wykonana dopiero na koniec dzięki wykorzystaniu polecenia `defer`) oraz usunięcie `mysql` (ta operacja również będzie wykonana dopiero na koniec dzięki wykorzystaniu polecenia `defer`). Różnica względem wcześniejszego przykładu polega na tym, że każdy etap został opakowany przez `test_structure.RunTestStage()`. Metoda `RunTestStage()` pobiera trzy argumenty:

t

Pierwszym argumentem jest wartość *t*, którą Go przekazuje jako argument dla każdego testu zautomatyzowanego. Tę wartość można wykorzystać do zarządzania stanem testu. Przykładowo niezaliczenie testu można wywołać za pomocą `t.Fail()`.

Nazwa etapu

Drugi argument pozwala na określenie nazwy dla danego etapu testu. Przykłady poznasz wkrótce, gdy zobaczysz, jak można tych nazw użyć do pominięcia etapów testu.

Kod do wykonania

Trzeci argument zawiera kod przeznaczony do wykonania dla danego etapu testu. To może być dowolna funkcja.

Przystępujemy do implementacji funkcji dla poszczególnych etapów testu, a zaczynamy od `deployDb()`.

```
func deployDb(t *testing.T, dbDir string) {
    dbOpts := createDbOpts(t, dbDir)
    // Zapisanie danych na dysku, aby inne etapy wykonywane później
    // miały możliwość odczytywania tych danych.
    test_structure.SaveTerraformOptions(t, dbDir, dbOpts)

    terraform.InitAndApply(t, dbOpts)
}
```

Podobnie jak wcześniej, w celu wdrożenia `mysql` kod wywołuje `createDbOpts()` i `terraform.InitAndApply()`. Jedyną nowością jest istnienie między tymi dwoma krokami wywołania `test_structure.SaveTerraformOptions()`. To powoduje zapis danych w `dbOpts` na dysku, aby późniejsze etapy mogły odczytywać te dane. Spójrz na przykładową implementację funkcji `teardownDb()`.

```
func teardownDb(t *testing.T, dbDir string) {
    dbOpts := test_structure.LoadTerraformOptions(t, dbDir)
    defer terraform.Destroy(t, dbOpts)
}
```

Ta funkcja używa `test_structure.LoadTerraformOptions()` do wczytania z dysku danych `dbOpts`, które zostały zapisane wcześniej przez funkcję `deployDb()`. Powodem przekazywania danych za pomocą dysku twardego zamiast pamięci jest to, że poszczególne etapy testu mogą być wykonywane w trakcie różnych uruchomień — a tym samym w ramach odmiennych procesów. Jak będziesz mógł zobaczyć nieco dalej w rozdziale, w trakcie kilku pierwszych uruchomień go test możesz wykonywać `deployDb()` i pomijać `teardownDb()`, natomiast w późniejszych zrobić zupełnie odwrotnie, czyli wykonywać funkcję `teardownDb()` i pomijać `deployDb()`. Aby mieć gwarancję użycia tej samej bazy danych podczas tych testów, jej informacje muszą być przechowywane na dysku.

Przystępujemy do implementacji funkcji `deployHelloApp()`.

```
func deployApp(t *testing.T, dbDir string, helloAppDir string) {
    dbOpts := test_structure.LoadTerraformOptions(t, dbDir)
    helloOpts := createHelloOpts(dbOpts, helloAppDir)

    // Zapisanie danych na dysku, aby inne etapy wykonywane później
    // miały możliwość odczytywania tych danych.
    test_structure.SaveTerraformOptions(t, helloAppDir, helloOpts)

    terraform.InitAndApply(t, helloOpts)
}
```

Ta funkcja ponownie wykorzystuje funkcję `createHelloOpts()` i wywołuje w niej `terraform.InitAndApply()`. Trzeba w tym miejscu dodać, że jedynym nowym aspektem jest użycie `test_structure.LoadTerraformOptions()` do wczytania `dbOpts` z dysku i użycie `test_structure`.

➔`SaveTerraformOptions()` do zapisania `hello0pts` na dysku. W tym momencie prawdopodobnie już się domyślasz, jak wygląda implementacja metody `teardownApp()`:

```
func teardownApp(t *testing.T, helloAppDir string) {
    hello0pts := test_structure.LoadTerraformOptions(t, helloAppDir)
    defer terraform.Destroy(t, hello0pts)
}
```

Spójrz na implementację metody `validateApp()`:

```
func validateApp(t *testing.T, helloAppDir string) {
    hello0pts := test_structure.LoadTerraformOptions(t, helloAppDir)
    validateHelloApp(t, hello0pts)
}
```

Ostatecznie cały kod testu jest identyczny z pierwotnym testem integracji z wyjątkiem opakowania każdego etapu testu wywołaniem `test_structure.RunTestStage()` oraz wykonaniem nieco dodatkowej pracy związanej z zapisaniem i odczytaniem danych z dysku. Te proste zmiany otwierają drogę dla ważnej funkcjonalności, jaką jest możliwość nakazania Terratest pominięcia dowolnego etapu testu, np. o nazwie `foo`, przez zdefiniowanie zmiennej środowiskowej `SKIP_foo=true`. Przeanalizujemy teraz typowy sposób pracy, aby pokazać, jak to działa.

Pierwszym krokiem będzie nie wykonanie testu, ale pominięcie obu etapów przygotowań, aby moduły `mysql` i `hello-world-app` zostały wdrożone na końcu testu. Ponieważ wspomniane etapy noszą nazwy `teardown_db` i `teardown_app`, konieczne jest zdefiniowanie zmiennych środowiskowych, odpowiednio `SKIP_teardown_db` i `SKIP_teardown_app`, aby bezpośrednio nakazać Terratest pominięcie tych etapów.

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
```

(...)

The 'SKIP_deploy_db' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'deploy_app' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'validate_app' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'teardown_app' environment variable is set,
so skipping stage 'deploy_db'.

(...)

```
The 'teardown_db' environment variable is set,  
so skipping stage 'deploy_db'.
```

```
(...)
```

```
PASS
```

```
ok  terraform-up-and-running 423.650s
```

Teraz można przystąpić do iteracji modułu `hello-world-app`. Po wprowadzeniu każdej zmiany można ponownie uruchamiać testy, choć tym razem nakazać pominięcie nie tylko etapu `teardown`, ale również wdrożenia modułu `mysql` (ponieważ `mysql` wciąż działa), więc jedyne wykonywane zadania to polecenie `terraform apply` i operacje sprawdzające moduł `hello-world-app`.

```
$ SKIP_teardown_db=true \  
  SKIP_teardown_app=true \  
  SKIP_deploy_db=true \  
go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
```

```
(...)
```

```
The 'SKIP_deploy_db' environment variable is set,  
so skipping stage 'deploy_db'.
```

```
(...)
```

```
The 'deploy_app' environment variable is not set,  
so executing stage 'deploy_db'.
```

```
(...)
```

```
The 'validate_app' environment variable is not set,  
so executing stage 'deploy_db'.
```

```
(...)
```

```
The 'teardown_app' environment variable is set,  
so skipping stage 'deploy_db'.
```

```
(...)
```

```
The 'teardown_db' environment variable is set,  
so skipping stage 'deploy_db'.
```

```
(...)
```

```
PASS
```

```
ok  terraform-up-and-running 13.824s
```

Zwróć uwagę na szybkość wykonywania tych testów po wprowadzonych zmianach: zamiast czekać od 10 do 15 minut po każdej zmianie, teraz można każdą zmianę sprawdzić w ciągu 10 – 60 sekund (w zależności od konkretnej zmiany). Jeśli wziąć pod uwagę, że te etapy będą wykonywane dziesiątki lub nawet setki razy, oszczędność czasu jest po prostu ogromna.

Gdy zmiany wprowadzone w module `hello-world-app` działają zgodnie z oczekiwaniami, można przystąpić do operacji porządkowych. Uruchom testy raz jeszcze, ale tym razem pominięty etapy wdrożenia i weryfikacji, aby wykonane zostały jedynie etapy `teardown`.

```
$ SKIP_deploy_db=true \  
  SKIP_deploy_app=true \  
  SKIP_validate_app=true \  
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
```

(...)

The 'SKIP_deploy_db' environment variable is set,
so skipping stage 'deploy_db'.

(...)

The 'SKIP_deploy_app' environment variable is set,
so skipping stage 'deploy_app'.

(...)

The 'SKIP_validate_app' environment variable is set,
so skipping stage 'validate_app'.

(...)

The 'SKIP_teardown_app' environment variable is not set,
so executing stage 'teardown_app'.

(...)

The 'SKIP_teardown_db' environment variable is not set,
so executing stage 'teardown_db'.

(...)

PASS

ok terraform-up-and-running 340.02s

Zastosowanie etapów testu pozwala na szybkie otrzymywanie informacji z testów zautomatyzowanych, co niezwykle przyspiesza tempo pracy i zwiększa jakość programowania iteracyjnego. Wprowadzenie to nie będzie miało żadnego wpływu na długość wykonywania testów w środowisku ciągłej integracji, ale wpływ na środowisko programistyczne jest ogromny.

Ponowne próby

Po rozpoczęciu wykonywania — w regularnych odstępach czasu — testów zautomatyzowanych dla kodu infrastruktury prawdopodobnie zetkniesz się z nowym problemem: wariujących testów. Chodzi tutaj o testy, które czasami są niezaliczane z powodu przejściowych trudności, takich jak okazjonalny problem z uruchomieniem egzemplarza EC2, błąd spójności w Terraform, błąd TLS podczas komunikacji z S3 itd. Świat infrastruktury jest daleki od doskonałego i powinieneś spodziewać się sporadycznego niezaliczania testów oraz prawidłowo obsługiwać tę sytuację.

Aby zapewnić nieco większą niezawodność testów, można spróbować ponownie je wykonywać w przypadku znanych błędów. Przykładowo podczas pisania tej książki od czasu do czasu otrzymałem następujący komunikat błędu, który pojawiał się zwłaszcza w trakcie jednoczesnego wykonywania wielu testów:

```
* error loading the remote state: RequestError: send request failed
Post https://xxx.amazonaws.com/: dial tcp xx.xx.xx.xx:443:
connect: connection refused
```

Aby zapewnić większą niezawodność testów w przypadku takich błędów, można zezwolić na wielokrotne ponawianie prób w Terratest, co wymaga użycia wymienionych tutaj argumentów terraform. ↪Options: MaxRetries, TimeBetweenRetries i RetryableTerraformErrors.

```
func createHelloOpts(
    dbOpts *terraform.Options,
    terraformDir string) *terraform.Options {

    return &terraform.Options{
        TerraformDir: terraformDir,

        Vars: map[string]interface{}{
            "db_remote_state_bucket": dbOpts.BackendConfig["bucket"],
            "db_remote_state_key": dbOpts.BackendConfig["key"],
            "environment": dbOpts.Vars["db_name"],
        },

        // W przypadku znanych błędów testy będą powtarzane do 3 razy,
        // w 5-sekundowych odstępach czasu.
        MaxRetries: 3,
        TimeBetweenRetries: 5 * time.Second,
        RetryableTerraformErrors: map[string]string{
            "RequestError: send request failed": "Throttling issue?",
        },
    }
}
```

W argumencie RetryableTerraformErrors można zdefiniować mapowanie znanych błędów, po których wystąpieniu testy będą powtarzane: kluczami mapowania są komunikaty błędów wyszukiwane w dziennikach zdarzeń (można w tym miejscu wykorzystać wyrażenia regularne), natomiast wartości to informacje dodatkowe do umieszczenia w dziennikach zdarzeń, gdy Terratest dopasuje jeden z tych błędów i spowoduje ponowną próbę wykonania testów. Po napotkaniu jednego ze znanych błędów w dzienniku zdarzeń powinieneś zobaczyć komunikat z informacją o użyciu TimeBetweenRetries, a następnie polecenie ponownie wykonujące testy.

```
$ go test -v -timeout 30m
```

```
(...)
```

```
Running command terraform with args [apply -input=false -lock=false -auto-approve]
```

```
(...)
```

```
* error loading the remote state: RequestError: send request failed
Post https://s3.amazonaws.com/: dial tcp 11.22.33.44:443:
connect: connection refused
```

(...)

```
'terraform [apply]' failed with the error 'exit status code 1'  
but this error was expected and warrants a retry. Further details:  
Intermittent error, possibly due to throttling?
```

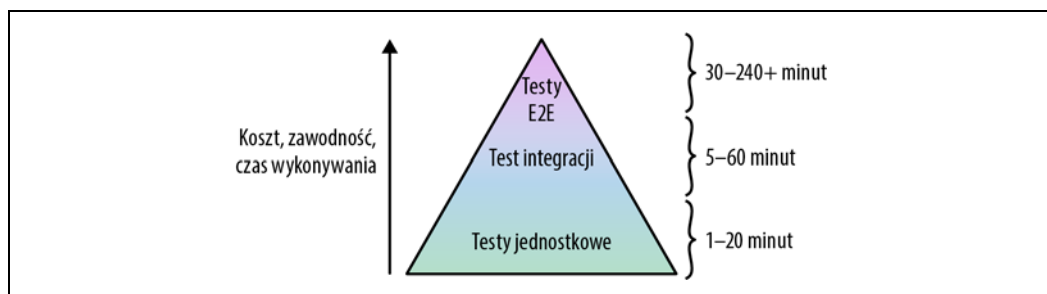
(...)

Running command terraform with args [apply -input=false -lock=false -auto-approve]

Testy typu E2E

Po przygotowaniu testów jednostkowych i testów integracji ostatnim rodzajem testów, które być może będziesz chciał dodać, są testy *typu E2E* (ang. *end-to-end*). W przypadku wspomnianego już wielokrotnie serwera WWW w języku Ruby test typu E2E może przeprowadzać wdrożenie serwera WWW, umieścić w nim niezbędne dane, a następnie przetestować ten serwer z poziomu przeglądarki WWW za pomocą narzędzia takiego jak Selenium. Testy typu E2E dla infrastruktury Terraform będą przedstawiały się podobnie: wdrożenie wszystkiego w środowisku odzwierciedlającym produkcyjne, a następnie przetestowanie z perspektywy użytkownika końcowego.

Wprawdzie testy E2E można tworzyć za pomocą dokładnie takiej samej strategii, jaka jest stosowana w testach integracji — tzn. zdefiniować etapy przeznaczone do uruchamiania terraform apply, przeprowadzania pewnych operacji sprawdzających, a następnie do uruchamiania terraform destroy — ale takie podejście jest rzadko stosowane w praktyce. To ma związek z tzw. *piramidą testów*, którą możesz zobaczyć na rysunku 7.1.



Rysunek 7.1. Piramida testów

Idea piramidy testów polega na tym, że zwykle powinienś celować w dużą liczbę testów jednostkowych (dolna część piramidy), mniejszą liczbę testów integracji (środkowa część piramidy) i jeszcze mniejszą liczbę testów E2E (górna część piramidy). To wynika z tego, że jeśli poruszać się w górę piramidy, nieustannie zwiększa się koszt i poziom skomplikowania podczas tworzenia testów, a także ich zawodność oraz czas wykonywania.

To prowadzi do *piątej reguły związanej z testowaniem*: mniejsze moduły są łatwiejsze i szybsze do testowania.

We wcześniejszej części rozdziału widziałeś, że trzeba wykonać dość sporo pracy związanej z przestrzeniami nazw, mechanizmem wstrzykiwania zależności, ponownymi próbami, obsługą błędów i etapami testów, aby przetestować nawet względnie prosty moduł `hello-world-app`. W przypadku

większej i bardziej skomplikowanej infrastruktury to zadanie stanie się jeszcze trudniejsze. Dlatego też jak największa część testów powinna być przeprowadzana na dole piramidy, ponieważ to oznacza najszybsze i najbardziej niezawodne otrzymywanie informacji zwrotnych.

W rzeczywistości, zanim dotrzesz na szczyt piramidy testów, wykonanie testów w celu wdrożenia zupełnie od początku skomplikowanej infrastruktury stanie się praktycznie niemożliwe z dwóch ważnych powodów:

Zbyt wolne

Wdrożenie całej infrastruktury zupełnie od początku, a następnie usunięcie wszystkich zasobów może zabrać bardzo dużo czasu, rzędu wielu godzin. Wykonywany tak długo zestaw testów ma dość niewielką wartość, ponieważ po prostu zbyt wolno dostarcza informacje zwrotne. Wykonanie takiego zestawu testów prawdopodobnie pozostawisz na noc, co oznacza, że dopiero rano zapoznasz się z informacjami o ewentualnym niepowodzeniu. Wówczas zaczniesz szukać błędów, wprowadzisz poprawkę i następnie zaczekasz do kolejnego dnia, aby sprawdzić, czy problem faktycznie został usunięty. W praktyce to oznacza ograniczenie do jednej poprawki dziennie. W takich sytuacjach programiści zaczynają nawzajem oskarżać się o niezaliczanie testów i skłaniają menedżerów do wdrażania produktu nawet pomimo niezaliczonych testów, co ostatecznie prowadzi do całkowitego ich ignorowania.

Zbyt zawodne

Jak już wcześniej wspomniałem, świat infrastruktury jest daleki od idealnego. Wraz ze wzrostem ilości wdrażanej infrastruktury rośnie również liczba sporadycznie występujących dziwnych sytuacji. Dla przykładu przyjmujemy założenie, że jeden zasób (np. egzemplarz EC2) charakteryzuje się 0,1-procentowym prawdopodobieństwem awarii ze względu na sporadycznie występujący błąd (faktyczna skala niepowodzeń jest w świecie DevOps większa). To oznacza, że prawdopodobieństwo bezbłędnego wykonania testu wdrażającego jeden zasób wynosi 99,9%. Jak to wygląda w przypadku dwóch zasobów? Aby test został uznany za zaliczony, oba zasoby muszą być wdrożone bez błędów, więc mnożymy prawdopodobieństwo: $99,9\% \times 99,9\% = 99,8\%$. W przypadku trzech zasobów ogólne prawdopodobieństwo sukcesu wynosi $99,9\% \times 99,9\% \times 99,9\% = 99,7\%$. Dla N zasobów wzór ma postać $99,9\%^N$.

Rozważmy teraz różne typy testów zautomatyzowanych. Jeżeli masz test jednostkowy modułu wdrażającego np. 20 zasobów, prawdopodobieństwo sukcesu wynosi $99,9\%^{20} = 98,0\%$. To oznacza, że dwa testy na każde 100 przeprowadzonych zakończą się niepowodzeniem. Jeżeli dodasz kilka ponownych prób wykonania testów, nadal można je uznawać za całkiem niezawodne. Teraz przyjmujemy założenie o istnieniu testu integracji trzech modułów wdrażających 60 zasobów. Prawdopodobieństwo sukcesu wynosi $99,9\%^{60} = 94,1\%$. Także w tym przypadku, przy zastosowaniu ponownych prób wykonania testów, można je uznać za wystarczająco stabilne, aby mogły być użyteczne. Co się stanie, gdy zechcesz przygotować test typu E2E dla całej infrastruktury składającej się np. z 30 modułów wdrażających około 600 zasobów? Prawdopodobieństwo sukcesu wynosi $99,9\%^{600} = 54,9\%$. To oznacza, że mniej więcej połowa testów zakończy się niepowodzeniem z powodu sporadycznie występujących problemów.

Część tych problemów można rozwiązać dzięki ponownym próbom wykonania testów, choć to bardzo szybko zmieni się w niekończącą się grę „whack-a-mole” (ang. *zabij kreta*). Dodasz

ponowną próbę w celu poradzenia sobie z przekroczeniem czasu oczekiwania na zakończenie fazy TLS, a staniesz przed kolejnym problemem, np. w postaci chwilowej niedostępności repozytorium APT szablonu Packer. Dodasz więc kolejną próbę dla szablonu Packer, a natkniesz się na nieudaną kompilację ze względu na błąd niespójności w Terraform. Gdy poradzisz sobie z tym błędem, pojawi się następny, np. chwilowa niedostępność serwisu GitHub. Skoro wykonywanie testów typu E2E trwa bardzo długo, skutkiem będzie możliwość podjęcia w ciągu dnia jednej lub może dwóch prób usunięcia problemu.

W praktyce niewiele firm korzystających ze skomplikowanej infrastruktury decyduje się na przeprowadzanie testów typu E2E wdrażających wszystko *zupełnie od początku*. Zamiast tego znacznie częściej spotykana strategia wykonywania testów typu E2E przedstawia się następująco:

1. Ponosi się jednorazowy koszt wdrożenia stałego środowiska testowego, które odwzorowuje produkcyjne i pozostaje uruchomione.
2. Po wprowadzeniu każdej zmiany w infrastrukturze działanie testu E2E polega na:
 - a. wprowadzeniu zmiennych infrastruktury w środowisku testowym,
 - b. uruchomieniu operacji sprawdzających w środowisku testowym (np. użyciu Selenium do przetestowania kodu z perspektywy użytkownika końcowego), aby mieć pewność co do prawidłowego działania całości.

Dzięki zmianie strategii testów typu E2E i zastosowaniu jedynie przyrostowych zmian zmniejsza się liczbę zasobów wdrażanych podczas testu z kilkuset do zaledwie kilku, co oznacza, że testy są znacznie krótsze i bardziej niezawodne.

Co więcej, takie podejście w zakresie testów E2E oznacza znacznie dokładniejsze odwzorowanie wdrożenia tych zmian w środowisku produkcyjnym. W końcu po wprowadzeniu zmiany środowisko produkcyjne nie jest przygotowywane zupełnie od początku. Zamiast tego poszczególne zmiany są wprowadzane przyrostowo, więc ten styl testów typu E2E ma ogromną zaletę: możliwość sprawdzenia nie tylko prawidłowości działania infrastruktury, ale również poprawności *procesu wdrożenia* tej infrastruktury.

Przykładowo kluczowe znaczenie będzie miało przetestowanie, czy uaktualnienie infrastruktury można przeprowadzić bez przestoju. Wprawdzie utworzony wcześniej moduł `asg-rolling-deploy` ma możliwość wdrożenia bez przestoju, ale jak można to sprawdzić? W tym celu dodamy teraz test zautomatyzowany.

Konieczne będzie wprowadzenie zaledwie kilku zmian w teście utworzonym w pliku `test/hello_world_integration_test.go`, ponieważ w tle moduł `hello-world-app` używa modułu `asg-rolling-deploy`. Pierwszym krokiem jest udostępnienie zmiennej `server_text` w `live/stage/services/hello-world-app/variables.tf`:

```
variable "server_text" {  
  description = "Komunikat, który powinien być zwrócony przez serwer WWW"  
  default     = "Witaj, świecie"  
  type        = string  
}
```

Tę zmienną należy przekazać do modułu `hello-world-app` w pliku `live/stage/services/hello-world-app/main.tf`:

```

module "hello_world_app" {
  source = "../../../../../modules/services/hello-world-app"

  server_text = var.server_text

  environment      = var.environment
  db_remote_state_bucket = var.db_remote_state_bucket
  db_remote_state_key   = var.db_remote_state_key

  instance_type     = "t2.micro"
  min_size          = 2
  max_size          = 2
  enable_autoscaling = false
}

```

Moduł `hello-world-app` używa argumentu `server_text` w jego skrypcie danych użytkownika, więc każda zmiana tej zmiennej wymusza (mamy nadzieję) wdrożenie bez przestoju. Sprawdźmy to przez dodanie kolejnego etapu testu o nazwie `redeploy_app` do pliku `test/hello_world_integration_test.go` na końcu testu, bezpośrednio po etapie `validate_app`.

```

func TestHelloWorldAppStageWithStages(t *testing.T) {
  t.Parallel()

  // Przechowywanie funkcji w krótkich nazwach funkcji ma pozwolić,
  // aby przykładowe fragmenty kodu lepiej zmieściły się w książce.
  stage := test_structure.RunTestStage

  // Wdrożenie bazy danych MySQL.
  defer stage(t, "teardown_db", func() { teardownDb(t, dbDirStage) })
  stage(t, "deploy_db", func() { deployDb(t, dbDirStage) })

  // Wdrożenie modułu hello-world-app.
  defer stage(t, "teardown_app", func() { teardownApp(t, appDirStage) })
  stage(t, "deploy_app", func() { deployApp(t, dbDirStage, appDirStage) })

  // Sprawdzenie poprawności działania modułu hello-world-app.
  stage(t, "validate_app", func() { validateApp(t, appDirStage) })

  // Ponowne wdrożenie modułu hello-world-app.
  stage(t, "redeploy_app", func() { redeployApp(t, appDirStage) })
}

```

Kolejnym krokiem jest ponowna implementacja nowej metody `redeployApp()`.

```

func redeployApp(t *testing.T, helloAppDir string) {
  helloOpts := test_structure.LoadTerraformOptions(t, helloAppDir)

  albDnsName := terraform.OutputRequired(t, helloOpts, "alb_dns_name")
  url := fmt.Sprintf("http://%s", albDnsName)

  // Rozpoczęcie sprawdzania co sekundę, czy aplikacja zwraca kod stanu 200 OK.
  stopChecking := make(chan bool, 1)
  waitGroup, _ := http_helper.ContinuouslyCheckUrl(
    t,
    url,
    stopChecking,
    1*time.Second,
  )
}

```

```

// Uaktualnienie komunikatu serwera i ponowne wdrożenie.
newServerText := "Witaj, świecie, v2!"
hello0pts.Vars["server_text"] = newServerText
terraform.Apply(t, hello0pts)

// Upewnienie się o wdrożeniu nowej wersji.
maxRetries := 10
timeBetweenRetries := 10 * time.Second
http_helper.HttpGetWithRetryWithCustomValidation(
    t,
    url,
    maxRetries,
    timeBetweenRetries,
    func(status int, body string) bool {
        return status == 200 &&
            strings.Contains(body, newServerText)
    },
)

// Zakończenie sprawdzania.
stopChecking <- true
waitGroup.Wait()
}

```

Ta metoda wykonuje następujące działania:

1. Używa metody pomocniczej Terratest `http_helper.ContinuouslyCheckUrl()` do uruchomienia w tle podprocedury (lekki wątek zarządzany przez środowisko uruchomieniowe Go), która co sekundę będzie wykonywała żądanie HTTP GET pod podany adres URL ALB i zakończy test niepowodzeniem po otrzymaniu odpowiedzi wraz z kodem stanu innym niż 200 OK.
2. Uaktualnia nową zmienną `server_text` i wykonuje polecenie `terraform apply`, aby zainicjować wdrożenie.
3. Po zakończeniu wdrożenia metoda sprawdza, czy odpowiedź udzielona przez serwer zawiera wartość zmiennej `server_text`.
4. Zatrzymanie podprocedury nieustannie wykonującej żądanie pod adres URL ALB.

Po uruchomieniu tego testu i przejściu do jego ostatniego etapu otrzymasz dane wyjściowe pokazujące nieustanne wykonywanie żądań HTTP GET do ALB i wykonywanie polecenia `terraform apply` odpowiedzialnego za wdrożenie.

```

$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'

```

(...)

```

Making an HTTP GET call to URL
http://hello-world-test-th1MBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-th1MBF-168938547.us-east-2.elb.amazonaws.com

```

```

Making an HTTP GET call to URL
http://hello-world-test-th1MBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-th1MBF-168938547.us-east-2.elb.amazonaws.com

```

(...)

```
Running command terraform with args  
[apply -input=false -lock=false -auto-approve]
```

(...)

```
Making an HTTP GET call to URL  
http://hello-world-test-th1MBF-168938547.us-east-2.elb.amazonaws.com  
Got response 200 and err <nil> from URL at  
http://hello-world-test-th1MBF-168938547.us-east-2.elb.amazonaws.com
```

```
Making an HTTP GET call to URL  
http://hello-world-test-th1MBF-168938547.us-east-2.elb.amazonaws.com  
Got response 200 and err <nil> from URL at  
http://hello-world-test-th1MBF-168938547.us-east-2.elb.amazonaws.com
```

(...)

```
PASS  
ok  terraform-up-and-running 551.04s
```

Jeżeli test zostanie zaliczony, to oznacza, że moduły `hello-world-app` i `asg-rolling-deploy` można zgodnie z obietnicą wdrożyć bez przestoju. Za każdym razem, gdy test typu E2E ma zastosowanie dla przyrostowej zmiany w środowisku testowym, tę strategię można wykorzystać do upewnienia się, że wdrożenie nie powoduje przestoju.

Inne podejścia w zakresie testów

W większej części rozdziału skoncentrowałem się na używaniu Terratest do testów zautomatyzowanych. W zakresie podejścia do testowania istnieją jeszcze dwie inne kategorie, które można wykorzystać:

- analiza statyczna,
- testowanie właściwości.

Podobnie jak poszczególne rodzaje testów zautomatyzowanych (jednostkowe, integracji i E2E) przechwytyują odmienne typy błędów, tak samo wymienione tutaj podejścia w zakresie testowania przechwytyują różne rodzaje błędów. Dlatego też najlepiej będzie zastosować różne podejścia, aby otrzymać jak najlepsze wyniki podczas testowania. Teraz dokładnie omówię nowe kategorie.

Analiza statyczna

Dostępnych jest wiele narzędzi pozwalających na analizowanie kodu Terraform bez jego uruchamianie, m.in.:

`terraform validate`

To polecenie jest wbudowane w Terraform i można je wykorzystać do sprawdzenia składni i typów Terraform (czyli działa na zasadzie podobnej do kompilatora).

tflint (<https://github.com/terraform-linters/tflint>)

Narzędzie typu „linter” potrafi przeskanować kod Terraform i na podstawie zestawu wbudowanych reguł przechwycić często popełniane błędy i potencjalne pomyłki.

HashiCorp Sentinel (<https://www.hashicorp.com/sentinel/>)

Framework typu „polityka jako kod” pozwala na wymuszenie stosowania reguł w różnych narzędziach HashiCorp. Przykładowo można zdefiniować politykę uniemożliwiającą tworzenie w grupach bezpieczeństwa kodu Terraform reguł zezwalających na dostęp do adresów 0.0.0.0/0. W chwili powstawania książki ten framework był dostępny jedynie z produktami HashiCorp Enterprise — m.in. Terraform Enterprise.

Testowanie właściwości

Istnieje wiele narzędzi testowania koncentrujących się na weryfikacji określonych „właściwości” infrastruktury:

- kitchen-terraform (<https://github.com/newcontext-oss/kitchen-terraform>),
- rspec-terraform (<https://github.com/bsnape/rspec-terraform>),
- serverspec (<https://serverspec.org/>),
- inspec (<https://www.inspec.io/>),
- goss (<https://github.com/aelsabbahy/goss>).

Większość tych narzędzi oferuje prosty język specjalizowany (ang. *domain-specific language*, DSL) przeznaczony do sprawdzania, czy wdrożona infrastruktura jest zgodna z podaną specyfikacją. Przykładowo, jeśli testujesz moduł Terraform wdrożony w egzemplarzu EC2, możesz skorzystać z przedstawionego tutaj kodu inspec do sprawdzenia, czy egzemplarz ma prawidłowe uprawnienia do określonych plików i zainstalowane wymagane zależności oraz czy nasłuchuje na konkretnym porcie.

```
describe file('/etc/myapp.conf') do
  it { should exist }
  its('mode') { should cmp 0644 }
end

describe apache_conf do
  its('Listen') { should cmp 8080 }
end

describe port(8080) do
  it { should be_listening }
end
```

Zaletą takich narzędzi jest to, że język DSL zwykle jest zwięzły, łatwy w użyciu oraz oferuje efektywny i deklaracyjny sposób na weryfikację ogromnej liczby właściwości dotyczących infrastruktury. To doskonale rozwiązanie ułatwiające stosowanie listy rzeczy do sprawdzenia w zakresie wymagań, a zwłaszcza zgodności z PCI, HIPAA itd. Natomiast wadą tych narzędzi jest ryzyko, że te wszystkie operacje sprawdzenia właściwości zostaną zaliczone, a mimo to infrastruktura Terraform nadal nie będzie działała. Dla porównania „sposób Terratest” na weryfikację tych samych właściwości polega na wykonaniu żądania HTTP do serwera i sprawdzeniu, czy otrzymana odpowiedź jest zgodna z oczekiwaniami.

Podsumowanie

W świecie infrastruktury wszystko ulega nieustannym zmianom: Terraform, Packer, Docker, Kubernetes, AWS, Google Cloud, Azure itd. To z kolei oznacza bardzo szybką rotację kodu infrastruktury lub — jeśli ująć to w zupełnie inny sposób:

Pozbawiony testów zautomatyzowanych kod infrastruktury jest nieprawidłowy.

Ta fraza jest jednocześnie aforyzmem i precyzyjnym zdaniem. Za każdym razem, gdy przystępuję do tworzenia kodu infrastruktury, niezależnie od wysiłku wkładanego w zachowanie przejrzystości tego kodu, ręcznego przetestowania i przeanalizowania, po przejściu do przygotowania testów zautomatyzowanych odnajduję wiele poważnych błędów. Coś magicznego dzieje się, gdy poświęci się nieco czasu na automatyzację procesu testowania. Praktycznie zawsze pozwala to na ujawnienie problemów, których w przeciwnym razie nigdy byś nie znalazł (natomiast zostałyby znalezione przez klientów). Błędy są znajdowane nie tylko tuż po dodaniu testów zautomatyzowanych, ale również po ich wykonaniu po każdej operacji zatwierdzenia. Dzięki temu są wyszukiwane i znajdowane non stop, zwłaszcza gdy zmienia się otaczający Cię świat DevOps.

Testy zautomatyzowane dodane do mojego kodu infrastruktury mają na celu wychwytywanie błędów nie tylko w nim, ale także w używanych przeze mnie narzędziach — to dotyczy również poważniejszych błędów w Terraform, Packer, Elasticsearch, Kafka, AWS itd. Tworzenie testów zautomatyzowanych, jak pokazałem w tym rozdziale, *nie* należy do łatwych zadań. Przygotowanie takich testów wymaga znacznego wysiłku, a dodatkowy wysiłek wiąże się z ich obsługą i dodawaniem kolejnej logiki zapewniającej ich niezawodność. Jeszcze więcej wysiłku wiąże się z zachowaniem przejrzystości środowiska testowego i zachowaniem kosztów pod kontrolą. Mimo to warto to zrobić.

Gdy tworzę moduł do wdrożenia, np. magazynu danych, wówczas po każdej operacji przekazania kodu do repozytorium testy uruchamiają dziesiątki kopii magazynu danych w różnych konfiguracjach, zapisują dane, odczytują je, a następnie wszystkie usuwają. Jeżeli te testy zostaną zaliczone, mam dużą pewność, że utworzony przeze mnie kod nadal działa. Testy zautomatyzowane pozwalają mi na spokojny sen. Godziny poświęcone na przygotowanie logiki i zapewnienie spójności zwracają się, gdy nie muszę o trzeciej nad ranem zmagać się z awarią infrastruktury.



Kod dla tej książki również zawiera testy!

Wszystkie przykładowe fragmenty kodu przygotowane dla tej książki również zawierają testy. Te przykłady i opracowane dla nich testy znajdziesz w repozytorium GitHub pod adresem <https://github.com/brikis98/terraform-up-and-running-code>.

W rozdziale przedstawiłem podstawowy proces testowania kodu Terraform, na podstawie tego procesu można wysunąć następujące wnioski:

Podczas testowania kodu Terraform nie istnieje tzw. komputer lokalny.

Dlatego też wszystkie operacje ręcznego testowania trzeba przeprowadzać przez wdrażanie rzeczywistych zasobów w jednym lub większej liczbie odizolowanych środowisk.

Należy regularnie przeprowadzać porządkowanie środowisk.

W przeciwnym razie środowiska staną się niemożliwe do zarządzania, a koszty związane z ich utrzymaniem wymkną się spod kontroli.

Nie można tworzyć czystych testów jednostkowych dla kodu Terraform.

Konieczne jest więc przygotowywanie testów zautomatyzowanych przez utworzenie kodu wdrażającego rzeczywiste zasoby w jednym lub większej liczbie odizolowanych środowisk.

Konieczne jest stosowanie przestrzeni nazw dla wszystkich zasobów.

To gwarantuje, że jednoczesne przeprowadzanie wielu testów nie będzie prowadziło do powstawania konfliktów między nimi.

Mniejsze moduły są łatwiejsze i szybsze do testowania.

To jest jeden z wniosków przedstawionych w rozdziale 6. i warto go tutaj powtórzyć: małe moduły są łatwiejsze do tworzenia i obsługi, a także w użyciu oraz w testowaniu.

Z rozdziału 8. dowiesz się, jak kod Terraform i testy zautomatyzowane wykorzystać w pracy zespołu. Poruszę m.in. tematy zarządzania środowiskami, konfiguracji mechanizmów ciągłej integracji i ciągłego wdrażania.

Używanie Terraform w zespołach

W trakcie lektury książki i pracy nad przykładowymi fragmentami kodu przez większość czasu samodzielnie wykonywałeś zadania. Jednak w rzeczywistości prawdopodobnie będziesz członkiem zespołu, co oznacza wiele nowych wyzwań. Musisz znaleźć sposób na przekonanie zespołu do używania Terraform oraz innych narzędzi infrastruktury jako kodu (IaC). Być może będziesz musiał stawić czoła sytuacji, w której jednocześnie większa liczba osób próbuje zrozumieć, używać i modyfikować tworzony przez Ciebie kod Terraform. Konieczne będzie określenie, jak umieścić Terraform w używanym stosie narzędzi i jak wdrożyć to narzędzie do systemu pracy stosowanego w firmie.

W rozdziale spróbuję zagłębić się w kluczowe procesy pozwalające na efektywną pracę z Terraform i ogólnie związane ze stosowaniem praktyk IaC w zespole. Poruszę przy tym następujące tematy:

- adaptacja infrastruktury jako kodu przez zespół,
- sposób pracy podczas wdrażania kodu aplikacji,
- sposób pracy podczas wdrażania kodu infrastruktury,
- zebranie wszystkiego w całość.

Poszczególne tematy zostaną omówione pojedynczo.



Przykładowe fragmenty kodu

Przypominam, że wszystkie przykładowe fragmenty kodu znajdują się w repozytorium GitHub pod adresem <https://github.com/brikis98/terraform-up-and-running-code>.

Adaptacja infrastruktury jako kodu przez zespół

Jeżeli Twój zespół przywykł do ręcznego zarządzania całą infrastrukturą, przejście do podejścia infrastruktury jako kodu będzie wymagało znacznie większego wysiłku niż tylko przedstawienie nowego narzędzia lub technologii. Konieczna będzie również zmiana kultury i sposobu pracy zespołu — a to jest już poważne wyzwanie, zwłaszcza w przypadku ogromnych firm. Ponieważ kultura i sposób pracy są nieco odmienne w poszczególnych zespołach, nie ma jednego uniwersalnego rozwiązania, choć warto w tym miejscu wspomnieć o kilku kwestiach:

- przekonanie szefa do pomysłu,
- stopniowe wprowadzanie zmian,
- zapewnienie zespołowi czasu na naukę.

Przekonanie szefa do pomysłu

Wielokrotnie spotykałem się z następującą sytuacją: programista odkrywa Terraform, zostaje zainspirowany możliwościami tego narzędzia, jest pełen zapału i entuzjazmu, pokazuje Terraform swoim współpracownikom, a szef mówi „nie”. Programista oczywiście popada w frustrację i zniechęcenie. Dlaczego nikt inny nie dostrzega korzyści, jakie może przynieść wykorzystanie nowego narzędzia? Dzięki temu narzędziu można zautomatyzować naprawdę wiele zadań! Można uniknąć wielu błędów! Jak inaczej spłacić ten cały dług technologiczny? Dlaczego nie dostrzega się możliwości drzemiących w Terraform?

Problem polega na tym, że choć programista widzi wszelkie zalety zastosowania narzędzia IaC takiego jak Terraform, to nie dostrzega związanego z tym kosztu. Oto kilka wybranych kwestii związanych z kosztem adaptacji podejścia IaC:

Brak umiejętności

Przejsie do stosowania podejścia opartego na IaC oznacza, że zespół operacji musi poświęcić większość swojego czasu na tworzenie ogromnych ilości kodu: modułów Terraform, testów w języku Go, receptur Chef itd. Wprawdzie część inżynierów zespołu operacji nie ma problemu z całodniowym tworzeniem kodu i lubi zmiany, ale część ma zupełnie przeciwne nastawienie. Wielu inżynierów zespołu operacji i administratorów systemu przywykło do ręcznego wprowadzania zmian i prawdopodobnie od czasu do czasu tworzą krótkie skrypty. W takim przypadku przejście do tworzenia oprogramowania przez niemal cały czas będzie wymagało zdobycia nowych umiejętności lub zatrudnienia nowych pracowników.

Nowe narzędzia

Twórcy oprogramowania mogą być przywiązani do używanych narzędzi, a część z nich nawet wręcz obsesyjnie przywiązana do pewnych narzędzi. Za każdym razem, gdy jest wprowadzane nowe narzędzie, część programistów będzie odczuwała deszczyk emocji związany z poznawaniem czegoś nowego, inni zaś będą preferowali pozostanie przy znanym im środowisku i sprzeciwiali się konieczności poświęcenia znacznej ilości czasu i energii na poznawanie nowych języków i technik.

Zmiana nastawienia

Jeżeli członkowie zespołu przywykli do ręcznego zarządzania infrastrukturą, wszelkie zmiany wprowadzają *bezpośrednio* np. przez nawiązanie połączenia SSH z serwerem i wydawanie poleceń. Przejście do podejścia IaC wymaga zmiany nastawienia, ponieważ w IaC zmiany są wprowadzane *pośrednio*, najpierw przez edycję kodu, następnie jest przekazanie do repozytorium, a później zlecenie pewnemu procesowi automatycznemu zastosowania tych zmian. Taka warstwa pośredniości może być frustrująca. W przypadku prostych zadań można odnieść wrażenie, że nowe podejście jest wolniejsze niż bezpośrednie zmiany, zwłaszcza gdy programista dopiero poznaje nowe narzędzia IaC i jeszcze nie potrafi efektywnie z nich korzystać.

Koszt utraconych korzyści

Jeżeli zdecydujesz się na inwestycję czasu i zasobów w jeden projekt, niejawnie decydujesz, że nie chcesz poświęcać czasu i zasobów na inne projekty. Jakie projekty trzeba będzie odłożyć na bok, aby można było przeprowadzić migrację do IaC? Jaką wagę mają te projekty?

Część programistów w zespole spojrzy na tę listę i będzie podekscytowana. Z kolei wielu będzie narzekać — w tym Twój szef. Zdobywanie nowych umiejętności, poznawanie nowych narzędzi i zmiana nastawienia może — choć nie musi — wyjść na dobre, a jednego można być pewnym: to nie odbywa się bez żadnego kosztu. Adaptacja IaC to znaczna inwestycja i, podobnie jak w wielu innych inwestycjach, konieczne jest rozważenie nie tylko potencjalnych korzyści, ale i strat.

Twój szef będzie szczególnie uczulony na koszt utraconych korzyści. Jednym z kluczowych obowiązków menedżera jest zapewnienie, by zespół pracował nad projektami o najwyższych priorytetach. Gdy pojawisz się przed nim i podekscytowany rozpoczniesz opowiadanie o Terraform, Twój szef może pomyśleć: „O nie, to brzmi jak ogromne przedsięwzięcie, ciekawe, ile czasu zajmie jego realizacja”. To nie oznacza, że Twój szef jest ślepy i nie dostrzega potencjalnych korzyści oferowanych przez Terraform. Jeżeli trzeba będzie poświęcić nieco czasu na adaptację nowej technologii, tego czasu może zabraknąć na wdrożenie nowej wersji aplikacji oczekiwanej od miesięcy, na przygotowanie audytu PCI (ang. *payment card industry*) lub na zagłębienie się w awarię z ubiegłego tygodnia. Dlatego też jeśli chcesz przekonać szefa do zaadaptowania przez Twój zespół podejścia IaC, musisz pokazać mu nie tylko samą wartość nowego rozwiązania, ale przede wszystkim to, że nowe podejście będzie miało znacznie większą wartość dla zespołu niż wszystko inne, nad czym obecnie ten zespół pracuje.

Jednym z najmniej efektywnych sposobów jest ograniczenie się do przedstawienia listy funkcjonalności ulubionego narzędzia IaC — w przypadku Terraform może to być deklaratywny sposób działania, obsługa wielu chmur i dostępność w postaci oprogramowania open source. To jest przykład jednego z wielu obszarów, na których programiści mogą wiele nauczyć się od handlowców. Większość z nich wie, że skoncentrowanie się na funkcjach to najbardziej nieefektywny sposób na sprzedaż produktu. Znacznie lepszą techniką jest koncentracja na korzyściach, tzn. zamiast mówić o tym, co produkt potrafi („produkt X może zrobić Y”), należy mówić o tym, co klient może zyskać, jeśli będzie używać danego produktu („możesz zrobić Y, mając produkt X”). Innymi słowy, należy pokazać klientowi nowe możliwości, jakie zyska po nabyciu danego produktu.

Przykładowo, zamiast mówić szefowi o deklaratywnej naturze Terraform, znacznie lepiej będzie podkreślić to, że zespół będzie mógł szybciej realizować projekty. Zamiast informować o obsłudze wielu platform chmury przez Terraform, należy wspomnieć o możliwości łatwej migracji między chmurami, gdy pewnego dnia zajdzie taka potrzeba, bez konieczności zmiany używanych narzędzi. Zamiast wyjaśniać szefowi, że Terraform to oprogramowanie typu open source, lepiej powiedzieć szefowi, jak dużo łatwiej będzie można zatrudniać do zespołu nowych programistów z ogromnej i aktywnej społeczności open source.

Koncentracja na korzyściach to dobry początek. Najlepsi handlowcy znają jeszcze bardziej efektywną strategię: koncentrację na problemach. Jeżeli będziesz obserwować rozmowę między doskonałym handlowcem i klientem, to zauważysz, że więcej mówi klient. Handlowiec spędza większość czasu na słuchaniu i szukaniu odpowiedzi na jedno ważne pytanie: jaki podstawowy problem klient próbuje

rozwiązać? Co jest jego największą bolączką? Zamiast spróbować sprzedać produkt o pewnych funkcjach i zapewniających określone korzyści, najlepszy handlowiec stara się rozwiązać problem klienta. Jeżeli rozwiązanie zawiera sprzedawany przez niego produkt, tym lepiej. Jednak handlowiec koncentruje się przede wszystkim na rozwiązaniu problemu, a nie na sprzedaży.

Podczas rozmowy z szefem staraj się zrozumieć najważniejsze problemy, przed którymi stoi on w tym kwartale. Może się okazać, że te problemy nie zostaną rozwiązane przez IaC. Nie ma w tym niczego złego. Te słowa możesz uznać za herezję jak na autora książki o Terraform, ale pamiętaj, że nie każdy zespół potrzebuje podejścia IaC. Adaptacja IaC wiąże się z dość wysokim kosztem i mimo że w dłuższej perspektywie czasu ten koszt się w pewnych sytuacjach zwraca, niekoniecznie tak jest we wszystkich — przykładowo, jeśli należysz do małego startupu zatrudniającego tylko jedną osobę w dziale operacji i pracujesz nad prototypem, który będzie gotowy dopiero za kilka miesięcy, lub dla rozrywki pracujesz nad projektem ubocznym, ręczne zarządzanie infrastrukturą jest właściwym rozwiązaniem. Czasami, nawet jeśli IaC będzie odpowiednim podejściem do zastosowania przez zespół, może nie mieć najwyższego priorytetu, a ze względu na ograniczone zasoby praca nad innymi projektami nadal będzie lepszym rozwiązaniem.

Jeśli stwierdzisz, że jeden z ważniejszych problemów Twojego szefa może być rozwiązany dzięki zastosowaniu podejścia IaC, powinieneś szefowi pokazać, jak takie rozwiązanie będzie wyglądało. Przykładowo jednym z największych zmartwień szefa w danym kwartale jest poprawienie czasu działania usługi. W zeszłym miesiącu doszło do wielu awarii, których skutkiem były godziny przestoju. Klienci narzekają, a prezes firmy uważnie patrzy Twojemu szefowi na ręce i codziennie sprawdza, jak wygląda sytuacja na tym polu. Zaczynasz się przyglądać tym awariom i odkrywasz, że połowa z nich była spowodowana przez błąd popełniony podczas wdrożenia, np. ktoś przypadkowo pominął ważny krok, serwer został błędnie skonfigurowany lub infrastruktura w środowisku roboczym nie odpowiadała tej w środowisku produkcyjnym.

Teraz, w rozmowie z szefem, zamiast wymieniać funkcje lub korzyści oferowane przez Terraform, lepiej będzie podejść do sprawy z innej strony, np.: „Mam pomysł, jak można o połowę zmniejszyć liczbę awarii”. Takie stwierdzenie na pewno zwróci uwagę szefa. Wykorzystaj tę sytuację do przedstawienia wizji, w której proces wdrożenia został w pełni zautomatyzowany, a ponadto stał się niezawodny i powtarzalny, co pozwoliło na wyeliminowanie popełnianych podczas ręcznego wdrażania błędów odpowiedzialnych za połowę wcześniejszych awarii. Ale to nie wszystko. W przypadku zautomatyzowanego wdrożenia można dodać testy zautomatyzowane i jeszcze bardziej ograniczyć liczbę awarii oraz umożliwić firmie częstsze wdrożenia. Pozwól szefowi wyobrazić sobie siebie w roli tego, który informuje prezesa o zmniejszeniu liczby awarii o połowę i o podwojeniu liczby wdrożeń. Następnie wspomnij, że na podstawie przeprowadzonej analizy jesteś przekonany o możliwości osiągnięcia tego za pomocą Terraform.

Oczywiście nie ma żadnej gwarancji, że szef się zgodzi, ale takie podejście ma znacznie większe szanse powodzenia. Szanse można zwiększyć jeszcze bardziej dzięki wprowadzaniu zmian stopniowo.

Stopniowe wprowadzanie zmian

Jedną z najważniejszych lekcji, jakie otrzymałem w mojej karierze, jest ta, że większość ogromnych projektów oprogramowania zakończy się niepowodzeniem. Podczas gdy trzy czwarte małych

(o budżecie poniżej miliona dolarów) projektów IT kończy się sukcesem, to tylko jedna dziesiąta ogromnych (budżet powyżej 10 milionów dolarów) projektów kończy się na czas oraz w ramach ustalonego budżetu, a ponad jedna trzecia ogromnych projektów w ogóle nie zostaje ukończona¹.

Dlatego też zawsze mam obawy, gdy widzę, jak zespół próbuje nie tylko zaadaptować IaC, ale jednocześnie wprowadzić to podejście w ogromnej infrastrukturze, we wszystkich zespołach i często jako część większego projektu. Wprawdzie nie mogę pomóc, ale jestem zaskoczony, że CEO lub CTO dużej firmy nakazuje migrację wszystkiego do chmury, wyłączenie starych centrów danych i próbuje w ciągu pół roku zrobić z każdego pracownika „specjalistę DevOps” (cokolwiek miałyby to znaczyć). Nie przesadzę, jeśli stwierdzę, że spotkałem się z taką sytuacją już dziesiątki razy, a wszystkie zakończyły się fiaskiem. Dwa lub trzy lata później każda z tych firm nadal pracuje nad migracją, stare centra danych wciąż działają i nikt nie potrafi powiedzieć, czym właściwie zajmuje się w dziedzinie DevOps.

Jeżeli chcesz z sukcesem zaadaptować podejście IaC lub odnieść sukces w innym dowolnym projekcie obejmującym migrację, jedynym sensownym rozwiązaniem jest stopniowe wprowadzanie zmian. Kluczem do podejścia *stopniowego* jest nie tylko podzielenie pracy na serię małych kroków, ale również podział każdego kroku w taki sposób, aby miał własną wartość — nawet jeśli późniejsze kroki nigdy nie zostaną wykonane.

Aby zrozumieć, dlaczego to jest tak ważne, spróbuj przeanalizować podejście zgoła odmienne, czyli *falszywe stopniowanie*². Przyjmujemy założenie, że masz ogromny projekt migracji podzielony na kilka mniejszych kroków, przy czym projekt nie będzie oferował żadnej rzeczywistej wartości aż do momentu ukończenia ostatniego kroku. Przykładowo pierwszym krokiem jest ponowne utworzenie frontendu, ale bez jego uruchomienia, ponieważ działanie tego frontendu opiera się na nowym backendzie. Następnie przystępujesz do ponownego utworzenia backendu, którego również nie uruchamiasz, ponieważ nie będzie działał aż do chwili zakończenia migracji danych do nowego magazynu danych. Ostatnim krokiem jest migracja danych. Dopiero po wykonaniu tego kroku będzie można wszystko uruchomić i zacząć odnosić jakąkolwiek wartość z tej całej pracy. Oczekiwanie do końca projektu na osiągnięcie jakiegokolwiek wartości jest bardzo ryzykowne. Jeżeli projekt zostanie anulowany, wstrzymany lub też istotnie zmodyfikowany, to pomimo ogromnych inwestycji wartością zwrótną będzie zero.

Z dokładnie taką sytuacją można się spotkać w przypadku wielu migracji ogromnych projektów. Projekt jest ogromny, aby rozpocząć z nim pracę, i podobnie jak w przypadku większości projektów oprogramowania, jego wykonanie zabiera więcej czasu, niż oczekiwano. Tymczasem następuje zmiana warunków na rynku albo początkowi udziałowcy tracą cierpliwość (np. nie ma problemu, gdy CEO poświęca 3 miesiące na spłatę debetu, ale po 12 miesiącach najwyższy czas rozpocząć dostarczanie nowych produktów) i projekt zostaje anulowany przed jego zakończeniem. W przypadku

¹ Przygotowany przez The Standish Group raport *CHAOS Manifesto 2013: Think Big, Act Small*, 2013. Znajdziesz go pod adresem http://athena.ecs.csus.edu/~buckley/CSc231_files/Standish_2013_Report.pdf.

² Zob. artykuł *How To Survive a Ground-Up Rewrite Without Losing Your Sanity* napisany przez Dana Milsteina i opublikowany 8 kwietnia 2013 roku w serwisie OnStartups.com (<https://www.onstartups.com/tabid/3339/bid/97052/How-To-Survive-a-Ground-Up-Rewrite-Without-Losing-Your-Sanity.aspx>).

falszywego stopniowania otrzymujesz najgorszy z możliwych wyników: ponosisz ogromny koszt i nie dostajesz absolutnie nic w zamian.

Dlatego też stopniowanie jest ważne. Każdy etap projektu powinien dostarczać pewną wartość, nawet jeśli nie zostanie on dokończony. Niezależnie od liczby wykonanych kroków zawsze otrzymujesz coś w zamian. Najlepszym sposobem na wykonanie takiego zadania jest skoncentrowanie się w danej chwili na rozwiązaniu jednego, małego i konkretnego problemu. Przykładowo, zamiast próbować przeprowadzić migrację wszystkiego do chmury, lepiej będzie wyznaczyć jedną, małą i konkretną aplikację lub zespół i dla nich zdecydować się na migrację. Zamiast przenosić wszystko i postawić na podejście DevOps, lepiej wyszukać pojedynczy, mały, konkretny problem (np. awarie podczas wdrożenia) i wprowadzić zmianę w miejscu rozwiązania tego problemu (np. automatyzację za pomocą Terraform najbardziej problemowych wdrożeń).

Jeżeli jesteś w stanie odnieść łatwe zwycięstwo przez natychmiastowe rozwiązanie rzeczywistego, małego problemu i zapewnić zespołowi sukces, rozpoczynasz nabieranie rozpędu. Taki zespół może stać się liderem i zachętą dla innych do przeprowadzenia migracji. Usunięcie problemu będzie chwalone ze strony CEO i okaże się pomocne podczas przekonywania do użycia IaC w innych projektach. Skutkiem może być kolejne łatwe zwycięstwo i następne itd. Gdy będziesz powtarzać ten proces — i za każdym razem zapewniać jakąś wartość — prawdopodobnie odniesiesz sukces podczas większej migracji. Jeśli natomiast ta większa migracja się nie powiedzie, przynajmniej jeden zespół więcej będzie miał sukces na koncie i jeden proces wdrożenia przebiegł lepiej, więc wynik i tak był wart poniesionych nakładów.

Zapewnienie zespołowi czasu na naukę

Na tym etapie powinno być już jasne (mam nadzieję), że adaptacja podejścia IaC może być poważną inwestycją. Zmiana podejścia nie jest operacją, którą można przeprowadzić z dnia na dzień. To nie jest również operacja, która odbędzie się w sposób magiczny tylko dlatego, że Twój szef się na to zgodził. Będzie wymagała znacznego wysiłku ze strony całego zespołu, zapewnienia zasobów szkoleniowych (np. dokumentacji, samouczków wideo i oczywiście tej książki), a także poświęcenia nieco czasu każdemu członkowi zespołu, aby pomóc mu w rozpoczęciu pracy z nową technologią.

Jeżeli zespół nie będzie miał wystarczająco czasu i zasobów, migracja IaC prawdopodobnie zakończy się niepowodzeniem. Niezależnie od tego, jak dobry kod stworzysz, jeśli cały zespół nie będzie stosował tego samego podejścia, operacja się nie uda. Zapoznaj się z przykładowym scenariuszem prowadzącym do nieudanej migracji:

1. Jeden z programistów zespołu jest pasjonatem podejścia typu IaC, poświęcił kilka miesięcy na tworzenie świetnego kodu Terraform, który wykorzystał do wdrożenia sporej ilości infrastruktury.
2. Ten programista jest szczęśliwy i produktywny, ale niestety pozostała część zespołu nie ma czasu na poznanie i zaadaptowanie Terraform.
3. Pewnego dnia nadchodzi nieunikniona awaria.
4. Teraz jeden z innych członków zespołu musi poradzić sobie z tą awarią. Do dyspozycji są dwie możliwości. Pierwsza: usunięcie problemu w zwykły sposób, stosowany „od zawsze”, przez

ręczne wprowadzenie zmian, co powinno zabrać kilka minut. Druga: usunięcie problemu przez wykorzystanie Terraform — ponieważ nie zna tego narzędzia, operacja może zabrać godziny lub dni. Członkowie zespołu są prawdopodobnie rozsądnymi ludźmi i dlatego niemal zawsze zostanie wybrana pierwsza możliwość.

5. W wyniku ręcznie wprowadzonej zmiany kod Terraform już dłużej nie odpowiada temu, co stało faktycznie wdrożone. Dlatego też, jeśli ktokolwiek w zespole zdecyduje się na drugą możliwość, to istnieje niebezpieczeństwo, że natrafi na dziwny błąd. W takim przypadku straci zaufanie do kodu Terraform i ponownie powróci do pierwszej opcji, a następnie ręcznie wprowadzi zmiany. Te zmiany oznaczają jeszcze większe oderwanie kodu Terraform od rzeczywistego wdrożenia, więc następna osoba zetknie się z kolejnymi dziwnymi błędami. Skutkiem będzie cykl, w którym członkowie zespołu będą ręcznie wprowadzali kolejne zmiany.
6. Bardzo szybko wszyscy powracają do ręcznego wprowadzania zmian, kod Terraform staje się zupełnie nieprzydatny, a miesiące poświęcone na jego utworzenie są całkowicie stracone.

To nie jest tylko hipotetyczny scenariusz, ale taki, z jakim spotykałem się wielokrotnie w różnych firmach. Mają one ogromne, kosztowne bazy kodu zawierające mnóstwo eleganckiego kodu Terraform, który pozostaje nieużywany. Aby uniknąć takiej sytuacji, trzeba do Terraform przekonać nie tylko szefa, ale również pozostałych członków zespołu oraz zapewnić im czas potrzebny na poznanie narzędzi i sposobu ich działania. Dzięki temu po wystąpieniu kolejnej awarii będą w stanie ją usunąć poprzez wprowadzenie zmian w kodzie Terraform, a nie bezpośrednio w infrastrukturze.

W szybszym zaadaptowaniu IaC może pomóc doskonale zdefiniowany proces zastosowania tego podejścia. Gdy dopiero poznasz IaC lub uczysz się w małym zespole, uruchamianie Terraform tymczasowo w komputerze programisty jest dobrym rozwiązaniem. Jednak wraz ze wzrostem stopnia użycia IaC w firmie konieczne jest zdefiniowanie bardziej systematycznego, powtarzalnego i zautomatyzowanego sposobu przeprowadzania wdrożeń.

Sposób pracy podczas wdrażania kodu aplikacji

W tym podrozdziale przedstawię typowy sposób pracy pozwalający na przekazanie kodu aplikacji (np. Ruby on Rails, Javy, JavaScriptu itd.) ze środowiska programistycznego do produkcyjnego. Zaprezentowana tutaj metoda jest dość często stosowana w świecie DevOps, więc prawdopodobnie powinieneś już znać jej etapy. W dalszej części rozdziału przejdę do sposobu pracy podczas przekazywania kodu infrastruktury (np. modułów Terraform) ze środowiska programistycznego do produkcyjnego. Ten sposób pracy nie jest aż tak powszechnie stosowany w branży, więc dobrze jest go porównać ze stosowanym podczas przekazywania kodu aplikacji. To pozwoli zrozumieć, jak kroki w procedurze dla aplikacji przekładają się na kroki w procedurze dla infrastruktury.

Spójrz na przykładowy sposób pracy z kodem aplikacji:

1. Użycie systemu kontroli wersji.
2. Lokalne uruchomienie kodu.
3. Wprowadzenie zmian w kodzie.
4. Przekazanie zmian do zatwierdzenia.

5. Uruchomienie testów zautomatyzowanych.
6. Połączenie kodu istniejącego z nowym i wydanie produktu.
7. Wdrożenie.

Teraz omówię te kroki po kolei.

Użycie systemu kontroli wersji

Cały kod powinien zostać umieszczony w systemie kontroli wersji. Żadnych wyjątków. To był numer 1 na klasycznej liście Joel Test (<https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/>), którą Joel Spolsky utworzył prawie 20 lat temu. Jedyne zmiany od tego czasu to: (a) wraz z pojawieniem się narzędzi takich jak GitHub praca z systemem kontroli wersji jest jeszcze łatwiejsza i (b) coraz więcej można przedstawić za pomocą kodu. To obejmuje dokumentację (np. pliki typu README utworzone w formacie Markdown), konfigurację aplikacji (np. pliki konfiguracyjne utworzone w formacie YAML), specyfikację (np. kod testu utworzony za pomocą RSpec), testy (np. testy zautomatyzowane utworzone za pomocą JUnit), bazy danych (np. utworzone w Active Record migracje schematu) oraz oczywiście infrastrukturę.

Podobnie jak w pozostałej części książki, przyjąłem założenie, że używasz Git jako systemu kontroli wersji. Dla przykładu — oto polecenie pozwalające na pobranie fragmentów kodu przygotowanych dla książki:

```
$ git clone https://github.com/brikis98/terraform-up-and-running-code.git
```

Domyślnie powoduje pobranie kodu z gałęzi master repozytorium, choć prawdopodobnie cała Twoja praca będzie znajdowała się w oddzielnej gałęzi. Spójrz na polecenie pozwalające na utworzenie nowej gałęzi o nazwie example-feature i przejście do niej za pomocą polecenia git checkout:

```
$ cd terraform-up-and-running-code
$ git checkout -b example-feature
Switched to a new branch 'example-feature'
```

Lokalne uruchomienie kodu

Gdy kod znajduje się w Twoim komputerze, możesz go uruchomić lokalnie. Przypomnij sobie z rozdziału 7., że przykładowy serwer WWW utworzony w języku Ruby można uruchomić za pomocą następującego polecenia:

```
$ cd code/ruby/08-terraform/team
$ ruby web-server.rb

[2019-06-15 15:43:17] INFO WEBrick 1.3.1
[2019-06-15 15:43:17] INFO ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]
[2019-06-15 15:43:17] INFO WEBrick::HTTPServer#start: pid=28618 port=8000
```

Teraz można przeprowadzić ręczny test, wykorzystując do tego polecenie curl.

```
$ curl http://localhost:8000
Witaj, świecie
```

Ewentualnie możesz uruchomić testy zautomatyzowane.


```
$ ruby web-server-test.rb
```

```
(...)
```

```
Finished in 0.633175 seconds.
```

```
-----  
8 tests, 24 assertions, 0 failures, 0 errors  
100% passed  
-----
```

Kluczowe znaczenie ma tutaj dostrzeżenie, że testy — zarówno ręczne, jak i zautomatyzowane — kodu aplikacji mogą działać całkowicie lokalnie w Twoim komputerze. Z dalszej części rozdziału dowiesz się, że nie dotyczy to sposobu pracy ze zmieniającą się infrastrukturą.

Wprowadzenie zmian w kodzie

Skoro możesz uruchomić kod aplikacji, masz również możliwość wprowadzania w nim zmian. To jest proces iteracyjny polegający na wprowadzeniu zmiany, ponownym uruchomieniu testów ręcznych lub zautomatyzowanych w celu sprawdzenia jej poprawności, wprowadzenie kolejnej zmiany, ponowne wykonanie testów itd.

Przykładowo można zmienić dane wyjściowe skryptu *web-server.rb* w taki sposób, aby był wyświetlany komunikat *Witaj, świecie v2*, a następnie ponownie uruchomić serwer i sprawdzić wynik całej operacji.

```
$ curl http://localhost:8000
```

```
Witaj, świecie v2
```

Ponownie uruchom testy i zobacz, czy będą zaliczone.

```
$ ruby web-server-test.rb
```

```
(...)
```

```
Failure: test_integration_hello(TestWebServer)  
web-server-test.rb:53:in `block in test_integration_hello'  
  50:   do_integration_test('/', lambda { |response|  
  51:     assert_equal(200, response.code.to_i)  
  52:     assert_equal('text/plain', response['Content-Type'])  
=> 53:     assert_equal('Witaj, świecie', response.body)  
  54:   })  
  55: end  
  56:  
web-server-test.rb:100:in `do_integration_test'  
web-server-test.rb:50:in `test_integration_hello'  
<"Witaj, świecie"> expected but was  
<"Witaj, świecie v2">
```

```
(...)
```

```
Finished in 0.236401 seconds.
```

```
-----  
8 tests, 24 assertions, 2 failures, 0 errors  
75% passed  
-----
```

Natychmiast otrzymujesz informację, że testy zautomatyzowane nadal oczekują komunikatu w poprzedniej postaci, więc szybko możesz wprowadzić niezbędną zmianę. Idea polega na zoptymalizowaniu sposobu pracy i maksymalnym skróceniu czasu między wprowadzeniem zmiany i sprawdzeniem, czy działa ona zgodnie z oczekiwaniami.

W trakcie pracy powinieneś regularnie przekazywać kod do repozytorium wraz z jasnymi komunikatami dotyczącymi wprowadzonych zmian.

```
$ git commit -m "Uaktualniony komunikat Witaj, świecie"
```

Przekazanie zmian do zatwierdzenia

Ostatecznie kod i testy będą działały w oczekiwany sposób, więc nadejdzie pora na zatwierdzenie zmian i przejrzanie kodu. Do tego celu można wykorzystać oddzielne narzędzie (np. Phabricator lub ReviewBoard) lub — jeśli używasz serwisu GitHub — masz możliwość utworzenia tzw. *żądania aktualizacji* (ang. *pull request*). Istnieje wiele różnych sposobów na przygotowanie żądania aktualizacji. Jednym z najłatwiejszych jest użycie polecenia `git push` wraz z nazwą gałęzi (tutaj `example-feature`) i `origin` (czyli przekazanie z powrotem do serwisu GitHub), a adres URL tego żądania zostanie automatycznie umieszczony w danych wyjściowych.

```
$ git push origin example-feature
```

```
(...)
```

```
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
```

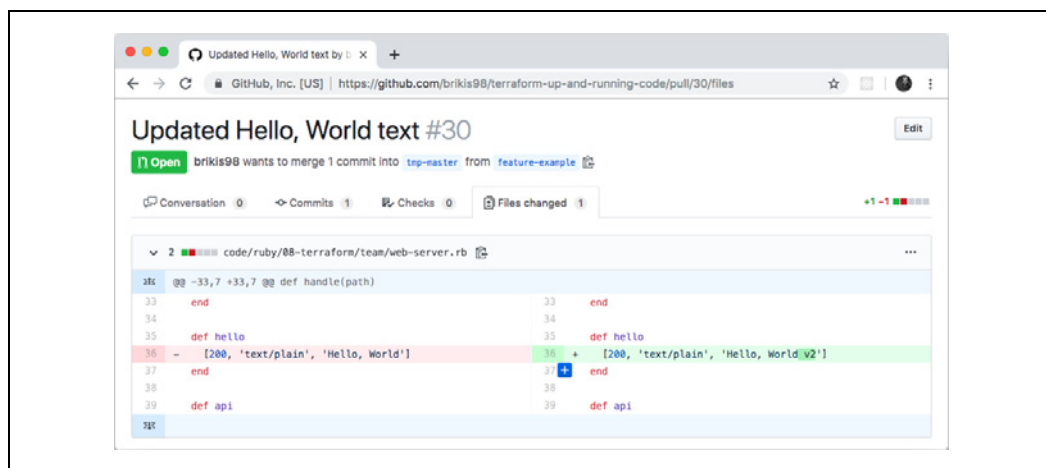
```
remote:
```

```
remote: Create a pull request for 'example-feature' on GitHub by visiting:
```

```
remote:      https://github.com/<WŁAŚCICIEL>/<REPOZYTORIUM>/pull/new/example-feature
```

```
remote:
```

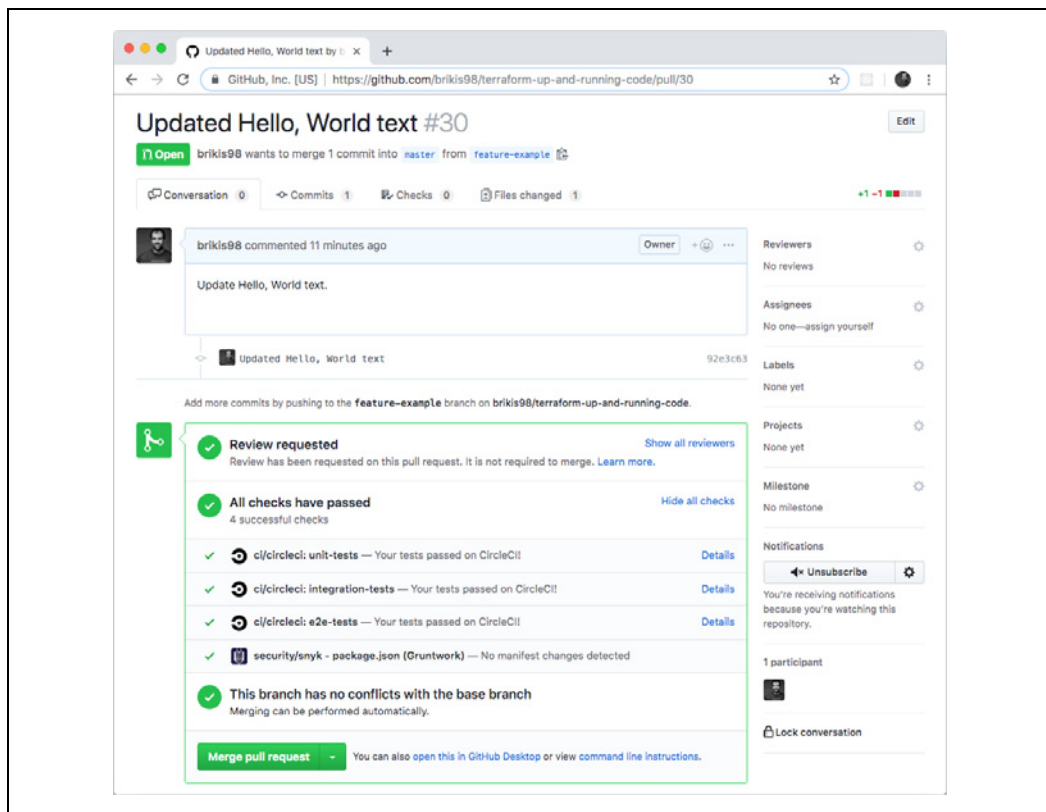
Otwórz ten adres URL w przeglądarce WWW, wypełnij pola tytułu i opisu, a następnie kliknij przycisk *Create*. Członkowie Twojego zespołu będą mogli teraz przejrzeć zmiany, jak pokazałem na rysunku 8.1.



Rysunek 8.1. Żądanie aktualizacji w GitHub

Uruchomienie testów zautomatyzowanych

Powinieneś przygotować zaczepy zatwierdzenia pozwalające na wykonywanie testów zautomatyzowanych po każdej operacji zatwierdzenia kodu w systemie kontroli wersji. Najczęściej stosowane rozwiązanie polega na użyciu serwera *ciągłej integracji* (ang. *continuous integration*, CI), takiego jak Jenkins, CircleCI lub TravisCI. Większość popularnych serwerów CI oferuje możliwość integracji z serwisem GitHub, więc po każdym przekazaniu kodu do repozytorium zostanie automatycznie zainicjowane nie tylko wykonanie testów, ale również umieszczenie wyników testów w samym żądaniu aktualizacji, jak pokazałem na rysunku 8.2.



Rysunek 8.2. Żądanie aktualizacji w serwisie GitHub pokazuje wyniki testów zautomatyzowanych z CircleCI

Na podstawie rysunku 8.2 wyraźnie widać, że serwer CircleCI wykonał testy jednostkowe, testy integracji, testy typu E2E oraz kilka innych statycznych operacji sprawdzenia (w postaci użycia narzędzia snyk do przeskanowania kodu pod względem luk w zabezpieczeniach) kodu w gałęzi i wszystkie te testy zostały zaliczone.

Połączenie kodu istniejącego z nowym i wydanie produktu

Członkowie Twojego zespołu powinni przejrzeć zmiany wprowadzone w kodzie, znaleźć potencjalne błędy, wymusić stosowanie odpowiednich reguł podczas tworzenia kodu (więcej informacji na ten

temat znajdziesz w dalszej części rozdziału), sprawdzić, czy testy zostały zaliczone oraz czy na pewno przygotowałeś testy do nowej funkcjonalności. Jeżeli wszystko wygląda prawidłowo, utworzony przez Ciebie kod może zostać dołączony do gałęzi master.

Następnym krokiem jest wydanie kodu. Jeżeli stosuje praktyki niemodyfikowalnej infrastruktury (wspomniałem o tym w rozdziale 1.), wydanie kodu aplikacji oznacza pakowanie kodu do nowego, niemodyfikowalnego i wersjonowanego produktu. W zależności od sposobu pakowania i wdrażania aplikacji tym nowym produktem może być nowy obraz Dockera, nowy obraz maszyny wirtualnej (np. AMI), nowy plik *.jar*, nowy plik *.tar* itd. Niezależnie od wybranego formatu należy się upewnić o utworzeniu niemodyfikowalnego produktu (nigdy go nie zmieniasz) i nadaniu mu unikatowego numeru wersji (aby można było odróżniać dany produkt od pozostałych).

Przykładowo, jeżeli do pakowania aplikacji jest używany Docker, numer wersji można przechowywać w tagu Dockera. Istnieje możliwość użycia identyfikatora operacji zatwierdzenia (wartość hash sha1) jako tagu, co pozwala na mapowanie wdrażanego obrazu Dockera na dokładny kod, który znalazł się w danym obrazie.

```
$ commit_id=$(git rev-parse HEAD)
$ docker build -t brikis98/ruby-web-server:$commit_id .
```

Te polecenia powodują utworzenie nowego obrazu Dockera o nazwie brikis98/ruby-web-server i tagu wraz z identyfikatorem ostatniej operacji zatwierdzenia, który ma postać podobną do następującej: 92e3c6380ba6d1e8c9134452ab6e26154e6ad849. Później w trakcie rozwiązywania problemu związanego z obrazem Dockera można przeanalizować umieszczony w nim kod przez porównanie identyfikatora operacji zatwierdzenia i tagu obrazu Dockera.

```
$ git checkout 92e3c6380ba6d1e8c9134452ab6e26154e6ad849
HEAD is now at 92e3c63 Uaktualniony komunikat Witaj, świecie
```

Wadą identyfikatora operacji zatwierdzenia jest to, że nie należy on do zbyt czytelnych lub możliwych do zapamiętania. Alternatywą jest utworzenie znacznika systemu Git.

```
$ git tag -a "v0.0.4" -m "Uaktualnienie komunikatu Witaj, świecie"
$ git push --follow-tags
```

Tag jest wskaźnikiem prowadzącym do określonej operacji zatwierdzenia w systemie Git, ale ma znacznie bardziej przyjazną nazwę. Ten tag Git można wykorzystać w obrazach Dockera.

```
$ git_tag=$(git describe --tags)
$ docker build -t brikis98/ruby-web-server:$git_tag .
```

Dlatego też podczas procesu debugowania należy pobrać kod oznaczony określonym tagiem.

```
$ git checkout v0.0.4
Note: checking out 'v0.0.4'.
(...)
HEAD is now at 92e3c63 Uaktualniony komunikat Witaj, świecie
```

Wdrożenie

Mając przygotowany wersjonowany produkt, można przystąpić do jego wdrożenia. Istnieje wiele różnych sposobów na wdrażanie kodu aplikacji, w zależności od jej typu, pakowania, sposobu

uruchamiania aplikacji, architektury sprzętowej, używanych narzędzi itd. Oto kilka kwestii do rozważenia:

- narzędzia wdrażania,
- strategie wdrażania,
- serwer wdrożenia,
- stosowanie produktu w różnych środowiskach.

Narzędzia wdrażania

Dostępnych jest wiele różnych narzędzi możliwych do zastosowania podczas wdrażania aplikacji, w zależności od sposobu jej pakowania i uruchamiania. W tym punkcie przedstawiłem tylko kilka wybranych:

Terraform

Jak miałeś okazję zobaczyć w książce, Terraform pozwala na wdrażanie określonych typów aplikacji. Przykładowo we wcześniejszych rozdziałach utworzyłeś moduł o nazwie `asg-rolling-deploy` umożliwiający wdrożenie bez przestoju w grupie ASG. Jeżeli aplikację pakujesz na postać obrazu AMI (np. za pomocą narzędzia Packer), podczas wdrożenia nowej wersji obrazu AMI z modulem `asg-rolling-deploy` można uaktualnić wartość parametru `ami` w kodzie Terraform i wydać polecenie `terraform apply`.

Narzędzia instrumentacji Dockera

Istnieje wiele różnych narzędzi instrumentacji zaprojektowanych do wdrażania i zarządzania aplikacjami umieszczonymi w kontenerach Dockera — m.in. Kubernetes (bez wątpienia to najpopularniejsze narzędzie), Apache Mesos, HashiCorp Nomad i Amazon ECS. Jeżeli aplikację pakujesz na postać obrazu Dockera, podczas wdrożenia nowej wersji tego obrazu za pomocą Kubernetesa możesz wydać polecenie `kubectl apply` (`kubectl` to aplikacja powłoki pozwalająca na pracę z Kubernetesem) i przekazać plik typu YAML definiujący nazwę obrazu i tagu do wdrożenia.

Skrypty

Terraform i większość narzędzi instrumentacji obsługuje jedynie ograniczoną liczbę strategii (do tego tematu powrócę w następnym punkcie). Jeżeli masz bardziej skomplikowane wymagania, prawdopodobnie będziesz musiał tworzyć własne skrypty w języku programowania ogólnego przeznaczenia (takim jak Python, Ruby itd.), za pomocą narzędzi zarządzania konfiguracją (np. Ansible, Chef) lub innych narzędzi automatyzacji serwera (np. Capistrano).

Najtrudniejszym zadaniem podczas tworzenia takich skryptów jest obsługa niepowodzeń. Co się stanie np. w sytuacji, gdy komputer wykonujący skrypt wdrożenia utraci połączenie z internetem lub w trakcie procesu ulegnie awarii? Utworzenie skryptu zapewniającego powtarzalność oraz naprawę po awarii i możliwość prawidłowego dokończenia wdrożenia nie jest łatwa. Skryptowi trzeba zapewnić możliwość rejestrowania gdzieś informacji o stanie (czasami te informacje można ustalić przez sprawdzenie infrastruktury) oraz przygotować tzw. automat skończony

(ang. *finite state machine*), który będzie potrafił obsłużyć wszystkie możliwe stany początkowe i przejściowe.

Strategie wdrażania

Opracowano wiele różnych strategii, które można wykorzystać podczas wdrażania aplikacji, w zależności od zdefiniowanych wymagań. Przyjmuję założenie, że istnieje pięć kopii starych wersji aplikacji i chcesz wdrożyć najnowszą z nich. Oto kilka strategii, z których możesz skorzystać:

Wdrażanie nieustanne wraz zastępowaniem

Wycofaj jedną ze starych kopii aplikacji, przeprowadź wdrożenie nowej kopii zastępującej starą, poczekaj na udostępnienie tej nowej kopii i zaliczenie przez nią operacji sprawdzenia stanu, rozpocznij przekazywanie nowej kopii odbiorcom, a następnie powtarzaj cały proces aż do chwili zastąpienia wszystkich starych kopii. Omawiany tutaj rodzaj wdrożenia gwarantuje, że nigdy nie będziesz miał więcej niż pięć kopii uruchomionej aplikacji, co może być użyteczne w sytuacji, gdy dysponujesz ograniczonymi zasobami (np. każda kopia aplikacji została uruchomiona w fizycznym serwerze) lub jeśli masz do czynienia z systemem, w którym każda aplikacja ma unikatową tożsamość (tak się często dzieje w przypadku usług systemów rozproszonych, takich jak Apache ZooKeeper). Warto w tym miejscu dodać, że ta strategia wdrożenia działa w przypadku większej liczby zastępowanych kopii (np. próbujesz zastąpić jednocześnie więcej niż tylko jedną kopię aplikacji przy założeniu o możliwości obsługi danego obciążenia i zachowaniu danych w przypadku uruchomienia mniejszej liczby aplikacji) oraz gdy w trakcie wdrożenia jednocześnie masz uruchomione stare i nowe wersje aplikacji.

Wdrażanie nieustanne bez zastępowania

Przeprowadź wdrożenie nowej kopii aplikacji, poczekaj na udostępnienie tej nowej kopii i zaliczenie przez nią operacji sprawdzenia stanu, rozpocznij przekazywanie nowej kopii odbiorcom, wycofaj starą kopię aplikacji, a następnie powtarzaj cały proces aż do chwili zastąpienia wszystkich starych kopii. Ta strategia sprawdzi się tylko wtedy, gdy dysponujesz elastyczną pojemnością (np. aplikacje zostały uruchomione w chmurze, co pozwala na uruchamianie nowych i zatrzymywanie istniejących serwerów wirtualnych na żądanie) oraz gdy aplikacja może tolerować więcej niż pięć jej kopii działających jednocześnie. Zaletą takiego rozwiązania jest to, że nigdy nie masz mniej niż pięć kopii uruchomionej aplikacji i nie musisz zapewnić funkcjonowania w warunkach zmniejszonej pojemności podczas wdrożenia. Warto w tym miejscu dodać, że ta strategia wdrożenia działa w przypadku większej liczby wdrażanych kopii (np. możesz jednocześnie wdrażać pięć nowych kopii, czyli dokładnie tak, jak to zostało zrobione w przypadku modułu `asg-rolling-deploy`) oraz gdy w trakcie wdrożenia jednocześnie masz uruchomione stare i nowe wersje aplikacji.

Wdrożenie typu niebieski-zielony

Przeprowadź wdrożenie pięciu nowych kopii aplikacji, poczekaj na udostępnienie tych nowych kopii i zaliczenie przez nie operacji sprawdzenia stanu, rozpocznij przekazywanie nowych kopii odbiorcom, a następnie wycofaj stare kopie. Ta strategia sprawdzi się tylko wtedy, gdy dysponujesz elastyczną pojemnością (np. aplikacje zostały uruchomione w chmurze, co pozwala na uruchamianie nowych i zatrzymywanie istniejących serwerów wirtualnych na żądanie)

oraz gdy aplikacja może tolerować więcej niż pięć jej kopii działających jednocześnie. Zaletą takiego rozwiązania jest to, że w danej chwili dostępna jest tylko jedna wersja aplikacji dla użytkowników i nigdy nie będziesz mieć do dyspozycji mniej niż pięć kopii uruchomionej aplikacji, więc podczas wdrożenia nie musisz się martwić działaniem w warunkach zmniejszonej pojemności.

Wdrożenie kanarkowe

Przeprowadź wdrożenie jednej nowej kopii aplikacji, poczekaj na udostępnienie tej nowej kopii i zaliczenie przez nią operacji sprawdzenia stanu, rozpocznij przekazywanie nowej kopii odbiorcom, a następnie wstrzymaj wdrożenie. W trakcie tej przerwy porównaj nową kopię aplikacji, nazywaną „kanarkową”, ze starymi kopiami, nazywanymi „kontrolnymi”. Oba rodzaje kopii można porównywać pod wieloma względami: poziomu użycia procesora i pamięci, opóźnienia, przepustowości, poziomu błędów w dziennikach zdarzeń, kodów stanu HTTP w udzielanych odpowiedziach itd. W idealnej sytuacji trudno będzie odróżnić od siebie dwa serwery, co powinno gwarantować, że nowy kod działa doskonale. W takim przypadku wznów wdrożenie i wykorzystaj jedną ze strategii wdrożenia ciągłego, aby dokończyć proces. Jeśli natomiast znajdziesz jakieś różnice, może to świadczyć o problemach z nowym kodem — należy anulować wdrożenie i wycofać aplikację kanarkową, zanim problem stanie się znacznie poważniejszy.

Nazwa tego wdrożenia pochodzi od kanarków w kopalni, które górnicy zabierali ze sobą pod ziemię. Jeżeli w tunelach znajdowały się niebezpieczne gazy (np. dwutlenek węgla), zabijały one kanarki, zanim zabiły górników. Ptaki były więc rodzajem systemu wczesnego ostrzegania górników, że dzieje się coś złego i należy natychmiast wydostać się na powierzchnię. Wdrożenie kanarkowe oferuje podobne korzyści — zapewnia systematyczny sposób na przetestowanie nowego kodu w środowisku produkcyjnym w taki sposób, że jeśli coś pójdzie źle, otrzymasz wczesne ostrzeżenie, o ile problem dotyczy jedynie małej grupy użytkowników i nadal jest wystarczająco dużo czasu na reakcję i uniknięcie dalszych szkód.

Wdrożenie kanarkowe jest często łączone z tzw. *przełącznikami funkcjonalności*, gdy nowe funkcje są opakowywane konstrukcjami warunkowymi i f. Domyślnie konstrukcja warunkowa i f przyjmuje wartość false, więc nowa funkcjonalność będzie niedostępna po początkowym wdrożeniu kodu. Skoro nowa funkcjonalność jest niedostępna, po wdrożeniu serwera kanarkowego powinien działać identycznie jak starsze, a ewentualne różnice mogą być natychmiast oznaczane jako problem i prowadzić do wycofania. Jeżeli nie wystąpiły żadne problemy, można włączyć daną funkcjonalność dla części użytkowników za pomocą wewnętrznego interfejsu opartego na przeglądarce WWW. Przykładowo początkowo funkcjonalność może zostać wdrożona jedynie dla pracowników. Jeżeli wszystko działa zgodnie z oczekiwaniami, można ją włączyć dla 1% użytkowników. Jeżeli ta funkcjonalność nadal działa dobrze, można ją wprowadzić dla 10% użytkowników itd. Jeśli na którymkolwiek etapie wystąpi problem, przełącznik funkcjonalności pozwala na wyłączenie nowej funkcji. Taki proces umożliwia oddzielenie *wdrożenia* nowego kodu od *udostępnienia* nowych funkcji.

Serwer wdrożenia

Wdrożenie powinno zostać przeprowadzone z poziomu serwera CI, a nie komputera programisty, ponieważ wiąże się to z wymienionymi tutaj korzyściami:

Pełna automatyzacja

Aby przeprowadzić wdrożenie z poziomu serwera CI, będziesz zmuszony do pełnej automatyzacji wszystkich etapów wdrożenia. To gwarantuje zdefiniowanie procesu wdrożenia jako kodu, co chroni przed przypadkowym pominięciem któregoś z etapów na skutek błędu, a samo wdrożenie stanie się szybkie i powtarzalne.

Zainicjowanie procesu ze spójnego środowiska

Jeżeli programiści będą przeprowadzali wdrożenia z poziomu swoich komputerów, zacząłby pojawiać się błędy wynikające z odmiennych sposobów konfiguracji poszczególnych komputerów, np. różnych systemów operacyjnych, różnych wersji zależności (różnych wersji Terraform), różnych konfiguracji i różnic w faktycznie wdrażanych komponentach (np. programista przypadkowo wdroży zmianę nieprzekazaną do systemu kontroli wersji). Wszystkie te problemy można wyeliminować dzięki przeprowadzaniu wdrożenia z poziomu jednego i tego samego serwera CI.

Lepsze zarządzanie uprawnieniami

Zamiast udzielać każdemu programiście uprawnienia do wdrażania, takie uprawnienia można nadać jedynie serwerowi CI (zwłaszcza jeśli chodzi o uprawnienia w środowisku produkcyjnym). Stosowanie dobrych praktyk w zakresie bezpieczeństwa znacznie łatwiej jest wymusić w pojedynczym serwerze niż wśród dziesiątek lub setek programistów dysponujących dostępem do środowiska produkcyjnego.

Stosowanie produktu w różnych środowiskach

Jeżeli stosowane są praktyki infrastruktury niemodyfikowalnej, przekazanie nowych zmian do wszystkich środowisk oznacza przekazywanie dokładnie tego samego wersjonowanego produktu. Przykładowo, jeśli istnieją środowiska programistyczne, robocze i produkcyjne, wydanie wersji 0.0.4 aplikacji będzie przebiegało według następującego schematu:

1. Wdrożenie wersji 0.0.4 aplikacji w środowisku programistycznym.
2. Uruchomienie w środowisku programistycznym testów ręcznych i zautomatyzowanych.
3. Jeżeli wersja 0.0.4 działa świetnie w środowisku programistycznym, kroki 1. i 2. należy powtórzyć podczas wdrażania wersji 0.0.4 aplikacji w środowisku roboczym (nosi to nazwę *promowania* produktu).
4. Jeżeli wersja 0.0.4 działa świetnie w środowisku roboczym, kroki 1. i 2. należy powtórzyć podczas promowania wersji 0.0.4 aplikacji do środowiska produkcyjnego.

Skoro we wszystkich środowiskach używany jest dokładnie ten sam produkt, to istnieje ogromne prawdopodobieństwo, że jeśli działa on dobrze w jednym środowisku, będzie działał równie dobrze w pozostałych środowiskach. W przypadku jakichkolwiek problemów zawsze można wycofać produkt i wdrożyć jego starszą wersję.

Sposób pracy podczas wdrażania kodu infrastruktury

W poprzednim podrozdziale zapoznałeś się ze sposobem pracy podczas wdrażania kodu aplikacji, więc teraz przechodzimy do sposobu pracy w trakcie wdrażania kodu infrastruktury. Gdy w tym podrozdziale wymieniam kod infrastruktury, mam na myśli kod utworzony za pomocą dowolnego narzędzia IaC (włączając w to oczywiście Terraform) i pozwalający na wdrażanie dowolnych zmian infrastruktury poza pojedynczą aplikacją. Przykładowo może to być wdrażanie baz danych, mechanizmów równoważenia obciążenia, konfiguracji sieci, ustawień DNS itd.

Spójrz na przykładowy sposób pracy z kodem infrastruktury:

1. Użycie systemu kontroli wersji.
2. Lokalne uruchomienie kodu.
3. Wprowadzenie zmian w kodzie.
4. Przekazanie zmian do zatwierdzenia.
5. Uruchomienie testów zautomatyzowanych.
6. Połączenie kodu istniejącego z nowym i wydanie produktu.
7. Wdrożenie.

Na pierwszy rzut oka może się wydawać, że sposób pracy jest identyczny z tym, który stosujemy dla kodu aplikacji. Jednak istnieją między nimi duże różnice. Wdrażanie zmian kodu infrastruktury jest znacznie bardziej skomplikowane, a stosowane w trakcie techniki nie są doskonale znane. Dlatego też możliwość powiązania poszczególnych kroków z odpowiadającymi im krokami, które zostały zdefiniowane w kodzie aplikacji, ułatwia procedurę wdrażania kodu infrastruktury. Przechodzę teraz do omówienia wymienionych tutaj kroków.

Użycie systemu kontroli wersji

Tak jak w przypadku kodu aplikacji, także cały kod infrastruktury powinien zostać umieszczony w systemie kontroli wersji. To oznacza możliwość wydania polecenia `git clone` w celu pobrania kodu, podobnie jak wcześniej. Jednak w przypadku systemu kontroli wersji dla kodu infrastruktury pod uwagę trzeba wziąć kilka dodatkowych kwestii:

- repozytoria *live* i *modules*,
- złotą regułę Terraform,
- problemy z gałęziami.

Repozytoria live i modules

Jak już wspomniałem w rozdziale 4., zwykle będziesz miał co najmniej dwa oddzielne repozytoria systemu kontroli wersji: dla modułów i oddzielne dla kodu wdrożonej infrastruktury. Repozytorium modułów służy do tworzenia wielokrotnego użycia, wersjonowanych modułów podobnych do tych, które tworzyłeś we wcześniejszych rozdziałach książki (`cluster/asg-rolling-deploy`, `data-stores/mysql`,

networking/alb i services/hello-world-app). Repozytorium typu *live* definiuje infrastrukturę wdrażaną w poszczególnych środowiskach (programistycznym, roboczym, produkcyjnym itd.).

Jednym ze wzorców, które doskonale się sprawdzają, jest przygotowanie w firmie zespołu odpowiedzialnego za infrastrukturę i specjalizującego się w tworzeniu wielokrotnego użycia, niezawodnych i charakteryzujących się jakością produkcyjną modułów. Ten zespół może przynieść dużą korzyść firmie poprzez opracowanie biblioteki modułów implementującej idee przedstawione w rozdziale 6.: każdy moduł oferuje API pozwalające na łączenie z innymi modułami, ma dokładną dokumentację (łącznie z umieszczoną w katalogu *examples* dokumentacją wykonywalną), zawiera rozbudowany zestaw testów zautomatyzowanych, stosuje wersjonowanie oraz implementuje wszystkie wymagania firmy dotyczące kodu infrastruktury o jakości produkcyjnej (np. zapewnienia bezpieczeństwa, zgodność, skalowalność, wysoką dostępność, monitorowanie itd.).

Jeżeli taką bibliotekę budujesz zupełnie od podstaw (lub kupujesz gotową³), wszystkie pozostałe zespoły w firmie będą mogły korzystać z tych modułów, wybierać je jak z katalogu usług oraz wdrażać własną infrastrukturę i nią zarządzać. To nie będzie wymagało poświęcania miesięcy pracy zespołu na przygotowanie całej infrastruktury zupełnie od początku. Także zespół DevOps nie stanie się wąskim gardłem z powodu konieczności wdrażania i zarządzania infrastrukturą dla każdego zespołu. Zamiast tego zespół DevOps może zająć się tworzeniem kodu infrastruktury, a wszystkie pozostałe zespoły mogą pracować niezależnie, wykorzystując wspomniane moduły. Skoro każdy zespół będzie używał tych samych modułów kanonicznych, wraz z rozwojem firmy i ze zmianą wymagań zespół DevOps może przygotować nowe wersje modułów dla wszystkich zespołów, a jednocześnie zagwarantować, że wszystko pozostanie spójne i możliwe do konserwacji.

Ta konserwacja będzie możliwa dopóty, dopóki jest stosowana złota reguła Terraform.

Złota reguła Terraform

Oto szybki sposób na sprawdzenie stanu kodu Terraform: przejdź do repozytorium *live*, losowo wybierz kilka katalogów, a następnie w każdym z nich wydaj polecenie `terraform apply`. Jeżeli dane wyjściowe będą za każdym razem zawierały komunikat „no changes”, czyli „bez zmian”, to jest doskonała wiadomość oznaczająca, że infrastruktura odpowiada temu, co faktycznie zostało wdrożone. Jeśli natomiast dane wyjściowe wskazują na drobne różnice i spotykasz się z okazijnymi wymówkami ze strony członków zespołu (w stylu „faktycznie, wprowadziłem tę drobną zmianę i zapomniałem ją wprowadzić w kodzie”), to oznacza, że kod nie odpowiada rzeczywistości i wkrótce mogą pojawić się problemy. Jeżeli wykonanie polecenia `terraform plan` kończy się całkowitym niepowodzeniem i dziwnymi błędami lub każde polecenie `terraform plan` powoduje wygenerowanie ogromnej ilości informacji o różnicach, taki kod Terraform nie ma odzwierciedlenia w rzeczywistości i praktycznie jest bezużyteczny.

Złoty standard — lub inaczej: ten, do którego dążysz — to określona przeze mnie tzw. *złota reguła Terraform*:

³ Przygotowana przez Gruntwork biblioteka Infrastructure as Code Library (<https://gruntwork.io/infrastructure-as-code-library/>) to kolekcja zawierająca ponad 300 000 wierszy kodu o jakości produkcyjnej wraz z komercyjną pomocą techniczną. Ta biblioteka kodu infrastruktury wielokrotnego użycia została z powodzeniem wykorzystana w środowiskach produkcyjnych setek firm.

Gałąź *master* w repozytorium *live* powinna w 100% odzwierciedlać to, co zostało faktycznie wdrożone w środowisku produkcyjnym.

Warto dokładniej zapoznać się z wybranymi fragmentami tego zdania:

„to, co zostało faktycznie wdrożone”

Jedynym sposobem na zagwarantowanie, że kod Terraform w repozytorium *live* faktycznie odzwierciedla to, co zostało wdrożone, jest *niewprowadzanie ręcznie zmian, nigdy*. Gdy zaczniesz korzystać z Terraform, nigdy nie wprowadzaj zmian w infrastrukturze za pomocą interfejsu użytkownik w postaci przeglądarki WWW, ręcznych wywołań API lub innego mechanizmu. Jak mogłeś zobaczyć w rozdziale 5., takie zmiany prowadzą nie tylko do skomplikowanych błędów, ale też niwelują wiele korzyści oferowanych przez podejście IaC.

„powinna w 100% odzwierciedlać”

Przeglądając repozytorium *live*, powinieneś mieć możliwość szybkiego ustalenia, które zasoby zostały wdrożone w poszczególnych środowiskach. Dlatego też każdy zasób powinien w 100% odpowiadać pewnym wierszom kodu, które znalazły się w repozytorium *live*. Wprawdzie wydaje się to aż nazbyt oczywiste, jednak mimo to zaskakująco często ta reguła jest łamana. Jednym z przykładów jej łamania jest wprowadzanie ręcznie zmian prowadzących do różnic między kodem i infrastrukturą. Znacznie subtelniejszym przykładem złamania tej reguły jest używanie przestrzeni roboczych do zarządzania środowiskami. Dlatego też, jeżeli korzystasz z przestrzeni roboczych, repozytorium *live* będzie zawierało tylko jedną kopię kodu, nawet jeśli służy on do wdrażania 3 lub 30 środowisk. Patrzenie jedynie na kod nie wystarczy, by ustalić, co właściwie zostało wdrożone, co prowadzi do błędów i utrudnia późniejszą konserwację infrastruktury. Stąd, jak to omówiłem w rozdziale 3., zamiast używania przestrzeni roboczych do zarządzania środowiskami lepszym rozwiązaniem jest zdefiniowanie każdego środowiska w oddzielnych katalogu, za pomocą oddzielnych plików. Dzięki temu, jedynie przeglądając repozytorium *live*, można wyraźnie zobaczyć, które środowisko zostało wdrożone. W dalszej części rozdziału pokażę, jak to zrobić przy minimalnej liczbie operacji kopiowania i wklejania.

„Gałąź *master*”

Do zrozumienia, co naprawdę zostało wdrożone w środowisku produkcyjnym, powinno wystarczyć przeglądanie tylko jednej gałęzi. Zwykle tą gałęzią jest *master*. To oznacza, że wszystkie zmiany wpływające na środowisko produkcyjne powinny trafiać bezpośrednio do gałęzi *master* (wprawdzie można tworzyć oddzielne gałęzie, ale tylko w celu przygotowania żądań aktualizacji przeznaczonych do połączenia z gałęzią *master*). Polecenie `terraform apply` dla środowiska produkcyjnego powinno być wydawane tylko z poziomu gałęzi *master*. W następnym podpunkcie wyjaśnię, dlaczego należy stosować takie podejście.

Problemy z gałęziami

W rozdziale 3. zobaczyłeś, że można skorzystać z mechanizmu nakładania blokad wbudowanego w backendy Terraform. Dzięki temu można mieć pewność, że gdy dwóch członków zespołu w tym samym czasie wyda polecenie `terraform apply` względem tego samego zestawu plików konfiguracyjnych Terraform, ich zmiany nie zostaną wzajemnie nadpisane. Niestety, to jest tylko częściowe rozwiązanie problemu. Nawet pomimo oferowania mechanizmu nakładania blokad na informacje

o stanie Terraform nie pomoże to w nałożeniu blokad na sam kod Terraform. Mam tutaj na myśli to, że jeśli dwóch członków zespołu będzie w tym środowisku wdrażało ten sam kod, ale pochodzący z różnych gałęzi, powstaną konflikty niemożliwe do uniknięcia za pomocą mechanizmu nakładania blokad.

Dla przykładu przyjmuję założenie, że jeden z członków zespołu, Anna, wprowadza pewne zmiany w konfiguracji Terraform dla aplikacji o nazwie foo składającej się z pojedynczego egzemplarza Amazon EC2.

```
resource "aws_instance" "foo" {
  ami      = "ami-0c55b159cbfafa1f0"
  instance_type = "t2.micro"
}
```

Ponieważ ta aplikacja notuje ogromny ruch sieciowy, Anna decyduje się na zmianę typu egzemplarza (wartość `instance_type`) z `t2.micro` na `t2.medium`.

```
resource "aws_instance" "foo" {
  ami      = "ami-0c55b159cbfafa1f0"
  instance_type = "t2.medium"
}
```

Oto dane wyjściowe, które Anna otrzyma po wydaniu polecenia `terraform plan`:

```
$ terraform plan
```

```
(...)
```

Terraform will perform the following actions:

```
# Egzemplarz aws_instance.foo zostanie uaktualniony w miejscu.
~ resource "aws_instance" "foo" {
    ami      = "ami-0c55b159cbfafa1f0"
    id       = "i-096430d595c80cb53"
    instance_state = "running"
    ~ instance_type   = "t2.micro" -> "t2.medium"
    (...)
}
```

Plan: 0 to add, 1 to change, 0 to destroy.

Te zmiany wyglądają prawidłowo, więc Anna przeprowadza wdrożenie w środowisku roboczym.

Tymczasem pojawia się Bartek i również rozpoczyna wprowadzanie zmian w konfiguracji Terraform w tej samej aplikacji, ale w innej gałęzi. Bartek chce jedynie dodać tag do aplikacji.

```
resource "aws_instance" "foo" {
  ami      = "ami-0c55b159cbfafa1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "foo"
  }
}
```

Pamiętaj, że Anna wprowadziła już zmiany w środowisku roboczym, ale ponieważ są one w innej gałęzi, kod Bartka nadal ma atrybut `instance_type` wraz z poprzednią wartością `t2.micro`. Oto dane

wyjściowe, które Anna otrzyma po wydaniu polecenia `terraform plan` (te dane zostały skrócone w celu zapewnienia większej przejrzystości):

```
$ terraform plan
```

```
(...)
```

Terraform will perform the following actions:

```
# Egzemplarz aws_instance.foo zostanie uaktualniony w miejscu.
~ resource "aws_instance" "foo" {
    ami           = "ami-0c55b159cbfaffe1f0"
    id            = "i-096430d595c80cb53"
    instance_state = "running"
~ instance_type  = "t2.medium" -> "t2.micro"
+ tags           = {
    + "Name" = "foo"
  }
  (...)
}
```

Plan: 0 to add, 1 to change, 0 to destroy.

O nie, wygląda na to, że zmiana wprowadzona przez Annę została wycofana. Jeżeli nadal przeprowadza ona testy w środowisku roboczym, będzie bardzo zdziwiona, gdy serwer zostanie nagle ponownie wdrożony i zacznie działać inaczej, niż Anna tego oczekuje. Dobrą wiadomością jest to, że jeśli Bartek dokładnie przeanalizuje dane wyjściowe polecenia `terraform plan`, będzie mógł dostrzec błąd, zanim dotknie on Annę. W tym momencie to bez znaczenia — chcę tutaj zwrócić uwagę na to, co się stanie podczas wprowadzania we współdzielonym środowisku zmian pochodzących z różnych gałęzi.

Oferowany przez backend Terraform mechanizm nakładania blokad nie będzie tutaj pomocny, ponieważ powstały konflikt nie ma nic wspólnego z jednoczesnym wprowadzaniem modyfikacji w pliku informacji o stanie. Bartek i Anna mogli wprowadzić swoje zmiany w odstępie tygodni, a mimo to problem pozostanie ten sam. Źródłem problemu jest to, że gałęzie i Terraform to niedobre połączenie. Terraform to niejawne mapowanie kodu Terraform na infrastrukturę wdrożoną w rzeczywistym świecie. Skoro istnieje tylko jeden rzeczywisty świat, nie ma sensu tworzenie wielu gałęzi w kodzie Terraform. Dlatego też każde środowisko współdzielone (np. robocze, produkcyjne) zawsze jest wdrażane z jednej gałęzi.

Lokalne uruchomienie kodu

Skoro masz już kod w swoim komputerze, kolejnym krokiem jest uruchomienie tego kodu. Trudność w przypadku Terraform polega na tym, że w przeciwieństwie do kodu aplikacji dla Terraform nie istnieje coś takiego jak „komputer lokalny”. Przykładowo nie można wdrożyć grupy ASG w swoim laptopie. Jak już wspomniałem w rozdziale 7., jedynym sposobem na ręczne przetestowanie kodu Terraform jest jego uruchomienie w odizolowanym środowisku, takim jak konto AWS przeznaczone dla programistów (lub jeszcze lepiej po jednym koncie AWS dla każdego programisty).

Gdy mamy przygotowane odizolowane środowisko, w celu ręcznego przetestowania kodu należy wydać polecenie `terraform apply`.

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

Do sprawdzenia wdrożonej infrastruktury można wykorzystać polecenie takie jak `curl`.

```
$ curl hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

```
Witaj, świecie
```

Aby uruchomić testy zautomatyzowane utworzone w języku Go, trzeba wydać polecenie `go test` z poziomu katalogu przeznaczonego dla testów.

```
$ go test -v -timeout 30m
```

```
(...)
```

```
PASS
```

```
ok  terraform-up-and-running 229.492s
```

Wprowadzenie zmian w kodzie

Skoro możesz uruchomić kod Terraform, masz możliwość rozpoczęcia iteracyjnego wprowadzania zmian, podobnie jak w przypadku kodu aplikacji. Po każdej wprowadzonej zmianie można ponownie wydać polecenie `terraform apply` w celu jej wprowadzenia, a następnie użyć polecenia `curl` do sprawdzenia, czy zmiany działają zgodnie z oczekiwaniami.

```
$ curl hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

```
Witaj, świecie v2
```

Eventualnie możesz ponownie wydać polecenie `go test` i upewnić się, że testy wciąż są zaliczane.

```
$ go test -v -timeout 30m
```

```
(...)
```

```
PASS
```

```
ok  terraform-up-and-running 229.492s
```

Jedyna różnica między testami kodu aplikacji i infrastruktury polega na tym, że w przypadku tych drugich procedura zwykle trwa znacznie dłużej, więc należy dobrze przemyśleć, jak można skrócić cykl testu, aby jak najszybciej otrzymać informacje na temat wprowadzonych zmian. Z rozdziału 7. dowiedziałeś się o możliwości wykorzystania etapów testu, co pozwala na wykonywanie tylko wybranych etapów zestawu testu, co z kolei potrafi znacznie skrócić czas wykonywania testów.

Przykładowo, jeśli masz test wdrażający bazę danych, aplikację, weryfikujący ich działania, usuwający aplikację, a później usuwający bazę danych, to podczas początkowego wykonywania testów można pominąć dwa etapy usuwania aplikacji i bazy danych, aby pozostawić je działające.

```
$ SKIP_teardown_db=true \  
  SKIP_teardown_app=true \  
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
```

(...)

```
PASS  
ok   terraform-up-and-running 423.650s
```

Następnie po każdej zmianie wprowadzonej w aplikacji można pominąć wdrożenie bazy danych i oba etapy usunięcia komponentów, co oznacza pozostawienie jedynie etapów wdrożenia aplikacji i jej weryfikacji.

```
$ SKIP_teardown_db=true \  
  SKIP_teardown_app=true \  
  SKIP_deploy_db=true \  
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
```

(...)

```
PASS  
ok   terraform-up-and-running 13.824s
```

W ten sposób można skrócić z minut do zaledwie sekund czas oczekiwania na informacje o wyniku wprowadzonych zmian, co oznacza spore zwiększenie produktywności programisty.

Podczas wprowadzania zmian należy pamiętać o regularnym przekazywaniu (zatwierdzaniu) kodu do repozytorium.

```
$ git commit -m "Uaktualniony komunikat Witaj, świecie"
```

Przekazanie zmian do zatwierdzenia

Gdy kod działa zgodnie z oczekiwaniami, można utworzyć żądanie aktualizacji, aby wprowadzone przez Ciebie zmiany zostały przejrzane i zatwierdzone, podobnie jak ma to miejsce w przypadku kodu aplikacji. Twój zespół przejrzy wprowadzone zmiany, poszuka błędów, a także wymusi stosowanie się do *reguł tworzenia kodu źródłowego*. Gdy stworzysz kod w ramach zespołu, niezależnie od rodzaju tworzonego kodu powinny być zdefiniowane reguły jego tworzenia, które mają być przestrzegane przez wszystkich. Jedną z moich ulubionych definicji czystego kodu pochodzi z wywiadu, który przeprowadziłem z Nickiem Dellamaggiore na potrzeby jednej z moich wcześniejszych książek — *Hello, Startup* (<https://www.hello-startup.net/>):

Jeżeli spojrzeć na plik przygotowany przez 10 różnych inżynierów, praktycznie powinno być nie do odróżnienia, który fragment został utworzony przez daną osobę. W takim przypadku kod jest dla mnie czysty.

Aby osiągnąć taki stan, należy przeprowadzać sprawdzanie kodu, publikować reguły dotyczące stylu tworzenia kodu, stosowanych wzorców i idiomów języka. Gdy wszyscy członkowie zespołu poznają te reguły, każdy z nich stanie się bardziej produktywny, ponieważ będzie wiedział, jak tworzyć kod w dokładnie taki sam sposób. W tym momencie kwestią będzie to, co należy napisać, a nie jak to zrobić.

— Nick Dellamaggiore, szef infrastruktury w Coursera

Reguły tworzenia kodu źródłowego, które będą miały sens dla poszczególnych zespołów, będą odmienne. Dlatego też tutaj zdecydowałem się na przedstawienie kilku tych, które przez większość zespołów są uznawane za użyteczne:

- Dokumentacja.
- Testy zautomatyzowane.
- Układ plików.
- Przewodnik po stylu.

Dokumentacja

W pewnym sensie kod Terraform jest sam w sobie formą dokumentacji. Za pomocą prostego języka dokładnie opisuje wdrożoną infrastrukturę oraz sposób jej konfiguracji. Jednak nie istnieje coś takiego jak kod samodokumentujący się. Wprowadzie doskonale utworzony kod może wskazywać *sposób* działania, ale żaden znany mi język programowania (w tym Terraform) nie wyjaśni *powodów* danego działania.

Dlatego każde oprogramowanie, łącznie z IaC, poza samym kodem wymaga również dokumentacji. Istnieje kilka różnych typów dokumentacji, które należy wziąć pod uwagę, a następnie na etapie recenzji kodu sprawdzać, czy została utworzona.

Dokumentacja pisemna

Większość modułów Terraform powinna zawierać plik typu README wyjaśniający działanie modułu, powody jego utworzenia, sposób jego użycia i modyfikacji. Właściwie taki plik powinien zostać utworzony jako pierwszy, jeszcze przed przystąpieniem do pracy nad rzeczywistym kodem Terraform. To wymusi na programiście rozważenie tego, *co* jest tworzone i *dlaczego*, jeszcze przed zagłębieniem się w kod i zagubieniem w szczegółach związanych ze *sposobem* opracowania projektu⁴. Poświęcenie 20 minut na utworzenie pliku typu README może uchronić przed spędzaniem później wielu godzin na tworzeniu kodu rozwiązującego niewłaściwy problem. Poza prostym plikiem typu README można przygotować także samouczki, dokumentację API, strony Wiki oraz dokumenty projektowe, które głębiej przedstawiają sposób działania kodu i wyjaśniają powody jego utworzenia w dany sposób.

Dokumentacja kodu

W samym kodzie można stosować komentarze jako formę dokumentacji. Każdy tekst rozpoczynający się od znaku # jest w Terraform traktowany jako komentarz. Nie używaj komentarzy do wyjaśniania sposobu działania kodu — kod powinien mówić sam za siebie. Zawieraj w nich tylko te informacje, które nie mogą być wyrażone za pomocą kodu, np. sposób jego użycia lub powody wyboru określonego wzorca. Terraform pozwala także, aby zmienne danych wejściowych i wyjściowych deklarowały parametr `description`, który jest doskonałym miejscem na opisanie sposobu użycia danej zmiennej.

⁴ Rozpoczęcie pracy od utworzenia pliku typu README jest nazywane podejściem RDD (ang. *readme driven development*), o którym więcej informacji znajdziesz na stronie <https://tom.preston-werner.com/2010/08/23/readme-driven-development.html>.

Przykładowe fragmenty kodu

Jak już wspomniałem w rozdziale 6., każdy moduł Terraform powinien zawierać przykładowy kod przedstawiający sposób używania danego modułu. To jest doskonałe miejsce na przedstawienie zalecanego sposobu użycia, umożliwienie użytkownikom wypróbowania modułu bez konieczności utworzenia jakiegokolwiek kodu oraz podstawowy sposób na dodanie testów zautomatyzowanych dla modułu.

Testy zautomatyzowane

Cały rozdział 7. został poświęcony na omówienie tematu testowania kodu Terraform, więc nie będę tego w tym miejscu powtarzał. Ograniczę się jedynie do stwierdzenia, że pozbawiony testów kod infrastruktury jest nieprawidłowy. Dlatego też jeden z najważniejszych komentarzy, jakie możesz dodać podczas recenzji kodu, brzmi: „jak to przetestowałeś?”.

Układ plików

Zespół powinien definiować konwencje dotyczące miejsca przechowywania kodu Terraform i stosowanego układu plików. Skoro układ plików kodu Terraform określa również sposób przechowywania informacji o stanie, szczególną ostrożność należy zachować w kwestii tego, jak układ pliku wpływa na możliwość zapewnienia izolacji, np. jak zagwarantuje, że zmiany w środowisku roboczym przypadkowo nie spowodują problemów w środowisku produkcyjnym. Podczas sprawdzania kodu być może będziesz chciał wymusić stosowanie układu plików przedstawionego w rozdziale 3., zapewniającego izolację między poszczególnymi środowiskami (np. roboczym i produkcyjnym) oraz między różnymi komponentami (np. topologią sieci dla całego środowiska i pojedynczą aplikacją w tym środowisku).

Przewodnik po stylu

Każdy zespół powinien wymuszać stosowanie zbioru konwencji dotyczących stylu tworzenia kodu, czyli m.in. użycia białych znaków, znaków nowego wiersza, wcięć, nawiasów klamrowych, nazewnictwa zmiennych. Wprawdzie programiści uwielbiają debaty dotyczące stosowania spacji kontra tabulatorów lub na temat tego, w którym miejscu umieścić nawias klamrowy, ale rzeczywisty wybór nie ma tutaj znaczenia. Liczy się tylko spójne stosowanie konwencji w całej bazie kodu. Narzędzia formatowania są dostępne dla większości edytorów kodu i zintegrowanych środowisk programistycznych (ang. *integrated development environment*, IDE), a także jako zaczepy zatwierdzania w systemie kontroli wersji — to wszystko powinno pomóc w wymuszeniu spójnego stosowania konwencji tworzenia kodu.

Terraform ma wbudowane polecenie `fmt`, którego działanie polega na ponownym sformatowaniu kodu w taki sposób, aby automatycznie był stosowany spójny styl.

\$ terraform fmt

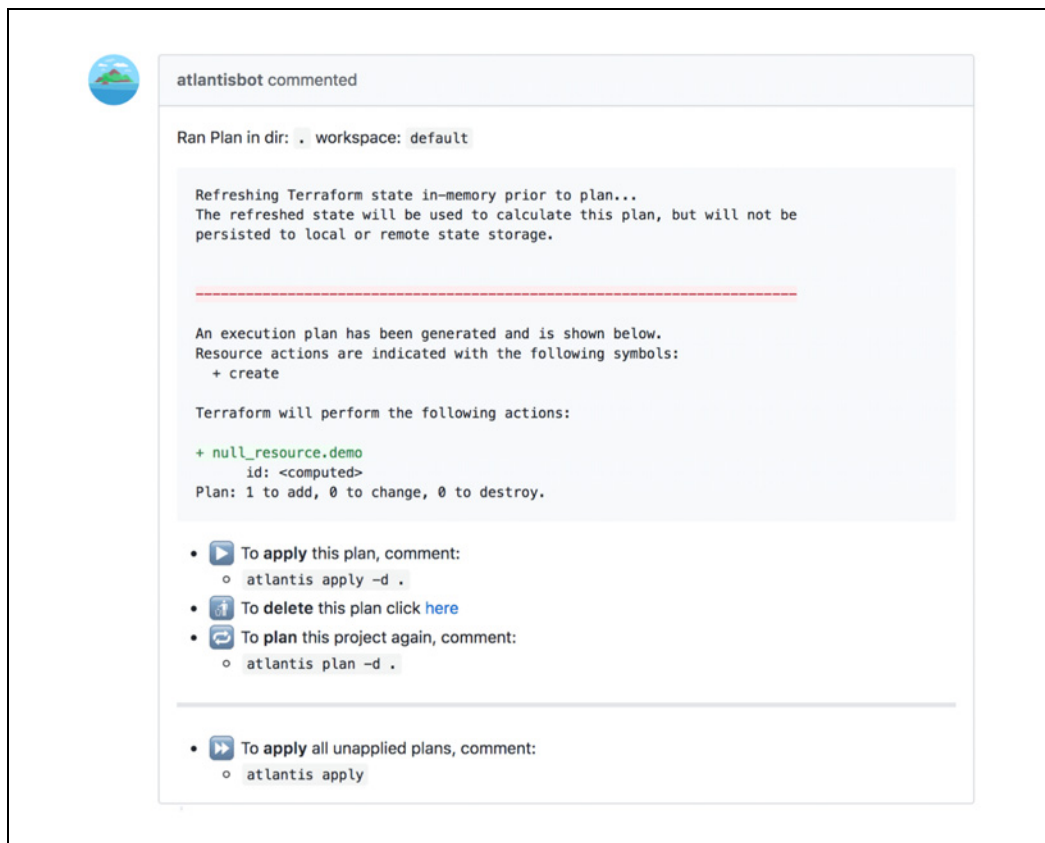
To polecenie można wydawać jako część zaczepu zatwierdzania i tym samym zapewnić, że cały kod umieszczany w systemie kontroli wersji będzie automatycznie stosował spójny styl.

Uruchomienie testów zautomatyzowanych

Podobnie jak w przypadku kodu aplikacji, także kod infrastruktury powinien zawierać zaczepy zatwierdzania uruchamiające testy zautomatyzowane w serwerze CI po każdej operacji zatwierdzenia i przedstawiające wynik tych testów w postaci żądania aktualizacji. Z rozdziału 7. dowiedziałeś się, jak tworzyć testy jednostkowe, testy integracji i testy typu E2E. Istnieje jeszcze jeden niezwykle ważny typ testu, który należy przeprowadzać: `terraform plan`. Reguła jest bardzo prosta:

Zawsze wykonuj polecenie `terraform plan` przed wydaniem polecenia `terraform apply`.

Terraform automatycznie wyświetli dane wyjściowe polecenia `terraform plan` po wydaniu polecenia `terraform apply`, więc powinieneś zatrzymać się na chwilę i zawsze dokładnie zapoznać się z danymi wyjściowymi polecenia `terraform plan`. Możesz być zaskoczony typem błędów, które można wychwycić dzięki poświęceniu 30 sekund na przejrzenie sekcji *diff* w danych wyjściowych. Doskonałym rozwiązaniem będzie zintegrowanie polecenia `terraform plan` z procedurą recenzji kodu. Przykładowo Atlantis (<https://www.runatlantis.io/>) to narzędzie typu open source, które automatycznie wydaje polecenie `terraform plan` podczas operacji zatwierdzenia oraz umieszcza dane wyjściowe tego polecenia jako komentarz w żądaniu aktualizacji (zobacz rysunek 8.3).



Rysunek 8.3. Atlantis może automatycznie dodać dane wyjściowe polecenia `terraform plan` jako komentarz w żądaniu aktualizacji

Polecenie `terraform plan` pozwala również na umieszczenie w pliku informacji o znalezionych różnicach:

```
$ terraform plan -out=example.plan
```

Następnie można wydać polecenie `terraform apply` wraz z zapisanym planem i dzięki temu mieć pewność, że zostaną zastosowane *dokładnie* te zmiany, które wcześniej zobaczyłeś.

```
$ terraform apply example.plan
```

Warto w tym miejscu wspomnieć, że podobnie jak w przypadku informacji o stanie Terraform, także pliki zapisanego planu mogą zawierać informacje poufne. Przykładowo, jeśli wdrażasz bazę danych za pomocą Terraform, plik planu może zawierać hasło do tej bazy. Ponieważ pliki planu nie są szyfrowane, to jeśli chcesz je przechowywać przez dłuższy okres czasu, musisz je w pewien sposób szyfrować.

Połączenie kodu istniejącego z nowym i wydanie produktu

Gdy członkowie zespołu mieli możliwość przejrzenia zmian w kodzie i danych wyjściowych polecenia `terraform plan`, a także po zaliczeniu wszystkich testów zmiany można wprowadzić do gałęzi `master` i wydać kod. Podobnie jak w przypadku kodu aplikacji, także tutaj można skorzystać z tagów Git do utworzenia wersjonowanego wydania produktu.

```
$ git tag -a "v0.0.6" -m "Uaktualniony tekst hello-world-example"
$ git push --follow-tags
```

O ile w przypadku kodu aplikacji bardzo często są tworzone oddzielne produkty do wdrożenia, np. obraz Dockera lub maszyny wirtualnej, o tyle Terraform natywnie obsługuje pobieranie kodu źródłowego z repozytorium Git, a kod o określonym tagu *pozostaje* niemodyfikowalny, więc wdrożona będzie wersjonowana wersja produktu.

Wdrożenie

Skoro masz niemodyfikowalny i wersjonowany produkt, nadeszła pora na jego wdrożenie. Oto kilka kluczowych kwestii do rozważenia podczas wdrażania kodu Terraform:

- narzędzia wdrażania,
- strategię wdrażania,
- serwer wdrożenia,
- stosowanie produktu w różnych środowiskach.

Narzędzia wdrażania

Podczas wdrażania kodu Terraform samo narzędzie Terraform jest podstawowym wykorzystywanym narzędziem. Jednak do dyspozycji masz także kilka innych, które mogą okazać się użyteczne:

Atlantis

Wspomniane już wcześniej narzędzie typu open source, które może nie tylko umieścić dane wyjściowe polecenia `terraform plan` w żądaniu aktualizacji, ale również wykonać polecenie

terraform apply po dodaniu specjalnego komentarza do żądania aktualizacji. Wprawdzie to narzędzie oferuje wygodny interfejs działający w przeglądarce WWW dla wdrożeń Terraform, ale trzeba mieć świadomość, że nie obsługuje wersjonowania. To znacznie utrudnia obsługę i debugowanie większych projektów.

Terraform Enterprise

Produkty korporacyjne HashiCorp zapewniają działający w przeglądarce WWW interfejs przeznaczony do wykonywania poleceń terraform plan i terraform apply, jak również do zarządzania zmiennymi, danymi wrażliwymi i uprawnieniami dostępu.

Terragrunt

To jest opakowanie typu open source dla Terraform wypełniające pewne luki w Terraform. Sposób użycia tego narzędzia przedstawię w dalszej części rozdziału na przykładzie wdrożenia wersjonowanego kodu Terraform w różnych środowiskach przy użyciu minimalnej liczby operacji kopiowania i wklejania.

Skrypty

Jak zawsze można tworzyć skrypty w języku programowania ogólnego przeznaczenia, takim jak Python, Ruby lub Bash, i tym samym dostosować do własnych potrzeb sposób działania Terraform.

Strategie wdrażania

Terraform sam w sobie nie oferuje żadnych strategii wdrażania. Nie istnieje wbudowana obsługa dla wdrożeń ciągłych lub typu niebieski-zielony, a także nie ma możliwości przełączania większości zmian Terraform (tzn. nie można włączyć lub wyłączyć zmiany bazy danych, zmianę wprowadzasz lub nie). W zasadzie jesteś ograniczony do terraform apply, co oznacza zastosowanie konfiguracji zdefiniowanej w kodzie. Oczywiście w kodzie można czasem implementować własne strategie wdrażania, takie jak ciągle wdrażanie bez przestoju w module asg-rolling-deploy opracowanym w poprzednich rozdziałach. Skoro Terraform jest językiem deklaratywnym, to kontrola nad wdrożeniami jest dość ograniczona.

Ze względu na te ograniczenia pod uwagę trzeba koniecznie wziąć to, co się stanie w przypadku niepowodzenia operacji wdrożenia. W przypadku wdrażania aplikacji wiele typów błędów można wychwycić przez strategię wdrożenia. Przykładowo, jeśli aplikacja nie zaliczy operacji sprawdzenia jej stanu, mechanizm równoważenia obciążenia nigdy nie przekaże do niej ruchu sieciowego, więc błąd aplikacji nie będzie miał wpływu na użytkowników. Co więcej, wdrażanie ciągle lub typu niebieski-zielony może automatycznie przywrócić poprzednią wersję aplikacji, jeśli okaże się, że nowa zawiera błędy.

Z kolei Terraform *nie oferuje możliwości automatycznego przywracania poprzedniej wersji, gdy nowa zawiera błędy*. To po części wynika z dowolności kodu infrastruktury — często nie można bezpiecznie lub w ogóle nie można tego zrobić. Przykładowo, jeśli wdrożenie aplikacji zakończyło się niepowodzeniem, prawie zawsze można bezpiecznie przywrócić jej starszą wersję. Jeśli natomiast wprowadzenie zmiany Terraform zakończyło się niepowodzeniem, a ta zmiana spowodowała usunięcie bazy danych lub serwera, nie można łatwo powrócić do poprzedniego stanu (sprzed wdrożenia).

Co więcej, jak się dowiedziałeś z rozdziału 5., błędy dość często istnieją w samym narzędziu Terraform. Dlatego też przyjęta strategia wdrożenia powinna zakładać, że błędy są czymś (względnie) normalnym, i zapewnić możliwość radzenia sobie z nimi.

Ponowne próby

Pewne typy błędów są tymczasowe i znikają po ponownym wydaniu polecenia `terraform apply`. Wykorzystane narzędzia wdrożenia Terraform powinny wykrywać te znane błędy i automatycznie ponawiać próbę po krótkiej przerwie. Terraform ma wbudowaną funkcjonalność automatycznego ponawiania próby w przypadku wystąpienia znanego błędu.

Błędy informacji o stanie Terraform

Sporadycznie Terraform nie zapisze informacji o stanie po wykonaniu polecenia `terraform apply`. Przykładowo, jeśli w trakcie wykonywania polecenia `terraform apply` nastąpi zerwanie połączenia z internetem, nie tylko to polecenie zakończy się niepowodzeniem, ale Terraform nie będzie mieć możliwości zapisania w zdalnym backendzie (np. Amazon S3) uaktualnionego pliku informacji o stanie. W takich sytuacjach Terraform zapisze informacje o stanie w pliku na dysku o nazwie `errored.tfstate`. Upewnij się, że serwer CI nie usuwa tych plików (np. w trakcie operacji porządkujących przestrzeń roboczą po zakończeniu kompilacji). Jeżeli po nieudanym wdrożeniu będziesz mieć dostęp do tego pliku, po przywróceniu połączenia z internetem możesz przekazać ten plik do zdalnego backendu (np. S3) za pomocą polecenia `terraform state push`, aby informacje o stanie nie zostały utracone.

```
$ terraform state push errored.tfstate
```

Błędy związane ze zwalnianiem blokad

Sporadycznie Terraform nie zwolni nałożonej blokady. Przykładowo, jeżeli serwer CI ulegnie awarii w trakcie wykonywania polecenia `terraform apply`, stan pozostanie trwale zablokowany. Każdy, kto spróbuje wykonać polecenie `terraform apply` dla tego samego modułu, otrzyma komunikat błędu informujący o zablokowaniu stanu i zawierający identyfikator blokady. Jeżeli masz całkowitą pewność, że to jest przypadkowo niezwolniona blokada, możesz wymusić jej zwolnienie za pomocą polecenia `terraform force-unlock`, przekazując identyfikator blokady odczytany ze wspomnianego komunikatu błędu.

```
$ terraform force-unlock <LOCK_ID>
```

Serwer wdrożenia

Podobnie jak w przypadku kodu aplikacji, wszystkie zmiany kodu infrastruktury powinny być wprowadzane z poziomu serwera CI, a nie z komputera programisty. Polecenia `terraform xxx` można wydawać z poziomu Jenkins, CircleCI, Terraform Enterprise, Atlantis oraz każdego innej, sensownie zabezpieczonej platformy automatyzacji. W ten sposób zyskujesz takie same korzyści jak w przypadku kodu aplikacji: wymuszenie pełnej automatyzacji procesu wdrożenia, gwarancję przeprowadzania wdrożenia z poziomu spójnego środowiska, a także lepszą kontrolę nad tym, kto ma uprawnienia do środowiska produkcyjnego.

Mając to wszystko na uwadze, uprawnienia do wdrożenia kodu infrastruktury są nieco trudniejsze do zdefiniowania niż dla kodu aplikacji. W przypadku kodu aplikacji zwykle nadaje się serwerowi

CI minimalny zestaw uprawnień pozwalający na wdrażanie aplikacji. Przykładowo w celu wdrożenia grupy ASG serwer CI zwykle wymaga jedynie kilku konkretnych uprawnień `ec2` i `autoscaling`. Jednak do wdrożenia dowolnych zmian w kodzie infrastruktury (np. kod Terraform może próbować wdrożyć bazę danych, VPC lub zupełnie nowe konto AWS) serwer CI będzie wymagał różnych uprawnień — tzn. uprawnień administratora.

Skoro serwery CI zostały zaprojektowane do wykonywania dowolnego kodu, często są trudne do zabezpieczenia (np. spróbuj dołączyć do listy dyskusyjnej dotyczącej zapewnienia bezpieczeństwa serwera Jenkins, a przekonasz się, jak często pojawiają się informacje o znalezionych lukach w zabezpieczeniach), więc nadanie serwerowi CI trwałych uprawnień administratora może być ryzykowne. Istnieje kilka działań, które można podjąć, aby pomóc w minimalizacji tego ryzyka:

- Nie udostępniaj serwera CI w publicznym internecie. Serwer CI powinien działać w prywatnej podsiaci, bez żadnego publicznego adresu IP, aby pozostał dostępny jedynie poprzez połączenie VPN. Takie rozwiązanie zdecydowanie jest znacznie bezpieczniejsze, choć jednocześnie wiąże się z pewnym kosztem: zaczepty sieciowe z systemów zewnętrznych nie będą działały — przykładowo GitHub nie będzie miał możliwości automatycznego uruchomienia kompilacji w serwerze CI. Zamiast tego należy skonfigurować serwer CI w taki sposób, aby uaktualnienia pobierał z systemu kontroli wersji.
- Zabezpieczenie serwera CI. Udostępnij serwer CI tylko przez HTTPS, wymagaj uwierzytelniania wszystkich użytkowników i stosu praktyki zabezpieczania serwerów (np. zabezpieczenia zapory sieciowej, instalacji fail2ban, włączenia audytu rejestrowania danych itd.).
- Rozważ rezygnację z nadawania serwerowi CI trwałych uprawnień administratora. Zamiast tego nadaj uprawnienia pozwalające na automatyczne wdrażanie pewnych typów zmian kodu infrastruktury, natomiast wszystkie bardziej wrażliwe zmiany (np. związane z dodawaniem lub usuwaniem użytkowników oraz z dostępem do danych poufnych) powinny wymagać od administratora dostarczenia serwerowi CI tymczasowych uprawnień administracyjnych (np. tylko na godzinę).

Stosowanie produktu w różnych środowiskach

Podobnie jak w przypadku produktu aplikacji, między środowiskami należy promować niemodyfikowalny, wersjonowany produkt. Przykładowo promuj wersję 0.0.6 produktu ze środowiska programistycznego do roboczego i później do produkcyjnego⁵. Reguła jest tutaj bardzo prosta:

Przed wprowadzeniem zmian w środowisku produkcyjnym zawsze testuj je w środowisku przedprodukcyjnym.

Skoro w Terraform wszystko i tak jest zautomatyzowane, nie będzie Cię kosztować zbyt wiele wypróbowanie zmian w środowisku roboczym przed ich wprowadzeniem w środowisku produkcyjnym, za to takie podejście pomoże w wychwyceniu ogromnej liczby błędów. Testowanie w środowisku przed-

⁵ Podziękowania za określenie kolejności promowania kodu Terraform w środowiskach należą się Kiefowi Morrisowi — zapoznaj się z artykułem *Using Pipelines to Manage Environments with Infrastructure as Code* opublikowanym na stronie <https://medium.com/@kief/https-medium-com-kief-using-pipelines-to-manage-environments-with-infrastructure-as-code-b37285a1cbf5#.59368x6da>.

produkcyjnym jest szczególnie ważne, ponieważ, jak wspomniałem we wcześniejszej części rozdziału, Terraform nie przeprowadza automatycznego wycofania zmian w przypadku błędów. Jeżeli wydasz polecenie `terraform apply` i coś pójdzie źle, musisz to naprawić ręcznie. Łatwiejsze i mniej stresujące będzie przechwytywanie błędów w środowisku przedprodukcyjnym niż w produkcyjnym.

Proces promowania kodu Terraform między środowiskami jest podobny do procesu promowania produktu aplikacji z wyjątkiem kroku dodatkowego w postaci wykonania polecenia `terraform plan` i przejrzania jego danych wyjściowych. Ten krok jest zwykle niepotrzebny podczas wdrażania aplikacji, ponieważ większość tych wdrożeń jest podobna i charakteryzuje się względnie niewielkim ryzykiem. Jednak każde wdrożenie infrastruktury może być zupełnie inne od pozostałych, a pomyłki (np. usunięcie bazy danych) będą bardzo kosztowne, więc możliwość spojrzenia na dane wyjściowe polecenia `terraform plan` i ich przeanalizowanie nie będzie czasem straconym.

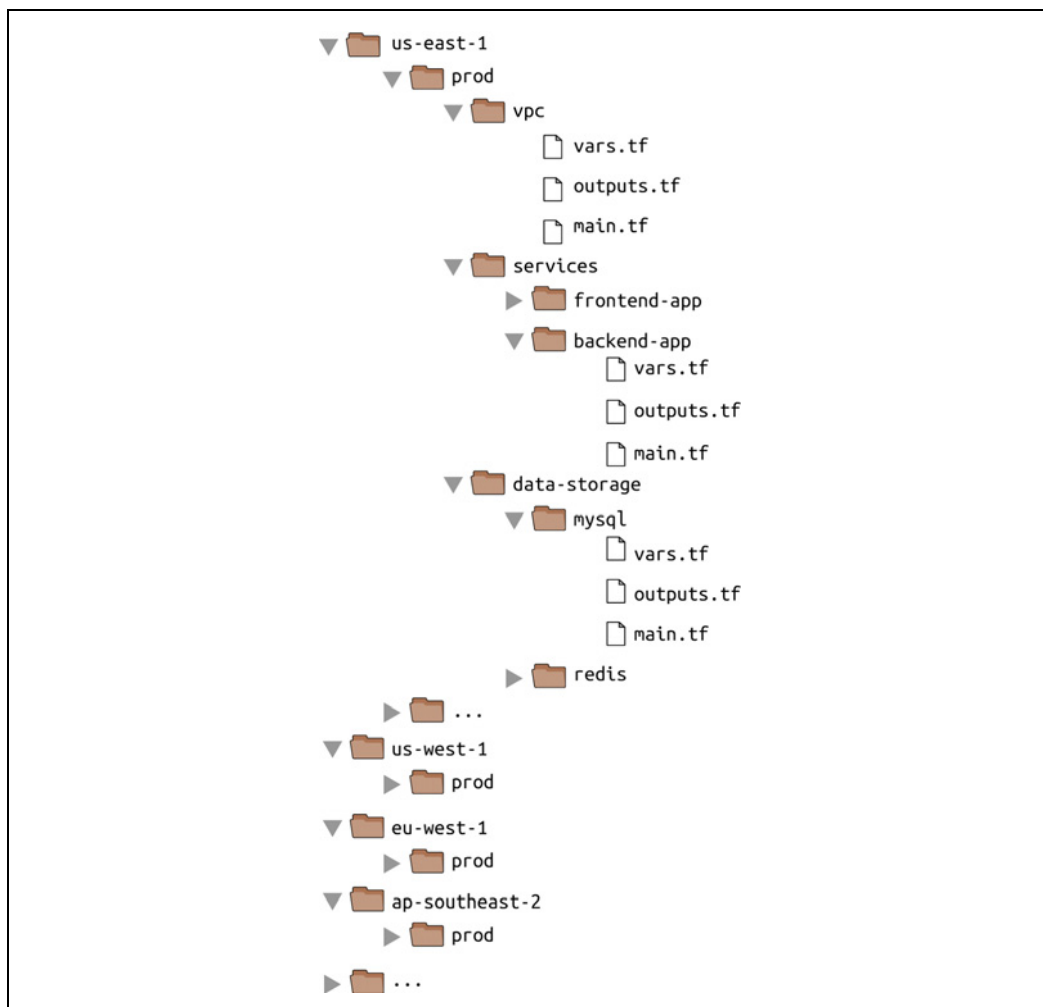
Spójrz na przykład procesu promocji modułu Terraform w wersji 0.0.6 między środowiskami programistycznym, roboczym i produkcyjnym:

1. Uaktualnienie modułu do wersji 0.0.6 aplikacji w środowisku programistycznym i wydanie polecenia `terraform plan`.
2. Poproszenie kogoś o sprawdzenie i zaaprobowanie planu, np. przez wysłanie zautomatyzowanej wiadomości za pomocą serwisu Slack.
3. Jeżeli plan zostanie zaaprobowany, wdrożenie wersji 0.0.6 w środowisku programistycznym przez wydanie polecenia `terraform apply`.
4. Uruchomienie w środowisku programistycznym testów ręcznych i zautomatyzowanych.
5. Jeżeli wersja 0.0.6 działa świetnie w środowisku programistycznym, kroki od 1. do 4. należy powtórzyć podczas wdrażania wersji 0.0.6 aplikacji w środowisku roboczym.
6. Jeżeli wersja 0.0.6 działa świetnie w środowisku roboczym, kroki od 1. do 4. należy powtórzyć podczas promowania wersji 0.0.6 do środowiska produkcyjnego.

Ważna kwestia do rozważenia wiąże się z obsługą powtarzającego się kodu między środowiskami w repozytorium *live*. Dla przykładu przeanalizuj repozytorium *live* pokazane na rysunku 8.4.

Repozytorium *live* ma ogromną liczbę regionów, w każdym z nich znajduje się duża liczba modułów, z których większość jest kopiowana i wklejana. Oczywiście poszczególne moduły mają plik *main.tf* odwołujący się do modułu w repozytorium *modules*, więc liczba operacji kopiowania i wklejania nie jest aż tak ogromna, jak można by sądzić. Jeżeli chcesz jedynie utworzyć egzemplarz pojedynczego modułu, wciąż masz ogromną ilość kodu koniecznego do powielenia między wszystkimi środowiskami:

- blok konfiguracji `provider`,
- blok konfiguracji `backend`,
- przygotowanie wszystkich zmiennych danych wejściowych dla modułu,
- wyświetlenie wartości wszystkich zmiennych danych wyjściowych modułu.

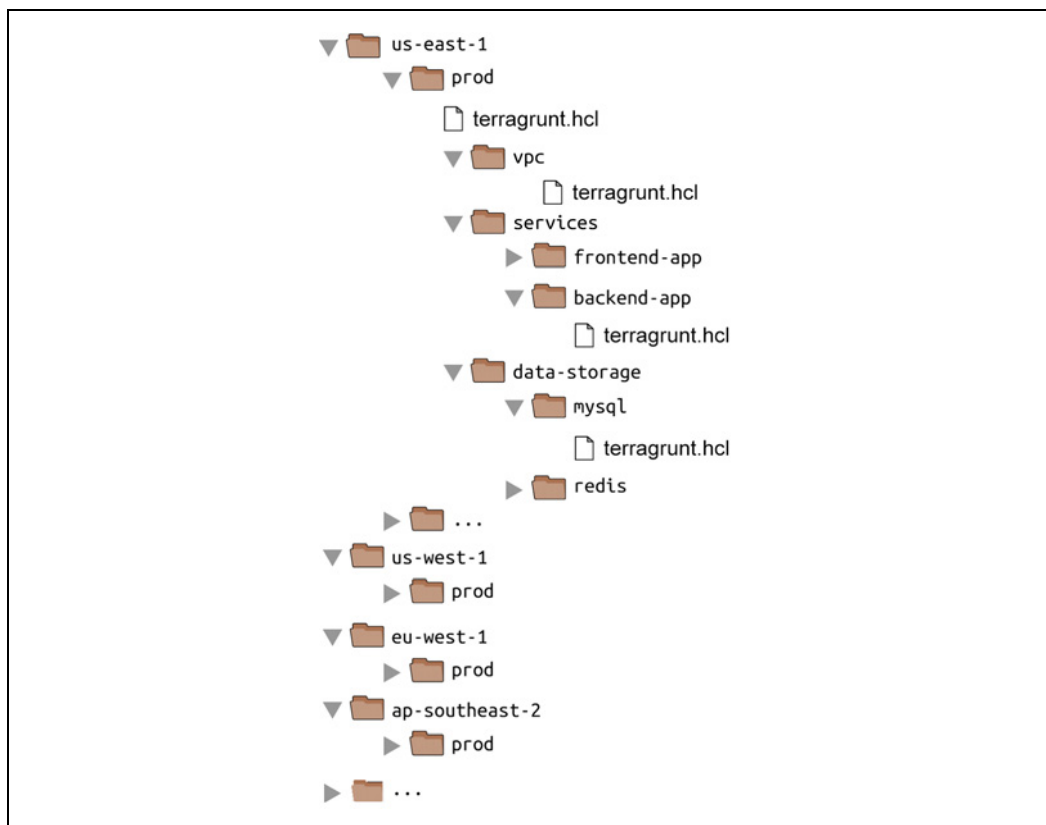


Rysunek 8.4. Układ plików z ogromną liczbą skopiowanych i wklejonych środowisk oraz modułów w poszczególnych środowiskach

To może oznaczać dodanie dziesiątek lub nawet setek wierszy praktycznie identycznego kodu w każdym module, skopiowanego i wklejonego do każdego środowiska. Aby w tym kodzie zastosować regułę DRY i ułatwić promowanie kodu Terraform między środowiskami, można wykorzystać narzędzie typu open source — Terragrunt (<https://github.com/gruntwork-io/terragrunt>), o którym wspomniałem już wcześniej w książce. Terragrunt to lekkie opakowanie dla Terraform, po którego zainstalowaniu (zapoznaj się z dokumentem README, <https://github.com/gruntwork-io/terragrunt> ↪[#install-terragrunt](#)) będziesz mógł wydawać wszystkie standardowe polecenia terraform, przy czym jako pliku binarnego użyjesz *terragrunt*:

```
$ terragrunt plan
$ terragrunt apply
$ terragrunt output
```


Terragrunt wykona polecenie Terraform, ale opierając się na konfiguracji zdefiniowanej w pliku *terragrunt.hcl*, można ustalić pewne zachowanie dodatkowe. Podstawowa idea polega na tym, aby cały kod Terraform był dokładnie raz zdefiniowany w repozytorium *modules*, a w repozytorium *live* znalazły się pliki *terragrunt.hcl* zapewniające możliwość stosowania reguły DRY podczas konfigurowania i wdrażania poszczególnych modułów w różnych środowiskach. Skutkiem będzie repozytorium *live* z mniejszą liczbą plików i wierszy kodu, jak pokazałem na rysunku 8.5.



Rysunek 8.5. Układ plików w przypadku użycia Terragrunt

Pracę rozpocznij od dodania bloku konfiguracji provider do plików *modules/data-stores/mysql/main.tf* i *modules/services/hello-world-app/main.tf*.

```

provider "aws" {
  region = "us-east-2"

  # Zezwolenie na użycie dowolnej wersji 2.x dostawcy AWS.
  version = "~> 2.0"
}

```

Następnym krokiem jest dodanie bloku konfiguracji backend do plików *modules/data-stores/mysql/main.tf* i *modules/services/hello-world-app/main.tf*, ale pozostawienie pustego bloku config (wkrótce dowiesz się, jak ten blok zostanie wypełniony za pomocą Terragrunt).

```

terraform {
  # Wymagana jest dowolna wersja 0.12.x Terraform.
  required_version = ">= 0.12, < 0.13"

  # Konfiguracja częściowa. Pozostała część zostanie wypełniona przez Terragrunt.
  backend "s3" {}
}

```

Zmiany należy zatwierdzić i wydać nową wersję repozytorium *modules*.

```

$ git add modules/data-stores/mysql/main.tf
$ git add modules/services/hello-world-app/main.tf
$ git commit -m "Uaktualnienie modułów mysql i hello-world-app pod kątem Terragrunt"
$ git tag -a "v0.0.7" -m "Uaktualnienie komunikatu Witaj, świecie"
$ git push --follow-tags

```

Teraz przejdź do repozytorium *live* i usuń wszystkie pliki *.tf*. Cały skopiowany i wklejony kod Terraform zastąpisz jednym plikiem *terragrunt.hcl* dla każdego modułu. Dla przykładu spójrz na zawartość wymienionego pliku znajdującego się pod adresem *live/stage/data-stores/mysql/terragrunt.hcl*:

```

terraform {
  source = "github.com/<OWNER>/modules//data-stores/mysql?ref=v0.0.7"
}

inputs = {
  db_name      = "example_stage"
  db_username  = "admin"
  # Zdefiniowanie hasła za pomocą zmiennej środowiskowej TF_VAR_db_password.
}

```

Jak możesz zobaczyć, w pliku *terragrunt.hcl* jest stosowana dokładnie ta sama składnia HCL (ang. *hashicorp configuration language*), jak w samym kodzie Terraform. Po wydaniu polecenia *terragrunt apply* i znalezieniu parametru *source* w pliku *terragrunt.hcl* Terraform przeprowadzi następującą procedurę:

1. Pobranie do katalogu tymczasowego kodu spod adresu URL wymienionego w parametrze *source*. Obsługiwana jest dokładnie ta sama składnia adresu URL, jak w przypadku parametru *source* modułów Terraform. Dlatego też można korzystać ze ścieżek dostępu do plików lokalnych, adresów URL Git, wersjonowanych adresów URL Git (za pomocą parametru *ref*, jak pokazałem w przykładzie) itd.
2. Wydanie w katalogu tymczasowym polecenia *terraform apply* i przekazanie zmiennych danych wejściowych, które zostały wymienione w bloku *inputs = { ... }*.

Zaletą takiego podejścia jest to, że kod w repozytorium *live* został zredukowany do zaledwie jednego pliku *terragrunt.hcl* na moduł i zawiera tylko wskaźnik prowadzący do używanego modułu (w określonej wersji) plus zmienne danych wejściowych do zdefiniowania w tym konkretnym środowisku. Dzięki temu w maksymalnie możliwy sposób wykorzystujesz regułę DRY.

Terragrunt pomaga również w stosowaniu reguły DRY względem konfiguracji backend. Zamiast definiować *bucket*, *key*, *dynamodb_table* itd. w każdym module, te wartości można zdefiniować w jednym pliku *terragrunt.hcl* dla każdego środowiska. Dla przykładu utwórz plik *live/stage/terragrunt.hcl* o przedstawionej tutaj zawartości:

```
remote_state {
  backend = "s3"
  config = {
    bucket      = "<NAZWA_KUBEŁKA>"
    key         = "${path_relative_to_include()}/terraform.tfstate"
    region      = "us-east-2"
    encrypt     = true
    dynamodb_table = "<NAZWA_TABELI>"
  }
}
```

W bloku `remote_state` należy podać konfigurację backend w dokładnie taki sam sposób jak zwykle, z wyjątkiem wartości `key` używającej funkcji wbudowanej Terragrunt o nazwie `path_relative_to_include()`. Wartością zwrótną tej funkcji jest względna ścieżka dostępu między głównym plikiem *terragrunt.hcl* i wszelkimi modułami potomnymi, które go zawierają. Przykładowo w celu dołączenia tego pliku głównego w *live/stage/data-stores/mysql/terragrunt.hcl* należy dodać blok `include`.

```
terraform {
  source = "github.com/<WŁAŚCICIEL>/modules//data-stores/mysql?ref=v0.0.7"
}

include {
  path = find_in_parent_folders()
}

inputs = {
  db_name      = "example_stage"
  db_username  = "admin"

  # Zdefiniowanie hasła za pomocą zmiennej środowiskowej TF_VAR_db_password.
}
```

Blok `include` odszukuje główny plik *terragrunt.hcl*, wykorzystując do tego funkcję wbudowaną Terragrunt o nazwie `find_in_parent_folders()`. Wszystkie ustawienia, w tym `remote_state`, są automatycznie dziedziczone po pliku nadrzędnym. W efekcie moduł `mysql` będzie stosował te same ustawienia backend, jakie zostały zdefiniowane w pliku głównym, a wartość `key` będzie automatycznie określona jako *data-stores/mysql/terraform.tfstate*. To oznacza, że informacje o stanie Terraform będą przechowywane w strukturze plików takiej samej jak w repozytorium *live*, co ułatwia ustalenie, który moduł wygenerował dany plik informacji o stanie.

W celu wdrożenia modułu należy wydać polecenie `terragrunt apply`.

```
$ terragrunt apply
```

```
[terragrunt] Reading Terragrunt config file at terragrunt.hcl
```

```
[terragrunt] Downloading Terraform configurations from
github.com/<OWNER>/modules//data-stores/mysql?ref=v0.0.7
```

```
[terragrunt] Running command: terraform init -backend-config=(...)
```

```
(...)
```

```
[terragrunt] Running command: terraform apply
```

(...)

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

W dzienniku zdarzeń danych wyjściowych Terragrunt powinieneś zobaczyć plik *terragrunt.hcl*, pobranie podanego modułu, wydanie polecenia `terraform init` w celu skonfigurowania backendu (nawet automatyczne utworzenie kubelka S3 i tabeli DynamoDB, o ile nie istniały), a następnie wydanie polecenia `terraform apply` odpowiedzialnego za wdrożenie wszystkiego.

Teraz można przystąpić do wdrożenia modułu `hello-world-app` w środowisku roboczym przez dodanie pliku *live/stage/services/hello-world-app/terragrunt.hcl* i wydanie polecenia `terragrunt apply`.

```
terraform {
  source = "github.com/<WŁAŚCICIEL>/modules//services/hello-world-app?ref=v0.0.7"
}

include {
  path = find_in_parent_folders()
}

inputs = {
  environment = "stage"

  min_size = 2
  max_size = 2

  enable_autoscaling = false

  db_remote_state_bucket = "<NAZWA_KUBEŁKA>"
  db_remote_state_key    = "<WARTOŚĆ_KLUCZA>"
}
```

Ten moduł używa również bloku `include` do pobrania ustawień z głównego pliku *terragrunt.hcl*, więc będzie dziedziczył te same ustawienia bloku `backend` z wyjątkiem parametru `key`, który automatycznie otrzyma zgodnie z oczekiwaniami wartość *services/hello-world-app/terraform.tfstate*. Gdy wszystko działa prawidłowo w środowisku roboczym, analogiczny plik *terragrunt.hcl* należy utworzyć w *live/prod* i do środowiska produkcyjnego promować dokładnie ten sam produkt w wersji 0.0.7 przez wydanie polecenia `terragrunt apply` w każdym module.

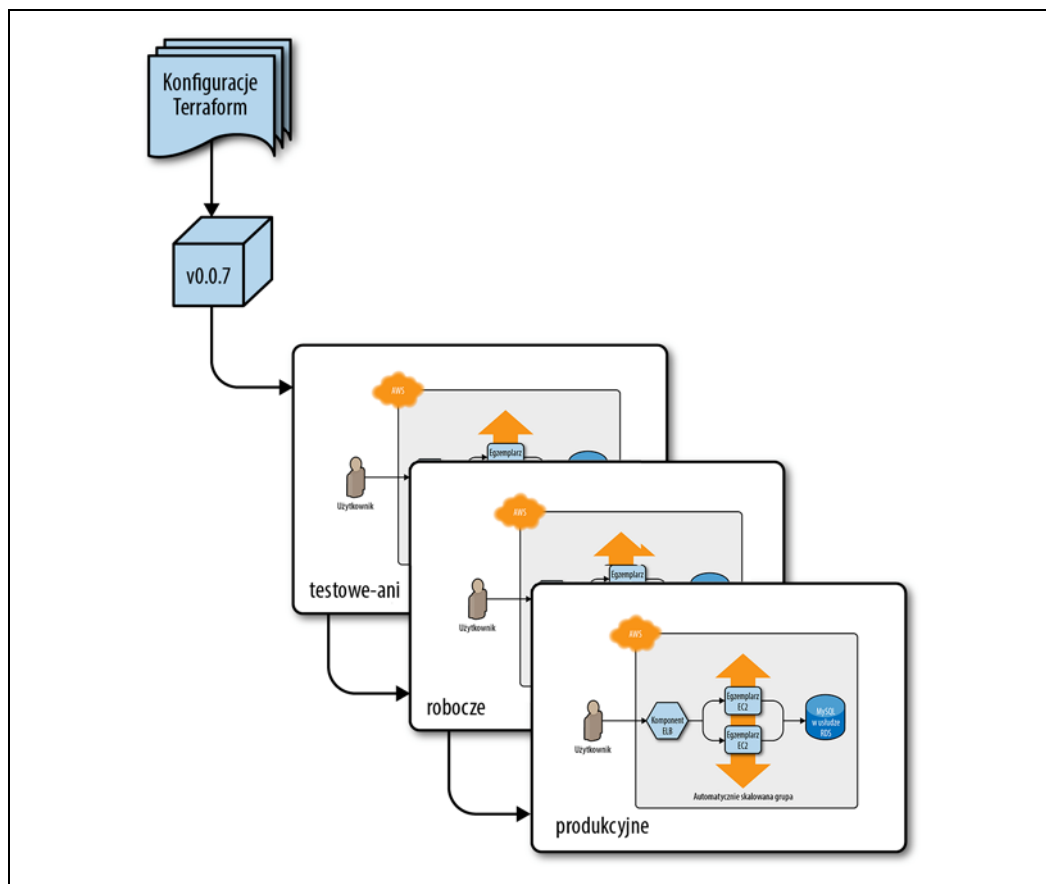
Zebranie wszystkiego w całość

Dowiedziałeś się, jak kod aplikacji i infrastruktury przekazać ze środowiska programistycznego do produkcyjnego. W tabeli 8.1 zestawilem obok siebie kroki przeprowadzane w obu tych procesach.

Jeżeli zastosujesz ten proces, będziesz w stanie uruchamiać kod aplikacji i infrastruktury w środowisku programistycznym, przetestować go, przeanalizować, zmienić na wersjonowany i niemodyfikowalny produkt, który następnie będzie przekazywany między środowiskami, jak pokazałem na rysunku 8.6.

Tabela 8.1. Sposoby pracy z kodem aplikacji i infrastruktury

	Kod aplikacji	Kod infrastruktury
Użyj systemu kontroli wersji	polecenie <code>git clone</code> tylko jedno repozytorium dla aplikacji użycie gałęzi	polecenie <code>git clone</code> repozytoria <i>live</i> i <i>modules</i> gałęzie nie są używane
Lokalne uruchomienie kodu	uruchomienie kodu w komputerze lokalnym polecenie <code>ruby web-server.rb</code> polecenie <code>ruby web-server-test.rb</code>	uruchomienie kodu w odizolowanym środowisku polecenie <code>terraform apply</code> polecenie <code>go test</code>
Wprowadzanie zmian w kodzie	zmiana kodu polecenie <code>ruby web-server.rb</code> polecenie <code>ruby web-server-test.rb</code>	zmiana kodu polecenie <code>terraform apply</code> polecenie <code>go test</code>
Przekazanie zmian do zatwierdzenia	wysłanie żądania aktualizacji wymuszenie zastosowania określonych reguł tworzenia kodu	użycie etapów wykonywania testów wysłanie żądania aktualizacji wymuszenie zastosowania określonych reguł tworzenia kodu
Uruchomienie testów zautomatyzowanych	uruchomienie testów w serwerze CI testy jednostkowe testy integracji testy typu E2E analiza statyczna	uruchomienie testów w serwerze CI testy jednostkowe testy integracji testy typu E2E analiza statyczna polecenie <code>terraform plan</code>
Połączenie kodu istniejącego z nowym i wydanie produktu	polecenie <code>git tag</code> utworzenie wersjonowanego, niemodyfikowalnego produktu	polecenie <code>git tag</code> użycie repozytorium wraz z tagiem jako wersjonowanego, niemodyfikowalnego produktu
Wdrożenie	wdrożenie za pomocą Terraform, narzędzi instrumentacji (np. Kubernetes, Mesos), skryptów wiele strategii wdrożenia: ciągle, typu niebieski-zielony, kanarkowe uruchomienie wdrożenia z poziomu serwera CI nadanie serwerowi CI ograniczonych uprawnień promocja niemodyfikowalnego, wersjonowanego produktu między poszczególnymi środowiskami	wdrożenie za pomocą Terraform, Atlantis, Terraform Enterprise, Terragrunt, skryptów ograniczona liczba strategii wdrożenia: upewnij się co do obsługi błędów za pomocą np. wielu prób i plik <i>errored.tfstate</i> uruchomienie wdrożenia z poziomu serwera CI nadanie serwerowi CI uprawnień administratora promocja niemodyfikowalnego, wersjonowanego produktu między poszczególnymi środowiskami



Rysunek 8.6. Promowanie między środowiskami niemodyfikowalnego, wersjonowanego produktu utworzonego na podstawie kodu Terraform

Podsumowanie

Skoro dotarłeś do tego miejsca w książce, wiesz już prawie wszystko to, co powinieneś, aby używać Terraform w rzeczywistych projektach. Dowiedziałeś się, jak tworzyć kod Terraform, jak zarządzać informacjami o stanie Terraform, jak tworzyć moduły Terraform wielokrotnego użycia, jak definiować pętle, jak definiować konstrukcje warunkowe, jak przeprowadzać wdrożenia, jak tworzyć kod Terraform o jakości produkcyjnej, jak przetestować kod Terraform, a także jak używać Terraform podczas pracy w zespole. Zapoznałeś się z przykładami wdrożenia i zarządzania serwerami, klastrami serwerów, mechanizmami równoważenia obciążenia, bazami danych, zaplanowanymi akcjami, alarmami CloudWatch, użytkownikami IAM, modułami wielokrotnego użycia, wdrożeniami bez przestoju, testami zautomatyzowanymi itd. Uf! Nie zapomnij o wydaniu polecenia `terraform destroy` w każdym module po zakończeniu z nim pracy.

Prawdziwie potężne możliwości Terraform, i ogólnie podejście IaC, wiążą się z zarządzaniem operacjami i aplikacją za pomocą tych samych reguł tworzenia kodu, jak w przypadku stosowanych podczas tworzenia samej aplikacji. To pozwala na pełne wykorzystanie możliwości inżynierii oprogramowania podczas przygotowywania infrastruktury, dołączania modułów, analizowania kodu, stosowania systemu kontroli wersji oraz przeprowadzania testów zautomatyzowanych.

Jeżeli prawidłowo używasz Terraform, Twój zespół będzie w stanie szybciej przeprowadzać wdrożenia i wprowadzać zmiany. Same wdrożenia powinny stać się (mam nadzieję) rutynowe i nudne — w świecie operacji nuda jest bardzo pożądana. Jeżeli naprawdę dobrze wykonasz swoją pracę, to zamiast poświęcać cały czas na ręczne zarządzanie infrastrukturą, Twój zespół będzie miał go znacznie więcej na zajęcie się usprawnieniem tej infrastruktury i jeszcze szybszy rozwój.

To już koniec książki, choć jednocześnie dopiero początek Twojej przygody z Terraform. Aby dowiedzieć się więcej na temat narzędzia Terraform, podejścia IaC i praktyk DevOps, zajrzyj do dodatku A, w którym zamieściłem listę zasobów wartych przejrzania. Jeżeli masz jakiegokolwiek uwagi lub pytania, napisz do mnie na adres jim@ybrikman.com. Dziękuję, że przeczytałeś tę książkę!

Polecane zasoby

W dodatku wymieniałem wybrane z najlepszych zasobów w postaci książek, blogów, newsletterów i prelekcji dotyczących DevOps i podejścia infrastruktury jako kodu.

Książki

- Kief Morris, *Infrastructure as Code: Managing Servers in the Cloud* (O'Reilly)
- Betsy Beyer, Chris Jones, Jennifer Petoff, Niall Richard Murph, *Site Reliability Engineering. Jak Google zarządza systemami produkcyjnymi* (Helion)
- Gene Kim, Patrick Debois, John Willis, Jez Humble, John Allspaw, *DevOps. Światowej klasy zwinność, niezawodność i bezpieczeństwo w Twojej organizacji* (Helion)
- Martin Kleppmann, *Przetwarzanie danych w dużej skali. Niezawodność, skalowalność i łatwość konserwacji systemów* (Helion)
- Jez Humble, David Farley, *Ciągle dostarczanie oprogramowania. Automatyzacja kompilacji, testowania i wdrażania* (Helion)
- Michael T. Nygard, *Release It! Design and Deploy Production-Ready Software* (The Pragmatic Bookshelf)
- Marko Luksa, *Kubernetes In Action* (Manning)
- Gary Gruver, Tommy Mouser, *Leading the Transformation: Applying Agile and DevOps Principles at Scale* (IT Revolution Press)
- Kevin Behr, Gene Kim, George Spafford, *Visible Ops Handbook* (Information Technology Process Institute)
- Jennifer Davis, Katherine Daniels, *Effective DevOps* (O'Reilly)
- Jez Humble, Joanne Molesky, Barry O'Reilly, *Metoda Lean Enterprise. W poszukiwaniu innowacji* (O'Reilly)
- Yevgeniy Brikman, *Hello, Startup: A Programmer's Guide to Building Products, Technologies, and Teams* (O'Reilly)

Blogi

- High Scalability (<http://highscalability.com/>)
- Code as Craft (<https://codeascraft.com/>)
- dev2ops (<http://dev2ops.org/>)
- Blog AWS (<https://aws.amazon.com/blogs/aws/>)
- Kitchen Soap (<https://www.kitchensoap.com/>)
- Blog Paula Hammanta (<https://paulhammant.com/>)
- Blog Martina Fowlera (<https://martinfowler.com/bliki/>)
- Blog Gruntwork (<https://blog.gruntwork.io/>)
- Blog Yevgeniya Brikmana (<https://www.ybrikman.com/writing/>)

Prelekcje

- Yevgeniy Brikman, *Reusable, composable, battle-tested Terraform modules* (<https://www.ybrikman.com/writing/2017/10/13/reusable-composable-battle-tested-terraform-modules/>)
- Yevgeniy Brikman, *5 Lessons Learned From Writing Over 300,000 Lines of Infrastructure Code* (<https://blog.gruntwork.io/5-lessons-learned-from-writing-over-300-000-lines-of-infrastructure-code-36ba7fadeac1>)
- Yevgeniy Brikman, *Infrastructure as code: running microservices on AWS using Docker, Terraform, and ECS* (<https://www.ybrikman.com/writing/2016/03/31/infrastructure-as-code-microservices-aws-docker-terraform-ecs/>)
- Yevgeniy Brikman, *Agility Requires Safety* (<https://www.ybrikman.com/writing/2016/02/14/agility-requires-safety/>)
- Jez Humble, *Adopting Continuous Delivery* (<https://www.youtube.com/watch?v=ZLBhVEo1OG4>)
- Michael Rembetsy, Patrick McDonnell, *Continuously Deploying Culture* (<https://vimeo.com/51310058>)
- John Allspaw, Paul Hammond, *10+ Deploys Per Day: Dev and Ops Cooperation at Flickr* (<https://www.youtube.com/watch?v=LdOe18KhtT4>)
- Rachel Potvin, *Why Google Stores Billions of Lines of Code in a Single Repository* (<https://www.youtube.com/watch?v=W71BTkUbdqE>)
- Rich Hickey, *The Language of the System* (https://www.youtube.com/watch?v=ROor6_NGIWU)
- Ben Christensen, *Don't Build a Distributed Monolith* (<https://www.youtube.com/watch?v=-czp0Y4Z36Y>)
- Glenn Vanderburg, *Real Software Engineering* (<https://www.youtube.com/watch?v=NP9AIUT9nos>)

Newslettery

- Devops Weekly (<https://www.devopsweekly.com/>)
- Faun, dawniej DevOpsLinks (<https://www.faun.dev/>)
- Newsletter Gruntwork (<https://www.gruntwork.io/newsletter/>)
- Newsletter Terraform: Up & Running (<https://www.terraformupandrunning.com/>)

Fora internetowe

- Grupa Terraform w serwisie Grupy Google (<https://groups.google.com/forum/#!forum/terraform-tool>)
- DevOps Reddit (<https://www.reddit.com/r/devops/>)

O autorze

Yevgeniy (Jim) Brikman uwielbia programować, pisać, prowadzić wykłady, podróżować oraz podnosić ciężary. Jest współzałożycielem Gruntwork, firmy świadczącej usługi w zakresie praktyk DevOps. Jest także autorem innej książki wydanej przez O'Reilly Media, zatytułowanej *Hello, Startup: A Programmer's Guide to Building Products, Technologies and Teams*. Wcześniej pracował również jako inżynier oprogramowania w LinkedIn, TripAdvisor, Cisco Systems i Thomson Financial. Tytuł licencjata oraz magistra otrzymał na Uniwersytecie Cornella. Więcej informacji o autorze znajdziesz w witrynie <https://www.ybrikman.com/>.

Kolofon

Zwierzę znajdujące się na okładce książki to smok latający (łac. *Draco volans*) — mała jaszczurka, której nazwa wzięła się od możliwości latania, a raczej przenoszenia się z drzewa na drzewo, dzięki skrzydłom tworzonym przez rozpiętą pomiędzy wydłużonymi żebrami skórę nazywaną *patagia*. Te skrzydła mają jaskrawe kolory i umożliwiają jaszczurce szybowanie na odległość do 8 metrów. Smok latający występuje w wielu krajach Azji Południowej, takich jak Indonezja, Wietnam, Tajlandia, Filipiny i Singapur.

Smok latający odżywia się owadami, a dorosły osobnik może osiągać długość powyżej 20 centymetrów. Najchętniej zamieszkuje tereny zalesione, gdzie szybuje pomiędzy drzewami w poszukiwaniu pożywienia i unika w ten sposób wrogów. Samice schodzą z drzew tylko po to, aby złożyć jaja w ukrytych dziurach w ziemi. Samce są bardziej terytorialne i często ścigają rywali pomiędzy drzewami.


Chociaż kiedyś uważano go za zwierzę trujące, smok latający nie stanowi żadnego zagrożenia dla człowieka, a czasem wręcz jest traktowany jako zwierzę domowe. Gatunek nie jest obecnie zagrożony.

Wiele zwierząt prezentowanych na okładkach książek wydawanych przez O'Reilly jest zagrożonych wyginięciem; wszystkie są niezwykle istotne dla naszej planety. Jeżeli chcesz dowiedzieć się więcej o tym, jak możesz pomóc, odwiedź witrynę <https://www.oreilly.com/animals.csp>.

Ilustracja z okładki pochodzi ze zbiorów *Johnson's Natural History*.

PROGRAM PARTNERSKI

— GRUPY HELION —

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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA
Helion 

KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

NASZE SZKOLENIA SĄ PROWADZONE
ZGODNIE Z METODĄ

BLENDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie
z dostępem do nowoczesnych narzędzi - videokursów,
e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

WWW.HELIONSZKOLENIA.PL