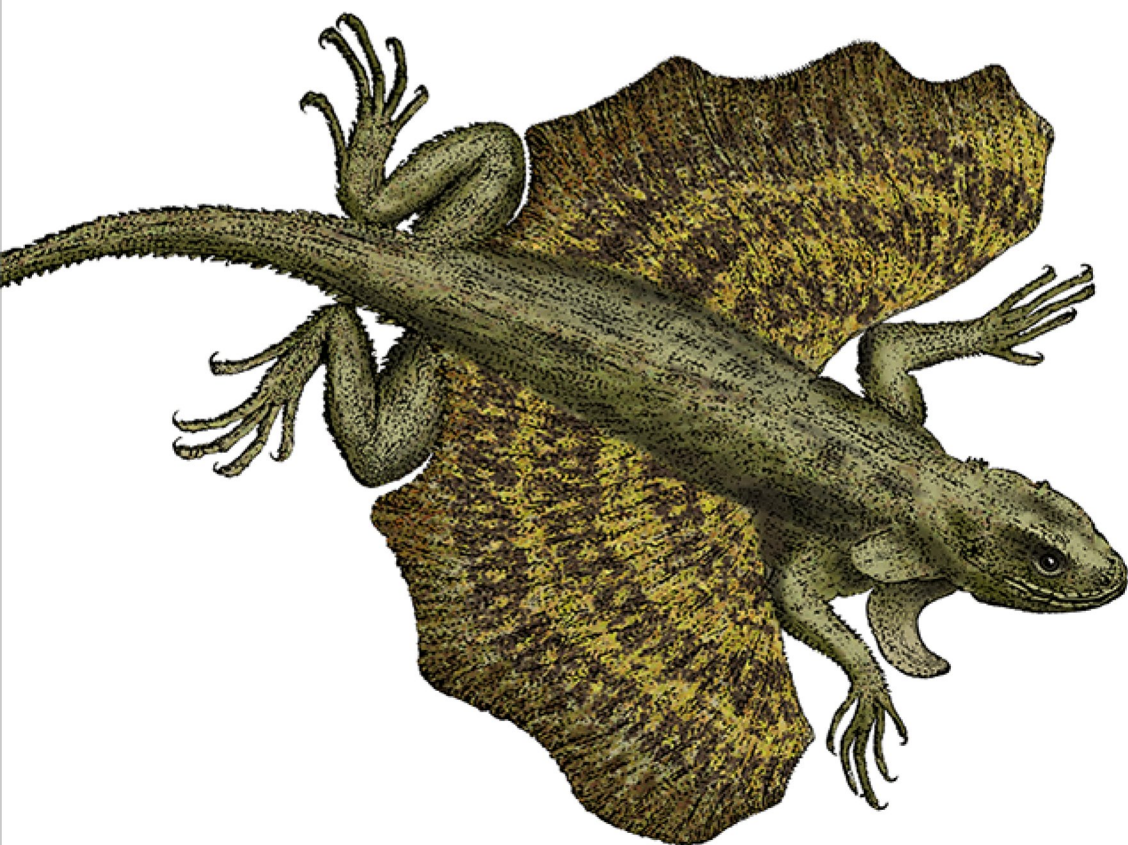


O'REILLY®

Wydanie III

Terraform

Tworzenie infrastruktury
za pomocą kodu



Helion 

Yevgeniy Brikman

Tytuł oryginału: Terraform: Up and Running: Writing Infrastructure as Code, 3rd Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-8322-347-6

© 2023 Helion S.A.

Authorized Polish translation of the English edition of *Terraform: Up and Running*, 3E ISBN 9781098116743 © 2022 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 jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

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

Autor oraz wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

https://helion.pl/user/opinie/terra3_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<https://ftp.helion.pl/przyklady/terra3.zip>

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

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

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

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

Dla Mamy, Taty, Lyalyi i Molly

Spis treści

Wprowadzenie	9
1. Dlaczego Terraform?	23
Powstanie ruchu DevOps	23
Infrastruktura jako kod	25
Skrypty tymczasowe	26
Narzędzia zarządzania konfiguracją	27
Narzędzia szablonów serwera	29
Narzędzia instrumentacji	33
Narzędzia provisioningu	35
Korzyści płynące z infrastruktury jako kodu	36
Jak działa Terraform?	38
Porównanie Terraform z innymi narzędziami IaC	40
Zarządzanie konfiguracją kontra provisioning	41
Infrastruktura niemodyfikowalna kontra modyfikowalna	41
Język proceduralny kontra deklaratywny	42
Język ogólnego przeznaczenia kontra język specjalizowany	45
Serwer główny kontra jego brak	46
Agent kontra jego brak	47
Rozwiązanie płatne kontra bezpłatne	50
Duża społeczność kontra mała	51
Rozwiązanie dojrzałe kontra najnowsze	52
Używanie razem wielu narzędzi	53
Podsumowanie	55
2. Rozpoczęcie pracy z Terraform	57
Utworzenie konta AWS	58
Instalacja Terraform	61
Wdrożenie pojedynczego serwera	62
Wdrożenie pojedynczego serwera WWW	70
Wdrażanie konfigurowalnego serwera WWW	77

Wdrażanie klastra serwerów WWW	82
Wdrożenie mechanizmu równoważenia obciążenia	86
Porządkowanie	94
Podsumowanie	95
3. Zarządzanie informacjami o stanie Terraform	96
Czym są informacje o stanie Terraform?	96
Współdzielony magazyn danych dla plików informacji o stanie	98
Ograniczenia backendu Terraform	106
Izolowanie plików informacji o stanie	107
Izolacja za pomocą przestrzeni roboczych	109
Izolacja za pomocą układu plików	114
Źródło danych terraform_remote_state	118
Podsumowanie	126
4. Zastosowanie modułów do tworzenia infrastruktury Terraform wielokrotnego użycia	127
Podstawy modułów	130
Dane wejściowe modułu	132
Wartości lokalne modułu	136
Dane wyjściowe modułu	138
Problemy z modułami	140
Ścieżki dostępu do pliku	140
Osadzony blok kodu	141
Wersjonowanie modułu	143
Podsumowanie	149
5. Sztuczki i podpowiedzi dotyczące Terraform — pętle, konstrukcje if, wdrażanie i problemy	150
Pętle	151
Pętla za pomocą parametru count	151
Pętla za pomocą wyrażenia for_each	157
Pętla za pomocą wyrażenia for	163
Pętla za pomocą dyrektywy for ciągu tekstowego	166
Wyrażenie warunkowe	167
Wyrażenie warunkowe z użyciem parametru count	167
Definiowanie warunku za pomocą for_each i wyrażen	173
Wyrażenia warunkowe wraz z dyrektywą if ciągu tekstowego	174
Wdrożenie bez przestoju	175
Problemy związane z Terraform	184
Ograniczenia parametru count i wyrażenia for_each	184
Ograniczenia wdrożenia bez przestoju	185

Awarie poprawnych planów	188
Trudności podczas refaktoryzacji	190
Podsumowanie	193
6. Zarządzanie danymi poufnymi za pomocą Terraform	194
Podstawy zarządzania danymi poufnymi	195
Narzędzia przeznaczone do zarządzania danymi poufnymi	196
Rodzaje przechowywanych danych poufnych	196
Przechowywanie danych poufnych	197
Interfejs używany w celu dostępu do danych poufnych	197
Porównanie narzędzi przeznaczonych do zarządzania danymi poufnymi	198
Narzędzia przeznaczone do zarządzania danymi poufnymi w Terraform	199
Dostawcy	199
Zasoby i źródła danych	209
Pliki informacji o stanie i pliki planu	218
Podsumowanie	220
7. Praca z wieloma dostawcami	223
Praca z pojedynczym dostawcą	223
Czym jest dostawca?	224
Jak odbywa się instalacja dostawcy?	225
W jaki sposób używać dostawców?	227
Praca z wieloma kopiami tego samego dostawcy	228
Praca z wieloma regionami AWS	228
Praca z wieloma kontami AWS	238
Tworzenie modułów, które mogą działać z wieloma dostawcami	245
Praca z wieloma różnymi dostawcami	248
Krótkie wprowadzenie do Dockera	249
Krótkie wprowadzenie do Kubernetes	252
Wdrażanie kontenerów Dockera w AWS za pomocą Elastic Kubernetes Service	262
Podsumowanie	270
8. Produkcyjny kod Terraform	271
Dlaczego przygotowanie infrastruktury o jakości produkcyjnej trwa tak długo?	273
Lista rzeczy do zrobienia podczas tworzenia infrastruktury o jakości produkcyjnej	275
Moduły infrastruktury o jakości produkcyjnej	277
Małe moduły	277
Moduły łączone z innymi	281
Moduły możliwe do testowania	286
Moduły wersjonowane	293
Moduły wykraczające poza Terraform	300
Podsumowanie	307

9. Testowanie kodu Terraform	308
Testy ręczne	309
Podstawy ręcznego przeprowadzania testów	310
Uporządkowanie środowiska po zakończeniu testów	312
Testy zautomatyzowane	313
Testy jednostkowe	314
Testy integracji	338
Testy typu E2E	351
Inne podejścia w zakresie testów	354
Podsumowanie	361
10. Używanie Terraform w zespołach	365
Adaptacja infrastruktury jako kodu przez zespół	365
Przekonanie szefa do pomysłu	366
Stopniowe wprowadzanie zmian	368
Zapewnienie zespołowi czasu na naukę	370
Sposób pracy podczas wdrażania kodu aplikacji	371
Użycie systemu kontroli wersji	372
Lokalne uruchomienie kodu	372
Wprowadzenie zmian w kodzie	373
Przekazanie zmian do zatwierdzenia	373
Uruchomienie testów zautomatyzowanych	374
Połączenie kodu istniejącego z nowym i wydanie produktu	374
Wdrożenie	376
Sposób pracy podczas wdrażania kodu infrastruktury	380
Użycie systemu kontroli wersji	380
Lokalne uruchomienie kodu	384
Wprowadzenie zmian w kodzie	385
Przekazanie zmian do zatwierdzenia	385
Uruchomienie testów zautomatyzowanych	388
Połączenie kodu istniejącego z nowym i wydanie produktu	388
Wdrożenie	389
Zebranie wszystkiego w całość	400
Podsumowanie	401
A Polecane zasoby	405

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 usługach AWS:

```
provider "aws" {  
  region = "us-east-2"  
}  
resource "aws_instance" "example" {  
  ami           = "ami-0fb653ca2d3203ac1"  
  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 niezawodne i 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 (serwery wirtualne, klastry Kubernetes, kontenery Dockera, 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 powłoce 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, Pulumi, CloudFormation, 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 bezpiecznie zarządzać informacjami poufnymi podczas pracy z Terraform?
- Jak używać Terraform z wieloma regionami, kontami i chmurami?
- 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.

Choć Terraform zdobył niezwykłą popularność, to wciąż pozostaje stosunkowo nową technologią. Pomimo jego rosnącej powszechności 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 sześć z siedmiu 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 wykorzystywanie go

przez dużą liczbę firm w różnych sytuacjach 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, Pulumi, 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 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., „Zarządzanie danymi poufnymi za pomocą Terraform”

Wprowadzenie do zarządzania danymi poufnymi; ogólne omówienie różnych typów danych poufnych; różne sposoby na przechowywanie danych poufnych i na uzyskiwanie do nich dostępu; porównanie najczęściej używanych narzędzi przeznaczonych do zarządzania danymi poufnymi: HashiCorp Vault, AWS Secrets Manager i Azure Key Vault; zarządzanie danymi poufnymi podczas pracy z dostawcami, m.in. uwierzytelnianie za pomocą zmiennych środowiskowych, role IAM i tożsamości OIDC; zarządzanie danymi poufnymi podczas pracy z zasobami i ze źródłami danych, m.in. używanie zmiennych środowiskowych, zaszyfrowanych plików i scentralizowanych magazynów danych poufnych; bezpieczne obsługiwanie plików informacji o stanie i plików planu.

Rozdział 7., „Praca z wieloma dostawcami”

Dokładniejsza analiza sposobu działania dostawców Terraform, m.in. kwestie związane z instalacją, kontrolowaniem wersji i używaniem dostawców w kodzie; wykorzystywanie wielu kopii tego samego dostawcy, m.in. wdrażanie w wielu regionach AWS, wdrażanie w wielu kontach AWS i tworzenie modułów wielokrotnego użycia, korzystających z wielu dostawców; używanie wielu różnych dostawców, m.in. przykłady zastosowania Terraform do uruchamiania klastra Kubernetes (EKS) w AWS i wdrażanie aplikacji Dockera w klastrze.

Rozdział 8., „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; weryfikacja zmiennej; wersjonowanie Terraform; dostawcy Terraform; moduły Terraform i Terragrunt; luki w zabezpieczeniach Terraform.

Rozdział 9., „Testowanie kodu Terraform”

Ręczne testowanie kodu Terraform; środowisko odizolowane i porządkowanie go; zautomatyzowane testy kodu Terraform; Terratest; testy jednostkowe kodu Terraform; testy integracji; testy typu E2E; mechanizm wstrzykiwania zależności; jednoczesne przeprowadzanie testów; etapy przeprowadzania testów; ponowne wykonywanie testów; piramida testów; analiza statyczna; planowanie testów i testowanie serwera.

Rozdział 10., „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.

Zmiany wprowadzone w wydaniu trzecim względem wydania drugiego

Pierwsze wydanie książki pojawiło się w 2017 roku, drugie w 2019 i trudno uwierzyć, że teraz, w 2022 roku, pracuję nad trzecim. Czas płynie. To niezwykle, ile się zmieniło na przestrzeni lat!

Jeżeli czytałeś drugie wydanie książki i chciałbyś się dowiedzieć, co nowego jest w trzecim wydaniu, lub po prostu jesteś ciekaw, jak wyglądała ewolucja Terraform w latach 2019 – 2022, zapoznaj się z przedstawionymi tutaj punktami, w których wymieniłem najważniejsze zmiany względem poprzedniego wydania książki:

Setki stron uaktualnionej treści

Wydanie trzecie zawiera około 100 stron więcej niż poprzednie. Oceniam również, że mniej więcej 30 – 50% stron wydania drugiego zostało uaktualnionych. Skąd się wzięło tyle zmian? Od poprzedniej publikacji pojawiło się sześć głównych wersji Terraform: 0.13, 0.14, 0.15, 1.0, 1.1 i 1.2. Co więcej, wielu dostawców Terraform również zostało uaktualnionych, m.in. dostawca AWS, który wtedy był w wersji 2, a obecnie jest w wersji 4. Ponadto w ciągu ostatnich kilku lat znacznie zwiększyła się społeczność Terraform, co doprowadziło do powstania wielu nowych dobrych praktyk, narzędzi i modułów. W wydaniu trzecim starałem się uchwycić jak najwięcej tych zmian. Pojawiły się dwa zupełnie nowe rozdziały, a wszystkie dotychczasowe zostały uaktualnione, zgodnie z przedstawionymi tutaj informacjami.

Nowa funkcjonalność dostawcy

Terraform oferuje znacznie usprawnioną pracę z dostawcami. W wydaniu trzecim dodałem zupełnie nowy rozdział, siódmy, przedstawiający sposób pracy z wieloma dostawcami, np. wdrażanie do wielu regionów, kont i chmur. Ponadto ze względu na duże zainteresowanie czytelników w rozdziale siódmym znalazło się wiele zupełnie nowych przykładów pokazujących używanie Terraform, Kubernetes, Dockera, AWS i EKS do uruchamiania aplikacji w kontenerach. Uaktualniłem także wszystkie pozostałe rozdziały w celu podkreślenia nowych funkcji dostawcy

z ostatnich kilku wydań oprogramowania, m.in. `required_providers`, czyli bloku wprowadzonego w Terraform 0.13, pliku blokady wprowadzonego w Terraform 0.14 i parametru `configuration_aliases` wprowadzonego w Terraform 0.15.

Lepsza obsługa informacji poufnych

Podczas używania kodu Terraform często zachodzi potrzeba pracy z wieloma rodzajami informacji poufnych: hasłami baz danych, kluczami API, danymi uwierzytelniającymi dostawcy chmury, certyfikatami TLS itd. W wydaniu trzecim dodałem zupełnie nowy rozdział, szósty, poświęcony temu zagadnieniu. Znajdziesz w nim m.in. porównanie najczęściej spotykanych narzędzi przeznaczonych do zarządzania informacjami poufnymi, wiele przykładowych fragmentów kodu pokazujących różne techniki bezpiecznego używania informacji poufnych w Terraform, takie jak zmienne środowiskowe, szyfrowane pliki, scentralizowane magazyny informacji poufnych, role IAM, tożsamości OIDC itd.

Nowa funkcjonalność modułu

W wersji 0.13 Terraform dodano możliwość używania parametru `count`, wyrażeń `for_each` i `depends_on` w blokach `module`. Dzięki temu moduły mają zdecydowanie większe możliwości, stały się elastyczniejsze i można ich wielokrotnie używać. Przykłady wykorzystania tych nowych funkcjonalności znajdziesz w rozdziałach 5. i 7.

Nowa funkcjonalność weryfikacji

W rozdziale 8. dodałem przykłady użycia wprowadzonej w Terraform 0.13 funkcjonalności `validation`, przeznaczonej do prostego sprawdzania zmiennych (np. wymuszanie zastosowania wartości minimalnej i maksymalnej), a także wprowadzonych w Terraform 1.2 funkcjonalności `precondition` i `postcondition`, przeznaczonych do prostego sprawdzania zasobów i źródeł danych przed wykonaniem polecenia `terraform apply` (np. wymuszenie, by użytkownik AMI korzystał z architektury `x86_64`) lub już po jego wykonaniu (np. sprawdzenie, czy użyty wolumin EBS został zaszyfrowany). W rozdziale 6. pokazałem, jak używać wprowadzonego w Terraform 0.14 i 0.15 parametru `sensitive`, który gwarantuje, że informacje poufne nie będą rejestrowane podczas wykonywania poleceń `terraform plan` lub `terraform apply`.

Nowa funkcjonalność refaktoryzacji

W Terraform 1.1 wprowadzono blok `moved`, zapewniający znacznie lepszy sposób obsługi określonych typów refaktoryzacji, np. zmiany nazwy zasobu. Wcześniej ten rodzaj refaktoryzacji wymagał od użytkowników ręcznego i podatnego na błędy wykonywania operacji `terraform state mv`, podczas gdy teraz, jak zobaczysz w nowym przykładzie zamieszczonym w rozdziale 5., ten proces można w pełni zautomatyzować. Dzięki temu uaktualnienia stają się bezpieczniejsze i bardziej zgodne.

Więcej opcji w zakresie testów

Narzędzia przeznaczone do zautomatyzowanego testowania kodu Terraform są nieustannie usprawniane. W rozdziale 9. dodałem przykład i porównanie narzędzi analizy statycznej dla Terraform, m.in. `tfsec`, `tflint`, `terrascan` i polecenie `validate`; narzędzia testowania polecenia `terraform plan` w Terraform, m.in. `Terratest`, `OPA` i `Sentinel`; narzędzia testowania serwera, m.in. `inspec`, `serverspec` i `goss`. Dodałem również porównanie wszystkich rozwiązań w zakresie testowania, co pozwala wybrać najodpowiedniejsze w danej sytuacji.

Większa stabilność

Wydanie 1.0 to krok milowy w rozwoju Terraform. Nie tylko świadczy o tym, że narzędzie osiągnęło określony poziom dojrzałości, ale również wiąże się z wieloma obietnicami dotyczącymi zgodności. To przede wszystkim obietnica, że wszystkie wydania 1.x będą zgodne wstecz, więc uaktualnienie wersji 1.x nie powinno wymagać zmian w kodzie, rozwiązaniu lub plikach stanu. Pliki stanu Terraform są teraz zgodne z wydaniem Terraform 0.14 i 0.15, a także ze wszystkimi wydaniami 1.x. Źródła danych zdalnego stanu Terraform są zgodne z wydaniami: 0.12.30, 0.13.6, 0.14.0, 0.15.0 i wszystkimi wydaniami 1.x. Uaktualniłem też rozdział 8. poprzez dodanie egzemplarzy pokazujących, jak lepiej zarządzać wersjonowaniem Terraform (za pomocą `tfenv`), Terragrunt (m.in. używając do tego `tgswitch`) i dostawcami Terraform (np. jak używać pliku blokady).

Większa dojrzałość

Oprogramowanie Terraform pobrano ponad 100 milionów razy, pracuje nad nim ponad 1500 osób i jest używane przez mniej więcej 79% firm z listy Fortune 500¹. Można więc stwierdzić, że na przestrzeni ostatnich lat ekosystem Terraform znacznie się zwiększył i dojrzał. Jeszcze większa rzesza programistów używa Terraform, mamy więcej dostawców, modułów wielokrotnego użycia, narzędzi, wtyczek, klas, książek i samouczków Terraform niż kiedykolwiek wcześniej. A do tego firma HashiCorp, która opracowała Terraform, w 2021 roku przeprowadzała pierwszą ofertę publiczną. Tym samym za Terraform nie stoi już mały startup, a ogromna, stabilna i publicznie notowana firma, dla której Terraform to najważniejszy produkt.

Wiele innych zmian

Wprowadzono wiele innych zmian, m.in. Terraform Cloud (bazujący na przeglądarce WWW graficzny interfejs użytkownika dla Terraform), usprawnione, popularne i opracowane przez społeczność narzędzia, takie jak Terragrunt i Terratest, oraz polecenie `tfenv`; dodano wiele nowych funkcjonalności związanych z dostawcami (np. nowe sposoby na wdrażanie bez przestoju i natychmiastowe wdrożenie, na którego temat więcej dowiesz się w rozdziale 5.), a także nowe funkcje, np. przykłady użycia funkcji `one()` (rozdział 5.) i `try()` (rozdział 7.); usunięto sporo starych funkcjonalności — np. źródło danych `template_file`, wiele parametrów `aws_s3_bucket`, `listmap`, obsługa zewnętrznych odniesień do narzędzia provisioningu `destroy` — itd.

Zmiany wprowadzone w wydaniu drugim względem wydania pierwszego

Cofnijmy się jeszcze bardziej w czasie: wydanie drugie zawierało około 150 stron nowej treści dodanej względem wydania pierwszego. Oto krótkie podsumowanie tych zmian, pokazujące również, co się zmieniło w Terraform w latach 2017 – 2019:

¹ Zgodnie z informacjami zamieszczonymi na stronie HashiCorp S1 (<https://www.sec.gov/Archives/edgar/data/1720671/000119312521319849/d205906ds1.htm>).

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 aż do wersji 0.12. W tym wydaniu znalazła się niesamowita nowa funkcjonalność, o której wspomniałem w wydaniu drugim, choć dla użytkowników to oznacza znacznie więcej pracy podczas uaktualniania oprogramowania².

Usprawnienia w zakresie testów zautomatyzowanych

Narzędzia i praktyki w zakresie tworzenia testów zautomatyzowanych dla kodu Terraform uległy dużym zmianom w latach 2017 – 2019. W wydaniu drugim pojawił się całkowicie nowy rozdział 7. poświęcony testowaniu, przedstawiał 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 wydaniu drugim pojawił się całkowicie nowy rozdział 6., zawierający 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. w wydaniu drugim został napisany całkowicie od początku, aby odzwierciedlić zmiany w sposobach, na jakie zespoły włączają Terraform do swojego działania. Zamieściłem 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, rozbudowany system ograniczeń, wyrażenia warunkowe obliczane z opóźnieniem, obsługę wyrażeń null, for_each i for, dynamiczne bloki kodu wewnętrznego itd. Wszystkie przykłady przedstawione w wydaniu drugim zostały uaktualnione do HCL2, a dokładne omówienie funkcjonalności oferowanej przez nowy język zamieszczono 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 zostały w wydaniu drugim omówione w rozdziale 3.

² Zapoznaj się z przewodnikami uaktualniania Terraform, które znajdziesz na stronie <https://www.terraform.io/language/upgrade-guides>.

Podział dostawców Terraform

W wydaniu 0.10 podstawowy kod Terraform został oddzielony od kodu obsługującego poszczególne wszystkie 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 modulem. Więcej informacji na ten temat w wydaniu drugim przedstawiłem w rozdziałach 2. i 7.

Ogromny wzrost liczby dostawców

Od 2016 do 2019 roku zwiększyła się lista dostawców chmur obsługiwanych przez Terraform, od (jedynie) AWS, GCP i Azure do ponad 100 oficjalnych dostawców i wielu opracowanych przez społeczność³. 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 w wydaniu drugim zamieszczono 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, te dane były zapisywane lokalnie w pliku o nazwie `errored.tfstate`. Wydanie 0.12 zawiera całkowicie przeprojektowaną obsługę błędów, która teraz przechwytytuje 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

Miedzy 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`

³ Lista dostawców Terraform znajduje się na stronie <https://registry.terraform.io/browse/providers>.

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/>.

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 Cloud 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 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 umieszczone w katalogu `code` i ułożone przede wszystkim według narzędzia lub języka (np. Terraform, Packer, OPA), a następnie według rozdziału. Wyjątkiem jest kod

w języku Go użyty dla testów zautomatyzowanych w rozdziale 9. — ten kod znajduje się w katalogu *terraform*. Został umieszczony w zalecanym w rozdziale układzie katalogów: *examples*, *modules* i *test*. W tabeli 0.1 zamieściłem kilka przykładów pokazujących, gdzie możesz znaleźć różne typy przykładowych fragmentów kodu w repozytorium.

Tabela 0.1. Położenie różnych typów przykładowych fragmentów kodu w repozytorium

Typ kodu źródłowego	Rozdział	Nazwa katalogu w repozytorium
Terraform	2.	<i>code/terraform/02-intro-to-terraform-syntax</i>
Terraform	5.	<i>code/terraform/05-tips-and-tricks</i>
Packer	1.	<i>code/packer/01-why-terraform</i>
OPA	9.	<i>code/opa/09-testing-terraform-code</i>
Go	9.	<i>code/terraform/09-testing-terraform-code/test</i>

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, a „oficjalne” rozwiązania sprawdzaj jedynie na koniec pracy.

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.

0 wersjach Terraform

Wszystkie przykłady przedstawione w książce zostały przetestowane wraz z wydaniem Terraform 1.x i AWS Provider 4.x, które były najnowsze w chwili powstawania książki. Skoro Terraform to względnie nowe narzędzie, 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. Tworzenie infrastruktury za pomocą kodu*. Wydanie III, ISBN 978-83-8322-346-9, Helion, Gliwice 2023.

Konwencje zastosowane w książce

W tej książce zastosowano następujące konwencje typograficzne:

Kursywa

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

Pogrubienie

Wskazuje nowe pojęcia.

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 wskazówkę.



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, Virginii Wilson, która pomogła mi przy drugim wydaniu, i Corbina Collinsa za pomoc nad trzecim wydaniem.

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 trzeciego 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, Taylor Dolezal i Anton Babenko

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

Czytelnicy pierwszego i drugiego wydania

Dzięki czytelnikom pierwszego i drugiego wydania książki możliwe było przygotowanie trzeciej edycji. Dziękuję wam za to. Wasze uwagi, pytania, zgłoszenia i nieustanne zgłaszanie uaktualnień zmotywowały mnie do napisania wielu 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 oprogramowaniem 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, Docker i Kubernetes. 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 wykorzystwała 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, wymieniłem 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, wdrożenia, uaktualnienia i usunięcia infrastruktury. To pokazuje ważną zmianę w nastawieniu, polegającą na tym, że wszystkie aspekty operacji są traktowane jako oprogramowanie — nawet te związane ze sprzętem (np. fizyczne przygotowanie serwera do pracy). 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:

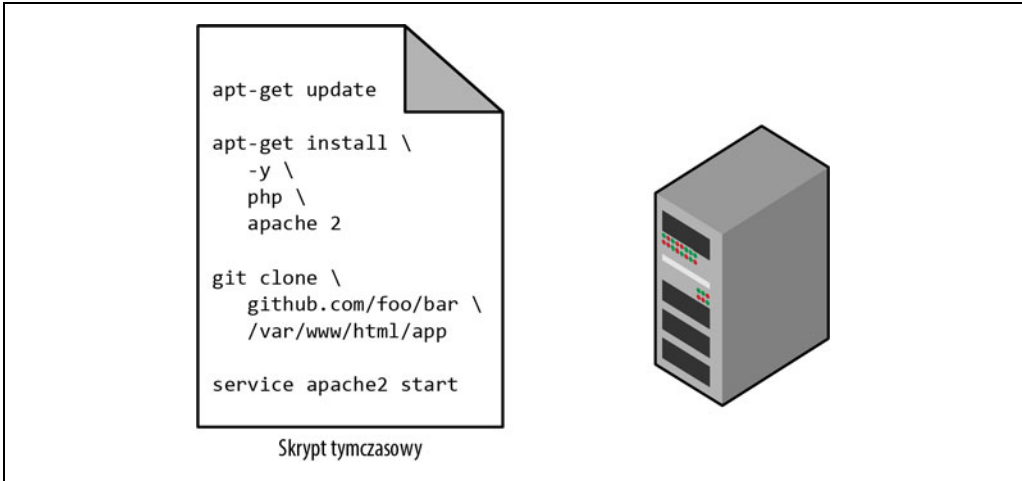
- skrypty tymczasowe,
- narzędzia zarządzania konfiguracją,
- narzędzia szablonów serwera,
- narzędzia instrumentacji,
- narzędzia provisioningu.

Dalej po kolei przedstawię te kategorie.

¹ Te informacje pochodzą z książki *DevOps. Światowej klasy zwinność, niezawodność i bezpieczeństwo w Twojej organizacji* Helion, Gliwice 2017, 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. Najprostszy sposób automatyzacji polega na utworzeniu skryptu tymczasowego i jego wykonywaniu 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/brikis98/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, zaś w przypadku języków programowania ogólnego przeznaczenia każdy programista ma własny styl i inaczej

wykonuje pewne zadania. Żadna z wymienionych kwestii nie stanowi poważnego problemu w ośmioletnim 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 i Ansible 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

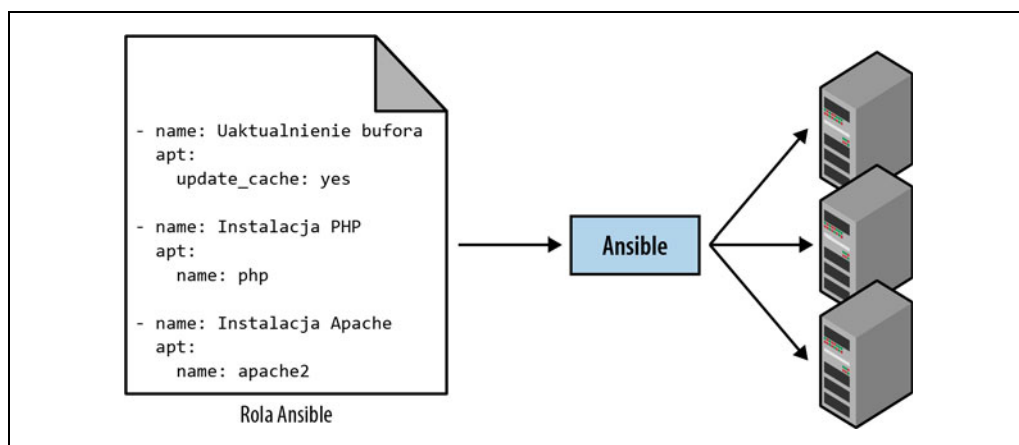
Utworzenie jednorazowo wykonywanego skryptu tymczasowego nie należy do zbyt trudnych zadań. Z kolei opracowanie skryptu tymczasowego, który będzie działał prawidłowo nawet

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. *scenariusz Ansible*:

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

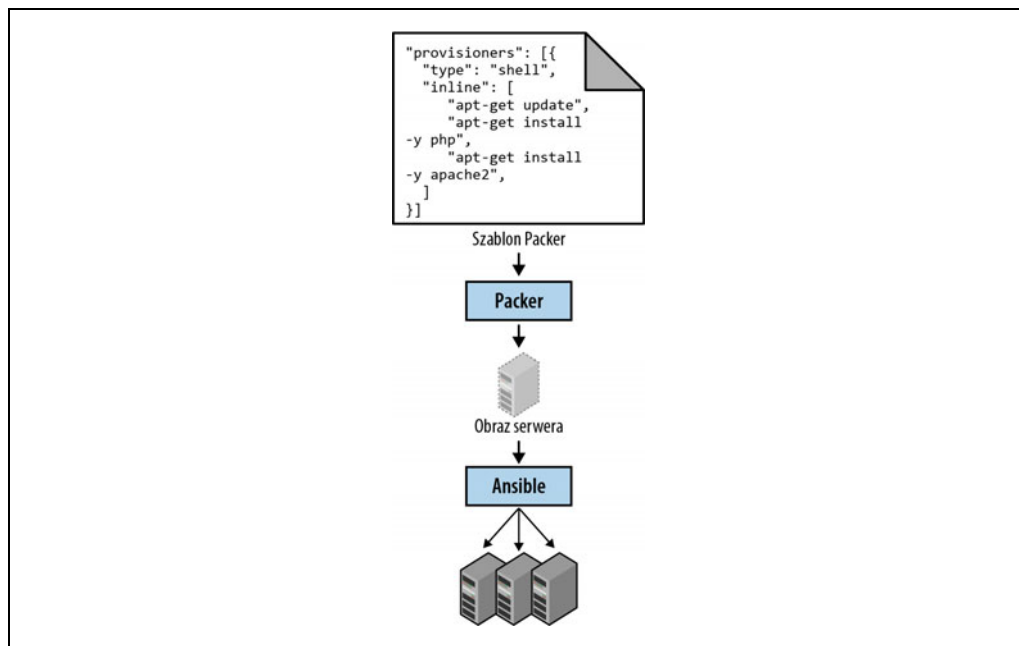
Na końcu można za pomocą przedstawionego polecenia wykonać zdefiniowany kod scenariusza (ang. *playbook*):

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

To nakazuje Ansible równoczesne skonfigurowanie wszystkich pięciu serwerów. Ewentualnie za pomocą polecenia o nazwie `serial` umieszczonego we wspomnianym scenariuszu 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.

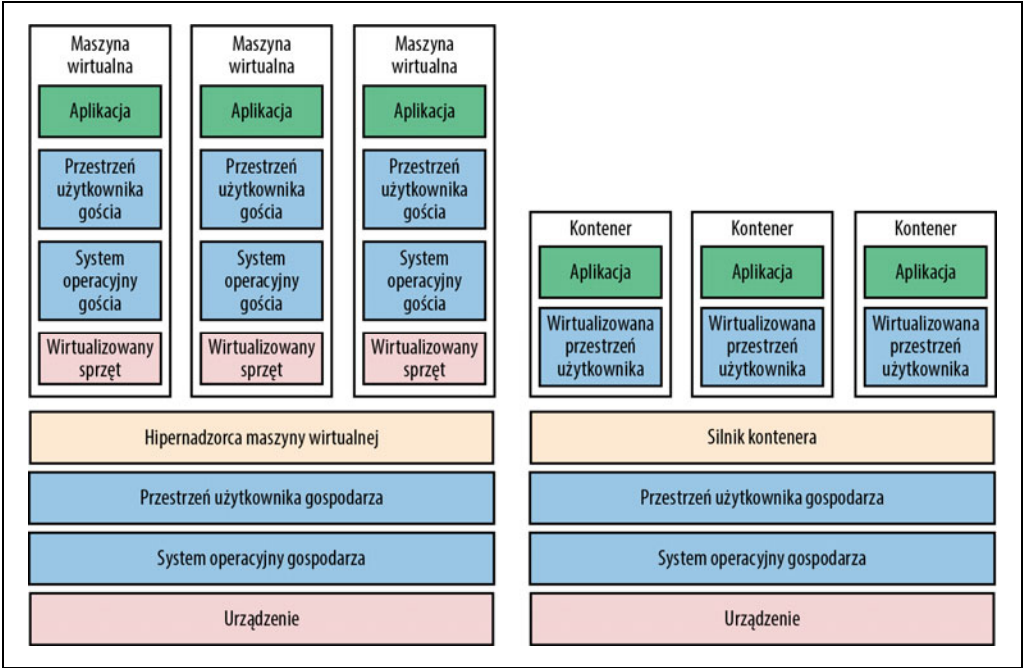
Narzędzia szablonów serwera

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.



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

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



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

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.

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 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 stanowi żadnego obciążenia dla procesora i pamięci. Obrazy kontenerów jako kodu można zdefiniować za pomocą takich narzędzi jak Docker i CoreOS rkt. Przykład użycia Dockera pokażę w rozdziale 7.

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-0fb653ca2d3203ac1",
    "ssh_username": "ubuntu"
  }],
  "provisioners": [{
    "type": "shell",
    "inline": [
      "sudo apt-get update",
```

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. 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.

```

    "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"
  ],
  "pause_before": "60s"
}]
}

```

Ten szablon narzędzia Packer przeprowadza tę samą konfigurację serwera WWW Apache, którą wcześniej widziałeś w przykładzie *setup-webserver.sh*, używając tego samego kodu Bash. Jedyną różnicą 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), przeznaczone do rzeczywistego uruchomienia tego oprogramowania.

Utworzenie obrazu AMI na podstawie tego szablonu odbywa się po wydaniu polecenia `packer build webserver.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 bezpośrednio 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. 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

Wprowadzając 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

```

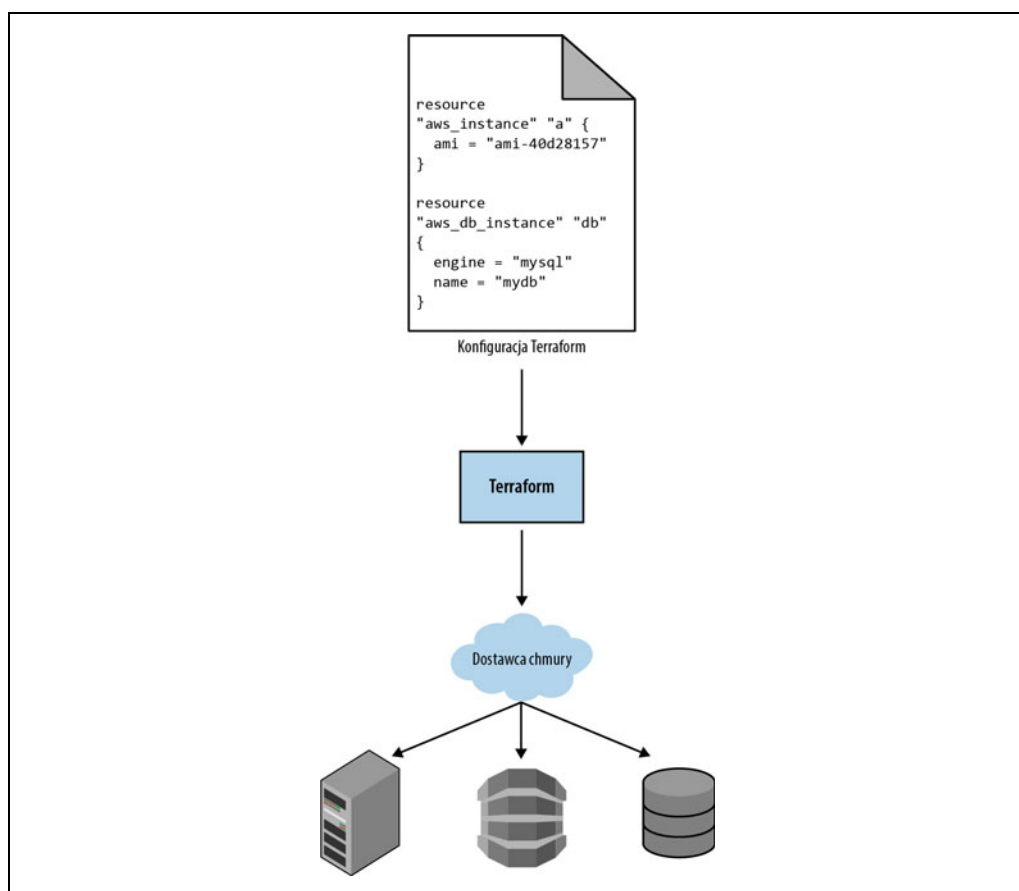
Ten plik nakazuje Kubernetesowi utworzenie tzw. *obiektu Deployment*, czyli deklaratywnego rozwiązania pozwalającego na zdefiniowanie następujących kwestii:

- 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 żądania.
- 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. Istnieje również możliwość użycia Terraform do zarządzania klastrem Kubernetes i wdrożonymi w nim aplikacjami. Przykłady takich rozwiązań przedstawię w rozdziale 7.

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, OpenStack Heat i Pulumi, 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 podsieci, ustawienia zapory sieciowej, reguły routingu, certyfikaty SSL (ang. *secure socket layer*) i praktycznie każdy inny aspekt infrastruktury, jak pokazałem na rysunku 1.5.



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

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-0fb653ca2d3203ac1"

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

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

Jeżeli informacje o stanie infrastruktury zostaną zamknięte w głowie jednego administratora systemu, który pójdzie na urlop, opuści firmę lub zostanie potrącony przez autobus⁴, wówczas może się okazać, że nie jesteś w stanie dłużej zarządzać własną infrastrukturą. Natomiast jeśli stan infrastruktury zostanie zdefiniowany w plikach kodu źródłowego, będą one 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.

⁴ W tym miejscu pojawia się wyrażenie *współczynnik autobusu*: określa liczbę osób, które można stracić (np. w wyniku potrącenia przez autobus), zanim nie będzie możliwości dalszego funkcjonowania firmy. Nigdy nie chcesz, aby wartość tego współczynnika wynosiła 1.

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

Szczęście

Jest jeszcze jeden bardzo ważny i zarazem często niedoceniany powód, dla którego powinienś 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 nieprzyjemne 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.

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 (a dokładnie 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).

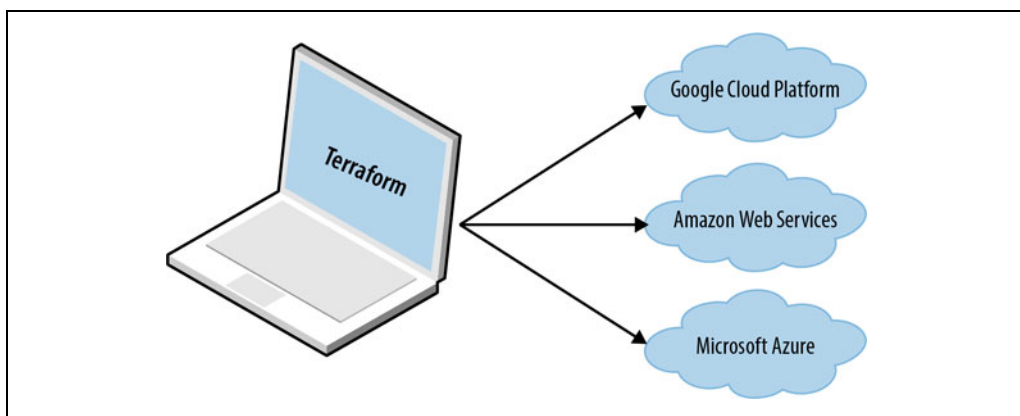
Skąd Terraform wie, które wywołanie API ma zostać wykonane. Odpowiedź kryje się w stworzonych *konfiguracjach Terraform*, które są plikami tekstowymi określającymi infrastrukturę przeznaczoną do utworzenia. Wspomniane konfiguracje to „kod” w wyrażeniu „infrastruktura jako kod”. Spójrz na przykładową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

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}

resource "google_dns_record_set" "a" {
  name         = "demo.google-example.com"
  managed_zone = "example-zone"
  type         = "A"
  ttl          = 300
  rrrdatas     = [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 wywołanie API do AWS w celu wdrożenia serwera, a następnie wywołanie 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 wykorzystasteś 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 zaledwie kilku poleceń?

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 jednocześnie sporo oferuje komercyjną pomoc techniczną. O ile wcześniej nie zdarzyło Ci się używać 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, Pulumi, 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,

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.

- język proceduralny kontra deklaracyjny,
- język ogólnego przeznaczenia kontra język specjalizowany,
- serwer główny kontra jego brak,
- agent kontra jego brak,
- rozwiązanie płatne kontra bezpłatne,
- duża społeczność kontra mała,
- rozwiązanie dojrzałe kontra najnowsze,
- używanie razem wielu narzędzi.

Zarządzanie konfiguracją kontra provisioning

Jak miałeś okazję zobaczyć wcześniej, Chef, Puppet i Ansible to narzędzia zarządzania konfiguracją, podczas gdy CloudFormation, Terraform, OpenStack Heat i Pulumi to narzędzia provisioningu.

Wprawdzie granica między tymi typami narzędzi nie jest do końca wyraźnie ustalona, ale 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 sytuacji.

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. W rozdziale 7. poznasz przykłady używania Terraform i Dockera razem — takie połączenie jest obecnie szczególnie popularne.

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 Ansible.

Infrastruktura niemodyfikowalna kontra modyfikowalna

Narzędzia konfiguracji takie jak Chef, Puppet i Ansible zwykle domyślnie stosują paradygmat infrastruktury niemodyfikowalnej.

Przykładowo, jeśli nakazesz 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 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, SaltStack, 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-0fb653ca2d3203ac1` (Ubuntu 20.04). Spójrz na uproszczony przykład szablonu Ansible pokazujący, jak osiągnąć żądany efekt za pomocą podejścia proceduralnego.

```
- ec2:
  count: 10
  image: ami-0fb653ca2d3203ac1
  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-0fb653ca2d3203ac1"
  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-0fb653ca2d3203ac1
  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-0fb653ca2d3203ac1"
  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-0fb653ca2d3203ac1"
+   instance_type = "t2.micro"
+   (...)
+ }

# aws_instance.example[12] will be created
+ resource "aws_instance" "example" {
+   ami           = "ami-0fb653ca2d3203ac1"
+   instance_type = "t2.micro"
+   (...)
+ }
```

```
# aws_instance.example[13] will be created
+ resource "aws_instance" "example" {
  + ami           = "ami-0fb653ca2d3203ac1"
  + instance_type = "t2.micro"
  + (...)
}

# aws_instance.example[14] will be created
+ resource "aws_instance" "example" {
  + ami           = "ami-0fb653ca2d3203ac1"
  + 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 dokładanie 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, strefie dostępności, a także na podstawie innych parametrów. 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, w przypadku bazy kodu Ansible lub Chef 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 opisaniu żądanego stanu, a Terraform ustali, jak automatycznie przejść z jednego stanu do drugiego. W efekcie baza kodu Terraform zwykle pozostaje mała i łatwa do zrozumienia.

Język ogólnego przeznaczenia kontra język specjalizowany

Chef i Pulumi pozwalają na używanie **języków programowania ogólnego przeznaczenia** (ang. *general-purpose programming language*, GPL) podczas zarządzania infrastrukturą jako kodem: Chef obsługuje język Ruby, Pulumi zaś obsługuje wiele tego rodzaju języków, np. JavaScript, TypeScript, Python, Go, C#, Javę i inne. Z kolei Terraform, Puppet, Ansible, CloudFormation i OpenStack Heat do obsługi infrastruktury jako kodu używają **języka specjalizowanego** (ang. *domain-specific language*, DSL): w przypadku Terraform to HCL, Puppet używa Puppet Language, a Ansible, CloudFormation i Open Stack Heat korzystają z YAML (CloudFormation obsługuje również JSON).

Granica między językami ogólnego przeznaczenia i specjalizowanym nie jest ściśle ustalona — to bardziej model mentalny niż jasno wyznaczona granica. Jednak podstawowa idea polega na tym, że język specjalizowany jest przeznaczony do korzystania w konkretnej domenie, podczas gdy język ogólnego przeznaczenia można stosować w szerokiej gamie domen. Przykładowo kod w języku HCL tworzony dla Terraform działa jedynie w Terraform i jest ograniczony wyłącznie do funkcjonalności obsługiwanej przez Terraform, np. wdrażanie infrastruktury. Mamy tutaj przeciwieństwo względem użycia języka programowania ogólnego przeznaczenia, np. JavaScriptu w Pulumi, ponieważ wówczas tworzony kod może nie tylko zarządzać infrastrukturą za pomocą bibliotek Pulumi, ale również praktycznie wykonywać dowolne zadania programistyczne, jak uruchomienie aplikacji internetowej (w rzeczywistości Pulumi oferuje API automatyzacji pozwalające na osadzenie Pulumi w kodzie aplikacji), zdefiniowanie skomplikowanej logiki kontrolnej (pętle, konstrukcje warunkowe i abstrakcje — to wszystko jest łatwiejsze w GPL niż w DSL), wykonywanie różnych operacji sprawdzania poprawności i testów, integracja z innymi narzędziami i API itd.

Język specjalizowany ma wiele zalet w porównaniu z językiem ogólnego przeznaczenia.

Łatwiejszy w nauce

Skoro język specjalizowany z natury zajmuje się tylko jedną dziedziną, zwykle będzie mniejszy i prostszy niż język programowania ogólnego przeznaczenia, a tym samym łatwiejszy w poznawaniu niż znaczna część języków typu GPL. Większość programistów będzie w stanie opanować Terraform zdecydowanie szybciej niż np. Javę.

Bardziej przejrzysty i zwięzły

Skoro język specjalizowany został przeznaczony do konkretnego celu, wszystkie wbudowane w nim słowa kluczowe zostały przygotowane do wykonywania pojedynczego celu, a kod utworzony w języku specjalizowanym zwykle jest łatwiejszy do zrozumienia i zwięźlejszy niż przeznaczony do tego samego celu kod opracowany w języku programowania ogólnego przeznaczenia.

Kod odpowiedzialny za wdrożenie pojedynczego serwera w AWS zwykle będzie krótszy i łatwiejszy do zrozumienia po utworzeniu go w Terraform niż np. w Javie.

Bardziej ujednolicony

Większość języków specjalizowanych ma ograniczone możliwości. Wiąże się to z pewnymi wadami, do czego jeszcze powrócę, jedną z zalet natomiast jest to, że kod utworzony w języku specjalizowanym zwykle korzysta z ujednoliconej, przewidywalnej struktury. Dlatego też prościej się po nim poruszać i łatwiej go zrozumieć niż kod utworzony w języku specjalizowanym ogólnego przeznaczenia, w którym ten sam problem może być całkowicie odmiennie rozwiązywany przez różnych programistów. Naprawdę istnieje tylko jeden sposób na wdrożenie serwera w AWS za pomocą Terraform i są setki sposobów na zrobienie tego samego w Javie.

Z kolei język ogólnego przeznaczenia ma kilka zalet w porównaniu z językiem specjalizowanym.

Prawdopodobnie brak konieczności poznawania czegokolwiek nowego

Skoro język programowania ogólnego przeznaczenia jest używany w wielu domenach, istnieje prawdopodobieństwo, że w ogóle nie trzeba będzie się uczyć nowego języka. To w szczególności dotyczy narzędzia Pulumi obsługującego wiele najpopularniejszych języków programowania, m.in. JavaScript, Pythona i Javę. Jeżeli znasz Javę, pracę z Pulumi możesz rozpocząć znacznie szybciej niż w sytuacji, w której musisz poznać HCL, aby zacząć korzystać z Terraform.

Większy ekosystem i więcej dojrzałych narzędzi

Skoro język programowania ogólnego przeznaczenia jest używany w wielu domenach, ma znacznie większą społeczność użytkowników i bardziej dopracowane narzędzia niż w wypadku typowego języka specjalizowanego. Liczba i jakość zintegrowanych środowisk programistycznych, tzw. IDE (ang. *integrated development environment*), bibliotek, wzorców, narzędzi testowania itd. dla Javy znacznie przekracza liczbę tego rodzaju dodatków dostępnych dla Terraform.

Większe możliwości

Język programowania ogólnego przeznaczenia z natury może być użyty do wykonania praktycznie dowolnego zadania programistycznego, zapewnia więc dużo większe możliwości i funkcjonalność niż język specjalizowany. Konkretnie zadania, takie jak konstrukcje sterowania (pętle i wyrażenia warunkowe), zautomatyzowane testy, wielokrotne używanie kodu, abstrakcja i integracja z innymi narzędziami, są znacznie łatwiejsze w Javie niż w wypadku Terraform.

Serwer główny kontra jego brak

Domyślnie Chef i Puppet 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 otwarcie 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, Terraform i Pulumi 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 Terraform 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 i Puppet wymagają zainstalowania **oprogramowania agenta** (np. Chef Client, Puppet Agent) 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.

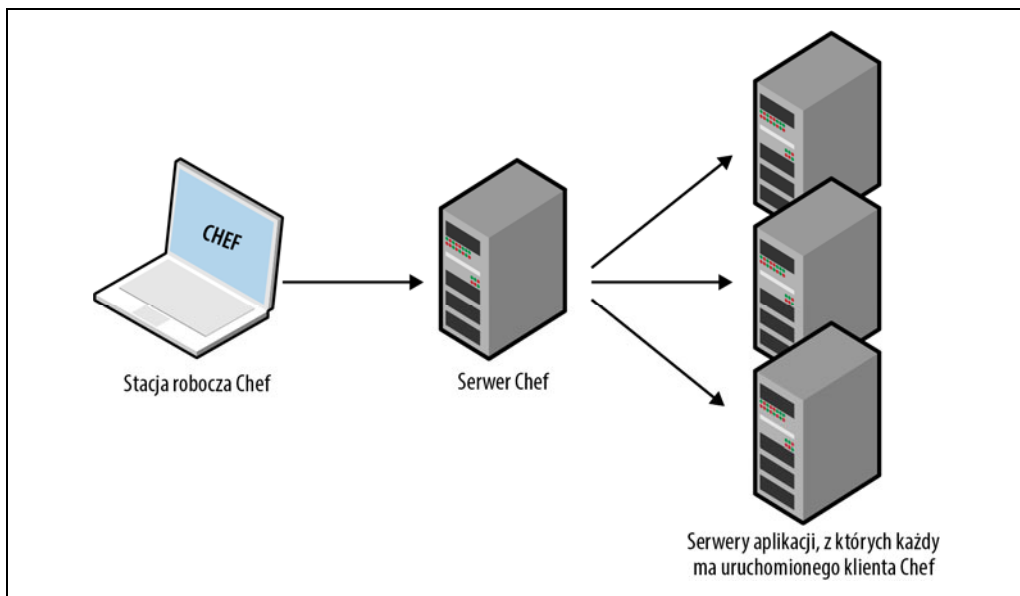
Bezpieczeństwo

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.

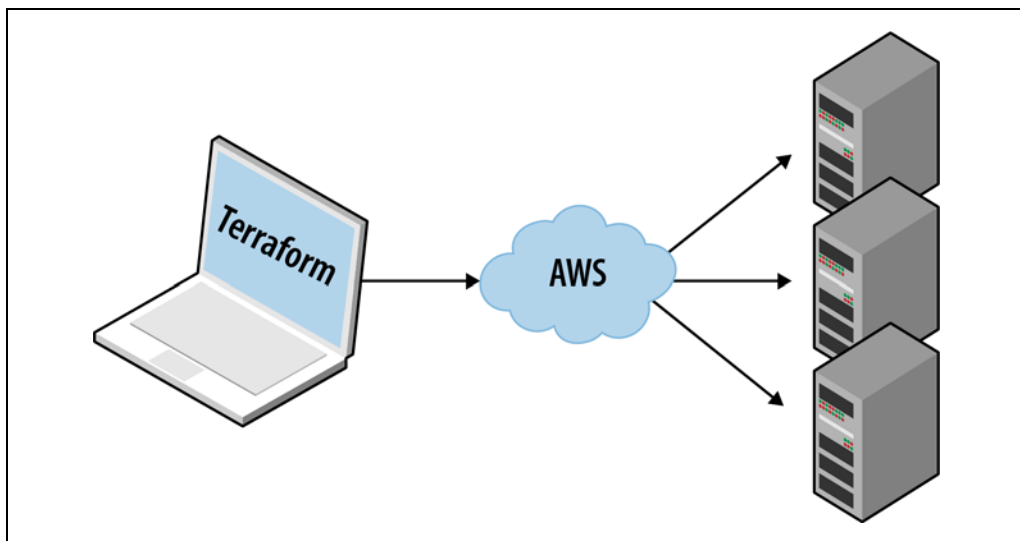
Także i w tym zakresie Chef i Puppet różnią się poziomem obsługi dla trybów pracy bez agenta, ale należy mieć świadomość, że taki tryb nie zapewnia dostępu do pełnych możliwości narzędzia zarządzania konfiguracją. Dlatego też w rzeczywistych wdrożeniach domyślna konfiguracja dla Chef i Puppet niemal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

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, Terraform i Pulumi 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.



Rysunek 1.7. Typowa architektura Chef i Puppet 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



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

Rozwiązanie płatne kontra bezpłatne

CloudFormation i OpenStack Heat to produkty całkowicie bezpłatne; wprawdzie wdrażane przy ich pomocy zasoby mogą kosztować, ale nie trzeba ponosić żadnych kosztów związanych z używaniem wymienionych narzędzi. Terraform, Chef, Puppet, Ansible i Pulumi są dostępne w wersjach bezpłatnych i płatnych. Można np. skorzystać z bezpłatnej i dostępnej jako oprogramowanie typu open source wersji Terraform albo też użyć płatnego produktu HashiCorp o nazwie Terraform Cloud. Ceny, dostępne pakiety i różnice wersji płatnych to wszystko kwestie wykraczające poza zakres tematyczny książki. W tym miejscu jednak chciałbym się skupić na ustaleniu, czy wersja bezpłatna jest na tyle ograniczona, że w środowisku produkcyjnym praktycznie *wymusza* użycie wersji płatnej.

Chcę wyraźnie powiedzieć, że nie ma niczego złego w oferowaniu przez firmę płatnej usługi dla dowolnego z wymienionych narzędzi. W rzeczywistości, jeśli używasz tych narzędzi w środowisku produkcyjnym, gorąco zachęcam do użycia płatnych usług, wiele z nich bowiem jest wartych każdego wydanego grosza. Trzeba jednak pamiętać, że usługa płatna jest poza Twoją kontrolą — nagle może przestać być dostępna lub oferująca ją firma zostanie przejęta (Chef, Puppet i Ansible to produkty, które zostały przejęte, co miało poważny wpływ na ofertę ich płatnych wersji) albo też zmieni się model cen (np. w wypadku Pulumi ceny zmieniły się w 2021 roku, część użytkowników skorzystała na zmianach, z kolei dla innych to oznaczało 10-krotną podwyżkę). Ponadto produkt może ulec zmianie lub zostać całkowicie wycofany z rynku — należy wiedzieć, czy wybrane narzędzie typu IaC pozostanie użyteczne, gdy z jakiegokolwiek powodu przestaną być dla niego oferowane płatne usługi.

Z mojego doświadczenia wynika, że bezpłatne wersje Terraform, Chef, Puppet i Ansible mogą być wykorzystywane w środowiskach produkcyjnych. Ich płatne wersje są jeszcze lepsze, ale jeśli nie byłyby dostępne, nadal można korzystać z wersji bezpłatnych. Z kolei Pulumi to narzędzie, którego w wersji bezpłatnej trudno używać w środowisku produkcyjnym — najlepiej skorzystać z płatnego wydania znanego pod nazwą Pulumi Service.

Kluczowym aspektem zarządzania infrastrukturą jako kodem jest zarządzanie stanem (z rozdziału 3. dowiesz się, jak Terraform zarządza informacjami o stanie), a Pulumi domyślnie używa Pulumi Service jako backendu dla magazynu danych. Wprawdzie istnieje możliwość przejścia do innego obsługiwanego backendu dla magazynu danych informacji o stanie, np. Amazon S3, Azure Blob Storage lub Google Cloud Storage, ale dokumentacja backendu Pulumi (<https://www.pulumi.com/docs/intro/concepts/state/>) wyjaśnia, że tylko Pulumi Service obsługuje transakcyjne punkty kontrolne (na potrzeby odporności na awarie i odzyskiwania), współbieżne blokowanie stanu (w celu uniknięcia uszkodzenia informacji o stanie infrastruktury w środowisku zespołowym) oraz zapewnia szyfrowanie informacji o stanie podczas ich przekazywania i przechowywania. Według mnie bez tych funkcjonalności nie ma praktycznego sensu używanie Pulumi w jakimkolwiek środowisku produkcyjnym (z więcej niż tylko jednym programistą). Dlatego też, jeśli zamierzasz używać Pulumi, przygotuj się na konieczność wykupienia usługi Pulumi Service.

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 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 wymieniłem popularne i stosujące praktyki IaC narzędzia wraz z danymi, które zebrałem w czerwcu 2022 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 bibliotek typu open source dostępnych dla narzędzia, liczba dotyczących danego narzędzia pytań zadanych w serwisie StackOverflow⁷.

Tabela 1.1. Porównanie społeczności wybranych narzędzi IaC

	Kod źródłowy	Chmura	Liczba pracujących nad projektem	Liczba gwiazdek	Liczba bibliotek	Liczba pytań w serwisie StackOverflow
Chief	otwarty	wszystkie	640	6910	3695 ^a	8295
Puppet	otwarty	wszystkie	571	6581	6871 ^b	3996
Ansible	otwarty	wszystkie	5328	53 479	31 329 ^c	22 052
Pulumi	otwarty	wszystkie	1402	12 723	15 ^d	327
CloudFormation	zamknięty	AWS	?	?	369 ^e	7252
Heat	otwarty	wszystkie	395	379	0 ^f	103
Terraform	otwarty	wszystkie	1621	33 019	9641 ^g	13 370

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

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

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

^d Liczba pakietów udostępnionych w rejestrze Pulumi Registry (<https://www.pulumi.com/registry/>).

^e Liczba szablonów udostępnionych w koncie AWS Quick Start (<https://aws.amazon.com/quickstart>).

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

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

⁷ Większość tych danych — m.in. liczba osób pracujących nad projektem i gwiazdek — pochodzi z repozytoriów typu open source (przede wszystkim GitHub) dla danego projektu. Ponieważ CloudFormation to rozwiązanie oparte na zamkniętym kodzie źródłowym, część tych informacji jest niedostępna.

Oczywiście to nie jest doskonałe porównanie typu jeden do jednego. Przykładowo dla części narzędzi istnieje więcej niż tylko jedno repozytorium. Dla Terraform od 2017 roku istnieją oddzielne repozytoria dla kodu dostawców (np. kod dotyczący AWS, Google Cloud, Azure itd.), więc pomiar aktywności na bazie jedynie repozytorium podstawowego daje znacznie zaniżony wynik. Część narzędzi oferuje alternatywy dla serwisu StackOverflow itd.

Mając to na względzie, warto zwrócić uwagę na kilka oczywistych trendów. Po pierwsze, poza CloudFormation (to rozwiązanie zamknięte działające jedynie z AWS) wszystkie wymienione w tabeli narzędzia stosujące praktyki IaC są dostępne jako oprogramowanie typu open source i działają z wieloma dostawcami chmury. Po drugie, Ansible i Terraform prowadzą w kategorii popularności.

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. (Uwaga: narzędzia Pulumi nie ma w tabeli, ponieważ nie było uwzględnione w porównaniu zamieszczonym w pierwszym wydaniu książki).

Tabela 1.2. Zmiana wartości dotyczących społeczności wybranych narzędzi IaC dla danych zebranych między wrześniem 2016 roku i czerwcem 2022 roku

	Kod źródłowy	Chmura	Liczba pracujących nad projektem	Liczba gwiazdek	Liczba bibliotek	Liczba pytań w serwisie StackOverflow
Chief	otwarty	wszystkie	+34%	+56%	+21%	+98%
Puppet	otwarty	wszystkie	+32%	+58%	+55%	+51%
Ansible	otwarty	wszystkie	+258%	+183%	+289%	+507%
CloudFormation	zamknięty	AWS	?	?	+54% ^a	+1083%
Heat	otwarty	wszystkie	+40%	+34%	0	+98%
Terraform	otwarty	wszystkie	+148%	+476%	+24 003%	+10 106%

^a We wcześniejszych wydaniach książki użyłem szablonów CloudFormation z repozytorium GitHub awslabs, które obecnie już nie istnieje. W tym wydaniu więc skorzystałem z AWS Quick Start, dlatego tych wartości nie można bezpośrednio porównywać.

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. 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. Czy mamy do czynienia z technologią istniejącą od lat, w której wszystkie wzorce użycia, najlepsze praktyki, problemy i potencjalne sposoby awarii są doskonale znane? A może to jest zupełnie nowa technologia, której trzeba

będzie się uczyć na błędach? W tabeli 1.3 wymienilem daty wydania i obecne numery wersji poszczególnych narzędzi stosujących praktyki IaC, analizowanych w tym podrozdziale (stan na czerwiec 2022 roku), a także — być może subiektywne — odczucia dotyczące dojrzałości poszczególnych narzędzi IaC.

Tabela 1.3. Porównanie dojrzałości wybranych narzędzi IaC w czerwcu 2022 roku

	Pierwsze wydanie	Obecne wydanie	Postrzegana dojrzałość
Chef	2009	17.10.3	wysoka
Puppet	2005	7.17.0	wysoka
Ansible	2012	5.9.0	średnia
Pulumi	2017	3.34.1	niska
CloudFormation	2011	???	średnia
Heat	2012	18.0.0	niska
Terraform	2014	1.2.3	średnia

To również nie jest dokładne porównanie: sam wiek narzędzia nie ma wpływu na jego dojrzałość, podobnie jak wyższy numer wersji (poszczególne narzędzia stosują odmienne schematy wersjonowania). Mimo to trendy powinny być wyraźnie widoczne. Pulumi jest najmłodszym narzędziem IaC w tym porównaniu i bezsprzecznie najmniej dojrzałym. To staje się oczywiste podczas szukania dokumentacji, najlepszych praktyk, modułów opracowanych przez społeczność itd. Obecnie Terraform to nieco bardziej dojrzałe rozwiązanie: narzędzia zostały poprawione, najlepsze praktyki są lepiej rozumiane, dostępnych jest znacznie więcej zasobów szkoleniowych (m.in. ta książka!). Ponadto narzędzie osiągnęło już wersję 1.0.0 i jest uznawane za znacznie stabilniejsze i w większym stopniu niezawodne niż podczas pracy nad pierwszym i drugim wydaniem książki. Chef i Puppet to najstarsze i bez wątpienia najbardziej dojrzałe narzędzia w tym porównaniu.

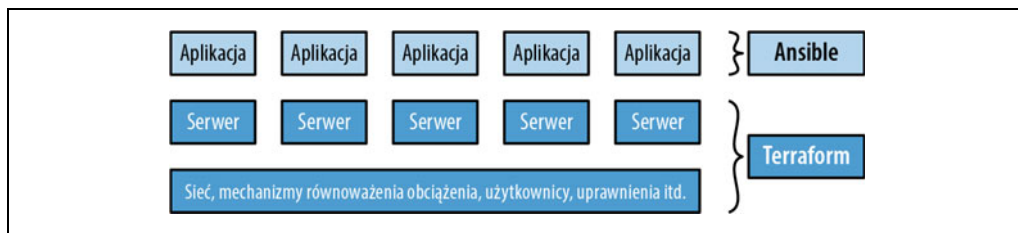
Używanie razem wielu narzędzi

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.

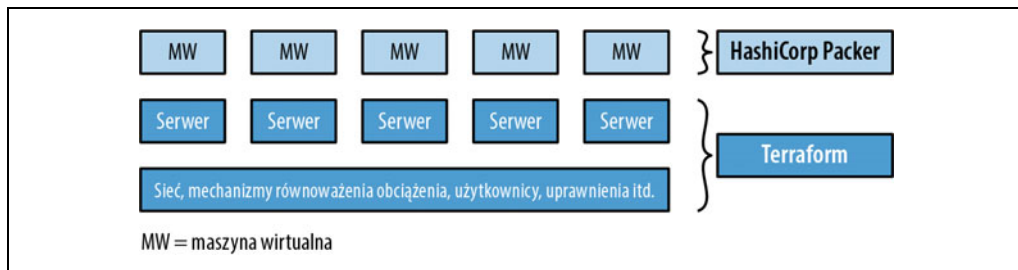


Rysunek 1.9. Terraform wdraża infrastrukturę, m.in. serwery, Ansible natomiast wdraża aplikacje w tych serwerach

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 odszukania 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 serwerów za pomocą wspomnianych obrazów maszyn wirtualnych i 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.

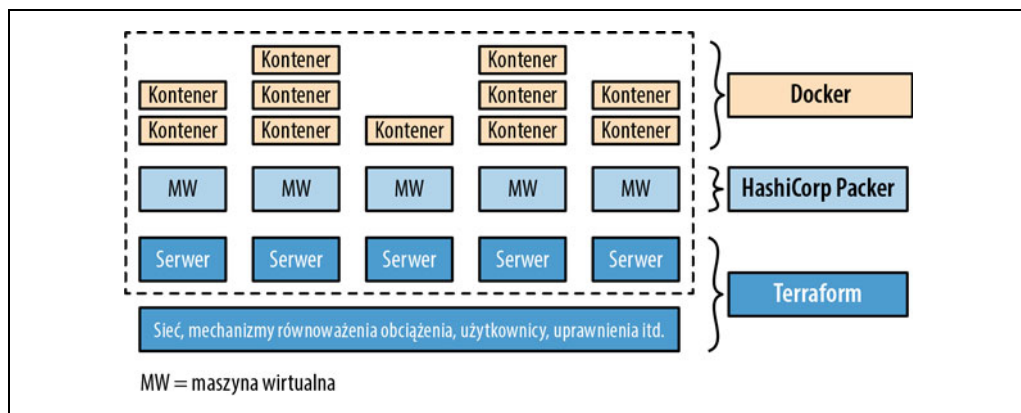


Rysunek 1.10. Terraform wdraża infrastrukturę, m.in. serwery, Packer natomiast tworzy maszyny wirtualne działające w tych serwerach

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 strategię 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 klastra serwerów, z których każdy będzie uruchamiał wspomniany obraz maszyny wirtualnej, i 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.



Rysunek 1.11. Terraform wdraża infrastrukturę, m.in. serwery, Packer natomiast tworzy maszyny wirtualne działające w tych serwerach, a Kubernetes nimi zarządza jako klastrem działających kontenerów Dockera

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 dodatkowej 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 dodatkowych warstw abstrakcji (Kubernetes, Docker, Packer) związanych z poznawaniem, zarządzaniem i debugowaniem rozwiązania.

Przykłady takiego podejścia przedstawię w rozdziale 7.

Podsumowanie

Po połączeniu wszystkiego w całość w tabeli 1.4 wymieniłem 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, Puppet do przygotowania infrastruktury niemodyfikowalnej itd.).

Tabela 1.4. Porównanie najczęściej stosowanego sposobu użycia popularnych narzędzi IaC

	Chef	Puppet	Ansible	Pulumi	Cloud Formation	Heat	Terraform
Kod źródłowy	otwarty	otwarty	otwarty	otwarty	zamknięty	otwarty	otwarty
Chmura	wszystkie	wszystkie	wszystkie	wszystkie	AWS	wszystkie	wszystkie
Typ	konfiguracja zarządzania	konfiguracja zarządzania	konfiguracja zarządzania	provisioning	provisioning	provisioning	provisioning
Infrastruktura	zmienna	zmienna	zmienna	niezmienna	niezmienna	niezmienna	niezmienna
Paradygmat	proceduralny	deklaratywny	proceduralny	deklaratywny	deklaratywny	deklaratywny	deklaratywny
Język	GPL	DSL	DSL	GPL	DSL	DSL	DSL
Serwer główny	tak	tak	nie	nie	nie	nie	nie
Agent	tak	tak	nie	nie	nie	nie	nie
Usługa płatna	opcjonalnie	opcjonalnie	opcjonalnie	tak	nie dotyczy	nie dotyczy	opcjonalnie
Spółeczność	duża	duża	ogromna	mała	mała	mała	ogromna
Dojrzałość	wysoka	wysoka	średnia	niska	średnia	niska	średnia

W firmie Gruntwork chcieliśmy zastosować rozwiązanie typu open source, niezależne od chmury narzędzie provisioningu zapewniające obsługę infrastruktury niemodyfikowalnej, dojrzałą bazę kodu, obsługę niezmienną infrastruktury, język deklaratywny, architekturę pozbawioną serwera głównego i agenta, a także — opcjonalnie — oferujące płatną usługę. 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.

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 32%, czyli znacznie więcej niż trzech kolejnych konkurentów (Microsoft, Google i IBM) razem wziętych¹.
- 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 wirtualnych².
- AWS oferuje hojnie wyposażone (<https://aws.amazon.com/free/>) konto (bezpłatne przez pierwszy rok³), 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 Źródło: *Global cloud services spend hits record US\$49.4 billion in Q3 2021*, <https://www.canalys.com/newsroom/global-cloud-services-q3-2021>, Canalys, 28 października 2021.

² Jeżeli terminologia stosowana przez AWS jest dezorientująca, zapoznaj się z jej wyjaśnieniem opublikowanym na stronie <https://expeditedsecurity.com/aws-in-plain-english/>.

³ Więcej informacji na ten temat znajdziesz w dokumentacji AWS Free Tier, <https://aws.amazon.com/free>.

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

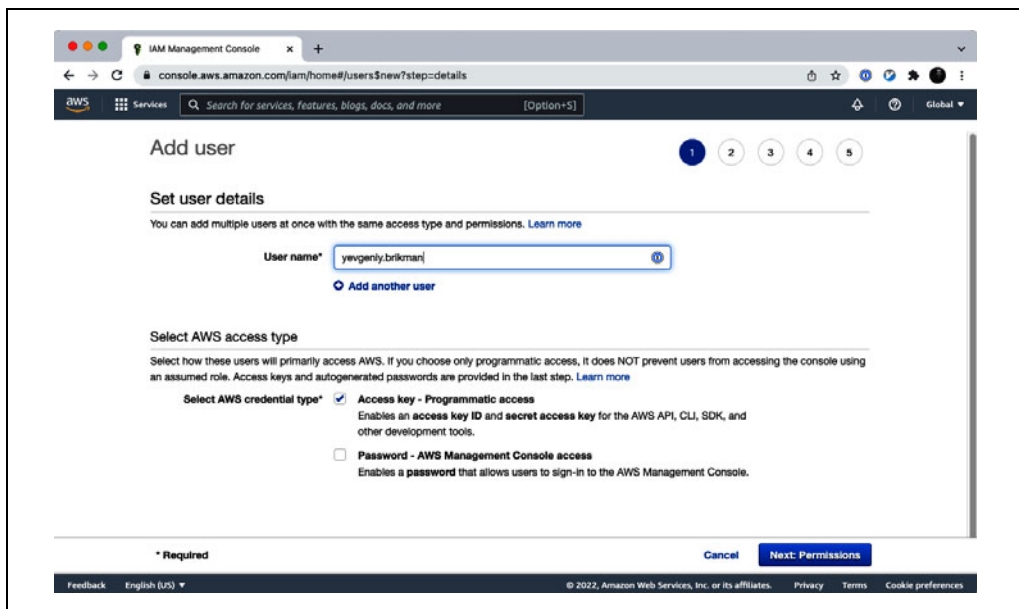
Przypominam, że wszystkie przykładowe fragmenty kodu znajdują się w repozytorium GitHub pod adresem <https://github.com/brikis98/terraform-up-and-running-code>.

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 nie wskazane. Konto użytkownika *root* powinieneś wykorzystać *tylko* do tworzenia innych kont użytkowników o znacznie bardziej ograniczonych uprawnieniach, a następnie natychmiast przejść do jednego z tych kont⁴.

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 *Add user*. Podaj nazwę użytkownika i upewnij się, że zaznaczona jest opcja *Access key — Programmatic access*, jak pokażalem 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).

⁴ 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

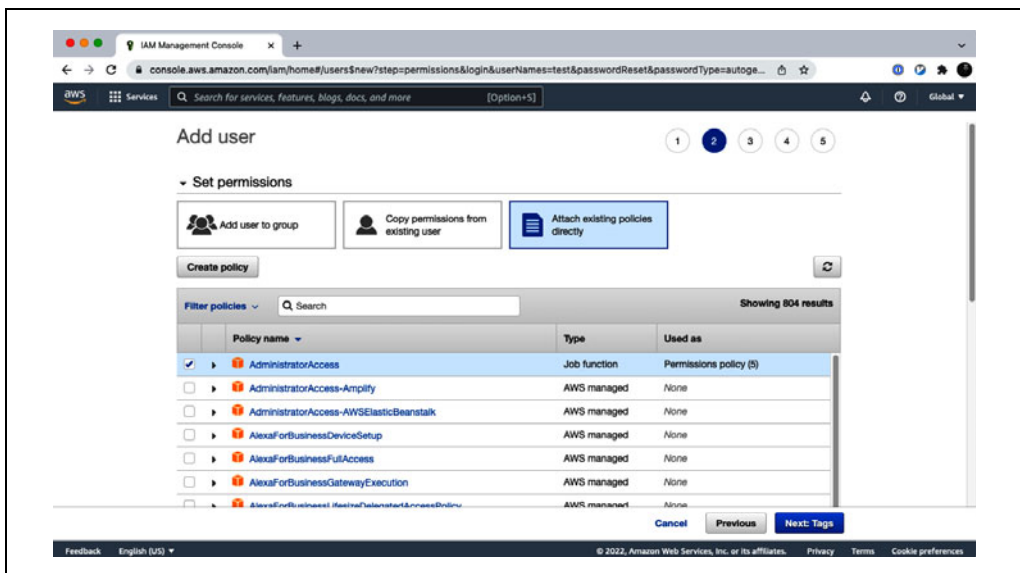
Kliknij przycisk *Next*. AWS zapyta o dodanie uprawnień dla użytkownika. Domyślnie nowy użytkownik IAM nie ma żadnych uprawnień i nie może niczego zrobić w ramach konta AWS. Aby zdefiniować uprawnienia użytkownikowi IAM do wykonania jakiegokolwiek zadania, konto trzeba powiązać ze zdefiniowaną polityką 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**⁵ (ang. *managed policies*).

By uruchomić przykłady przedstawione w książce, najłatwiej będzie dodać użytkownikowi IAM politykę zarządzaną `AdministratorAccess` (odszukaj ją i kliknij pole wyboru obok jej nazwy), jak pokazałem na rysunku 2.2⁶.

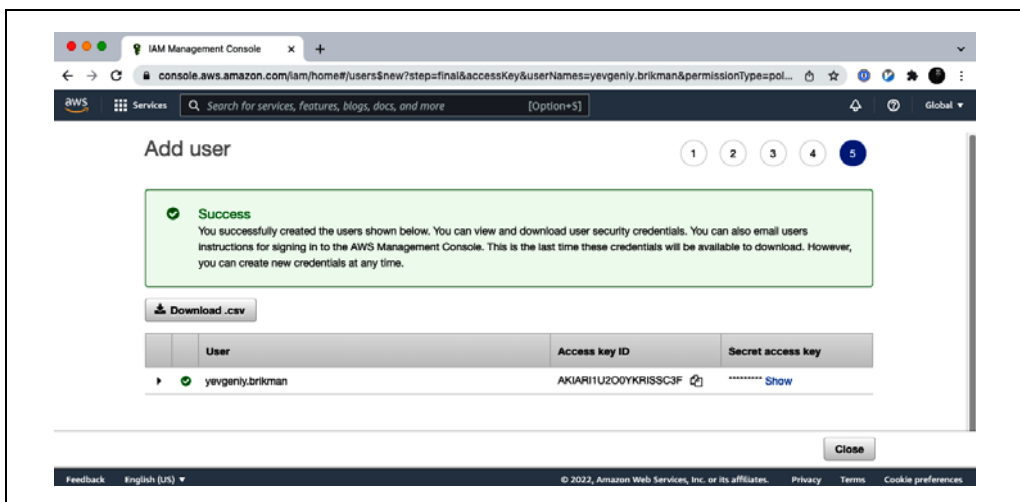
Kilkukrotnie kliknij przycisk *Next*, a następnie przycisk *Create user*. AWS wyświetli dane uwierzytelniające dla tego użytkownika składające się z wartości *Access Key ID* i *Secret Access Key*, jak możesz zobaczyć na rysunku 2.3. Musisz je natychmiast zapisać, ponieważ nigdy więcej nie zostaną

⁵ Więcej informacji na temat polityk IAM znajdziesz na stronie https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_managed-vs-inline.html.

⁶ Przyjąłem założenie, że przykłady zamieszczone w książce uruchamiasz w ramach konta AWS przeznaczonego wyłącznie do nauki i testowania. W takim wypadku szerokie uprawnienia nadawane przez politykę zarządzaną `AdministratorAccess` nie stanowią dużego ryzyka. Jeżeli jednak te przykłady wykonujesz w bardziej wrażliwym środowisku — czego absolutnie nie zalecam! — i jesteś w stanie utworzyć niestandardową politykę IAM, kod w wersji z okrojonymi uprawnieniami znajdziesz w repozytorium GitHub pod adresem <https://github.com/brikis98/terraform-up-and-running-code/blob/master/code/json/02-intro-to-terraform-syntax/iam-policy.json>.



Rysunek 2.2. Dodawanie polityki zarządzanej AdministratorAccess do nowego konta użytkownika IAM



Rysunek 2.3. Dane uwierzytelniające AWS powinieneś przechowywać w bezpiecznym miejscu. Nigdy nikomu ich nie udostępniaj (nie przejmuj się, te pokazane na rysunku zostały spreparowane)

wyświetlone, a będą potrzebne w dalszej części rozdziału. Nie zapominaj, że te dane uwierzytelniające zapewniają dostęp do konta AWS, 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 kliknij przycisk *Close*. Teraz możesz przejść do pracy z Terraform.



Uwaga dotycząca domyślnej wirtualnej chmury prywatnej

We wszystkich przykładach w książce używam *Default VPC*, czyli domyślnej wirtualnej chmury prywatnej (ang. *virtual private cloud*) konta AWS. 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 utworzonego po 2013 roku. Dlatego też, jeśli z jakiegokolwiek powodu usuwasz 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 trzeba będzie uaktualnić niemalże wszystkie przykłady, aby zawierały parametr `vpc_id` lub `subnet_id` określający niestandardową chmurę VPC.

Instalacja Terraform

Terraform najłatwiej zainstalować za pomocą domyślnego menedżera pakietów systemu operacyjnego. Jeżeli np. w systemie macOS używasz menedżera Homebrew, możesz zastosować takie polecenia:

```
$ brew tap hashicorp/tap
$ brew install hashicorp/tap/terraform
```

Jeśli natomiast w Windows używasz Chocolatey, możesz użyć tego polecenia:

```
$ choco install terraform
```

W oficjalnej dokumentacji Terraform (<https://learn.hashicorp.com/tutorials/terraform/install-cli?in=terraform/aws-get-started>) znajdziesz dokładne informacje dotyczące instalacji w innych systemach operacyjnych, m.in. w różnych dystrybucjach systemu Linux.

Ewentualnie 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 powinieneś dodać do zmiennej systemowej PATH.

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

```
$ terraform
Usage: terraform [global options] <command> [args]
```

```
The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.
```

```
Main commands:
```

<code>init</code>	Prepare your working directory for other commands
<code>validate</code>	Check whether the configuration is valid
<code>plan</code>	Show changes required by the current configuration

apply	Create or update infrastructure
destroy	Destroy previously-created infrastructure

(...)

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)
```

W wierszu poleceń Windows natomiast trzeba użyć poleceń:

```
PS C:\> set AWS_ACCESS_KEY_ID=(your access key id)
PS C:\> set AWS_SECRET_ACCESS_KEY=(your secret access key)
```

Warto w tym miejscu dodać, że te zmienne pozostaną aktywne jedynie w bieżącej powłoce. Jeżeli więc ponownie uruchomisz komputer lub przejdziesz 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 `aws 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>. Szerzej o uwierzytelnianiu dostawców Terraform piszę w rozdziale 6.

Wdrożenie pojedynczego serwera

Kod Terraform jest tworzony w języku HCL (ang. *hashicorp configuration language*) i umieszczany w plikach wraz z rozszerzeniem `.tf`. 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.

⁷ 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/language/syntax/configuration>.

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). 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.⁸ Mamy jeszcze inne ustawienia, które można skonfigurować dla dostawcy. W tym miejscu jednak chcę zachować prostotę przykładu, dokładniejsze omówienie konfiguracji dostawcy znajdziesz w rozdziale 7.

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-0fb653ca2d3203ac1"  
  instance_type = "t2.micro"  
}
```

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

⁸ 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>.

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 20.04 w regionie `us-east-2`. Ten obraz jest bezpłatny. Trzeba pamiętać, że identyfikatory obrazów AMI są różne w poszczególnych regionach AWS, jeśli więc zmienisz wartość parametru `region` na inną niż `us-east-2`, musisz samodzielnie odszukać identyfikator obrazu AMI Ubuntu dla tego regionu⁹, a następnie skopiować go do parametru `ami`. W rozdziale 7. wyjaśnię, jak można całkowicie automatycznie pobierać identyfikatory obrazów AMI.

`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.



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 stworzysz 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://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance>. 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...
```

- Reusing previous version of hashicorp/aws from the dependency lock file
- Using hashicorp/aws v4.19.0 from the shared cache directory

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

⁹ Wyszukiwanie identyfikatorów AMI okazuje się zaskakująco skomplikowane, jak to zostało wyjaśnione w poście bloga Gruntwork: <https://blog.gruntwork.io/locating-aws-ami-owner-id-and-image-name-for-packer-builds-7616fe46b49a>.

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, można go więc dodać do pliku *.gitignore*. Informacje o pobranym kodzie dostawcy Terraform zapisuje w pliku *.terraform.lock.hcl* (więcej na jego temat dowiesz się z rozdziału 8.). 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-0fb653ca2d3203ac1"
+   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 jakiejkolwiek 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-0fb653ca2d3203ac1"
  + 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 10.). 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.

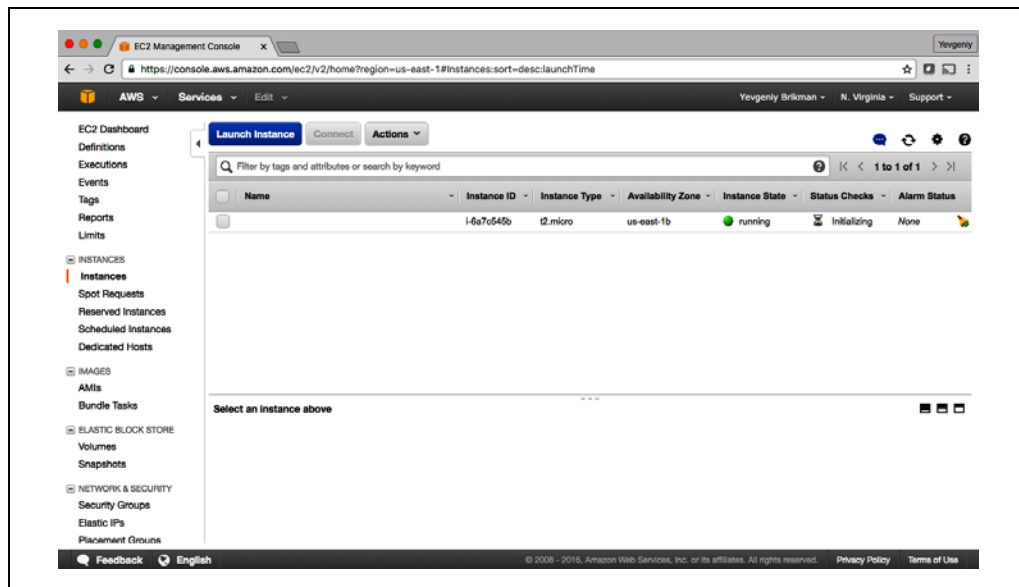
Only 'yes' will be accepted to approve.

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>) — powinny się pojawić dane podobne do pokazanych na rysunku 2.4.



Rysunek 2.4. Konsola AWS pokazuje wdrożony pojedynczy egzemplarz EC2

Oczywiście ten egzemplarz istnieje, choć trzeba przyznać, że to nie jest najbardziej ekscytujący przykład na świecie. Zrobmy 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-0fb653ca2d3203ac1"
  instance_type = "t2.micro"

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

Ponownie użyj polecenia `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-0fb653ca2d3203ac1"
  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?

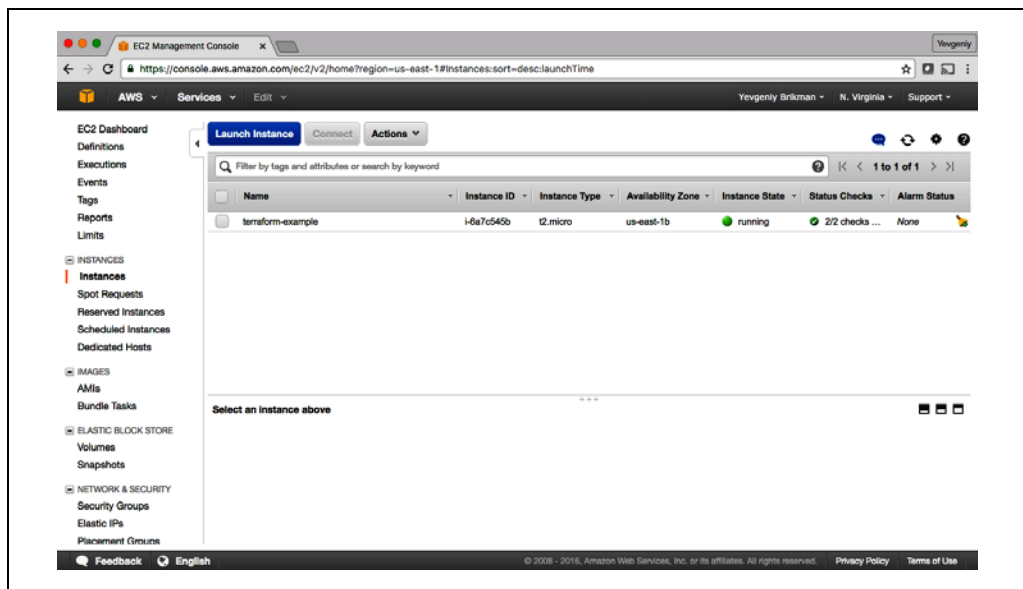
Terraform will perform the actions described above.

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 masz już utworzony pewien kod Terraform, być może chcesz 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 i pliku blokady. (Więcej informacji na temat tego pliku znajdziesz w rozdziale 8. Teraz wystarczy wiedzieć, że te pliki powinny być dodane do systemu kontroli wersji razem z kodem Terraform):

```
$ git init
$ git add main.tf .terraform.lock.hcl
$ git commit -m "Pierwsze przekazanie plików 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 main
```

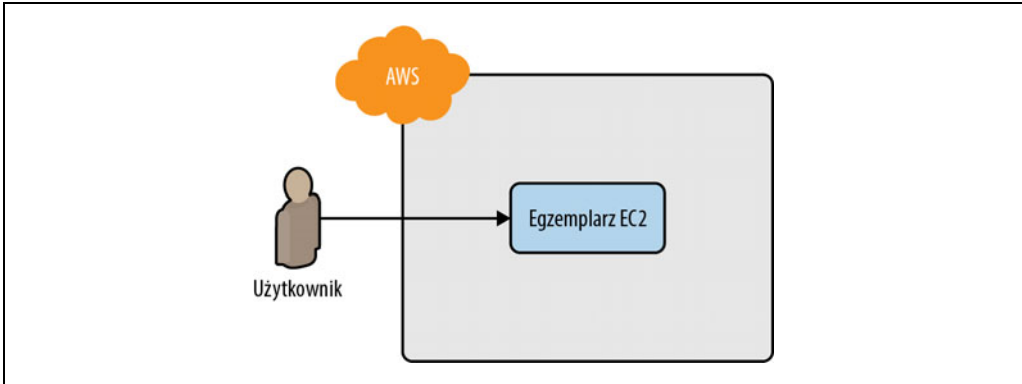
Natomiast w celu zobaczenia zmian wprowadzonych przez współpracowników musisz *pobrać* zmiany ze zdalnego repozytorium za pomocą następującego polecenia:

```
$ git pull origin main
```

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 10.

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*¹⁰.

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



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 przekazywać ruch sieciowy do portów o wyższych numerach w serwerze WWW.

¹⁰ Użyteczną listę serwerów HTTP tworzonych za pomocą tylko jednego wiersza kodu znajdziesz na stronie <https://gist.github.com/willurd/5720255>.

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.

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 20.04 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-0fb653ca2d3203ac1"
  instance_type = "t2.micro"

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

  user_data_replace_on_change = true

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

Dwie ważne kwestie wiążą się z przedstawionym fragmentem kodu.

- Znaczniki *<<-EOF* i *EOF* to składnia typu *heredoc* w Terraform pozwalająca na tworzenie wielowierszowych ciągów tekstowych bez konieczności wstawiania znaków nowego wiersza (*\n*).
- Parametr *user_data_replace_on_change* ma przypisaną wartość *true*, więc po zmianie parametru *user_data* i użyciu polecenia *terraform apply* Terraform zakończy działanie egzemplarza początkowego i uruchomi zupełnie nowy. Zachowanie domyślne Terraform polega na uaktualnieniu egzemplarza początkowego. Skoro jednak konfiguracja danych użytkownika odbywa się tylko w trakcie pierwszego uruchomienia, a egzemplarz początkowy już przeszedł przez proces uruchamiania, to konieczne jest wymuszenie utworzenia nowego egzemplarza, aby w ten sposób mieć pewność, że zostanie wykonany skrypt *user_data*.

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 IP¹¹.

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-0fb653ca2d3203ac1"`) 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>
```

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 argumencie `vpc_security_group_ids` zasobu `aws_instance`.

¹¹ 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`.


```

resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  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

  user_data_replace_on_change = true

  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`.

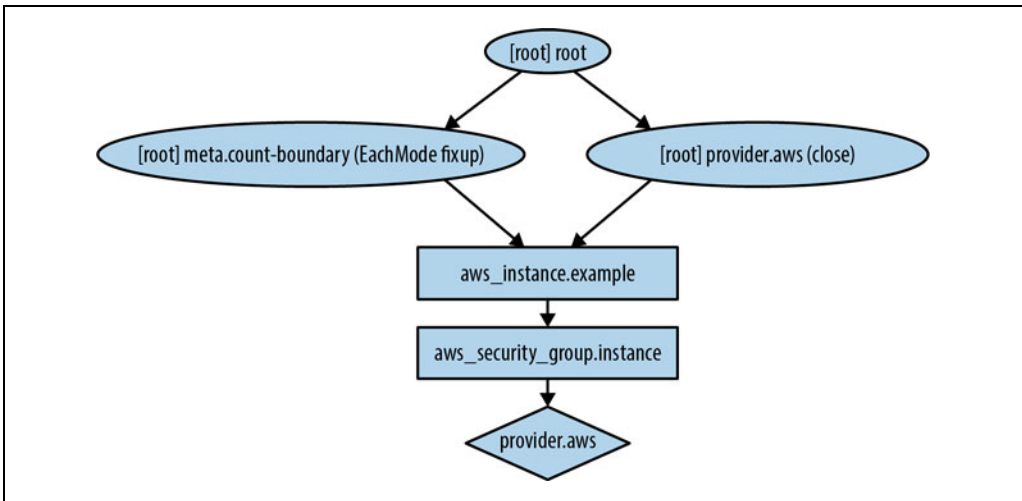
\$ terraform graph

```

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/>)¹².



Rysunek 2.7. Wykres zależności dla egzemplarza EC2 i jego grupy bezpieczeństwa wygenerowany przez narzędzie Graphviz

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-0fb653ca2d3203ac1"
  ~ 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 = [
```

¹² Wprawdzie polecenie `graph` może być użyteczne podczas wizualizacji związku między małą liczbą zasobów, ale w wypadku dziesiątek lub setek zasobów wykres staje się zbyt duży i zbyt zagmatwany, aby był użyteczny.

```

    - "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?

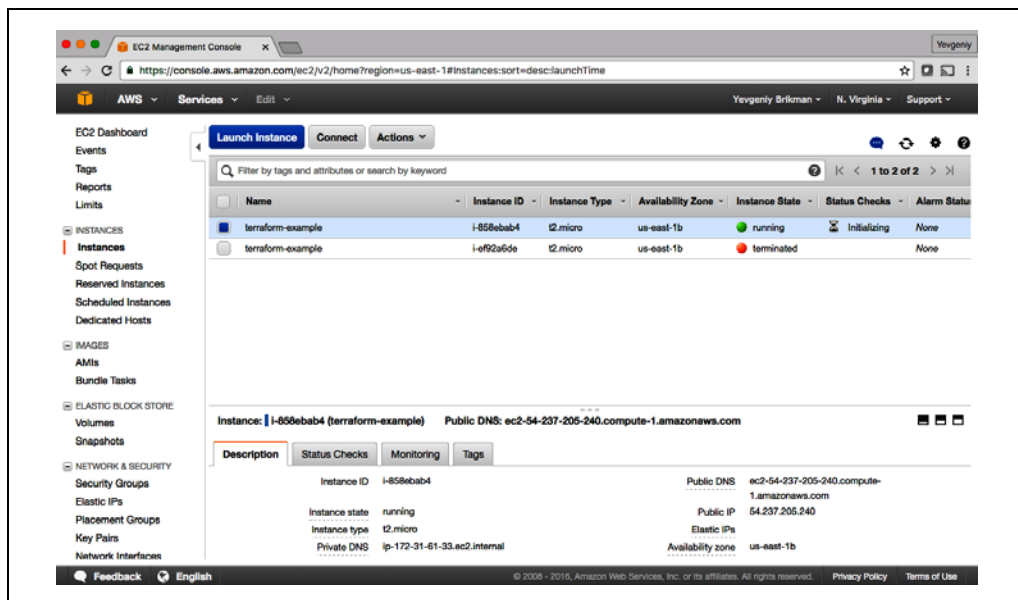
Terraform will perform the actions described above.

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. Z powodu użycia opcji `user_data_replace_on_change` i zmiany parametru `user_data` to 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).



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

Bezpieczeństwo sieci

W celu zachowania prostoty wszystkich przykładów przedstawionych w książce są one wdrażane nie tylko w domyślnej prywatnej wirtualnej chmurze (o czym wspomniałem już wcześniej), ale również w domyślnych **podsięciach** danej sieci VPC. Standardowo sieć VPC jest podzielona na jedną lub więcej podsięci, z których każda ma własny adres IP. Podsieci w sieci domyślnej VPC są **podsięciami 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 ogniw. 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 **podsięciach prywatnych**, których adresy IP są dostępne jedynie wewnątrz danego VPC, a nie z zewnątrz (np. z publicznego internetu). Jedynie serwery, które powinny działać w podsięciach 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).

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
```

Świetnie! Masz uruchomiony serwer WWW w chmurze AWS.

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 kategoryczną reprezentację w systemie¹³. 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 wymienione tutaj 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 powłoce (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`. Zdefiniowanie ograniczenia typu zawsze będzie dobrym

¹³ Zob. Andy Hunt, Dave Thomas, *Pragmatyczny programista. Od czeladnika do mistrza*, Helion, Gliwice, 2011.

pomysłem, ponieważ pomaga to w wychwytywaniu prostych błędów. Jeżeli nie podasz typu, przyjęte zostanie założenie, że jest nim any.

validation

To pozwala na zdefiniowanie niestandardowych reguł weryfikacji zmiennej danych wejściowych, wykraczających poza sprawdzenie zaledwie typu, np. możliwe jest wymuszenie użycia minimalnej lub maksymalnej wartości liczbowej. Przykłady tego rodzaju weryfikacji przedstawię w rozdziale 8.

sensitive

Jeżeli temu parametrowi przypiszesz wartość true dla zmiennej danych wejściowych, Terraform nie będzie go zapisywał podczas wykonywania polecenia terraform plan lub terraform apply. Tego parametru należy używać podczas przekazywania za pomocą zmiennych jakichkolwiek danych poufnych do kodu Terraform, np. haseł, kluczy API itd. Do tematu danych poufnych powrócę w rozdziale 6.

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

```
variable "number_example" {
  description = "Przykład zmiennej typu number w Terraform"
  type       = number
  default    = 42
}
```

To natomiast przykład zmiennej sprawdzającej, czy wartość jest listą:

```
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 plan` lub `terraform apply`. To jest użyteczna możliwość, gdy zmienna danych wyjściowych zawiera informacje wrażliwe, takie jak hasło lub klucz prywatny. Jeżeli zmienna danych wyjściowych odwołuje się do zmiennej danych wejściowych lub atrybutu oznaczonego za pomocą `sensitive = true`, *wymagane* jest oznaczenie zmiennej danych wyjściowych za pomocą `sensitive = true`, aby w ten sposób wskazać na celowe wyświetlenie danych poufnych.

`depends_on`

Standardowo Terraform automatycznie określa wykres zależności na podstawie odniesień w kodzie. Jednak w rzadkich sytuacjach trzeba będzie użyć pewnych podpowiedzi. Przykładowo masz zmienną danych wyjściowych zwracającą adres IP serwera, przy czym adres IP będzie niedostępny aż do chwili prawidłowego skonfigurowania dla tego serwera grupy bezpieczeństwa (zapory sieciowej). W takim wypadku za pomocą `depends_on` można wyraźnie wskazać Terraform istnienie zależności między zmienną danych wyjściowych zawierającą adres IP i zasobem grupy bezpieczeństwa.

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 za pomocą szybkiego testu potwierdzić poprawność wdrożenia.

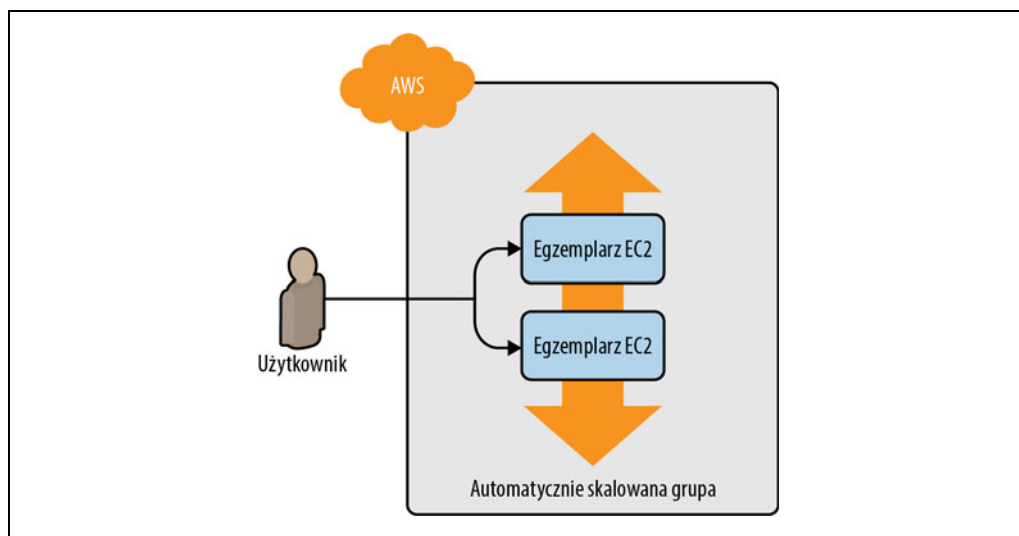
Zmienne danych wejściowych i wyjściowych mają również znaczenie podczas tworzenia konfigurowalnej infrastruktury wielokrotnego użycia jako kodu; więcej informacji na ten temat znajdziesz w rozdziale 4.

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 sieciowego¹⁴.

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 klastry serwerów WWW za pomocą automatycznie skalowanej grupy

Pierwszym krokiem podczas tworzenia ASG jest przygotowanie **konfiguracji startowej** określającej sposób skonfigurowania poszczególnych egzemplarzy EC2 w ASG¹⁵. Zasób `aws_launch_configuration` używa praktycznie tych samych parametrów znanych z zasobu `aws_instance`, choć nie obsługuje tagów (trzeba będzie się tym później zająć w zasobie `aws_autoscaling_group`), parametru `user_data_replace_on_change` (ASG domyślnie uruchamia nowy egzemplarz, ten parametr zatem jest niepotrzebny), a dwa z tych parametrów mają inne nazwy — zamiast `ami` mamy `image_id` i zamiast `vpc_security_group` mamy `security_groups` — więc wystarczy zamienić poprzednią nazwę parametru na nową:

¹⁴ Szczegółowe informacje na temat tworzenia wysoko dostępnych i skalowanych systemów w AWS znajdziesz w artykule Josha Padnicka *A Comprehensive Guide to Building a Scalable Web App on Amazon Web Services — Part 1*, opublikowanym na stronie <https://www.airpair.com/aws/posts/building-a-scalable-web-app-on-amazon-web-services-p1>.

¹⁵ Obecnie w automatycznie skalowanej grupie należy korzystać z tzw. *szablону startowego* (i zasobu `aws_launch_template_resource`), a nie z konfiguracji startowej. Jednak w przykładach zamieszczonych w książce pozostanę przy konfiguracji startowej, ponieważ takie podejście okazuje się użyteczne podczas omawiania wybranych koncepcji wdrożeń bez przestoju (zob. rozdział 5.).

```

resource "aws_launch_configuration" "example" {
  image_id      = "ami-0fb653ca2d3203ac1"
  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-0fb653ca2d3203ac1"
  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.
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 ramce „Bezpieczeństwo sieci”). Każda podsieć istnieje w odizolowanej strefie dostępności AWS (tzn. odizolowanym centrum danych), więc wdrożenie egzemplarzy w wielu podsieciach 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 źródło 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_subnets`, i wyszukania podsieci w danym VPC:

```
data "aws_subnets" "default" {
  filter {
    name     = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}
```

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

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_id          = data.aws_subnets.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.

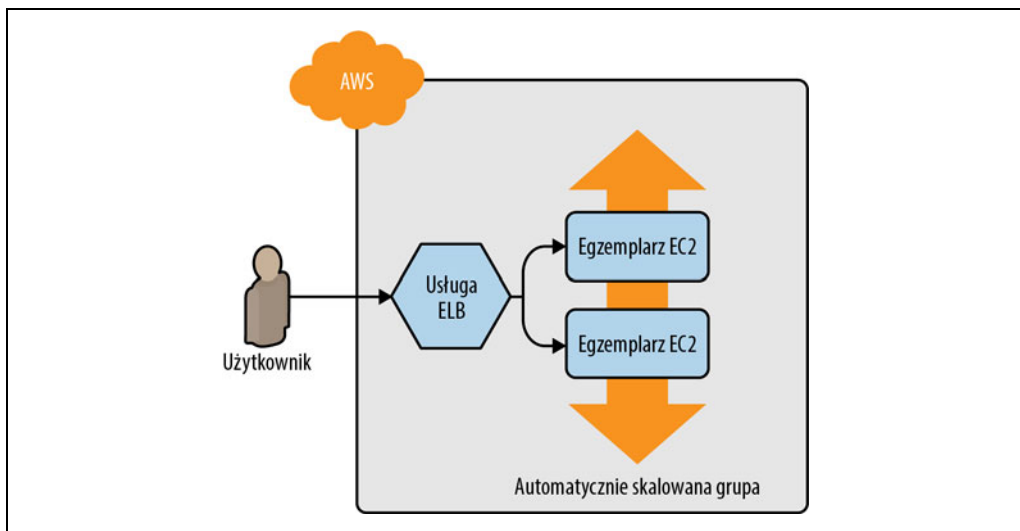
Dostawca AWS oferuje trzy odmienne rodzaje mechanizmu równoważenia obciążenia:

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.

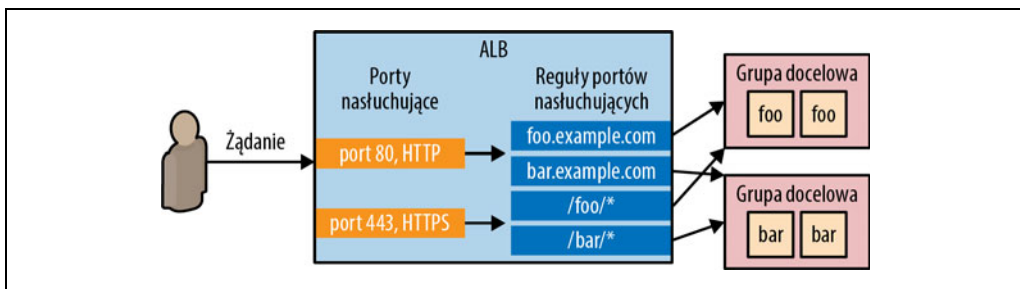
CLB (ang. *classic load balancer*)

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.

Porty nasłuchujące

Reguły portów nasłuchujących

Grupa docelowa



Rysunek 2.11. Mechanizm równoważenia obciążenia typu ALB składa się z portów nasłuchujących, reguł portów nasłuchujących i grup docelowych

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_subnets.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_subnets`¹⁶. 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
    }
  }
}
```

¹⁶ 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).

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.

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_subnets.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_subnets.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 {
    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.

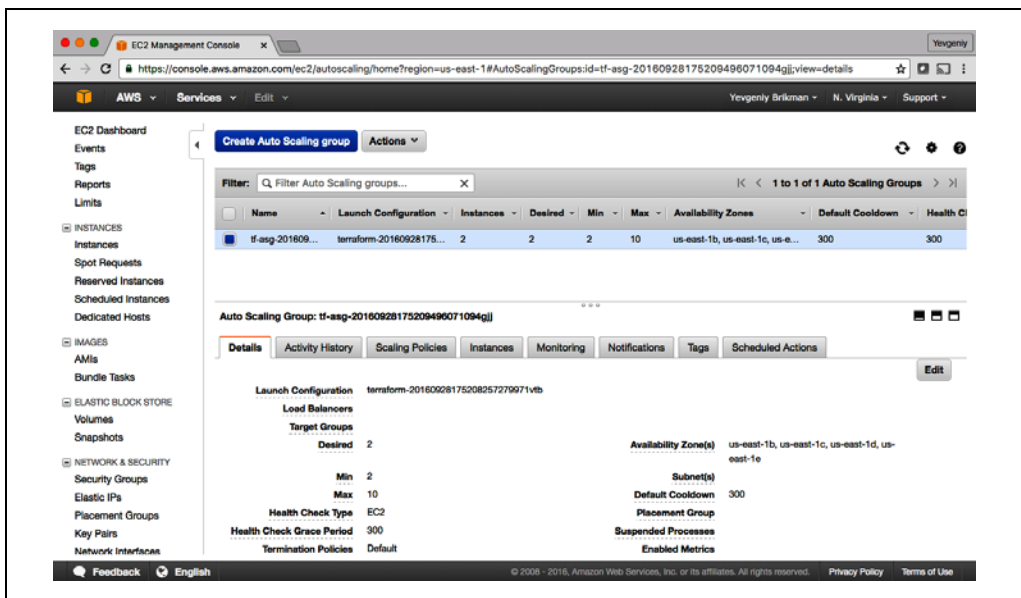
```
output "alb_dns_name" {
  value       = aws_lb.example.dns_name
  description = "Nazwa domeny mechanizmu równoważenia obciążenia"
}
```

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`:

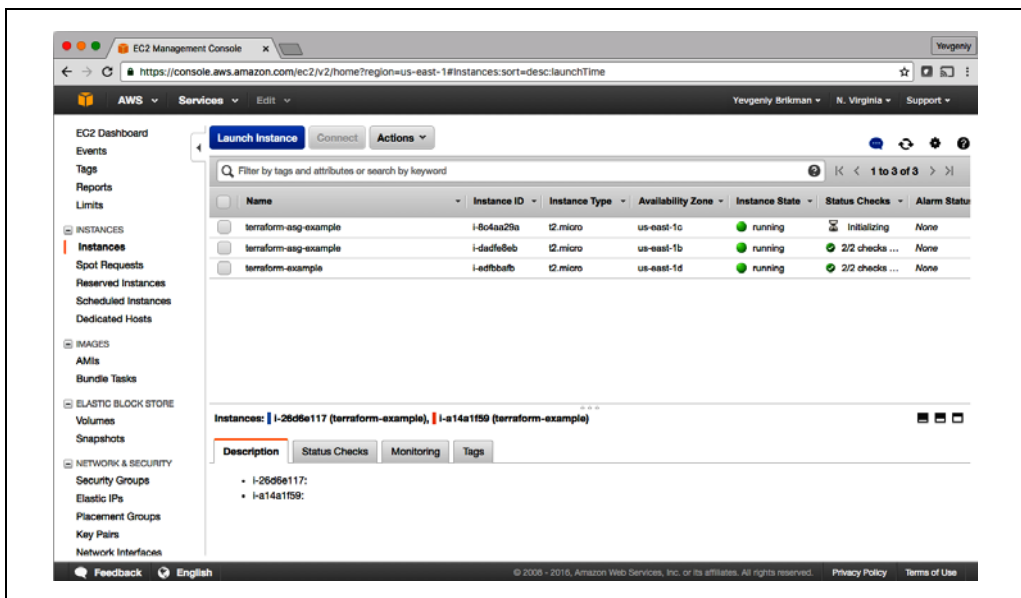
```
Outputs:
alb_dns_name = terraform-asg-example-123.us-east-2.elb.amazonaws.com
```

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.

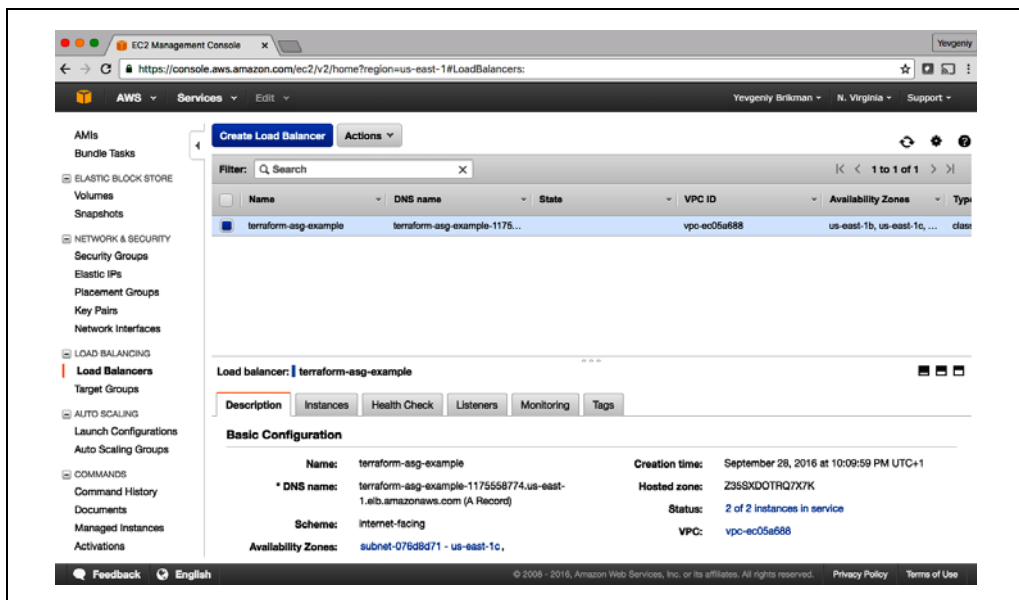


Rysunek 2.12. Grupa ASG w konsoli AWS EC2



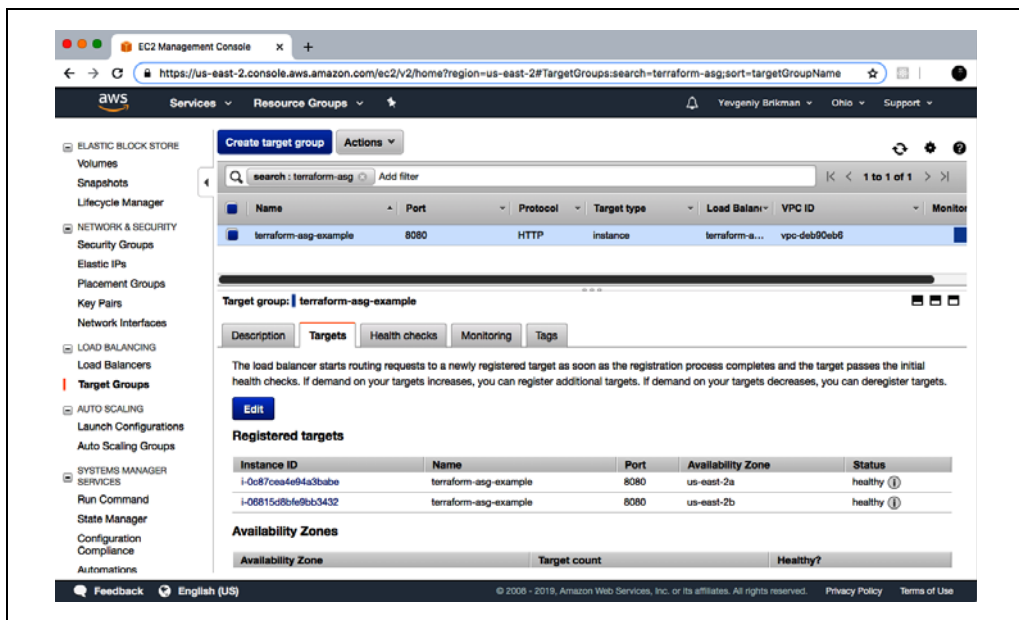
Rysunek 2.13. Uruchamianie egzemplarzy EC2 w grupie ASG

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



Rysunek 2.14. Mechanizm równoważenia obciążenia w przykładowej aplikacji

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



Rysunek 2.15. Informacje o grupie docelowej

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. Poczekaj do chwili, 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
```

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.

Na tym etapie można zobaczyć, jak klaster 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. Co ciekawe, krótko po zatrzymaniu egzemplarza grupa ASG automatycznie wykrywa dostępność mniej niż dwóch uruchomionych egzemplarzy i uruchamia nowy, aby zastąpić zatrzymany (samodzielna naprawa). Zmianę wielkości grupy ASG możesz obserwować przez dodanie parametru `desired_capacity` do kodu Terraform i ponowne użycie 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:

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

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

# Zasob 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 powinniśmy korzystać rzadko (o ile w ogóle) w środowisku produkcyjnym. Nie ma czegoś takiego jak cofnięcie wymienionego polecenia, więc Terraform daje ostatnią możliwość na przejrzanie 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 załazek 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-0fb653ca2d3203ac1"
  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": "1.2.3",
  "serial": 1,
  "lineage": "86545604-7463-4aa5-e9e8-a2a221de98d2",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "example",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0fb653ca2d3203ac1",
            "availability_zone": "us-east-2c",
            "id": "i-0bc4bbe5b84387543",
            "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-0bc4bbe5b84387543`. 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.



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 powinienś 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.).

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:

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.

Isolowanie 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). Wprawdzie kod Terraform zdecydowanie powinienś 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 nowych 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 w systemie kontroli wersji plików zawierających takie dane jest złym pomysłem.

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 i Terraform Enterprise.

Zdalny backend pozwala na rozwiązywanie 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 zachowujesz 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 99,99999999% 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. kubelka 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 kubelek 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"  
}
```

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/>.

Teraz utwórz kubełek S3 za pomocą zasobu `aws_s3_bucket`.

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

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

Ten fragment kodu definiuje takie argumenty:

`bucket`

To jest nazwa kubełka S3. Zwróć uwagę na to, że nazwy kubełków 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 inną. 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 kubełek S3, przechowującego wszystkie informacje o stanie Terraform. Oczywiście, jeśli faktycznie będziesz chciał usunąć dany zasób, wystarczy, że umieścisz znak komentarza na początku omawianego ustawienia.

Dodajemy kolejne warstwy zabezpieczeń dla tworzonego kubełka S3.

Po pierwsze, używamy zasobu `aws_s3_bucket_versioning` pozwalającego na włączenie wersjonowania w kubełku S3, aby każde uaktualnienie pliku w kubełku powodowało utworzenie nowej wersji tego pliku. To pozwala na sprawdzanie starszych wersji pliku i przywracanie ich w dowolnym momencie, co będzie użytecznym mechanizmem awaryjnym, na wypadek gdyby coś poszło źle.

```
# Włączenie wersjonowania, co pozwala na zachowanie pełnej
# historii informacji o stanie.
resource "aws_s3_bucket_versioning" "enabled" {
  bucket = aws_s3_bucket.terraform_state.id
  versioning_configuration {
    status = "Enabled"
  }
}
```

Po drugie, używamy zasobu `aws_s3_bucket_server_side_encryption_configuration`, który włącza szyfrowanie po stronie serwera domyślnie dla wszystkich danych zapisywanych we wskazanym kubełku S3. Dzięki temu masz pewność, że pliki informacji o stanie i wszelkie znajdujące się w nich dane wrażliwe zawsze będą zaszyfrowane na dysku podczas ich przechowywania w S3.

```
# Domyślne włączenie szyfrowania po stronie serwera.
resource "aws_s3_bucket_server_side_encryption_configuration" "default" {
  bucket = aws_s3_bucket.terraform_state.id

  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}
```

Po trzecie, używamy zasobu `aws_s3_bucket_public_access_block` w celu uniemożliwienia całego publicznego dostępu do kubełka S3. Wprowadzie kubełek S3 domyślnie jest prywatny, ale bardzo często służy do udostępniania treści statycznej — np. obrazów, czcionek, CSS, JS, HTML — co powoduje, że można dość łatwo udostępnić go publicznie. Skoro pliki informacji o stanie Terraform mogą zawierać dane wrażliwe i poufne, warto dodać kolejną warstwę zabezpieczenia i mieć pewność, że nikt z zespołu przypadkowo nie udostępni publicznie kubełka S3.

```
# Wyraźne zablokowanie dostępu publicznego do kubełka S3.
resource "aws_s3_bucket_public_access_block" "public_access" {
  bucket = aws_s3_bucket.terraform_state.id
  block_public_acls = true
  block_public_policy = true
  ignore_public_acls = true
  restrict_public_buckets = true
}
```

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 Terraform³.

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"
  }
}
```

Wydać polecenie `terraform init` w celu pobrania kodu dostawcy, a następnie `terraform apply` w celu wdrożenia projektu. Wprowadzie po przeprowadzeniu wdrożenia otrzymasz kubełek S3 i tabelę DynamoDB, ale informacje o stanie Terraform nadal będą przechowywane lokalnie. W celu

³ Cennik usługi DynamoDB znajduje się na stronie <https://aws.amazon.com/dynamodb/pricing/>.

skonfigurowania Terraform do umieszczenia w kubelku 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 kubelka S3). Spójrz na przykładową konfigurację backendu dla kubelka S3.

```
terraform {  
  backend "s3" {  
    # Zmień tę nazwę na nazwę Twojego kubelka!  
    bucket      = "terraform-up-and-running-state"  
    key         = "global/s3/terraform.tfstate"  
    region      = "us-east-2"  
  
    # Zmień tę nazwę na nazwę Twojej tabeli DynamoDB!  
    dynamodb_table = "terraform-up-and-running-locks"  
    encrypt        = true  
  }  
}
```

Przeanalizujemy poszczególne ustawienia:

bucket

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

key

To jest ścieżka dostępu do pliku kubelka 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 kubelek. Upewnij się o umieszczeniu w tym miejscu wartości przedstawiającej region, w którym wcześniej utworzyłeś kubelek.

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 kubelka S3, więc to jest druga warstwa zapewniająca, że dane zawsze pozostaną zaszyfrowane.

Aby nakazać Terraform przechowywanie w kubelku 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
```

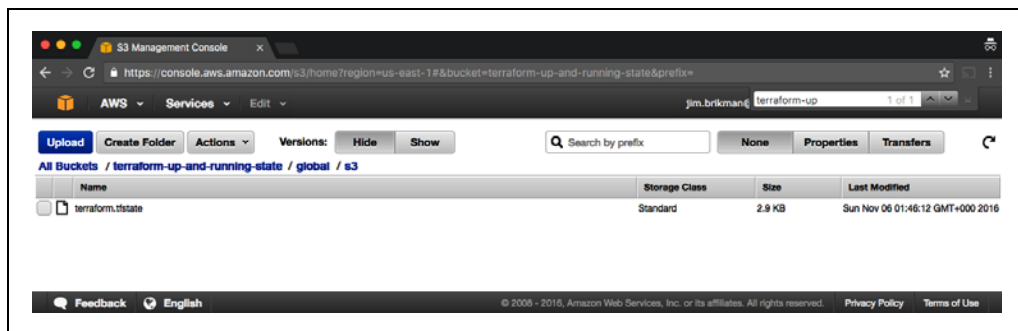
```
Initializing the backend...
Acquiring state lock. This may take a few moments...
Do you want to copy existing state to the new backend?
Pre-existing state was found while migrating the previous "local" backend
to the newly configured "s3" backend. No existing state was found in the
newly configured "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 kubelka 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 kubelka. Otrzymasz wynik podobny do pokazanego na rysunku 3.1.



Rysunek 3.1. Informacje o stanie Terraform są przechowywane w kubelku S3 i można je wyświetlić za pomocą konsoli AWS

Po włączeniu tego backendu, przed wykonaniem polecenia Terraform automatycznie pobierze z utworzonego kubelka S3 najnowsze informacje o stanie, natomiast po wykonaniu polecenia umieści w kubelku S3 najnowsze informacje o stanie Terraform. Jeżeli chcesz 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 nazwę tabeli DynamoDB. Wyдай 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:

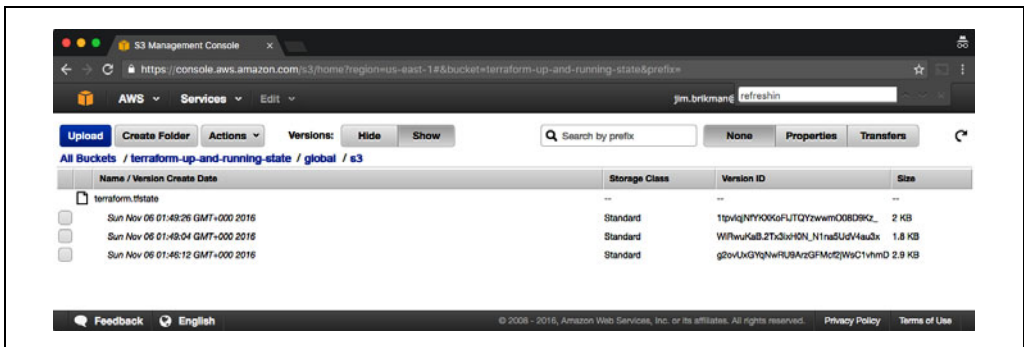
```

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 kubku S3 jest kilka wersji pliku *terraform.tfstate*, jak pokazałem na rysunku 3.2.



Rysunek 3.2. Jeżeli włączysz wersjonowanie dla kubka S3, każda zmiana pliku stanu będzie przechowywana w postaci oddzielnej wersji

Masz potwierdzenie, że Terraform automatycznie pobiera i przekazuje dane do kubka S3. Kubelek 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 kubełka 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 kubełka 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 kubełka S3 i tabeli DynamoDB, a następnie wydaj polecenie `terraform init`, aby przechowywane lokalnie informacje o stanie skopiować do S3.

Jeżeli kiedykolwiek będziesz chciał usunąć kubełek 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 kubełka S3 i tabeli DynamoDB.

Ten dwuetapowy proces jest nieco dziwny, ale dobrą wiadomością jest możliwość współdzielenia pojedynczego kubełka 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 kubełek 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 kubełka S3, regionu, tabeli DynamoDB itd. do każdego modułu Terraform. (Więcej informacji na temat modułów Terraform znajdziesz w rozdziałach 4. i 8., 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.

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. Tego samego pliku *backend.hcl* można użyć we wszystkich modułach, co znacząco zmniejsza ilość powielonego kodu. Mimo wszystko nadal trzeba definiować unikatową wartość `key` dla każdego modułu.

Inną możliwością jest wykorzystanie Terragrunt (<https://terragrunt.gruntwork.io/>), 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 10.

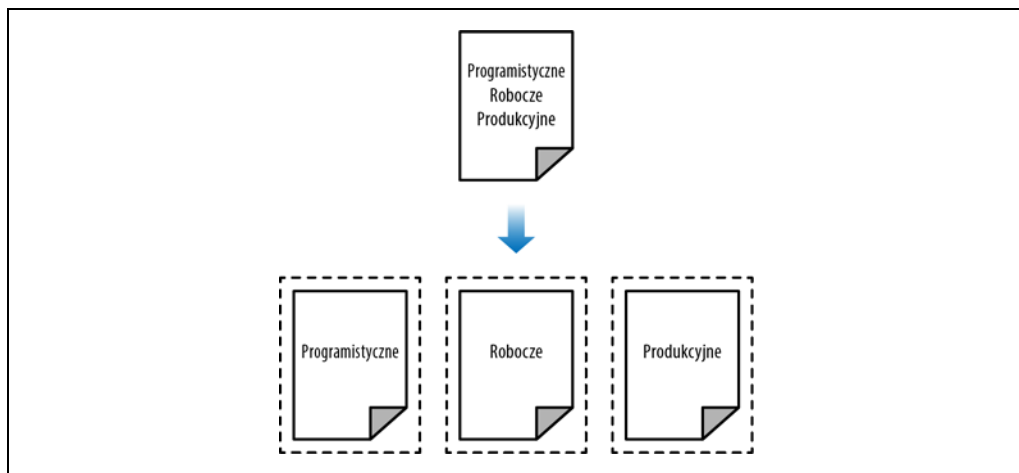
Izolowanie plików informacji o stanie

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 kruszące 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 uszkodzona⁴.

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. Zapewnienie izolacji (tzw. grodzi) między środowiskami poprzez ich zdefiniowanie w oddzielnych konfiguracjach 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.

⁴ 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.

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.

Izolacja 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-0fb653ca2d3203ac1"
  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
  }
}
```

Wyдай 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
```

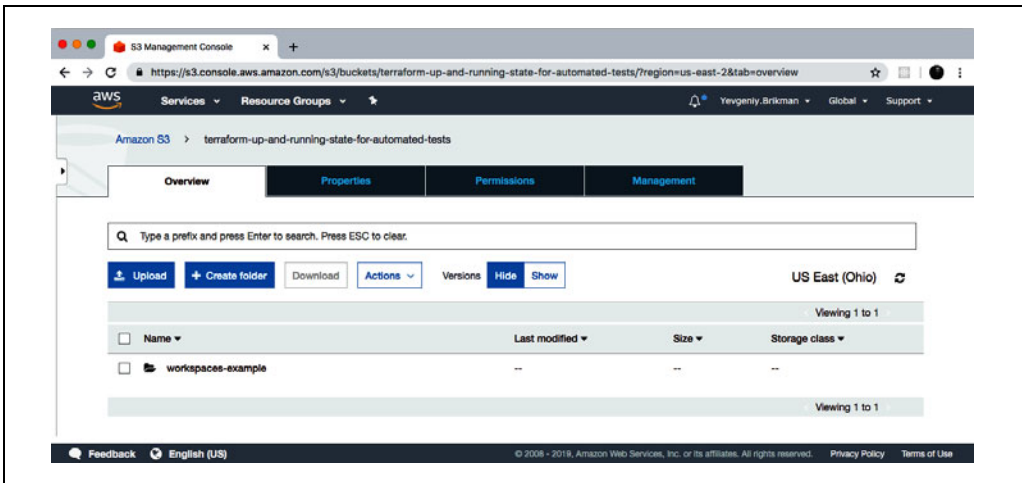
```
(...)
```

```
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. W wypadku użycia domyślnej przestrzeni roboczej kubelki S3 to po prostu pojedynczy katalog zawierający plik z informacjami o stanie

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-0fb653ca2d3203ac1"
+   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`.

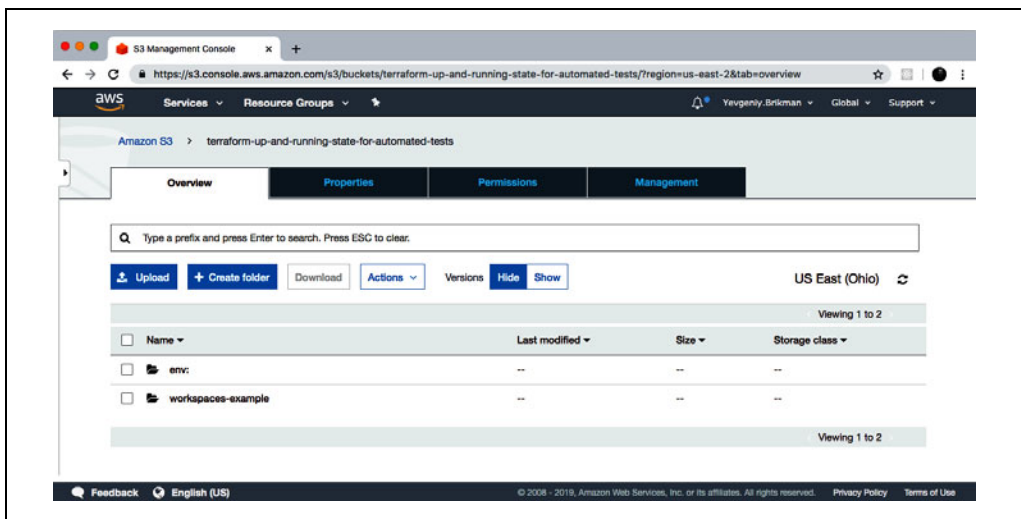
```
$ terraform workspace select example1
```

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

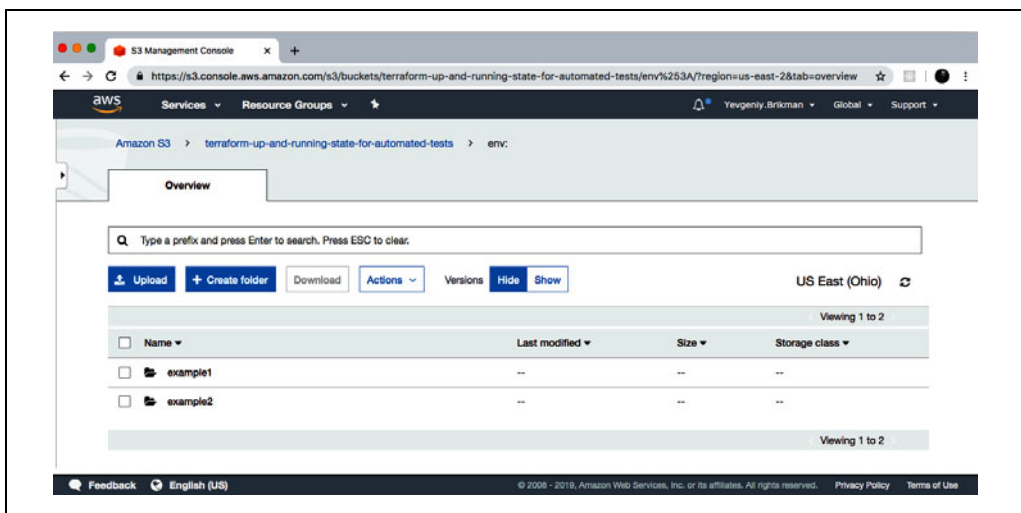
Aby dokładnie zrozumieć sposób działania przestrzeni roboczej, ponownie spójrz na kubełek S3. Powinieneś zobaczyć nowy katalog o nazwie `env`, jak pokazałem na rysunku 3.5.

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. Po rozpoczęciu korzystania z przestrzeni roboczych kubelek S3 będzie zawierał wiele katalogów i umieszczonych w nich plików z informacjami o stanie



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-0fb653ca2d3203ac1"
  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.

Ze względu na wymienione wady przestrzeń robocza nie jest odpowiednim mechanizmem podczas izolowania poszczególnych środowisk, np. roboczego i produkcyjnego⁵. 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.

⁵ Dokumentacja przestrzeni roboczej (<https://www.terraform.io/language/state/workspaces>) to potwierdza, przy czym te informacje znajdują się gdzieś głęboko w wielu akapitach tekstu. Ponadto, ponieważ w tej dokumentacji przestrzeń robocza jest nazywana „środowiskiem”, wielu użytkowników pozostaje zdezorientowanych i nie wie, kiedy używać, a kiedy nie używać przestrzeni roboczej.

Isolacja 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 błahego 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).



Rysunek 3.7. Typowy układ plików w projekcie Terraform korzysta z oddzielnych katalogów dla poszczególnych środowisk i komponentów w tych środowiskach

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.

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 wejściowych lub zasobu.

Pamiętaj, że to jest *minimalna* konwencja, do której należy się stosować, ponieważ praktycznie zawsze podczas pracy z Terraform dobrze jest mieć możliwość bardzo szybkiego przejścia do zmiennych danych wejściowych, zmiennych danych wyjściowych i zasobów. Oczywiście można wykroczyć poza tę konwencję. Zapoznaj się z kilkoma przykładami:

dependencies.tf

Przyjęło się umieszczać wszystkie źródła danych w pliku *dependencies.tf*, aby łatwiej dostrzec zewnętrzne zależności kodu.

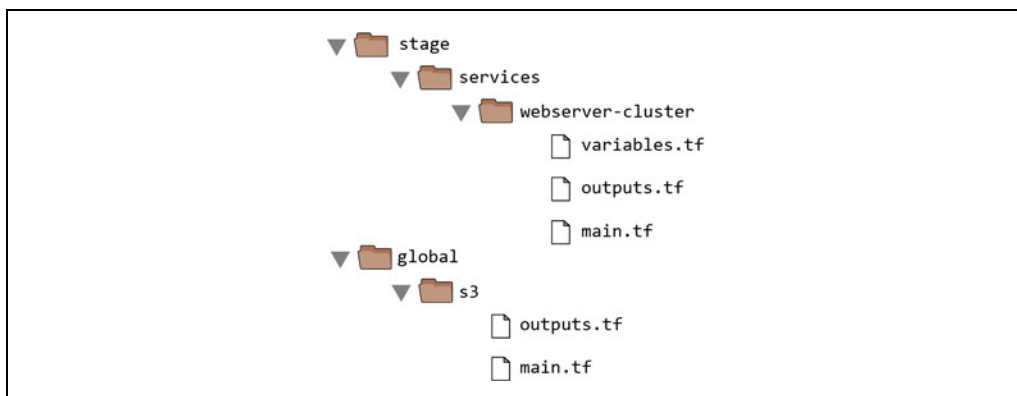
providers.tf

Bloki provider można umieścić w pliku *providers.tf*, co pozwala łatwo sprawdzić, których dostawców używa kod i jakie metody uwierzytelniania trzeba dostarczyć.

main-xx.tf

Jeżeli plik Terraform jest ogromny, ponieważ zawiera znaczną liczbę zasobów, można go podzielić i poszczególne funkcjonalności umieścić w oddzielnych logicznych plikach (np. *main-iam.tf* może zawierać wszystkie zasoby IAM, *main-s3.tf* może zawierać wszystkie zasoby S3 itd.). Użycie prefiksu *main-* ułatwia przeglądanie listy plików w katalogu, a gdy jest ona ułożona alfabetycznie, wtedy wszystkie zasoby pozostają zgrupowane razem. Warto w tym miejscu dodać, że jeśli zmagasz się z zarządzaniem dużą ilością zasobów i ich podziałem na wiele mniejszych plików, to może być również sygnał wskazujący na potrzebę podzielenia kodu na mniejsze moduły, co jest tematem, w który zagłębiamy się w rozdziale 4.

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. Przeniesienie kodu klastra serwera WWW do katalogu `stage/services/webserver-cluster` wskazuje, że to jest robocza wersja 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. klastr serwera WWW powinien zostać przeniesiony do `stage/services/webserver-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`.

Klastr 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 do 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.

Ten układ plików ma wiele zalet:

Przejrzysty kod i układ środowiska

Taki układ plików znacznie ułatwia przeglądanie kodu i dokładne ustalenie komponentów wdrożonych w poszczególnych środowiskach.

Izolacja

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.

Praca z wieloma katalogami

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.

Rozwiązanie: jeżeli używasz Terragrunt, ten proces można zautomatyzować za pomocą polecenia `apply-all` (<https://terragrunt.gruntwork.io/docs/features/execute-terraform-commands-on-multiple-modules-at-once/>).

Kopiowanie i wklejanie

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`.

Rozwiązanie: 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.

Zależności zasobu

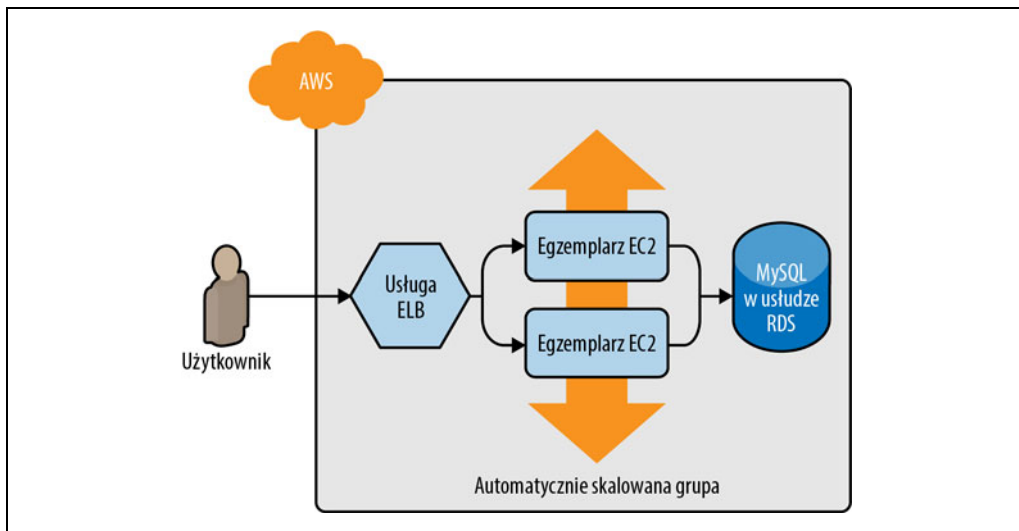
Pojawia się jeszcze inny problem z takim układem plików: znacznie trudniej jest wykorzystać zależności zasobu. Jeżeli kod aplikacji został zdefiniowany w tych samych plikach konfiguracyjnych, w których znajduje się kod bazy danych, to aplikacja będzie miała bezpośredni dostęp do atrybutów bazy danych 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.

Rozwiązanie: Jedną z możliwości jest używanie bloków `dependency` w Terragrunt, co dokładniej pokażę w rozdziale 10. Inna opcja to użycie źródła danych `terraform_remote_state`, co zostanie zaprezentowane w następnym podrozdziale.

Źródło danych `terraform_remote_state`

W rozdziale 2. źródło danych wykorzystałeś do pobrania z AWS informacji tylko do odczytu, takich jak źródło danych `aws_subnets`, 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.



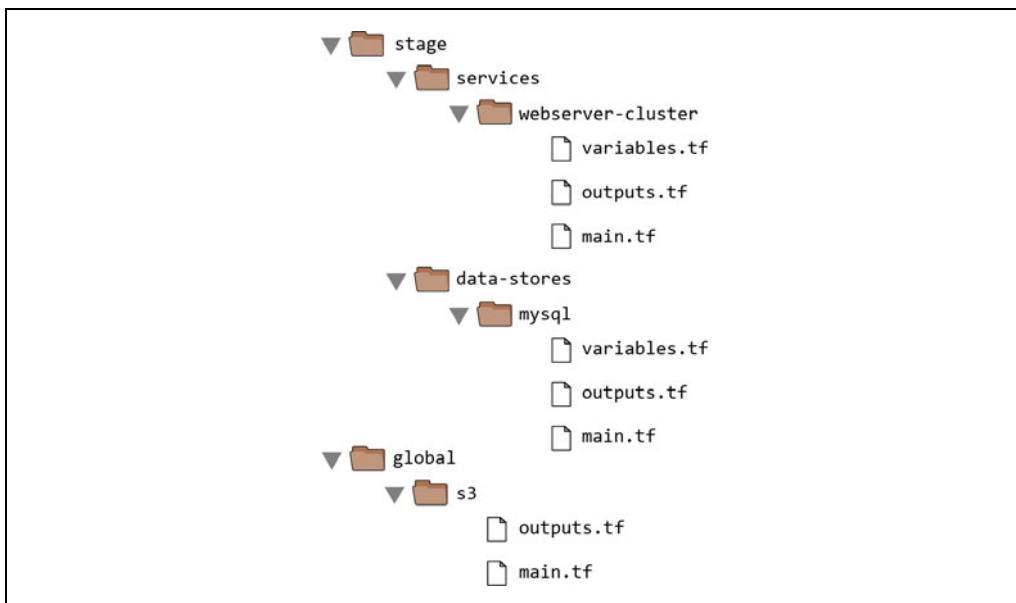
Rysunek 3.9. Klaster serwera WWW komunikuje się z bazą danych MySQL wdrożoną za pomocą oferowanej przez Amazon usługi RDS

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.

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"
  skip_final_snapshot = true
  db_name          = "example_database"
```



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

```
# Jak należy zdefiniować nazwę użytkownika i hasło?
username      = "???"
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 razem z takimi ustawieniami:

- silnik bazy danych MySQL;
- 10 GB pamięci masowej;
- egzemplarz `db.t2.micro` posiadający jeden wirtualny procesor, 1 GB pamięci RAM i dostępny w ramach bezpłatnego planu AWS;
- wyłączona możliwość utworzenia ostatecznej migawki, ponieważ ten kod jest przeznaczony jedynie do celów edukacyjnych i testowych (jeśli nie wyłączysz tworzenia migawki albo nie podasz dla niej nazwy za pomocą `final_snapshot_idenfifier`, operacja `destroy` zakończy się niepowodzeniem).

Zwróć uwagę na to, że wśród parametrów koniecznych do przekazania zasobowi `aws_db_instance` jest nazwa użytkownika głównego i hasło główne do bazy danych. Ponieważ to są dane wrażliwe, nie powinny być umieszczane bezpośrednio w kodzie zdefiniowanym w pliku zwykłego tekstu! W rozdziale 6. zaprezentuję różne możliwości w zakresie przekazywania danych wrażliwych do zasobów Terraform. W tym miejscu zostanie zastosowana opcja pozwalająca uniknąć przechowywania danych wrażliwych w plikach zwykłego tekstu i łatwa w użyciu: obsługa danych wrażliwych

przez 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 zmiennych środowiskowych.

W takim przypadku należy w pliku `stage/data-stores/mysql/variables.tf` zdefiniować zmienne o nazwach `db_username` i `db_password`.

```
variable "db_username" {
  description = "Nazwa użytkownika bazy danych"
  type        = string
  sensitive   = true
}

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

Po pierwsze, zauważ oznaczenie tych zmiennych za pomocą `sensitive = true`, co wskazuje na przechowywanie przez nie danych wrażliwych. To gwarantuje, że ich wartości nie zostaną zapisane podczas wykonywania poleceń `terraform plan` lub `terraform apply`. Po drugie, zwróć uwagę, że te zmienne nie mają wartości default. To zamierzone działanie. W pliku zwykłego tekstu nie należy przechowywać haseł ani żadnych innych danych wrażliwych. Zamiast tego należy wykorzystać zmienną środowiskową.

Zanim się tym zajmiemy, najpierw należy dokończyć tworzenie kodu. Zaczynamy od przekazania dwóch nowych zmiennych za pomocą zasobu `aws_db_instance`:

```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine            = "mysql"
  allocated_storage = 10
  instance_class    = "db.t2.micro"
  skip_final_snapshot = true
  db_name            = "example_database"

  username = var.db_username
  password = var.db_password
}
```

Po skonfigurowaniu danych wrażliwych następnym krokiem jest skonfigurowanie w kubku 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 kubka!
    bucket = "terraform-up-and-running-state"
    key    = "stage/data-stores/mysql/terraform.tfstate"
    region = "us-east-2"

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

Ostatnim krokiem jest dodanie dwóch zmiennych danych wyjściowych do pliku *stage/data-stores/mysql/outputs.tf* przekazujących adres bazy danych i numer portu.

```
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"
}
```

Teraz możesz przekazać nazwę użytkownika bazy danych i jego hasło za pomocą zmiennych środowiskowych. 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 zdefiniowanych wcześniej zmiennych danych wejściowych *db_username* i *db_password* utworzenie zmiennych środowiskowych *TF_VAR_username* i *TF_VAR_db_password* w systemach Linux, UNIX i macOS odbywa się za pomocą wymienionego tutaj polecenia:

```
$ export TF_VAR_db_username="(UŻYTKOWNIK_BAZY_DANYCH)"
$ export TF_VAR_db_password="(HASŁO_DO_BAZY_DANYCH)"
```

W Windows polecenia przedstawiają się następująco:

```
C:\> set TF_VAR_db_username="(UŻYTKOWNIK_BAZY_DANYCH)"
C:\> set TF_VAR_db_password="(HASŁO_DO_BAZY_DANYCH)"
```

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. Po zastosowaniu polecenia `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 = "terraform-up-and-running.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*.

Jeżeli powrócisz do kodu klastra serwera WWW, będzie on 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"

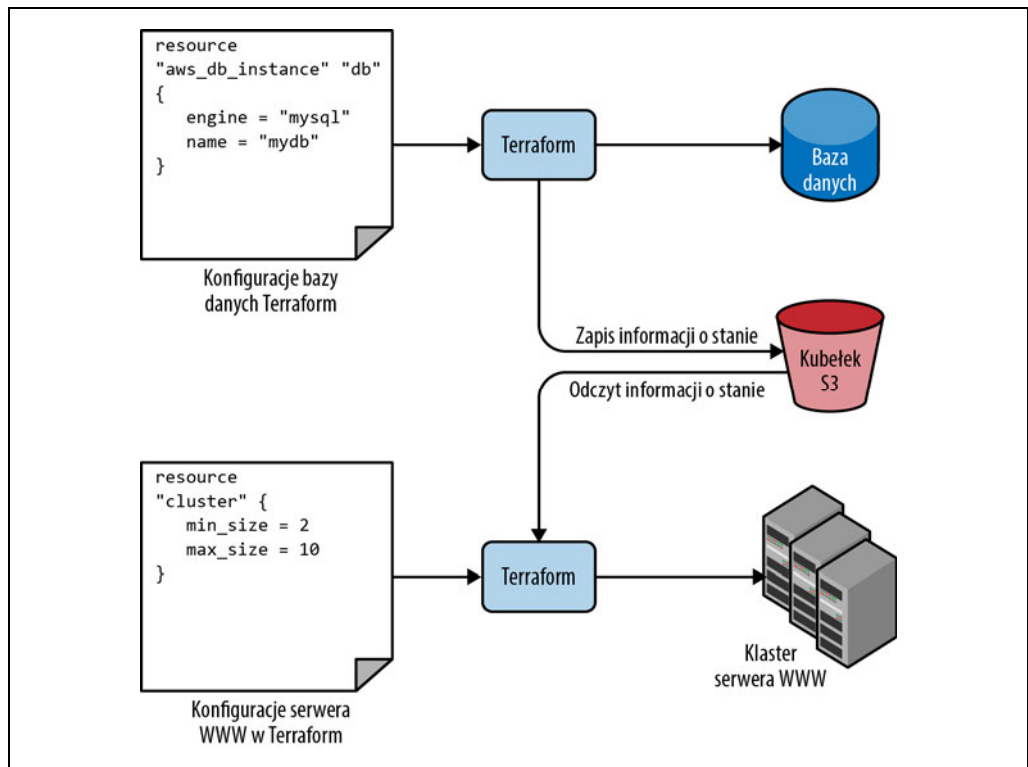
  config = {
    bucket = "(NAZWA_KUBEŁKA)"
  }
}
```

```

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 pokażalem 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 FMT⁶. 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 mapowaniach⁷. Jedną z nich jest funkcja `templatefile()`.

```
templatefile(<ŚCIEŻKA_DOSTĘPU>, <ZMIENNE>)
```

Ta funkcja odczytuje plik znajdujący się w podanej ścieżce dostępu, generuje go w postaci szablonu i zwraca jego zawartość w postaci ciągu tekstowego. Określenie „wygenerowanie jako szablonu” oznacza, że podany plik może używać składni interpolacji ciągu tekstowego w Terraform

⁶ Dokumentacja składni `sprintf` znajduje się na stronie <https://pkg.go.dev/fmt>.

⁷ Pełna lista funkcji wbudowanych znajduje się na stronie <https://www.terraform.io/language/functions>.

(\${...}), a Terraform wygeneruje zawartość tego pliku z wykorzystaniem podanych zmiennych. Aby zobaczyć to rozwiązanie w akcji, skrypt danych użytkownika można umieścić w pliku `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, jak się wkrótce przekonasz, dostępne są jedynie zmienne wymienione w drugim parametrze wywołania `templatefile()`. 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` w celu wywołania funkcji `templatefile()` i przekazania wymaganych zmiennych.

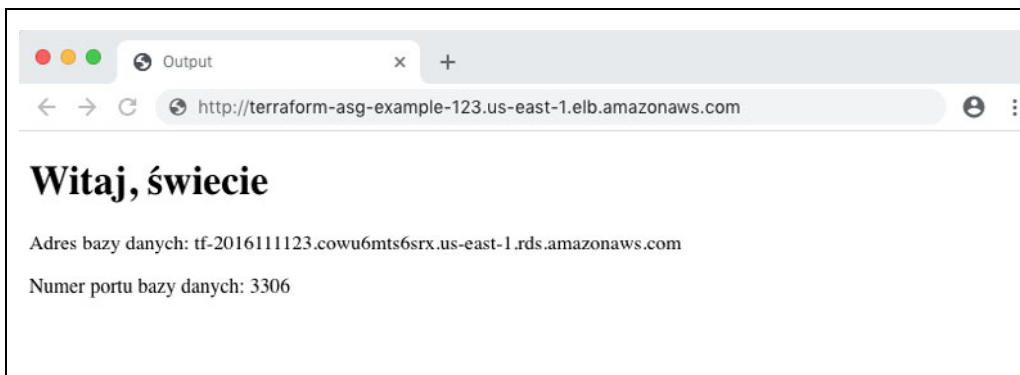
```
resource "aws_launch_configuration" "example" {
  image_id      = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
  security_groups = [aws_security_group.instance.id]

  # Wygenerowanie w postaci szablonu skryptu danych użytkownika.
  user_data = templatefile("user-data.sh", {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  })

  # Wymagane podczas używania konfiguracji startowej w automatycznie skalowanej grupie.
  lifecycle {
    create_before_destroy = true
  }
}
```

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

Gratulacje! 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

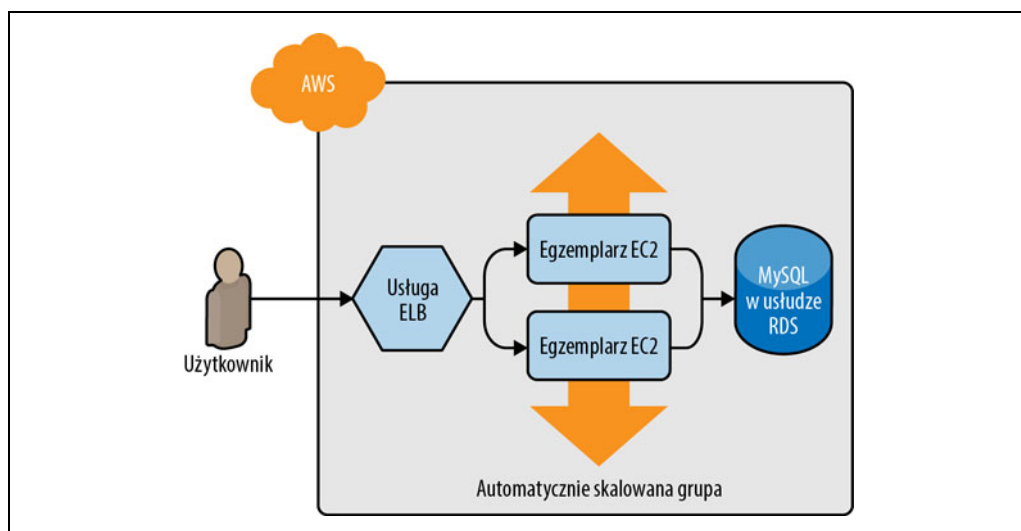
Powodem, dla którego tak wiele wysiłku wkłada się w izolację, nakładanie blokad i obsługę informacji o stanie, jest to, że infrastruktura wyrażona w postaci kodu (IaC) wiąże się z innymi kompromisami niż w przypadku typowego programowania. Gdy tworzysz kod typowej aplikacji, większość błędów jest stosunkowo niewielka i prowadzi do uszkodzenia jedynie fragmentu tej aplikacji. Kiedy natomiast tworzysz 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 kodem⁸.

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.

⁸ Więcej informacji na temat mechanizmów bezpieczeństwa znajdziesz na stronie <https://www.ybrikman.com/writing/2016/02/14/agility-requires-safety/>.

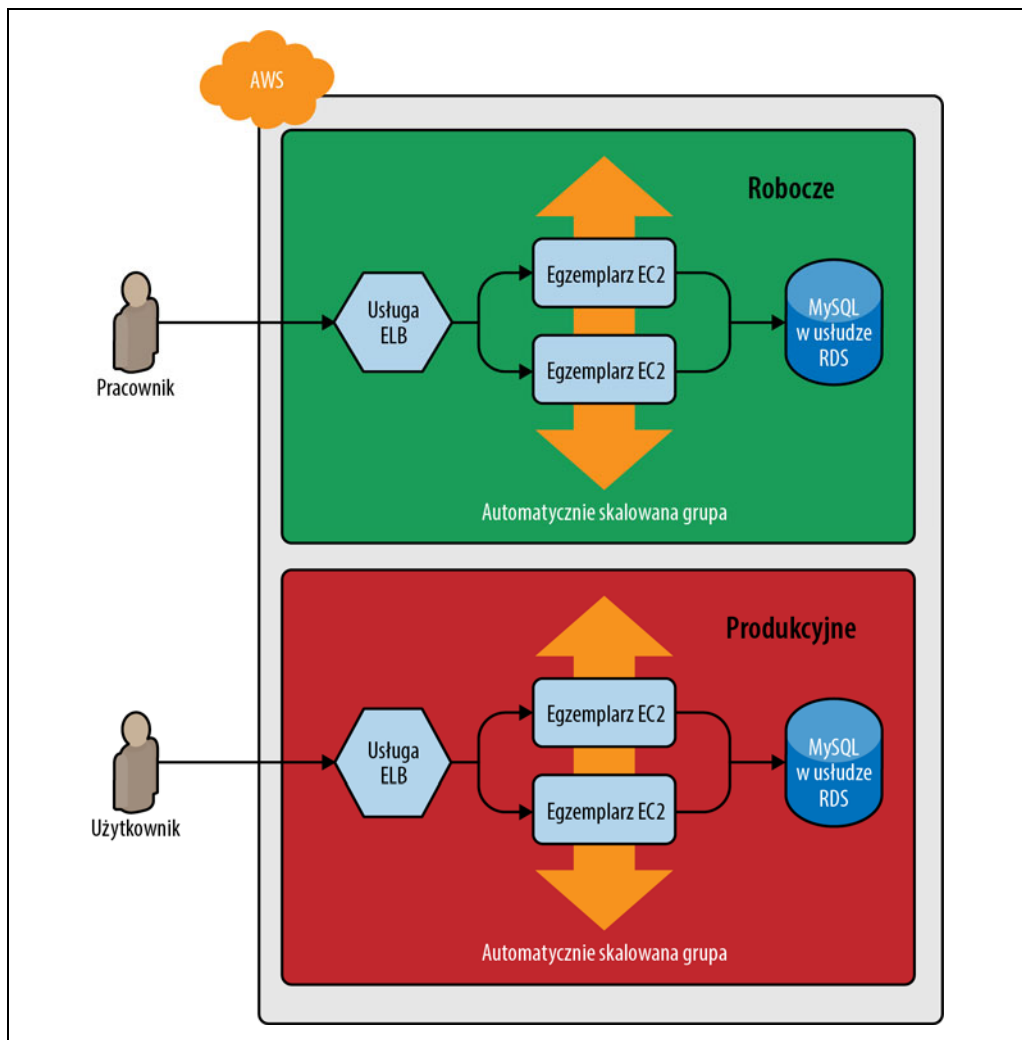
Zastosowanie modułów do tworzenia infrastruktury Terraform wielokrotnego użycia

Na końcu rozdziału 3. wdrożyłeś architekturę pokazaną na rysunku 4.1.



Rysunek 4.1. Architektura wdrożenia oparta na mechanizmie równoważenia obciążenia, klastrze serwerów 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.

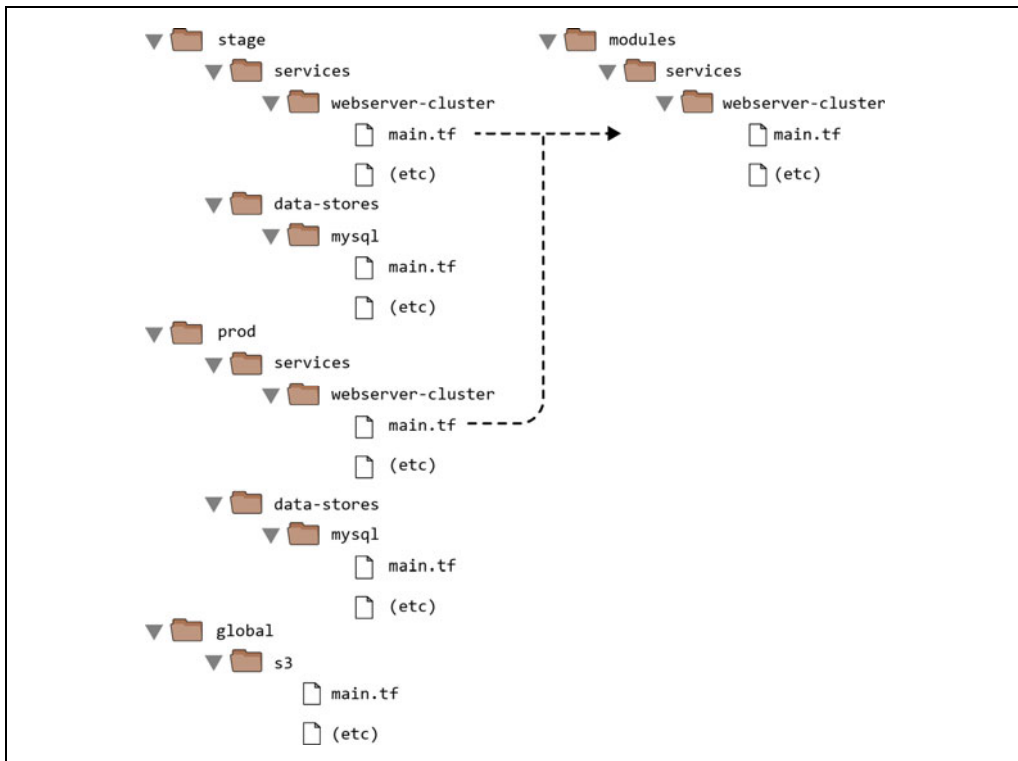
```
# Funkcja zdefiniowana w jednym miejscu.
def example_function()
  puts "Witaj, świecie"
```



```
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.

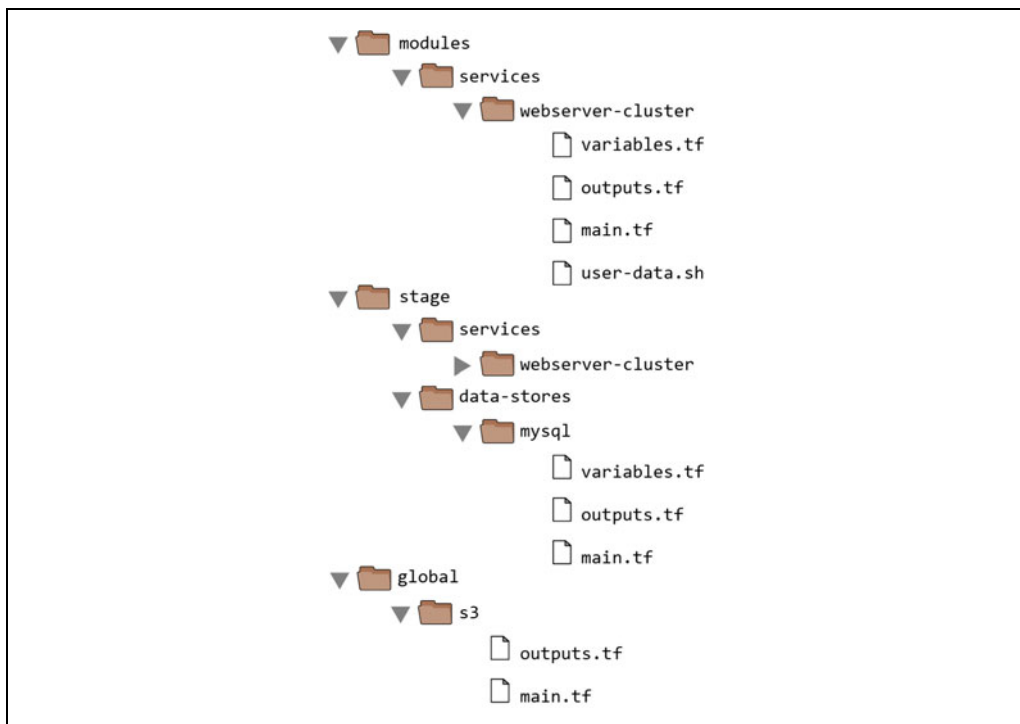
Otwórz plik *main.tf* w katalogu *modules/services/webserver-cluster* i usuń definicję `provider`. Dostawca powinien zostać skonfigurowany przez użytkownika modułu, a nie w samym module. (Więcej informacji na temat pracy z dostawcami znajdziesz w rozdziale 7.).

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. `webserver_cluster`), 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. Przeznaczony do tworzenia klastra serwera WWW kod wielokrotnego użycia należy przenieść do katalogu `modules/services/webserver-cluster`

```
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 znasz 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 użyjesz polecenia `terraform apply` dla tego kodu, musisz wiedzieć o problemie związanym z modułem `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.

```
# Funkcja pobierająca dwa parametry danych wejściowych.
def example_function(param1, param2)
  puts "Witaj, #{param1} #{param2}"
end

# Przekazanie funkcji dwóch parametrów danych wejściowych.
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      = "webservers-stage"
  db_remote_state_bucket = "(NAZWA_KUBEŁKA)"
  db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"
}

```

To samo należy powtórzyć w pliku *prod/services/webserver-cluster/main.tf* w środowisku produkcyjnym, ale z odmiennymi, właściwymi dla niego wartościami:

```
module "webserver_cluster" {
  source = "../../modules/services/webserver-cluster"

  cluster_name      = "webserver-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 zmienne danych wejściowych dla nazwy kubełka i zdanego stanu bazy danych, ale równie dobrze można zapewnić konfigurowalność pozostałych parametrów modułu. 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-0fb653ca2d3203ac1"
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]

  user_data = templatefile("user-data.sh", {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
  })
}
```

```

    db_port      = data.terraform_remote_state.db.outputs.port
  })

  # Wymagane podczas używania konfiguracji startowej wraz z automatycznie skalowaną grupą.
  lifecycle {
    create_before_destroy = true
  }
}

```

Powinienes 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_subnets.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 kłster 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 kłster 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 zdefiniowanie 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, powinienś 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.

```
# Funkcja zwracająca wartość.
def example_function(param1, param2)
    return "Witaj, #{param1} #{param2}"
end

# W innych miejscach kodu można wywołać tę funkcję i otrzymać jej wartość zwrótną.
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 `templatefile()` w celu odczytania z dysku zawartości tego pliku. Problem z funkcją `templatefile()` polega na tym, że używana ścieżka dostępu do pliku musi być względna (ponieważ narzędzie Terraform może być uruchamiane w różnych komputerach, każdy z innym układem dysku). Jednak względna dla którego komponentu?

Domyślnie Terraform interpretuje ścieżkę dostępu względem katalogu bieżącego. Takie rozwiązanie jest stosowane podczas użycia funkcji `templatefile()` w pliku konfiguracyjnym Terraform, znajdującym się w tym samym katalogu, w którym zostało wydane polecenie `terraform apply` (o ile funkcja `templatefile()` jest używana w module głównym), ale nie działa, gdy wywołanie `templatefile()` pojawia się w module zdefiniowanym w oddzielnym katalogu (moduł wielokrotnego użycia).

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 wypadku skryptu danych użytkownika potrzebna jest ścieżka dostępu względem samego modułu, więc podczas wywołania funkcji `templatefile()` w pliku `modules/services/webserver-cluster/main.tf` należy skorzystać z `path.module`.

```
user_data = templatefile("${path.module}/user-data.sh", {
  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. *Osadzony blok kodu* to argument umieszczony w zasobie i mający format:

```
resource "xxx" "yyy" {
  <NAZWA> {
    [KONFIGURACJA...]
  }
}
```

gdzie NAZWA to nazwa osadzonego bloku kodu (np. `ingress`), a KONFIGURACJA składa się z co najmniej jednego argumentu ściśle związanego z danym osadzonym blokiem kodu (np. `from_port` i `to_port`). W wypadku `aws_security_group_resource` można zdefiniować reguły przychodzącego i wychodzącego ruchu sieciowego za pomocą osadzonych bloków kodu (np. `ingress { ... }`) lub oddzielnych zasobów `aws_security_group_rule`.

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. Oto moja rada: podczas tworzenia modułu zawsze powinieneś preferować wykorzystanie zewnętrznego zasobu.

Zaletą używania oddzielnych zasobów jest możliwość ich dodawania gdziekolwiek, osadzony blok kodu natomiast może być dodany tylko w module tworzącym zasób. Tak więc używanie jedynie oddzielnych zasobów zapewnia modułowi większą elastyczność i konfigurowalność.

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
  }
}

```

Dzięki tym osadzonym blokom kodu użytkownik modułu nie ma możliwości dodawania z zewnątrz kolejnych reguł wejścia lub wyjścia. Aby zapewnić większą elastyczność, trzeba 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
}

```

Ponadto należy wyeksportować identyfikator `aws_security_group` jako zmienną 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. Można to zrobić bardzo łatwo, przez dodanie zasobu `aws_security_group_rule` w pliku `stage/services/webserver-cluster/main.tf`:

```

module "webserver_cluster" {
  source = "../../modules/services/webserver-cluster"

  # (Parametry zostały ukryte w celu zachowania przejrzystości).
}

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 dotyka 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

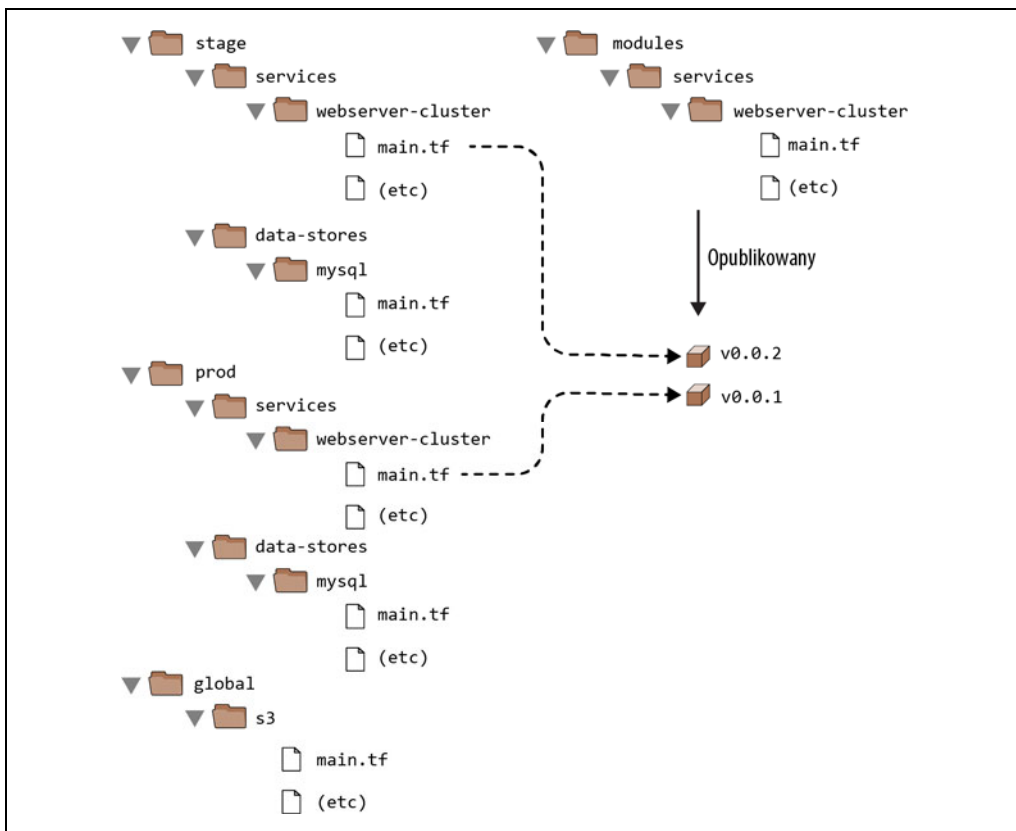
Przykłady w tym rozdziale prowadzą do utworzenia dwóch środowisk odizolowanych w kodzie Terraform, a także odizolowanych w kategoriach użycia oddzielnymi mechanizmami równoważenia obciążenia, serwerów i baz danych, choć jednocześnie nieodizolowanych na poziomie sieci. Aby zachować prostotę przykładów 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.



Rysunek 4.5. Dzięki wersjonowaniu modułów można używać różnych wersji modułu w odmiennych środowiskach, np. v0.0.1 w produkcyjnym i v0.0.2 w roboczym

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:

modules

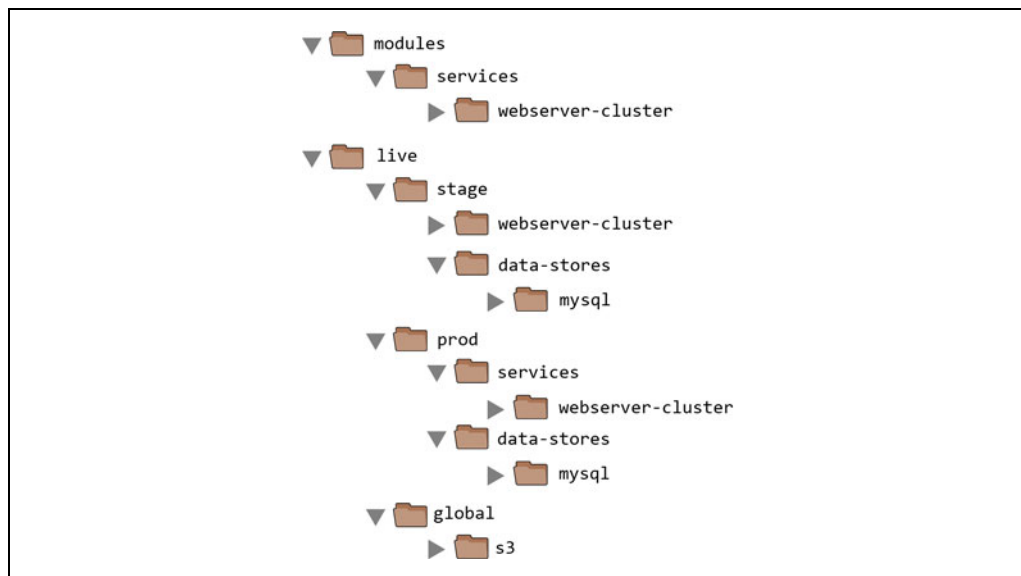
To repozytorium definiuje moduły wielokrotnego użycia. Każdy moduł potraktuj jako „matrycę” pozwalającą na zdefiniowanie określonego fragmentu infrastruktury.

¹ Więcej informacji na temat obsługiwanych adresów URL znajdziesz na stronie <https://developer.hashicorp.com/terraform/language/modules/sources>.

live

To repozytorium definiuje działającą infrastrukturę, która jest wykorzystywana w poszczególnych środowiskach (robozym, 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.



Rysunek 4.6. Moduły wersjonowane wielokrotnego użycia powinny być przechowywane w jednym repozytorium (tutaj to będzie *modules*), konfiguracja środowisk natomiast w innym (tutaj — *live*)

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*:

```
$ 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 main
```

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://docs.github.com/en/repositories/releasing-projects-on-github/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 `github.com/foo/modules` (zwróć uwagę na konieczność użycia podwójnego ukośnika w adresie URL repozytorium Git).

```
module "webserver_cluster" {
  source = "github.com/foo/modules//services/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 (musiałem 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=v0.3.0"
```

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=v0.3.0
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`.



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 umieszczania 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:acme/modules.git
```

Jeżeli wykonanie tych poleceń zakończy się sukcesem, Terraform będzie mieć możliwość wykorzystania także prywatnego repozytorium Git.

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*:

```
$ cd modules
$ git add .
$ git commit -m "Wprowadzono pewne zmiany w module webserver-cluster"
$ git push origin main
```

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.

² Więcej informacji na temat pracy z kluczami SSH podczas uzyskiwania dostępu do repozytorium znajdziesz na stronie <https://docs.github.com/en/authentication/connecting-to-github-with-ssh>.

```
module "webserver_cluster" {
  source = "github.com/foo/modules//services/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.

```
module "webserver_cluster" {
  source = "github.com/foo/modules//services/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, publikowania jego nowej wersji i ponownego wykonywania `terraform init` 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 wypadku 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 wiesz 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, blok cyklu życiowego o nazwie `create_before_destroy` i 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 odmiennej sytuacji:

- parametr `count` do przeprowadzania iteracji przez zasoby,
- wyrażenia `for_each` do przeprowadzania iteracji przez zasoby, osadzone bloki kodu w zasobie i moduły,
- 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.0", "neo.1", "neo.2").

Terraform will perform the following actions:

```
# Utworzony będzie zasób aws_iam_user.example[0].
+ resource "aws_iam_user" "example" {
+   + name           = "neo.0"
+   (...)
+ }

# Utworzony będzie zasób aws_iam_user.example[1].
+ resource "aws_iam_user" "example" {
+   + name           = "neo.1"
+   (...)
+ }

# Utworzony będzie zasób aws_iam_user.example[2].
+ resource "aws_iam_user" "example" {
+   + name           = "neo.2"
+   (...)
+ }
```

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 = var.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]
```

Funkcją `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" {
+   name      = "neo"
+   (...)
+ }

# Utworzony będzie zasób aws_iam_user.example[1].
+ resource "aws_iam_user" "example" {
+   name      = "trinity"
+   (...)
+ }

# Utworzony będzie zasób aws_iam_user.example[2].
+ resource "aws_iam_user" "example" {
```

```

+ name      = "morpheus"
(...)
}

```

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 "first_arn" {
  value      = aws_iam_user.example[0].arn
  description = "Wartość ARN dla pierwszego użytkownika"
}

```

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 `first_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:
```

```

first_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",
]

```

W wydaniu Terraform 0.13 parametr count może być używany także w modułach. Załóżmy, że istnieje moduł `modules/landing-zone/iam-user` odpowiadający za utworzenie pojedynczego użytkownika IAM:

```

resource "aws_iam_user" "example" {
  name = var.user_name
}

```

Nazwa użytkownika jest przekazywana do modułu za pomocą zmiennej danych wejściowych:

```
variable "user_name" {
  description = "Nazwa użytkownika do użycia"
  type        = string
}
```

Za pomocą zmiennej danych wyjściowych moduł zwraca wartość ARN utworzonego użytkownika IAM.

```
output "user_arn" {
  value        = aws_iam_user.example.arn
  description = "Wartość ARN utworzonego użytkownika IAM"
}
```

Tego modułu można użyć razem z parametrem `count` w celu utworzenia trzech użytkowników IAM, jak pokazałem w kolejnym fragmencie kodu.

```
module "users" {
  source = "../../modules/landing-zone/iam-user"

  count      = length(var.user_names)
  user_name = var.user_names[count.index]
}
```

Przedstawiony kod używa parametru `count` do iteracji przez listę nazw użytkowników.

```
variable "user_names" {
  description = "Utworzenie użytkowników IAM o podanych nazwach"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

Dane wyjściowe to wartości ARN utworzonych użytkowników:

```
output "user_arns" {
  value        = module.users[*].user_arn
  description = "Wartości ARNs dla utworzonych użytkowników"
}
```

Dodanie parametru `count` do zasobu zmienia go w tablicę zasobów, dodanie parametru `count` do modułu natomiast zmienia go w tablicę modułów.

Jeżeli zastosujesz polecenie `terraform apply` do tego kodu, zostaną wygenerowane przedstawione tutaj dane wyjściowe.

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
all_arns = [
  "arn:aws:iam::123456789012:user/neo",
  "arn:aws:iam::123456789012:user/trinity",
  "arn:aws:iam::123456789012:user/morpheus",
]
```

Jak widzisz, parametr `count` działa mniej więcej identycznie w wypadku zasobów i modułów.

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.

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_subnets.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 kubelków 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 kubelka o indeksie 1 zmień na morpheus i usuń kubelek 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. Możesz stracić dostępność (brak możliwości użycia użytkownika IAM w trakcie wykonywania polecenia terraform apply), a co gorsza, możesz również utracić dane (jeżeli usuwanym zasobem jest baza danych, skutkiem może być utrata wszystkich znajdujących się w niej danych!).

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, lub (c) wielu kopii w module. Najpierw przedstawię użycie for_each do utworzenia wielu kopii zasobu.

Składnia wygląda następująco:

```
resource "<DOSTAWCA>_<TYP>" "<NAZWA>" {  
  for_each = <KOLEKCJA>  
  
  [KONFIGURACJA ...]  
}
```

gdzie 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 `first_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`.

Wyrażenie `for_each` działa mniej więcej identycznie z modułami. Za pomocą używanego wcześniej modułu `iam-user` możesz utworzyć trzech użytkowników IAM, stosując do tego wyrażenie `for_each` w przedstawiony tutaj sposób.

```

module "users" {
  source = "../../modules/landing-zone/iam-user"

  for_each = toset(var.user_names)
  user_name = each.value
}

```

Wartości ARN dla tych użytkowników zostaną wyświetlone następująco:

```

output "user_arns" {
  value     = values(module.users)[*].user_arn
  description = "Wartości ARN dla utworzonych użytkowników"
}

```

Po użyciu polecenia `terraform apply` otrzymasz oczekiwane dane wyjściowe:

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 3 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",
]

```

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

  custom_tags = {
    Owner      = "team-foo"
    ManagedBy  = "terraform"
  }
}

```

W tym kodzie zostały zdefiniowane dwa użyteczne tagi: Owner określa zespół będący właścicielem tej grupy ASG, natomiast ManagedBy określa, że dana infrastruktura została wdrożona za pomocą Terraform. (To wskazuje, że ta infrastruktura nie powinna być modyfikowana ręcznie).

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_subnets.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”, `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_subnets.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 plan`, powinieneś otrzymać plan podobny do następującego:

```
$ terraform plan
```

```
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                = "ManagedBy"
+   propagate_at_launch = true
+   value              = "terraform"
+ }
}

```

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

Wymuszanie standardu używania tagów

Zwykle dobrym rozwiązaniem jest opracowanie przez zespół standardu dotyczącego używania tagów i tworzenie zgodnie z nim modułów Terraform. Jedną z możliwości jest ręczne upewnienie się, że każdy zasób w każdym module ma przypisane poprawne tagi. Dla wielu zasobów takie podejście okazuje się żmudne i podatne na błędy. Jeżeli istnieją tagi, które mają być zastosowane dla *wszystkich* zasobów AWS, znacznie lepszym podejściem będzie dodanie w każdym module bloku `default_tags` do dostawcy `aws`:

```

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

  # Tagi, które domyślnie zostaną zastosowane dla wszystkich zasobów AWS.
  default_tags {
    tags = {
      Owner      = "team-foo"
      ManagedBy = "Terraform"
    }
  }
}

```

Ten fragment kodu gwarantuje, że każdy zasób AWS tworzony w module będzie zawierał tagi `Owner` i `ManagedBy`. (Wyjątkiem są zasoby nieobsługujące tagów i zasób `aws_autoscaling_group`, który nie obsługuje tagów i nie działa z blokiem `default_tags`. Dlatego też wcześniej trzeba było wykonać nieco dodatkowej pracy i zdefiniować tagi w module `webserver-cluster`). Blok `default_tags` gwarantuje, że wszystkie zasoby będą miały przypisany podstawowy zbiór tagów, a jednocześnie pozwala na nadpisywanie tagów w poszczególnych zasobach. Z rozdziału 9. dowiesz się, jak definiować politykę jako kod i wymuszać jej stosowanie, aby „wszystkie zasoby miały tag `ManagedBy`”, za pomocą narzędzi typu OPA.

Pętla za pomocą wyrażenia `for`

Dotychczas dowiedziałeś się, jak używać pętli do tworzenia wielu kopii całych zasobów i bloków osadzonych. Być może się zastanawiasz, co jest potrzebne do zdefiniowania pojedynczej zmiennej lub parametru.

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:

```
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:

```
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:

```
bios = [
    "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 kwadratowym, (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:

```
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 wyrażeń 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 = "%{ for name in var.names }${name}, %{ endfor }"
}
```

Po wydaniu polecenia `terraform apply` otrzymasz następujące dane wyjściowe:

```
$ terraform apply

(...)
```

Outputs:

```
for_directive = "neo, trinity, morpheus, "
```

Istnieje także wersja składni dyrektywy `for` pozwalająca na używanie indeksu w pętli:

```
%{ for <INDEKS>, <ELEMENT> in <KOLEKCJA> }<ZAWARTOŚĆ>%{ endfor }
```

Oto przykład użycia indeksu:

```
output "for_directive_index" {  
  value = "%{ for i, name in var.names }${i}} ${name}, %{ endfor }"  
}
```

Po użyciu polecenia `terraform apply` otrzymasz takie dane wyjściowe:

```
$ terraform apply
```

```
(...)
```

Outputs:

```
for_directive_index = "(0) neo, (1) trinity, (2) morpheus, "
```

Zwróć uwagę na dodatkowy przecinek i spację na końcu. Możesz się pozbyć tych znaków za pomocą wyrażenia warunkowego — zwłaszcza w dyrektywie `if` — co omówiłem w następnym podrozdziale.

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 odmiennej sytuacji:

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 if

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 parametru 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"
    ManagedBy = "terraform"
  }
}

```

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`). Teraz już wiesz, jak można używać konstrukcji `if-else`.

Skoro masz możliwość tworzenia odpowiednich zasobów na podstawie warunku `if-else`, co możesz zrobić, jeśli zajdzie potrzeba uzyskania dostępu do atrybutu faktycznie utworzonego zasobu? Przykładowo chcesz dodać zmienną danych wyjściowych o nazwie `neo_cloudwatch_policy_arn` zawierającą wartość ARN rzeczywiście dołączonej biblioteki.

Najprostsze rozwiązanie polega na użyciu składni operatora trójargumentowego:

```
output "neo_cloudwatch_policy_arn" {
  value = (
    var.give_neo_cloudwatch_full_access
    ? aws_iam_user_policy_attachment.neo_cloudwatch_full_access[0].policy_arn
    : aws_iam_user_policy_attachment.neo_cloudwatch_read_only[0].policy_arn
  )
}
```

Wprawdzie takie rozwiązanie działa, ale kod jest nieco zawodny: jeżeli kiedykolwiek zmienisz warunek w parametrze `count` zasobów `aws_iam_user_policy_attachment` — to może nastąpić prawdopodobnie w przyszłości i będzie zależało od wielu zmiennych, a nie tylko od `var.give_neo_cloudwatch_full_access` — pojawi się niebezpieczeństwo, że zapomnisz uaktualnić warunek w tej zmiennej danych wyjściowych. W efekcie otrzymasz dezorientujący błąd podczas próby uzyskania dostępu do elementu tablicy, który może nie istnieć.

Znacznie bezpieczniejsze podejście polega na wykorzystaniu zalet funkcji `concat()` i `one()`. Funkcja `concat()` pobiera dwie lub więcej list jako dane wejściowe, a następnie łączy je w pojedynczą listę. Z kolei funkcja `one()` pobiera listę jako dane wejściowe. Jeśli lista ma zero elementów, wartością zwrótną funkcji będzie `null`. Jeżeli natomiast lista ma więcej niż jeden element, funkcja powoduje wygenerowanie komunikatu błędu. Po połączeniu wywołań obu tych funkcji i wyrażenia `splat` otrzymujemy taki fragment kodu:

```
output "neo_cloudwatch_policy_arn" {
  value = one(concat(
    aws_iam_user_policy_attachment.neo_cloudwatch_full_access[*].policy_arn,
    aws_iam_user_policy_attachment.neo_cloudwatch_read_only[*].policy_arn
  ))
}
```

W zależności od wyniku wykonania konstrukcji warunkowej `if-else` tablica `neo_cloudwatch_full_access` będzie pusta lub `neo_cloudwatch_read_only` będzie zawierać tylko jeden element albo na odwrót, po ich połączeniu więc otrzymasz listę z jednym elementem, który następnie zostanie zwrócony przez funkcję `one()`. Takie rozwiązanie będzie działało poprawnie niezależnie od zmian wprowadzanych w konstrukcji `if-else`.

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, zero bloków osadzonych lub zero modułów. Natomiast skutkiem przekazania niepustej kolekcji jest utworzenie jednego lub więcej zasobów, bloków osadzonych lub modułów. 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, nie zostaną więc zdefiniowane żadne tagi. Innymi słowy, masz już pewną logikę warunkową. Można jednak 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 wyrażeń 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).

Wcześniej w rozdziale dyrektywa `for` została użyta do zdefiniowania pętli w celu wyświetlenia wielu imion rozdzielonych przecinkami. Problem polegał na pojawieniu się dodatkowego przecinka i spacji na końcu ciągu tekstowego. Można go rozwiązać za pomocą dyrektywy ciągu tekstowego `if`, w zaprezentowany tutaj sposób:

```
output "for_directive_index_if" {
  value = <<EOF
  %{ for i, name in var.names }
    ${name}%{ if i < length(var.names) - 1 }, %{ endif }
  %{ endfor }
  EOF
}
```

Mamy tutaj pewne zmiany względem wcześniejszej wersji:

- Kod został zdefiniowany z wykorzystaniem składni *HEREDOC*, która jest używana podczas definiowania wielowierszowych ciągów tekstowych. To pozwala na umieszczenie kodu w kilku wierszach, co poprawia jego czytelność.
- Dyrektywa ciągu tekstowego `if` została użyta, aby po ostatnim elemencie nie było przecinka i spacji.

Po użyciu polecenia `terraform apply` otrzymasz takie dane wyjściowe:

```
$ terraform apply
```

```
(...)
```

```
Outputs:
```

```
for_directive_index_if = <<EOT
neo,

trinity,

morpheus

EOT
```

Ups! Przecinek na końcu ostatniego elementu zniknął, ale pojawiło się wiele białych znaków (spacje i znaki nowego wiersza). Każdy biały znak w *HEREDOC* będzie się również znajdował w ostatecznym ciągu tekstowym. Aby rozwiązać tę kwestię, w dyrektywie ciągu tekstowego możesz skorzystać ze **znacznika usuwania białych znaków** (`-`), który spowoduje usunięcie wszystkich białych znaków 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_index_if_strip" {
  value = <<EOF
  %{~ for i, name in var.names ~}
  ${name}%{ if i < length(var.names) - 1 }, %{ endif }
  %{~ endfor ~}
  EOF
}
```

Wypróbujemy teraz uaktualnioną wersję:

```
$ terraform apply
```

```
(...)
```

Outputs:

```
for_directive_index_if_strip = "neo, trinity, morpheus"
```

W porządku, nastąpiła znaczna poprawa: pozbyliśmy się dodatkowych białych znaków i przecinków. Dane wyjściowe mogą być jeszcze bardziej eleganckie po wykorzystaniu klauzuli `else` o takiej składni:

```
%{ 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, w którym klauzula `else` została użyta w celu dodania kropki na końcu.

```
output "for_directive_index_if_else_strip" {
  value = <<EOF
  %{~ for i, name in var.names ~}
  ${name}%{ if i < length(var.names) - 1 }, %{ else }.%{ endif }
  %{~ endfor ~}
  EOF
}
```

Po użyciu polecenia `terraform apply` otrzymasz takie dane wyjściowe:

```
$ terraform apply
```

```
(...)
```

Outputs:

```
for_directive_index_if_else_strip = "neo, trinity, morpheus."
```

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/variables.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"
  type        = string
  default     = "ami-0fb653ca2d3203ac1"
}

variable "server_text" {
  description = "Ciąg tekstowy zwracany przez serwer"
  type        = string
  default     = "Witaj, świecie"
}
```

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`, uaktualnij parametr `image_id` w celu użycia `var.ami` i wywołanie `templatefile()` w parametrze `user_data`, aby przekazać wartość za pomocą `var.server_text`.

```
resource "aws_launch_configuration" "example" {
  image_id      = var.ami
  instance_type = var.instance_type
  security_groups = [aws_security_group.instance.id]

  user_data = templatefile("${path.module}/user-data.sh", {
    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
  })

  # Wymagane tylko w przypadku konfiguracji startowej z automatycznie skalowaną grupą.
  lifecycle {
    create_before_destroy = true
  }
}
```

Teraz w środowisku roboczym (`live/stage/services/webserver-cluster/main.tf`) można przypisać nowe parametry `ami` i `server_text`:

```
module "webserver_cluster" {
  source = "../../../../../modules/services/webserver-cluster"
```



```

ami          = "ami-0fb653ca2d3203ac1"
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 powinieneś 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-0fb653ca2d3203ac1"
    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 przestoju¹:

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 zmieni, 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" {
  # Wyraźna 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_subnets.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:
      key => upper(value)
    }
  }
```

¹ Podziękowania za opracowanie tej techniki należą się Paulowi Hinze'owi (<https://groups.google.com/g/terraform-tool/c/7GdhvIOAc80/m/iNQ93riiLwAJ>).

```

    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-0fb653ca2d3203ac1"
    instance_type = "t2.micro"
  ~ name        = "terraform-2019" -> (known after apply)
  ~ user_data    = "bd7c0a" -> "4919a" # Wymuszone zastąpienie.
  (...)
}

(...)

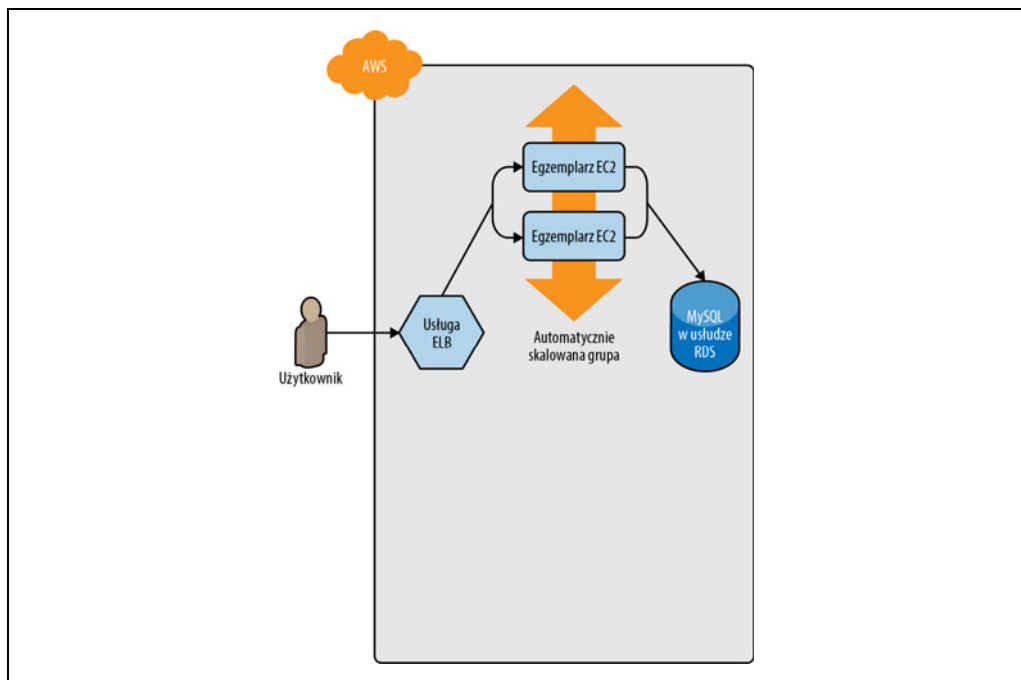
```

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

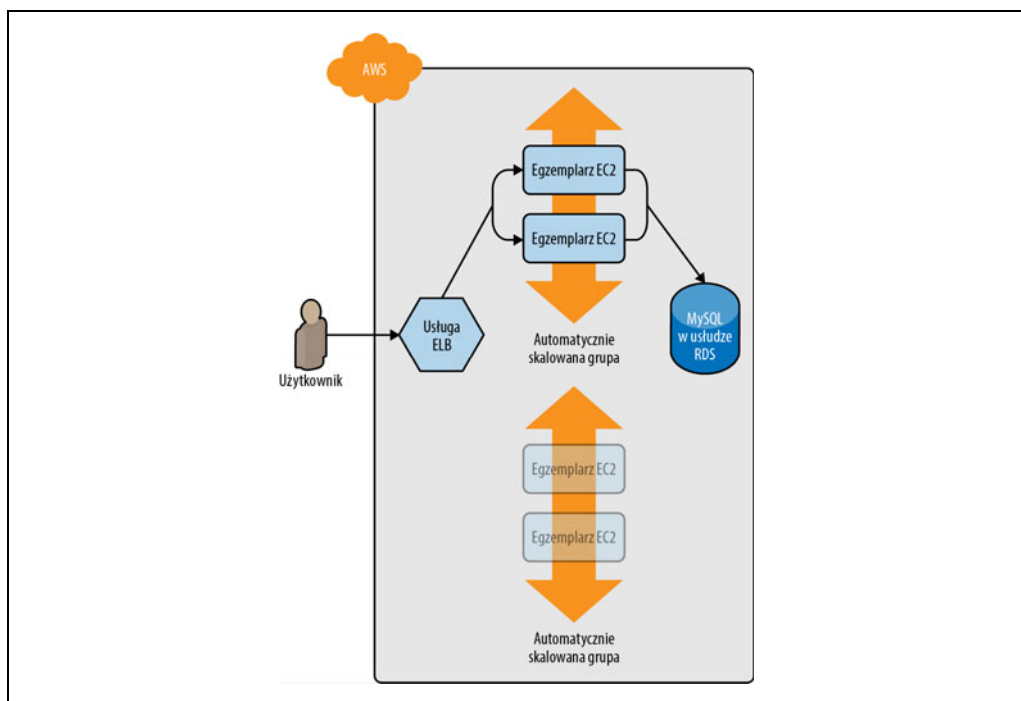
Kluczową kwestią, na którą należy zwrócić uwagę, jest to, że zasób `aws_autoscaling_group` ma obok nazwy parametru umieszczony komunikat `forces replacement` oznaczający zastąpienie go nową grupą ASG działającą wraz z nowym obrazem AMI lub danymi użytkownika. Wydadaj polecenie `terraform apply`, aby 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 5.1).

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 5.2).

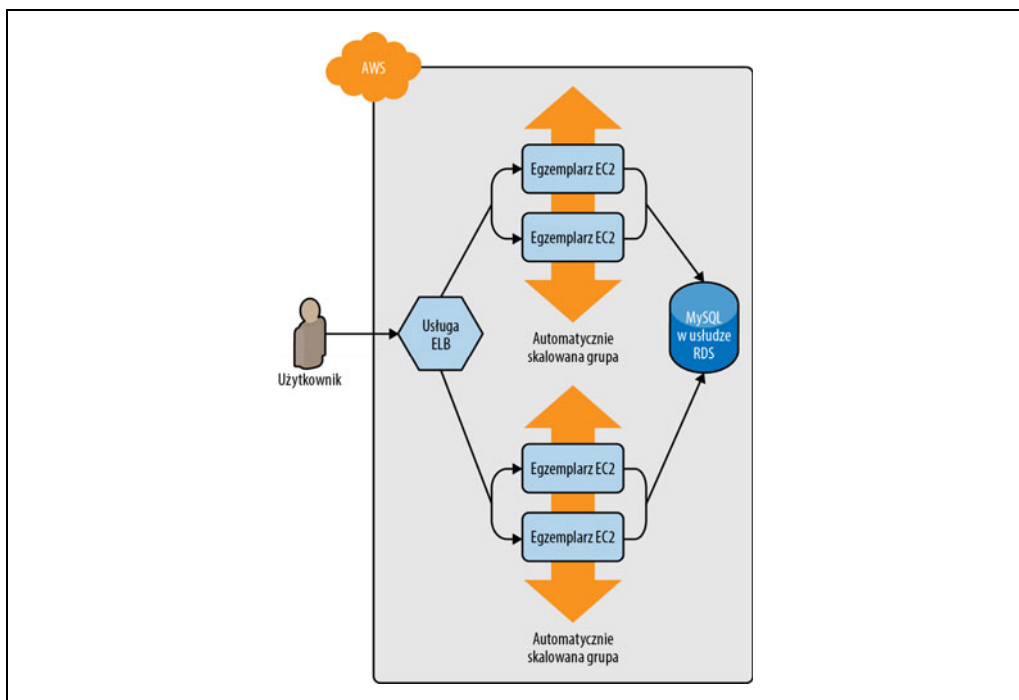


Rysunek 5.1. Początkowo masz pierwotną grupę ASG wraz z uruchomionym kodem w wersji v1



Rysunek 5.2. Terraform rozpoczyna wdrażanie nowej grupy ASG wraz z kodem w wersji v2

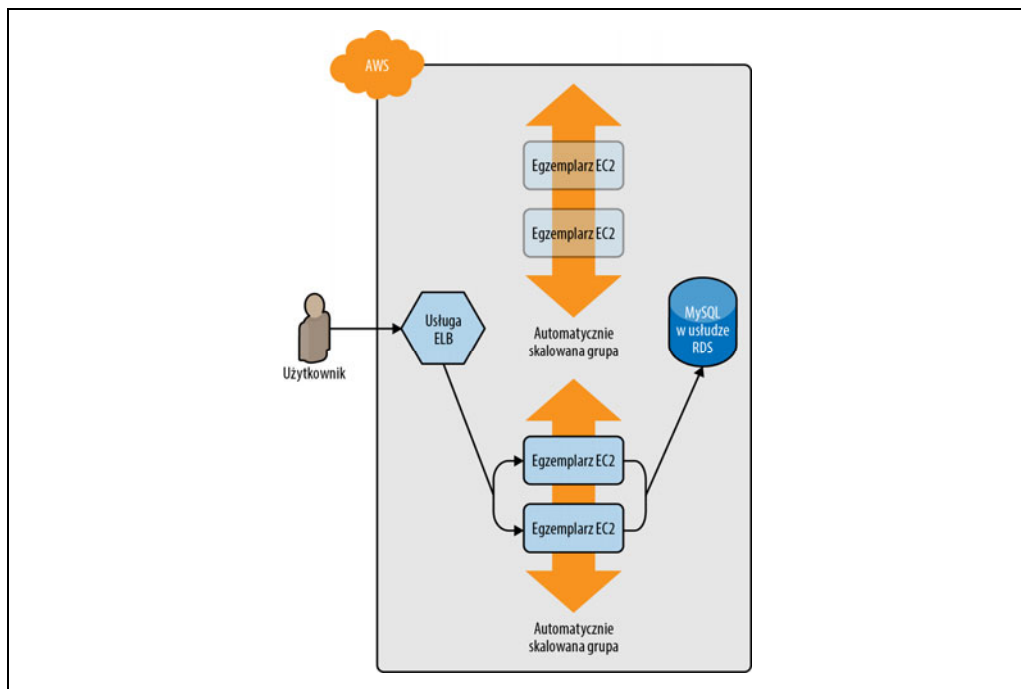
Po 1 – 2 minutach serwery nowej grupy ASG są uruchomione, nawiązały połączenie z bazą danych, zostały zarejestrowane w mechanizmie równoważenia obciążenia oraz rozpoczęły procedurę sprawdzenia swojego stanu. Na tym etapie jednocześnie działają wersje v1 i v2 aplikacji, a wersja używana przez użytkownika zależy od tego, do której został on przekierowany przez mechanizm równoważenia obciążenia (zobacz rysunek 5.3).



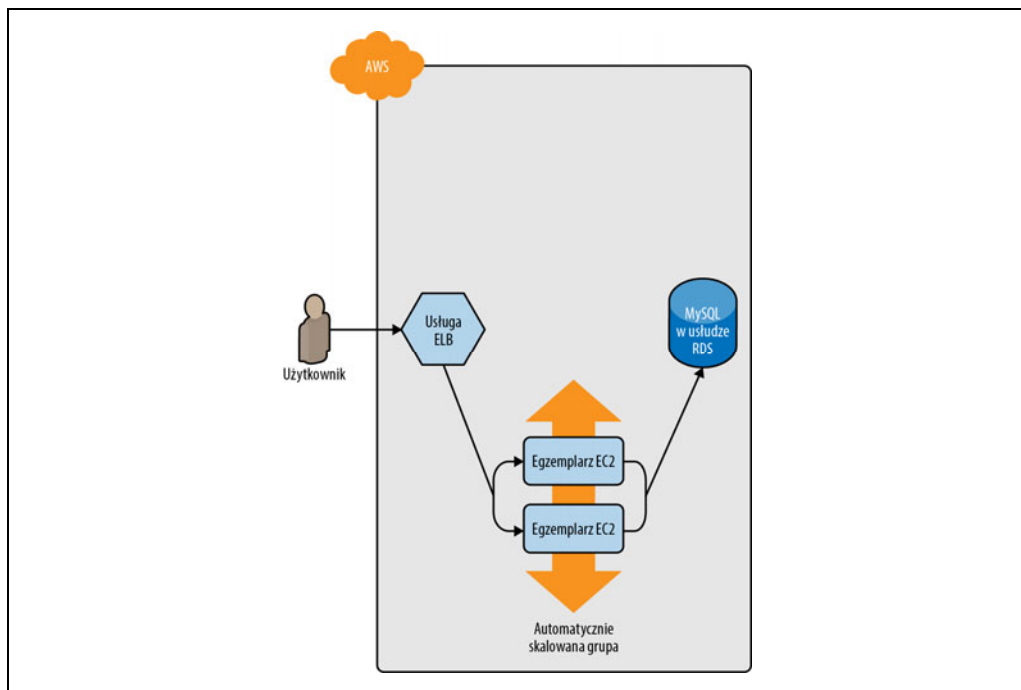
Rysunek 5.3. Serwery w nowej grupie ASG są uruchomione, nawiązały połączenie z bazą danych, zostały zarejestrowane przez mechanizm równoważenia obciążenia i rozpoczęły obsługę ruchu sieciowego

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 5.4).

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 5.5).



Rysunek 5.4. Serwery w starej grupie ASG zaczynają być wyłączane



Rysunek 5.5. W tym momencie pozostała tylko nowa grupa ASG, w której działa kod w wersji v2

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 5.6.



Rysunek 5.6. Nowa wersja kodu została wdrożona

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_ebl>; sleep 1; done
```

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>`):

```
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.

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 poważne ograniczenie, o którym trzeba wiedzieć: 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-0fb653ca2d3203ac1"
  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-0fb653ca2d3203ac1"
  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 domyślasz 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-0fb653ca2d3203ac1"
  instance_type  = "t2.micro"
}
```

Po użyciu polecenia `terraform plan` 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.

Ograniczenia wdrożenia bez przestoju

Wykorzystanie ustawienia `create_before_destroy` w grupie ASG to doskonała technika wdrożenia bez przestoju, choć niestety niepozbawiona wad.

Pierwszym ograniczeniem jest to, że nie działa wraz z politykami automatycznego skalowania. A dokładniej 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, np. zmiana parametru `recurrence` w zasobie `aws_autoscaling_schedule`, a także przypisanie parametrowi `desired_capacity` grupy ASG wartości zwróconej przez skrypt używający API AWS do ustalenia liczby egzemplarzy działających przed wdrożeniem.

Drugim i większym ograniczeniem jest to, że w razie ważnych i skomplikowanych zadań, np. wdrożenia bez przestoju, naprawdę chcesz skorzystać z natywnego i najlepszego rozwiązania, a nie ze sztuczek wymagających łączenia na chybił trafił `create_before_destroy`, `min_elb_capacity`, niestandardowych skryptów itd. Okazuje się, że w wypadku automatycznie skalowanej grupy AWS oferuje natywne rozwiązanie o nazwie *odświeżenie egzemplarza*.

Powróć do zasobu `aws_autoscaling_group` i wycofaj zmiany wprowadzone we wdrożeniu bez przestoju:

- W miejsce wartości `name` przywróć `var.cluster_name`, zamiast polegać na nazwie `aws_launch_configuration`.
- Usuń ustawienia `create_before_destroy` i `min_elb_capacity`.

Następnie uaktualnij zasób `aws_autoscaling_group`, aby korzystał z bloku `instance_refresh`, jak pokazano w kolejnym fragmencie kodu.

```
resource "aws_autoscaling_group" "example" {
  name = var.cluster_name
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier = data.aws_subnets.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

  # Użycie natychmiastowego odświeżania w celu przekazania zmian do grupy ASG.
  instance_refresh {
    strategy = "Rolling"
    preferences {
      min_healthy_percentage = 50
    }
  }
}
```

Jeżeli wdrożysz tę grupę ASG, a następnie zmienisz wybrane parametry (np. `server_text`) i zastosujesz polecenie `terraform plan`, różnica będzie się sprowadzała do zaledwie uaktualnienia `aws_launch_configuration`:

Terraform will perform the following actions:

```
# Zasób module.webserver_cluster.aws_autoscaling_group.ex zostanie uaktualniony w miejscu.
~ resource "aws_autoscaling_group" "example" {
```

```

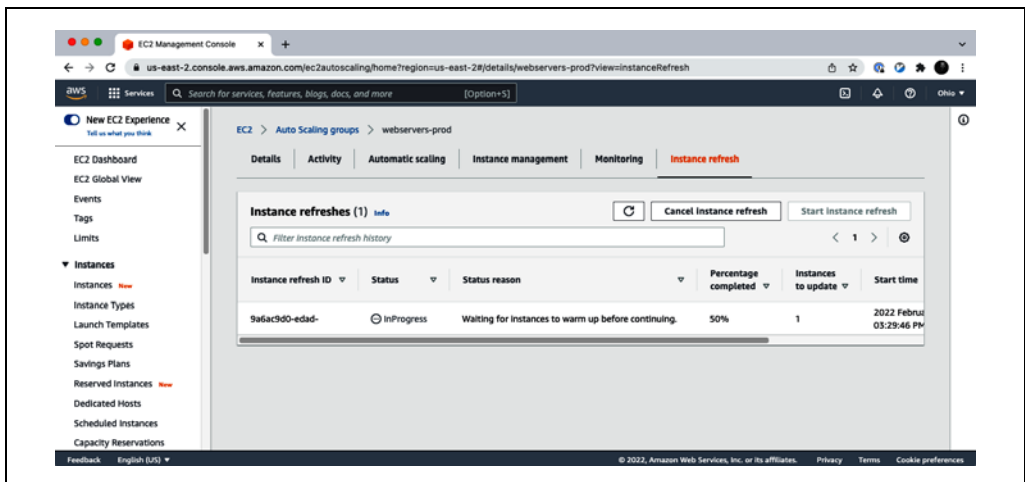
    id = "webserver-stage-terraform-20190516"
    ~ launch_configuration = "terraform-20190516" -> (known after apply)
    (...)
  }

# Zasób module.webserver_cluster.aws_launch_configuration.ex musi zostać zastąpiony.
+/- resource "aws_launch_configuration" "example" {
  ~ id = "terraform-20190516" -> (known after apply)
    image_id = "ami-0fb653ca2d3203ac1"
    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.

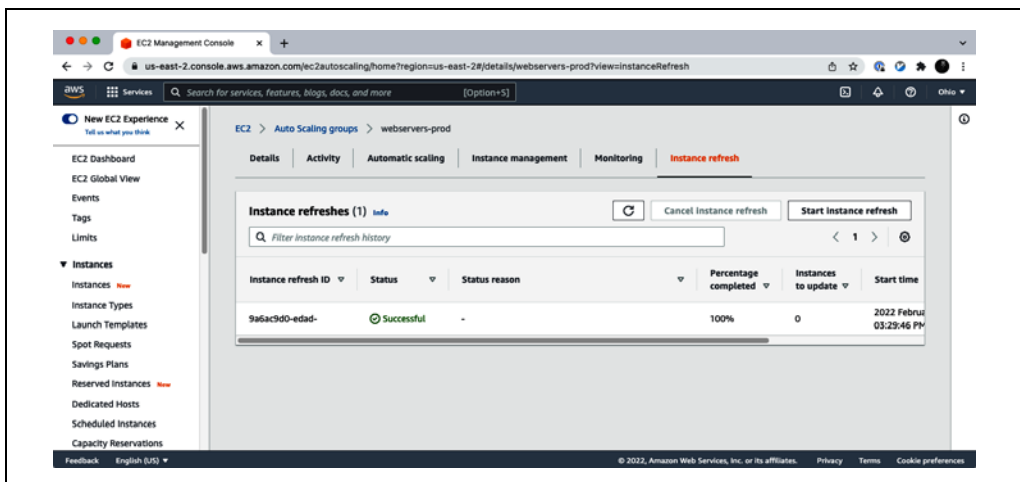
Jeżeli użyjesz polecenia `terraform apply`, zostanie ono wykonane bardzo szybko. Na początku nic nowego nie zostanie wdrożone. Jednak w tle, na skutek modyfikacji konfiguracji startowej, AWS rozpocznie proces odświeżania egzemplarza, jak pokazałem na rysunku 5.7.



Rysunek 5.7. Trwający proces odświeżania egzemplarza

Początkowo AWS uruchomi jeden nowy egzemplarz, poczeka na zaliczenie przez niego testów, wyłączy jeden stary egzemplarz. Następnie ten proces zostanie powtórzony dla drugiego egzemplarza i będzie kontynuowany aż do odświeżenia egzemplarza, jak pokazałem na rysunku 5.8.

Ten proces jest w całości zarządzany przez AWS, można go skonfigurować, świetnie radzi sobie z obsługą błędów i nie wymaga sztuczek. Jediną jego wadą jest to, że czasami bywa wolny (zastąpienie dwóch serwerów może zabrać do 20 minut). Pomijając tę wadę, jest to znacznie skuteczniejsze rozwiązanie w zakresie wdrażania bez przestoju.



Rysunek 5.8. Ukończenie procesu odświeżania egzemplarza

Ogólnie rzecz biorąc, gdy tylko istnieje możliwość, należy preferować dobre, natywne opcje, takie jak odświeżanie egzemplarza. Wprawdzie te opcje nie zawsze były dostępne we wcześniejszych wersjach Terraform, ale obecnie wiele zasobów zapewnia obsługę natywnych opcji wdrożeń. Jeśli np. używasz Amazon ECS (Elastic Container Service) do wdrażania kontenerów Dockera, zasób `aws_ecs_service` natywnie obsługuje wdrożenie bez przestoju dzięki parametrom `deployment_maximum_percent` i `deployment_minimum_healthy_percent`. Jeżeli używasz Kubernetes do wdrażania kontenerów Dockera, zasób `kubernetes_deployment` natywnie obsługuje wdrożenie bez przestoju dzięki parametrowi `strategy` o wartości `RollingUpdate` i dostarczeniu konfiguracji za pomocą bloku `rolling_update`. Zajrzyj do dokumentacji używanego zasobu, a następnie, gdzie tylko możesz, korzystaj z natywnej funkcjonalności.

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.
  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:

Jeśli 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` powinienś 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 takimi narzędziami jak `terraformer` (<https://github.com/GoogleCloudPlatform/terraformer>) i `terracognita` (<https://github.com/cycloidio/terracognita>), które potrafią automatycznie zaimportować z obsługiwanych środowisk chmury 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_subnets.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. Podobne problemy mogą się pojawić w wypadku zmiany identyfikatora powiązanego z modułem, podziału modułu na wiele innych, dodania `count` lub `for_each` do zasobu albo modułu, który wcześniej nie miał podanego parametru czy wyrażenia.

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`.

Refaktoryzacja może wymagać zmiany stanu

Jeżeli chcesz przeprowadzić refaktoryzację kodu bez przypadkowego spowodowania przestoju, 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`. Ewentualnie można to zrobić automatycznie, poprzez dodanie bloku `moved` do kodu.

Zacniemy od przyjrzenia się poleceniu `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.

Konieczność pamiętania poleceń powłoki w celu ich ręcznego wykonywania jest uciążliwa i stwarza ryzyko popełnienia błędu, zwłaszcza jeśli moduł jest refaktoryzowany dziesiątki razy przez firmę. Wówczas każdy zespół musi pamiętać o wykonaniu polecenia `terraform state mv` w celu uniknięcia przestoju. Na szczęście w Terraform 1.1 wprowadzono możliwość automatycznej obsługi dzięki blokom `moved`. W trakcie każdej refaktoryzacji należy dodać blok `moved` w celu przechwycenia informacji o tym, jak powinien być uaktualniony stan. Przykładowo w celu przechwycenia informacji, że zasób `aws_security_group` ma zmieniony identyfikator z `instance` na `cluster_instance`, należy dodać ten blok `moved`:

```
moved {
  from = aws_security_group.instance
  to   = aws_security_group.cluster_instance
}
```

Teraz, po wykonaniu polecenia `terraform apply` dla tego kodu, Terraform automatycznie wykryje konieczność uaktualnienia pliku stanu:

Terraform will perform the following actions:

```
# Zasób aws_security_group.instance został przeniesiony
# do aws_security_group.cluster_instance.
resource "aws_security_group" "cluster_instance" {
  name = "moved-example-security-group"
```



```

tags                = {}
# (8 unchanged attributes hidden)
}

```

Plan: 0 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:

Jeżeli wpiszesz **yes**, Terraform automatycznie uaktualni informacje o stanie, a jeśli wynik polecenia `terraform plan` nie będzie wskazywał zasobów przeznaczonych do dodania, zmiany lub usunięcia, Terraform nie wprowadzi żadnych zmian — dokładnie tego oczekujemy!

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`.

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://developer.hashicorp.com/terraform/language/functions>) 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 zapewniające bezpieczną obsługę danych poufnych i wrażliwych.

Zarządzanie danymi poufnymi za pomocą Terraform

W pewnym momencie Twoje oprogramowanie będzie zawierało wiele różnego rodzaju danych poufnych, takich jak hasła baz danych, klucze API, certyfikaty TLS, klucze SSH i GPG itd. To wszystko są informacje wrażliwe, które, jeśli dostaną się w niepowołane ręce, mogą wyrządzić wiele szkód firmie i jej klientom. Jeżeli tworzysz oprogramowanie, Twoim zadaniem jest zapewnienie bezpieczeństwa tym informacjom.

Rozważ np. przedstawiony tutaj fragment kodu Terraform odpowiedzialny za wdrożenie bazy danych.

```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine           = "mysql"
  allocated_storage = 10
  instance_class   = "db.t2.micro"
  skip_final_snapshot = true
  db_name          = var.db_name

  # Jak można bezpiecznie zdefiniować te parametry?
  username = "???"
  password = "???"
}
```

Ten kod wymaga zdefiniowania dwóch rodzajów informacji tajnych, nazwy użytkownika i hasła, czyli danych uwierzytelniających głównego użytkownika bazy danych. Jeżeli trafią w nieodpowiednie ręce, może to mieć katastrofalne skutki, ponieważ te dane uwierzytelniające zapewniają dostęp na poziomie superużytkownika do tej bazy danych i wszystkiego, co jest w niej przechowywane. W jaki sposób można zapewnić bezpieczeństwo tajnych informacji?

To jest fragment szerszego tematu **zarządzania danymi poufnymi**, na którym skoncentrowałem się w tym rozdziale. Zostaną w nim omówione takie zagadnienia:

- podstawy zarządzania danymi poufnymi,
- narzędzia przeznaczone do zarządzania danymi poufnymi,
- narzędzia przeznaczone do zarządzania danymi poufnymi w Terraform.

Podstawy zarządzania danymi poufnymi

Pierwsza zasada dotycząca zarządzania danymi poufnymi brzmi:

Danych poufnych nie przechowuj w postaci zwykłego tekstu.

Druga zasada dotycząca zarządzania danymi poufnymi brzmi:

DANYCH POUFNYCH NIE PRZECHOWUJ W POSTACI ZWYKŁEGO TEKSTU.

Naprawdę tego nie rób. Przykładowo *nie* umieszczaj danych uwierzytelniających do bazy danych bezpośrednio w kodzie Terraform, który następnie zostanie przekazany do systemu kontroli wersji.

```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine           = "mysql"
  allocated_storage = 10
  instance_class   = "db.t2.micro"
  skip_final_snapshot = true
  db_name          = var.db_name

  # NIGDY TEGO NIE RÓB!!!
  username = "admin"
  password = "hasło"
  # NIGDY TEGO NIE RÓB!!!
}
```

Umieszczanie danych poufnych w postaci zwykłego tekstu, który następnie trafia do systemu kontroli wersji, to *fatalny pomysł*. Oto zaledwie kilka powodów:

Każdy z dostępem do systemu kontroli wersji będzie miał również dostęp do tych danych poufnych

W poprzednim przykładzie każdy programista w firmie, który ma dostęp do kodu Terraform, uzyskuje również dostęp do najważniejszych danych uwierzytelniających do bazy danych.

Każdy komputer z dostępem do systemu kontroli wersji będzie zawierał kopię tych danych poufnych

Każdy komputer, w którym kiedykolwiek pobrano dane repozytorium, nadal może zawierać na dysku lokalnym te dane poufne. To obejmuje komputery wszystkich programistów w zespole, wszystkie komputery zaangażowane w obsługę systemu ciągłej integracji (np. Jenkins, CircleCI, GitLab itd.), każdy komputer zaangażowany w kontrolę wersji (np. GitHub, GitLab, BitBucket), każdy komputer zaangażowany we wdrożenie (np. wszystkie środowiska przedprodukcyjne i produkcyjne), każdy komputer zaangażowany w tworzenie kopii zapasowej (np. CrashPlan, Time Machine itd.) itd.

Każdy fragment uruchamianego oprogramowania będzie miał dostęp do tych danych poufnych

Skoro dane poufne są przechowywane w plikach zwykłego tekstu na tak dużej liczbie dysków, to każdy fragment uruchamianego oprogramowania w dowolnym z tych komputerów może potencjalnie odczytywać wspomniane dane poufne.

Nie ma możliwości audytu ani odebrania dostępu do tych danych poufnych

Gdy dane poufne znajdują się w postaci zwykłego tekstu na setkach dysków, wówczas nie wiadomo, kto z nich korzysta (brak możliwości przeprowadzenia audytu dzienników zdarzeń), a także nie ma łatwego sposobu na odebranie dostępu do nich.

Ujmując rzecz najkrócej: jeśli dane poufne przechowujesz w postaci zwykłego tekstu, to osobom nieupoważnionym (np. hakerom, konkurencji, niezadowolonym byłym pracownikom) zapewniasz niezliczone możliwości w zakresie uzyskania dostępu do najbardziej wrażliwych danych firmy — np. przez złamanie systemu kontroli wersji lub dowolnego z komputerów używanych w firmie, a także przez złamanie dowolnego fragmentu używanego w nich oprogramowania. Ponadto nie będziesz o tym wiedzieć, a nawet jeśli się dowiesz, nie masz łatwej możliwości reakcji.

Dlatego też kluczowe znaczenie ma używanie właściwego *narzędzia przeznaczonego do zarządzania danymi poufnymi*, w którym będą przechowywane informacje wrażliwe.

Narzędzia przeznaczone do zarządzania danymi poufnymi

Dokładne omówienie wszystkich aspektów związanych z zarządzaniem danymi poufnymi wykracza poza zakres tematyczny książki. Jednak w celu umożliwienia użycia z Terraform narzędzi przeznaczonych do zarządzania danymi poufnymi warto przynajmniej pokrótce zapoznać się z tymi kwestiami:

- rodzaje przechowywanych danych poufnych,
- sposoby przechowywania danych poufnych,
- interfejs używany w celu uzyskania dostępu do danych poufnych,
- porównanie narzędzi przeznaczonych do zarządzania danymi poufnymi.

Rodzaje przechowywanych danych poufnych

Mamy trzy podstawowe typy danych poufnych: prywatne, klienta i infrastruktury.

Osobiste dane poufne

Należą do danej osoby. Przykładem mogą być nazwy użytkownika i hasła do odwiedzanych witryn internetowych, klucze SSH (Secure Shell) i PGP (Pretty Good Privacy).

Dane poufne klienta

Należą do klientów. Jeżeli uruchamiasz oprogramowanie przeznaczone dla innych pracowników firmy — np. zarządzasz wewnętrznym serwerem Active Directory w firmie — ci pracownicy są Twoimi klientami. Przykładem takich danych mogą być nazwy użytkowników i hasła używane przez klientów podczas logowania się do Twojego produktu, dane osobowe, dane dotyczące zdrowia klientów.

Dane poufne infrastruktury

Należą do Twojej infrastruktury. Przykładem mogą być hasło bazy danych, klucze API i certyfikaty TLS.

Większość narzędzi przeznaczonych do zarządzania danymi poufnymi została opracowana do przechowywania dokładnie jednego ich typu. Wprawdzie można spróbować wymusić przechowywanie także innych typów, ale to zwykle jest zły pomysł z perspektywy bezpieczeństwa i użyteczności. Przykładowo sposób przechowywania haseł dotyczących infrastruktury jest zupełnie inny niż w wypadku przechowywania danych poufnych użytkownika: w tym pierwszym przypadku zwykle jest używany algorytm szyfrowania, taki jak AES (ang. *advanced encryption standard*), prawdopodobnie z liczbą jednorazową, ponieważ musi być możliwość deszyfrowania danych poufnych i przywrócenia pierwotnego hasła. Z kolei w drugiej sytuacji zwykle jest stosowany algorytm haszowania (np. *bcrypt*) z ciągiem zaburzającym, ponieważ nie powinno być możliwości odzyskania pierwotnie zdefiniowanego hasła. Wykorzystanie błędnego podejścia może się okazać katastrofalne w skutkach, używaj więc właściwego narzędzia do danego zadania.

Przechowywanie danych poufnych

Istnieją dwie najczęściej stosowane strategie podczas przechowywania danych poufnych: magazyn danych bazujący na pliku lub też centralny magazyn danych.

W przypadku **magazynu danych bazującego na pliku** dane poufne są przechowywane w zaszyfrowanych plikach, które następnie trafiają do systemu kontroli wersji. Zasyfrowanie plików wymaga użycia klucza szyfrowania. Ten klucz sam jest przykładem informacji tajnych. W ten sposób powstaje pytanie: jak bezpiecznie przechowywać ten klucz? Nie można przechowywać go w postaci zwykłego tekstu i umieścić w systemie kontroli wersji, ponieważ wówczas szyfrowanie w ogóle nie ma sensu. Wprawdzie ten klucz można zasyfrować za pomocą drugiego klucza, ale to nie rozwiązuje problemu i nadal trzeba ustalić, jak bezpiecznie przechowywać ten drugi klucz.

Najczęściej stosowanym rozwiązaniem tego problemu jest przechowywanie klucza w tzw. **usłudze zarządzania kluczami** (ang. *key management service*, KMS) zapewnianej przez dostawcę chmury, np. AWS KMS, GCP KMS i Azure Key Vault. To rozwiązuje problem dzięki zaufaniu, że dostawca chmury jest w stanie bezpiecznie przechowywać informacje poufne i zarządzać dostępem do nich. Kolejna możliwość to użycie kluczy PGP. Każdy programista ma swój klucz PGP składający się z **klucza publicznego** i **klucza prywatnego**. Jeżeli dane poufne zasyfrujesz za pomocą jednego lub więcej kluczy publicznych, te dane będą mogły być odszyfrowane jedynie przez programistów posiadających klucze prywatne odpowiadające kluczom prywatnym użytym podczas szyfrowania. Z kolei klucze prywatne są chronione za pomocą hasła pamiętanego przez programistów albo przechowywane w osobistym menedżerze danych poufnych.

Scentralizowany magazyn danych poufnych to zwykle usługa sieciowa działająca w sieci, szyfrująca dane poufne i przechowująca je w magazynie danych, takim jak MySQL, PostgreSQL, DynamoDB itd. Aby zasyfrować te dane poufne, centralny magazyn danych poufnych wymaga klucza szyfrowania. Zwykle tym kluczem zarządza sama usługa albo polega na rozwiązaniu KMS oferowanym przez dostawcę chmury.

Interfejs używany w celu dostępu do danych poufnych

W przypadku większości narzędzi przeznaczonych do zarządzania danymi poufnymi dostęp można uzyskać za pomocą API, CLI i (lub) interfejsu użytkownika.

Praktycznie wszystkie centralne magazyny danych poufnych udostępniają API, którego można używać za pomocą żądań sieciowych, np. API REST używane przez HTTP. Takie API okazuje się wygodne, gdy kod wymaga programowej możliwości odczytywania danych poufnych. Przykładowo podczas uruchamiania aplikacji można wykonać żądanie API do centralnego magazynu danych poufnych w celu pobrania hasła bazy danych. Ponadto, jak zobaczysz w dalszej części rozdziału, można stworzyć kod Terraform, który w tle będzie używał API centralnego magazynu danych poufnych do pobierania informacji tajnych.

Wszystkie bazujące na pliku magazyny danych poufnych działają za pomocą **interfejsu powłoki** (ang. *command-line interface*, CLI). Wiele centralnych magazynów danych poufnych również oferuje narzędzia CLI, które w tle wykonują żądania API do usługi. Narzędzia CLI zapewniają programistom wygodny dostęp do danych poufnych (np. szyfrowanie pliku odbywa się za pomocą kilku poleceń) i podczas tworzenia skryptów (np. przeznaczonego do szyfrowania danych poufnych).

Część centralnych magazynów danych poufnych udostępnia również **interfejs użytkownika** (ang. *user interface*, UI) w postaci przeglądarki WWW i aplikacji, zarówno tradycyjnej, jak i mobilnej. Dla każdego członka zespołu to potencjalnie jeszcze wygodniejsze rozwiązanie w zakresie uzyskania dostępu do danych poufnych.

Porównanie narzędzi przeznaczonych do zarządzania danymi poufnymi

W tabeli 6.1 przedstawiłem porównanie wybranych narzędzi przeznaczonych do zarządzania danymi poufnymi, podzielonych według trzech kryteriów omówionych w poprzednich sekcjach.

Tabela 6.1. Porównanie wybranych narzędzi przeznaczonych do zarządzania danymi poufnymi

	Typ danych poufnych	Magazyn danych poufnych	Interfejs danych poufnych
HashiCorp Vault	dane poufne infrastruktury ^a	usługa centralna	UI, API, CLI
AWS Secrets Manager	dane poufne infrastruktury	usługa centralna	UI, API, CLI
Google Secrets Manager	dane poufne infrastruktury	usługa centralna	UI, API, CLI
Azure Key Vault	dane poufne infrastruktury	usługa centralna	UI, API, CLI
Confidant	dane poufne infrastruktury	usługa centralna	UI, API, CLI
Keywhiz	dane poufne infrastruktury	usługa centralna	API, CLI
sops	dane poufne infrastruktury	pliki	CLI
git-secret	dane poufne infrastruktury	pliki	CLI
1Password	osobiste dane poufne	usługa centralna	UI, API, CLI
LastPass	osobiste dane poufne	usługa centralna	UI, API, CLI
Bitwarden	osobiste dane poufne	usługa centralna	UI, API, CLI
KeePass	osobiste dane poufne	pliki	UI, CLI
Keychain (macOS)	osobiste dane poufne	pliki	UI, CLI
Credential Manager (Windows)	osobiste dane poufne	pliki	UI, CLI
pass	osobiste dane poufne	pliki	CLI
Active Directory	dane poufne klienta	usługa centralna	UI, API, CLI

Tabela 6.1. Porównanie wybranych narzędzi przeznaczonych do zarządzania danymi poufnymi (ciąg dalszy)

	Typ danych poufnych	Magazyn danych poufnych	Interfejs danych poufnych
Autho	dane poufne klienta	usługa centralna	UI, API, CLI
Okta	dane poufne klienta	usługa centralna	UI, API, CLI
OneLogin	dane poufne klienta	usługa centralna	UI, API, CLI
Ping	dane poufne klienta	usługa centralna	UI, API, CLI
AWS Cognito	dane poufne klienta	usługa centralna	UI, API, CLI

^a HashiCorp Vault obsługuje wiele *magazynów danych poufnych*, z których większość została zaprojektowana do przechowywania danych poufnych infrastruktury. Kilka zapewnia również obsługę danych poufnych klienta.

Skoro to książka dotycząca Terraform, od tego miejsca będę się koncentrować przede wszystkim na narzędziach przeznaczonych do zarządzania danymi poufnymi infrastruktury, dostępnych za pomocą API lub CLI. (Od czasu do czasu wspomnę również o narzędziach przeznaczonych do zarządzania osobistymi danymi poufnymi, ponieważ często zawierają one informacje tajne niezbędne do uwierzytelnienia w narzędziach przechowujących dane poufne infrastruktury).

Narzędzia przeznaczone do zarządzania danymi poufnymi w Terraform

Przejdę teraz do używania z Terraform narzędzi przeznaczonych do zarządzania danymi poufnymi. Omówię trzy miejsca, w których kod Terraform może mieć styczność z danymi poufnymi:

- dostawcy,
- zasoby i źródła danych,
- pliki informacji o stanie i pliki planu.

Dostawcy

Podczas pracy z kodem Terraform pierwsze miejsce, w którym pojawia się konieczność pracy z danymi poufnymi, zwykle ma związek z uwierzytelnieniem dostawcy. Chcesz np. zastosować polecenie `terraform apply` dla kodu używającego dostawcy AWS. W takim wypadku musisz najpierw się uwierzytelnić w AWS, co zwykle oznacza użycie kluczy dostępu, które zaliczają się do danych poufnych. W jaki sposób można przechowywać te dane? I jak je udostępnić dla kodu Terraform?

Istnieje wiele potencjalnych odpowiedzi na te pytania. Na pewno sposobem, którego *nie* należy stosować, jest umieszczanie danych poufnych bezpośrednio w kodzie, czyli w postaci zwykłego tekstu (z takim rozwiązaniem można się czasami spotkać w dokumentacji Terraform):

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

  # NIGDY TEGO NIE RÓB!!!
  access_key = "(KLUCZ_DOSTĘPU)"
```

```
secret_key = "(KLUCZ_TAJNY)"  
# NIGDY TEGO NIE RÓB!!!  
}
```

Przechowywanie danych uwierzytelniających w taki sposób, w postaci zwykłego tekstu, *nie* jest bezpieczne, jak to już wyjaśniłem we wcześniejszej części rozdziału. Co więcej, to podejście okazuje się również niepraktyczne, ponieważ umieszczenie na stałe danych uwierzytelniających oznacza używanie tego samego zestawu tych danych dla wszystkich użytkowników modułu. W większości przypadków oczekujesz używania odmiennych danych uwierzytelniających w poszczególnych komputerach (np. gdy różni programiści w serwerze CI wydają polecenie `terraform apply`) i środowiskach (programistycznych, roboczych, produkcyjnych).

Mamy wiele technik uznawanych za znacznie bezpieczniejsze w zakresie przechowywania danych uwierzytelniających i dostarczania ich dostawcom Terraform. Zapoznaj się z tymi technikami, pogrupowanymi według użytkowników korzystających z Terraform.

Użytkownicy

To są programiści używający Terraform w swoich komputerach.

Urządzenia

To są systemy zautomatyzowane (np. serwer CI) używające Terraform, gdy w pobliżu nie ma użytkownika człowieka.

Użytkownicy

Praktycznie każdy dostawca Terraform pozwala na podanie danych uwierzytelniających, w taki czy inny sposób, zamiast bezpośredniego umieszczania ich w kodzie. Najczęstszym rozwiązaniem jest stosowanie zmiennych środowiskowych. Zobacz, jak można używać zmiennych środowiskowych do uwierzytelniania w AWS:

```
$ export AWS_ACCESS_KEY_ID=(IDENTYFIKATOR_KLUCZA_DOSTĘPU)  
$ export AWS_SECRET_ACCESS_KEY=(TWÓJ_TAJNY_KLUCZ_DOSTĘPU)
```

Zdefiniowanie danych uwierzytelniających jako zmiennych środowiskowych pozwala uniknąć przechowywania tych danych w postaci zwykłego tekstu i gwarantuje, że każdy programista wykonujący kod Terraform musi dostarczyć własne dane uwierzytelniające. Ponadto mamy pewność, że dane uwierzytelniające są przechowywane jedynie w pamięci, a nie na dysku¹.

Ważne pytanie, które może zrodzić się w tej chwili, dotyczy miejsca przechowywania identyfikatora klucza dostępu i tajnego klucza dostępu. Te wartości są zbyt duże i losowe, aby można było je zapamiętać. Z kolei, jeśli umieścisz je w komputerze w postaci zwykłego tekstu, narażasz te dane na niebezpieczeństwo. Skoro ten podpunkt dotyczy użytkowników programistów, rozwiązaniem będzie przechowywanie kluczy dostępu (i innych danych poufnych) w menedżerze danych poufnych

¹ Trzeba pamiętać, że w większości powłok w systemach Linux/UNIX/macOS każde wydawane polecenie jest przechowywane na dysku w postaci pewnego rodzaju pliku historii (np. `~/.bash_history`). Dlatego też przedstawione w tym miejscu polecenia `export` mają na początku spację, która w wielu powłokach oznacza, że dane polecenie nie zostanie zapisane w pliku historii. Być może konieczne będzie przypisanie zmiennej środowiskowej `HISTCONTROL` wartości `ignoreboth`, aby zapewnić takie zachowanie powłoki, o ile domyślnie nie działa ona w taki właśnie sposób.

opracowanym dla tego rodzaju informacji osobistych. Przykładowo klucze dostępu można umieścić w menedżerze 1Password lub LastPass, a następnie kopiować i wklejać w poleceniach export wydawanych w powłoce.

Jeżeli dane uwierzytelniające są często używane w powłoce, znacznie wygodniejszym rozwiązaniem będzie użycie menedżera danych poufnych zapewniającego obsługę interfejsu CLI. Przykładowo 1Password oferuje takie narzędzie w postaci polecenia o nazwie `op`. W systemach macOS i Linux polecenia `op` można użyć do uwierzytelnienia 1Password w powłoce, np.:

```
$ eval $(op signin my)
```

Jeżeli w aplikacji 1Password przechowujesz klucze dostępu w elemencie `aws-dev` o polach `id` i `secret`, to po uwierzytelnieniu możesz za pomocą polecenia `op` zastosować ten klucz do zdefiniowania zmiennych środowiskowych:

```
$ export AWS_ACCESS_KEY_ID=$(op get item 'aws-dev' --fields 'id')
$ export AWS_SECRET_ACCESS_KEY=$(op get item 'aws-dev' --fields 'secret')
```

Wprawdzie narzędzia takie jak 1Password i `op` sprawdzają się doskonale do zarządzania ogólnego przeznaczenia danymi poufnymi, ale dla pewnych dostawców istnieją dedykowane im narzędzia CLI, które znacznie ułatwiają pracę. Przykładowo w celu uwierzytelnienia w AWS można użyć narzędzia typu open source o nazwie `aws-vault`. W kolejnym fragmencie kodu pokazałem, jak można zapisać klucze dostępu za pomocą polecenia `aws-vault` w *profilu* o nazwie `dev`.

```
$ aws-vault add dev
Enter Access Key Id: (IDENTYFIKATOR_KLUCZA_DOSTĘPU)
Enter Secret Key: (TWÓJ_TAJNY_KLUCZ_DOSTĘPU)
```

W tle `aws-vault` będzie przechowywać te dane uwierzytelniające bezpiecznie w natywnym menedżerze haseł systemu operacyjnego (np. Keychain w macOS, Credential Manager w Windows). Po zapisaniu tych danych uwierzytelniających uwierzytelnienie w AWS dla dowolnego polecenia powłoki można przeprowadzić następująco:

```
$ aws-vault exec <PROFIL> -- <POLECENIE>
```

gdzie `PROFIL` to nazwa profilu utworzonego wcześniej za pomocą polecenia `add` (np. `dev`), a `POLECENIE` to polecenie przeznaczone do wykonania. Zobacz dla przykładu, jak można użyć zapisanych wcześniej danych uwierzytelniających `dev` do wydania polecenia `terraform apply`.

```
$ aws-vault exec dev -- terraform apply
```

Polecenie `exec` automatycznie używa AWS STS do pobrania tymczasowych danych uwierzytelniających i w postaci zmiennych środowiskowych udostępnia je wykonywanemu poleceniu (tutaj to `terraform apply`). Dzięki temu trwale nie tylko dane uwierzytelniające są bezpiecznie przechowywane (w natywnym menedżerze haseł systemu operacyjnego), ale również dowolnemu wykonywanemu procesowi są udostępniane tymczasowe dane uwierzytelniające, niebezpieczeństwo ich wycieku więc zostało ograniczone do minimum. `aws-vault` zapewnia natywną obsługę ról IAM, użycia uwierzytelniania wielopoziomowego (ang. *multifactor authentication*, MFA), logowania w kontach konsoli internetowych itd.

Urządzenia

Podczas gdy użytkownik człowiek może polegać na zapamiętaniu hasła, to co można zrobić, gdy takiego użytkownika nie ma? Jeśli np. przygotowujesz rozwiązanie ciągłej integracji i ciągłego wdrażania (CI/CD) w celu automatycznego wykonywania kodu Terraform, to w jaki sposób możesz uwierzytelnić taki potok? W takiej sytuacji masz do czynienia z uwierzytelnianiem dla *urządzenia*. Pojawia się wówczas pytanie: jak jedno urządzenie (np. serwer CI) może uwierzytelnić inne (np. serwery API AWS) bez konieczności przechowywania jakichkolwiek danych poufnych w postaci zwykłego tekstu?

Rozwiązanie w dużej mierze zależy od używanych urządzeń, czyli z jakiego odbywa się uwierzytelnienie i w jakim urządzeniu jest uwierzytelnianie to pierwsze. Przeanalizujmy trzy przykłady:

- CircleCI jako serwer CI, z przechowywanymi danymi poufnymi,
- egzemplarz EC2 z uruchomionym serwerem CI Jenkins, z rolami IAM,
- GitHub Actions jako serwer CI, z tożsamością OIDC.



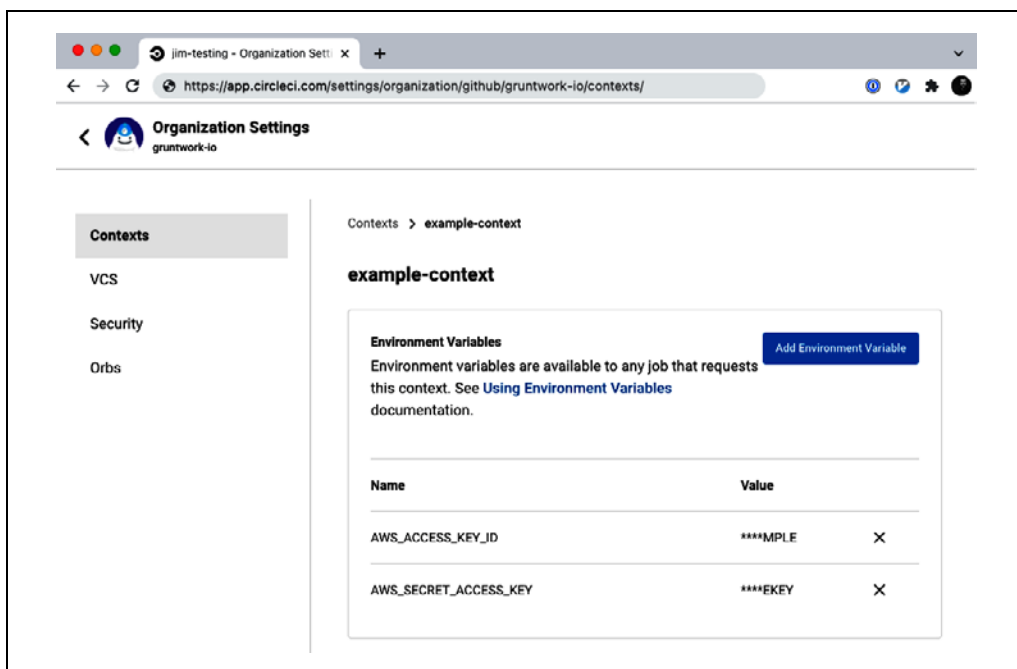
Ostrzeżenie: to są przykłady uproszczone

Przykłady zamieszczone w tym podpunkcie pokazują, jak zapewnić pełną obsługę uwierzytelniania dostawcy w kontekście potoku CI/CD, przy czym wszystkie pozostałe aspekty potoku CI/CD zostały bardzo uproszczone. Pełniejsze, opracowane od początku do końca i przygotowane do użycia w produkcji, przykłady potoków CD/CI znajdziesz w rozdziale 9.

CircleCI jako serwer CI, z przechowywanymi danymi poufnymi. Wyobraź sobie użycie CircleCI, czyli popularnej platformy zarządzanego CI/CD, do wykonywania kodu Terraform. Dzięki CircleCI można skonfigurować kroki kompilacji w pliku `.circleci/config.yml`, w którym definiuje się zadanie do wykonania polecenia `terraform apply`. Takie zadanie może przedstawiać się tak, jak pokazałem w kolejnym fragmencie kodu.

```
version: '2.1'
orbs:
  # Instalacja Terraform za pomocą CircleCi Orb.
  terraform: circleci/terraform@1.1.0
jobs:
  # Zdefiniowanie zadania przeznaczonego do wykonania 'terraform apply'.
  terraform_apply:
    executor: terraform/default
    steps:
      - checkout          # Sklonowanie repozytorium git z kodem.
      - terraform/init    # Wykonanie polecenia 'terraform init'.
      - terraform/apply   # Wykonanie polecenia 'terraform apply'.
workflows:
  # Utworzenie przepływu przeznaczonego do wykonania zdefiniowanego tutaj zadania 'terraform apply'.
  deploy:
    jobs:
      - terraform_apply
  # Wykonanie tego zadania jedynie podczas przekazywania kodu do gałęzi main.
  filters:
    branches:
      only:
        - main
```

W wypadku narzędzia takiego jak CircleCI sposobem na uwierzytelnienie dostawcy jest utworzenie w nim użytkownika urządzenia (przeznaczonego jedynie na potrzeby automatyzacji, a nie używania przez człowieka), przechowywanie danych uwierzytelniających dla tego urządzenia w tym, co w CircleCI jest określane mianem *kontekstu CircleCI*, a po rozpoczęciu kompilacji CircleCI udostępni dane uwierzytelniające w tym kontekście jako zmienne środowiskowe. Jeśli np. kod Terraform musi uwierzytelnić się w AWS, można w AWS utworzyć nowego użytkownika IAM, nadać mu uprawnienia niezbędne do wdrażania zmian Terraform, a następnie ręcznie skopiować do kontekstu CircleCI klucze dostępu tego użytkownika, jak pokazałem na rysunku 6.1.



Rysunek 6.1. Kontekst CircleCI z danymi uwierzytelniającymi AWS

Pozostało jeszcze uaktualnienie bloku workflows w pliku `.circleci/config.yml`, aby kontekst CircleCI wykorzystać za pomocą parametru `context`.

```
workflows:
  # Utworzenie przepływu przeznaczonego do wykonania zdefiniowanego tutaj zadania 'terraform apply'.
  deploy:
    jobs:
      - terraform_apply
    # Wykonanie tego zadania jedynie podczas przekazywania kodu do gałęzi main.
    filters:
      branches:
        only:
          - main
    # Udostępnienie danych poufnych w kontekście CircleCI za pomocą zmiennych środowiskowych.
    context:
      - example-context
```

Po rozpoczęciu kompilacji CircleCI automatycznie udostępni dane poufne w tym kontekście jako zmienne środowiskowe — w omawianym przykładzie będą to `AWS_ACCESS_KEY_ID` i `AWS_SECRET_ACCESS_KEY` — a polecenie `terraform apply` automatycznie użyje tych zmiennych w celu uwierzytelnienia u dostawcy.

Największe wady tego podejścia to (a) konieczność ręcznego zarządzania danymi uwierzytelniającymi i (b) konieczność użycia trwałych danych uwierzytelniających, które, raz zapisane w CircleCI, będą rzadko (o ile w ogóle) zmieniane. Dwa kolejne przykłady pokazują alternatywne podejścia.

Egzemplarz EC2 z uruchomionym serwerem CI Jenkins, z rolami IAM. Jeżeli używasz egzemplarza EC2 do uruchamiania kodu Terraform — np. masz serwer Jenkins działający w EC2 w charakterze serwera CI — zalecanym rozwiązaniem jest nadanie temu egzemplarzowi roli IAM przeznaczonej do uwierzytelniania. Rola IAM jest podobna do użytkownika IAM pod tym względem, że stanowi encję w AWS, której można nadać uprawnienia IAM. Jednak w przeciwieństwie do użytkownika IAM roli IAM nie można powiązać z żadną osobą i nie może ona mieć trwałych danych uwierzytelniających (hasła czy kluczy dostępu). Zamiast tego rola może być *przyjęta* przez inne encje IAM: np. użytkownik IAM może przyjąć rolę tymczasowo w celu uzyskania dostępu do innych uprawnień niż te, którymi zwykle dysponuje. Wiele usług AWS, takich jak egzemplarze EC2, może przyjmować role IAM w celu nadawania tym usługom uprawnień w ramach konta AWS.

To, co przedstawia kolejny fragment kodu, możesz spotkać wielokrotnie podczas wdrażania egzemplarza EC2:

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

W celu utworzenia roli IAM trzeba zacząć od zdefiniowania **polityki przyjęcia roli**, czyli polityki określającej, kto może przyjąć tę rolę IAM. Wprawdzie politykę IAM można zapisać w postaci czystego kodu JSON, ale Terraform ma wygodne źródło danych `aws_iam_policy_document` pozwalające na utworzenie tego kodu JSON. Spójrz na przykład użycia `aws_iam_policy_document` do zdefiniowania polityki przyjęcia roli pozwalającej usłudze EC2 na przyjęcie roli IAM.

```
data "aws_iam_policy_document" "assume_role" {
  statement {
    effect = "Allow"
    actions = ["sts:AssumeRole"]

    principals {
      type        = "Service"
      identifiers = ["ec2.amazonaws.com"]
    }
  }
}
```

Teraz można użyć zasobu `aws_iam_role` do utworzenia roli IAM i przekazania jej do kodu JSON w `aws_iam_policy_document`, aby wykorzystać jako politykę przyjęcia roli.

```
resource "aws_iam_role" "instance" {
  name_prefix   = var.name
  assume_role_policy = data.aws_iam_policy_document.assume_role.json
}
```

W ten sposób masz rolę IAM, ale domyślnie role IAM nie nadają żadnych uprawnień. Dlatego też następnym krokiem jest przypisanie jednej lub więcej polityk IAM do roli IAM określającej faktyczne możliwości użytkownika po przyjęciu tej roli. Załóżmy, że wdrożony w EC2 serwer Jenkins jest używany do wykonywania kodu Terraform. Źródło danych `aws_iam_policy_document` można wykorzystać do zdefiniowania polityki IAM, zgodnie z którą użytkownik otrzyma uprawnienia administratora dla egzemplarzy EC2:

```
data "aws_iam_policy_document" "ec2_admin_permissions" {
  statement {
    effect    = "Allow"
    actions   = ["ec2:*"]
    resources = ["*"]
  }
}
```

Tę politykę można przypisać do roli IAM za pomocą zasobu `aws_iam_role_policy`.

```
resource "aws_iam_role_policy" "example" {
  role = aws_iam_role.instance.id
  policy = data.aws_iam_policy_document.ec2_admin_permissions.json
}
```

Ostatnim krokiem jest umożliwienie egzemplarzowi EC2 automatycznego przyjmowania roli IAM przez utworzenie **profilu egzemplarza**.

```
resource "aws_iam_instance_profile" "instance" {
  role = aws_iam_role.instance.name
}
```

Następnie egzemplarzowi EC2 nakazujesz użycie tego profilu za pomocą parametru `iam_instance_profile`.

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"

  # Dołączenie profilu egzemplarza.
  iam_instance_profile = aws_iam_instance_profile.instance.name
}
```

W tle AWS uruchamia **punkt końcowy metadanych egzemplarza** w każdym egzemplarzu EC2 pod adresem `http://169.254.169.254`. To jest punkt końcowy, do którego dostęp może uzyskać jedynie proces uruchomiony w samym egzemplarzu — te procesy używają podanego punktu końcowego w celu pobrania metadanych dotyczących egzemplarza. Jeśli np. nawiążesz połączenie SSH z egzemplarzem EC2, to za pomocą polecenia `curl` możesz wykonać żądanie do podanego punktu końcowego.

```
$ ssh ubuntu@<IP_EGZEMPLARZA>
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.11.0-1022-aws x86_64)
(...)

$ curl http://169.254.169.254/latest/meta-data/
ami-id
ami-launch-index
ami-manifest-path
block-device-mapping/
```

```
events/  
hibernation/  
hostname  
identity-credentials/  
(...)
```

Jeżeli egzemplarz ma przypisaną rolę IAM (za pomocą profilu egzemplarza), to metadane będą zawierały dane uwierzytelniające AWS, które mogą być wykorzystane do uwierzytelnienia w AWS i przyjęcia roli IAM. Dowolne narzędzie używające AWS SDK, np. Terraform, wie, jak automatycznie korzystać z danych uwierzytelniających tego punktu końcowego metadanych egzemplarza. Dlatego tuż po zastosowaniu polecenia `terraform apply` w egzemplarzu EC2 z tą rolą IAM kod Terraform uwierzytelnia się jako ta rola IAM, a tym samym kod otrzyma uprawnienia administratora EC2 niezbędne do zakończonego sukcesem wykonania kodu².

W wypadku dowolnego zautomatyzowanego procesu działającego w AWS, np. serwera CI, (a) rola IAM zapewnia możliwość uwierzytelnienia bez konieczności ręcznego zarządzania danymi uwierzytelniającymi i (b) dane uwierzytelniające dostarczane AWS przez punkt końcowy metadanych egzemplarza zawsze są tymczasowe i automatycznie rotowane. To są dwie ogromne korzyści względem trwałych danych uwierzytelniających ręcznie zarządzanych za pomocą narzędzia takiego jak CircleCI, działającego poza kontem AWS. Jak jednak zobaczysz w następnym przykładzie, w niektórych przypadkach istnieje możliwość uzyskania tych samych korzyści także podczas używania narzędzi zewnętrznych.

GitHub Actions jako serwer CI, z tożsamością OIDC. To jest inna, popularna i zarządzana platforma CI/CD, której można użyć do wykonywania kodu Terraform. W przeszłości GitHub Actions wymagała ręcznego kopiowania danych uwierzytelniających, podobnie jak w wypadku CircleCI. Jednak od 2021 roku GitHub Actions oferuje lepszą alternatywę: *OIDC* (Open ID Connect). Korzystając z *OIDC*, można utworzyć zaufane połączenie między systemem CI i dostawcą chmury (GitHub Actions obsługuje AWS, Azure i Google Cloud), aby system CI mógł uwierzytelnić się u tych dostawców bez konieczności ręcznego zarządzania jakimikolwiek danymi uwierzytelniającymi.

Sposób działania GitHub Actions można zdefiniować w plikach YAML umieszczonych w katalogu `.github/workflows`, np. takim jak przedstawiony tutaj `terraform.yml`.

```
name: Terraform Apply  
# Wykonanie tego zadania jedynie podczas przekazywania kodu do gałęzi main.  
on:  
  push:  
    branches:
```

² Domyślnie punkt końcowy metadanych egzemplarza jest otwarty dla wszystkich użytkowników systemu operacyjnego korzystających z egzemplarza EC2. Zalecam do zabezpieczenia tego punktu końcowego, aby tylko określone użytkownicy systemu operacyjnego mieli do niego dostęp: np. jeśli w egzemplarzu EC2 uruchamiasz aplikację jako użytkownik `app`, możesz użyć `iptables` lub `nftables` w celu umożliwienia tylko temu użytkownikowi dostępu do punktu końcowego metadanych egzemplarza. W ten sposób, jeśli atakujący znajdzie lukę w zabezpieczeniach i uzyska możliwość wykonania kodu w egzemplarzu, będzie miał do dyspozycji uprawnienia roli IAM, jak po uwierzytelnieniu jako użytkownik `app` (a nie dowolny użytkownik systemu). Jeszcze lepiej, jeśli uprawnienia roli IAM są potrzebne jedynie podczas rozruchu (np. w celu odczytania hasła bazy danych), wówczas można całkowicie wyłączyć punkt końcowy metadanych egzemplarza po rozruchu, aby atakujący, który później uzyska dostęp, nie mógł w ogóle skorzystać z tego punktu końcowego.

```

- 'main'
jobs:
  TerraformApply:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      # Wykonanie kodu Terraform za pomocą akcji setup-terraform firmy HashiCorp.
      - uses: hashicorp/setup-terraform@v1
        with:
          terraform_version: 1.1.0
          terraform_wrapper: false
        run: |
          terraform init
          terraform apply -auto-approve

```

Jeżeli kod Terraform komunikuje się z dostawcą takim jak AWS, konieczne jest zapewnienie sposobu pozwalającego na uwierzytelnienie się u tego dostawcy. Aby to zrobić za pomocą OIDC³, pierwszym krokiem jest *utworzenie dostawcy tożsamości IAM OIDC* w koncie AWS — z wykorzystaniem do tego zasobu `aws_iam_openid_connect_provider` — i skonfigurowanie go, aby ufał odciskowi palca GitHub Actions pobranemu za pomocą źródła danych `tls_certificate`.

```

# Utworzenie dostawcy tożsamości IAM OIDC ufającego GitHub.
resource "aws_iam_openid_connect_provider" "github_actions" {
  url = "https://token.actions.githubusercontent.com"
  client_id_list = ["sts.amazonaws.com"]
  thumbprint_list = [
    data.tls_certificate.github.certificates[0].sha1_fingerprint
  ]
}

# Pobranie odcisku palca GitHub OIDC.
data "tls_certificate" "github" {
  url = "https://token.actions.githubusercontent.com"
}

```

W tym momencie można już utworzyć rolę IAM dokładnie tak jak wcześniej — np. rolę IAM z dołączonymi uprawnieniami administracyjnymi EC2 — przy czym polityka przyjęcia roli będzie dla tych ról IAM wyglądała nieco inaczej.

```

data "aws_iam_policy_document" "assume_role_policy" {
  statement {
    actions = ["sts:AssumeRoleWithWebIdentity"]
    effect = "Allow"

    principals {
      identifiers = [aws_iam_openid_connect_provider.github_actions.arn]
      type = "Federated"
    }
  }

  condition {
    test = "StringEquals"
  }
}

```

³ W trakcie powstawania książki obsługa OIDC między GitHub Actions i AWS była względną nowością i jej szczegóły ulegały zmianie. Najnowsze informacje na ten temat znajdziesz w dokumentacji GitHub OIDC (<https://docs.github.com/en/actions/deployment/security-hardening-your-deployments/configuring-openid-connect-in-amazon-web-services>).

```

variable = "token.actions.githubusercontent.com:sub"
# Repozytoria i gałęzie zdefiniowane w var.allowed_repos_branches
# będą dostępne podczas przyjmowania tej roli IAM.
values = [
  for a in var.allowed_repos_branches :
    "repo:${a["org"]}/${a["repo"]}:ref:refs/heads/${a["branch"]}"
]
}
}
}

```

Ta polityka pozwala dostawcy tożsamości IAM OIDC na przyjęcie roli za pomocą uwierzytelnienia federacyjnego. Zwróć uwagę na blok `condition` gwarantujący, że tylko konkretne repozytoria i gałęzie wymienione za pomocą zmiennej danych wejściowych `allowed_repos_branches` będą mogły przyjmować tę rolę IAM.

```

variable "allowed_repos_branches" {
  description = "Repozytoria i gałęzie GitHub, które mogą przyjmować tę rolę IAM"
  type = list(object({
    org    = string
    repo   = string
    branch = string
  }))
# Przykład:
# allowed_repos_branches = [
# {
#   org    = "brikis98"
#   repo   = "terraform-up-and-running-code"
#   branch = "main"
# }
# ]
}

```

Bardzo duże znaczenie ma zagwarantowanie, że przypadkowo nie pozwolisz, aby *wszystkie* repozytoria GitHub były uwierzytelnione w Twoim koncie AWS. Teraz w GitHub Actions możesz skonfigurować kompilację przyjmującą tę rolę IAM. Przede wszystkim na początku kompilacji nadaj uprawnienia `id-token: write`.

```

permissions:
  id-token: write

```

Następnie dodaj krok kompilacji tuż przed wykonaniem kodu Terraform w celu uwierzytelnienia w AWS za pomocą akcji `configure-aws-credentials`.

```

# Uwierzytelnienie w AWS za pomocą tożsamości OIDC.
- uses: aws-actions/configure-aws-credentials@v1
  with:
    # Określenie roli IAM do przyjęcia.
    role-to-assume: arn:aws:iam::123456789012:role/example-role
    aws-region: us-east-2

# Wykonanie Terraform za pomocą akcji HashiCorp setup-terraform.
- uses: hashicorp/setup-terraform@v1
  with:
    terraform_version: 1.1.0
    terraform_wrapper: false

```



```
run: |
    terraform init
    terraform apply -auto-approve
```

Po uruchomieniu tej kompilacji w jednym z repozytoriów i jednej z gałęzi wymienionych w zmiennej `allowed_repos_branches` GitHub automatycznie będzie w stanie przyjąć rolę IAM przy użyciu tymczasowych danych uwierzytelniających, a Terraform przeprowadzi uwierzytelnienie w AWS za pomocą tej roli. To wszystko odbędzie się bez konieczności ręcznego zarządzania jakimikolwiek danymi uwierzytelniającymi.

Zasoby i źródła danych

Następnym miejscem w kodzie Terraform, gdzie mogą pojawiać się dane poufne, są zasoby i źródła danych. Przykładowo we wcześniejszej części rozdziału pokazałem przekazywanie danych uwierzytelniających bazy danych do zasobu `aws_db_instance`.

```
resource "aws_db_instance" "example" {
    identifier_prefix = "terraform-up-and-running"
    engine           = "mysql"
    allocated_storage = 10
    instance_class   = "db.t2.micro"
    skip_final_snapshot = true
    db_name           = var.db_name

    # NIGDY TEGO NIE RÓB!!!
    username = "admin"
    password = "password"
    # NIGDY TEGO NIE RÓB!!!
}
```

Wprawdzie w rozdziale wspomniałem już o tym wielokrotnie, ale warto to powtórzyć: przechowywanie tych danych uwierzytelniających w kodzie, w postaci zwykłego tekstu, jest złym pomysłem. Czy istnieje lepsze rozwiązanie?

Mamy trzy główne techniki, których można użyć:

- zmienne środowiskowe,
- szyfrowane pliki,
- magazyny danych poufnych.

Zmienne środowiskowe

Pierwsza technika, przedstawiona już wcześniej w rozdziale 3., a także we wcześniejszej części rozdziału podczas analizy dostawców, polega na uniknięciu przechowywania danych poufnych w postaci zwykłego tekstu w kodzie dzięki wykorzystaniu oferowanej przez Terraform natywnej obsługi odczytu zmiennych środowiskowych.

Aby zastosować tę technikę, należy zadeklarować zmienne dla danych poufnych, które mają być przekazywane.

```
variable "db_username" {
    description = "Nazwa użytkownika bazy danych"
```

```

type          = string
sensitive     = true
}

variable "db_password" {
  description = "Hasło użytkownika bazy danych"
  type        = string
  sensitive   = true
}

```

Podobnie jak w rozdziale 3., także tutaj te zmienne są oznaczone za pomocą `sensitive = true`, aby wskazać, że przechowują dane poufne. (Terraform nie będzie zapisywał w dziennikach zdarzeń tych wartości podczas wykonywania poleceń `terraform plan` lub `terraform apply`). Ponadto te zmienne nie mają właściwości `default` (dane poufne nie są przechowywane w postaci zwykłego tekstu). Następnym krokiem jest przekazanie zmiennych do zasobów Terraform, które ich wymagają.

```

resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine            = "mysql"
  allocated_storage = 10
  instance_class    = "db.t2.micro"
  skip_final_snapshot = true
  db_name            = var.db_name

  # Przekazanie danych poufnych do zasobu.
  username = var.db_username
  password = var.db_password
}

```

Teraz można przekazać wartość dla każdej zmiennej `foo` przez zdefiniowanie zmiennej środowiskowej `TF_VAR_foo`.

```

$ export TF_VAR_db_username=(NAZWA_UŻYTKOWNIKA_BAZY_DANYCH)
$ export TF_VAR_db_password=(HASŁO_UŻYTKOWNIKA_BAZY_DANYCH)

```

Wprawdzie przekazywanie danych uwierzytelniających za pomocą zmiennych środowiskowych pomoże w uniknięciu przechowywania tych danych w postaci zwykłego tekstu w kodzie, ale nie dostarcza odpowiedzi na ważne pytanie: jak można bezpiecznie przechowywać dane poufne? Użyteczną cechą używania zmiennych środowiskowych jest to, że współdziałają one praktycznie z dowolnym rodzajem rozwiązania w zakresie zarządzania danymi poufnymi. Jedną z możliwości jest ich przechowywanie w osobistym menedżerze informacji tajnych (takim jak `1Password`) i ręczne przypisywanie zmiennym środowiskowym w powłoce. Inną możliwością jest przechowywanie danych poufnych w centralnym magazynie danych poufnych (takim jak `HasiCorp Vault`) i tworzenie skryptów używających API lub CLI tego magazynu danych poufnych podczas odczytywania informacji tajnych i ich przypisywania zmiennym środowiskowym.

Używanie zmiennych środowiskowych ma takie zalety:

- Dane poufne w postaci zwykłego tekstu nie będą się znajdowały ani w kodzie, ani w systemie kontroli wersji.
- Przechowywanie danych poufnych jest łatwe, o ile można użyć innego dowolnego rozwiązania w zakresie zarządzania danymi poufnymi. Jeżeli firma stosuje już takie rozwiązanie, z reguły udaje się znaleźć sposób, aby współdziałało ono ze zmiennymi środowiskowymi.

- Pobieranie danych poufnych jest łatwe, podobnie jak odczytywanie zmiennych środowiskowych stanowi proste zadanie w praktycznie każdym języku programowania.
- Integracja z testami zautomatyzowanymi jest prosta, ponieważ zmiennym środowiskowym bardzo łatwo można przypisać wartości imitujące.
- Używanie zmiennych środowiskowych nie wiąże się z żadnym kosztem, w przeciwieństwie do innych, omówionych w dalszej części rozdziału rozwiązań w zakresie zarządzania danymi poufnymi.

Używanie zmiennych środowiskowych ma również pewne wady:

- Nie wszystko zostaje zdefiniowane w samym kodzie Terraform. To powoduje, że zrozumienie sposobu działania takiego kodu i jego późniejsza obsługa stają się trudniejsze. Każdy używający tego kodu musi znać kroki dodatkowe konieczne do wykonania w celu ręcznego zdefiniowania zmiennych środowiskowych albo uruchomienia skryptu opakowującego.
- Standaryzacja praktyk dotyczących zarządzania danymi poufnymi staje się trudniejsza. Skoro zarządzanie danymi poufnymi odbywa się poza Terraform, kod nie wymusza żadnych właściwości bezpieczeństwa i wciąż istnieje możliwość, że ktoś będzie zarządzał tymi danymi w sposób niezapewniający im bezpieczeństwa (np. będą przechowywane w postaci zwykłego tekstu).
- Skoro dane poufne nie są wersjonowane, pakowane i testowane razem z kodem, zwiększa się prawdopodobieństwo występowania błędów, takich jak dodawanie danych poufnych w jednym środowisku (np. roboczym) i pominięcie ich dodania w innym (np. produkcyjnym).

Szyfrowane pliki

Druga technika polega na szyfrowaniu danych poufnych, przechowywaniu szyfrogramu w pliku i umieszczaniu go w systemie kontroli wersji.

Aby zaszyfrować dowolne dane, np. pewne dane poufne znajdujące się w pliku, potrzebny jest klucz szyfrowania. Jak już wspomniałem we wcześniejszej części rozdziału, ten klucz szyfrowania sam stanowi przykład danych poufnych, trzeba go więc bezpiecznie przechowywać. Typowe rozwiązanie polega na wykorzystaniu KMS dostawcy chmury (np. AWS KMS, Google KMS, Azure Key Vault) lub też kluczy PGP jednego lub więcej programistów zespołu.

Spójrz na przykład z wykorzystaniem AWS KMS. Trzeba zacząć od utworzenia CMK (*customer managed key*), czyli klucza szyfrowania, którym AWS zarządza za Ciebie. W tym celu najpierw musisz zdefiniować **politykę klucza**, czyli politykę IAM Policy określającą, kto może używać tego CMK. Aby zachować prostotę przykładu, utworzymy politykę klucza nadającą użytkownikowi bieżącemu uprawnienia administratora do CMK. Informacje dotyczące użytkownika bieżącego — np. nazwę użytkownika i ARN — można pobrać za pomocą źródła danych `aws_caller_identity`.

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

data "aws_caller_identity" "self" {}
```

Teraz danych wyjściowych źródła danych `aws_caller_identity` można użyć w źródle danych `aws_iam_policy_document` w celu utworzenia polityki klucza nadającej użytkownikowi bieżącemu uprawnienia administratora do CMK.

```
data "aws_iam_policy_document" "cmk_admin_policy" {
  statement {
    effect      = "Allow"
    resources   = ["*"]
    actions     = ["kms:*"]
    principals {
      type       = "AWS"
      identifiers = [data.aws_caller_identity.self.arn]
    }
  }
}
```

Następnym krokiem jest utworzenie CMK za pomocą zasobu `aws_kms_key`.

```
resource "aws_kms_key" "cmk" {
  policy = data.aws_iam_policy_document.cmk_admin_policy.json
}
```

Zauważ, że domyślnie KMS CMK jest identyfikowany na podstawie jedynie długiej wartości liczbowej, np. `b7670b0e-ed67-28e4-9b15-0d61e1485be3`, dobrą praktyką więc będzie utworzenie również czytelnego dla człowieka *aliasu* dla CMK przy użyciu do tego zasobu `aws_kms_alias`.

```
resource "aws_kms_alias" "cmk" {
  name           = "alias/kms-cmk-example"
  target_key_id = aws_kms_key.cmk.id
}
```

Dzięki temu aliasowi podczas pracy z API AWS i CLI do CMK można się odwoływać za pomocą `alias/kms-cmk-example` zamiast długiego identyfikatora typu `b7670b0e-ed67-28e4-9b15-0d61e1485be3`. Po utworzeniu klucza CMK można zacząć go stosować do szyfrowania i deszyfrowania danych. Pamiętaj, że nie będziesz w stanie poznać (a tym samym przypadkowo ujawnić) klucza szyfrowania. Tylko AWS ma dostęp do tego klucza szyfrowania, choć możesz go wykorzystać za pomocą API AWS i CLI, jak to teraz przedstawię.

Zacznij od utworzenia pliku o nazwie *db-creds.yml* przechowującego pewne dane poufne, np. dane uwierzytelniające bazy danych.

```
username: admin
password: password
```

Uwaga: *nie* umieszczaj tego pliku w systemie kontroli wersji, ponieważ jeszcze nie został zaszyfrowany! Aby zaszyfrować te dane, musisz użyć polecenia `aws kms encrypt` i otrzymany szyfrogram zapisać w nowym pliku. Spójrz na mały skrypt powłoki Bash (dla systemów operacyjnych Linux/UNIX/macOS) o nazwie *encrypt.sh*, który przeprowadza tę operację z użyciem AWS CLI:

```
CMK_ID="$1"
AWS_REGION="$2"
INPUT_FILE="$3"
OUTPUT_FILE="$4"

echo "Szyfrowanie pliku $INPUT_FILE za pomocą CMK $CMK_ID..."
```

```

ciphertext=$(aws kms encrypt \
  --key-id "$CMK_ID" \
  --region "$AWS_REGION" \
  --plaintext "fileb://$INPUT_FILE" \
  --output text \
  --query CiphertextBlob)

echo "Zapis danych do pliku $OUTPUT_FILE..."
echo "$ciphertext" > "$OUTPUT_FILE"

echo "Gotowe!"

```

Oto przykład użycia pliku *encrypt.sh* do szyfrowania pliku *db-creds.yml* za pomocą utworzonego wcześniej klucza KMS CMK i umieszczenia otrzymanego szyfrogramu w nowym pliku o nazwie *db-creds.yml.encrypted*:

```

$ ./encrypt.sh \
  alias/kms-cmk-example \
  us-east-2 \
  db-creds.yml \
  db-creds.yml.encrypted

```

```

Szyfrowanie pliku db-creds.yml za pomocą CMK alias/kms-cmk-example...
Zapis danych do pliku db-creds.yml.encrypted...
Gotowe!

```

Teraz można usunąć plik *db-creds.yml* (w postaci zwykłego tekstu) i bezpiecznie umieścić (zaszyfrowany) plik *db-creds.yml.encrypted* w systemie kontroli wersji. W tym momencie masz zaszyfrowany plik z pewnymi danymi poufnymi i być może się zastanawiasz, jak można je wykorzystać w kodzie Terraform.

Trzeba zacząć od deszyfrowania danych poufnych z tego pliku i użyć do tego źródła danych `aws_kms_secrets`.

```

data "aws_kms_secrets" "creds" {
  secret {
    name     = "db"
    payload = file("${path.module}/db-creds.yml.encrypted")
  }
}

```

Ten fragment kodu za pomocą funkcji pomocniczej `file()` odczytuje z dysku plik *db-creds.yml.encrypted* i — przy założeniu, że masz uprawnienia dostępu do odpowiedniego klucza w KMS — deszyfruje jego zawartość. W ten sposób otrzymujesz zawartość pierwotnego pliku *db-creds.yml*, kolejnym krokiem jest więc przetworzenie danych YAML w przedstawiony tutaj sposób:

```

locals {
  db_creds = yamldecode(data.aws_kms_secrets.creds.plaintext["db"])
}

```

Ten kod pobiera ze źródła danych `aws_kms_secrets` dane poufne dotyczące bazy danych, przetwarza YAML i umieszcza wynik w zmiennej lokalnej `db_creds`. Następnie można z niej odczytać nazwę użytkownika i hasło i przekazać je do zasobu `aws_db_instance`.

```

resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine            = "mysql"
}

```

```

allocated_storage = 10
instance_class     = "db.t2.micro"
skip_final_snapshot = true
db_name            = var.db_name

# Przekazanie danych poufnych do zasobu.
username = local.db_creds.username
password = local.db_creds.password
}

```

W ten sposób masz możliwość przechowywania danych poufnych w zaszyfrowanym pliku, z którego możesz je pobierać w kodzie Terraform. Taki plik można bezpiecznie przekazać do systemu kontroli wersji.

Warto w tym miejscu dodać, że praca z zaszyfrowanymi plikami jest niewygodna. W celu wprowadzenia zmiany należy lokalnie deszyfrować plik za pomocą długiego polecenia `aws kms decrypt`, dokonać odpowiednich zmian i ponownie zaszyfrować plik z użyciem innego długiego polecenia — `aws kms encrypt`. W trakcie tego procesu trzeba zachować wyjątkową ostrożność, aby pliku w postaci zwykłego tekstu przypadkowo nie przekazać do systemu kontroli wersji ani nie pozostawić w komputerze. Ten proces jest żmudny i podatny na błędy.

Jednym ze sposobów na ułatwienie sobie pracy w tym zakresie jest zastosowanie narzędzia typu open source o nazwie `sops` (<https://github.com/mozilla/sops>). Po użyciu polecenia `sops <PLIK>` narzędzie `sops` automatycznie deszyfruje plik, a jego zawartość otworzy w domyślnym edytorze tekstu. Po zakończeniu edycji i zamknięciu edytora tekstu narzędzie `sops` automatycznie szyfruje plik. Dzięki temu operacje szyfrowania i deszyfrowania odbywają się praktycznie w sposób niewidoczny dla użytkownika, który unika konieczności wydawania długich poleceń `aws kms`. Tym samym ogranicza się niebezpieczeństwo przekazania do systemu kontroli wersji niezaszyfrowanego pliku zawierającego dane poufne. Obecnie (2022 rok) narzędzie `sops` zapewnia obsługę plików szyfrowanych za pomocą AWS KMS, GCP KMS, Azure Key Vault lub kluczy PGP. Terraform nie oferuje jeszcze natywnej obsługi deszyfrowania plików zaszyfrowanych za pomocą `sops`, musisz więc skierować się do dostawcy zewnętrznego, np. `carlpett/sops` (<https://registry.terraform.io/providers/carlpett/sops/latest/docs>), albo — jeśli używasz `Terragrunt` — możesz skorzystać z funkcji wbudowanej `sops_decrypt_file()`, na której temat więcej informacji znajdziesz na stronie https://terragrunt.gruntwork.io/docs/reference/built-in-functions/#sops_decrypt_file.

Używanie zaszyfrowanych plików ma sporo korzyści, które wymienię poniżej:

- Unika się przechowywania w kodzie i w systemie kontroli wersji danych poufnych zapisanych w postaci zwykłego tekstu.
- Dane poufne są w systemie kontroli wersji zapisane w postaci zaszyfrowanej, są więc wersjonowane, spakowane i przetestowane razem z pozostałą częścią kodu. To pozwala zmniejszyć liczbę błędów konfiguracji, takich jak dodawanie nowych danych poufnych w jednym środowisku (np. roboczym), a pominięcie ich dodania w innym środowisku (np. produkcyjnym).
- Pobieranie danych poufnych jest łatwe — przy założeniu, że używany format szyfrowania jest obsługiwany natywnie przez Terraform lub wtyczkę zewnętrzną.

- Takie rozwiązanie działa z wieloma różnymi rodzajami szyfrowania, np. AWS KMS, GCP KMS, PGP itd.
- Wszystko jest zdefiniowane w kodzie. Nie ma żadnych ręcznych kroków dodatkowych ani wymaganych skryptów opakowujących (choć integracja z narzędziem sops wymaga wtyczki opracowanej przez podmiot zewnętrzny).

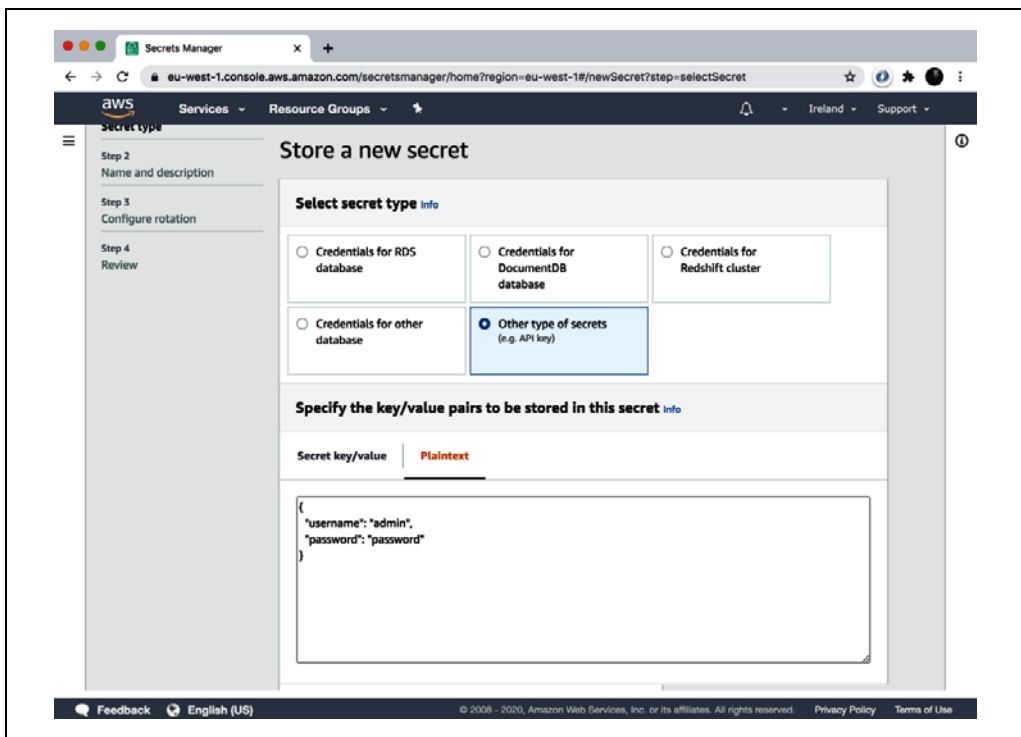
Używanie zaszyfrowanych plików ma również wady:

- Przechowywanie danych poufnych staje się trudniejsze. Konieczne jest wydawanie wielu poleceń (np. `aws kms encrypt`) albo też sięgnięcie po narzędzia zewnętrzne, takie jak sops. Istnieje również pewna trudność w poznaniu prawidłowego i bezpiecznego używania tego rodzaju narzędzi.
- Integracja z testami zautomatyzowanymi jest trudniejsza. Konieczne staje się wykonanie pewnej pracy dodatkowej w celu utworzenia kluczy szyfrowania i udostępnienia zaszyfrowanych danych testowych w środowisku testowym.
- Dane poufne pozostają zaszyfrowane, ale nadal są przechowywane w systemie kontroli wersji, więc ich rotacja i wycofywanie stają się trudne. Jeżeli komukolwiek uda się złamać klucz szyfrowania, będzie mógł deszyfrować wszystkie dane poufne zaszyfrowane za pomocą tego klucza.
- Możliwość sprawdzenia, kto uzyskał dostęp do danych poufnych, jest minimalna. Jeżeli korzystasz z oferowanej przez chmurę usługi zarządzania kluczem (np. AWS KMS), prawdopodobnie będzie ona zawierała dziennik zdarzeń zawierający informacje o tym, kto używa klucza szyfrowania. Jednak na podstawie tego dziennika nie będzie można stwierdzić, do czego ten klucz szyfrowania był używany (np. do których danych poufnych został uzyskany dostęp).
- Większość usług zarządzania kluczami jest płatna. Przykładowo każdy klucz przechowywany w AWS KMS kosztuje 1 dolar miesięcznie plus 3 centy za 10 000 wywołań API — każda operacja szyfrowania lub deszyfrowania wymaga jednego wywołania API. Typowy wzorzec użycia — niewielka liczba kluczy w KMS i aplikacje używające ich tylko do deszyfrowania danych poufnych podczas uruchamiania — zwykle oznacza koszt 1 – 10 dolarów miesięcznie. W wypadku większych wdrożeń, gdy masz dziesiątki aplikacji i setki danych poufnych, koszt to zwykle 10 – 50 dolarów miesięcznie.
- Standaryzacja praktyk zarządzania danymi poufnymi jest trudniejsza. Poszczególni programiści i zespoły mogą używać różnych sposobów przechowywania kluczy szyfrowania lub zarządzania zaszyfrowanymi plikami, a pomyłki zdarzają się dość często — np. niewłaściwe użycie szyfrowania albo przypadkowe umieszczenie w systemie kontroli wersji pliku zwykłego tekstu zawierającego dane poufne.

Magazyn danych poufnych

Trzecia technika polega na przechowywaniu danych poufnych w centralnym magazynie danych poufnych.

Do najpopularniejszych magazynów danych poufnych zaliczamy AWS Secrets Manager, Google Secret Manager, Azure Key Vault i HashiCorp Vault. W tym miejscu pokażę przykład użycia AWS Secrets Manager. Pierwszym krokiem jest umieszczenie w AWS Secrets Manager danych uwierzytelniających bazy danych, do czego można użyć konsoli AWS, jak pokazałem na rysunku 6.2.



Rysunek 6.2. Przechowywanie danych poufnych w formacie JSON w AWS Secrets Manager

Zwróć uwagę, że dane poufne pokazane na rysunku 6.2 są w formacie JSON, który jest formatem zalecanym do przechowywania danych w AWS Secrets Manager.

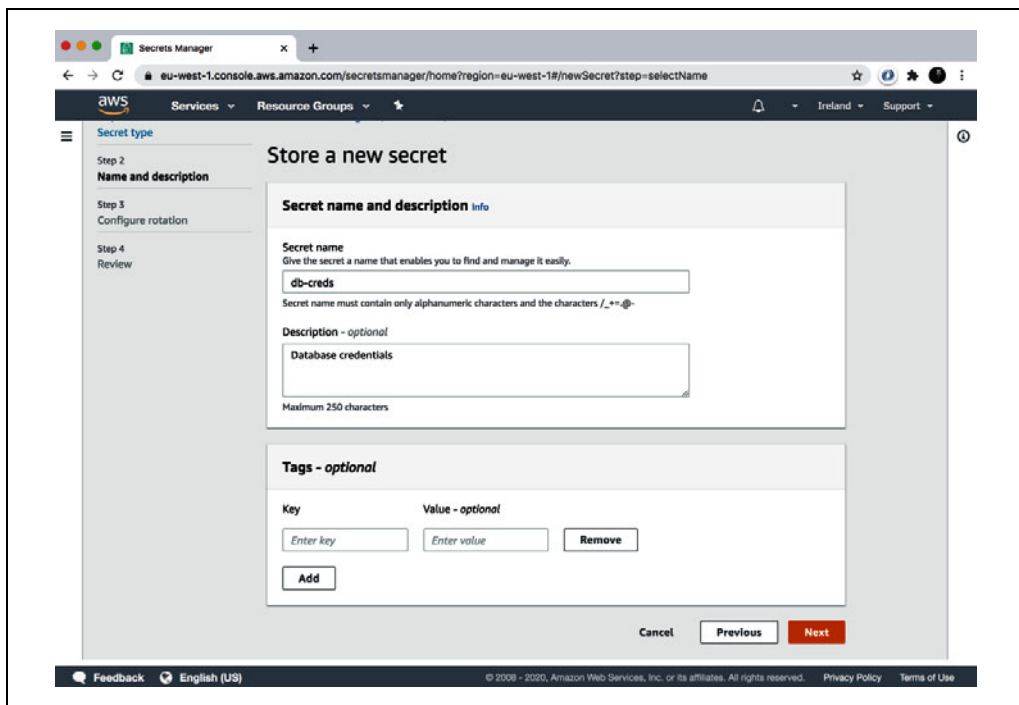
Przejdź do następnego kroku i upewnij się, że danym poufnym została nadana unikatowa nazwa, np. db-creds, jak pokazałem na rysunku 6.3.

Kliknij przyciski *Next* i *Store* w celu zapisania danych poufnych. Teraz w kodzie Terraform możesz skorzystać ze źródła danych `aws_secretsmanager_secret_version` podczas odczytywania danych poufnych db-creds.

```
data "aws_secretsmanager_secret_version" "creds" {
  secret_id = "db-creds"
}
```

Skoro dane poufne są przechowywane w formacie JSON, można użyć funkcji `jsondecode()` do ich przetworzenia na postać zmiennej lokalnej `db_creds`:

```
locals {
  db_creds = jsondecode(
    data.aws_secretsmanager_secret_version.creds.secret_string
  )
}
```

Rysunek 6.3. Nadanie danym poufnym unikatowej nazwy w AWS Secrets Manager

Teraz będzie można odczytać dane uwierzytelniające bazy danych z `db_creds` i przekazać je zasobowi `aws_db_instance`:

```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine           = "mysql"
  allocated_storage = 10
  instance_class   = "db.t2.micro"
  skip_final_snapshot = true
  db_name          = var.db_name

  # Przekazanie danych poufnych do zasobu.
  username = local.db_creds.username
  password = local.db_creds.password
}
```

Używanie magazynu danych poufnych ma takie zalety:

- Unika się przechowywania w kodzie i w systemie kontroli wersji danych poufnych zapisanych w postaci zwykłego tekstu.
- Wszystko jest zdefiniowane w kodzie. Nie ma żadnych ręcznych kroków dodatkowych ani wymaganych skryptów opakowujących.
- Dodawanie danych poufnych jest łatwe, zazwyczaj wykorzystuje się interfejs użytkownika bazujący na przeglądarce WWW.

- Magazyn danych poufnych zwykle obsługuje rotację i ich odbieranie, co jest użyteczne w sytuacji, gdy dojdzie do złamania danych poufnych. Istnieje nawet możliwość rotacji według harmonogramu (np. co 30 dni) jako rodzaju zabezpieczenia.
- Magazyn danych poufnych zwykle zapewnia obsługę szczegółowych dzienników zdarzeń audytu, pokazujących dokładnie, kto uzyskiwał do nich dostęp.
- Magazyn danych poufnych ułatwia standaryzację wszystkich praktyk dotyczących danych poufnych, ponieważ wymaga stosowania określonego typu szyfrowania, sposobu przechowywania, wzorców dostępu itd.

Używanie magazynu danych poufnych ma również wady:

- Dane poufne nie są wersjonowane, pakowane i testowane razem z pozostałą częścią kodu. To może prowadzić do pewnych błędów konfiguracji, takich jak dodawanie nowych danych poufnych w jednym środowisku (np. roboczym), a pominięcie ich dodania w innym środowisku (np. produkcyjnym).
- Większość zarządzanych magazynów danych poufnych jest płatna. Przykładowo każdy element danych poufnych w AWS Secrets Manager kosztuje 40 centów miesięcznie plus 5 centów za 10 000 wywołań API w celu pobrania tych danych. Typowy wzorzec użycia — wiele danych poufnych przechowywanych w różnych środowiskach i sporo aplikacji odczytujących te dane podczas uruchamiania — zwykle oznacza koszt 10 – 25 dolarów miesięcznie. W razie większych wdrożeń, gdy masz setki danych poufnych i dziesiątki korzystających z nich aplikacji, koszt może sięgać setek dolarów miesięcznie.
- Jeżeli używasz częściowo zarządzanego magazynu danych poufnych, np. HashiCorp Vault, musisz zapłacić za korzystanie z takiego magazynu danych (np. ponieść koszt 3 – 5 egzemplarzy AWS EC2 niezbędnych do działania Vault w trybie wysokiej dostępności), a także poświęcić czas i ponieść koszt związany z pracą programistów odpowiedzialnych za wdrożenie i skonfigurowanie tego magazynu danych, a także zarządzanie nim, uaktualnianie go i monitorowanie. Czas pracy programisty jest niezwykle kosztowny. W zależności od tego, ile czasu będzie poświęcał na przygotowanie magazynu danych poufnych i zarządzanie nim, to może oznaczać koszt liczony w setkach dolarów miesięcznie.
- Pobieranie danych poufnych jest trudniejsze, zwłaszcza w środowiskach zautomatyzowanych (np. uruchomienie aplikacji i próba odczytania hasła bazy danych), trzeba więc określić sposób bezpiecznego uwierzytelniania między różnymi maszynami.
- Integracja z testami zautomatyzowanymi jest trudniejsza, ponieważ większość testowanego kodu zależy teraz od jego działania w systemie zewnętrznym, który trzeba imitować albo też umieścić w nim testowane dane.

Pliki informacji o stanie i pliki planu

Mamy jeszcze dwa kolejne miejsca, z których mogą pochodzić dane poufne podczas pracy z Terraform:

- pliki informacji o stanie,
- pliki planu.

Pliki informacji o stanie

Mam nadzieję, że ten rozdział przekonał Cię, aby danych poufnych nie przechowywać w postaci zwykłego tekstu — mamy lepsze alternatywy. Jednak czymś, co przyciąga uwagę wielu użytkowników Terraform, niezależnie od używanej techniki, jest to, że *każde dane poufne przekazywane do zasobów Terraform i źródeł danych będą ostatecznie zapisane w postaci zwykłego tekstu w pliku informacji o stanie Terraform!*

Przykładowo — niezależnie od tego, w jaki sposób odczytujesz dane uwierzytniające bazy danych (zmienne środowiskowe, zaszyfrowane pliki, centralny magazyn danych poufnych) — jeśli przekazasz je do zasobu, takiego jak `aws_db_instance`:

```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  engine           = "mysql"
  allocated_storage = 10
  instance_class   = "db.t2.micro"
  skip_final_snapshot = true
  db_name          = var.db_name

  # Przekazanie danych poufnych do zasobu.
  username = local.db_creds.username
  password = local.db_creds.password
}
```

to Terraform będzie przechowywać te dane poufne w pliku `terraform.tfstate`, czyli w postaci zwykłego tekstu. To spowodowało zgłoszenie w 2014 roku błędu (<https://github.com/hashicorp/terraform/issues/516>), dla którego dotychczas nie zaproponowano dobrego rozwiązania. Istnieją pewne obejścia pozwalające na pozbycie się danych poufnych z plików informacji o stanie, ale są one zawodne i prawdopodobnie przestają działać po pojawieniu się nowych wydań Terraform, dlatego nie zalecam ich stosowania.

Na razie, niezależnie od tego, której z omówionych w rozdziale technik do zarządzania danymi poufnymi użyjesz, musisz wykonać jeszcze wymienione tutaj zadania:

Przechowywanie plików informacji o stanie w backendzie obsługującym szyfrowanie

Zamiast informacje o stanie przechowywać w lokalnym pliku `terraform.tfstate` i umieszczać go w systemie kontroli wersji, możesz skorzystać z jednego z backendów Terraform natywnie obsługujących szyfrowanie, np. S3, GCS i Azure Blob Storage. Te backendy będą szyfrowały pliki informacji o stanie zarówno podczas ich przekazywania (np. za pomocą TLS), jak i na dysku (np. za pomocą AES-256).

Ścisłe kontrolowanie tego, kto może uzyskać dostęp do backendu Terraform

Skoro pliki informacji o stanie Terraform mogą zawierać dane poufne, trzeba ściśle kontrolować, kto ma dostęp do backendu, *przynajmniej* z taką samą ostrożnością, z jaką kontrolowany jest dostęp do samych danych poufnych. Jeśli np. używasz backendu S3, możesz skonfigurować politykę IAM Policy, której jedynym celem będzie nadanie dostępu do produkcyjnego kubelka S3 tylko niewielkiej grupie zaufanych programistów lub jedynie serwerowi CI używanemu do wdrażania aplikacji w produkcji.

Pliki planu

Wielokrotnie zdarzyło Ci się spotkać z użyciem polecenia `terraform plan`. Tymczasem jedną z funkcjonalności, których nie pokazałem, jest przechowywanie danych wyjściowych tego polecenia w pliku:

```
$ terraform plan -out=example.plan
```

Wynikiem wykonania tego polecenia będzie umieszczenie planu Terraform w pliku o nazwie *example.plan*. Następnie można użyć polecenia `terraform apply` razem z planem zapisanym w pliku i tym samym zagwarantować, że Terraform wprowadzi *dokładnie* te zmiany, które wcześniej zostały przeanalizowane.

```
$ terraform apply example.plan
```

To jest jedna z użytecznych funkcjonalności Terraform, choć wiąże się z pewną kwestią. Podobnie jak w wypadku pliku informacji o stanie, także tutaj *wszelkie dane poufne przekazane do zasobów Terraform i magazynów danych ostatecznie trafią do pliku planu Terraform, który ma postać zwykłego tekstu!* Jeśli np. wykonasz polecenie `terraform plan` dla kodu `aws_db_instance` i zapiszesz wynik w pliku planu, ten plik (w postaci zwykłego tekstu) będzie zawierał nazwę użytkownika bazy danych i hasło.

Dlatego też, jeśli zamierzasz używać pliku planu, musisz wykonać jeszcze wymienione tutaj zadania:

Szyfrowanie plików planu Terraform

Jeżeli zamierzasz zapisywać plany Terraform w plikach, musisz znaleźć sposób na ich szyfrowanie zarówno podczas ich przekazywania (np. za pomocą TLS), jak i na dysku (np. za pomocą AES-256). Przykładowo pliki planu możesz przechowywać w kubelku S3 obsługującym oba typy szyfrowania.

Ścisłe kontrolowanie tego, kto może uzyskać dostęp do backendu Terraform

Skoro pliki planu Terraform mogą zawierać dane poufne, trzeba ściśle kontrolować, kto ma dostęp do backendu, *przynajmniej* z taką samą ostrożnością, z jaką kontrolowany jest dostęp do samych danych poufnych. Jeśli np. używasz S3 do przechowywania plików planu, możesz skonfigurować politykę IAM Policy, której jedynym celem będzie nadanie dostępu do produkcyjnego kubelka S3 tylko niewielkiej grupie zaufanych programistów albo jedynie serwerowi CI używanemu do wdrażania aplikacji w produkcji.

Podsumowanie

Oto najważniejsze wnioski płynące z tego rozdziału. Po pierwsze, jeżeli masz zapamiętać tylko jedno po lekturze rozdziału: nigdy *nie* należy przechowywać danych poufnych w plikach zwykłego tekstu.

Po drugie, w celu przekazania danych poufnych do dostawców użytkowników może użyć osobistego menedżera danych poufnych i zdefiniować zmienne środowiskowe, urządzenie natomiast może używać przechowywanych danych poufnych, ról IAM lub tożsamości OIDC. W tabeli 6.2 wymieniłem cechy poszczególnych rozwiązań w tym zakresie.

Tabela 6.2. Porównanie metod przekazywania danych poufnych do dostawców Terraform przez użytkowników lub urządzenia (np. serwer CI)

	Przechowywane dane uwierzytelniające	Role IAM	Tożsamość OIDC
Przykład	CircleCI	Jenkins w egzemplarzu EC2	GitHub Actions
Uniknięcie ręcznego zarządzania danymi uwierzytelniającymi	×	✓	✓
Uniknięcie trwałych danych uwierzytelniających	×	✓	✓
Praca u dostawcy chmury	×	✓	×
Praca poza dostawcą chmury	✓	×	✓
Powszechna obsługa w 2022 roku	✓	✓	×

Po trzecie, w celu przekazywania danych poufnych do zasobów i źródeł danych korzystaj ze zmiennych środowiskowych, zaszyfrowanych plików lub centralnych magazynów danych. W tabeli 6.3 wymienilem cechy poszczególnych rozwiązań w tym zakresie.

Tabela 6.3. Porównanie metod przekazywania danych poufnych do zasobów i źródeł danych Terraform

	Zmienne środowiskowe	Zaszyfrowane pliki	Centralny magazyn danych
Uniknięcie przechowywania danych poufnych w postaci zwykłego tekstu	✓	✓	✓
Wszystkie dane poufne zdefiniowane jako kod	×	✓	✓
Dziennik zdarzeń audytu dostępu do kluczy szyfrowania	×	✓	✓
Dziennik zdarzeń audytu dostępu do poszczególnych danych poufnych	×	×	✓
Łatwa rotacja i odbieranie danych poufnych	×	×	✓
Łatwa standaryzacja zarządzania danymi poufnymi	×	×	✓
Dane poufne są wersjonowane w kodzie	×	✓	×
Łatwe przechowywanie danych poufnych	✓	×	✓
Łatwe pobieranie danych poufnych	✓	✓	×
Łatwa integracja z testami zautomatyzowanymi	✓	×	×
Koszt	0	\$	\$\$\$

Po czwarte, niezależnie od sposobu przekazywania danych poufnych do zasobów i magazynów danych trzeba pamiętać, że Terraform te dane poufne umieści w plikach stanu i plikach planu, które mają postać plików zwykłego tekstu. Dlatego też trzeba się upewnić o ich szyfrowaniu (na czas transportu i na dysku) i o stosowaniu ścisłej kontroli dostępu do tych plików.

Teraz już wiesz, jak zarządzać danymi poufnymi podczas pracy z Terraform. Skoro wyjaśniłem kwestie związane z bezpiecznym przekazywaniem danych poufnych do dostawców Terraform, możemy przejść do rozdziału 7., z którego dowiesz się, jak używać Terraform z wieloma dostawcami (np. wiele regionów, wiele kont, wiele chmur).

Praca z wieloma dostawcami

Dotychczas praktycznie każdy przykład zamieszczony w książce zawierał blok `provider`:

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

Ten blok konfiguruje kod do wdrożenia w ramach pojedynczego konta AWS w pojedynczym regionie AWS. Takie rozwiązanie rodzi kilka pytań:

- Co można zrobić w sytuacji, gdy zachodzi potrzeba wdrożenia w wielu regionach AWS?
- Co można zrobić w sytuacji, gdy zachodzi potrzeba wdrożenia w wielu kontach AWS?
- Co można zrobić w sytuacji, gdy zachodzi potrzeba wdrożenia w innych chmurach, np. Azure lub GCP?

Aby udzielić odpowiedzi na te pytania, w rozdziale znacznie dokładniej omówię kwestię dostawców w Terraform:

- Praca z pojedynczym dostawcą.
- Praca z wieloma kopiami tego samego dostawcy.
- Praca z wieloma różnymi dostawcami.

Praca z pojedynczym dostawcą

Dotychczas dostawców używaliśmy niemalże „magicznie”. Takie podejście sprawdzało się świetnie w wypadku prostych przykładów z jednym prostym dostawcą. Jeśli natomiast chcesz pracować z wieloma regionami, kontami, chmurami itd., musisz się nieco bardziej zagłębić w temat dostawców. Zaczynamy od dokładnego zapoznania się z pojedynczym dostawcą, aby lepiej zrozumieć sposób jego działania.

- Czym jest dostawca?
- W jaki sposób można instalować dostawców
- Jak można używać dostawców?

Czym jest dostawca?

Gdy w rozdziale 2. wprowadziłem dostawców, określiłem ich jako *platformy*, z którymi współdziała Terraform, np. AWS, Azure, Google Cloud, DigitalOcean itd. W jaki sposób Terraform współpracuje z tymi platformami?

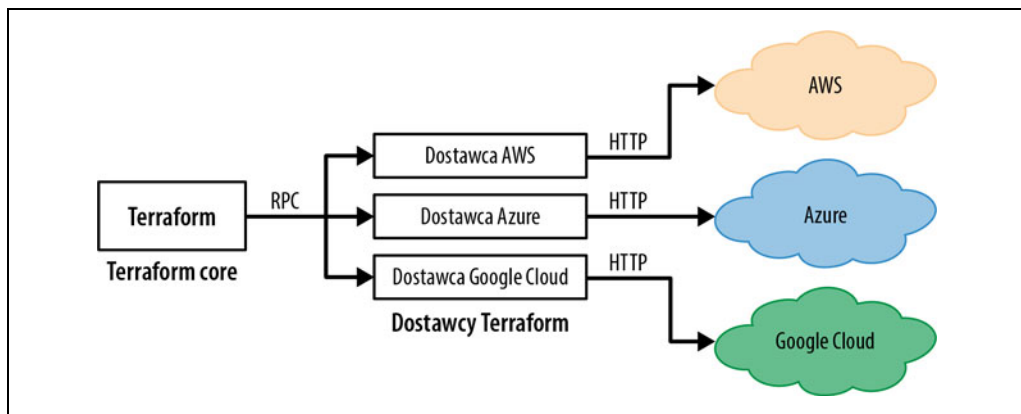
W tle Terraform składa się z dwóch części:

Core

To jest plik binarny *terraform* dostarczający całą podstawową funkcjonalność Terraform i używany na wszystkich platformach np. jako interfejs powłoki (polecenia typu `terraform plan` i `terraform apply`), analizator składni i interpreter kodu Terraform (HCL), a także pozwalający na tworzenie grafu zależności na podstawie zasobów i źródeł danych, tworzenie logiki odczytu i zapisu plików informacji o stanie itd. W tle kod został utworzony w języku programowania Go, znajduje się w repozytorium GitHub (<https://github.com/hashicorp/terraform>), jest udostępniany jako oprogramowanie open source, został opracowany i jest obsługiwany przez HashiCorp.

Dostawcy

Dostawcy Terraform to wtyczki dla Terraform core. Każda wtyczka została utworzona w Go i implementuje określony interfejs. Terraform core wie, jak zainstalować i uruchamiać wtyczkę. Poszczególne wtyczki zostały przeznaczone do pracy na pewnej platformie, np. AWS, Azure lub Google Cloud. Terraform core komunikuje się z wtyczkami za pomocą zdalnych wywołań procedur (ang. *remote procedure calls*, RPC), z kolei te wtyczki z odpowiadającymi im platformami komunikują się poprzez sieć (np. wywołania HTTP), jak pokazałem na rysunku 7.1.



Rysunek 7.1. Interakcje zachodzące między Terraform core, dostawcami i światem zewnętrznym

Kod każdej wtyczki zwykle znajduje się w oddzielnym repozytorium. Przykładowo cała funkcjonalność AWS używana dotychczas w książce pochodzi z wtyczki o nazwie Terraform AWS Provider (lub po prostu dostawca AWS), którą znajdziesz w repozytorium pod adresem <https://github.com/hashicorp/terraform-provider-aws>. Wprawdzie firma HashiCorp utworzyła większość początkowych wtyczek dostawców, ale wciąż pomaga w obsłudze wielu z nich. Obecnie większość pracy związanej z obsługą poszczególnych wtyczek wykonuje firma będąca właścicielem

danej platformy: np. pracownicy AWS zajmują się dostawcą AWS, pracownicy Microsoftu zajmują się dostawcą Azure, natomiast pracownicy Google zajmują się dostawcą Google Cloud itd.

Każdy dostawca stosuje określony prefiks i udostępnia co najmniej jeden zasób i źródło danych, których nazwa obejmuje ten prefiks. Dlatego też wszystkie zasoby i źródła danych dostawcy AWS używają prefiksu `aws_` (np. `aws_instance`, `aws_autoscaling_group`, `aws_ami`), wszystkie zasoby i źródła danych dostawcy Azure używają prefiksu `azurerm_` (`azurerm_virtual_machine`, `azurerm_virtual_machine_scale_set`, `azurerm_image`) itd.

Jak odbywa się instalacja dostawcy?

W wypadku oficjalnych dostawców Terraform, np. dla AWS, Azure i Google Cloud, wystarczające jest dodanie bloku `providers` do utworzonego kodu¹.

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

Tuż po użyciu polecenia `terraform init` Terraform automatycznie pobierze od dostawcy niezbędny kod:

```
$ terraform init
```

```
Initializing provider plugins...
- Finding hashicorp/aws versions matching "4.19.0"...
- Installing hashicorp/aws v4.19.0...
- Installed hashicorp/aws v4.19.0 (signed by HashiCorp)
```

To rozwiązanie działa niemal magicznie, prawda? Skąd Terraform wie, którego dostawcy chcesz używać? Ewentualnie która wersja dostawcy Cię interesuje? Skąd ją pobrać? Wprawdzie można na tej „magii” polegać podczas uczenia się i poznawania Terraform, ale w trakcie tworzenia kodu produkcyjnego prawdopodobnie chcesz mieć nieco większą kontrolę nad tym, jak Terraform instaluje dostawców. To jest możliwe dzięki blokowi `required_providers`, którego składnia przedstawia się następująco:

```
terraform {
  required_providers {
    <NAZWA_LOKALNA> = {
      source = "<ADRES_URL>"
      version = "<WERSJA>"
    }
  }
}
```

gdzie:

`NAZWA_LOKALNA`

To jest **nazwa lokalna** używana w danym module dla dostawcy. Każdemu dostawcy trzeba nadać unikatową nazwę i używać jej w blokach `provider` konfiguracji. W większości przypadków

¹ W rzeczywistości można nawet pominąć blok `provider` i dodać dowolny zasób albo źródło danych z oficjalnego dostawcy, a Terraform na podstawie prefiksu ustali, który dostawca powinien zostać użyty. Jeśli np. dodasz zasób `aws_instance`, Terraform użyje dostawcy AWS wybranego na podstawie prefiksu `aws_`.

będziesz używać *preferowanej nazwy lokalnej* danego dostawcy, np. w wypadku dostawcy AWS preferowaną nazwą lokalną jest `aws`. Dlatego też blok dostawcy jest zapisywany w postaci `provider "aws" { ... }`. Jednak w rzadkich sytuacjach może się zdarzyć, że dwóch dostawców będzie miało tę samą preferowaną nazwę lokalną — np. dwóch dostawców zajmuje się obsługą żądań HTTP i ma preferowaną nazwę lokalną `http` — wówczas można używać zdefiniowanej nazwy lokalnej w celu rozróżniania tych dostawców.

ADRES_URL

To jest adres URL, z którego Terraform powinien pobierać dostawcę. Adres URL ma postać `[<NAZWA_HOSTA>]/<PRZESTRZEŃ_NAZW>/<TYP>`, gdzie `NAZWA_HOSTA` to nazwa hosta rejestru Terraform rozprowadzającego danego dostawcę, `PRZESTRZEŃ_NAZW` to organizacyjna przestrzeń nazw (zwykle to nazwa firmy), a `TYP` to nazwa platformy, którą zarządza ten dostawca (`TYP` jest zwykle preferowaną nazwą lokalną). Przykładowo pełny adres URL dostawcy AWS rozpowszechnianego przez publiczne repozytorium Terraform (<https://registry.terraform.io/>) to `registry.terraform.io/hashicorp/aws`. Zwróć uwagę, że `NAZWA_HOSTA` jest opcjonalna i jeśli ją pominiesz, Terraform domyślnie pobierze dostawcę z oficjalnego repozytorium publicznego. Dlatego też krótszym i częściej spotykanym zapisem adresu URL tego samego dostawcy jest `hashicorp/aws`. `NAZWE_HOSTA` zwykle dołącza się w wypadku dostawców niestandardowych, pobieranych z repozytoriów prywatnych (np. repozytorium prywatnego uruchomionego w ramach rozwiązania Terraform Cloud lub Terraform Enterprise).

WERSJA

To jest ograniczenie wersji. Można np. podać konkretną wersję, taką jak `4.0.19`, albo też zakres wersji, np. `> 4.0, < 4.3`. Więcej informacji na temat obsługi wersjonowania znajdziesz w rozdziale 8.

Przykładowo w celu zainstalowania dostawcy AWS w wersji 4.0 można użyć przedstawionego tutaj fragmentu kodu:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

Teraz już w pełni rozumiesz poznany wcześniej magiczny sposób działania podczas instalacji dostawcy. Jeżeli do kodu dodasz nowy blok `provider` o nazwie `foo` i nie określisz bloku `required_providers`, to podczas wykonywania polecenia `terraform init` Terraform automatycznie wykona takie operacje:

- Próba pobrania dostawcy `foo` przy założeniu, że `NAZWA_HOSTA` to oficjalny publiczny rejestr Terraform, `PRZESTRZEŃ_NAZW` to `hashicorp`, adres URL ma więc postać `registry.terraform.io/hashicorp/foo`.
- Jeżeli to poprawny adres URL, zostanie zainstalowana najnowsza dostępna wersja dostawcy `foo`.

Jeżeli chcesz zainstalować dowolnego dostawcę, a nie dostawcę w przestrzeni nazw hashi corp (np. chcesz używać dostawcy z Datadog, Cloudflare, Confluent albo opracowanego samodzielnie), lub też jeśli chcesz kontrolować wersję używanego dostawcy, musisz skorzystać z bloku `required_providers`.



Zawsze dołączaj blok `required_providers`

Jak się dowiesz z rozdziału 8., duże znaczenie ma kontrolowanie wersji używanego dostawcy. Dlatego też zalecam, aby *zawsze* umieszczać w kodzie blok `required_providers`.

W jaki sposób używać dostawców?

Skoro już nieco wiesz o dostawcach, warto powrócić do tematu ich używania. Pierwszym krokiem jest dodanie do kodu bloku `required_providers`, aby wskazać dostawcę przeznaczony do użycia.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

Teraz można już dodać blok `provider` konfigurujący tego dostawcę.

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

Wprawdzie w omawianym przykładzie konfigurujemy jedynie używany region dostawcy AWS, ale dostępnych jest wiele innych ustawień. Zawsze należy sprawdzać szczegóły dostawcy w dokumentacji: zwykle znajduje się ona w tym samym rejestrze, z którego jest pobierany dostawca (wskazany w adresie URL `source`). Przykładowo dokumentacja dostawcy AWS (<https://registry.terraform.io/providers/hashicorp/aws/latest/docs>) znajduje się w publicznym rejestrze Terraform. W dokumentacji zwykle jest wyjaśnione, jak skonfigurować dostawcę do pracy z różnymi użytkownikami, rolami, regionami, kontami itd.

Po skonfigurowaniu dostawcy wszystkie umieszczone w kodzie zasoby i źródła danych z tego dostawcy (czyli posiadające ten sam prefiks) będą automatycznie używały tej konfiguracji. Przykładowo po określeniu, że dostawca `aws` korzysta z regionu `us-east-2`, zasoby o prefiksie `aws_` w kodzie będą automatycznie wdrażane do `us-east-2`.

Co zrobić w sytuacji, gdy część tych zasobów ma być wdrożona w `us-east-2`, podczas gdy część w innym regionie, np. `us-west-1`? Co w sytuacji, gdy część zasobów ma być wdrożona w ramach zupełnie innego konta AWS? W tym celu musisz się dowiedzieć, jak utworzyć wiele kopii tego samego dostawcy, co zostanie omówione w następnym podrozdziale.

Praca z wieloma kopiami tego samego dostawcy

Aby się dowiedzieć, jak pracować z wieloma kopiami tego samego dostawcy, zapoznaj się z kilkoma najczęściej występującymi sytuacjami:

- Praca z wieloma regionami AWS.
- Praca z wieloma kontami AWS.
- Tworzenie modułów, które mogą współdziałać z wieloma dostawcami.

Praca z wieloma regionami AWS

Większość dostawców chmury pozwala na wdrażanie w centrach danych (tzw. regionach) znajdujących się na całym świecie. Jednak podczas konfigurowania dostawcy Terraform zwykle korzysta się z tylko jednego regionu. W przedstawionych dotychczas przykładach wdrożenia odbywały się do pojedynczego regionu AWS — `us-east-2`.

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

Co można zrobić w sytuacji, gdy zachodzi potrzeba wdrożenia do wielu regionów? Jak np. wdrożyć część zasobów do `us-east-2`, inne zaś do `us-west-1`? Kuszące może być zdefiniowanie dwóch bloków konfiguracyjnych `provider`, jak pokazałem w kolejnym fragmencie kodu.

```
provider "aws" {  
  region = "us-east-2"  
}  
  
provider "aws" {  
  region = "us-west-1"  
}
```

Jednak w tym wypadku powstaje problem: w jaki sposób wskazać, który blok `provider` powinien być używany dla poszczególnych zasobów, źródeł danych i modułów? Zacznijmy od źródeł danych. Załóżmy, że masz dwie kopie źródła danych `aws_region`, które zwracają bieżący region AWS.

```
data "aws_region" "region_1" {  
}  
  
data "aws_region" "region_2" {  
}
```

Jak sprawić, aby źródło danych `region_1` używało dostawcy `us-east-2`, a źródło danych `region_2` — `us-west-1`? Rozwiązaniem jest dodanie aliasu do poszczególnych dostawców.

```
provider "aws" {  
  region = "us-east-2"  
  alias  = "region_1"  
}  
  
provider "aws" {  
  region = "us-west-1"  
  alias  = "region_2"  
}
```

Alias to własna nazwa dla bloku provider, którą można wyraźnie przekazać poszczególnym zasobom, źródłom danych i modułom, aby była używana konfiguracja w tym konkretnym dostawcy. Jeżeli chcesz nakazać źródłom danych `aws_region` używanie określonego dostawcy, parametr `provider` musisz zdefiniować w przedstawiony tutaj sposób.

```
data "aws_region" "region_1" {
  provider = aws.region_1
}
```

```
data "aws_region" "region_2" {
  provider = aws.region_2
}
```

Dodaj zmienne danych wyjściowych, aby sprawdzić, czy takie rozwiązanie działa zgodnie z oczekiwaniami.

```
output "region_1" {
  value       = data.aws_region.region_1.name
  description = "Nazwa pierwszego regionu"
}

output "region_2" {
  value       = data.aws_region.region_2.name
  description = "Nazwa drugiego regionu"
}
```

Teraz można już użyć polecenia `terraform apply`.

```
$ terraform apply
```

```
(...)
```

```
Outputs:
```

```
region_1 = "us-east-2"
region_2 = "us-west-1"
```

W omawianym przykładzie każdy zasób `aws_region` korzysta z innego dostawcy, a tym samym z innego regionu AWS. Ta sama technika definiowania parametru `provider` sprawdza się również podczas pracy z zasobami. Dla przykładu w kolejnym fragmencie kodu pokazałem, jak można wdrożyć dwa egzemplarze EC2 w różnych regionach.

```
resource "aws_instance" "region_1" {
  provider = aws.region_1

  # Zwróć uwagę na różne identyfikatory AMI!
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

```
resource "aws_instance" "region_2" {
  provider = aws.region_2

  # Zwróć uwagę na różne identyfikatory AMI!
  ami           = "ami-01f87c43e618bf8f0"
  instance_type = "t2.micro"
}
```

Zwróć uwagę, jak zasób `aws_instance` definiuje parametr `provider` w celu zagwarantowania wdrożenia w odpowiednim regionie. Zauważ również, że parametr `ami` ma odmienną wartość w poszczególnych zasobach `aws_instance`. To wynika z tego, że identyfikatory AMI są unikatowe dla regionów, identyfikator Ubuntu 20.04 w regionie `us-east-2` jest więc inny niż identyfikator Ubuntu 20.04 w regionie `us-west-1`. Konieczność ręcznego wyszukiwania i zarządzania tymi identyfikatorami AMI jest żmudna i podatna na błędy. Na szczęście istnieje lepsza alternatywa: użycie źródła danych `aws_ami`, które po otrzymaniu zbioru filtrów potrafi automatycznie odszukać identyfikator AMI. Zobacz, jak można dwukrotnie użyć tego źródła danych, jednokrotnie w każdym regionie, do wyszukania identyfikatora AMI dla Ubuntu 20.04.

```
data "aws_ami" "ubuntu_region_1" {
  provider = aws.region_1

  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}

data "aws_ami" "ubuntu_region_2" {
  provider = aws.region_2

  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}
```

Zwróć uwagę, jak każde źródło danych definiuje parametr `provider` w celu zagwarantowania, że w odpowiednim regionie zostanie znaleziony identyfikator AMI. Powróć do kodu `aws_instance` i uaktualnij parametr `ami` w celu użycia danych wyjściowych tych źródeł zamiast na stałe zdefiniowanych wartości.

```
resource "aws_instance" "region_1" {
  provider = aws.region_1

  ami           = data.aws_ami.ubuntu_region_1.id
  instance_type = "t2.micro"
}

resource "aws_instance" "region_2" {
  provider = aws.region_2

  ami           = data.aws_ami.ubuntu_region_2.id
  instance_type = "t2.micro"
}
```

Teraz już znacznie lepiej. Niezależnie od regionu, do którego odbywa się wdrożenie, automatycznie otrzymasz poprawny identyfikator obrazu AMI Ubuntu. Aby sprawdzić, czy te egzemplarze EC2 faktycznie są wdrażane w różnych regionach, dodaj zmienne danych wyjściowych wyświetlające strefę dostępności (w poszczególnych regionach), w której faktycznie został wdrożony egzemplarz.

```
output "instance_region_1_az" {
  value      = aws_instance.region_1.availability_zone
  description = "Strefa dostępności w pierwszym regionie, w którym został wdrożony egzemplarz"
}

output "instance_region_2_az" {
  value      = aws_instance.region_2.availability_zone
  description = "Strefa dostępności w drugim regionie, w którym został wdrożony egzemplarz"
}
```

Teraz zastosuj polecenie `terraform apply`.

```
$ terraform apply
```

```
(...)
```

```
Outputs:
```

```
instance_region_1_az = "us-east-2a"
instance_region_2_az = "us-west-1b"
```

W porządku, teraz już wiesz, jak źródła danych i zasoby mogą być wdrażane w różnych regionach. Jak to wygląda w wypadku modułów? W rozdziale 3. np. użyliśmy Amazon RDS do wdrożenia pojedynczego egzemplarza bazy danych MySQL w środowisku roboczym (*stage/data-stores/mysql*):

```
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"
  skip_final_snapshot = true

  username = var.db_username
  password = var.db_password
}
```

Takie rozwiązanie jest akceptowalne w środowisku roboczym, w produkcji natomiast pojedyncza baza danych stanowi pojedynczy punkt awarii. Na szczęście Amazon RDS natywnie obsługuje **replikację**, w której dane są automatycznie kopiowane z podstawowej bazy danych do zapasowej — *replika* tylko do odczytu — co jest użyteczne z perspektywy skalowalności i jako rodzaj zabezpieczenia, gdy podstawowa baza danych stanie się niedostępna. Istnieje możliwość replikacji nawet do zupełnie innego regionu AWS, więc nawet jeśli jeden region stanie się niedostępny (np. przez poważną awarię w us-east-2), będzie można przełączyć się do innego (np. us-west-1).

Zmienimy teraz kod MySQL w środowisku roboczym na moduł `mysql` wielokrotnego użycia obsługujący replikację. Zaczniemy od skopiowania całej zawartości `stage/data-stores/mysql` (powinna obejmować `main.tf`, `variables.tf` i `outputs.tf`) do nowego katalogu `modules/data-stores/mysql`. Teraz otwórz `modules/data-stores/mysql/variables.tf` i udostępnij dwie nowe zmienne:

```
variable "backup_retention_period" {
  description = "Dni, przez które będzie zachowana kopia zapasowa. Aby włączyć replikację,
  ↳ to musi być wartość większa niż 0"
  type        = number
  default     = null
}

variable "replicate_source_db" {
  description = "Jeżeli będzie podana, odbędzie się replikacja bazy danych RDS o podanym ARN"
  type        = string
  default     = null
}
```

Jak wkrótce zobaczysz, definiujesz zmienną `backup_retention_period` w podstawowej bazie danych, aby włączyć replikację, i zmienną `replicate_source_db` w zapasowej bazie danych, aby zmienić ją na replikę. Przejdź do pliku `modules/data-stores/mysql/main.tf` i uaktualnij zasób `aws_db_instance` w przedstawiony tutaj sposób:

1. Zmienne `backup_retention_period` i `replicate_source_db` przekaż do parametrów o takich samych nazwach w zasobie `aws_db_instance`.
2. Jeżeli egzemplarz bazy danych jest repliką, AWS nie pozwoli na zdefiniowanie parametrów `engine`, `db_name` lub `password`, ponieważ są one dziedziczone po podstawowej bazie danych. Dlatego też w zasobie `aws_db_instance` trzeba dodać pewną logikę warunkową, aby te parametry nie były definiowane w wypadku istnienia zmiennej `replicate_source_db`.

Spójrz, jak te zasoby powinny wyglądać po wprowadzonych zmianach.

```
resource "aws_db_instance" "example" {
  identifier_prefix = "terraform-up-and-running"
  allocated_storage = 10
  instance_class    = "db.t2.micro"
  skip_final_snapshot = true

  # Włączenie tworzenia kopii zapasowej.
  backup_retention_period = var.backup_retention_period

  # Jeżeli ta zmienna jest zdefiniowana, ta baza danych będzie repliką.
  replicate_source_db = var.replicate_source_db

  # Te parametry będą zdefiniowane tylko w razie braku zmiennej replicate_source_db.
  engine   = var.replicate_source_db == null ? "mysql" : null
  db_name  = var.replicate_source_db == null ? var.db_name : null
  username = var.replicate_source_db == null ? var.db_username : null
  password = var.replicate_source_db == null ? var.db_password : null
}
```

Zwróć uwagę, że w przypadku replik to oznacza, że zmienne `db_name`, `db_username` i `db_password` w tym module powinny być opcjonalne. Dlatego też dobrym pomysłem będzie powrót do pliku `modules/data-stores/mysql/variables.tf` i przypisanie wartości `null` wymienionym zmiennym.


```

variable "db_name" {
  description = "Nazwa bazy danych"
  type        = string
  default     = null
}

variable "db_username" {
  description = "Nazwa użytkownika bazy danych"
  type        = string
  sensitive   = true
  default     = null
}

variable "db_password" {
  description = "Hasło bazy danych"
  type        = string
  sensitive   = true
  default     = null
}

```

W celu użycia zmiennej `replicate_source_db` konieczne jest przypisanie jej wartości ARN innej bazy danych RDS, trzeba więc również uaktualnić `modules/data-stores/mysql/outputs.tf` i dodać ARN bazy danych jako zmienną danych wyjściowych.

```

output "arn" {
  value     = aws_db_instance.example.arn
  description = "Wartość ARN bazy danych"
}

```

I jeszcze jedno: do tego modułu trzeba dodać blok `required_providers`, aby wskazać, że moduł oczekuje użycia dostawcy AWS, a także w celu określenia oczekiwanej wersji tego dostawcy.

```

terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

```

Za chwilę się przekonasz, dlaczego to ma tak duże znaczenie również podczas pracy z wieloma regionami.

W porządku, teraz możesz używać modułu `mysql` do wdrażania w środowisku produkcyjnym podstawowej bazy danych MySQL i jej repliki. Rozpocznij od utworzenia pliku `live/prod/data-stores/mysql/variables.tf`, aby udostępnić zmienne danych wejściowych przeznaczone dla nazwy użytkownika i hasła bazy danych (jak wyjaśniłem w rozdziale 6., te dane można przekazać także za pomocą zmiennych środowiskowych).

```

variable "db_username" {
  description = "Nazwa użytkownika bazy danych"
  type        = string
  sensitive   = true
}

variable "db_password" {
  description = "Hasło bazy danych"
}

```

```

    type          = string
    sensitive     = true
}

```

Następnym krokiem jest utworzenie pliku *live/prod/data-stores/mysql/main.tf* i użycie modułu `mysql` do skonfigurowania podstawowej bazy danych, jak pokazałem w kolejnym fragmencie kodu.

```

module "mysql_primary" {
    source = "../../../../../modules/data-stores/mysql"

    db_name      = "prod_db"
    db_username  = var.db_username
    db_password  = var.db_password

    # Ta zmienna jest konieczna do zapewnienia obsługi replikacji.
    backup_retention_period = 1
}

```

Teraz dodajemy drugie użycie modułu `mysql` w celu utworzenia repliki.

```

module "mysql_replica" {
    source = "../../../../../modules/data-stores/mysql"

    # Utworzenie repliki podstawowej bazy danych.
    replicate_source_db = module.mysql_primary.arn
}

```

To eleganckie i krótkie rozwiązanie. Polega ono na przekazaniu wartości ARN podstawowej bazy danych do parametru `replicate_source_db`, co powinno spowodować uruchomienie bazy danych RDS jako repliki.

Mamy tutaj jeden problem: w jaki sposób nakazać wdrożenie w różnych regionach podstawowej bazy danych i jej repliki? W tym celu należy zdefiniować dwa bloki `provider`, każdy z własnym aliasem.

```

provider "aws" {
    region = "us-east-2"
    alias  = "primary"
}

provider "aws" {
    region = "us-west-1"
    alias  = "replica"
}

```

Aby wskazać modułowi dostawcę przeznaczonego do użycia, skorzystaj z parametru `providers`. Spójrz na przykład konfiguracji podstawowej bazy danych, która ma używać dostawcy `primary` (czyli w omawianym przykładzie `us-east-2`).

```

module "mysql_primary" {
    source = "../../../../../modules/data-stores/mysql"

    providers = {
        aws = aws.primary
    }

    db_name      = "prod_db"
    db_username  = var.db_username
    db_password  = var.db_password
}

```

```

# Ta zmienna jest konieczna do zapewnienia obsługi replikacji.
backup_retention_period = 1
}

```

A teraz zobacz, jak skonfigurować replikę MySQL do użycia dostawcy replica (w omawianym przykładzie to us-west-1).

```

module "mysql_replica" {
  source = "../../../../../modules/data-stores/mysql"

  providers = {
    aws = aws.replica
  }

  # Utworzenie repliki podstawowej bazy danych.
  replicate_source_db = module.mysql_primary.arn
}

```

Zwróć uwagę, że w przypadku modułów parametr `providers` (to liczba mnoga w języku angielskim) jest mapowaniem, podczas gdy w wypadku zasobów i źródeł danych parametr `provider` (to liczba pojedyncza w języku angielskim) jest pojedynczą wartością. To wynika z tego, że każdy zasób i każde źródło danych są wdrażane w dokładnie jednym dostawcy, moduł natomiast może zawierać wiele zasobów i źródeł danych, a tym samym używać wielu dostawców (przykład wykorzystania wielu dostawców przez moduł przedstawię w dalszej części rozdziału). W mapowaniu przekazywanym do modułu za pomocą parametru `providers` klucz musi być dopasowany do nazwy lokalnej dostawcy w mapowaniu `required_providers` modułu (w omawianym przykładzie oba klucze to `aws`). To jest kolejny powód, dla którego wyraźne definiowanie `required_providers` w każdym module to świetne rozwiązanie.

Ostatnim krokiem jest utworzenie pliku `live/prod/data-stores/mysql/outputs.tf` razem z takimi zmiennymi danych wyjściowych:

```

output "primary_address" {
  value      = module.mysql_primary.address
  description = "Nawiązanie połączenia z podstawową bazą danych w tym punkcie końcowym"
}

output "primary_port" {
  value      = module.mysql_primary.port
  description = "Port, na którym nasłuchuje podstawowa baza danych"
}

output "primary_arn" {
  value      = module.mysql_primary.arn
  description = "Wartość ARN podstawowej bazy danych"
}

output "replica_address" {
  value      = module.mysql_replica.address
  description = "Nawiązanie połączenia z repliką bazy danych w tym punkcie końcowym"
}

output "replica_port" {
  value      = module.mysql_replica.port
  description = "Port, na którym nasłuchuje replika bazy danych"
}

```

```
output "replica_arn" {
  value      = module.mysql_replica.arn
  description = "Wartość ARN repliki bazy danych"
}
```

Wreszcie można wdrożyć rozwiązanie. Użycie polecenia `terraform apply` spowoduje uruchomienie podstawowej bazy danych i jej repliki, co może zabrać dużo czasu, nawet 20-30 minut, więc cierpliwości!

\$ terraform apply

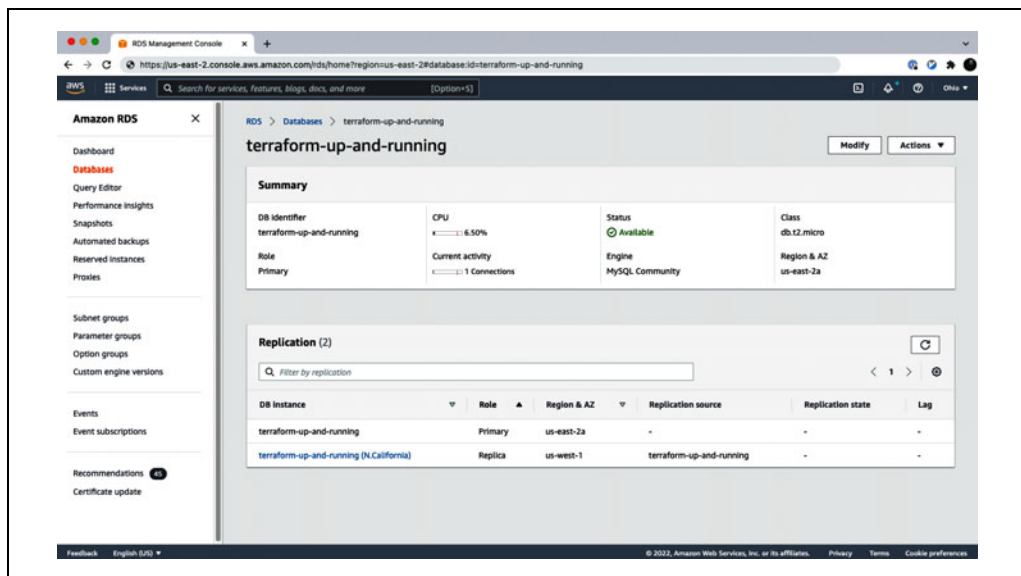
(...)

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

```
primary_address = "terraform-up-and-running.cmyd6qwb.us-east-2.rds.amazonaws.com"
primary_arn      = "arn:aws:rds:us-east-2:111111111111:db:terraform-up-and-running"
primary_port     = 3306
replica_address  = "terraform-up-and-running.drctpdoe.us-west-1.rds.amazonaws.com"
replica_arn      = "arn:aws:rds:us-west-1:111111111111:db:terraform-up-and-running"
replica_port     = 3306
```

W ten sposób masz replikację bazy danych między regionami! Możesz zalogować się do konsoli RDS ([console.aws.amazon.com](https://console.aws.amazon.com/rds)) w celu potwierdzenia działania replikacji. Jak pokazałem na rysunku 7.2, w omawianym przykładzie powinno być widać, że podstawowa baza danych znajduje się w regionie `us-east-2`, jej replika zaś w `us-west-1`.



Rysunek 7.2. Konsola RDS pokazuje, że podstawowa baza danych znajduje się w regionie `us-east-2`, jej replika zaś w `us-west-1`

Jako ćwiczenie pozostawię Ci uaktualnienie środowiska roboczego (`stage/data-stores/mysql`) w taki sposób, aby również używało modułu `mysql` (`modules/data-stores/mysql`). W tym wypadku jednak

konfiguracja *nie* powinna zawierać replikacji, ponieważ zwykle w środowiskach przedprodukcyjnych nie potrzebujemy aż takiego poziomu dostępności.

Jak możesz zobaczyć w omówionych przykładach, dzięki użyciu wielu dostawców razem z aliasami wdrażanie zasobów w wielu regionach za pomocą Terraform jest dość łatwym zadaniem. Zanim jednak przejdziesz dalej, zapoznaj się z dwoma ostrzeżeniami.

Ostrzeżenie 1.: Używanie wielu regionów jest trudne

Aby uruchomić infrastrukturę w wielu regionach na całym świecie, zwłaszcza w trybie „aktywny – aktywny”, gdy w tym samym czasie więcej niż tylko jeden region aktywnie odpowiada na żądania użytkownika (w przeciwieństwie do sytuacji, w której tylko jeden region pozostaje w trybie czuwania), konieczne jest rozwiązanie wielu trudnych problemów, takich jak opóźnienie między regionami, wybór między jednym zapisującym (to oznacza niższą dostępność i większe opóźnienia) i wieloma zapisującymi (to oznacza spójność końcową lub sharding), ustalenie sposobu generowania unikatowych identyfikatorów (standardowe automatyczne generowanie identyfikatorów przez bazę danych okazuje się niewystarczające), konieczność spełnienia lokalnych wymagań dotyczących danych itd. Wprawdzie te zagadnienia wykraczają poza zakres tematyczny książki, ale uznałem, że przynajmniej muszę o nich wspomnieć. Dzięki temu masz jasność, że wdrożenia w wielu regionach są znacznie trudniejsze niż jedynie użycie kilku dodatkowych aliasów w kodzie Terraform.

Ostrzeżenie 2.: Oszczędnie korzystaj z aliasów

Korzystanie z aliasów w Terraform jest bardzo łatwe, mimo to odradzam ich zbyt częste używanie, zwłaszcza podczas tworzenia infrastruktury składającej się z wielu regionów. Jednym z podstawowych powodów przygotowania infrastruktury bazującej na wielu regionach jest zapewnienie odporności na awarię jednego regionu. Przykładowo w razie awarii regionu us-east-2 infrastruktura znajdująca się w regionie us-west-1 nadal będzie funkcjonować. Jeśli natomiast użyjesz pojedynczego modułu Terraform wykorzystującego aliasy do wdrożenia w obu regionach, gdy jeden z nich ulegnie awarii, moduł nie będzie w stanie nawiązać z nim połączenia, każda próba wykonania `terraform plan` lub `terraform apply` zakończy się więc niepowodzeniem. A zatem, jeśli zachodzi potrzeba nieustannego uaktualniania — i mamy poważną awarię — kod Terraform nie będzie działał.

Ogólnie rzecz biorąc, jak wyjaśniłem w rozdziale 3., należy zachować pełną izolację środowisk: zamiast za pomocą aliasów zarządzać wieloma regionami w jednym module, a każdym regionem zarządzać w oddzielnych modułach. W ten sposób zmniejsza się pole rażenia w razie zarówno błędów popełnionych przez Ciebie (np. przypadkowe uszkodzenie czegoś w jednym regionie prawdopodobnie nie będzie miało wpływu na inne elementy), jak i problemów w innych miejscach (np. awaria jednego regionu prawdopodobnie nie będzie miała wpływu na pozostałe regiony).

Kiedy sensowne będzie używanie aliasów? Zwykle sprawdzają się, gdy infrastruktura wdrażana przez wielu dostawców z aliasami jest prawdziwie powiązana i zawsze chcesz ją wdrażać razem. Jeśli np. jako CDN (ang. *content delivery network*) chcesz użyć Amazon CloudFront i za pomocą AWS Certification Manager (ACM) przygotować przeznaczony dla CDN certyfikat TLS, to AWS wymaga utworzenia certyfikatu w regionie us-west-1, niezależnie od tego, jakie inne

regiony będą używane z CloudFront. W takim wypadku kod może mieć dwa bloki provider: jeden przeznaczony dla regionu używanego przez CloudFront i jeden z aliasem na stałe zdefiniowanym specjalnie dla us-east-1 na potrzeby konfiguracji certyfikatu. Innym przykładem sytuacji, w której sprawdzają się aliasy, jest wdrażanie zasobów przeznaczonych do używania w wielu regionach. I tak AWS zaleca wdrażanie GuardDuty, zautomatyzowanej usługi wykrywania zagrożeń, w każdym regionie używanym w ramach konta AWS. Wówczas sensowne jest posiadanie modułu razem z blokiem provider i własnym aliasem dla każdego regionu AWS.

Poza kilkoma skrajnymi przypadkami używanie aliasów do obsługi wielu regionów zdarza się względnie rzadko. Znacznie częstszym przykładem zastosowania aliasów jest sytuacja, gdy masz wielu dostawców wymagających uaktualniania na różne sposoby, np. uaktualniania za pomocą odmiennych kont AWS.

Praca z wieloma kontami AWS

Dotychczas w książce używaliśmy pojedynczego konta AWS dla całej infrastruktury. W przypadku kodu produkcyjnego znacznie częstsze rozwiązanie polega na używaniu wielu kont AWS — umieszczenie środowiska roboczego w koncie roboczym, środowiska produkcyjnego w koncie produkcyjnym itd. Ta koncepcja ma zastosowanie także dla innych chmur, np. Azure i Google Cloud. Zwróć uwagę na użycie słowa *konto* w książce, nawet pomimo tego, że niektórzy dostawcy chmury mogą stosować nieco odmienną terminologię dla tej samej koncepcji (np. w Google Cloud mamy *projekty* zamiast kont).

Oto najważniejsze powody używania wielu kont.

Izolacja (inaczej kategoryzowanie)

Poszczególne konta są używane do odizolowania od siebie różnych środowisk i ograniczenia „pola rażenia” w razie problemów. Przykładowo umieszczenie środowisk roboczego i produkcyjnego w oddzielnych kontach gwarantuje, że jeśli komuś uda się skutecznie zaatakować środowisko robocze, nie będzie miało dostępu do środowiska produkcyjnego. Podobnie izolacja zapewnia, że programista wprowadzający zmiany w środowisku roboczym praktycznie niczego nie uszkodzi w środowisku produkcyjnym.

Uwierzytelnianie i autoryzacja

Jeżeli wszystko znajduje się w jednym koncie, wówczas trudne jest nadanie uprawnień do pewnych elementów (np. środowiska roboczego) i uniknięcie przypadkowego nadania uprawnień do innych (np. środowiska produkcyjnego). Korzystanie z wielu kont ułatwia zachowanie większej kontroli, ponieważ uprawnienia nadawane w jednym koncie nie mają wpływu na inne konta.

Wymagania uwierzytelniania w wielu kontach pomagają w ograniczeniu niebezpieczeństwa pomyłki. Gdy wszystko znajduje się w jednym koncie, zbyt łatwo jest popełnić błąd, np. gdy się sądzi, że zmiana została wprowadzona w środowisku roboczym, podczas gdy w rzeczywistości zmiana odbywa się w środowisku produkcyjnym. (Taka pomyłka może mieć katastrofalne skutki, jeśli np. polega na usunięciu wszystkich tabel bazy danych). W przypadku wielu kont natomiast popełnienie takiego błędu jest mniej prawdopodobne, a uwierzytelnianie w poszczególnych kontach wymaga oddzielnego zbioru kroków.

Pamiętaj, że wiele kont *nie* oznacza konieczności posiadania przez programistów oddzielnych profili użytkowników (np. oddzielnych użytkowników IAM w poszczególnych kontach AWS). W rzeczywistości to byłby antywzorzec, ponieważ konieczne byłoby zarządzanie wieloma zbiorami danych uwierzytelniających, uprawnieniami itd. Zamiast tego można skonfigurować po prostu wszystkie najważniejsze chmury, aby każdy programista miał dokładnie jeden profil użytkownika, który będzie mógł używać do uwierzytelniania dowolnego konta, do jakiego ma dostęp. Mechanizm uwierzytelniania między kontami różni się w zależności od używanej chmury: np. w AWS można uwierzytelniać się między kontami przez przyjęcie roli IAM, co zostanie wkrótce omówione.

Audyt i raportowanie

Poprawnie skonfigurowana struktura konta pozwala na zachowanie wyniku audytu wszystkich zmian wprowadzanych w poszczególnych środowiskach, sprawdzenie spełnienia wymagań zgodności i wykrywanie anomalii. Co więcej, można mieć skonsolidowany billing i w jednym miejscu informacje o wszystkich opłatach, przy czym koszty pozostają rozdzielone na poszczególne konta, usługi, tagi itd. Takie rozwiązanie jest szczególnie użyteczne w ogromnych organizacjach, ponieważ pozwala na śledzenie finansów i budżetów przez zespoły poprzez po prostu sprawdzenie, którego konta dotyczą dane opłaty.

Przeanalizujemy teraz przykład wielu kont, posługując się chmurą AWS. Przede wszystkim trzeba utworzyć nowe konto AWS na potrzeby testów. Ponieważ masz już jedno konto AWS, utwórz nowe *konto potomne*. W tym celu skorzystaj z sekcji *AWS Organizations* pozwalającej, aby wszystkie informacje billingowe z kont potomnych trafiały do konta nadrzędnego, czasami nazywanego *kontem głównym* (ang. *root account*). Sekcja *AWS Organizations* udostępnia również panel, który jest używany do zarządzania wszystkimi kontami potomnymi.

Przejdź do konsoli *AWS Organizations* (console.aws.amazon.com) i kliknij przycisk *Add an AWS account*, co widać na rysunku 7.3.

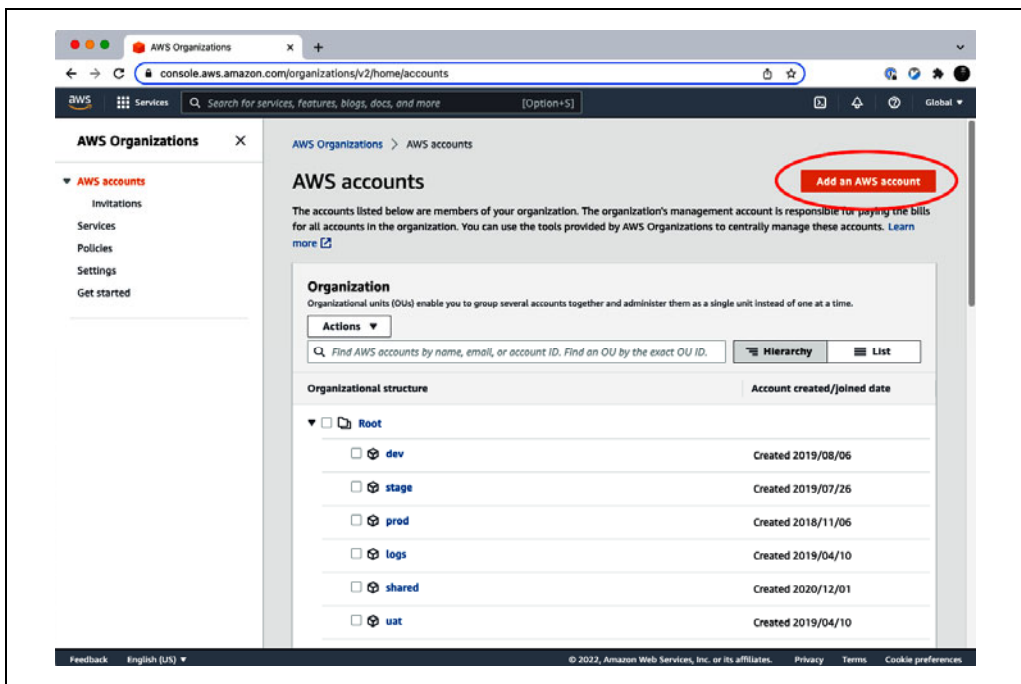
Na wyświetlonej stronie podaj niezbędne informacje, jak pokazałem na rysunku 7.4.

Nazwa konta AWS

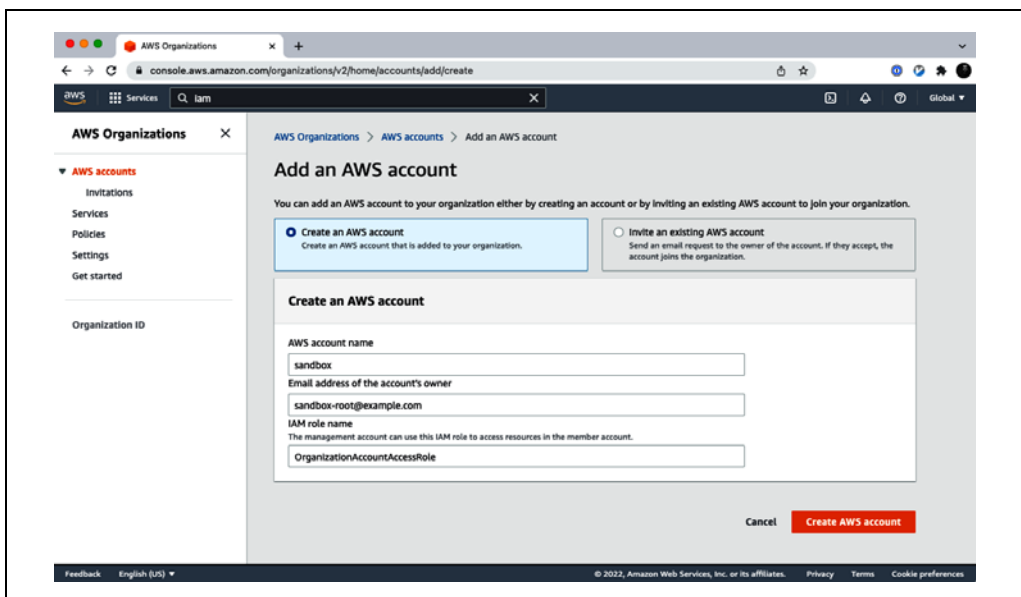
To jest nazwa używana przez konto. Jeśli np. konto jest przeznaczone dla środowiska roboczego, możesz mu nadać nazwę „robocze”.

Adres e-mail właściciela konta

To jest adres e-mail użytkownika głównego konta AWS. Pamiętaj, że każde konto AWS musi używać innego adresu e-mail dla użytkownika głównego, nie możesz więc skorzystać z adresu e-mail utworzonego dla pierwszego (głównego) konta AWS. (Zapoznaj się z ramką zatytułowaną „Jak otrzymać wiele aliasów dla adresu e-mail?” w dalszej części rozdziału, w której znajdziesz informacje, jak poradzić sobie z tym problemem). Jak wygląda kwestia hasła użytkownika głównego? Domyślnie AWS nie konfiguruje hasła dla użytkownika głównego nowego konta potomnego (wkrótce poznasz alternatywne podejście w zakresie uwierzytelniania konta potomnego). Jeżeli kiedykolwiek wystąpi konieczność zalogowania się jako ten użytkownik główny, po utworzeniu konta potomnego musisz przejść przez procedurę wyzerowania hasła i użyć podanego adresu e-mail.



Rysunek 7.3. Używanie AWS Organizations do utworzenia nowego konta AWS



Rysunek 7.4. Szczegóły dotyczące nowego konta AWS

Nazwa roli IAM

Gdy AWS Organizations tworzy nowe konto potomne AWS, automatycznie tworzy w nim również rolę IAM, która ma uprawnienia administratora i może być przyjęta przez konto nadrzędne. To jest wygodne rozwiązanie, ponieważ pozwala uwierzytelnić się w koncie potomnym AWS bez konieczności tworzenia jakichkolwiek użytkowników lub ról IAM. Zalecam pozostawienie tej roli IAM wartości domyślnej `OrganizationAccountAccessRole`.

Jak otrzymać wiele aliasów dla adresu e-mail?

Jeżeli korzystasz z poczty Gmail, możesz uzyskać wiele adresów e-mail dla pojedynczego adresu przez wykorzystanie tego, że Gmail ignoruje wszystko, co znajduje się po znaku + w adresie e-mail. Jeśli np. adres e-mail to *przyklad@gmail.com*, możesz wysłać wiadomość e-mail pod takie adresy jak *przyklad+foo@gmail.com* i *przyklad+dowolny-tekst@gmail.com*, a trafią one do skrzynki odbiorczej konta pocztowego *przyklad@gmail.com*. To działa również wtedy, gdy firma używa poczty Gmail w połączeniu z usługą Google Workspace, nawet z własną domeną, np. wiadomości wysłane pod adresy *przyklad+dev@nazwa-firmy.pl* i *przyklad+robocze@nazwa-firmy.pl* trafią do skrzynki odbiorczej konta pocztowego *przyklad@nazwa-firmy.pl*.

Taka możliwość okazuje się użyteczna podczas tworzenia wielu kont potomnych AWS, ponieważ zamiast licznych oddzielnych adresów e-mail możesz użyć np. *przyklad+dev@nazwa-firmy.pl* dla konta programistycznego, *przyklad+robocze@nazwa-firmy.pl* dla konta roboczego itd. Dla AWS to będą całkowicie unikatowe adresy e-mail, podczas gdy w rzeczywistości wiadomości będą trafiały do jednej skrzynki odbiorczej.

Kliknij przycisk *Create AWS account*, odczekaj kilka minut, aż AWS utworzy konto, a następnie zanotuj jego 12-cyfrowy identyfikator. Na pozostałą część rozdziału przyjąłem takie założenia:

- identyfikator nadrzędnego konta AWS: 111111111111,
- identyfikator potomnego konta AWS: 222222222222.

W nowym koncie potomnym możesz się uwierzytelnić za pomocą konsoli AWS, przez kliknięcie nazwy użytkownika i później przycisku *Switch role* (Zmień rolę), jak pokazałem na rysunku 7.5.

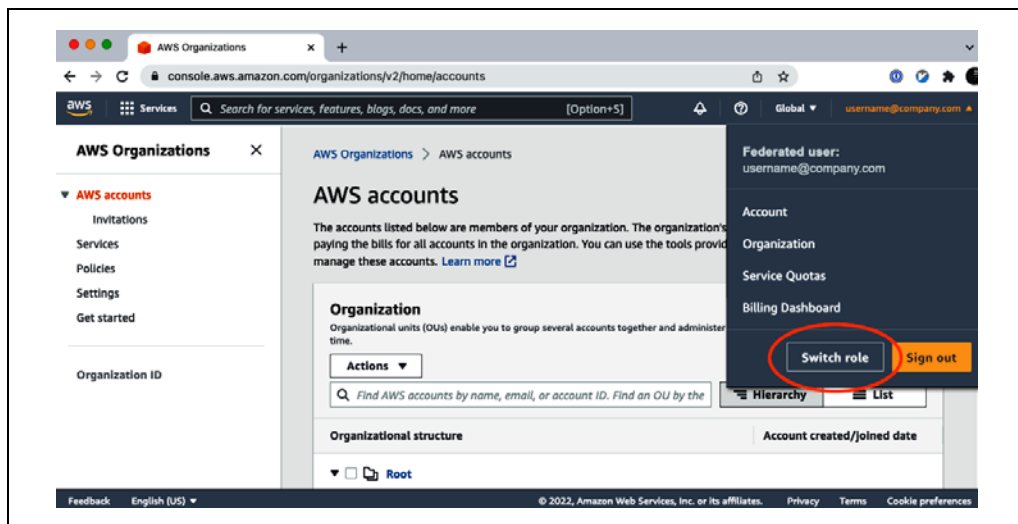
Następnie podaj szczegóły roli IAM, którą chcesz przyjąć, co widać na rysunku 7.6.

Account

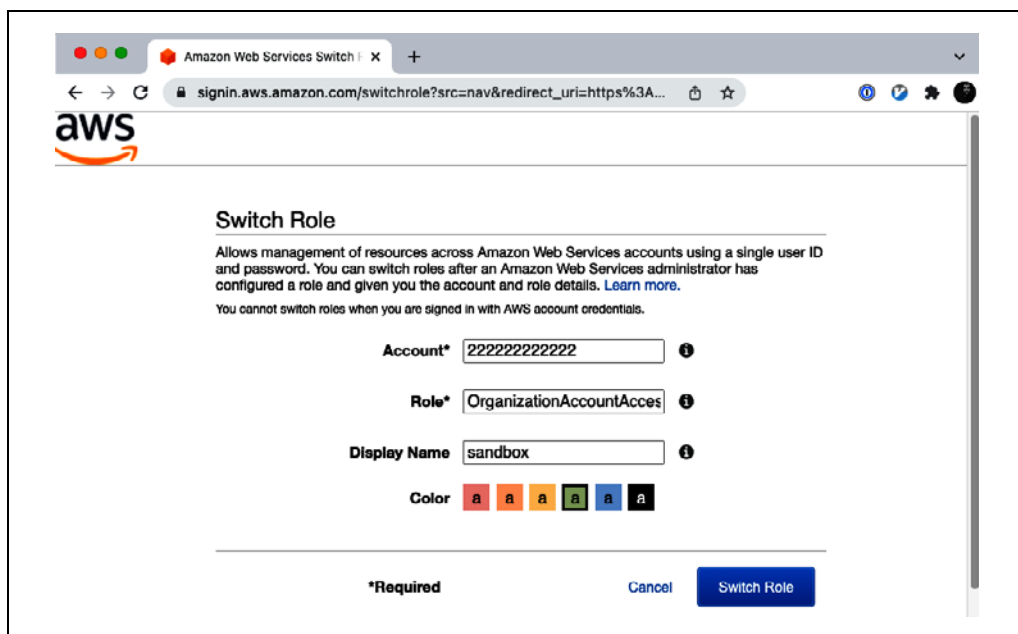
To jest 12-cyfrowy identyfikator konta AWS, do którego chcesz przejść. Możesz podać identyfikator nowo utworzonego konta potomnego.

Role

To jest nazwa roli do przyjęcia w danym koncie AWS. Podaj w tym miejscu nazwę użytą dla roli IAM podczas tworzenia nowego konta potomnego — domyślnie to `OrganizationAccountAccessRole`.



Rysunek 7.5. Przycisk Switch role



Rysunek 7.6. Podaj szczegóły roli, do której chcesz przejść

Display name

AWS wyświetli w nawigacji skrót pozwalający w przyszłości przejście do tego konta za pomocą pojedynczego kliknięcia. To jest nazwa użyta w tym skrótce. Będzie dotyczyła jedynie danego użytkownika IAM w tej przeglądarce WWW.

Kliknij przycisk *Switch Role* i gotowe — AWS zaloguje Cię w konsoli internetowej nowego konta AWS.

Przystępujemy teraz do utworzenia przykładowego modułu Terraform (*examples/multi-account-root*) pozwalającego na uwierzytelnianie w wielu kontach AWS. Podobnie jak w przykładzie dotyczącym wielu regionów AWS, także tu konieczne jest dodanie dwóch bloków provider w pliku *main.tf*, każdy z innym aliasem. Pierwszy blok provider jest przeznaczony dla nadrzędnego konta AWS.

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

Z kolei drugi jest przeznaczony dla potomnego konta AWS.

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

Aby mieć możliwość uwierzytelnienia w potomnym koncie AWS, trzeba przyjąć rolę IAM. W konsoli internetowej można to zrobić przez kliknięcie przycisku *Switch Role*. W kodzie Terraform natomiast potrzebny jest blok `assume_role` dodany do bloku provider potomnego konta AWS.

```
provider "aws" {  
  region = "us-east-2"  
  alias   = "child"  
  
  assume_role {  
    role_arn = "arn:aws:iam::<IDENTYFIKATOR_KONTA>:role/<NAZWA_ROLI>"  
  }  
}
```

W parametrze `role_arn` wartość `IDENTYFIKATOR_KONTA` trzeba zastąpić identyfikatorem konta potomnego, a `NAZWA_ROLI` nazwą roli IAM w tym koncie, podobnie jak podczas przełączania ról w konsoli internetowej. Zobacz, jak ten fragment kodu wygląda w wypadku konta o identyfikatorze `222222222222` i roli o nazwie `OrganizationAccountAccessRole`:

```
provider "aws" {  
  region = "us-east-2"  
  alias   = "child"  
  
  assume_role {  
    role_arn = "arn:aws:iam::222222222222:role/OrganizationAccountAccessRole"  
  }  
}
```

Teraz w celu sprawdzenia, czy to rozwiązanie faktycznie działa, dodaj dwa źródła danych `aws_caller_identity` i każde z nich skonfiguruj z użyciem innego dostawcy.

```
data "aws_caller_identity" "parent" {  
  provider = aws.parent  
}  
  
data "aws_caller_identity" "child" {  
  provider = aws.child  
}
```

W pliku *outputs.tf* dodaj dwie zmienne danych wyjściowych w celu wyświetlenia identyfikatorów kont.

```
output "parent_account_id" {
  value      = data.aws_caller_identity.parent.account_id
  description = "Identyfikator konta nadrzędnego AWS"
}

output "child_account_id" {
  value      = data.aws_caller_identity.child.account_id
  description = "Identyfikator konta potomnego AWS"
}
```

Po wykonaniu polecenia `terraform apply` zobaczysz różne identyfikatory dla poszczególnych kont.

```
$ terraform apply
```

```
(...)
```

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

```
Outputs:
```

```
parent_account_id = "111111111111"
child_account_id  = "222222222222"
```

W ten sposób dzięki użyciu aliasów dostawców i bloków `assume_role` wiesz, jak można tworzyć kod Terraform działający w różnych kontach AWS.

Podobnie jak w wypadku pracy z wieloma regionami, także tutaj warto pamiętać o dwóch ostrzeżeniach.

Ostrzeżenie 1.: Działające dla wielu kont role IAM to rodzaj podwójnego opt-in

Aby rola IAM pozwalała na uzyskanie dostępu do konta AWS z poziomu innego konta — rola IAM w koncie o identyfikatorze 222222222222 może być przyjęta przez konto o identyfikatorze 111111111111 — konieczne jest nadanie uprawnień w *obu* kontach:

- Po pierwsze, konto AWS zawierające tę rolę IAM (np. konto potomne 222222222222) musi mieć skonfigurowaną politykę przyjęcia roli w taki sposób, aby to drugie konto (np. nadrzędne 111111111111) było uznawane za zaufane. Tak się dzieje magicznie w wypadku roli `OrganizationAccountAccessRole`, ponieważ AWS Organizations automatycznie konfiguruje politykę przyjęcia roli i zaufanie dla konta nadrzędnego. Jednak w każdej niestandardowej roli IAM, którą utworzysz, musisz pamiętać o wyraźnym nadaniu uprawnień `sts:AssumeRole`.
- Po drugie, w koncie AWS, z którego przyjmowana jest rola (np. konto nadrzędne 111111111111), konieczne jest *również* nadanie uprawnień użytkownika w celu przyjęcia tej roli IAM. I to dzieje się magicznie, ponieważ w rozdziale 2. użytkownikowi IAM zostały nadane uprawnienia `AdministratorAccess`, które zapewniają uprawnienia do praktycznie wszystkich zadań w nadrzędnym koncie AWS, czyli także do przyjmowania ról IAM. W większości rzeczywistych przypadków użytkownik nie będzie (i nie powinien być!) administratorem. To oznacza konieczność wyraźnego nadania użytkownikowi uprawnień `sts:AssumeRole` do roli IAM, którą powinien mieć możliwość przyjąć.

Ostrzeżenie 2.: Oszczędnie korzystaj z aliasów

Wspomniałem o tym w przykładzie dotyczącym wielu regionów, ale warto przypomnieć ponownie. Korzystanie z aliasów w Terraform jest bardzo łatwe, mimo to odradzam ich zbyt częste używanie, także podczas tworzenia kodu obsługującego liczne konta. Wiele kont tworzy się głównie w celu zapewnienia separacji między nimi, jeśli więc coś złego stanie się w jednym koncie, to nie będzie miało wpływu na inne. Moduły wdrażane w wielu kontach są sprzeczne z tą zasadą. Decyduj się na takie rozwiązanie tylko wtedy, gdy *celowo* chcesz mieć zasoby w wielu kontach połączone ze sobą i wdrożone razem.

Tworzenie modułów, które mogą działać z wieloma dostawcami

Podczas pracy z modułami Terraform zwykle masz styczność z dwoma typami modułów:

Moduły wielokrotnego użycia

To są moduły niskiego poziomu, nieprzeznaczone do bezpośredniego wdrożenia, a zamiast tego łączone z innymi modułami, zasobami i źródłami danych.

Moduły główne

To są moduły wysokiego poziomu łączące wiele modułów wielokrotnego użycia w pojedynczą jednostkę, która jest przeznaczona do wdrożenia bezpośrednio przez zastosowanie polecenia `terraform apply` (w rzeczywistości definicja modułu głównego jest taka: to moduł, w którym można użyć polecenia `terraform apply`).

Zaprezentowane dotychczas przykłady wykorzystujące wielu dostawców opierały się na umieszczeniu wszystkich bloków `provider` w module głównym. Co można zrobić, gdy chce się utworzyć moduł wielokrotnego użycia przeznaczony do działania z wieloma dostawcami? Co zrobić np. w sytuacji, gdy przedstawiony w poprzednim punkcie kod obsługi wielu kont chce się umieścić w module wielokrotnego użycia? Pierwszym krokiem może być pobranie całego kodu i umieszczenie niezmiennego w katalogu `modules/multi-account`. Następnie można utworzyć nowy przykład w celu przetestowania kodu — w katalogu `examples/multi-account-module` możesz umieścić plik `main.tf` o takiej zawartości:

```
module "multi_account_example" {  
  source = "../modules/multi-account"  
}
```

Jeżeli uruchomisz ten kod, będzie działał, choć pojawi się pewien problem: wszystkie bloki konfiguracyjne `provider` są teraz ukryte w samym module (`modules/multi-account`). Zdefiniowanie bloków `provider` w module wielokrotnego użycia jest antywzorcem z kilku powodów.

Problemy z konfiguracją

Jeżeli bloki `provider` są zdefiniowane w module wielokrotnego użycia, dany moduł będzie kontrolował całą konfigurację dla danego dostawcy. Przykładowo przeznaczona do użycia wartość `ARN` i `region` roli `IAM` są obecnie na stałe zdefiniowane w module `modules/multi-account`. Oczywiście można udostępnić zmienne danych wejściowych pozwalające użytkownikom na zdefiniowanie regionu i wartości `ARN` roli, ale to jedynie wierzchołek góry lodowej. Jeżeli zajrzysz do dokumentacji dostawcy `AWS`, znajdziesz tam niemalże 50 różnych opcji

konfiguracyjnych, które można mu przekazać! Wiele tych parametrów będzie ważnych dla użytkowników modułu, ponieważ kontrolują sposób uwierzytelniania AWS, używany region, używane konto (lub rolę IAM), punkty końcowe używane podczas prowadzenia komunikacji z AWS, tagi przeznaczone do stosowania lub ignorowania, a także znacznie więcej. Konieczność udostępnienia tych 50 dodatkowych zmiennych w module może spowodować, że stanie się on bardzo trudny w użyciu i obsłudze.

Problemy z powielaniem kodu

Jeżeli wspomniane 50 ustawień lub ich część udostępnisz w module wcześniej, w ten sposób doprowadzisz do powielenia kodu dla użytkowników modułu. To wynika z tego, że dość często łączy się ze sobą wiele modułów i jeśli w każdym z nich musisz przekazać podzbiór 50 ustawień, aby zapewnić poprawne uwierzytelnienie, musisz kopiować i wklejać wiele parametrów, co będzie żmudne i podatne na błędy.

Problemy z wydajnością działania

Za każdym razem, gdy w kodzie umieścisz blok `provider`, Terraform uruchamia nowy proces przeznaczony do działania tego dostawcy i komunikuje się z tym procesem za pomocą RPC. Jeżeli masz kilka bloków `provider`, takie rozwiązanie po prostu działa. Po jego skalowaniu w górę natomiast mogą pojawiać się problemy związane z wydajnością działania. Oto rzeczywisty przykład: kilka lat temu utworzyłem wielokrotnego użycia moduł dla CloudTrail, AWS Config, GuardDuty, IAM Access Analyzer i Macie. Każda z tych usług AWS miała być wdrażana we wszystkich regionach konta AWS (jest ich około 25). Dlatego w każdym z modułów umieściłem 25 bloków `provider`. Następnie utworzyłem pojedynczy moduł główny w celu wdrożenia ich wszystkich jako „punktu wyjścia” w moich kontach AWS. Jeżeli pomnożysz 5 modułów przez 25 bloków `provider` w każdym z nich, otrzymasz w sumie 125 bloków `provider`. Po użyciu polecenia `terraform apply` Terraform uruchomił 125 procesów, z których każdy wykonywał setki wywołań API i RPC. W przypadku tysięcy jednoczesnych żądań sieciowych procesor w moim komputerze zaczął mieć trudności z ich obsługą i wykonanie zwykłego polecenia `terraform plan` mogło zabrać 20 minut. Co gorsze, obciążana jest również sieć, co prowadzi do problemów w wywołaniach API i polecenie `terraform apply` czasami kończy się niepowodzeniem.

Dlatego też najlepszą praktyką jest, aby nigdy *nie* definiować żadnych bloków `provider` w modułach wielokrotnego użycia. Zamiast tego należy pozostawić użytkownikom tworzenie w module głównym niezbędnych im bloków `provider`. Jak jednak wówczas utworzyć moduł, który będzie mógł współdziałać z wieloma dostawcami? Jeżeli moduł nie zawiera bloku `provider`, to jak można prawidłowo zdefiniować aliasy, do których będzie się można później odwoływać w zasobach i źródłach danych?

Rozwiązaniem jest używanie *aliasów konfiguracji*. Są bardzo podobne do przedstawionych wcześniej aliasów dostawcy, ale nie zostały zdefiniowane w blokach `provider`. Zamiast tego są definiowane w blokach `required_providers`.

Otwórz plik `modules/multi-account/main.tf`, usuń zagnieżdżone bloki `provider` i zastąp je blokiem `required_providers` o takiej konfiguracji:

```
terraform {  
  required_providers {
```

```

aws = {
  source          = "hashicorp/aws"
  version         = "~> 4.0"
  configuration_aliases = [aws.parent, aws.child]
}
}

```

Podobnie jak w wypadku zwykłych aliasów dostawców, także aliasy konfiguracji można przekazywać do zasobów i źródeł danych za pomocą parametru `provider`.

```

data "aws_caller_identity" "parent" {
  provider = aws.parent
}

data "aws_caller_identity" "child" {
  provider = aws.child
}

```

Względem zwykłego aliasu dostawcy podstawowa różnica polega na tym, że alias konfiguracji nie tworzy żadnych dostawców. Zamiast tego wymusza na użytkownikach modułu wyraźne przekazanie za pomocą mapowania `providers` dostawcy dla każdego aliasu konfiguracji.

Otwórz plik `examples/multi-account-module/main.tf` i zdefiniuj blok `provider`, podobnie jak wcześniej.

```

provider "aws" {
  region = "us-east-2"
  alias  = "parent"
}

provider "aws" {
  region = "us-east-2"
  alias  = "child"

  assume_role {
    role_arn = "arn:aws:iam::222222222222:role/OrganizationAccountAccessRole"
  }
}

```

Teraz możesz w pokazany tutaj sposób przekazać te bloki do modułu `modules/multi-account`:

```

module "multi_account_example" {
  source = "../../modules/multi-account"
  providers = {
    aws.parent = aws.parent
    aws.child  = aws.child
  }
}

```

Klucze mapowania `providers` muszą odpowiadać nazwą aliasów konfiguracji w module. Jeżeli w mapowaniu brakuje nazwy dla któregośkolwiek aliasu konfiguracji, Terraform wygeneruje komunikat błędu. W ten sposób podczas tworzenia modułu wielokrotnego użycia można zdefiniować dostawców wymaganych przez moduł, a Terraform zagwarantuje, że użytkownicy będą przekazywać niezbędnych dostawców. Z kolei podczas tworzenia modułu głównego możesz zdefiniować bloki `provider` tylko raz, a następnie odniesienia do nich przekazywać do modułów wielokrotnego użycia.

Praca z wieloma różnymi dostawcami

We wcześniejszej części rozdziału wyjaśniłem, jak pracować z wieloma dostawcami, gdy wszyscy są tego samego typu: np. wiele kopii dostawcy aws. W tym podrozdziale pokażę, jak pracować z wieloma różnymi dostawcami.

Czytelnicy dwóch pierwszych wydań książki często pytali o przykłady użycia wielu różnych chmur ze sobą (*multicloud*), ale nie byłem w stanie zaproponować im niczego użytecznego. Po części to wynikało z tego, że jednoczesne używanie wielu chmur najczęściej jest złym pomysłem². Nawet jeśli coś Cię do tego zmusi (większość ogromnych firm używa wielu chmur, bez względu na to, czy tego chcą), rzadko się zdarza konieczność zarządzania wieloma chmurami w jednym module, dokładnie z tego samego powodu, dla którego w jednym module rzadko zarządza się wieloma regionami lub kontami. W sytuacji gdy używasz wielu chmur, lepiej będzie, jeśli każdą z nich będziesz zarządzać w oddzielnym module.

Co więcej, konwersja każdego przykładu AWS w książce na odpowiadające mu rozwiązanie w innej chmurze (Azure i Google Cloud) jest niepraktyczna: książka stałaby się zbyt obszerna i mówiłaby wiele o poszczególnych chmurach, niewiele zaś o nowych koncepcjach Terraform, a przecież to jest jej tematem. Jeżeli chcesz zobaczyć przykłady kodu Terraform dla podobnej infrastruktury utworzonej za pomocą innych chmur, zajrzyj do katalogu *examples* w repozytorium Terraform (<https://github.com/gruntwork-io/terratest>). Jak się dowiesz z rozdziału 9., Terratest dostarcza zbiór narzędzi przeznaczonych do tworzenia testów zautomatyzowanych dla różnych typów infrastruktury kodu i różnych typów chmury. Dlatego też w katalogu *examples* wymienionego repozytorium znajdziesz kod Terraform przeznaczony dla podobnej infrastruktury w AWS, Google Cloud i Azure, m.in. poszczególne serwery, grupy serwerów, bazy danych itd. Ponadto w katalogu *test* znajdziesz testy zautomatyzowane dla tych wszystkich przykładów.

W książce zamiast nierzeczywistego przykładu multicloud zdecydowałem się na pokazanie, jak można używać jednocześnie wielu różnych dostawców, w nieco bardziej realnym scenariuszu (oczekiwanym przez wielu czytelników dwóch pierwszych wydań książki): jak używać dostawcy AWS w połączeniu z dostawcą Kubernetes, aby wdrażać aplikacje w kontenerach Dockera. Pod wieloma względami Kubernetes jest chmurą samą w sobie — pozwala uruchamiać aplikacje, sieci, magazyny danych, mechanizmy równoważenia obciążenia, magazyny danych poufnych itd. Dlatego też w pewnym sensie to przykład rozwiązania obejmującego wielu dostawców i wiele chmur. Skoro Kubernetes jest chmurą, to oznacza sporo nauki. Zamierzam więc wyjaśniać wszystko po kolei. Rozpocznę od krótkiego wprowadzenia do Dockera i Kubernetes, a później przejdę do pełnego przykładu z wykorzystaniem AWS i Kubernetes:

- Krótkie wprowadzenie do Dockera.
- Krótkie wprowadzenie do Kubernetes.
- Wdrażanie kontenerów Dockera w AWS za pomocą EKS (Elastic Kubernetes Service).

² Zapoznaj się z artykułem *Multi-Cloud is the Worst Practice* na stronie <https://www.lastweekinaws.com/blog/multi-cloud-is-the-worst-practice/>.

Krótkie wprowadzenie do Dockera

Jak pamiętasz z rozdziału 1., obraz Dockera jest samodzielną „migawką” systemu operacyjnego (ang. *operating system*, OS), oprogramowania, plików i innych ważnych szczegółów. W tym punkcie zobaczysz Dockera w akcji.

Jeżeli jeszcze nie masz zainstalowanego Dockera, przeprowadź procedurę omówioną na stronie <https://docs.docker.com/get-docker/> w celu zainstalowania oprogramowania Docker Desktop w Twoim systemie operacyjnym. Po instalacji będziesz mieć dostępne polecenie `docker`. Za pomocą polecenia `docker run` możesz lokalnie uruchamiać obrazy Dockera:

```
$ docker run <OBRAZ> [POLECENIE]
```

gdzie `OBRAZ` to obraz Dockera przeznaczony do uruchomienia, a `POLECENIE` to opcjonalne polecenie do wykonania. Dla przykładu zobacz, w jaki sposób możesz uruchomić powłokę Bash w obrazie Dockera zawierającym Ubuntu 20.04 (zwróć uwagę, że to polecenie zawiera opcję `-it`, otrzymasz więc powłokę interaktywną, w której możesz wydawać polecenia).

```
$ docker run -it ubuntu:20.04 bash
```

```
Unable to find image 'ubuntu:20.04' locally
20.04: Pulling from library/ubuntu
Digest: sha256:669e010b58baf5beb2836b253c1fd5768333f0d1dbcb834f7c07a4dc93f474be
Status: Downloaded newer image for ubuntu:20.04
```

```
root@d96ad3779966:/#
```

Voilà, teraz jesteś w Ubuntu. Jeżeli nigdy wcześniej nie zdarzyło Ci się używać Ubuntu, to może się wydawać magią. Spróbuj wykonać kilka poleceń. Sprawdź np. zawartość pliku `/etc/os-release`, aby w ten sposób potwierdzić, że faktycznie korzystasz z Ubuntu:

```
root@d96ad3779966:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.3 LTS (Focal Fossa)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04.3 LTS"
VERSION_ID="20.04"
VERSION_CODENAME=focal
```

Jak to możliwe? Przede wszystkim w lokalnym systemie plików Docker poszukuje obrazu `ubuntu:20.04`. Jeżeli nie masz go pobranego, Docker automatycznie pobierze go z serwisu Docker Hub, czyli z **rejestru Dockera** zawierającego udostępnione obrazy. Obraz o nazwie `ubuntu:20.04` jest publicznym obrazem Dockera — oficjalnie obsługiwanym przez tworzący go zespół — możesz więc go pobrać bez konieczności jakiegokolwiek uwierzytelniania. Istnieje również możliwość tworzenia prywatnych obrazów Dockera, które mogą być używane tylko przez uwierzytelnionych użytkowników.

Po pobraniu obrazu Docker go uruchamia i wykonuje polecenie `bash` — to jest interaktywna powłoka Bash, w której można wydawać polecenia. Użyj polecenia `ls`, a otrzymasz listę plików.

```
root@d96ad3779966:/# ls -al
total 56
drwxr-xr-x  1 root root 4096 Feb 22 14:22 .
drwxr-xr-x  1 root root 4096 Feb 22 14:22 ..
```

```

lrwxrwxrwx 1 root root 7 Jan 13 16:59 bin -> usr/bin
drwxr-xr-x 2 root root 4096 Apr 15 2020 boot
drwxr-xr-x 5 root root 360 Feb 22 14:22 dev
drwxr-xr-x 1 root root 4096 Feb 22 14:22 etc
drwxr-xr-x 2 root root 4096 Apr 15 2020 home
lrwxrwxrwx 1 root root 7 Jan 13 16:59 lib -> usr/lib
drwxr-xr-x 2 root root 4096 Jan 13 16:59 media
(...)

```

Zauważysz, że to nie jest Twój system plików. Obraz Dockera jest uruchomiony w kontenerze odizolowanym od przestrzeni użytkownika: gdy znajdujesz się w kontenerze, masz dostęp jedynie do jego systemu plików, pamięci, sieci itd. Wszelkie dane w innych kontenerach czy w systemie operacyjnym gospodarza są niedostępne, wszystkie dane kontenera zaś są niedostępne dla innych kontenerów lub systemu operacyjnego gospodarza. To jest cecha, dzięki której Docker świetnie sprawdza się podczas uruchamiania aplikacji: format obrazu jest samodzielny i dlatego obrazy Dockera działają w dokładnie taki sam sposób niezależnie od miejsca ich uruchomienia i niezależnie od tego, co jeszcze zostało uruchomione.

Aby zobaczyć to w akcji, zapisz dowolny tekst w pliku *test.txt* za pomocą tego polecenia:

```
root@d96ad3779966:/# echo "Witaj, świecie!" > test.txt
```

Teraz opuść kontener przez naciśnięcie klawiszy *Ctrl+D* (Windows i Linux) lub *Cmd+D* (macOS), co spowoduje powrót do powłoki systemu operacyjnego komputera gospodarza. Jeżeli będziesz szukać utworzonego przed chwilą pliku *test.txt*, nie znajdziesz go: istnieje on tylko w systemie plików kontenera, który jest zupełnie odizolowany od systemu operacyjnego gospodarza.

Ponownie spróbuj uruchomić ten sam obraz Dockera:

```
$ docker run -it ubuntu:20.04 bash
root@3e0081565a5d:/#
```

Zauważ, że ponieważ obraz *ubuntu:20.04* został wcześniej pobrany, tym razem kontener jest uruchamiany niemal natychmiast. To kolejny powód, dla którego Docker jest użyteczny podczas uruchamiania aplikacji — w przeciwieństwie do maszyn wirtualnych kontenery są lekkie, uruchamiają się szybko, a także potrzebują niewielkiej ilości pamięci i mocy obliczeniowej procesora.

Być może zwróciło Twoją uwagę to, że po drugim uruchomieniu kontenera znak zachęty wygląda nieco inaczej. W tym momencie jesteś w zupełnie nowym kontenerze, a wszelkie dane zapisane w poprzednim nie są dłużej potrzebne. Po użyciu polecenia *ls -al* zobaczysz, że utworzony wcześniej plik *test.txt* nie istnieje. Kontenery są odizolowane nie tylko od systemu operacyjnego gospodarza, ale i od siebie.

Ponownie naciśnij klawisze *Ctrl+D* lub *Cmd+D* w celu zamknięcia kontenera. W ten sposób powrócisz do systemu operacyjnego gospodarza. Zastosuj w nim polecenie *docker ps -a*.

```
$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
3e0081565a5d	ubuntu:20.04	"bash"	5 min ago	Exited (0) 16 sec ago
d96ad3779966	ubuntu:20.04	"bash"	14 min ago	Exited (0) 5 min ago

Zobaczysz wszystkie kontenery w systemie, również zatrzymane (czyli te, które zostały opuszczone). Zatrzymany wcześniej kontener możesz ponownie uruchomić za pomocą polecenia *docker start*

<ID>, gdzie ID to identyfikator odczytany z kolumny `CONTAINER ID` w danych wyjściowych polecenia `docker ps`. Dla przykładu zobacz, jak możesz ponownie uruchomić pierwszy kontener (i dołączyć powłokę interaktywną za pomocą opcji `-ia`):

```
$ docker start -ia d96ad3779966
root@d96ad3779966:/#
```

Aby potwierdzić, że faktycznie znajdujesz się w pierwszym uruchomionym wcześniej kontenerze, wyświetl zawartość pliku `test.txt`:

```
root@d96ad3779966:/# cat test.txt
Witaj, świecie!
```

Teraz pokażę, jak można użyć kontenera do uruchomienia aplikacji internetowej. Naciśnij klawisze `Ctrl+D` lub `Cmd+D` w celu zamknięcia kontenera. W ten sposób powrócisz do systemu operacyjnego gospodarza — tu uruchom nowy kontener.

```
$ docker run training/webapp
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Obraz `training/webapp` (<https://github.com/docker-training/webapp>) zawiera utworzoną w Pythonie prostą aplikację internetową typu „Witaj, świecie!”, przeznaczoną na potrzeby testów. Po uruchomieniu kontenera następuje uruchomienie aplikacji internetowej, która nasłuchuje żądań na porcie 5000. Jeżeli w systemie operacyjnym gospodarza przejdiesz do nowego okna powłoki i spróbujesz uzyskać dostęp do tej aplikacji, zobaczysz, że nie działa.

```
$ curl localhost:5000
curl: (7) Failed to connect to localhost port 5000: Connection refused
```

W czym problem? W rzeczywistości to nie problem, a sposób działania aplikacji! Kontenery Dockera są odizolowane od systemu operacyjnego gospodarza i od innych kontenerów nie tylko na poziomie systemu operacyjnego, ale również w kategoriach sieci. Dlatego wprowadź kontener nasłuchuje na porcie 5000, ale tylko *wewnątrz* kontenera, który jest niedostępny w systemie operacyjnym gospodarza. Jeżeli w systemie operacyjnym gospodarza chcesz udostępnić port kontenera, musisz użyć opcji `-p`.

Zacznij od naciśnięcia klawiszy `Ctrl+C` w celu wyłączenia kontenera `training/webapp`. Zwróć uwagę, że tym razem to klawisz `C`, a nie `D`, niezależnie od systemu operacyjnego, ponieważ kończysz działanie procesu, a nie opuszczasz powłoki interaktywnej. Teraz ponownie uruchom kontener, ale tym razem z użyciem opcji `-p`, jak pokazałem w kolejnym poleceniu:

```
$ docker run -p 5000:5000 training/webapp
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Dodanie opcji `-p 5000:5000` nakazuje Dockerowi udostępnienie portu 5000 kontenera jako portu 5000 w systemie operacyjnym gospodarza. W innym oknie powłoki systemu operacyjnego będzie można potwierdzić działanie aplikacji internetowej:

```
$ curl localhost:5000
Hello world!
```



Sprzątanie po kontenerach

Po każdym wykonaniu polecenia `docker run` pozostawiasz kontenery, które zajmują nieco miejsca na dysku. Prawdopodobnie chcesz się ich pozbyć za pomocą polecenia `docker rm <ID_KONTENERA>`, gdzie `ID_KONTENERA` to identyfikator danego kontenera wyświetlony przez polecenie `docker ps`. Ewentualnie możesz dodać opcję `--rm` do polecenia `docker run`, aby kontener został usunięty automatycznie po tym, jak praca z nim zostanie zakończona.

Krótkie wprowadzenie do Kubernetes

Kubernetes to narzędzie instrumentacji dla Dockera, co oznacza, że jest platformą przeznaczoną do uruchamiania i zarządzania kontenerami Dockera w serwerach. Oferuje przy tym spore możliwości, m.in. obsługę harmonogramu (określenie, które serwery powinny działać w przypadku danego obciążenia), automatyczne uzdrawianie (automatyczne ponowne wdrażanie kontenerów), automatyczne skalowanie (skalowanie w górę lub w dół liczby kontenerów w odpowiedzi na aktualne obciążenie), równoważenie obciążenia (rozkład ruchu sieciowego między kontenerami) itd.

W tle Kubernetes składa się z dwóch podstawowych elementów:

Plaszczyzna kontrolna

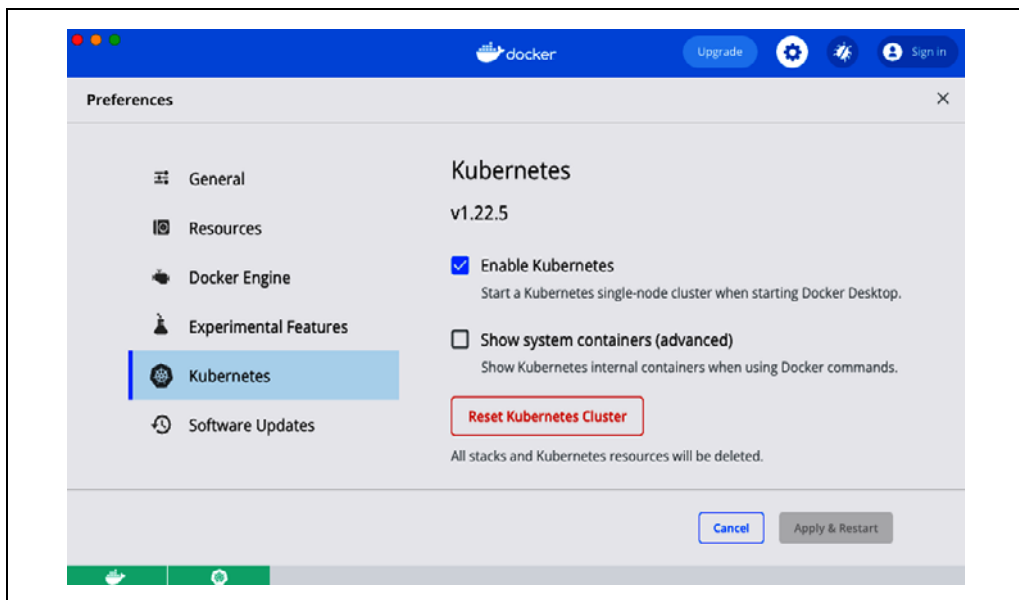
Plaszczyzna kontrolna jest odpowiedzialna za zarządzanie klastrem Kubernetes. To jest „mózg” operacji, odpowiada za przechowywanie stanu klastra, monitorowanie kontenerów i koordynację działań w klastrze. Gwarantuje działanie API serwera. Ponadto dostarcza API przeznaczone do użycia z poziomu narzędzi powłoki (np. `kubectl`), bazujący na przeglądarce WWW interfejs użytkownika (np. Kubernetes Dashboard), a także narzędzia IaC (np. Terraform) służące do kontrolowania tego, co się dzieje w klastrze.

Węzły robocze

Węzły robocze to serwery, w których faktycznie są uruchamiane kontenery. Węzły robocze są w całości zarządzane przez plaszczyznę kontrolną. Wskazuje ona poszczególnym węzłom, które kontenery powinny być uruchamiane.

Kubernetes to oprogramowanie typu open source, a jedną z jego mocnych stron jest możliwość uruchomienia wszędzie: w dowolnej publicznej chmurze (np. AWS, Azure, Google Cloud), w Twoim własnym centrum danych, a nawet w stacji roboczej programisty. Nieco dalej w rozdziale pokażę, jak można uruchamiać Kubernetes w chmurze (w AWS), zaczniemy natomiast od nieco prostszego i mniejszego rozwiązania — lokalnie działającego Kubernetes. To jest łatwe rozwiązanie, o ile masz zainstalowaną względnie nową wersję oprogramowania Docker Desktop, ponieważ wówczas możliwość lokalnego uruchomienia klastra Kubernetes to kwestia zaledwie kilku kliknięć myszą.

Jeżeli przejdziesz do preferencji Docker Desktop, po lewej stronie okna znajdziesz sekcję dotyczącą Kubernetes, jak pokazałem na rysunku 7.7.



Rysunek 7.7. Włączenie Kubernetes w preferencjach oprogramowania Docker Desktop

Jeżeli pole wyboru *Enable Kubernetes* nie zostało zaznaczone, zaznacz je, a następnie kliknij przycisk *Apply & Restart*. Poczekaj kilka minut na dokończenie operacji. W tym czasie zapoznaj się z zamieszczonymi w witrynie internetowej Kubernetes informacjami (<https://kubernetes.io/docs/tasks/tools/>) na temat instalacji narzędzia w postaci polecenia `kubectl`, które służy w powłoce do pracy z Kubernetes.

Aby używać polecenia `kubectl`, trzeba najpierw uaktualnić jego plik konfiguracyjny, którym jest `$HOME/.kube/config` (tzn. plik `config` w katalogu `.kube` w katalogu domowym użytkownika). W tym pliku należy wskazać klastę Kubernetes, z którym będzie nawiązywane połączenie. Podczas włączania Kubernetes w Docker Desktop ten plik konfiguracyjny jest automatycznie uaktualniany przez dodanie do niego sekcji `docker-desktop`. Dlatego też musisz jedynie wskazać `kubectl`, że chcesz korzystać z tej konfiguracji:

```
$ kubectl config use-context docker-desktop
Switched to context "docker-desktop".
```

Teraz za pomocą polecenia `kubectl get nodes` możesz sprawdzić, czy klastę Kubernetes działa:

```
$ kubectl get nodes
NAME                STATUS    ROLES                  AGE   VERSION
docker-desktop      Ready     control-plane,master   95m   v1.22.5
```

Polecenie `kubectl get nodes` wyświetla informacje o wszystkich węzłach klastra. Skoro Kubernetes działa lokalnie, Twój komputer jest jedynym węzłem, w którym zostały uruchomione płaszczyna kontrolna i węzeł roboczy. Możesz przystąpić do uruchamiania pewnych kontenerów Dockera!

Aby wdrożyć cokolwiek w Dockerze, tworzysz *obiekty* Kubernetes, które są trwałymi encjami zapisywanymi w klastrze Kubernetes (za pomocą API serwera) i rejestrującymi Twoje intencje: np. uruchomienie określonego obrazu Dockera. Klastę działa w tzw. **pętli**, która nieustannie sprawdza przechowywane obiekty i stara się, aby jego stan był zgodny z intencjami użytkownika.

Dostępnych jest wiele różnych typów obiektów Kubernetes. W książce np. będą użyte dwa typy:

Kubernetes Deployment

Typ obiektu *Kubernetes Deployment* przedstawia deklaracyjny sposób zarządzania aplikacją w Kubernetes. Deklarujesz obiekty Dockera przeznaczone do uruchomienia, podajesz ich liczbę (nazywaną *replikami*), a także różne ustawienia tych obrazów (np. procesor, pamięć, numery portów, zmienne środowiskowe) i strategię wprowadzania w nich uaktualnień, a Kubernetes Deployment będzie działał w taki sposób, aby zawsze zagwarantować spełnienie zadeklarowanych wymagań. Jeśli np. określisz, że chcesz mieć trzy repliki, a jeden węzeł roboczy ulegnie awarii i pozostaną tylko dwie repliki, to obiekt Deployment automatycznie uruchomi trzecią replikę w jednym z węzłów roboczych.

Kubernetes Service

Typ obiektu *Kubernetes Service* to sposób na wyrażenie aplikacji internetowej uruchomionej w Kubernetes jako usługi sieciowej. Przykładowo z Kubernetes Service można skorzystać, by skonfigurować mechanizm równoważenia obciążenia, który udostępni publiczny punkt końcowy i będzie przekazywać z niego ruch sieciowy do replik w Kubernetes Deployment.

Idiomatyczny sposób pracy z Kubernetes polega na utworzeniu pliku YAML określającego wymagania — np. jednego pliku YAML definiującego Kubernetes Deployment i innego definiującego Kubernetes Service — a następnie na użyciu polecenia `kubectl apply` w celu przekazania tych obiektów do klastra. Jednak korzystanie z niezmodyfikowanych plików YAML ma pewne wady, takie jak brak możliwości wielokrotnego użycia kodu (np. zmiennych, modułów), abstrakcji (np. pętle i konstrukcje warunkowe), czystych standardów dotyczących przechowywania plików YAML i zarządzania nimi (np. śledzenie zmian w klastrze na przestrzeni czasu) itd. Dlatego też wielu użytkowników zwraca się w kierunku alternatyw, takich jak Helm i Terraform. Skoro to jest książka poświęcona Terraform, pokażę, jak utworzyć moduł Terraform o nazwie `k8s-app` (K8S to akronim oznaczający Kubernetes, podobnie jak I18N oznacza Internationalization), pozwalający na wdrożenie aplikacji w Kubernetes za pomocą Kubernetes Deployment i Kubernetes Service.

Utwórz nowy moduł w katalogu `modules/services/k8s-app`. Następnie w tym katalogu utwórz plik o nazwie `variables.tf` definiujący API modułu za pomocą takich zmiennych danych wejściowych:

```
variable "name" {
  description = "Nazwa używana przez wszystkie zasoby tworzone przez ten moduł"
  type        = string
}

variable "image" {
  description = "Obraz Dockera przeznaczony do uruchomienia"
  type        = string
}

variable "container_port" {
  description = "Port, na którym nasłuchuje obraz Dockera"
  type        = number
}

variable "replicas" {
  description = "Liczba replik do uruchomienia"
```

```

    type      = number
  }

  variable "environment_variables" {
    description = "Zmienne środowiskowe do zdefiniowania dla aplikacji"
    type        = map(string)
    default     = {}
  }

```

W ten sposób otrzymujesz wszystkie zmienne danych wejściowych niezbędnych do utworzenia obiektów typu Kubernetes Deployment i Service. Następnie utwórz plik *main.tf* i umieść w nim blok `required_providers` definiujący dostawcę Kubernetes.

```

terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    kubernetes = {
      source = "hashicorp/kubernetes"
      version = "~> 2.0"
    }
  }
}

```

Nowy dostawca, świetnie! Wykorzystamy go teraz do utworzenia obiektu typu Kubernetes Deployment za pomocą zasobu `kubernetes_deployment`:

```

resource "kubernetes_deployment" "app" {
}

```

Istnieje jeszcze kilka ustawień do skonfigurowania za pomocą zasobu `kubernetes_deployment`, przeanalizujemy je więc pojedynczo. Przede wszystkim trzeba skonfigurować blok `metadata`.

```

resource "kubernetes_deployment" "app" {
  metadata {
    name = var.name
  }
}

```

Każdy obiekt Kubernetes zawiera metadane, z których można skorzystać w ramach identyfikacji i używania go w wywołaniach API. Wcześniej w kodzie została zdefiniowana nazwa zmiennej danych wejściowych (`name`).

Pozostała część konfiguracji zasobu `kubernetes_deployment` znajduje się w bloku `spec`.

```

resource "kubernetes_deployment" "app" {
  metadata {
    name = var.name
  }

  spec {
  }
}

```

Zaczynamy od umieszczenia w bloku `spec` elementu określającego liczbę replik, które mają być utworzone.

```

spec {
  replicas = var.replicas
}

```

Następnie przechodzimy do zdefiniowania bloku `template`.

```
spec {
  replicas = var.replicas
  template {
  }
}
```

W Kubernetes, zamiast wdrażać kontenery pojedynczo, wdraża się tzw. **pody**, czyli grupy kontenerów przeznaczone do wspólnego wdrożenia. Przykładowo można mieć poda z kontenerem zawierającym aplikację internetową (np. pokazaną wcześniej aplikację Pythona) i kontener zbierający wskaźniki dotyczące aplikacji internetowej, które następnie trafiają do pewnej usługi centralnej (np. Datadog). Blok `template` to miejsce zdefiniowania tzw. **szablonu poda**, który określa kontenery przeznaczone do uruchomienia, używane porty, ustawiane zmienne środowiskowe itd.

Ważnym składnikiem w szablonie poda są etykiety stosowane dla poda. Będziesz ich ponownie używać w wielu miejscach — np. Kubernetes Service używa etykiet do identyfikacji podów wymagających mechanizmu równoważenia obciążenia — więc zdefiniujemy je w zmiennej lokalnej o nazwie `pod_labels`.

```
locals {
  pod_labels = {
    app = var.name
  }
}
```

Teraz `pod_labels` można użyć w bloku `metadata` szablonu poda.

```
spec {
  replicas = var.replicas

  template {
    metadata {
      labels = local.pod_labels
    }
  }
}
```

Kolejnym krokiem jest dodanie bloku `spec` w `template`.

```
spec {
  replicas = var.replicas

  template {
    metadata {
      labels = local.pod_labels
    }

    spec {
      container {
        name = var.name
        image = var.image

        port {
          container_port = var.container_port
        }
      }
    }
  }
}
```



```

dynamic "env" {
  for_each = var.environment_variables
  content {
    name = env.key
    value = env.value
  }
}
}
}
}
}

```

Mamy tutaj całkiem sporo elementów, zostaną więc omówione pojedynczo.

container

W bloku spec można zdefiniować jeden lub więcej bloków container określających kontenery Dockera przeznaczone do uruchomienia w podzie. W celu zachowania prostoty przykładu w omawianej sytuacji mamy tylko jeden blok container. Pozostałe elementy będą się znajdowały w tym bloku container.

name

Nazwa używana przez kontener. W omawianym przykładzie wartością jest zmienna danych wejściowych name.

image

Obraz Dockera przeznaczony do uruchomienia w komputerze. W omawianym przykładzie wartością jest zmienna danych wejściowych image.

port

Port udostępniany przez kontener. W celu zachowania prostoty przykładu założyłem, że kontener musi nasłuchiwać tylko na jednym porcie. W omawianym przykładzie wartością jest zmienna danych wejściowych container_port.

env

Zmienne środowiskowe udostępniane przez kontener. W omawianym przykładzie użyłem bloku dynamic razem z dwiema konstrukcjami for_each (te dwie koncepcje prawdopodobnie pamiętasz z rozdziału 5.) w celu przypisania wartości w zmiennej danych wejściowych environment_variables.

W ten sposób omówiłem szablon poda. Pozostał już tylko jeden element do dodania w zasobie kubernetes_deployment — blok selector.

```

spec {
  replicas = var.replicas

  template {
    metadata {
      labels = local.pod_labels
    }

    spec {
      container {
        name = var.name
        image = var.image

```

```

    port {
        container_port = var.container_port
    }

    dynamic "env" {
        for_each = var.environment_variables
        content {
            name = env.key
            value = env.value
        }
    }
}

selector {
    match_labels = local.pod_labels
}
}

```

Blok selector wskazuje cel dla Kubernetes Deployment. Dzięki użyciu `pod_labels` nakazujemy zarządzanie zdefiniowanym przed chwilą szablonem poda. Dlaczego obiekt Kubernetes Deployment po prostu nie przyjmuje założenia, że zdefiniowany w nim szablon poda jest celem? Cóż, Kubernetes stara się zapewnić maksymalną elastyczność i system, którego komponenty nie są ściśle ze sobą powiązane. Istnieje np. możliwość zdefiniowania obiektu Kubernetes Deployment dla podów określonych oddzielnie. Dlatego też zawsze trzeba używać bloku selector do wskazania celu dla Kubernetes Deployment.

W ten sposób zakończyłem omawianie zasobu `kubernetes_deployment`. Następnym krokiem jest użycie zasobu `kubernetes_service` w celu utworzenia Kubernetes Service.

```

resource "kubernetes_service" "app" {
    metadata {
        name = var.name
    }

    spec {
        type = "LoadBalancer"
        port {
            port          = 80
            target_port = var.container_port
            protocol      = "TCP"
        }
        selector = local.pod_labels
    }
}

```

Przeanalizujmy te parametry.

`metadata`

Podobnie jak w wypadku obiektu Deployment, także obiekt Service używa metadanych do identyfikacji i wskazania obiektu w wywołaniach API. W omawianym przykładzie nazwą obiektu Service jest wartość zmiennej danych wejściowych `name`.

type

Ten obiekt Service został skonfigurowany jako typ LoadBalancer, który w zależności od sposobu skonfigurowania klastra Kubernetes będzie wdrażał różnego rodzaju mechanizmy równoważenia obciążenia. Przykładowo w AWS z EKS można otrzymać Elastic Load Balancer, podczas gdy w Google Cloud z GKE można otrzymać Cloud Load Balancer.

port

Mechanizm równoważenia obciążenia został skonfigurowany w taki sposób, aby ruch sieciowy kierowany do portu 80 (port domyślny dla HTTP) został przekazany do portu, na którym nasłuchuje kontener.

selector

Podobnie jak w wypadku obiektu Deployment, także obiekt Service używa selektora do wskazania, który obiekt Service jest celem. Przypisanie selektorowi wartości pod_labels oznacza, że obiekty Deployment i Service będą działały z tymi samymi podami.

Ostatnim krokiem jest udostępnienie punktu końcowego Service (nazwa hosta mechanizmu równoważenia obciążenia) jako zmiennej danych wyjściowych w *outputs.tf*.

```
locals {
  status = kubernetes_service.app.status
}

output "service_endpoint" {
  value = try(
    "http://${local.status[0]["load_balancer"][0]["ingress"][0]["hostname"]}",
    "(error parsing hostname from status)"
  )
  description = "Punkt końcowy K8S Service"
}
```

Ten nieco zawiliły fragment kodu wymaga pewnego wyjaśnienia. Zasób kubernetes_service ma atrybut danych wyjściowych o nazwie status, który zwraca najnowsze informacje o stanie obiektu Service. W omawianym przykładzie ten atrybut jest przechowywany w zmiennej lokalnej o nazwie status. W wypadku obiektu Service typu LoadBalancer zmienna status będzie zawierała skomplikowany obiekt, który będzie podobny do przedstawionego tutaj.

```
[
  {
    load_balancer = [
      {
        ingress = [
          {
            hostname = "<NAZWA_HOSTA>"
          }
        ]
      }
    ]
  }
]
```

W tym bardzo zagnieżdżonym obiekcie gdzieś głęboko znajduje się nazwa hosta mechanizmu równoważenia obciążenia, który nas interesuje. Dlatego też zmienna danych wyjściowych service_endpoint

musi korzystać ze skomplikowanej sekwencji operacji wyszukiwania tablicy (np. [0]) i wyszukiwania mapowania (np. ["load_balancer"]) w celu wyodrębnienia nazwy hosta. Co się stanie, jeśli atrybut status zwrócony przez zasób `kubernetes_service` będzie wyglądał nieco inaczej? W takim wypadku wszelkie tablice i operacje wyszukiwania mapowania mogą się zakończyć niepowodzeniem i doprowadzić do wygenerowania dezorientującego błędu.

Aby w elegancki sposób obsłużyć ten błąd, całe wyrażenie zostało opakowane wywołaniem funkcji `try()`, której składnia przedstawia się następująco:

```
try(ARG1, ARG2, ..., ARGN)
```

Ta funkcja sprawdza wszystkie przekazane jej argumenty i zwraca pierwszy, który nie powoduje wygenerowania żadnych błędów. Dlatego też zmienna danych wyjściowych `service_endpoint` będzie zawierała nazwę hosta (pierwszy argument) lub, jeśli odczyt nazwy hosta zakończy się błędem, komunikat błędu `error parsing hostname from status` (drugi argument).

W porządku, w ten sposób powstał moduł `k8s-app`. Aby go użyć, należy dodać nowy przykład w *examples/kubernetes-local* i utworzyć plik *main.tf* o takiej zawartości:

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name       = "simple-webapp"
  image      = "training/webapp"
  replicas   = 2
  container_port = 5000
}
```

To konfiguruje moduł do wdrożenia obrazu Dockera `training/webapp` (uruchomionego wcześniej), z dwiema replikami, nasłuchującego na porcie 5000. Wszystkie obiekty Kubernetes będą miały nazwę `simple-webapp` (na podstawie wartości metadanych). Aby ten moduł wdrożyć w lokalnym klastrze Kubernetes, trzeba dodać ten blok `provider`:

```
provider "kubernetes" {
  config_path = "~/.kube/config"
  config_context = "docker-desktop"
}
```

Ten kod nakazuje dostawcy Kubernetes uwierzytliwienie lokalnego klastra Kubernetes przez użycie kontekstu `docker-desktop` z pliku konfiguracyjnego `kubect1`. Użyj polecenia `terraform apply` i zobacz, jak to działa.

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
service_endpoint = "http://localhost"
```

Odczekaj kilka sekund niezbędnych do uruchomienia egzemplarzy, a następnie wypróbuj `service_endpoint`.

```
$ curl http://localhost
Hello world!
```

Sukces!

Dane wyjściowe są niemalże identyczne z wygenerowanymi przez polecenie `docker run`, zatem możesz się zastanawiać, czy warto podejmować dodatkowy wysiłek. Zobacz więc, co się dzieje w tle. Polecenie `kubectl` możesz wykorzystać do poruszania się po klastrze. Zacznij od użycia polecenia `kubectl get deployments`.

```
$ kubectl get deployments
NAME          READY  UP-TO-DATE  AVAILABLE  AGE
simple-webapp  2/2    2            2           3m21s
```

Możesz zobaczyć obiekt Kubernetes Deployment o nazwie `simple-webapp`, czyli o nazwie podanej w bloku metadata. Te dane Deployment wskazują na gotowość dwóch podów z dwóch (dwie repliki). Aby zobaczyć te pody, użyj polecenia `kubectl get pods`.

```
$ kubectl get pods
NAME                                READY  STATUS   RESTARTS  AGE
simple-webapp-d45b496fd-7d447       1/1    Running   0          2m36s
simple-webapp-d45b496fd-v16j7       1/1    Running   0          2m36s
```

Mamy tutaj jedną różnicę względem `docker run`: uruchomionych jest kilka kontenerów, a nie tylko jeden. Co więcej, te kontenery są aktywnie monitorowane i zarządzane. Jeśli np. jeden ulegnie awarii, jego zamiennik zostanie wdrożony automatycznie. Możesz to zobaczyć w akcji dzięki poleceniu `docker ps`.

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS
b60f5147954a   training/webapp "python app.py"         3 seconds ago   Up 2 seconds
c350ec648185   training/webapp "python app.py"         12 minutes ago   Up 12 minutes
```

Pobierz wartość `CONTAINER ID` jednego z tych kontenerów i użyj polecenia `docker kill`, aby zakończyć jego działanie.

```
$ docker kill b60f5147954a
```

Jeżeli szybko ponownie wydasz polecenie `docker ps`, zobaczysz, że pozostał uruchomiony tylko jeden kontener.

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS
c350ec648185   training/webapp "python app.py"         12 minutes ago   Up 12 minutes
```

Jednak już po kilku sekundach Kubernetes Deployment wykryje istnienie tylko jednej repliki zamiast dwóch i automatycznie zostanie uruchomiony nowy kontener.

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS
56a216b8a829   training/webapp "python app.py"         1 second ago     Up 5 seconds
c350ec648185   training/webapp "python app.py"         12 minutes ago   Up 12 minutes
```

Kubernetes gwarantuje, że zawsze będzie uruchomiona żądana liczba replik. Co więcej, działa też mechanizm równoważenia obciążenia, aby ruch sieciowy był rozkładany między tymi replikami. Możesz się o tym przekonać za pomocą polecenia `kubectl get services`.

```
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	4h26m
simple-webapp	LoadBalancer	10.110.25.79	localhost	80:30234/TCP	4m58s

Pierwsza usługa na liście to sam Kubernetes, możesz ją zignorować. Druga to utworzony obiekt Service, o nazwie `simple-webapp` (na podstawie bloku metadata). Ta usługa obsługuje mechanizm równoważenia obciążenia dla aplikacji: możesz poznać dostępny adres IP (`localhost`) i nasłuchiwany port (80).

Kubernetes Deployment zapewnia również automatyczne aktualizacje. W wypadku obrazu `training/webapp` zabawna sztuczka polega na tym, że jeśli zmiennej środowiskowej `PROVIDER` zostanie przypisana pewna wartość, zostanie ona użyta zamiast słowa *world* w komunikacie „Hello, world!” (Witaj, świecie!). Uaktualnij kod w pliku `examples/kubernetes-local/main.tf`, aby zmienić wartość tej zmiennej środowiskowej.

```
module "simple_webapp" {
  source = "../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Terraform"
  }
}
```

Raz jeszcze użyj polecenia `terraform apply`.

```
$ terraform apply
```

```
(...)
```

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

```
Outputs:
```

```
service_endpoint = "http://localhost"
```

Po kilku sekundach ponownie wypróbuj punkt końcowy.

```
$ curl http://localhost
Hello Terraform!
```

Jak widzisz, obiekt Deployment automatycznie wprowadził zmianę. W tle obiekt Deployment domyślnie wprowadza zmiany, podobnie jak w przypadku automatycznie skalowanej grupy. (Warto w tym miejscu wspomnieć o możliwości zmiany ustawień wdrożenia przez dodanie bloku `strategy` do zasobu `kubernetes_deployment`).

Wdrażanie kontenerów Dockera w AWS za pomocą Elastic Kubernetes Service

Kubernetes skrywa jeszcze jednego asa w rękawie: to rozwiązanie jest przenośne. Istnieje możliwość wielokrotnego użycia zarówno obrazów Dockera, jak i konfiguracji Kubernetes w zupełnie innym klastrze, a otrzymane wyniki będą podobne. Aby zobaczyć to w akcji, zajmiemy się teraz wdrożeniem klastra Kubernetes w AWS.

Konfigurowanie i zarządzanie bezpiecznym, charakteryzującym się wysoką dostępnością i skalowalnym klastrem Kubernetes w chmurze, utworzonym zupełnie od początku, jest dość skomplikowane. Na szczęście wielu dostawców chmury oferuje zarządzane usługi Kubernetes, w których za programistę są uruchamiane płaszczyzny kontrolne i węzły robocze, np. Elastic Kubernetes Service (EKS) w AWS, Azure Kubernetes Service (AKS) w Azure i Google Kubernetes Engine (GKE) w Google Cloud. Zamierzam pokazać, jak można wdrożyć bardzo prosty klastrek EKS w AWS.

Utwórz nowy moduł *modules/services/eks-cluster* i zdefiniuj dla niego API w pliku *variables.tf* o takiej zawartości:

```
variable "name" {
  description = "Używana nazwa klastra EKS"
  type        = string
}

variable "min_size" {
  description = "Minimalna liczba węzłów w klastrze EKS"
  type        = number
}

variable "max_size" {
  description = "Maksymalna liczba węzłów w klastrze EKS"
  type        = number
}

variable "desired_size" {
  description = "Oczekiwana liczba węzłów w klastrze EKS"
  type        = number
}

variable "instance_types" {
  description = "Typy egzemplarzy EC2 uruchomione w grupie węzła"
  type        = list(string)
}
```

Ten kod udostępnia zmienne danych wejściowych definiujące nazwę klastra EKS, liczbę i typy egzemplarzy przeznaczonych do uruchamiania w węzłach roboczych. Następnie w pliku *main.tf* utwórz rolę IAM dla płaszczyzny kontrolnej.

```
# Utworzenie roli IAM dla płaszczyzny kontrolnej.
resource "aws_iam_role" "cluster" {
  name               = "${var.name}-cluster-role"
  assume_role_policy = data.aws_iam_policy_document.cluster_assume_role.json
}

# Umożliwienie EKS przyjęcia roli IAM.
data "aws_iam_policy_document" "cluster_assume_role" {
  statement {
    effect = "Allow"
    actions = ["sts:AssumeRole"]
    principals {
      type        = "Service"
      identifiers = ["eks.amazonaws.com"]
    }
  }
}
```

```
# Dodanie uprawnień wymaganych przez rolę IAM.
resource "aws_iam_role_policy_attachment" "AmazonEKSClusterPolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"
  role       = aws_iam_role.cluster.name
}
```

Ta rola IAM może być przyjęta przez usługę EKS i ma dodane uprawnienie polityki zarządzanej roli IAM, które nadaje płaszczyźnie kontrolnej niezbędne uprawnienia. Teraz dodaj źródła danych `aws_vpc` i `aws_subnets` w celu pobrania informacji o domyślnej wirtualnej chmurze prywatnej i jej sieciach.

*# Skoro kod służy jedynie w celach dydaktycznych, użyjemy Default VPC i podsieci.
W przypadku rzeczywistych rozwiązań należy utworzyć własny VPC i prywatne podsieci.*

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

data "aws_subnets" "default" {
  filter {
    name     = "vpc-id"
    values   = [data.aws_vpc.default.id]
  }
}
```

Kolejny krok to utworzenie płaszczyzny kontrolnej dla klastra EKS za pomocą zasobu `aws_eks_cluster`.

```
resource "aws_eks_cluster" "cluster" {
  name     = var.name
  role_arn = aws_iam_role.cluster.arn
  version  = "1.21"

  vpc_config {
    subnet_ids = data.aws_subnets.default.ids
  }

  # Trzeba się upewnić, że uprawnienia roli IAM są tworzone przed i usuwane po
  # klastrze EKS. W przeciwnym razie EKS nie będzie w stanie poprawnie usunąć
  # zarządzanej przez EKS infrastruktury EC2, np. grup bezpieczeństwa.
  depends_on = [
    aws_iam_role_policy_attachment.AmazonEKSClusterPolicy
  ]
}
```

Przedstawiony fragment kodu konfiguruje płaszczyznę kontrolną w celu użycia utworzonej roli IAM i wdrożenia w domyślnej wirtualnej chmurze prywatnej (Default VPC) i podsieciach.

Kolejne zadanie wiąże się z węzłami roboczymi. EKS obsługuje wiele różnych typów węzłów roboczych: samodzielnie zarządzane egzemplarze EC2 (np. utworzoną automatycznie skalowaną grupę), zarządzane przez AWS egzemplarze EC2 (znane pod nazwą *zarządzanej grupy węzła*) i Fargate (bez użycia serwera)³. Najprostsze rozwiązanie w przypadku przykładów zamieszczonych w rozdziale polega na użyciu zarządzanej grupy węzła.

³ Porównanie różnych typów węzłów roboczych EKS znajdziesz w poście na blogu Gruntwork, opublikowanym na stronie <https://blog.gruntwork.io/comprehensive-guide-to-eks-worker-nodes-94e241092cbe>.

W celu wdrożenia zarządzanej grupy węzła trzeba rozpocząć od utworzenia nowej roli IAM.

```
# Utworzenie roli IAM dla grupy węzła.
resource "aws_iam_role" "node_group" {
  name           = "${var.name}-node-group"
  assume_role_policy = data.aws_iam_policy_document.node_assume_role.json
}

# Umożliwienie egzemplarzom EC2 przyjęcia roli IAM.
data "aws_iam_policy_document" "node_assume_role" {
  statement {
    effect = "Allow"
    actions = ["sts:AssumeRole"]
    principals {
      type       = "Service"
      identifiers = ["ec2.amazonaws.com"]
    }
  }
}

# Dodanie uprawnień wymaganych przez grupę węzła.
resource "aws_iam_role_policy_attachment" "AmazonEKSWorkerNodePolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy"
  role       = aws_iam_role.node_group.name
}

resource "aws_iam_role_policy_attachment" "AmazonEC2ContainerRegistryReadOnly" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly"
  role       = aws_iam_role.node_group.name
}

resource "aws_iam_role_policy_attachment" "AmazonEKS_CNI_Policy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy"
  role       = aws_iam_role.node_group.name
}
```

Ta rola IAM może być przyjęta przez usługę EC2 (to ma sens, ponieważ w tle egzemplarze EC2 używają grupy węzła) i ma wiele dołączonych polityk zarządzania rolą IAM, które nadają zarządzanej grupie węzła niezbędne jej uprawnienia. Teraz można użyć zasobu `aws_eks_node_group` do utworzenia samej zarządzanej grupy węzła.

```
resource "aws_eks_node_group" "nodes" {
  cluster_name   = aws_eks_cluster.cluster.name
  node_group_name = var.name
  node_role_arn  = aws_iam_role.node_group.arn
  subnet_ids     = data.aws_subnets.default.ids
  instance_types = var.instance_types

  scaling_config {
    min_size     = var.min_size
    max_size     = var.max_size
    desired_size = var.desired_size
  }
}

# Trzeba się upewnić, że uprawnienia roli IAM są tworzone przed i usuwane po
# grupie węzła EKS. W przeciwnym razie EKS nie będzie w stanie poprawnie usunąć
# egzemplarzy EC2 i Elastic Network Interfaces.
depends_on = [
```

```

    aws_iam_role_policy_attachment.AmazonEKSWorkerNodePolicy,
    aws_iam_role_policy_attachment.AmazonEC2ContainerRegistryReadOnly,
    aws_iam_role_policy_attachment.AmazonEKS_CNI_Policy,
  ]
}

```

Ten kod konfiguruje zarządzaną grupę węzła, a także używa płaszczyzny kontrolnej i utworzonej roli IAM do wdrożenia w domyślnej wirtualnej chmurze prywatnej. Używane są parametry nazwy, liczby i typu egzemplarza, przekazane w zmiennych danych wejściowych.

W pliku *outputs.tf* należy dodać takie zmienne danych wyjściowych:

```

output "cluster_name" {
  value      = aws_eks_cluster.cluster.name
  description = "Nazwa klastra EKS"
}

output "cluster_arn" {
  value      = aws_eks_cluster.cluster.arn
  description = "Wartość ARN klastra EKS"
}

output "cluster_endpoint" {
  value      = aws_eks_cluster.cluster.endpoint
  description = "Punkt końcowy klastra EKS"
}

output "cluster_certificate_authority" {
  value      = aws_eks_cluster.cluster.certificate_authority
  description = "Urząd certyfikacji klastra EKS"
}

```

W tym momencie moduł *eks-cluster* jest gotowy do wdrożenia. Użyjemy go i utworzonego wcześniej modułu *k8s-app* do wdrożenia klastra EKS, a następnie do wdrożenia w nim obrazu Dockera *training/webapp*. Utwórz plik *examples/kubernetes-eks/main.tf* i skonfiguruj moduł *eks-cluster* w przedstawiony tutaj sposób.

```

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

module "eks_cluster" {
  source = "../../modules/services/eks-cluster"

  name      = "example-eks-cluster"
  min_size  = 1
  max_size  = 2
  desired_size = 1

  # Ze względu na sposób działania EKS z ENI, t3.small to najmniejszy
  # typ egzemplarza, który może być użyty dla węzła roboczego. Jeżeli
  # spróbujesz użyć mniejszego, np. t2.micro, który ma tylko 4 ENI, to
  # wszystkie zostaną użyte przez usługi systemowe (np. kube-proxy)
  # i nie będzie możliwości wdrożenia własnych podów.
  instance_types = ["t3.small"]
}

```

Kolejnym krokiem jest konfiguracja k8s-app w przedstawiony tutaj sposób.

```
provider "kubernetes" {
  host = module.eks_cluster.cluster_endpoint
  cluster_ca_certificate = base64decode(
    module.eks_cluster.cluster_certificate_authority[0].data
  )
  token = data.aws_eks_cluster_auth.cluster.token
}

data "aws_eks_cluster_auth" "cluster" {
  name = module.eks_cluster.cluster_name
}

module "simple_webapp" {
  source = "../modules/services/k8s-app"

  name          = "simple-webapp"
  image         = "training/webapp"
  replicas      = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Terraform"
  }

  # Wdrożenie może się odbywać tylko po wdrożeniu klastra.
  depends_on = [module.eks_cluster]
}
```

Ten fragment kodu konfiguruje dostawcę Kubernetes do uwierzytelnienia w klastrze EKS, a nie w lokalnym klastrze Kubernetes (utworzonym przez oprogramowanie Docker Desktop). Następnie moduł k8s-app jest używany do wdrożenia obrazu Dockera training/webapp dokładnie w taki sam sposób, jak w przypadku wdrożenia za pomocą Docker Desktop. Jedyna różnica polega na dodaniu parametru depends_on w celu zagwarantowania, że Terraform będzie wdrażać obraz Dockera tylko po wdrożeniu klastra EKS.

Punkt końcowy usługi trzeba przekazać jako zmienną danych wyjściowych.

```
output "service_endpoint" {
  value          = module.simple_webapp.service_endpoint
  description    = "Punkt końcowy K8S Service"
}
```

W tym momencie możesz wdrażać! Jak zwykle użyj polecenia terraform apply. Wdrożenie klastra EKS może zająć 10 – 20 minut, więc cierpliwości!

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 10 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
service_endpoint = "http://774696355.us-east-2.elb.amazonaws.com"
```

Poczekaj, aż aplikacja internetowa zostanie uruchomiona i przejdzie procedurę sprawdzenia, a następnie przetestuj `service_endpoint`.

```
$ curl http://774696355.us-east-2.elb.amazonaws.com
Hello Terraform!
```

Gotowe! Ten sam obraz Dockera i kod Kubernetes, które wcześniej zostały uruchomione w komputerze lokalnym, teraz działają w klastrze EKS, wdrożonym w chmurze AWS. W obu przypadkach masz dostęp do tej samej funkcjonalności. Dla przykładu spróbuj uaktualnić zmienną `environment_variables`, aby miała inną wartość `PROVIDER`, np. `Readers`.

```
module "simple_webapp" {
  source = "../modules/services/k8s-app"

  name       = "simple-webapp"
  image      = "training/webapp"
  replicas   = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Readers"
  }

  # Wdrożenie może się odbywać tylko po wdrożeniu klastra.
  depends_on = [module.eks_cluster]
}
```

Ponownie użyj polecenia `terraform apply`, a po kilku sekundach obiekt Kubernetes Deployment wprowadzi zmiany.

```
$ curl http://774696355.us-east-2.elb.amazonaws.com
Hello Readers!
```

To jest jedna z zalet używania Dockera: zmiany mogą być wprowadzane naprawdę bardzo szybko.

Możesz ponownie użyć `kubectl` do sprawdzenia, co się dzieje w klastrze. Aby uwierzytelnić `kubectl` w klastrze EKS, musisz skorzystać z polecenia `aws eks update-kubeconfig` w celu automatycznego uaktualnienia pliku `$HOME/.kube/config`.

```
$ aws eks update-kubeconfig --region <REGION> --name <NAZWA_KLASTRA_EKS>
```

W tym poleceniu `REGION` to nazwa regionu AWS, a `NAZWA_KLASTRA_EKS` to nazwa Twojego klastra EKS. W omawianym przykładzie wdrożenie w module Terraform odbywa się w regionie `us-east-2`, a nazwą klastra jest `kubernetes-example`, polecenie ma zatem taką postać:

```
$ aws eks update-kubeconfig --region us-east-2 --name kubernetes-example
```

Podobnie jak wcześniej, można skorzystać z polecenia `kubectl get nodes` w celu przeanalizowania węzłów roboczych klastra. Jednak tym razem dodajemy opcję `-o wide`, aby otrzymać nieco więcej informacji.

```
$ kubectl get nodes -o wide
```

NAME	STATUS	AGE	EXTERNAL-IP	OS-IMAGE
xxx.us-east-2.compute.internal	Ready	22m	3.134.78.187	Amazon Linux 2

Wygenerowane dane wyjściowe zostały bardzo ograniczone, aby zmieściły się na stronie książki. W rzeczywistości otrzymasz ich znacznie więcej — m.in. węzeł roboczy, wewnętrzny i zewnętrzny adres IP, informacje o wersji, informacje dotyczące systemu operacyjnego itd.

Polecenie `kubectl get deployments` pozwala na przeanalizowanie obiektów `Deployment`.

```
$ kubectl get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
simple-webapp  2/2     2             2           19m
```

Jeżeli chcesz wyświetlić pody, użyj polecenia `kubectl get pods`.

```
$ kubectl get pods
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
simple-webapp  2/2     2             2           19m
```

Na koniec możesz zastosować polecenie `kubectl get services` w celu wyświetlenia obiektów `Service`.

```
$ kubectl get services
NAME          TYPE          EXTERNAL-IP          PORT(S)
kubernetes    ClusterIP     <none>                443/TCP
simple-webapp  LoadBalancer 774696355.us-east-2.elb.amazonaws.com 80/TCP
```

Wygenerowane dane wyjściowe powinny zawierać informacje o mechanizmie równoważenia obciążenia i adresie URL użytym do jego przetestowania.

W ten sposób masz dwóch różnych dostawców, obu działających w tej samej chmurze i pomagających we wdrażaniu skonteneryzowanych obciążeń.

Podobnie jak we wcześniejszej części rozdziału, także tutaj chciałbym zamieścić dwa ostrzeżenia.

Ostrzeżenie 1.: Te przykłady Kubernetes są bardzo uproszczone

Kubernetes to skomplikowana technologia, intensywnie rozwijana i stale się zmieniająca. Próba jej dokładnego wyjaśnienia z pewnością wymagałaby oddzielnej książki. Ponieważ ta pozycja jest poświęcona Terraform, a nie Kubernetes, moim celem w przykładach dotyczących Kubernetes było zachowanie maksymalnej prostoty i minimalizmu. Dlatego też, choć mam nadzieję, że zaprezentowane przykłady są użyteczne w celach dydaktycznych i podczas eksperymentów, to jeśli zamierzasz korzystać z Kubernetes w rzeczywistych projektach, musisz zmienić wiele aspektów tego kodu, np. skonfigurować pewną liczbę dodatkowych usług i ustawień w module `eks-cluster` (takich jak kontrolery `ingress`, szyfrowanie, grupy bezpieczeństwa, uwierzytelnianie OIDC, mapowanie RBAC (ang. *role-based access control*), VPC-CNI, kube-proxy, CoreDNS), udostępnić wiele innych ustawień w module `k8s-app` (np. zarządzanie danymi poufnymi, woluminami, sprawdzaniem dostępności, sprawdzaniem możliwości odczytu, etykietami, adnotacjami, wieloma portami, wieloma kontenerami), a także użyć niestandardowej wirtualnej chmury prywatnej z prywatnymi podsieciami dla klastra EKS, zamiast używać domyślnej wirtualnej chmury prywatnej i publicznych podsięci⁴.

Ostrzeżenie 2.: Oszczędnie korzystaj z wielu dostawców

Wprawdzie bez wątpienia można korzystać z wielu dostawców w pojedynczym module, ale nie zachęcam do nadużywania tej możliwości, z podobnych powodów, dla których nie zalecam zbyt częstego używania aliasów dostawców — w większości przypadków każdy dostawca powinien być odizolowany we własnym module. To pozwala zarządzać nim oddzielnie i ograniczyć pole rażenia w razie błędów lub ataków.

⁴ Ewentualnie możesz skorzystać z gotowych modułów Kubernetes o jakości produkcyjnej, takich jak AWS Infrastructure as Code Library (<https://gruntwork.io/infrastructure-as-code-library/>).

Co więcej, Terraform nie oferuje zbyt dobrej obsługi kolejności zależności między dostawcami. Przykładowo w rozwiązaniu bazującym na Kubernetes mieliśmy pojedynczy moduł wdrożony w klastrze EKS, z użyciem dostawcy AWS, a aplikacja Kubernetes w tym klastrze używała dostawcy Kubernetes. Okazuje się, że w dokumentacji dostawcy Kubernetes (<https://registry.terraform.io/providers/hashicorp/kubernetes/latest/docs>) zdecydowanie odradza się stosowanie takiego podejścia.

Podczas używania interpolacji w celu przekazania do dostawcy Kubernetes danych uwierzytelniających pochodzących z innych zasobów NIE POWINNY być one tworzone w tym samym module, w którym jest używany zasób dostawcy Kubernetes. To może czasami prowadzić do nieprzewidywalnych błędów, trudnych do debugowania i diagnozowania. Podstawowa przyczyna błędów wiąże się z kolejnością, w jakiej Terraform przetwarza bloki dostawców, i rzeczywistą kolejnością zasobów.

Przykład zamieszczony w książce radzi sobie z tymi problemami dzięki zależności od źródła danych `aws_eks_cluster_auth`, przy czym to jest rodzaj sztuczki. Dlatego też w kodzie produkcyjnym zawsze zalecam wdrażanie klastra EKS w jednym module, a aplikacji Kubernetes w innym, już po wdrożeniu klastra.

Podsumowanie

Mam nadzieję, że dokładnie wiesz, jak pracować z wieloma dostawcami w kodzie Terraform, i potrafisz odpowiedzieć na trzy pytania, które pojawiły się na początku rozdziału.

Co można zrobić w sytuacji, gdy zachodzi potrzeba wdrożenia w wielu regionach AWS?

Skorzystaj z wielu bloków `provider`, każdy skonfigurowany z innymi parametrami `region` i `alias`.

Co można zrobić w sytuacji, gdy zachodzi potrzeba wdrożenia w wielu kontach AWS?

Skorzystaj z wielu bloków `provider`, każdy skonfigurowany z innym blokiem `assume_role` i parametrem `alias`.

Co można zrobić w sytuacji, gdy zachodzi potrzeba wdrożenia w innych chmurach, np. Azure lub GCP?

Skorzystaj z wielu bloków `provider`, każdy skonfigurowany dla odpowiedniej chmury.

Wyjaśniłem również, że używanie wielu dostawców w jednym module to najczęściej antywzorzec. Dlatego też rzeczywista odpowiedź na te pytania, zwłaszcza w projektach produkcyjnych, to używanie poszczególnych dostawców w oddzielnych modułach, aby zapewnić zastosowanie oddzielnych regionów i kont, a także odizolowanie chmur od siebie — to pozwala ograniczyć pole rażenia w razie problemów.

Przechodzimy teraz do rozdziału 8., w którym przedstawię kilka innych wzorców pokazujących tworzenie modułów Terraform dla rzeczywistych projektów używanych w produkcji — to są moduły, na których może się opierać działanie firmy.

Produkcyjny kod Terraform

Tworzenie infrastruktury o jakości produkcyjnej jest zadaniem trudnym, stresującym i czasochłonnym. Gdy używam określenia *infrastruktura o jakości produkcyjnej*, mam na myśli infrastrukturę niezbędną 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 8.1.

Tabela 8.1. 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 wersjonowane,
 - 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

Myślę, 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 kilkanaście lat 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 ziemie. 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! Mógłbym pożyczyć przepustkę od sąsiada...”

„Nie, on nie pożyczy mi przepustki, dopóki nie zwrócę poduszki Moshi Moshi, którą pożyczył od niego mój syn”.

„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

¹ Douglas R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*. 20th 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.

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 splecione 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 liczbie ważnych — i czasochłonnych — szczegółów. Wspomniana wcześniej lista jest tematem kolejnego podrozdziału.

³ Frederick P. Brooks jr., *The Mythical Man-Month: Essays on Software Engineering. Anniversary edition*, Addison-Wesley Professional, Reading 1995.

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 8.2). Ta lista zawiera większość elementów o znaczeniu kluczowym, które trzeba wziąć pod uwagę podczas przygotowywania wspomnianej infrastruktury.

Tabela 8.2. 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, Ansible, Docker, Packer
Konfigurowanie	Konfigurowanie oprogramowania w środowisku uruchomieniowym. Obejmuje m.in. ustawienia portu, certyfikaty TLS, wykrywanie usług, serwery główne i podrzędne, replikację itd.	Chef, Ansible, Kubernetes
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	ASG, Kubernetes, ECS
Wysoka dostępność	Odporność na awarie procesów, serwerów, usług, centrów danych i regionów	Wykorzystanie wielu centrów danych i wielu regionów
Skalowanie	Skalowanie w górę lub w dół w odpowiedzi na aktualne obciążenie. Skalowanie poziome (więcej serwerów) i (lub) pionowe (większe serwery)	Automatyczne skalowanie, replikacja
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, VisualVM
Obsługa sieci	Konfigurowanie statycznych i dynamicznych adresów IP, portów, wykrywania usług, zapór sieciowych, DNS, dostępu SSH i VPN	VPC, zapory sieciowe, Route 53

Tabela 8.2. Lista rzeczy do sprawdzenia podczas przygotowywania infrastruktury przeznaczonej do wdrożenia w środowisku produkcyjnym (ciąg dalszy)

Zadanie	Opis	Przykładowe narzędzia
Zapewnienie bezpieczeństwa	Szyfrowanie transmisji (TLS) i danych na dysku, uwierzytelnianie, autoryzacja, zarządzanie danymi wrażliwymi, zabezpieczanie serwera	ACM, Let's Encrypt, KMS, Vault
Obsługa wskaźników	Dostępność wskaźników m.in. biznesowych, aplikacji i serwera, obsługa zdarzeń, monitorowania, śledzenia i powiadamiania	CloudWatch, DataDog
Obsługa dzienników zdarzeń	Rotacja dzienników zdarzeń na dysku. Agregowanie w pewnym położeniu centralnym danych dzienników zdarzeń	Elastic Stack, Sumo Logic
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	Kopia zapasowa AWS, migawki RDS
Optimalizacja kosztów	Wybór odpowiednich egzemplarzy, używanie egzemplarzy typu spot zarezerwowanych, automatyczne skalowanie i pozbywanie się nieużywanych zasobów	Automatyczne skalowanie, Infracost
Obsługa dokumentacji	Dokumentowanie kodu, architektury i praktyk. Tworzenie tzw. scenariuszy w celu reagowania na incydenty	Pliki README, wiki, Slack, IaC
Przeprowadzanie testów	Tworzenie testów zautomatyzowanych dla infrastruktury jako kodu. Wykonywanie testów po wprowadzeniu każdej zmiany oraz każdej nocy	Terratest, tfint, OPA, InSpec

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 wersjonowane,
- 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 20 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 powróć w rozdziale 9.

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 — od razu wiesz, że ten kod śmierzdi i znacznie lepszym rozwiązaniem będzie przeprowadzenie jego refaktoryzacji na postać mniejszych, oddzielnych funkcji, z których każda wykonuje jedno zadanie. Tę samą strategię należy zastosować w Terraform. Wyobraź sobie, że masz do czynienia z architekturą pokazaną na rysunku 8.1.

Jeżeli ta architektura zostanie zdefiniowana w postaci pojedynczego, ogromnego modułu Terraform, który składa się z 20 000 wierszy, od razu powinieneś 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 8.2.

Zbudowany dotychczas moduł `webserver-cluster` (zobacz rozdział 5.) 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.

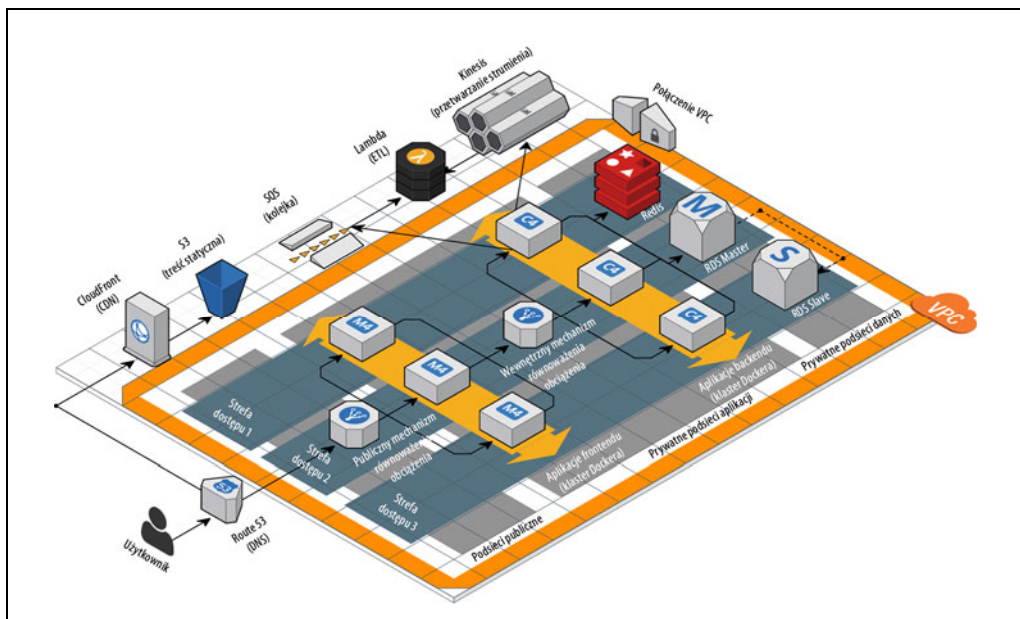
Mechanizm równoważenia obciążenia (ALB)

Moduł `webserver-cluster` wdraża mechanizm równoważenia obciążenia.

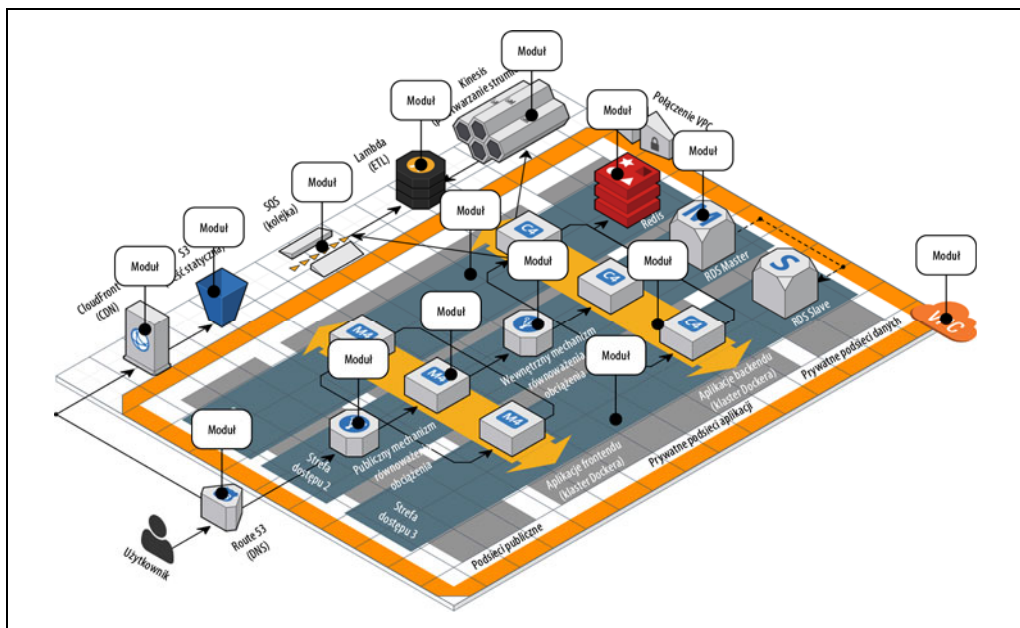
Aplikacja typu Witaj, świecie

Moduł `webserver-cluster` wdraża również prostą aplikację typu *Witaj, świecie*.

⁴ Robert C. Martin, *Czysty kod. Podręcznik dobrego programisty*, Helion, Gliwice 2010.



Rysunek 8.1. Przykład względnie skomplikowanej architektury AWS



Rysunek 8.2. Przykład względnie skomplikowanej architektury AWS podzielonej na wiele mniejszych modułów

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*, używający w tle modułów `asg-rolling-deploy` i `alb`.

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_subnets.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`:

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
```

⁵ Peter H. Salus, *A Quarter-Century of Unix*, Addison-Wesley Professional, New York 1994.

```
# 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/asg-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     = null
}

```

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 przekaz 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_subnets`), które nie zostały skopiowane do modułu `asg-rolling-deploy`.

```

resource "aws_autoscaling_group" "example" {
  name                = var.cluster_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

  # (...)
}

```

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. Przekaz zmienną `user_data` do zasobu `aws_launch_configuration`.

```

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.
  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:

- *aws_lb_target_group*,
- *aws_lb_listener_rule*,
- *terraform_remote_state* (dla baz danych),
- *aws_vpc*,
- *aws_subnets*.

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
  instance_type = var.instance_type

  user_data = templatefile("${path.module}/user-data.sh", {
    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
  })

  min_size          = var.min_size
  max_size          = var.max_size
  enable_autoscaling = var.enable_autoscaling

  subnet_ids      = data.aws_subnets.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_subnets.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 {
    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`).

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 *examples* (przykłady), 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          = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  min_size      = 1
  max_size      = 1
  enable_autoscaling = false

  subnet_ids = data.aws_subnets.default.ids
}

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

data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}

data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}
```

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 9., 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 jednoczesne 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 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 rozpoczniessz 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 rozpoczniessz 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 dokładniej przedstawię w rozdziale 9., całkowicie poświęconym tematowi testowania.

W tym punkcie skoncentruję się na tworzeniu *modułów samoweryfikujących się*, czyli modułów, które mogą sprawdzać swój sposób działania, aby uniknąć określonego typu błędów. W tym zakresie Terraform oferuje dwa rozwiązania:

- weryfikację,
- bloki precondition i postcondition.

Weryfikacja

Począwszy od wydania Terraform 0.13 można do dowolnej zmiennej danych wejściowych dodawać **bloki weryfikacji** w celu przeprowadzania operacji sprawdzenia wykraczających poza prostą kontrolę typu. Przykładowo do zmiennej `instance_type` można dodać blok `validation`, aby mieć pewność nie tylko co do przekazania przez użytkownika wartości w postaci ciągu tekstowego (wymuszenie za pomocą ograniczenia `type`), ale także tego, że ciąg tekstowy będzie miał jedną z dwóch postaci dozwolonych w ramach bezpłatnego konta AWS:

```
variable "instance_type" {
  description = "Egzemplarz EC2 przeznaczony do uruchomienia (np. t2.micro)"
  type        = string

  validation {
    condition     = contains(["t2.micro", "t3.micro"], var.instance_type)
    error_message = "Konto bezpłatne pozwala na użycie jedynie: t2.micro | t3.micro."
  }
}
```

Sposób działania bloku `validation` wygląda następująco: parametr `condition` powinien przyjąć wartość `true`, jeśli wartość jest poprawna, w przeciwnym razie wartością parametru powinna być `false`. Parametr `error_message` pozwala na zdefiniowanie komunikatu wyświetlanego użytkownikowi w przypadku przekazania nieprawidłowej wartości. Zobacz, co się stanie, gdy zmiennej `instance_type` spróbujesz przypisać wartość `m4.large`, która jest nieprawidłowa w wypadku bezpłatnego konta AWS:

```
$ terraform apply -var instance_type="m4.large"
Error: Invalid value for variable

on main.tf line 17:
1: variable "instance_type" {
  | var.instance_type is "m4.large"

Konto bezpłatne pozwala na użycie jedynie: t2.micro | t3.micro.

This was checked by the validation rule at main.tf:21,3-13.
```

W każdej zmiennej może istnieć wiele bloków validation przeznaczonych do sprawdzania różnych warunków.

```
variable "min_size" {
  description = "Minimalna liczba egzemplarzy EC2 w ASG"
  type        = number

  validation {
    condition      = var.min_size > 0
    error_message = "Grupa ASG nie może być pusta, ponieważ wtedy mamy awarię!"
  }

  validation {
    condition      = var.min_size <= 10
    error_message = "Grupa ASG powinna mieć co najwyżej 10 egzemplarzy, aby ograniczać
↳ koszty."
  }
}
```

Należy pamiętać o poważnym ograniczeniu bloków validation: condition w bloku validation może odwoływać się *jedynie* do obejmującej go zmiennej danych wejściowych. W razie próby odwołania się do jakiegokolwiek innej zmiennej danych wejściowych, zmiennych lokalnych, zasobów lub źródeł danych skutkiem będzie komunikat błędu. Wprawdzie blok validation jest użyteczny podczas prostego oczyszczania danych wejściowych, ale nie można go zastosować do bardziej złożonych zadań. Dlatego też nie można go użyć do sprawdzania między wieloma zmiennymi (np. „musi być zdefiniowana dokładnie jedna z tych dwóch zmiennych danych wejściowych”) albo podczas sprawdzania dynamicznego (np. by ustalić, czy użytkownik IAM korzysta z architektury x86_64). Do przeprowadzenia tego rodzaju dynamicznych operacji sprawdzenia trzeba wykorzystać bloki precondition i postcondition, omówione w następnym podpunkcie.

Bloki precondition i postcondition

W Terraform 1.2 można używać bloków precondition i postcondition podczas pracy z zasobami, ze źródłami danych i zmiennymi danych wyjściowych, aby przy użyciu tych bloków przeprowadzać bardziej dynamiczne operacje sprawdzania. Blok precondition służy do przechwytywania błędów jeszcze przed wykonaniem polecenia terraform apply. Przykładowo blok precondition można zastosować do przygotowania znacznie bardziej niezawodnej operacji sprawdzenia, czy użytkownik przekazał wartość instance_type dozwoloną w bezpłatnym koncie AWS. Wcześniej ta operacja odbywała się za pomocą bloku validation i miała na stałe zdefiniowaną listę typów egzemplarzy. Niestety, tego rodzaju listy szybko stają się nieaktualne. Zamiast tego można skorzystać ze źródła danych instance_type_data, które zawsze będzie miało aktualne informacje pochodzące z AWS:

```
data "aws_ec2_instance_type" "instance" {
  instance_type = var.instance_type
}
```

Następnie do zasobu aws_launch_configuration można dodać blok precondition, który sprawdzi, czy typ egzemplarza jest dozwolony w ramach bezpłatnego konta AWS:

```
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 z automatycznie skalowaną grupą.
lifecycle {
  create_before_destroy = true
  precondition {
    condition = data.aws_ec2_instance_type.instance.free_tier_eligible
    error_message = "${var.instance_type} jest niedostępny w bezpłatnym koncie AWS!"
  }
}
}
}

```

Podobnie jak `validation`, bloki `precondition` i — jak się wkrótce przekonasz — `postcondition` zawierają zmienną `condition`, której wartość musi wynosić `true` lub `false`. W tym drugim wypadku potrzeba jest zmienna `error_message` zawierająca komunikat błędu wyświetlany użytkownikowi. Jeżeli teraz użyjesz polecenia `terraform apply` razem z nazwą egzemplarza niedostępnego w ramach bezpłatnego konta AWS, otrzymasz komunikat błędu:

```

$ terraform apply -var instance_type="m4.large"
Error: Resource precondition failed

   on main.tf line 25, in resource "aws_launch_configuration" "example":
   18:   condition = data.aws_ec2_instance_type.instance.free_tier_eligible
      |_____
      | data.aws_ec2_instance_type.instance.free_tier_eligible is false
      |
m4.large jest niedostępny w bezpłatnym koncie AWS!

```

Blok `postcondition` jest przeznaczony do wychwytywania błędów po wykonaniu polecenia `terraform apply`. Można np. dodać blok `postcondition` do zasobu `aws_autoscaling_group` w celu sprawdzenia, czy grupa ASG została wdrożona w więcej niż tylko jednej strefie dostępności (ang. *availability zone*, AZ), a tym samym sprawdzić, czy można tolerować awarię co najmniej jednej strefy dostępności.

```

resource "aws_autoscaling_group" "example" {
  name                = var.cluster_name
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier = var.subnet_ids

  lifecycle {
    postcondition {
      condition = length(self.availability_zones) > 1
      error_message = "W celu zapewnienia wysokiej dostępności trzeba użyć więcej niż tylko jednej AZ!"
    }
  }
}

# (...)
}

```

Zwróć uwagę na użycie słowa kluczowego `self` w parametrze `condition`. Wyrażenie `self` używa takiej składni:

```
self.<ATRYBUT>
```

Tej składni specjalnej można używać wyłącznie w blokach `postcondition`, `connection` i `provisioner` (przykłady tych dwóch ostatnich bloków znajdziesz w dalszej części rozdziału) w celu odwołania się do ATRYBUTU zasobu zawierającego dany blok. Jeżeli spróbujesz użyć standardowej składni `aws_autoscaling_group.example.<ATRYBUT>`, powstanie błąd zależności cyklicznej, ponieważ zasób nie może odwoływać się do samego siebie, wyrażenie `self` jest więc rodzajem sztuczki dodanej w celu rozwiązywania tego problemu.

Jeżeli zastosujesz polecenie `terraform apply` dla tego modułu, Terraform go wdroży. Jeśli okaże się, że przekazane przez użytkownika w zmiennej danych wejściowych `subnet_id` podsieci znajdowały się w tej samej strefie dostępności, blok `postcondition` spowoduje wyświetlenie błędu. Tym samym zawsze otrzymasz ostrzeżenie, gdy grupa ASG nie jest skonfigurowana pod kątem wysokiej dostępności.

Kiedy używać bloków `validation`, `precondition` i `postcondition`?

Jak możesz zobaczyć, bloki `validation`, `precondition` i `postcondition` są całkiem podobne i dlatego być może zastanawia Cię, kiedy powinny być używane.

Bloku `validation` używaj podczas prostego oczyszczania danych wejściowych

Bloku `validation` używaj we wszystkich modułach produkcyjnych, aby uniemożliwić użytkownikom przekazywania niepoprawnych zmiennych do modułów. Celem jest przechwycenie podstawowych błędów danych wejściowych *przed* wprowadzeniem jakichkolwiek zmian. Wprawdzie bloki `precondition` mają znacznie potężniejsze możliwości, ale mimo to bloków `validation` należy używać do sprawdzania zmiennych, o ile tylko istnieje taka możliwość. Blok `validation` jest definiowany razem ze sprawdzaną zmienną, co prowadzi do powstania znacznie czytelniejszego i łatwiejszego w obsłudze API.

Bloku `precondition` używaj do sprawdzania podstawowych założeń

Bloku `precondition` używaj we wszystkich modułach produkcyjnych, aby sprawdzić założenia, które muszą być prawdziwe *przed* wprowadzeniem jakichkolwiek zmian. To obejmuje sprawdzenia zmiennych, których nie można zweryfikować za pomocą bloków `validation` (np. sprawdzenie odniesienia do wielu zmiennych lub źródeł danych), a także sprawdzenia zasobów i źródeł danych. Celem jest jak najwcześniejsze wychwycenie jak największej liczby błędów, zanim wyrażą one jakiekolwiek szkody.

Bloku `postcondition` używaj do wymuszania podstawowych gwarancji

Bloku `postcondition` używaj we wszystkich modułach produkcyjnych, aby sprawdzić, jak moduł będzie działał *po* wprowadzeniu zmian. Celem jest zagwarantowanie użytkownikowi modułu, że po użyciu polecenia `terraform apply` dany moduł będzie działał zgodnie z oczekiwaniami albo zakończy działanie z komunikatem błędu. Osoba zajmująca się modulem otrzymuje jasny sygnał, jaki sposób działania modułu ma być zagwarantowany, aby podczas refaktoryzacji nie doszło do przypadkowych strat.

Narzędzi testów zautomatyzowanych używaj do wymuszania bardziej zaawansowanych założeń i gwarancji

Bloki `validation`, `precondition` i `postcondition` są użytecznymi narzędziami, choć mogą przeprowadzać jedynie podstawowe sprawdzenia. To wynika z tego, że mogą używać jedynie źródeł danych, zasobów i wbudowanych konstrukcji języka Terraform przeznaczonych do tego rodzaju operacji. Bardzo często nie ma miejsca dla znacznie bardziej zaawansowanego sposobu działania. Jeśli np. opracowujesz moduł przeznaczony do wdrożenia usługi sieciowej, chcesz dodać sprawdzenie po wdrożeniu, aby mieć pewność, że usługa sieciowa może udzielać odpowiedzi na żądania HTTP. W tym celu możesz spróbować skorzystać z bloku `postcondition` przez wykonywanie żądań HTTP do usługi za pomocą dostawcy `http` w Terraform (<https://registry.terraform.io/providers/hashicorp/http/latest/docs>). Większość wdrożeń jednak jest przeprowadzana asynchronicznie, co może oznaczać konieczność wielokrotnego wykonywania żądań HTTP, a wymieniony dostawca nie ma wbudowanego mechanizmu powtarzania. Co więcej, jeśli wdrożysz wewnętrzną usługę sieciową, może być ona niedostępna w publicznym internecie, znajdzie więc konieczność podłączenia jej najpierw do sieci wewnętrznej lub VPN, co okazuje się trudne, gdy do dyspozycji jest tylko czysty kod Terraform. Dlatego też z myślą o bardziej niezawodnych operacjach sprawdzeń należy skorzystać z narzędzi zautomatyzowanych, takich jak OPA i Terratest, które zostaną omówione w rozdziale 9.

Moduły wersjonowane

Mamy dwa rodzaje wersjonowania, które może być stosowane podczas pracy z modułami:

- wersjonowanie zależności modułu,
- wersjonowanie samego modułu.

Rozpocznę od wersjonowania zależności modułu. Kod Terraform ma trzy typy zależności:

Plik wykonywalny Terraform

Wersja używanego przez Ciebie pliku binarnego *terraform*.

Dostawcy

Wersje poszczególnych dostawców, np. *aws*, od których zależy działanie kodu.

Moduły

Wersje poszczególnych modułów zależnych, pobieranych w blokach `module`.

Ogólnie rzecz biorąc, należy stosować **przypinanie wersji** dla wszystkich zależności. To oznacza przypięcie każdego z tych trzech typów zależności do konkretnie ustalonej wersji. Wdrożenie powinno być przewidywalne i powtarzalne: jeżeli kod nie uległ zmianie, wykonanie polecenia `terraform apply` zawsze powinno prowadzić do tego samego wyniku, niezależnie od tego, czy polecenie zostanie wykonane dzisiaj, za trzy miesiące lub za trzy lata. Aby tak się stało, trzeba uniknąć przypadkowego pobierania nowych wersji zależności. Zamiast tego uaktualnienie wersji powinno być świadomą i zamierzoną akcją, która będzie widoczna w kodzie umieszczonym w systemie kontroli wersji.

Przeanalizujemy teraz sposób przypinania wersji dla wszystkich trzech typów zależności w Terraform.

Aby przypiąć wersję pierwszego typu zależności, pliku wykonywalnego Terraform, należy użyć argumentu `required_version`. Absolutnym minimum powinno być podanie wymaganej wersji głównej Terraform.

```
terraform {  
  # Wymagana dowolna wersja 1.x Terraform.  
  required_version = ">= 1.0.0, < 2.0.0"  
}
```

To ma znaczenie krytyczne, ponieważ każda wersja główna jest niezgodna wstecz. Dlatego też uaktualnienie z wersji 1.0.0 do 2.0.0 wymaga zmian w kodzie, nie chcesz więc, aby taka zmiana została przeprowadzana przypadkowo. Przedstawiony tutaj kod pozwala na używanie jedynie dowolnej wersji 1.x.x Terraform z modulem, wersje 1.0.0 i 1.2.3 zatem działają. Dlatego, jeśli przypadkowo użyjesz wersji Terraform 0.14.3 lub 2.0.0 i spróbujesz wydać polecenie `terraform apply`, natychmiast otrzymasz komunikat błędu.

```
$ terraform apply  
Error: Unsupported Terraform Core version
```

```
This configuration does not support Terraform version 0.14.3. 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 ściślejsze deklarowanie wersji:

```
terraform {  
  # Wymagana jest dokładnie wersja 1.2.3 Terraform.  
  required_version = "1.2.3"  
}
```

W przeszłości, przed wydaniem wersji 1.0.0, było to absolutnie wymagane, ponieważ każde wydanie potencjalnie wprowadzało zmiany niezgodne z wcześniejszymi wersjami. To dotyczyło także pliku informacji o stanie: np. ten plik zapisany przez wersję 0.12.1 nie mógł być odczytywany przez wersję 0.12.0. Na szczęście po wydaniu wersji 1.0.0 to nie stanowi już problemu: zgodnie z oficjalnie opublikowaną obietnicą *Terraform v1.x Compatibility Promises* (<https://developer.hashicorp.com/terraform/language/v1-compatibility-promises>) uaktualnienia między wersjami 1.x nie powinny wymagać zmian w kodzie ani w sposobie pracy.

Mimo wszystko nadal możesz nie chcieć *przypadkowego* uaktualnienia do nowej wersji Terraform. Kolejne wydania wprowadzają nowe funkcjonalności i jeśli w jednym z komputerów (w stacji roboczej programisty lub na serwerze CI) taka nowa funkcjonalność zacznie być używana, w innych komputerach zaś wciąż będzie starsza wersja Terraform, mogą pojawić się problemy. Ponadto nowe wersje Terraform mogą zawierać błędy i zapewne chcesz je najpierw przetestować w środowiskach przedprodukcyjnych, a dopiero później zainstalować w produkcji. Dlatego też przypięcie wersji głównej Terraform to absolutne minimum. Zalecam także przypięcie wersji mniejszej i poprawki, wreszcie — świadome przeprowadzanie uaktualnień, ostrożnie, spójnie we wszystkich środowiskach.

Czasami może się zdarzyć używanie odmiennych wersji Terraform w pojedynczej bazie kodu. Przykładowo w środowisku roboczym testujesz wersję Terraform 1.2.3, podczas gdy w produkcyjnym

nadal jest używane wydanie 1.0.0. Konieczność obsługi wielu wersji Terraform, niezależnie od tego, czy we własnym komputerze, czy na serwerze ciągłej integracji, może być wyzwaniem. Na szczęście istnieje narzędzie typu open source o nazwie tfenv (<https://github.com/tfutils/tfenv>), czyli menedżer wersji Terraform, dzięki któremu staje się to znacznie łatwiejsze.

Na najbardziej podstawowym poziomie tfenv używamy do zainstalowania wielu wersji Terraform i przełączania się między nimi. I tak z polecenia `tfenv install` można skorzystać, by zainstalować konkretną wersję Terraform.

```
$ tfenv install 1.2.3
```

```
Installing Terraform v1.2.3
```

```
Downloading release tarball from
```

```
https://releases.hashicorp.com/terraform/1.2.3/terraform\_1.2.3\_darwin\_amd64.zip
```

```
Archive: tfenv_download.ZUS3Qn/terraform_1.2.3_darwin_amd64.zip
```

```
  inflating: /opt/homebrew/Cellar/tfenv/2.2.2/versions/1.2.3/terraform
```

```
Installation of terraform v1.2.3 successful.
```



tfenv i chip Apple Silicon (M1, M2)

W czerwcu 2022 roku narzędzie tfenv nadal nie potrafi zainstalować poprawnej wersji Terraform w komputerze wyposażonym w chip Apple Silicon, np. Mac z procesorem M1 lub M2 (więcej informacji na ten temat znajdziesz na stronie <https://github.com/tfutils/tfenv/issues/306>). Rozwiązaniem jest przypisanie zmiennej środowiskowej `TFENV_ARCH` wartości `arm64`:

```
$ export TFENV_ARCH=arm64
```

```
$ tfenv install 1.2.3
```

Zainstalowane wersje są wyświetlane za pomocą polecenia `tfenv list`:

```
$ tfenv list
```

```
1.2.3
```

```
1.1.4
```

```
1.1.0
```

```
* 1.0.0 (set by /opt/homebrew/Cellar/tfenv/2.2.2/version)
```

Jeżeli chcesz wybrać dowolną wersję Terraform z tej listy, użyj polecenia `tfenv use`:

```
$ tfenv use 1.2.3
```

```
Switching default version to v1.2.3
```

```
Switching completed
```

Te polecenia okazują się przydatne podczas pracy z wieloma wersjami Terraform. Jednak prawdziwie potężne możliwości tfenv kryją się w obsłudze plików `.terraform-version`. Polecenie `tfenv` będzie automatycznie wyszukiwało pliku `.terraform-version` w katalogu bieżącym, a także w katalogach nadrzędnych, aż do katalogu głównego projektu — tzn. katalogu głównego systemu kontroli wersji (np. katalogu zawierającego katalog `.git`). Jeżeli znajdzie ten plik, każde polecenie `terraform` będzie automatycznie używało wersji zdefiniowanej w tym pliku.

Jeśli chcesz np. wypróbować wersję Terraform 1.2.3 w środowisku roboczym, a przy tym pozostać przy wersji 1.0.0 w środowisku produkcyjnym, możesz skorzystać z takiej struktury katalogów:

```
live
└─ stage
```

```

L vpc
L mysql
L frontend-app
L .terraform-version
L prod
L vpc
L mysql
L frontend-app
L .terraform-version

```

Zawartość pliku *live/stage/.terraform-version* będzie taka:

```
1.2.3
```

Tymczasem zawartość pliku *live/prod/.terraform-version* będzie wyglądać następująco:

```
1.0.0
```

Teraz każde polecenie terraform wydane w katalogu *stage* lub dowolnym jego podkatalogu będzie automatycznie używało Terraform 1.2.3. Można to sprawdzić za pomocą polecenia `terraform version`:

```

$ cd stage/vpc
$ terraform version
Terraform v1.2.3

```

Podobnie każde polecenie terraform wydane w katalogu *prod* będzie automatycznie używało Terraform 1.0.0.

```

$ cd prod/vpc
$ terraform version
Terraform v1.0.0

```

Takie rozwiązanie działa automatycznie w stacji roboczej programisty i w serwerze ciągłej integracji, o ile wszyscy mają zainstalowane polecenie `tfnv`. Jeśli natomiast używasz Terragrunt, to `tgswitch` (<https://github.com/warrenbox/tgswitch>) oferuje podobną funkcjonalność automatycznego wybierania wersji Terragrunt na podstawie pliku *.terragrunt-version*.

Zwróć teraz uwagę na drugi typ zależności w kodzie Terraform: dostawcy. Jak pokazałem w rozdziale 7., do przypięcia wersji dostawcy można użyć bloku `required_providers`:

```

terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

```

Ten kod powoduje przypięcie wersji dostawcy AWS do dowolnej wersji 4.x (składnia `~> 4.0` jest odpowiednikiem `>= 4.0, < 5.0`). Także w wypadku dostawcy absolutnym minimum będzie przypięcie wersji głównej, aby uniknąć przypadkowego wprowadzenia zmian niezgodnych wstecz. W Terraform 0.14.0 i nowszych wydaniach nie trzeba przypinać wersji mniejszej lub poprawki

dla dostawców, ponieważ to odbywa się automatycznie ze względu na *plik blokady*. W trakcie pierwszego wydania polecenia `terraform init` następuje utworzenie pliku `.terraform.lock.hcl`, w którym zostają zapisane poniższe informacje:

Dokładna wersja każdego użytego dostawcy

Jeżeli plik `.terraform.lock.hcl` umieścisz w systemie kontroli wersji (co należy zrobić!), to w przyszłości po ponownym wydaniu polecenia `terraform init`, w tym lub innym komputerze, Terraform pobierze *dokładnie* tę samą wersję poszczególnych dostawców. Dlatego też nie trzeba przypisać wersji mniejszej i poprawki w bloku `required_providers`, ponieważ to i tak jest domyślny sposób działania. Jeżeli chcesz wyraźnie uaktualnić wersję dostawcy, możesz zmienić ograniczenie w bloku `required_providers`, a następnie użyć polecenia `terraform init -upgrade`. Terraform pobierze nowych dostawców dopasowanych do wersji podanej w ograniczeniu i uaktualnia zawartość pliku `.terraform.lock.hcl`. Dlatego też te uaktualnienia należy przeglądać i umieszczać w systemie kontroli wersji.

Suma kontrolna każdego dostawcy

Terraform rejestruje sumę kontrolną każdego pobieranego dostawcy. Kolejne wykonania polecenia `terraform init` będą wyświetlały komunikat błędu, jeśli ta suma kontrolna ulegnie zmianie. To jest rodzaj zabezpieczenia, aby mieć pewność, że w przyszłości nikt nie zastąpi kodu dostawcy kodem o złośliwym działaniu. Jeżeli dostawca jest podpisany kryptograficznie (tak jest w wypadku większości dostawców HashiCorp), Terraform sprawdza ten podpis, co jest dodatkowym zabezpieczeniem i potwierdzeniem, że kod jest wart zaufania.



Plik blokady i wiele systemów operacyjnych

Domyślnie Terraform rejestruje sumę kontrolną jedynie na platformie, na której zostało wykonane polecenie `terraform init`. Jeśli będzie to np. Linux, w pliku `.terraform.lock.hcl` Terraform zarejestruje sumy kontrolne plików binarnych dostawców wyłącznie w wersji dla systemu Linux. Jeżeli ten plik trafi do systemu kontroli wersji i później polecenie `terraform init` zostanie wykonane w systemie np. macOS, otrzymasz komunikat błędu, ponieważ plik nie będzie zawierał sum kontrolnych dostawców w wersji dla macOS. Gdy system korzysta z różnych systemów operacyjnych, wówczas trzeba użyć polecenia `terraform providers lock` w celu zarejestrowania sum kontrolnych dla każdej używanej platformy:

```
terraform providers lock \  
-platform=windows_amd64 \ # 64-bitowy Windows.  
-platform=darwin_amd64 \  # 64-bitowy macOS.  
-platform=darwin_arm64 \   # 64-bitowy macOS (ARM).  
-platform=linux_amd64      # 64-bitowy Linux.
```

Przechodzimy teraz do trzeciego typu zależności: modułów. Jak już wspomniałem we wcześniejszej części rozdziału, gorąco zachęcam do przypinania wersji modułu za pomocą adresu URL przypisanego zmiennej `source` (zamiast używać lokalnych ścieżek dostępu) i parametru `ref`, któremu jest przypisany tag Gita:

```
source = "git@github.com:foo/modules.git//services/hello-world-app?ref=v0.0.5"
```

Jeżeli będziesz używać takich adresów URL, w trakcie każdego wykonania polecenia `terraform init` Terraform zawsze pobierze dokładnie ten sam kod dla modułu.

Po omówieniu wersjonowania zależności kodu przechodzimy do wersji samego kodu. 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 powinieneś umieścić w pliku *live/stage/services/hello-world-app/main.tf*:

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

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      = "Nowy serwer"
  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
  ami              = data.aws_ami.ubuntu.id
}

data "aws_ami" "ubuntu" {
  most_recent = true
  owners     = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}
```

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. `rds`).
- 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.

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>"
  # (...)
}
```

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 RDS z Terraform Registry:

⁶ Szczegółowe informacje związane z publikowaniem modułów w repozytorium Terraform Registry znajdziesz na stronie <https://developer.hashicorp.com/terraform/registry/modules/publish>.

```
module "rds" {
  source = "terraform-aws-modules/rds/aws"
  version = "4.4.0"

  # (...)
}
```

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 możesz mieć katalog *modules/packer* zawierający szablon Packer i skrypty Bash używane do skonfigurowania obrazu AMI, a także moduł Terraform *modules/asg-rolling-deploy* w celu wdrożenia tego obrazu AMI.

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) i *file* (skopiowanie plików do zdalnego zasobu)⁷.

⁷ Pełną listę dostępnych bloków *provisioner* znajdziesz na stronie <https://developer.hashicorp.com/terraform/language/resources/provisioners/syntax>.

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      = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo \"Witaj, świecie z $(uname -smp)\""
  }
}
```

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.

```
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}

resource "aws_instance" "example" {
  ami               = data.aws_ami.ubuntu.id
  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 źródła danych `aws_ami` w celu znalezienia 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               = data.aws_ami.ubuntu.id
  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           = data.aws_ami.ubuntu.id
  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. 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...

```

(Te polecenia powtarzają się jeszcze kilkakrotnie...)

```

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 jedyną zaletę bloku `provisioner` podczas wykonywania kodu w egzemplarzu EC2 można uznać dowolną wielkość skryptu bloku `provisioner`, podczas gdy skrypt danych użytkownika jest ograniczony do 16 KB.

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 argumencie triggers.

```
resource "null_resource" "example" {
  # Użycie 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 8.2 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 8.1 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 dla wszystkich zależności, czyli pliku wykonywalnego Terraform, dostawców Terraform i modułów Terraform.
4. Utwórz katalog *test* i umieść w nim zautomatyzowane testy dla każdego przykładu.

Ostatni punkt — sposoby tworzenia zautomatyzowanych testów dla kodu infrastruktury — będzie tematem następnego rozdziału.

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'
    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
```

Ten fragment kodu spowoduje udzielenie odpowiedzi wraz z kodem stanu 200 OK i komunikatem Witaj, świecie dla żądań do adresu URL /, kodem stanu 201 i treścią JSON dla adresu URL /api, a także 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
```

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 8. Najłatwiejszy sposób na ręczne przetestowanie modułu `alb` polega na użyciu przykładowego kodu utworzonego w katalogu *examples/alb*.

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

module "alb" {
  source = "../../modules/networking/alb"

  alb_name   = "terraform-up-and-running"
  subnet_ids = data.aws_subnets.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
```

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.



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 wprowadzić można wykorzystać wraz z dowolnym typem infrastruktury, ale procedura weryfikacji będzie się zmieniała w zależności od przedmiotu testu.

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 rzeczywistości, 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. Przykładami takich narzędzi są `cloud-nuke` (<https://github.com/gruntwork-io/cloud-nuke>) i `aws-nuke` (<https://github.com/rebuy-de/aws-nuke>).

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.

¹ AWS nie wymaga żadnych dodatkowych opłat za kolejne konto. Jeśli korzystasz z usługi AWS Organizations, możesz tworzyć wiele kont „potomnych” połączonych (także pod względem rachunków) z jednym kontem „głównym”, jak pokazałem w rozdziale 7.

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
```



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.

Testy zautomatyzowane

Idea stojąca za testami zautomatyzowanymi polega na tworzeniu kodu sprawdzającego, czy rzeczywisty kod działa zgodnie z oczekiwaniami. Jak zobaczysz w rozdziale 10., 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 tzw. **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. aplikacje, 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 spójrz 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
      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 konieczne jest utworzenie egzemplarzy `WebServer`, `request` i `response` lub też ich imitacji, co będzie wymagało 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 nieznalesionych — 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

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
end

```

Oto jak zostają wykonane testy jednostkowe:

```

$ 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 0,000572 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.

Testy jednostkowe kodu Terraform

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 wielokrotnego użycia, 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 operację porządkową.

Innymi słowy, wykonywane są *dokładnie* te same kroki jak w przypadku ręcznego przeprowadzania testów, choć poszczególne kroki zostały zapisane w postaci kodu. Szczerze 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.

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://terratest.gruntwork.io/>), która zapewnia obsługę testowania z użyciem różnych narzędzi infrastruktury jako kodu (takich jak Terraform, Packer,

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 o nazwie LocalStack (<https://registry.terraform.io/providers/hashicorp/aws/latest/docs/guides/custom-service-endpoints#localstack>). 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.

Docker, Helm) i 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>. Wymagana jest wersja minimum 1.13.
2. Utworzenie katalogu przeznaczonego dla kodu testowego, np. o nazwie `test`.
3. Wydanie polecenia `go mod init <NAZWA>` w utworzonym przed chwilą katalogu, gdzie NAZWA to nazwa katalogu używanego przez ten zbiór testów, zwykle w formacie `github.com/<NAZWA_ORGANIZACJI>/<NAZWA_PROJEKTU>` (np. `go mod init github.com/brikis98/terraform-up-and-running`). To powinno spowodować utworzenie pliku `go.mod`, używanego następnie do śledzenia zależności w kodzie Go.

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`:

```
$ go test -v
```

Opcja `-v` powoduje wyświetlenie szczegółowych danych wyjściowych, co gwarantuje, że zawsze będą wyświetlone pełne dane wyjściowe. Powinny zostać wygenerowane takie dane wyjściowe:

```
=== RUN    TestGoIsWorking

Jeżeli widzisz ten komunikat, wszystko działa prawidłowo!

--- PASS: TestGoIsWorking (0.00s)
PASS
ok      github.com/brikis98/terraform-up-and-running-code    0.192s
```

Jeżeli pierwszy test został zaliczony, możesz usunąć plik `go_sanity_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).

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 z wykorzystaniem do tego pakietu `http_helper` (upewnij się o zaimportowaniu `github.com/gruntwork-io/terratest/modules/http-helper`):

```
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,
    nil,
    expectedStatus,
    expectedBody,
    maxRetries,
    timeBetweenRetries,
)
}
```

Metoda `http_helper.HttpGetWithRetry()` spróbuje wykonać żądanie HTTP GET pod podany adres URL i sprawdzi, czy udzielona na nie odpowiedź zawiera oczekiwany kod stanu i treść. Jeżeli nie — 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. Tego rodzaju logika jest powszechnie stosowana podczas testów infrastruktury, ponieważ zwykle istnieje pewna przerwa między zakończeniem wykonywania `terraform apply` i osiągnięciem pewnej gotowości przez wdrożoną infrastrukturę (np. czasu wymagają operacje sprawdzenia stanu, uaktualnienia DNS itd.). Skoro nie wiadomo, ile to potrwa, najlepszym rozwiązaniem jest powtarzanie żądania aż do chwili przekroczenia czasu oczekiwania.

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.HttpGetWithRetry()` 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,
    nil,
    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.



Wersja Terratest

Kod zamieszczony w książce został utworzony za pomocą Terratest v0.39.0. Terratest to wciąż narzędzie, które nie osiągnęło wersji 1.0.0, więc nowsze wydania mogą zawierać zmiany niezgodne wstecz. Aby mieć pewność co do prawidłowego działania przykładów zamieszczonych w książce, zalecam instalację Terratest w tej konkretnej wersji, 0.39.0, a nie najnowszej dostępnej. W tym celu przejdź do pliku `go.mod` i na jego końcu dodaj wiersz:

```
require github.com/gruntwork-io/terratest v0.39.0
```

Ponieważ mamy do czynienia z zupełnie nowym projektem w Go, jednorazową operacją jest nakazanie mu pobrania zależności (w tym Terratest). Najłatwiejszym rozwiązaniem będzie w tym miejscu użycie polecenia:

```
go mod tidy
```

To pobierze wszystkie zależności i utworzy plik `go.sum` w celu zablokowania konkretnych wersji użytych w projekcie.

Skoro powoduje on wdrożenie infrastruktury w AWS, przed wykonaniem testu konieczne jest uwierzytelnienie konta AWS w zwykły sposób (zapoznaj się z opcjami uwierzytelnienia omówionymi w rozdziale 2.). 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 `a1b` 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.

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
```

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 wykorzystwała 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"
}

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
    ami                 = data.aws_ami.ubuntu.id
  }

  data "aws_ami" "ubuntu" {
    most_recent = true
    owners      = ["099720109477"] # Canonical.

    filter {
      name   = "name"
      values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
    }
  }
}

```

Problem związany z zależnościami staje się widoczny, gdy dostrzeżesz parametry `db_remote_state_bucket` i `db_remote_state_key`: w module `hello-world-app` zostało przyjęte założenie o wdrożeniu modułu `mysql`. Ponadto wymagane jest przekazanie szczegółowych informacji o kubelku 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_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [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`). użytą tutaj wartością domyślną jest `null`.

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
ami                = data.aws_ami.ubuntu.id
}

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 kubełka S3 dla bazy danych informacji o stanie Terraform"
  type        = string
  default     = null
}

variable "db_remote_state_key" {
  description = "Ścieżka dostępu w kubełku 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_subnets" "default" {
  count = var.subnet_ids == null ? 1 : 0
  filter {
    name     = "vpc-id"
    values   = [data.aws_vpc.default.id]
  }
}

```

Teraz można uaktualnić wszelkie odwołania do tych źródeł danych w celu warunkowego użycia zmiennej danych wejściowych lub źródła danych. Przechwycimy 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_subnets.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_subnets i użycia local.vpc_id.

```

data "aws_subnets" "default" {
  count = var.subnet_ids == null ? 1 : 0
  filter {
    name     = "vpc-id"
    values   = [local.vpc_id]
  }
}

```

Następnie zmodyfikuj w taki sposób, aby parametr subnet_ids w module alb korzystał z local.subnet_ids.

```

module "alb" {
  source = "../../networking/alb"

  alb_name = "hello-world-${var.environment}"
  subnet_ids = local.subnet_ids
}

```

W module `asg` natomiast należy wprowadzić takie uaktualnienia: parametr `subnet_ids` do `local.subnet_ids`, w zasobie `user_data` trzeba uaktualnić zmienne `db_address` i `db_port`, aby odczytywały dane z `local.mysql_config`.

```
module "asg" {
  source = "../../cluster/asg-rolling-deploy"

  cluster_name = "hello-world-${var.environment}"
  ami          = var.ami
  instance_type = var.instance_type

  user_data = templatefile("${path.module}/user-data.sh", {
    server_port = var.server_port
    db_address  = local.mysql_config.address
    db_port     = local.mysql_config.port
    server_text = var.server_text
  })

  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
}
```

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
    }
}
```

Przełącz 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
    ami           = data.aws_ami.ubuntu.id
}
```

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)

    maxRetries := 10
    timeBetweenRetries := 10 * time.Second

    http_helper.HttpGetWithRetryWithCustomValidation(
        t,
        url,
        nil,
        maxRetries,
        timeBetweenRetries,
        func(status int, body string) bool {
```

```

        return status == 200 &&
            strings.Contains(body, "Witaj, świecie")
    },
)
}

```

Ten kod jest niemalże identyczny z testem jednostkowym dla przykładu modułu `alb`. Występują między nimi tylko dwie różnice:

- Ustawienie `TerraformDir` wskazuje przykład `hello-world-app` (upewnij się o podaniu odpowiedniej ścieżki dostępu dla Twojego systemu plików).
- Zamiast metody `http_helper.HttpGetWithRetry()` do sprawdzenia pod kątem odpowiedzi z kodem stanu 404 w teście została użyta metoda `http_helper.HttpGetWithRetryWithCustomValidation()` sprawdzająca, czy udzielona odpowiedź zawiera kod stanu 200 i treść w postaci komunikatu `Witaj, świecie`. Skrypt danych użytkownika moduły `hello-world-app` zwraca odpowiedź z kodem stanu 200, zawierającą nie tylko komunikat serwera, ale także inny tekst, np. kod HTML.

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_subnets.default.ids
}
```

Kolejnym krokiem jest przypisanie unikatowej wartości zmiennej w pliku *test/alb_example_test.go*:

```
package test

import (
    "fmt"
    "github.com/stretchr/testify/require"

    "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()),
        },
    }
    // (...)
}
```

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 (62⁶ 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
  ami           = data.aws_ami.ubuntu.id
}
```

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{
        // 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,
            },
            "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.

Zauważ, że domyślnie liczba jednocześnie wykonywanych testów Go odpowiada liczbie procesorów komputera. Dlatego też, jeśli masz tylko jeden procesor, domyślnie testy wciąż będą wykonywane sekwencyjnie, a nie równolegle. Można to zmienić za pomocą zmiennej środowiskowej `GOMAXPROCS` albo argumentu `-parallel` polecenia `go`. Przykładowo w celu wymuszenia na Go jednoczesnego wykonywania dwóch testów polecenie `go` powinno mieć taką postać:

```
$ go test -v -timeout 30m -parallel 2
```



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. 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 powinien mieć już zdefiniowany w plikach *live/stage/data-stores/mysql* i *live/stage/services/hello-world-app*. Możesz utworzyć test integracji dla (fragmentów) środowiska roboczego.

Oczywiście, jak już wspomniałem w rozdziale, wszystkie testy zautomatyzowane powinny działać w odizolowanym koncie AWS. Dlatego też podczas testowania kodu przeznaczonego dla środowiska roboczego powinieneś uwierzytelnić odizolowane środowisko testowe, a następnie w nim przeprowadzić te testy. Jeżeli moduły mają zdefiniowane na stałe jakiegokolwiek 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 za pomocą parametru danych wejściowych `db_name`.

```
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 szczegóły implementacji umieścimy w nim później.

```
// Te wartości zastąp 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_username": "admin",
            "db_password": "password",
        },
    }
}

```

Jak dotąd niezbyt wiele się dzieje: kod wskazuje terraform.Options w przekazanym katalogu i przypisuje wartości zmiennych db_name, db_username 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.

```

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:

```
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_username": "admin",
            "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 kubelku) 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"
}
```

Przekaż 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
  ami              = data.aws_ami.ubuntu.id
}
```

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,
        nil,
        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`. 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 `helloOpts` na dysku. W tym momencie prawdopodobnie już się domyślasz, jak wygląda implementacja metody `teardownApp()`:

```
func teardownApp(t *testing.T, helloAppDir string) {
    helloOpts := test_structure.LoadTerraformOptions(t, helloAppDir)
    defer terraform.Destroy(t, helloOpts)
}
```

Spójrz na implementację metody `validateApp()`:

```
func validateApp(t *testing.T, helloAppDir string) {
    helloOpts := test_structure.LoadTerraformOptions(t, helloAppDir)
    validateHelloApp(t, helloOpts)
}
```

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 modułu 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 pomiń 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. Wprawdzie 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 powinienś 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 otrzymywałem 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?
```

```
(...)
```

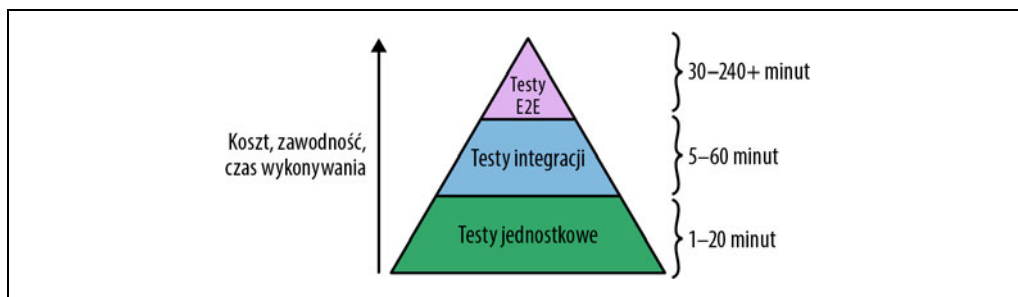
```
Running command terraform with args [apply -input=false -lock=false -auto-approve]
```

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 9.1.

Idea piramidy testów polega na tym, że zwykle powinieneś celować w dużą liczbę testów jednostkowych (dolna część piramidy), mniejszą liczbę testów integracji (środkowa część piramidy) i jeszcze mniejszą liczbę testów E2E (górną 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.



Rysunek 9.1. Piramida testów

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 przestrzennymi nazwami, 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 wspominałem, 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 poradenia 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. zastosowaniu zmiany 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.

Inne podejścia w zakresie testów

W większej części rozdziału skoncentrowałem się na używaniu Terratest do testowania kodu za pomocą cyklu poleceń terraform apply i terraform destroy. Wprawdzie to jest reguła podczas testowania, ale w zakresie podejścia do testowania istnieją jeszcze trzy inne kategorie, które można wykorzystać:

- analiza statyczna,
- testowanie planu,
- testowanie serwera.

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 dokładniej omówię nowe kategorie.

Analiza statyczna

Analiza statyczna to najprostsze rozwiązanie w zakresie testowania kodu Terraform: przetwarzasz go i analizujesz bez faktycznego wykonywania w jakikolwiek sposób. W tabeli 9.1 wymieniłem wybrane narzędzia współpracujące z Terraform w tym zakresie i porównałem je pod kątem popularności i dojrzałości, na podstawie liczby gwiazdek zebranych przez poszczególne projekty w serwisie GitHub (stan na luty 2022 roku).

Tabela 9.1. Porównanie wybranych najpopularniejszych narzędzi analizy statycznej dla Terraform

	terraform validate	tfsec	tflint	Terrascan
Krótki opis	wbudowane polecenie Terraform	wyszukuje potencjalne problemy z zabezpieczeniami	Linter Terraform	wykrywa złamanie polityk zgodności i bezpieczeństwa
Licencja	(tak samo jak w Terraform)	MIT	MPL 2.0	Apache 2.0
Firma stojąca za produktem	(tak samo jak w Terraform)	Aqua Security	brak	Accurics
Gwiazdki	(tak samo jak w Terraform)	3874	2853	2768
Współpracownicy	(tak samo jak w Terraform)	96	77	63
Pierwsze wydanie	(tak samo jak w Terraform)	2019	2016	2017
Najnowsze wydanie	(tak samo jak w Terraform)	1.1.2	0.34.1	1.13.0
Wbudowane sprawdzenia	tylko składnia	AWS, Azure, GCP, Kubernetes, DigitalOcean itd.	AWS, Azure i GCP	AWS, Azure, GCP, Kubernetes itd.
Niestandardowe sprawdzenia	nieobsługiwane	zdefiniowane w YAML lub JSON	zdefiniowane we wtyczce Go	zdefiniowane w Rego

Najprostszym z tych narzędzi jest `terraform validate`, które jest wbudowane w Terraform i potrafi wychwytywać problemy dotyczące składni. Jeśli np. zapomnisz o zdefiniowaniu parametru `alb_name` w `examples/alb` i wykonasz polecenie `terraform validate`, otrzymasz dane wyjściowe podobne do tych:

```
$ terraform validate
```

```
Error: Missing required argument

on main.tf line 20, in module "alb":
 20: module "alb" {

The argument "alb_name" is required, but no definition was found.
```

Pamiętaj, że działanie tego narzędzia ogranicza się jedynie do sprawdzenia składni, pozostałe natomiast pozwalają na wymuszanie innych typów polityki. Można np. zastosować narzędzia typu `tfsec` i `tflint` do wymuszania polityki:

- Grupy bezpieczeństwa nie mogą być zbyt otwarte, np. blokowanie reguł ruchu przychodzącego pozwalających na dostęp ze wszystkich adresów IP (blok CIDR `0.0.0.0/0`).
- Wszystkie egzemplarze EC2 muszą stosować określoną konwencję tagów.

Idea polega na **definiowaniu polityki jako kodu**, co pozwala zapewnić za jego pomocą bezpieczeństwo, zgodność i niezawodność. W kolejnych punktach poznasz inne narzędzia pozwalające definiować politykę w postaci kodu.

Zalety narzędzi analizy statycznej:

- Wydajność działania.
- Łatwość użycia.
- Stabilność (brak wariujących testów).
- Brak konieczności uwierzytelniania u prawdziwego dostawcy (np. w rzeczywistym koncie AWS).
- Brak konieczności wdrażania i usuwania zasobów.

Wady narzędzi analizy statycznej:

- Ograniczona liczba typów błędów, które te narzędzia są w stanie wychwytywać. To są jedynie te błędy, które można znaleźć podczas statycznego odczytywania kodu, bez jego wykonania: błędy składni, błędy typu i niewielki podzbiór błędów logiki biznesowej. Przykładowo istnieje możliwość wychwycenia niezastosowania polityki dla wartości statycznych, takich jak na stałe zdefiniowana grupa bezpieczeństwa pozwalająca na dostęp z bloku CIDR `0.0.0.0/0`. Nie można wykrywać przypadków złamania polityki dla wartości dynamicznych, takich jak grupa bezpieczeństwa z blokiem CIDR odczytywanym ze zmiennej lub pliku.
- Testy nie sprawdzają funkcjonalności. Istnieje więc niebezpieczeństwo, że pomimo zaliczenia wszystkich testów infrastruktura nie będzie działała.

Testowanie planu

Innym sposobem na przetestowanie kodu jest zastosowanie polecenia `terraform plan` i przeanalizowanie danych wyjściowych planu. Kod jest wykonywany, zatem to więcej niż jedynie analiza statyczna, ale jednocześnie mniej niż testy jednostkowe lub integracji, ponieważ kod nie jest wykonywany w pełni: polecenie `terraform plan` wykonuje kroki odczytujące dane (np. pobieranie stanu, wykonywanie źródeł danych), ale nie kroki zapisujące dane (np. tworzenie lub modyfikowanie zasobów). W tabeli 9.2 wymienilem wybrane narzędzia przeznaczone do testowania planu i porównałem je pod kątem popularności i dojrzałości, na podstawie liczby gwiazdek zebranych przez poszczególne projekty w serwisie GitHub (stan na luty 2022 roku).

Tabela 9.2. Porównanie wybranych z najpopularniejszych narzędzi testowania planu w Terraform

	Terratest	Open Policy Agent (OPA)	HashiCorp Sentinel	Checkov	terraform-compliance
Krótki opis	biblioteka Go dla testowania IaC	silnik polityk ogólnego przeznaczenia	polityka jako kod dla produktów korporacyjnych HashiCorp	polityka jako kod dla każdego	framework testów BDD dla Terraform
Licencja	Apache 2.0	Apache 2.0	komercyjna/licencja własnościowa	Apache 2.0	MIT
Firma stojąca za produktem	Gruntwork	Styła	HashiCorp	Bridgecrew	brak
Gwiazdki	5888	6207	(to nie jest open source)	3578	1104
Współpracownicy	157	237	(to nie jest open source)	199	36
Pierwsze wydanie	2016	2016	2017	2019	2018
Najnowsze wydanie	0.40.0	0.37.1	0.18.5	2.0.810	1.3.31
Wbudowane sprawdzenia	brak	brak	brak	AWS, Azure, GCP, Kubernetes itd.	brak
Niestandardowe sprawdzenia	zdefiniowane w Go	zdefiniowane w Rego	zdefiniowane w Sentinel	zdefiniowane w Pythonie lub YAML	zdefiniowane w BDD

Skoro znasz już narzędzie Terratest, zobacz teraz, jak można z niego skorzystać przy testowaniu planu wykonania kodu *examples/alb*. Jeżeli ręcznie wykonasz polecenie `terraform plan`, otrzymasz dane wyjściowe, których fragment przedstawia się następująco:

Terraform will perform the following actions:

```
# Zostanie wykonany kod module.alb.aws_lb.example
+ resource "aws_lb" "example" {
+   arn                        = (known after apply)
+   load_balancer_type        = "application"
+   name                      = "test-4Ti6CP"
+   (...)
}
```

```
}
(...)
```

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

W jaki sposób te dane wyjściowe mogą być przetestowane programistycznie? Oto podstawowa struktura testu używającego metody pomocniczej `InitAndPlan()` TerraTest w celu automatycznego wykonywania poleceń `terraform init` i `terraform plan`.

```
func TestALBExamplePlan(t *testing.T) {
    t.Parallel()

    albName := fmt.Sprintf("test-%s", random.UniqueId())

    opts := &terraform.Options{
        // Należy uaktualnić tę względną ścieżkę dostępu,
        // aby prowadziła do katalogu examples/alb!
        TerraformDir: "../examples/alb",
        Vars: map[string]interface{}{
            "alb_name": albName,
        },
    }

    planString := terraform.InitAndPlan(t, opts)
}
```

Nawet ten minimalny test oferuje pewną wartość, ponieważ sprawdza, czy kod pozwala na zakończone sukcesem wykonanie polecenia `terraform plan`, które z kolei sprawdza poprawność składni i możliwość użycia wszystkich API odczytu danych. Można pójść dalej. Jednym z drobnych usprawnień jest sprawdzenie, czy na końcu planu otrzymujemy oczekiwane liczby dotyczące zasobów, 5 to `add`, 0 to `change`, 0 to `destroy`. Do tego celu służy metoda pomocnicza `GetResourceCount()`.

```
// Przykład pokazujący, jak sprawdzić liczniki add/change/destroy danych wyjściowych planu Terraform.
resourceCounts := terraform.GetResourceCount(t, planString)
require.Equal(t, 5, resourceCounts.Add)
require.Equal(t, 0, resourceCounts.Change)
require.Equal(t, 0, resourceCounts.Destroy)
```

Jeszcze więcej można zrobić za pomocą metody pomocniczej `InitAndPlanAndShowWithStructNoLogTempPlanFile()` pozwalającej skonwertować dane wyjściowe polecenia `terraform plan` na postać struktury, która zapewnia programowy dostęp do wszystkich wartości i zmian w danych wyjściowych `terraform plan`. Można np. sprawdzić, czy dane wyjściowe zawierają zasób `aws_lb` o adresie `module.alb.aws_lb.example` i czy atrybut `name` tego zasobu ma oczekiwaną wartość:

```
// Przykład sprawdzenia określonych wartości w danych wyjściowych terraform plan.
planStruct :=
    terraform.InitAndPlanAndShowWithStructNoLogTempPlanFile(t, opts)
alb, exists :=
    planStruct.ResourcePlannedValuesMap["module.alb.aws_lb.example"]
require.True(t, exists, "aws_lb resource must exist")

name, exists := alb.AttributeValues["name"]
require.True(t, exists, "missing name parameter")
require.Equal(t, albName, name)
```

Zaletą podejścia Terratest do testowania planu jest niezwykła elastyczność, ponieważ można tworzyć dowolny kod w Go przeznaczony do sprawdzenia praktycznie czegośkolwiek. To jednocześnie w pewnych kategoriach można uznać za poważną wadę, ponieważ utrudnia rozpoczęcie tworzenia testów.

Niektóre zespoły preferują bardziej deklaracyjny język do definiowania polityki jako kodu. W ostatnich latach narzędzie Open Policy Agent (OPA) stało się dość popularnym narzędziem *polityki jako kodu*, ponieważ pozwala na definiowanie polityki firmy jako kodu w deklaracyjnym języku Rego.

Przykładowo wiele firm oznacza tagami polityki, które mają być wymuszane. W wypadku kodu Terraform powszechne jest gwarantowanie, że każdy zasób zarządzany przez Terraform ma tag `ManagedBy = terraform`. Oto prosta polityka *enforce_tagging.rego*, której można użyć do sprawdzenia pod kątem tego tagu.

```
package terraform

allow {
  resource_change := input.resource_changes[_]
  resource_change.change.after.tags["ManagedBy"]
}
```

Ta polityka będzie szukała zmian w danych wyjściowych terraform plan, wyodrębni tag `ManagedBy` i — jeśli jest on zdefiniowany — zmiennej OPA o nazwie `allow` przypisze wartość `true`. W przeciwnym razie wartością tej zmiennej będzie `false`.

Rozważ przedstawiony tutaj moduł Terraform:

```
resource "aws_instance" "example" {
  ami          = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
}
```

Ten moduł nie ma ustawienia w postaci wymaganego tagu `ManagedBy`. W jaki sposób można to przechwycić za pomocą OPA?

Pierwszym krokiem jest wykonanie polecenia `terraform plan` i zapisanie jego danych wyjściowych do pliku.

```
$ terraform plan -out tfplan.binary
```

OPA operuje wyłącznie na danych JSON, następnym krokiem więc jest konwersja planu pliku na JSON przy użyciu polecenia `terraform show`.

```
$ terraform show -json tfplan.binary > tfplan.json
```

Można również zastosować polecenie `opa eval` w celu sprawdzenia planu względem polityki *enforce_tagging.rego*:

```
$ opa eval \
  --data enforce_tagging.rego \
  --input tfplan.json \
  --format pretty \
  data.terraform.allow

undefined
```

Skoro tag `ManagedBy` nie został zdefiniowany, dane wyjściowe wygenerowane przez OPA to `undefined`. Teraz spróbuj zdefiniować tag `ManagedBy`:

```
resource "aws_instance" "example" {
  ami      = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  tags = {
    ManagedBy = "terraform"
  }
}
```

Ponownie użyj poleceń: `terraform plan`, `terraform show` i `opa eval`.

```
$ terraform plan -out tfplan.binary

$ terraform show -json tfplan.binary > tfplan.json

$ opa eval \
  --data enforce_tagging.rego \
  --input tfplan.json \
  --format pretty \
  data.terraform.allow

true
```

Tym razem zostały wygenerowane dane wyjściowe `true`, wskazujące na zastosowanie polityki.

Dzięki użyciu narzędzi takich jak OPA można wymuszać wymagania firmy przez utworzenie biblioteki takich polityk i zdefiniowanie potoku CI/CD stosującego te polityki w modułach Terraform po każdym ich przekazaniu do repozytorium.

Zalety narzędzi testowania planu:

- Wydajność działania — nie tak duża jak w wypadku czystych narzędzi analizy statycznej, ale znacznie większa niż w przypadku testów jednostkowych lub integracji.
- Łatwość użycia — nie tak duża jak w wypadku czystych narzędzi analizy statycznej, ale znacznie większa niż w przypadku testów jednostkowych lub integracji.
- Stabilność (mniej wariujących testów) — nie tak duża jak w wypadku czystych narzędzi analizy statycznej, ale znacznie większa niż w przypadku testów jednostkowych lub integracji.
- Brak konieczności
- wdrażania i usuwania zasobów.

Wady narzędzi testowania planu:

- Ograniczona liczba typów błędów, które te narzędzia są w stanie wychwytywać. Wychwytyują więcej błędów niż w przypadku czystej analizy statycznej, ale nawet nie zbliżają się do liczby błędów wychwytywanych przez testy jednostkowe i integracji.
- Brak konieczności uwierzytelniania u prawdziwego dostawcy (np. w rzeczywistym koncie AWS). To jest wymagane do działania polecenia `terraform plan`.

- Testy nie sprawdzają funkcjonalności. Istnieje więc niebezpieczeństwo, że pomimo zaliczenia wszystkich testów infrastruktura nie będzie działała.

Testowanie serwera

Istnieje jeszcze zbiór narzędzi testowania koncentrujących się na sprawdzaniu, czy serwery (w tym wirtualne) zostały prawidłowo skonfigurowane. Nie znam żadnej powszechnie używanej nazwy dla tych narzędzi, będę się więc posługiwał mianem **testowania serwera**. To nie są narzędzia ogólnego przeznaczenia do testowania wszystkich aspektów kodu Terraform. W rzeczywistości większość tych narzędzi została początkowo opracowana z przeznaczeniem do współdziałania z narzędziami zarządzania konfiguracją, takimi jak Chef i Puppet, które są całkowicie skoncentrowane na uruchamianiu serwerów. Jednak wraz z rosnącą popularnością Terraform powszechne stawało się używanie tych narzędzi do uruchamiania serwerów — tego rodzaju narzędzia mogą być przydatne podczas sprawdzania, czy uruchamiane serwery działają zgodnie z oczekiwaniami. W tabeli 9.3 wymienilem wybrane narzędzia testowania serwera i porównałem je pod kątem popularności i dojrzałości, na podstawie liczby gwiazdek zebranych przez poszczególne projekty w serwisie GitHub (stan na luty 2022 roku).

Tabela 9.3. Porównanie wybranych najpopularniejszych narzędzi testowania serwera

	InSpec	Serverspec	Goss
Krótki opis	framework audytu i testowania	testy RSpec dla Twoich serwerów	szybkie i łatwe testowanie i weryfikowanie serwera
Licencja	Apache 2.0	MIT	Apache 2.0
Firma stojąca za produktem	Chef	brak	brak
Gwiazdki	2472	2426	4607
Współpracownicy	279	128	89
Pierwsze wydanie	2016	2013	2015
Najnowsze wydanie	4.52.9	2.42.0	0.3.16
Wbudowane sprawdzenia	brak	brak	brak
Niestandardowe sprawdzenia	zdefiniowane w języku specjalizowanym bazującym na języku Ruby	zdefiniowane w języku specjalizowanym bazującym na języku Ruby	zdefiniowane w YAML

Większość tych narzędzi oferuje prosty **język specjalizowany** (ang. *domain-specific language*, DSL) przeznaczony do sprawdzania, czy wdrożone serwery są zgodne 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

```

Zalety narzędzi testowania serwera:

- Możliwość bardzo łatwej weryfikacji określonych właściwości serwera. Języki specjalizowane tych narzędzi są znacznie łatwiejsze w użyciu podczas wykonywania najczęstszych rodzajów operacji sprawdzania niż w przypadku ich przeprowadzania zupełnie od zera.
- Możliwość opracowania biblioteki dla polityk sprawdzania. Ponieważ każde sprawdzenie przygotowuje się dość szybko, na podstawie poprzedniego punktu można stwierdzić, że omawiane narzędzia to dobry sposób na weryfikację listy wymagań, zwłaszcza związanych ze zgodnością z PCI, HIPPA itd.
- Możliwość przechwytywania wielu typów błędów. Ponieważ trzeba wykonać polecenie terraform apply i zweryfikować prawdziwy, działający serwer, tego rodzaju testy są w stanie wychwycić znacznie więcej typów błędów niż analiza statyczna i testowanie planu.

Wady narzędzi testowania serwera:

- Wydajność działania. Te testy działają jedynie we wdrożonych serwerach, konieczne jest więc wykonanie pełnego cyklu poleceń terraform apply (i prawdopodobnie także terraform destroy), co może zająć sporo czasu.
- Mniejsza stabilność (pewna liczba wariujących testów). Skoro trzeba wykonać polecenie terraform apply i zaczekać na rzeczywiste wdrożenie serwerów, można się spodziewać różnych problemów i czasami wariujących testów.
- Konieczność uwierzytelniania u prawdziwego dostawcy (np. w rzeczywistym koncie AWS). To jest wymagane do działania polecenia terraform apply, aby można było wdrożyć serwery. Ponadto te narzędzia testowania wymagają dodatkowych metod uwierzytelniania — np. SSH — w celu nawiązania połączenia z testowanymi serwerami.
- Konieczność wdrożenia i późniejszego usunięcia rzeczywistych zasobów. To wymaga czasu i wiąże się z kosztami.
- Dokładne sprawdzenie jedynie sposobu działania serwerów, a nie pozostałych komponentów infrastruktury.
- Testy nie sprawdzają funkcjonalności. Istnieje więc niebezpieczeństwo, że pomimo zaliczenia wszystkich testów infrastruktura nie będzie działała.

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.

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.

Należy regularnie przeprowadzać porządkowanie środowisk

W przeciwnym razie środowiska staną się niemożliwe do zarządzania, a koszty związane z ich utrzymaniem wymkną się spod kontroli.

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 8. i warto go tutaj powtórzyć: małe moduły są łatwiejsze do tworzenia i obsługi, a także w użyciu oraz w testowaniu.

W rozdziale przedstawiłem wiele różnych podejść w zakresie testowania: testy jednostkowe, testy integracji, testy typu E2E, analiza statyczna itd. W tabeli 9.4 znalazło się krótkie podsumowanie poszczególnych typów testów.

Tabela 9.4. Porównanie podejść w zakresie testowania (im więcej czarnych kwadratów, tym lepiej)

	Analiza statyczna	Testowanie planu	Testowanie serwera	Testy jednostkowe	Testy integracji	Testy typu E2E
Wydajność działania	■■■■■	■■■■□	■■■□□	■■□□□	■□□□□	□□□□□
Koszt działania	■■■■■	■■■■□	■■■□□	■■□□□	■□□□□	□□□□□
Stabilność i niezawodność	■■■■■	■■■■□	■■■□□	■■□□□	■□□□□	□□□□□
Łatwość użycia	■■■■■	■■■■□	■■■□□	■■□□□	■□□□□	□□□□□
Sprawdzanie składni	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■
Sprawdzanie polityki	■■□□□	■■■■□	■■■■□	■■■■■	■■■■■	■■■■■
Sprawdzanie działania serwera	□□□□□	□□□□□	■■■■■	■■■■■	■■■■■	■■■■■
Sprawdzanie działania innych komponentów infrastruktury	□□□□□	□□□□□	■■□□□	■■■■□	■■■■■	■■■■■
Sprawdzanie współdziałania całej infrastruktury	□□□□□	□□□□□	□□□□□	■□□□□	■■■□□	■■■■■

Które z tych podejść należy stosować? Odpowiedź brzmi: połączyć je wszystkie! Poszczególne typy mają pewne wady i zalety, konieczne jest więc połączenie różnych typów testów, aby mieć pewność, że kod działa zgodnie z oczekiwaniami. To oczywiście nie oznacza, że poszczególne typy będą używane w równych proporcjach. Przypomnij sobie piramidę testów i to, że ogólnie chcesz mieć więcej testów jednostkowych, mniej testów integracji i tylko niewielką liczbę wysokiej wartości testów typu E2E. Co więcej, nie musisz wszystkich rodzajów testów dołączać jednocześnie. Zamiast tego wybierz te najbardziej wartościowe w danej chwili i rozpocznij od ich zdefiniowania. Praktycznie każdy rodzaj testowania jest lepszy niż brak testów. Dlatego też, jeśli na początek możesz pozwolić sobie jedynie na analizę statyczną, wykorzystaj ją jako punkt wyjścia i później stopniowo rozbudowuj testy.

Z rozdziału 10. 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 podkreślić to, że infrastruktura będzie łatwiejsza w obsłudze. Zamiast informować o ogromnej popularności Terraform, należy wspomnieć o możliwości łatwego wykorzystania wielu istniejących modułów i wtyczek pozwalających na szybsze wykonywanie zadań. 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 jakkolwiek 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 fałszywego stopniowania otrzymujesz najgorszy z możliwych wyników: ponosisz ogromny koszt i nie dostajesz absolutnie nic w zamian.

¹ Przygotowany przez The Standish Group raport *CHAOS Manifesto 2013: Think Big, Act Small*, 2013. Znajdziesz go pod adresem https://www.standishgroup.com/sample_research_files/CM2013-8+9.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>).

Dlatego też stopniowanie jest ważne. Każdy etap projektu powinien dostarczać pewną wartość, nawet jeśli nie zostanie on dokończony. Niezależnie od liczby wykonanych kroków zawsze otrzymasz 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 tworzysz, 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. 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.

4. W wyniku ręcznie wprowadzonej zmiany kod Terraform już dłużej nie odpowiada temu, co zostało 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.
5. 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 poznajesz 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 ponad 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/10-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
```

Możesz uaktualnić i ponownie uruchomić testy zautomatyzowane. Idea polega na zoptymalizowaniu sposobu pracy i maksymalnym skróceniu czasu między wprowadzeniem zmiany i sprawdzeniem, czy działa ona zgodnie z oczekiwaniami.

W trakcie pracy powinienś regularnie przekazywać kod do repozytorium wraz z jasnymi komunikatami dotyczącymi wprowadzonych zmian.

```
$ git commit -m "Uaktualniony komunikat Witaj, świecie"
```

Przekazanie zmian do zatwierdzenia

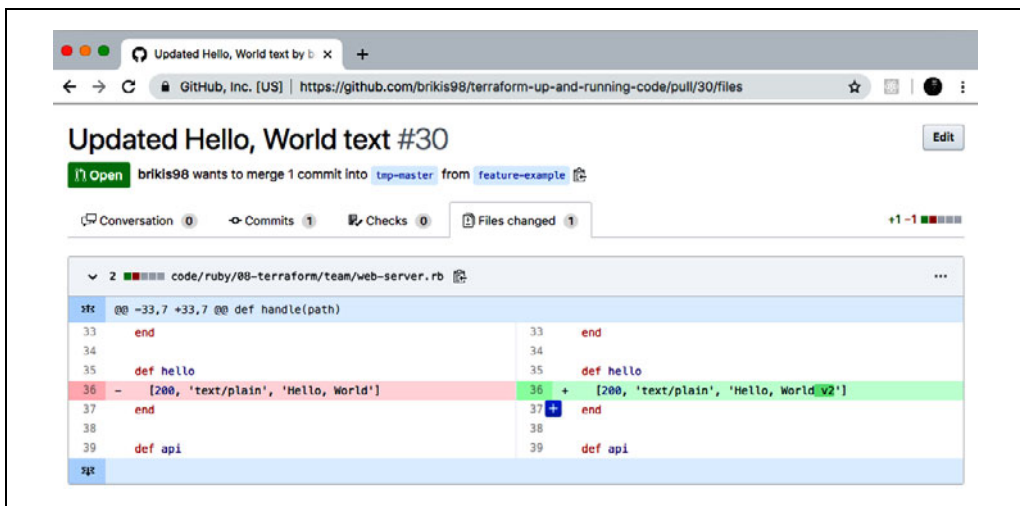
Ostatecznie kod i testy będą działały w oczekiwany sposób, więc nadejdzie pora na zatwierdzenie zmian i przejrzanie kodu. Do tego celu można wykorzystać oddzielne narzędzie (np. Phabricator lub ReviewBoard) lub — jeśli używasz serwisu GitHub — masz możliwość utworzenia tzw. **żądania aktualizacji** (ang. *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 10.1.



Rysunek 10.1. Członkowie zespołu mogą przejrzeć Twoje zmiany w kodzie w żądaniu aktualizacji w GitHub

Uruchomienie testów zautomatyzowanych

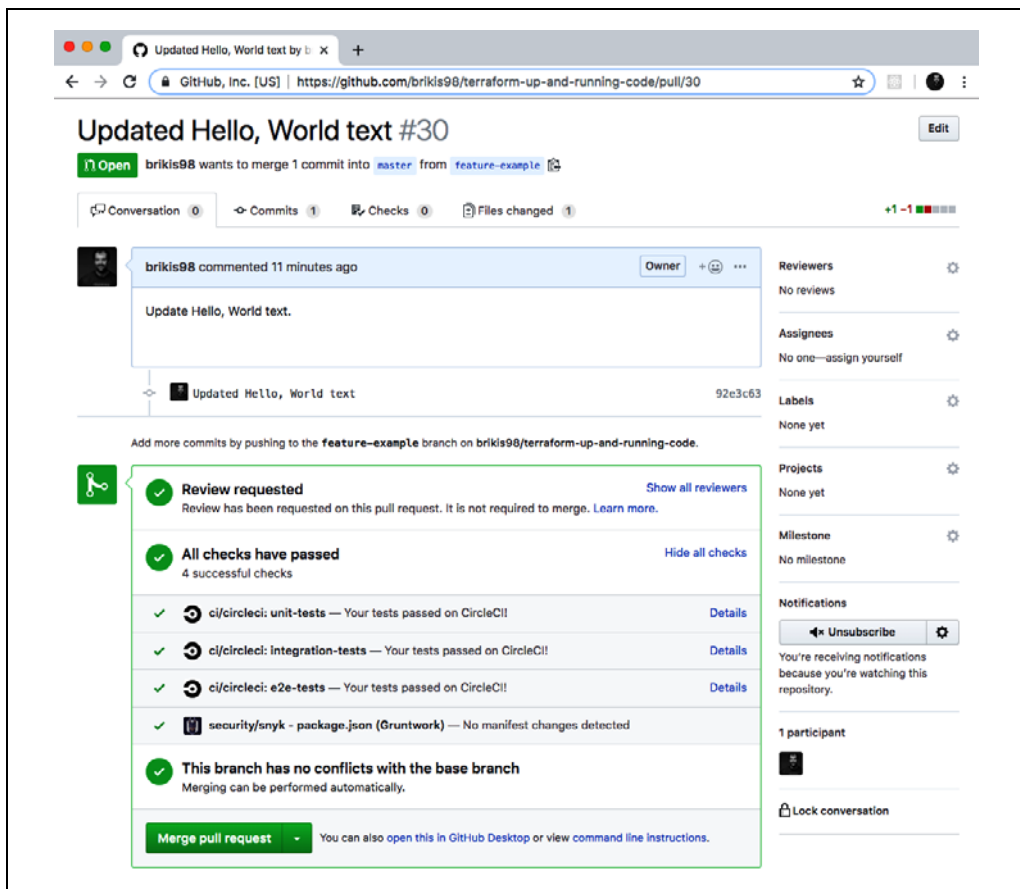
Powinieneś przygotować zaczepy zatwierdzenia pozwalające na wykonywanie testów zautomatyzowanych po każdej operacji zatwierdzenia kodu w systemie kontroli wersji. Najczęściej stosowane rozwiązanie polega na użyciu serwera **ciągłej integracji** (ang. *continuous integration*, CI), takiego jak Jenkins, CircleCI lub GitHub Actions. 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 10.2.

Na podstawie rysunku 10.2 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 `main`.

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



Rysunek 10.2. Żądanie aktualizacji w serwisie GitHub pokazuje wyniki testów zautomatyzowanych z CircleCI

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 briks98/ruby-web-server:$commit_id .
```

Te polecenia powodują utworzenie nowego obrazu Dockera o nazwie `briks98/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), Amazon ECS, HashiCorp Nomad i Apache Mesos. W rozdziale 7. pokazałem przykład użycia Kubernetes do wdrożenia kontenera Dockera.

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 implementujące te wymagania.

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 któregośkolwiek 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, zaczną 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 8.: 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 okazjnymi 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**:

Gałąź *main* w repozytorium *live* powinna w 100% odzwierciedlać to, co zostało faktycznie wdrożone w środowisku produkcyjnym.

³ Przygotowana przez Gruntwork biblioteka Infrastructure as Code Library (<https://gruntwork.io/infrastructure-as-code-library/>).

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 użytkownika 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 *main*”

Do zrozumienia, co naprawdę zostało wdrożone w środowisku produkcyjnym, powinno wystarczyć przeglądanie tylko jednej gałęzi. Zwykle tą gałęzią jest *main*. To oznacza, że wszystkie zmiany wpływające na środowisko produkcyjne powinny trafiać bezpośrednio do gałęzi *main* (wprawdzie można tworzyć oddzielne gałęzie, ale tylko w celu przygotowania żądań aktualizacji przeznaczonych do połączenia z gałęzią *main*). Polecenie `terraform apply` dla środowiska produkcyjnego powinno być wydawane tylko z poziomu gałęzi *main*. W następnym podpunkcie wyjaśnię, dlaczego należy stosować takie podejście.

Problemy z gałęziami

W rozdziale 3. zobaczyłeś, że możesz 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       = data.aws_ami.ubuntu.id
  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       = data.aws_ami.ubuntu.id
  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-0fb653ca2d3203ac1"
    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       = data.aws_ami.ubuntu.id
  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 Bartek 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-0fb653ca2d3203ac1"
    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 9., 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
```

Ewentualnie 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 9. 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.

Podczas wprowadzania zmian należy pamiętać o regularnym przekazywaniu (zatwierdzaniu) kodu do repozytorium.

```
$ git commit -m "Aktualniony komunikat Witaj, świecie"
```

Przekazanie zmian do zatwierdzenia

Gdy kod działa zgodnie z oczekiwaniami, można utworzyć żądanie aktualizacji, aby wprowadzone przez Ciebie zmiany zostały przejrane 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. Wprawdzie 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.

Przykładowe fragmenty kodu

Jak już wspomniałem w rozdziale 8., 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 9. 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).

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>.

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 liczy się tylko spójne stosowanie konwencji w całej bazie 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 9. 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 10.3).

Terraform Cloud i Terraform Enterprise, czyli płatne narzędzia oferowane przez HashiCorp, również obsługują automatycznie wykonywanie polecenia `terraform plan` w trakcie żądania aktualizacji.

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 `main` 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
```



Rysunek 10.3. Atlantis może automatycznie dodać dane wyjściowe polecenia terraform plan jako komentarz w żądaniu aktualizacji

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,
- strategie 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 ciągłe 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 ciągłe 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).

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. Terragrunt ma wbudowaną funkcjonalność automatycznego ponawiania próby w przypadku wystąpienia znanego błędu (<https://terragrunt.gruntwork.io/docs/features/auto-retry/>).

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 <IDENTYFIKATOR_BLOKADY>
```

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, GitHub Actions, Terraform Cloud, 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. I w tym tkwi problem.

Problem polega na tym, że serwery CI są (a) trudne do zabezpieczenia⁵, (b) dostępne dla wszystkich programistów w firmie i (c) używane do wykonywania dowolnego kodu. Dlatego też nadanie serwerowi CI trwałych uprawnień administratora może być ryzykowne. W praktyce oznacza to, że każdy członek zespołu otrzymuje uprawnienia administratora i może zmienić serwer CI w niezwykle cenny cel ataku.

Istnieje kilka działań, które można podjąć, aby pomóc w minimalizacji tego ryzyka:

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.).

Nie udostępniaj serwera CI w publicznym internecie

Serwer CI powinien działać w prywatnej podsięci, bez żadnego publicznego adresu IP, aby pozostał dostępny jedynie poprzez połączenie VPN. W ten sposób dostęp do niego będą mieli jedynie użytkownicy z poprawnym dostępem sieciowym (np. za pomocą certyfikatu VPN). Takie podejście jednocześnie wiąże się z pewnym kosztem: zaczepy 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. To stosunkowo niewielki koszt znacznego podniesienia poziomu bezpieczeństwa serwera CI.

Wymuszaj zatwierdzanie wdrożeń

Skonfiguruj potok CI/CD w taki sposób, aby każde wdrożenie musiało być zatwierdzane przez co najmniej jedną osobę (inną niż żądająca przeprowadzenia operacji danego wdrożenia). W trakcie zatwierdzania osoba przeglądająca kod powinna mieć możliwość zobaczenia proponowanych zmian w kodzie i danych wyjściowych polecenia `terraform plan`. To będzie ostatnie sprawdzenie przed zgodą na wykonanie polecenia `terraform apply`. Dzięki temu masz pewność, że każde wdrożenie, zmiana kodu i dane wyjściowe polecenia `terraform plan` zostały przeanalizowane przez co najmniej dwie osoby.

⁵ Zapoznaj się z 10 rzeczywistymi historiami dotyczącymi uszkodzenia potoków CI/CD (<https://research.nccgroup.com/2022/01/13/10-real-world-stories-of-how-weve-compromised-ci-cd-pipelines/>).

Rozważ rezygnację z nadawania serwerowi CI trwałych uprawnień administratora

Jak wyjaśniłem w rozdziale 6., zamiast ręcznego zarządzania trwałymi danymi uwierzytelniającymi (np. klucze dostępu AWS kopiowane i wklejane w serwerze CI) należy preferować mechanizmy uwierzytelniania, które wykorzystują tymczasowe dane uwierzytelniające, np. rolę IAM i OIDC.

Nie nadawaj serwerowi CI trwałych uprawnień administratora

Zamiast tego odizoluj uprawnienia administratora w całkowicie oddzielnym i odizolowanym *serwerze roboczym*, np. kontenerze. Taki serwer roboczy powinien być bardzo silnie zabezpieczony, aby żaden programista w ogóle nie miał do niego dostępu. Jedynie serwer CI może korzystać z serwera roboczego i to za pomocą wyjątkowo ograniczonego API. Przykładowo wspomniane API może pozwalać na wykonywanie tylko ściśle określonych poleceń (np. `terraform plan` lub `terraform apply`) w ściśle określonych repozytoriach (np. *live*) w konkretnych gałęziach (np. *main*) itd. Dzięki temu, nawet jeśli atakującemu uda się uzyskać dostęp do serwera CI, nadal nie będzie miał dostępu do administracyjnych danych uwierzytelniających. Będzie mógł jedynie żądać wdrożenia pewnego kodu znajdującego się w systemie kontroli wersji, co niekoniecznie będzie miało aż tak katastrofalne skutki jak pełny wyciek administracyjnych danych uwierzytelniających⁶.

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 produkcyjnego⁷. 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 przedprodukcyjnym 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 zatwierdzenia wdrożenia, o czym wspomniałem w poprzednim punkcie, co oznacza wykonanie polecenia `terraform plan` i przejrzanie

⁶ Zapoznaj się z frameworkiem Gruntwork Pipelines (<https://gruntwork.io/pipelines>), aby poznać rzeczywisty przykład wzorca takiego serwera roboczego.

⁷ 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>.

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.

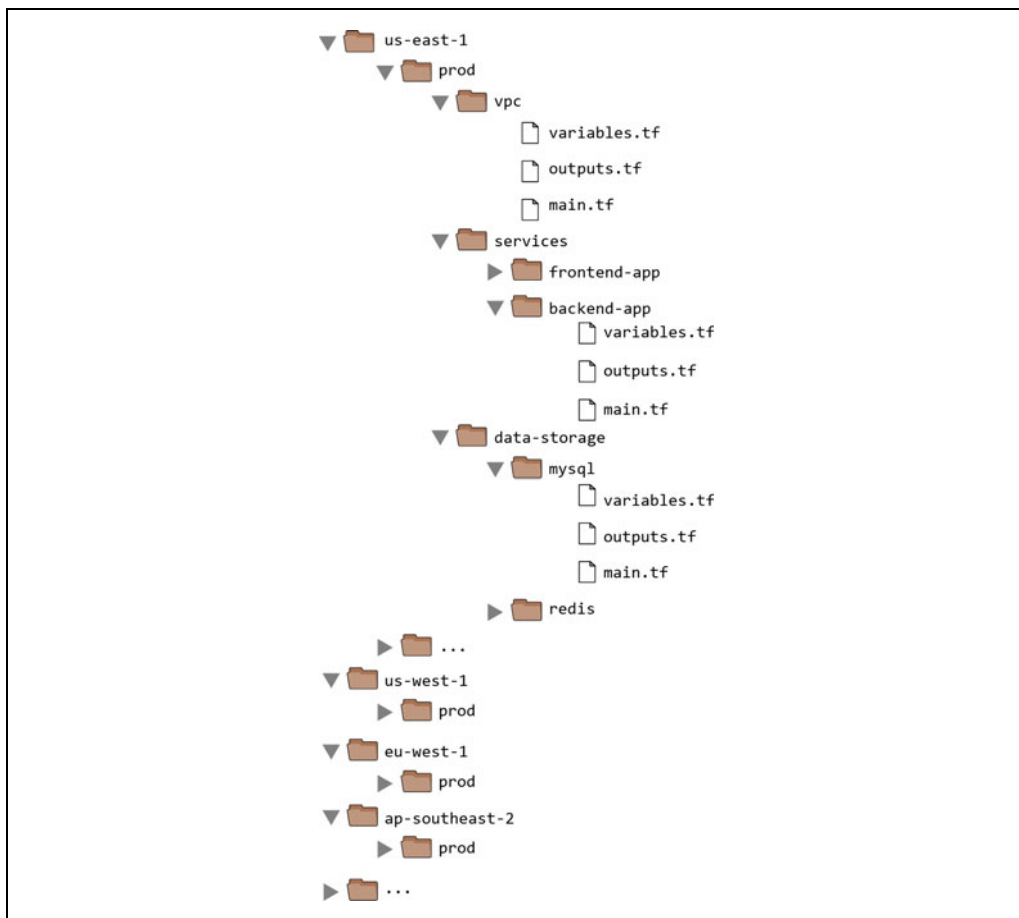
Ważna kwestia do rozważenia wiąże się z obsługą powtarzającego się kodu między środowiskami w repozytorium *live*. Dla przykładu przeanalizuj repozytorium *live* pokazane na rysunku 10.4.

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.

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, o którym wspomniałem już wcześniej w książce. Terragrunt to lekkie opakowanie dla Terraform, po którego zainstalowaniu będziesz mógł wydawać wszystkie standardowe polecenia `terraform`, przy czym jako pliku binarnego użyjesz *terragrunt*:

```
$ terragrunt plan
$ terragrunt apply
$ terragrunt output
```



Rysunek 10.4. Układ plików z ogromną liczbą skopiowanych i wklejonych środowisk oraz modułów w poszczególnych środowiskach

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 10.5.

Pracę rozpocznij od zainstalowania Terragrunt z wykorzystaniem informacji zamieszczonych w witrynie internetowej Terragrunt (<https://terragrunt.gruntwork.io/docs/getting-started/install/>). Następnie do plików *modules/data-stores/mysql/main.tf* i *modules/services/hello-world-app/main.tf* dodaj blok konfiguracji provider.

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



Rysunek 10.5. Użycie Terragrunt w repozytorium live pozwala na zmniejszenie liczby powielanych plików

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 *terraformunt.hcl* dla każdego modułu. Dla przykładu spójrz na zawartość wymienionego pliku znajdującego się pod adresem *live/stage/data-stores/mysql/terraformunt.hcl*:

```
terraform {
  source = "github.com/<WŁAŚCICIEL>/modules//data-stores/mysql?ref=v0.0.7"
}

inputs = {
  db_name      = "example_stage"

  # Zdefiniowanie nazwy użytkownika za pomocą zmiennej środowiskowej TF_VAR_db_username.
  # 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"

  generate = {
    path      = "backend.tf"
    if_exists = "overwrite"
  }

  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` Terragrunt może dynamicznie wygenerować konfigurację backend dla każdego modułu i zapisać ją w pliku wskazanym przez parametr `generate`. Zwróć uwagę, że wartość `key` w bloku `config` używa 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"

  # Zdefiniowanie nazwy użytkownika za pomocą zmiennej środowiskowej TF_VAR_db_username.
  # 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
DEBU[0001] Reading Terragrunt config file at terragrunt.hcl
DEBU[0001] Included config live/stage/terragrunt.hcl
DEBU[0001] Downloading Terraform configurations into .terragrunt-cache
DEBU[0001] Generated file backend.tf
DEBU[0013] Running command: terraform init

(...)

Initializing the backend...

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

(...)

DEBU[0024] Running command: terraform apply

(...)

Terraform will perform the following actions:

(...)

Plan: 5 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: yes

(...)

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

Normalnie Terragrunt wyświetla tylko dane wyjściowe Terraform, ponieważ jednak w omawianym przykładzie została użyta opcja `--terragrunt-log-level debug`, wygenerowane dane wyjściowe pokazują również działania Terragrunt podejmowane w tle.

1. Odczyt pliku *terragrunt.hcl* w katalogu *mysql*, w którym zostało wykonane polecenie `terraform apply`.
2. Pobranie wszystkich ustawień z dołączonego pliku *terragrunt.hcl*.
3. Pobranie kodu Terraform znajdującego się pod adresem URL podanym w *source* i umieszczenie go w katalogu *.terragrunt-cache*.
4. Wygenerowanie pliku *backend.tf* razem z blokiem konfiguracyjnym *backend*.
5. Wykrycie, że polecenie `terraform init` nie zostało wykonane, i automatyczne jego wykonanie. (Terragrunt automatycznie utworzy nawet kubelek S3 i tabele DynamoDB, jeśli jeszcze nie istnieją).
6. Wykonanie polecenia `terraform apply` w celu wdrożenia zmian.

Całkiem niezłe jak na kilka niewielkich plików *terragrunt.hcl*!

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()
}

dependency "mysql" {
  config_path = "../..//data-stores/mysql"
}

inputs = {
  environment = "stage"
  ami         = "ami-0fb653ca2d3203ac1"

  min_size = 2
  max_size = 2

  enable_autoscaling = false

  mysql_config = dependency.mysql.outputs
}
```

Ten plik *terragrunt.hcl* używa adresu URL zdefiniowanego w *source* i *inputs*, podobnie jak pokazałem wcześniej. 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*. Jedyną nowością w tym pliku *terragrunt.hcl* jest blok *dependency*:

```
dependency "mysql" {
  config_path = ".././data-stores/mysql"
}
```

To jest funkcjonalność Terragrunt, która może być używana do automatycznego odczytywania zmiennych danych wyjściowych innego modułu Terragrunt. Dlatego też można je przekazać jako zmienne danych wejściowych dla bieżącego modułu, co pokazałem w kolejnym poleceniu:

```
mysql_config = dependency.mysql.outputs
```

Innymi słowy, blok `dependency` to alternatywa dla używania źródeł danych `terraform_remote_state` do przekazywania danych między modułami. Wprawdzie `terraform_remote_state` ma zaletę w postaci rozwiązania natywnego dla Terraform, ale wadą jest ściślejsze powiązanie modułów ze sobą, ponieważ każdy z nich musi mieć informacje o stanie innych modułów. Użycie bloku `dependency` pozwala modułom na udostępnianie ogólnych danych wejściowych typu `mysql_config` i `vpc_id`, zamiast korzystać ze źródeł danych, co oznacza mniejsze powiązanie modułów ze sobą, a tym samym ich łatwiejsze testowanie i wielokrotne używanie.

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 10.1 zestawilem obok siebie kroki przeprowadzane w obu tych procesach.

Tabela 10.1. Sposoby pracy z kodem aplikacji i infrastruktury

	Kod aplikacji	Kod infrastruktury
Użyj systemu kontroli wersji	<ul style="list-style-type: none"> ✓ polecenie <code>git clone</code> ✓ tylko jedno repozytorium dla aplikacji ✓ użycie gałęzi 	<ul style="list-style-type: none"> ✓ polecenie <code>git clone</code> ✓ repozytoria <i>live</i> i <i>modules</i> ✓ gałęzie nie są używane
Lokalne uruchomienie kodu	<ul style="list-style-type: none"> ✓ uruchomienie kodu w komputerze lokalnym ✓ polecenie ruby <code>web-server.rb</code> ✓ polecenie ruby <code>web-server-test.rb</code> 	<ul style="list-style-type: none"> ✓ uruchomienie kodu w odizolowanym środowisku ✓ polecenie <code>terraform apply</code> ✓ polecenie <code>go test</code>
Wprowadzanie zmian w kodzie	<ul style="list-style-type: none"> ✓ zmiana kodu ✓ polecenie ruby <code>web-server.rb</code> ✓ polecenie ruby <code>web-server-test.rb</code> 	<ul style="list-style-type: none"> ✓ zmiana kodu ✓ polecenie <code>terraform apply</code> ✓ polecenie <code>go test</code> ✓ użycie etapów wykonywania testów
Przekazanie zmian do zatwierdzenia	<ul style="list-style-type: none"> ✓ wysłanie żądania aktualizacji ✓ wymuszenie zastosowania określonych reguł tworzenia kodu 	<ul style="list-style-type: none"> ✓ wysłanie żądania aktualizacji ✓ wymuszenie zastosowania określonych reguł tworzenia kodu

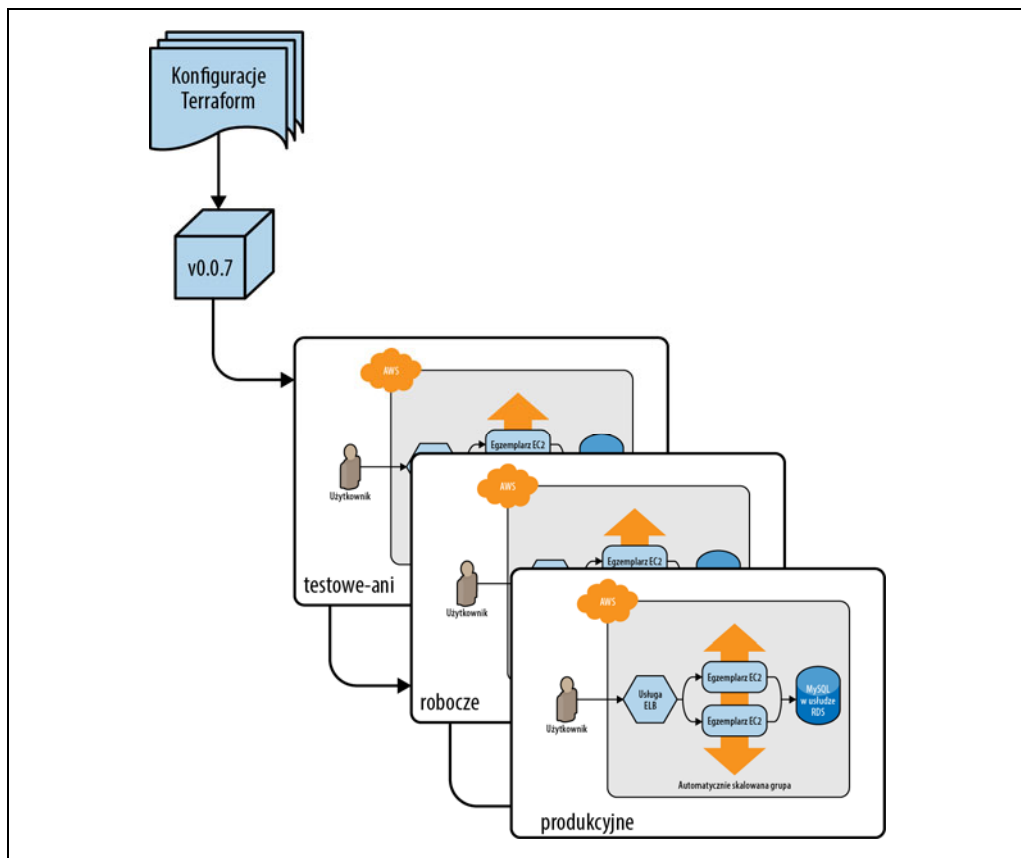
Tabela 10.1. Sposoby pracy z kodem aplikacji i infrastruktury (ciąg dalszy)

	Kod aplikacji	Kod infrastruktury
Uruchomienie testów zautomatyzowanych	<ul style="list-style-type: none"> ✓ uruchomienie testów w serwerze CI ✓ testy jednostkowe ✓ testy integracji ✓ testy typu E2E ✓ analiza statyczna 	<ul style="list-style-type: none"> ✓ 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	<ul style="list-style-type: none"> ✓ polecenie <code>git tag</code> ✓ utworzenie wersjonowanego, niemodyfikowalnego produktu 	<ul style="list-style-type: none"> ✓ polecenie <code>git tag</code> ✓ użycie repozytorium wraz z tagiem jako wersjonowanego, niemodyfikowalnego produktu
Wdrożenie	<ul style="list-style-type: none"> ✓ wdrożenie za pomocą Terraform, narzędzi instrumentacji (np. Kubernetes, Mesos), skryptów ✓ wiele strategii wdrożenia: ciągłe, typu niebieski-zielony, kanarkowe ✓ uruchomienie wdrożenia z poziomu serwera CI ✓ nadanie serwerowi CI ograniczonych uprawnień ✓ promocja niemodyfikowalnego, wersjonowanego produktu między poszczególnymi środowiskami ✓ automatyczne wdrożenie po akceptacji żądania aktualizacji 	<ul style="list-style-type: none"> ✓ wdrożenie za pomocą Terraform, Atlantis, Terraform Cloud, Terraform Enterprise, Terragrunt, skryptów ✓ ograniczona liczba strategii wdrożenia: upewnij się co do obsługi błędów za pomocą np. wielu prób i plik <code>errored.tfstate</code> ✓ uruchomienie wdrożenia z poziomu serwera CI ✓ nadanie serwerowi CI tymczasowych uprawnień administratora jedynie w celu wywołania odizolowanego i zabezpieczonego serwera roboczego, który ma uprawnienia administratora ✓ promocja niemodyfikowalnego, wersjonowanego produktu między poszczególnymi środowiskami ✓ przeprowadzenie — po akceptacji żądania aktualizacji — procesu zatwierdzenia wdrożenia, w trakcie którego ktoś sprawdza dane wyjściowe polecenia <code>terraform plan</code>, a następnie odbywa się automatyczne wdrożenie

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 10.6.

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 zarządzać danymi poufnymi, jak pracować z wieloma regionami, kontami i chmurami, jak tworzyć kod Terraform



Rysunek 10.6. Promowanie między środowiskami niemodyfikowalnego, wersjonowanego produktu utworzonego na podstawie kodu 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, menedżerem danych poufnych AWS, klastrami Kubernetes, testami zautomatyzowanymi itd. Uff! 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 przejrzenia. Jeżeli masz jakiegokolwiek uwagi lub pytania, napisz do mnie na adres jim@ybrikman.com. Dziękuję, że przeczytałeś tę książkę!

Polecane zasoby

W dodatku wymieniłem wybrane z najlepszych zasobów w postaci książek, blogów, newsletterów i prelekcji dotyczących DevOps i podejścia infrastruktury jako kodu.

Książki

- Kief Morris, *Infrastruktura jako kod. Dynamiczne systemy w epoce chmury* (Helion)
- 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, *Ciągłe dostarczanie oprogramowania. Automatyzacja kompilacji, testowania i wdrażania* (Helion)
- Michael T. Nygard, *Release It! Design and Deploy Production-Ready Software* (The Pragmatic Bookshelf)
- Marko Lukša, *Kubernetes w akcji* (PWN)
- 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* (Helion)
- 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/>)
- 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, *Automated Testing for Terraform, Docker, Packer, Kubernetes, and More* (<https://www.infoq.com/presentations/automated-testing-terraform-docker-packer/>)
- 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)
- Glenn Vanderburg, *Real Software Engineering* (<https://www.youtube.com/watch?v=NP9AIUT9nos>)

Newslettery

- Devops Weekly (<https://www.devopsweekly.com/>)
- Newsletter Gruntwork (<https://www.gruntwork.io/newsletter/>)
- Newsletter Terraform: Up & Running (<https://www.terraformupandrunning.com/>)
- Terraform Weekly Newsletter (<https://weekly.tf/>)

Fora internetowe

- Grupa Terraform w witrynie internetowej HashiCorp (<https://discuss.hashicorp.com/c/terraform-core/27>)
- Grupa Terraform w Reddit (<https://www.reddit.com/r/Terraform/>)
- 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, stworzonym 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.

Ilustracja na okładce, autorstwa Karen Montgomery, powstała na podstawie czarno-białej ryciny pochodzącej 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 