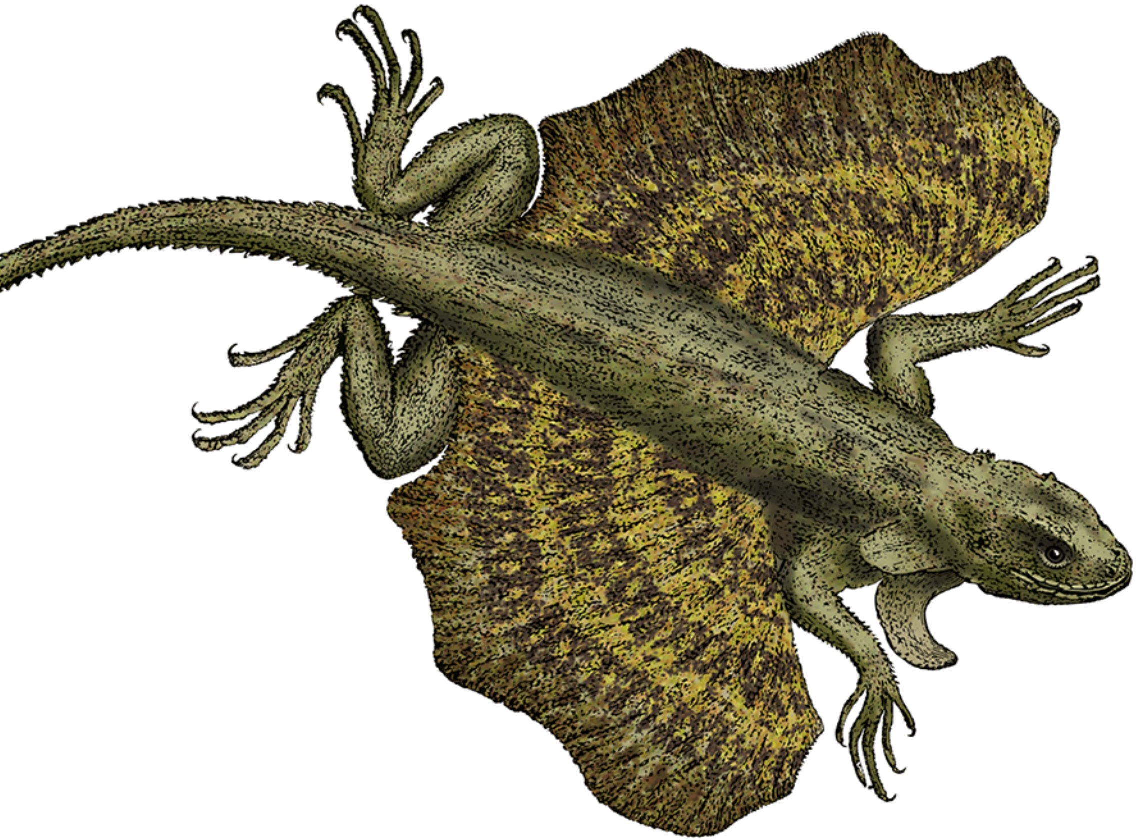
O'REILLY®

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: 978-83-8322-346-9

© 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 jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

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

Autor oraz 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
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)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

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

Kup ksi k Pole ksi k

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 wyrażeń 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

Kup książkę Poleć książkę

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

Kup książkę Poleć książkę

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

Kup ksi k Pole ksi k

ROZDZIAŁ 1.
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

Kup ksi k Pole ksi k

(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, koszmarnych 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.

Kup ksi k Pole ksi k

Wyniki firm, które zdecydowały się na zastosowanie podejścia DevOps, są zdumiewające. Przykładowo firma Nordstorm przekonała się, że zastosowanie praktyk DevOps w organizacji pozwoliło na zwiększenie o 100% liczby funkcji dostarczanych każdego miesiąca, skrócenie liczby usterek o połowę, skrócenie o 60% *czasu realizacji* (ang. *lead time*) — w tym kontekście to opóźnienie między pojawieniem się pomysłu i uruchomienie kodu w środowisku produkcyjnym — oraz zmniejszenie liczby incydentów produkcyjnych o 60 – 90%. Gdy dział LaserJet Firmware w HP zaczął stosować praktyki DevOps, czas poświęcany przez programistów na tworzenie kodu wzrósł z 5% do 40%, a ogólny koszt prac programistycznych spadł o 40%. Z kolei firma Etsy wykorzystała praktyki DevOps w celu przejścia od stresujących i rzadkich wdrożeń powodujących przestoje i awarie do wielokrotnych wdrożeń w ciągu dnia (od 25 do 50) wraz ze znacznie niższą liczbą przestojów[1].

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.

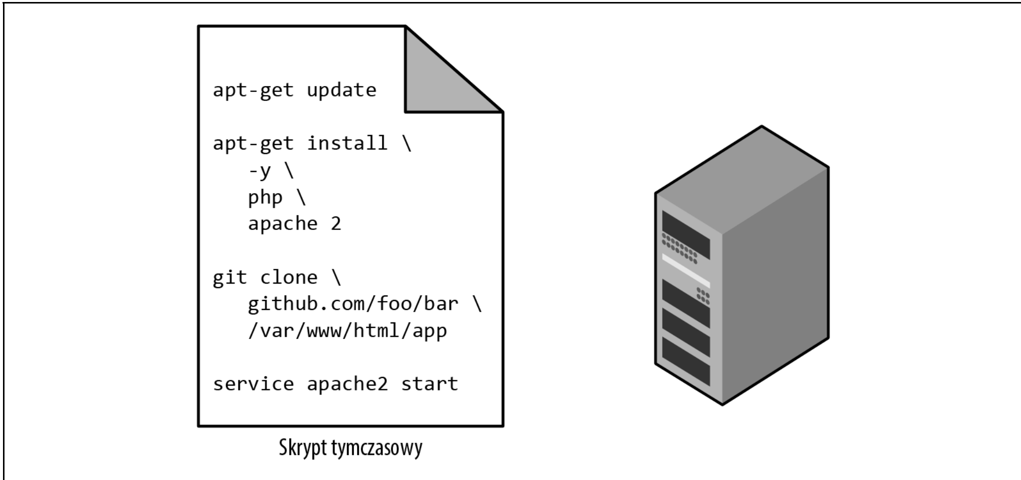
Dalej po kolei przedstawię te kategorie.

[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.

Kup książkę Poleć książkę

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

Kup ksi k Pole ksi k

wykonuje pewne zadania. Żadna z wymienionych kwestii nie stanowi poważnego problemu w ośmiowierszowym skrypcie instalującym serwer Apache, ale sytuacja szybko wymknie się spod kontroli, gdy skrypty tymczasowe będą używane do zarządzania dziesiątkami serwerów, baz danych, mechanizmów równoważenia obciążenia, konfiguracji sieciowych itd.

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

Narzędzia zarządzania konfiguracją

Chef, Puppet 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 wymieniłem 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

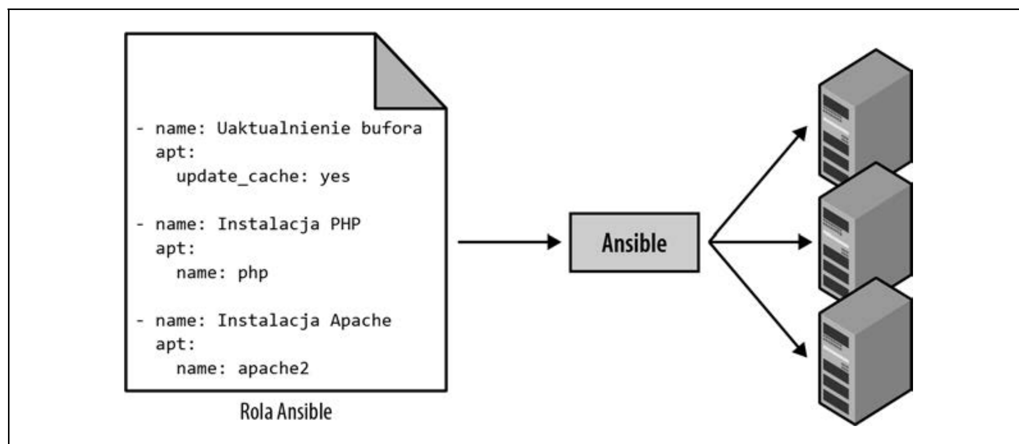
Kup ksi k Pole ksi k

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 **kodem 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
```


Kup ksi k Pole ksi k

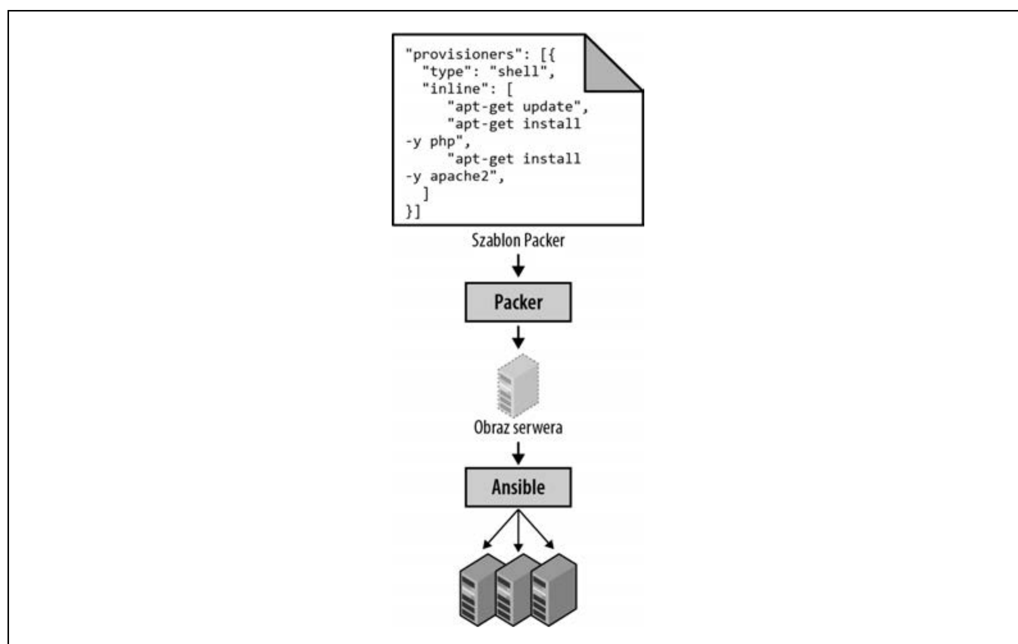
Na końcu można za pomocą przedstawionego polecenia wykonać zdefiniowany kod scenariusza (ang. *playbook*):

```
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

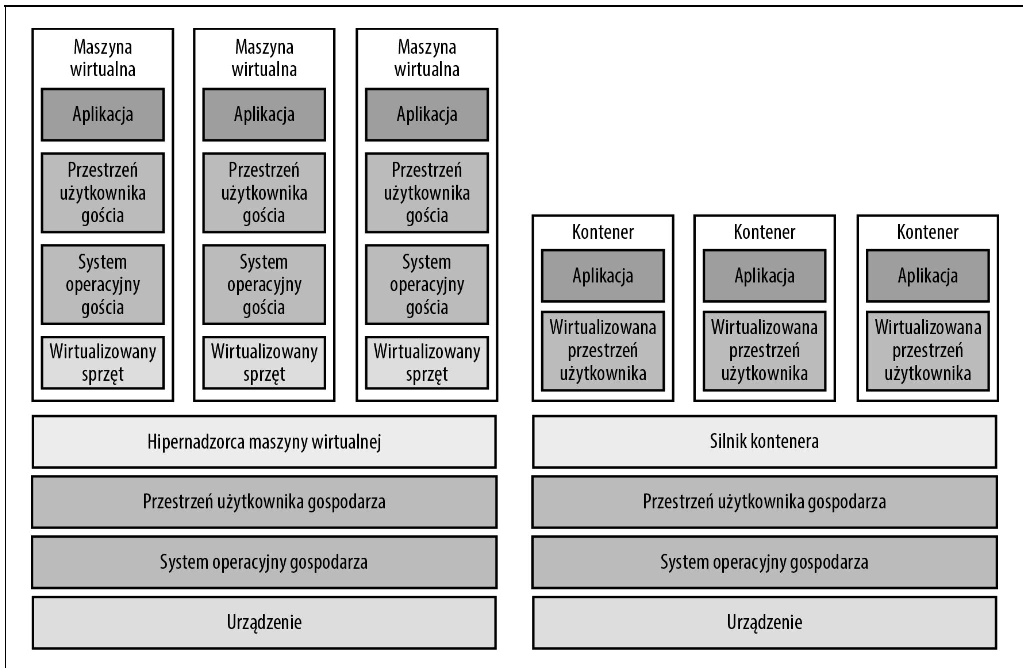
Zyskującym ostatnio popularność rozwiązaniem alternatywnym dla zarządzania konfiguracją jest wykorzystanie **narzędzi szablonów serwera**, takich jak Docker, Packer i Vagrant. Zamiast na uruchamianiu ogromnej liczby serwerów i konfigurowaniu ich przez wykonywanie tego samego kodu w każdym z nich idea 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

Kup książkę Poleć książkę

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. **hipernadzorcy** (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.

Kup ksi k Pole ksi k

Kontenery

Kontener emuluje przestrzeń użytkownika systemu operacyjnego[2]. 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 wirtualne[3]. 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 podejrzane działania, lepiej jest skorzystać z oferowanych przez maszyny wirtualne zalet większej izolacji.

Kup ksi k Pole ksi k

```
      "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. Jedyna różnica między tym i poprzednim przykładem polega na tym, że szablon Packer nie uruchamia serwera WWW Apache (za pomocą wywołania `sudo service apache2 start`). To wynika z faktu stosowania szablonów zwykle do instalowania oprogramowania w obrazach, więc to oprogramowanie powinno działać właściwie tylko po uruchomieniu danego obrazu (np. przez wdrożenie go w serwerze), przeznaczonego 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.

Kup ksi k Pole ksi k

Narzędzia instrumentacji

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

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

Obsługa tych zadań jest domeną **narzędzi instrumentacji**, takich jak Kubernetes, Marathon/Mesos, Amazon Elastic Container Service (Amazon ECS), Docker Swarm i Nomad. Przykładowo Kubernetes pozwala na zdefiniowanie sposobu zarządzania kontenerami Dockera jako kodem. Najpierw wdrażasz *klaster Kubernetes*, czyli grupę serwerów zarządzanych przez Kubernetes, a następnie używasz jej do uruchamiania kontenerów Dockera. Większość dostawców chmury oferuje natywną obsługę w zakresie wdrażania zarządzanych klastrów Kubernetes, np. Amazon Elastic Container Service for Kubernetes (Amazon EKS), Google Kubernetes Engine (GKE) i Azure Kubernetes Service (AKS).

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

```
apiVersion: apps/v1

# Użycie obiektu Deployment do wdrożenia wielu replik kontenerów
# Dockera oraz do deklaratywnego przekazywania im uaktualnień.
kind: Deployment

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

# Specyfikacja konfigurująca dany obiekt Deployment.
spec:
  # Określenie sposobu wyszukiwania kontenerów przez ten obiekt Deployment.
  selector:
    matchLabels:
      app: example-app

  # Nakazanie obiektowi Deployment uruchomienie
  # trzech replik kontenerów Dockera.
```


Kup ksi k Pole ksi k

```
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:

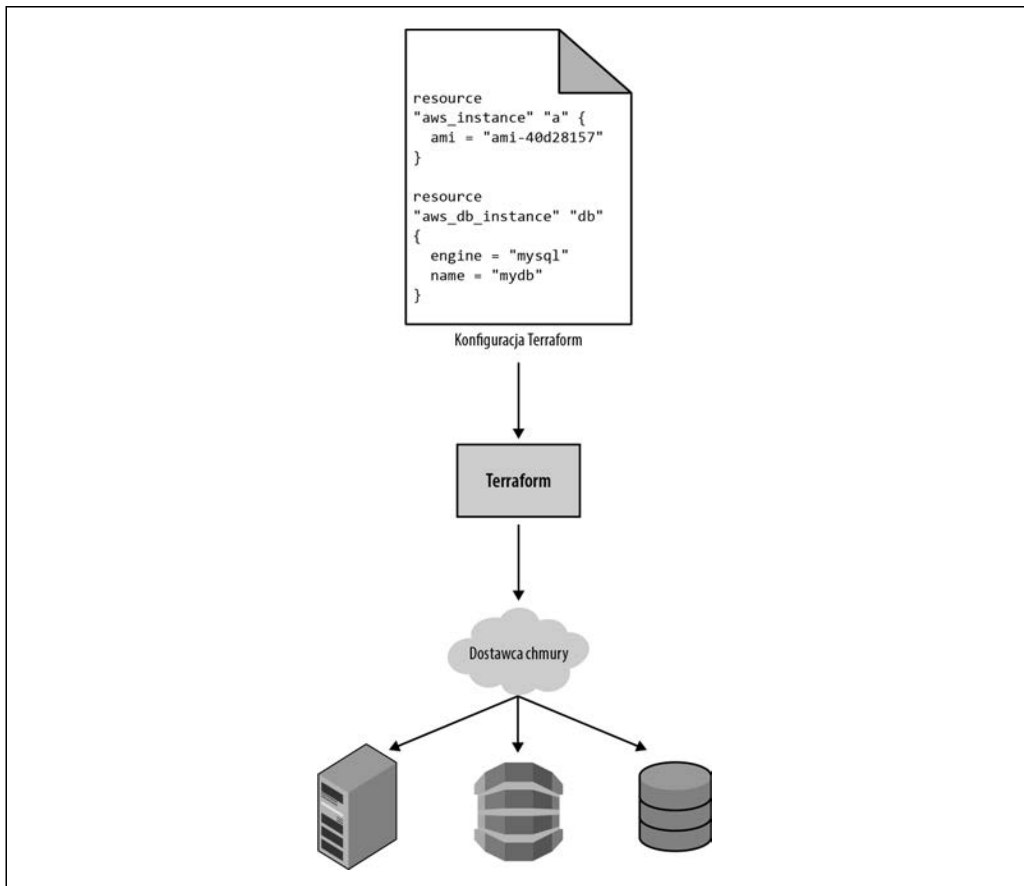
- Jeden lub więcej kontenerów Dockera, które będą razem uruchamiane. Taka grupa kontenerów jest w Kubernetesie określana mianem *pod*. Zdefiniowany w tym fragmencie kodu pod zawiera jeden kontener Dockera, w którym jest uruchamiany serwer WWW Apache.
- Ustawienia dla każdego kontenera Dockera w podzie. 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 klaster, 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.

Kup ksi k Pole ksi k

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

Kup ksi k Pole ksi k

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.

Kup ksi k Pole ksi k

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 autobus[4], 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 komponentach[5].

---

[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/*).

Kup ksi k Pole ksi k

Szczęście

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

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

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 tworzonych *konfiguracjach Terraform*, które są plikami tekstowymi określającymi infrastrukturę przeznaczoną do utworzenia. Wspomniane konfiguracje to „kod” w wyrażeniu „infrastruktura jako kod”. Spójrz na przykładową konfigurację Terraform.

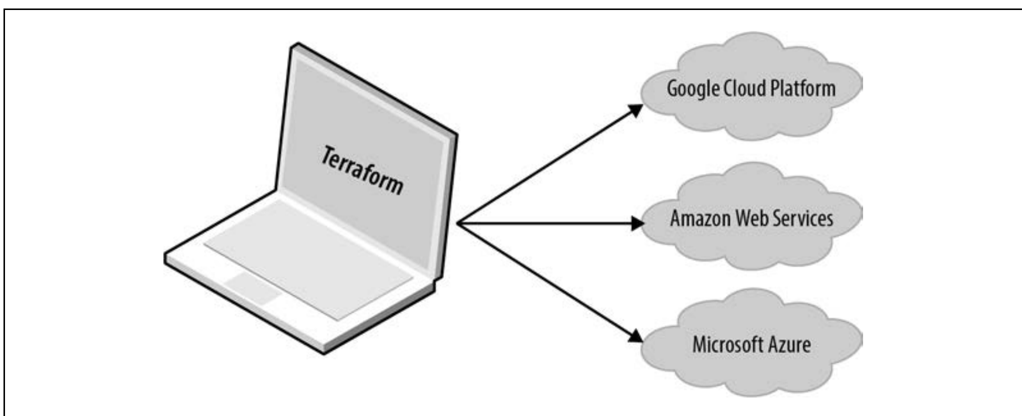
```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}

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


Kup ksi k Pole ksi k

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

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



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

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

Pełna przenośność między dostawcami chmury

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

Kup ksi k Pole ksi k

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 książki[6]. 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.

Kup ksi k Pole ksi k

- język proceduralny kontra deklaratywny,
- 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 nakażesz narzędziu Chef zainstalowanie nowej wersji OpenSSL, nastąpi uruchomienie procesu aktualizacji oprogramowania w istniejących serwerach i zmiany zostaną w nich wprowadzone. Wraz z upływem czasu i kolejnymi aktualizacjami każdy serwer ma unikatową historię zmian. W efekcie poszczególne serwery nieco się różnią od siebie, co prowadzi do powstawania drobnych błędów konfiguracji, które są trudne do zdiagnozowania i reprodukcji (to jest dokładnie ten sam problem związany ze zmianą konfiguracji jak w przypadku ręcznego zarządzania serwerami, choć zdecydowanie mniej kłopotliwy, gdy stosowane jest narzędzie zarządzania konfiguracją).

Kup ksi k Pole ksi k

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, niezmienność 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
```


Kup ksi k Pole ksi k

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"
    + (...)
  }
```


Kup ksi k Pole ksi k

```
# 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 wyśledzenia 10 wdrożonych wcześniej serwerów (a może to było 15 serwerów?) i ostrożnie uaktualnić każdy z nich. Natomiast w przypadku podejścia deklaratywnego wracasz do dokładnie 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.

Kup ksi k Pole ksi k

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 przeznczony 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.

Kup ksi k Pole ksi k

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. Konkretne 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 powłoce) 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

Kup książkę Poleć książkę

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 otworzenie 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.

Kup ksi k

Pole ksi k

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

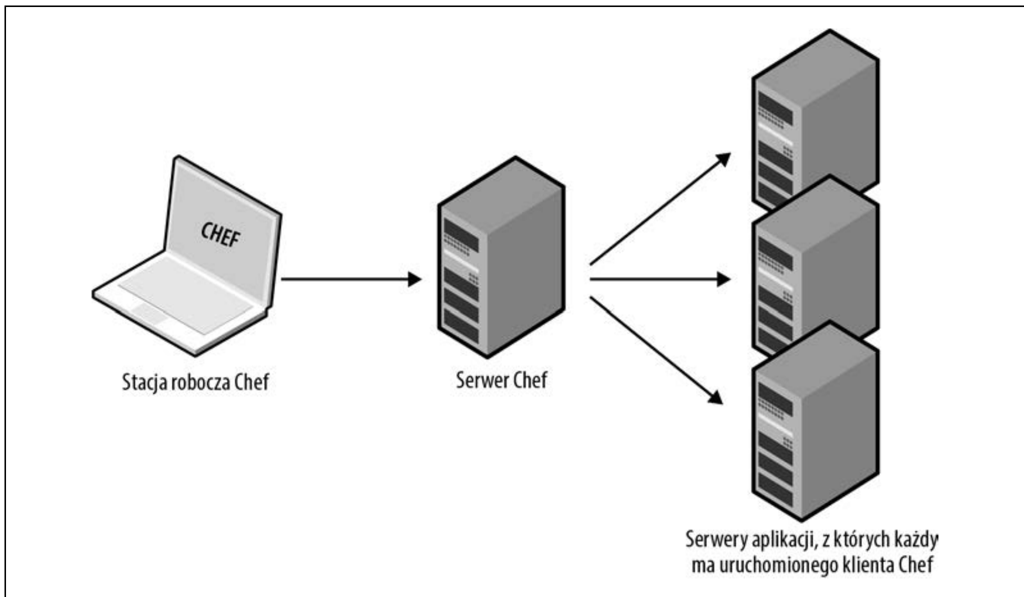
Jeżeli oprogramowanie agenta pobiera konfigurację z serwera głównego (lub innego serwera w przypadku nieużywania serwera głównego), konieczne jest otworzenie portów dla ruchu wychodzącego w każdym serwerze. Jeśli natomiast serwer główny przekazuje konfigurację do agenta, w każdym serwerze konieczne jest otworzenie 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.

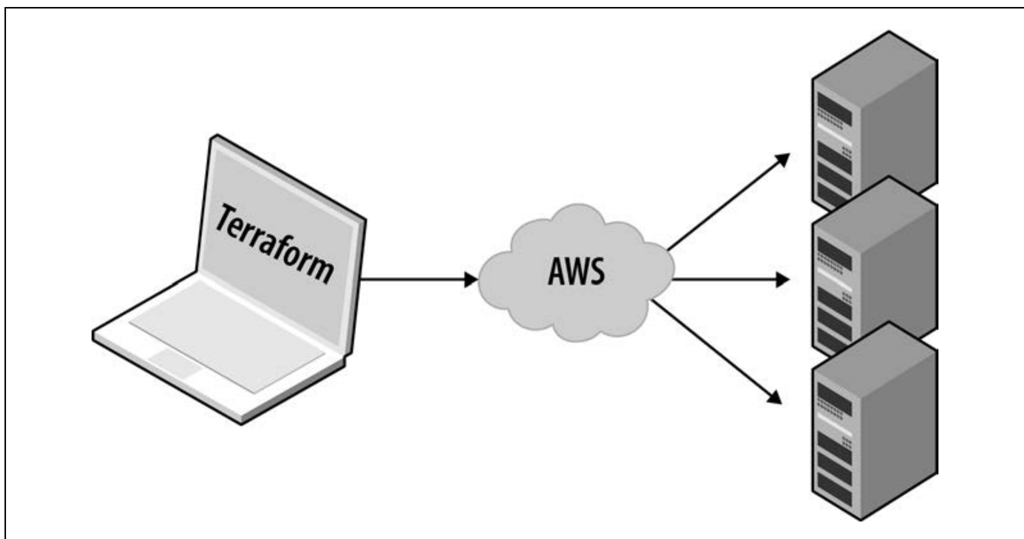
Wszystkie te ruchome elementy wprowadzają do infrastruktury ogromną liczbę nowych trybów awarii. Za każdym razem, gdy o trzeciej nad ranem otrzymasz zgłoszenie błędu, musisz ustalić, czy to jest wynik błędu w kodzie aplikacji, czy w kodzie IaC, czy w kliencie zarządzania konfiguracją, czy w serwerze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

Kup ksi k Pole ksi k



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

Kup ksi k
Pole ksi k

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.

Kup ksi k   Pole ksi k

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 StackOverflow[7].

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.

Kup książkę Poleć książkę

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

Kup ksi k Pole ksi k

będzie się uczyć na błędach? W tabeli 1.3 wymieniłem 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.

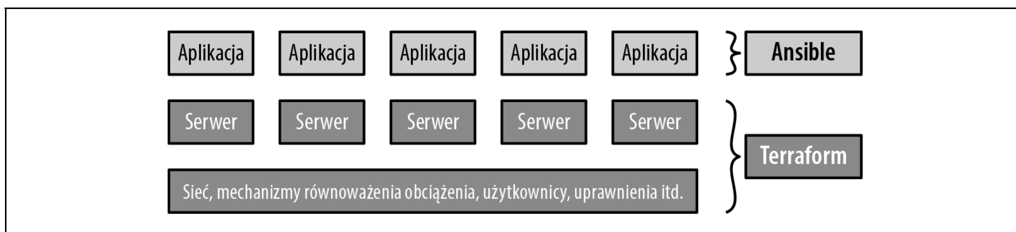
Używanie razem wielu narzędzi

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.

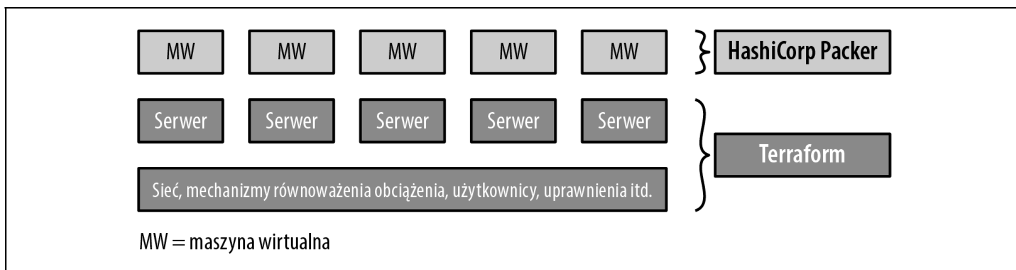


Rysunek 1.9. Terraform wdraża infrastrukturę, m.in. serwery, Ansible natomiast wdraża aplikacje w tych serwerach

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.



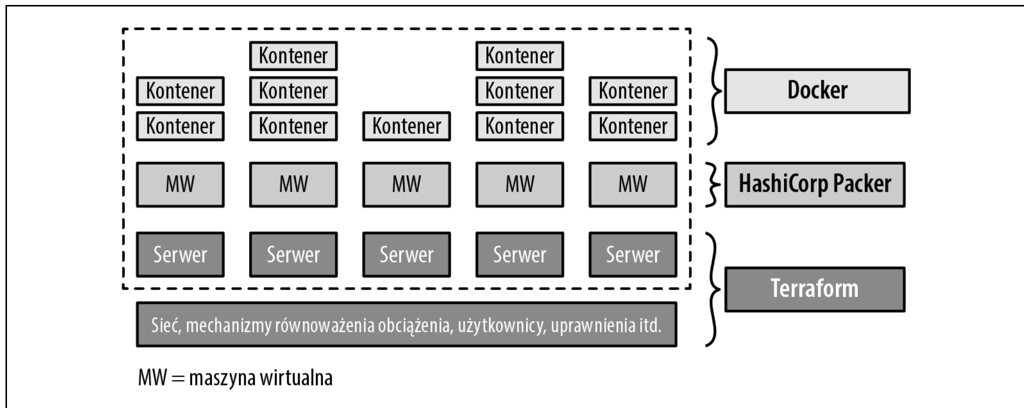
Rysunek 1.10. Terraform wdraża infrastrukturę, m.in. serwery, Packer natomiast tworzy maszyny wirtualne działające w tych serwerach

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 strategie wdrażania możliwe do implementacji za pomocą Terraform są ograniczone (nie możesz natywnie zastosować wdrożenia typu niebiesko-zielony). Dlatego skutkiem będzie tworzenie ogromnej liczby skomplikowanych skryptów wdrożenia lub zwrócenie się ku narzędziom instrumentacji, co przedstawię w następnym punkcie.

Kup ksi k Pole ksi k

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.).

Kup ksi k Pole ksi k

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
Społeczność	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 niezmiennej 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.

Kup ksi k   Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

Infrastruktura jako kod: od sukcesu dzieli Cię kilka poleceń!

Jeszcze do niedawna praca administratora systemu polegała na ręcznym przygotowywaniu infrastruktury do działania. Mozolne konfigurowanie serwerów, baz danych i elementów sieci niosło za sobą ryzyko przestojów środowiska produkcyjnego i wprowadzenia błędnych ustawień, a wdrożenia przebiegały powoli i łączyły się z nieuchronnym poszukiwaniem błędów. Dziś do tego rodzaju prac służy Terraform, narzędzie open source przeznaczone do tworzenia i wdrażania kodu infrastruktury, a także zarządzania nim; rozwiązanie stosowane w takich platformach jak Amazon Web Services, Google Cloud, Azure i wiele innych.

To trzecie, wzbogacone i uzupełnione wydanie praktycznego samouczka, dzięki któremu błyskawicznie rozpoczniesz pracę z Terraform. Zapoznasz się z językiem programowania Terraform i zasadami tworzenia kodu. Szybko zaczniesz wdrażać infrastrukturę i zarządzać nią za pomocą zaledwie kilku poleceń. Istotną częścią książki jest pokazanie metodologii DevOps w działaniu, a także wyjaśnienie zasad kodowania infrastruktury. Dziesiątki jasnych przykładów kodu, które można samodzielnie wypróbować w akcji, ułatwią zrozumienie podstaw. Nie musisz być weteranem DevOps ani doświadczonym administratorem systemów — z tym podręcznikiem nawet początkujący programiści sprawnie przygotują pełny stos, który zapewni obsługę ogromnego ruchu sieciowego w rzeczywistych środowiskach produkcyjnych.

Dzięki książce:

- zrozumiesz, kiedy używać Terraform, a kiedy innych narzędzi
- wdrożysz klastry serwerów, mechanizmy równoważenia obciążenia i bazy danych
- nauczysz się tworzyć infrastrukturę Terraform
- przetestujesz moduły Terraform wieloma metodami
- skonfigurujesz potoki CI/CD i zaczniesz się posługiwać zaawansowaną składnią Terraform
- nauczysz się pracować w środowiskach chmurowych

Yevgeniy (Jim) Brikman jest pasjonatem programowania i pisania o kodowaniu. Zafascynowany możliwościami, jakie daje DevOps, współzałożył firmę Gruntwork, która zajmuje się ułatwieniami w tworzeniu oprogramowania. Wcześniej pracował jako inżynier oprogramowania dla takich tuzów jak LinkedIn, TripAdvisor, Cisco Systems i Thomson Financial.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-8322-346-9

Cena: 99,00 zł