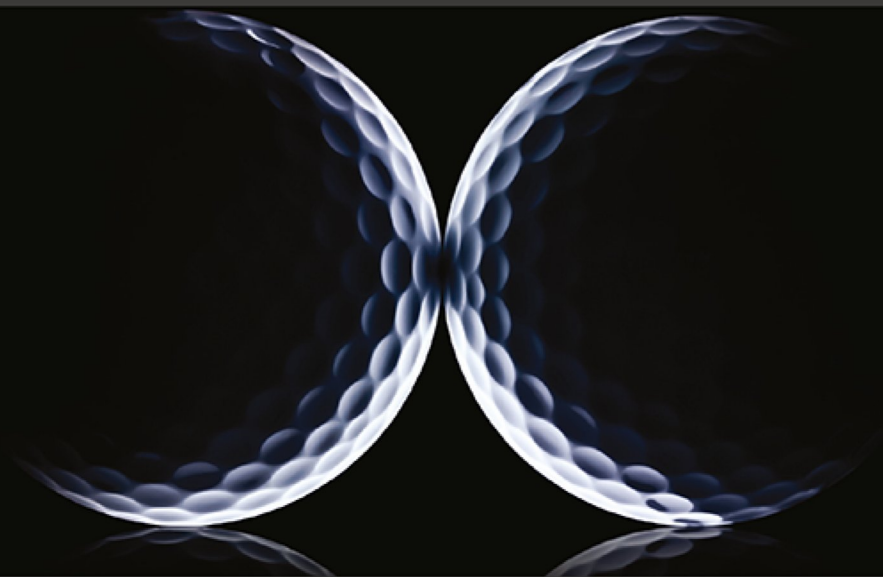


DevOps

w praktyce

Wydanie II

Wdrażanie narzędzi Terraform, Azure DevOps,
Kubernetes i Jenkins



Mikael Krief

Helion 



Tytuł oryginału: Learning DevOps: A comprehensive guide to accelerating DevOps culture adoption with Terraform, Azure DevOps, Kubernetes, and Jenkins, 2nd Edition

Tłumaczenie: Łukasz Wójcicki

ISBN: 978-83-8322-199-1

Copyright © Packt Publishing 2022. First published in the English language under the title 'Learning DevOps - Second Edition – (9781801818964)'

Polish edition copyright © 2023 by Helion S.A.

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/devpr2_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

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

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

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

*Chciałbym zadedykować tę książkę mojej żonie i dzieciom,
którzy są źródłem mojego szczęścia.*

Spis treści |

| | |
|-----------------------------|-----------|
| O autorze | 15 |
| O recenzentach | 16 |
| Przedmowa | 17 |

CZĘŚĆ 1. DevOps i infrastruktura jako kod

Rozdział 1

| | |
|---|-----------|
| Kultura DevOps i praktyki kodowania infrastruktury | 25 |
| Pierwsze kroki z DevOps | 25 |
| Wdrażanie CI/CD i ciągłe wdrażanie | 29 |
| Ciągła integracja (CI) | 29 |
| Ciągłe dostarczanie (CD) | 31 |
| Ciągłe wdrażanie | 33 |
| Zrozumienie praktyk IaC | 35 |
| Korzyści IaC | 36 |
| Języki i narzędzia IaC | 36 |
| Topologia IaC | 39 |
| Ewolucja kultury DevOps | 44 |
| Podsumowanie | 45 |
| Pytania | 45 |
| Dalsza lektura | 46 |

Rozdział 2

| | |
|--|-----------|
| Udostępnianie infrastruktury chmury za pomocą Terraform | 47 |
| Wymagania techniczne | 48 |
| Instalacja Terraform | 48 |
| Instalacja ręczna | 49 |
| Instalacja za pomocą skryptu | 49 |
| Integracja Terraform z Azure Cloud Shell | 53 |

| | |
|---|----|
| Konfigurowanie Terraform dla platformy Azure | 55 |
| Tworzenie jednostki usługi Azure SP | 55 |
| Konfiguracja dostawcy Terraform | 57 |
| Konfiguracja Terraform w celu rozwoju aplikacji i testowania | 58 |
| Tworzenie skryptu Terraform w celu wdrożenia infrastruktury Azure | 60 |
| Postępowanie zgodnie z dobrymi praktykami Terraform | 63 |
| Uruchamianie Terraform w celu wdrożenia | 66 |
| Inicjalizacja | 67 |
| Podgląd zmian | 69 |
| Stosowanie zmian | 70 |
| Zrozumienie cyklu życia Terraform z różnymi opcjami wiersza polecenia | 72 |
| Używanie polecenia destroy w celu przebudowy | 72 |
| Formatowanie i walidacja konfiguracji | 74 |
| Cykl życia Terraform w procesie CI/CD | 75 |
| Ochrona pliku stanu za pomocą zdalnego zaplecza | 77 |
| Podsumowanie | 82 |
| Pytania | 82 |
| Dalsza lektura | 83 |

Rozdział 3

| | |
|--|-----------|
| Używanie Ansible do konfigurowania infrastruktury IaaS | 84 |
| Wymagania techniczne | 85 |
| Instalacja Ansible | 86 |
| Instalacja Ansible za pomocą skryptu | 86 |
| Integracja Ansible z Azure Cloud Shell | 88 |
| Artefakty Ansible | 89 |
| Konfiguracja Ansible | 90 |
| Tworzenie pliku inwentarza Ansible | 92 |
| Plik inwentarza | 93 |
| Konfigurowanie hostów w pliku inwentarza | 94 |
| Testowanie pliku inwentarza | 95 |
| Uruchomienie pierwszego playbooka | 97 |
| Tworzenie prostego playbooka | 97 |
| Opis modułów Ansible | 98 |
| Ulepszanie playbooków za pomocą ról | 99 |
| Uruchomienie Ansible | 100 |
| Korzystanie z podglądu lub z opcji testowej pracy (ang. dry run) | 102 |
| Zwiększanie poziomu logowania | 104 |

| | |
|--|-----|
| Ochrona danych za pomocą Ansible Vault | 104 |
| Używanie zmiennych w Ansible w celu lepszej konfiguracji | 105 |
| Ochrona wrażliwych danych za pomocą Ansible Vault | 108 |
| Korzystanie z dynamicznego pliku inwentarza dla infrastruktury Azure | 110 |
| Podsumowanie | 115 |
| Pytania | 115 |
| Dalsza lektura | 115 |

Rozdział 4

| | |
|--|------------|
| Optymalizacja wdrażania infrastruktury za pomocą Packera | 117 |
| Wymagania techniczne | 118 |
| Opis Packera | 119 |
| Instalacja Packera | 119 |
| Tworzenie szablonów Packera dla maszyn wirtualnych Azure za pomocą skryptów | 125 |
| Struktura szablonu Packera | 125 |
| Tworzenie obrazu platformy Azure za pomocą szablonu Packera | 130 |
| Tworzenie szablonów Packera przy użyciu Ansible | 133 |
| Tworzenie playbooka Ansible | 134 |
| Integracja playbooka Ansible z szablonem Packera | 135 |
| Uruchamianie Packera | 136 |
| Konfigurowanie Packera do uwierzytelniania na platformie Azure | 136 |
| Sprawdzanie poprawności szablonu Packera | 137 |
| Uruchamianie Packera w celu wygenerowania naszego obrazu maszyny wirtualnej | 138 |
| Tworzenie szablonów Packera w formacie HCL | 140 |
| Korzystanie z obrazów utworzonych przez Packera za pomocą Terraform | 143 |
| Podsumowanie | 145 |
| Pytania | 145 |
| Dalsza lektura | 146 |

Rozdział 5

| | |
|--|------------|
| Tworzenie środowiska programistycznego z Vagrantem | 147 |
| Wymagania techniczne | 148 |
| Instalacja Vagranta | 148 |
| Instalacja ręczna (w systemie Windows) | 148 |
| Instalowanie Vagranta za pomocą skryptu w systemie Windows | 150 |
| Instalowanie Vagranta za pomocą skryptu w systemie Linux | 151 |

| | |
|--|-----|
| Tworzenie pliku konfiguracyjnego Vagranta | 152 |
| Używanie Vagrant Cloud dla boksów Vagranta | 152 |
| Tworzenie pliku konfiguracyjnego Vagranta | 154 |
| Tworzenie lokalnej maszyny wirtualnej za pomocą interfejsu Vagrant CLI | 156 |
| Tworzenie maszyny wirtualnej | 156 |
| Łączenie z maszyną wirtualną | 158 |
| Podsumowanie | 159 |
| Pytania | 160 |
| Dalsza lektura | 160 |

CZĘŚĆ 2. Potok CI/CD

Rozdział 6

| | |
|---|------------|
| Zarządzanie kodem źródłowym za pomocą Gita | 163 |
| Wymagania techniczne | 164 |
| Przegląd Gita i jego głównych poleceń | 164 |
| Instalacja Gita | 166 |
| Konfiguracja Gita | 176 |
| Terminologia Gita | 176 |
| Polecenia Gita | 177 |
| Zrozumienie procesu Gita i wzorca Gitflow | 181 |
| Zaczynamy od procesu Gita | 181 |
| Izolacja kodu za pomocą gałęzi | 190 |
| Strategia tworzenia gałęzi z Gitflow | 195 |
| Podsumowanie | 198 |
| Pytania | 198 |
| Dalsza lektura | 199 |

Rozdział 7

| | |
|---|------------|
| Ciągła integracja i ciągłe wdrażanie | 200 |
| Wymagania techniczne | 201 |
| Zasady CI/CD | 201 |
| CI | 201 |
| CD | 202 |
| Korzystanie z menedżera pakietów w procesie CI/CD | 203 |
| Prywatne repozytorium NuGet i npm | 205 |
| Repozytorium Nexusa OSS | 205 |
| Azure Artifacts | 206 |

| | |
|---|-----|
| Używanie Jenkinsa do implementacji CI/CD | 208 |
| Instalowanie i konfigurowanie Jenkinsa | 208 |
| Konfiguracja webhooka GitHuba | 210 |
| Konfiguracja zadania CI w Jenkinsie | 211 |
| Wykonywanie zadania Jenkinsa | 215 |
| Korzystanie z Azure Pipelines dla CI/CD | 216 |
| Wersjonowanie kodu za pomocą Gita w Azure Repos | 218 |
| Tworzenie potoku CI | 219 |
| Tworzenie potoku CD — nowa wersja aplikacji | 228 |
| Tworzenie pełnej definicji potoku w pliku YAML | 234 |
| Korzystanie z GitLab CI | 240 |
| Uwierzytelnianie w GitLabie | 241 |
| Tworzenie nowego projektu i zarządzanie kodem źródłowym | 242 |
| Tworzenie potoku CI | 245 |
| Dostęp do szczegółów wykonania potoku CI | 247 |
| Podsumowanie | 249 |
| Pytania | 249 |
| Dalsza lektura | 249 |

Rozdział 8

| | |
|--|------------|
| Wdrażanie infrastruktury jako kodu za pomocą potoku CI/CD | 251 |
| Wymagania techniczne | 252 |
| Uruchamianie Packera w Azure Pipelines | 252 |
| Uruchamianie Terraform i Ansible w Azure Pipelines | 255 |
| Podsumowanie | 260 |
| Pytania | 261 |
| Dalsza lektura | 261 |

CZĘŚĆ 3. Konteneryzowane mikrouслуги wykorzystujące platformę Docker i Kubernetes

Rozdział 9

| | |
|---|------------|
| Konteneryzacja aplikacji za pomocą Dockera | 265 |
| Wymagania techniczne | 266 |
| Instalowanie Dockera | 267 |
| Rejestracja w Docker Hubie | 267 |
| Instalacja Dockera | 268 |
| Przegląd elementów Dockera | 272 |

| | |
|--|-----|
| Tworzenie pliku Dockerfile | 273 |
| Tworzenie pliku Dockerfile | 273 |
| Przegląd instrukcji Dockerfile | 274 |
| Budowanie i uruchamianie kontenera na komputerze lokalnym | 276 |
| Tworzenie obrazu Dockera | 276 |
| Tworzenie nowego kontenera obrazu | 278 |
| Lokalne testowanie kontenera | 279 |
| Wysyłanie obrazu do Docker Huba | 279 |
| Wysyłanie obrazu Dockera do rejestru prywatnego (ACR) | 283 |
| Wdrażanie kontenera do ACI za pomocą potoku CI/CD | 285 |
| Tworzenie kodu Terraform dla ACI | 286 |
| Tworzenie potoku CI/CD dla kontenera | 287 |
| Korzystanie z Dockera przy użyciu narzędzi wiersza poleceń | 294 |
| Pierwsze kroki z Docker Compose | 296 |
| Instalowanie Docker Compose | 297 |
| Tworzenie pliku konfiguracyjnego dla Docker Compose | 298 |
| Wykonywanie Docker Compose | 299 |
| Wdrażanie kontenerów Docker Compose w ACI | 300 |
| Podsumowanie | 303 |
| Pytania | 303 |
| Dalsza lektura | 304 |

Rozdział 10

| | |
|--|------------|
| Efektywne zarządzanie kontenerami za pomocą Kubernetesa | 305 |
| Wymagania techniczne | 306 |
| Instalacja Kubernetesa | 306 |
| Przegląd architektury Kubernetesa | 307 |
| Instalacja Kubernetesa na komputerze lokalnym | 307 |
| Instalacja pulpitu nawigacyjnego Kubernetesa | 309 |
| Pierwszy przykład wdrożenia aplikacji w Kubernetesie | 312 |
| Używanie Helma jako menedżera pakietów | 316 |
| Instalacja klienta Helma | 316 |
| Korzystanie z publicznego pakietu Helma, dostępnego w Artifact Hubie | 317 |
| Tworzenie niestandardowego charta Helma | 321 |
| Publikowanie charta Helma w rejestrze prywatnym (ACR) | 323 |
| Korzystanie z AKS | 325 |
| Tworzenie usługi AKS | 326 |
| Konfigurowanie pliku kubeconfig dla AKS | 327 |
| Zalety AKS | 328 |
| Tworzenie potoku CI/CD dla Kubernetesa za pomocą Azure Pipelines | 329 |

| | |
|---|-----|
| Monitorowanie aplikacji i metryk w Kubernetesie | 330 |
| Korzystanie z wiersza poleceń kubectl | 330 |
| Korzystanie z interfejsu webowego | 331 |
| Korzystanie z narzędzi | 332 |
| Podsumowanie | 334 |
| Pytania | 334 |
| Dalsza lektura | 335 |

CZĘŚĆ 4. Testowanie aplikacji

Rozdział 11

| | |
|--|------------|
| Testowanie interfejsów API za pomocą Postmana | 339 |
| Wymagania techniczne | 340 |
| Tworzenie kolekcji żądań Postmana | 340 |
| Instalacja Postmana | 341 |
| Tworzenie kolekcji | 342 |
| Tworzenie pierwszego żądania | 343 |
| Wykorzystywanie środowisk i zmiennych do dynamizowania żądań | 346 |
| Tworzenie testów Postmana | 349 |
| Wykonywanie lokalnych testów za pomocą żądań Postmana | 351 |
| Zrozumienie koncepcji Newmana | 354 |
| Przygotowywanie kolekcji Postmana dla Newmana | 356 |
| Eksportowanie kolekcji | 356 |
| Eksportowanie środowisk | 357 |
| Korzystanie z wiersza poleceń Newmana | 359 |
| Integracja Newmana z procesem potoku CI/CD | 361 |
| Budowa i udostępnianie konfiguracji | 362 |
| Wykonanie potoku | 366 |
| Podsumowanie | 367 |
| Pytania | 368 |
| Dalsza lektura | 368 |

Rozdział 12

| | |
|---|------------|
| Statyczna analiza kodu za pomocą SonarQube | 369 |
| Wymagania techniczne | 370 |
| Odkrywanie SonarQube | 370 |
| Instalacja SonarQube | 371 |
| Przegląd architektury SonarQube | 371 |
| Instalacja SonarQube | 372 |

| | |
|---|-----|
| Analiza w czasie rzeczywistym za pomocą SonarLint | 378 |
| Wykonywanie SonarQube w procesie CI | 380 |
| Konfigurowanie SonarQube | 381 |
| Tworzenie potoku CI dla SonarQube w Azure Pipelines | 381 |
| Podsumowanie | 385 |
| Pytania | 385 |
| Dalsza lektura | 385 |

Rozdział 13

| | |
|--|------------|
| Testy bezpieczeństwa i wydajności | 386 |
| Wymagania techniczne | 386 |
| Stosowanie zabezpieczeń internetowych i testów penetracyjnych za pomocą narzędzia ZAP | 387 |
| Korzystanie z ZAP-a w celu testowania bezpieczeństwa | 388 |
| Sposoby automatyzacji wykonywania ZAP-a | 390 |
| Uruchamianie testów wydajności za pomocą Postmana | 392 |
| Podsumowanie | 395 |
| Pytania | 395 |
| Dalsza lektura | 395 |

CZĘŚĆ 5. Więcej informacji na temat DevOps

Rozdział 14

| | |
|--|------------|
| Bezpieczeństwo w procesie DevOps z wykorzystaniem DevSecOps | 399 |
| Wymagania techniczne | 400 |
| Testowanie infrastruktury Azure za pomocą InSpec | 400 |
| Omówienie InSpec | 401 |
| Instalacja InSpec | 402 |
| Konfigurowanie platformy Azure dla InSpec | 403 |
| Tworzenie testów InSpec | 405 |
| Wykonywanie InSpec | 408 |
| Ochrona poufnych danych dzięki Vault od HashiCorp | 410 |
| Lokalna instalacja programu Vault | 411 |
| Uruchamianie serwera Vault | 413 |
| Zapisywanie haseł w Vault | 415 |
| Odczytywanie sekretów z Vault | 416 |
| Korzystanie z interfejsu webowego (UI) programu Vault | 418 |
| Pobieranie sekretów Vault w Terraform | 421 |

| | |
|----------------------|-----|
| Podsumowanie | 425 |
| Pytania | 425 |
| Dalsza lektura | 425 |

Rozdział 15

| | |
|--|------------|
| Skrócenie czasu przestoju wdrażania | 426 |
| Wymagania techniczne | 427 |
| Skrócenie czasu przestojów we wdrażaniu dzięki Terraform | 427 |
| Zrozumienie zielono-niebieskich koncepcji i wzorców wdrażania | 429 |
| Korzystanie z wdrożenia zielono-niebieskiego w celu ulepszenia środowiska produkcyjnego | 430 |
| Opis wzorca Canary release | 430 |
| Badanie wzorca Dark launch | 432 |
| Stosowanie wdrożeń zielono-niebieskich na platformie Azure | 432 |
| Używanie App Service z gniazdami | 433 |
| Korzystanie z usługi Azure Traffic Manager | 434 |
| Wprowadzenie flag funkcjonalności | 436 |
| Używanie frameworka open source dla flag funkcjonalności | 438 |
| Korzystanie z narzędzia LaunchDarkly | 443 |
| Podsumowanie | 447 |
| Pytania | 448 |
| Dalsza lektura | 448 |

Rozdział 16

| | |
|--|------------|
| DevOps dla projektów open source | 449 |
| Wymagania techniczne | 450 |
| Przechowywanie kodu źródłowego w GitHubie | 451 |
| Tworzenie nowego repozytorium na GitHubie | 451 |
| Przyczynianie się do rozwoju projektu w GitHubie | 453 |
| Przyczynianie się do rozwoju projektów open source przy użyciu żądań pobierania | 455 |
| Zarządzanie plikiem dziennika zmian i informacjami o wydaniu | 459 |
| Udostępnianie plików binarnych w wydaniach GitHuba | 461 |
| Wprowadzenie do GitHub Actions | 464 |
| Analiza kodu za pomocą SonarCloud | 468 |
| Wykrywanie luk w zabezpieczeniach za pomocą narzędzia WhiteSource Bolt | 473 |
| Podsumowanie | 476 |
| Pytania | 477 |
| Dalsza lektura | 478 |

Rozdział 17**Najlepsze praktyki DevOps479**

| | |
|---|-----|
| Pełna automatyzacja | 480 |
| Wybór odpowiedniego narzędzia | 480 |
| Tworzenie całej konfiguracji za pomocą kodu | 481 |
| Projektowanie architektury systemu | 482 |
| Budowanie dobrego potoku CI/CD | 484 |
| Testy integracyjne | 485 |
| Przesunięcie bezpieczeństwa w lewo dzięki DevSecOps | 486 |
| Monitorowanie systemu | 487 |
| Ewoluuujące zarządzanie projektami | 488 |
| Podsumowanie | 489 |
| Pytania | 489 |
| Dalsza lektura | 490 |

Odpowiedzi491

| | |
|---|-----|
| Rozdział 1. Kultura DevOps i praktyki kodowania infrastruktury | 491 |
| Rozdział 2. Udostępnianie infrastruktury chmury za pomocą Terraform | 491 |
| Rozdział 3. Używanie Ansible do konfigurowania infrastruktury IaaS | 492 |
| Rozdział 4. Optymalizacja wdrażania infrastruktury za pomocą Packera | 492 |
| Rozdział 5. Tworzenie środowiska programistycznego z Vagrantem | 493 |
| Rozdział 6. Zarządzanie kodem źródłowym za pomocą Gita | 493 |
| Rozdział 7. Ciągła integracja i ciągłe wdrażanie | 494 |
| Rozdział 8. Wdrażanie infrastruktury jako kodu za pomocą potoku CI/CD | 494 |
| Rozdział 9. Konteneryzacja aplikacji za pomocą Dockera | 494 |
| Rozdział 10. Efektywne zarządzanie kontenerami za pomocą Kubernetesa | 495 |
| Rozdział 11. Testowanie interfejsów API za pomocą Postmana | 495 |
| Rozdział 12. Statyczna analiza kodu za pomocą SonarQube | 495 |
| Rozdział 13. Testy bezpieczeństwa i wydajności | 496 |
| Rozdział 14. Bezpieczeństwo w procesie DevOps z wykorzystaniem DevSecOps | 496 |
| Rozdział 15. Skrócenie czasu przestoju wdrażania | 496 |
| Rozdział 16. DevOps dla projektów open source | 497 |
| Rozdział 17. Najlepsze praktyki DevOps | 497 |

Skorowidz499

O autorze

Mikael Krief (ur. 1980) mieszka we Francji i pracuje jako inżynier DevOps.

Uwielbia dzielić się swoją pasją, będąc członkiem różnych społeczności, np. HashiCorp User Group. W 2019 r. napisał pierwsze wydanie tej książki, a w 2020 opublikował *Terraform Cookbook* (Packt Publishing). Bierze udział w wielu projektach publicznych, pisze blogi i książki oraz przemawia na konferencjach.

Interesuje się produktami HashiCorp i specjalizuje się w wykorzystaniu Terraform w kilku projektach firmowych.

Za cały swój wkład i pasję otrzymał nagrodę Microsoftu **Most Valuable Professional** (MVP), którą Microsoft przyznaje mu przez ostatnie 6 lat, a od 2020 r. jest nominowany i wybierany na ambasadora HashiCorp.

*Chciałbym podziękować mojej rodzinie za zaakceptowanie tego,
że musiałem pracować nad tą książką przez długie godziny w czasie,
który mógłbym poświęcić rodzinie.*

*Chciałbym podziękować Meeti Rajani
za umożliwienie mi napisania drugiego wydania tej książki,
co było bardzo wzbogacającym doświadczeniem.*

*Specjalne podziękowania dla Romy Dias i Vaidehi Sawant
za ich cenny wkład*

*i czas poświęcony na przejrzenie tej książki
oraz dla całego zespołu Packt za wsparcie podczas pisania tej książki.*

O recenzentach

Kevin Bridges jest starszym inżynierem DevOps w firmie Alegeus. Pracował z szeroką gamą narzędzi wspierających stabilne wydania i poprawki. Nie posiada dużego doświadczenia związanego z zagadnieniami chmurowymi (Azure). W czerwcu 2021 r. otrzymał certyfikat Azure Fundamentals.

Kevin spędził 20 lat w armii w charakterze czynnym i rezerwowym. W 2009 r. ukończył Granite State College z tytułem Bachelor of Science w dziedzinie technologii informatycznych, a w maju 2015 r. uzyskał tytuł MBA na University of New Hampshire (Manchester).

Kevin uwielbia upraszczać skomplikowane rzeczy.

*Chciałbym podziękować całej mojej rodzinie,
przyjaciołom i współpracownikom za wsparcie.*

Deb Bhattacharya stosuje agile i DevOps, aby organizacje odnosiły większe sukcesy.

W ciągu 20 lat Deb pomogła ponad 50 zespołom w 4 krajach być bardziej zwinnymi (ang. *agile*) i bardziej DevOps. Jest pasjonatką tych dziedzin.

Inną pasją Deb jest sport. Kiedy była młodsza, grała zawodowo w tenisa stołowego. Wygrała wiele turniejów, ale przede wszystkim była trenerem zwycięskich drużyn tenisa stołowego. Nadal lubi grać w tenisa stołowego.

Deb wykorzystuje swoje praktyczne doświadczenie programistyczne i trenerskie, aby rozwijać wysokowydajne zespoły agile i DevOps. Te dwie pasje łącznie się tu łączą.

*Jestem wdzięczna Anandowi Athanemu, delivery managerowi,
któremu podlegałam jakieś 20 lat temu, kiedy byłam liderem inżynierii.
Pisałam pracę magisterską i Anand zasugerował Rational Unified
Process (RUP). Od tego zaczęła się moja przygoda z agile.
Nauka i wnoszenie wkładu do powyższych technologii wciąż trwają.*

Przedmowa |

Obecnie, wraz z rozwojem technologii i stale rosnącą konkurencją, firmy stają przed prawdziwym wyzwaniem szybszego projektowania i dostarczania produktów — a wszystko to przy zachowaniu satysfakcji użytkowników.

Jednym z rozwiązań dla tego wyzwania jest wprowadzenie (do firm) kultury współpracy między różnymi zespołami, takimi jak deweloperzy i operatorzy, testerzy i zespoły ds. bezpieczeństwa. Ta kultura została już sprawdzona i nazywa się kulturą DevOps. Zapewnia ona współpracę zespołów oraz praktyki, które skracają czas wprowadzania produktu na rynek — dzięki krótszym cyklom wdrażania aplikacji i wnoszeniu rzeczywistej wartości do produktów i aplikacji firmy.

Co więcej, wraz z przechodzeniem firm w kierunku chmury infrastruktury aplikacji ewoluują, a kultura DevOps pozwala na lepszą skalowalność i wydajność aplikacji, generując w ten sposób zyski finansowe dla firm.

Jeśli chcesz dowiedzieć się więcej o kulturze DevOps i zastosować jej praktyki w swoich projektach, ta książka wprowadzi Cię w podstawy praktyk DevOps za pomocą różnych narzędzi i laboratoriów.

Omówimy tutaj podstawy kultury i praktyk DevOps, a następnie przyjrzymy się różnym laboratoriom używanym do wdrażania praktyk DevOps takich jak IaC i wykorzystującym potoki Git i CI/CD, automatyzację testów, analizę kodu i DevSecOps wraz z dodatkowymi zabezpieczeniami Twoich procesów.

Część książki jest poświęcona konteneryzacji aplikacji, z omówieniem prostego użycia Dockera i zarządzania kontenerami w Kubernetesie. Obejmuje tematy dotyczące redukcji przestojów podczas wdrażania i praktyki DevOps w projektach open source.

Książkę kończy rozdział poświęcony niektórym dobrym praktykom DevOps, które można wdrożyć w całym cyklu życia projektów.

W tej drugiej edycji wszystkie narzędzia są zaktualizowane. Dowiesz się również coś na temat narzędzia Vagrant od HashiCorp oraz więcej na temat wdrażania Kubernetesa.

Książka ma na celu poprowadzenie Cię krok po kroku przez wdrożenie praktyk DevOps przy użyciu różnych narzędzi, które w większości są otwartoźródłowe lub są najlepsze na rynku.

Moim celem jako autora jest podzielenie się z Tobą codziennymi doświadczeniami. Mam nadzieję, że będzie to dla Ciebie przydatne i znajdzie zastosowanie w Twoich projektach.

Dla kogo jest ta książka

Ta książka jest przeznaczona dla każdego, kto chce rozpocząć wdrażanie praktyk DevOps. Nie jest wymagana żadna konkretna wiedza na temat programowania ani obsługi systemu.

Co zawiera ta książka

Rozdział 1., „Kultura DevOps i praktyki kodowania infrastruktury”, wyjaśnia cele kultury DevOps i wyszczególnia różne praktyki DevOps — potoki IaC i CI/CD — które zostaną omówione w tej książce.

Rozdział 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”, zawiera szczegółowe informacje dotyczące udostępniania infrastruktury chmurowej z IaC przy użyciu Terraform, w tym jego instalację, wiersz poleceń, cykl życia, praktyczne wykorzystanie do udostępniania przykładowej infrastruktury Azure i do ochrony plików stanu Terraform za pomocą zdalnych backendów.

Rozdział 3., „Używanie Ansible do konfigurowania infrastruktury IaaS”, dotyczy konfiguracji maszyn wirtualnych za pomocą Ansible, w tym instalacji Ansible, wiersza poleceń, konfigurowania ról dla pliku inwentarza i playbooka, jego wykorzystania do konfigurowania maszyn wirtualnych na platformie Azure, ochrony danych za pomocą Ansible Vault i korzystania z dynamicznego inwentarza.

Rozdział 4., „Optymalizacja wdrażania infrastruktury za pomocą Packera”, obejmuje zastosowanie narzędzia Packer do tworzenia obrazów maszyn wirtualnych, w tym jego instalację i sposób użycia do tworzenia obrazów na platformie Azure.

Rozdział 5., „Tworzenie środowiska programistycznego z Vagrantem”, wyjaśnia, jak zbudować lokalne środowisko programistyczne przy użyciu IaC i Vagranta.

Rozdział 6., „Zarządzanie kodem źródłowym za pomocą Gita”, omawia korzystanie z Gita, w tym jego instalację, główne polecenia, przepływ pracy, przegląd systemu tworzenia gałęzi i przykład przepływu pracy za pomocą Gitflow.

Rozdział 7., „Ciągła integracja i ciągłe wdrażanie”, pokazuje tworzenie kompleksowego potoku CI/CD przy użyciu trzech różnych narzędzi: Jenkins, GitLab CI i Azure Pipelines. Każde z tych narzędzi zostało szczegółowo omówione.

Rozdział 8., „Wdrażanie infrastruktury jako kodu za pomocą potoku CI/CD”, omawia użycie potoków CI/CD z Azure Pipelines do automatycznego uruchamiania narzędzi Packer, Terraform i Ansible.

Rozdział 9., „Konteneryzacja aplikacji za pomocą Dockera”, obejmuje opis korzystania z Dockera, w tym omawia jego lokalną instalację, rejestr Docker Hub, tworzenie pliku Dockerfile i demonstrację, w jaki sposób go używać. Przykładowa aplikacja zostanie umieszczona w kontenerze, wykonana lokalnie, a następnie wdrożona w instancji kontenera platformy Azure za pośrednictwem potoku CI/CD.

Rozdział 10., „Efektywne zarządzanie kontenerami za pomocą Kubernetesa”, wyjaśnia podstawowe użycie Kubernetesa, w tym lokalną instalację i wdrażanie aplikacji, a następnie przykład Kubernetesa zarządzanego za pomocą Azure Kubernetes Services.

Rozdział 11., „Testowanie interfejsów API za pomocą Postmana”, szczegółowo opisuje wykorzystanie Postmana do przetestowania przykładowego interfejsu API, w tym jego lokalnego użycia i automatyzacji w potoku CI/CD z wykorzystaniem Newmana i Azure Pipelines.

Rozdział 12., „Statyczna analiza kodu za pomocą SonarQube”, wyjaśnia użycie SonarQube do analizy kodu statycznego w aplikacji, w tym jego instalację, analizę w czasie rzeczywistym za pomocą narzędzia SonarLint i integrację SonarQube z potokiem CI w Azure Pipelines.

Rozdział 13., „Testy bezpieczeństwa i wydajności”, omawia bezpieczeństwo i wydajność aplikacji webowych, w tym demonstrację, jak używać narzędzia ZAP do testowania reguł OWASP i Postmana do testowania wydajności API.

Rozdział 14., „Bezpieczeństwo w procesie DevOps z wykorzystaniem DevSecOps”, wyjaśnia, jak korzystać z integracji zabezpieczeń w procesie DevOps poprzez testowanie zgodności infrastruktury za pomocą InSpec i użycie Vault do ochrony poufnych danych.

Rozdział 15., „Skrócenie czasu przestoju wdrażania”, przedstawia ograniczanie przestojów wdrażania za pomocą Terraform, koncepcje i wzorce wdrażania zielono-niebieskiego oraz sposoby ich stosowania na platformie Azure. Znaczny nacisk kładzie się również na wykorzystanie flag funkcjonalności w aplikacji.

Rozdział 16., „DevOps dla projektów open source”, jest poświęcony tematyce open source. Szczegółowo opisuje narzędzia, procesy i praktyki dla projektów open source we współpracy z GitHubem, żądaniami pobrania, plikami dziennika zmian, udostępnianiem binarnym w wydaniach GitHuba oraz kompleksowym przykładem potoku CI w GitHub Actions.

Omówiona została także analiza kodu open source i problem bezpieczeństwa z wykorzystaniem narzędzi SonarCloud i WhiteSource Bolt.

Rozdział 17., „Najlepsze praktyki DevOps”, zawiera przegląd listy dobrych praktyk DevOps dotyczących automatyzacji, IaC, potoków CI/CD, testowania, bezpieczeństwa, monitorowania projektów i zarządzania projektami.

Jak najlepiej skorzystać z tej książki

Do zrozumienia tej książki nie jest wymagana żadna wiedza programistyczna. Jedyne języki, które zobaczysz, to języki deklaratywne, takie jak JSON lub YAML. Oprócz tego nie jest wymagane żadne specjalne IDE. Jeśli go nie masz, możesz skorzystać z Visual Studio Code, który jest bezpłatny i wieloplatformowy. Jest dostępny tutaj: <https://code.visualstudio.com/>.

Przykładowym dostawcą usług w chmurze w tej książce jest Microsoft Azure. Jeśli nie masz subskrypcji, możesz utworzyć bezpłatne konto tutaj: <https://azure.microsoft.com/en-us/free/>.

Jeśli chodzi o systemy operacyjne, których będziesz potrzebować, nie ma żadnych rzeczywistych wymagań wstępnych. Większość narzędzi, których będziemy używać, jest wieloplatformowa i kompatybilna z systemami Windows, Linux i macOS. Ich instalacje zostaną szczegółowo opisane w odpowiednich rozdziałach.

| Oprogramowanie omówione w książce | Wymagany system operacyjny |
|-----------------------------------|----------------------------|
| Terraform | Windows, macOS lub Linux |
| Ansible | macOS lub Linux |
| Packer | Windows, macOS lub Linux |
| Vagrant | Windows, macOS lub Linux |
| Docker | Windows, macOS lub Linux |
| Azure CLI | Windows, macOS lub Linux |
| Helm | Windows, macOS lub Linux |
| Postman | Windows, macOS lub Linux |
| SonarLint | Windows, macOS lub Linux |
| Java Runtime | Windows, macOS lub Linux |

Jeśli korzystasz z cyfrowej wersji tej książki, radzimy pisać kod źródłowy samodzielnie lub uzyskać dostęp do kodu z repozytorium GitHuba książki (link znajduje się w następnej sekcji). Pomoże to uniknąć potencjalnych błędów związanych z kopiowaniem i wklejaniem kodu.

Pobieranie przykładowych plików z kodem źródłowym

Możesz pobrać przykładowe pliki z kodem źródłowym dla tej książki z GitHuba — spod adresu <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition>. Jeśli istnieje aktualizacja kodu, zostanie ona zaktualizowana w repozytorium GitHuba.

Kod w wersji z polskiego wydania książki można również pobrać pod adresem: <https://ftp.helion.pl/przyklady/devpr2.zip>.

Code in Action

Filmy dotyczące tej książki (w języku angielskim) można obejrzeć na kanale Code in Action pod adresem <https://bit.ly/36xzV7u>.

Pobieranie kolorowych zdjęć

Udostępniamy również plik PDF zawierający kolorowe zrzuty ekranu i diagramy używane w tej książce. Można go pobrać tutaj: <https://ftp.helion.pl/przyklady/devpr2.zip>.

Stosowane konwencje

W tej książce zastosowano wiele konwencji tekstowych.

Kod w tekście: wskazuje kod źródłowy w tekście.

Wyróżnienie w tekście: wskazuje nazwy folderów, nazwy plików, rozszerzenia plików, ścieżki, adresy URL i łącza Twittera. Oto przykład: „Przejdź do folderu, w którym utworzyliśmy plik *Vagrantfile*”.

Blok kodu źródłowego jest sformatowany w następujący sposób:

```
pool:
  vmImage: ubuntu-latest
steps:
- task: DotNetCoreCLI@2
  displayName: "Restore"
  inputs:
```

Gdy chcemy zwrócić Twoją uwagę na określoną część bloku kodu, odpowiednie wiersze lub elementy są pogrubione:

```
[ inputs:
  command: 'test'
  projects: '**/tests/*.csproj'
  arguments: '--configuration Release'
- task: DotNetCoreCLI@2
```

Każda instrukcja lub dane wyjściowe są zapisywane w następujący sposób:

```
sudo apt-get update && sudo apt-get install -y gnupg
↳softwareproperties-common curl \
```

Pogrubienie: oznacza nowy termin lub ważne słowo.

Uwaga

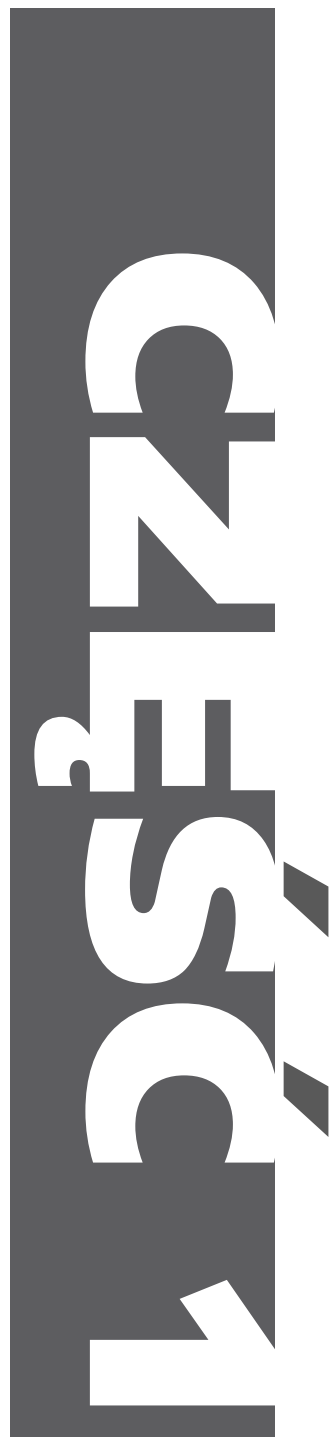
Wskazówki lub ważne uwagi wyglądają tak.

DevOps i infrastruktura jako kod

Celem części pierwszej jest przedstawienie kultury DevOps oraz wskazówek w celu uzyskania dobrych praktyk kodowania infrastruktury. W tej części wyjaśniono zastosowanie DevOps w infrastrukturze chmury, demonstrując udostępnianie (ang. *provisioning*) przy użyciu Terraform i konfigurację za pomocą Ansible. Następnie ulepszymy ten proces, tworząc szablon tej infrastruktury za pomocą narzędzia Packer.

Sekcja ta składa się z następujących rozdziałów:

- Rozdział 1., „Kultura DevOps i praktyki kodowania infrastruktury”.
- Rozdział 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”.
- Rozdział 3., „Używanie Ansible do konfiguracji infrastruktury IaaS”.
- Rozdział 4., „Optymalizacja wdrażania infrastruktury za pomocą Packera”.
- Rozdział 5., „Tworzenie środowiska programistycznego z Vagrantem”.



Kultura DevOps i praktyki kodowania infrastruktury

Rozdział 1

DevOps, termin, który coraz częściej słyszymy w przedsiębiorstwach z frazami typu „stosujemy DevOps” czy „używamy narzędzi DevOps”, to skrót pochodzący od słów „Development” i „Operations”.

DevOps to kultura, która różni się od tradycyjnych kultur korporacyjnych i wymaga zmiany sposobu myślenia, procesów i narzędzi. Często wiąże się to z praktykami **ciągłej integracji** (ang. *continuous integration* — CI) i **ciągłego dostarczania** (ang. *continuous delivery* — CD), które są związane z inżynierią oprogramowania, ale także z **infrastrukturą jako kod** (ang. *Infrastructure as Code* — IaC), polegającą na *kodyfikacji* struktury i konfiguracji infrastruktury.

W tym rozdziale zobaczymy, czym jest kultura DevOps, jakie są zasady DevOps i jakie korzyści przynoszą firmie. Następnie wyjaśnimy praktyki CI/CD, a na koniec omówimy szczegółowo IaC z jej wzorcami i praktykami.

W tym rozdziale omówimy następujące tematy:

- pierwsze kroki z DevOps,
- wdrażanie CI/CD i ciągłe wdrażanie,
- zrozumienie praktyk IaC.

Obejrzyj następujący film na kanale Code in Action: <https://bit.ly/3JJAMAb>.

Pierwsze kroki z DevOps

Termin *DevOps* został wprowadzony w latach 2007 – 2009 przez Patricka Debois, Gene’a Kima i Johna Willisa i reprezentuje połączenie słów *Development* (Dev) i *Operations* (Ops). Dało to początek ruchowi, który opowiada się za łączeniem razem programistów i operacji. Zapewnia to użytkownikom dodaną wartość biznesową dużo szybciej, co czyni ją bardziej konkurencyjną na rynku.

Kultura DevOps to zestaw praktyk zmniejszających bariery między *programistami*, którzy chcą szybciej wprowadzać innowacje i dostarczać, a *zespołami operacyjnymi*, które chcą zagwarantować stabilność systemów produkcyjnych i jakość wprowadzanych zmian systemowych.

Kultura DevOps to także rozszerzenie zwinnych (ang. *agile*) procesów (scrum, XP itd.), co pozwala skrócić czas dostawy, angażując programistów i zespoły biznesowe. Procesy te są często trudne do przeprowadzenia z powodu niewłączania operacji do tych samych zespołów.

Komunikacja i to połączenie między Dev i Ops umożliwiają lepsze śledzenie kompleksowych wdrożeń produkcyjnych i częstsze wdrożenia o wyższej jakości, co pozwala zaoszczędzić pieniądze dla firmy.

Aby ułatwić tę współpracę i poprawić komunikację między programistami a zespołami operacyjnymi, w procesach należy wprowadzić kilka kluczowych elementów, jak pokazano poniżej:

- Częstsze wdrożenia aplikacji z integracją i ciągłym dostarczaniem (tzw. **CI/CD**).
- Wdrażanie i automatyzacja testów jednostkowych i integracyjnych z procesem skoncentrowanym na **projektowaniu opartym na zachowaniu** (ang. *behavior-driven design* — **BDD**) lub **projektowaniu opartym na testach** (ang. *test-driven design* — **TDD**).
- Wdrożenie sposobu zbierania informacji zwrotnych od użytkowników.
- Monitorowanie aplikacji i infrastruktury.

Ruch DevOps opiera się na trzech założeniach:

- **Kultura współpracy.** To jest istota DevOps — fakt, że zespoły nie są już rozdzielone (jeden zespół programistów, jeden zespół Ops, jeden zespół testerów itd.). Ludzie ci łączą się, tworząc multidyscyplinarne zespoły, które mają ten sam cel: jak najszybsze dostarczenie wartości dodanej do produktu.
- **Procesy.** Aby oczekiwać szybkiego wdrożenia, zespoły te muszą śledzić procesy rozwoju oparte na metodologiach agile z iteracyjnymi fazami, które pozwalają na lepszą funkcjonalność, wyższą jakość i szybszą informację zwrotną. Te procesy powinny być zintegrowane nie tylko z przepływem pracy programistycznej z ciągłą integracją, ale także z przepływem pracy wdrażania z ciągłym dostarczaniem i wdrażaniem. Proces DevOps podzielony jest na kilka faz:
 - A. Planowanie i ustalanie priorytetów funkcjonalności.
 - B. Rozwój.
 - C. Ciągła integracja i dostarczanie.

- D. Ciągłe wdrażanie.
- E. Ciągłe monitorowanie.

Fazy te są przeprowadzane cyklicznie i iteracyjnie przez cały czas trwania projektu.

- **Narzędzia.** Wybór narzędzi i produktów używanych przez zespoły jest bardzo ważny w DevOps. W rzeczywistości, kiedy zespoły zostały podzielone na Dev i Ops, każdy zespół używał swoich specyficznych narzędzi — narzędzi do wdrażania dla programistów i narzędzi infrastruktury dla Ops — co jeszcze bardziej zwiększyło luki komunikacyjne.

W zespołach, które łączą programowanie i operację, oraz w tej kulturze jedności używane narzędzia muszą być użyteczne i możliwe do wykorzystania przez wszystkich członków.

Deweloperzy muszą zapoznać się z narzędziami monitorującymi używanymi przez zespoły Ops w celu jak najwcześniejszego wykrywania problemów z wydajnością i z narzędziami bezpieczeństwa dostarczanych przez Ops w celu ochrony dostępu do różnych zasobów.

Ops z kolei musi zautomatyzować proces tworzenia i aktualizacji infrastruktury oraz zintegrować kod z menedżerem kodu. Wszystkie te czynności tworzą praktyki IaC. Można je jednak wykonać tylko we współpracy z programistami, którzy znają infrastrukturę potrzebną dla aplikacji. Zadania zespołów operacyjnych należy również zintegrować z procesami i narzędziami wydawania aplikacji.

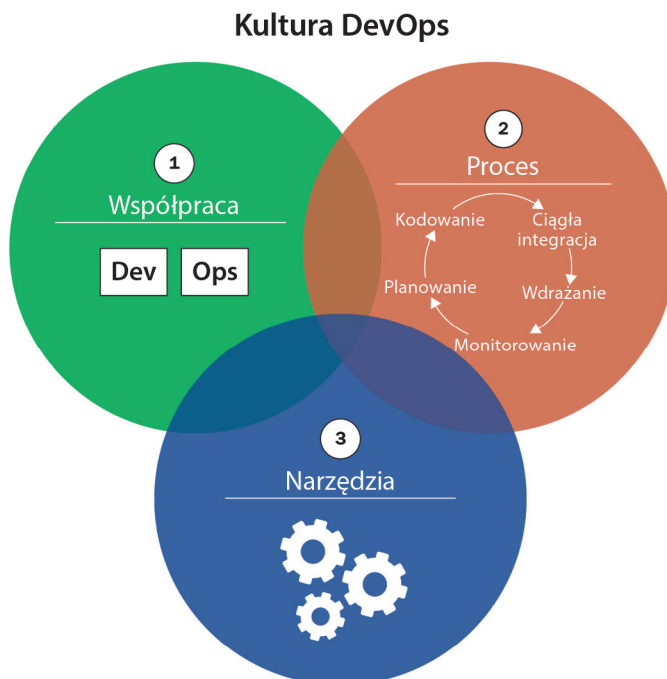
Poniższy diagram ilustruje trzy założenia kultury DevOps — współpracę między Dev i Ops, procesy i wykorzystanie narzędzi.

Możemy nawiązać do kultury DevOps za pomocą definicji Donovan Brown (http://donovanbrown.com/post/what-is-devops):

„DevOps to połączenie ludzi, procesów i produktów, które umożliwia ciągłe dostarczanie wartości naszym użytkownikom końcowym”.

Korzyści z ustanowienia kultury DevOps w przedsiębiorstwie są następujące:

- Lepsza współpraca i komunikacja w zespołach, co ma wpływ na ludzi i na społeczne więzi w firmie.
- Krótsze czasy realizacji produkcji, co skutkuje lepszą wydajnością i satysfakcją użytkownika końcowego.
- Zmniejszone koszty infrastruktury dzięki IaC.



Rysunek 1.1. Kultura DevOps

- Znaczna oszczędność czasu dzięki cyklom iteracyjnym zmniejszającym liczbę błędów aplikacji i narzędziom automatyzacji, które ograniczają liczbę zadań wykonywanych ręcznie, dzięki czemu zespoły skupiają się bardziej na opracowywaniu nowych funkcji o wartości dodanej dla biznesu.

Uwaga

Aby uzyskać więcej informacji na temat kultury DevOps i jej wpływu na transformację przedsiębiorstw, przeczytaj książki *The Phoenix Project: A Novel about IT, DevOps and Helping Your Business Win* Gene'a Kima i Kevina Behra oraz *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations* Gene'a Kima, Jeza Humble'a, Patricka Debois i Johna Willisa.

W tej sekcji poznaliśmy podstawowe pojęcia kultury DevOps. Przyjrzyjmy się teraz pierwszej praktyce kultury DevOps: implementacji CI/CD i ciągłemu wdrażaniu.

Wdrażanie CI/CD i ciągłe wdrażanie

Wcześniej dowiedzieliśmy się, że jedną z kluczowych praktyk DevOps jest proces ciągłej integracji i ciągłego dostarczania, znany również jako **CI/CD**. W rzeczywistości za akronimami CI/CD kryją się trzy praktyki:

- **ciągła integracja (CI),**
- **ciągłe dostarczanie (CD),**
- **ciągłe wdrażanie.**

Czemu odpowiada każda z tych praktyk? Jakie są ich wymagania wstępne i które z nich są najlepsze? Gdzie mają zastosowanie?

Przyjrzyjmy się szczegółowo każdej z tych praktyk, zaczynając od ciągłej integracji.

Ciągła integracja (CI)

W poniższej definicji podanej przez Martina Fowlera wymienia się trzy kluczowe rzeczy: *członkowie zespołu integrują się i to jak najszybciej*:

„Ciągła integracja to praktyka tworzenia oprogramowania, w której członkowie zespołu często integrują swoją pracę... Każda integracja jest weryfikowana przez zautomatyzowaną kompilację (w tym test), aby jak najszybciej wykryć błędy integracji”.

Oznacza to, że CI to automatyczny proces, który pozwala na sprawdzenie kompletności kodu aplikacji za każdym razem, gdy członek zespołu dokona zmiany. Tę weryfikację należy przeprowadzić jak najszybciej.

Kulturę DevOps w CI widzimy bardzo wyraźnie, w duchu współpracy i komunikacji, ponieważ realizacja CI wpływa na wszystkich członków pod względem metodologii pracy, a tym samym współpracy; ponadto CI wymaga implementacji procesów (wprowadzanie zmian, zatwierdzanie, pobieranie i przegląd kodu itd.) wykorzystujących automatyzację za pomocą narzędzi dostosowanych do całego zespołu (Git, Jenkins, Azure DevOps itd.). Wreszcie CI musi działać szybko, aby jak najszybciej zebrać informacje zwrotne na temat integracji kodu, a tym samym być w stanie szybciej dostarczać nowe funkcje użytkownikom.

Wdrażanie CI

Dlatego aby skonfigurować CI, konieczne jest posiadanie **menedżera kodu źródłowego** (ang. *Source Code Manager* — **SCM**), który scentralizuje kod wszystkich członków. Ten menedżer może być dowolnego typu: Git, SVN lub *Team Foundation Version Control* (**TFVC**). Ważne jest również posiadanie automatycznego menedżera kompilacji (serwera CI), który obsługuje ciągłą integrację, takiego jak Jenkins, GitLab CI, TeamCity, Azure Pipelines, GitHub Actions, Travis CI i Circle CI.

Uwaga

W tej książce jako SCM użyjemy Gita i przyjrzymy się nieco dokładniej jego konkretnym zastosowaniom.

Każdy członek zespołu będzie pracował nad kodem aplikacji codziennie, iteracyjnie i przyrostowo (np. w metodach agile i scrum). Każde zadanie lub funkcja musi być oddzielona od innych rozwiązań za pomocą stosowania gałęzi (ang. *branch*).

Regularnie, nawet kilka razy dziennie, członkowie archiwizują lub zatwierdzają (ang. *commit*) swój kod, najlepiej za pomocą małych paczek (ang. *trunks*), które można łatwo naprawić w przypadku błędu. Zostanie on zintegrowany z resztą kodu aplikacji, z resztą zatwierdzeń innych członków.

Integracja wszystkich zatwierdzeń jest punktem wyjścia procesu CI.

Ten proces, który jest wykonywany przez serwer CI, musi być zautomatyzowany i uruchamiany przy każdym zatwierdzeniu. Serwer pobierze kod, a następnie wykona następujące czynności:

- Zbuduje pakiet aplikacji — kompilacja, transformacja plików itd.
- Wykona testy jednostkowe (z testem pokrycia — ang. *code coverage*).

Uwaga

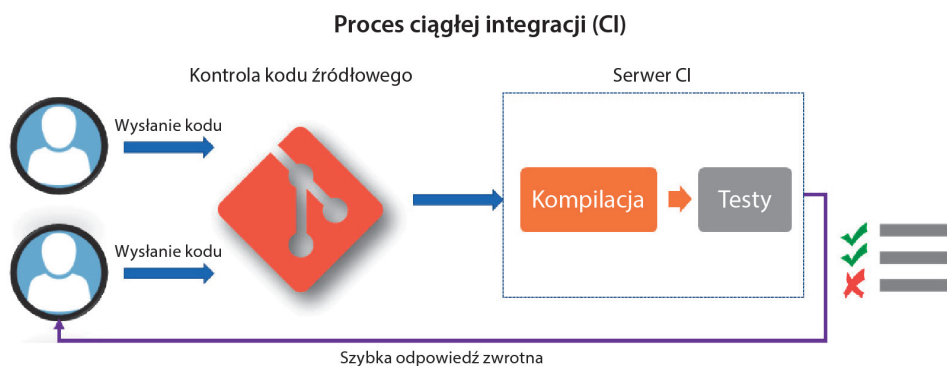
Możliwe jest również wzbogacenie tego procesu o statyczną analizę kodu i podatności. Przyjrzymy się temu w rozdziale 12., „Statyczna analiza kodu za pomocą SonarQube”, który jest poświęcony testom.

Proces CI musi zostać zoptymalizowany najwcześniej, jak to możliwe, aby mógł działać natychmiast, a programiści mogli szybko zebrać informacje zwrotne na temat integracji ich kodu. Na przykład kod, który został zarchiwizowany i nie kompiluje się lub dla którego test kończy się niepowodzeniem, może wpływać na cały zespół i blokować go.

Czasami złe praktyki mogą spowodować niepowodzenie testów podczas CI. Przed dezaktywacją wykonania tego testu musisz wziąć pod uwagę to, że *nie jest ważne, aby koniecznie zadbać o szybkie dostarczenie kodu*.

Wręcz przeciwnie, ta praktyka może mieć poważne konsekwencje, gdy błędy wykryte przez testy zostaną ujawnione w produkcji. Czas zaoszczędzony podczas CI zostanie stracony na naprawianie błędów za pomocą poprawek i ich szybkie ponowne wdrażanie, co może powodować stres. Jest to przeciwieństwo kultury DevOps, ponieważ użytkownicy końcowi otrzymują niską jakość aplikacji i nie ma prawdziwych informacji zwrotnych; zamiast opracowywać nowe funkcje, spędzamy czas na poprawianiu błędów.

Dzięki zoptymalizowanemu i kompletnemu procesowi CI programista może szybko naprawić swoje problemy i ulepszyć kod lub przedyskutować go z resztą zespołu i zatwierdzić swój kod do nowej integracji. Spójrzmy na poniższy diagram:



Rysunek 1.2. Proces ciągłej integracji

Ten diagram przedstawia cykliczne etapy ciągłej integracji. Obejmuje to kod przesyłany do SCM przez członków zespołu oraz kompilację i test wykonywany przez serwer CI. Celem tego procesu jest zapewnienie członkom szybkiej informacji zwrotnej.

Teraz, gdy dowiedzieliśmy się, czym jest ciągła integracja, spójrzmy na ciągłe dostarczanie.

Ciągłe dostarczanie (CD)

Po zakończeniu ciągłej integracji następnym krokiem jest automatyczne wdrożenie aplikacji w co najmniej jednym środowisku nieprodukcyjnym, określanym jako **staging**. Ten proces nazywa się **ciągłym dostarczaniem** (ang. *continuous delivery* — CD).

CD często rozpoczyna pracę z pakietem aplikacji przygotowywanym przez CI, który zostanie zainstalowany na podstawie listy zautomatyzowanych zadań. Mogą to być zadania dowolnego typu: rozpakowanie, zatrzymanie i ponowne uruchomienie usługi, skopiowanie plików, zamiana konfiguracji itd. Podczas procesu CD mogą być również wykonane testy funkcjonalne i akceptacyjne.

W przeciwieństwie do CI, CD ma na celu przetestowanie całej aplikacji ze wszystkimi jej zależnościami. Jest to szczególnie widoczne w aplikacjach mikroserwisowych składających się z kilku usług i interfejsów API; CI przetestuje tylko rozwijaną mikrousługę, podczas gdy po wdrożeniu w środowisku przejściowym będzie można przetestować i zweryfikować całą aplikację, a także interfejsy API i mikrousługi, z których się składa.

W praktyce obecnie bardzo często łączy się CI z CD w środowisku integracyjnym; oznacza to, że CI wdraża się w tym samym czasie w środowisku. Jest to konieczne, aby programiści mogli nie tylko wykonywać testy jednostkowe, ale także weryfikować aplikację jako całość (UI i funkcjonalną) przy każdym zatwierdzeniu, wraz z integracją opracowań innych członków zespołu.

Ważne jest, aby pakiet generowany podczas CI, który zostanie wdrożony również podczas CD, był tym samym, który zostanie zainstalowany we wszystkich środowiskach, i tak powinno być do czasu produkcji. Mogą jednak występować zmiany pliku konfiguracyjnego, które różnią się w zależności od środowiska, ale kod aplikacji (pliki binarne, pliki DLL, obrazy Dockera i JAR) musi pozostać niezmienny.

Niezmienny charakter kodu jest jedyną gwarancją, że aplikacja weryfikowana w środowisku będzie tej samej jakości co wersja, która została wdrożona w poprzednim środowisku, a także ta sama, która zostanie wdrożona w następnym środowisku. Jeśli zmiany (ulepszenia lub poprawki błędów) mają zostać wprowadzone w kodzie po weryfikacji w jednym z tych środowisk, po ich wykonaniu modyfikacje będą musiały ponownie przejść cykl CI i CD.

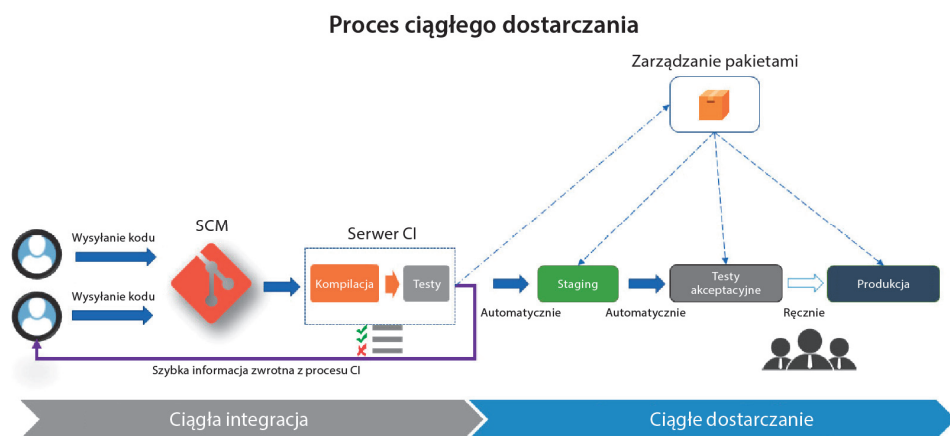
Narzędzia skonfigurowane dla CI/CD są często używane z innymi rozwiązaniami, takimi jak:

- **Menedżer pakietów** — stanowi przestrzeń przechowywania pakietów wygenerowanych przez CI i odzyskanych przez CD. Menedżery te muszą obsługiwać źródła danych, wersjonowanie i różne typy pakietów. Na rynku jest ich kilka, np.: Nexus, ProGet, Artifactory i Azure Artifacts.
- **Menedżer konfiguracji** — umożliwia zarządzanie zmianami konfiguracji podczas procesu CD; większość narzędzi CD zawiera mechanizm konfiguracyjny z systemem zmiennych.

Podczas procesu CD wdrażanie aplikacji w każdym środowisku przejściowym jest wyzwalane w następujący sposób:

- Może być wyzwolone automatycznie po pomyślnym wykonaniu w poprzednim środowisku. Na przykład możemy sobie wyobrazić przypadek, w którym wdrożenie w środowisku przedprodukcyjnym jest uruchamiane automatycznie po pomyślnym przeprowadzeniu testów integracyjnych w środowisku dedykowanym.
- Może być uruchomione ręcznie w przypadku wrażliwych środowisk, takich jak środowisko produkcyjne, po ręcznym zatwierdzeniu przez osobę odpowiedzialną za sprawdzenie poprawności działania aplikacji w środowisku.

W procesie CD ważne jest to, że wdrożenie do środowiska produkcyjnego — czyli dla użytkownika końcowego — jest uruchamiane ręcznie przez zatwierdzonych użytkowników.



Rysunek 1.3. Proces ciągłego dostarczania

Powyższy diagram wyraźnie pokazuje, że proces CD jest kontynuacją procesu CI. Reprezentuje łańcuch kroków CD, które są automatyczne dla środowisk przejściowych i ręczne dla wdrożeń produkcyjnych. Pokazuje również, że pakiet jest generowany przez CI i przechowywany w menedżerze pakietów oraz że jest to ten sam pakiet, który jest wdrażany w różnych środowiskach.

Teraz, gdy przyjrzelśmy się CD, przyjrzyjmy się praktykom ciągłego wdrażania.

Ciągłe wdrażanie

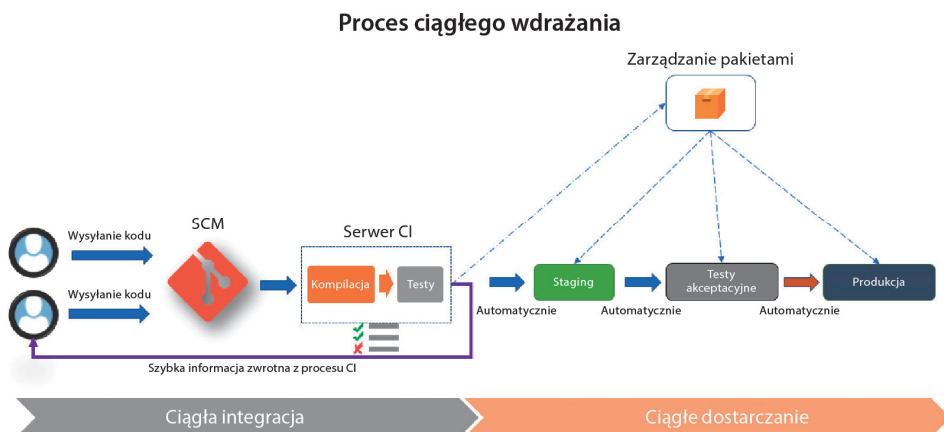
Ciągłe wdrażanie jest rozszerzeniem procesu CD. Automatyzuje cały potok CI/CD od momentu, gdy programista zatwierdzi swój kod, do wdrożenia w środowisku produkcyjnym, przez wszystkie etapy weryfikacji.

Taka praktyka jest rzadko wdrażana w przedsiębiorstwach, ponieważ wymaga wykonania różnych testów (jednostkowych, funkcjonalnych, integracyjnych, wydajnościowych itd.) dla aplikacji. Pomyślne wykonanie tych testów jest wystarczające do sprawdzenia poprawności działania aplikacji pod kątem wszystkich tych zależności. Pozwala również na automatyczne wdrażanie w środowisku produkcyjnym bez konieczności wykonywania jakichkolwiek czynności zatwierdzania.

Ciągły proces wdrażania musi również uwzględniać wszystkie etapy przywracania aplikacji w razie wystąpienia problemu produkcyjnego.

Ciągłe wdrażanie można zastosować przy użyciu i implementacji technik przełączników (ang. *feature toggle*), co obejmuje hermetyzację funkcji aplikacji i aktywowanie danej funkcjonalności na żądanie, bezpośrednio w środowisku produkcyjnym, bez konieczności ponownego wdrażania kodu aplikacji.

Inną techniką jest wykorzystanie niebiesko-zielonej infrastruktury produkcyjnej, która składa się z dwóch środowisk produkcyjnych: jednego niebieskiego i jednego zielonego. Najpierw wdrażamy się do niebieskiego środowiska, a następnie do zielonego; zapewni to, że nie będą wymagane żadne przestoje.



Rysunek 1.4. Proces ciągłego wdrażania

Uwaga

Przełącznikom i niebiesko-zielonemu wykorzystaniu wdrożeń bardziej szczegółowo przyjrzemy się w rozdziale 15., „Skrócenie czasu przestoju wdrażania”.

Powyższy diagram jest prawie taki sam jak w przypadku CD, ale z tą różnicą, że przedstawia automatyczne, kompleksowe wdrożenie.

Procesy CI/CD są zatem istotną częścią kultury DevOps, przy czym CI pozwala zespołom integrować i testować spójność kodu oraz regularnie uzyskiwać szybkie informacje zwrotne. CD automatycznie wdraża się w jednym lub kilku środowiskach staging, dzięki czemu oferuje możliwość przetestowania całej aplikacji, dopóki nie zostanie ona wdrożona w środowisku produkcyjnym.

Wreszcie ciągłe wdrażanie automatyzuje możliwość wdrażania aplikacji od zatwierdzenia do środowiska produkcyjnego.

Uwaga

W rozdziale 7., „Ciągła integracja i ciągłe wdrażanie”, dowiemy się, jak wdrożyć wszystkie te procesy we współpracy z narzędziami Jenkins, Azure DevOps i GitLab CI.

W tej sekcji omówiliśmy praktyki, które są niezbędne dla kultury DevOps, czyli ciągłą integrację, ciągłe dostarczanie i ciągłe wdrażanie.

W następnej sekcji przyjrzymy się innej praktyce DevOps, znanej jako IaC.

Zrozumienie praktyk IaC

IaC to praktyka polegająca na pisaniu kodu zasobów składających się na infrastrukturę.

Praktyka ta zaczęła przynosić efekty wraz z rozwojem kultury DevOps i modernizacją infrastruktury chmury. Rzeczywiście, zespoły operacyjne, które ręcznie wdrażają infrastrukturę, poświęcają czas na wprowadzenie zmian w infrastrukturze ze względu na niespójną obsługę i ryzyko błędów. Ponadto, wraz z modernizacją chmury i jej skalowalnością, sposób budowania infrastruktury wymaga przeglądu wyposażenia (ang. *provisioning*) i praktyki zmian poprzez dostosowanie bardziej zautomatyzowanej metody.

IaC to proces pisania kodu etapów udostępniania i konfiguracji komponentów infrastruktury, który pomaga zautomatyzować jej wdrażanie w powtarzalny i spójny sposób.

Zanim przyjrzymy się korzystaniu z IaC, zobaczymy, jakie są korzyści płynące z tej praktyki.

Korzyści IaC

Korzyści płynące z IaC są następujące:

- Standaryzacja konfiguracji infrastruktury zmniejsza ryzyko błędów.
- Kod opisujący infrastrukturę jest wersjonowany i kontrolowany w menedżerze kodu źródłowego.
- Kod jest zintegrowany z potokami CI/CD.
- Wdrożenia, które wprowadzają zmiany w infrastrukturze, są szybsze i bardziej wydajne.
- Lepsze zarządzanie, kontrola i redukcja kosztów infrastruktury.

IaC przynosi również korzyści zespołowi DevOps, umożliwiając Ops większą wydajność w zakresie zadań związanych z ulepszaniem infrastruktury zamiast poświęcania czasu na ręczną konfigurację. Daje również Dev możliwość ulepszania swojej infrastruktury i wprowadzania zmian bez konieczności proszenia o więcej zasobów operacyjnych.

IaC umożliwia też tworzenie samoobsługowych, efemerycznych środowisk, które zapewnią programistom i testerom większą elastyczność w testowaniu nowych funkcji w izolacji i niezależnie od innych środowisk.

Języki i narzędzia IaC

Języki i narzędzia używane do napisania konfiguracji infrastruktury mogą być różnych typów: **skryptowe**, **deklaratywne** i **programowe**. Zbadamy je w kolejnych sekcjach.

Typy skryptowe

Zaliczają się do nich języki skryptowe, takie jak Bash, PowerShell lub inne, które korzystają z różnych klientów (SDK) dostarczanych przez dostawcę chmury. Na przykład możesz uzyskać dostęp do zasobów platformy Azure za pomocą interfejsu wiersza poleceń platformy Azure lub programu Azure PowerShell.

Oto polecenie, które tworzy grupę zasobów na platformie Azure:

- Korzystając z interfejsu Azure CLI (dokumentacja jest dostępna pod adresem <https://bit.ly/2V1Ofxf>):
az group create --location westeurope --resource-group MyAppResourcegroup
- Korzystając z Azure PowerShell (dokumentacja jest dostępna pod adresem <https://bit.ly/2VcASeh>):
New-AzResourceGroup -Name MyAppResourcegroup -Location westeurope

Problem z tymi językami i narzędziami polega na tym, że wymagają one dużej ilości linii kodu. Dzieje się tak, ponieważ musimy zarządzać różnymi stanami manipulowanych zasobów i konieczne jest napisanie wszystkich kroków tworzenia lub aktualizowania pożądanej infrastruktury.

Jednak te języki i narzędzia mogą być bardzo przydatne w wypadku zadań, które automatyzują powtarzalne czynności wykonywane na liście zasobów (wybór elementu i zapytania) lub które wymagają złożonego przetwarzania z pewną logiką do wykonania na zasobach infrastruktury, np. skrypt automatyzujący maszyny wirtualne, które mają usuwany określony tag.

Typy deklaratywne

Zaliczają się do nich języki, w których wystarczy zapisać pożądany stan systemu lub infrastruktury w postaci konfiguracji i właściwości. Tak jest np. w przypadku narzędzi Terraform i Vagrant firmy HashiCorp, Ansible, szablonu Azure ARM, Azure Bicep (<https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview?tabs=bicep>), PowerShell DSC, Puppet i Chef. Wszystko, co użytkownik musi zrobić, to opisać ostateczny stan pożądanej infrastruktury; narzędzie zadba o jego zastosowanie.

Na przykład poniższy kod Terraform umożliwia zdefiniowanie żądanej konfiguracji grupy zasobów platformy Azure:

```
resource "azurerm_resource_group" "myrg" {
  name = "MyAppResourceGroup"
  location = "West Europe"

  tags = {
    environment = "Bookdemo"
  }
}
```

W tym przykładzie, jeśli chcesz dodać lub zmodyfikować tag, po prostu zmodyfikuj właściwość tags w poprzednim kodzie, a Terraform sam wykona aktualizację.

Oto kolejny przykład, który pozwala zainstalować i uruchomić usługę nginx na serwerze za pomocą Ansible:

```
---
- hosts: all
  tasks:
    - name: install and check nginx latest version
      apt: name=nginx state=latest
    - name: start nginx
      service:
        name: nginx
        state: started
```

Aby się upewnić, że usługa nie jest zainstalowana, po prostu zmień poprzedni kod dla elementu `service` — ustaw wartość `stopped` dla właściwości `state`:

```
---
- hosts: all
  tasks:
    - name: stop nginx
  service:
    name: nginx
    state: stopped
    - name: check nginx is not installed
  apt: name=nginx state=absent
```

W tym przykładzie wystarczyło zmienić właściwość `state`, aby wskazać pożądany stan usługi.

Uwaga

Aby uzyskać szczegółowe informacje na temat korzystania z Terraform i Ansible, zobacz rozdział 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”, i rozdział 3., „Używanie Ansible do konfigurowania infrastruktury IaaS”.

Typy programowe

Od kilku lat obserwuje się, że dwa rodzaje kodu IaC, czyli języki skryptowe lub deklaratywne, używane są przez zespół **operacyjny**. Zwykle nie dotyczy to programistów.

W celu zacieśnienia więzi między programistami a operacjami widzimy pojawianie się narzędzi IaC, które są bardziej oparte na językach znanych programistom, takich jak TypeScript, Java, Python i C#.

Wśród narzędzi IaC, które pozwalają nam udostępniać infrastrukturę za pomocą języka programowania, mamy **Pulumi** (<https://www.pulumi.com/>) i **Terraform CDK** (<https://github.com/hashicorp/terraform-cdk>).

Poniżej znajduje się przykład kodu TypeScriptu napisanego za pomocą Terraform CDK:

```
import { Construct } from 'constructs';
import { App, TerraformStack, TerraformOutput } from 'cdktf';
import {
  ResourceGroup,
} from './.gen/providers/azurerm';
class AzureRgCDK extends TerraformStack {
  constructor(scope: Construct, name: string) {
    super(scope, name);
    new AzurermProvider(this, 'azureFeature', {
      features: [{}],
    });
    const rg = new ResourceGroup(this, 'cdktf-rg', {
```

```
        name: 'MyAppResourceGroup',
        location: 'West Europe',
    });
  }
}
const app = new App();
new AzureRgCDK(app, 'azure-rg-demo');
app.synth();
```

W tym przykładzie, napisanym w języku TypeScript, używamy bibliotek dwuwarstwowych: pakietu npm i Terraform CDK o nazwie `cdktf`. Pakiet npm używany do udostępniania zasobów platformy Azure nosi nazwę `'gen/providers/azurerm'`.

Następnie deklarujemy nową klasę, która inicjuje dostawcę platformy Azure, i definiujemy tworzenie grupy zasobów za pomocą metody `new ResourceGroup`.

Na koniec, aby utworzyć grupę zasobów, tworzymy wystąpienie tej klasy i wywołujemy metodę `app.synth` zestawu CDK.

Uwaga

Aby uzyskać więcej informacji na temat Terraform CDK, proponuję przeczytać następujące wpisy na blogu i obejrzeć następujący film:

<https://www.hashicorp.com/blog/cdk-for-terraform-enabling-python-and-typescript-support>,
<https://www.hashicorp.com/blog/announcing-cdk-for-terraform-0-1>,
<https://www.youtube.com/watch?v=5hSdb0nadRQ>.

Topologia IaC

W infrastrukturze chmurowej IaC dzieli się na kilka typologii:

- wdrażanie i udostępnianie infrastruktury,
- konfiguracja serwera i tworzenie szablonów,
- konteneryzacja,
- konfiguracja i wdrożenie w Kubernetesie.

Przyjrzymy się szerzej każdej z nich.

Wdrażanie i udostępnianie infrastruktury

Udostępnianie to czynność tworzenia instancji zasobów tworzących infrastrukturę. Mogą to być zasoby typu **platforma jako usługa** (ang. *Platform-as-a-Service* — PaaS) i bezserwerowe, takie jak aplikacja internetowa, funkcja platformy Azure, Event Hub, ale także cała zarządzana część sieci, taka jak sieć wirtualna, podsieci, tabele routingu

lub Azure Firewall. W przypadku zasobów maszyn wirtualnych proces udostępniania tworzy lub aktualizuje tylko zasób w chmurze maszyny wirtualnej, ale nie jego zawartość.

Dostępnych jest wiele narzędzi do udostępniania, takich jak Terraform, szablony ARM, Azure CLI, Azure PowerShell, a także Google Cloud Deployment Manager. Oczywiście jest ich znacznie więcej, ale trudno je wszystkie wymienić. W tej książce przyjrzymy się szczegółowo wykorzystaniu Terraform.

Konfiguracja serwera

Ten krok dotyczy konfigurowania maszyn wirtualnych, takich jak utwardzenie (ang. *hardening*), montowanie dysków, konfiguracja sieci (zaporę ogniową, proxy itd.) oraz instalacja middleware.

Istnieją różne narzędzia konfiguracyjne, takie jak Ansible, PowerShell DSC, Chef, Puppet i SaltStack. Oczywiście jest ich znacznie więcej, ale w tej książce szczegółowo przyjrzymy się wykorzystaniu Ansible w celu konfiguracji maszyny wirtualnej.

Aby zoptymalizować czas udostępniania i konfiguracji serwera, można tworzyć modele serwerów, zwanych obrazami, i ich używać. Zawierają one całą konfigurację (hardening, middleware itd.) serwerów. Podczas przygotowywania serwera wskażemy odpowiadający mu szablon. Tak więc za kilka minut będziemy mieli serwer, który został skonfigurowany i jest gotowy do użycia.

Istnieje również wiele narzędzi IaC do tworzenia szablonów serwerów, np. **Aminator** (używany przez Netflix) i **HashiCorp Packer**.

Oto przykład kodu Packera służącego do tworzenia obrazu Ubuntu z aktualizacjami pakietów:

```
{
  "builders": [{
    "type": "azure-arm",
    "os_type": "Linux",
    "image_publisher": "Canonical",
    "image_offer": "UbuntuServer",
    "image_sku": "16.04-LTS",
    "managed_image_resource_group_name": "demoBook",
    "managed_image_name": "SampleUbuntuImage",
    "location": "West Europe",
    "vm_size": "Standard_DS2_v2"
  }],
  "provisioners": [{
    "execute_command": "chmod +x {{ .Path }}; {{ .Vars }} sudo -E sh  
↳ '{{ .Path }}'",
```

```
    "inline": [
      "apt-get update",
      "apt-get upgrade -y",
      "/usr/sbin/waagent -force -deprovision+user && export
      HISTSIZE=0 && sync"
    ],
    "inline_shebang": "/bin/sh -x",
    "type": "shell"
  }]
}
```

Ten skrypt tworzy obraz szablonu dla maszyny wirtualnej Standard_DS2_V2 na podstawie systemu operacyjnego Ubuntu (sekcja `builders`). Dodatkowo podczas tworzenia obrazu Packer aktualizuje wszystkie pakiety za pomocą polecenia `apt-get update`.

Następnie Packer wyrejestruje obraz, aby usunąć wszystkie informacje o użytkowniku (sekcja `provisioners`).

Uwaga

Packer zostanie szczegółowo omówiony w rozdziale 4., „Optymalizacja wdrażania infrastruktury za pomocą Packera”.

Niezmienna infrastruktura z kontenerami

Konteneryzacja polega na wdrażaniu aplikacji w kontenerach zamiast na maszynach wirtualnych.

Obecnie najpowszechniejszym oprogramowaniem służącym do konteneryzacji jest **Docker**. Obraz Dockera jest konfigurowany za pomocą kodu w pliku *Dockerfile*. Plik ten zawiera deklarację obrazu bazowego, który reprezentuje używany system operacyjny, dodatkowe oprogramowanie middleware, pliki i binaria niezbędne dla aplikacji oraz konfigurację sieciową portów. W przeciwieństwie do maszyn wirtualnych, kontenery są uważane za niezmiennie; konfiguracja kontenera nie może być modyfikowana podczas jego wykonywania.

Oto prosty przykład pliku *Dockerfile*:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y nginx
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]
EXPOSE 80
```

W tym obrazie Dockera używamy podstawowego obrazu Ubuntu, instalujemy `nginx` i udostępniamy port 80.

Uwaga

Część dotycząca platformy Docker zostanie szczegółowo omówiona w rozdziale 9., „Konteneryzacja aplikacji za pomocą Dockera”.

Konfiguracja i wdrożenie w Kubernetesie

Kubernetes jest orkiestratorem kontenerów — to technologia, która w większości uosabia IaC (moim zdaniem) ze względu na sposób wdrażania kontenerów, architekturę sieci (równoważenie obciążenia, porty itd.) i zarządzanie woluminami. Chroni także poufne informacje, które są zapisane w plikach specyfikacji YAML.

Oto prosty przykład pliku specyfikacji YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-demo
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.7.9
    ports:
      - containerPort: 80
```

W poprzednim pliku specyfikacji możemy zobaczyć nazwę obrazu do wdrożenia (nginx), port do otwarcia (80) i liczbę replik (2).

Uwaga

Część dotycząca Kubernetesa zostanie szczegółowo omówiona w rozdziale 10., „Efektywne zarządzanie kontenerami za pomocą Kubernetesa”.

IaC, podobnie jak tworzenie oprogramowania, wymaga od nas wdrożenia praktyk i procesów, które umożliwiają ewolucję i utrzymanie kodu infrastruktury.

Wśród tych praktyk są te związane z tworzeniem oprogramowania:

- Stosuj dobre zasady nazewnictwa.
- Nie przeciążaj kodu niepotrzebnymi komentarzami.
- Używaj małych funkcji.
- Zaimplementuj obsługę błędów.

Uwaga

Aby dowiedzieć się więcej o dobrych praktykach tworzenia oprogramowania, przeczytaj doskonałą książkę związaną z tym tematem, *Clean Code* Roberta Martina.

Są jeszcze inne konkretne praktyki, które moim zdaniem zasługują na więcej uwagi:

- **Kod musi być w pełni zautomatyzowany.** Podczas wykonywania IaC należy zakodować i zautomatyzować wszystkie etapy udostępniania i nie należy pozostawiać ręcznych działań poza kodem, które zakłócają automatyzację infrastruktury — co może generować błędy. A jeśli to konieczne, nie wahać się użyć kilku narzędzi, takich jak Terraform i Bash, ze skryptami Azure CLI.
- **Kod musi się znajdować w menedżerze kodu źródłowego.** Kod infrastruktury musi również znajdować się w SCM, aby można go było wersjonować, śledzić, scalać i przywracać, a tym samym zapewniać lepszą widoczność kodu między programistami i zespołem operacyjnym.
- **Kod infrastruktury musi być trzymany z kodem aplikacji.** W niektórych przypadkach może to być trudne, ale jeśli to możliwe, znacznie lepiej jest umieścić kod infrastruktury w tym samym repozytorium co kod aplikacji. Ma to na celu zapewnienie lepszej organizacji pracy między programistami i operatorami, którzy będą dzielić ten sam obszar roboczy.
- **Oddzielenie ról i katalogów.** Dobrze jest oddzielić kod od infrastruktury — zgodnie z rolą kodu. Umożliwia to utworzenie jednego katalogu do udostępniania i konfigurowania maszyn wirtualnych i innego, który będzie zawierał kod do testowania integracji całej infrastruktury.
- **Integracja z procesem CI/CD.** Jednym z celów IaC jest możliwość automatyzacji wdrażania infrastruktury. Tak więc od początku jego implementacji konieczne jest utworzenie procesu CI/CD, który zintegruje kod, przetestuje go i wdroży w różnych środowiskach. Niektóre narzędzia, takie jak Terratest, umożliwiają tworzenie testów dla kodu infrastruktury. Jedną z najlepszych praktyk jest integracja procesu CI/CD infrastruktury w tym samym potoku co aplikacja.
- **Kod musi być idempotentny.** Wykonanie kodu wdrażania infrastruktury musi być idempotentne — czyli kod powinien być ciągle automatycznie wykonywalny. Oznacza to, że skrypty muszą uwzględniać stan infrastruktury

podczas jej uruchamiania i nie generować błędów, jeśli zasób do utworzenia już istnieje lub zasób do usunięcia został już usunięty. Zobaczymy, że języki deklaratywne, takie jak Terraform, natywnie przyjmują ten aspekt idempotencji. Kod infrastruktury raz w pełni zautomatyzowany musi umożliwiać budowę i niszczenie infrastruktury aplikacji.

- **Kod jako dokumentacja.** Kod infrastruktury musi być jasny i musi służyć jako dokumentacja. Dokumentacja infrastruktury zajmuje dużo czasu i w wielu przypadkach nie jest aktualizowana w miarę rozwoju infrastruktury.
- **Kod musi być modułowy.** W infrastrukturze komponenty często mają ten sam kod — jedyną różnicą jest wartość ich właściwości. Również te komponenty są kilkakrotnie wykorzystywane w aplikacjach firmy. Dlatego ważne jest, aby zoptymalizować czasy pisania kodu przez podzielenie go na moduły (lub role w przypadku Ansible), które będą wywoływane jako funkcje. Kolejną zaletą korzystania z modułów jest możliwość standaryzacji nazewnictwa zasobów i zgodność niektórych właściwości.
- **Posiadanie środowiska programistycznego.** Problem z IaC polega na tym, że trudno jest testować kod infrastruktury będącej w fazie rozwoju w środowiskach używanych do integracji, a także testować aplikację, ponieważ zmiana infrastruktury może wpływać na kod. Dlatego ważne jest, aby mieć środowisko programistyczne nawet dla IaC, które może w każdej chwili ulec uszkodzeniu lub nawet zostać zniszczone.

W przypadku testów infrastruktury lokalnej niektóre narzędzia symulują środowisko lokalne, np. Vagrant (od HashiCorp), więc należy ich używać do testowania skryptów kodu tak często, jak to możliwe.

Oczywiście pełna lista dobrych praktyk jest dłuższa; również wszystkie metody i procesy praktyk inżynierii oprogramowania mają zastosowanie.

Dlatego IaC, podobnie jak procesy CI/CD, jest kluczową praktyką **kultury DevOps**, która umożliwia wdrażanie i konfigurowanie infrastruktury poprzez pisanie kodu. Jednak IaC może być skuteczna tylko przy użyciu odpowiednich narzędzi i wdrażaniu dobrych praktyk.

W tej sekcji zawarliśmy omówienie niektórych najlepszych praktyk DevOps. Teraz przedstawimy krótki przegląd ewolucji kultury DevOps.

Ewolucja kultury DevOps

Z czasem i doświadczeniem zdobytym przy stosowaniu kultury DevOps możemy zaobserwować ewolucję praktyk, a także zespołów pragnących dołączyć do tego ruchu.

Tak jest np. w przypadku praktyki **GitOps**, która coraz częściej zaczyna się pojawiać w firmach.

Przepływ pracy GitOps, który jest powszechnie stosowany w Kubernetesie, polega na użyciu Gita jako jedyne źródła prawdy; oznacza to, że repozytorium Gita zawiera kod stanu infrastruktury, a także kod aplikacji do wdrożenia.

Kontroler będzie nadzorował pobieranie źródła Gita podczas zatwierdzania kodu, wykonywania testów i ponownego wdrażania aplikacji.

Uwaga

Aby uzyskać więcej informacji o kulturze, praktykach i przepływach pracy GitOps, przeczytaj oficjalny przewodnik po GitOps: <https://www.weave.works/technologies/gitops/>.

Podsumowanie

W tym rozdziale zobaczyliśmy, że kultura DevOps to złożenie współpracy, procesów i narzędzi. Następnie szczegółowo opisaliśmy poszczególne etapy procesu CI/CD i wyjaśniliśmy różnicę między ciągłą integracją, ciągłym dostarczaniem i ciągłym wdrażaniem. W ostatniej części wyjaśniliśmy, jak korzystać z IaC z jej najlepszymi praktykami, a także omówiliśmy ewolucję kultury DevOps.

Poznaliśmy podstawy kultury DevOps i jej praktyk, co nadaje ton pozostałym rozdziałom tej książki, w których omówimy, jak zastosować tę kulturę za pomocą narzędzi i praktyk.

W następnym rozdziale zaczniemy od omówienia implementacji procesu *infrastruktury jako kodu* i sposobu dostarczania infrastruktury za pomocą Terraform.

Pytania

1. Od jakich słów pochodzi skrót DevOps?
2. Czy DevOps to termin, który reprezentuje nazwę narzędzia, kulturę, społeczeństwo czy tytuł książki?
3. Jakie są trzy założenia kultury DevOps?
4. Jaki jest cel ciągłej integracji?
5. Jaka jest różnica między ciągłym dostarczaniem a ciągłym wdrażaniem?
6. Co oznacza IaC?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o kulturze DevOps, oto kilka zasobów:

- Centrum zasobów DevOps (zasoby firmy Microsoft) — <https://docs.microsoft.com/en-us/azure/devops/learn/>.
- Raport na temat DevOps z 2020 r. (zasoby Puppeta) — <https://puppet.com/resources/report/2020-state-of-devops-report>.

Udostępnianie infrastruktury chmury za pomocą Terraform

Rozdział

2

W poprzednim rozdziale przedstawiliśmy narzędzia, praktyki i zalety procesu *infrastruktura jako kod* (IaC) oraz jej wpływ na kulturę DevOps. Spośród wszystkich wymienionych narzędzi IaC szczególnie popularnym i potężnym jest **Terraform**, który stanowi część pakietu narzędzi HashiCorp.

W tym rozdziale omówimy podstawy korzystania z Terraform do udostępniania infrastruktury chmury na przykładzie platformy Azure. Zaczniemy od przeglądu jego mocnych stron w porównaniu z innymi narzędziami IaC. Dowiemy się, jak zainstalować go zarówno w trybie ręcznym, jak i automatycznym, a następnie utworzymy nasz pierwszy skrypt Terraform do udostępniania infrastruktury Azure z wykorzystaniem najlepszych praktyk i automatyzacji Terraform w trybie *ciągłej integracji* (CI)/*ciągłego wdrażania* (CD). Na koniec zajmiemy się implementacją zdalnego zaplecza (ang. *remote backend*) dla pliku stanu Terraform.

W tym rozdziale omówimy następujące tematy:

- instalacja Terraform,
- konfiguracja Terraform dla platformy Azure,
- napisanie skryptu Terraform w celu wdrożenia infrastruktury platformy Azure,
- uruchamianie Terraform do wdrożenia,
- zrozumienie cyklu życia Terraform z różnymi opcjami wiersza poleceń,
- ochrona pliku stanu za pomocą zdalnego zaplecza.

Wymagania techniczne

W tym rozdziale wyjaśniono, w jaki sposób można użyć Terraform do udostępnienia infrastruktury platformy Azure jako przykładu infrastruktury chmury. Dlatego będziesz potrzebować subskrypcji platformy Azure, którą możesz pobrać bezpłatnie pod adresem <https://azure.microsoft.com/en-us/free/>.

Ponadto do pisania kodu Terraform będziemy potrzebować edytora kodu. Istnieje kilka edytorów, ale będę używał **Visual Studio Code**. Jest darmowy, lekki, wieloplatformowy i ma kilka rozszerzeń dla Terraform. Możesz go pobrać pod adresem <https://code.visualstudio.com/>. Pełny kod źródłowy tego rozdziału jest dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP02>.

Obejrzyj następujący film z kanału Code in Action: <https://bit.ly/3p7x63h>.

Instalacja Terraform

Terraform to narzędzie wiersza poleceń, które w swojej podstawowej wersji jest otwartoźródłowe, używa **języka konfiguracji HashiCorp** (ang. *HashiCorp Configuration Language* — **HCL**), jest deklaratywny i stosunkowo łatwy do zrozumienia. Jego główną zaletą jest użycie tego samego języka do wdrożenia u wielu dostawców chmury, takich jak Azure, AWS i Google — pełna lista jest dostępna pod adresem <https://www.terraform.io/docs/providers/>.

Terraform ma również inne zalety:

- Jest wieloplatformowy i można go zainstalować w systemach Windows, Linux i macOS.
- Umożliwia podgląd zmian w infrastrukturze przed ich wdrożeniem.
- Umożliwia zrównoleglenie operacji poprzez uwzględnienie zależności zasobów.
- Integruje bardzo dużą liczbę dostawców.

Terraform można zainstalować w systemie na wiele sposobów. Zacznijmy od przyjrzenia się ręcznej metodzie instalacji.

Instalacja ręczna

Aby ręcznie zainstalować Terraform, wykonaj następujące czynności:

1. Przejdź do oficjalnej strony pobierania pod adresem <https://www.terraform.io/downloads.html>. Następnie pobierz pakiet odpowiadający Twojemu systemowi operacyjnemu.
2. Po pobraniu rozpakuj i skopiuj plik binarny do katalogu wykonawczego (np. do katalogu `c:\Terraform`).
3. Następnie zmienna środowiskowa PATH musi być uzupełniona ścieżką do katalogu binarnego. Aby uzyskać szczegółowe instrukcje, obejrzyj wideo pod adresem <https://learn.hashicorp.com/tutorials/terraform/install-cli>.

Teraz, gdy nauczyliśmy się ręcznie instalować Terraform, przyjrzyjmy się dostępnym opcjom instalacji za pomocą skryptu.

Instalacja za pomocą skryptu

Instalacja za pomocą skryptu automatyzuje instalację lub aktualizację Terraform na serwerze zdalnym, który będzie odpowiedzialny za wykonywanie kodu Terraform, np. na podrzędnym serwerze Jenkins lub agencie Azure Pipelines.

Instalowanie Terraform za pomocą skryptu w systemie Linux

Aby zainstalować plik binarny Terraform w systemie Linux, mamy dwa rozwiązania. Pierwszym rozwiązaniem jest instalacja Terraform za pomocą następującego skryptu:

```
TERRAFORM_VERSION="1.0.0" #Zaktualizuj do żądanej wersji
curl -Os https://releases.hashicorp.com/terraform/${TERRAFORM_VERSION}/
↳terraform_${TERRAFORM_VERSION}_linux_amd64.zip \
&& curl -Os https://releases.hashicorp.com/terraform/${TERRAFORM_VERSION}/
↳terraform_${TERRAFORM_VERSION}_SHA256SUMS \
&& curl https://keybase.io/hashicorp/pgp_keys.asc | gpg
--import \
&& curl -Os https://releases.hashicorp.com/terraform/${TERRAFORM_VERSION}/
↳terraform_${TERRAFORM_VERSION}_SHA256SUMS.sig \
&& gpg --verify terraform_${TERRAFORM_VERSION}_SHA256SUMS.sig terraform_
↳${TERRAFORM_VERSION}_SHA256SUMS \
&& shasum -a 256 -c terraform_${TERRAFORM_VERSION}_SHA256SUMS 2>&1 |
↳grep "${TERRAFORM_VERSION}_linux_amd64.zip:\sOK" \
&& unzip -o terraform_${TERRAFORM_VERSION}_linux_amd64.zip -d /usr/local/bin
```

Ten skrypt wykonuje następujące czynności:

- Nadaje parametrowi `TERRAFORM_VERSION` wartość wersji do pobrania.
- Pobiera pakiet Terraform, sprawdzając sumę kontrolną.
- Rozpakowuje pakiet w lokalnym katalogu użytkownika.

Ważna uwaga

Ten skrypt jest również dostępny w źródle GitHuba dla tej książki pod adresem https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP02/Terraform_install_Linux.sh.

Aby wykonać ten skrypt, wykonaj następujące kroki:

1. Otwórz terminal.
2. Skopiuj i wklej poprzedni skrypt.
3. Wykonaj go, naciskając *Enter* w terminalu.

Poniższy zrzut ekranu przedstawia wykonanie skryptu instalującego Terraform w systemie Linux:

```
root@ubuntu-bionic:/learningdevops/CHAP02# sh Terraform_install_Linux.sh
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           % Done    0      0     0    20523      0 --:--:-- --:--:-- --:--:-- 20469
gpg: key 34365D9472D7468F: "HashiCorp Security (hashicorp.com/security) <security@hashicorp.com>" not changed
gpg: Total number processed: 1
gpg:      unchanged: 1
gpg: Signature made Tue Jun  8 11:21:42 2021 UTC
gpg:    using RSA key B36CBA91A2C0730C435FC280B0B441097685B676
gpg: Good signature from "HashiCorp Security (hashicorp.com/security) <security@hashicorp.com>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:       There is no indication that the signature belongs to the owner.
Primary key fingerprint: C874 011F 0AB4 0511 0D02  1055 3436 5D94 72D7 468F
Subkey fingerprint: B36C BA91 A2C0 730C 435F  C280 B0B4 4109 7685 B676
terraform_1.0.0_linux_amd64.zip: OK
Archive:  terraform_1.0.0_linux_amd64.zip
inflating: /usr/local/bin/terraform
```

Rysunek 2.1. Skrypt instalacyjny Terraform w systemie Linux

Podczas wykonywania powyższego skryptu możemy zobaczyć pobranie pakietu Terraform ZIP (za pomocą narzędzia `curl`) i operację rozpakowania tego pakietu w folderze `/usr/local/bin`.

Zaletą tego rozwiązania jest to, że możemy wybrać folder instalacyjny Terraform i skrypt może być stosowany w różnych dystrybucjach Linuksa. Dzieje się tak, ponieważ używa typowych narzędzi, w tym `curl` i `unzip`.

Drugim rozwiązaniem instalacji Terraform w systemie Linux jest użycie menedżera pakietów `apt`. Możesz to zrobić za pomocą następującego skryptu dla dystrybucji Ubuntu:


```
sudo apt-get update && sudo apt-get install -y gnupg
↳softwareproperties-common curl \
&& curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add - \
&& sudo apt-add-repository "deb [arch=amd64] https://apt.releases.
↳hashicorp.com $(lsb_release -cs) main" \
&& sudo apt-get update && sudo apt-get install terraform
```

Ten skrypt wykonuje następujące czynności:

- Dodaje repozytorium apt HashiCorp.
- Aktualizuje lokalne repozytorium.
- Pobiera interfejs Terraform CLI.

Ważna uwaga

Aby uzyskać dodatkowe informacje na temat tego skryptu i instalacji Terraform w innych dystrybucjach, zapoznaj się z dokumentacją pod adresem <https://learn.hashicorp.com/tutorials/terraform/install-cli> i w zakładce *Linux*.

Zaletą tego rozwiązania jest możliwość użycia menedżera pakietów Linux. Można je zintegrować z popularnymi narzędziami konfiguracyjnymi (np.: Ansible, Puppet lub Docker).

Właśnie omówiliśmy instalację Terraform w systemie Linux. Przyjrzyjmy się teraz jego instalacji w systemie Windows.

Instalowanie Terraform za pomocą skryptu w systemie Windows

Jeśli korzystamy z systemu Windows, możemy użyć **Chocolatey**, który jest darmowym publicznym menedżerem pakietów, takim jak **NuGet** czy **npm**, ale dedykowanym do oprogramowania. Jest szeroko stosowany do automatyzacji oprogramowania na serwerach Windowsa lub nawet na komputerach lokalnych.

Ważna uwaga

Oficjalna strona Chocolatey znajduje się pod adresem <https://chocolatey.org/>, a dokumentacja dotycząca jego instalacji znajduje się pod adresem <https://chocolatey.org/install>.

Po zainstalowaniu Chocolatey wystarczy uruchomić następujące polecenie w PowerShell lub narzędziu CMD:

```
choco install terraform -y
```

Poniżej znajduje się zrzut ekranu instalacji Terraform dla Windowsa za pomocą Chocolatey:

```
PS C:\WINDOWS\system32> choco install terraform -y
chocolatey v0.10.15
Installing the following packages:
terraform
By installing you accept licenses for the packages.
Progress: Downloading terraform 1.0.0... 100%

terraform v1.0.0 [Approved]
terraform package files install completed. Performing other installation steps.
Removing old terraform plugins
Downloading terraform 64 bit
  from 'https://releases.hashicorp.com/terraform/1.0.0/terraform_1.0.0_windows_amd64.zip'
Progress: 100% - Completed download of C:\Users\mika\AppData\Local\Temp\chocolatey\terraform\1.0.0\terraform_1.0.0_windows_amd64.zip (31.79 MB).
Download of terraform_1.0.0_windows_amd64.zip (31.79 MB) completed.
Hashes match.
Extracting C:\Users\mika\AppData\Local\Temp\chocolatey\terraform\1.0.0\terraform_1.0.0_windows_amd64.zip to C:\ProgramData\chocolatey\lib\terraform\tools...
C:\ProgramData\chocolatey\lib\terraform\tools
ShimGen has successfully created a shim for terraform.exe
The install of terraform was successful.
Software installed to 'C:\ProgramData\chocolatey\lib\terraform\tools'

Chocolatey installed 1/1 packages.
See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).
```

Rysunek 2.2. Instalacja Terraform w systemie Windows

Wykonanie polecenia `choco install terraform` powoduje zainstalowanie najnowszej wersji Terraform firmy Chocolatey.

Po zainstalowaniu możemy sprawdzić wersję Terraform, uruchamiając następujące polecenie:

```
terraform version
```

To polecenie wyświetla numer wersji zainstalowanego Terraform.

Możemy również sprawdzić różne polecenia oferowane przez Terraform, uruchamiając następujące polecenie:

```
terraform --help
```

Poniższy zrzut ekranu przedstawia różne polecenia i ich funkcje.

Przyjrzyjmy się teraz instalacji Terraform w systemie macOS.

Instalowanie Terraform za pomocą skryptu w systemie macOS

W systemie macOS, aby zainstalować Terraform, możemy użyć **Homebrew**, menedżera pakietów macOS (<https://brew.sh/>). Wykonaj następujące polecenie w terminalu:

```
brew install terraform
```

To wszystko, jeśli chodzi o instalowanie Terraform za pomocą skryptu. Rzućmy okiem na inne rozwiązanie, które używa Terraform na platformie Azure bez konieczności jego instalowania — **Azure Cloud Shell**.

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

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
  apply          Builds or changes infrastructure
  console        Interactive console for Terraform interpolations
  destroy        Destroy Terraform-managed infrastructure
  env            Workspace management
  fmt            Rewrites config files to canonical format
  get            Download and install modules for the configuration
  graph          Create a visual graph of Terraform resources
  import         Import existing infrastructure into Terraform
  init           Initialize a Terraform working directory
  output         Read an output from a state file
  plan           Generate and show an execution plan
  providers      Prints a tree of the providers used in the configuration
  push           Upload this Terraform module to Atlas to run
  refresh        Update local state file against real resources
  show           Inspect Terraform state or plan
  taint         Manually mark a resource for recreation
  untaint        Manually unmark a resource as tainted
  validate       Validates the Terraform files
  version        Prints the Terraform version
  workspace      Workspace management

All other commands:
  debug          Debug output management (experimental)
  force-unlock   Manually unlock the terraform state
  state          Advanced state management
```

Rysunek 2.3. Dostępne polecenia Terraform

Integracja Terraform z Azure Cloud Shell

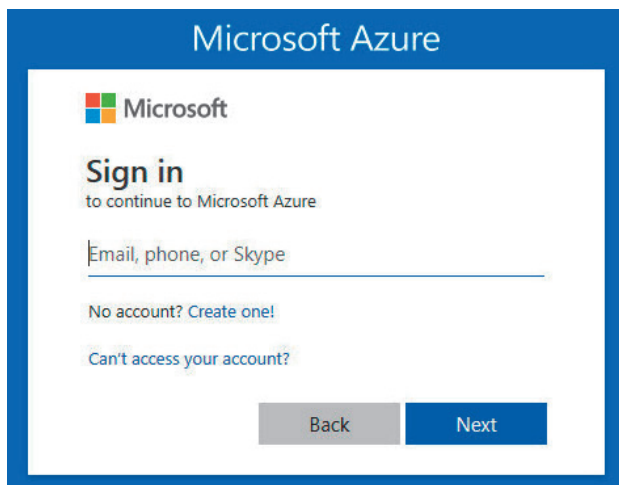
Jeśli używamy Terraform do wdrożenia fragmentu infrastruktury na platformie Azure, powinniśmy również wiedzieć, że zespół platformy Azure zintegrował Terraform z usługą Azure Cloud Shell.

Ważna uwaga

Aby dowiedzieć się więcej o Azure Cloud Shell, zapoznaj się z jego dokumentacją pod adresem <https://azure.microsoft.com/en-us/features/cloud-shell/>.

Aby użyć go z Azure Cloud Shell, wykonaj następujące kroki:

1. Połącz się z portalem Azure, otwierając stronę <https://portal.azure.com>, i zaloguj się przy użyciu swojego konta platformy Azure:

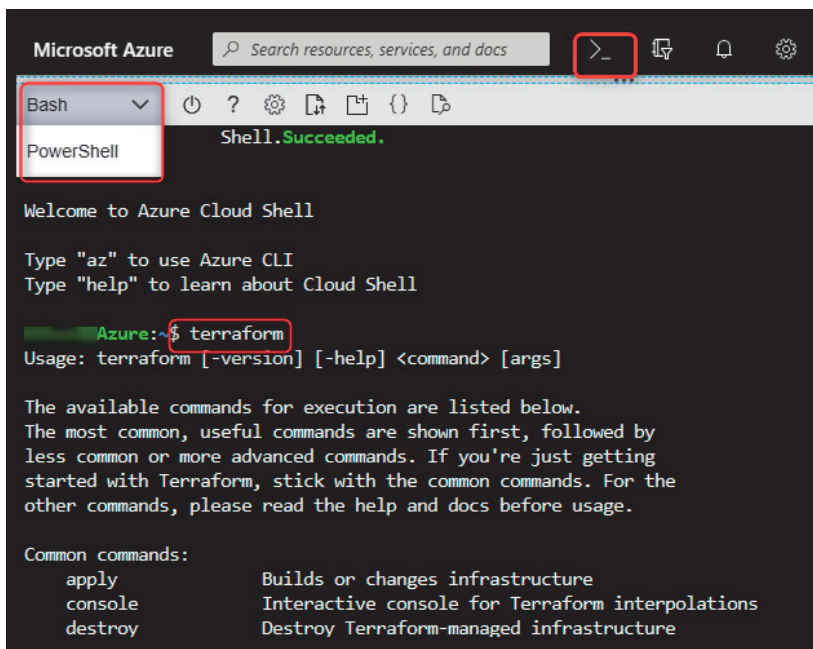


Rysunek 2.4. Strona logowania Azure

2. Otwórz Cloud Shell i wybierz żądany tryb, czyli *Bash* lub *PowerShell*.

3. Następnie w powłoce możesz uruchomić wiersz poleceń Terraform.

Poniżej znajduje się zrzut ekranu wykonania Terraform w Azure Cloud Shell:



Rysunek 2.5. Azure Cloud Shell

Zaletą korzystania z tego rozwiązania jest to, że nie potrzebujemy żadnego oprogramowania do instalacji; możemy po prostu przesłać pliki Terraform do Cloud Shell i uruchomić je w Cloud Shell. Dodatkowo jesteśmy już połączeni z platformą Azure, więc nie jest wymagana konfiguracja (zapoznaj się z sekcją „Konfigurowanie Terraform dla platformy Azure”).

Jednak to rozwiązanie może być używane tylko w trybie programowania, a nie do lokalnego lub automatycznego korzystania z Terraform. Dlatego w tym rozdziale omówimy konfigurację Terraform dla platformy Azure.

Teraz, gdy zainstalowaliśmy Terraform, możemy zacząć używać go lokalnie w celu udostępnienia infrastruktury platformy Azure. Zaczniemy od pierwszego kroku, czyli skonfigurowania Terraform dla Azure.

Konfigurowanie Terraform dla platformy Azure

Przed napisaniem kodu Terraform, w którym należy udostępnić infrastrukturę chmury, taką jak platforma Azure, musimy skonfigurować Terraform, by umożliwić manipulowanie zasobami w ramach subskrypcji platformy Azure.

Aby to zrobić, najpierw utworzymy nową jednostkę Azure (ang. *service principal* — **SP**) w usłudze Azure **Active Directory (AD)**, która to jednostka na platformie Azure jest *użytkownikiem aplikacji* mającym uprawnienia do zarządzania zasobami platformy Azure.

Ważna uwaga

Aby uzyskać więcej informacji na temat jednostki Azure SP, zapoznaj się z dokumentacją pod adresem <https://docs.microsoft.com/en-us/azure/active-directory/develop/app-objects-and-service-principals>.

W przypadku tej jednostki SP platformy Azure musimy przypisać do niej uprawnienia do współtworzenia subskrypcji, w której będziemy tworzyć zasoby.

Tworzenie jednostki usługi Azure SP

Operację tę można wykonać za pośrednictwem portalu Azure (wszystkie kroki są szczegółowo opisane w oficjalnej dokumentacji pod adresem <https://docs.microsoft.com/en-us/azure/active-directory/develop/howto-create-service-principal-portal>) lub za

pomocą skryptu, wykonując polecenie `az cli` (które możemy uruchomić w Azure Cloud Shell).

Poniżej znajduje się szablon skryptu `az cli`, który musisz uruchomić, aby utworzyć jednostkę SP. Tutaj musisz wpisać swoją nazwę SP, rolę i zakres:

```
az ad sp create-for-rbac --name="<nazwa SP>" --role="Contributor"
↳--scopes="/subscriptions/<ID subskrypcji>"
```

Spójrz na następujący przykład:

```
az ad sp create-for-rbac --name="SPForTerraform" --role="Contributor"
↳--scopes="/subscriptions/8921-1444-..."
```

Ten skrypt tworzy nową jednostkę SP o nazwie `SPForTerraform` i nadaje jej uprawnienia na współautora do identyfikatora subskrypcji, czyli `8921...`

Ważna uwaga

Aby uzyskać więcej informacji na temat polecenia Azure CLI i tworzenia jednostki Azure SP, zapoznaj się z dokumentacją pod adresem <https://docs.microsoft.com/en-us/cli/azure/create-an-azure-service-principal-azure-cli?view=Azure-cli-latest>.

Poniższy zrzut ekranu pokazuje wykonanie skryptu, który tworzy jednostkę usługi Azure SP:

```
Bash
Type "az" to use Azure CLI 2.0
Type "help" to learn about Cloud Shell

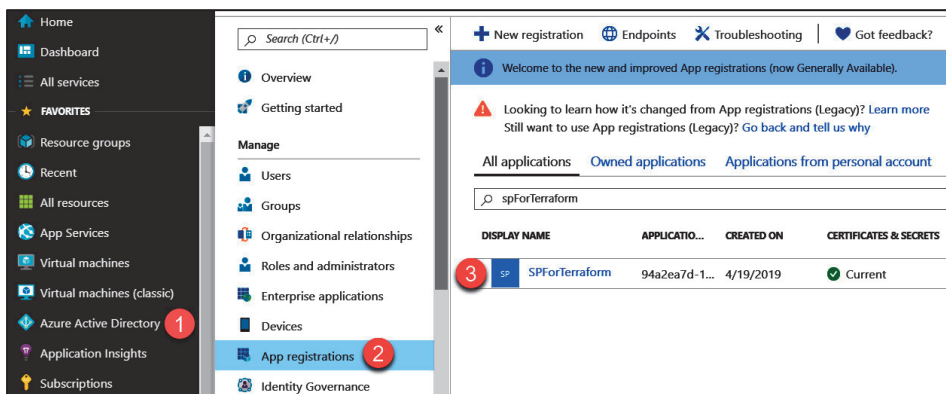
mika1@Azure:~$ az ad sp create-for-rbac --name="SPForTerraform" --role="Contributor" --scopes="/subscriptions/1da42ac9-ee3e-4fdb-
Changing "SPForTerraform" to a valid URI of "http://SPForTerraform", which is the required format used for service principal names
Retrying role assignment creation: 1/36
{
  "appId": "94a2ea7d-10c9-46b3-803",
  "displayName": "SPForTerraform",
  "name": "http://SPForTerraform",
  "password": "a1ca14d7-0aa0-",
  "tenant": "2e3a33f9-"
}
```

Rysunek 2.6. Tworzenie Azure SP

W wyniku utworzenia tej jednostki zwrócone zostały trzy identyfikatory:

- Identyfikator aplikacji, który jest również nazywany identyfikatorem klienta (ang. *Client ID*).
- Tajny klucz klienta (ang. *Client Secret*).
- Identyfikator dzierżawcy (ang. *Tenant ID*).

SP jest tworzona w usłudze Azure AD. Zobacz poniższy zrzut ekranu:



Rysunek 2.7. Lista rejestracji aplikacji w portalu Azure

Właśnie odkryliśmy, jak utworzyć jednostkę SP w usłudze Azure AD, i zezwoliliśmy na manipulowanie zasobami naszych subskrypcji Azure.

Nauczmy się teraz, jak skonfigurować Terraform, aby skorzystać z naszej jednostki Azure SP.

Konfiguracja dostawcy Terraform

Po utworzeniu Azure SP będziemy konfigurować Terraform, żeby połączyć się z Azure za pomocą tej SP. Aby to zrobić, wykonaj następujące kroki:

1. W wybranym katalogu utwórz nową nazwę pliku, *provider.tf* (rozszerzenie *.tf* wskazuje na pliki Terraform), który zawiera następujący kod:

```
provider "azurerm" {  
  features {}  
  subscription_id = "<ID subskrypcji>"  
  client_id = "<ID klienta>"  
  client_secret = "<Klucz klienta>"  
  tenant_id = "<ID dzierżawcy>"  
}
```

W poprzednim kodzie wskazujemy, że używamy przez nas dostawcą jest *azurerm*. Informacje o uwierzytelnianiu Azure zawiera jednostka SP, która została utworzona. Dodajemy nowe właściwości (ang. *features*), które zapewniają możliwość dostosowania zachowania zasobów dostawcy Azure.

Jednak ze względów bezpieczeństwa nie jest zalecane umieszczanie informacji identyfikacyjnych w jawnym tekście w konfiguracji, szczególnie jeśli wiesz, że ten kod może być dostępny dla innych osób.

2. Dlatego ulepszymy poprzedni kod, zastępując go tym:

```
provider "azurerm" {  
  features {}  
}
```

3. Tak więc usuwamy dane uwierzytelniające w konfiguracji Terraform i przekazujemy te wartości do określonych zmiennych środowiskowych Terraform:

- ARM_SUBSCRIPTION_ID,
- ARM_CLIENT_ID,
- ARM_CLIENT_SECRET,
- ARM_TENANT_ID.

Ważna uwaga

Aby uzyskać więcej informacji na temat dostawcy azurerm, zapoznaj się z dokumentacją pod adresem <https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs>. Jak ustawić te zmienne środowiskowe, dowiemy się w dalszej części tego rozdziału, w sekcji „Uruchamianie Terraform w celu wdrożenia”.

W rezultacie kod Terraform nie zawiera już żadnych informacji identyfikacyjnych.

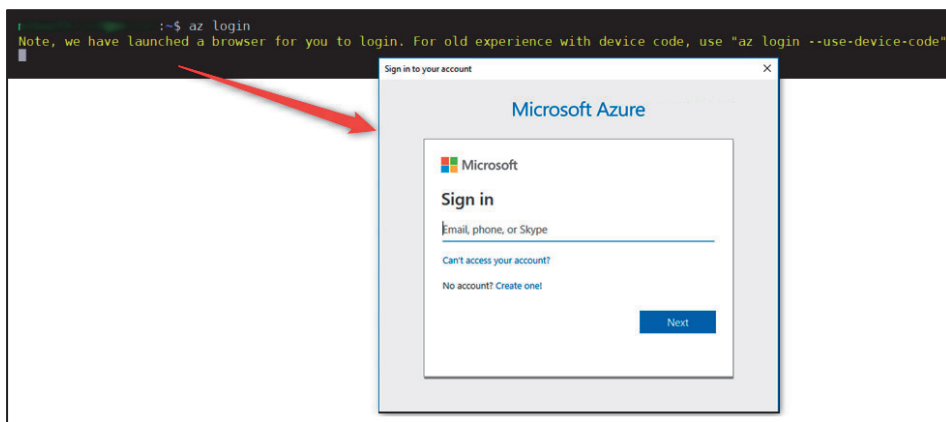
Właśnie nauczyliśmy się, jak konfigurować Terraform w celu uwierzytelniania w Azure. Teraz wyjaśnimy, jak szybko skonfigurować Terraform, aby przeprowadzić rozwój aplikacji i testowanie.

Konfiguracja Terraform w celu rozwoju aplikacji i testowania

Kiedy pracujesz lokalnie i chcesz szybko przetestować kod Terraform — np. w środowisku piaskownicy (ang. *sandbox*) — może być wygodniej i szybciej użyć własnego konta Azure zamiast korzystania z jednostki SP.

Aby to zrobić, można wcześniej połączyć się z Azure za pomocą polecenia `az login`. Wprowadź informacje identyfikacyjne w oknie, które się otworzy.

Poniżej znajduje się zrzut ekranu okna logowania Azure:



Rysunek 2.8. Ekran logowania Azure

Jeśli uzyskano dostęp do kilku subskrypcji, pożądaną można wybrać za pomocą następującego polecenia:

```
az account set --subscription="<ID subskrypcji>"
```

Następnie konfigurujemy dostawcę (ang. *provider*) Terraform, tak jak zrobiliśmy poprzednio, czyli ustawiamy "azurerm" { }.

Oczywiście tej metody uwierzytelniania nie należy stosować w przypadku wykonywania na zdalnym serwerze.

Uwaga

Aby uzyskać więcej informacji na temat konfiguracji dostawcy, zapoznaj się z dokumentacją pod adresem <https://www.terraform.io/docs/providers/azurerm/index.html>.

Dlatego konfiguracja Terraform dla Azure jest definiowana przez konfigurację dostawcy, który wykorzystuje informacje z jednostki SP.

Po zakończeniu tej konfiguracji możemy zacząć tworzyć konfigurację Terraform w celu udostępnienia zasobów Azure i zarządzania nimi.

Tworzenie skryptu Terraform w celu wdrożenia infrastruktury Azure

Aby zilustrować wykorzystanie Terraform do wdrażania zasobów na platformie Azure, zdefiniujemy prostą architekturę Azure z Terraform, złożoną z następujących komponentów:

- Grupa zasobów Azure.
- Konfiguracja sieci, która składa się z sieci wirtualnej i podsieci.
- W tej podsieci utworzymy maszynę wirtualną, która ma publiczny adres IP.

Zrobimy to w tym samym katalogu, w którym wcześniej utworzyliśmy plik *provider.tf*. Utworzymy plik *main.tf* z następującą zawartością:

1. Zaczniemy od kodu, który zapewnia grupę zasobów:

```
resource "azurerm_resource_group" "rg" {  
  name = "bookRg"  
  location = "West Europe"  
  tags {  
    environment = "Terraform Azure"  
  }  
}
```

Każdy kawałek kodu Terraform składa się z tego samego modelu składni, a składnia obiektu Terraform składa się z czterech części:

- Blok `resource` lub `data`.
- Nazwa zarządzanego zasobu (np. `azurerm_resource_group`).
- Wewnętrzny ID Terraform (np. `rg`).
- Lista właściwości odpowiadających właściwościom zasobu (tzn. `name` i `location`).

Ważna uwaga

Więcej informacji na temat składni Terraform jest dostępnych na stronie <https://www.terraform.io/docs/configuration-0-11/resources.html>.

Ten kod wykorzystuje zasób Terraform `azurerm_resource_group` i zapewnia grupę zasobów o nazwie `bookRg`, która zostanie przechowywana w lokalizacji Europy Zachodniej.

2. Następnie napiszemy kod dla części sieciowej:

```
resource "azurerm_virtual_network" "vnet" {
  name = "book-vnet"
  location = "West Europe"
  address_space = ["10.0.0.0/16"]
  resource_group_name = azurerm_resource_group.rg.name
}
resource "azurerm_subnet" "subnet" {
  name = "book-subnet"
  virtual_network_name = azurerm_virtual_network.vnet.name
  resource_group_name = azurerm_resource_group.rg.name
  address_prefix = "10.0.10.0/24"
}
```

W powyższym kodzie Terraform tworzymy sieć wirtualną book-vnet, a w niej tworzymy podsieć o nazwie book-subnet.

Jeśli przyjrzymy się uważnie temu kodowi, zobaczymy, że nie umieszczamy wyraźnych identyfikatorów wskazujących zależności między zasobami, ale używamy wskaźników do zasobów Terraform.

Sieć wirtualna i podsieć są właściwością grupy zasobów `${azurerm_resource_group.rg.name}`, co informuje Terraform, że sieć wirtualna i podsieć zostaną utworzone tuż po grupie zasobów. Jeśli chodzi o podsieć, jest ona zależna od swojej sieci wirtualnej przy użyciu wartości `${azurerm_virtual_network.vnet.name}`; jest to jawna koncepcja zależności.

Teraz napiszmy kod Terraform udostępniający zasoby dla maszyny wirtualnej, który składa się z następujących elementów:

- Interfejs sieciowy.
- Publiczny adres IP.
- Obiekt usługi Azure Storage do rozruchu diagnostycznego (dzienniki informacji o rozruchu).
- Maszyna wirtualna.

Przykładowy kod *interfejsu sieciowego* z konfiguracją IP wygląda następująco:

```
resource "azurerm_network_interface" "nic" {
  name = "book-nic"
  location = "West Europe"
  resource_group_name = azurerm_resource_group.rg.name
  ip_configuration {
    name = "bookipconfig"
    subnet_id = azurerm_subnet.subnet.id
    private_ip_address_allocation = "Dynamic"
```

```
        public_ip_address_id = azurerm_public_ip.pip.id
    }
}
```

W tym kodzie Terraform używamy bloku `azurerm_network_interface` (https://www.terraform.io/docs/providers/azurerm/r/network_interface.html). W nim konfigurujemy nazwę, region, grupę zasobów i konfigurację IP z dynamicznym adresem IP interfejsu sieciowego.

Kod dotyczący publicznego adresu IP, który ma adres IP z właśnie utworzonej podsieci, wygląda następująco:

```
resource "azurerm_public_ip" "pip" {
  name = "book-ip"
  location = "West Europe"
  resource_group_name = "${azurerm_resource_group.rg.name}"
  public_ip_address_allocation = "Dynamic"
  domain_name_label = "bookdevops"
}
```

W powyższym kodzie Terraform używamy bloku `azurerm_public_ip` — opis na stronie https://www.terraform.io/docs/providers/azurerm/r/public_ip.html. W nim konfigurujemy dynamiczną alokację adresu IP i etykietę DNS.

Kod dla obiektu storage, którego używamy do dzienników diagnostyki rozruchu, jest następujący:

```
resource "azurerm_storage_account" "stor" {
  name = "bookstor"
  location = "West Europe"
  resource_group_name = azurerm_resource_group.rg.name
  account_tier = "Standard"
  account_replication_type = "LRS"
}
```

W tym kodzie Terraform używamy bloku `azurerm_storage_account`, opisanego pod adresem https://www.terraform.io/docs/providers/azurerm/r/storage_account.html. W nim konfigurujemy nazwę, region, grupę zasobów i typ magazynu — w naszym przypadku jest to Standard LRS.

Ważna uwaga

Dokumentację obiektu storage można znaleźć pod adresem <https://docs.microsoft.com/en-us/azure/storage/common/storage-account-overview>.

Kod maszyny wirtualnej Ubuntu, który zawiera identyfikator utworzonego wcześniej interfejsu sieciowego, wygląda następująco:

```
resource "azurerm_linux_virtual_machine" "vm" {
  name = "bookvm"
  location = "West Europe"
  resource_group_name = azurerm_resource_group.rg.name
  vm_size = "Standard_DS1_v2"
  network_interface_ids = ["${azurerm_network_interface.nic.id}"]
  storage_image_reference {
    publisher = "Canonical"
    offer = "UbuntuServer"
    sku = "16.04-LTS"
    version = "latest"
  }
  ....
}
```

W tym kodzie Terraform używamy bloku `azurerm_linux_virtual_machine`, opisanego pod adresem https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/linux_virtual_machine. W nim konfigurujemy nazwę, rozmiar (`Standard_DS1_v2`), odniesienie do obiektu Terraform `network_interface` i typ systemu operacyjnego maszyny wirtualnej (Ubuntu).

Wszystkie te sekcje kodu są dokładnie takie same jak poprzednie, z użyciem jawnej zależności do określenia relacji między zasobami.

Ważna uwaga

Ten kompletny kod źródłowy jest dostępny pod adresem https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP02/terraform_simple_script.

Właśnie utworzyliśmy kompletny skrypt Terraform, który pozwala nam udostępniać małą infrastrukturę Azure. Jednak, jak w każdym języku, istnieją dobre praktyki dotyczące separacji plików, stosowania jasnego i czytelnego kodu, a także wykorzystania wbudowanych funkcji.

Postępowanie zgodnie z dobrymi praktykami Terraform

Właśnie przyjrzelśmy się przykładowi kodu Terraform, który udostępniał infrastrukturę platformy Azure. Warto również zapoznać się z kilkoma dobrymi praktykami dotyczącymi pisania kodu Terraform.

Lepsza widoczność dzięki separacji plików

Podczas wykonywania kodu Terraform wszystkie pliki konfiguracyjne w katalogu *execution*, które mają rozszerzenie *.tf*, są wykonywane automatycznie; w naszym przykładzie mamy *provider.tf* i *main.tf*. Dobrze jest rozdzielić kod na kilka plików, aby poprawić czytelność kodu i jego ewolucję.

Korzystając z naszego przykładowego skryptu, możemy zrobić lepiej, dzieląc go na poniższe:

- *Rg.tf* — zawiera kod dla grupy zasobów.
- *Network.tf* — zawiera kod dla sieci wirtualnej i podsieci.
- *Compute.tf* — zawiera kod dla interfejsu sieciowego, publicznego adresu IP, magazynu i maszyny wirtualnej.

Ważna uwaga

Pełny kod z osobnymi plikami można znaleźć pod adresem https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP02/terraform_separate_files.

Ochrona danych wrażliwych

Należy zachować ostrożność w przypadku poufnych danych w konfiguracji Terraform, takich jak hasła i uprawnienia dostępu. Dowiedzieliśmy się już, że do uwierzytelniania dostępu do Azure nie trzeba zostawiać ich w kodzie. Dodatkowo w naszym przykładzie dotyczącym konta administratora maszyny wirtualnej zwróć uwagę na to, że hasło konta administratora maszyny wirtualnej zostało wyraźnie określone w tej konfiguracji Terraform. Aby temu zaradzić, możemy użyć menedżera do przechowywania haseł, takiego jak Azure Key Vault lub HashiCorp Vault, i uzyskać te hasła za pośrednictwem Terraform.

Dynamizacja konfiguracji za pomocą zmiennych i funkcji interpolacji

Podczas tworzenia konfiguracji Terraform ważne jest, aby od samego początku brać pod uwagę, że infrastruktura, która będzie hostowała aplikację, jest bardzo często taka sama na wszystkich etapach. Jednak tylko niektóre informacje będą się różnić w zależności od etapu, np. nazwa zasobów i liczba instancji.

Aby zapewnić większą elastyczność kodu, musimy użyć zmiennych w kodzie, wykonując następujące czynności:

1. Zadeklaruj zmienne, dodając następujący przykładowy kod w globalnym kodzie Terraform. Alternatywnie możemy dodać go w innym pliku (np. *variable.tf*) w celu lepszej czytelności kodu.

```
variable "resource_group_name" {
    description = "Name of the resource group"
}
variable "location" {
    description = "Location of the resource"
    default = "West Europe"
}
variable "application_name" {
    description = "Name of the application"
}
```

2. Umieść ich wartości w innym pliku *.tfvars* o nazwie *terraform.tfvars*, ze składnią *nazwa_zmiennej=wartość*, jak poniżej:

```
resource_group_name = "bookRg"
application_name = "book"
```

3. Użyj tych zmiennych w kodzie jako `var.<nazwa zmiennej>`; np. w kodzie grupy zasobów Terraform możemy napisać:

```
resource "azurerm_resource_group" "rg" {
    name = var.resoure_group_name
    location = var.location
    tags {
        environment = "Terraform Azure"
    }
}
```

Oprócz tego Terraform ma obszerną listę wbudowanych funkcji, których można używać do manipulowania danymi lub zmiennymi. Aby dowiedzieć się więcej o tych funkcjach, zapoznaj się z oficjalną dokumentacją pod adresem <https://www.terraform.io/docs/configuration/functions.html>.

Oczywiście istnieje wiele innych dobrych praktyk, ale należy je stosować od pierwszych linii kodu, by zapewnić, że Twój kod jest dobrze utrzymany.

Ważna uwaga

Pełny i ostateczny kod tego przykładu jest dostępny pod adresem https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP02/terraform_vars_interp.

Korzystając z najlepszych praktyk, napisaliśmy konfigurację Terraform, która pozwala na utworzenie prostej infrastruktury chmurowej na platformie Azure, zapewniającej

sieć i maszynę wirtualną. Przyjrzyjmy się teraz, jak uruchomić Terraform z naszym kodem, aby udostępnić tę infrastrukturę.

Uruchamianie Terraform w celu wdrożenia

Po napisaniu konfiguracji Terraform musimy teraz uruchomić Terraform, aby wdrożyć naszą infrastrukturę.

Jednak przed wykonaniem należy zagwarantować uwierzytelnianie za pomocą jednostki usługi Azure SP, aby zapewnić, że Terraform może zarządzać zasobami platformy Azure.

By to zrobić, możemy ustawić zmienne środowiskowe określone dla Terraform, aby zawierały informacje o SP utworzonej wcześniej w sekcji „Konfigurowanie Terraform dla platformy Azure”, lub możemy użyć skryptu `az cli`.

Poniższy skrypt eksportuje cztery zmienne środowiskowe Terraform w systemie operacyjnym Linux:

```
export ARM_SUBSCRIPTION_ID=xxxxx-xxxxx-xxxx-xxxx
export ARM_CLIENT_ID=xxxxx-xxxxx-xxxx-xxxx
export ARM_CLIENT_SECRET=xxxxxxxxxxxxxxxxxxxx
export ARM_TENANT_ID=xxxxx-xxxxx-xxxx-xxxx
```

Dodatkowo możemy użyć skryptu `az cli` z poleceniem `login`:

az login

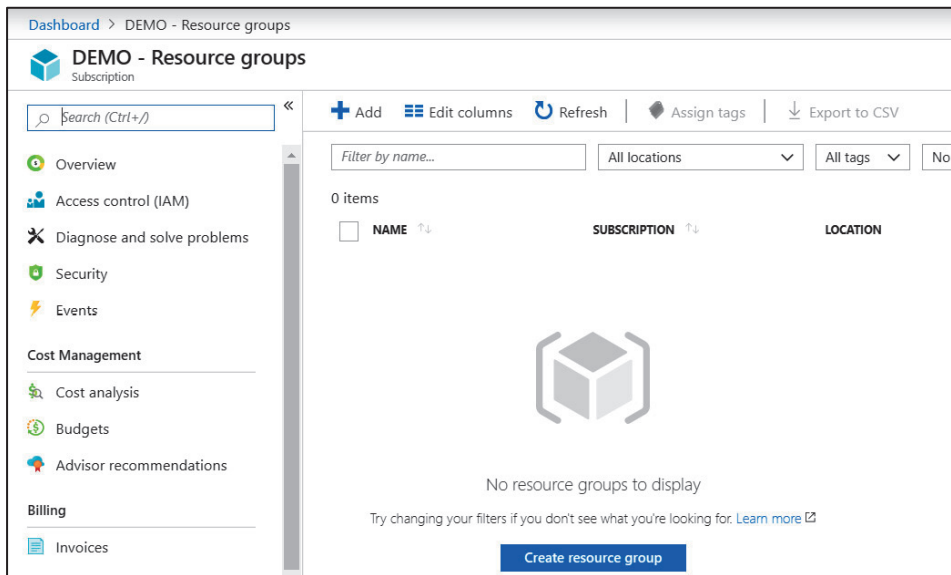
Po uwierzytelnieniu możemy uruchomić proces Terraform.

W naszym scenariuszu zaczynamy od pustej subskrypcji platformy Azure bez żadnych grup zasobów tej platformy; jednak w świecie rzeczywistym nasza subskrypcja może już zawierać grupę zasobów.

Przed uruchomieniem Terraform w portalu Azure sprawdź, czy nie masz grupy zasobów w swojej subskrypcji, w sposób jak na rysunku 2.9.

Aby uruchomić Terraform, musimy otworzyć terminal, taki jak CMD, PowerShell lub Bash, i przejść do katalogu, w którym znajdują się zapisane wcześniej pliki konfiguracyjne Terraform.

Konfiguracja Terraform jest wykonywana w kilku krokach: inicjalizacja, podgląd zmian i zastosowanie tych zmian.



Rysunek 2.9. Brak grupy zasobów na platformie Azure

Teraz przyjrzyjmy się szczegółowo wykonaniu tych kroków, zaczynając od kroku inicjalizacji.

Inicjalizacja

Krok inicjowania umożliwia Terraform wykonanie następujących czynności:

- Inicjalizacja kontekstu Terraform, aby sprawdzić i nawiązać połączenie między dostawcą Terraform a usługą zdalną — w naszym przypadku dotyczy to platformy Azure.
- Pobranie pluginu(ów) dostawcy(ów) — w naszym przypadku będzie to dostawca `azurerm`.
- Sprawdzenie zmiennych kodu.

Aby wykonać inicjalizację, uruchom polecenie `init`:

```
terraform init
```

Poniżej znajduje się zrzut ekranu wykonania `terraform init`.

```
root@ubuntu-bionic:/learningdevops/CHAP02/terraform_separate_files# terraform init
Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "2.63.0"...
- Installing hashicorp/azurerm v2.63.0...
- Installed hashicorp/azurerm v2.63.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

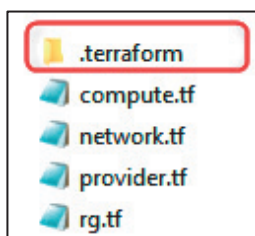
If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Rysunek 2.10. Wykonywanie polecenia terraform init

Jak dowiedzieliśmy się podczas wykonywania poprzedniego polecenia, Terraform wykonuje następujące czynności:

- Pobiera najnowszą wersję wtyczki azurerm.
- Tworzy katalog roboczy *.terraform*.

Poniżej znajduje się zrzut ekranu katalogu *.terraform*:



Rysunek 2.11. Katalog konfiguracji Terraform

Ważna uwaga

Aby uzyskać więcej informacji na temat poleceń `init`, zapoznaj się z dokumentacją pod adresem <https://www.terraform.io/docs/commands/init.html>.

Po zakończeniu kroku inicjalizacji możemy przejść do kolejnego kroku, czyli podglądu zmian.

Podgląd zmian

Następnym krokiem jest podgląd zmian wprowadzonych w infrastrukturze przed ich zastosowaniem.

Aby to zrobić, uruchom Terraform z opcją `plan`. Polecenie `plan` automatycznie używa pliku `terraform.tfvars` do ustawiania zmiennych.

Uruchom polecenie `plan`:

```
terraform plan
```

Poniżej pokazano wykonanie polecenia `terraform plan`:

```
~/mnt/d/DevOps/Learning-DevOps/CHAP02/terraform_separate_files$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

-----

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

+ azurerm_network_interface.nic
  id:                                     <computed>
  applied_dns_servers.#:                 <computed>
  dns_servers.#:                         <computed>
  enable_accelerated_networking:         "false"
  enable_ip_forwarding:                  "false"
  internal_dns_name_label:                <computed>
  internal_fqdn:                         <computed>
  ip_configuration.#:                    <computed>
  ip_configuration.0.application_gateway_backend_address_pools_ids.#: <computed>
  ip_configuration.0.application_security_group_ids.#: <computed>
  ip_configuration.0.load_balancer_backend_address_pools_ids.#: <computed>
  ip_configuration.0.load_balancer_inbound_nat_rules_ids.#: <computed>
  ip_configuration.0.name:                "bookipconfig"
  ip_configuration.0.primary:              <computed>
  ip_configuration.0.private_ip_address_allocation:         "dynamic"
  ip_configuration.0.private_ip_address_version:            "IPv4"
  ip_configuration.0.public_ip_address_id:                  "${azurerm_public_ip.pip.id}"
  ip_configuration.0.subnet_id:                             "${azurerm_subnet.subnet.id}"
  location:                                                  "westeurope"

+ azurerm_virtual_network.vnet
  id:                                     <computed>
  address_space.#:                       "1"
  address_space.0:                       "10.0.0.0/16"
  location:                               "westeurope"
  name:                                   "book-vnet"
  resource_group_name:                    "bookRg"
  subnet.#:                               <computed>
  tags.%:                                 <computed>

Plan: 7 to add, 0 to change, 0 to destroy.

-----

Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
```

Rysunek 2.12. Wykonywanie polecenia `terraform plan`

Podczas wykonywania polecenia `plan` zostanie wyświetlona nazwa i właściwości zasobów, na które zmiana będzie miała wpływ. Wyświetlona zostanie również liczba nowych zasobów i liczba zasobów, które zostaną zmodyfikowane, a także ilość zasobów, które zostaną usunięte.

Ważna uwaga

Aby uzyskać więcej informacji na temat polecenia `plan`, zapoznaj się z dokumentacją pod adresem <https://www.terraform.io/docs/commands/plan.html>.

Właśnie sprawdziliśmy prognozę zmian, które zostaną dokonane w naszej infrastrukturze. Teraz zobaczmy, jak je zastosować.

Stosowanie zmian

Po ustaleniu, że polecenie `plan` odpowiada naszym oczekiwaniom, ostatnim krokiem jest zastosowanie kodu Terraform w czasie rzeczywistym do udostępniania zasobów i zastosowania zmian w naszej infrastrukturze.

Aby to zrobić, wykonamy polecenie `apply`:

```
terraform apply
```

Ta komenda wykonuje taką samą operację jak komenda `plan` i interaktywnie prosi użytkownika o potwierdzenie, że chcemy wprowadzić zmiany.

Poniżej znajduje się zrzut ekranu po wykonaniu polecenia `terraform apply`:

```
:/mnt/d/DevOps/Learning-DevOps/CHAP02/terraform_separate_files$ terraform apply
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

+ azurerm_network_interface.nic
  id:                                     <computed>
  applied_dns_servers.#:                 <computed>

tags.%:                                <computed>

Plan: 7 to add, 0 to change, 0 to destroy.

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

Enter a value: 
```

Rysunek 2.13. Potwierdzenie zmian do zastosowania w Terraform

Potwierdzenie zmian jest dokonywane przez wpisanie `yes` (lub `no` w celu anulowania), a następnie Terraform zastosuje zmiany w infrastrukturze.

Poniżej znajduje się zrzut ekranu wykonania polecenia `terraform apply`:

```
Only 'yes' will be accepted to approve.
Enter a value: yes
azurerm_resource_group.rg: Creating...
  location:      "" => "westeurope"
  name:          "" => "bookRg"
  tags.%:        "" => "1"
  tags.environment: "" => "Terraform Azure"
azurerm_virtual_machine.vm: Still creating... (2m30s elapsed)
azurerm_virtual_machine.vm: Creation complete after 2m32s (ID: /subscriptions/1da42a
apply complete! Resources: 7 added, 0 changed, 0 destroyed.
```

Rysunek 2.14. Wykonywanie polecenia `terraform apply`

Dane wyjściowe polecenia `apply` zawierają wszystkie akcje wykonywane przez Terraform wraz ze wszystkimi zmianami i z zasobami, których ono dotyczy. Proces kończy się wierszem podsumowującym, który wyświetla sumę wszystkich dodanych, zmienionych lub zniszczonych zasobów.

Ważna uwaga

Aby uzyskać więcej informacji na temat polecenia `apply`, zapoznaj się z dokumentacją pod adresem <https://www.terraform.io/docs/commands/apply.html>.

Ponieważ polecenie Terraform `apply` zostało wykonane poprawnie, możemy sprawdzić w Azure Portal, czy istnieją zasoby opisane w kodzie Terraform.

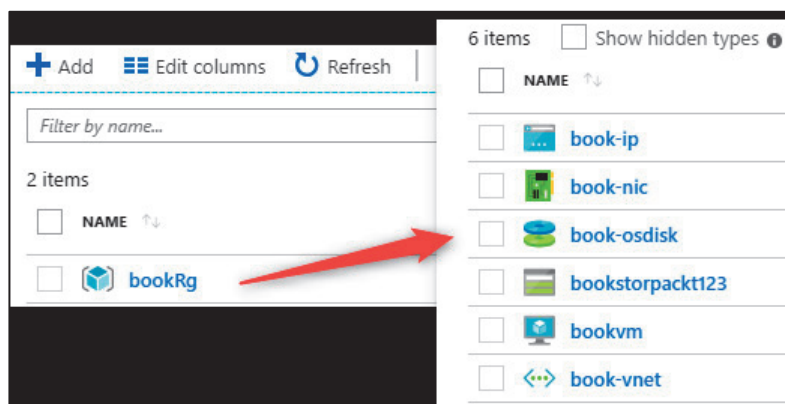
Na następnej stronie znajduje się zrzut ekranu zasobów platformy Azure (rysunek 2.15).

Z portalu możemy się dowiedzieć, że zasoby określone w kodzie Terraform zostały pomyślnie zainicjowane.

Tak więc właśnie zobaczyliśmy, jak Terraform przydaje się do udostępniania infrastruktury za pomocą trzech głównych poleceń:

- `init`, służącego do inicjalizacji kontekstu,
- `plan`, służącego do podglądu zmian,
- `apply`, służącego do zastosowania zmian.

W następnej sekcji omówimy inne polecenia Terraform i cykl życia Terraform.



Rysunek 2.15. Lista udostępnionych zasobów na platformie Azure

Zrozumienie cyklu życia Terraform z różnymi opcjami wiersza polecenia

Właśnie odkryliśmy, że wprowadzanie zmian w infrastrukturze przy użyciu Terraform odbywa się głównie za pomocą trzech poleceń. Obejmują one inicjowanie, podgląd zmian i stosowanie zmian. Jednak Terraform ma inne bardzo praktyczne i ważne polecenia, których można użyć do zarządzania cyklem życia naszej infrastruktury. Należy też rozważyć pytanie, jak wykonać Terraform w kontekście automatyzacji, takim jak potok CI/CD.

Wśród innych operacji, które można wykonać na elemencie infrastruktury, znajduje się usuwanie zasobów. Odbywa się to w celu przebudowania lub usunięcia tymczasowej infrastruktury.

Używanie polecenia destroy w celu przebudowy

Jednym z etapów cyklu życia infrastruktury utrzymywanej przez IaC jest usuwanie infrastruktury; nie zapominaj, że jednym z celów i korzyści IaC jest możliwość dokonywania szybkich zmian w infrastrukturze, a także tworzenia środowisk na żądanie. Oznacza to, że tworzymy i utrzymujemy środowiska tak długo, jak długo ich potrzebujemy, a gdy przestaną być używane, niszczymy je, zapewniając w ten sposób firmie oszczędności finansowe.

By to osiągnąć, konieczne jest zautomatyzowanie usuwania infrastruktury, aby móc ją szybko odbudować.

Aby zniszczyć infrastrukturę, która została wcześniej udostępniona za pomocą Terraform, wykonaj następujące polecenie:

```
terraform destroy
```

Wykonanie tego polecenia powinno dać następujący wynik:

```
~/mnt/d/DevOps/Learning-DevOps/CHAP02/terraform_separate_files$ terraform destroy
azure_rm_resource_group.rg: Refreshing state... (ID: /subscriptions/1da42ac9-ee3e-4fdb-b294-f7a607f589d5/resourceGroups/bookRg)
azure_rm_storage_account.stor: Refreshing state... (ID: /subscriptions/1da42ac9-ee3e-4fdb-b294-f7a607f589d5/storageAccounts/bookstorpackt123)
azure_rm_public_ip.pip: Refreshing state... (ID: /subscriptions/1da42ac9-ee3e-4fdb-b294-f7a607f589d5/network/publicIPAddresses/book-ip)
azure_rm_virtual_network.vnet: Refreshing state... (ID: /subscriptions/1da42ac9-ee3e-4fdb-b294-f7a607f589d5/network/virtualNetworks/book-vnet)
azure_rm_subnet.subnet: Refreshing state... (ID: /subscriptions/1da42ac9-ee3e-4fdb-b294-f7a607f589d5/networks/book-vnet/subnets/book-subnet)
azure_rm_network_interface.nic: Refreshing state... (ID: /subscriptions/1da42ac9-ee3e-4fdb-b294-f7a607f589d5/network/networkInterfaces/book-nic)
azure_rm_virtual_machine.vm: Refreshing state... (ID: /subscriptions/1da42ac9-ee3e-4fdb-b294-f7a607f589d5/compute/virtualMachines/bookvm)

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
- destroy

Terraform will perform the following actions:

- azure_rm_network_interface.nic
- azure_rm_public_ip.pip
- azure_rm_resource_group.rg
- azure_rm_storage_account.stor
- azure_rm_subnet.subnet
- azure_rm_virtual_machine.vm
- azure_rm_virtual_network.vnet

Plan: 0 to add, 0 to change, 7 to destroy.

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes
```

Rysunek 2.16. Potwierdzenie wykonania polecenia terraform destroy

To polecenie, podobnie jak `apply`, wymaga potwierdzenia przez użytkownika przed zastosowaniem:

```
azure_rm_resource_group.rg: Still destroying... (ID: /subscriptions/1da42ac9-ee3e-4fdb-b294-f7a607f589d5/resourceGroups/bookRg, 2m40s elapsed)
azure_rm_resource_group.rg: Destruction complete after 2m47s

Destroy complete! Resources: 7 destroyed.
```

Rysunek 2.17. Wykonywanie polecenia terraform destroy

Po zweryfikowaniu poczekaj na komunikat potwierdzający, że infrastruktura została zniszczona.

Polecenie `destroy` niszczy tylko zasoby skonfigurowane w bieżącej konfiguracji Terraform. Nie ma ono wpływu na inne zasoby (utworzone ręcznie lub przez inny kod Terraform). Jeśli jednak nasz kod Terraform udostępnia grupę zasobów, `destroy` zniszczy całą jej zawartość.

Właśnie odkryliśmy, że Terraform umożliwia również niszczenie zasobów w wierszu poleceń. Teraz nauczymy się formatować i sprawdzać poprawność kodu Terraform.

Uwaga

Aby uzyskać więcej informacji na temat polecenia `destroy`, zapoznaj się z dokumentacją pod adresem <https://www.terraform.io/docs/commands/destroy.html>.

Formatowanie i walidacja konfiguracji

Po nauczeniu się, jak niszczyć zasoby za pomocą Terraform, należy również podkreślić znaczenie posiadania dobrze sformatowanego kodu, który spełnia reguły stylów Terraform, i sprawdzić, czy kod nie zawiera błędów składniowych ani błędów zmiennych.

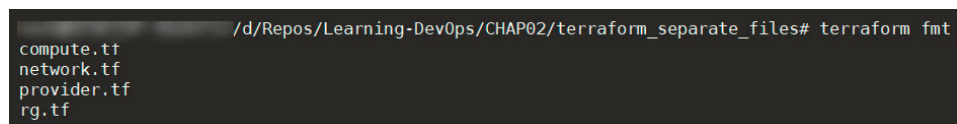
Formatowanie kodu

Terraform ma polecenie, które umożliwia prawidłowe dopasowanie kodu do stylów i konwencji Terraform.

Następujące polecenie automatycznie formatuje kod:

```
terraform fmt
```

Poniżej znajduje się zrzut ekranu formatowania pliku przez Terraform:



```
/d/Repos/Learning-DevOps/CHAP02/terraform_separate_files# terraform fmt
compute.tf
network.tf
provider.tf
rg.tf
```

Rysunek 2.18. Wykonywanie polecenia `terraform fmt`

Polecenie przeformatowuje kod i wskazuje listę uporządkowanych plików.

Ważna uwaga

Więcej informacji na temat przewodnika po stylu Terraform można znaleźć na stronie <https://www.terraform.io/docs/configuration/style.html>. Dodatkowo, aby uzyskać informacje o poleceniu `terraform fmt`, odwiedź <https://www.terraform.io/docs/commands/fmt.html>.

Walidacja kodu

Terraform zawiera również polecenie, które weryfikuje kod i pozwala nam wykryć możliwe błędy przed wykonaniem polecenia `plan` lub `apply`.

Przeanalizujmy następujący kod:

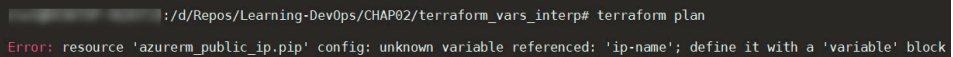
```
resource "azurerm_public_ip" "pip" {
  name = var.ip-name
  location = var.location
```



```
resource_group_name = "${azurerm_resource_group.rg.name}"
allocation_method = "Dynamic"
domain_name_label = "bookdevops"
}
```

We właściwości `name` używamy zmiennej `ip-name`, która nie została zadeklarowana ani zainicjowana żadną wartością.

Wykonanie polecenia `terraform plan` powinno zwrócić błąd:



```
~/d/Repos/Learning-DevOps/CHAP02/terraform_vars_interp# terraform plan
Error: resource 'azurerm_public_ip.pip' config: unknown variable referenced: 'ip-name'; define it with a 'variable' block
```

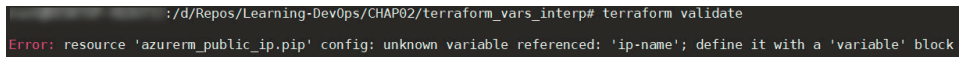
Rysunek 2.19. Wykonanie polecenia `plan` z błędem

Z powodu tego błędu w procesie CI/CD może zostać opóźnione wdrożenie infrastruktury.

Aby wykryć błędy w kodzie Terraform tak wcześnie, jak to możliwe w cyklu rozwoju, wykonaj następujące polecenie. Zweryfikuje to wszystkie pliki Terraform w katalogu:

terraform validate

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia:



```
~/d/Repos/Learning-DevOps/CHAP02/terraform_vars_interp# terraform validate
Error: resource 'azurerm_public_ip.pip' config: unknown variable referenced: 'ip-name'; define it with a 'variable' block
```

Rysunek 2.20. Wykonanie polecenia `terraform validate`

Tutaj obserwujemy ten sam błąd, który został zwrócony przez polecenie `plan`.

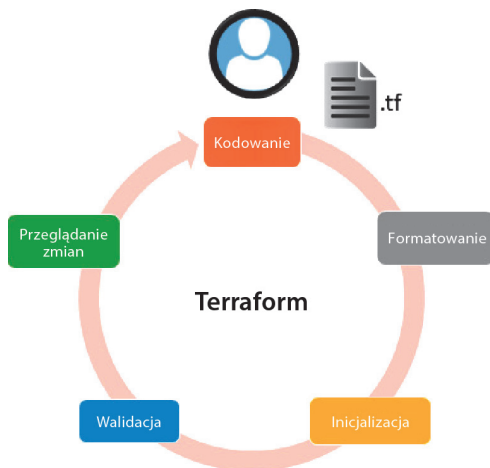
Właśnie odkryliśmy główne polecenia Terraform. Zapoznajmy się bardziej szczegółowo z integracją Terraform w procesie CI/CD.

Cykl życia Terraform w procesie CI/CD

Do tej pory widzieliśmy i wykonywaliśmy na komputerze lokalnym różne polecenia Terraform, które pozwalają nam inicjować, przeglądać, stosować i niszczyć infrastrukturę oraz formatować i weryfikować kod Terraform. W przypadku lokalnego korzystania z Terraform w kontekście programistycznym cykl życia wykonania jest jak na rysunku 2.21.

Poniższe kroki wyjaśniają sekwencję z poprzedniego diagramu:

1. Tworzenie kodu.
2. Formatowanie kodu za pomocą `terraform fmt`.

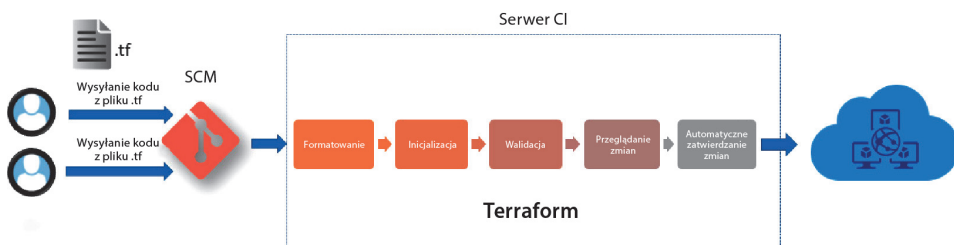


Rysunek 2.21. Proces CI/CD Terraform

3. Inicjalizacja za pomocą `terraform init`.
4. Walidacja kodu za pomocą `terraform validate`.
5. Przeglądanie stosowanych zmian za pomocą `transform plan`.
6. Ręczna weryfikacja zmian Terraform w infrastrukturze.

Jednak proces IaC, podobnie jak aplikacja, musi zostać wdrożony lub wykonany podczas **automatycznego procesu CI/CD**. Zaczyna się on od archiwizacji kodu Terraform członków zespołu. Następnie uruchamia proces CI i wykonuje polecenia Terraform, które omówiliśmy w tym rozdziale.

Poniżej znajduje się zrzut ekranu cyklu życia Terraform w automatyzacji CI/CD:



Rysunek 2.22. Przepływ pracy Terraform CI/CD

Kroki CI/CD wykonywane przez serwer CI (na którym zainstalowano Terraform) dla Terraform są następujące:

1. Pobieranie kodu z SCM.
2. Formatowanie kodu za pomocą `terraform fmt`.

3. Inicjalizacja za pomocą `terraform init`.
4. Walidacja kodu za pomocą `terraform validate`.
5. Wyświetlanie podglądu zmian w infrastrukturze za pomocą `terraform plan -out=out.tfplan`.
6. Stosowanie zmian w trybie automatycznym za pomocą `terraform apply --auto-approve out.tfplan`.

Jeśli dodamy opcję `--auto-approve` do poleceń `apply` i `destroy`, Terraform może również wykonywać te operacje w trybie automatycznym. Ma to na celu uniknięcie pytania użytkownika o potwierdzenie dotyczące zweryfikowania zmian, które należy zastosować. Dzięki tej automatyzacji Terraform można zintegrować z narzędziami CI/CD.

W poleceniu `plan` dodawana jest opcja `out` w celu określenia pliku w formacie `.tfplan`, który odpowiada plikowi zawierającemu dane wyjściowe polecenia `plan`. Plik `out.tfplan` jest następnie używany przez komendę `apply`. Zaletą tej procedury jest możliwość wykonania aplikacji na późniejszym planie, co można wykorzystać w przypadku wycofania zmian (ang. *rollback*).

W tej sekcji dowiedzieliśmy się, że oprócz zwykłych poleceń `init`, `plan`, `apply` i `destroy` Terraform ma również opcje, które pozwolą nam poprawić czytelność i zweryfikować składnię kodu. Dodatkowo wyjaśniliśmy, że Terraform umożliwia doskonałą integrację z potokiem CI/CD, z opcjami cyklu życia i automatyzacji.

W następnej sekcji sprawdzimy, czym jest plik `tfstate` i jak go chronić za pomocą zdalnego zaplecza.

Ochrona pliku stanu za pomocą zdalnego zaplecza

Gdy Terraform obsługuje zasoby, zapisuje stan tych zasobów w pliku stanu Terraform. Ten plik jest w formacie JSON i zachowuje zasoby i ich właściwości podczas wykonywania Terraform.

Domyślnie plik o nazwie `terraform.tfstate` jest tworzony lokalnie podczas pierwszego wykonania polecenia `apply`. Następnie będzie on używany przez Terraform za każdym razem, gdy wykonywane jest polecenie `plan`, w celu porównania jego stanu (zapisanego w pliku stanu) ze stanem infrastruktury docelowej. Na koniec zwróci podgląd tego, co zostanie zastosowane.

Podczas korzystania z Terraform ten lokalnie przechowywany plik stanu stwarza wiele problemów:

- Ponieważ ten plik zawiera stan infrastruktury, nie należy go usuwać. W przypadku usunięcia Terraform może nie zachowywać się zgodnie z oczekiwaniami podczas wykonywania.
- Musi być dostępny w tym samym czasie dla wszystkich członków zespołu obsługujących zasoby na tej samej infrastrukturze.
- Ten plik może zawierać dane wrażliwe, dlatego musi być zabezpieczony.
- W przypadku udostępniania wielu środowisk konieczna jest możliwość korzystania z wielu plików stanu.

We wszystkich tych punktach nie jest możliwe przechowywanie tego pliku stanu lokalnie ani nawet archiwizowanie go w SCM.

Aby rozwiązać ten problem, Terraform umożliwia przechowywanie tego pliku stanu w udostępnionym i bezpiecznym magazynie zwanym **zdalnym zapleczem** (ang. *remote backend*).

Ważna uwaga

Terraform obsługuje kilka typów zdalnych zapleczy; pełna lista jest dostępna pod adresem <https://www.terraform.io/docs/backends/types/remote.html>.

W naszym przypadku użyjemy **zdalnego zaplecza azurerm** do przechowywania naszych plików stanu przy użyciu konta magazynu i obiektu blob dla pliku stanu.

Dlatego zaimplementujemy i użyjemy zdalnego backendu w trzech krokach:

1. Utworzenie konta magazynu.
2. Konfiguracja Terraform dla zdalnego zaplecza.
3. Wykonanie Terraform za pomocą zdalnego zaplecza.

Przyjrzyjmy się szczegółowo wykonaniu tych kroków:

1. Aby utworzyć konto magazynu Azure i kontener obiektów blob, możemy skorzystać z portalu Azure (<https://docs.microsoft.com/en-gb/azure/storage/common/storage-quickstart-create-account?tabs=Azure-portal>) lub ze skryptu az cli:

```
# 1. Utwórz grupę zasobów
az group create --name MyRgRemoteBackend --location westeurope
# 2. Utwórz konto magazynu
```

```
az storage account create --resource-group MyRgRemoteBackend --name
↳storageremotetf --sku Standard_LRS --encryption-services blob
# 3. Pobierz klucz konta magazynu
ACCOUNT_KEY=$(az storage account keys list --resourcegroup MyRgRemoteBackend
--account-name storageremotetf --query [0].value -o tsv)
# 4. Utwórz kontener obiektów blob
AZ Storage Container create --name tfbackends --accountname storageremotetf
--account-key $ACCOUNT_KEY
```

Ten skrypt tworzy grupę zasobów MyRgRemoteBackend i konto magazynu o nazwie storageremotetf.

Następnie skrypt pobiera klucz konta magazynu i tworzy kontener obiektów blob, tfbackends wewnątrz tego konta magazynu.

Ten skrypt można uruchomić w Azure Cloud Shell, a zaletą używania skryptu zamiast korzystania z portalu Azure jest to, że ten skrypt można zintegrować z procesem CI/CD.

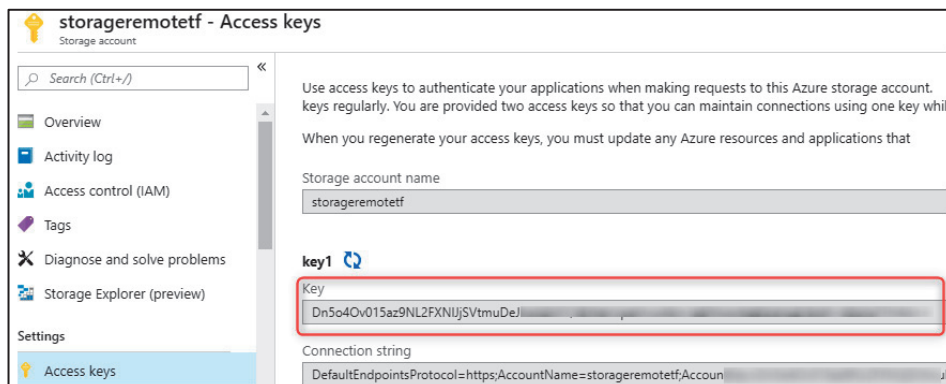
2. Następnie, aby skonfigurować Terraform do korzystania z wcześniej utworzonego zdalnego zaplecza, musimy dodać sekcję konfiguracyjną w pliku *Terraform.tf*:

```
terraform {
  backend "azurerm" {
    storage_account_name = "storageremotetfdemo"
    container_name = "tfbackends"
    key = "myappli.tfstate"
    snapshot = true
  }
}
```

Właściwość `storage_account_name` zawiera nazwę konta magazynu, właściwość `container_name` zawiera nazwę kontenera, właściwość `key` zawiera nazwę obiektu stanu blob, a właściwość `snapshot` umożliwia wykonanie migawki tego obiektu w każdej edycji przez wykonanie programu Terraform.

Jednak nadal istnieje jeszcze jedna informacja, którą należy dostarczyć do Terraform, aby mógł nawiązać połączenie i mieć uprawnienia do konta magazynu. Ta informacja to klucz dostępu, który jest prywatnym kluczem uwierzytelniania i autoryzacji konta magazynu. Aby dostarczyć klucz magazynu do Terraform, podobnie jak w przypadku informacji o usłudze Azure SP, ustaw zmienną środowiskową `ARM_STORAGE_KEY` z wartością klucza dostępu do konta magazynu.

Poniżej znajduje się zrzut ekranu klucza dostępu do usługi Azure Storage:



Rysunek 2.23. Klucz dostępu do magazynu Azure

Ważna uwaga

Terraform obsługuje inne typy uwierzytelniania na koncie magazynu, takie jak użycie tokena SAS lub SP. Aby uzyskać więcej informacji na temat konfigurowania Terraform dla zdalnego zaplecza azurerm, zapoznaj się z dokumentacją pod adresem <https://www.terraform.io/docs/backends/types/azurerm.html>.

3. Po zakończeniu konfiguracji Terraform można go uruchomić za pomocą nowego zdalnego zaplecza. To właśnie podczas wykonywania polecenia `init` Terraform inicjuje kontekst pliku stanu. Domyślnie polecenie `init` wykonywane jest w czasie wywoływania komendy `terraform init`.

Jeśli jednak wiele stanów Terraform jest używanych do zarządzania wieloma środowiskami, możliwe jest utworzenie kilku zdalnych konfiguracji zaplecza za pomocą uproszczonego kodu w pliku `.tf`:

```
terraform {
  backend "azurerm" {}
}
```

Teraz utwórz kilka plików `backend.tfvars`, które zawierają tylko właściwości backendów.

Te właściwości zaplecza to nazwa konta magazynu, nazwa kontenera obiektów blob i nazwa obiektu blob pliku stanu:

```
storage_account_name = "storageremotetf"
container_name       = "tfbackends"
key                  = "myappli.tfstate"
snapshot             = true
```

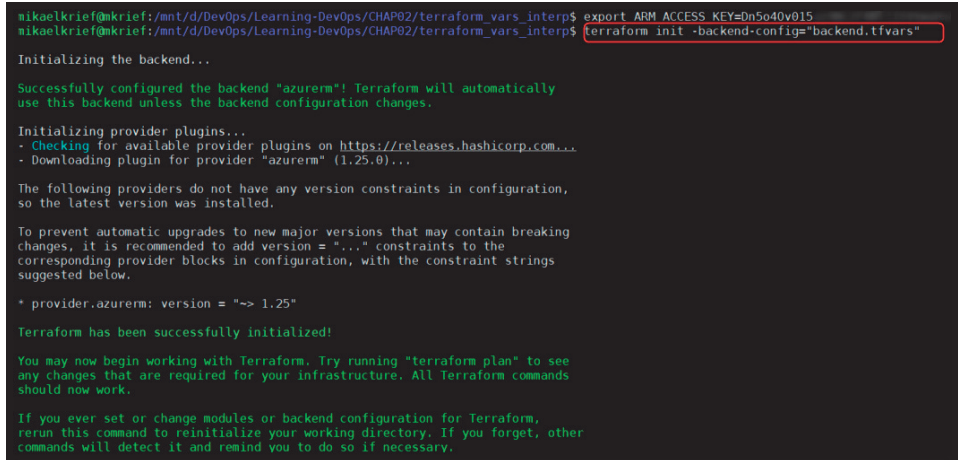
W tym scenariuszu, wykonując polecenie `init`, możemy określić backend. Plik `tfvars` używany jest z następującym poleceniem:

```
terraform init -backend-config="backend.tfvars"
```

Argument `-backend-config` jest ścieżką do pliku konfiguracyjnego zaplecza.

Wolę ten sposób postępowania, ponieważ pozwala mi oddzielić kod poprzez uzewnętrznienie wartości właściwości zaplecza w celu lepszej czytelności kodu.

Oto wykonanie Terraform:



```
mikaelkrief@mikrief:/mnt/d/DevOps/Learning-DevOps/CHAP02/terraform_vars_interp$ export ARM_ACCESS_KEY=Dn5o40v015
mikaelkrief@mikrief:/mnt/d/DevOps/Learning-DevOps/CHAP02/terraform_vars_interp$ terraform init -backend-config="backend.tfvars"

Initializing the backend...

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

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...
- Downloading plugin for provider "azurerm" (1.25.0)...

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

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

* provider.azurerm: version = "~> 1.25"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Rysunek 2.24. Polecenie `terraform init` z konfiguracją zaplecza

W tym wykonaniu możemy zaobserwować eksport zmiennej środowiskowej `ARM_ACCESS_KEY` wraz z poleceniem `terraform init`, które określa konfigurację zaplecza za pomocą opcji `-backend-config`.

Dzięki temu zdalnemu zapleczu plik stanu nie będzie już przechowywany lokalnie, ale na koncie magazynu, które jest przestrzenią współużytkowaną. Dzięki temu może być używany przez kilku użytkowników naraz. Jednocześnie to konto zapewnia bezpieczeństwo w celu ochrony poufnych danych pliku stanu i możliwość tworzenia kopii zapasowych lub przywracania plików stanu, które są zarówno istotnymi, jak i krytycznymi elementami Terraform.

Uwaga

Cały kod źródłowy tego rozdziału jest dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP02>, a ostateczny kod Terraform znajduje się w folderze `terraform_vars_interp`.

Podsumowanie

W tym rozdziale poświęconym Terraform dowiedzieliśmy się, że jego instalację można wykonać ręcznie lub za pomocą skryptów.

Aby zastosować Terraform, szczegółowo opisaliśmy różne kroki jego konfiguracji w celu udostępnienia infrastruktury platformy Azure przy użyciu usługi Azure SP.

Dodatkowo omówiliśmy krok po kroku jego lokalne wykonanie z głównymi opcjami poleceń, takimi jak `init`, `plan`, `apply` i `destroy`, wraz z jego cyklem życia w procesie CI/CD. Ten rozdział zakończyliśmy, przyglądając się ochronie pliku stanu w zdalnym zapleczu platformy Azure.

Dlatego Terraform jest narzędziem zgodnym z zasadami IaC. Kod Terraform jest czytelny i zrozumiały dla użytkowników, a jego wykonanie bardzo dobrze integruje się z potokiem CI/CD, co pozwala na automatyczne udostępnianie infrastruktury chmury.

W tej książce będziemy nadal omawiać Terraform. Opiszemy sposoby jego używania z Packerem, Azure Kubernetes Services i poruszymy temat redukcji przestojów.

W następnym rozdziale przyjrzymy się kolejnemu etapowi IaC, czyli zarządzaniu konfiguracją przy użyciu Ansible. Omówimy jego instalację, wykorzystanie do konfigurowania naszej maszyny wirtualnej i sposób ochrony kluczy za pomocą Ansible Vault.

Pytania

1. Jaki język jest używany przez Terraform?
2. Jaka jest rola Terraform?
3. Czy Terraform jest narzędziem skryptowym?
4. Które polecenie pozwala wyświetlić zainstalowaną wersję?
5. W przypadku korzystania z Terraform dla Azure jaka jest nazwa obiektu platformy Azure, który łączy Terraform z platformą Azure?
6. Jakie są trzy główne polecenia przepływu pracy Terraform?
7. Która komenda Terraform pozwala na niszczenie zasobów?
8. Jaka opcja została dodana do polecenia `apply`, aby zautomatyzować stosowanie zmian w infrastrukturze?
9. Do czego służy plik stanu Terraform?
10. Czy dobrą praktyką jest pozostawienie pliku stanu Terraform lokalnie? Jeśli nie, co należy zrobić?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o Terraform, oto kilka zasobów:

- Oficjalna dokumentacja Terraform — <https://www.terraform.io/>.
- Informacje o pobieraniu i instalacji Terraform — <https://www.terraform.io/downloads.html>.
- Dostawca Terraform Azure — <https://www.terraform.io/docs/providers/azurerm/index.html>.
- Oficjalna dokumentacja platformy Azure dla Terraform — <https://docs.microsoft.com/en-us/azure/terraform/terraform-overview>.
- Książka *Terraform Cookbook* — <https://www.packtpub.com/product/terraform-cookbook/9781800207554>.
- *Getting Started with Terraform, Second Edition* — <https://www.packtpub.com/networking-and-servers/getting-started-terraform-second-edition>.
- Nauka online Terraform — <https://learn.hashicorp.com/terraform>.

Używanie Ansible do konfigurowania infrastruktury IaaS

W poprzednim rozdziale omówiliśmy udostępnianie infrastruktury chmury Azure za pomocą Terraform. Jeśli ta infrastruktura zawiera **maszyny wirtualne (VM)**, to po ich udostępnieniu konieczne jest skonfigurowanie ich systemów i zainstalowanie całego niezbędnego oprogramowania. Taka konfiguracja będzie niezbędna do prawidłowego działania aplikacji, które będą hostowane na maszynie wirtualnej.

Dostępnych jest kilka narzędzi IaC (ang. *Infrastructure as Code*) do konfigurowania maszyn wirtualnych, a najbardziej znane to Ansible, Puppet, Chef, SaltStack i PowerShell DSC. Ansible firmy Red Hat (<https://www.ansible.com/overview/it-automation>) wyróżnia się wieloma zaletami, np.:

- Jest deklaratywny i używa łatwego do odczytania języka YAML.
- Działa tylko z jednym plikiem wykonywalnym.
- Konfiguracja Ansible nie wymaga instalowania agentów na maszynach wirtualnych.
- Żeby Ansible mógł się połączyć ze zdalnymi maszynami wirtualnymi, wymagane jest proste połączenie SSL/WinRM.
- Zawiera silnik szablonów i mechanizm do szyfrowania/odszyfrowywania wrażliwych danych.
- Jest idempotentny.

Główne przypadki użycia Ansible są następujące:

- Konfiguracja maszyny wirtualnej z middleware i utwardzaniem (ang. *hardening*), o czym dowiemy się w tym rozdziale.
- Udostępnianie infrastruktury, takiej jak Terraform, ale przy użyciu konfiguracji YAML.
- Badanie zgodności bezpieczeństwa, czyli sprawdzenie, czy konfiguracja systemu lub sieci jest zgodna z wymaganiami przedsiębiorstwa.

W tym rozdziale dowiemy się, jak zainstalować Ansible, a następnie użyć go do skonfigurowania maszyny wirtualnej ze zbiorem urządzeń (ang. *inventory*) i z playbookiem. Dowiemy się również, jak chronić poufne dane za pomocą Ansible Vault — zanim omówimy sposób korzystania z dynamicznego spisu maszyn na platformie Azure.

W tym rozdziale zostaną omówione następujące tematy:

- instalacja Ansible,
- tworzenie pliku inwentarza Ansible,
- uruchamianie pierwszego playbooka,
- uruchamianie Ansible,
- ochrona danych za pomocą Ansible Vault,
- korzystanie z dynamicznego pliku inwentarza dla infrastruktury Azure.

Wymagania techniczne

Aby rozpocząć, musisz spełnić następujące wymagania techniczne:

- Aby zainstalować Ansible, potrzebujemy systemu operacyjnego, takiego jak Red Hat, Debian, CentOS, macOS lub którykolwiek z BSD. Jeśli masz system Windows, możesz zainstalować podsystem Windows dla systemu Linux (ang. *Windows Subsystem for Linux* — WSL); zapoznaj się z dokumentacją pod adresem <https://docs.microsoft.com/en-us/windows/wsl/install-win10>.
- Na komputerze, na którym działa Ansible, musi być zainstalowany Python 2 (wersja 2.7) lub Python 3 (wersja 3.5+). Możesz go pobrać tutaj: <https://www.python.org/downloads/>. Więcej informacji na temat wymagań Ansible można znaleźć w dokumentacji tutaj: https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html#control-node-requirements.
- Playbook Ansible używa plików konfiguracyjnych YAML, więc każdy edytor kodu będzie z nim współpracował; my będziemy używać Visual Studio Code. Możesz go pobrać tutaj: <https://code.visualstudio.com/>.
- Większość tego rozdziału nie skupia się na konkretnym dostawcy chmury, z wyjątkiem ostatniej sekcji dotyczącej Azure. Potrzebna do tego będzie subskrypcja Azure, którą możemy pobrać bezpłatnie tutaj: <https://azure.microsoft.com/en-us/free/>.
- Aby uruchomić dynamiczny spis maszyn dla platformy Azure, musimy zainstalować zestaw SDK języka Azure Python: <https://docs.microsoft.com/en-us/azure/python/python-sdk-azure-install?view=azure-python>.

- Pełny kod źródłowy tego rozdziału jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP03>.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/3HdwbVc>.

Instalacja Ansible

Zanim zaczniemy korzystać z Ansible, musimy wiedzieć, w jakim systemie operacyjnym możemy go używać oraz jak go zainstalować i skonfigurować. Następnie musimy poznać niektóre zagadnienia potrzebne do pracy.

W tej sekcji przyjrzymy się, jak zainstalować Ansible na komputerze lokalnym lub serwerze i jak zintegrować Ansible z Azure Cloud Shell. Następnie porozmawiamy o różnych elementach, które składają się na Ansible. Na koniec skonfigurujemy Ansible.

Na początek dowiemy się, jak pobrać i zainstalować Ansible za pomocą automatycznego skryptu.

Instalacja Ansible za pomocą skryptu

W przeciwieństwie do Terraform, Ansible nie jest wieloplatformowy i można go zainstalować tylko na Red Hat, Debian, CentOS, macOS lub dowolnym BSD. Instalujemy go za pomocą skryptu, który różni się w zależności od systemu operacyjnego.

Aby zainstalować najnowszą wersję na Ubuntu, musimy uruchomić następujący skrypt w terminalu Bash:

```
sudo apt-get update
sudo apt-get install software-properties-common
sudo apt-add-repository --yes --update ppa:ansible/ansible
sudo apt-get install ansible
```

Ważna uwaga

Ten skrypt jest również dostępny tutaj: https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP03/install_ansible_ubuntu.sh.

Ten skrypt aktualizuje niezbędne pakiety, instaluje zależność `software-properties-common`, dodaje repozytorium Ansible i instaluje najnowszą wersję Ansible.

Aby zainstalować Ansible lokalnie na komputerze z systemem Windows, nie ma natywnego rozwiązania, ale można go zainstalować na lokalnej maszynie wirtualnej

Ważna uwaga

Skrypty instalacyjne Ansible dla wszystkich typów dystrybucji są dostępne tutaj: https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html#installing-ansible-on-specific-operating-systems.

VirtualBox lub WSL. WSL umożliwia programistom korzystającym z systemu operacyjnego Windows testowanie swoich skryptów i aplikacji bezpośrednio na stacjach roboczych bez konieczności instalowania maszyny wirtualnej.

Ważna uwaga

Przeczytaj ten artykuł, aby dowiedzieć się, jak zainstalować Ansible w lokalnym środowisku VirtualBox: <https://phoenixnap.com/kb/install-ansible-on-windows>. By uzyskać więcej informacji na temat WSL, przeczytaj dokumentację tutaj: <https://docs.microsoft.com/en-us/windows/wsl/about>.

Aby wiedzieć, czy Ansible został pomyślnie zainstalowany, możemy uruchomić następujące polecenie, by sprawdzić jego zainstalowaną wersję:

```
ansible --version
```

Wynik tego polecenia dostarcza pewnych informacji o zainstalowanej wersji Ansible, np.:

```
mikael@vmAnsible:~$ ansible --version
ansible 2.8.3
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/home/mikael/.ansible/plugins/modules', u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/dist-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.15+ (default, Oct 7 2019, 17:39:04) [GCC 7.4.0]
```

Rysunek 3.1. Polecenie ansible --version

Aby wyświetlić listę wszystkich poleceń i opcji Ansible, wykonaj polecenie `ansible` z argumentem `--help`:

```
ansible --help
```

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia.

Jak widać, instalacja Ansible na komputerze lokalnym lub zdalnym jest dość prosta i można ją zautomatyzować za pomocą skryptu. Jeśli wdrożymy infrastrukturę na platformie Azure, możemy również użyć Ansible, ponieważ jest on zintegrowany z Azure Cloud Shell.

Przyjrzyjmy się teraz, jak Ansible jest zintegrowany z Azure Cloud Shell.

```
~$ ansible --help
Usage: ansible <host-pattern> [options]

Define and run a single task 'playbook' against a set of hosts

Options:
  -a MODULE_ARGS, --args=MODULE_ARGS
                        module arguments
  --ask-vault-pass     ask for vault password
  -B SECONDS, --background=SECONDS
                        run asynchronously, failing after X seconds
                        (default=N/A)
  -C, --check          don't make any changes; instead, try to predict some
                        of the changes that may occur
  -D, --diff           when changing (small) files and templates, show the
                        differences in those files; works great with --check
  -e EXTRA_VARS, --extra-vars=EXTRA_VARS
                        set additional variables as key=value or YAML/JSON, if
```

Rysunek 3.2. Polecenie ansible --help

Integracja Ansible z Azure Cloud Shell

Jak dowiedzieliśmy się w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”, Azure Cloud Shell integruje narzędzia innych firm, których można używać na platformie Azure bez konieczności instalowania ich na maszynie wirtualnej. Wśród tych narzędzi jest Terraform, który szczegółowo omówiliśmy w poprzednim rozdziale, ale jest też Ansible, który Microsoft zintegrował, żeby umożliwić nam automatyczne konfigurowanie maszyn wirtualnych hostowanych na platformie Azure.

Aby korzystać z Ansible w chmurze Azure, musimy wykonać następujące czynności:

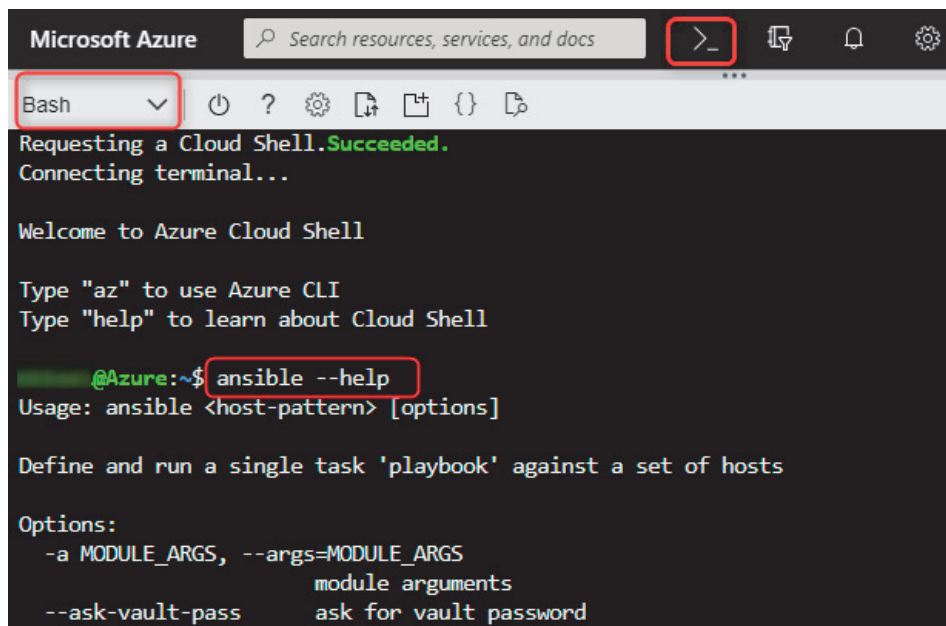
1. Połącz się z portalem Azure pod adresem <https://portal.azure.com>.
2. Otwórz Cloud Shell.
3. Wybierz tryb *Bash*.
4. W terminalu, który się otworzy, mamy teraz dostęp do wszystkich poleceń Ansible.

Poniższy zrzut ekranu przedstawia polecenie ansible w Azure Cloud Shell.

W ten sposób będzie można używać Ansible do programowania i testowania bez instalowania żadnego oprogramowania.

Ponadto Ansible ma moduły, które pozwalają nam udostępniać infrastrukturę Azure (takie jak Terraform, ale ten aspekt Ansible nie zostanie omówiony w tej książce), więc jego integracja z Azure Cloud Shell pozwala na uproszczone uwierzytelnianie.

Zanim zaczniemy używać Ansible, przejrzymy jego ważne koncepcje, które będą nam służyć w tym rozdziale.



The screenshot shows the Microsoft Azure Cloud Shell interface. At the top, there's a search bar and a dropdown menu set to 'Bash'. The terminal output shows the successful connection to the Cloud Shell and a welcome message. The user has entered the command 'ansible --help', which is highlighted with a red box. The output of the command shows the usage and options for the 'ansible' command.

```
Microsoft Azure Search resources, services, and docs >_
Bash
Requesting a Cloud Shell.Succeeded.
Connecting terminal...

Welcome to Azure Cloud Shell

Type "az" to use Azure CLI
Type "help" to learn about Cloud Shell

@Azure:~$ ansible --help
Usage: ansible <host-pattern> [options]

Define and run a single task 'playbook' against a set of hosts

Options:
  -a MODULE_ARGS, --args=MODULE_ARGS      module arguments
  --ask-vault-pass                          ask for vault password
```

Rysunek 3.3. Ansible w Azure Cloud Shell

Ważna uwaga

Szczegółowa dokumentacja dotycząca integracji Ansible z Azure Cloud Shell jest dostępna tutaj: <https://docs.microsoft.com/en-us/azure/ansible/ansible-run-playbook-in-cloudshell>.

Artefakty Ansible

Aby skonfigurować system, Ansible potrzebuje kilku głównych artefaktów:

- **Hosty** — są to systemy docelowe, które skonfiguruje Ansible; host może być również systemem lokalnym.
- **Inwentarz** — jest to plik w formacie INI lub YAML zawierający listę docelowych hostów, na których Ansible wykona działania konfiguracyjne. Takim inwentarzem może być również skrypt, tak jak w przypadku spisu dynamicznego.

Ważna uwaga

Przyjrzymy się, jak zaimplementować spis Ansible, w sekcji „Tworzenie pliku inwentarza Ansible”, natomiast sposób implementacji spisu dynamicznego omówimy w sekcji „Korzystanie z dynamicznego pliku inwentarza dla infrastruktury Azure”.

- **Playbook** — jest to skrypt konfiguracyjny Ansible, który zostanie wykonany w celu skonfigurowania hostów.

Ważna uwaga

Jak tworzyć playbooksi, nauczymy się w części „Uruchomienie pierwszego playbooka”, w dalszej części tego rozdziału.

Po nauczaniu się, jak zainstalować Ansible, przyjrzelśmy się podstawowym elementom Ansible, którymi są hosty, inwentarz i playbooksi. Teraz nauczymy się konfigurować Ansible.

Konfiguracja Ansible

Domyślnie konfiguracja Ansible znajduje się w `/etc/ansible/ansible`. Plik `cfg` jest tworzony podczas instalacji Ansible i zawiera kilka kluczy konfiguracyjnych, takich jak np.: połączenie SSL, użytkownik, protokół, transport.

Jak wspomnieliśmy wcześniej, ten plik jest tworzony domyślnie podczas instalacji Ansible. Aby ułatwić użytkownikowi rozpoczęcie pracy, umieszczana jest w nim początkowa treść. Ta treść zawiera wiele kluczy konfiguracyjnych, które są zakomentowane, aby nie były stosowane przez Ansible. Mogą być aktywowane w dowolnym momencie przez użytkownika.

Uwaga

Jeśli używamy Ansible w Azure Cloud Shell, musimy ręcznie utworzyć ten plik (`ansible.cfg`) na naszym dysku w chmurze Azure i ustawić zmienną środowiskową `ANSIBLE_CONFIG` na ścieżkę do utworzonego pliku. Dokumentacja tej zmiennej środowiskowej jest dostępna tutaj: https://docs.ansible.com/ansible/latest/reference_appendices/config.html#envvar-ANSIBLE_CONFIG.

Poniższy zrzut ekranu pokazuje wyciąg z pliku konfiguracyjnego `/etc/ansible/ansible.cfg` z kilkoma zakomentowanymi kluczami — symbolem #.

Jeśli chcemy zmienić domyślną konfigurację Ansible, możemy zmodyfikować ten plik.

Ważna uwaga

Więcej informacji na temat wszystkich kluczy konfiguracyjnych Ansible można znaleźć w oficjalnej dokumentacji: https://docs.ansible.com/ansible/latest/reference_appendices/config.html#ansible-configuration-settings.

```
# config file for ansible -- https://ansible.com/
# =====

# nearly all parameters can be overridden in ansible-playbook
# or with command line flags. ansible will read ANSIBLE_CONFIG,
# ansible.cfg in the current working directory, .ansible.cfg in
# the home directory or /etc/ansible/ansible.cfg, whichever it
# finds first

[defaults]

# some basic default values...

#inventory      = /etc/ansible/hosts
#library        = /usr/share/my_modules/
#module_utils   = /usr/share/my_module_utils/
#remote_tmp     = ~/.ansible/tmp
#local_tmp      = ~/.ansible/tmp
#plugin_filters_cfg = /etc/ansible/plugin_filters.yml
#forks          = 5
#poll_interval  = 15
#sudo_user      = root
#ask_sudo_pass  = True
#ask_pass       = True
#transport      = smart
#remote_port    = 22
#module_lang    = C
#module_set_locale = False
```

Rysunek 3.4. Plik konfiguracyjny Ansible

Możemy również przeglądać i modyfikować tę konfigurację za pomocą polecenia `ansible-config`. Na przykład aby wyświetlić plik konfiguracyjny Ansible, możemy wykonać następujące polecenie:

```
ansible-config view
```

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia.

W tej sekcji dowiedzieliśmy się, jak zainstalować Ansible, i zbadaliśmy niektóre z artefaktów Ansible. Na koniec przyjrzelśmy się różnym sposobom konfiguracji Ansible.

W następnej sekcji szczegółowo opiszemy statyczny spis Ansible i sposób jego tworzenia na hostach docelowych.

```
:/home/n # ansible-config view
# config file for ansible -- https://ansible.com/
# =====

# nearly all parameters can be overridden in ansible-playbook
# or with command line flags. ansible will read ANSIBLE_CONFIG,
# ansible.cfg in the current working directory, .ansible.cfg in
# the home directory or /etc/ansible/ansible.cfg, whichever it
# finds first

[defaults]
# some basic default values...

#inventory      = /etc/ansible/hosts
#library        = /usr/share/my_modules/
#module_utils   = /usr/share/my_module_utils/
#remote_tmp     = ~/.ansible/tmp
#local_tmp      = ~/.ansible/tmp
#plugin_filters_cfg = /etc/ansible/plugin_filters.yml
#forks          = 5
#poll_interval  = 15
#sudo_user      = root
#ask_sudo_pass  = True
#ask_pass       = True
#transport      = smart
#remote_port    = 22
#module_lang    = C
#module_set_locale = False
```

Rysunek 3.5. Wyświetlenie konfiguracji Ansible za pomocą CLI

Tworzenie pliku inwentarza Ansible

Inwentarz zawiera listę hostów, na których Ansible będzie wykonywał czynności administracyjne i konfiguracyjne.

Istnieją dwa rodzaje inwentarzy:

- **Inwentarz statyczny** — hosty są wymienione w pliku tekstowym w formacie INI (lub YAML); jest to podstawowy tryb inwentarza Ansible. Inwentaryzacja statyczna jest stosowana w przypadkach, gdy znamy adresy hostów (IP lub FQDN).
- **Inwentarz dynamiczny** — lista hostów jest generowana dynamicznie przez zewnętrzny skrypt (np. za pomocą skryptu Pythona). Inwentaryzacja dynamiczna jest używana, jeśli nie mamy adresów hostów, np. w przypadku infrastruktury składającej się ze środowisk na żądanie.

W tej części dowiemy się, jak utworzyć inwentarz statyczny w formacie `ini`, zaczynając od prostego przykładu. Następnie przyjrzymy się grupom i konfiguracji hostów.

Zacznijmy od nauki tworzenia statycznego pliku inwentarza.

Plik inwentarza

Aby Ansible mógł konfigurować hosty podczas uruchamiania playbooka, musi mieć plik zawierający listę hostów; jest to lista adresów IP lub **w pełni kwalifikowanej nazwy domeny** (ang. *fully qualified domain name* — **FQDN**) komputerów docelowych. Ta lista hostów jest zapisywana w statycznym pliku zwanym **plikiem inwentarza**.

Domyślnie Ansible zawiera plik inwentarza, który jest tworzony podczas instalacji; ten plik nazywa się `/etc/ansible/hosts` i zawiera kilka przykładów konfiguracji inwentarza. W naszym przypadku ręcznie utworzymy i wypełnimy ten plik w wybranym przez nas katalogu, takim jak `devopsansible`.

Zróbmy to krok po kroku:

1. Najpierw musimy utworzyć katalog za pomocą następującego podstawowego polecenia:

```
mkdir devopsansible
cd devopsible
```

2. Teraz utworzymy plik o nazwie `myinventory` (bez rozszerzenia), w którym zapiszemy adresy IP lub FQDN hostów docelowych, jak pokazano w poniższym przykładzie:

```
192.10.14.10
mywebserver.entreprise.com
localhost
```

Gdy Ansible zostanie uruchomiony na podstawie tego pliku inwentarza, wykona wszystkie żądane działania (playbook) na wszystkich hostach wymienionych w tym pliku.

Ważna uwaga

Więcej informacji na temat pliku inwentarza można znaleźć w dokumentacji pod adresem https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html.

Jeśli używasz Ansible w firmie, ten sam kod Ansible (lub playbook) zawiera akcje konfiguracyjne, które są wykonywane dla wszystkich maszyn wirtualnych aplikacji. Ponieważ te maszyny wirtualne mają różne role w aplikacji, np. rolę serwera WWW lub serwera bazy danych, musimy podzielić nasz spis, aby pogrupować maszyny wirtualne według ról funkcjonalnych.

Aby pogrupować maszyny wirtualne według roli w pliku inwentarza, zorganizujemy nasze maszyny wirtualne w grupy, które zostaną zapisane w nawiasach []. Tworzymy następujący plik:

```
[webserver]
192.10.20.31
mywebserver.example.com
[database]
192.20.34.20
```

W tym przykładzie zdefiniowaliśmy dwie grupy: `webserver` i `database`. Wszystkie hosty są rozdzielone pomiędzy te grupy.

W innym przykładzie możemy również pogrupować hosty według środowisk:

```
[dev]
192.10.20.31
192.10.20.32
[qa]
192.20.34.20
192.20.34.21
[prod]
192.10.12.10
192.10.12.11
```

W dalszej części tego rozdziału dowiemy się, jak te grupy będą wykorzystywane podczas tworzenia playbooków.

Teraz nauczymy się, jak uzupełnić nasz plik inwentarza konfiguracją hostów.

Konfigurowanie hostów w pliku inwentarza

Jak widzieliśmy, cała konfiguracja Ansible znajduje się w pliku *ansible.cfg*. Ta konfiguracja jest jednak ogólna i dotyczy wszystkich wykonań Ansible, a także połączenia z hostami.

W przypadku używania Ansible do konfigurowania maszyn wirtualnych w różnych środowiskach lub ról z różnymi uprawnieniami ważne jest, aby mieć różne konfiguracje połączeń — tzn. różnych użytkowników administracyjnych i różne klucze SSL dla środowisk. Z tego powodu możliwe jest nadpisanie domyślnej konfiguracji Ansible w pliku inwentarza poprzez skonfigurowanie określonych parametrów dla hosta zgodnie z definicją w pliku inwentarza.

Główne parametry konfiguracyjne, które można zmienić, są następujące:

- `ansible_user` — użytkownik, który łączy się ze zdalnym hostem.
- `ansible_port` — możliwa jest zmiana domyślnej wartości portu SSH.

- `ansible_host` — alias hosta.
- `ansible_connection` — typ połączenia ze zdalnym hostem; może to być połączenie Paramiko, SSH lub lokalne.
- `ansible_private_key_file` — klucz prywatny używany do łączenia się ze zdalnym hostem.

Ważna uwaga

Pełna lista parametrów jest dostępna w następującej dokumentacji: https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#list-of-behavioral-inventory-parameters.

Oto przykład pliku inwentarza, w którym skonfigurowaliśmy połączenie hostów:

```
[webserver]
webserver1 ansible_host=192.10.20.31 ansible_port=2222
webserver2 ansible_host=192.10.20.31 ansible_port=2222
[database]
database1 ansible_host=192.20.34.20
ansible_user=databaseuser
database2 ansible_host=192.20.34.21
ansible_user=databaseuser
[dev]
webserver1
database1
[qa]
webserver2
database2
```

W tym przykładzie można zobaczyć, że:

- Informacje o połączeniu zostały określone obok każdego hosta.
- Implementacja aliasu (taka jak `webserver1` i `webserver2`) jest używana w innej grupie (takiej jak grupa `qa` w tym przykładzie).

Po wdrożeniu pliku inwentarza Ansible dowiemy się teraz, jak go testować.

Testowanie pliku inwentarza

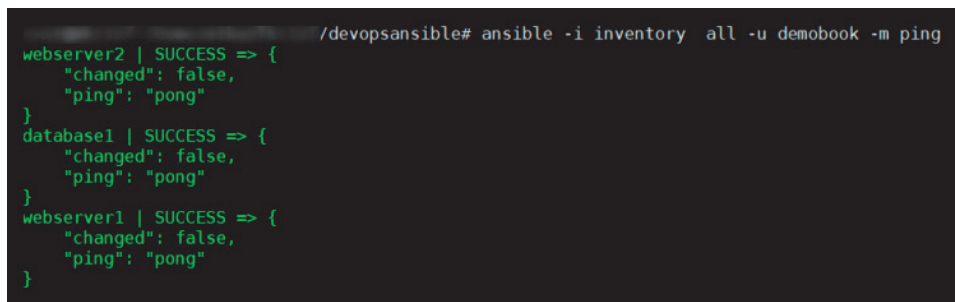
Po zapisaniu pliku inwentarza możliwe jest sprawdzenie, czy wszystkie wymienione hosty są dostępne z Ansible. W tym celu możemy wykonać następujące polecenie:

```
ansible -i inventory all -u demobook -m ping
```

Argument `-i` określa ścieżkę do pliku inwentarza, argument `-u` odpowiada zdalnej nazwie użytkownika używanej do łączenia się ze zdalnym komputerem, a `-m` określa

polecenie do wykonania. Tutaj wykonujemy polecenie ping na wszystkich maszynach z pliku inwentarza.

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia:



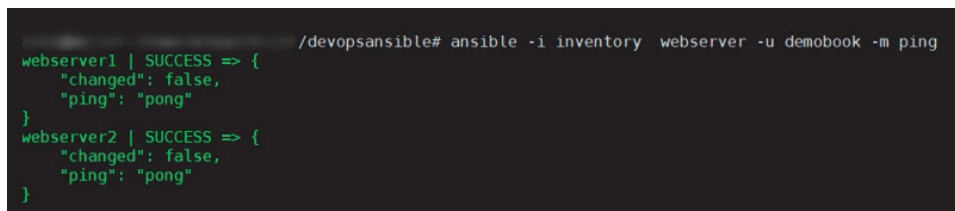
```
/devopsansible# ansible -i inventory all -u demobook -m ping
webserver2 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
database1 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
webserver1 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
```

Rysunek 3.6. Testowanie polecenia ansible ping z opcją all

Możemy również przetestować połączenie na hostach określonej grupy, wywołując to polecenie z nazwą grupy zamiast all. W naszym przypadku wykonamy to polecenie tak

```
ansible -i inventory webserver -u demobook -m ping
```

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia:



```
/devopsansible# ansible -i inventory webserver -u demobook -m ping
webserver1 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
webserver2 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
```

Rysunek 3.7. Testowanie polecenia ansible ping dla określonego hosta

W tej sekcji dowiedzieliśmy się, że Ansible wymaga pliku inwentarza do konfiguracji hostów. Następnie utworzyliśmy i przetestowaliśmy nasz pierwszy plik, zanim jeszcze nauczyliśmy się konfigurować ten plik.

W następnej sekcji dowiemy się, jak skonfigurować i napisać kod wykonujący akcję w playbooku Ansible.

Uruchomienie pierwszego playbooka

Jednym z istotnych elementów Ansible są jego playbooksi, ponieważ, jak określono we wstępie, zawierają one kod akcji lub zadań, które należy wykonać, aby skonfigurować maszynę wirtualną lub nią administrować.

Rzeczywiście, po udostępnieniu maszyny wirtualnej należy ją skonfigurować i zainstalować całe oprogramowanie pośrednie potrzebne do uruchamiania aplikacji, które będą hostowane na tej maszynie. Niezbędne jest również wykonywanie zadań administracyjnych dotyczących konfiguracji katalogów i dostępu do nich.

W tej sekcji zobaczymy, z czego składa się playbook, poznamy jego moduły i dowiemy się, jak go ulepszyć za pomocą ról.

Zacznijmy uczyć się, jak utworzyć prosty playbook.

Tworzenie prostego playbooka

Kod playbooka jest napisany w YAML, języku deklaratywnym, który pozwala nam łatwo zwizualizować kroki konfiguracji.

Aby zrozumieć, jak wygląda playbook, spójrzmy na prosty i klasyczny przykład, czyli instalowanie serwera NGINX na maszynie wirtualnej Ubuntu. Wcześniej utworzyliśmy działający katalog *devopsansible*, wewnątrz którego utworzymy plik *playbook.yml* i wstawimy do niego następujący kod:

```
---
- hosts: all
  tasks:
    - name: install and check nginx latest version
      apt: name=nginx state=latest
    - name: start nginx
      service:
        name: nginx
        state: started
---
```

Przyjrzyjmy się temu szczegółowo:

- Plik YAML zaczyna się i kończy opcjonalnymi znakami ---.
- Właściwość - hosts zawiera listę hostów do skonfigurowania. Tutaj zapisaliśmy wartość tej właściwości jako all, aby zainstalować NGINX na wszystkich maszynach wirtualnych wymienionych w naszym pliku inwentarza. Jeśli chcemy zainstalować go tylko dla określonej grupy, np. dla grupy webserver, odnotujemy to w następujący sposób:

- hosts: webserver

- Następnie wskazujemy listę zadań lub akcji do wykonania na tych maszynach wirtualnych za pomocą właściwości `tasks`.
- Pod elementem `tasks` opisujemy listę zadań i każdemu z nich nadajemy nazwę, która służy jako etykieta, we właściwości `name`. Pod nazwą wywołujemy funkcję, która ma zostać wykonana przy użyciu *modułów Ansible* i ich właściwości. W naszym przykładzie wykorzystaliśmy dwa moduły:
 - `apt` — pozwala nam pobrać pakiet (polecenie `apt-get`), aby uzyskać najnowszą wersję pakietu `nginx`.
 - `service` — pozwala nam na uruchomienie lub zatrzymanie usługi — w tym przykładzie, aby uruchomić usługę `NGINX`.

Widzimy, że do korzystania z Ansible nie jest wymagana żadna wiedza z zakresu programowania ani pisania skryptów IT; ważne jest, aby znać listę działań, które możesz wykonać na maszynach wirtualnych potrzebnych do konfiguracji. Playbook Ansible jest zatem sekwencją działań, które są zakodowane w modułach Ansible.

Właśnie widzieliśmy, że zadania używane w playbookach wykorzystują moduły. W następnej sekcji przedstawimy krótki przegląd modułów i ich wykorzystania.

Opis modułów Ansible

W poprzednim podrozdziale dowiedzieliśmy się, że w playbookach Ansible używamy modułów. Dzięki temu Ansible jest dziś tak popularny i istnieje długa lista natywnych modułów publicznych dostarczanych przez Ansible (ponad 200). Pełna lista dostępna jest tutaj: https://docs.ansible.com/ansible/latest/collections/index_module.html.

Moduły te pozwalają nam wykonywać wszystkie zadania i operacje, które należy wykonać na maszynie wirtualnej w celu jej konfiguracji i administrowania nią, bez konieczności pisania jakichkolwiek wierszy kodu lub skryptów.

W ramach własnych potrzeb możemy również tworzyć niestandardowe moduły i publikować je wewnętrznie w prywatnym rejestrze. Więcej informacji można znaleźć tutaj: https://docs.ansible.com/ansible/latest/dev_guide/developing_modules_general.html.

Teraz, gdy nauczyliśmy się, jak tworzyć prosty playbook i jak korzystać z modułów, jeszcze bardziej go ulepszymy — dzięki rołom.

Ulepszanie playbooków za pomocą ról

Podczas konfigurowania maszyn wirtualnych zauważamy pewną powtarzalność zadań dla każdej aplikacji. Na przykład kilka aplikacji wymaga identycznej instalacji NGINX, którą należy wykonać w ten sam sposób.

W przypadku Ansible to powtórzenie będzie wymagało zduplikowania kodu z playbooka, jak widać w naszym przykładzie w sekcji „Uruchomienie pierwszego playbooka”, na kilka playbooków (ponieważ każda aplikacja zawiera playbook). Aby uniknąć tego powielania, a tym samym zaoszczędzić czas, uniknąć błędów i ujednolicić czynności związane z instalacją i konfiguracją, możemy umieścić taki kod w katalogu zwanym *rolą* (ang. *role*), z którego może korzystać kilka playbooków.

Aby utworzyć rolę `nginx` odpowiadającą naszemu przykładowi, utworzymy następujący katalog i drzewo plików w naszym katalogu `devopsansible`:



Rysunek 3.8. Folder Ansible

Następnie w pliku `main.yml`, który znajduje się w `tasks`, skopiujemy i wkleimy następujący kod z naszego playbooka:

- name: install and check nginx latest version
apt: name=nginx state=latest
- name: start nginx
service:
name: nginx
state: started

Następnie zmodyfikujemy nasz playbook, aby używać tej roli z następującą zawartością:

```
---
- hosts: webserver
  roles:
    - nginx
```

W taki sposób dostarczymy listę ról (nazwy katalogów ról), które będą używane. Tak więc rola `nginx` jest teraz scentralizowana i może być używana w kilku playbookach bez konieczności przepisywania kodu.

Poniżej znajduje się kod playbooka, który konfiguruje serwer WWW maszyny wirtualnej z Apache i inną maszyną wirtualną zawierającą bazę danych MySQL:

```
---
- hosts: webserver
  roles:
    - php
    - apache
- hosts: database
  roles:
    - mysql
```

Ważna uwaga

Aby uzyskać więcej informacji na temat tworzenia ról, przeczytaj oficjalną dokumentację pod adresem https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_roles.html.

Jednak zanim zaczniemy tworzyć rolę, możemy skorzystać z Ansible Galaxy (<https://galaxy.ansible.com/>), który zawiera dużą liczbę ról dostarczanych przez społeczność i pokrywa wiele potrzeb konfiguracyjnych i administracyjnych.

W swojej firmie możemy również tworzyć niestandardowe role i publikować je w prywatnej galaktyce. Więcej informacji można znaleźć tutaj: https://docs.ansible.com/ansible/latest/dev_guide/developing_modules_general.html.

W tej sekcji dowiedzieliśmy się, jak utworzyć playbook, a także jak go ulepszyć za pomocą ról. Wszystkie nasze artefakty są w końcu gotowe, więc teraz będziemy mogli uruchomić Ansible.

Uruchomienie Ansible

Do tej pory nauczyliśmy się instalować Ansible, opisaliśmy hosty w pliku inwentarza i skonfigurowaliśmy nasz playbook Ansible. Teraz możemy uruchomić Ansible, aby skonfigurować nasze maszyny wirtualne.

W tym celu uruchomimy narzędzie Ansible za pomocą polecenia `ansible-playbook`:

```
ansible-playbook -i inventory playbook.yml
```

Podstawowe opcje tego polecenia są następujące:

- Argument `-i` opisujący ścieżkę do pliku inwentarza.
- Ścieżka do playbooka.

Poniżej znajduje się wykonanie tego polecenia:

```
/devopsansible# ansible-playbook -i inventory playbook.yml --check

PLAY [webserver] *****

TASK [Gathering Facts] *****
ok: [webserver1]
ok: [webserver2]

TASK [nginx : install and check nginx latest version] *****
changed: [webserver1]
changed: [webserver2]

TASK [nginx : start nginx] *****
changed: [webserver2]
changed: [webserver1]

PLAY RECAP *****
webserver1      : ok=3    changed=2    unreachable=0    failed=0
webserver2      : ok=3    changed=2    unreachable=0    failed=0
```

Rysunek 3.9. Wykonywanie playbooka Ansible

Wykonanie tego polecenia powoduje zastosowanie playbooka do hostów w pliku inwentarza. Dzieje się to w kilku krokach:

1. Zbieranie faktów — Ansible sprawdza, czy hosty są osiągalne.
2. Playbook jest wykonywany na hostach.
3. `PLAY Recap` — jest to stan zmian, które zostały wykonane na każdym hoście; wartość tego statusu może być następująca:

Tabela 3.1. Wartości `PLAY Recap`

| | |
|--------------------------|---|
| <code>ok</code> | Liczba zadań playbooka, które zostały poprawnie zastosowane na hoście |
| <code>changed</code> | Liczba zmian, które zostały zastosowane |
| <code>unreachable</code> | Host jest nieosiągalny |
| <code>failed</code> | Na tym hoście uruchomienie się nie powiodło |

Jeśli będziemy musieli uaktualnić nasz playbook, aby dodać lub zmodyfikować oprogramowanie na naszych maszynach wirtualnych, to podczas drugiego wykonania Ansible

zobaczymy, że nie została ponownie zastosowana pełna konfiguracja maszyn wirtualnych — tylko różnice.

Poniższy zrzut ekranu pokazuje drugie wykonanie Ansible bez wykonanych zmian w naszym playbooku:

```
/devopsansible# ansible-playbook -i inventory playbook.yml

PLAY [webserver] *****
TASK [Gathering Facts] *****
ok: [webserver1]
ok: [webserver2]

TASK [nginx : install and check nginx latest version] *****
ok: [webserver2]
ok: [webserver1]

TASK [nginx : start nginx] *****
ok: [webserver1]
ok: [webserver2]

PLAY RECAP *****
webserver1      : ok=3    changed=0    unreachable=0    failed=0
webserver2      : ok=3    changed=0    unreachable=0    failed=0
```

Rysunek 3.10. Wykonanie playbooka Ansible z inną informacją

Tutaj widzimy, że Ansible nie wykonał zmian w hostach (changed=0).

Możemy również dodać kilka przydatnych opcji do tego polecenia, aby zapewnić:

- podgląd zmian Ansible przed ich zastosowaniem,
- więcej logów w danych wyjściowych.

Te opcje są ważne nie tylko w fazie tworzenia playbooka, ale także w czasie debugowania — w przypadku błędów podczas wykonywania.

Przyjrzyjmy się teraz, jak korzystać z tych opcji podglądu.

Korzystanie z podglądu lub z opcji testowej pracy (ang. dry run)

Podczas kodowania playbooka Ansible często musimy testować różne kroki bez stosowania ich bezpośrednio do infrastruktury. W związku z tym bardzo przydatne, zwłaszcza podczas automatyzacji konfiguracji maszyny wirtualnej za pomocą Ansible, jest posiadanie podglądu jej wykonania. Dzięki temu możemy sprawdzić, czy składnia playbooka zachowuje dobrą spójność z konfiguracją systemu, która już istnieje na hoście.

Dzięki Ansible można sprawdzić wykonanie playbooka na hostach, dodając opcję `--check` do polecenia:

```
ansible-playbook -i inventory playbook.yml --check
```

Oto przykład wykonania testowego:

```
/devopsansible# ansible-playbook -i inventory playbook.yml --check

PLAY [webserver] *****
TASK [Gathering Facts] *****
ok: [webserver1]
ok: [webserver2]

TASK [nginx : install and check nginx latest version] *****
changed: [webserver1]
changed: [webserver2]

TASK [nginx : start nginx] *****
changed: [webserver2]
changed: [webserver1]

PLAY RECAP *****
webserver1      : ok=3    changed=2    unreachable=0    failed=0
webserver2      : ok=3    changed=2    unreachable=0    failed=0
```

Rysunek 3.11. Wykonanie testu playbooka Ansible

Dzięki tej opcji Ansible nie stosuje zmian konfiguracyjnych do hosta; sprawdza tylko i wyświetla podgląd zmian, które zostały wprowadzone na hostach.

Ważna uwaga

Więcej informacji na temat opcji `--check` można znaleźć w następującej dokumentacji: https://docs.ansible.com/ansible/latest/user_guide/playbooks_checkmode.html.

Właśnie dowiedzieliśmy się, że Ansible pozwala nam sprawdzić playbook przed zastosowaniem go do hosta; trzeba też wiedzieć, że istnieją inne narzędzia do testowania funkcjonalności playbooka (bez konieczności symulowania jego wykonania), takie jak Vagrant firmy HashiCorp.

Vagrant pozwala nam bardzo szybko utworzyć lokalne środowisko testowe składające się z maszyn wirtualnych, na którym możemy uruchomić nasze playbooki i zobaczyć wyniki. Więcej informacji na temat korzystania z Ansible i Vagranta można znaleźć w następującej dokumentacji: https://docs.ansible.com/ansible/latest/scenario_guides/guide_vagrant.html.

Właśnie dowiedzieliśmy się, jak wyświetlić podgląd zmian, które zostaną zastosowane przez Ansible. Przyjrzyjmy się teraz, jak zwiększyć poziom logowania procesu wykonywania Ansible.

Zwiększanie poziomu logowania

W przypadku błędów możliwe jest dodanie większej liczby logów wyjściowych poprzez dodanie opcji `-v`, `-vvv` lub `-vvvv` do polecenia Ansible.

Opcja `-v` włącza podstawowy tryb logowania, opcja `-vvv` włącza tryb szczegółowy z większą liczbą danych wyjściowych, a opcja `-vvvv` dodaje tryb szczegółowy i informacje o połączeniu.

Wykonanie następującego polecenia powoduje zastosowanie instrukcji playbooka i wyświetlenie większej liczby informacji przy użyciu dodanej opcji `-v`:

```
ansible-playbook -i inventory playbook.yml -v
```

Może to być przydatne do debugowania w przypadku wystąpienia błędów Ansible.

Ważna uwaga

Pełna dokumentacja polecenia `ansible-playbook` jest dostępna tutaj: <https://docs.ansible.com/ansible/2.4/ansible-playbook.html>.

Właśnie dowiedzieliśmy się, jak uruchomić Ansible z jego plikiem inwentarza i z playbookiem, badając niektóre opcje, które pozwalają na:

- podgląd zmian, które zostaną wprowadzone przez Ansible,
- zwiększenie poziomu logów w celu ułatwienia debugowania.

W następnej sekcji omówimy bezpieczeństwo danych podczas korzystania z Ansible Vault.

Ochrona danych za pomocą Ansible Vault

Do tej pory nauczyliśmy się używać Ansible z plikiem inwentarza zawierającym listę hostów do skonfigurowania oraz z playbookiem zawierającym kod działań konfiguracyjnych dla hosta. Jednak we wszystkich narzędziach IaC konieczne będzie wyodrębnienie pewnych danych, które są specyficzne dla kontekstu lub środowiska.

W tej sekcji przyjrzymy się, jak używać zmiennych w Ansible i jak chronić poufne dane za pomocą Ansible Vault.

Aby zilustrować ochronę zmiennych, uzupełnimy nasz przykład instalacją serwera bazy danych MySQL na serwerze.

Zacznijmy od przyjrzenia się wykorzystaniu i użyteczności zmiennych w Ansible.

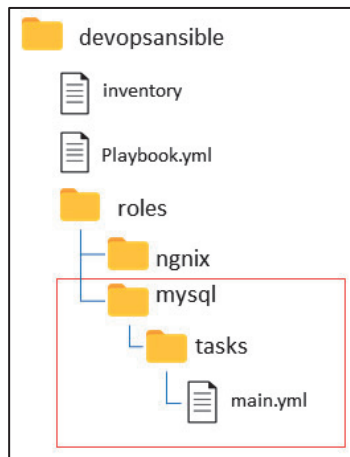
Używanie zmiennych w Ansible w celu lepszej konfiguracji

Podczas wdrażania infrastruktury z IaC używany kod często składa się z dwóch części:

- Część opisująca elementy lub zasoby składające się na infrastrukturę.
- Kolejna część, która różnicuje właściwości tej infrastruktury w różnych środowiskach.

Różnicowanie wykonania w zależności od środowiska odbywa się za pomocą zmiennych. Ansible ma cały system, który pozwala nam korzystać ze zmiennych w playbookach.

Aby dowiedzieć się, jak używać zmiennych w Ansible, uzupełnimy nasz kod i dodamy rolę `mysql` do katalogu `roles` w następującej strukturze drzewa:



Rysunek 3.12. Struktura folderów ról w Ansible

W pliku `main.yml` tej roli napiszemy następujący kod:

```
---
- name: Update apt cache
  apt: update_cache=yes cache_valid_time=3600
- name: Install required software
  apt: name="{{ packages }}" state=present
vars:
packages:
- python-mysqldb
```

```
- mysql-server
- name: Create mysql user
mysql_user:
  name={{ mysql_user }}
  password={{ mysql_password }}
  priv=*.*:ALL
  state=present
```

W tym kodzie niektóre statyczne informacje zostały zastąpione zmiennymi. Są to:

- `packages` — zawiera listę pakietów do zainstalowania. Jest zdefiniowana w powyższym kodzie.
- `mysql_user` — zawiera nazwę administratora bazy danych MySQL.
- `mysql_password` — zawiera hasło administratora bazy danych MySQL.

Zadania tej roli są następujące:

- aktualizacja pakietów,
- instalacja serwera MySQL i pakietów Python MySQL,
- tworzenie użytkownika MySQL.

Ważna uwaga

Pełny kod źródłowy tej roli jest dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP03/devopsansible/roles/mysql>.

Jak widać, w zadaniu tworzenia użytkownika umieściliśmy zmienne `mysql_user` i `mysql_password` jako nazwę użytkownika i hasło. W związku z tym informacje te mogą być różne w zależności od środowiska lub mogą być tworzone dynamicznie podczas uruchamiania Ansible.

Aby zdefiniować wartości tych zmiennych, utworzymy katalog `group_vars`, który będzie zawierał wszystkie wartości zmiennych dla każdej grupy zdefiniowanej w naszym pliku inwentarza.

Następnie w katalogu `group_vars` utworzymy podkatalog `database` odpowiadający grupie bazy danych zdefiniowanej w pliku inwentarza oraz plik `main.yml`.

W pliku `main.yml` umieszczamy żądane wartości tych zmiennych w następujący sposób:

```
---
mysql_user: mydbuserdef
mysql_password: mydbpassworddef
```


Na koniec uzupełnimy nasz playbook, wywołując rolę `mysql` przez dodanie następującego kodu:

```
- hosts: database
  become: true
  roles:
    - mysql
```

Ansible możemy uruchomić tym samym poleceniem co poprzednio: `ansible-playbook -i inventory playbook.yml`. Generowane są następujące dane wyjściowe:

```
root@DESKTOP-9Q2U73J:/d/devopsansible# ansible-playbook -i inventory playbook.yml
PLAY [webserver] *****
TASK [Gathering Facts] *****
ok: [webserver1]
ok: [webserver2]

TASK [nginx : install and check nginx latest version] *****
ok: [webserver1]
ok: [webserver2]

TASK [nginx : start nginx] *****
ok: [webserver1]
ok: [webserver2]

PLAY [database] *****
TASK [Gathering Facts] *****
ok: [database1]

TASK [mysql : Update apt cache] *****
ok: [database1]

TASK [mysql : Install required software] *****
changed: [database1]

TASK [mysql : Create mysql user] *****
changed: [database1]

PLAY RECAP *****
database1      : ok=4    changed=2    unreachable=0    failed=0
webserver1     : ok=3    changed=0    unreachable=0    failed=0
webserver2     : ok=3    changed=0    unreachable=0    failed=0
```

Rysunek 3.13. Wywołanie MySQL w playbooku Ansible

Ansible zaktualizował serwer bazy danych, wprowadzając dwie zmiany: listę pakietów do zainstalowania i administratora MySQL. Właśnie nauczyliśmy się używać zmiennych w Ansible, ale ich zawartość jest dostępna w kodzie, co rodzi problemy z bezpieczeństwem.

Ważna uwaga

Aby uzyskać więcej informacji na temat zmiennych Ansible, przeczytaj pełną dokumentację tutaj: https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html.

Teraz nauczmy się, jak używać Ansible Vault do ochrony zmiennych w playbooku.

Ochrona wrażliwych danych za pomocą Ansible Vault

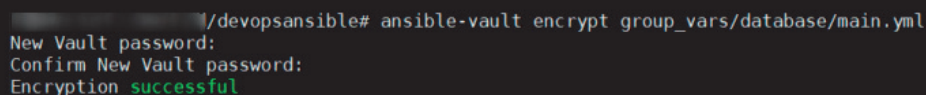
Konfiguracja systemu często wymaga poufnych informacji, które nie powinny trafić w niepowołane ręce. W narzędziu Ansible znajduje się narzędzie podrzędne o nazwie **Ansible Vault**, które chroni dane przesyłane do Ansible za pośrednictwem playbooków.

W tej sekcji dowiemy się, jak wykorzystać Ansible Vault w celu zaszyfrowania i odszyfrowania informacji na temat użytkownika MySQL.

Pierwszym krokiem jest zaszyfrowanie pliku *group_vars/database/main.yml*, który zawiera wartości zmiennych. Robimy to, wykonując następujące polecenie:

```
ansible-vault encrypt group_vars/database/main.yml
```

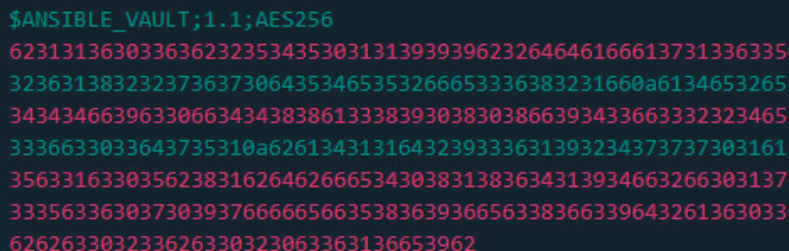
Ansible Vault prosi o podanie hasła, które będzie wymagane do odszyfrowania pliku, a następnie pokazuje wykonanie tego polecenia, szyfrując zawartość pliku:



```
/devopsansible# ansible-vault encrypt group_vars/database/main.yml
New Vault password:
Confirm New Vault password:
Encryption successful
```

Rysunek 3.14. Szyfrowanie za pomocą Ansible Vault

Po wykonaniu tego polecenia zawartość pliku jest szyfrowana, więc wartości nie są już jasne. Oto jego zawartość:



```
$ANSIBLE_VAULT;1.1;AES256
623131363033636232353435303131393939623264646166613731336335
3236313832323736373064353465353266653336383231660a6134653265
343434663963306634343838613338393038303866393433663332323465
3336633033643735310a62613431316432393336313932343737303161
356331633035623831626462666534303831383634313934663266303137
333563363037303937666665663538363936656338366339643261363033
626263303233626330323063363136653962
```

Rysunek 3.15. Zaszyfrowany plik

Aby odszyfrować plik w celu jego modyfikacji, musisz wykonać polecenie deszyfrowania:

```
ansible-vault decrypt group_vars/database/main.yml
```

Ansible Vault żąda hasła, które zostało użyte do zaszyfrowania pliku. Plik ponownie stanie się czytelny.

W procesie automatyzacji lepiej jest przechowywać hasło w pliku w chronionej lokalizacji; np. w pliku `~/.vault_pass.txt`.

Następnie, aby zaszyfrować plik zmiennej za pomocą tego pliku, musimy wykonać polecenie `ansible-vault` i dodać opcję `--vault-password-file`:

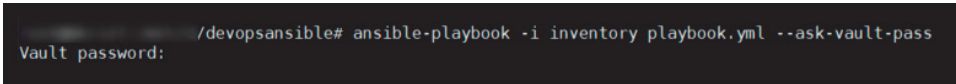
```
ansible-vault encrypt group_vars/database/main.yml --vault-password-file  
~/.vault_pass.txt
```

Teraz, gdy plik został zaszyfrowany, a dane są chronione, uruchomimy Ansible.

W **trybie interaktywnym** uruchomimy następujące polecenie:

```
ansible-playbook -i inventory playbook.yml --ask-vault-pass
```

Ansible prosi użytkownika o wprowadzenie hasła, jak pokazano na poniższym zrzucie ekranu:



```
/devopsansible# ansible-playbook -i inventory playbook.yml --ask-vault-pass  
Vault password:
```

Rysunek 3.16. Odszyfrowanie pliku za pomocą Ansible Vault

W **trybie automatycznym** — czyli w potoku CI/CD — możemy dodać parametr `-vault-password-file` ze ścieżką do pliku zawierającego hasło do odszyfrowania danych:

```
ansible-playbook -i inventory playbook.yml --vault-password-file  
~/.vault_pass.txt
```

Dzięki temu uruchomiliśmy Ansible z danymi, które nie są już czytelne w kodzie — za pomocą polecenia `ansible-vault`.

Ważna uwaga

Cały kod źródłowy pliku inwentarza, playbooka i ról jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP03/devopsansible>.

W tej sekcji dowiedzieliśmy się, jak chronić poufne dane w naszych playbookach za pomocą narzędzia `ansible-vault`. Zaszyfrowaliśmy i odszyfrowaliśmy pliki, a następnie ponownie uruchomiliśmy Ansible z tymi zaszyfrowanymi plikami.

W dalszej części dowiemy się, jak korzystać z Ansible z dynamicznym plikiem inwentarza.

Korzystanie z dynamicznego pliku inwentarza dla infrastruktury Azure

Podczas konfigurowania infrastruktury składającej się z kilku maszyn wirtualnych wraz ze środowiskami efemerycznymi, które są budowane na żądanie, często się obserwuje, że utrzymywanie statycznej inwentaryzacji, jak widzieliśmy w sekcji „Tworzenie pliku inwentarza Ansible”, może szybko stać się skomplikowane, a jej konserwacja zajmuje dużo czasu.

Aby rozwiązać ten problem, Ansible umożliwia dynamiczne tworzenie pliku inwentarza poprzez wywołanie skryptu (np. w Pythonie) dostarczanego przez dostawców chmury lub skryptu, który możemy opracować samodzielnie, a który zwraca zawartość pliku inwentarza.

W tej sekcji przyjrzymy się różnym sposobom używania Ansible do konfigurowania maszyn wirtualnych na platformie Azure przy użyciu dynamicznego pliku inwentarza. Zaczniemy:

1. Pierwszym krokiem jest skonfigurowanie Ansible, aby mieć dostęp do zasobów platformy Azure. W tym celu utworzymy jednostkę Azure SP w usłudze Azure AD dokładnie w taki sam sposób jak w przypadku Terraform (zob. sekcję „Konfigurowanie Terraform dla platformy Azure” w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”). Następnie musimy wyeksportować informacje o czterech identyfikatorach jednostki usługi do następujących zmiennych środowiskowych:

```
export AZURE_SUBSCRIPTION_ID=<ID subskrypcji>
export AZURE_CLIENT_ID=<ID klienta>
export AZURE_SECRET=<Klucz klienta>
export AZURE_TENANT=<ID dzierżawcy>
```

Ważna uwaga

Aby uzyskać więcej informacji na temat zmiennych środowiskowych platformy Azure dla Ansible, zapoznaj się z dokumentacją platformy Azure tutaj: https://docs.ansible.com/ansible/latest/scenario_guides/guide_azure.html.

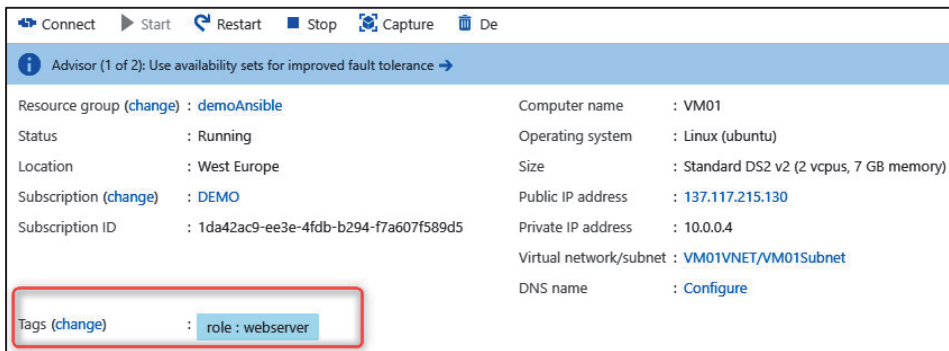
2. Następnie, aby móc generować plik inwentaryzacji z grupami i filtrować maszyny wirtualne, lepiej jest dodać tagi do maszyn wirtualnych. Tagi można dodawać za pomocą Terraform, polecenia `az cli` lub skryptu Azure PowerShell.

Oto przykładowy skrypt wykorzystujący polecenie `az cli`:

```
az resource tag --tags role=webserver -n VM01 -g
demoAnsible --resource-type "Microsoft.Compute/virtualMachines"
```

Powyższy skrypt dodaje tag `role` o wartości `webserver` do maszyny wirtualnej VM01. Następnie musimy wykonać tę samą operację na maszynie wirtualnej VM02 (wystarczy zmienić wartość parametru `-n` na VM02 w tym skrypcie).

Poniższy zrzut ekranu przedstawia tag maszyny wirtualnej w Azure Portal:



Rysunek 3.17. Tag role na platformie Azure

Teraz musimy dodać do naszej maszyny wirtualnej tag `database` za pomocą tego skryptu:

```
az resource tag --tags role=database -n VM04 -g
demoAnsible --resource-type "Microsoft.Compute/virtualMachines"
```

Ten skrypt dodaje do VM04 tag `role`, który ma wartość `database`.

Ważna uwaga

Dokumentację polecenia `az cli` do zarządzania tagami platformy Azure można znaleźć tutaj: <https://docs.microsoft.com/fr-fr/cli/azure/resource?view=azure-cli-latest&viewFallbackFrom=azure-cli-latest.md#az-resource-tag>.

3. Aby skorzystać z dynamicznego pliku inwentarza na platformie Azure, musimy wykonać następujące czynności:
 - Zainstaluj moduł Ansible Azure na komputerze za pomocą następującego skryptu:

```
wget -q https://raw.githubusercontent.com/ansible-collections/azure/
dev/requirements-azure.txt;
pip3 install -r requirements-azure.txt;
```

- Możemy również zainstalować moduł Azure za pomocą Ansible Galaxy, wykonując następujące polecenie:

```
ansible-galaxy collection install azure.azcollection
```

Uwaga

Aby uzyskać więcej informacji na temat *kolekcji platformy Azure*, zapoznaj się z dokumentacją pod adresem <https://galaxy.ansible.com/azure/azcollection>.

Utwórz nowy plik o nazwie *inv.azure_rm.yml* (nazwa tego pliku musi kończyć się na *azure_rm*) i zapisz w tym pliku następującą konfigurację:

- Użyj wtyczki *azure_rm*.
- Zezwól na grupowanie zwróconej listy maszyn wirtualnych według tagów *role*.
- Filtruj tylko w grupie zasobów *demoAnsible*.

Zawartość tego pliku będzie wyglądać następująco:

```
plugin: azure_rm
include_vm_resource_groups:
- demoAnsible
auth_source: auto
keyed_groups:
- key: tags.role
leading_separator : false
```

Ważna uwaga

Pełny kod źródłowy pliku *inv.azure_rm.yml* jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP03>.

4. Po skonfigurowaniu wszystkich artefaktów dla naszego dynamicznego pliku inwentarza Ansible na platformie Azure dobrze jest przetestować jego funkcjonalność, co obejmuje wykonanie następujących czynności:
 - A. Zapewnienie braku błędów wykonania.
 - B. Zapewnienie, że połączenie i uwierzytelnianie w naszym środowisku Azure są wykonywane poprawnie.
 - C. Zapewnienie jego wykonania zwraca maszyny wirtualne platformy Azure z naszej infrastruktury.

Ważna uwaga

Jak wspomniano w sekcji „Wymagania techniczne”, przed uruchomieniem poniższych poleceń musimy mieć zainstalowany na komputerze moduł Azure Python.

Aby przeprowadzić ten test, wykonaj następujące polecenie:

```
ansible-inventory -i inv.azure_rm.yml --list
```

Pozwala nam ono wyświetlić skrypt inwentarzowy w formacie listy.

Oto mały przykładowy ekran z tego wykonania:

```
root@LP-FYL22X2: /Learning-DevOps-Second-Edition/CHAP03/devopsansible# ansible-inventory -i inv.azure_rm.yml --list
{
  "_meta": {
    "hostvars": {
      "bookvm01_73ec": {
        "ansible_host": "20.61.2.21",
        "availability_zone": [
          ""
        ],
        "computer_name": "bookvm01",
        "default_inventory_hostname": "bookvm01_73ec",
        "id": "/subscriptions/.../resourceGroups/demoAnsible/providers/Microsoft.Compute/virtualMachines/bookvm01",
        "image": {
          "id": "/subscriptions/.../resourceGroups/demoAnsible/providers/Microsoft.Compute/galleries/demo/images/linux/versions/1.0.0"
        },
        "location": "westeurope",
        "mac_address": "08-4C-8D-88-FE-24",
        "name": "bookvm01",
        "network_interface": "bookvm01i663",
        "network_interface_id": "/subscriptions/.../resourceGroups/demoAnsible/providers/Microsoft.Network/networkInterfaces/bookvm01i663",
        "os_disk": {
          "name": "bookvm01_disk1_3d7f0532b1f4d1a98d5ddb47b65d8af",
          "operating_system_type": "linux"
        },
        "os_profile": {
          "system": "linux"
        }
      }
    }
  }
}
```

Rysunek 3.18. Dynamiczna lista inwentarza maszyn wirtualnych Ansible

Możemy również wyświetlić ten spis w formie wykresu, uruchamiając to samo polecenie, ale z opcją `--graph`:

```
root@LP-FYL22X2: /Learning-DevOps-Second-Edition/CHAP03/devopsansible# ansible-inventory -i inv.azure_rm.yml --graph
@all:
  |--@database:
  | |--bookvm2_2f86
  |--@ungrouped:
  |--@webserver:
  | |--bookvm01_73ec
  | |--bookvm1_6823
```

Rysunek 3.19. Dynamiczna lista inwentaryzacji maszyn wirtualnych Ansible pogrupowanych według ról

Dzięki opcji `--graph` uzyskujemy lepszą wizualizację maszyn wirtualnych według ich tagów.

Po zakończeniu testu możemy przejść do ostatniego kroku, czyli wykonania Ansible z dynamicznym plikiem inwentarza.

5. Po przetestowaniu naszego dynamicznego pliku inwentarza na platformie Azure wystarczy uruchomić go z Ansible, używając tagów, które zastosowaliśmy do maszyn wirtualnych. W tym celu musimy uruchomić nasz playbook za pomocą następującego polecenia:

```
ansible-playbook playbook.yaml -i inv.azure_rm.yml -u demobook -ask-pass
```

Ważna uwaga

W naszym laboratorium używamy VM z nazwą użytkownika i hasłem, dlatego w poprzednim poleceniu używamy parametru `-u` (dla nazwy użytkownika VM) oraz parametru `-ask-pass` (aby zapytać o hasło użytkownika VM). Ale zaleca się używanie kluczy publicznych/prywatnych SSH zamiast hasła.

Poniższy zrzut ekranu przedstawia wykonanie playbooka Ansible z dynamicznym plikiem inwentarza:

```
root@LP-FVL72X2: /Learning-DevOps-Second-Edition/CHAP03/devopsansible# ansible-playbook playbook.yml -i inv.azure_rm.yml -u denobook --ask-pass
SSH password:
PLAY [webserver] *****
TASK [Gathering Facts] *****
ok: [bookvm01_73ec]
ok: [bookvm1_6823]
TASK [nginx : install and check nginx latest version] *****
ok: [bookvm01_73ec]
ok: [bookvm1_6823]
TASK [nginx : start nginx] *****
ok: [bookvm1_6823]
ok: [bookvm01_73ec]
```

Rysunek 3.20. Dynamiczne wykonanie pliku inwentarza Ansible

Od teraz za każdym razem, gdy maszyna wirtualna w naszej infrastrukturze Azure będzie miała tag `role=webserver`, zostanie on automatycznie uwzględniony przez dynamiczną inwentaryzację, więc żadne modyfikacje kodu nie będą konieczne.

Ważna uwaga

Aby poznać inne sposoby korzystania z dynamicznych inwentaryzacji na platformie Azure, zapoznaj się z dokumentacją platformy Azure pod adresem <https://docs.microsoft.com/en-us/azure/developer/ansible/dynamic-inventory-configure?tabs=azure-cli>.

Korzystając z dynamicznej inwentaryzacji, możemy w pełni wykorzystać skalowalność chmury dzięki automatycznej konfiguracji VM, bez konieczności wprowadzania zmian w kodzie.

W tej sekcji dowiedzieliśmy się, jak korzystać z dynamicznej inwentaryzacji na platformie Azure, wdrażając jej konfigurację i tworząc skrypt służący do odzyskiwania danych przed wykonaniem dynamicznej inwentaryzacji za pomocą Ansible.

Podsumowanie

W tym rozdziale zobaczyliśmy, że Ansible jest bardzo potężnym i kompletnym narzędziem, które pozwala nam zautomatyzować konfigurację serwerów i administrowanie nimi. Do pracy wykorzystuje plik inwentarza zawierający listę hostów do skonfigurowania i playbook, w którym zakodowana jest lista czynności konfiguracyjnych.

Role, moduły i zmienne pozwalają też na lepsze zarządzanie kodem playbooka. Ansible ma również rozwiązanie, które chroni wrażliwe dane playbooka. Wreszcie, w przypadku dynamicznych środowisk, pisanie inwentaryzacji można uprościć, wdrażając dynamiczne inwentaryzacje.

W kolejnym rozdziale dowiemy się, jak zoptymalizować wdrażanie infrastruktury z wykorzystaniem Packera do tworzenia szablonów serwerów.

Pytania

1. Jaka jest rola Ansible opisana w tym rozdziale?
2. Czy możemy zainstalować Ansible w systemie operacyjnym Windows?
3. Jakie dwa artefakty, które badaliśmy w tym rozdziale, są potrzebne do uruchomienia Ansible?
4. Jak nazywa się opcja, która została dodana do polecenia `ansible-playbook`, a która służy do podglądu zmian, które zostaną zastosowane?
5. Jak nazywa się narzędzie służące do szyfrowania i deszyfrowania danych Ansible?
6. W przypadku korzystania z dynamicznej inwentaryzacji na platformie Azure za pomocą jakich właściwości maszyn wirtualnych skrypt inwentaryzacji zwraca listę maszyn wirtualnych?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o Ansible, oto kilka zasobów:

- Dokumentacja Ansible — <https://docs.ansible.com/ansible/latest/index.html>.
- Film przedstawiający szybkie omówienie tematu — https://www.ansible.com/resources/videos/quick-start-video?extIdCarryOver=true&sc_cid=701f200000010H6uAAG.

- Ansible w dokumentacji platformy Azure — <https://docs.microsoft.com/en-us/azure/ansible/>.
- Rozszerzenie Ansible dla Visual Studio Code — <https://marketplace.visualstudio.com/search?term=ansible&target=VSCode&category=All%20categories&sortBy=Relevance>.
- Książka *Mastering Ansible* — <https://www.packtpub.com/virtualization-and-cloud/mastering-ansible-third-edition>.
- Szkolenie Ansible — <https://www.ansible.com/resources/webinars-training>.

Optymalizacja wdrażania infrastruktury za pomocą Packera

Rozdział

4

W poprzednich rozdziałach dowiedzieliśmy się, jak udostępniać infrastrukturę chmury za pomocą Terraform, a następnie kontynuowaliśmy automatyczną konfigurację maszyn wirtualnych za pomocą Ansible. Ta automatyzacja pozwala nam czerpać korzyści z rzeczywistej poprawy produktywności i bardzo widocznej oszczędności czasu.

Jednak pomimo tej automatyzacji zauważamy następujące problemy:

- Konfiguracja maszyny wirtualnej może być bardzo czasochłonna, ponieważ zależy od jej zabezpieczenia i od oprogramowania, które zostanie zainstalowane i skonfigurowane na tej maszynie.
- W każdym środowisku lub aplikacji wersje oprogramowania nie są identyczne, ponieważ skrypt automatyzacji niekoniecznie jest identyczny lub utrzymywany w czasie. Stąd np. środowisko produkcyjne, które jest bardziej krytyczne, będzie częściej dysponować najnowszą wersją pakietów, co nie ma miejsca w środowiskach przedprodukcyjnych. W takiej sytuacji często spotykamy się z problemami z zachowaniem aplikacji w środowisku produkcyjnym.
- Procedura zachowania zgodności konfiguracji i bezpieczeństwa nie jest często stosowana ani aktualizowana.

Aby rozwiązać te problemy, wszyscy dostawcy usług w chmurze zintegrowali ze swoją platformą usługę, która umożliwia im tworzenie lub generowanie niestandardowych obrazów maszyn wirtualnych. Te obrazy zawierają wszystkie konfiguracje maszyn wirtualnych wraz z poprawkami bezpieczeństwa i niezbędnym oprogramowaniem. Mogą służyć jako podstawa do tworzenia maszyn wirtualnych dla aplikacji.

Korzyści z używania tych obrazów są następujące:

- Udostępnianie maszyny wirtualnej za pomocą obrazu jest bardzo szybkie.
- Każda maszyna wirtualna ma jednolitą konfigurację, a przede wszystkim zawiera poprawki bezpieczeństwa.

Wśród narzędzi IaC znajduje się Packer (narzędzie HashiCorp), który umożliwia tworzenie obrazów maszyn wirtualnych z pliku (lub szablonu).

W tym rozdziale dowiemy się, jak zainstalować Packera w różnych trybach. Omówimy składnię szablonów Packera w celu tworzenia niestandardowych obrazów maszyn wirtualnych na platformie Azure, które używają skryptów lub playbooków Ansible.

Szczegółowo opiszemy wykonanie Packera za pomocą tych szablonów w formatach JSON i HCL. Na koniec zobaczymy, jak Terraform wykorzystuje obrazy utworzone przez Packera. W tym rozdziale zrozumiemy, że Packer to proste narzędzie, które upraszcza tworzenie maszyn wirtualnych w procesie DevOps i bardzo dobrze integruje się z Terraform.

W tym rozdziale omówimy:

- opis Packera,
- tworzenie szablonów Packera za pomocą skryptów,
- tworzenie szablonów Packera przy użyciu Ansible,
- uruchamianie Packera,
- tworzenie szablonów Packera w formacie HCL,
- korzystanie z obrazów utworzonych przez Packera za pomocą Terraform.

Wymagania techniczne

W tym rozdziale wyjaśnimy, jak używać Packera do tworzenia obrazu maszyny wirtualnej w infrastrukturze chmurowej Azure. Potrzebna będzie więc subskrypcja platformy Azure, którą można bezpłatnie pobrać tutaj: <https://azure.microsoft.com/en-us/free/>.

W sekcji „Tworzenie szablonów Packera przy użyciu Ansible” nauczymy się tworzyć szablony Packera, które używają Ansible. Będziesz więc musiał zainstalować Ansible na swoim komputerze i zrozumieć, jak działa. Jest to szczegółowo opisane w rozdziale 3, „Używanie Ansible do konfigurowania infrastruktury IaaS”.

Ostatnia część rozdziału zawiera przykład użycia Terraform z obrazem Packera; do jego zastosowania konieczne będzie zainstalowanie Terraform i zrozumienie jego

działania, co szczegółowo opisano w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”.

Cały kod źródłowy tego rozdziału jest dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP04>.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/35exxSo>.

Opis Packera

Packer jest częścią pakietu narzędzi open source HashiCorp (oficjalna strona Packera: <https://www.packer.io/>). Jest to narzędzie wiersza poleceń typu open source, które pozwala nam tworzyć niestandardowe obrazy maszyn wirtualnych dowolnego systemu operacyjnego (te obrazy są również nazywane **szablonami** — ang. *templates*) na kilku platformach z pliku JSON.

Działanie Packera jest proste: opiera się na podstawowym systemie operacyjnym dostarczonym przez różnych dostawców chmury i konfiguruje tymczasową maszynę wirtualną, wykonując skrypty opisane w szablonie JSON lub HCL. Następnie z tej tymczasowej maszyny wirtualnej Packer generuje niestandardowy obraz, gotowy do użycia, służący do udostępniania maszyn wirtualnych.

Oprócz obrazów maszyn wirtualnych Packer udostępnia również inne rodzaje obrazów, takie jak obrazy Dockera lub obrazy Vagranta.

Po tym krótkim omówieniu Packera przyjrzymy się różnym trybom instalacji.

Instalacja Packera

Packer, podobnie jak Terraform, jest narzędziem wieloplatformowym i można go zainstalować w systemach Windows, Linux lub macOS. Instalacja Packera jest prawie taka sama jak w przypadku instalacji Terraform (zob. rozdział 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”) i można ją przeprowadzić na dwa sposoby: ręcznie lub za pomocą skryptu.

Instalacja ręczna

Aby ręcznie zainstalować Packera, wykonaj następujące czynności:

1. Przejdź do oficjalnej strony narzędzia (<https://www.packer.io/downloads.html>) i pobierz pakiet odpowiadający Twojemu systemowi operacyjnemu.
2. Po pobraniu rozpakuj i skopiuj plik binarny do katalogu wykonawczego (np. do `c:\Packer`).

3. Następnie zmienna środowiskowa PATH musi być ustawiona ze ścieżką do katalogu binarnego.

Uwaga

Szczegółowe instrukcje dotyczące aktualizacji zmiennej środowiskowej PATH w systemie Windows można znaleźć w tym artykule: <https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/>, a w przypadku systemu Linux zapoznaj się z tym: <https://www.techrepublic.com/article/how-to-add-directories-to-your-path-in-linux/>.

Teraz, gdy nauczyliśmy się ręcznie instalować Packera, przyjrzyjmy się dostępnym opcjom instalacji za pomocą skryptu.

Instalacja za pomocą skryptu

Możliwe jest również zainstalowanie Packera automatycznie, za pomocą skryptu, który można zainstalować na zdalnym serwerze i używać w procesie CI/CD. Rzeczywiście, Packer może być używany lokalnie, jak zobaczymy w tym rozdziale, ale jego prawdziwym celem jest integracja z potokiem CI/CD. Ten automatyczny potok DevOps umożliwia tworzenie i publikowanie jednolitych obrazów maszyn wirtualnych, które zagwarantują integralność oprogramowania i bezpieczeństwa maszyn wirtualnych opartych na tych obrazach.

Zobaczmy strukturę tych skryptów dla różnych systemów operacyjnych, czyli Linux, Windows i macOS.

Instalowanie Packera za pomocą skryptu w systemie Linux

Aby zainstalować binarkę Packera w systemie Linux, mamy dwa rozwiązania:

- Pierwszym rozwiązaniem jest zainstalowanie Packera za pomocą następującego skryptu:

```
PACKER_VERSION="1.7.3" #Zaktualizuj do żądanej wersji
curl -Os https://releases.hashicorp.com/packer/${PACKER_VERSION}/
↳packer_${PACKER_VERSION}_linux_amd64.zip \
&& curl -Os https://releases.hashicorp.com/packer/${PACKER_VERSION}/
↳packer_${PACKER_VERSION}_SHA256SUMS \
&& curl https://keybase.io/hashicorp/pgp_keys.asc | gpg --import \
&& curl -Os https://releases.hashicorp.com/packer/${PACKER_VERSION}/
↳packer_${PACKER_VERSION}_SHA256SUMS.sig \
&& gpg --verify packer_${PACKER_VERSION}_SHA256SUMS.sig packer_${
↳PACKER_VERSION}_SHA256SUMS \
&& shasum -a 256 -c packer_${PACKER_VERSION}_SHA256SUMS 2>&1 | grep
↳"${PACKER_VERSION}_linux_amd64.zip:sOK" \
&& unzip -o packer_${PACKER_VERSION}_linux_amd64.zip -d /usr/local/bin
```

Uwaga

Kod tego skryptu jest również dostępny tutaj: https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP04/install_packer.sh.

Ten skrypt wykonuje następujące czynności:

- Pobiera pakiet Packera w wersji 1.7.3 i sprawdza sumę kontrolną.
- Rozpakowuje i kopiuje pakiet do lokalnego katalogu, `/usr/local/bin` (domyślnie ten folder znajduje się w zmiennej środowiskowej `PATH`).

Poniżej znajduje się zrzut ekranu wykonania skryptu dla instalacji Packera w systemie Linux:

```
/CHAP04# sh install_packer.sh
Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 1696 100 1696 0 0 3981 0 --:--:-- --:--:-- --:--:-- 3971
gpg: key 51852D87348FFC4C: "HashiCorp Security <security@hashicorp.com>" not changed
gpg: Total number processed: 1
gpg: unchanged: 1
gpg: Signature made Thu Apr 11 20:30:03 2019 DST
gpg: using RSA key 91A6E7F85D05C65630BEF18951852D87348FFC4C
gpg: Good signature from "HashiCorp Security <security@hashicorp.com>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg: There is no indication that the signature belongs to the owner.
Primary key fingerprint: 91A6 E7F8 5D05 C656 30BE F189 5185 2D87 348F FC4C
packer 1.4.0 linux_amd64.zip: OK
Archive: packer_1.4.0_linux_amd64.zip
inflating: /usr/local/bin/packer
```

Rysunek 4.1. Wykonanie skryptu instalacyjnego Packera

Zaletą tego rozwiązania jest to, że możemy wybrać folder instalacyjny Packera i zastosować go w różnych dystrybucjach Linuksa, ponieważ korzysta z popularnych narzędzi, takich jak `curl` i `unzip`.

- Drugim rozwiązaniem instalacji Packera w systemie Linux jest użycie menedżera pakietów `apt` za pomocą następującego skryptu dla dystrybucji Ubuntu:

```
sudo apt-get update && sudo apt-get install -y gnupg
↪software-properties-common curl \
&& curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key
↪add - \
&& sudo apt-add-repository "deb [arch=amd64]
↪https://apt.releases.hashicorp.com $(lsb_release -cs) main" \
&& sudo apt-get update && sudo apt-get install packer
```

Ten skrypt wykonuje następujące czynności:

- Dodaje do `apt` repozytorium HashiCorp.
- Aktualizuje lokalne repozytorium.
- Pobiera Packer CL.

Ważna uwaga

Aby uzyskać więcej informacji na temat tego skryptu i instalacji Packera w innych dystrybucjach, przeczytaj dokumentację pod adresem <https://learn.hashicorp.com/tutorials/packer/get-started-install-cli?in=packer/docker-get-started> i przejdź do zakładki *Linux*.

Instalowanie Packera za pomocą skryptu w systemie Windows

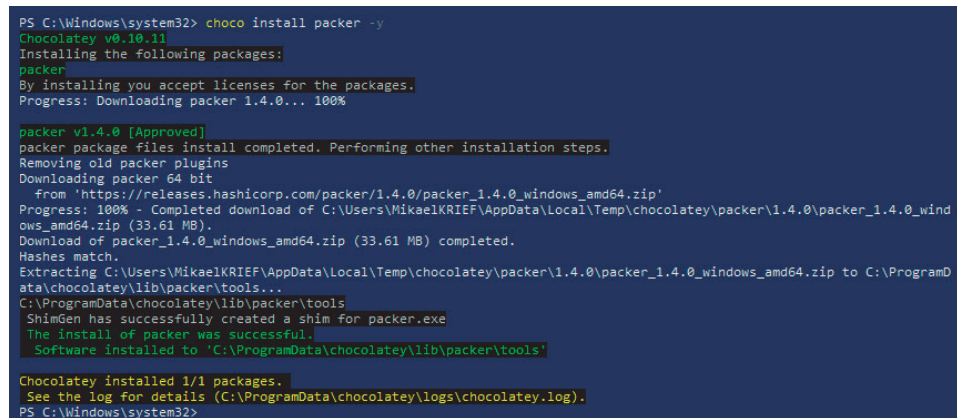
W systemie Windows możemy użyć **Chocolatey**, który jest menedżerem pakietów oprogramowania. Chocolatey to darmowy menedżer pakietów publicznych, taki jak NuGet lub npm, ale dedykowany do oprogramowania. Jest szeroko stosowany do automatyzacji oprogramowania na serwerach Windowsa lub nawet na komputerach lokalnych.

Oficjalna strona Chocolatey znajduje się tutaj: <https://chocolatey.org/>, a dokumentacja instalacji znajduje się tutaj: <https://chocolatey.org/install>.

Po zainstalowaniu Chocolatey wystarczy uruchomić następujące polecenie w PowerShell lub w narzędziu CMD:

```
choco install packer -y
```

Poniżej znajduje się zrzut ekranu instalacji Packera dla Windowsa za pomocą Chocolatey:



```
PS C:\Windows\system32> choco install packer -y
Chocolatey v0.10.11
Installing the following packages:
packer
By installing you accept licenses for the packages.
Progress: Downloading packer 1.4.0... 100%

packer v1.4.0 [Approved]
packer package files install completed. Performing other installation steps.
Removing old packer plugins
Downloading packer 64 bit
from 'https://releases.hashicorp.com/packer/1.4.0/packer_1.4.0_windows_amd64.zip'
Progress: 100% - Completed download of C:\Users\MikaelKRIEF\AppData\Local\Temp\chocolatey\packer\1.4.0\packer_1.4.0_wi
ows_amd64.zip (33.61 MB).
Download of packer_1.4.0_windows_amd64.zip (33.61 MB) completed.
Hashes match.
Extracting C:\Users\MikaelKRIEF\AppData\Local\Temp\chocolatey\packer\1.4.0\packer_1.4.0_windows_amd64.zip to C:\ProgramD
ata\chocolatey\lib\packer\tools...
C:\ProgramData\chocolatey\lib\packer\tools
ShimGen has successfully created a shim for packer.exe
The install of packer was successful.
Software installed to 'C:\ProgramData\chocolatey\lib\packer\tools'

Chocolatey installed 1/1 packages.
See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).
PS C:\Windows\system32>
```

Rysunek 4.2. Instalowanie Packera przy użyciu Chocolatey

Wykonanie polecenia `choco install packer -y` instaluje najnowszą wersję Packera za pomocą Chocolatey.

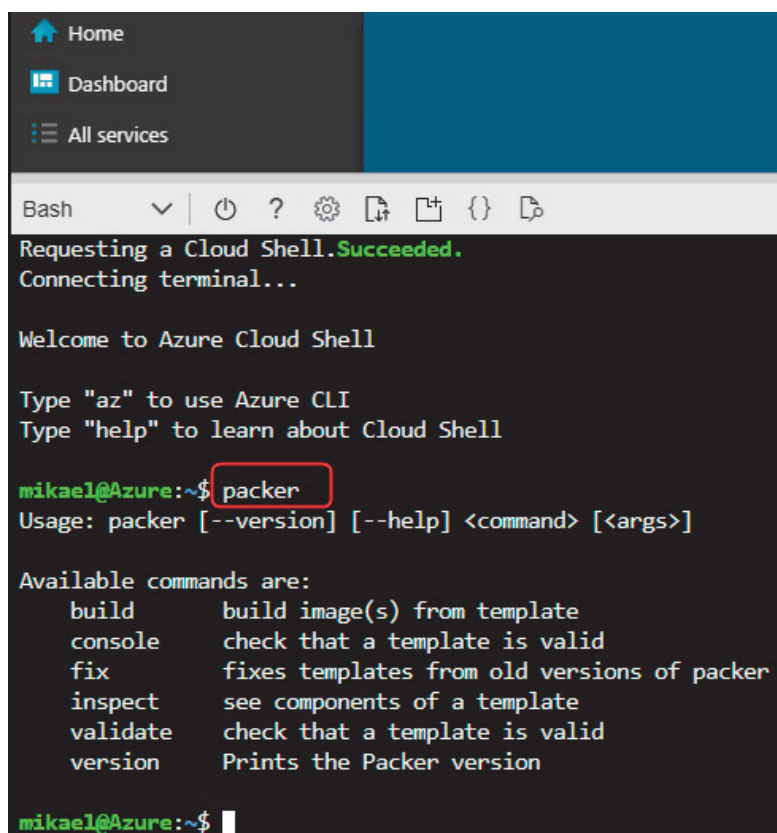
Instalowanie Packera za pomocą skryptu w systemie macOS

Aby zainstalować Packera w systemie macOS, możemy użyć **Homebrew**, menedżera pakietów macOS (<https://brew.sh/>), wykonując następujące polecenie w naszym terminalu:

```
brew install packer
```

Integracja Packera z Azure Cloud Shell

Podobnie jak Terraform, którego szczegółowy opis poznaliśmy w sekcji „Integracja Terraform z Azure Cloud Shell” w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”, Packer jest również zintegrowany z Azure Cloud Shell, jak pokazano na poniższym zrzucie ekranu:



```
Home
Dashboard
All services

Bash
Requesting a Cloud Shell.Succeeded.
Connecting terminal...

Welcome to Azure Cloud Shell

Type "az" to use Azure CLI
Type "help" to learn about Cloud Shell

mikael@Azure:~$ packer
Usage: packer [--version] [--help] <command> [<args>]

Available commands are:
  build      build image(s) from template
  console    check that a template is valid
  fix        fixes templates from old versions of packer
  inspect    see components of a template
  validate   check that a template is valid
  version    Prints the Packer version

mikael@Azure:~$
```

Rysunek 4.3. Packer w Azure Cloud Shell

Teraz, gdy widzieliśmy już instalację Packera na różnych systemach operacyjnych i jego integrację z Azure Cloud Shell, sprawdzimy jego zainstalowaną wersję.

Sprawdzanie instalacji Packera

Po zainstalowaniu możemy sprawdzić zainstalowaną wersję Packera, uruchamiając następujące polecenie:

```
packer --version
```

Polecenie to wyświetla zainstalowaną wersję Packera:

```
PS C:\Users\mkrief> packer --version  
1.7.3
```

Rysunek 4.4. Polecenie pokazujące wersję Packera

Aby zobaczyć wszystkie opcje Packera, możemy wykonać następujące polecenie:

```
packer --help
```

Po wykonaniu zobaczymy listę dostępnych opcji, jak pokazano na poniższym zrzucie ekranu:

```
PS C:\Users\mkrief> packer --help  
Usage: packer [--version] [--help] <command> [<args>]  
  
Available commands are:  
  build          build image(s) from template  
  console        creates a console for testing variable interpolation  
  fix            fixes templates from old versions of packer  
  fmt            Rewrites HCL2 config files to canonical format  
  hcl2_upgrade   transform a JSON template into an HCL2 configuration  
  init           Install missing plugins or upgrade plugins  
  inspect        see components of a template  
  validate       check that a template is valid  
  version        Prints the Packer version
```

Rysunek 4.5. Polecenie help Packera

Właśnie widzieliśmy procedurę ręcznej instalacji Packera i instalacji za pomocą skryptu w różnych systemach operacyjnych, a także jego integrację z Azure Cloud Shell.

Teraz napiszemy szablon, by przy użyciu skryptów utworzyć obraz maszyny wirtualnej na platformie Azure za pomocą programu Packer.

Tworzenie szablonów Packera dla maszyn wirtualnych Azure za pomocą skryptów

Jak wspomniano we wstępie, aby utworzyć obraz maszyny wirtualnej, Packer opiera się na pliku (szablonie) zapisanym w formacie JSON lub w języku **HashiCorp Configuration Language (HCL)**, który został wprowadzony w Packerze od wersji 1.5.0 (przełączaj następujący post na blogu, aby uzyskać więcej informacji: <https://www.packer.io/guides/hcl>). Najpierw przyjrzymy się strukturze i kompozycji tego szablonu, a następnie przejdziemy do praktycznego tworzenia szablonu, który utworzy obraz maszyny wirtualnej na platformie Azure.

Struktura szablonu Packera

Szablon programu Packer składa się z kilku głównych sekcji, takich jak `builders`, `provisioners` i `variables`. Format JSON szablonu jest następujący:

```
{
  "variables": {
    // lista zmiennych
    ...
  },
  "builders": [
    {
      // właściwości builders
      ...
    }
  ],
  "provisioners": [
    {
      // lista skryptów do wykonania w celu udostępnienia obrazu
      ...
    }
  ]
}
```

Przyjrzymy się szczegółom każdej sekcji.

Sekcja builders

Sekcja `builders` jest obowiązkowa i zawiera wszystkie właściwości, które definiują obraz i jego lokalizację, takie jak jego nazwa, typ obrazu, dostawca chmury, na którym zostanie wygenerowany obraz, informacje o połączeniu z chmurą, obraz bazowy do wykorzystania i inne właściwości, które są specyficzne dla typu obrazu.

Oto przykładowy kod sekcji builders:

```
{
  "builders": [{
    "type": "azure-rm",
    "client_id": "xxxxxxx",
    "client_secret": "xxxxxxx",
    "subscription_id": "xxxxxxxxxx",
    "tenant_id": "xxxxxx",
    "os_type": "Linux",
    "image_publisher": "Canonical",
    "image_offer": "UbuntuServer",
    "location": "westus"
    .....
  }]
}
```

W tym przykładzie sekcja builders definiuje obraz, który będzie przechowywany w chmurze platformy Azure i jest oparty na systemie operacyjnym Linux Ubuntu. Konfigurujemy również klucze uwierzytelniające dla chmury.

Uwaga

Dokumentacja w sekcji builders jest tutaj: <https://www.packer.io/docs/builders>.

Jeśli chcemy utworzyć ten sam obraz, ale na kilku dostawcach, możemy wskazać w tym samym pliku szablonu wiele bloków builders, które będą zawierać właściwości dostawcy. Oto przykładowy kod w formacie JSON:

```
{
  "builders": [
    {
      "type": "azure-rm",
      "location": "westus",
      ....
    },
    {
      "type": "docker",
      "image": "alpine:latest",
      ...
    }
  ]
}
```

Aby przetłumaczyć ten sam kod bloku na format HCL, tworzymy dwa bloki sources, które definiują różne właściwości dostawcy, i wywołujemy je w sekcji build w następujący sposób:

```
build {  
    sources = ["sources.azure-arm.azurevm", "sources.docker.docker-img"]  
    ...  
}
```

W tym przykładzie kodu definiujemy w szablonie Packera informacje dotyczące obrazu maszyny wirtualnej platformy Azure oraz informacje dotyczące obrazu Dockera bazującego na Alpine. Zaletą tego jest ujednolicenie skryptów, które zostaną szczegółowo opisane w sekcji udostępniania tych dwóch obrazów.

Zaraz po szczegółach każdej sekcji zobaczymy konkretny przykład z `builders` do tworzenia obrazu na platformie Azure.

Przejdźmy do omówienia sekcji `provisioners`.

Sekcja `provisioners`

Sekcja `provisioners`, która jest opcjonalna, zawiera listę skryptów, które zostaną wykonane przez Packera na tymczasowym obrazie bazowym maszyny wirtualnej w celu zbudowania naszego niestandardowego obrazu maszyny wirtualnej zgodnie z naszymi potrzebami.

Jeśli szablon programu pakującego nie zawiera sekcji `provisioners`, na obrazach podstawowych nie zostanie wykonana żadna konfiguracja.

Akcje zdefiniowane w tej sekcji są dostępne zarówno dla obrazów systemu Windows, jak i Linux i mogą być różne, np.: wykonywanie lokalnego lub zdalnego skryptu, wykonywanie polecenia lub kopiowanie pliku.

Uwaga

Typ `provisioners` proponowany natywnie przez Packera jest szczegółowo opisany w dokumentacji: <https://www.packer.io/docs/provisioners/index.html>.

Możliwe jest również rozszerzenie Packera poprzez tworzenie niestandardowych typów `provisioners`. Aby dowiedzieć się więcej na ten temat, zapoznaj się z dokumentacją tutaj: <https://www.packer.io/docs/extending/custom-provisioners.html>.

Poniżej znajduje się przykładowa sekcja `provisioners` dla formatu JSON:

```
{  
    ...  
    "provisioners": [  
        {  
            "type": "shell",  
            "script": "hardening-config.sh"  
        },  
    ],  
}
```

```

        {
            "type": "file",
            "source": "scripts/installers",
            "destination": "/tmp/scripts"
        }
    ]
    ...
}

```

A poniżej znajduje się ten sam kod blokowy w formacie HCL:

```

provisioner "shell" {
    scripts = ["hardening-config.sh"]
}
provisioner "file" {
    source = "scripts/installers"
    destination = "/tmp/scripts"
}

```

W tej sekcji provisioners Packer prześle i uruchomi lokalny skrypt *hardening-config.sh*, aby zastosować konfigurację utwardzania na zdalnym tymczasowym bazowym obrazie maszyny wirtualnej, a następnie skopiuje zawartość lokalnego folderu *scripts/installers* do folderu zdalnego, */tmp/scripts*, aby skonfigurować obraz.

Tak więc w tej sekcji wymieniamy wszystkie działania konfiguracyjne dla tworzonego obrazu.

Jednak podczas tworzenia obrazu maszyny wirtualnej należy go uogólnić — innymi słowy: usunąć wszystkie dane osobowe użytkownika, które zostały użyte do utworzenia tego obrazu.

W przypadku obrazu maszyny wirtualnej z systemem Windows użyjemy narzędzia Sysprep jako ostatniego kroku bloku provisioners:

```

"provisioners": [
    ...
    {
        "type": "powershell",
        "inline": ["& C:\windows\System32\Sysprep\Sysprep.exe /oobe
        ↪/generalize /shutdown /quiet"]
    }
]

```

Inny przykład użycia Sysprep w szablonach Packera jest dostępny tutaj: <https://www.packer.io/docs/builders/azure.html>.

Aby usunąć osobiste informacje o użytkowniku z obrazu systemu Linux, użyjemy następującego kodu:

```

"provisioners": [
    ....
    {

```

```

    "type": "shell",
    "execute_command": "sudo sh -c '{{ .Vars }} {{ .Path }}'",
    "inline": [
        "/usr/sbin/waagent -force -deprovision+user && export HISTSIZE=0 && sync"
    ]
}
]

```

Uwaga

Więcej informacji na temat sekcji provisioners znajdziesz w dokumentacji tutaj: https://developer.hashicorp.com/packer/docs/templates/legacy_json_templates/provisioners, a listę akcji znajdziesz tutaj: <https://www.packer.io/docs/provisioners/index.html>.

Po omówieniu sekcji provisioners porozmawiamy o zmiennych.

Sekcja variables

W szablonie Packera możemy często używać wartości, które nie są statyczne.

Opcjonalna sekcja variables służy do definiowania zmiennych, które będą przyjmowały wartości argumentów wiersza poleceń lub zmiennych środowiskowych. Te zmienne będą następnie używane w sekcjach builders lub provisioners.

Oto przykład sekcji variables:

```

{
  "variables": {
    "access_key": "{{env 'ACCESS_KEY'}}",
    "image_folder": "/image",
    "vm_size": "Standard_DS2_v2"
  },
  ....
}

```

W tym przykładzie inicjujemy:

- Zmienną access_key wartością zmiennej środowiskowej ACCESS_KEY.
- Zmienną image_folder wartością /image.
- Rozmiar obrazu maszyny wirtualnej, która jest zmienną vm_size.

Aby użyć tych tzw. zmiennych użytkownika, używamy notacji {{user 'variablename' }}.

Oto przykład zastosowania tych zmiennych w sekcji builders:

```

"builders": [
  {
    "type": "azure-arm",

```

```

        "access_key": "{{user 'access_key'}}",
        "vm_size": "{{user 'vm_size'}}",
        ...
    }
],

```

W sekcji `provisioners` używamy zmiennych zdefiniowanych w sekcji `variables`, jak następuje:

```

"provisioners": [
  {
    "type": "shell",
    "inline": [
      "mkdir {{user 'image_folder'}}",
      "chmod 777 {{user 'image_folder'}}",
      ...
    ],
    "execute_command": "sudo sh -c '{{ .Vars }} {{ .Path }}'"
  },
  ...
]

```

Dlatego definiujemy właściwości obrazu za pomocą zmiennych, które zostaną dostarczone podczas wykonywania szablonu Packera. Możemy również użyć tych zmiennych w sekcji `provisioners` do scentralizowania tych właściwości i nie trzeba ich ponownie definiować, np. w przypadku ścieżki obrazów (`/image`), która będzie powtarzana kilka razy w szablonach.

Uwaga

Oprócz zmiennych dostarczonych przez użytkownika możliwe jest ich pobranie z innych źródeł, np. z kluczy przechowywanych w HashiCorp Vault lub Consul. Więcej informacji na temat zmiennych można znaleźć w dokumentacji: https://www.packer.io/docs/templates/legacy_json_templates/user-variables.

Właśnie widzieliśmy strukturę szablonu Packera z głównymi sekcjami, które go tworzą, czyli `builders`, `provisioners` i `variables`. Teraz spójrzmy na konkretny przykład tworzenia szablonu Packera, aby utworzyć obraz na platformie Azure.

Tworzenie obrazu platformy Azure za pomocą szablonu Packera

Dzięki wszystkim elementom, które widzieliśmy wcześniej, będziemy teraz mogli utworzyć szablon programu Packer, który utworzy obraz maszyny wirtualnej na platformie Azure.

W tym celu musimy najpierw utworzyć jednostkę usługi Azure AD (SP), która będzie miała uprawnienia do tworzenia zasobów w naszej subskrypcji. Tworzenie jest dokładnie takie samo jak w przypadku Terraform. Aby uzyskać więcej informacji, zobacz sekcję „Konfigurowanie Terraform dla platformy Azure” w rozdziale 2, „Udostępnianie infrastruktury chmury za pomocą Terraform”. Następnie na dysku lokalnym utworzymy plik szablonu Packera.

Jeśli chcesz użyć formatu szablonu JSON, utwórz plik *azure_linux.json*, który będzie naszym szablonem Packera. Tworzenie sekcji builders rozpoczniemy w następujący sposób:

```
... "builders": [{
    "type": "azure-arm",
    "client_id": "{{user 'clientid'}}",
    "client_secret": "{{user 'clientsecret'}}",
    "subscription_id": "{{user 'subscriptionid'}}",
    "tenant_id": "{{user 'tenantid'}}",

    "os_type": "Linux",
    "image_publisher": "Canonical",
    "image_offer": "UbuntuServer",
    "image_sku": "18.04-LTS",
    "location": "West Europe",
    "vm_size": "Standard_DS2_v3",

    "managed_image_resource_group_name": "{{user 'resource_group'}}",
    "managed_image_name": "{{user 'image_name'}}-{{user
↳ 'image_version'}}",
    "azure_tags": {
        "version": "{{user 'image_version'}}",
        "role": "WebServer"
    }
}], ...
```

W tej sekcji opisano:

- Typ *azure_rm*, który wskazuje dostawcę.
- Właściwości *client_id*, *secret_client*, *subscription_id* i *tenant_id*, które zawierają informacje z poprzednio utworzonej SP. Ze względów bezpieczeństwa wartości te nie są zapisywane w postaci zwykłego tekstu w szablonie JSON; zostaną one umieszczone w zmiennych (które zobaczymy zaraz po szczegółach sekcji builders).
- Właściwości *managed_image_resource_group_name* i *managed_image_name* wskazują grupę zasobów i nazwę tworzonego obrazu. Nazwa obrazu jest również umieszczana w zmiennej z nazwą i numerem wersji.

- Pozostałe właściwości odpowiadają informacjom o typie systemu operacyjnego (Ubuntu 18), rozmiarze (Standard_DS2_v3), regionie i tagu.

Teraz utworzymy sekcję `variables`, która definiuje elementy, które nie są stałe:

```
... "variables": {
  "subscriptionid": "{{env 'AZURE_SUBSCRIPTION_ID'}}",
  "clientid": "{{env 'AZURE_CLIENT_ID'}}",
  "clientsecret": "{{env 'AZURE_CLIENT_SECRET'}}",
  "tenantid": "{{env 'AZURE_TENANT_ID'}}",

  "resource_group": "rg_images",
  "image_name": "linuxWeb",
  "image_version": "0.0.1"
},...
```

Zdefiniowaliśmy zmienne i ich wartości domyślne w następujący sposób:

- Cztery informacje uwierzytelniające z SP zostaną przekazane albo w wierszu poleceń Packera, albo jako zmienne środowiskowe.
- Grupa zasobów, nazwa, rozmiar i region generowanego obrazu również znajdują się w `variables`.
- Zdefiniowana jest zmienna `image_version` zawierająca wersję obrazu (używaną w nazwie obrazu).

Dzięki tym zmiennym będziemy mogli użyć tego samego pliku szablonu JSON do wygenerowania kilku obrazów o różnych nazwach i rozmiarach (zobaczmy to po uruchomieniu Packera).

Ostatnią czynnością jest utworzenie sekcji `provisioners` obrazu:

```
"provisioners": [
{
  "type": "shell",
  "execute_command": "sudo sh -c '{{ .Vars }} {{ .Path }}'",
  "inline": [
    "apt-get update",
    "apt-get -y install nginx"
  ]
},
{
  "type": "shell",
  "execute_command": "sudo sh -c '{{ .Vars }} {{ .Path }}'",
  "inline": [
    "/usr/sbin/waagent -force -deprovision+user && export HISTSIZE=0 &&
    ↪sync"
  ]
}
]
```

Oto, co robi poprzedni blok kodu:

- Aktualizuje pakiety za pomocą `apt-get update` i `upgrade`.
- Instaluje NGINX.

W ostatnim kroku przed utworzeniem obrazu maszyna wirtualna jest wyrejestrowywana za pomocą następującego polecenia:

```
/usr/sbin/waagent -force -deprovision+user && export HITSIZE=0 && sync
```

Dzięki temu usunięte zostaną informacje o użytkowniku, który zainstalował całe oprogramowanie na tymczasowej maszynie wirtualnej.

Uwaga

Pełny kod źródłowy szablonu Packera jest dostępny tutaj: https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP04/templates/azure_linux.json.

Właśnie widzieliśmy strukturę szablonu Packera, który składa się głównie z trzech sekcji: `variables`, `builders` i `provisioners`. Zobaczyliśmy konkretny przykład tworzenia szablonu Packera w celu wygenerowania niestandardowego obrazu maszyny wirtualnej w Azure. Skorzystaliśmy ze skryptów lub z poleceń udostępniania.

Mamy szablon Packera gotowy do uruchomienia. Jednak najpierw zobaczymy inny typ `provisioners` — korzystający z Ansible.

Tworzenie szablonów Packera przy użyciu Ansible

Właśnie dowiedzieliśmy się, jak utworzyć szablon Packera, który używa skryptów (np. `apt-get`). Możliwe jest również użycie `playbooków` Ansible do tworzenia obrazu. Rzeczywiście, kiedy używamy IaC do konfigurowania maszyn wirtualnych, często jesteśmy przyzwyczajeni do konfigurowania maszyn wirtualnych bezpośrednio za pomocą Ansible, zanim pomyślimy o przekształceniu ich w obrazy maszyn wirtualnych.

Interesujące w Packerze jest to, że możemy ponownie wykorzystać te same skrypty z `playbooka`, których używaliśmy do konfigurowania maszyn wirtualnych, do tworzenia naszych obrazów maszyn wirtualnych. To ogromna oszczędność czasu, ponieważ nie musimy przepisywać skryptów.

Aby zastosować to w praktyce, napiszemy co następuje:

- Playbook Ansible, który instaluje NGINX.
- Szablon Packera, który wykorzystuje Ansible z naszym playbookiem.

Zacznijmy od utworzenia playbooka Ansible.

Tworzenie playbooka Ansible

Playbook, który zamierzamy utworzyć, jest prawie identyczny z tym, który utworzyliśmy w rozdziale 3., „Używanie Ansible do konfigurowania infrastruktury IaaS”, ale z pewnymi zmianami.

Poniższy kod przedstawia przykładowy playbook:

```
---
- hosts: 127.0.0.1
  become: true
  connection: local
  tasks:
    - name: installing Ngnix latest version
      apt:
        name: nginx
        state: latest
    - name: starting Nginx service
      service:
        name: nginx
        state: started
```

Wprowadzone zmiany są następujące:

- Nie ma pliku inwentarza, ponieważ to Packer zarządza zdalnym hostem, będącym tymczasową maszyną wirtualną, która zostanie użyta do utworzenia obrazu.
- Wartością hosts jest zatem lokalny adres IP.
- W playbooku zachowujemy tylko instalację NGINX i usunęliśmy zadanie, które instalowało bazę danych MySQL.

Uwaga

Kod tego playbooka jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP04/templates/ansible/playbookdemo.yml>.

Teraz, gdy utworzyliśmy playbook Ansible, zobaczymy, jak zintegrować jego wykonanie z szablonem Packera.

Integracja playbooka Ansible z szablonem Packera

Jeśli chodzi o szablon Packera, sekcje `builders` i `variables` JSON są identyczne z jednym z szablonów korzystających ze skryptów opisanych wcześniej w sekcji „Tworzenie szablonów Packera przy użyciu Ansible”. Inna jest sekcja `provisioners`, którą napiszemy w następujący sposób:

```
provisioners: [
{
  "type": "shell",
  "execute_command": "sudo sh -c '{{ .Vars }} {{ .Path }}'",
  "inline": [
    "add-apt-repository ppa:ansible/ansible", "apt-get update",
    ↪ "apt-get install ansible -y"
  ]
},
{
  "type": "ansible-local",
  "playbook_file": "ansible/playbookdemo.yml"
},
{
  "type": "shell",
  "execute_command": "sudo sh -c '{{ .Vars }} {{ .Path }}'",
  "script": "clean.sh"
},
.....//Wyrejestruj VM
]
```

Akcje opisane w tej sekcji `provisioners`, które Packer wykona przy wykonaniu tego szablonu, są następujące:

1. Instalacja Ansible na tymczasowej maszynie wirtualnej.
2. Na tej tymczasowej maszynie wirtualnej dostawca `ansible-local` uruchamia plik `playbookdemo.yaml`, który instaluje i uruchamia NGINX. Dokumentacja tego dostawcy znajduje się tutaj: <https://www.packer.io/docs/provisioners/ansible/ansible-local>.
3. Skrypt `clean.sh` usuwa Ansible i zależne od niego pakiety, które nie są już używane.
4. Packer wyrejestrowuje maszynę wirtualną, aby usunąć informacje o użytkownikach lokalnych.

Jak widzimy, Packer wykona Ansible na tymczasowej maszynie wirtualnej, która zostanie użyta do utworzenia obrazu. Możliwe jest również zdalne użycie Ansible za pomocą dostawcy Packera, którego dokumentacja znajduje się tutaj: <https://www.packer.io/docs/provisioners/ansible/ansible>.

Uwaga

Kompletny szablon Packera jest dostępny tutaj: https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP04/templates/azure_linux_ansible.json, a kod źródłowy skryptu `clean.sh` jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP04/templates/clean.sh>.

Do tej pory dowiedzieliśmy się, że szablon Packera składa się z sekcji `builders`, `variables` i `provisioners`. Widzieliśmy, że możliwe jest użycie Ansible w szablonie Packera.

Teraz uruchomimy Packera z tymi szablonami JSON, aby utworzyć obraz maszyny wirtualnej na platformie Azure.

Uruchamianie Packera

Po utworzeniu szablonów Packera następnym krokiem jest jego uruchomienie w celu wygenerowania niestandardowego obrazu maszyny wirtualnej. Będzie on używany do szybkiego udostępniania maszyn wirtualnych, które są już skonfigurowane i gotowe do użycia dla Twoich aplikacji.

Przypominamy, że aby wygenerować ten obraz, Packer z naszego szablonu JSON *utworzy tymczasową maszynę wirtualną*, na której *wykona wszystkie czynności konfiguracyjne* opisane w tym szablonie, a następnie wygeneruje obraz. Na koniec *usunie tymczasową maszynę wirtualną* i wszystkie jej zależności.

Aby wygenerować nasz obraz maszyny wirtualnej na platformie Azure, wykonaj następujące kroki:

1. Skonfiguruj Packera do uwierzytelniania na platformie Azure.
2. Sprawdź nasz szablon Packera.
3. Uruchom Packera, aby wygenerować nasz obraz.

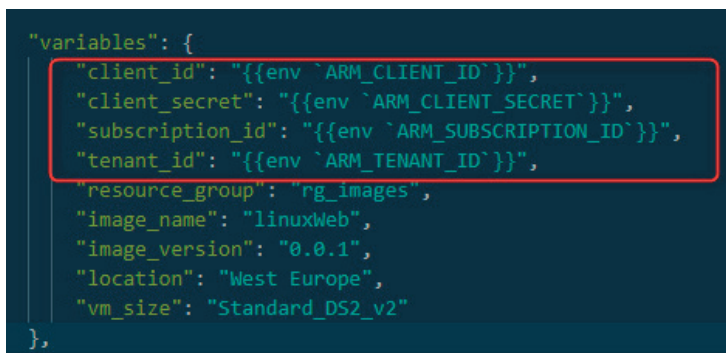
Przyjrzyjmy się szczegółowo wykonaniu każdego z tych kroków.

Konfigurowanie Packera do uwierzytelniania na platformie Azure

Aby umożliwić Packerowi tworzenie zasobów na platformie Azure, użyjemy usługi Azure AD SP utworzonej wcześniej w tym rozdziale w sekcji „Tworzenie obrazu platformy Azure za pomocą szablonu Packera”. Aby uruchomić Packera na platformie Azure, użyjemy czterech informacji uwierzytelniających (`subscription_id`, `client_id`,

client_secret i tenant_id) tej SP w zmiennych środowiskowych dostarczonych w naszym szablonie Packera w sekcji variables.

W poniższym szablonie mamy cztery zmienne (client_id, client_secret, subscription_id i tenant_id), które przyjmują jako wartości zawartość czterech zmiennych środowiskowych (ARM_CLIENT_ID, ARM_CLIENT_SECRET, ARM_SUBSCRIPTION_ID i ARM_TENANT_ID):

A screenshot of a JSON configuration file for a Packer template. The file is displayed on a dark blue background with light green and yellow text. A red rectangular box highlights the 'variables' section, which contains four entries: 'client_id', 'client_secret', 'subscription_id', and 'tenant_id'. Each entry is assigned a value using the '{{env `ARM_...`}}' syntax. Below the highlighted section, other variables like 'resource_group', 'image_name', 'image_version', 'location', and 'vm_size' are listed without being highlighted.

```
"variables": {  
  "client_id": "{{env `ARM_CLIENT_ID`}}",  
  "client_secret": "{{env `ARM_CLIENT_SECRET`}}",  
  "subscription_id": "{{env `ARM_SUBSCRIPTION_ID`}}",  
  "tenant_id": "{{env `ARM_TENANT_ID`}}",  
  "resource_group": "rg_images",  
  "image_name": "linuxWeb",  
  "image_version": "0.0.1",  
  "location": "West Europe",  
  "vm_size": "Standard_DS2_v2"  
},
```

Rysunek 4.6. Zmienne szablonu Packera

Możemy więc ustawić te zmienne środowiskowe w następujący sposób (to jest przykład Linuksa):

```
export ARM_SUBSCRIPTION_ID=<subscription_id>export ARM_CLIENT_ID=<client  
ID>export ARM_SECRET_SECRET=<client Secret>export ARM_TENANT_ID=<tenant ID>
```

Uwaga

Aby ustawić zmienną środowiskową w systemie Windows, możemy użyć polecenia PowerShell \$env.

Pierwszy krok uwierzytelniania jest zakończony i teraz sprawdzimy napisany przez nas szablon Packera.

Sprawdzanie poprawności szablonu Packera

Przed wykonaniem Packera w celu wygenerowania obrazu wykonamy polecenie sprawdzania poprawności, aby sprawdzić, czy nasz szablon jest poprawny.

Tak więc w folderze zawierającym szablon Packera wykonujemy następujące polecenie na szablonie:

```
packer validate azure_linux.json
```

Dane wyjściowe z wykonania tego polecenia zwracają stan sprawdzenia, czy szablon jest prawidłowy — jak pokazano na poniższym zrzucie ekranu:

```
/templates# packer validate azure_linux.json  
Template validated successfully.
```

Rysunek 4.7. Sprawdzanie poprawności przez Packera

Nasz szablon Packera ma poprawną składnię, więc możemy go uruchomić, aby wygenerować nasz obraz.

Uruchamianie Packera w celu wygenerowania naszego obrazu maszyny wirtualnej

Aby wygenerować nasz obraz za pomocą Packera, uruchomimy go za pomocą polecenia `build` na pliku szablonu:

```
packer build azure_linux.json
```

W danych wyjściowych po wykonaniu Packera możemy zobaczyć różne wykonywane przez niego akcje. Pierwszą z nich jest utworzenie tymczasowej maszyny wirtualnej, jak pokazano na poniższym zrzucie ekranu:

```
/CHAP04/templates# packer build azure_linux.json  
azure-arm output will be in this color.  
==> azure-arm: Running builder ...  
==> azure-arm: Getting tokens using client secret  
==> azure-arm: Creating Azure Resource Manager (ARM) client ...  
==> azure-arm: WARNING: Zone resiliency may not be supported in West Europe, checkout the docs at https://  
==> azure-arm: Creating resource group ...  
==> azure-arm:   -> ResourceGroupName : 'packer-Resource-Group-xwzj9da4y4'  
==> azure-arm:   -> Location           : 'West Europe'  
==> azure-arm:   -> Tags                :  
==> azure-arm:   -> version             : 0.0.1  
==> azure-arm:   -> role                : WebServer  
==> azure-arm: Validating deployment template ...  
==> azure-arm:   -> ResourceGroupName : 'packer-Resource-Group-xwzj9da4y4'  
==> azure-arm:   -> DeploymentName    : 'pkrdpwxzj9da4y4'  
==> azure-arm: Deploying deployment template ...  
==> azure-arm:   -> ResourceGroupName : 'packer-Resource-Group-xwzj9da4y4'  
==> azure-arm:   -> DeploymentName    : 'pkrdpwxzj9da4y4'
```

Rysunek 4.8. Utworzenie tymczasowej maszyny wirtualnej przez Packera

W portalu Azure widzimy tymczasową grupę zasobów i jej zasoby utworzone przez Packera, jak pokazano na poniższym zrzucie ekranu.

Czas wykonania Packera zależy od działań, które mają zostać wykonane na tymczasowej maszynie wirtualnej. Pod koniec wykonywania Packer wskazuje, że wygenerował obraz, i usuwa zasoby tymczasowe.





Rysunek 4.9. Tymczasowa grupa zasobów Packera w Azure Portal

Poniższy zrzut ekranu zawiera ostatnie dane wyjściowe dla wykonania Packera, czyli usunięcie tymczasowej grupy zasobów i wygenerowanie obrazu:

```
==> azure-arm: Querying the machine's properties ...
==> azure-arm: -> ResourceGroupName : 'packer-Resource-Group-uh810tdrw'
==> azure-arm: -> ComputeName       : 'pkrmuht810tdrw'
==> azure-arm: -> Managed OS Disk   : '/subscriptions/8a7aace5-.../resourceGroups/packer-Resource-Group-uh810tdrw/providers/Microsoft
,Compute/disk/pkrosh810tdrw'
==> azure-arm: Querying the machine's additional disks properties ...
==> azure-arm: -> ResourceGroupName : 'packer-Resource-Group-uh810tdrw'
==> azure-arm: -> ComputeName       : 'pkrmuht810tdrw'
==> azure-arm: Powering off machine ...
==> azure-arm: -> ResourceGroupName : 'packer-Resource-Group-uh810tdrw'
==> azure-arm: -> ComputeName       : 'pkrmuht810tdrw'
==> azure-arm: Capturing image ...
==> azure-arm: -> Compute ResourceGroupName : 'packer-Resource-Group-uh810tdrw'
==> azure-arm: -> Compute Name       : 'pkrmuht810tdrw'
==> azure-arm: -> Compute Location    : 'West Europe'
==> azure-arm: -> Image ResourceGroupName : 'rg_images'
==> azure-arm: -> Image Name          : 'linuxWeb-0.0.2'
==> azure-arm: -> Image Location     : 'westeurope'
==> azure-arm: Deleting resource group ...
==> azure-arm: -> ResourceGroupName : 'packer-Resource-Group-uh810tdrw'
==> azure-arm: The resource group was created by Packer, deleting ...
==> azure-arm: -> OS Disk : skipping, managed disk was used...
==> azure-arm: Deleting the temporary OS disk ...
==> azure-arm: Deleting the temporary Additional disk ...
==> azure-arm: -> Additional Disk : skipping, managed disk was used...
Build 'azure-arm' finished.
==> Builds finished. The artifacts of successful builds are:
--> azure-arm: Azure.ResourceManagement.VMImage:
OSType: Linux
ManagedImageResourceGroupName: rg_images
ManagedImageName: linuxWeb-0.0.2
ManagedImageId: /subscriptions/8a7aace5-.../resourceGroups/rg_images/providers/Microsoft.Compute/images/linuxWeb-0.0.2
ManagedImageLocation: westeurope
```

Rysunek 4.10. Dane wyjściowe Packera

Po wykonaniu Packera w portalu Azure sprawdzamy, czy obraz jest obecny. Poniższy zrzut ekranu pokazuje nasz wygenerowany obraz:

| <input type="checkbox"/> | NAME ↑↓ | TYPE ↑↓ | LOCATION ↑↓ |
|--------------------------|--|---------|-------------|
| <input type="checkbox"/> |  linuxWeb-0.0.1 | Image | West Europe |
| <input type="checkbox"/> |  linuxWebAnsible-0.0.1 | Image | West Europe |

Rysunek 4.11. Obraz maszyny wirtualnej Azure utworzony przez Packera

Na tym ekranie możemy zobaczyć nasz obraz i obrazy, które wygenerowaliśmy za pomocą szablonu Packera, który wykorzystuje Ansible.

Interesujące jest również to, że możemy nadpisać zmienne naszego szablonu podczas wykonywania polecenia `packer build`, jak w poniższym przykładzie:

```
packer build -var 'image_version=0.0.2' azure_linux.json
```

Do polecenia `build` możemy przekazać wszystkie zmienne za pomocą opcji `-var`.

Dzięki tej opcji możemy zmienić nazwę obrazu bez zmiany zawartości szablonu. Możemy to zrobić dla wszystkich zmiennych, które są zdefiniowane w naszym szablonie.

Uwaga

Pełna dokumentacja polecenia `build` jest dostępna tutaj: <https://www.packer.io/docs/commands/build.html>.

Poznaliśmy polecenia Packera służące do sprawdzania składni szablonu Packera w formacie JSON. Następnie uruchomiliśmy Packera na szablonie, który generuje obraz maszyny wirtualnej na platformie Azure.

Teraz poznamy podstawowe etapy tworzenia szablonów Packera w formacie HCL.

Tworzenie szablonów Packera w formacie HCL

Od wydania wersji Packera 1.5.0 możliwe jest tworzenie szablonów przy użyciu formatu HCL, o czym dowiedzieliśmy się szczegółowo w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”.

Uwaga

Integracja HCL w Packerze jest obecnie w fazie beta i będzie zalecana przez HashiCorp od wersji 1.7.0.

Format szablonu HCL jest bardzo podobny do formatu JSON i składa się z bloków `variable`, `source`, `build` i `provisioner`. Poniższy kod przedstawia strukturę szablonu HCL Packera.

Aby utworzyć szablon HCL, utwórz plik *.pkr.hcl*, który zawiera następujący kod:

```
packer {
  required_plugins
  {
    azure =
    {
      version = ">= 1.0.0"
      source = "github.com/hashicorp/azure"
    }
  }
}

Variable "var name" {
  ...
}

Source "name" {
  ...
}

Build {
  Source = []
  Provisioner "" {}
}
```

Zaczynamy od konfiguracji wtyczki Packera, która została wprowadzona od wersji 1.7.0. W tym bloku wymieniamy wszystkie wtyczki, które będą używane w kodzie.

Uwaga

Aby uzyskać więcej informacji na temat wtyczek Packera, przeczytaj dokumentację pod adresem <https://www.packer.io/docs/plugins>.

Następnie w bloku *variable* deklarujemy zmienne użytkownika, np.: poświadczenia Azure, nazwę maszyny wirtualnej, rozmiar maszyny wirtualnej lub inne zmienne. Poniższy kod przedstawia przykładowy blok *variable*:

```
variable "image_folder" {
  default = "/image"
}

variable "vm_size" {
  default = "Standard_DS2_v2"
}
```

Uwaga

Aby uzyskać więcej informacji i szczegółów na temat zmiennych HCL, przeczytaj dokumentację pod adresem <https://www.packer.io/guides/hcl/variables>.

Blok `source` zawiera właściwości obrazu docelowego, aby skompilować obraz platformy Azure lub obraz platformy Docker. Poniższy kod pokazuje przykłady dwóch bloków `source`:

- Pierwsza deklaracja `source` dotyczy maszyny wirtualnej platformy Azure:

```
source "azure-arm" "azurevm" {
  os_type = "Linux"
  location = "West Europe"
  vm_size = "Standard_DS2_V2"
  ....
}
```

- Druga deklaracja `source` dotyczy obrazu Dockera:

```
source "docker" "docker-img"
{
  image = "ubuntu"
  export_path = "imagedocker.tar"
}
```

Blok `build` zawiera listę źródeł do użycia i skrypty `provisioner` służące do konfigurowania obrazów. Poniższy kod przedstawia przykładowy blok `build`:

```
build {
  sources = ["sources.azure-arm.azurevm", "sources.docker.dockerimg"]
  provisioner "shell" {
    inline = [
      "apt-get update",
      "apt-get -y install nginx"
    ]
    execute_command = "chmod +x {{ .Path }}; {{ .Vars }} sudo -E sh \
↳ '{{ .Path }}'"
    inline_shebang = "/bin/sh -x"
  }

  provisioner "shell" {
    inline = [
      "sleep 30",
      "/usr/sbin/waagent -force -deprovision+user && export \
↳ HISTSIZE=0 && sync"
    ]
    execute_command = "chmod +x {{ .Path }}; {{ .Vars }} sudo -E sh \
↳ '{{ .Ścieżka }}'"
    inline_shebang = "/bin/sh -x"
  }
}
```

W powyższym kodzie właściwość `source` zawiera listę źródeł zadeklarowanych tuż przed szczegółami bloku `source`.

Następnie używamy listy bloków `provisioner` typu `shell`, aby zainstalować pakiet NGINX i wyczyścić obraz z danych osobowych — dokładnie te same operacje, które wykonaliśmy w formacie JSON.

Uwaga

Kod źródłowy tego formatu szablonu HCL jest dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP04/templates/pkr.hcl>.

Na koniec, aby uruchomić Packera przy użyciu szablonu HCL, wykonujemy następujące polecenie, by pobrać wtyczki:

```
packer init .pkr.hcl
```

Następnie sprawdzamy składnię szablonu, uruchamiając komendę `validate`:

```
packer validate .pkr.hcl
```

Na koniec budujemy żądany obraz, uruchamiając następujące polecenie:

```
packer build .pkr.hcl
```

Po wykonaniu powyższego polecenia zostanie utworzony obraz, dokładnie tak, jak widzieliśmy już w formacie JSON.

Uwaga

Aby przenieść szablony Packera z formatu JSON do formatu HCL, przeczytaj tę dokumentację: <https://learn.hashicorp.com/tutorials/packer/hcl2-upgrade>.

Omówiliśmy, jak tworzyć i wykonywać szablony Packera przy użyciu formatu HCL, który stanie się formatem zalecanym przez HashiCorp. Dowiemy się teraz, jak udostępnić za pomocą Terraform maszynę wirtualną opartą na tym właśnie wygenerowanym obrazie.

Korzystanie z obrazów utworzonych przez Packera za pomocą Terraform

Teraz, gdy wygenerowaliśmy niestandardowy obraz maszyny wirtualnej, udostępnimy za jego pomocą nową maszynę wirtualną. Do udostępniania maszyny wirtualnej będziemy nadal korzystać z praktyk IaC przy użyciu Terraform firmy HashiCorp.

Aby to zrobić, użyjemy skryptu Terraform utworzonego w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”, i zmodyfikujemy go tak, aby używał nie-standardowego obrazu.

W skrypcie *compute.tf* dodaj następujący blok danych, który będzie wskazywał na obraz maszyny wirtualnej wygenerowany za pomocą Packera w ostatniej sekcji:

```
## UZYSKAJ NIESTANDARDOWY OBRAZ UTWORZONY PRZEZ PACKERA
data "azurerm_image" "customnignix" {
  name = "linuxWeb-0.0.1"
  resource_group_name = "rg_images"
}
```

W tym kodzie dodajemy blok danych Terraform *azurerm_image*, który pozwala nam pobrać właściwości obrazu maszyny wirtualnej na platformie Azure. Określamy w nim właściwość *name*, dla której podajemy nazwę niestandardowego obrazu, i właściwość *resource_group_name*, gdzie określamy grupę zasobów obrazu.

Aby uzyskać więcej informacji na temat tego bloku danych *azurerm_image* i jego właściwości, zapoznaj się z dokumentacją: <https://www.terraform.io/docs/providers/azurerm/d/image.html>.

Następnie w kodzie VM Terraform i kodzie zasobu *azurerm_virtual_machine* (nadal w pliku *compute.tf*) sekcja *storage_image_reference* jest modyfikowana następującym kodem:

```
resource "azurerm_virtual_machine" "vm" {
...
  ## UŻYJ NIESTANDARDOWEGO OBRAZU
  storage_image_reference {
    id = "${data.azurerm_image.customnignix.id}"
  }
...
}
```

W tym kodzie właściwość *ID* wykorzystuje *id* obrazu z danych bloku, który dodaliśmy wcześniej.

Uwaga

Cały kod skryptu *compute.tf* jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP04/terraform/compute.tf>, a pełny kod Terraform jest tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP04/terraform>.

Podczas wykonywania tego kodu, który jest identyczny z klasycznym wykonaniem Terraform, jak pokazano w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą

Terraform”, udostępniana maszyna wirtualna będzie oparta na niestandardowym obrazie wygenerowanym przez Packera.

Widzieliśmy, że zmieniając trochę nasz poprzedni kod Terraform, poprzez dodanie bloku danych, który pobiera informacje z obrazu maszyny wirtualnej, i używając identyfikatora tego obrazu, możemy w Terraform używać niestandardowych obrazów maszyn wirtualnych generowanych przez Packera.

Podsumowanie

W tym rozdziale zobaczyliśmy, jak zainstalować Packera i używać go do tworzenia niestandardowych obrazów maszyn wirtualnych. Obraz maszyny wirtualnej został utworzony z dwóch szablonów Packera: pierwszy za pomocą skryptów i drugi za pomocą Ansible.

Następnie zobaczyliśmy, jak tworzyć szablony Packera w formacie HCL. Na koniec zmodyfikowaliśmy nasz kod Terraform, aby używał naszego obrazu maszyny wirtualnej.

Ten rozdział kończy implementację praktyk IaC. Zaczynaliśmy od **Terraform** w celu udostępnienia infrastruktury chmury, następnie wykorzystaliśmy **Ansible** do konfiguracji serwera, a zakończyliśmy na **Packerze** do tworzenia obrazu maszyny wirtualnej.

Dzięki tym obrazom maszyn wirtualnych utworzonych przez Packera będziemy w stanie skrócić czas udostępniania infrastruktury dzięki szybszemu wdrażaniu, gotowym do użycia maszynom wirtualnym, a tym samym skróceniu przestojów.

Oczywiście nie są to jedyne narzędzia IaC; istnieje wiele innych na rynku i będziesz musiał przeprowadzić monitorowanie technologii, aby znaleźć te, które najlepiej odpowiadają Twoim potrzebom.

W następnym rozdziale rozpoczniemy nową część, jaką jest implementacja CI/CD, i dowiemy się, jak używać Gita do sourcingu kodu.

Pytania

1. Jak są dwa sposoby instalacji Packera?
2. Jakie są obowiązkowe sekcje szablonu Packera używane do tworzenia obrazu maszyny wirtualnej na platformie Azure?
3. Które polecenie służy do sprawdzania poprawności szablonu Packera?
4. Które polecenie służy do generowania obrazu Packera?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej na temat Packera, oto kilka zasobów:

- Dokumentacja Packera — <https://www.packer.io/docs/>.
- Nauka Packera — <https://learn.hashicorp.com/packer>.
- Korzystanie z Packera na platformie Azure — <https://docs.microsoft.com/en-us/azure/virtual-machines/linux/build-image-with-packer>.
- Projektowanie niezmienniej infrastruktury za pomocą Packera (film Pluralsight) — <https://www.pluralsight.com/courses/packer-designing-immutable-infrastructure>.

Tworzenie środowiska programistycznego z Vagrantem

Rozdział

5

W poprzednich rozdziałach dowiedzieliśmy się, jak udostępniać część infrastruktury za pomocą Terraform, jak zainstalować oprogramowanie za pomocą Ansible i jak tworzyć obrazy **maszyn wirtualnych (VM)** za pomocą Packera.

Jednym z problemów, na które często zwracają uwagę zespoły operacyjne, jest fakt, że muszą być w stanie przetestować wszystkie skrypty automatyzacji w środowiskach izolowanych, czyli takich, które nie znajdują się na komputerze lokalnym. Na przykład skrypty utworzone za pomocą Ansible pod Linuxem są trudne do przetestowania na lokalnym komputerze z systemem Windows.

By rozwiązać ten problem, możemy wykorzystać systemy wirtualizacji, takie jak Hyper-V lub VirtualBox, pozwalające na posiadanie maszyn wirtualnych z różnymi systemami operacyjnymi, które działają lokalnie. Aby pójść jeszcze dalej, możemy zautomatyzować tworzenie i udostępnianie tych maszyn za pomocą narzędzia do tworzenia środowisk wirtualnych firmy HashiCorp o nazwie **Vagrant**.

W tym rozdziale poznamy podstawy korzystania z Vagranta. Omówimy jego instalację, sposób pisania plików Vagranta, a na koniec zbadamy jego wykonanie, aby utworzyć i udostępnić maszynę wirtualną z systemem Linux.

W tym rozdziale omówimy następujące główne tematy:

- instalacja Vagranta,
- tworzenie pliku konfiguracyjnego Vagranta,
- udostępnianie lokalnej maszyny wirtualnej za pomocą interfejsu Vagrant CLI.

Wymagania techniczne

Vagrant może utworzyć maszynę wirtualną na dowolnym hipernadzorcy (ang. *hypervisor*). W tym rozdziale użyjemy VirtualBox jako lokalnego hiperwizora VM. Możesz go pobrać i zainstalować ze strony <https://www.virtualbox.org/wiki/Downloads>.

W systemie Windows, jeśli masz już zainstalowany Hyper-V, aby korzystać z VirtualBox, musisz go wyłączyć. Zapoznaj się z następującą dokumentacją: <https://www.vagrantup.com/docs/installation#windows-virtualbox-and-hyper-v>.

Wszystkie polecenia wykonane w tym rozdziale zostaną wykonane w konsoli terminala, takiej jak PowerShell lub Bash.

Pełny kod źródłowy tego rozdziału jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP05/>.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/3s5a049>.

Instalacja Vagranta

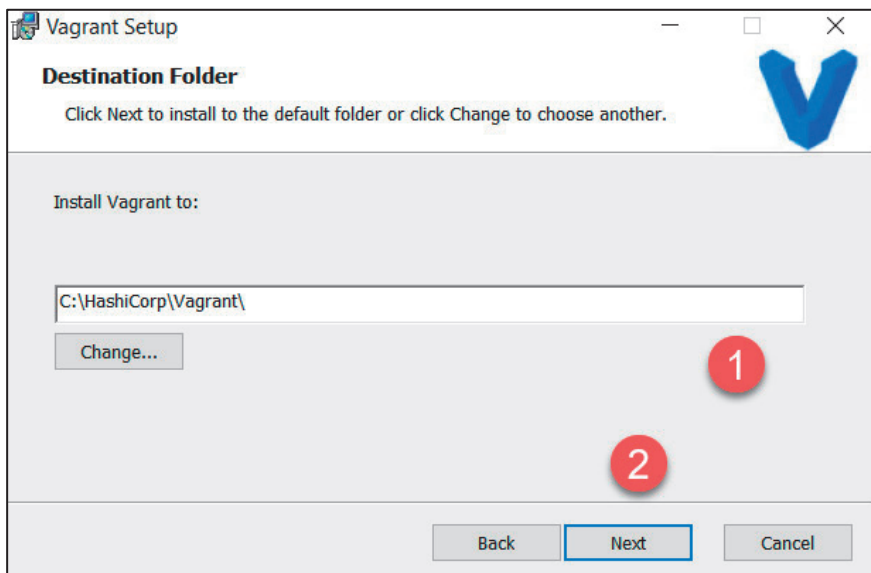
Vagrant to wieloplatformowe narzędzie, które można zainstalować w systemach Windows, Linux lub macOS. W systemie Windows instalację można przeprowadzić na dwa sposoby: ręcznie lub za pomocą skryptu.

Instalacja ręczna (w systemie Windows)

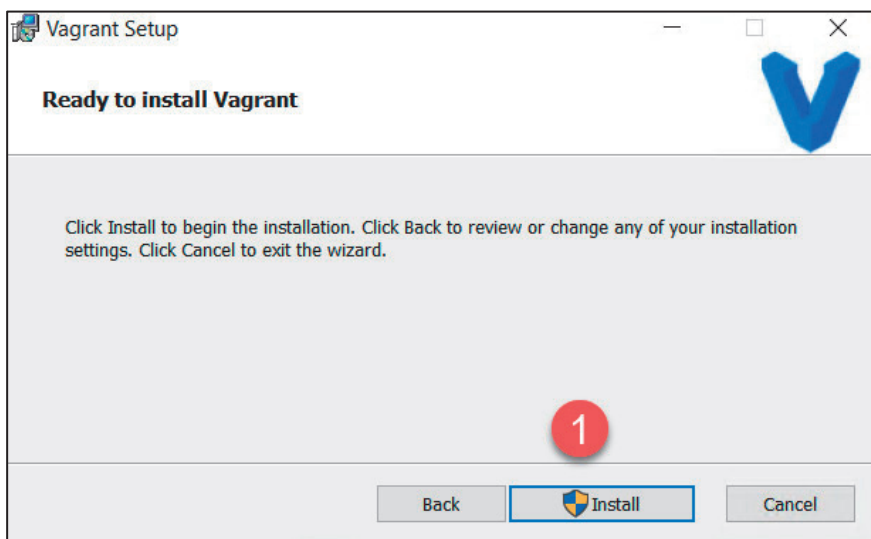
Aby ręcznie zainstalować Vagranta w systemie Windows, wykonaj następujące czynności:

1. Przejdź do oficjalnej strony pobierania (<https://www.vagrantup.com/downloads>), kliknij zakładkę *Windows* i pobierz pakiet MSI odpowiadający typowi procesora Twojego systemu operacyjnego (32- lub 64-bitowy).
2. Po pobraniu kliknij pobrany plik MSI i wybierz katalog instalacyjny (zachowaj wartość domyślną) — rysunek 5.1.
3. Następnie kliknij przycisk *Install* — rysunek 5.2.

Nauczyłeś się ręcznie instalować Vagranta w systemie Windows. W następnej sekcji omówimy instalację Vagranta przy użyciu Chocolatey lub skryptu.



Rysunek 5.1. Wybór folderu dla Vagranta



Rysunek 5.2. Przycisk instalacji Vagranta

Instalowanie Vagranta za pomocą skryptu w systemie Windows

W systemie Windows możemy użyć Chocolatey, który jest menedżerem pakietów oprogramowania.

Uwaga

W tym rozdziale nie będę ponownie opowiadać o Chocolatey, ponieważ przedstawiłmy już go w poprzednich rozdziałach. Aby uzyskać dodatkowe informacje na temat Chocolatey, zapoznaj się z dokumentacją na stronie <https://chocolatey.org/>.

Po zainstalowaniu Chocolatey wystarczy uruchomić następujące polecenie w PowerShell lub narzędziu CMD:

```
choco install vagrant -y
```

Poniżej znajduje się zrzut ekranu instalacji Vagranta dla systemu Windows przy użyciu Chocolatey:

```
PS C:\Users\mkrief\Documents> choco install vagrant -y
Chocolatey v0.10.15
Installing the following packages:
vagrant
By installing you accept licenses for the packages.
Progress: Downloading vagrant 2.2.18.20210807... 100%

vagrant v2.2.18.20210807
vagrant package files install completed. Performing other installation steps.
Downloading vagrant 64 bit
from 'https://releases.hashicorp.com/vagrant/2.2.18/vagrant_2.2.18_x86_64.msi'
Progress: 100% - Completed download of C:\Users\mkrief\AppData\Local\Temp\chocolatey\vagrant\2.2.18.20210807\vagrant_2.2.18_x86_64.msi (253.13 MB).
Download of vagrant_2.2.18_x86_64.msi (253.13 MB) completed.
Hashes match.
Installing vagrant...
vagrant has been installed.
Updating installed plugins...
All plugins are up to date.
Repairing currently installed global plugins. This may take a few minutes...
Installed plugins successfully repaired!
vagrant may be able to be automatically uninstalled.
Environment Vars (like PATH) have changed. Close/reopen your shell to
see the changes (or in powershell/cmd.exe just type 'refreshenv').
The install of vagrant was successful.
Software installed as 'msi', install location is likely default.

Chocolatey installed 1/1 packages.
See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).

Packages requiring reboot:
- vagrant (exit code 3010)

The recent package changes indicate a reboot is necessary.
Please reboot at your earliest convenience.
```

Rysunek 5.3. Instalacja Vagranta przy użyciu Chocolatey

Aby sfinalizować instalację Vagranta, musimy ponownie uruchomić komputer.

Instalowanie Vagranta za pomocą skryptu w systemie Linux

W systemie Linux możemy wykonać następujący skrypt, aby automatycznie zainstalować Vagranta:

```
sudo apt-get update && sudo apt-get install -y gnupg
↳softwareproperties-common curl \
&& curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add - \
&& sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com
↳$(lsb_release -cs) main" \
&& sudo apt-get update && sudo apt-get install vagrant
```

Ten skrypt wykonuje następujące czynności:

- Dodaje repozytorium apt HashiCorp.
- Aktualizuje lokalne repozytorium.
- Pobiera i instaluje interfejs Vagrant CLI.

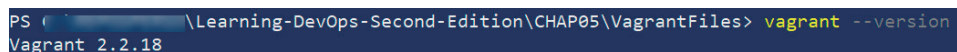
Uwaga

Aby uzyskać dodatkowe informacje na temat wszystkich instalacji Vagranta w innych systemach operacyjnych, zapoznaj się z dokumentacją pod adresem <https://www.vagrantup.com/downloads>.

Po zainstalowaniu Vagranta możemy przetestować instalację Vagrant CLI, uruchamiając następujące polecenie:

```
vagrant --version
```

Polecenie to wyświetla zainstalowaną wersję Vagranta, a poniższy zrzut ekranu pokazuje wykonanie tego polecenia:



```
PS C:\Learning-DevOps-Second-Edition\CHAP05\VagrantFiles> vagrant --version
Vagrant 2.2.18
```

Rysunek 5.4. Wyświetlanie zainstalowanej wersji Vagranta

Właśnie poznaliśmy procedurę ręcznej instalacji Vagranta i dowiedzieliśmy się, jak go zainstalować za pomocą skryptu w systemie Linux.

Teraz opiszemy kilka głównych elementów Vagranta i utworzymy konfigurację szablonu Vagranta, aby utworzyć lokalną podstawową maszynę wirtualną z systemem Linux i skonfigurować ją za pomocą Ansible do testowania skryptów.

Tworzenie pliku konfiguracyjnego Vagranta

Zanim omówimy, jak napisać plik konfiguracyjny Vagranta, warto wspomnieć o kilku artefaktach Vagranta.

Ważnymi elementami Vagranta są:

- Plik binarny Vagranta (CLI). Dowiedzieliśmy się, jak go pobrać i zainstalować, w poprzedniej sekcji, „Instalacja Vagranta”.
- W pliku konfiguracyjnym zostanie użyty podstawowy obraz maszyny wirtualnej, zwany **Vagrant Box**. Ten obraz może być publiczny, tzn. opublikowany w chmurze Vagranta, lub lokalny na komputerze.
- Plik konfiguracyjny definiuje budowę naszej maszyny wirtualnej, którą chcemy utworzyć lokalnie.

Teraz, gdy przyjrzelśmy się różnym elementom Vagranta, napiszemy plik konfiguracyjny naszej maszyny wirtualnej. Na początek sprawdzimy, z jakiej maszyny bazowej będziemy korzystać.

Używanie Vagrant Cloud dla boksów Vagranta

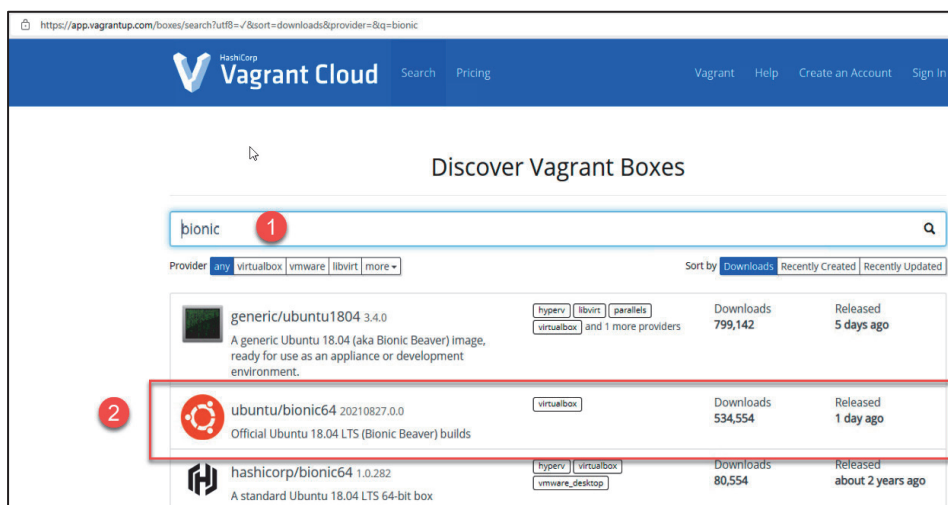
Aby korzystać z podstawowych obrazów Vagranta, HashiCorp utworzył portal, który umożliwia publikowanie i udostępnianie obrazów maszyn wirtualnych zgodnych z Vagrantem.

Aby uzyskać dostęp do tych obrazów, zwanych boksami, możesz przejść do witryny pod adresem <https://app.vagrantup.com/boxes/search>, a następnie przeprowadzić wyszukiwanie boksów, który Cię interesuje, według następujących kryteriów:

- System operacyjny (Windows lub Linux).
- Obsługiwany hiperwizor.
- Oprogramowanie, które zostało zainstalowane.
- Wydawca.

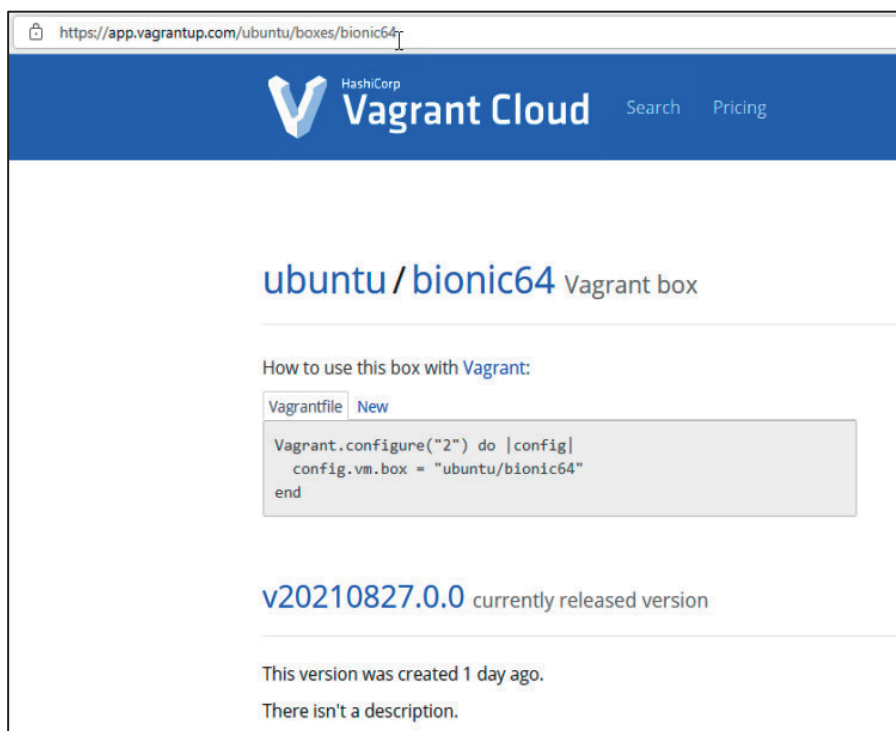
Poniższy zrzut ekranu (rysunek 5.5) pokazuje przykład wyszukiwania w Vagrant Cloud.

W tym przykładzie szukamy podstawowego boksów Bionic. Na liście wyników znajdziemy oficjalny box `ubuntu/bionic64`, który jest kompatybilny z hiperwizorem VirtualBox.



Rysunek 5.5. Boksy w Vagrant Cloud

Po kliknięciu wybranego boks portal wyświetla więcej informacji, takich jak szczegóły konfiguracji i dziennik zmian:



Rysunek 5.6. Szczegóły boks w Vagrant Cloud

Uwaga

Aby uzyskać więcej informacji o Vagrant Cloud oraz o tym, jak tworzyć i publikować niestandardowe boksy, proponuję zapoznać się z oficjalną dokumentacją pod adresem <https://www.vagrantup.com/vagrant-cloud>.

Teraz, gdy wybraliśmy nasz podstawowy box, napiszemy plik konfiguracyjny Vagranta dla naszej maszyny wirtualnej.

Tworzenie pliku konfiguracyjnego Vagranta

Aby utworzyć lokalną maszynę wirtualną za pomocą Vagranta, musimy zapisać jej konfigurację w pliku konfiguracyjnym.

Ta konfiguracja będzie zawierać informacje o maszynie wirtualnej, takie jak:

- Używany box.
- Konfiguracja sprzętowa, taka jak pamięć RAM i procesor.
- Konfiguracja sieci.
- Skrypty używane do udostępniania i konfigurowania maszyny wirtualnej.
- Lokalny folder do udostępnienia dla VM.

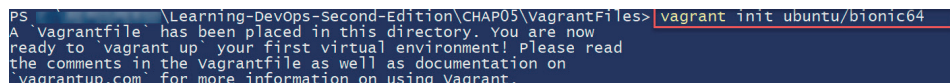
W tej sekcji dowiemy się, jak utworzyć podstawowy plik konfiguracyjny Vagranta, który tworzy lokalną maszynę wirtualną zawierającą już zainstalowany plik binarny Ansible.

Aby utworzyć konfigurację Vagranta, wykonaj następujące czynności:

1. Utwórz folder o nazwie *VagrantFiles*.
2. Wewnątrz tego folderu otwórz nową konsolę terminala i uruchom polecenie `vagrant init` z nazwą boksa:

`vagrant init ubuntu/bionic64`

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia:



```
PS > .\Learning-DevOps-Second-Edition\CHAP05\VagrantFiles> vagrant init ubuntu/bionic64
A 'Vagrantfile' has been placed in this directory. You are now
ready to 'vagrant up' your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
vagrantup.com for more information on using Vagrant.
```

Rysunek 5.7. Inicjalizacja Vagranta

Powyższe polecenie tworzy nowy plik *Vagrantfile* z podstawową konfiguracją dla maszyny wirtualnej *bionic* o zawartości:


```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/bionic64"
end
```

Następnie dodamy skrypty do zainstalowania Ansible podczas udostępniania tej maszyny wirtualnej.

3. W folderze *VagrantFiles* utwórz nowy folder o nazwie *scripts*, a następnie utwórz nowy plik skryptu o nazwie *ansible.sh* z następującą zawartością:

```
apt-get update
sudo apt-get --assume-yes install software-properties-common
sudo apt-add-repository --yes --update ppa:ansible/ansible
sudo apt-get --assume-yes install ansible
```

4. W pliku *Vagrantfile* zaktualizuj konfigurację za pomocą następujących wierszy:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/bionic64"
  config.vm.provision "shell", path: "scripts/ansible.sh"
end
```

5. Aby przetestować lokalny playbook Ansible, możemy udostępnić lokalny katalog maszynie wirtualnej, dodając następujące wiersze do pliku konfiguracyjnego:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/bionic64"
  config.vm.provision "shell", path:
    ↪ "scripts/ansible.sh"

  config.vm.synced_folder "C:\\<ścieżka>\\CHAP03\\devopsansible",
    ↪ "/learningdevops"
end
```

W poprzedniej konfiguracji udostępniliśmy lokalny folder *c:\...\Chap03\devopsansible* (jest to przykład lokalnego folderu) między maszyną lokalną a folderem */learningdevops* na maszynie wirtualnej.

Uwaga

Więcej informacji na temat tego pliku konfiguracyjnego Vagranta można znaleźć w pełnej dokumentacji na stronie <https://www.vagrantup.com/docs/vagrantfile>.

6. Ostatnim krokiem jest sprawdzenie poprawności tego pliku konfiguracyjnego, uruchamiając następujące polecenie:

```
vagrant validate
```

Dane wyjściowe tego polecenia pokazano na poniższym zrzucie ekranu:

```
PS > .\Learning-DevOps-Second-Edition\CHAP05\VagrantFiles> vagrant validate  
Vagrantfile validated successfully.
```

Rysunek 5.8. Weryfikacja konfiguracji Vagranta

Uwaga

W tej sekcji dowiedzieliśmy się, jak utworzyć maszynę wirtualną za pomocą pliku konfiguracyjnego Vagranta. Jeśli chcesz utworzyć wiele maszyn wirtualnych w tym samym pliku konfiguracyjnym, zapoznaj się z oficjalną dokumentacją pod adresem <https://www.vagrantup.com/docs/multi-machine>.

Właśnie zobaczyliśmy, jak wybrać box i utworzyć plik konfiguracyjny Vagranta. W następnej sekcji dowiemy się, jak używać Vagrant CLI i pliku konfiguracyjnego do lokalnego tworzenia maszyny wirtualnej.

Tworzenie lokalnej maszyny wirtualnej za pomocą interfejsu Vagrant CLI

Teraz, gdy napisaliśmy już plik konfiguracyjny, możemy utworzyć naszą maszynę wirtualną lokalnie.

Aby to wykonać, użyjemy kilku poleceń Vagrant CLI. Aby wyświetlić wszystkie dostępne polecenia, uruchomimy polecenie `vagrant --help` — rysunek 5.9.

Uwaga

Wszystkie szczegóły dotyczące poleceń Vagrant CLI są udokumentowane na stronie <https://www.vagrantup.com/docs/cli>.

W kolejnym rozdziale dowiemy się, jak utworzyć VM, następnie połączymy się z tą maszyną, a na koniec wykonamy kilka skryptów.

Tworzenie maszyny wirtualnej

Aby utworzyć maszynę wirtualną, przejdź do folderu, w którym utworzyliśmy plik *Vagrantfile*, i uruchom następujące polecenie:

```
vagrant up
```

```

PS > .\Learning-DevOps-Second-Edition\CHAP05\VagrantFiles> vagrant --help
Usage: vagrant [options] <command> [<args>]

    -h, --help                Print this help.

Common commands:
autocomplete    manages autocomplete installation on host
box             manages boxes: installation, removal, etc.
cloud           manages everything related to Vagrant Cloud
destroy         stops and deletes all traces of the vagrant machine
global-status   outputs status Vagrant environments for this user
halt            stops the vagrant machine
help            shows the help for a subcommand
init            initializes a new Vagrant environment by creating a Vagrantfile
login
package        packages a running vagrant environment into a box
plugin          manages plugins: install, uninstall, update, etc.
port           displays information about guest port mappings
powershell     connects to machine via powershell remoting
provision       provisions the vagrant machine
push           deploys code in this environment to a configured destination
rdp            connects to machine via RDP
reload         restarts vagrant machine, loads new Vagrantfile configuration
resume         resume a suspended vagrant machine
snapshot       manages snapshots: saving, restoring, etc.
ssh            connects to machine via SSH
ssh-config     outputs OpenSSH valid configuration to connect to the machine
status         outputs status of the vagrant machine
suspend       suspends the machine
up            starts and provisions the vagrant environment
upload         upload to machine via communicator
validate       validates the Vagrantfile
vbguest        plugin: vagrant-vbguest: install VirtualBox Guest Additions to the machine
version        prints current and latest Vagrant version
winrm          executes commands on a machine via WinRM
winrm-config   outputs WinRM configuration to connect to the machine

```

Rysunek 5.9. Wyświetlanie poleceń Vagranta

Dane wyjściowe pokazano na poniższym zrzucie ekranu:

```

Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/bionic64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/bionic64' version '20210818.0.0' is up to date...

==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
default: SSH address: 127.0.0.1:2222
default: SSH username: vagrant
default: SSH auth method: private key
default: Vagrant insecure key detected. Vagrant will automatically replace
default: Removing insecure key from the guest if it's present...
default: Key inserted! Disconnecting and reconnecting using new SSH key...
==> default: Machine booted and ready!
Got different reports about installed GuestAdditions version:
0 upgraded, 41 newly installed, 0 to remove and 0 not upgraded.

update-initramfs: Generating /boot/initrd.img-4.15.0-134-generic
Unmounting Virtualbox Guest Additions ISO from: /mnt
==> default: Checking for guest additions in VM...
==> default: Mounting shared folders...
default: /vagrant => C:/REPOSPERSO/Learning-DevOps-Second-Edition/CHAP05/VagrantFiles
default: /learningdevops => C:/REPOSPERSO/Learning-DevOps-Second-Edition/CHAP03/devopsansible
==> default: Running provisioner: shell...
default: Running: C:/Users/mkrief/AppData/Local/Temp/vagrant-shell20210829-5524-is6k7s.sh
default: Hit:1 http://archive.ubuntu.com/ubuntu bionic InRelease
default: Get:2 http://archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]

default: Setting up python-cryptography (2.1.4-1ubuntu1.4) ...
default: Setting up python-paramiko (2.0.0-1ubuntu1.2) ...
default: Setting up ansible (2.9.24-1ppa-bionic) ...
default: Processing triggers for mime-support (3.60ubuntu1) ...
default: Processing triggers for man-db (2.8.3-2ubuntu0.1) ...

```

Rysunek 5.10. Wykonanie polecenia vagrant up

Powyższe polecenie wykonuje następujące kroki:

1. Importuje box z Vagrant Cloud.
2. Tworzy nową maszynę wirtualną w hipernadzorcy (w naszym przypadku jest to VirtualBox).
3. Tworzy połączenie SSH za pomocą kluczy SSH (Vagrant tworzy prywatne/publiczne klucze SSH).
4. Montuje udostępniony folder.
5. Stosuje skrypt Ansible do udostępniania.

Po udostępnieniu maszyny wirtualnej możemy wyświetlić tę maszynę wirtualną w hipernadzorcy — tutaj w VirtualBox.

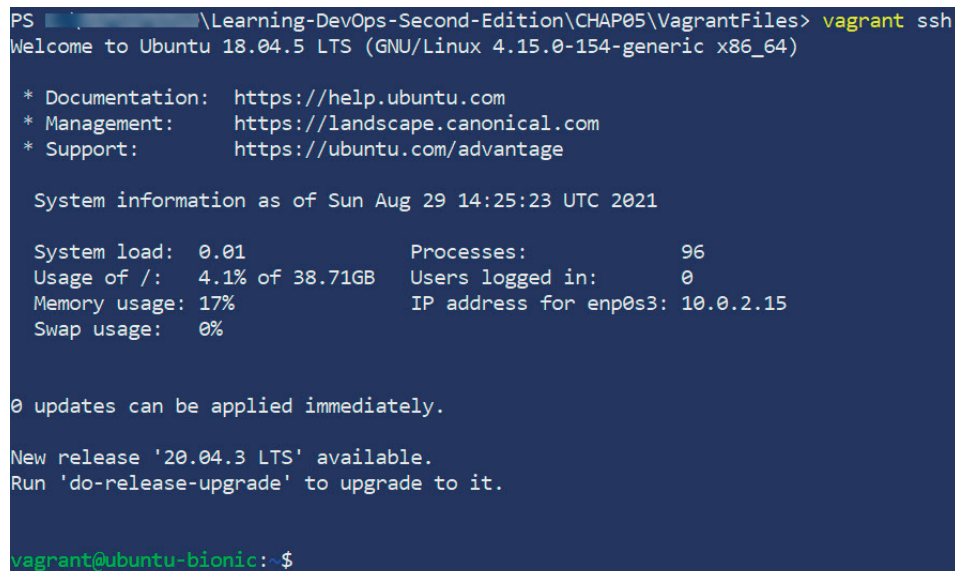
Teraz, gdy maszyna wirtualna działa i została udostępniona, możemy ją podłączyć.

Łączenie z maszyną wirtualną

Aby podłączyć VM za pomocą SSH, możemy użyć domyślnego polecenia SSH dedykowanego dla Vagranta (w trybie administratora):

```
vagrant ssh
```

Dane wyjściowe pokazano na poniższym zrzucie ekranu:



```
PS C:\Learning-DevOps-Second-Edition\CHAP05\VagrantFiles> vagrant ssh
Welcome to Ubuntu 18.04.5 LTS (GNU/Linux 4.15.0-154-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Sun Aug 29 14:25:23 UTC 2021

System load:  0.01               Processes:    96
Usage of /:   4.1% of 38.71GB    Users logged in: 0
Memory usage: 17%               IP address for enp0s3: 10.0.2.15
Swap usage:   0%

0 updates can be applied immediately.

New release '20.04.3 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

vagrant@ubuntu-bionic:~$
```

Rysunek 5.11. Podłączanie za pomocą SSH do maszyny wirtualnej Vagranta

Wykonanie tego polecenia oznacza, że Vagrant automatycznie łączy poświadczenia SSH z maszyną wirtualną i możemy uruchamiać dowolne polecenia lub skrypty wewnątrz maszyny wirtualnej.

Na przykład możemy uruchomić polecenie `ansible --version`, by wyświetlić zainstalowaną wersję Ansible:

```
vagrant@ubuntu-bionic:~$ ansible --version
ansible 2.9.24
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/home/vagrant/.ansible/plugins/modules', u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/dist-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.17 (default, Feb 27 2021, 15:10:58) [GCC 7.5.0]
vagrant@ubuntu-bionic:~$
```

Rysunek 5.12. Wykonywanie poleceń wewnątrz maszyny wirtualnej

Po uruchomieniu żądanych testów, jeśli chcemy zniszczyć tę maszynę wirtualną, aby ją zmodyfikować, możemy uruchomić polecenie `vagrant destroy`:

```
PS C:\Learning-DevOps-Second-Edition\CHAP05\VagrantFiles> vagrant destroy
default: Are you sure you want to destroy the 'default' VM? [y/N] y
==> default: Forcing shutdown of VM...
==> default: Destroying VM and associated drives...
```

Rysunek 5.13. Zniszczenie VM Vagranta

W tej sekcji dowiedzieliśmy się, jak utworzyć maszynę wirtualną i połączyć się z nią za pomocą interfejsu Vagrant CLI.

Podsumowanie

W tym rozdziale dowiedzieliśmy się, że możliwe jest tworzenie maszyn wirtualnych lokalnie przy użyciu narzędzia Vagrant firmy HashiCorp w celu uzyskania izolowanego środowiska programistycznego.

Wyjaśniliśmy, jak je pobrać i zainstalować. Następnie dowiedzieliśmy się, jak napisać plik konfiguracyjny Vagranta, korzystając z boksów Bionic, polecenia `ansible install` i folderu udostępnionego lokalnie.

Na koniec dowiedzieliśmy się, jak wykonywać polecenia Vagranta, aby utworzyć tę maszynę wirtualną i połączyć się z nią w celach testowych.

W następnym rozdziale rozpoczniemy nowy temat, jakim jest implementacja CI/CD, i dowiemy się, jak używać Gita jako źródła kodu.

Pytania

1. Jaka jest rola Vagranta?
2. Jakie jest polecenie Vagranta do tworzenia maszyny wirtualnej?
3. Jakie jest polecenie Vagranta, aby połączyć się za pomocą SSH z maszyną wirtualną?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o narzędziu Vagrant, zapoznaj się z następującymi zasobami:

- Oficjalna dokumentacja Vagranta — <https://www.vagrantup.com/docs/index>.
- Oficjalna dokumentacja Vagrant Cloud — <https://www.vagrantup.com/vagrant-cloud>.

Potok CI/CD

W tej części omówimy proces potoku DevOps, zaczynając od zasad ciągłej integracji i ciągłego wdrażania. Będziemy przy tym używać narzędzi takich jak Jenkins, Azure Pipelines i GitLab.

Sekcja ta składa się z następujących rozdziałów:

- Rozdział 6., „Zarządzanie kodem źródłowym za pomocą Gita”.
- Rozdział 7., „Ciągła integracja i ciągłe wdrażanie”.
- Rozdział 8., „Wdrażanie infrastruktury jako kodu za pomocą potoku CI/CD”.



Zarządzanie kodem źródłowym za pomocą Gita

Rozdział

6

Kilka lat temu, kiedy byliśmy programistami i pisaliśmy kod jako część zespołu, napotykaliliśmy powtarzające się problemy, które w większości były następujące:

- Jak udostępnić mój kod członkom mojego zespołu?
- Jak zaktualizować wersję mojego kodu?
- Jak śledzić zmiany w moim kodzie?
- Jak odzyskać stary stan mojego kodu lub jego części?

Z biegiem czasu problemy te zostały rozwiązane wraz z pojawieniem się menedżerów kodu źródłowego, zwanych również **systemami kontroli wersji** (ang. *version control system* — **VCS**) lub częściej określanymi jako **menedżery kontroli wersji** (ang. *version control manager* — **VCM**).

Celem systemów VCS jest przede wszystkim wykonanie następujących czynności:

- współpraca programistów przy kodzie,
- pobieranie kodu,
- wersjonowanie kodu,
- śledzenie zmian kodu.

Wraz z pojawieniem się zwinnych (ang. *agile*) metod i kultury **DevOps** stosowanie VCS w procesach stało się obowiązkowe. Jak wspomniano w rozdziale 1., „Kultura DevOps i praktyki kodowania infrastruktury”, wdrożenie procesu **ciągłej integracji/ciągłego wdrażania (CI/CD)** można wykonać tylko, jeżeli istnieje VCS.

W tym rozdziale zobaczymy, jak korzystać z jednego z najbardziej znanych systemów VCS, jakim jest Git. Zaczniemy od omówienia Gita i dowiemy się, jak go zainstalować. Następnie opiszemy jego główne polecenia, aby zapoznać każdego programistę z ich zastosowaniami. Na koniec prześledzimy aktualny proces przepływu pracy Gita

i korzystania z Gitflow. Celem tego rozdziału jest pokazanie codziennej pracy z Gitem w prostych procesach.

Ten rozdział obejmuje następujące tematy:

- przegląd Gita i jego głównych poleceń,
- zrozumienie procesu Gita i wzorca Gitflow.

Ten rozdział nie obejmuje instalacji serwera Gita, więc jeśli chcesz dowiedzieć się więcej na ten temat, możesz zapoznać się z dokumentacją pod następującym linkiem: <https://git-scm.com/book/en/v2/Git-on-the-Server-The-Protocols>.

Wymagania techniczne

Korzystanie z Gita nie wymaga żadnych rozwiązań technicznych; potrzebujemy tylko terminala wiersza poleceń.

Aby zilustrować jego użycie w tym rozdziale, użyjemy Azure Repos z platformy Azure DevOps (dawniej **Visual Studio Team System** — **VSTS**), która jest platformą chmurową z menedżerem repozytorium Gita. Możesz zarejestrować się tutaj bezpłatnie: <https://visualstudio.microsoft.com/team-services/>.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/3LTcoOi>.

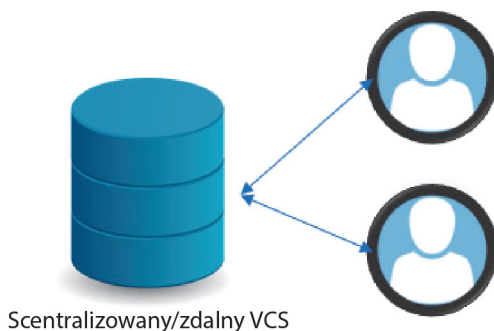
Przegląd Gita i jego głównych poleceń

Aby zrozumieć działanie Gita, należy wiedzieć, że istnieją dwa typy systemów VCS: systemy scentralizowane i rozproszone.

Jako pierwsze pojawiły się **systemy scentralizowane**, takie jak **Subversion (SVN)**, **Concurrent Version System (CVS)**, **Team Foundation Version Control (TFVC)** i **Microsoft Visual SourceSafe (VSS)**. Systemy te składają się ze zdalnego serwera, który centralizuje kod wszystkich programistów.

Możemy przedstawić scentralizowany system kontroli kodu źródłowego jak na rysunku 6.1.

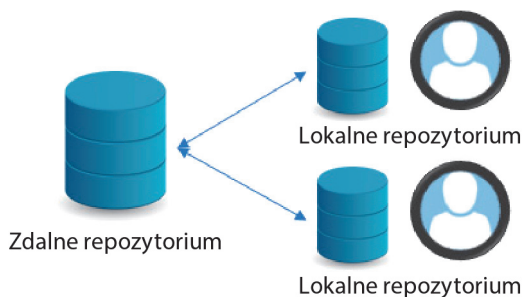
Wszyscy programiści mogą archiwizować i pobierać swój kod ze zdalnego serwera. System pozwala na lepszą współpracę między zespołami i daje gwarancję backupu kodu. Ma jednak swoje wady, takie jak:



Rysunek 6.1. Scentralizowana kontrola kodu źródłowego

- W przypadku braku połączenia (z powodu problemu z siecią lub rozłączenia z internetem) między programistami a zdalnym serwerem nie można wykonywać dalszych działań związanych z archiwizacją lub odzyskiwaniem kodu.
- Jeśli zdalny serwer przestanie działać, kod i historia zmian zostaną utracone.

Drugi rodzaj VCS, który pojawił się później, to **systemy rozproszone**, takie jak Mercurial czy Git. Te systemy składają się ze zdalnego repozytorium i z lokalnej kopii tego repozytorium na lokalnym komputerze każdego dewelopera, jak pokazano na poniższym zrzucie ekranu:



Rysunek 6.2. Rozproszona kontrola kodu źródłowego

Dzięki temu rozproszonemu systemowi nawet w przypadku odłączenia od zdalnego repozytorium programiści mogą kontynuować pracę z lokalnym repozytorium, a synchronizacja zostanie wykonana, gdy zdalne repozytorium będzie ponownie dostępne. Kopia kodu i jego historia zmian znajdują się również w lokalnym repozytorium.

Git jest zatem rozproszonym VCS-em, który został utworzony w 2005 r. przez Linusa Torvaldsa i społeczność programistów Linuksa.

Uwaga

Aby dowiedzieć się więcej o historii Gita, zajrzyj na tę stronę: <https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git>.

Od momentu powstania Git stał się bardzo potężnym i dojrzałym narzędziem, z którego może korzystać programista.

Git to bezpłatne, wieloplatformowe narzędzie, które można zainstalować na komputerze lokalnym dla osób, które tworzą kod — w trybie klienta — ale można je również zainstalować na serwerach w celu hostowania i zarządzania zdalnymi repozytoriami.

Git to narzędzie wiersza poleceń z wieloma opcjami. Obecnie istnieje wiele narzędzi graficznych — takich jak Git GUI, GitKraken, GitHub Desktop lub Sourcetree — które umożliwiają łatwiejszą i graficzną interakcję z operacjami Gita bez konieczności samodzielnego korzystania z wiersza poleceń. Jednak te narzędzia graficzne nie zawierają wszystkich operacji i opcji dostępnych w wierszu poleceń. Na szczęście wiele edytorów kodu, takich jak **Visual Studio Code (VS Code)**, Visual Studio, JetBrains i Sublime Text, umożliwia bezpośrednią integrację kodu z Gitem i ze zdalnymi repozytoriami.

W przypadku repozytoriów zdalnych dostępnych jest kilka rozwiązań chmurowych i bezpłatnych, takich jak GitHub, GitLab, Azure DevOps lub Bitbucket Cloud. Istnieją również inne rozwiązania, zwane rozwiązaniami lokalnymi, które można zainstalować w firmie, takie jak Azure DevOps Server, Bitbucket lub GitHub Enterprise.

W tym rozdziale wykorzystamy Gita — np. z usługą Azure DevOps jako zdalnym repozytorium — oraz GitLaba i GitHuba w przyszłych rozdziałach.

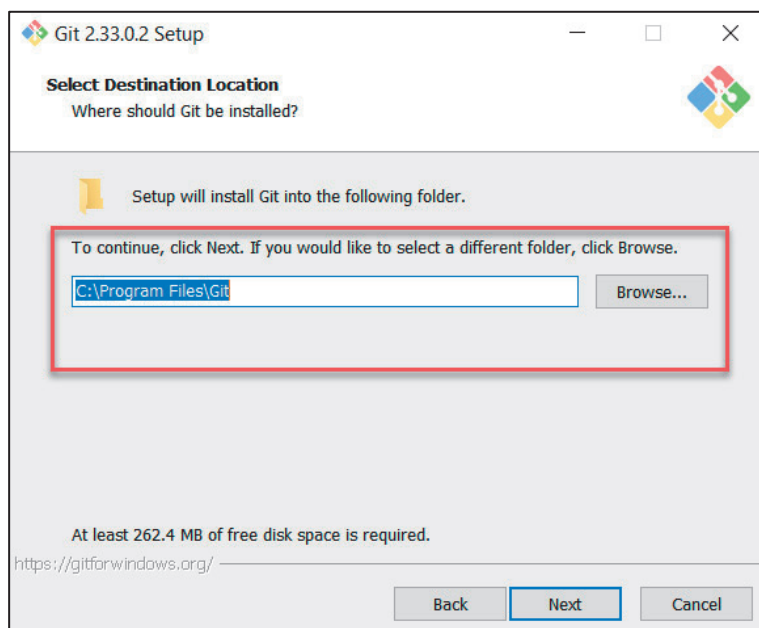
W tej sekcji omówiliśmy Gita, a teraz zobaczymy, jak zainstalować go na komputerze lokalnym, aby rozwijać i wersjonować nasz kod źródłowy.

Instalacja Gita

Omówimy teraz szczegółowo kroki instalacji i konfiguracji Gita w systemach Windows, Linux i macOS.

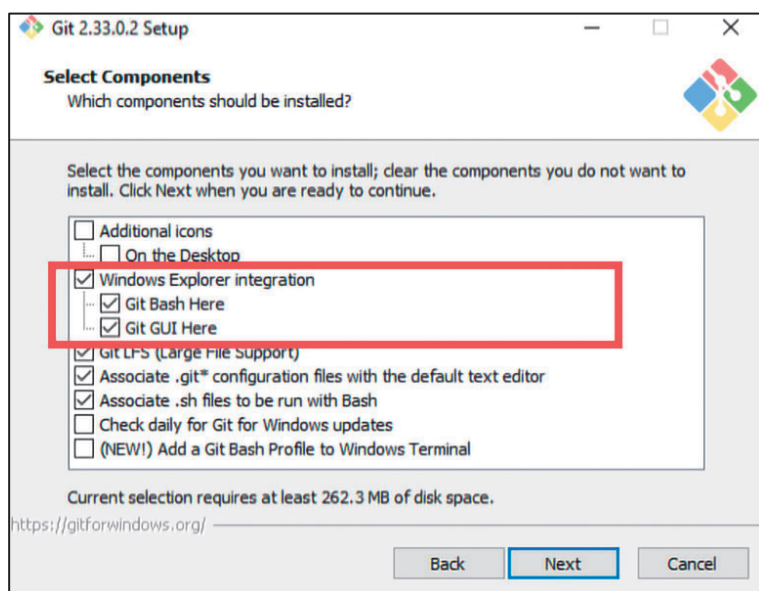
Aby zainstalować Gita ręcznie na komputerze z systemem Windows, musimy pobrać plik wykonywalny narzędzia **Git for Windows** ze strony <https://gitforwindows.org/>, a po pobraniu kliknąć plik wykonywalny i wykonać kolejne kroki instalacyjne:

1. Wybierz ścieżkę instalacji plików binarnych Gita. Zachowujemy domyślną ścieżkę, jak pokazano na poniższym zrzucie ekranu:



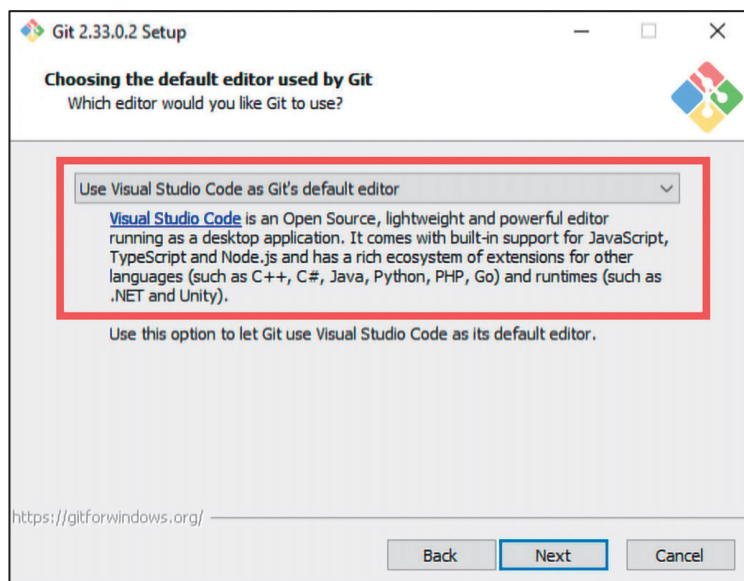
Rysunek 6.3. Ścieżka instalacji Gita

2. Wybierz komponenty Gita, zaznaczając pole *Windows Explorer integration*, jak pokazano na poniższym zrzucie ekranu:



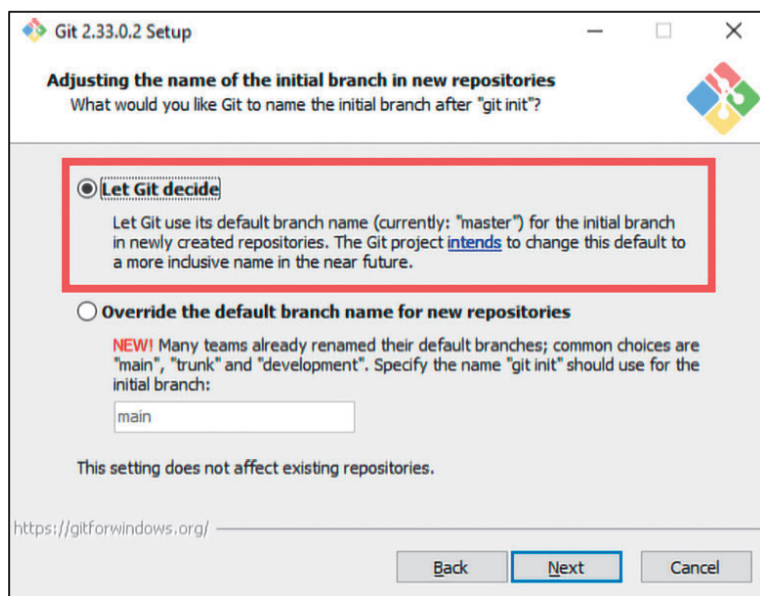
Rysunek 6.4. Wybór komponentów instalacyjnych Gita

3. Wybierz *zintegrowane środowisko programistyczne* (ang. *integrated development environment* — IDE) jako edytor kodu; w naszym przypadku używamy VS Code — opcja *Use Visual Studio Code as Git's default editor*, jak pokazano na poniższym zrzucie ekranu:

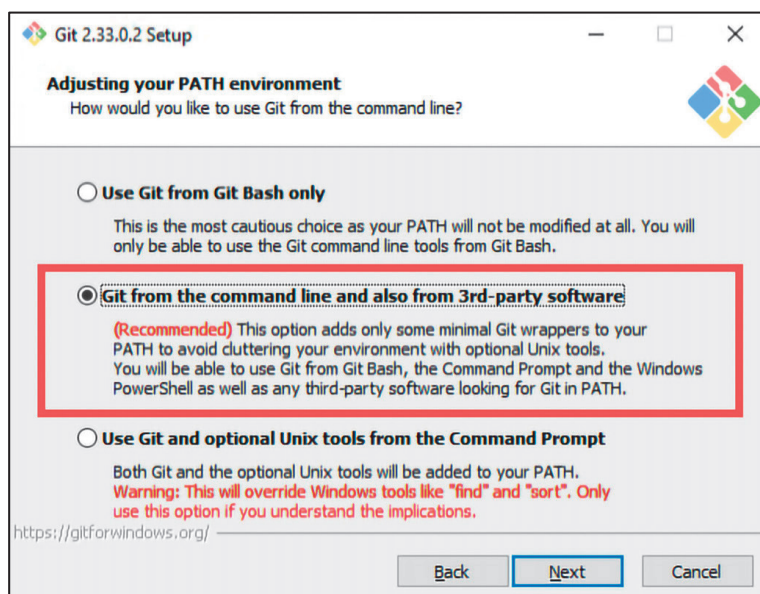


Rysunek 6.5. Wybór edytora Gita

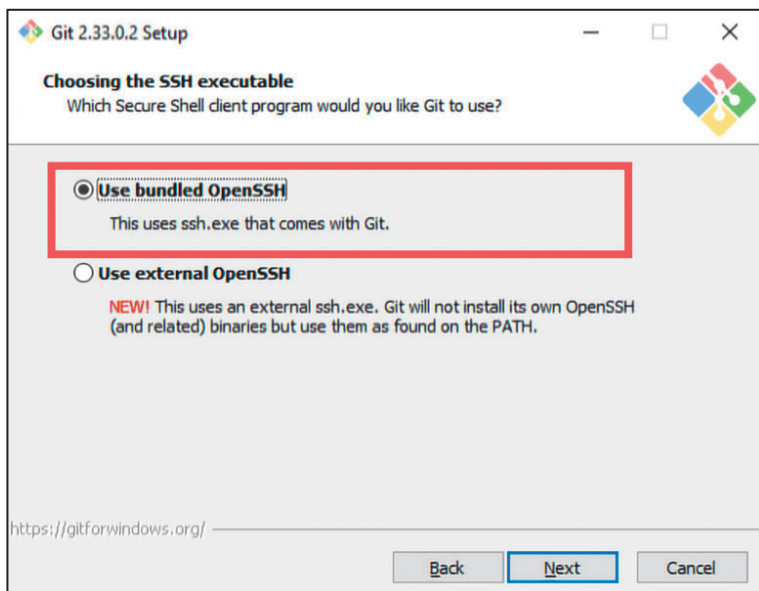
4. Skonfiguruj nazwę gałęzi (ang. *branch*) domyślnej. Możemy zachować opcję *master* jako domyślną nazwę gałęzi, jak pokazano na rysunku 6.6.
5. Wybierając opcję *Adjusting your PATH environment*, możemy pozostawić domyślny wybór zaproponowany przez instalatora, jak pokazano na rysunku 6.7.
6. Wybierz klienta *Secure Shell (SSH)*, zachowując domyślną opcję korzystania ze zintegrowanego klienta *ssh.exe*, jak pokazano na rysunku 6.8.
7. Wybierz rodzaj warstwy transportowej *HyperText Transfer Protocol Secure (HTTPS)*, którą również zostawimy z domyślną opcją *Use the OpenSSL library*, jak pokazano na rysunku 6.9.
8. Wybierz opcję kodowania plików końcowych. Wybierzemy również domyślną opcję, która archiwizuje pliki w formacie Unix, jak pokazano na rysunku 6.10.
9. Wybierz domyślny emulator terminala dla Git Bash. Wybieramy MinTTY, co ilustruje rysunek 6.11.



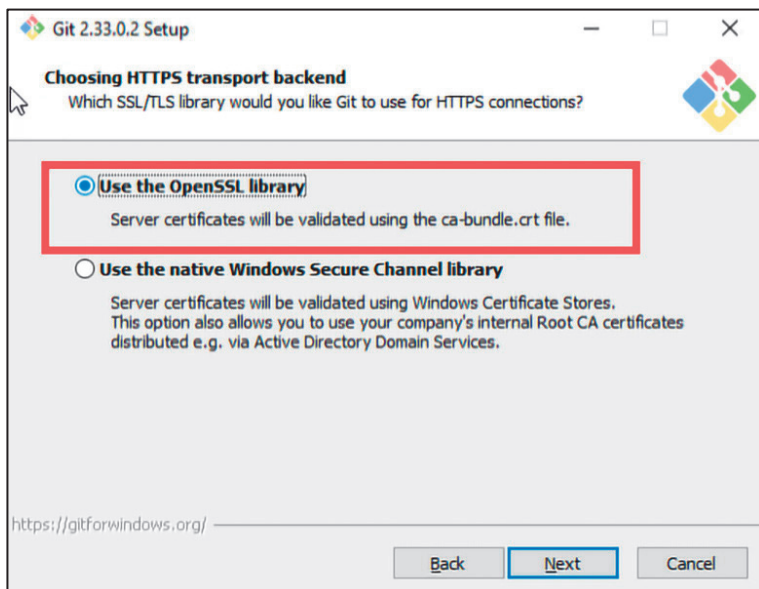
Rysunek 6.6. Domyślna nazwa gałęzi Gita



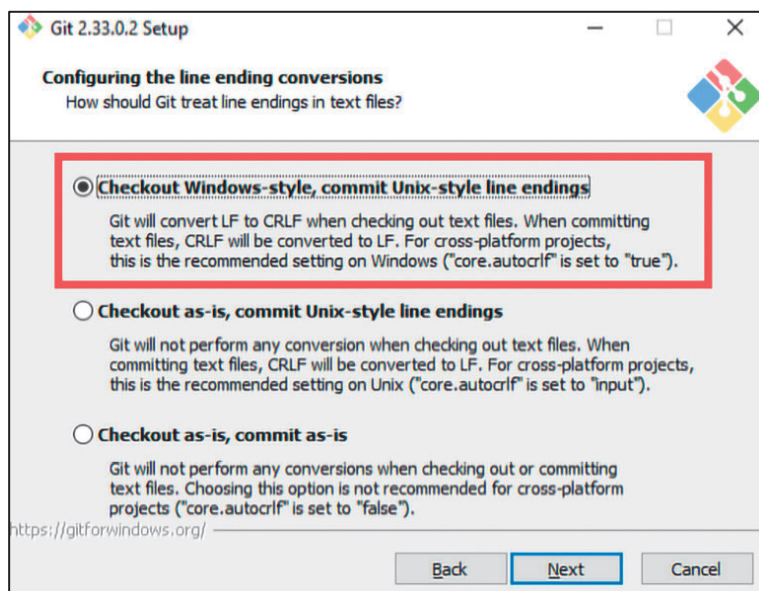
Rysunek 6.7. Konfiguracja PATH podczas instalacji Gita



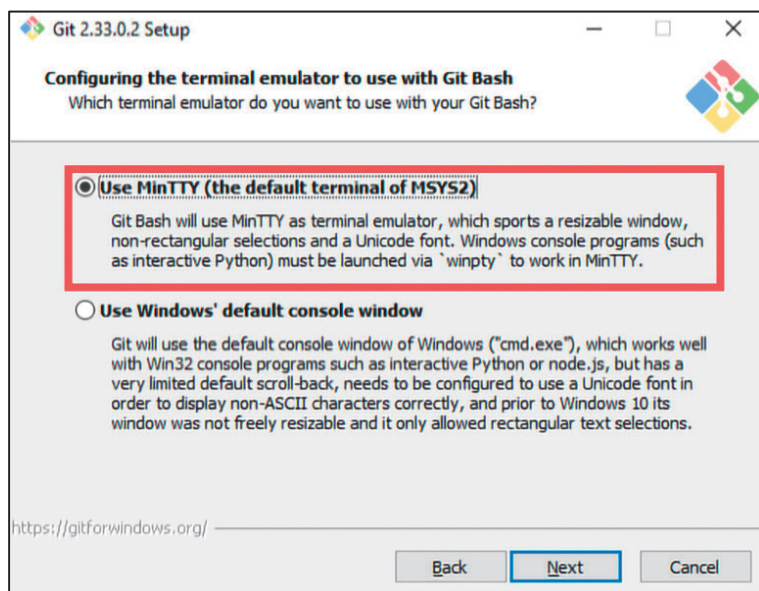
Rysunek 6.8. Wybór SSH podczas instalacji Gita



Rysunek 6.9. Wybór OpenSSL podczas instalacji Gita

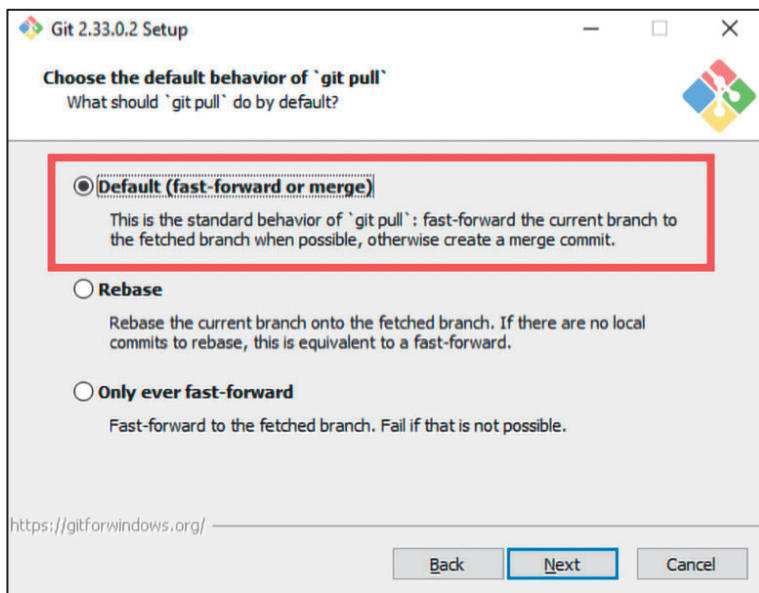


Rysunek 6.10. Wybór kodowania plików Gita



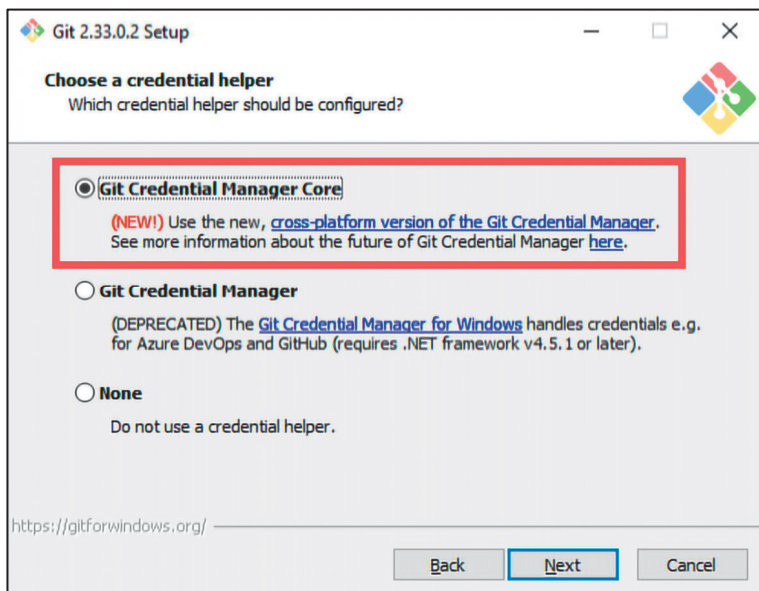
Rysunek 6.11. Emulator terminala Gita

10. Wybierz domyślne zachowanie polecenia `git pull`. Zachowujemy domyślną opcję, jak pokazano na poniższym zrzucie ekranu:



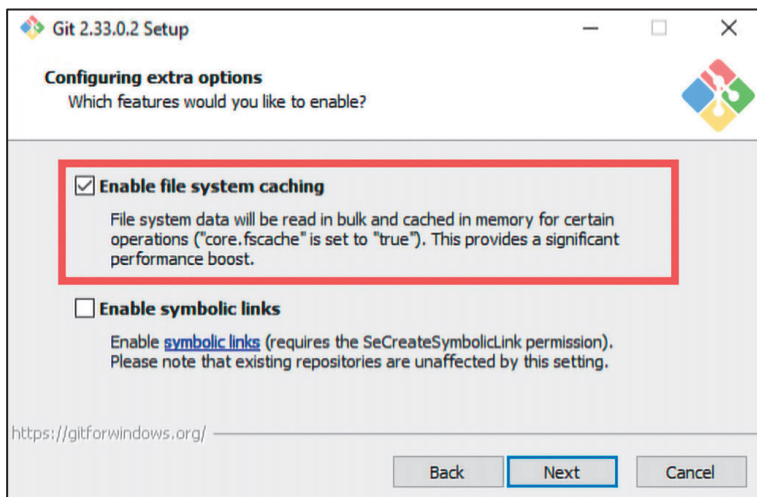
Rysunek 6.12. Domyślne zachowanie polecenia git pull

11. Wybierz menedżer poświadczeń, zachowując domyślną opcję, jak pokazano na poniższym zrzucie ekranu:



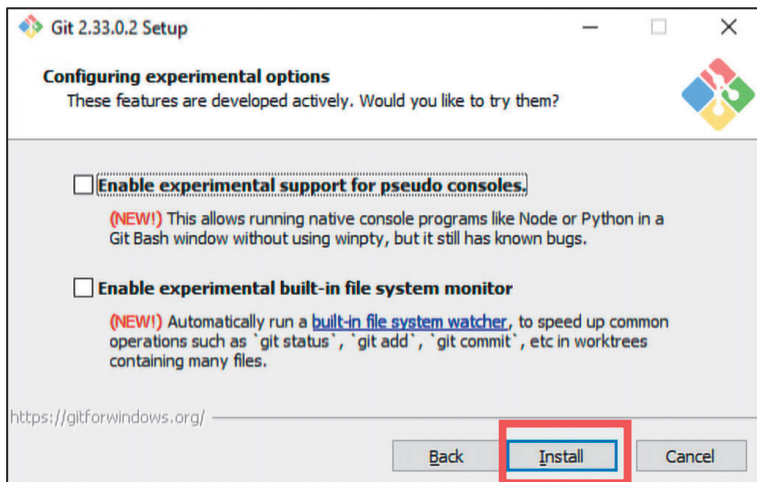
Rysunek 6.13. Menedżer poświadczeń Gita

12. Na kolejnym ekranie włączamy buforowanie systemu plików w następujący sposób:



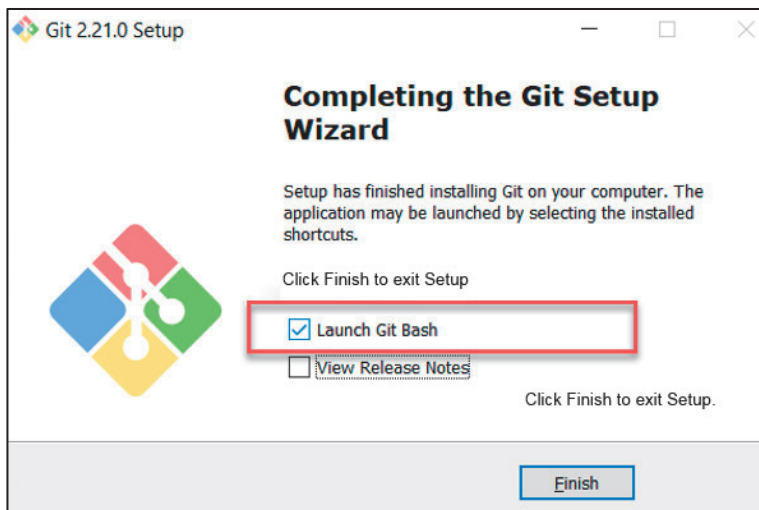
Rysunek 6.14. Buforowanie plików Gita

13. Następnie zakończ konfigurację instalacji, klikając przycisk *Install*, jak pokazano na poniższym zrzucie ekranu:



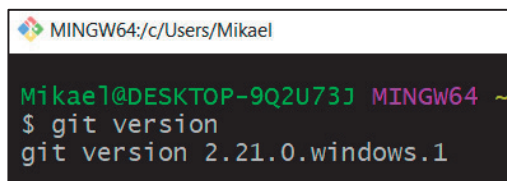
Rysunek 6.15. Ostatni krok instalacji Gita

Pod koniec narzędzie instalacyjne proponuje otwarcie Git Basha, który jest emulatorem wiersza poleceń Linuksa dedykowanym poleceniom Gita, jak pokazano na poniższym zrzucie ekranu:



Rysunek 6.16. Koniec instalacji Gita

Po instalacji możemy od razu sprawdzić stan instalacji Gita bezpośrednio w oknie Git Basha, uruchamiając polecenie `git version` w następujący sposób:



Rysunek 6.17. Wyświetlanie wersji Gita

Możemy również zainstalować Gita za pomocą automatycznego skryptu i Chocolatey, menedżera pakietów oprogramowania Windows, którego używaliśmy już w poprzednich rozdziałach o Terraform i Packerze. (Przypominamy, że dokumentacja Chocolatey jest dostępna tutaj: <https://chocolatey.org/>).

Aby zainstalować Gita dla Windowsa za pomocą Chocolatey, musimy wykonać następujące polecenie w działającym terminalu:

```
choco install -y git
```

Wynik tego wykonania jest wyświetlany na poniższym zrzucie ekranu:

```
PS C:\WINDOWS\system32> choco install -y git
Chocolatey v0.10.15
Installing the following packages:
git
By installing you accept licenses for the packages.
Progress: Downloading git.install 2.33.0.2... 100%
Progress: Downloading git 2.33.0.2... 100%

git.install v2.33.0.2 [Approved]
git.install package files install completed. Performing other installation steps.
Using Git LFS
Installing 64-bit git.install...
git.install has been installed.
WARNING: Can't find git.install install location
git.install can be automatically uninstalled.
Environment Vars (like PATH) have changed. Close/reopen your shell to
see the changes (or in powershell/cmd.exe just type `refreshenv`).
The install of git.install was successful.
Software installed to 'C:\Program Files\Git\'

git v2.33.0.2 [Approved]
git package files install completed. Performing other installation steps.
The install of git was successful.
Software install location not explicitly set, could be in package or
default install location if installer.

Chocolatey installed 2/2 packages.
See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).
```

Rysunek 6.18. Instalacja Gita za pomocą Chocolatey

Aby zainstalować Gita na komputerze z systemem **Linux** dla dystrybucji Debian takich jak Ubuntu, uruchamiamy polecenie `apt-get` w następujący sposób:

```
apt-get install git
```

Dla **CentOS** lub **Fedora** instalujemy Gita za pomocą polecenia `yum`:

```
yum install git
```

W przypadku systemu **macOS** możemy pobrać i zainstalować Gita za pomocą Homebrew (<https://brew.sh/>), który jest menedżerem pakietów dedykowanym dla macOS. Wykonaj to polecenie w terminalu:

```
brew install git
```

Instalacja Gita została zakończona, więc teraz przejdziemy do jego konfiguracji.

Konfiguracja Gita

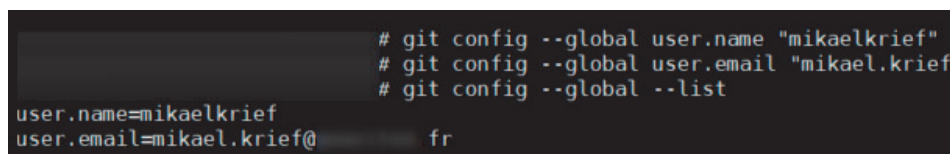
Konfiguracja Gita wymaga od nas skonfigurowania nazwy użytkownika i adresu e-mail, które będą używane podczas zatwierdzania (ang. *commit*) kodu. Aby przeprowadzić tę konfigurację, wykonujemy następujące polecenia w terminalu — albo tym, który jest natywny dla Twojego **systemu operacyjnego (OS)**, albo Git Bash dla Windowsa:

```
git config --global user.name "<Twoja nazwa użytkownika>"
git config --global user.email "<Twój adres e-mail>"
```

Następnie możemy sprawdzić wartości konfiguracyjne, wykonując następujące polecenie:

```
git config --global --list
```

Zobacz poniższy zrzut ekranu:



```
# git config --global user.name "mikaelkrief"
# git config --global user.email "mikael.krief"
# git config --global --list
user.name=mikaelkrief
user.email=mikael.krief@fr
```

Rysunek 6.19. Konfiguracja Gita

Git jest już skonfigurowany i gotowy do użycia, ale przed jego zastosowaniem przedstawimy terminologię.

Terminologia Gita

Git to narzędzie, które jest bardzo bogate w obiekty i terminologię. Posiada własne koncepcje.

Przed użyciem ważne jest, aby mieć pewną wiedzę na temat jego elementów i terminów, które go tworzą:

- **Repozytorium** — podstawowy element Gita; jest to przestrzeń do przechowywania, w której jest śledzony i wersjonowany kod źródłowy. Istnieją **zdalne repozytoria**, które centralizują kod zespołu i umożliwiają współpracę zespołową. Istnieje również **lokalne repozytorium**, które jest kopią repozytorium na lokalnym komputerze.
- **Klonowanie** (ang. *clone*) — czynność tworzenia lokalnej kopii zdalnego repozytorium.
- **Zatwierdzenie** (ang. *commit*) — zmiana dokonana na jednym lub kilku plikach, a zapisywana w lokalnym repozytorium. Każde zatwierdzenie jest

unikatowe i identyfikowane za pomocą unikalnego numeru, zwanego **SHA-1**, za pomocą którego można śledzić zmiany w kodzie.

- **Gałąź** (ang. *branch*) — kod znajdujący się w repozytorium jest domyślnie przechowywany w gałęzi *master*. Gałąź może tworzyć inne gałęzie, które będą repliką gałęzi głównej, na której programiści wprowadzają zmiany, co pozwoli nam pracować w izolacji bez wpływu na gałąź *master*. W każdej chwili możemy połączyć jedną gałąź z drugą.
- **Merge** — akcja polegająca na połączeniu kodu jednej gałęzi z drugą.
- **Checkout** — akcja, która pozwala nam przełączać się z jednej gałęzi do drugiej.
- **Fetch** — akcja pobierania kodu ze zdalnego repozytorium bez łączenia go z repozytorium lokalnym.
- **Pull** — akcja polegająca na aktualizacji lokalnego repozytorium zdalnym repozytorium. Jest równoważna operacjom *merge* i *fetch*.
- **Push** — działanie odwrotne do *pull*; pozwala nam aktualizować zdalne repozytorium na podstawie lokalnego repozytorium.
- **Pull request** (PR) — funkcja Gita (zainicjowana przez GitHuba), która uruchamia graficzny interfejs użytkownika (GUI) lub interfejs webowy w celu omówienia proponowanych zmian między użytkownikami zespołów przed zintegrowaniem ich z główną gałęzią. Jeśli chcesz uzyskać więcej informacji na temat PR, przeczytaj dokumentację GitHuba tutaj: <https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests>.

Oto pojęcia, które należy znać podczas korzystania z usługi Git. Oczywiście ta lista nie jest wyczerpująca i istnieją inne ważne terminy i pojęcia. Informacje możemy znaleźć na kilku stronach. Oto krótka lista:

- **Terminologia GitHuba** — <https://help.github.com/en/articles/github-glossary>.
- **Słowniczek Atlassian** — <https://www.atlassian.com/git/glossary/terminology>.

Po wyjaśnieniu tych pojęć możemy zobaczyć, jak używać Gita za pomocą poleceń.

Polecenia Gita

Skoro już zdobyliśmy wiedzę teoretyczną na temat Gita, możemy przejść do praktyki. Najlepszym sposobem na to jest nauka używania Gita za pomocą poleceń; gdy to opaujemy, będzie nam łatwiej korzystać z narzędzi graficznych.

Oto prezentacja poleceń Gita, które są teraz częścią (lub powinny być częścią) codziennego życia programistów. Ich zastosowanie w praktyce zobaczymy w następnej sekcji, „Zrozumienie procesu Gita i wzorca Gitflow”.

Pierwsze polecenie pozwala nam pobrać kod ze zdalnego repozytorium.

Pobieranie zdalnego repozytorium

Pierwszym poleceniem, które należy znać, jest polecenie `clone`, które tworzy kopię zdalnego repozytorium w celu utworzenia repozytorium lokalnego:

```
git clone <url zdalnego repozytorium>
```

Jedynym wymaganym parametrem jest **URL** (ang. *Uniform Resource Locator*) repozytorium (każde repozytorium można zidentyfikować za pomocą unikalnego adresu URL). Po wykonaniu tego polecenia zawartość zdalnego repozytorium jest pobierana na komputer lokalny, a lokalne repozytorium jest automatycznie tworzone i konfigurowane.

Inicjowanie lokalnego repozytorium

Zauważ, że `init` to polecenie Gita, które pozwala na utworzenie lokalnego repozytorium. Aby to zrobić w katalogu, który będzie zawierał Twoje lokalne repozytorium, uruchom następujące proste polecenie:

```
git init
```

Polecenie to tworzy katalog `.git`, który zawiera wszystkie foldery i pliki konfiguracyjne lokalnego repozytorium.

Konfiguracja lokalnego repozytorium

Po użyciu polecenia `init` nowe repozytorium lokalne musi zostać skonfigurowane poprzez ustawienie repozytorium zdalnego. Dokonamy tego za pomocą następującego polecenia:

```
git remote add <nazwa> <url zdalnego repozytorium>
```

Nazwa przekazana jako parametr umożliwia lokalną identyfikację tego zdalnego repozytorium; jest to odpowiednik aliasu. Możliwe jest również skonfigurowanie kilku zdalnych urządzeń w naszym lokalnym repozytorium.

Dodawanie pliku do następnego zatwierdzenia

Dokonanie zatwierdzenia (które zobaczymy dalej) polega na zarchiwizowaniu naszych zmian w naszym lokalnym repozytorium. Kiedy edytujemy pliki, możemy wybrać, które zostaną uwzględnione w następnym zatwierdzeniu; jest to koncepcja etapowa. Pozostałe niewybrane pliki zostaną odłożone na bok do późniejszego zatwierdzenia.

Aby dodać pliki do następnego zatwierdzenia, wykonujemy polecenie `add` w następujący sposób:

```
git add <ścieżka plików do dodania>
```

Na przykład jeśli chcemy, aby wszystkie pliki zostały zmodyfikowane przy następnym zatwierdzeniu, wykonujemy polecenie `git add`. Pliki do dodania możemy również filtrować za pomocą **wyrażeń regularnych (RegEx)**, np. `git add *.txt`.

Zatwierdzanie zmian

Zatwierdzenie to jednostka Gita zawierająca listę zmian wprowadzonych do plików, które zostały zarejestrowane w lokalnym repozytorium. Dokonywanie zatwierdzenia polega zatem na archiwizacji zmian dokonanych w plikach, które zostały wcześniej wybrane za pomocą polecenia `add`.

Polecenie, które dokonuje zatwierdzenia, jest pokazane tutaj:

```
git commit -m "<Twoja wiadomość dotycząca zatwierdzenia>"
```

Parametr `-m` odpowiada wiadomości lub opisowi, który przypisujemy do tego zatwierdzenia.

Wiadomość jest bardzo ważna, ponieważ będziemy mogli zidentyfikować przyczynę zmian w plikach.

Możliwe jest również zatwierdzenie wszystkich plików zmodyfikowanych od ostatniego zatwierdzenia bez konieczności wykonywania polecenia `add`, poprzez wykonanie polecenia `git commit -a -m "<wiadomość>"`.

Po wykonaniu zatwierdzenia zmiany są archiwizowane w lokalnym repozytorium.

Aktualizacja zdalnego repozytorium

Kiedy wykonujemy zatwierdzenia, są one przechowywane w lokalnym repozytorium, a kiedy jesteśmy gotowi udostępnić je reszcie zespołu w celu walidacji lub wdrożenia, musimy je opublikować w repozytorium zdalnym. Aby zaktualizować zdalne repozytorium zatwierdzeniami dokonanymi w lokalnym repozytorium, wykonuje się operację `push` za pomocą tego polecenia:

```
git push <alias> <branch>
```

Alias przekazany jako parametr odpowiada aliasowi zdalnego repozytorium skonfigurowanemu w konfiguracji Gita repozytorium lokalnego (wykonanej za pomocą polecenia `git remote add`).

Parametr `branch` to gałąź do aktualizacji — domyślnie jest to gałąź `master`.

Synchronizacja lokalnego repozytorium na podstawie zdalnego

Jak wspomnieliśmy wcześniej, wiersz poleceń Gita służy do aktualizacji zdalnego repozytorium na podstawie lokalnego repozytorium. Teraz, aby wykonać operację odwrotną — tzn. zaktualizować repozytorium o wszystkie zmiany innych członków, które zostały przekazane do zdalnego repozytorium — wykonamy operację `pull` za pomocą następującego polecenia:

```
git pull
```

Wykonanie tego polecenia prowadzi do dwóch operacji:

1. Scalanie kodu lokalnego z kodem zdalnym.
2. Zatwierdzanie lokalnego repozytorium.

Z drugiej strony, jeśli nie chcemy zatwierdzać — np. aby móc wprowadzać inne zmiany — to zamiast polecenia `pull` musimy wykonać `fetch` za pomocą tego polecenia:

```
git fetch
```

Jego wykonanie łączy tylko kod lokalny z kodem zdalnym, a żeby go zarchiwizować, będziemy musieli wykonać polecenie `commit`, aby zaktualizować lokalne repozytorium.

Zarządzanie gałęziami

Domyślnie podczas tworzenia repozytorium kod umieszczany jest w głównej gałęzi o nazwie `master`. Aby móc odizolować rozwój gałęzi `master` — np. opracować nową funkcję, naprawić błąd, a nawet przeprowadzić eksperymenty techniczne — możemy stworzyć nowe gałęzie z innych gałęzi i scalać je razem, gdy chcemy scalić ich kod.

Aby utworzyć gałąź z aktualnie załadowanej lokalnie gałęzi, wykonujemy następujące polecenie:

```
git branch <nazwa żądanej gałęzi>
```

Aby przejść do innej gałęzi, wykonujemy następujące polecenie:

```
git checkout <nazwa gałęzi>
```

Polecenie to zmienia gałąź i ładuje bieżący katalog roboczy z zawartością tej gałęzi.

Aby scalić gałąź z gałęzią bieżącą, wykonujemy polecenie `merge` w następujący sposób:

```
git merge <nazwa gałęzi>
```

Jako parametr podajemy nazwę gałęzi, którą chcemy scalić.

Na koniec, aby wyświetlić listę gałęzi lokalnych, wykonujemy polecenie `branch` w następujący sposób:

```
git branch
```

Zarządzanie gałęziami nie jest łatwe przy użyciu wiersza poleceń. Wspomniane już narzędzia graficzne Gita pozwalają na lepszą wizualizację i zarządzanie nimi. Przyjrzymy się im szczegółowo w kolejnej sekcji, która zajmie się procesem Gita. To wszystko na temat zwykłych poleceń Gita, chociaż jest ich o wiele więcej.

W następnej sekcji zastosujemy to wszystko w praktyce i przyjrzymy się omówieniu wzorca Gitflow.

Zrozumienie procesu Gita i wzorca Gitflow

Do tej pory poznaliśmy podstawy bardzo wydajnego systemu VCS, jakim jest Git, wraz z jego instalacją, konfiguracją i niektórymi z najczęstszych poleceń. W tej sekcji zastosujemy to wszystko w praktyce za pomocą studium przypadku pokazującego, który proces Gita należy wykorzystać w całym cyklu życia projektu.

W tym studium przypadku dla repozytoriów zdalnych użyjemy **Azure Repos** (jedna z usług Azure DevOps), będącej bezpłatną platformą chmurową Gita, którą można wykorzystać do projektów osobistych, a nawet biznesowych. Aby dowiedzieć się więcej o usłudze Azure DevOps, zapoznaj się z dokumentacją tutaj: <https://azure.microsoft.com/en-us/products/>. Często będziemy o tym mówić w tej książce.

Przyjrzymy się najpierw, jak przebiega współpraca z Gitem, a potem zobaczymy, jak wyizolować kod za pomocą gałęzi.

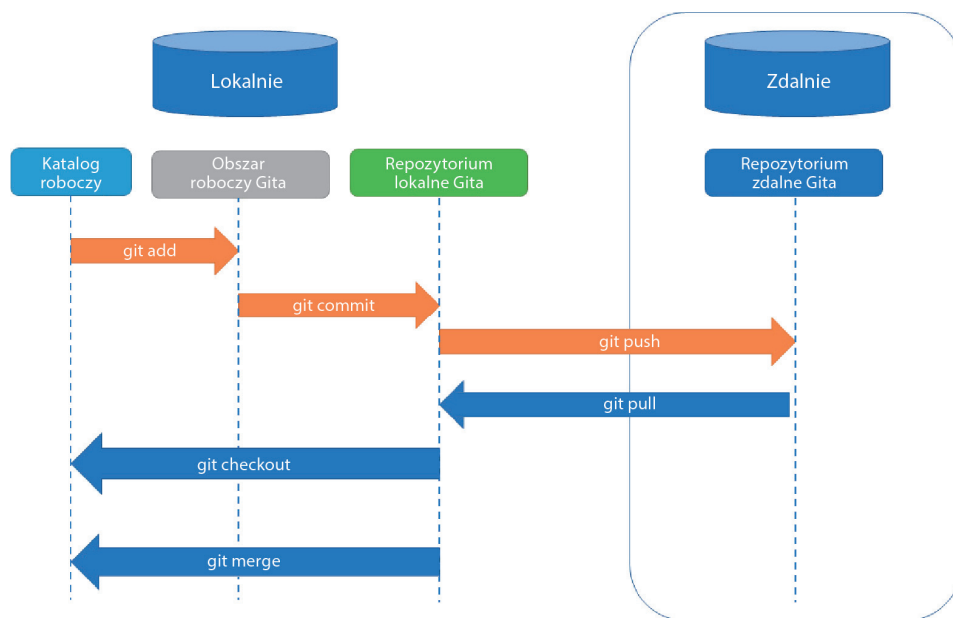
Zaczynamy od procesu Gita

W tym laboratorium wyjaśnimy proces współpracy Gita z zespołem dwóch programistów, którzy rozpoczynają nowy projekt tworzenia aplikacji.

Oto kroki procesu Gita, które omówimy szczegółowo w tej sekcji:

1. Pierwszy deweloper zatwierdza kod w lokalnym repozytorium i przesyła go do zdalnego repozytorium.
2. Następnie drugi programista pobiera wysłany kod ze zdalnego repozytorium.
3. Drugi deweloper aktualizuje ten kod, tworzy zatwierdzenie i przesyła nową wersję kodu do zdalnego repozytorium.
4. Na koniec pierwszy programista pobiera ostatnią wersję kodu do lokalnego repozytorium.

Przepływ danych Gitflow, o którym się dowiemy, zilustrowano na poniższym diagramie:



Rysunek 6.20. Przepływ danych w Gicie

Zanim jednak zaczniemy pracować z poleceniami Gita, przyjrzymy się krokom tworzenia i konfigurowania repozytorium Gita w Azure Repos.

Tworzenie i konfigurowanie repozytorium Gita

Na początek utworzymy zdalne repozytorium Gita w Azure DevOps, które posłuży do współpracy z innymi członkami zespołu. Wykonaj następujące kroki:

1. W Azure DevOps utworzymy nowy projekt jak na rysunku 6.21.
2. Wprowadzamy nazwę i opis projektu. Jak widać na poprzednim zrzucie, w polu *Version control* wybrano *Git*. Następnie w menu po lewej stronie klikamy usługę Azure Repos jak na rysunku 6.22.
3. Domyślne repozytorium jest już utworzone i ma taką samą nazwę jak projekt, co widać na rysunku 6.23.

Teraz, gdy repozytorium Gita zostało utworzone w Azure Repos, zobaczmy, jak zainicjować i skonfigurować nasz lokalny katalog roboczy do pracy z repozytorium Gita.

Create a project to get started


Project name *

BookDemo ✓


Description

Demo project

Visibility

 Public

Anyone on the internet can view the project. Certain features like TFVC are not supported.

 Private

Only people you give access to will be able to view this project.

^ Advanced

Version control ?

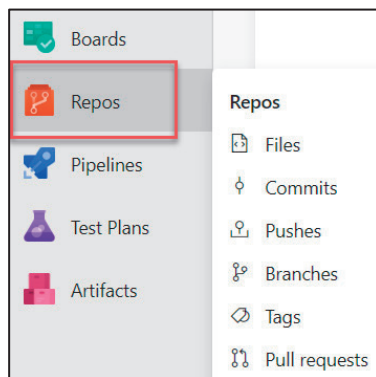
Git

Work item process ?

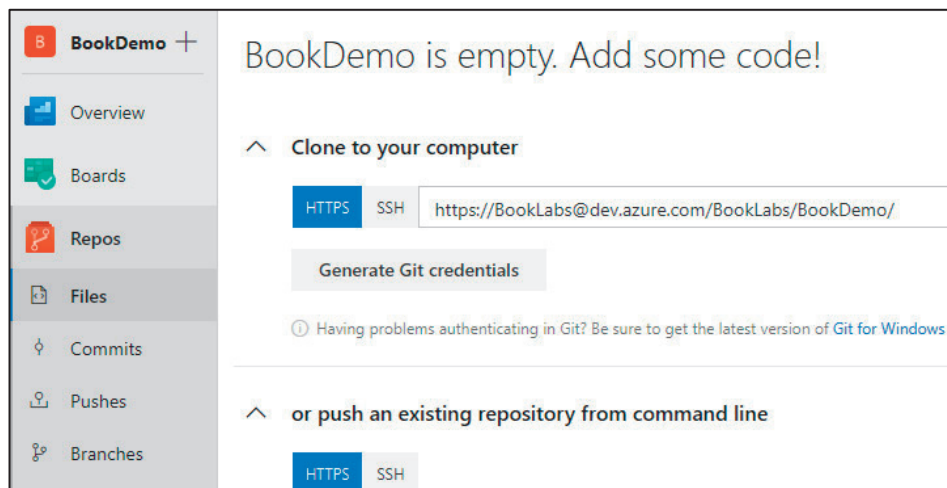
Basic

+ Create project

Rysunek 6.21. Tworzenie projektu Azure DevOps

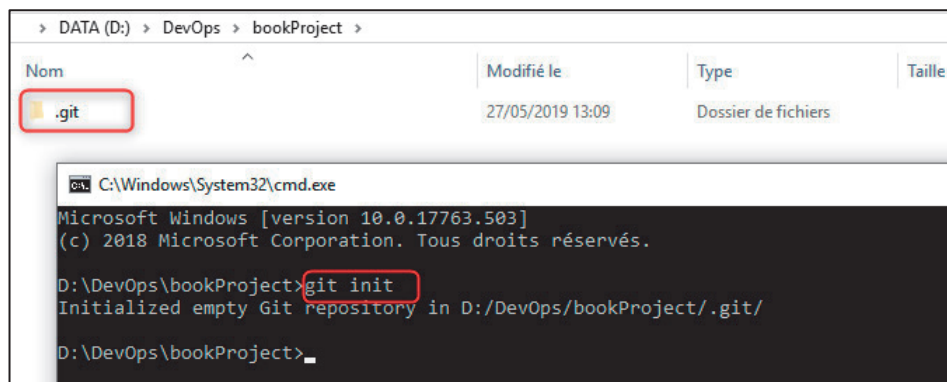


Rysunek 6.22. Menu usługi Azure Repos



Rysunek 6.23. Nowe repozytorium Azure DevOps

4. Aby zainicjować to repozytorium, utworzymy lokalny katalog i nazwiemy go (np. *bookProject*). Będzie zawierało kod aplikacji.
5. Po utworzeniu tego katalogu zainicjujemy lokalne repozytorium Gita, wykonując w folderze *bookProject* polecenie `git init` w następujący sposób:

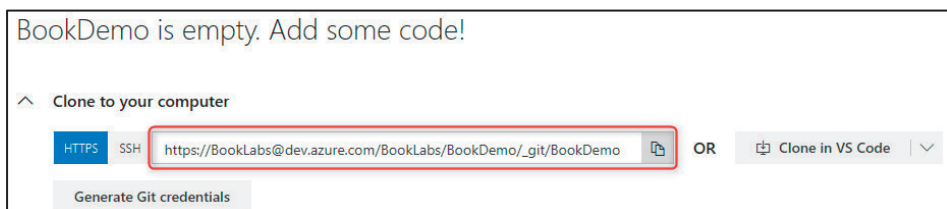


Rysunek 6.24. Polecenie `git init`

Wynikiem powyższego polecenia jest wiadomość potwierdzająca i nowy katalog *.git*. Ten nowy katalog będzie zawierał wszystkie informacje i konfigurację lokalnego repozytorium Gita.

6. Następnie skonfigurujemy to repozytorium lokalne, aby było połączone ze zdalnym repozytorium Azure DevOps. Połączenie to pozwoli na synchronizację repozytoriów lokalnych i zdalnych. W tym celu w Azure

DevOps otrzymujemy adres URL repozytorium, który znajduje się w informacjach o repozytorium, jak pokazano na poniższym zrzucie ekranu:



Rysunek 6.25. Adres URL repozytorium Azure DevOps

7. Uruchamiamy polecenie `git remote add` w folderze *bookProject*, który utworzyliśmy w kroku 4., w następujący sposób:

`git remote add origin <url zdalnego repozytorium>`

Wykonując powyższe polecenie, tworzymy alias *origin*, który będzie wskazywał na adres URL zdalnego repozytorium.

Uwaga

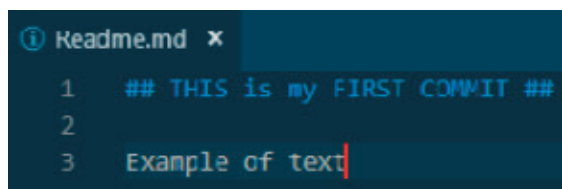
Procedura inicjalizacji i konfiguracji dotyczy również pustego katalogu, jak w naszym przypadku, lub katalogu, który zawiera już kod, który chcemy zarchiwizować.

Tak więc mamy teraz lokalne repozytorium Gita, nad którym będziemy pracować. Po utworzeniu i skonfigurowaniu naszego repozytorium zaczniemy tworzyć kod dla naszej aplikacji i współpracować z innymi przy użyciu procesu Gita, zaczynając od zatwierdzenia kodu.

Zatwierdzanie kodu

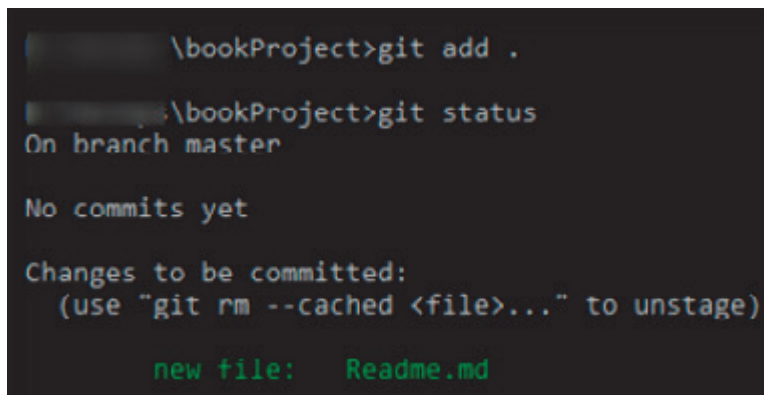
Pierwszym krokiem w procesie jest zatwierdzenie kodu, które umożliwia przechowywanie zmian kodu w lokalnym repozytorium Gita.

1. W naszym katalogu utworzymy plik *Readme.md* zawierający następujący przykładowy tekst:



Rysunek 6.26. Przykładowy plik README

2. Aby dokonać zatwierdzenia tego pliku w naszym lokalnym repozytorium, musimy dodać go do listy następnego zatwierdzenia, wykonując polecenie `git add .` (znak `.` na końcu oznacza włączenie wszystkich plików do modyfikacji, dodania lub usunięcia).
3. Możemy również zobaczyć status zmian, które zostaną wprowadzone w lokalnym repozytorium, uruchamiając polecenie `git status`, jak pokazano na poniższym zrzucie ekranu:



```
\bookProject>git add .  
  
\bookProject>git status  
On branch master  
  
No commits yet  
  
Changes to be committed:  
  (use "git rm --cached <file>..." to unstage)  
  
    new file:   README.md
```

Rysunek 6.27. Polecenie `git add .`

Widzimy, że tworzony jest plik *Readme.md*, który również zostanie dodany w czasie zatwierdzenia.

Ostatnią operacją do wykonania jest walidacja zmiany poprzez jej archiwizację w lokalnym repozytorium. W tym celu wykonujemy następujące polecenie:

```
git commit -m "Add file readme.md"
```

Wykonujemy zatwierdzenie z opisem, aby śledzić zmiany. Rysunek 6.28 przedstawia zrzut ekranu z wykonania poprzedniego polecenia.

Uwaga

W tym przykładzie zauważyliśmy również komunikat Gita, który zawiera informacje o konfiguracji Gita użytkownika.

Teraz, gdy nasze lokalne repozytorium Gita jest na bieżąco z naszymi zmianami, wystarczy je zarchiwizować w repozytorium zdalnym.


```
D:\DevOps\bookProject>git commit -m "Add file readme.md"
[master (root-commit) 0ffca3e] Add file readme.md
Committer: Mikael KRIFT <mikael.krief@younited-credit.fr>
your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

1 file changed, 3 insertions(+)
create mode 100644 Readme.md
```

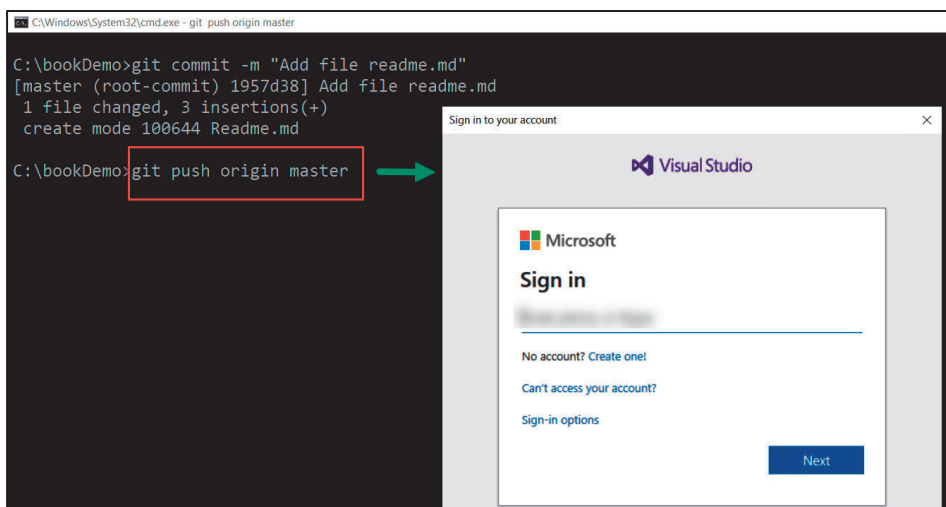
Rysunek 6.28. Polecenie git commit

Archiwizacja w zdalnym repozytorium

Aby zarchiwizować lokalne zmiany i umożliwić zespołowi pracę nad tym kodem, wyślemy nasze zatwierdzenie do zdalnego repozytorium, wykonując następujące polecenie:

git push origin master

Poleceniu `git push` wskazujemy alias i gałąź zdalnego repozytorium. Podczas pierwszego wykonania tego polecenia zostaniemy poproszeni o uwierzytelnienie w repozytorium Azure DevOps, jak pokazano na poniższym zrzucie ekranu:



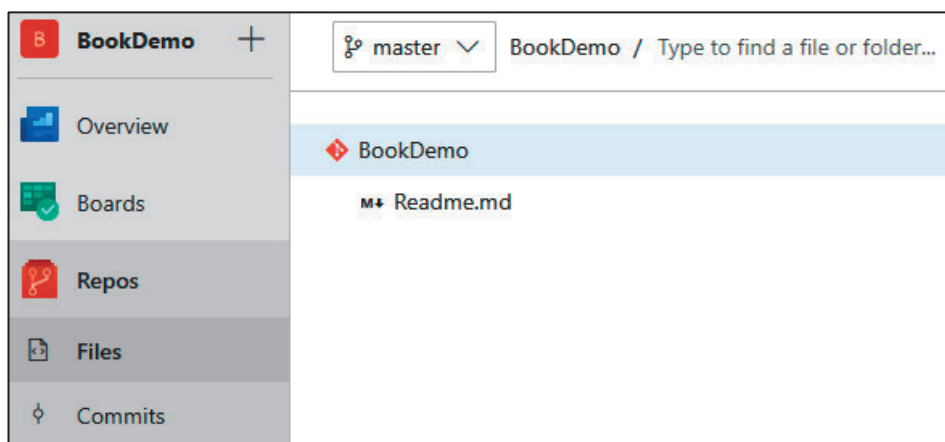
Rysunek 6.29. git push na Azure DevOps

Natomiast po uwierzytelnieniu wykonywane jest polecenie push:

```
\bookProject>git push origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 275 bytes | 275.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote: Analyzing objects... (3/3) (6 ms)
remote: Storing packfile... done (324 ms)
remote: Storing index... done (148 ms)
To https://dev.azure.com/BookLabs/BookDemo/_git/BookDemo
 * [new branch]      master -> master
```

Rysunek 6.30. Wynik git push

Zdalne repozytorium jest na bieżąco z naszymi zmianami, a w interfejsie Azure Repos możemy zobaczyć kod zdalnego repozytorium. Poniższy zrzut ekranu przedstawia katalog w Azure Repos, który zawiera dodany plik:



Rysunek 6.31. Nowy plik dodany do Azure Repos

To wszystko: wykonaliśmy pierwszy duży krok procesu Gita, który polega na zainicjowaniu repozytorium i przesłaniu kodu do zdalnego repozytorium. Kolejnym krokiem jest wprowadzenie zmian w kodzie przez innego członka zespołu.

Klonowanie repozytorium

Kiedy inny członek zespołu chce pobrać cały kod zdalnego repozytorium po raz pierwszy, wykonuje operację klonowania. Aby to zrobić, musi wykonać następujące polecenie:

```
git clone <adres URL repozytorium>
```

Wykonanie tego polecenia powoduje:

1. Utworzenie nowego katalogu o nazwie repozytorium.
2. Utworzenie lokalnego repozytorium wraz z jego inicjalizacją i konfiguracją.
3. Pobranie zdalnego kodu.

Członek zespołu może zatem wprowadzać zmiany w kodzie.

Aktualizacja kodu

Gdy kod zostanie zmodyfikowany, a programista zaktualizuje swoje zmiany w zdalnym repozytorium, wykonane zostaną dokładnie te same czynności co w przypadku inicjacji zdalnego repozytorium. Wykonane zostaną następujące polecenia:

```
git add .  
git commit -m "aktualizacja kodu"  
git push origin master
```

Dodaliśmy pliki do następnego zatwierdzenia, utworzyliśmy zatwierdzenie i wysłaliśmy je do zdalnego repozytorium.

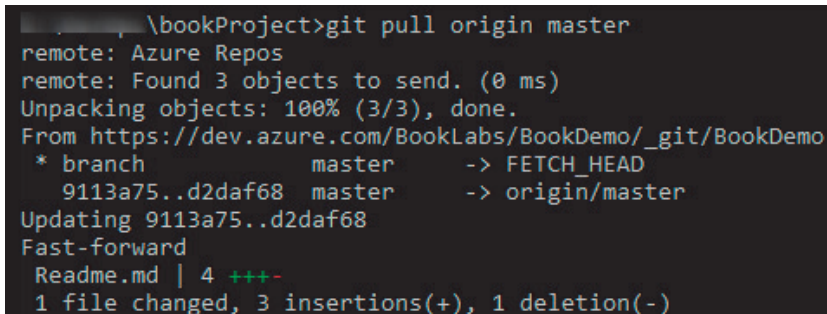
Ostatnim krokiem jest pobranie aktualizacji przez innych członków.

Pobieranie aktualizacji

Gdy jeden członek zaktualizuje zdalne repozytorium, możliwe jest pobranie tych aktualizacji i zaktualizowanie naszego lokalnego repozytorium o zmiany. Zrobimy to, uruchamiając polecenie `git pull` w następujący sposób:

```
git pull origin master
```

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia:



```
.\bookProject>git pull origin master  
remote: Azure Repos  
remote: Found 3 objects to send. (0 ms)  
Unpacking objects: 100% (3/3), done.  
From https://dev.azure.com/BookLabs/BookDemo/_git/BookDemo  
* branch          master      -> FETCH_HEAD  
   9113a75..d2daf68 master      -> origin/master  
Updating 9113a75..d2daf68  
Fast-forward  
  README.md | 4 +++-  
  1 file changed, 3 insertions(+), 1 deletion(-)
```

Rysunek 6.32. Polecenie `git pull`

Podczas wykonywania powyższego polecenia wskazujemy, że alias origin i gałąź master mają być zaktualizowane w lokalnym repozytorium. Zostaną wyświetlone wysłane zatwierdzenia. Po zakończeniu wykonywania tego polecenia nasze lokalne repozytorium jest zaktualizowane.

W pozostałej części procesu wykonywane są takie same operacje, jak widzieliśmy wcześniej w sekcji „Aktualizacja kodu”.

Właśnie opisaliśmy prosty proces korzystania z Gita. Możliwe jest również odizolowanie jednej części kodu od drugiej za pomocą gałęzi.

Izolacja kodu za pomocą gałęzi

Podczas tworzenia aplikacji często musimy zmodyfikować część kodu, nie chcąc wpływać na istniejący, stabilny kod aplikacji. W tym celu wykorzystamy funkcję obecną we wszystkich systemach VCS, która pozwala nam tworzyć gałęzie i zarządzać nimi.

Mechanika korzystania z gałęzi jest dość prosta, jak opisano poniżej:

1. Z gałęzi tworzymy kolejną gałąź.
2. Pracujemy na tej nowej gałęzi.
3. Jeśli pracujemy w zespołach, tworzymy PR, aby przedyskutować zmiany w kodzie. Jeśli recenzenci zgadzają się z tymi zmianami, zatwierdzają je.
4. Aby zastosować zmiany wprowadzone w nowej gałęzi do gałęzi oryginalnej, wykonywana jest operacja scalania.

Koncepcyjnie system gałęzi jest reprezentowany w następujący sposób:



Rysunek 6.33. Schemat rozgałęzień

Na tym diagramie widzimy, że gałąź **B** jest tworzona z gałęzi **A**. Te dwie gałęzie zostaną zmodyfikowane przez zespoły programistów w celu dodania nowej funkcji do naszej aplikacji. Ale w końcu gałąź **A** jest łączona z gałęzią **B**, a zatem zmienia się kod gałęzi **B**.

Teraz, gdy już poznaliśmy podstawy, przyjrzyjmy się, jak używać gałęzi, wykonując następujące kroki:

1. Celem tego ćwiczenia jest utworzenie gałęzi **Feature1** z gałęzi **master**. Następnie, po kilku zmianach w gałęzi **Feature1**, połączymy zmiany kodu z gałęzi **Feature1**

z gałęzią master. Najpierw utwórz gałąź Feature1 w lokalnym repozytorium z gałęzi master za pomocą następującego polecenia:

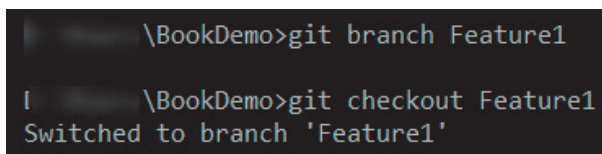
```
git branch Feature1
```

Wykonanie tego polecenia utworzyło nową gałąź Feature1, która zawiera dokładnie ten sam kod co gałąź nadrzędna, master.

2. Aby załadować kod tej gałęzi do katalogu roboczego, wykonujemy następujące polecenie:

```
git checkout Feature1
```

Wynik powyższego przełączenia gałęzi:



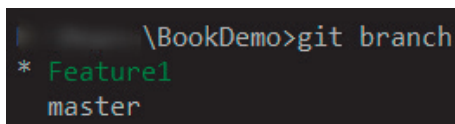
```
\BookDemo>git branch Feature1
I \BookDemo>git checkout Feature1
Switched to branch 'Feature1'
```

Rysunek 6.34. Przełączanie gałęzi Gita

3. Możemy również zobaczyć listę gałęzi naszego lokalnego repozytorium za pomocą następującego polecenia:

```
git branch
```

Wynik działania tego polecenia:



```
\BookDemo>git branch
* Feature1
master
```

Rysunek 6.35. Polecenie git branch

Podczas wykonywania widzimy dwie gałęzie — jedna z nich jest gałęzią aktywną.

4. Teraz wprowadzimy zmiany w naszym kodzie na aktywnej gałęzi. Mechanizm aktualizacji jest identyczny ze wszystkim, co widzieliśmy wcześniej, więc wykonamy następujące polecenia:

```
git add .
```

```
git commit -m "Dodaj kod funkcji 1"
```

5. W przypadku operacji push jako parametr określimy nazwę gałęzi Feature1 — za pomocą tego polecenia:

```
git push origin Feature1
```

Poniższy zrzut ekranu przedstawia wykonanie kodu, który wykonuje wszystkie te kroki:

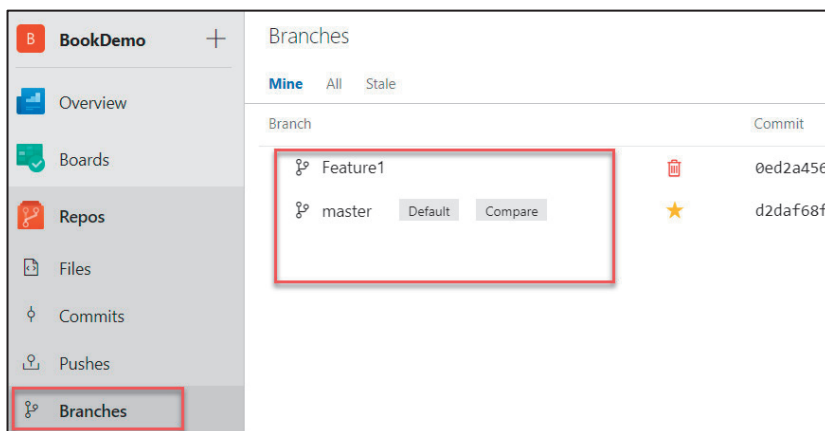
```
\BookDemo>git add .

\BookDemo>git commit -m "Add feature 1 code"
[Feature1 0ed2a45] Add feature 1 code
1 file changed, 3 insertions(+), 1 deletion(-)

\BookDemo>git push origin Feature1
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 306 bytes | 153.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Analyzing objects... (3/3) (97 ms)
remote: Storing packfile... done (188 ms)
remote: Storing index... done (92 ms)
To https://dev.azure.com/BookLabs/BookDemo/_git/BookDemo
 * [new branch]      Feature1 -> Feature1
```

Rysunek 6.36. Dodawanie gałęzi Gita

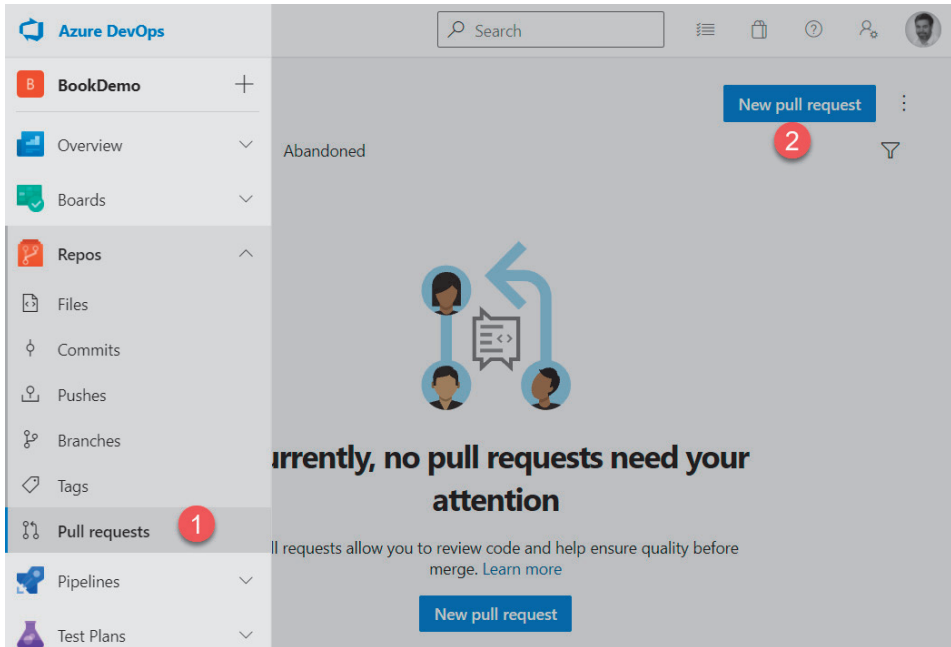
6. Za pomocą poprzednich poleceń, które wykonaliśmy, utworzyliśmy zatwierdzenie w gałęzi Feature1. Następnie wykonaliśmy polecenie push, aby opublikować nową gałąź, a także jej zatwierdzenie w zdalnym repozytorium. W interfejsie Azure Repos możemy zobaczyć nasze dwie gałęzie w sekcji *Branches*, jak pokazano na poniższym zrzucie ekranu:



Rysunek 6.37. Lista gałęzi Azure Repos

7. Przed scaleniem kodu tworzymy PR do dyskusji o zmianach. Poniższe zrzuty ekranu przedstawiają kroki tworzenia i zatwierdzania PR w Azure DevOps.

Zaczynamy od utworzenia PR w następujący sposób:



Rysunek 6.38. Azure Repos — tworzenie PR

Następnie wypełniamy formularz PR, wybierając gałąź źródłową i docelową, wpisując tytuł i opis oraz wybierając recenzentów. Możemy również zobaczyć różnicę w kodzie, który zostanie scalony. Proces ilustruje rysunek 6.39.

Na koniec recenzenci przejrzą, zatwierdzą lub odrzucą zmiany (mogą również scalać bezpośrednio), jak pokazano na rysunku 6.40.

8. Ostatnim krokiem w naszym procesie jest połączenie kodu gałęzi Feature1 z gałęzią master. Aby wykonać to scalenie, najpierw do naszego katalogu roboczego załadujemy gałąź master, a następnie połączymy gałąź Feature1 z gałęzią bieżącą, która jest gałęzią master.

Aby to zrobić, wykonamy następujące polecenia:

```
git checkout master  
git merge Feature1
```

Feature1 into master

Overview Files 2 Commits 2

Title

Update feature

Description

Add commit messages

Describe the code that is being reviewed

Markdown supported. Drag & drop, paste, or select files to insert. Link work items.

Reviewers

Add required reviewers

Mikael Krief Search users and groups to add as reviewers

Work items to link

Search work items by ID or title

Tags

Create

Rysunek 6.39. Formularz PR Azure Repos

Update feature

Active 2 Mikael Krief Feature1 into master

Overview Files Updates Commits

No merge conflicts

Last checked Just now

Description

Approve Complete

Rysunek 6.40. Zatwierdzenie PR Azure Repos

Wykonanie tych poleceń jest wyświetlane w następujący sposób:

```
\bookProject>git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

\bookProject>git merge Feature1
Updating 0ed2a45..3/b/8ab
Fast forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Rysunek 6.41. Scalanie gałęzi

Na końcu operacji kod gałęzi master zawiera zmiany dokonane w gałęzi Feature1.

Właśnie zobaczyliśmy przykład użycia gałęzi i ich łączenie. Teraz spójrzmy na wzorzec gałęzi Gitflow i jego użyteczność.

Strategia tworzenia gałęzi z Gitflow

Kiedy zaczynamy używać gałęzi w Gicie, często pojawia się pytanie: *Która strategia tworzenia gałęzi jest właściwa do zastosowania?* Innymi słowy: według jakiego klucza należy wyizolować nasz kod — według środowiska, funkcjonalności, motywu czy wydania?

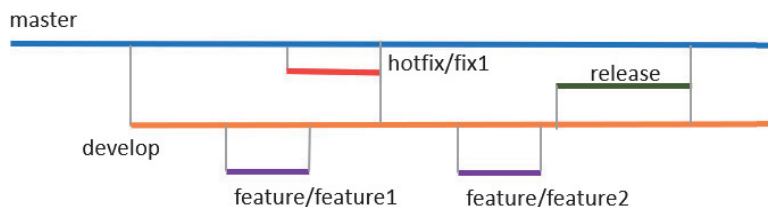
Na to pytanie nie ma uniwersalnej odpowiedzi, a zarządzanie gałęziami w ramach projektu zależy od kontekstu projektu. Istnieją jednak wzorce strategii, które zostały zatwierdzone przez kilka społeczności i wielu użytkowników, co pozwala na lepszy proces współpracy w projekcie w ramach Gita.

Przyjrzymy się bliżej jednemu z tych wzorców gałęzi, czyli Gitflow.

Wzorzec Gitflow

Jednym z takich wzorców jest **Gitflow**, opracowany przez *nvie*, który jest bardzo popularny i ma bardzo łatwy do nauczenia przepływ pracy.

Diagram gałęzi Gitflow jest pokazany tutaj:



Rysunek 6.42. Diagram Gitflow

Przyjrzyjmy się szczegółom przepływu pracy, a także celowi każdej z tych gałęzi w kolejności ich tworzenia:

1. Przede wszystkim mamy gałąź `master`, zawierającą kod, który jest aktualnie w produkcji. Żaden programista nie pracuje nad nim bezpośrednio.
2. Z gałęzi `master` tworzymy gałąź `develop`, czyli gałąź, która będzie zawierała zmiany, które zostaną wdrożone w kolejnym dostarczeniu aplikacji.
3. Dla każdej funkcjonalności aplikacji tworzona jest gałąź `feature/<nazwa właściwości>` (/ utworzy zatem katalog `feature`) z gałęzi deweloperskiej.
4. Zaraz po zakończeniu kodowania nowej właściwości dana gałąź jest scalana z gałęzią `develop`.
5. Następnie, gdy tylko będziemy chcieli wdrożyć wszystkie najnowsze opracowane właściwości, tworzymy gałąź `release` z `develop`.
6. Zawartość gałęzi `release` jest wdrażana sukcesywnie we wszystkich środowiskach.
7. Zaraz po wdrożeniu w środowisku produkcyjnym gałąź `release` jest scalana z gałęzią `master` i w przypadku tej operacji gałąź `master` zawiera kod produkcyjny. W związku z tym kod znajdujący się w gałęzi `release` i w gałęzi właściwości nie jest już potrzebny, a gałęzie te można usunąć.
8. W przypadku wykrycia błędu w środowisku produkcyjnym tworzymy gałąź `hotfix/<nazwa błędu>`; następnie, po naprawieniu błędu, łączymy tę gałąź z gałęzią główną i rozwijamy gałęzie, aby propagować poprawkę w kolejnych gałęziach i wdrożeniach.

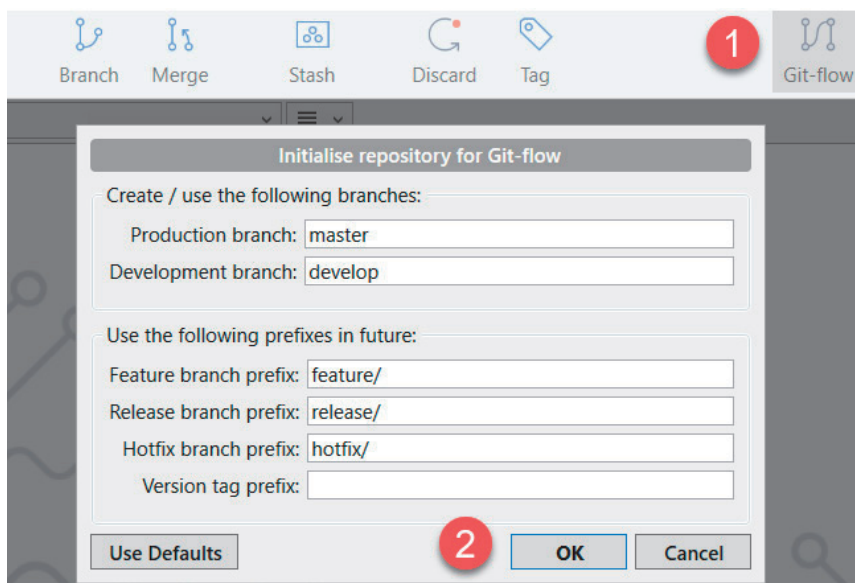
Po obejrzeniu przepływu pracy wzorca Gitflow omówimy narzędzia ułatwiające jego implementację.

Narzędzia Gitflow

Aby pomóc zespołom i programistom w korzystaniu z Gitflow, *nvie* utworzyła narzędzie wiersza poleceń Gita, które umożliwia łatwe tworzenie gałęzi zgodnie z powyższym wzorcem. Narzędzie to jest dostępne na stronie GitHuba *nvie* tutaj: <https://github.com/nvie/gitflow>.

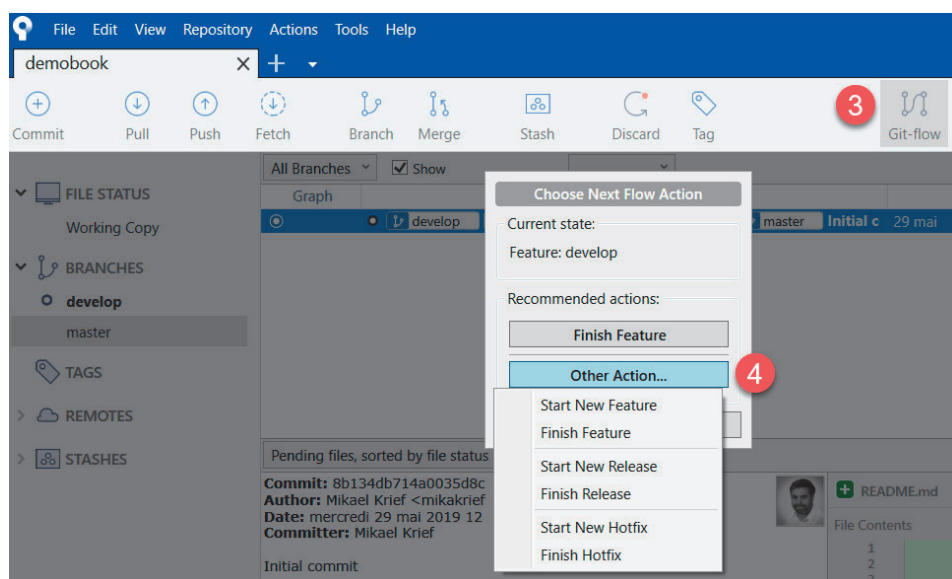
Istnieją również narzędzia graficzne Gita, które obsługują model Gitflow, takie jak GitKraken (<https://www.gitkraken.com/>) i Sourcetree (<https://www.sourcetreeapp.com/>). Narzędzia te pozwalają nam korzystać z procesu Gitflow za pomocą interfejsu graficznego, jak pokazano na poniższym zrzucie ekranu. Wykorzystano tutaj Sourcetree z funkcją *Git-flow*, która pozwala nam utworzyć gałąź przy użyciu wzorca Gitflow.

Poniższy zrzut ekranu przedstawia konfigurację Sourcetree dla wzorca Gitflow:



Rysunek 6.43. Narzędzie Git-flow od Sourcetree

Następnie, po tej konfiguracji, możemy łatwo utworzyć gałąź o tej nazwie, jak pokazano na poniższym zrzucie ekranu:



Rysunek 6.44. Tworzenie gałęzi za pomocą Sourcetree

Jak widać na poprzednich zrzutach ekranu, korzystamy z Sourcetree i z interfejsu graficznego, który pozwala nam intuicyjnie tworzyć gałęzie.

Uwaga

Aby uzyskać więcej dokumentacji na temat Gitflow i jego procesu, przeczytaj ten artykuł: <https://jeffkreeftmeijer.com/git-flow/>.

Aby zacząć korzystać z Gitflow, sugeruję przyzwyczaić się do małych projektów — przekonasz się, że mechanizm jest taki sam dla większych projektów.

W tej sekcji zapoznaliśmy się z procesem przepływu pracy Gita z wykorzystaniem linii poleceń. Następnie rozmawialiśmy o zarządzaniu gałęziami deweloperskimi. Na końcu zapoznaliśmy się z zarządzaniem gałęziami za pomocą Gitflow — pozwala to na proste zarządzanie gałęziami i lepszy przepływ pracy deweloperskiej za pomocą Gita.

Podsumowanie

Git jest dziś niezbędnym narzędziem dla wszystkich programistów; pozwala nam korzystać z wierszy poleceń lub narzędzi graficznych oraz udostępniać i wersjonować kod w celu lepszej współpracy w zespole.

W tym rozdziale zobaczyliśmy, jak zainstalować Gita w różnych systemach operacyjnych, i omówiliśmy główne polecenia. Następnie zobaczyliśmy przepływ pracy Gita z zastosowaniem wierszy poleceń. Na koniec przedstawiliśmy izolację kodu z implementacją gałęzi i wykorzystaniem wzorca Gitflow, co daje prosty model strategii tworzenia gałęzi.

W kolejnym rozdziale porozmawiamy o CI/CD, jednej z kluczowych praktyk kultury DevOps.

Pytania

1. Co to jest Git?
2. Jakie polecenie służy do inicjalizacji repozytorium?
3. Jak nazywa się artefakt Gita służący do zapisywania części kodu?
4. Która komenda Gita pozwala na zapisanie kodu w lokalnym repozytorium?
5. Które polecenie Gita umożliwia wysłanie kodu do zdalnego repozytorium?

6. Które polecenie Gita umożliwia aktualizację lokalnego repozytorium ze zdalnego repozytorium?
7. Który mechanizm Gita jest używany do izolacji kodu?
8. Co to jest Gitflow?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o Gicie, oto kilka zasobów:

- Dokumentacja Gita — <https://www.git-scm.com/doc>.
- Książka *Pro Git* — <https://git-scm.com/book/en/v2>.
- Książka *Mastering Git* — <https://www.packtpub.com/application-development/mastering-git>.
- *Set up Git* — <https://try.github.io/>.
- *Learn Git Branching* — <https://learngitbranching.js.org/>.
- Samouczki Atlassian Git — <https://www.atlassian.com/git/tutorials>.
- *git-flow cheat sheet* — <https://danielkummer.github.io/git-flow-cheatsheet/>.

Ciągła integracja i ciągłe wdrażanie

Jednym z głównych filarów kultury DevOps jest **wdrażanie procesów ciągłej integracji (CI)** i ciągłego wdrażania, jak wyjaśniliśmy w rozdziale 1., „Kultura DevOps i praktyki kodowania infrastruktury”.

W poprzednim rozdziale przyjrzelśmy się wykorzystaniu Gita z jego poleceniami i przepływem pracy, a w tym rozdziale przyjrzymy się ważnej roli, jaką Git odgrywa w przepływie CI/CD.

CI to proces, który zapewnia szybką informację zwrotną na temat spójności i jakości kodu wszystkim członkom zespołu. Występuje w czasie zatwierdzania kodu każdego użytkownika. Kod jest pobierany ze zdalnego repozytorium, a następnie jest scalany, kompilowany i testowany.

Ciągłe dostarczanie (CD) to automatyzacja procesu, który wdraża aplikację na różnych etapach (lub w różnych środowiskach).

W tym rozdziale wyjaśnimy zasady procesu CI/CD i omówimy jego praktyczne zastosowanie. Zapoznamy się z różnymi narzędziami, takimi jak **Jenkins**, **Azure Pipelines** i **GitLab CI**. W przypadku każdego z tych narzędzi przedstawimy zalety, wady i najlepsze praktyki oraz przyjrzymy się praktycznemu przykładowi implementacji potoku CI/CD.

Poznasz koncepcję potoku CI/CD. Następnie przyjrzymy się menedżerom pakietów i roli, jaką odgrywają w potoku.

Ponadto dowiesz się, jak zainstalować Jenkinsa, i na koniec — jak zbudować potok CI/CD na Jenkinsie, Azure Pipelines i GitLab CI.

Ten rozdział obejmuje następujące tematy:

- zasady CI/CD,
- korzystanie z menedżera pakietów w procesie CI/CD,
- używanie Jenkinsa do implementacji CI/CD,
- korzystanie z Azure Pipelines dla CI/CD,
- korzystanie z GitLab CI.

Wymagania techniczne

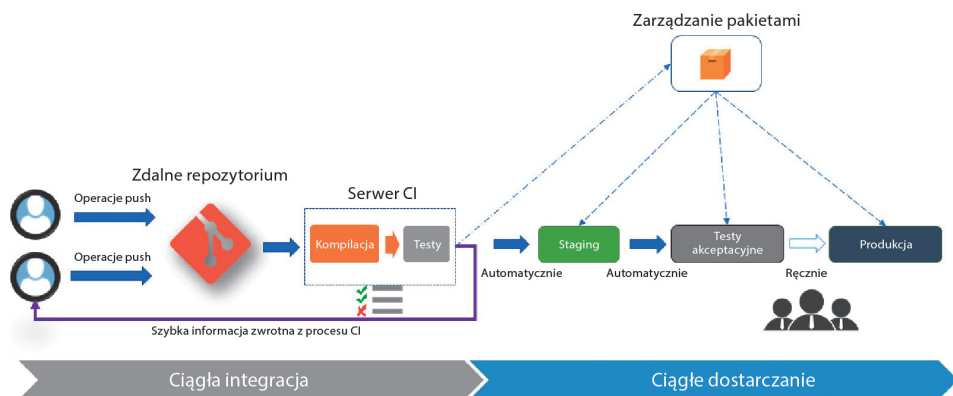
Jedynym wymaganiem tego rozdziału jest posiadanie zainstalowanego w systemie Gita, jak opisano w poprzednim rozdziale.

Kod źródłowy tego rozdziału jest dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP07>.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/3s9G8Pj>.

Zasady CI/CD

Aby wdrożyć potok CI/CD, ważne jest poznanie różnych elementów, które będą wymagane do zbudowania wydajnego i bezpiecznego potoku. Aby zrozumieć zasady CI/CD, spójrz na poniższy diagram, który przedstawia różne etapy przepływu CI/CD. Widzieliśmy je już w rozdziale 1., „Kultura DevOps i praktyki kodowania infrastruktury”:



Rysunek 7.1. Przebieg pracy CI/CD

Przyjrzyjmy się szczegółowo każdemu z tych kroków, aby opisać poszczególne elementy procesu CI/CD.

CI

Podczas fazy CI sprawdzany jest kod zarchiwizowany przez członków zespołu. Faza ta musi zostać wykonana przy każdym zatwierdzeniu, które zostało wysłane do zdalnego repozytorium.

Niezbędnym warunkiem wstępnym jest skonfigurowanie **systemu kontroli wersji kodu źródłowego** (ang. *source version control* — **SVC**) typu Git. Umożliwia on scentralizowanie kodu wszystkich członków zespołu.

Zespół będzie musiał zdecydować się na gałąź kodu, która zostanie wykorzystana dla CI. Na przykład dla Gitflow możemy użyć gałęzi `master` lub gałęzi `develop`; po prostu musi to być aktywna gałąź, która regularnie przyjmuje zmiany w kodzie.

Ponadto CI osiąga się dzięki automatycznemu zestawowi zadań, które są wykonywane na serwerze, zgodnie z podobnymi wzorcami wykonywanymi na laptopie programisty, który ma niezbędne narzędzia dla CI; ten serwer nazywa się **serwerem CI**.

Serwer CI może być serwerem typu on-premises (lokalnym), zainstalowanym w firmowym centrum danych, takim jak Jenkins lub TeamCity, lub może być typu chmurowego, o którego instalację i utrzymanie nie musimy się martwić, np. Azure Pipelines lub GitLab CI.

Zadania wykonywane w fazie CI muszą być zautomatyzowane i uwzględniać wszystkie elementy niezbędne do weryfikacji kodu.

Te zadania to generalnie kompilacja kodu i wykonanie testów jednostkowych kodu. Możemy również dodać statyczną analizę kodu za pomocą SonarQube (lub SonarCloud), którą omówimy w rozdziale 12., „Statyczna analiza kodu za pomocą SonarQube”.

Pod koniec zadań weryfikacyjnych w wielu przypadkach CI generuje pakiet aplikacji, który zostanie wdrożony w różnych środowiskach (zwanym również **obszarami pośrednimi** — ang. *stages*).

Aby móc hostować ten pakiet, potrzebujemy menedżera pakietów, zwanego również **menedżerem repozytorium**, który może być zainstalowany lokalnie (ang. *on-premises*), takiego jak Nexus, Artifactory, ProGet, lub rozwiązania typu **oprogramowanie jako usługa** (ang. *software-as-a-service* — SaaS), np.: Azure Pipelines, Azure Artifacts lub rejestr pakietów GitHuba. Ten pakiet musi być również neutralny pod względem konfiguracji środowiska i musi być wersjonowany, aby w razie potrzeby wdrożyć aplikację z poprzedniej wersji.

CD

Po spakowaniu aplikacji i przechowywaniu jej w menedżerze pakietów (podczas CI) proces CD jest gotowy do pobrania pakietu i wdrożenia go w różnych środowiskach.

Wdrożenie w każdym środowisku składa się z szeregu zautomatyzowanych zadań, które są również wykonywane na zdalnym serwerze, mającym dostęp do różnych środowisk.

Niezbędne jest zatem zaangażowanie Dev, Ops, a także zespołu bezpieczeństwa we wdrażanie narzędzi i procesów CI/CD. Będzie to połączenie ludzi z narzędziami i procesami, które będą wdrażać aplikacje na różnych serwerach lub zasobach w chmurze z poszanowaniem zasad sieciowych, ale także standardów bezpieczeństwa firmy.

W fazie wdrożenia często konieczna jest modyfikacja konfiguracji aplikacji w generowanym pakiecie w celu dostosowania jej do środowiska docelowego. Dlatego konieczne jest zintegrowanie **menedżera konfiguracji**, który jest już obecny w popularnych narzędziach CI/CD, takich jak Jenkins, Azure Pipelines lub Octopus Deploy. Ponadto, gdy pojawia się nowy klucz konfiguracyjny, dobrą praktyką jest wprowadzanie go do każdego środowiska, w tym produkcji, przy zaangażowaniu zespołu Ops.

Wreszcie wyzwanie wdrożenia może odbywać się automatycznie. Jednak w przypadku środowisk, które są bardziej krytyczne (np. środowiska produkcyjne), firmy ściśle regulowane mogą mieć ograniczenia, które wymagają ręcznego wyzwania przy kontroli osób upoważnionych.

Poniżej wymieniono różne narzędzia do konfigurowania potoku CI/CD:

- SVC,
- menedżer pakietów,
- serwer CI,
- menedżer konfiguracji.

Ale nie zapominajmy, że wszystkie te narzędzia będą naprawdę skuteczne tylko wtedy, gdy zespoły deweloperskie i operacyjne będą współpracować.

Właśnie przyjrzeliśmy się zasadom implementacji potoku CI/CD. W dalszej części rozdziału przyjrzymy się praktycznej implementacji za pomocą różnych narzędzi, zaczynając od menedżerów pakietów.

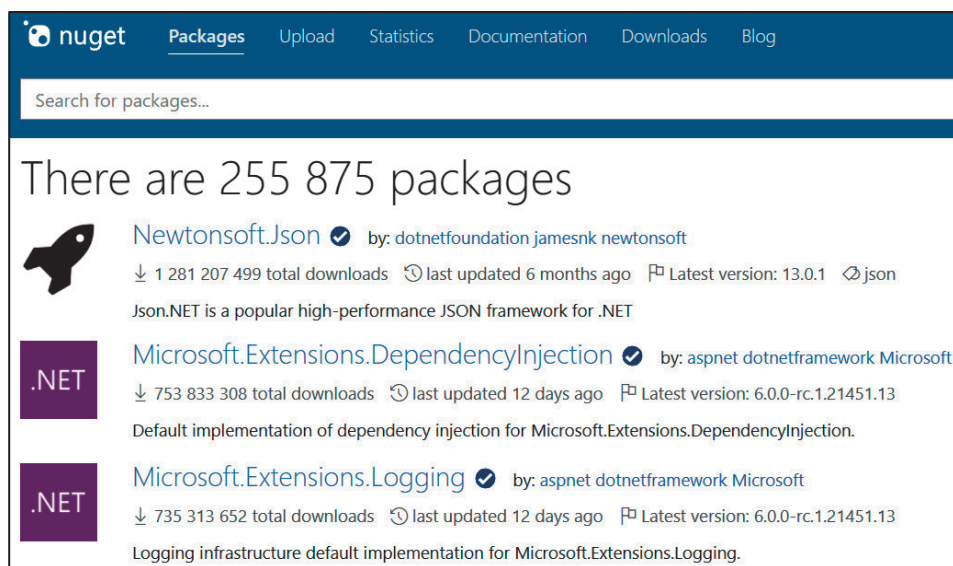
Korzystanie z menedżera pakietów w procesie CI/CD

Menedżer pakietów to centralne repozytorium, które skupia i udostępnia pakiety, biblioteki programistyczne, narzędzia i oprogramowanie.

Dla klientów konsumenckich, którzy korzystają z menedżerów pakietów, korzyścią jest możliwość śledzenia, aktualizacji, instalacji i usuwania zainstalowanych pakietów.

Istnieje wiele publicznych menedżerów pakietów, takich jak NuGet, *Node Package Manager* (npm), Maven, Bower i Chocolatey, które udostępniają struktury lub narzędzia dla deweloperów w różnych językach i na różnych platformach.

Poniższy zrzut ekranu pochodzi z menedżera pakietów NuGet, który publicznie udostępnia ponad 150 tys. platform .NET:



Rysunek 7.2. Menedżer pakietów NuGet

Jedną z zalet używania tego typu menedżera pakietów przez programistów jest to, że nie muszą oni przechowywać pakietów z kodem źródłowym aplikacji. Mogą stosować referencje w pliku konfiguracyjnym, dzięki czemu pakiety będą automatycznie pobierane.

W aplikacji korporacyjnej sprawy mają się nieco inaczej. Chociaż programiści używają pakietów z menedżerów publicznych, niektóre elementy generowane w przedsiębiorstwie muszą pozostać wewnętrzne.

Rzeczywiście, często zdarza się, że platformy (takie jak biblioteki NuGet lub npm) są opracowywane wewnętrznie i nie mogą być udostępniane publicznie. Co więcej, jak widzieliśmy w potoku CI/CD, musimy utworzyć pakiet dla naszej aplikacji i przechowywać go w menedżerze pakietów, który będzie prywatny dla firmy.

Dlatego zaleca się przyjrzenie się menedżerom pakietów takim jak NuGet i npm, których można używać w przedsiębiorstwie lub na potrzeby osobiste.

Prywatne repozytorium NuGet i npm

Jeśli chcesz scentralizować pakiety NuGet lub npm, możesz utworzyć własne repozytorium lokalne.

Aby utworzyć instancję serwera NuGet, zapoznaj się z dokumentacją firmy Microsoft: <https://docs.microsoft.com/en-us/nuget/hosting-packages/overview>.

npm możemy też zainstalować lokalnie za pomocą pakietu `npm local-npm`, którego dokumentacja jest dostępna tutaj: <https://www.npmjs.com/package/local-npm>.

Problem z instalacją jednego repozytorium dla jednego typu pakietu polega na tym, że musimy zainstalować i utrzymywać repozytorium oraz jego infrastrukturę dla różnych typów pakietów.

Dlatego lepiej jest przełączyć się na uniwersalne rozwiązania repozytorium, takie jak Nexus (Sonatype), ProGet i Artifactory dla rozwiązań lokalnych oraz Azure Artifacts, MyGet lub Artifactory dla rozwiązań SaaS.

Aby zrozumieć, jak działa menedżer pakietów, przyjrzymy się Nexus Repository OSS i Azure Artifacts.

Repozytorium Nexusa OSS

Nexus Repository to produkt firmy Sonatype (<https://www.sonatype.com/>), która specjalizuje się w narzędziach *development-security-operations* (**DevSecOps**) integrujących mechanizmy bezpieczeństwa w kodzie aplikacji.

Repozytorium Nexusa istnieje w wersji open source/bezpłatnej, a jego dokumentacja jest dostępna pod adresem <https://www.sonatype.com/nexus-repository-oss?smtNoRedir=1> i <https://help.sonatype.com/repomanager3>.

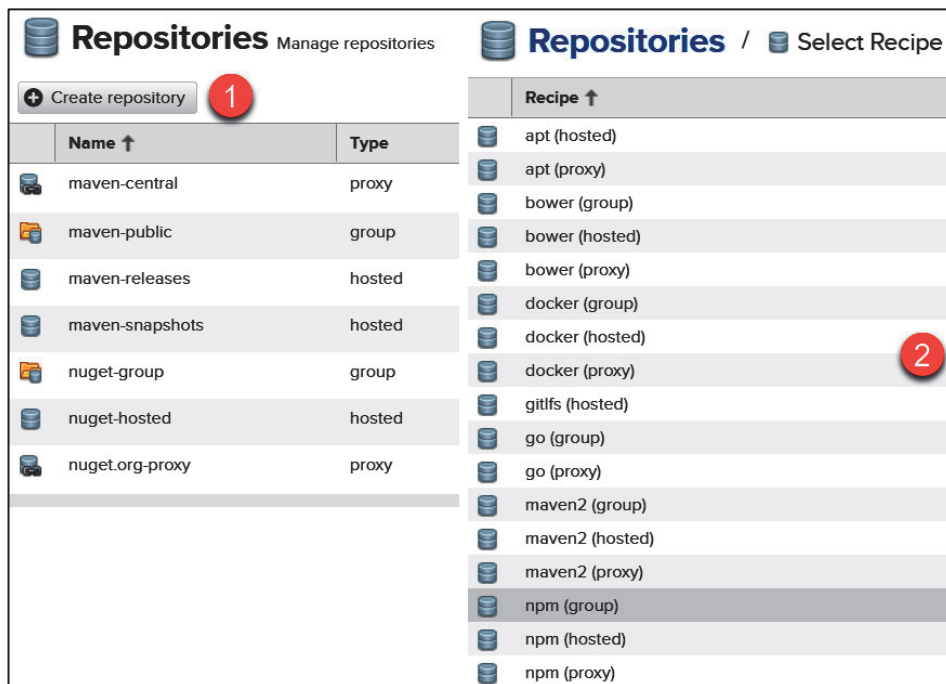
Przed zainstalowaniem i użyciem Nexusa weź pod uwagę wymagania programowe i sprzętowe wyszczególnione w dokumentacji: <https://help.sonatype.com/repomanager3/installation/system-requirements>.

Aby zapoznać się z krokami instalacji i konfiguracji, przyjrzyj się procedurze instalacji, która jest dostępna tutaj: <https://help.sonatype.com/repomanager3/installation>.

Z Nexusa można również korzystać za pośrednictwem kontenera Dockera (Dockerowi przyjrzymy się szczegółowo w następnym rozdziale) — dokumentacja na ten temat: <https://hub.docker.com/r/sonatype/nexus3/>.

Po zainstalowaniu repozytorium Nexusa musimy utworzyć repozytorium, wykonując następujące kroki:

1. W sekcji *Repositories* kliknij przycisk *Create repository*.
2. Następnie wybierz typ pakietów (np.: npm, NuGet lub Bower), które będą przechowywane w repozytorium, jak pokazano na poniższym zrzucie ekranu:



Rysunek 7.3. Menedżer pakietów Nexus

Nexus to wysokowydajne i szeroko stosowane repozytorium korporacyjne, ale jego instalacja i konserwacja wymagają wysiłku. Nie dotyczy to menedżerów pakietów SaaS, takich jak Azure Artifacts, którym przyjrzymy się za chwilę.

Azure Artifacts

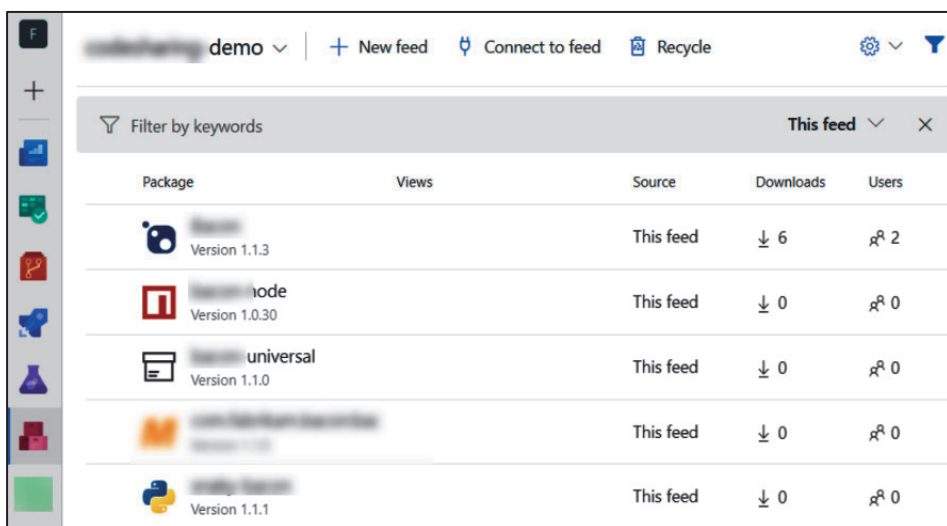
Azure Artifacts to jedna z usług świadczonych przez Azure DevOps. Przyjrzelśmy się jej już w poprzednim rozdziale i omówimy ją również później, w sekcji „Korzystanie z Azure Pipelines dla CI/CD” w tym rozdziale. Jest hostowana w chmurze, dzięki czemu umożliwia zarządzanie pakietami prywatnymi.






Obsługiwane obecnie pakiety to NuGet, npm, Maven, Gradle, Python, a także pakiety uniwersalne. Główna różnica w porównaniu z Nexusem polega na tym, że w Azure Artifacts źródło danych nie jest podzielone według typu pakietu, a jedno źródło może zawierać różne typy pakietów.

Jedną z zalet Azure Artifacts jest to, że jest w pełni zintegrowana z innymi usługami Azure DevOps, takimi jak Azure Pipelines, co pozwala na zarządzanie potokami CI/CD, o czym wkrótce się przekonamy.

W Azure Artifacts istnieje również typ pakietu zwany **pakietem uniwersalnym**, umożliwiający przechowywanie w źródle danych wszystkich typów plików (nazywanych **pakietem**), które mogą być używane przez inne usługi lub użytkowników.

Oto przykład źródła danych Azure Artifacts zawierającego kilka typów pakietów, w którym możemy zobaczyć jeden pakiet NuGet, jeden pakiet npm, jeden pakiet uniwersalny, jeden pakiet Mavena i jeden pakiet Pythona:



| Package | Views | Source | Downloads | Users |
|--|-------|-----------|-----------|-------|
|  demo Version 1.1.3 | | This feed | ↓ 6 | 👤 2 |
|  node Version 1.0.30 | | This feed | ↓ 0 | 👤 0 |
|  universal Version 1.1.0 | | This feed | ↓ 0 | 👤 0 |
|  Version 1.1.1 | | This feed | ↓ 0 | 👤 0 |
|  Version 1.1.1 | | This feed | ↓ 0 | 👤 0 |

Rysunek 7.4. Pakiety Azure Artifacts

Usługa Azure Artifacts ma tę zaletę, że działa w trybie SaaS, więc nie trzeba zarządzać instalacją ani infrastrukturą. Dokumentacja jest dostępna tutaj: <https://azure.microsoft.com/en-us/services/devops/artifacts/>.

Zakończyliśmy omówienie menedżerów pakietów z lokalną instancją serwera NuGet, npm, Nexus i Azure Artifacts. Oczywiście istnieje wiele innych narzędzi do zarządzania pakietami, które należy rozważyć zgodnie z potrzebami firmy.

Po przyjrzeniu się menedżerom pakietów zaimplementujemy teraz potok CI/CD za pomocą dobrze znanego narzędzia o nazwie **Jenkins**.

Używanie Jenkinsa do implementacji CI/CD

Jenkins to jedno z najstarszych narzędzi CI, pierwotnie wydane w 2011 r. Jest otwartoźródłowe i rozwijane w Javie.

Jenkins stał się sławny dzięki dużej społeczności pracującej przy nim i jego wtyczkach. Istnieje ponad 1500 wtyczek Jenkinsa, które pozwalają wykonywać wszystkie rodzaje działań w ramach Twoich zadań. Jeśli mimo to jedno z Twoich zadań nie ma wtyczki, możesz ją utworzyć samodzielnie.

W tej sekcji przyjrzymy się instalacji i konfiguracji Jenkinsa i utworzymy zadanie CI, które zostanie wykonane podczas zatwierdzania kodu znajdującego się w repozytorium Gita.

Kod źródłowy aplikacji demonstracyjnej to projekt Javy, który jest otwartoźródłowy i jest dostępny w przestrzeni repozytorium GitHuba Microsoftu tutaj: <https://github.com/microsoft/MyShuttle2>. Aby móc z niego korzystać, musisz połączyć go z Twoim kontem GitHuba.

Zanim omówimy zadanie Jenkinsa, zobaczmy, jak zainstalować i skonfigurować Jenkinsa.

Instalowanie i konfigurowanie Jenkinsa

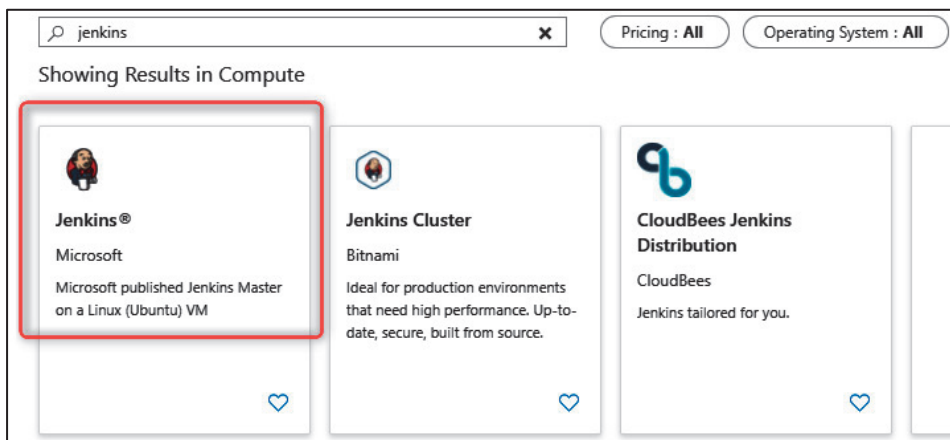
Jenkins to wieloplatformowe narzędzie, które można zainstalować w dowolnym środowisku, takim jak **maszyny wirtualne (VM)** lub nawet kontenery Dockera. Jego dokumentacja instalacji jest dostępna tutaj: <https://jenkins.io/doc/book/installing/>.

Na potrzeby naszej demonstracji i szybkiego dostępu do konfiguracji potoku CI/CD użyjemy Jenkinsa na maszynie wirtualnej platformy Azure. W rzeczywistości witryna Azure Marketplace zawiera maszynę wirtualną z zainstalowanym oprogramowaniem Jenkins i jego wymaganiami wstępnymi.

Poniższe kroki pokazują, jak utworzyć maszynę wirtualną platformy Azure z Jenkinsem, i przedstawiają jej podstawową konfigurację:

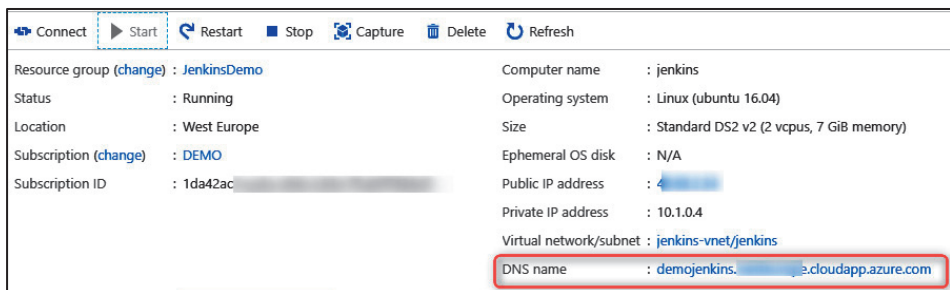
1. Aby uzyskać wszystkie kroki potrzebne do utworzenia maszyny wirtualnej platformy Azure z już zainstalowanym programem Jenkins, zapoznaj się z dokumentacją dostępną tutaj: <https://docs.microsoft.com/en-us/azure/jenkins/install-jenkins-solution-template>.

Poniższy zrzut ekranu przedstawia integrację Jenkinsa w portalu Azure Marketplace:



Rysunek 7.5. Jenkins w Azure Marketplace

2. Po zainstalowaniu i utworzeniu maszyny uzyskamy dostęp do usługi Jenkins w przeglądarce, podając jej adres URL (ang. *Uniform Resource Locator*) w portalu Azure w polu *DNS name*, jak pokazano na poniższym zrzucie ekranu:



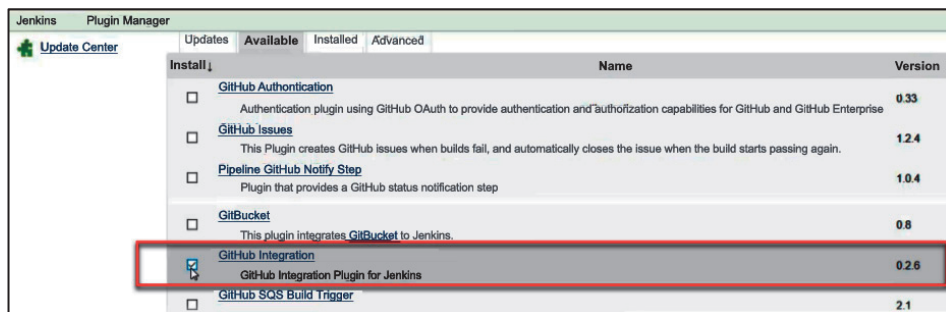
Rysunek 7.6. Nazwa Jenkinsa w Azure Domain Name System (DNS)

3. Postępuj zgodnie z instrukcjami wyświetlanymi na stronie głównej Jenkinsa, aby umożliwić dostęp do instancji Jenkinsa za pośrednictwem bezpiecznego połączenia **Secure Sockets Layer (SSL)**. Aby uzyskać więcej informacji na temat tego kroku, przeczytaj dokumentację pod adresem <https://docs.microsoft.com/en-us/azure/jenkins/install-jenkins-solution-template#connect-to-jenkins> i ten artykuł: <https://jenkins.io/blog/2017/04/20/secure-jenkins-on-azure/>.
4. Następnie postępuj zgodnie z instrukcjami konfiguracji w komunikacie *Unlock Jenkins* wyświetlanym na ekranie Jenkinsa. Po zakończeniu konfiguracji przygotowujemy Jenkinsa do utworzenia zadania CI.

Aby korzystać z funkcji GitHuba w Jenkinsie, zainstalowaliśmy również **wtyczkę integracyjną GitHuba** (ang. *GitHub integration plugin*) z pomocą systemu zarządzania

wtyczkami Jenkinsa — zgodnie z tą dokumentacją: <https://jenkins.io/doc/book/managing/plugins/>.

Poniższy zrzut ekranu przedstawia instalację wtyczki GitHuba:



Rysunek 7.7. Integracja Jenkinsa z GitHubem

Teraz, gdy zainstalowaliśmy wtyczkę GitHuba w Jenkinsie, przyjrzyjmy się, jak skonfigurować GitHuba z webhookiem w celu jego integracji z Jenkinsem.

Konfiguracja webhooka GitHuba

Aby Jenkins mógł uruchomić nowe zadanie, musimy najpierw utworzyć webhook w repozytorium GitHuba. Ten webhook będzie używany do powiadamiania Jenkinsa, gdy tylko pojawi się nowa operacja push w repozytorium.

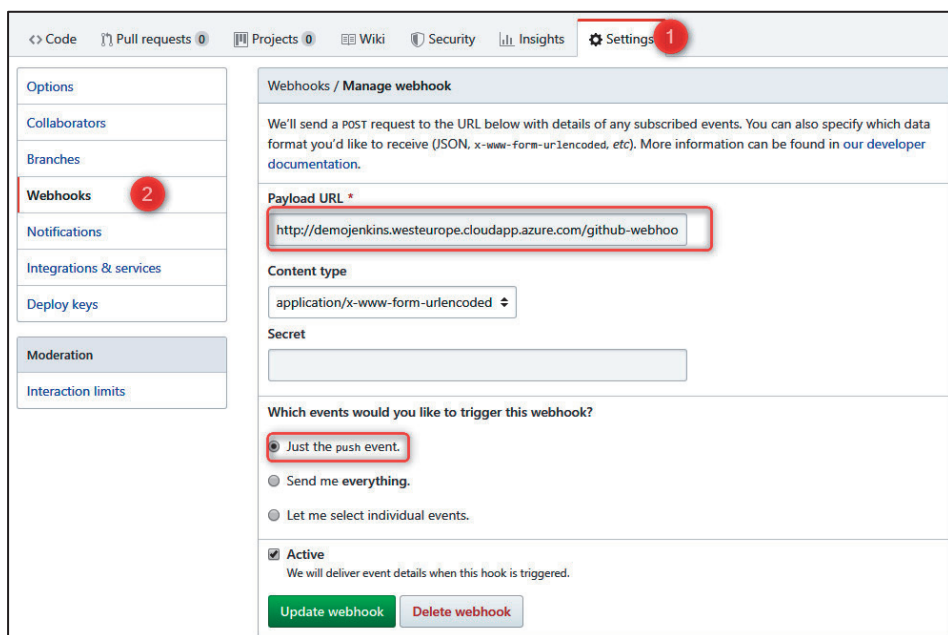
Aby to zrobić, wykonaj następujące kroki:

1. W repozytorium GitHuba przejdź do zakładki *Settings/Webhooks*.
2. Kliknij przycisk *Add Webhook*.
3. W polu *Payload URL* wpisz adres URL Jenkinsa, a następnie dopisz `/github-webhook/`, pozostaw pole z kluczem (*Secret*) bez zmian i wybierz opcję *Just the push event*.
4. Sprawdź poprawność webhooka.

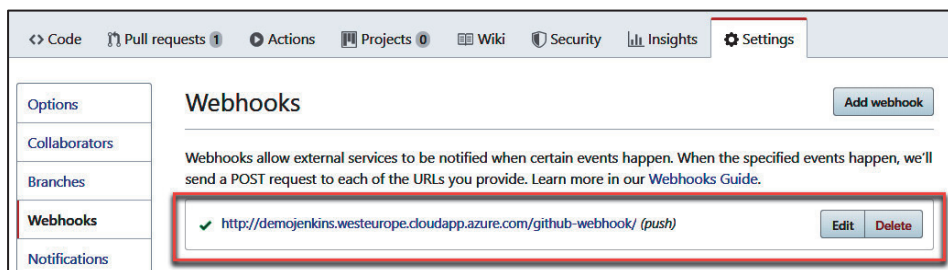
Rysunek 7.8 przedstawia konfigurację webhooka GitHuba dla Jenkinsa.

5. Na końcu możemy sprawdzić w interfejsie GitHuba, czy webhook jest dobrze skonfigurowany i czy komunikuje się z Jenkinsem, jak pokazano na rysunku 7.9.

Konfiguracja GitHuba została zakończona. Przejdziemy teraz do utworzenia nowego zadania CI w Jenkinsie.



Rysunek 7.8. Konfiguracja webhooka GitHuba dla Jenkinsa

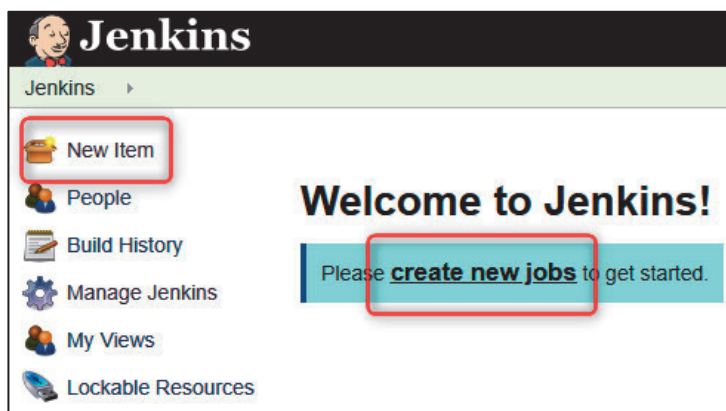


Rysunek 7.9. Weryfikacja webhooka GitHuba

Konfiguracja zadania CI w Jenkinsie

Aby skonfigurować Jenkinsa, wykonaj następujące kroki:

1. Najpierw utworzymy nowe zadanie, klikając *New Item* lub link *create new jobs*, jak pokazano na rysunku 7.10.
2. W formularzu konfiguracji zadania wprowadź nazwę zadania — np. **demoCI**, wybierz szablon *Freestyle project*, a następnie zweryfikuj go, klikając przycisk *OK*, jak pokazano na rysunku 7.11.



Rysunek 7.10. Tworzenie nowego zadania w Jenkinsie

Enter an item name

demoCI 1

» Required field

Freestyle project 2
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM

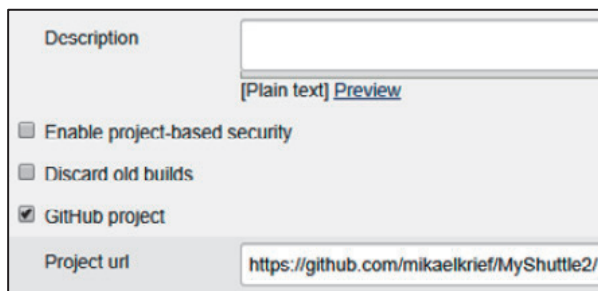
Maven project
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces

OK 3

Rysunek 7.11. Nazwa zadania Jenkinsa

3. Następnie konfigurowujemy zadanie z następującymi parametrami:

- W polu *GitHub project* wpisujemy adres URL repozytorium GitHuba jak na rysunku 7.12.
- W sekcji *Source Code Management* wprowadź adres URL repozytorium GitHuba i gałąź kodu jak na rysunku 7.13.
- W sekcji *Build Triggers* zaznacz *GitHub hook trigger for GITScm polling* jak na rysunku 7.14.



Description

[Plain text] [Preview](#)

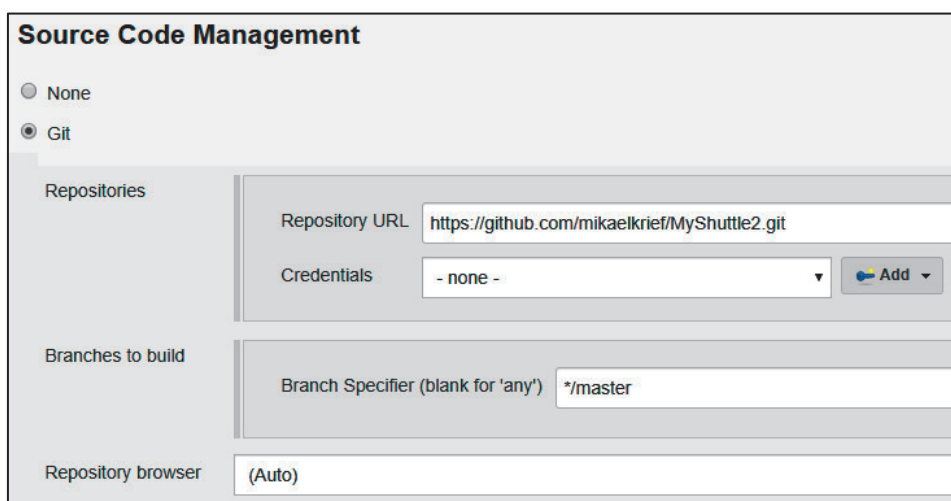
☐ Enable project-based security

☐ Discard old builds

☒ GitHub project

Project url

Rysunek 7.12. Zadanie Jenkinsa — GitHub



Source Code Management

☐ None

☒ Git

Repositories

Repository URL

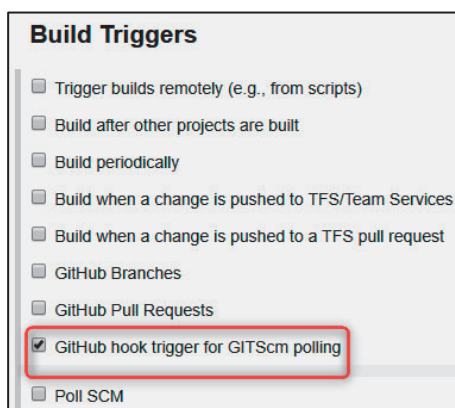
Credentials [Add](#)

Branches to build

Branch Specifier (blank for 'any')

Repository browser

Rysunek 7.13. Zadanie Jenkinsa — konfiguracja GitHuba



Build Triggers

☐ Trigger builds remotely (e.g., from scripts)

☐ Build after other projects are built

☐ Build periodically

☐ Build when a change is pushed to TFS/Team Services

☐ Build when a change is pushed to a TFS pull request

☐ GitHub Branches

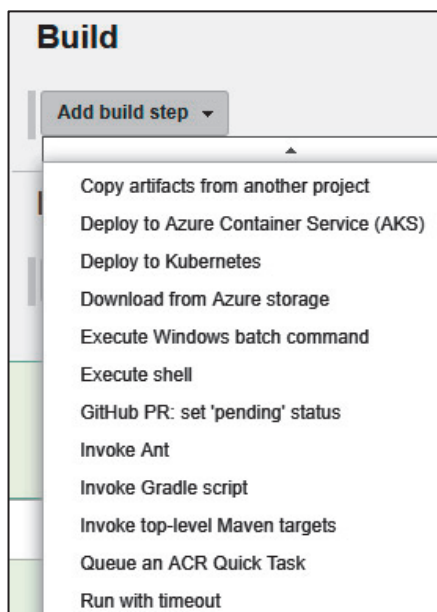
☐ GitHub Pull Requests

☒ GitHub hook trigger for GITScm polling

☐ Poll SCM

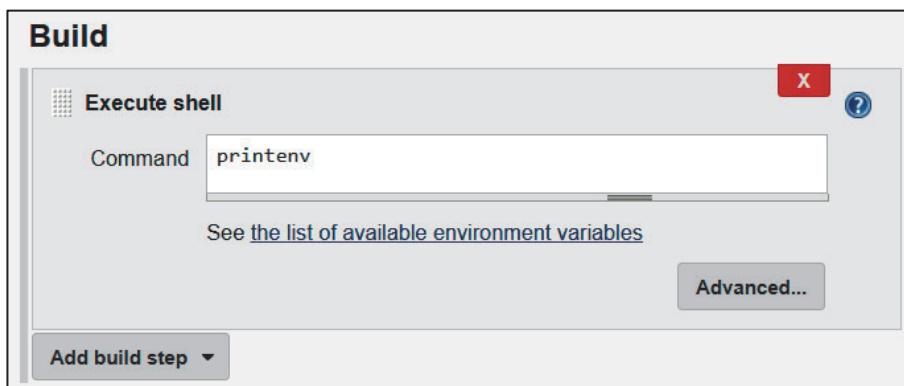
Rysunek 7.14. Zadanie Jenkinsa — konfiguracja GitHuba

- W sekcji *Build*, na liście rozwijanej *Add build step* wybierzemy krok *Execute shell* dla tego laboratorium. Do CI możesz dodać tyle akcji, ile potrzeba (kompilacja, kopiowanie plików i testy). Na poniższym zrzucie ekranu możesz zobaczyć kilka przykładów możliwych działań:



Rysunek 7.15. Zadanie Jenkinsa — dodawanie kroku kompilacji

- Wewnątrz pola tekstowego polecenia powłoki wpisujemy polecenie `printenv`, które ma zostać wykonane podczas wykonywania potoku zadania, jak pokazano na poniższym zrzucie ekranu:



Rysunek 7.16. Przykład uruchamiania zadania Jenkinsa

4. Teraz kończymy konfigurację, klikając *Apply*, a następnie przycisk *Save*.

Nasze zadanie CI Jenkinsa zostało utworzone i jest skonfigurowane tak, aby było uruchamiane podczas zatwierdzania i wykonywania różnych działań.

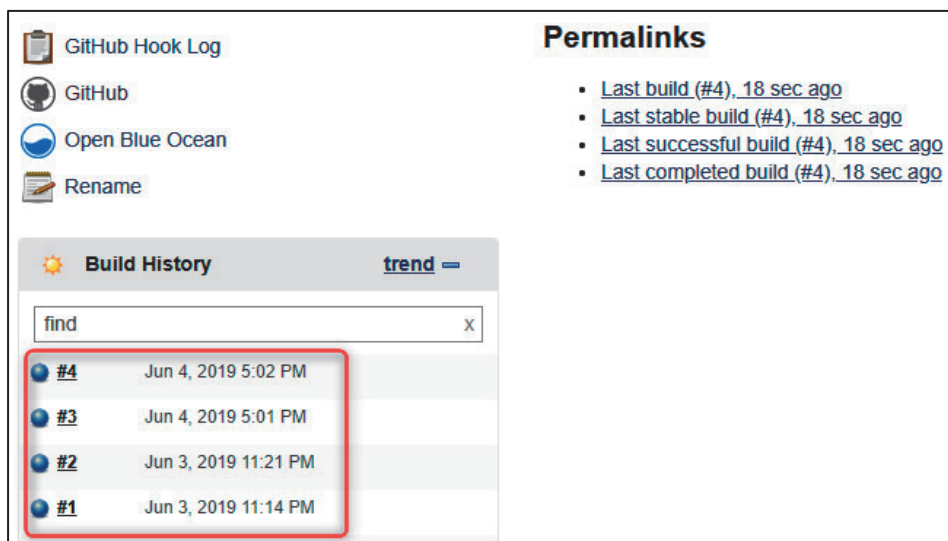
Uruchomimy je teraz ręcznie, aby przetestować jego prawidłowe działanie.

Wykonywanie zadania Jenkinsa

Aby przetestować wykonanie zadania, wykonamy następujące kroki:

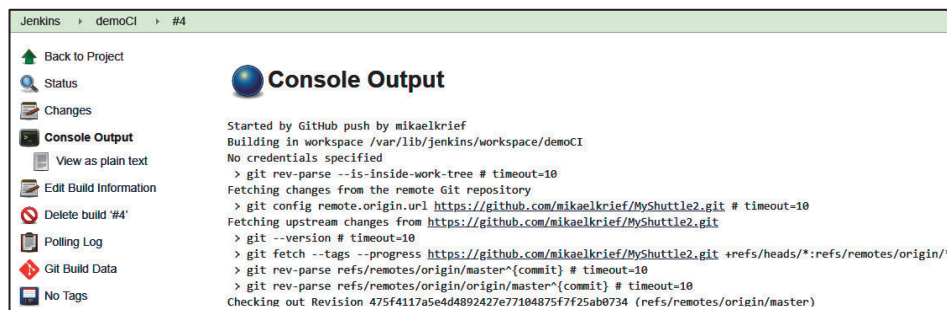
1. Najpierw zmodyfikujemy kod naszego repozytorium GitHuba — np. modyfikując plik *Readme.md*.
2. Następnie zatwierdzimy zmianę do gałęzi *master* — bezpośrednio z interfejsu webowego GitHuba.
3. To, co widzimy w Jenkinsie zaraz po wykonaniu tego zatwierdzenia, to to, że zadanie *DemoCI* jest w kolejce i działa.

Poniższy zrzut ekranu przedstawia kolejkę wykonywania zadań:



Rysunek 7.17. Historia wykonywania zadań Jenkinsa

4. Klikając zadanie, a następnie link w menu *Console Output*, możemy zobaczyć dzienniki wykonania zadania, jak pokazano na poniższym zrzucie ekranu:



Rysunek 7.18. Dane wyjściowe konsoli zadań Jenkinsa

Właśnie utworzyliśmy zadanie CI w Jenkinsie, które działa podczas zatwierdzania repozytorium Gita (w naszym przykładzie GitHuba).

W tej sekcji przyjrzelśmy się tworzeniu potoku w Jenkinsie. Zobaczymy teraz, jak utworzyć potok CI/CD za pomocą innego narzędzia DevOps: Azure Pipelines.

Korzystanie z Azure Pipelines dla CI/CD

Azure Pipelines to jedna z usług oferowanych przez Azure DevOps. Wcześniej była znana jako **Visual Studio Team Services (VSTS)**.

Azure DevOps to kompletna platforma DevOps dostarczana przez firmę Microsoft, w pełni dostępna za pośrednictwem przeglądarki internetowej i nie wymagająca instalacji. Jest to bardzo przydatne z następujących powodów:

- Narzędzia DevOps zarządzają swoim kodem za pośrednictwem **systemu kontroli wersji (VCS)**.
- Azure DevOps zarządza projektem w trybie agile.
- Wdraża aplikacje w potoku CI/CD w celu scentralizowania pakietów.
- Wykonuje ręczne plany testów.

Każda z tych funkcji jest połączona w usługi, które podsumowano w tej tabeli 7.1.

Usługa **Azure DevOps** jest bezpłatna dla maksymalnie pięciu użytkowników. Poza tym istnieje wersja licencyjna płatna za każdego użytkownika. Aby uzyskać więcej informacji na temat licencjonowania, zapoznaj się z opisem produktu pod adresem <https://azure.microsoft.com/en-us/pricing/details/devops/azure-devops-services/>, który zawiera również kalkulator umożliwiający oszacowanie kosztów.

Tabela 7.1. Lista usług Azure DevOps

| Nazwa usługi | Opis | Dokumentacja |
|------------------|--|---|
| Azure Repos | VCS, który omówiliśmy w poprzednim rozdziale | https://azure.microsoft.com/en-us/services/devops/repos/ |
| Azure Boards | Usługa na potrzeby zarządzania projektami w trybie agile ze sprintami, z backlogami i boardami | https://azure.microsoft.com/en-us/services/devops/boards/ |
| Azure Pipelines | Usługa, która pozwala na zarządzanie potokami CI/CD | https://azure.microsoft.com/en-us/services/devops/pipelines/ |
| Azure Artifacts | Prywatny menedżer pakietów | https://azure.microsoft.com/en-us/services/devops/artifacts/ |
| Azure Test Plans | Pozwala na tworzenie ręcznych planów testów i zarządzanie nimi | https://azure.microsoft.com/en-us/services/devops/test-plans/ |

Uwaga

Istnieje również Azure DevOps Server, który jest tym samym produktem co Azure DevOps, ale instaluje się lokalnie. Aby poznać różnice między tymi dwoma produktami, przeczytaj dokumentację tutaj: <https://docs.microsoft.com/en-us/azure/devops/user-guide/about-azure-devops-services-tfs?view=azure-devops>.

Aby zarejestrować się w usłudze Azure DevOps i utworzyć konto zwane organizacją, potrzebujemy konta Microsoft Live lub konta GitHuba. Następnie wykonaj następujące kroki:

1. W przeglądarce przejdź do tego adresu URL: <https://azure.microsoft.com/en-us/services/devops/>.
2. Kliknij przycisk *Sign In*.
3. Na następnej stronie wybierz konto, którego chcesz użyć (*Live* lub *GitHub*).
4. Jak tylko się zarejestrujemy, pierwszym sugerowanym krokiem jest utworzenie organizacji z wybraną unikatową nazwą i lokalizacją platformy Azure — przyjmij np. *BookLabs* jako nazwę organizacji i *West Europe* jako lokalizację.
5. W tej organizacji będziemy teraz mogli tworzyć projekty za pomocą naszego potoku CI/CD, jak dowiedzieliśmy się w rozdziale 6., „Zarządzanie kodem źródłowym za pomocą Gita”.

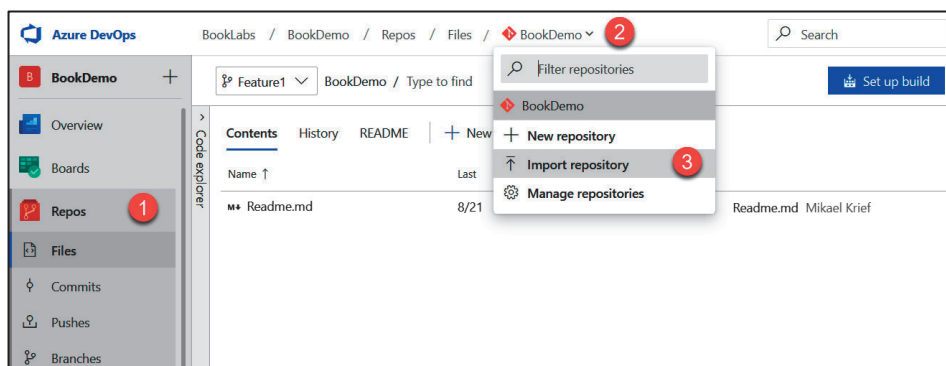
W tym laboratorium pokażemy, jak skonfigurować testy **E2E** (ang. *end-to-end*) w CI i potok CD, zaczynając od użycia Azure Repos do wersji naszego kodu. Następnie w Azure

Pipelines przyjrzymy się procesowi CI i zakończymy automatycznym wdrożeniem aplikacji w kolejnej wersji.

Wersjonowanie kodu za pomocą Gita w Azure Repos

Jak wspomnieliśmy, pierwszym warunkiem wstępnym skonfigurowania procesu CI jest posiadanie wersjonowania kodu aplikacji w SVC. Zrobimy to w Azure Repos, wykonując następujące kroki:

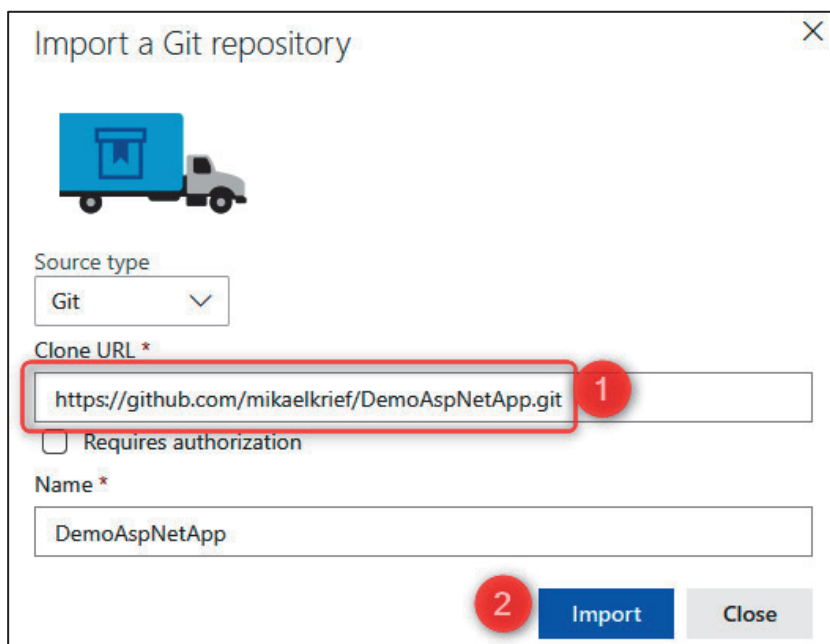
1. Aby uruchomić nasze laboratorium, utworzymy nowy projekt; ta operacja została już omówiona w rozdziale 6., „Zarządzanie kodem źródłowym za pomocą Gita”, w sekcji „Zaczynamy od procesu Gita”.
2. Następnie w Azure Repos zaimportujemy kod z innego repozytorium Gita za pomocą opcji *Import repository* w menu repozytorium, jak pokazano na poniższym zrzucie ekranu:



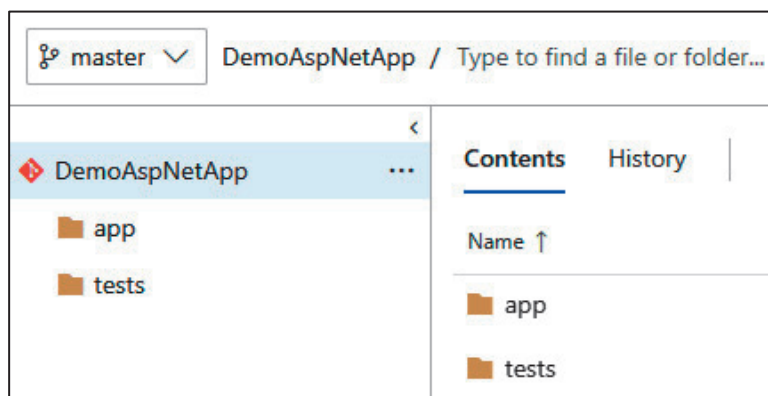
Rysunek 7.19. Menu Import repository w Azure Repos

3. Po otwarciu okna *Import a Git repository* wprowadzamy adres URL repozytorium Gita, którego źródła chcemy zaimportować. W naszym laboratorium zaimportujemy źródła znalezione w repozytorium <https://github.com/mikaelkrief/DemoAspNetApp.git>, jak widać na rysunku 7.20.
4. Klikamy przycisk *Import*. Kod został zaimportowany do naszego repozytorium. Rysunek 7.21 przedstawia kod zaimportowany do naszego repozytorium Azure Repos. Zawiera on kod aplikacji ASP.NET i jego testy jednostkowe.

Teraz, gdy mamy kod w Azure Repos, skonfigurujemy potok CI, który będzie sprawdzał i testował kod przy każdym zatwierdzeniu użytkownika.



Rysunek 7.20. Import repozytorium w Azure Repos

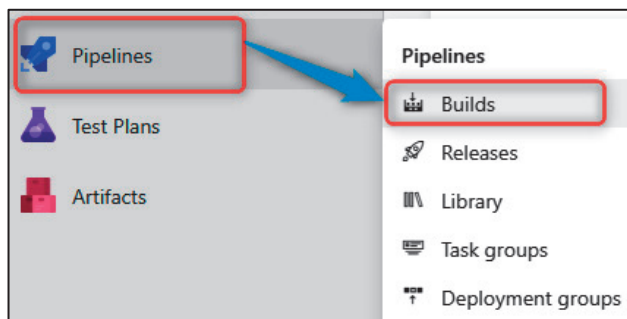


Rysunek 7.21. Zakończono importowanie repozytorium Azure Repos

Tworzenie potoku CI

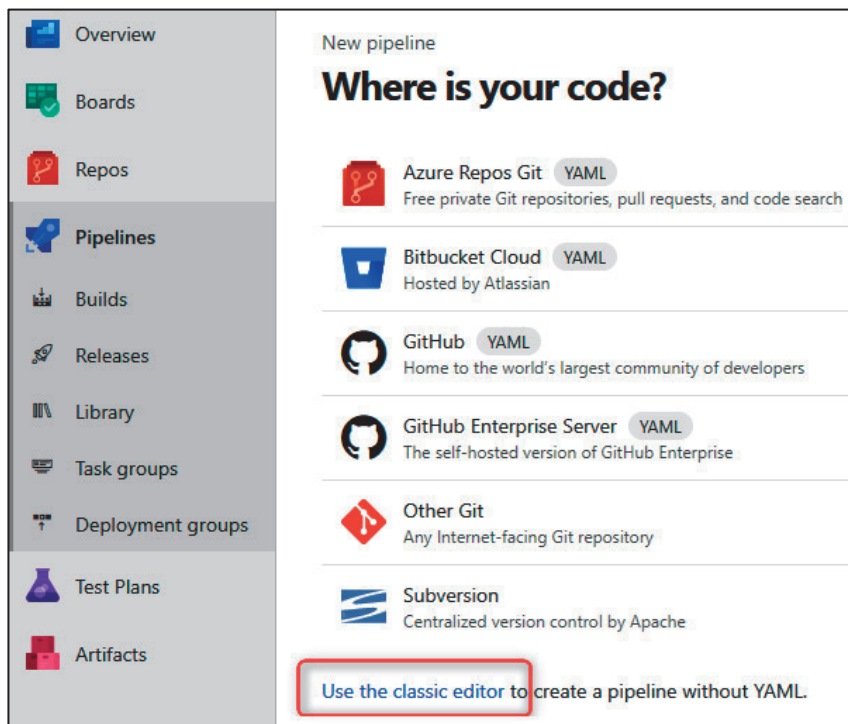
Utworzymy potok CI w Azure Pipelines, wykonując następujące kroki:

1. Aby utworzyć ten potok, otwórz menu *Pipelines/Builds*, jak pokazano na poniższym zrzucie ekranu. Następnie kliknij przycisk *New pipeline*:



Rysunek 7.22. Azure Pipelines — tworzenie kompilacji

2. W trybie konfiguracji wybieramy opcję *Use the classic editor*, jak pokazano na poniższym zrzucie ekranu:



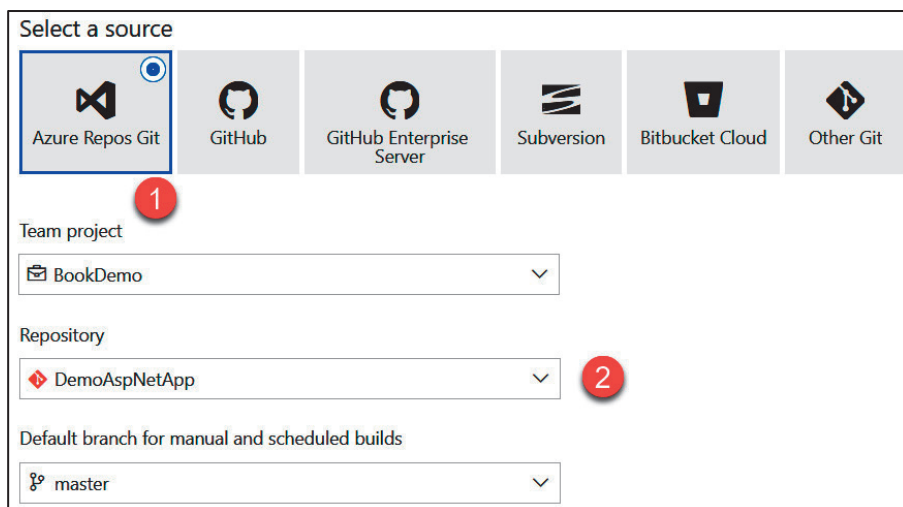
Rysunek 7.23. Link do klasycznego edytora Azure Pipelines

W Azure Pipelines mamy do wyboru klasyczny tryb edytora, który pozwala nam skonfigurować potok za pomocą interfejsu graficznego, lub tryb potoku YAML, który obejmuje użycie pliku YAML (ang. *YAML Ain't Markup Language*) opisującego konfigurację potoku.

W tym laboratorium użyjemy klasycznego trybu edytora, aby zwizualizować różne opcje i kroki konfiguracji.

3. Pierwszy krok konfiguracji potoku polega na wybraniu repozytorium zawierającego źródła aplikacji.

Obecnie Azure Pipelines obsługuje kilka typów systemów Git, takich jak **Azure Repos**, GitHub, Bitbucket i **Subversion (SVN)**. My wybierzemy Azure Repos Git, który zawiera zaimportowane źródła kodu — jak pokazano na poniższym zrzucie ekranu:



Select a source

Azure Repos Git GitHub GitHub Enterprise Server Subversion Bitbucket Cloud Other Git

1

Team project

BookDemo

Repository

DemoAspNetApp

2

Default branch for manual and scheduled builds

master

Rysunek 7.24. Azure Pipelines — wybór repozytorium kodu źródłowego

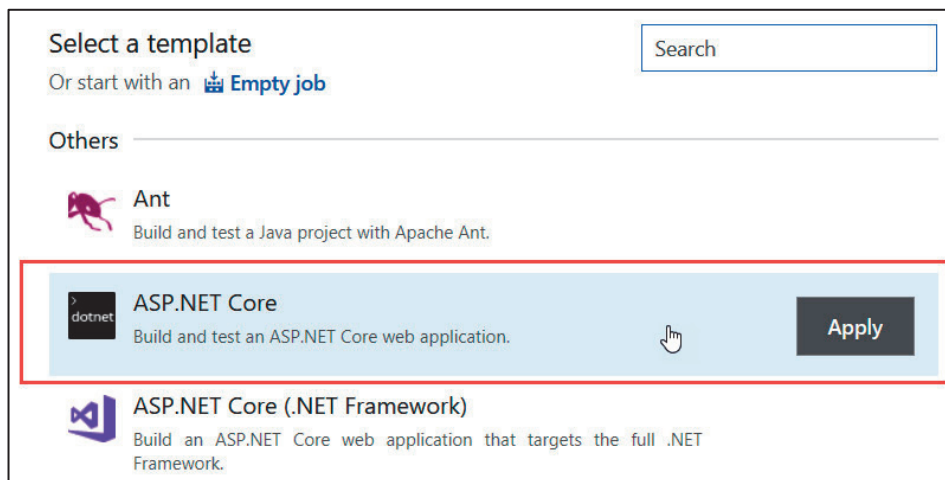
4. Azure Pipelines proponuje wybranie szablonu kompilacji, który będzie zawierał wszystkie wstępnie skonfigurowane kroki kompilacji; istnieje również możliwość rozpoczęcia od pustego szablonu.

Ponieważ nasz projekt jest aplikacją ASP.NET, wybierzemy szablon *ASP.NET Core*, jak pokazano na poniższym zrzucie ekranu.

Po wybraniu szablonu dochodzimy do strony konfiguracyjnej definicji kompilacji.

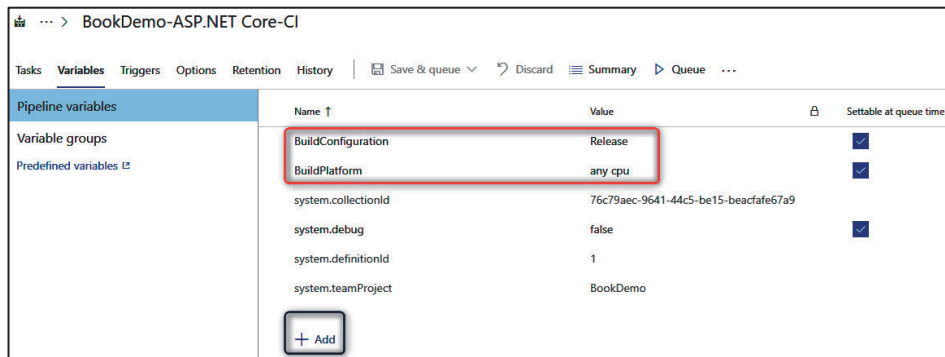
Konfiguracja definicji kompilacji składa się z czterech głównych sekcji, przedstawionych w następujący sposób:

- *Variables*,
- *Steps*,
- *Triggers*,
- *Options*.



Rysunek 7.25. Azure Pipelines — wybór szablonu

5. Konfigurujemy sekcję *Variables*, która pozwala nam wypełnić listę zmiennych w postaci klucza, tworząc wartość, którą można wykorzystać w krokach. Poniższy ekran pokazuje zakładkę *Variables* naszej definicji kompilacji:



Rysunek 7.26. Azure Pipelines — zakładka Variables

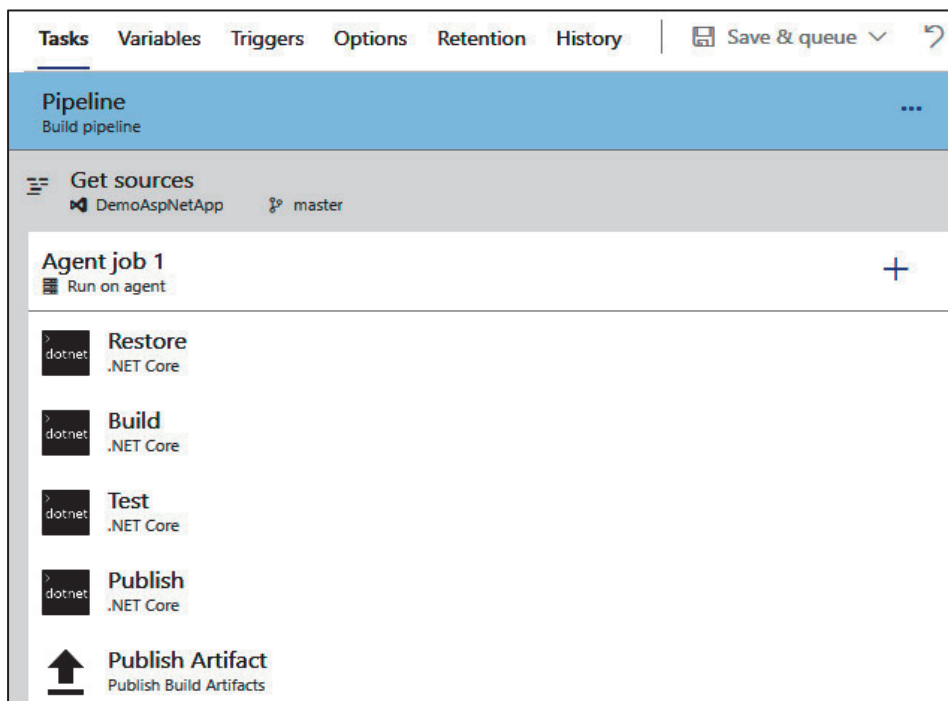
Po przejściu do zakładki *Variables* widzimy zmienne `BuildConfiguration` i `BuildPlatform`, które są już wstępnie wypełnione przez szablon, oraz przycisk `+ Add`, który pozwala nam dodać inne zmienne, jeśli chcemy.

Uwaga

Dokumentacja dotycząca zmiennych jest dostępna tutaj: <https://docs.microsoft.com/en-us/azure/devops/pipelines/process/variables?view=azure-devops&tabs=classic%2Cbatch>.

6. Konfigurujemy zakładkę *Tasks*, która zawiera konfigurację wszystkich kroków do wykonania w kompilacji.

Oto zrzut ekranu tej karty:



Rysunek 7.27. Azure Pipelines — lista zadań

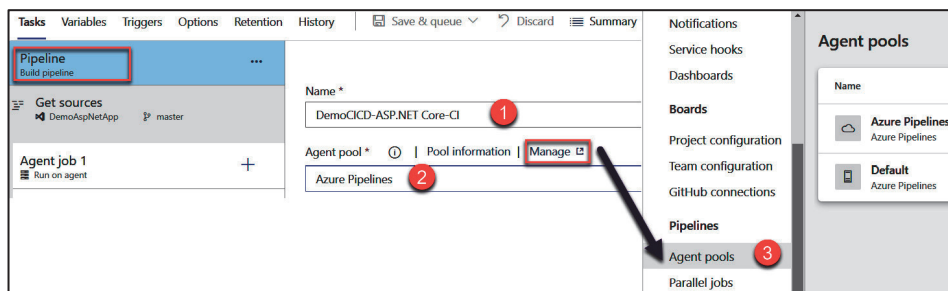
Na powyższym zrzucie pierwsza część to *Pipeline*, która pozwala nam skonfigurować nazwę definicji kompilacji oraz agenta, którego będziemy używać. W Azure DevOps potoki są wykonywane na agentach zainstalowanych na maszynach wirtualnych lub kontenerach.

Usługa Azure DevOps oferuje bezpłatne agenty dla wielu **systemów operacyjnych** (ang. *operating systems* — **OS**), nazywane agentami hostowanymi, ale możliwe jest również zainstalowanie własnych agentów, nazywanych **agentami hostowanymi samodzielnie** (ang. *self-hosted*).

Uwaga

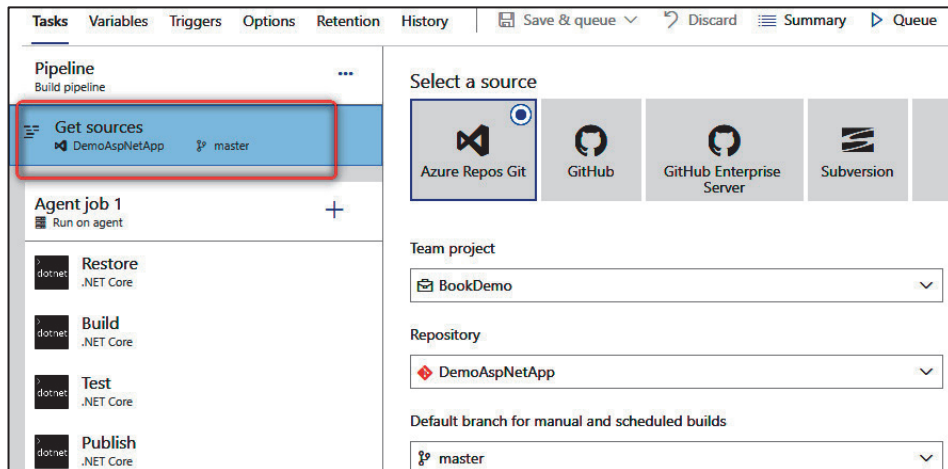
Aby dowiedzieć się więcej o agentach hostowanych i hostowanych samodzielnie, zapoznaj się z następującą dokumentacją: <https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/agents?view=azure-devops>.

Poniższy zrzut ekranu przedstawia konfigurację sekcji *Pipeline*:



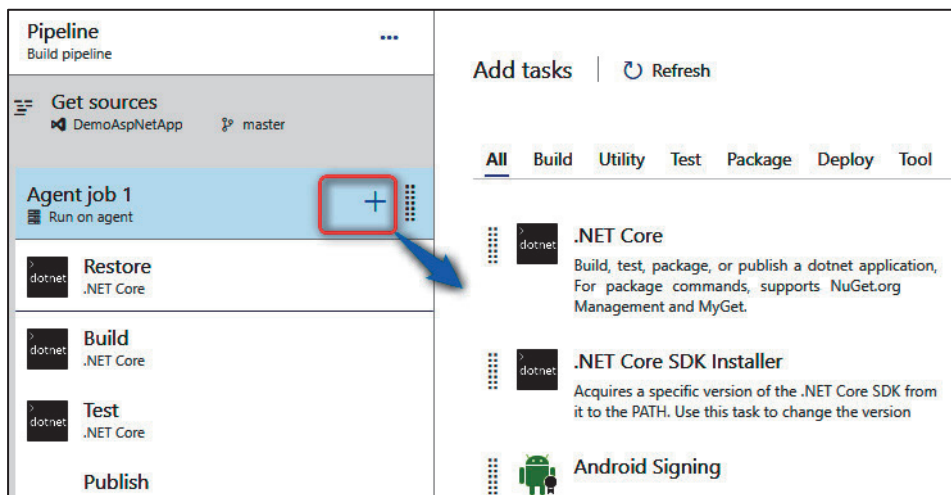
Rysunek 7.28. Azure Pipelines — konfiguracja puli agentów

7. Następnie mamy fazę *Get sources*, zawierającą konfigurację kodu źródłowego, którą zrobiliśmy na początku; można ją jednak tutaj zmodyfikować, co ilustruje poniższy zrzut ekranu:



Rysunek 7.29. Azure Pipelines — konfiguracja kodu źródłowego

8. Mamy część *Agent job*, która zawiera uporządkowaną listę zadań do wykonania w potoku. Każde z tych zadań jest konfigurowane w panelu po prawej stronie. Możemy dodać zadania, klikając przycisk **+**, a następnie wybrać je z katalogu Azure Pipelines jak na rysunku 7.30.
Domyślnie Azure Pipelines zawiera bardzo bogaty katalog zadań; ich lista jest dostępna tutaj: <https://docs.microsoft.com/en-us/azure/devops/pipelines/tasks/index?view=azure-devops>.



Rysunek 7.30. Azure Pipelines — dodawanie zadania

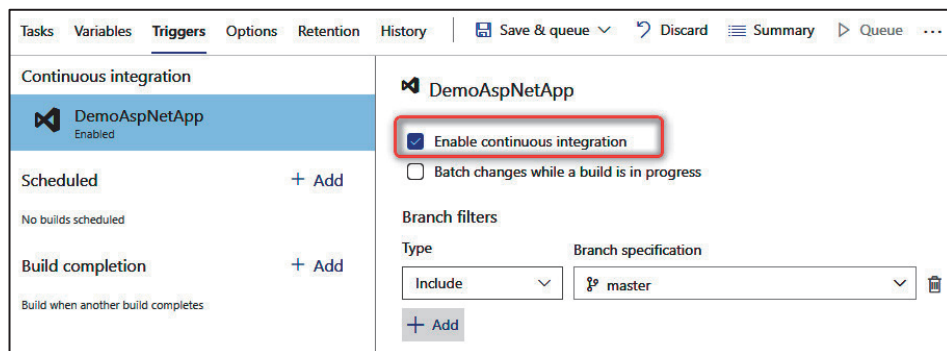
Możemy również zainstalować zadania, które można znaleźć w portalu Azure Marketplace: <https://marketplace.visualstudio.com/search?target=AzureDevOps&category=Azure%20Pipelines&sortBy=Downloads>. W razie potrzeby możesz też tworzyć zadania dla swoich potrzeb, postępując zgodnie z dokumentacją tutaj: <https://docs.microsoft.com/en-us/azure/devops/extend/get-started/node?view=azure-devops>.

Zdefiniujemy pięć zadań w potoku CI w następujący sposób:

Tabela 7.2. Lista kroków Azure Pipelines

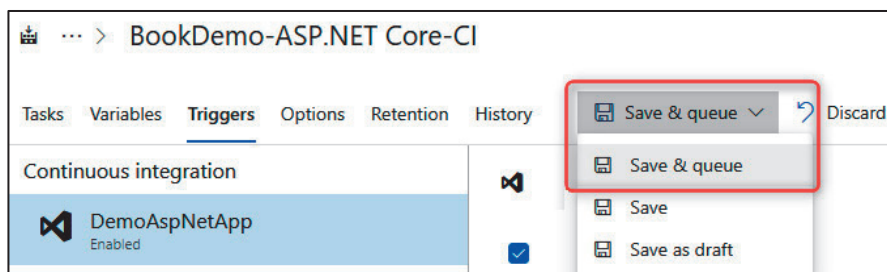
| Krok/zadanie | Opis |
|--------------------------------|--|
| <i>Restore</i> | Odzyskuje pakiety powiązane z projektem |
| <i>Build</i> | Kompiluje projekt i generuje pliki binarne |
| <i>Test</i> | Uruchamia testy jednostkowe |
| <i>Publish</i> | Tworzy pakiet ZIP, który zawiera pliki binarne projektu |
| <i>Publish build artifacts</i> | Definiuje artefakt, który jest naszym pakietem ZIP. Będzie publikowany w Azure DevOps i używany w fazie wdrażania — tak jak widzieliśmy w poprzedniej sekcji podczas korzystania z menedżera pakietów w procesie CI/CD |

- Ostatnią ważną konfiguracją naszego potoku CI jest konfiguracja wyzwalacza kompilacji na karcie *Triggers*, aby włączyć CI — jak pokazano na poniższym zrzucie ekranu:



Rysunek 7.31. Azure Pipelines — włączanie CI

10. Konfiguracja naszego CI lub potoku kompilacji jest zakończona; sprawdzamy i testujemy jego wykonanie po raz pierwszy, klikając przycisk *Save & queue*, jak pokazano na poniższym zrzucie ekranu:



Rysunek 7.32. Azure Pipelines — zapisywanie i kolejowanie potoku

11. Pod koniec wykonywania kompilacji dostajemy kilka informacji, które pomagają nam przeanalizować stan potoku:
- Rysunek 7.33 przedstawia dzienniki wykonania. Wyświetla szczegóły wykonania każdego zadania zdefiniowanego w potoku.
 - Następnie przedstawiono wyniki wykonania testów jednostkowych. Rysunek 7.34 przedstawia raport wykonania testów jednostkowych z kilkoma ważnymi metrykami, takimi jak liczba zaliczonych/niezaliczonych testów i czas wykonania testu.

Teraz, gdy wiemy, że nasza kompilacja CI jest skonfigurowana i działa, utworzymy i skonfigurujemy wersję wdrożeniową dla potoku CD.

✓ #20190606.2: add solution

Manually run today at 13:39 by Mikael Krief DemoAspNetApp master 6dd6855

Logs

Summary

Tests

Agent job 1

Pool: Hosted Ubuntu 1604 · Agent: Hosted Agent

✓ Prepare job · succeeded

✓ Initialize job · succeeded

✓ Checkout · succeeded

✓ Restore · succeeded

✓ Build · succeeded

✓ Test · succeeded

✓ Publish · succeeded

✓ Publish Artifact · succeeded

✓ Post-job: Checkout · succeeded

✓ Finalize Job · succeeded

✓ Report build status · succeeded

Rysunek 7.33. Wynik uruchomienia Azure Pipelines

✓ #20190606.2: add solution

Manually run today at 13:39 by Mikael Krief DemoAspNetApp master 6dd6855

Logs

Summary

Tests

Summary

1 Run(s) Completed (1 Passed, 0 Failed)

1

Total tests

1 ● Passed

0 ● Failed

0 ● Others

100%

Pass percentage

6s 974ms

Run duration ⓘ

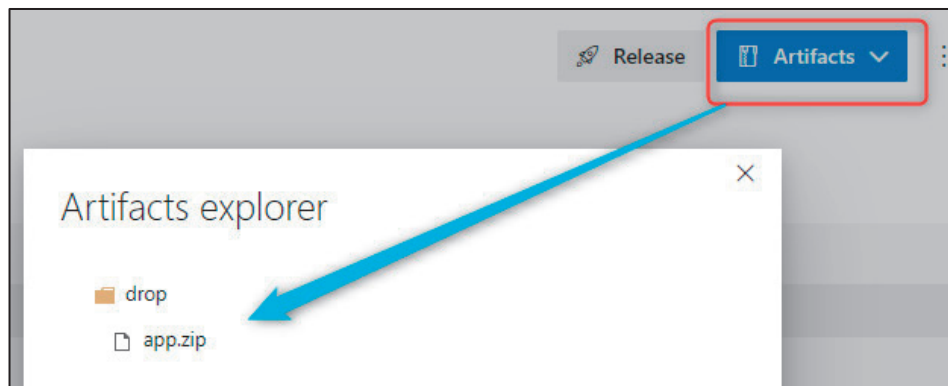
↑ +2s 787ms

Bug ▾

Link

Rysunek 7.34. Azure Pipelines — podsumowanie wykonania testu

- Poniższy zrzut ekranu przedstawia opublikowane artefakty. Daje to możliwość eksplorowania lub pobierania opublikowanych artefaktów zdefiniowanych w zadaniu potoku *Publish Build Artifact*.

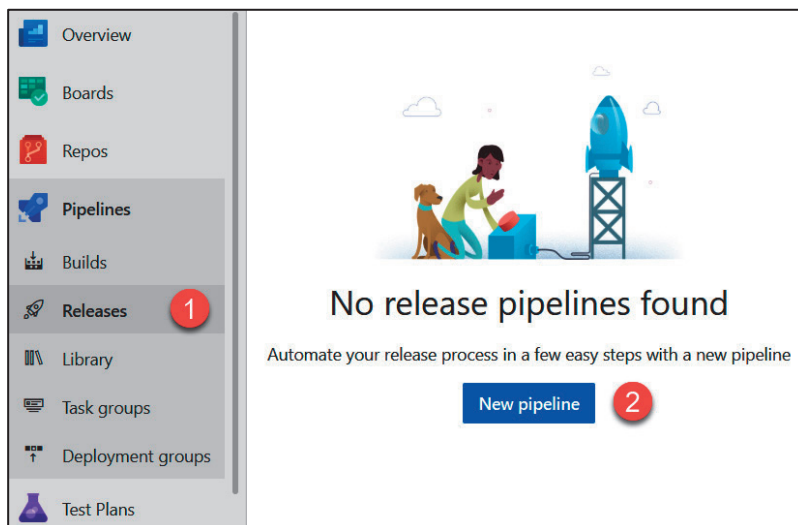


Rysunek 7.35. Azure Pipelines — przeglądanie artefaktów

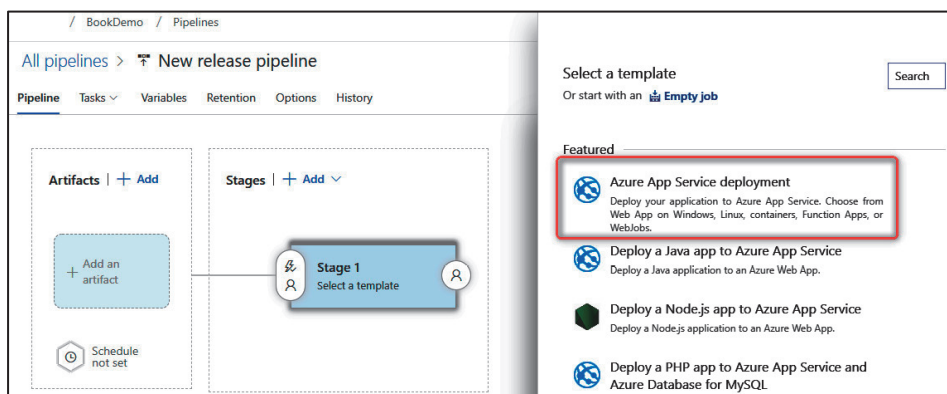
Tworzenie potoku CD — nowa wersja aplikacji

W Azure Pipelines element, który umożliwia wdrażanie na różnych etapach lub w różnych środowiskach, nazywa się **wersją aplikacji** (ang. *release*). Teraz utworzymy definicję wersji, która wdroży nasze wygenerowane przez kompilację artefakty w aplikacji internetowej platformy Azure.

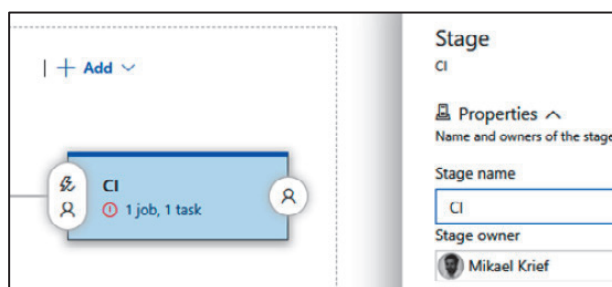
1. Aby utworzyć definicję wersji, przechodzimy do menu *Releases* i klikamy *New pipeline* jak na rysunku 7.36.
2. Jeśli chodzi o kompilację, pierwszym krokiem konfiguracji jest wybranie szablonu już skonfigurowanego. Na potrzeby tego laboratorium wybierzemy szablon *Azure App Service deployment*, jak pokazano na rysunku 7.37.
3. W następnym oknie pierwszy etap zostaje nazwany — np. *CI*, czyli środowisko CI, jak pokazano na rysunku 7.38.
4. Konfigurujemy punkt wejścia nowej wersji w części artefaktów, dodając artefakt, który jest definicją kompilacji utworzoną wcześniej w sekcji „Tworzenie potoku CI” — rysunek 7.39.
5. Konfigurujemy automatyczny wyzwalacz nowej wersji dla każdego pomyślnego wykonania kompilacji — rysunek 7.40.



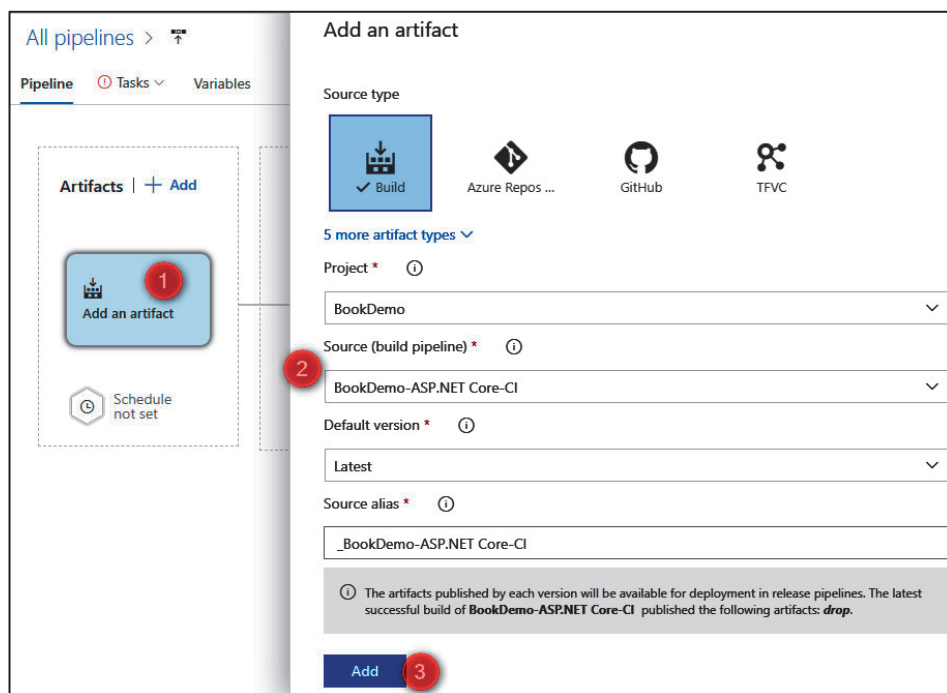
Rysunek 7.36. Azure Pipelines — tworzenie definicji nowej wersji aplikacji



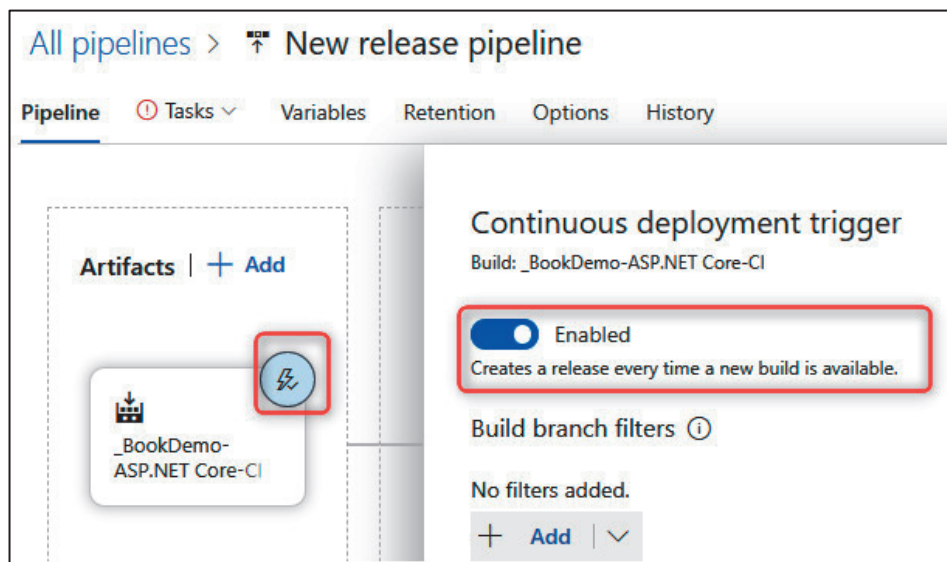
Rysunek 7.37. Azure Pipelines — szablon Azure App Service deployment



Rysunek 7.38. Azure Pipelines — nazwa etapu



Rysunek 7.39. Azure Pipelines — dodawanie artefaktu nowej wersji aplikacji

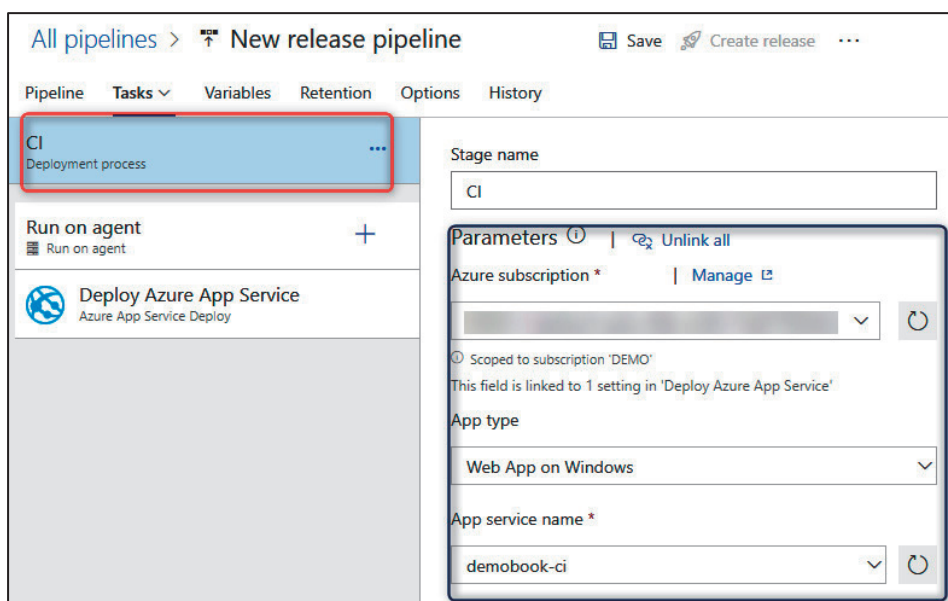


Rysunek 7.40. Azure Pipelines — wersja włączająca proces CD

6. Teraz konfigurujemy kroki, które zostaną wykonane na etapie CI; klikając etap, otrzymujemy dokładnie to samo okno konfiguracyjne co dla kompilacji, z następującymi informacjami:

- Wybór agenta w sekcji *Run on agent*.
- Konfiguracja kroków wraz z ich parametrami.

W naszym przypadku widzimy zadanie wdrożenia w aplikacji internetowej platformy Azure, które było już obecne w szablonie. Najpierw wypełniamy parametry, które znajdują się w nagłówku CI, ponieważ są one wspólne dla procesu CI, jak pokazano na poniższym zrzucie ekranu:



Rysunek 7.41. Azure Pipelines — konfiguracja wersji

Wypełniamy następujące parametry:

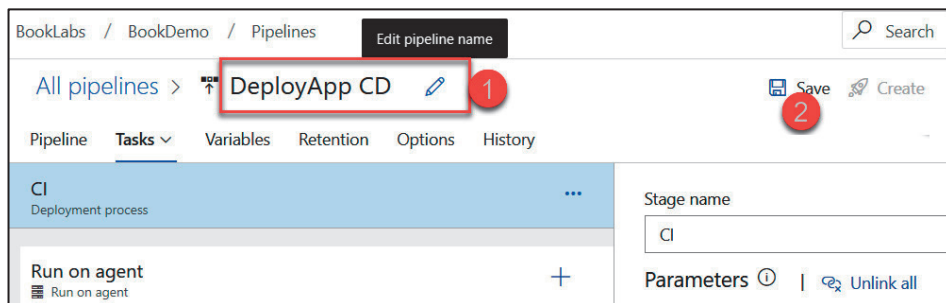
- Połączenie z subskrypcją platformy Azure.
- Nazwa aplikacji internetowej platformy Azure, w której chcemy wdrożyć nasz pakiet ZIP.

Wtedy w parametrach zadania wdrożenia Azure nie mamy nic do modyfikacji.

7. Musimy tylko zmienić nazwę wersji na nazwę, która po prostu opisuje, co robi, a następnie zapisać ją jak na rysunku 7.42.

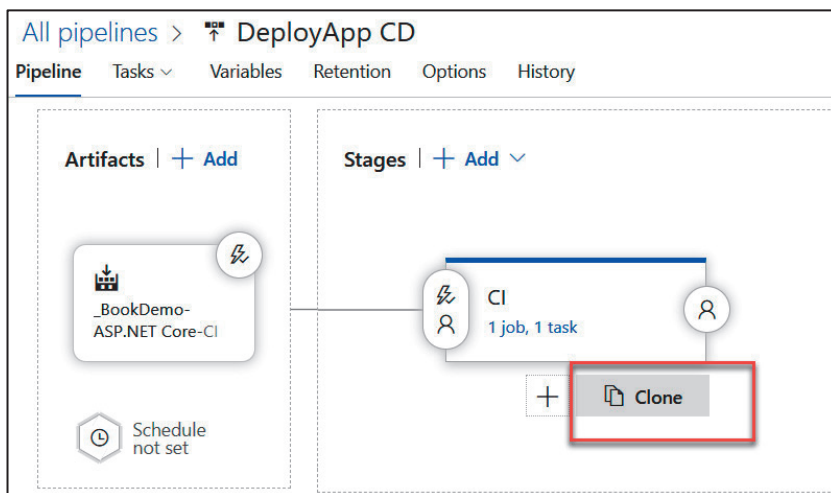
Uwaga

Aplikacja internetowa musi być już utworzona przed wdrożeniem w niej aplikacji. Jeśli nie zostanie utworzona, możesz użyć **polecenia platformy Azure** (ang. *command-line interface* — **CLI**) az **webapp create** udokumentowanego na stronie <https://docs.microsoft.com/en-us/cli/azure/webapp?view=azure-cli-latest#az-webapp-create> lub polecenia PowerShell **New-AzureRmWebApp** udokumentowanego pod adresem <https://docs.microsoft.com/en-us/powershell/module/azurerm.websites/new-azurermwebapp?view=azurerm-ps-6.13.0>.



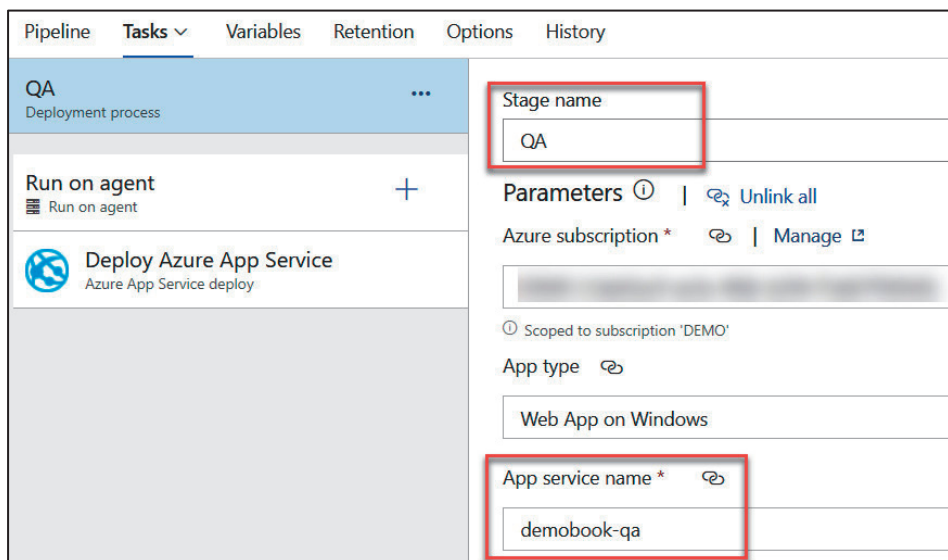
Rysunek 7.42. Azure Pipelines — edycja nazwy wersji

8. Teraz kończymy naszą definicję wersji wraz z wdrożeniem innych środowisk (lub etapów), którymi są np. *QA* i *PROD*. Aby uprościć pracę, sklonujemy ustawienia środowiska CI w naszej wersji i zmienimy nazwę ustawień *App service name* na nazwę aplikacji webowej. Rysunek 7.43 przedstawia działanie środowiska klonowania.



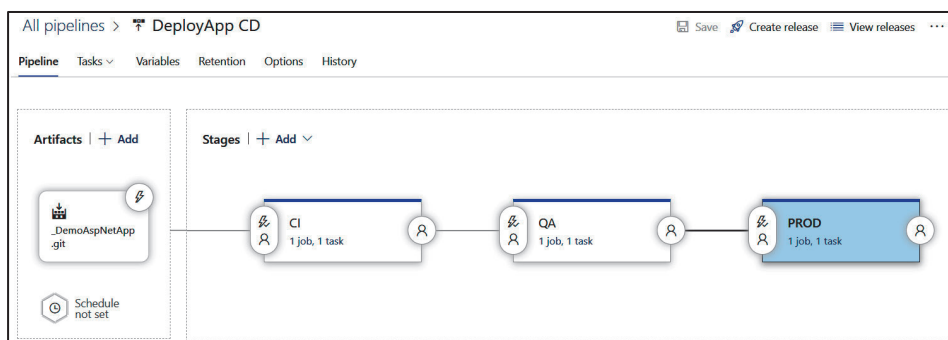
Rysunek 7.43. Etap klonowania wydania platformy Azure

Poniższy zrzut ekranu przedstawia ustawienia *App service name* w Azure z nazwą aplikacji webowej nowego środowiska:



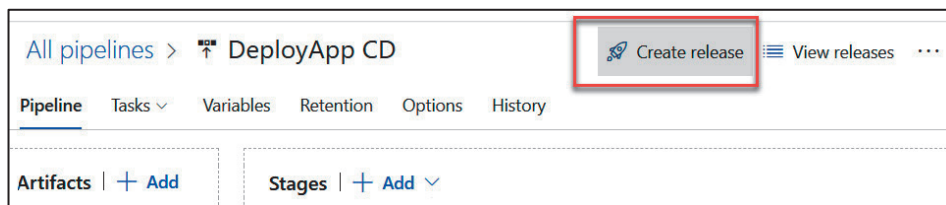
Rysunek 7.44. Wersja Azure — edycja nazwy usługi aplikacji

9. W końcu otrzymujemy definicję wersji w następujący sposób:

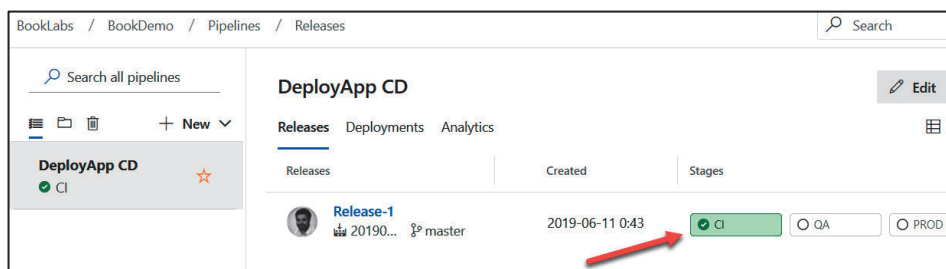


Rysunek 7.45. Azure Pipelines — definicja wersji

10. Aby uruchomić wdrożenie naszej aplikacji, utworzymy nową wersję, klikając przycisk *Create release*, jak pokazano na rysunku 7.46.
11. Pod koniec wykonania wdrożenia możemy zobaczyć jego stan, jak pokazano na rysunku 7.47.



Rysunek 7.46. Azure Pipelines — tworzenie wersji



Rysunek 7.47. Azure Pipelines — stan wdrożenia

Na tym rzucie ekranu widzimy, że wdrożenie w środowisku integracyjnym zostało pomyślnie zakończone.

Postępując zgodnie z krokami w tym laboratorium, utworzyliśmy potok CI/CD w **trybie interfejsu użytkownika** (ang. *user interface* — **UI**) (nazywanym klasycznym). Teraz dowiemy się, jak utworzyć potok za pomocą kodu w pliku YAML.

Tworzenie pełnej definicji potoku w pliku YAML

W poprzednich sekcjach omówiliśmy tworzenie potoku CI i CD w Azure DevOps za pomocą interfejsu użytkownika, co ma pewne zalety. Jednym z jego niewygodnych punktów jest trudność w zautomatyzowaniu tworzenia potoku dla wielu projektów.

Aby rozwiązać ten problem, Azure DevOps ma możliwość zapisania całej definicji potoku (CI i CD) w pliku YAML, który jest zapisywany w repozytorium.

Uwaga

Celem tej sekcji jest omówienie potoku YAML usługi Azure DevOps tylko dla procesu CI. Nie jest to zaawansowane laboratorium, a jeśli chcesz dowiedzieć się więcej, przeczytaj dokumentację tutaj: <https://docs.microsoft.com/en-us/azure/devops/pipelines/yaml-schema>.

W poniższym laboratorium nauczymy się tworzyć podstawową definicję potoku w pliku YAML.

Oto jak to zrobić:

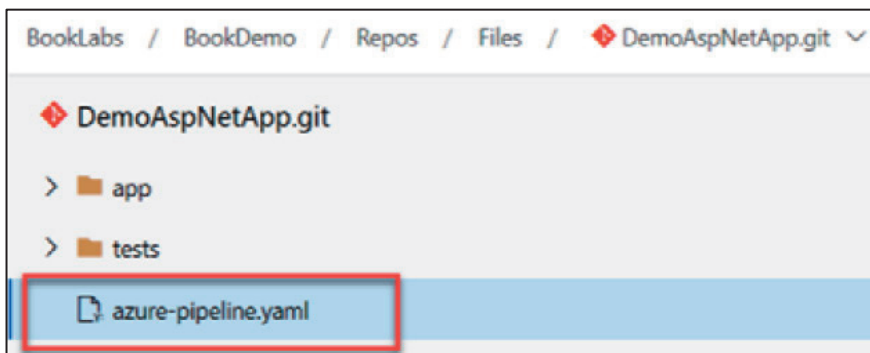
1. W Azure Repos, wewnątrz utworzonego wcześniej repozytorium *DemoBook*, dodaj nowy plik o nazwie *azure-pipeline.yaml* i następującej zawartości:

```
trigger:
- master
pool:
  vmImage: ubuntu-latest
steps:
- task: DotNetCoreCLI@2
  displayName: "Restore"
  inputs:
    command: restore
    projects: '**/*.csproj'
- task: DotNetCoreCLI@2
  displayName: "build"
  inputs:
    command: 'build'
    projects: '**/*.csproj'
    arguments: '--configuration Release'
- task: DotNetCoreCLI@2
  displayName: "Run tests"
  inputs:
    command: 'test'
    projects: '**/tests/*.csproj'
    arguments: '--configuration Release'
- task: DotNetCoreCLI@2
  displayName: "Code coverage"
  inputs:
    command: test
    projects: '**/*Tests/*.csproj'
    arguments: '--configuration Release --collect "Code coverage"'
```

W tym pliku mamy trzy sekcje: *trigger*, *pool* i *steps*. W sekcji *trigger* wskazujemy gałąź kodu, która wyzwala potok, więc tutaj każde zatwierdzenie w gałęzi *master* spowoduje wykonanie potoku.

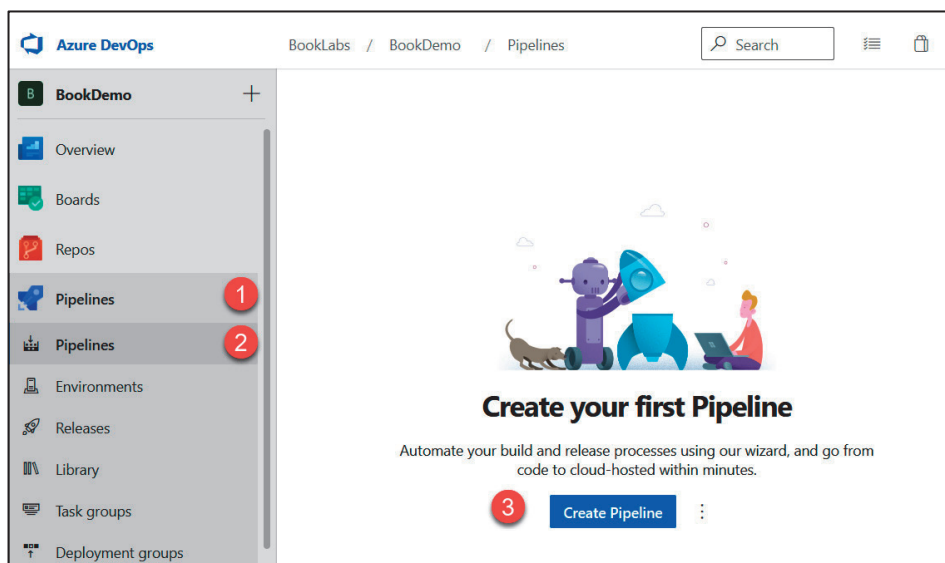
Następnie sekcja *pool* wskazuje pulę agentów, która ma być użyta do wykonania potoku. Na koniec w sekcji *steps* tworzymy wszystkie zadania, które zostaną wykonane podczas potoku. W naszym laboratorium zadeklarowaliśmy cztery zadania:

- Przywracanie pakietów NuGet dla projektu.
 - Kompilacja projektu.
 - Wykonanie testów.
 - Publikacja wykonanych testów pokrycia (ang. *code coverage*).
2. Zatwierdzamy i wysyłamy ten plik do katalogu głównego repozytorium w następujący sposób:



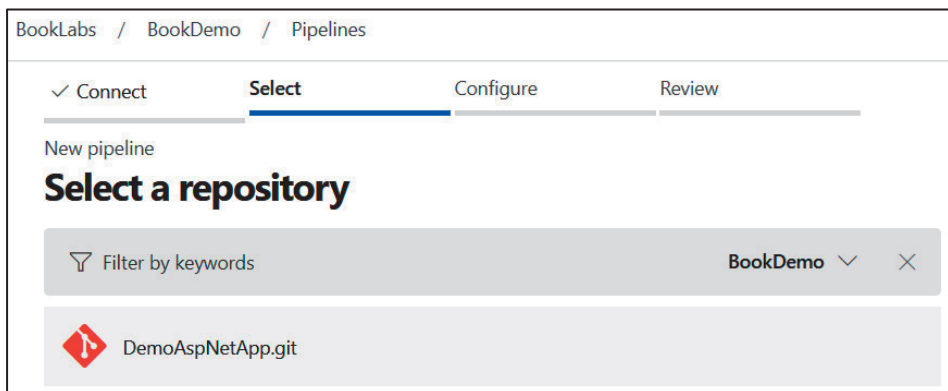
Rysunek 7.48. Azure Pipelines — potok YAML

3. W Azure Pipelines otwórz menu *Pipeline* i kliknij przycisk *Create Pipeline*, jak pokazano na poniższym zrzucie ekranu:



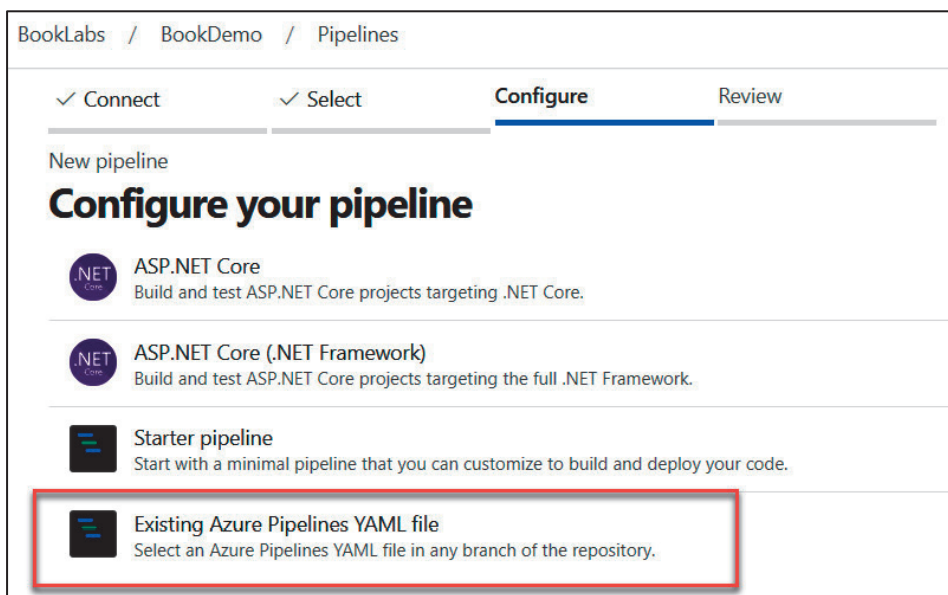
Rysunek 7.49. Azure Pipelines — tworzenie nowego potoku YAML

4. Wybierz źródło kodu (tutaj wybieramy *Azure Repos Git*) i repozytorium zawierające plik YAML potoku, jak pokazano na poniższym zrzucie ekranu:



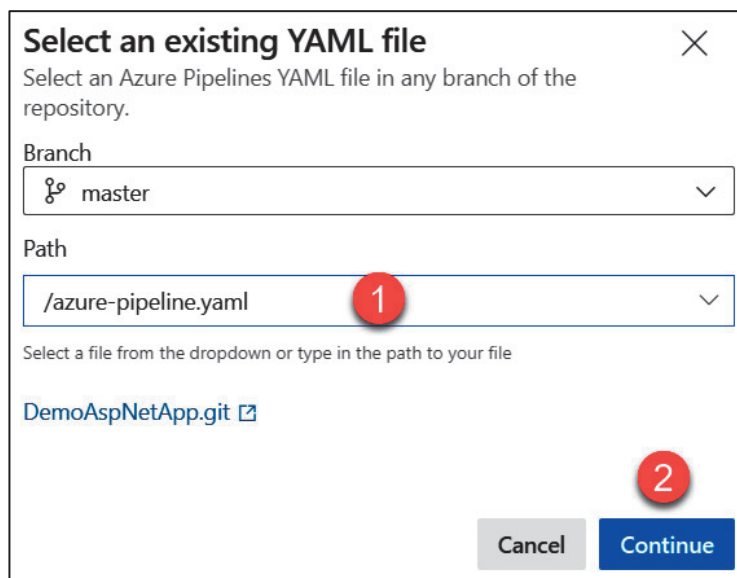
Rysunek 7.50. Azure Pipelines — wybór repozytorium

5. Następnie skonfiguruj potok platformy Azure, wybierając opcję użycia istniejącego pliku YAML, jak pokazano na poniższym zrzucie ekranu:



Rysunek 7.51. Azure Pipelines — wybór istniejącego pliku YAML

Następnie wybierz ścieżkę pliku YAML w następujący sposób:



Select an existing YAML file

Select an Azure Pipelines YAML file in any branch of the repository.

Branch

master

Path

/azure-pipeline.yaml

Select a file from the dropdown or type in the path to your file

[DemoAspNetApp.git](#)

Cancel Continue

Rysunek 7.52. Azure Pipelines — wybór pliku YAML

Potwierdź, klikając przycisk *Continue*.

6. Zawartość pliku jest wyświetlana na ekranie. Na koniec, aby uruchomić potok, kliknij przycisk *Run*, jak pokazano na rysunku 7.53.

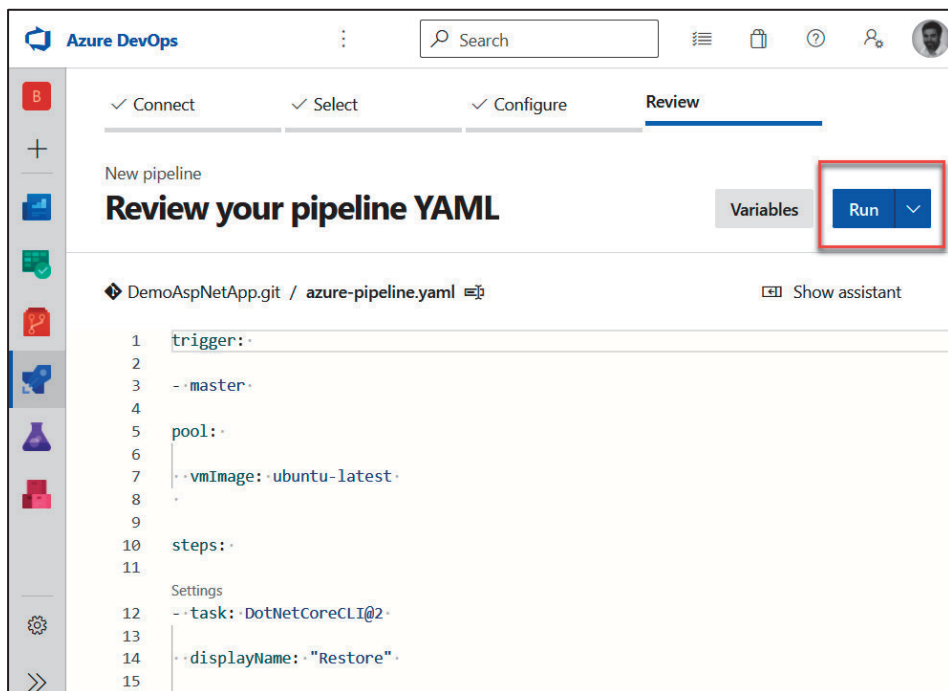
Potok jest tworzony i wykonywany.

7. Na koniec możemy wyświetlić szczegóły wykonania potoku, jak pokazano na rysunku 7.54.

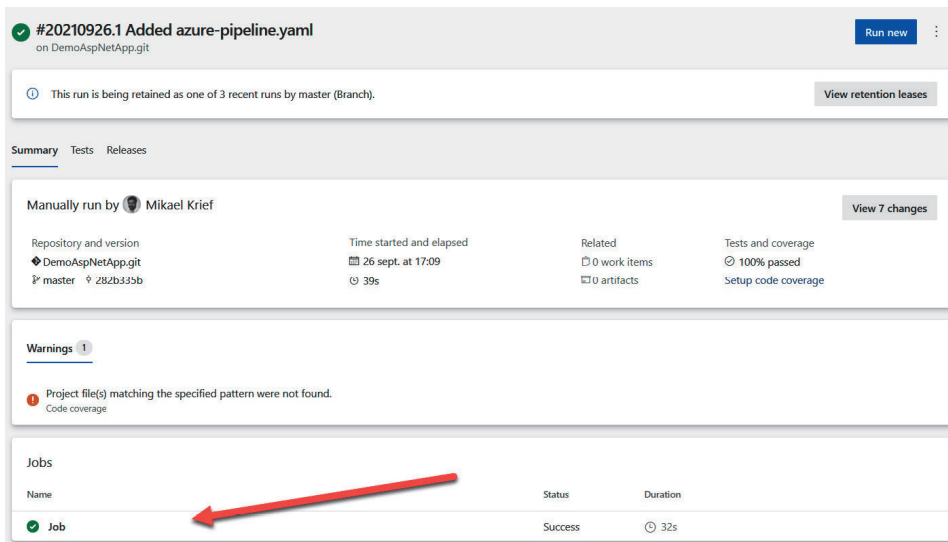
Kliknij *Job*, jak pokazano na poprzednim zrzucie ekranu, aby wyświetlić szczegóły wykonania każdego zadania. Powinieneś wtedy zobaczyć ekran jak na rysunku 7.55.

W tej sekcji poznaliśmy podstawy korzystania z plików YAML, które zawierają definicję potoku i mają wiele innych bardzo interesujących funkcji. Aby dowiedzieć się więcej, zapoznaj się z dokumentacją tutaj: <https://docs.microsoft.com/en-us/azure/devops/pipelines/?view=azure-devops>.

Przyjrzyjmy się teraz tworzeniu potoku CI za pomocą GitLab CI.



Rysunek 7.53. Azure Pipelines — uruchamianie potoku



Rysunek 7.54. Azure Pipelines — wykonanie zadania

| | |
|--|--|
| <div>← Jobs in run #20210926.1</div> <div>DemoAspNetApp.git</div> <div>Jobs</div> <div><div>✓ Job 32s</div><div>✓ Initialize job <1s</div><div>✓ Checkout DemoAspNet... 2s</div><div>✓ Restore 14s</div><div>✓ build 8s</div><div>✓ Run tests 5s</div><div>✓ Code coverage <1s</div><div>✓ Post-job: Checkout D... <1s</div><div>✓ Finalize Job <1s</div><div>✓ Report build status <1s</div></div> | |
| <div>✓ Job</div> <div>1 Pool: Azure Pipelines</div> <div>2 Image: ubuntu-latest</div> <div>3 Agent: Hosted Agent</div> <div>4 Started: Today at 17:09</div> <div>5 Duration: 32s</div> <div>6</div> <div>7 ▶ Job preparation parameters</div> <div>8 100% tests passed</div> | |

Rysunek 7.55. Azure Pipelines — szczegóły wykonania

Korzystanie z GitLab CI

W poprzednich sekcjach tego rozdziału dowiedzieliśmy się, jak tworzyć potoki CI/CD za pomocą Jenkinsa i Azure Pipelines.

Teraz spójrzmy na laboratorium korzystające z innego narzędzia DevOps, które zyskuje na popularności: **GitLab CI**.

GitLab CI to jedna z usług oferowanych przez serwis GitLab (<https://about.gitlab.com/>), który podobnie jak Azure DevOps jest platformą chmurową o następujących atrybutach:

- menedżer kodu źródłowego,
- menedżer potoku CI/CD,
- panel do zarządzania projektami.

Inne usługi, które oferuje, są wymienione tutaj: <https://about.gitlab.com/features/>.

GitLab posiada darmowy model cenowy z dodatkowymi usługami, które są płatne; cennik jest dostępny pod adresem <https://about.gitlab.com/pricing/>. Różnice między Azure DevOps i GitLabem są szczegółowo opisane w tym linku: <https://about.gitlab.com/devops-tools/azure-devops-vs-gitlab/>.

W tym laboratorium dowiemy się o następujących kwestiach:

1. Uwierzytelnianie w GitLabie.
2. Tworzenie nowego projektu i wersjonowanie jego kodu w GitLabie.
3. Utworzenie i wykonanie potoku CI w GitLab CI.

Uwierzytelnianie w GitLabie

Utworzenie konta GitLaba jest bezpłatne i można to zrobić, tworząc konto GitLaba lub korzystając z kont zewnętrznych, takich jak Google, GitHub, Twitter lub Bitbucket.

Aby utworzyć konto GitLaba, musimy wejść na https://gitlab.com/users/sign_in#register-pane i wybrać rodzaj uwierzytelnienia.

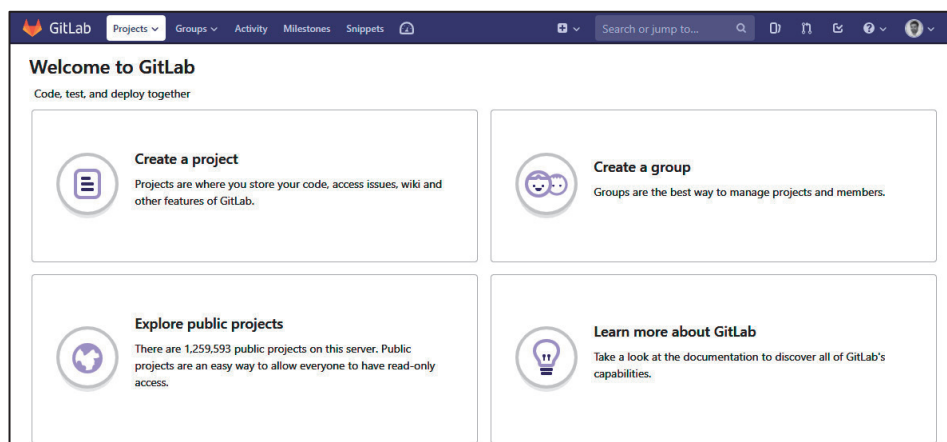
Poniższy zrzut ekranu przedstawia formularz uwierzytelniania GitLaba:

The screenshot shows the GitLab.com login interface. On the left, there is a sidebar with the GitLab logo and a list of links: 'Explore projects on GitLab.com (no login needed)', 'More information about GitLab.com', 'GitLab.com Support Forum', and 'GitLab Homepage'. Below these links, a message states: 'By signing up for and by signing in to this service you accept our:', followed by 'Privacy policy' and 'GitLab.com Terms'. The main content area has two tabs: 'Sign in' (active) and 'Register'. Under the 'Sign in' tab, there are input fields for 'Username or email' and 'Password', each with a toggle for password visibility. Below the password field is a 'Remember me' checkbox and a 'Forgot your password?' link. A large green 'Sign in' button is positioned below these fields. Under the 'Sign in with' section, there are buttons for 'Google', 'Twitter', 'GitHub', 'Bitbucket', and 'Salesforce'. A 'Remember me' checkbox is also present at the bottom of this section.

Rysunek 7.56. Rejestracja w GitLabie

Po utworzeniu i uwierzytelnieniu konta zostaniesz przeniesiony na stronę główną swojego konta, która oferuje wszystkie funkcje pokazane na poniższym zrzucie ekranu.

Teraz, gdy przeszliśmy uwierzytelnienie, idźmy dalej i utwórzmy nowy projekt.

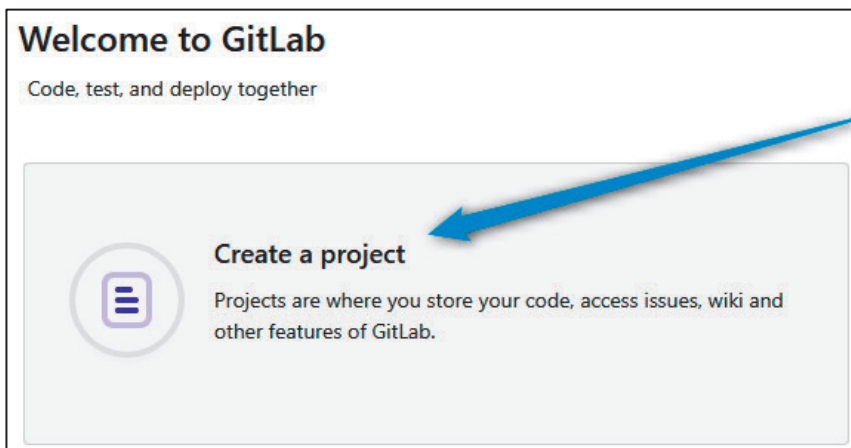


Rysunek 7.57. Strona główna GitLaba

Tworzenie nowego projektu i zarządzanie kodem źródłowym

Aby utworzyć nowy projekt w GitLabie, wykonaj następujące kroki:

1. Kliknij *Create a project* na stronie głównej, jak pokazano na poniższym zrzucie ekranu:








Rysunek 7.58. Nowy projekt GitLaba

- Następnie mamy do wyboru kilka opcji, jak niżej:
 - Aby utworzyć pusty projekt (bez kodu), formularz poprosi o wpisanie nazwy projektu, jak pokazano na poniższym zrzucie ekranu:

| Blank project | Create from template | Import project | CI/CD for external repo |
|--|----------------------|--|---------------------------------------|
| Project name <input type="text" value="BookDemo"/> | | | |
| Project URL <input type="text" value="https://gitlab.com/mikakrief/"/> | | Project slug <input type="text" value="bookdemo"/> | |
| Want to house several dependent projects under the same namespace? Create a group. | | | |
| Project description (optional) <input type="text" value="Description format"/> | | | |
| Visibility Level ⓘ <input checked="" type="radio"/> Private Project access must be granted explicitly to each user. <input type="radio"/> Internal The project can be accessed by any logged in user. <input type="radio"/> Public The project can be accessed without any authentication. | | | |
| <input type="checkbox"/> Initialize repository with a README Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository. | | | |
| <input type="button" value="Create project"/> | | | <input type="button" value="Cancel"/> |

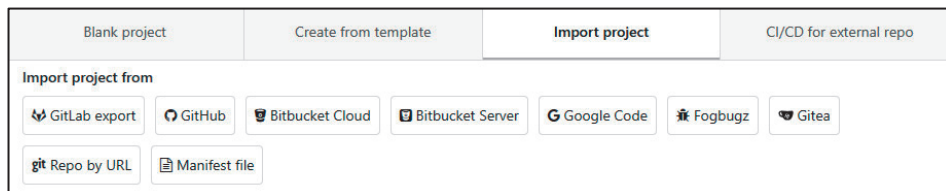
Rysunek 7.59. Konfiguracja projektu GitLaba

- Aby utworzyć nowy projekt z wbudowanego projektu szablonu, wykonaj następujące czynności:

| Blank project | Create from template | Import project | CI/CD for external repo |
|--|----------------------|--|---|
| <input type="text" value="Learn how to contribute to the built-in templates"/> | | | |
| Built-in 17 Instance 0 Group 0 | | | |
|  Ruby on Rails Includes an MVC structure, Gemfile, Rakefile, along with many others, to help you get started. | | <input type="button" value="Preview"/> | <input type="button" value="Use template"/> |
|  Spring Includes an MVC structure, mvnw and pom.xml to help you get started. | | <input type="button" value="Preview"/> | <input type="button" value="Use template"/> |
|  NodeJS Express Includes an MVC structure to help you get started. | | <input type="button" value="Preview"/> | <input type="button" value="Use template"/> |
|  iOS (Swift) A ready-to-go template for use with iOS Swift apps. | | <input type="button" value="Preview"/> | <input type="button" value="Use template"/> |
|  .NET Core A .NET Core console application template, customizable for any .NET Core project | | <input type="button" value="Preview"/> | <input type="button" value="Use template"/> |

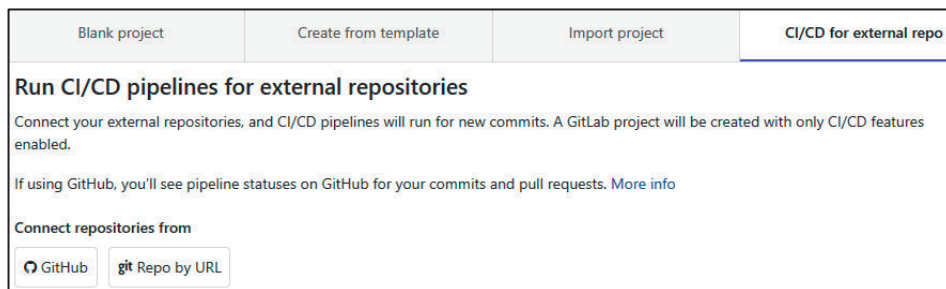
Rysunek 7.60. Szablon projektu GitLaba

- Aby zaimportować kod z wewnętrznego lub zewnętrznego repozytorium innej platformy SVC, wykonaj następujące czynności:



Rysunek 7.61. GitLab — importowanie kodu

- Kod do zaimportowania znajduje się w zewnętrznym repozytorium SVC, jak pokazano na poniższym zrzucie ekranu:



Rysunek 7.62. GitLab — CI/CD dla zewnętrznych repozytoriów

W naszym przypadku, dla tego laboratorium, zaczniemy od pierwszej opcji, która jest pustym projektem. W formularzu wybieramy nazwę projektu, np. *BookDemo*, a następnie zatwierdzamy ją, klikając przycisk *Create a project*.

2. Po utworzeniu projektu dostaniemy stronę wskazującą różne polecenia Gita, które należy wykonać, aby wysłać kod.
3. W tym celu na naszym lokalnym dysku utworzymy nowy plik *gitlab-ci-demo.yml*, a następnie skopiujemy zawartość naszego przykładu, który można znaleźć pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP07>, w pliku *gitlab-ci-demo.yml*.
4. Następnie wykonamy następujące polecenia w terminalu, aby wysłać kod do repozytorium, jak szczegółowo opisano w rozdziale 6., „Zarządzanie kodem źródłowym za pomocą Gita”:

```
git init
git remote add origin <Url repozytorium gita>
git add .
```

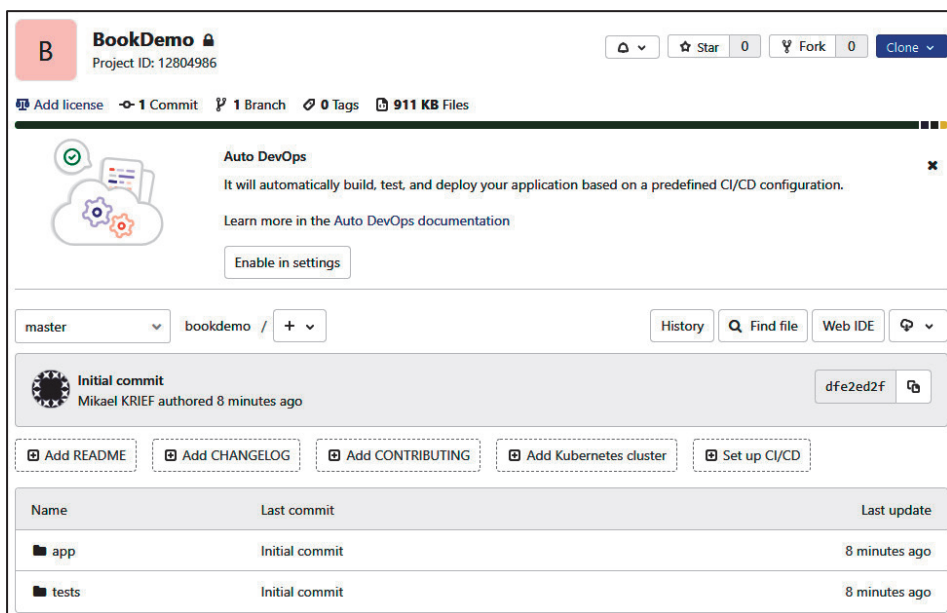
```
git commit -m "Początkowe zatwierdzenie"
git push -u origin master
```

Uwaga

Podczas wykonywania poleceń zostaniesz poproszony o Twój **identyfikator konta GitLab** (ID), ponieważ jest to projekt prywatny. Po zalogowaniu się na swoje konto w portalu internetowym GitLab Twoja nazwa użytkownika będzie widoczna na stronie Twojego konta pod adresem <https://gitlab.com/profile/account>.

Po wykonaniu tych poleceń otrzymamy zdalne repozytorium GitLab z naszym kodem laboratoryjnym.

Poniższy zrzut ekranu przedstawia zdalne repozytorium GitLab:



Rysunek 7.63. Repozytorium GitLab

Kod naszej aplikacji został przesłany i możemy teraz utworzyć nasz proces CI za pomocą GitLab.

Tworzenie potoku CI

W GitLab CI tworzenie potoku CI (i CD) odbywa się nie za pomocą graficznego interfejsu użytkownika (GUI), ale za pomocą pliku YAML w katalogu głównym projektu.

Ta metoda, polegająca na opisaniu procesu potoku w postaci kodu zapisanego w pliku, może być nazwana *Pipeline as Code* (PaC), podobnie jak *Infrastructure as Code* (IaC). Postępujemy w następujący sposób:

1. Aby utworzyć ten potok, utworzymy w katalogu głównym kodu aplikacji plik `.gitlab-ci.yml` o następującej treści:

```
image: microsoft/dotnet:latest
stages:
  - build
  - test

variables:
  BuildConfiguration: "Release"

build:
  stage: build
  script:
    - "cd app"
    - "dotnet restore"
    - "dotnet build --configuration
$BuildConfiguration"
test:
  stage: test
  script:
    - "cd tests"
    - "dotnet test --configuration
$BuildConfiguration"
```

Uwaga

Kod źródłowy z tego pliku jest również dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP07/gitlab-ci-demo.yml>.

Na początku tego kodu używamy obrazu Dockera `microsoft/dotnet:latest`, który zostanie zamontowany w kontenerze i w którym będą wykonywane akcje potoku.

Następnie definiujemy dwa etapy: jeden dla kompilacji i jeden dla wykonania testu, a także zmienną `BuildConfiguration`, która będzie używana w skryptach.

Na koniec wskazujemy katalogi, w których mają się wykonać skrypty każdego etapu. Te skrypty .NET Core są identyczne z tymi, które widzieliśmy w sekcji „Korzystanie z Azure Pipelines dla CI/CD”.

Uwaga

Pełna dokumentacja dotycząca formatu i składni pliku `.gitlab-ci.yml` jest dostępna tutaj: <https://docs.gitlab.com/ee/ci/yaml/>.

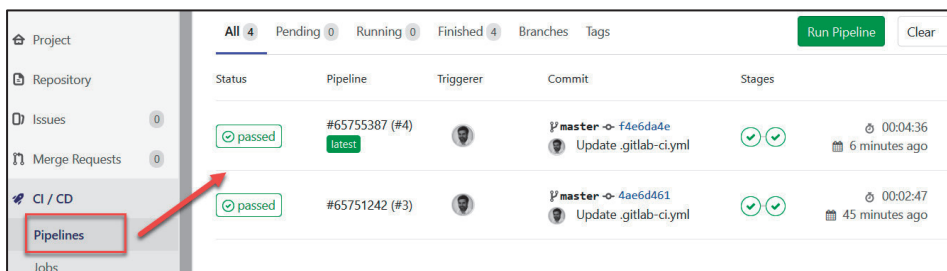
2. Następnie zatwierdzamy i wysyłamy ten plik do zdalnego repozytorium.
3. Zaraz po wysłaniu kodu widzimy, że proces CI został uruchomiony.

Nasz potok CI został zatem wyzwolony, gdy kod został wysłany do repozytorium. Zobaczmy teraz, jak uzyskać szczegóły jego wykonania.

Dostęp do szczegółów wykonania potoku CI

Aby uzyskać dostęp do szczegółów wykonania wykonanego potoku CI, wykonaj następujące kroki:

1. W menu GitLab CI przejdź do *CI / CD / Pipelines*, a zobaczysz listę wykonanych potoków, jak pokazano na poniższym zrzucie ekranu:



Rysunek 7.64. Potoki GitLaba

2. Aby wyświetlić szczegóły potoku, klikamy żądane wykonanie potoku, jak pokazano na rysunku 7.65.
3. Możemy zobaczyć stan wykonania, a także dwa etapy, które zdefiniowałeś w pliku YAML potoku. Aby wyświetlić szczegóły dzienników wykonania dla etapu, wybieramy go, jak pokazano na rysunku 7.66.

Teraz możemy zobaczyć wykonanie skryptów napisanych w pliku potoku.

W tej sekcji widzieliśmy implementację potoku CI w GitLab CI z inicjalizacją zdalnego repozytorium i utworzeniem pliku YAML do konfiguracji potoku, a także wykonanie potoku.

The screenshot shows a GitLab CI pipeline execution page. At the top, a green status bar indicates the pipeline is 'passed'. The pipeline is identified as '#65755387 (#4)' and was triggered 15 minutes ago by 'Mikael Krief'. The main title of the pipeline is 'Update .gitlab-ci.yml'. Below this, it states '2 jobs for master in 4 minutes and 36 seconds'. A 'latest' tag is visible. The commit hash 'f4e6da4e' is shown with a link to the commit. The pipeline is divided into two sections: 'Pipeline' and 'Jobs 2'. The 'Jobs' section shows two jobs: 'build' and 'test', both of which are marked as 'passed' with green checkmarks. The 'build' job is followed by a refresh icon, and the 'test' job is followed by a refresh icon.

Rysunek 7.65. Wykonanie potoku GitLaba

The screenshot shows a GitLab CI job execution log for 'Job #229390643', triggered 19 minutes ago by 'Mikael Krief'. The log is displayed in a dark-themed terminal window. The log content is as follows:

```
Running with gitlab-runner 11.11.2 (ac2a293c)
on docker-auto-scale ed2dce3a
Using Docker executor with image microsoft/dotnet:latest ...
Pulling docker image microsoft/dotnet:latest ...
Using docker image sha256:08663b8eaa01a928bf4b22c6d7892a5306dc76a40d34e9449465d7f8d0c5ec38 for microsoft/dotnet:latest ...
Running on runner-ed2dce3a-project-12804986-concurrent-0 via runner-ed2dce3a-srm-1560285231-e19116a0...
Initialized empty Git repository in /builds/mikakrief/bookdemo/.git/
Fetching changes...
Created fresh repository.
From https://gitlab.com/mikakrief/bookdemo
* [new branch]      master -> origin/master
Checking out f4e6da4e as master...

Skipping Git submodules setup
$ cd app
$ dotnet restore
Restore completed in 13.32 sec for /builds/mikakrief/bookdemo/app/app.csproj.
```

Rysunek 7.66. Szczegóły dziennika wykonania GitLaba

Podsumowanie

W tym rozdziale przyjrzelśmy się jednemu z najważniejszych tematów w DevOps: procesowi CI/CD. Zaczęliśmy od prezentacji zasad CI i CD. Następnie skupiliśmy się na menedżerach pakietów, przyglądając się NuGet, npm, Nexus i Azure Artifacts.

Na koniec zobaczyliśmy, jak zaimplementować i wykonać potok E2E CI/CD przy użyciu trzech różnych narzędzi: Jenkins, Azure Pipelines i GitLab CI. Dla każdego z nich przyjrzelśmy się archiwizowaniu kodu źródłowego aplikacji wraz z tworzeniem potoku i jego wykonaniem.

Po przeczytaniu tego rozdziału powinniśmy być w stanie utworzyć potok dla CI i CD z zarządzaniem kodem źródłowym. Ponadto będziemy mogli wybrać i wykorzystać menedżer pakietów do centralizacji i dystrybucji naszych pakietów.

W następnym rozdziale opowiemy o tworzeniu potoku CI/CD dla projektu IaC przy użyciu Azure DevOps w celu wykonania kodu Packera, Terraform i Ansible.

Pytania

1. Jakie są warunki wstępne dla wdrożenia potoku CI?
2. Kiedy zostanie uruchomiony potok CI?
3. Jakie jest zastosowanie menedżera pakietów?
4. Jakie typy pakietów są przechowywane w menedżerze pakietów NuGet?
5. Z którą platformą zintegrowana jest usługa Azure Artifacts?
6. Czy Jenkins jest usługą chmurową?
7. Jak nazywa się w Azure DevOps usługa umożliwiająca zarządzanie potokami CI/CD?
8. Jakie trzy usługi oferuje GitLab?
9. Który element w GitLab CI pozwala na zbudowanie potoku CI?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o potokach CI/CD, oto kilka zasobów, z którymi możesz się zapoznać:

- *Hands-On Continuous Integration and Delivery* — <https://www.packtpub.com/virtualization-and-cloud/hands-continuous-integration-and-delivery>.

- *Continuous Integration, Delivery, and Deployment* — <https://www.packtpub.com/application-development/continuous-integration-delivery-and-deployment>.
- *Mastering Jenkins* — <https://www.packtpub.com/application-development/mastering-jenkins>.
- *Azure DevOps Server 2019 Cookbook* — <https://www.packtpub.com/networking-and-servers/azure-devops-server-2019-cookbooks-econd-edition>.
- *Mastering GitLab 12* — <https://www.packtpub.com/cloud-networking/mastering-gitlab-12>.

Wdrażanie infrastruktury jako kodu za pomocą potoku CI/CD

Rozdział

8

W pierwszej części tej książki dowiedzieliśmy się wiele na temat **infrastruktury jako kodu (IaC)** i narzędzi takich jak Terraform, Packer i Ansible. W drugiej części omówiliśmy Gita i poświęciliśmy rozdział ciągłej integracji i ciągłemu wdrażaniu aplikacji.

Nie powinniśmy jednak zaniedbywać stosowania CI/CD w praktyce jako IaC, co pozwoli nam zorkiestrować i zautomatyzować cały proces udostępniania infrastruktury.

W drugim wydaniu książki chciałem dodać ten rozdział, aby ponownie wykorzystać wszystko, czego nauczyliśmy się w poprzednich rozdziałach, i pokazać, jak zaimplementować IaC za pomocą potoku CI/CD, który będzie się składał z Packera, Terraform i Ansible.

Narzędzie potoku, które zostanie użyte, to Azure Pipelines. Omówiliśmy je szczegółowo w rozdziale 7., „Ciągła integracja i ciągłe wdrażanie”. Dowiemy się więc, jak napisać potok Azure w YAML, aby wygenerować obraz za pomocą Packera. Następnie zobaczymy, jak napisać potok w języku YAML, aby udostępnić **maszynę wirtualną (VM)** za pomocą Terraform, i jak uzupełnić ten potok o wykonanie Ansible, aby zainstalować nginx na tej maszynie wirtualnej.

W tym rozdziale omówiono następujące tematy:

- uruchamianie Packera w Azure Pipelines,
- uruchamianie Terraform i Ansible w Azure Pipelines.

Wymagania techniczne

Ten rozdział wymaga:

- Subskrypcji Azure na potrzeby tworzenia obrazu Packera i udostępniania maszyny wirtualnej; bezpłatne konto możesz utworzyć tutaj: <https://azure.microsoft.com/en-us/free/search/>.
- Konta Azure DevOps do tworzenia definicji potoków YAML; bezpłatne konto możesz utworzyć tutaj: <https://azure.microsoft.com/en-us/pricing/details/devops/azure-devops-services/>.

Kod źródłowy tego rozdziału jest dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP08>.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/3BDT9ne>.

Uruchamianie Packera w Azure Pipelines

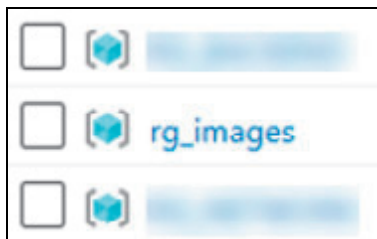
Aby uruchomić nasz potok IaC, automatycznie utworzymy obraz maszyny wirtualnej platformy Azure za pomocą Packera. Aby wykonać tę operację, skorzystamy z Azure Pipelines i potoku w formacie YAML. Celem tego potoku jest automatyczne wykonanie polecenia Packera.

Ważna uwaga

Aby uzyskać więcej informacji na temat tworzenia potoku YAML w Azure DevOps, przeczytaj rozdział 7, „Ciągła integracja i ciągłe wdrażanie”.

Szablon Packera, którego użyjemy w laboratorium w tej sekcji, to kod, którego nauczyliśmy się w rozdziale 4, „Optymalizacja wdrażania infrastruktury za pomocą Packera”. Ten kod źródłowy jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP08/packer/.pkr.hcl>.

Zanim zaczniemy używać Packera, musimy najpierw utworzyć grupę zasobów platformy Azure, która będzie przechowywać obraz maszyny wirtualnej utworzony przez potok. W naszym laboratorium nazwaliśmy tę grupę zasobów platformy Azure *rg_images*. Poniższy zrzut ekranu przedstawia grupę zasobów platformy Azure:



Rysunek 8.1. Grupa zasobów platformy Azure dla obrazu Packera

Teraz możemy napisać kod potoku, który będzie wykonywał polecenia na szablonie Packera.

Aby to zrobić, w tym samym folderze co szablon Packera utwórz nowy plik YAML opisujący kroki potoku, a następnie napisz trzy bloki kodu w następujący sposób:

1. Pierwszym krokiem jest wykonanie polecenia `packer init`:

```
- script: packer init $(Build.SourcesDirectory)/CHAP08/packer/.pkr.hcl
  displayName: Packer init
```

Drugim krokiem jest sprawdzenie poprawności szablonu Packera poprzez uruchomienie komendy `packer validate`:

```
- script: packer validate $(Build.SourcesDirectory)/CHAP08/packer/
  ↪.pkr.hcl
  displayName: Packer validate template
```

2. Ostatnim krokiem jest zbudowanie obrazu za pomocą Packera poprzez uruchomienie komendy `packer build`:

```
- script: packer build $(Build.SourcesDirectory)/CHAP08/packer/.pkr.hcl
  displayName: Packer build template
  env:
    PKR_VAR_clientid: $(PKR_VAR_clientid)
    PKR_VAR_clientsecret: $(PKR_VAR_clientsecret)
    PKR_VAR_subscriptionid: $(PKR_VAR_subscriptionid)
    PKR_VAR_tenantid: $(PKR_VAR_tenantid)
```

W tym kroku dodajemy do skryptu zmienne środowiskowe odpowiadające kontu jednostki usługi Azure. Aby zmienne środowiskowe mogły być używane w szablonie Packera, muszą być zadeklarowane jako `PKR_VAR_<nazwa zmiennej>`.

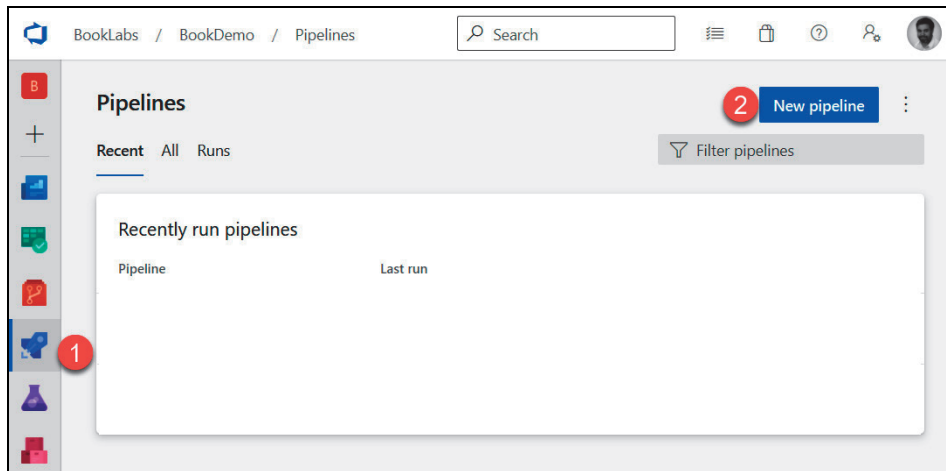
Ważna uwaga

Cały kod źródłowy tego pliku potoku YAML jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP08/packer/pipeline.yaml>.

Następnie zatwierdź i wyślij ten plik do repozytorium Gita. W tym laboratorium korzystamy z GitHuba, chociaż możesz korzystać z innych repozytoriów Gita, takich jak Azure Repos lub Bitbucket.

Ostatnim krokiem jest utworzenie i uruchomienie potoku Azure, wykonując następujące kroki:

1. W menu Azure Pipelines kliknij *New pipeline*:



Rysunek 8.2. Tworzenie nowego potoku w Azure Pipelines

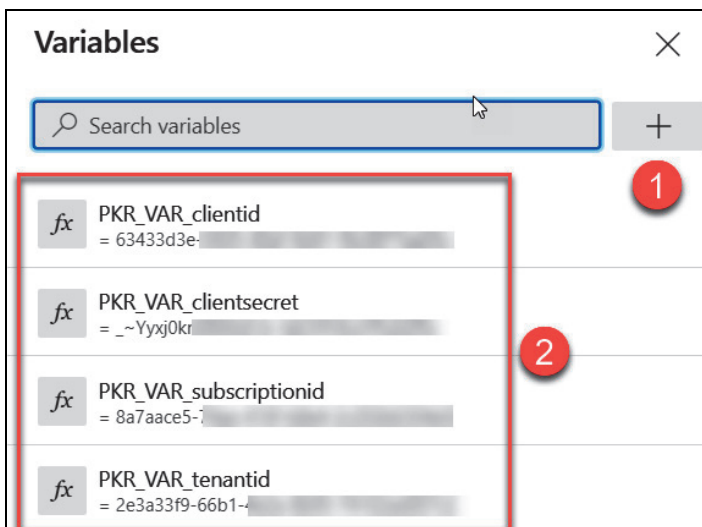
2. Wybierz repozytorium Gita, które zawiera szablony Packera i plik potoku YAML, a następnie opcję użycia istniejącego pliku potoku YAML. Wszystkie szczegóły dotyczące tego kroku są wyjaśnione w sekcji „Tworzenie pełnej definicji potoku w pliku YAML” w rozdziale 7., „Ciągła integracja i ciągłe wdrażanie”.

Po utworzeniu potoku, w trybie *Edit*, dodaj cztery zmienne, aby przechowywać informacje o poświadczeniach jednostki usługi platformy Azure (rysunek 8.3).

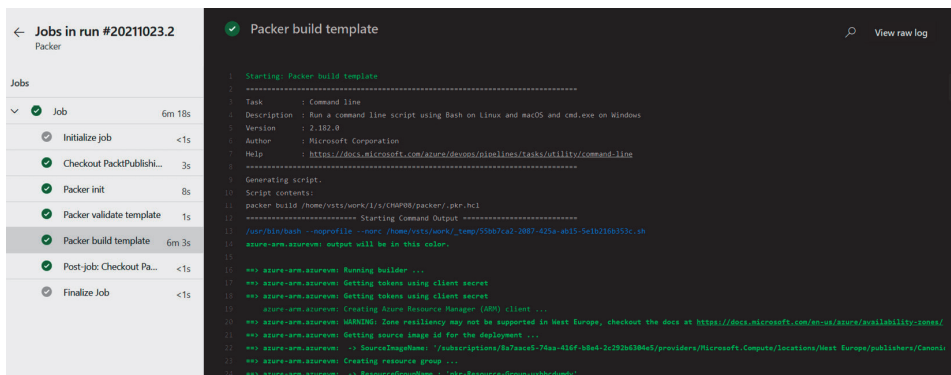
Są to zmienne, których użyliśmy w potoku YAML w ostatnim kroku skryptu.

3. Następnie możemy uruchomić potok i poczekać do końca wykonania. W panelu logowania możemy wyświetlić wszystkie szczegóły wykonania (rysunek 8.4).
4. W portalu Azure możemy zobaczyć wygenerowany obraz maszyny wirtualnej w grupie zasobów platformy Azure (rysunek 8.5).

W tej sekcji dowiedzieliśmy się, jak automatycznie tworzyć obrazy Packera przy użyciu potoków w Azure DevOps. W następnej sekcji będziemy kontynuować pracę z potokiem IaC, aby wykonywać polecenia Terraform i Ansible w automatycznych potokach.



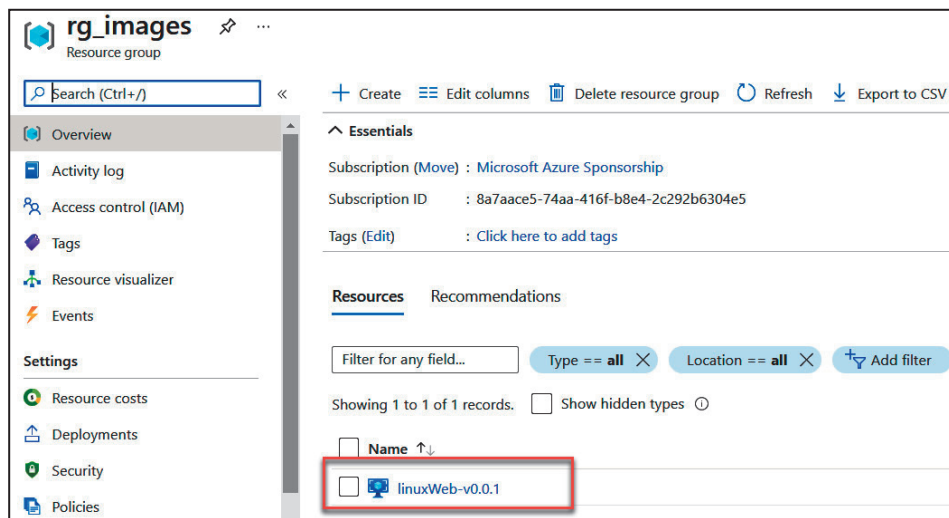
Rysunek 8.3. Zmienne Azure Pipelines



Rysunek 8.4. Dzienniki potoku Azure Pipelines Packer

Uruchamianie Terraform i Ansible w Azure Pipelines

Po utworzeniu potoku Packera utworzymy potok służący do udostępniania maszyny wirtualnej platformy Azure, która używa obrazu utworzonego za pomocą Packera. Następnie ten potok skonfiguruje maszynę wirtualną za pomocą narzędzia Ansible. W tym przykładzie użyjemy Ansible, aby zainstalować nginx na tej maszynie wirtualnej.



Rysunek 8.5. Obraz Packera na platformie Azure

Konfiguracja Terraform używana w tym kodzie spowoduje udostępnienie nowej grupy zasobów platformy Azure, sieci wirtualnej z podsiecią i maszyny wirtualnej z systemem Linux z rolą tag o wartości *webserver*.

Kompletny kod Terraform jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP08/terraform>. Nie będę tego wyjaśniał, ponieważ jest to ten sam kod, którego nauczyliśmy się w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”.

Pełny kod Ansible, którego użyliśmy do zainstalowania nginx na maszynie z systemem Linux, jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP08/ansible/playbookdemo.yml>. Dowiedzieliśmy się już o tym w rozdziale 3., „Używanie Ansible do konfiguracji infrastruktury IaaS”. W przypadku Ansible używamy **dynamicznego pliku inwentarza**, który wybiera hosty maszyn wirtualnych na podstawie określonej grupy zasobów platformy Azure i tagu na maszynie wirtualnej.

Kod konfiguracyjny tego dynamicznego pliku inwentarza jest dostępny tutaj: https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP08/ansible/inv.azure_rm.yml.

Teraz, gdy mamy już cały kod Terraform i Ansible, możemy napisać kod YAML potoków Azure, wykonując poniższe kroki. W tym celu utwórz nowy plik *azure-pipeline.yaml* z następującą zawartością:

1. Pierwszym krokiem jest uruchomienie pracy Terraform za pomocą poleceń `init`, `plan` i `apply`:

```
- script: terraform init --backend-config backend.
```

```
tfvars
```

```
  displayName: Terraform init
  workingDirectory: $(Build.SourcesDirectory)/CHAP08/terraform
  env:
    ARM_CLIENT_ID: $(AZURE_CLIENT_ID)
    ARM_CLIENT_SECRET: $(AZURE_SECRET)
    ARM_SUBSCRIPTION_ID: $(AZURE_SUBSCRIPTION_ID)
    ARM_TENANT_ID: $(AZURE_TENANT)
    ARM_ACCESS_KEY: $(AZURE_ACCESS_KEY)
```

```
- script: terraform plan
```

```
  displayName: Terraform plan
  workingDirectory: $(Build.SourcesDirectory)/CHAP08/terraform
  env:
    ARM_CLIENT_ID: $(AZURE_CLIENT_ID)
    ARM_CLIENT_SECRET: $(AZURE_SECRET)
    ARM_SUBSCRIPTION_ID: $(AZURE_SUBSCRIPTION_ID)
    ARM_TENANT_ID: $(AZURE_TENANT)
    ARM_ACCESS_KEY: $(AZURE_ACCESS_KEY)
```

```
- script: terraform apply -auto-approve
```

```
  displayName: Terraform apply
  workingDirectory: $(Build.SourcesDirectory)/CHAP08/terraform
  env:
    ARM_CLIENT_ID: $(AZURE_CLIENT_ID)
    ARM_CLIENT_SECRET: $(AZURE_SECRET)
    ARM_SUBSCRIPTION_ID: $(AZURE_SUBSCRIPTION_ID)
    ARM_TENANT_ID: $(AZURE_TENANT)
    ARM_ACCESS_KEY: $(AZURE_ACCESS_KEY)
```

W tych zadaniach skryptowych uruchamiamy następujące trzy polecenia:

- Polecenie `terraform init`, które korzysta z konfiguracji backendu Terraform.
- Polecenie `terraform plan` do podglądu zmian.
- Polecenie `terraform apply` do wprowadzania zmian i tworzenia maszyny wirtualnej, jeśli nie istnieje. Do tego polecenia dodajemy opcję `-auto-approve`, by automatycznie zastosować zmiany bez pytania o potwierdzenie.

Ważna uwaga

Opcjonalnie możemy dodać tutaj jeden krok, by ręcznie zatwierdzić wynik polecenia `plan` przed zastosowaniem zmian. W tym laboratorium jesteśmy pewni zmian i potok może bezpośrednio zastosować zmiany.

W przypadku uwierzytelniania platformy Azure używamy poświadczeń jednostki usługi i klucza dostępu do usługi Azure Storage (dla zaplecza stanu) jako zmiennych środowiskowych.

2. Następnie w tym potoku kontynuujemy wykonywanie Ansible z następującym kodem:

```
- script: pip install ansible[azure]==2.8.6
  displayName: Get requirements

- script: ansible-playbook playbookdemo.yml -i inv.azure_rm.yml
  displayName: Ansible playbook
  workingDirectory: $(Build.SourcesDirectory)/CHAP08/ansible
  env:
    AZURE_CLIENT_ID: $(AZURE_CLIENT_ID)
    AZURE_SECRET: $(AZURE_SECRET)
    AZURE_SUBSCRIPTION_ID: $(AZURE_SUBSCRIPTION_ID)
    AZURE_TENANT: $(AZURE_TENANT)
    ANSIBLE_HOST_KEY_CHECKING: False
```

W tym kodzie pierwszy skrypt instaluje wtyczki platformy Azure dla Ansible, które są wymagane dla dynamicznego pliku inwentarza.

Drugi skrypt wykonuje playbook Ansible i używa dynamicznego inwentarza skonfigurowanego w pliku *inv.azure_rm.yml*. Używamy również poświadczeń jednostki usługi Azure jako zmiennych środowiskowych.

Następnie zatwierdź i wyślij ten plik do repozytorium Gita.

Ostatnim krokiem jest utworzenie i uruchomienie potoku platformy Azure. Wykonaj następujące kroki:

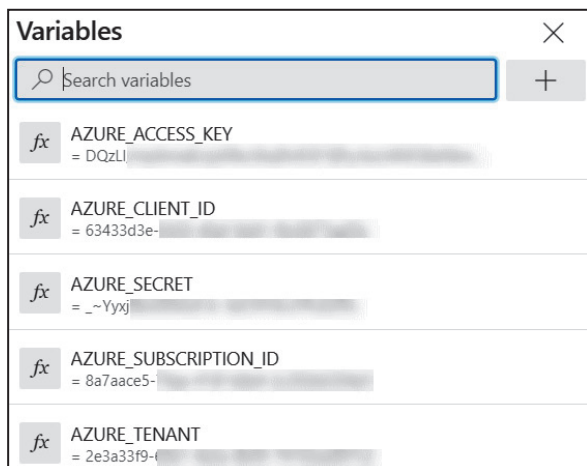
1. W menu *Azure Pipelines* kliknij *New pipeline*.
2. Wybierz repozytorium Gita, które zawiera szablony Packera i plik potoku YAML, a następnie opcję użycia istniejącego pliku potoku YAML.

Wszystkie szczegóły dotyczące tego kroku są wyjaśnione w sekcji „Tworzenie pełnej definicji potoku w pliku YAML” w rozdziale 7., „Ciągła integracja i ciągłe wdrażanie”.

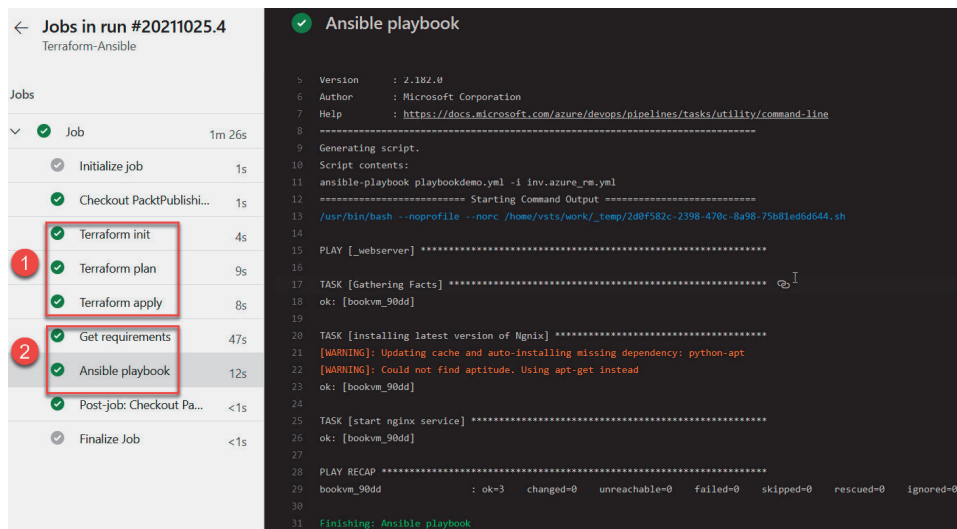
3. Po utworzeniu potoku, w trybie *Edit*, dodaj pięć zmiennych do przechowywania informacji o poświadczeniach jednostki usługi Azure i klucza dostępu do usługi Azure Storage dla zaplecza stanu Terraform (rysunek 8.6).

Są to zmienne, których używaliśmy w potoku YAML we wszystkich krokach skryptu.

4. Następnie możemy uruchomić potok i poczekać do końca wykonania. W panelu logowania możemy wyświetlić krok po kroku wszystkie szczegóły wykonania (rysunek 8.7).



Rysunek 8.6. Zmienne Terraform i Ansible w Azure Pipelines



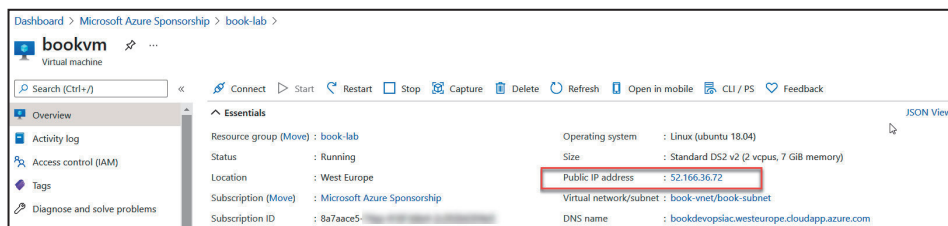
Rysunek 8.7. Szczegóły dziennika Terraform i Ansible Azure Pipelines

Na tym ekranie widzimy wykonanie polecenia Terraform, a następnie wykonanie Ansible.

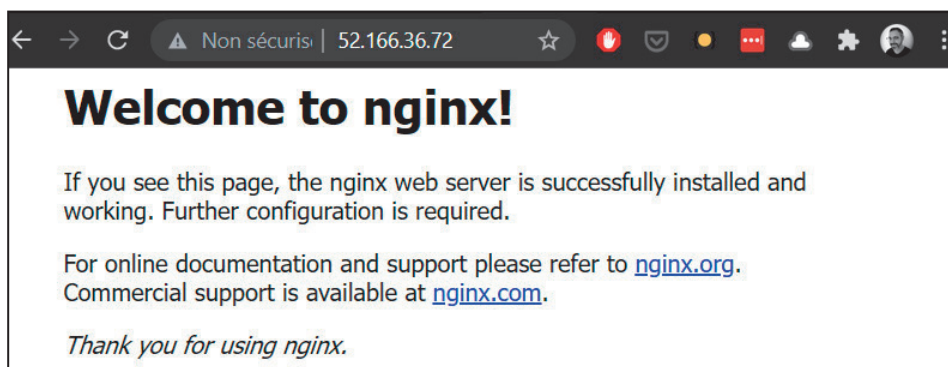
5. Na koniec, aby sprawdzić, czy wykonanie się powiodło, znajdź publiczny adres IP na tej maszynie wirtualnej w witrynie Azure Portal — rysunek 8.8.

Otwórz przeglądarkę i przejdź do tego publicznego adresu IP — rysunek 8.9.

Możemy teraz zobaczyć stronę główną nginx.



Rysunek 8.8. Publiczny adres IP maszyny wirtualnej Azure



Rysunek 8.9. Strona główna nginx

W tej sekcji dowiedzieliśmy się, jak utworzyć kod potoku do wykonywania Terraform i automatycznego uruchomienia Ansible za pomocą potoku YAML w Azure DevOps.

Podsumowanie

W tym rozdziale zaimplementowaliśmy przykładowy potok dla IaC. W pierwszej sekcji dowiedzieliśmy się, jak zintegrować wiersze poleceń Packera z potokami. W drugiej części kontynuowaliśmy automatyzację IaC za pomocą innego potoku, który uruchamiał Terraform w celu udostępnienia maszyny wirtualnej platformy Azure, i zainstalowaliśmy nginx na tej maszynie wirtualnej za pomocą Ansible.

Cały kod potoku YAML użyty w tym rozdziale miał zastosowanie do GitHuba i Azure Pipelines, a proces jest dokładnie taki sam dla innych narzędzi CI/CD, takich jak Jenkins lub GitLab CI.

W następnym rozdziale dowiemy się, jak zbudować i uruchomić kontener za pomocą Dockera.

Pytania

1. Jak nazywa się narzędzie użyte w tym rozdziale?
2. Jaka jest kolejność udostępniania IaC za pomocą narzędzi Terraform, Ansible i Packer?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o potokach IaC, oto kilka przykładów:

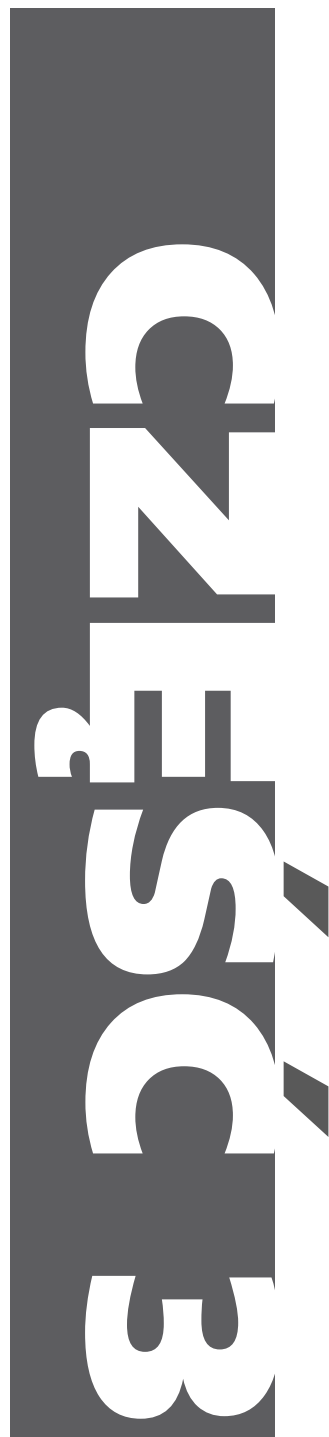
- Laboratorium DevOps oparte na Terraform na platformie Azure DevOps — <https://azuredevopslabs.com/labs/vstsextend/terraform/>.
- Laboratorium DevOps opisujące rozwiązania Ansible na platformie Azure DevOps — <https://www.azuredevopslabs.com/labs/vstsextend/ansible/>.

Konteneryzowane mikroustługi wykorzystujące platformę Docker i Kubernetes

W tej części przedstawimy podstawowe zastosowania Dockera i powiemy, jak tworzyć i uruchamiać kontenery za pomocą pliku Dockerfile. Następnie zbadamy rolę Kubernetesa i sposoby wdrażania bardziej złożonych aplikacji na Kubernetesie.

Część ta składa się z następujących rozdziałów:

- Rozdział 9., „Konteneryzacja aplikacji za pomocą Dockera”.
- Rozdział 10., „Efektywne zarządzanie kontenerami za pomocą Kubernetesa”.



Konteneryzacja aplikacji za pomocą Dockera

Rozdział

9

W ciągu ostatnich kilku lat w portalach społecznościowych i podczas wydarzeń branżowych omawiano przede wszystkim jedną technologię — Docker.

Docker to narzędzie służące do konteneryzacji, które stało się otwartoźródłowe w 2013 r. Umożliwia odizolowanie aplikacji od systemu hosta, dzięki czemu aplikacja staje się przenośna, a kod testowany na stacji roboczej programisty może być wdrożony w środowisku produkcyjnym bez obaw o zależności środowiska wykonawczego. W tym rozdziale porozmawiamy trochę o konteneryzacji aplikacji.

Kontener to system, który skupia w sobie aplikację i jej zależności. W przeciwieństwie do **maszyny wirtualnej (VM)** kontener zawiera tylko lekki **system operacyjny (OS)** i elementy wymagane dla systemu operacyjnego, takie jak biblioteki systemowe, pliki binarne i zależności kodu.

Aby dowiedzieć się więcej o różnicach między maszynami wirtualnymi a kontenerami i o tym, dlaczego w przyszłości kontenery zastąpią maszyny wirtualne, proponuję przeczytać ten artykuł na blogu: <https://blog.docker.com/2018/08/containers-replacing-virtual-machines/>.

Główna różnica między maszynami wirtualnymi a kontenerami polega na tym, że każda maszyna wirtualna hostowana na hipernadzorcy zawiera kompletny system operacyjny, a zatem jest całkowicie niezależna od systemu operacyjnego znajdującego się na hipernadzorcy.

Kontenery nie zawierają jednak kompletnego systemu operacyjnego — tylko kilka plików binarnych — ale są zależne od systemu operacyjnego i korzystają z jego zasobów (**jednostki centralnej — CPU, pamięci o dostępie swobodnym — RAM** i sieci).

W tym rozdziale dowiemy się, jak zainstalować Dockera na różnych platformach, jak utworzyć obraz Dockera i jak go zarejestrować w Docker Hubie. Następnie omówimy przykład potoku **ciągłej integracji/ciągłego wdrażania (CI/CD)**, który wdraża obraz

platformy Docker w **Azure Container Instances (ACI)**. Ponadto pokażemy, jak używać Dockera do uruchamiania narzędzi z **interfejsami wiersza poleceń (CLI)**.

Na koniec poznamy również podstawowe pojęcia dotyczące **Docker Compose** i sposobu wdrażania kontenerów Docker Compose w ACI.

W tym rozdziale omówiono następujące tematy:

- instalowanie Dockera,
- tworzenie pliku Dockerfile,
- budowanie i uruchamianie kontenera na komputerze lokalnym,
- wysyłanie obrazu do Docker Huba,
- wysyłanie obrazu Dockera do rejestru prywatnego (ACR),
- wdrażanie kontenera do ACI za pomocą potoku CI/CD,
- korzystanie z Dockera przy użyciu narzędzi wiersza poleceń,
- pierwsze kroki z Docker Compose,
- wdrażanie kontenerów Docker Compose w ACI.

Wymagania techniczne

Ten rozdział ma następujące wymagania techniczne:

- Subskrypcja platformy Azure. Możesz założyć darmowe konto tutaj: <https://azure.microsoft.com/en-us/free/>.
- W przypadku niektórych poleceń platformy Azure użyjemy interfejsu Azure CLI. Zapoznaj się z dokumentacją tutaj: <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli>.
- W ostatniej części tego rozdziału, w sekcji „Tworzenie potoku CI/CD dla kontenera”, omówimy Terraform i potok CI/CD, które zostały wyjaśnione w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”, i w rozdziale 7., „Ciągła integracja i ciągłe wdrażanie”.

Cały kod źródłowy skryptów zawartych w tym rozdziale jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP09>.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/3t1Jov8>.

Instalowanie Dockera

Demon Dockera jest darmowy i bardzo dobrze nadaje się dla programistów i małych zespołów, dlatego skorzystamy z niego w tej książce.

Docker to wieloplatformowe narzędzie, które można zainstalować w systemach Windows, Linux lub macOS, a także jest dostępne natywnie u niektórych dostawców chmury, takich jak **Amazon Web Services (AWS)** i Azure.

Do działania Docker potrzebuje następujących elementów:

- **Klient Dockera** — umożliwia wykonywanie różnych operacji w wierszu poleceń.
- **Demon Dockera** — silnik platformy Docker.
- **Rejestr Dockera** — rejestr publiczny (Docker Hub) lub prywatny rejestr obrazów Dockera.

Przed zainstalowaniem Dockera najpierw utworzymy konto w Docker Hubie.

Rejestracja w Docker Hubie

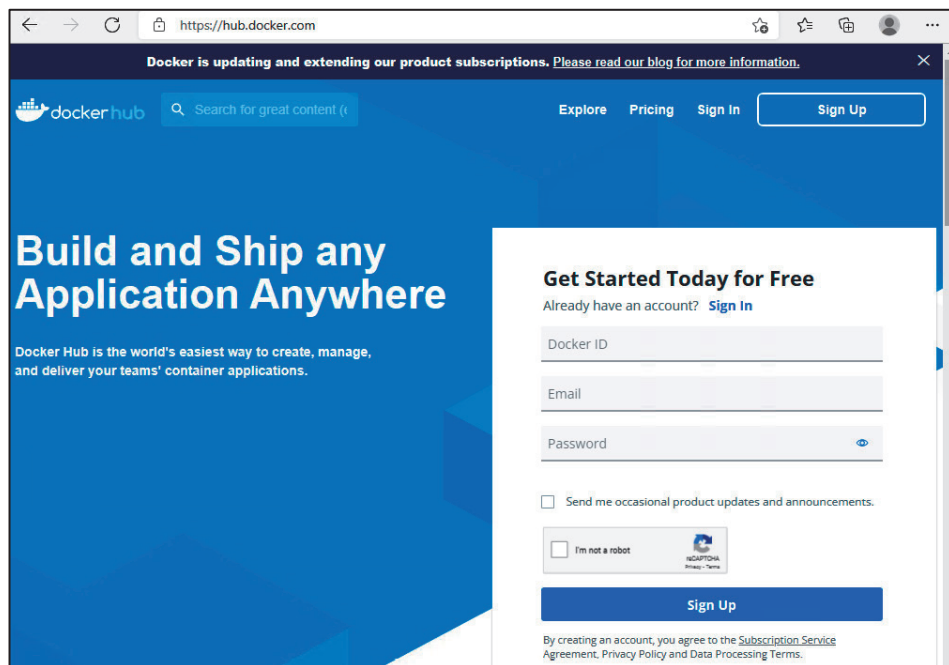
Docker Hub to przestrzeń publiczna zwana **rejestrem**, zawierająca ponad 2 mln publicznych obrazów platformy Docker, które zostały zdeponowane przez firmy, społeczności, a nawet użytkowników indywidualnych.

Aby zarejestrować się w Docker Hubie i wyświetlić publiczne obrazy platformy Docker, wykonaj następujące czynności:

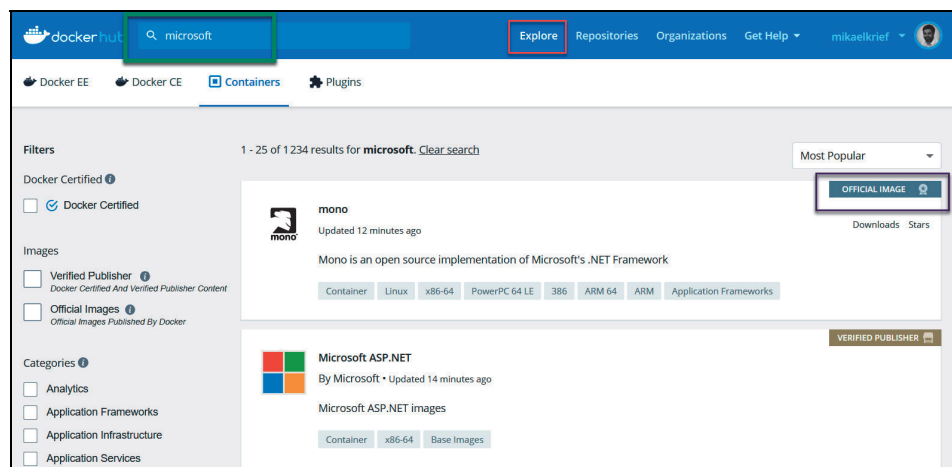
1. Przejdź pod adres <https://hub.docker.com/>, gdzie zobaczysz ekran jak na rysunku 9.1.
2. Wypełnij formularz **unikalnym identyfikatorem (ID)**, adresem e-mail i hasłem. Następnie kliknij przycisk *Sign Up*.
3. Po utworzeniu konta możesz zalogować się do witryny, a to konto umożliwi Ci przesyłanie różnych obrazów i pobranie **Docker Desktop**.
4. Aby wyświetlić i eksplorować obrazy dostępne w Docker Hubie, przejdź do sekcji *Explore*, jak pokazano na rysunku 9.2.

Lista obrazów Dockera jest wyświetlana z filtrem wyszukiwania, którego można użyć do znajdowania oficjalnych obrazów lub obrazów od zweryfikowanych wydawców, a także obrazów certyfikowanych przez Dockera.

Po utworzeniu konta w Docker Hubie przyjrzymy się instalacji Dockera w systemie Windows.



Rysunek 9.1. Strona logowania Docker Huba



Rysunek 9.2. Sekcja Explore w Docker Hubie

Instalacja Dockera

Omówimy teraz szczegółowo instalację Dockera w systemie Windows.

Przed zainstalowaniem Docker Desktop w systemie Windows lub macOS musimy sprawdzić wszystkie opcje licencji. Aby uzyskać więcej informacji na temat licencjonowania platformy Docker Desktop, przeczytaj stronę z cennikiem (<https://www.docker.com/pricing>) i stronę z często zadawanymi pytaniami (FAQ) (<https://www.docker.com/pricing/faq>).

Aby zainstalować Docker Desktop na komputerze z systemem Windows, należy najpierw sprawdzić wymagania sprzętowe, które są następujące:

- Windows 10/11 64-bitowy z co najmniej 4 gigabajtami (GB) pamięci RAM.
- Włączony **podsystem Windows dla systemu Linux 2** (ang. *Windows Subsystem for Linux 2* — **WSL 2**) lub funkcja Hyper-V. W razie jakichkolwiek problemów możesz zapoznać się z tą dokumentacją: <https://docs.docker.com/docker-for-windows/troubleshoot/#virtualization-must-be-enabled>.

Uwaga

Aby uzyskać więcej informacji o WSL, przeczytaj dokumentację tutaj: <https://docs.microsoft.com/en-us/windows/wsl/install>.

Więcej szczegółów na temat wymagań platformy Docker Desktop podano tutaj: <https://docs.docker.com/desktop/windows/install/>.

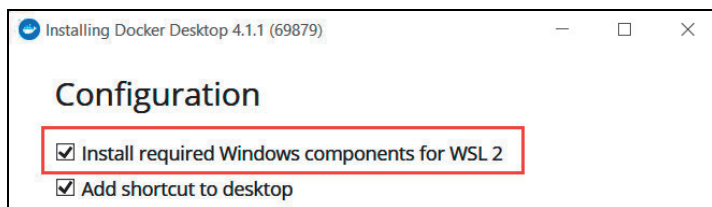
Aby zainstalować Docker Desktop, który jest tym samym plikiem binarnym co instalator Dockera dla systemów Windows i macOS, wykonaj następujące kroki:

1. Najpierw pobierz Docker Desktop, klikając przycisk *Docker Desktop for Windows* na stronie dokumentacji instalacji pod adresem <https://docs.docker.com/desktop/windows/install/>, jak pokazano na poniższym zrzucie ekranu:



Rysunek 9.3. Link do pobrania Docker Desktop

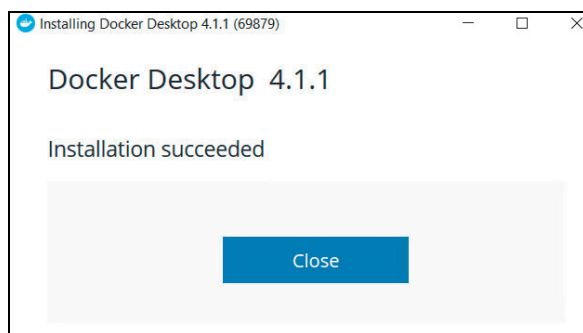
2. Po pobraniu kliknij pobrany plik wykonywalny (EXE).
3. Następnie wykonaj jeden krok konfiguracji, instalację wymaganych komponentów dla backendu WSL 2, jak pokazano na poniższym zrzucie ekranu:



Rysunek 9.4. Konfiguracja Docker Desktop

W naszym przypadku zaznaczymy tę opcję, aby zainstalować komponenty Windowsa korzystające z WSL 2 jako backendu.

4. Po zakończeniu instalacji otrzymamy komunikat potwierdzający i przycisk kończący instalację, jak pokazano na poniższym zrzucie ekranu:

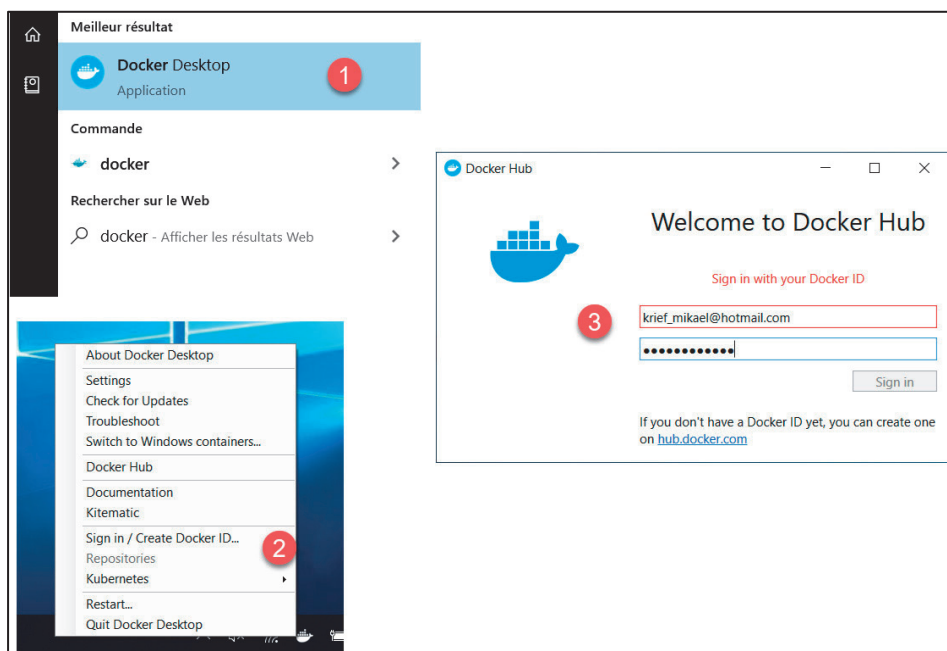


Rysunek 9.5. Koniec instalacji platformy Docker Desktop

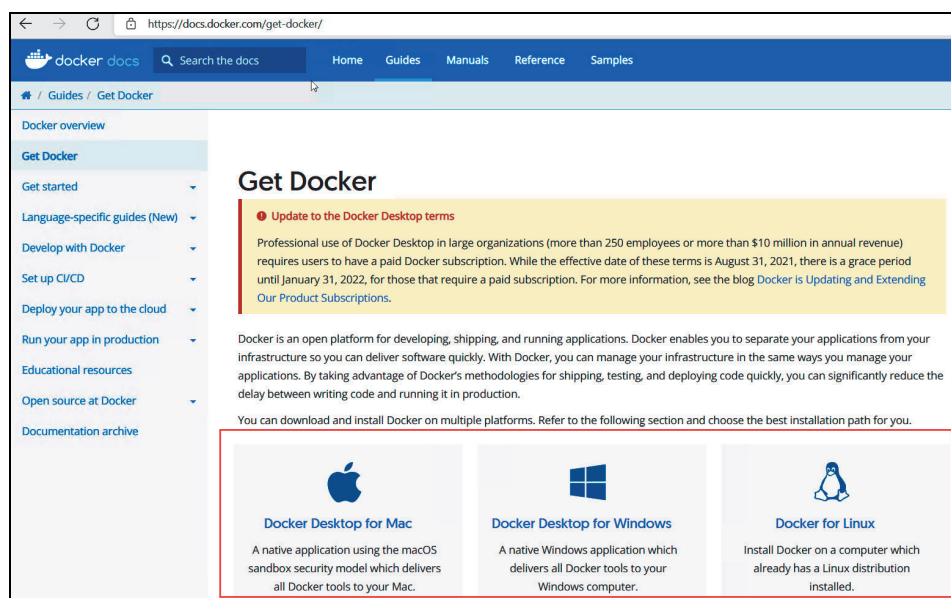
5. Na koniec, aby uruchomić Dockera, uruchom program Docker Desktop. Na pasku powiadomień pojawi się ikona wskazująca, że Docker się uruchamia. Następnie zostaniesz poproszony o zalogowanie się do Docker Huba. Poszczególne etapy uruchamiania Docker Desktop są pokazane na rysunku 9.6.

Otóż to! Zainstalowaliśmy i uruchomiliśmy Dockera w systemie Windows.

Aby zainstalować Dockera w innym systemie operacyjnym, możesz przeczytać dokumentację dla każdego z nich pod adresem <https://docs.docker.com/get-docker/>. Następnie możesz wybrać żądany system operacyjny na tej stronie, jak pokazano na rysunku 9.7.



Rysunek 9.6. Logowanie do Docker Huba z poziomu Docker Desktop



Rysunek 9.7. Dokumentacja instalacji Dockera

Aby sprawdzić instalację Dockera, otwórz okno terminala (będzie również działać w terminalu Windowsa PowerShell) i wykonaj następujące polecenie:

```
docker --help
```

Powinieneś być w stanie zobaczyć coś takiego:

```
PS C:\Users\mkrief> docker --help
Usage:
  docker [flags]
  docker [command]

Available Commands:
  compose      Docker Compose
  context      Manage contexts
  ecs
  exec         Run a command in a running container
  help        Help about any command
  inspect      Inspect containers
  kill        Kill one or more running containers
  login        Log in to a Docker registry or cloud backend
  logout       Log out from a Docker registry or cloud backend
  logs        Fetch the logs of a container
  prune       prune existing resources in current context
  ps          List containers
  rm          Remove containers
  run         Run a container
  secret      Manages secrets
  serve       Start an api server
  start       Start one or more stopped containers
  stop        Stop one or more running containers
  version     Show the Docker version information
  volume      Manages volumes
```

Rysunek 9.8. Polecenie docker --help

Jak widać na poprzednim zrzucie ekranu, polecenie wyświetla różne operacje dostępne w narzędziu klienta Dockera.

Zanim przyjrzymy się szczegółowo wykonywaniu poleceń przez Dockera, ważne jest, aby zapoznać się z jego budową.

Przegląd elementów Dockera

Przed wykonaniem poleceń Dockera omówimy niektóre z podstawowych elementów Dockera, którymi są pliki *Dockerfile*, *kontenery* i *woluminy*.

Przede wszystkim ważne jest, aby wiedzieć, że **obraz Dockera** jest jego podstawowym elementem i składa się z dokumentu tekstowego zwanego plikiem *Dockerfile*, zawierającego pliki binarne i pliki aplikacji, które chcemy konteneryzować.

Rejestr platformy Docker to scentralizowany system przechowywania udostępnianych obrazów platformy Docker. Ten rejestr może być publiczny — jak w przypadku Docker Huba — lub prywatny, np. **Azure Container Registry (ACR)** lub JFrog Artifactory.

Kontener to instancja wykonywana z obrazu platformy Docker. Możliwe jest posiadanie kilku wystąpień tego samego obrazu w kontenerze, który aplikacja będzie uruchamiać.

Wolumin to przestrzeń magazynowa, która jest fizycznie zlokalizowana w systemie operacyjnym hosta (czyli poza kontenerem) i w razie potrzeby może być współużytkowana przez wiele kontenerów. Przestrzeń ta pozwoli na przechowywanie trwałych elementów, takich jak pliki czy bazy danych.

Aby manipulować tymi elementami, użyjemy poleceń, które zostaną omówione w dalszej części tego rozdziału.

W tej sekcji omówiliśmy Docker Huba i różne kroki tworzenia konta. Następnie przyrzekliśmy się krokom instalacji Docker Desktop lokalnie i zakończyliśmy przeglądem elementów Dockera.

Rozpocniemy teraz pracę z Dockerem, a pierwszą operacją, której się przyjrzymy, będzie utworzenie obrazu Dockera z pliku *Dockerfile*.

Tworzenie pliku *Dockerfile*

Podstawowym elementem Dockera jest plik zwany plikiem *Dockerfile*, który zawiera instrukcje krok po kroku dotyczące budowania obrazu Dockera.

Aby zrozumieć, jak utworzyć plik *Dockerfile*, przyjrzymy się przykładowi, który pozwala nam zbudować obraz Dockera. Zawiera on serwer WWW Apache i aplikację internetową.

Zacznijmy od napisania pliku *Dockerfile*.

Tworzenie pliku *Dockerfile*

Aby napisać plik *Dockerfile*, najpierw utworzymy stronę w języku **HTML** (ang. *Hyper-Text Markup Language*), która będzie naszą aplikacją internetową. Tak więc utworzymy nowy katalog *appdocker*, a w nim stronę *index.html*, która zawiera przykładowy kod wyświetlający tekst powitalny w następujący sposób:

```
<html>
  <body>
    <h1>Witam w mojej nowej aplikacji</h1>
    To jest strona testowa dla mojego pliku Dockerfile.<br />
    Miłej zabawy...
  </body>
</html>
```

Następnie, w tym samym katalogu, tworzymy plik *Dockerfile* (bez rozszerzenia) z następującą zawartością, którą szczegółowo zaraz opiszemy:

```
FROM httpd:latest
COPY index.html /usr/local/apache2/htdocs/
```

Aby utworzyć plik *Dockerfile*, zacznij od instrukcji *FROM*. Wymagana instrukcja *FROM* definiuje obraz bazowy, którego użyjemy dla naszego obrazu Dockera — każdy obraz jest zbudowany z innego obrazu Dockera. Obraz bazowy można zapisać w Docker Hubie lub w innym rejestrze, takim jak JFrog Artifactory, Nexus Repository lub ACR.

W naszym przykładzie używamy obrazu *httpd* Apache oznaczonego jako najnowsza wersja, https://hub.docker.com/_/httpd/, i instrukcji *FROM httpd:latest*.

Następnie używamy instrukcji *COPY* do wykonania procesu konstrukcji obrazu. Docker kopiuje lokalny plik *index.html*, który właśnie utworzyliśmy, do katalogu obrazu */usr/local/apache2/htdocs/*.

Uwaga

Kod źródłowy pliku *Dockerfile* i stronę HTML można znaleźć tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP09/appdocker>.

Właśnie przyjrzelśmy się instrukcjom *FROM* i *COPY* pliku *Dockerfile*, ale są też inne instrukcje, które omówimy w następnej sekcji.

Przegląd instrukcji *Dockerfile*

Wspomnieliśmy wcześniej, że plik *Dockerfile* składa się z instrukcji, a także przyjrzelśmy się konkretnemu przykładowi z instrukcjami *FROM* i *COPY*. Istnieją inne instrukcje, które pozwolą Ci zbudować obraz Dockera. Oto przegląd głównych instrukcji, których można użyć w tym celu:

- *FROM* — służy do zdefiniowania obrazu bazowego dla naszego obrazu, jak pokazano w przykładzie szczegółowo opisanym w poprzedniej sekcji „Tworzenie pliku *Dockerfile*”.

- **COPY** i **ADD** — służą do kopiowania jednego lub więcej lokalnych plików do obrazu. Instrukcja **ADD** obsługuje dwie dodatkowe funkcje: odwoływanie się do adresu **URL** (ang. *Uniform Resource Locator*) i wyodrębnianie skompresowanych plików.

Uwaga

Więcej szczegółów na temat różnic między **COPY** a **ADD** można znaleźć w tym artykule: <https://nickjanetakis.com/blog/docker-tip-2-the-difference-between-copy-and-add-in-a-dockerfile>.

- **RUN** i **CMD** — te instrukcje jako parametr przyjmują polecenie, które zostanie wykonane podczas tworzenia obrazu. Instrukcja **RUN** tworzy warstwę, dzięki czemu obraz może być buforowany i wersjonowany. Instrukcja **CMD** definiuje domyślne polecenie, które ma być wykonane podczas wywoływania uruchomienia obrazu. **CMD** może zostać zastąpiona w czasie wykonywania dodatkowym parametrem.

Możesz skorzystać z następującego przykładu instrukcji **RUN** w pliku **Dockerfile**, aby wykonać polecenie **apt-get**:

RUN apt-get update

Za pomocą poprzedniej instrukcji aktualizujemy pakiety **apt**, które są już obecne w obrazie, i tworzymy warstwę. Możemy również użyć instrukcji **CMD** w poniższym przykładzie, która wyświetli komunikat **docker**:

CMD "echo docker"

- **ENV** — ta instrukcja umożliwia tworzenie instancji zmiennych środowiskowych, których można użyć do zbudowania obrazu. Te zmienne środowiskowe będą obowiązywać przez cały okres eksploatacji kontenera w następujący sposób:

ENV myvar=mykey

Poprzednie polecenie ustawia zmienną środowiskową **myvar** z wartością **mykey** dla kontenera.

- **WORKDIR** — ta instrukcja podaje katalog dla wykonania kontenera w następujący sposób:

WORKDIR usr/local/apache2

To był przegląd instrukcji **Dockerfile**. Istnieją inne powszechnie używane instrukcje, takie jak **EXPOSE**, **ENTRYPOINT** i **VOLUME**, które można znaleźć w oficjalnej dokumentacji pod adresem <https://docs.docker.com/engine/reference/builder/>.

Właśnie zaobserwowaliśmy, że tworzenie pliku **Dockerfile** odbywa się za pomocą różnych instrukcji, takich jak **FROM**, **COPY** i **RUN**, które są używane do tworzenia obrazu

Dockera. Przyjrzyjmy się teraz, jak uruchomić Dockera w celu skompilowania obrazu z pliku Dockerfile i jak uruchomić ten obraz lokalnie, aby go przetestować.

Budowanie i uruchamianie kontenera na komputerze lokalnym

Do tej pory w rozdziale omówiliśmy elementy Dockera i przyjrzelśmy się przykładowemu plikowi Dockerfile, który służy do konteneryzacji aplikacji internetowej. Teraz mamy już wszystkie elementy potrzebne do uruchomienia Dockera.

Wykonanie Dockera odbywa się za pomocą różnych operacji, jak opisano tutaj:

- Tworzenie obrazu Dockera z pliku Dockerfile.
- Tworzenie nowego kontenera lokalnie z tego obrazu.
- Testowanie naszej lokalnie konteneryzowanej aplikacji.

Przyjrzyjmy się szczegółowo każdej operacji.

Tworzenie obrazu Dockera

Zbudujemy obraz Dockera z naszego wcześniej utworzonego pliku Dockerfile, który zawiera następujące instrukcje:

```
FROM httpd:latest
COPY index.html /usr/local/apache2/htdocs/
```

Przejdziemy do terminala, aby przejść do katalogu zawierającego plik Dockerfile, a następnie wykonamy polecenie `docker build` z następującą składnią:

```
docker build -t demobook:v1 .
```

Argument `-t` wskazuje nazwę obrazu i jego tag. W naszym przykładzie nazywamy obraz *demobook*, a dodany przez nas tag to *v1*.

Kropka `.` na końcu polecenia określa, że będziemy używać plików z bieżącego katalogu. Poniższy zrzut ekranu pokazuje wykonanie tego polecenia.

W powyższym wykonaniu możemy zobaczyć trzy kroki budowania obrazów Dockera:

1. Docker pobiera zdefiniowany obraz bazowy.
2. Docker kopiuje plik *index.html* do obrazu.
3. Docker tworzy i oznacza obraz tagiem.

```

PS C:\Learning-DevOps-Second-Edition\CHAP09\appdocker> docker build -t demobook:v1 .
[*] Building 40.55 (7/7) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 988
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/httpd:latest
=> [internal] load build context
=> => transferring context: 191B
=> [1/2] FROM docker.io/library/httpd:latest@sha256:f78876d78442771406d7245b8d3425e8b0a86891c79811af94fb2e12af0fadeb
=> => resolve docker.io/library/httpd:latest@sha256:f78876d78442771406d7245b8d3425e8b0a86891c79811af94fb2e12af0fadeb
=> => sha256:4482565671564bb0b369534aa4040f113c5fe4eee6aabe2da04d144f663eed4 913.73kB / 913.73kB
=> => sha256:f78876d78442771406d7245b8d3425e8b0a86891c79811af94fb2e12af0fadeb 1.86kB / 1.86kB
=> => sha256:73c9b78280a69385893a9e3519ef57723702ad3e82e45f10744b4d88f406e 1.36kB / 1.36kB
=> => sha256:11224fc88faaf5c19959f08353c1366d3004ced1978cb9c5f32c73d0c139532 0.78kB / 0.78kB
=> => sha256:7d63c13d9b9b6ec5f05a2b07daadaca9c610d01102a662ae9b1d082185f1ffa 31.36MB / 31.36MB
=> => sha256:ca52f3eeea665ce537eeec1840e21d7d024ab70fb555a609cd748e710779db9e0 176B / 176B
=> => sha256:21d69ac90caf9d24441bfa860ed24c4bf82e421f95d9a2abf957c9b111978c03 24.11MB / 24.11MB
=> => sha256:462e88bc307455be86d7af71d19421a240793468d7ab879e36c86b54d8e0ec7d 296B / 296B
=> => extracting sha256:7d63c13d9b9b6ec5f05a2b07daadaca9c610d01102a662ae9b1d082185f1ffa
=> => extracting sha256:ca52f3eeea665ce537eeec1840e21d7d024ab70fb555a609cd748e710779db9e0
=> => extracting sha256:4482565671564bb0b369534aa4040f113c5fe4eee6aabe2da04d144f663eed4
=> => extracting sha256:21d69ac90caf9d24441bfa860ed24c4bf82e421f95d9a2abf957c9b111978c03
=> => extracting sha256:462e88bc307455be86d7af71d19421a240793468d7ab879e36c86b54d8e0ec7d
=> [2/2] COPY index.html /usr/local/apache2/htdocs/
=> => exporting to image
=> => exporting layers
=> => writing image sha256:9a3862a66c65d0a431b70b464d8deedff1a73927fad7475ac520f876644c3301
=> => naming to docker.io/library/demobook:v1

```

Rysunek 9.9. Polecenie docker build

Po wykonaniu polecenia `docker build` pobierany jest obraz podstawowy wskazany w pliku `Dockerfile` z Docker Huba, a następnie Docker wykonuje różne instrukcje wymienione w pliku `Dockerfile`.

Uwaga

Zwróć uwagę, że jeśli podczas pierwszego wykonania polecenia `docker build` pojawi się błąd `Get https://registry-1.docker.io/v2/library/httpd/manifests/latest: unauthorized: incorrect username or password`, należy wykonać polecenie `docker logout`. Następnie uruchom ponownie polecenie `docker build`, jak wskazano w tym artykule: <https://medium.com/@blacksourcel/fix-dockererror-unauthorized-incorrect-username-or-password-in-docker-f80c45951b6b>.

Pod koniec wykonania otrzymujemy przechowywany lokalnie obraz *demobook*.

Uwaga

Obraz platformy Docker jest przechowywany w lokalnym systemie folderów w zależności od systemu operacyjnego. Więcej informacji na temat lokalizacji obrazów platformy Docker można znaleźć w tym artykule: <http://www.scmgalaxy.com/tutorials/location-of-dockers-images-in-all-operating-systems/>.

Możemy również sprawdzić, czy obraz został pomyślnie utworzony, wykonując następujące polecenie Dockera:

```
docker image
```

Oto wynik wykonania poprzedniego polecenia:

```
PS \Learning-DevOps-Second-Edition\CHAP09\appdocker> docker images
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|-----------------------------|-------------------|--------------|---------------|-------|
| demobook | v1 | 9a3862a66c65 | 5 minutes ago | 143MB |
| gcr.io/k8s-minikube/kicbase | v0.0.12-snapshot3 | 25ac91b9c8d7 | 14 months ago | 952MB |

Rysunek 9.10. Polecenie docker image

To polecenie wyświetla listę obrazów Dockera na lokalnym komputerze. Możemy zobaczyć właśnie utworzony obraz *demobook*. Tak więc następnym razem, gdy obraz zostanie zbudowany, nie będziemy musieli ponownie pobierać obrazu httpd.

Teraz, po utworzeniu obrazu Dockera naszej aplikacji, utworzymy nowy kontener tego obrazu.

Tworzenie nowego kontenera obrazu

Aby utworzyć nowy kontener naszego obrazu Dockera, wykonamy w naszym terminalu polecenie `docker run` o następującej składni:

```
docker run -d --name demoapp -p 8080:80 demobook:v1
```

Parametr `-d` wskazuje, że kontener będzie działał w tle. W parametrze `--name` określamy nazwę kontenera. W parametrze `-p` wskazujemy żądaną translację portu. W naszym przykładzie oznaczałoby to, że port 80 kontenera zostanie przekierowany na port 8080 na naszym lokalnym komputerze. I wreszcie ostatnim parametrem polecenia jest nazwa obrazu i jego tag.

Wykonanie tego polecenia pokazano na poniższym zrzucie ekranu:

```
PS \Learning-DevOps-Second-Edition\CHAP09\appdocker> docker run -d --name demoapp -p 8080:80 demobook:v1
6cce2099b174cbe29fcd408d044cd5b2ebbf50cedfb1a5e2984e346ebee7e1a
```

Rysunek 9.11. Polecenie docker run

Po zakończeniu wykonywania to polecenie wyświetla identyfikator kontenera, a kontener kontynuuje działanie w tle. Możliwe jest również wyświetlenie listy kontenerów działających na komputerze lokalnym poprzez wykonanie następującego polecenia:

```
docker ps
```

Poniższy zrzut ekranu przedstawia wykonanie polecenia, gdzie zostaje zwrócony nasz kontener:

```
PS C:\Users\mkrief> docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|-------------|--------------------|-------------|-------------------|----------------------|---------|
| 6cce2099b174 | demobook:v1 | "httpd-foreground" | 2 weeks ago | Up About a minute | 0.0.0.0:8080->80/tcp | demoapp |

Rysunek 9.12. Polecenie docker ps

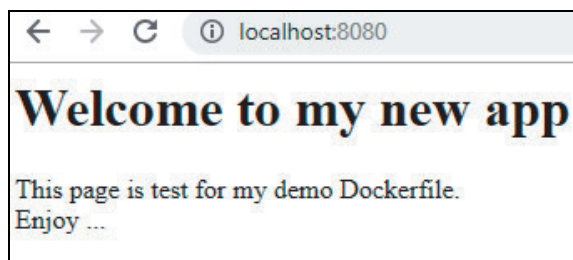
Po wykonaniu każdego kontenera dostajemy jego identyfikator, powiązany z nim obraz, jego nazwę, polecenie wykonania i informacje o porcie.

Zbudowaliśmy obraz Dockera i lokalnie utworzyliśmy nowy kontener dla tego obrazu. Zobaczmy teraz, jak uruchomić aplikację internetową znajdującą się w lokalnym kontenerze.

Lokalne testowanie kontenera

Wszystko, co działa w kontenerze, pozostaje w nim — to zasada izolacji kontenera. Jednak dzięki translacji portów, którą zrobiliśmy wcześniej, możesz przetestować swój kontener na komputerze lokalnym za pomocą polecenia `run`.

Aby to zrobić, otwórz przeglądarkę internetową i wpisz adres `http://localhost:8080`. Wartość 8080 reprezentuje port wskazany w poleceniu. Powinieneś być w stanie zobaczyć następujący wynik:



Rysunek 9.13. Uruchomiona aplikacja Docker

Widzimy wyświetlaną zawartość naszej strony `index.html`.

W tej sekcji przyjrzelśmy się różnym poleceniom Dockera, których można użyć do zbudowania obrazu Dockera. Potem utworzyliśmy instancję nowego kontenera z tego obrazu, a na koniec przetestowaliśmy go lokalnie.

W następnej sekcji zobaczymy, jak opublikować obraz Dockera w Docker Hubie.

Wysyłanie obrazu do Docker Huba

Celem utworzenia obrazu Dockera zawierającego aplikację jest możliwość używania go na serwerach z Dockerem i hostujących aplikacje firmy, tak jak w przypadku maszyny wirtualnej.

Aby obraz mógł zostać pobrany na inny komputer, musi zostać zapisany w rejestrze obrazów Dockera. Jak już wspomniano w tym rozdziale, istnieje kilka rejestrów Dockera, które można zainstalować lokalnie, tak jak w przypadku JFrog Artifactory i Nexus Repository.

Jeśli chcesz utworzyć obraz publiczny, możesz go wysłać (lub przekazać) do Docker Huba, który jest publicznym (i niekiedy bezpłatnym, w zależności od licencji) rejestrem platformy Docker. Zobaczmy teraz, jak przesłać obraz, który utworzyliśmy w poprzedniej sekcji, do Docker Huba. Aby to zrobić, musisz mieć konto w Docker Hubie, które utworzyłeś przed instalacją Docker Desktop.

Aby przekazać obraz Dockera do Docker Huba, wykonaj następujące czynności:

1. Zaloguj się do Docker Huba za pomocą następującego polecenia:

```
docker login -u <Twój login w Docker Hubie>
```

Podczas wykonywania polecenia zostaniesz poproszony o podanie hasła Docker Huba i wskazanie połączenia z rejestrem Dockera, jak pokazano na poniższym zrzucie ekranu:

```
PS > \Learning_DevOps\CHAP07\appdocker> docker login -u mikaelkr
Password:
Login Succeeded
```

Rysunek 9.14. Polecenie docker login

2. Pobierz identyfikator obrazu. Następny krok polega na pobraniu identyfikatora utworzonego obrazu. Aby to zrobić, wykonamy polecenie `docker images`, by wyświetlić listę obrazów z ich ID.

```
PS > \Learning-DevOps-Second-Edition\CHAP09\appdocker> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
demobook             v1                 9a3862a66c65       17 hours ago       143MB
gcr.io/k8s-minikube/kicbase v0.0.12-snapshot3 25ac91b9c8d7       14 months ago     952MB
```

Rysunek 9.15. Lista obrazów Dockera

3. Tagowanie obrazu dla Docker Huba. Za pomocą identyfikatora pobranego obrazu oznaczmy teraz obraz dla Docker Huba. W tym celu wykonywane jest następujące polecenie:

```
docker tag <identyfikator obrazu> <login do Docker Huba>/demobook:v1
```

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia na utworzonym obrazie:

```
PS > \Learning_DevOps\CHAP07\appdocker> docker tag (a121d88f6e18) (mikaelkr1eF) demobook:v1
```

Rysunek 9.16. docker tag

4. **Wysłanie obrazu Dockera do Docker Huba.** Po otagowaniu obrazu ostatnim krokiem jest przekazanie go do Docker Huba.

W tym celu wykonamy następujące polecenie:

```
docker push docker.io/<login do Docker Huba>/demobook:v1
```

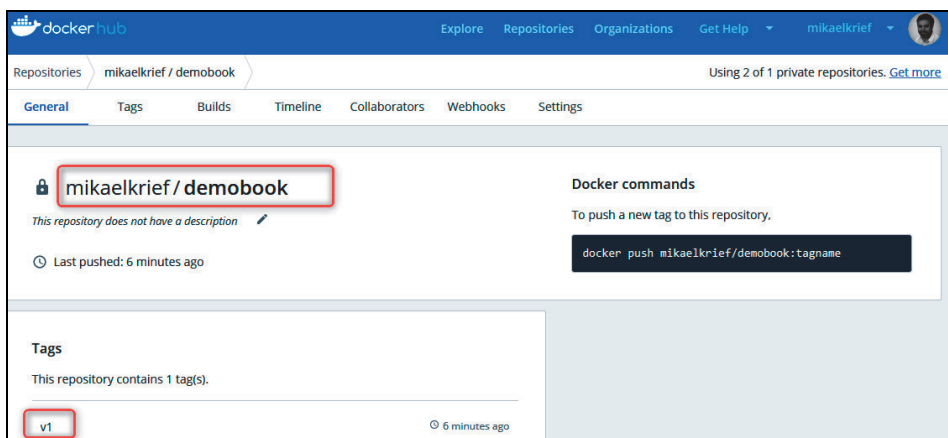
Poniższy zrzut ekranu pokazuje wykonanie poprzedniego polecenia:

```
PS > .\Learning-DevOps-Second-Edition\CHAP09\appdocker> docker push docker.io/mikaelkrief/demobook:v1
The push refers to repository [docker.io/mikaelkrief/demobook]
f083a28f9cfa: Pushed
4dcdec0b7a0e: Mounted from library/httpd
c86537ee54f9: Mounted from library/httpd
ecd2b49ef243: Mounted from library/httpd
7511c367f47a: Mounted from library/httpd
e8b689711f21: Mounted from library/httpd
v1: digest: sha256:380d9e0b2beb20c495b496c7047bd3d808f048307a5b5c84cc1e1de3fe119e79 size: 1572
```

Rysunek 9.17. docker push

Jak widać, obraz jest przesyłany do Docker Huba.

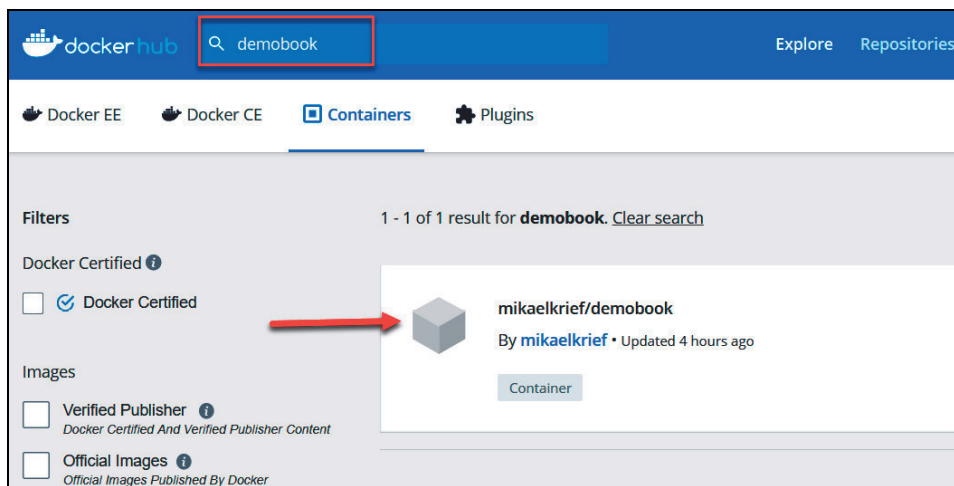
Aby w Docker Hubie wyświetlić wysłany obraz, łączymy się z portalem internetowym Docker Huba pod adresem <https://hub.docker.com/>. Widzimy, że obraz jest obecny, jak pokazano na poniższym zrzucie ekranu:



Rysunek 9.18. Przesłany otagowany obraz w Docker Hubie

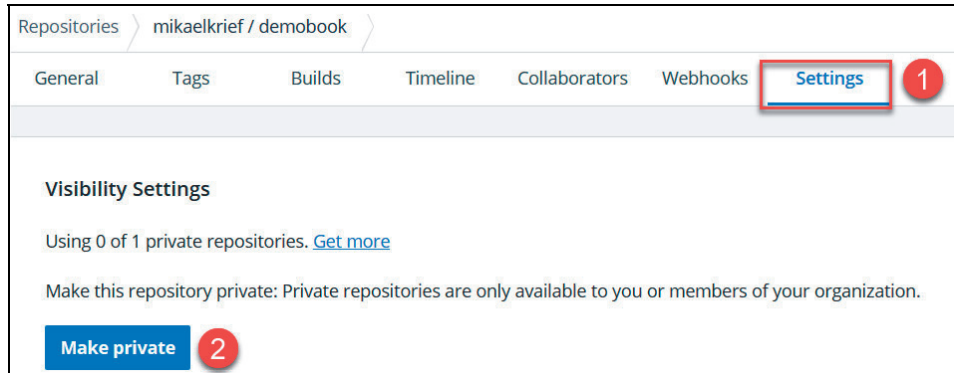
Domyślnie obraz przekazany do Docker Huba jest udostępniony publicznie — każdy może go wyświetlić w eksploratorze i z niego korzystać.

Możemy uzyskać dostęp do tego obrazu w wyszukiwarce Docker Huba, jak pokazano na poniższym zrzucie ekranu:



Rysunek 9.19. Znajdowanie obrazu w Docker Hubie

Aby uczynić ten obraz prywatnym — co oznacza, że tylko Ty jesteś uwierzytelniony do korzystania z niego — musisz przejść do zakładki *Settings* dla obrazu i kliknąć przycisk *Make private*, jak pokazano na poniższym zrzucie ekranu:



Rysunek 9.20. Ustawianie dostępu prywatnego dla obrazu Dockera

W tej sekcji przyjrzelśmy się etapom i poleceniom platformy Docker służącym do logowania się do usługi Docker Hub za pomocą wiersza poleceń, a następnie poleceniom tag i push służącym do przesyłania obrazu platformy Docker do usługi Docker Hub.

W następnej sekcji zobaczymy, jak wysłać obraz Dockera do prywatnego rejestru Dockera przy użyciu przykładowej instancji ACR.

Wysyłanie obrazu Dockera do rejestru prywatnego (ACR)

W poprzedniej sekcji dowiedzieliśmy się, jak przesłać obraz Dockera do Docker Huba, który jest rejestrem publicznym. Teraz dowiemy się, jak przesłać obraz Dockera do prywatnego rejestru.

Istnieje wiele rozwiązań lokalnych lub chmurowych, które umożliwiają korzystanie z prywatnego rejestru Dockera. Oto lista tych rozwiązań:

- Serwer rejestru platformy Docker — <https://docs.docker.com/registry/deploying/>.
- Artifactory firmy JFrog — <https://www.jfrog.com/confluence/display/JFROG/Docker+Registry>.
- Amazon Elastic Container Registry (ECR) — <https://aws.amazon.com/ecr/>.
- Rejestr kontenerów Google (ang. Google Container Registry — GCR) — <https://cloud.google.com/container-registry>.
- ACR — <https://azure.microsoft.com/en-us/services/container-registry/>.

W tej części przeanalizujemy użycie jednego z tych rozwiązań — ACR.

Aby wysłać obraz Dockera do ACR, wykonaj następujące kroki:

1. Przed wysłaniem obrazu Dockera utworzymy zasób ACR, korzystając z jednego z poniższych rozwiązań:
 - Interfejs wierszy poleceń platformy Azure — Azure CLI (<https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-azure-cli>). Skrypt `az cli`, który tworzy grupę zasobów i zasób ACR, jest pokazany tutaj:

```
az group create --name RG-ACR --location eastus
az acr create --resource-group RG-ACR --name acrdemo --sku Basic
```
 - PowerShell (<https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-powershell>). Skrypt PowerShell, który tworzy grupę zasobów i zasób ACR, jest pokazany tutaj:

```
New-AzResourceGroup -Name RG-ACR -Location EastUS
$registry = New-AzContainerRegistry -ResourceGroupName
"RG-ARC" -Name "acrdemo" -EnableAdminUser -Sku Basic
```
 - Portal platformy Azure (<https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-portal>).

W kolejnych krokach użyjemy zasobu ACR o nazwie demobookacr.

2. Następnie połączymy się z naszym kontem Azure, uruchamiając następującą komendę az cli:

```
az login
```

3. Łączymy się z utworzonym zasobem ACR (w kroku 1.) za pomocą polecenia az acr login, przekazując w argumencie --name nazwę zasobu ACR utworzonego w kroku 1. w następujący sposób:

```
az acr login --name demobookacr
```

Polecenie to połączy się w tle z rejestrem platformy Docker przy użyciu polecenia docker login.

4. Aby wysłać obraz Dockera do tego zasobu ACR, wykonamy kilka poleceń.

Pierwszym poleceniem jest utworzenie tagu do lokalnego obrazu, jak pokazano tutaj:

```
docker tag demobook:v1 demobookacr.azurecr.io/demobook:v1
```

Drugie polecenie polega na wysłaniu obrazu do zasobu ACR, jak pokazano tutaj:

```
docker push demobookacr.azurecr.io/demobook:v1
```

Te dwa polecenia są dokładnie takie same jak te, które dotyczyły tagowania i wysyłania obrazu Dockera do Docker Huba. Różnica polega na tym, że adres URL rejestru Dockera dla ACR to <acr name>.azurecr.io.

Poniższy zrzut ekranu przedstawia wykonanie tych poleceń:

```
PS C:\Users\mkrief> docker tag demobook:v1 demobookacr.azurecr.io/demobook:v1
PS C:\Users\mkrief> docker push demobookacr.azurecr.io/demobook:v1
The push refers to repository [demobookacr.azurecr.io/demobook]
f083a28f9cfa: Pushed
4dcdec0b7a0e: Pushed
c86537ee54f9: Pushed
ecd2b49ef243: Pushed
7511c367f47a: Pushed
e8b689711f21: Pushed
v1: digest: sha256:380d9e0b2beb20c495b496c7047bd3d808f048307a5b5c84cc1e1de3fe119e79 size: 1572
```

Rysunek 9.21. Wysyłanie obrazu Dockera do ACR

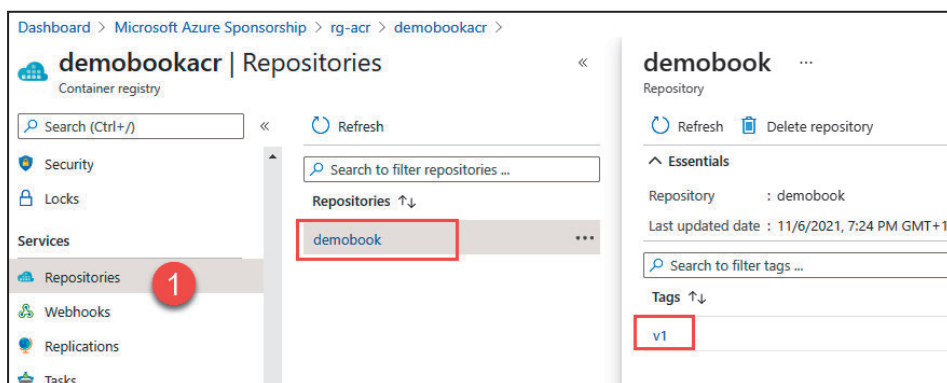
Poniższy zrzut ekranu pokazuje wysłany obraz Dockera w ACR.

5. Na koniec, aby wyciągnąć ten obraz Dockera, użyjemy następującego polecenia:

```
docker pull demobookacr.azurecr.io/demobook:v1
```

W tej sekcji dowiedzieliśmy się, jak używać Dockera do wysyłania i pobierania obrazów Dockera z prywatnego rejestru Dockera (czyli ACR).

W następnej sekcji zobaczymy, jak wdrożyć ten obraz za pomocą potoku CI/CD w zarządzanych usługach kontenerowych w chmurze — ACI i Terraform.



Rysunek 9.22. Obraz Dockera w ACR

Wdrażanie kontenera do ACI za pomocą potoku CI/CD

Jednym z głównych powodów, dla których platforma Docker szybko stała się atrakcyjna dla programistów i zespołów operacyjnych, jest to, że wdrożenie obrazów i kontenerów platformy Docker ułatwiły dla aplikacji korporacyjnych potoki CI i CD.

Aby zautomatyzować wdrażanie naszej aplikacji, utworzymy potok CI/CD, który zainstaluje obraz Dockera zawierający naszą aplikację w ACI.

ACI to usługa zarządzana przez platformę Azure, która umożliwia bardzo łatwe wdrażanie kontenerów bez martwienia się o architekturę sprzętową.

Uwaga

Aby dowiedzieć się więcej o ACI, przejdź na oficjalną stronę pod adresem <https://azure.microsoft.com/en-us/services/container-instances/>.

Ponadto dla **infrastruktury jako kodu (IaC)** skorzystamy z Terraform — o czym pisaliśmy w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”. Zrobimy to przy użyciu zasobu Azure ACI i jego integracji z obrazem Dockera.

Dlatego podzielimy tę sekcję na dwie części w następujący sposób:

- Konfiguracja kodu Terraform interfejsu Azure ACI i jego integracja z naszym obrazem platformy Docker.
- Przykład potoku CI/CD w Azure Pipelines, który umożliwia wykonanie kodu Terraform.

Na początek napiszemy kod Terraform, który umożliwia udostępnianie zasobu ACI na platformie Azure.

Tworzenie kodu Terraform dla ACI

Aby udostępnić zasób ACI za pomocą Terraform, przechodzimy do nowego katalogu *terraform-aci* i tworzymy plik Terraform *main.tf*.

W tym przykładzie udostępniemy kod Terraform dla grupy zasobów i zasobu ACI przy użyciu obiektu Terraform *azurerm_container_group*.

Uwaga

Dokumentacja do zasobu Terraform ACI jest dostępna tutaj: https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/container_group.

Plik *main.tf* zawiera następujący kod Terraform, który tworzy grupę zasobów:

```
resource "azurerm_resource_group" "acidemobook" {
  name = "demoBook"
  location = "westus2"
}
```

W tym pliku, *main.tf*, dodajemy deklaracje zmiennych w konfiguracji Terraform w następujący sposób:

```
variable "imageversion" {
  description = "Tag obrazu do wdrożenia"
}
variable "dockerhub-username" {
  description = "Tag obrazu do wdrożenia"
}
```

Dodajemy kod Terraform dla ACI z blokiem zasobów *azurerm_container_group* w następujący sposób:

```
resource "azurerm_container_group" "aci-myapp" {
  name = "aci-agent"
  location = "West Europe"
  resource_group_name = azurerm_resource_group.acidemobook.name
  os_type = "linux"
  container {
    name = "myappdemo"
    image = "docker.io/mikaelkrief/${var.dockerhub-username}
↳:${var.imageversion}"
    cpu = "0.5" memory = "1.5"
    ports {
```

```
        port = 80
        protocol = "TCP"
    }
}
```

W powyższym fragmencie kodu wykonujemy następujące czynności:

- Deklarujemy zmienne `imageversion` i `dockerhub-username`, które będą tworzone podczas potoku CI/CD i zawierają nazwę użytkownika oraz tag obrazu, który ma zostać wdrożony.
- Do zarządzania ACI używamy zasobu Terraform `azurerem_container_group`. We właściwości obrazu wskazujemy informacje o obrazie, który ma zostać wdrożony, czyli jego pełną nazwę w Docker Hubie, a także jego tag, który w naszym przykładzie jest przechowywany w zmiennej `imageversion`.

Na koniec, aby chronić plik stanu Terraform, możemy użyć zdalnego zaplecza Terraform przy użyciu usługi Azure Blob Storage, co omówiliśmy w sekcji „Ochrona pliku stanu za pomocą zdalnego zaplecza” rozdziału 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”.

Uwaga

Pełny kod źródłowy tego pliku Terraform jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP09/terraform-aci>.

Otrzymaliśmy kod Terraform, który pozwoli nam utworzyć zasób Azure ACI i wykona kontener naszego obrazu. Teraz utworzymy potok CI/CD, który automatycznie wdroży kontener aplikacji.

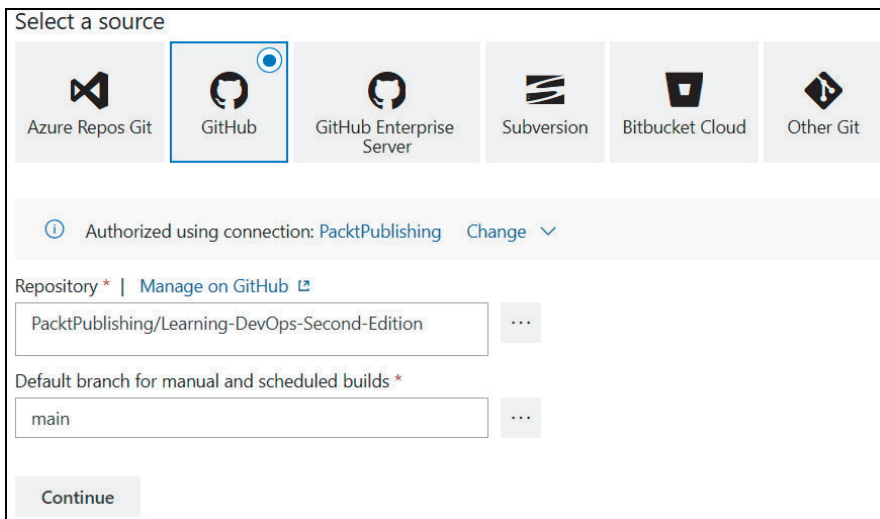
Tworzenie potoku CI/CD dla kontenera

Aby utworzyć potok CI/CD, który zbuduje nasz obraz i wykona kod Terraform, możemy użyć wszystkich narzędzi, które szczegółowo omówiliśmy w rozdziale 7., „Ciągła integracja i ciągłe wdrażanie”.

W tym rozdziale do wizualizacji potoku użyjemy Azure Pipelines, które jest jednym z wcześniej szczegółowo omówionych narzędzi. Zaleca się uważne przeczytanie sekcji „Korzystanie z Azure Pipelines dla CI/CD” w rozdziale 7., „Ciągła integracja i ciągłe wdrażanie”. Z tego powodu nie będziemy szczegółowo omawiać wszystkich etapów potoku, a jedynie te, które są istotne dla naszego tematu związanego z kontenerem.

Aby zaimplementować potok CI/CD w Azure Pipelines, wykonaj następujące kroki:

1. Utworzymy nową definicję kompilacji, której **kod źródłowy** będzie wskazywał na kopię repozytorium GitHuba (<https://github.com/PacktPublishing/Learning-DevOps-Second-Edition>), i wybierzemy folder główny tego repozytorium, jak pokazano na poniższym zrzucie ekranu:



Select a source

Azure Repos Git | **GitHub** | GitHub Enterprise Server | Subversion | Bitbucket Cloud | Other Git

Authorized using connection: PacktPublishing Change

Repository * | Manage on GitHub

PacktPublishing/Learning-DevOps-Second-Edition

Default branch for manual and scheduled builds *

main

Continue

Rysunek 9.23. Azure Pipelines ze źródłami GitHuba

Uwaga

Aby uzyskać więcej informacji o kopiach repozytorium (ang. *fork*) w GitHubie, przeczytaj rozdział 16., „DevOps dla projektów open source”.

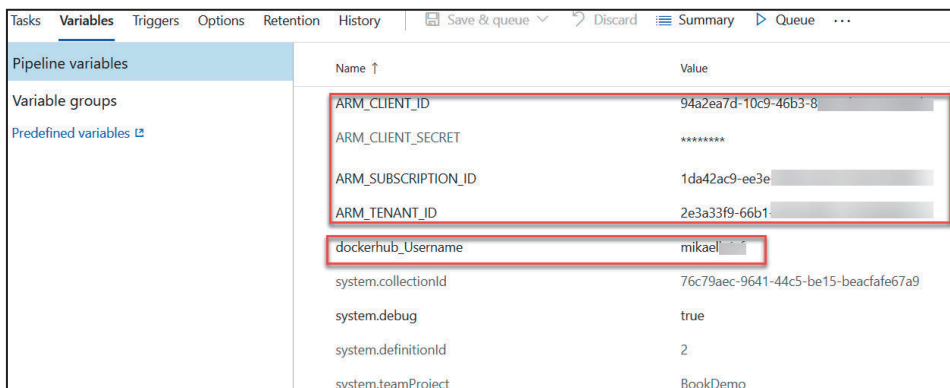
Możesz używać dowolnego systemu kontroli wersji kodu źródłowego dostępnego w Azure Pipelines.

2. Następnie na zakładce *Variables* zdefiniujemy zmienne, które będą używane w potoku. Rysunek 9.24 przedstawia informacje na karcie *Variables*.

Zdefiniowaliśmy cztery elementy informacji związane z połączeniem Terraform dla platformy Azure i nazwę użytkownika Docker Hub.

3. Następnie w zakładce *Tasks* musimy wykonać następujące czynności:

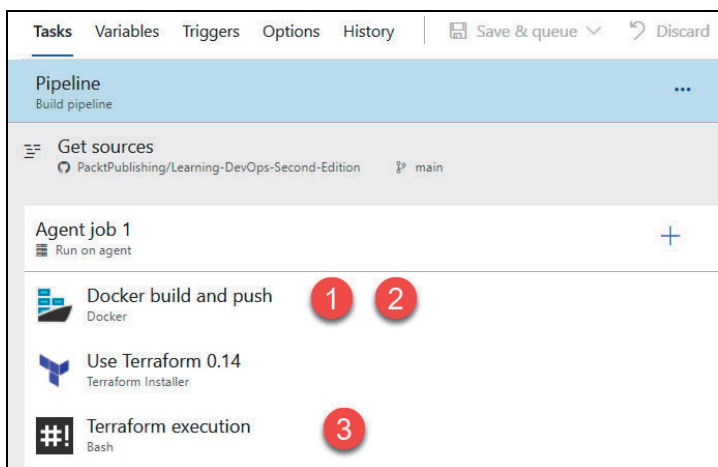
- A. Uruchom polecenie `docker build` na pliku `Dockerfile`.
- B. Prześlij obraz do Docker Hub.
- C. Uruchom kod Terraform, aby zaktualizować zasób ACI o nową wersję zaktualizowanego obrazu.



| Name ↑ | Value |
|---------------------|-------------------------------------|
| ARM_CLIENT_ID | 94a2ea7d-10c9-46b3-8... |
| ARM_CLIENT_SECRET | ***** |
| ARM_SUBSCRIPTION_ID | 1da42ac9-ee3e... |
| ARM_TENANT_ID | 2e3a33f9-66b1... |
| dockerhub_username | mikael... |
| system.collectionId | 76c79aec-9641-44c5-be15-beacafe67a9 |
| system.debug | true |
| system.definitionId | 2 |
| system.teamProject | BookDemo |

Rysunek 9.24. Zmienne potoku

Poniższy zrzut ekranu przedstawia konfigurację zadań:



Rysunek 9.25. Lista kroków potoku

Zadania wymienione w kroku 3. konfigurujemy za pomocą następujących kroków:

4. Pierwsze zadanie, *Docker build and push*, umożliwia zbudowanie obrazu Dockera i przekazanie go do Docker Hub. Jego konfiguracja jest dość prosta, jak widać na rysunku 9.26.

Oto wymagane parametry tego zadania:

- Połączenie z Docker Hubem przy użyciu połączenia usługi o nazwie DockerHub.
- Tag obrazu, który zostanie przekazany do Docker Hub.

The screenshot shows the configuration for the 'Docker build and push' task in an Azure Pipelines pipeline. The task is highlighted in the left sidebar. The configuration details on the right are as follows:

- Display name ***: Docker build and push
- Container Repository ^**:
 - Container registry: DockerHub
 - Container repository: \$(dockerHub_Username)/demobook
- Commands ^**:
 - Command *: buildAndPush
 - Dockerfile *: CHAP09/appdocker/Dockerfile
 - Build context: **
 - Tags: \$(Build.BuildNumber)

Rysunek 9.26. Parametry kroku wysyłania i budowania Dockera

5. Drugie zadanie, instalator Terraform o nazwie *Use Terraform 0.14*, umożliwia pobranie Terraform przez agenta potoku — poprzez określenie żądanej wersji Terraform.

Uwaga

To zadanie jest dostępne w Visual Studio Marketplace pod adresem <https://marketplace.visualstudio.com/items?itemName=charleszip.azur-pipelines-tasks-terraform&targetId=76c79aec-9641-44c5-be15-beacfafe67a9>.

Rysunek 9.27 przedstawia jego konfigurację, która jest bardzo prosta.

6. Ostatnie zadanie, *Bash*, umożliwia wykonanie Terraform w skrypcie Bash, a rysunek 9.28 pokazuje jego konfigurację.

Skonfigurowany skrypt wygląda tak:

```
export ARM_CLIENT_ID="$(ARM_CLIENT_ID)"
export ARM_CLIENT_SECRET="$(ARM_CLIENT_SECRET)"
export ARM_TENANT_ID="$(ARM_TENANT_ID)"
export ARM_SUBSCRIPTION_ID="$(ARM_SUBSCRIPTION_ID)"
terraform init -backend-config="backend.tfvars"
terraform apply -var "imageversion=$(Build.BuildNumber)"
-var "dockerhub-username=$(dockerhub_Username)" --autoapprove
```


Pipeline
Build pipeline

Get sources
PacktPublishing/Learning-DevOps-Second-Edition
main

Agent job 1
Run on agent

Docker build and push
Docker

Use Terraform 0.14
Terraform Installer

Terraform execution
Bash

Terraform Installer

Task version: 0.*

Display name *: Use Terraform 0.14

Version *: 0.14.10

Download URL

Control Options

Output Variables

Rysunek 9.27. Parametry kroku Terraform

Pipeline
Build pipeline

Get sources
PacktPublishing/Learning-DevOps-Second-Edition
main

Agent job 1
Run on agent

Docker build and push
Docker

Use Terraform 0.14
Terraform Installer

Terraform execution
Bash

Terraform execution

Task version: 3.*

Display name *: Terraform execution

Type: inline

Script *:

```
export ARM_CLIENT_ID="${ARM_CLIENT_ID}"
export ARM_CLIENT_SECRET="${ARM_CLIENT_SECRET}"
export ARM_TENANT_ID="${ARM_TENANT_ID}"
export ARM_SUBSCRIPTION_ID="${ARM_SUBSCRIPTION_ID}"

terraform init -backend-config="backend.tfvars"
terraform apply -var "imageversion=${Build.BuildNumber}" -var "dockerhub-username=${dockerhub_Username}" --auto-approve
```

Advanced

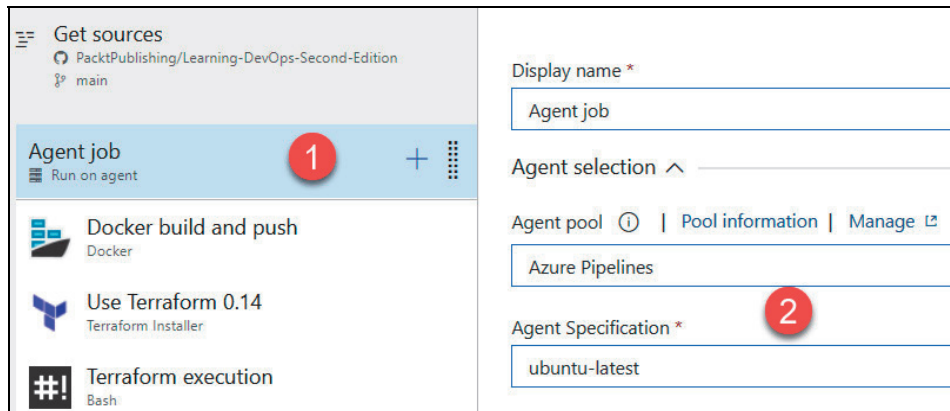
Working Directory: CHAP09/terraform-aci

Rysunek 9.28. Parametry kroku Bash

Ten skrypt wykonuje trzy akcje, które są wykonywane w następującej kolejności:

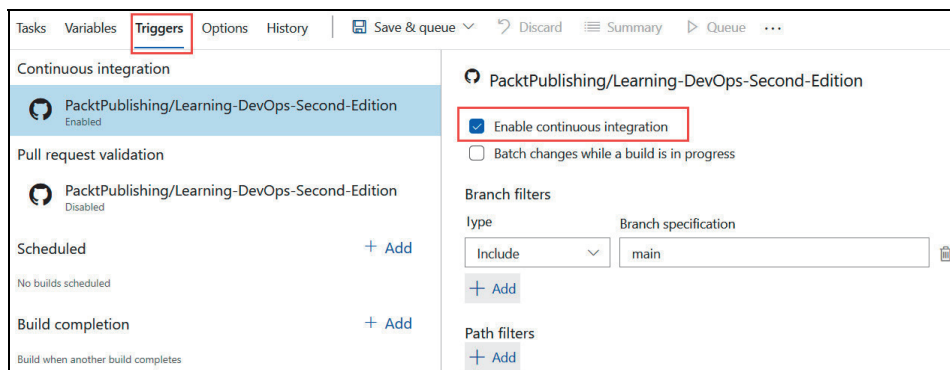
- A. Eksportuje zmienne środowiskowe wymagane przez Terraform.
- B. Wykonuje polecenie `terraform init`.
- C. Uruchamia Terraform, aby zastosować zmiany. Dwa parametry `-var` określają naszą nazwę użytkownika Docker Huba i tag, który ma zostać zastosowany. Te parametry umożliwiają wykonanie kontenera z nowym obrazem, który właśnie został przekazany do Docker Huba.

Następnie, aby skonfigurować agenta kompilacji, używanego w opcjach *Agent job*, używamy systemu Ubuntu 16.04 hostowanego przez agenta Azure Pipelines, jak pokazano na poniższym zrzucie ekranu:



Rysunek 9.29. Parametry pracy agenta

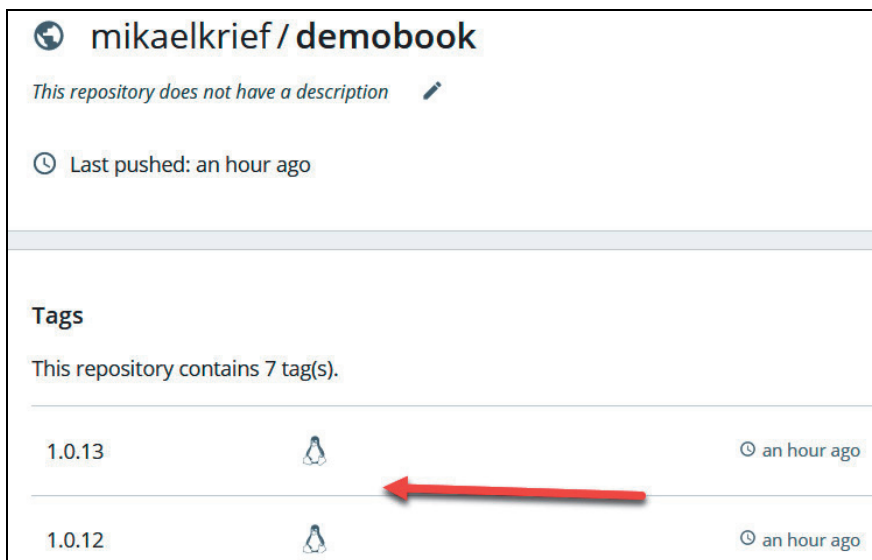
Ostatnią konfiguracją jest konfiguracja wyzwalacza na karcie *Triggers*, która włącza CI z wyzwalaczem tej kompilacji przy każdym zatwierdzeniu, jak pokazano na poniższym zrzucie ekranu:



Rysunek 9.30. Włączony CI

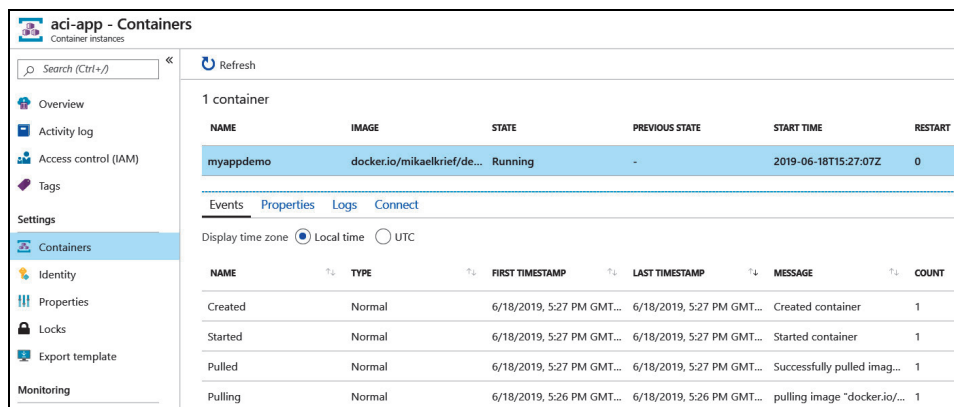
Ta operacja kończy konfigurację potoku CI/CD w Azure Pipelines.

Po uruchomieniu tej kompilacji powinniśmy być w stanie zobaczyć nową wersję obrazu Dockera na końcu jej wykonania, odpowiadającą numerowi kompilacji, która wysłała obraz Dockera do Docker Hub, jak pokazano na poniższym zrzucie ekranu:



Rysunek 9.31. Przesłany obraz Dockera, za pośrednictwem potoku, w Docker Hubie

W portalu Azure otrzymujemy nasz zasób ACI `aci-app` z naszym kontenerem `mydemoapp`, jak widać na poniższym zrzucie ekranu:



Rysunek 9.32. Kontenery ACI

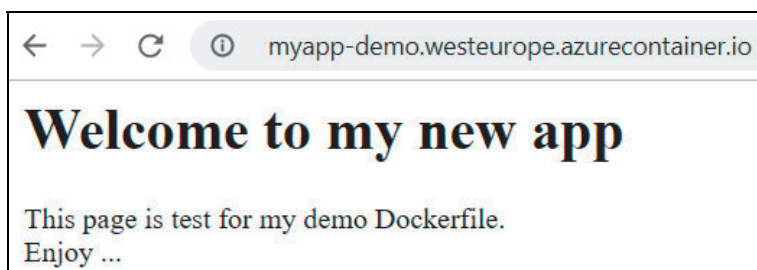
Zauważ, że kontener działa dobrze.

Teraz, aby uzyskać dostęp do naszej aplikacji, musimy pobrać **pełną, jednoznaczną nazwę domenową** (ang. *fully qualified domain name* — FQDN) kontenera podanego w portalu Azure. Poniższy zrzut ekranu pokazuje, gdzie możesz to znaleźć:



Rysunek 9.33. FQDN kontenera aplikacji w ACI

Otwieramy przeglądarkę internetową o tym adresie URL:



Rysunek 9.34. Testowanie aplikacji

Nasza aplikacja internetowa wyświetla się poprawnie.

Przy następnej aktualizacji aplikacji zostanie wyzwolona kompilacja CI/CD, nowa wersja obrazu zostanie wysłana do Docker Huba, a nowy kontener zostanie załadowany z nową wersją obrazu.

W tej sekcji przyjrzelśmy się tworzeniu kodu Terraform do zarządzania zasobem ACI i konstruowaniu potoku CI/CD w Azure Pipelines, co umożliwia wdrożenie obrazu aplikacji w Docker Hubie, a następnie zaktualizowanie zasobu ACI za pomocą nowej wersji obrazu.

W następnej sekcji omówimy inny przypadek użycia Dockera — czyli uruchamianie narzędzi wiersza poleceń.

Korzystanie z Dockera przy użyciu narzędzi wiersza poleceń

Do tej pory w tym rozdziale analizowaliśmy przypadki użycia Dockera do konteneryzacji aplikacji internetowej za pomocą nginx.

Innym przypadkiem użycia platformy Docker jest możliwość uruchamiania narzędzi wiersza poleceń znajdujących się w kontenerach platformy Docker.

Aby to zilustrować, uruchomimy przykładową konfigurację Terraform przy użyciu pliku binarnego, który znajduje się nie na komputerze lokalnym, ale w kontenerze Dockera.

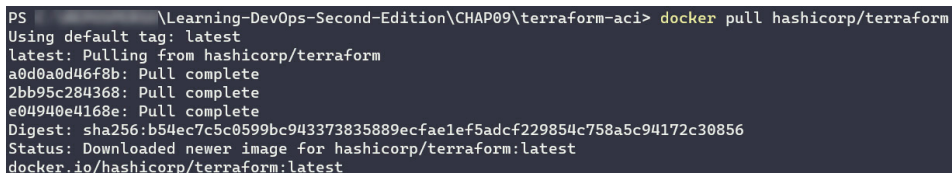
Konfiguracja Terraform, której używamy w tej sekcji, jest taka sama jak w poprzedniej sekcji, „Wdrażanie kontenera do ACI za pomocą potoku CI/CD”, a kod źródłowy jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP09/terraform-aci>.

Celem tego laboratorium jest uruchomienie tej konfiguracji Terraform przy użyciu pliku binarnego znajdującego się w kontenerze Dockera. Aby uruchomić to laboratorium, wykonaj następujące kroki:

1. Najpierw ściągamy oficjalny obraz Terraform z Docker Huba za pomocą tego polecenia:

```
docker pull hashicorp/terraform
```

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia:



```
PS > \Learning-DevOps-Second-Edition\CHAP09\terraform-aci> docker pull hashicorp/terraform
Using default tag: latest
latest: Pulling from hashicorp/terraform
a0d0a0d46f8b: Pull complete
2bb95c284368: Pull complete
e04940e4168e: Pull complete
Digest: sha256:b54ec7c5c0599bc943373835889ecfae1ef5adcdf229854c758a5c94172c30856
Status: Downloaded newer image for hashicorp/terraform:latest
docker.io/hashicorp/terraform:latest
```

Rysunek 9.35. Polecenie docker pull, służące do pobrania obrazu Terraform

2. Następnie w folderze zawierającym konfigurację Terraform uruchomimy przepływ pracy Terraform za pomocą następujących trzech poleceń docker run: Najpierw uruchamiamy polecenie terraform init w następujący sposób:

```
docker run -i -t -v ${PWD}:/usr/tf -w /usr/tf '
--env ARM_CLIENT_ID=<identyfikator klienta Azure> '
--env ARM_CLIENT_SECRET=<klucz klienta Azure> '
--env ARM_SUBSCRIPTION_ID=<subskrypcja Azure> '
--env ARM_TENANT_ID=<identyfikator dzierżawy Azure> '
--env ARM_ACCESS_KEY=<klucz dostępu Azure> '
hashicorp/terraform:latest'
init -backend-config="backend.tfvars"
```

W powyższym poleceniu docker run korzystamy z następujących argumentów:

- -v, aby utworzyć wolumin do montowania bieżącego katalogu lokalnego, który zawiera kod Terraform wewnątrz katalogu /usr/tf w kontenerze.
- -w, aby określić katalog roboczy.

- `--env` ze zmienną środowiskową niezbędną do uwierzytelniania Terraform na platformie Azure.
- Hashicorp/terraform:latest, czyli nazwa obrazu.
- `init -backend-config="backend.tfvars"`, który jest argumentem służącym do uruchomienia polecenia Terraform.

Teraz uruchamiamy polecenie `terraform plan` w następujący sposób:

```
docker run -i -t -v ${PWD}:/usr/tf -w /usr/tf '
--env ARM_CLIENT_ID=<identyfikator klienta Azure> " '
--env ARM_CLIENT_SECRET=<klucz klienta Azure> " '
--env ARM_SUBSCRIPTION_ID=<subskrypcja Azure> " '
--env ARM_TENANT_ID=<identyfikator dzierżawy Azure> " '
--env ARM_ACCESS_KEY=<klucz dostępu Azure> " '
hashicorp/terraform:latest '
plan -var dockerhub-username=<nazwa użytkownika Docker Huba> -out
↳ plan.tfplan
```

W powyższym poleceniu używamy tego samego argumentu, z poleceniem `plan` dla Terraform.

Na koniec uruchamiamy polecenie `terraform apply` w następujący sposób:

```
docker run -i -t -v ${PWD}:/usr/tf -w /usr/tf '
--env ARM_CLIENT_ID=<identyfikator klienta Azure> " '
--env ARM_CLIENT_SECRET=<klucz klienta Azure> " '
--env ARM_SUBSCRIPTION_ID=<subskrypcja Azure> " '
--env ARM_TENANT_ID=<identyfikator dzierżawy Azure> " '
--env ARM_ACCESS_KEY=<klucz dostępu Azure> " '
hashicorp/terraform:latest '
apply plan.tfplan
```

Właśnie przestudiowaliśmy podstawowy przykład użycia narzędzia (tutaj z Terraform), które działa w kontenerze Dockera. Takie zastosowanie Dockera ma następujące zalety:

- Nie ma potrzeby instalowania tych narzędzi na lokalnym komputerze; proces instalacji odbywa się za pomocą edytora narzędzi w obrazie Dockera.
- Możesz uruchomić kilka wersji tego samego narzędzia.

W kolejnej sekcji omówimy wykorzystanie usługi Docker Compose, która pozwala na zamontowanie kilku obrazów Dockera w tej samej grupie kontenerów.

Pierwsze kroki z Docker Compose

Do tej pory w rozdziale uczyliśmy się, jak utworzyć plik Dockerfile, utworzyć obraz Dockera i uruchomić kontener tego obrazu.

Obecnie aplikacje nie działają w trybie autonomicznym; wymagają innych zależności, takich jak usługa (np. inna aplikacja; **interfejs programowania aplikacji — API**) lub baza danych. Oznacza to, że w przypadku tych aplikacji przepływ pracy platformy Docker jest bardziej spójny. Rzeczywiście, gdy pracujemy z kilkoma aplikacjami Dockera, musimy wykonać dla każdej z nich polecenia `docker build` i `docker run`, co wymaga pewnego wysiłku.

Docker Compose to bardziej zaawansowane narzędzie Dockera, które pozwala nam na jednoczesne wdrożenie kilku kontenerów Dockera w tym samym cyklu wdrażania. Docker Compose pozwala nam również zarządzać elementami, które są wspólne dla tych kontenerów Dockera, takimi jak woluminy danych i konfiguracja sieci.

Uwaga

Aby uzyskać więcej informacji na temat Docker Compose, przeczytaj oficjalną dokumentację tutaj: <https://docs.docker.com/compose>.

W Docker Compose ta konfiguracja — zawierająca obrazy platformy Docker, woluminy i sieć stanowiące artefakty tej samej aplikacji — zawarta jest po prostu w pliku konfiguracyjnym w formacie **YAML** (ang. *YAML Ain't Markup Language*).

W tej sekcji opowiemy o instalacji Docker Compose w trybie podstawowym. Następnie napiszemy prosty plik konfiguracyjny Docker Compose, aby uruchomić aplikację nginx z bazą danych MySQL w tym samym kontekście. Na koniec utworzymy ten plik konfiguracyjny Docker Compose i wyświetlimy wynik w kontenerach Dockera.

Instalowanie Docker Compose

W systemie Windows lub macOS plik binarny **Docker Compose** jest już zainstalowane z programem Docker Desktop.

Postępuj zgodnie z tą dokumentacją, aby zainstalować plik binarny Docker Compose w systemie Linux: <https://docs.docker.com/compose/install/#install-compose-on-linux-systems>.

Możemy sprawdzić, czy Docker Compose jest poprawnie zainstalowane, uruchamiając następujące polecenie:

```
docker-compose version
```

Poniższy zrzut ekranu pokazuje wynik tego polecenia.

Polecenie to wyświetla wersję zainstalowanego pliku binarnego docker-compose.

```
PS C:\BEP00000000\Learning-DevOps-Second-Edition\CHAP09\docker-compose> docker-compose version
docker-compose version 1.29.2, build 5becea4c
docker-py version: 5.0.0
CPython version: 3.9.0
OpenSSL version: OpenSSL 1.1.1g  21 Apr 2020
```

Rysunek 9.36. Polecenie `docker-compose version`

Po zainstalowaniu Docker Compose napiszemy plik konfiguracyjny YAML dla Docker Compose.

Tworzenie pliku konfiguracyjnego dla Docker Compose

Aby wdrożyć kontenery za pomocą Docker Compose, napiszemy plik konfiguracyjny, by uruchomić kontener `nginx` połączony z kontenerem `mysql`.

W tym celu utworzymy nowy plik o nazwie `docker-compose.yml` z następującą zawartością obecną w dwóch blokach kodu.

Pierwszy fragment kodu w YAML służy do tworzenia kontenera `nginx` w następujący sposób:

```
version: '3' #wersja schematu YAML dla Docker Compose
services:
  nginx:
    image: nginx:latest
    container_name: nginx-container
    ports:
      - 8080:80
```

W poprzednim fragmencie kodu zaczynamy od właściwości `services`, która zawiera listę usług (lub aplikacji Dockera) do uruchomienia w Dockerze. Pierwszą usługą jest usługa `nginx`. Konfigurujemy obraz `nginx` dockera, nazwę kontenera i wystawiamy na zewnątrz port, którym jest 8080, dla lokalnego dostępu usługi `nginx`.

Następnie dodajemy YAML dla usługi `MySQL` z następującym kodem:

```
mysql:
  image: mysql:5.7
  container_name: mysql-container
  environment:
    MYSQL_ROOT_PASSWORD: secret
    MYSQL_DATABASE: mydb
    MYSQL_USER: myuser
    MYSQL_PASSWORD: password
```

W poprzednim fragmencie kodu konfigurujemy usługę `mysql` z obrazem Dockera, nazwą kontenera i wymaganą zmienną środowiskową do konfiguracji dostępu do bazy danych.

Uwaga

Pełny kod źródłowy tego pliku jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP09/docker-compose/docker-compose.yml>.

Właśnie utworzyliśmy plik YAML dla konfiguracji docker-compose. Teraz uruchomimy Docker Compose, aby wykonać kontenery opisane w tej konfiguracji.

Wykonywanie Docker Compose

Aby uruchomić kontenery Dockera opisane w pliku konfiguracyjnym YAML, uruchomimy podstawową operację Docker Compose, wykonując następujące polecenie w folderze zawierającym plik *docker-compose.yml*:

```
docker-compose up -d
```

Uwaga

Dodano opcję `-d` do uruchamiania kontenerów w trybie odłączonym.

Pełna dokumentacja interfejsu docker-compose jest dostępna tutaj: <https://docs.docker.com/compose/reference/>.

Poniższy zrzut ekranu pokazuje wykonanie polecenia `docker-compose up -d`:

```
PS \Learning-DevOps-Second-Edition\CHAP09\docker-compose> docker-compose up -d
[+] Running 12/12
 - mysql Pulled                                42.7s
 - b380bbd43752 Already exists                  0.0s
 - f23cbf2ecc5d Pull complete                  0.6s
 - 30cfc6c29c0a Pull complete                  2.6s
 - b38609286cbe Pull complete                  2.7s
 - 8211d9e66cd6 Pull complete                  2.8s
 - 2313f9eeca4a Pull complete                  8.7s
 - 7eb487d00da0 Pull complete                  8.7s
 - a71aacf913e7 Pull complete                  8.8s
 - 393153c555df Pull complete                  40.0s
 - 06628e2290d7 Pull complete                  40.1s
 - ff2ab8dac9ac Pull complete                  40.1s
[+] Running 2/2
 - Container mysql-container Started            4.0s
 - Container nginx-container Running            0.0s
```

Rysunek 9.37. Polecenie `docker-compose up -d`

Pod koniec tego wykonania Docker Compose wyświetla listę uruchomionych kontenerów. W naszym przykładzie są to `nginx` i `mysql`.

Aby sprawdzić, czy kontenery są uruchomione, wykonujemy polecenie `docker ps` w celu wyświetlenia listy uruchomionych kontenerów.

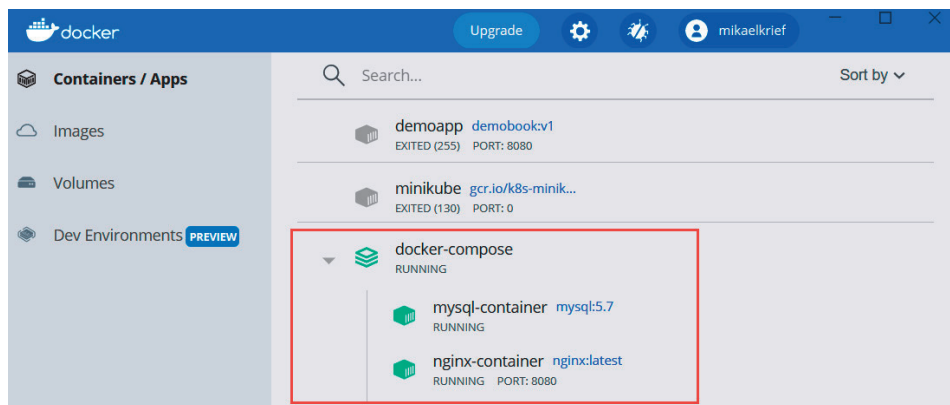
Poniższy zrzut ekranu przedstawia wykonanie polecenia `docker ps`:

```
PS C:\Users\mkrief> docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
2fc77d3debb0   nginx:latest   "/docker-entrypoint. ..." 3 minutes ago  Up 3 minutes  0.0.0.0:8080->80/tcp               nginx-container
7db499e87466   mysql:5.7      "docker-entrypoint.s ..." 3 minutes ago  Up 3 minutes  3306/tcp, 33060/tcp               mysql-container
```

Rysunek 9.38. Polecenie `docker ps`

Widzimy, że nasze dwa kontenery są uruchomione.

W systemie Windows lub macOS możemy również użyć Docker Desktop, który wyświetla na liście kontenerów uruchomione kontenery zamontowane przez Docker Compose wewnątrz grupy `docker-compose`, jak pokazano na poniższym zrzucie ekranu:



Rysunek 9.39. Lista kontenerów Docker Compose w Docker Desktop

W tej sekcji poznaliśmy kilka podstawowych reguł dla tworzenia plików konfiguracyjnych Docker Compose i lokalnego wykonywania Docker Compose w celu uruchamiania kontenerów Dockera.

W następnej sekcji omówimy wykonanie tych kontenerów zdalnie w ACI.

Wdrażanie kontenerów Docker Compose w ACI

ACI omówiliśmy w sekcji „Wdrażanie kontenera do ACI za pomocą potoku CI/CD”.

Teraz dowiemy się, jak uruchomić kontenery z konfiguracją Docker Compose w ACI, aby uruchomić zestaw kontenerów, które znajdują się w tych samych usługach aplikacji.

W tym laboratorium użyjemy tej samej konfiguracji Docker Compose, której nauczyliśmy się w sekcji „Korzystanie z Dockera przy użyciu narzędzi wiersza poleceń”. Jedyną różnicą jest to, że dla usługi nginx otwieramy port 80 zamiast 8080. Z portu 8080 korzystaliśmy lokalnie, ponieważ mój port 80 jest już używany przez inną usługę.

Aby wdrożyć kontenery w ACI, wykonamy następujące kroki:

1. Wewnątrz naszej subskrypcji Azure utwórz nową grupę zasobów o nazwie `rg-acicompose`.
2. Następnie w konsoli uruchom następującą komendę Dockera, aby zalogować się do Azure:

`docker login azure`

Wykonanie tego polecenia otwiera okno, które pozwala nam uwierzytelnić się w naszej subskrypcji Azure.

3. Utwórz nowy kontekst Dockera, uruchamiając następujące polecenie:

`docker context create aci demobookaci`

Uruchamiając to polecenie, wybieramy subskrypcję platformy Azure i grupę zasobów, którą utworzyliśmy w kroku 1., jak pokazano na poniższym zrzucie ekranu:

```
PS > \Learning-DevOps-Second-Edition\CHAP09\docker-compose> docker context create aci demobookaci
? Select a subscription ID Microsoft Azure Sponsorship (t
? Select a resource group rg-acicompose (westeurope)
Successfully created aci context "demobookaci"
```

Rysunek 9.40. Tworzenie kontekstu Dockera dla ACI

4. Sprawdź nowy kontekst platformy Docker, uruchamiając następujące polecenie:

`docker context ls`

Polecenie to wyświetla listę kontekstów Dockera i za pomocą symbolu * wskazuje bieżący kontekst, jak pokazano na poniższym zrzucie ekranu:

```
PS > \Learning-DevOps-Second-Edition\CHAP09\docker-compose> docker context ls
NAME      TYPE      DESCRIPTION          DOCKER ENDPOINT
ATOR
default *  moby      Current DOCKER_HOST based configuration  npipe:///./pipe/docker_engine
demobookaci  aci      rg-acicompose@westeurope                npipe:///./pipe/dockerDesktopLinuxEngine
desktop-linux  moby
```

Rysunek 9.41. Lista kontekstów Dockera

5. Wybierz nowo utworzony kontekst `demobookaci`, uruchamiając następujące polecenie:

`docker context use demobookaci`

6. Na koniec, aby wdrożyć aplikację konfiguracyjną Docker Compose wewnątrz tego zasobu ACI, uruchom następujące polecenie:

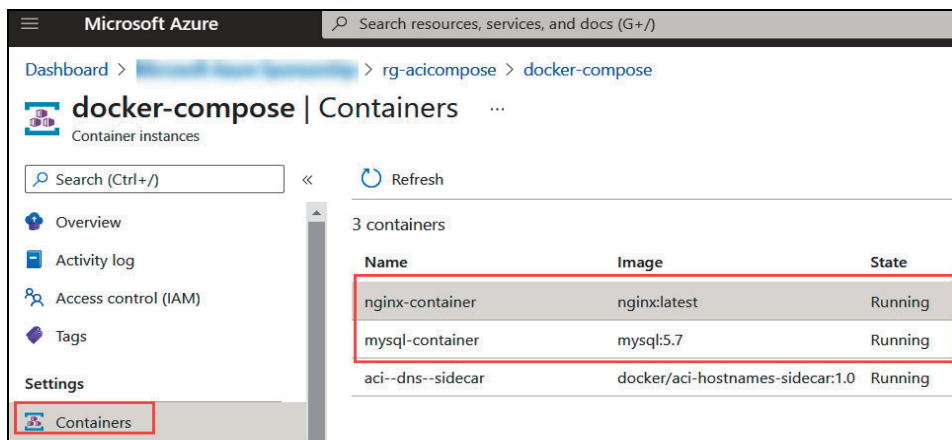
docker compose up

Poniższy zrzut ekranu przedstawia wynik wykonania tego polecenia:

```
PS > \Learning-DevOps-Second-Edition\CHAP09\docker-compose> docker compose up
[+] Running 3/3
 - Group docker-compose Created
 - nginx-container Created
 - mysql-container Created
```

Rysunek 9.42. Polecenie służące do wdrożenia w ACI — docker compose up

W naszej subskrypcji Azure możemy zobaczyć utworzony zasób ACI z dwoma kontenerami, jak pokazano na poniższym zrzucie ekranu:



Rysunek 9.43. Kontenery ACI utworzone przez Docker Compose

Na koniec, aby uzyskać dostęp do wdrożonej aplikacji, znajdź nazwę FQDN aplikacji we właściwościach ACI i uruchom aplikację w przeglądarce. Dokładnie tego nauczyliśmy się w sekcji „Wdrażanie kontenera do ACI za pomocą potoku CI/CD” w tym rozdziale.

Uwaga

Aby zapoznać się z kolejnym przykładem użycia Docker Compose w ACI, przeczytaj oficjalny samouczek tutaj: <https://docs.microsoft.com/en-us/azure/container-instances/tutorial-docker-compose>.

W tej sekcji dowiedzieliśmy się, jak wdrażać wiele kontenerów za pomocą Docker Compose w ACI przy użyciu pliku YAML *docker-compose* i niektórych poleceń Dockera dla kontekstu.

Podsumowanie

W tym rozdziale przedstawiliśmy Dockera i jego podstawowe założenia. Omówiliśmy niezbędne kroki, aby utworzyć konto w Docker Hubie, a następnie zainstalowaliśmy Dockera lokalnie za pomocą Docker Desktop.

Utworzyliśmy plik *Dockerfile*, który szczegółowo opisuje budowę obrazu Dockera dla aplikacji internetowej, a także przyjrzelśmy się głównym instrukcjom, z których ten plik się składa — FROM, COPY i RUN.

Wykonaliśmy polecenia `docker build` i `docker run`, aby zbudować obraz z naszego pliku *Dockerfile*. Utworzony obraz wykonaliśmy lokalnie, a następnie wysłaliśmy go do Docker Hub za pomocą polecenia `push`.

W drugiej części tego rozdziału wdrożyliśmy i wykonaliśmy potok CI/CD w Azure Pipelines, aby wdrożyć nasz kontener w zasobie ACI, który został udostępniony za pomocą Terraform. Następnie omówiliśmy użycie Dockera do uruchamiania narzędzi wiersza poleceń takich jak Terraform.

Wreszcie nauczyliśmy się instalować i używać Docker Compose do tworzenia wielu kontenerów aplikacji i wdrażania ich w ACI.

W kolejnym rozdziale będziemy kontynuować tematykę kontenerów i przyjrzymy się wykorzystaniu Kubernetesa, czyli narzędzia do zarządzania kontenerami na dużą skalę. Użyjemy usługi **Azure Kubernetes Service (AKS)** i Azure Pipelines do wdrożenia aplikacji w Kubernetesie z potokiem CI/CD.

Pytania

1. Co to jest Docker Hub?
2. Jaki jest podstawowy element pozwalający na utworzenie obrazu Dockera?
3. Jaka instrukcja w pliku Dockerfile definiuje obraz bazowy?
4. Które polecenie Dockera umożliwia utworzenie obrazu?
5. Które polecenie Dockera umożliwia utworzenie instancji nowego kontenera?
6. Które polecenie Dockera pozwala na opublikowanie obrazu w Docker Hubie?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o Dockerze, oto kilka świetnych książek:

- *Docker Cookbook* — <https://www.packtpub.com/virtualization-and-cloud/docker-cookbook-second-edition>.
- *Beginning DevOps with Docker* — <https://www.packtpub.com/virtualization-and-cloud/beginning-devops-docker>.

Efektywne zarządzanie kontenerami za pomocą Kubernetesa

Rozdział

10

W poprzednim rozdziale zapoznaliśmy się szczegółowo z kontenerami i Dockerem. Poznaliśmy budowę obrazu Dockera i tworzenie instancji nowego kontenera na maszynie lokalnej. Na koniec **skonfigurowaliśmy potok ciągłej integracji/ciągłego wdrażania (CI/CD)**, który kompiluje obraz, wdraża go w usłudze Docker Hub i wykonuje jego kontener w **Azure Container Instances (ACI)**.

Wszystko to działa dobrze i nie nastrocza zbyt wielu problemów przy pracy z kilkoma kontenerami. Jednak w przypadku tzw. **aplikacji mikrouslug** — czyli aplikacji, które składają się z kilku usług (każda z nich jest kontenerem) — będziemy musieli zarządzać tymi kontenerami i organizować je.

Na rynku dostępne są dwa główne narzędzia służące do orkiestracji kontenerów: Docker Swarm i Kubernetes.

Od jakiegoś czasu Kubernetes, znany również jako **K8S**, okazuje się prawdziwym liderem w dziedzinie zarządzania kontenerami i dlatego staje się *nieodzownym* elementem konteneryzacji aplikacji.

W tym rozdziale dowiemy się, jak zainstalować Kubernetesa na komputerze lokalnym, a także jak wdrożyć w nim aplikację zarówno w sposób standardowy, jak i za pomocą Helma. Narzędzie Helm poznamy bardziej szczegółowo, tworząc paczkę (ang. *chart*) i publikując ją w prywatnym rejestrze w **Azure Container Registry (ACR)**.

Następnie opowiemy o usłudze **Azure Kubernetes Service (AKS)** jako przykładzie klastra Kubernetesa, a na koniec dowiemy się, jak monitorować aplikacje i metryki w Kubernetesie.

W tym rozdziale zostaną omówione następujące tematy:

- instalacja Kubernetesa,
- pierwszy przykład wdrożenia aplikacji w Kubernetesie,
- używanie Helma jako menedżera pakietów,
- publikowanie charta Helma w prywatnym rejestrze (ACR),
- korzystanie z AKS,
- tworzenie potoku CI/CD dla Kubernetesa za pomocą Azure Pipelines,
- monitorowanie aplikacji i metryk w Kubernetesie.

Wymagania techniczne

Ten rozdział jest kontynuacją rozdziału o Dockerze, więc aby go poprawnie zrozumieć, konieczne jest przeczytanie poprzedniego rozdziału i zainstalowanie Docker Desktop (dla systemu operacyjnego Windows (OS)).

W części CI/CD tego rozdziału będziesz musiał pobrać kod źródłowy, który został dostarczony w poprzednim rozdziale o Dockerze, dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP09/appdocker>.

W przypadku sekcji dotyczącej paczek Helma w ACR konieczna jest subskrypcja platformy Azure (zarejestruj się bezpłatnie tutaj: <https://azure.microsoft.com/en-us/free/>) i instalacja **interfejsu wiersza poleceń** platformy Azure (CLI) dostępnego tutaj: <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli>.

Cały kod źródłowy tego rozdziału jest dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP10>.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/3p7ydtj>.

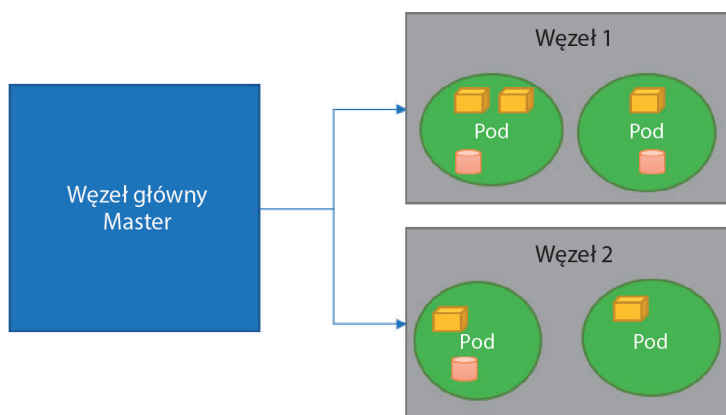
Instalacja Kubernetesa

Zanim zainstalujemy Kubernetesa, musimy zapoznać się z jego architekturą i głównymi komponentami, ponieważ Kubernetes nie jest prostym narzędziem, ale klastrem — tzn. składa się z **serwera głównego** i innych serwerów podrzędnych, zwanych **węzłami** (ang. *nodes*).

Poznaj uproszczoną architekturę Kubernetesa.

Przegląd architektury Kubernetesa

Kubernetes to platforma składająca się z kilku komponentów, które łączą się ze sobą i zwiększają na żądanie swoją liczbę, by umożliwić lepszą skalowalność aplikacji. Architekturę Kubernetesa, która jest modelem typu klient/serwer, można przedstawić w prosty sposób, jak pokazano na poniższym diagramie:



Rysunek 10.1. Architektura Kubernetesa

Na diagramie widzimy, że klastrowy składa się z komponentu głównego i węzłów (zwanymi również **węzłami roboczymi** — ang. *worker*), które reprezentują serwery podrzędne.

W każdym z tych węzłów znajdują się **pody**, które są wirtualnymi elementami. Będą one zawierać kontenery i woluminy.

Mówiąc prościej: możemy utworzyć jeden pod na aplikację, który będzie zawierał wszystkie kontenery aplikacji. Na przykład jeden pod może zawierać kontener serwera WWW, kontener bazy danych i wolumin, który będzie zawierał trwałe pliki obrazów i pliki bazy danych.

kubectl to narzędzie klienckie, które pozwala nam na interakcję z klastrem Kubernetesa.

Poznaliśmy główne wymagania, które pozwalają nam pracować z Kubernetesem, spójrzmy więc, jak go zainstalować na komputerze lokalnym.

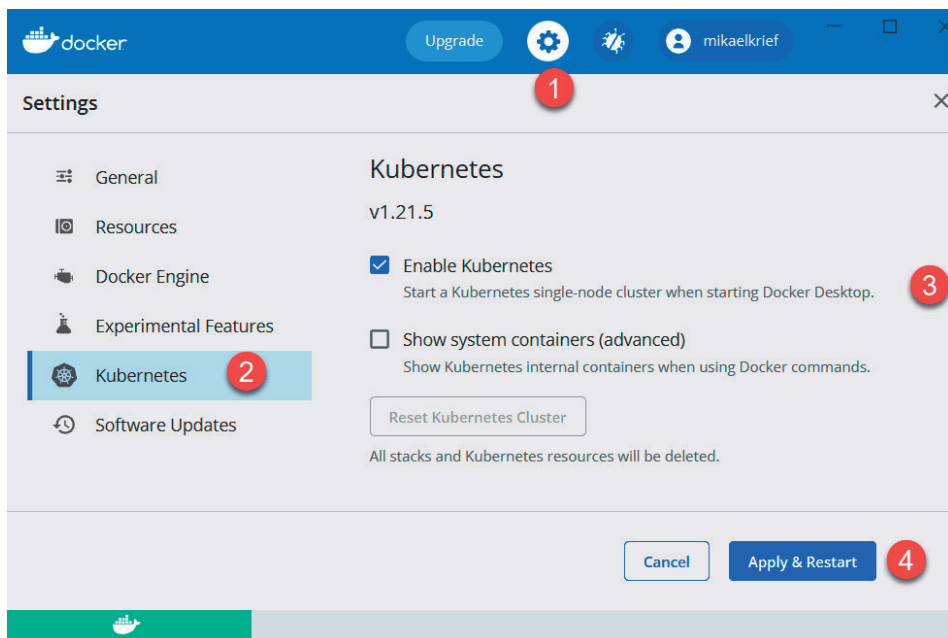
Instalacja Kubernetesa na komputerze lokalnym

Podczas tworzenia aplikacji kontenerowej, która ma być hostowana na Kubernetesie, bardzo ważne jest, aby móc uruchomić aplikację (wraz z jej kontenerami) na komputerze lokalnym przed wdrożeniem jej na zdalnych klastrach produkcyjnych Kubernetesa.

Istnieje kilka rozwiązań lokalnej instalacji Kubernetesa, które są szczegółowo opisane poniżej.

Pierwszym rozwiązaniem jest użycie **Docker Desktop** poprzez wykonanie następujących kroków:

1. Jeśli mamy już zainstalowany Docker Desktop, o którym dowiedzieliśmy się w rozdziale 9., „Konteneryzacja aplikacji za pomocą Dockera”, możemy aktywować opcję *Enable Kubernetes* w ustawieniach *Settings* na zakładce *Kubernetes*, jak pokazano na poniższym zrzucie ekranu:



Rysunek 10.2. Włączanie Kubernetesa w Docker Desktop

2. Po kliknięciu przycisku *Apply & Restart* Docker Desktop zainstaluje lokalny klaster Kubernetesa, a także narzędzie klienta *kubectl* na komputerze lokalnym.

Drugim sposobem lokalnej instalacji Kubernetesa jest zainstalowanie *minikube*, które instaluje również lokalnie uproszczony klaster Kubernetesa. Oto oficjalna dokumentacja, z którą możesz się zapoznać: <https://minikube.sigs.k8s.io/docs/start/>.

Uwaga

Istnieją inne rozwiązania do instalowania lokalnie Kubernetesa, takie jak *kind* lub *kubeadm*. Aby uzyskać więcej informacji, przeczytaj dokumentację tutaj: <https://kubernetes.io/docs/tasks/tools/>.

Po lokalnej instalacji Kubernetesa sprawdzimy poprawność instalacji, wykonując następujące polecenie w terminalu:

```
kubectl version --short
```

Poniższy zrzut ekranu przedstawia wyniki dla poprzedniego polecenia:

```
PS C:\Users\mkrief> kubectl version --short
Client Version: v1.20.5
Server Version: v1.20.9
```

Rysunek 10.3. kubectl zwraca numer wersji

Uwaga

Wszystkie operacje, które wykonujemy na naszym klastrze Kubernetesa, będą wykonywane za pomocą poleceń kubectl.

Po zainstalowaniu naszego klastra Kubernetesa będziemy potrzebować kolejnego elementu, jakim jest pulpit nawigacyjny Kubernetesa. Jest to aplikacja internetowa, która pozwala nam na podgląd statusu, a także wszystkich komponentów naszego klastra.

W następnej sekcji powiemy, jak zainstalować i przetestować pulpit nawigacyjny Kubernetesa.

Instalacja pulpitu nawigacyjnego Kubernetesa

W celu zainstalowania dashboardu Kubernetesa, czyli gotowej, konteneryzowanej aplikacji internetowej, która zostanie wdrożona w naszym klastrze, uruchomimy w terminalu następujące polecenie:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.4.0/aio/deploy/recommended.yaml
```

Jego wykonanie pokazano na poniższym zrzucie ekranu:

```
PS C:\Users\mkrief> kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.4.0/aio/deploy/recommended.yaml
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
```

Rysunek 10.4. Instalacja pulpitu nawigacyjnego Kubernetesa

Na zrzucie widać, że tworzone są różne artefakty, przedstawione w następujący sposób: dane uwierzytelniające (ang. *secrets*), dwie aplikacje webowe, **kontrole dostępu oparte na rolach** (ang. *role-based access control* — RBAC), uprawnienia i usługi.

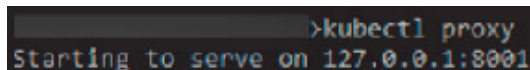
Uwaga

Należy zauważyć, że adres **URL** (ang. *Uniform Resource Locator*) wymieniony w parametrach polecenia instalującego pulpit nawigacyjny może się zmieniać w zależności od wersji pulpitu nawigacyjnego. Aby znaleźć ostatni prawidłowy adres URL, zapoznaj się z oficjalną dokumentacją, odwiedzając stronę <https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>.

Teraz, gdy zainstalowaliśmy pulpit nawigacyjny Kubernetesa, połączymy się z nim i skonfigurujemy go.

Aby otworzyć dashboard i połączyć się z nim z naszej lokalnej maszyny, musimy najpierw utworzyć proxy między klastrem Kubernetesa a naszą maszyną, wykonując następujące kroki:

1. Aby utworzyć proxy, wykonujemy polecenie `kubect1 proxy` w terminalu. Szczegóły wykonania przedstawia poniższy zrzut ekranu:



```
>kubect1 proxy
Starting to serve on 127.0.0.1:8001
```

Rysunek 10.5. Polecenie `kubect1 proxy`

Widzimy, że proxy jest otwarte na adresie localhost (127.0.0.1) na porcie 8001.

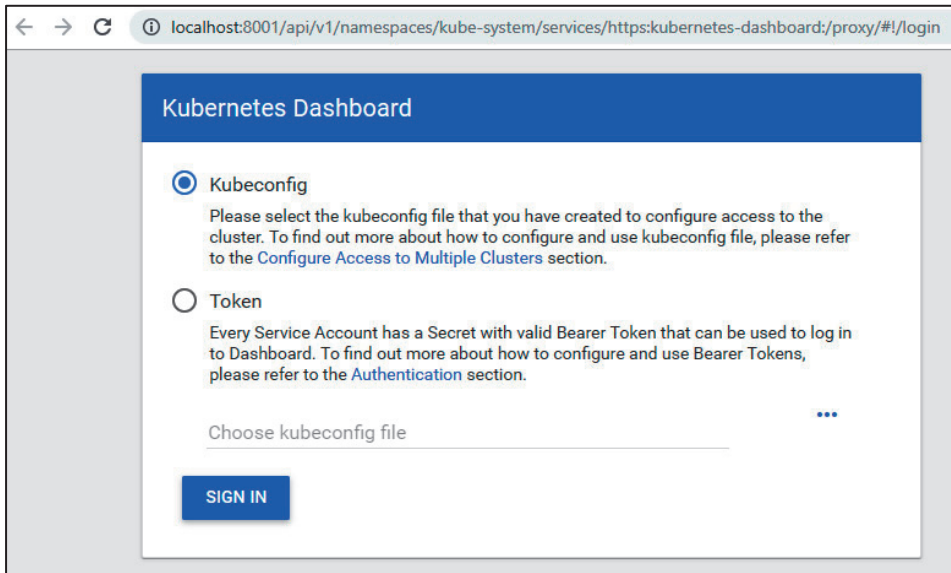
2. Następnie w przeglądarce internetowej otwórz następujący adres URL, <http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/#/login>, który jest lokalnym adresem URL (localhost i 8001). Jest on tworzony przez serwer proxy i wskazuje na zainstalowaną aplikację pulpitu nawigacyjnego Kubernetesa.

Poniższy zrzut ekranu pokazuje, jak wybrać plik konfiguracyjny Kubernetesa lub wprowadzić token uwierzytelniania:

3. Aby utworzyć nowy token uwierzytelniający użytkownika, wykonamy następujący skrypt w terminalu PowerShell:

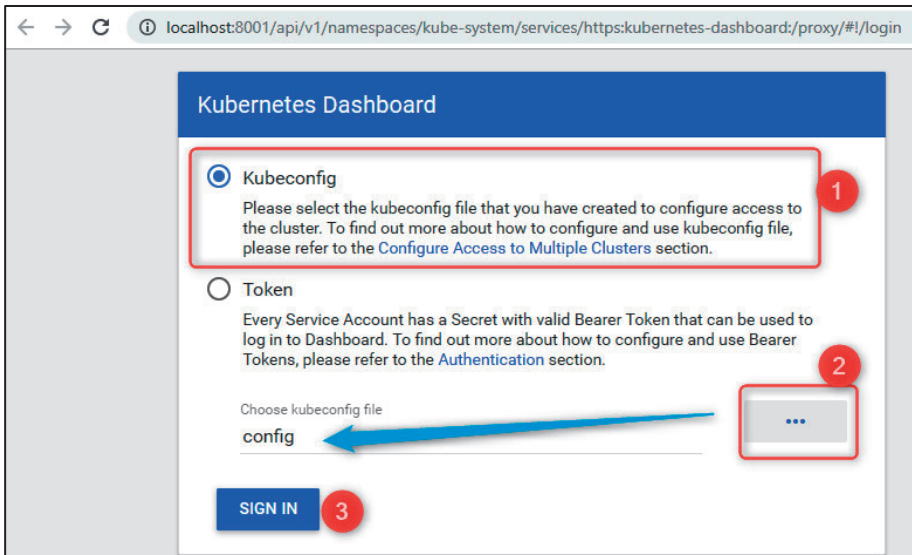
```
$TOKEN=((kubect1 -n kube-system describe secret default |
Select-String "token:") -split " ")[1]
kubect1 config set-credentials docker-for-desktop
--token="$TOKEN"
```

Wykonanie tego skryptu tworzy nowy token w lokalnym pliku konfiguracyjnym.



Rysunek 10.6. Uwierzytelnianie na pulpicie Kubernetesa

4. Na koniec na pulpicie wybierzemy plik konfiguracyjny, który znajduje się w folderze `C:\Users\<nazwa użytkownika>.kube\`, jak pokazano na poniższym zrzucie ekranu:

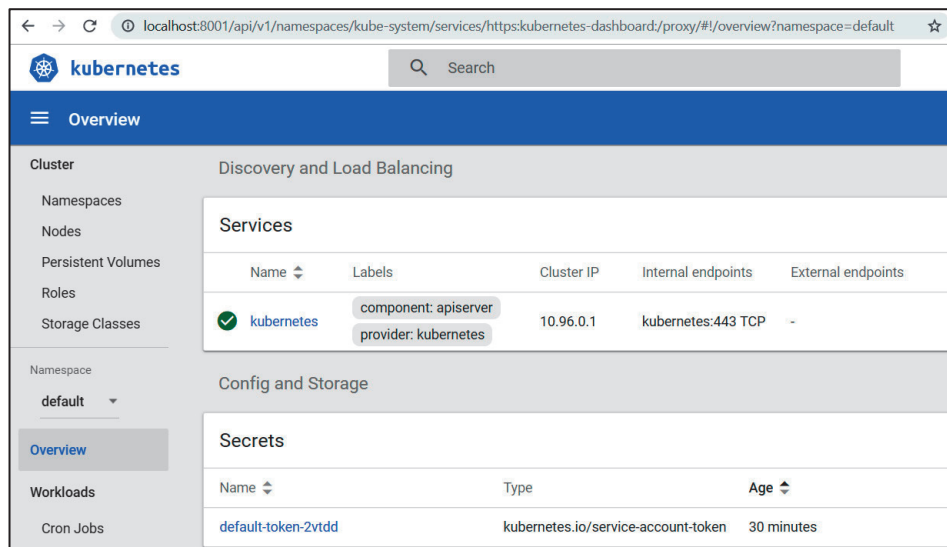


Rysunek 10.7. Uwierzytelnianie na pulpicie nawigacyjnym Kubernetesa za pomocą pliku kubeconfig

Uwaga

W przypadku uwierzytelniania tokenem przeczytaj ten wpis na blogu: <https://www.replex.io/blog/how-to-install-access-and-add-heapster-metrics-to-the-kubernetes-dashboard>.

5. Po kliknięciu przycisku *SIGN IN* dashboard wyświetli się w następujący sposób:



Rysunek 10.8. Lista zasobów pulpitu nawigacyjnego Kubernetesa

Właśnie dowiedzieliśmy się, jak zainstalować kłaster Kubernetesa na komputerze lokalnym, a następnie zainstalowaliśmy i skonfigurowaliśmy pulpit nawigacyjny Kubernetesa w tym klastrze. Teraz wdrożymy naszą pierwszą aplikację w lokalnym klastrze Kubernetesa przy użyciu plików **YAML** (ang. *YAML Ain't Markup Language*) i poleceń `kubectl`.

Pierwszy przykład wdrożenia aplikacji w Kubernetesie

Po zainstalowaniu naszego klastra Kubernetesa wdrożymy w nim aplikację. Przede wszystkim musimy wiedzieć, że wdrażając aplikację w Kubernetesie, tworzymy nową instancję kontenera Dockera w obiekcie *poda* Kubernetesa. Zatem najpierw musimy mieć obraz Dockera, który zawiera aplikację.

W naszym przykładzie użyjemy obrazu Dockera zawierającego aplikację webową, którą wysłaliśmy do Docker Huba w rozdziale 9, „Konteneryzacja aplikacji za pomocą Dockera”.

Aby wdrożyć tę instancję kontenera Dockera, utworzymy nowy folder *k8sdeploy*, a wewnątrz niego utworzymy plik YAML wdrożenia Kubernetesa (*myappdeployment.yml*) o następującej treści:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  selector:
    matchLabels:
      app: webapp
replicas: 2
template:
  metadata:
    labels:
      app: webapp
spec:
  containers:
    - name: demobookk8s
      image: mikaelkrief/demobook:latest
      ports:
        - containerPort: 80
```

W powyższym fragmencie kodu opisujemy nasze wdrożenie w następujący sposób:

- Właściwość *apiVersion* to wersja interfejsu API, której należy użyć.
- We właściwości *Kind* wskazujemy, że typem specyfikacji jest wdrożenie (ang. *deployment*).
- Właściwość *replicas* wskazuje liczbę podów, które Kubernetes utworzy w klastrze; tutaj wybieramy dwie instancje.

W tym przykładzie wybraliśmy dwie repliki, które mogą — co najmniej — rozłożyć obciążenie ruchu dla aplikacji (w przypadku dużego obciążenia możemy umieścić więcej replik), jednocześnie zapewniając prawidłowe działanie aplikacji. Dlatego jeśli jeden z dwóch podów ma problem, drugi (będący identyczną repliką) zapewni prawidłowe działanie aplikacji.

Następnie w sekcji *containers* wskazujemy obraz (z Docker Huba) wraz z nazwą (*name*) i tagiem (*tag*). Wreszcie właściwość *port* wskazuje port, którego kontener będzie używał w klastrze.

Uwaga

Ten kod źródłowy jest również dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP10/k8sdeploy/myapp-deployment.yml>.

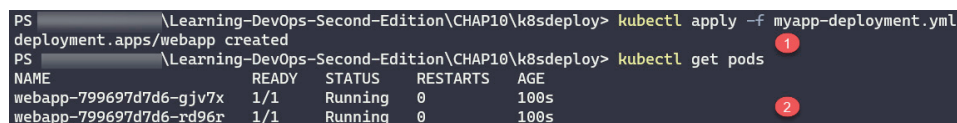
Aby wdrożyć naszą aplikację, przechodzimy do naszego terminala i wykonujemy jedno z podstawowych poleceń `kubectl` (`kubectl apply`) w następujący sposób:

```
kubectl apply -f myapp-deployment.yml
```

Parametr `-f` wskazuje plik YAML.

Polecenie to stosuje wdrożenie opisane w pliku YAML w klastrze Kubernetesa.

Po wykonaniu tego polecenia sprawdzimy stan wdrożenia, wyświetlając listę podów w klastrze. Aby to zrobić w terminalu, wykonujemy polecenie `kubectl get pods`, które zwraca listę podów klastra. Poniższy zrzut ekranu przedstawia wykonanie wdrożenia i wyświetla informacje w podach, których używamy do sprawdzenia wdrożenia:



```
PS \Learning-DevOps-Second-Edition\CHAP10\k8sdeploy> kubectl apply -f myapp-deployment.yml
deployment.apps/webapp created
PS \Learning-DevOps-Second-Edition\CHAP10\k8sdeploy> kubectl get pods
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-------------------------|-------|---------|----------|------|
| webapp-799697d7d6-gjv7x | 1/1 | Running | 0 | 100s |
| webapp-799697d7d6-rd96r | 1/1 | Running | 0 | 100s |

Rysunek 10.9. Polecenie `kubectl apply`

Na powyższym zrzucie ekranu widzimy, że drugie polecenie wyświetla nasze dwa pody z nazwą (`webapp`) określoną w pliku YAML, po której następuje **unikalny identyfikator (UID)**. Ponadto widzimy, że mają one stan *Running*.

Możemy również zwizualizować na pulpicie nawigacyjnym Kubernetesa stan naszego klastra, wdrożenie `webapp` z używanym obrazem platformy Docker i dwa utworzone pody. Aby uzyskać więcej informacji, możemy kliknąć różne linki elementów.

Nasza aplikacja została pomyślnie wdrożona w naszym klastrze Kubernetesa, ale na razie jest dostępna tylko wewnątrz klastra i żeby była użyteczna, musimy ją udostępnić poza klastrem.

Aby uzyskać dostęp do aplikacji webowej poza klastrem, musimy dodać do naszego klastra typ usługi i element `NodePort`. Aby dodać ten typ usługi i element `NodePort` w taki sam sposób jak dla wdrożenia, utworzymy drugi plik YAML (`myappservice.yml`) specyfikacji usługi w tym samym katalogu `k8sdeploy` z następującą zawartością:

```
---
apiVersion: v1
kind: Service
```



```
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
      nodePort: 31000
  selector:
    app: webapp
```

W powyższym fragmencie kodu określamy rodzaj usługi Service, a także typ usługi NodePort.

Następnie w sekcji ports określamy translację portu: port 80, który jest udostępniany wewnętrznie, i port 31000, który jest wystawiany zewnętrznie dla klastra.

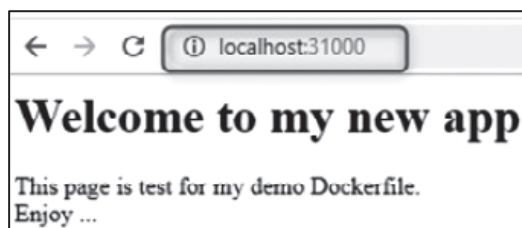
Uwaga

Kod źródłowy tego pliku jest również dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP10/k8sdeploy/myapp-service.yml>.

Aby utworzyć tę usługę w klastrze, wykonujemy polecenie `kubectl apply`, ale tym razem z naszym plikiem `myapp-service.yml` jako parametrem:

```
kubectl apply -f myapp-service.yml
```

Wykonanie polecenia tworzy usługę w ramach klastra, a do przetestowania naszej aplikacji otwieramy przeglądarkę internetową z adresem URL `http://localhost:31000`. Nasza strona wyświetla się w następujący sposób:



Rysunek 10.10. Demo aplikacji Kubernetesa

Nasza aplikacja jest teraz wdrożona w klastrze Kubernetesa i można do niej uzyskać dostęp spoza klastra.

W tej sekcji dowiedzieliśmy się, że wdrażanie aplikacji, a także tworzenie obiektów w Kubernetesie odbywa się za pomocą plików w formacie YAML i kilku poleceń `kubectl`.

Następnym krokiem jest użycie pakietów Helma w celu uproszczenia zarządzania plikami YAML.

Używanie Helma jako menedżera pakietów

Jak wspomniano wcześniej, wszystkie akcje, które wykonujemy na klastrze Kubernetesa, są wykonywane za pomocą narzędzia `kubectl` i **plików YAML**.

W firmie, która wdraża kilka aplikacji mikrousług w klastrze Kubernetesa, często zauważamy dużą liczbę plików YAML, co stwarza problem z ich utrzymaniem. Aby rozwiązać ten problem, możemy skorzystać z **Helma**, który jest menedżerem pakietów dla Kubernetesa.

Uwaga

Więcej informacji na temat menedżerów pakietów można również znaleźć w sekcji „Korzystanie z menedżera pakietów w procesie CI/CD” w rozdziale 7., „Ciągła integracja i ciągłe wdrażanie”.

Helm jest zatem repozytorium umożliwiającym udostępnianie pakietów zwanych **paczkami** (ang. *charts*), które zawierają gotowe do użycia szablony plików Kubernetesa.

Uwaga

Aby dowiedzieć się więcej na temat Helma i uzyskać dostęp do jego dokumentacji, odwiedź stronę <https://helm.sh/>.

Instalacja klienta Helma

Zobaczymy, jak zainstalować Helma w naszym lokalnym klastrze Kubernetesa, a później przejdziemy przez instalację aplikacji za jego pomocą.

Od wersji 3 Helm składa się tylko z jednego pliku binarnego: **narzędzia klienckiego**, które pozwala nam głównie instalować pakiety plików specyfikacji Kubernetesa na docelowym klastrze, wyświetlać listę pakietów repozytorium i wskazywać pakiety, które mają być zainstalowane.

Uwaga

W pierwszym wydaniu tej książki używaliśmy wersji Helma niższej niż 3.0 i nauczyliśmy się instalować wtyczkę Helm Tiller. Od wersji 3.0+ wtyczka Tiller nie musi być instalowana, więc instrukcja instalacji Tillera została usunięta w tym drugim wydaniu.

Aby zainstalować klienta Helma, zapoznaj się z dokumentacją instalacji pod adresem https://helm.sh/docs/using_helm/#installing-the-helm-client, w której szczegółowo opisano procedurę instalacji dla różnych systemów operacyjnych.

Na przykład w systemie Windows możemy go zainstalować za pomocą menedżera pakietów Chocolatey, wykonując następujące polecenie:

```
choco install kubernetes-helm -y
```

Aby sprawdzić poprawność instalacji, wykonaj polecenie `helm --help`, jak pokazano na poniższym zrzucie ekranu:

```
PS C:\Users\mkrief> helm --help
The Kubernetes package manager

Common actions for Helm:

- helm search:      search for charts
- helm pull:        download a chart to your local directory to view
- helm install:     upload the chart to Kubernetes
- helm list:        list releases of charts
```

Rysunek 10.11. Polecenie `helm --help`

Wykonanie polecenia mówi nam, że Helm jest poprawnie zainstalowany. Teraz dowiemy się, jak korzystać z publicznego pakietu Helma.

Korzystanie z publicznego pakietu Helma, dostępnego w Artifact Hubie

Pakiety zawarte w repozytorium Helma nazywane są **chartami**. Charty składają się z plików, które są szablonami plików Kubernetesa dla aplikacji.

Dzięki chartom możliwe jest wdrożenie aplikacji w Kubernetesie bez konieczności pisania jakichkolwiek plików YAML. Tak więc aby wdrożyć aplikację, użyjemy odpowiadającego jej charta i prześlemy jej kilka zmiennych konfiguracyjnych.

Po zainstalowaniu Helma zainstalujemy chart, który znajduje się w publicznym repozytorium Helma o nazwie **Artifact Hub**, dostępnym tutaj: <https://artifacthub.io/>. Ale najpierw, żeby wyświetlić listę publicznych chartów, uruchamiamy następujące polecenie:

helm search hub

Parametr `hub` określa nazwę w Artifact Hubie.

By wyszukać konkretny pakiet, możemy uruchomić polecenie `helm search hub <nazwa pakietu>`, a jeśli chcemy znaleźć np. wszystkie pakiety `wordpress`, uruchamiamy następujące polecenie:

```
helm search hub wordpress
```

Oto wynik wykonania polecenia, który zawiera wiele chartów:

```
PS C:\Users\mkrief> helm search hub wordpress
```

| URL | CHART VERSION | APP VERSION | DESCRIPTION |
|---|----------------|---------------------|--|
| https://artifacthub.io/packages/helm/kube-wordpress | 0.1.0 | 1.1 | this is my wordpress package |
| https://artifacthub.io/packages/helm/bitnami/wordpress | 12.2.3 | 5.8.2 | Web publishing platform for building blogs and ... |
| https://artifacthub.io/packages/helm/bitnami/wordpress | 12.2.3 | 5.8.2 | Web publishing platform for building blogs and ... |
| https://artifacthub.io/packages/helm/groundhops/wordpress | 0.4.0 | 5.8.2-apache | A Helm chart for Wordpress on Kubernetes |
| https://artifacthub.io/packages/helm/riftbit/wordpress | 12.1.16 | 5.8.1 | Web publishing platform for building blogs and ... |
| https://artifacthub.io/packages/helm/homeenterp/wordpress | 0.1.0 | 5.8.0-php8.0-apache | Blog server |
| https://artifacthub.io/packages/helm/mcouliba/wordpress | 0.1.0 | 1.16.0 | A Helm chart for Kubernetes |
| https://artifacthub.io/packages/helm/securecode/wordpress | 3.4.0 | 4.0 | Insecure & Outdated Wordpress Instance: Never e... |
| https://artifacthub.io/packages/helm/wordpress/wordpress | 1.0.0 | | This is the Helm Chart that creates the Wordpre... |
| https://artifacthub.io/packages/helm/bitpoke/wordpress | 0.11.1 | 0.11.1 | Bitpoke Wordpress Operator Helm Chart |
| https://artifacthub.io/packages/helm/presslabs/wordpress | 0.11.0-alpha.3 | 0.11.0-alpha.3 | Presslabs Wordpress Operator Helm Chart |
| https://artifacthub.io/packages/helm/presslabs/wordpress | 0.12.0-rc.2 | v0.12.0-rc.2 | A Helm chart for deploying a WordPress site on ... |
| https://artifacthub.io/packages/helm/phntom/bin/wordpress | 0.0.3 | 0.0.3 | www.binaryvision.co.il static wordpress |
| https://artifacthub.io/packages/helm/gh-shessel/wordpress | 1.0.35 | 5.8.2 | Web publishing platform for building blogs and ... |
| https://artifacthub.io/packages/helm/sonu-wordpress/wordpress | 1.0.0 | 2 | This is my custom chart to deploy wordpress and... |
| https://artifacthub.io/packages/helm/uvaize-wordpress/wordpress | 0.2.0 | 1.1.0 | Wordpress for Kubernetes |
| https://artifacthub.io/packages/helm/wordpress/wordpress | 0.2.0 | 1.1.0 | Wordpress for Kubernetes |
| https://artifacthub.io/packages/helm/wordpress/wordpress | 1.0.0 | | This is my custom chart to deploy wordpress and... |
| https://artifacthub.io/packages/helm/bitpoke/stack | 0.11.0-rc.3 | 0.11.0-rc.3 | Your Open-Source, Cloud-Native WordPress Infras... |
| https://artifacthub.io/packages/helm/securecode/wordpress | 3.4.0 | W3.8.20 | A Helm chart for the Wordpress security scanner... |
| https://artifacthub.io/packages/helm/viveksahu2/wordpress | 1.0.0 | 2 | This is my custom chart to deploy wordpress and... |
| https://artifacthub.io/packages/helm/presslabs/wordpress | 0.11.0-rc.2 | v0.11.0-rc.2 | Open-Source Wordpress Infrastructure on Kubernetes |
| https://artifacthub.io/packages/helm/presslabs/wordpress | 0.12.0-rc.2 | v0.12.0-rc.2 | Open-Source Wordpress Infrastructure on Kubernetes |
| https://artifacthub.io/packages/helm/six/wordpress | 0.2.0 | 1.1.0 | Wordpress for Kubernetes |
| https://artifacthub.io/packages/helm/jinchi-chart/wordpress | 0.2.0 | 1.1.0 | Wordpress for Kubernetes |
| https://artifacthub.io/packages/helm/wordpress/wordpress | 0.1.0 | 1.1 | Wordpress for Kubernetes |
| https://artifacthub.io/packages/helm/presslabs/wordpress | 0.11.6 | 0.11.6 | Presslabs Wordpress Operator Helm Chart |

Rysunek 10.12. Wyszukiwanie pakietów Helma

Aby łatwo znaleźć pakiety, przejdź do witryny Artifact Hub (<https://artifacthub.io/>) i wyszukaj pakiet `wordpress`, jak pokazano na rysunku 10.13.

Ta strona zawiera listę wszystkich pakietów `wordpress` od wszystkich wydawców.

Następnie kliknij żądany pakiet, aby wyświetlić szczegóły pakietu, jak pokazano na rysunku 10.14.

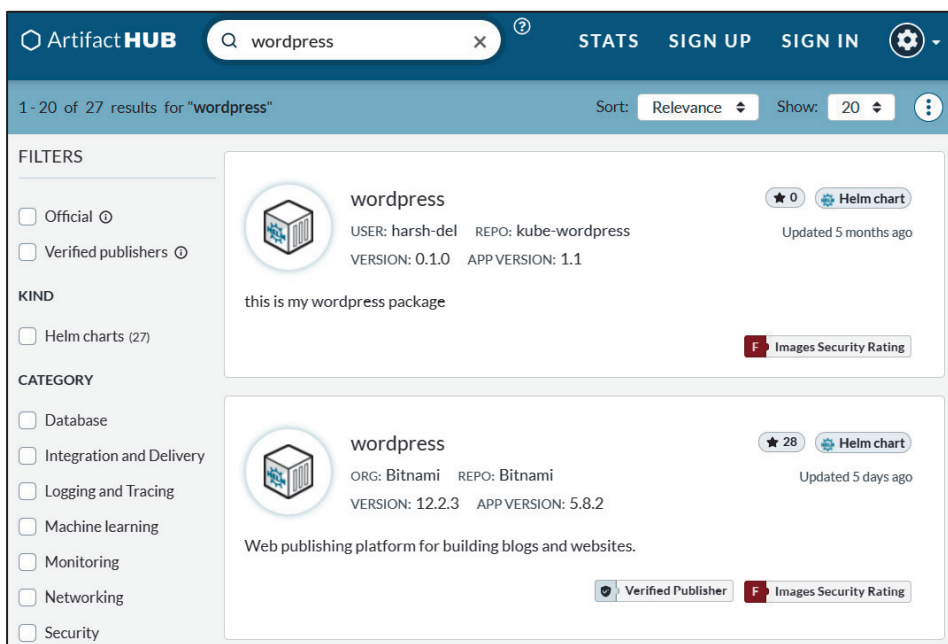
Na tej stronie możemy zobaczyć aktualną wersję pakietu i opis instalacji pakietu.

Zainstalujmy teraz aplikację za pomocą Helma.

Uwaga

Możliwe jest również utworzenie prywatnego lub korporacyjnego repozytorium Helma za pomocą narzędzi takich jak Nexus, Artifactory, a nawet ACR.

Aby zilustrować użycie Helma, wdrożymy aplikację `WordPress` w naszym klastrze `Kubernetesa` za pomocą charta Helma.



Rysunek 10.13. Wyszukiwanie pakietu wordpress w Artifact Hubie



Rysunek 10.14. Szczegóły pakietu wordpress z Artifact Huba

W tym celu wykonaj następujące polecenia (wymienione tutaj: <https://artifacthub.io/packages/helm/bitnami/wordpress>):

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install wpdemo bitnami/wordpress
```

Pierwsze polecenie, `helm repo add`, dodaje lokalnie indeks repozytorium `bitnami`. Następnie używamy polecenia `helm install <nazwa wydania> <nazwa pakietu>`, aby zainstalować żądany pakiet w Kubernetesie.

Poniższy zrzut ekranu pokazuje wykonanie tych dwóch poleceń:

```
PS C:\Users\mkrief> helm repo add bitnami https://charts.bitnami.com/bitnami
"bitnami" has been added to your repositories
PS C:\Users\mkrief> helm install wpdemo bitnami/wordpress
NAME: wpdemo
LAST DEPLOYED: Sat Dec 4 19:29:46 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: wordpress
CHART VERSION: 12.2.3
APP VERSION: 5.8.2

** Please be patient while the chart is being deployed **

Your WordPress site can be accessed through the following DNS name from within your cluster:

    wpdemo-wordpress.default.svc.cluster.local (port 80)

To access your WordPress site from outside the cluster follow the steps below:

1. Get the WordPress URL by running these commands:

    NOTE: It may take a few minutes for the LoadBalancer IP to be available.
    Watch the status with: 'kubectl get svc --namespace default -w wpdemo-wordpress'

    export SERVICE_IP=$(kubectl get svc --namespace default wpdemo-wordpress --template "{{ range (index .status.loadBalancer.ingress 0) }}{{.}}{{ end }}" )
    echo "WordPress URL: http://$SERVICE_IP/"
    echo "WordPress Admin URL: http://$SERVICE_IP/admin"

2. Open a browser and access WordPress using the obtained URL.

3. Login with the following credentials below to see your blog:

    echo Username: user
    echo Password: $(kubectl get secret --namespace default wpdemo-wordpress -o jsonpath="{.data.wordpress-password}" | base64 --decode)
```

Rysunek 10.15. Instalowanie aplikacji za pomocą Helma

Po wykonaniu powyższych poleceń Helm instaluje instancję WordPressa o nazwie `wpdemo` i wszystkie składniki Kubernetesa w lokalnym klastrze.

Możemy również wyświetlić listę pakietów Helma, które są zainstalowane w klastrze, wykonując następujące polecenie:

```
helm ls
```

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia:

```
PS C:\Users\mkrief> helm ls
NAME      NAMESPACE    REVISION    UPDATED           STATUS      CHART              APP VERSION
wpdemo    default       1           2021-12-04 19:29:46.8115125 +0100 CET    deployed   wordpress-12.2.3   5.8.2
```

Rysunek 10.16. Lista zainstalowanych pakietów Helma

Jeśli chcemy usunąć pakiet i wszystkie jego składniki (np. aby usunąć aplikację zainstalowaną z tym pakietem), wykonujemy polecenie `helm delete` w następujący sposób:

```
helm delete wpdemo
```

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia:

```
PS C:\Users\mkrief> helm delete wpdemo  
release "wpdemo" uninstalled
```

Rysunek 10.17. Polecenie helm delete

Omówiliśmy instalację charta Helma z Artifact Huba w klastrze Kubernetesa. W następnej sekcji dowiemy się, jak utworzyć niestandardowy pakiet charta.

Tworzenie niestandardowego charta Helma

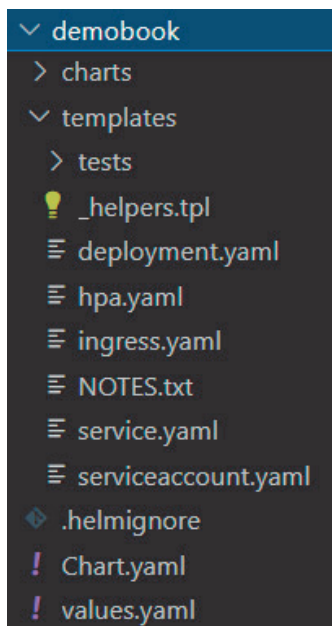
Właśnie nauczyliśmy się instalować publiczny chart Helma z Artifact Huba, ale w firmach często mamy niestandardowe aplikacje, które wymagają od nas tworzenia niestandardowych chartów Helma.

Oto podstawowe kroki, aby utworzyć niestandardowy chart Helma:

1. Wewnątrz folderu, który będzie zawierał szablon pliku charta Helma, uruchom polecenie `helm create <nazwa charta>` w następujący sposób:

```
helm create demobook
```

Wykonanie tego polecenia utworzy strukturę katalogów i podstawowe pliki szablonów dla naszego charta, jak pokazano na poniższym zrzucie ekranu:



Rysunek 10.18. Folder struktury charta Helma

2. Następnie dostosuj szablony i wartości charta, postępując zgodnie z dokumentacją techniczną: https://helm.sh/docs/chart_template_guide/.
3. Na koniec opublikuj chart w naszym klastrze Kubernetesa, uruchamiając polecenie `helm install <nazwa charta> <ścieżka do charta>` w następujący sposób:

```
helm install demochart ./demobook
```

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia:

```
PS Learning-DevOps-Second-Edition\CHAP10> helm install demochart ./demobook
NAME: demochart
LAST DEPLOYED: Sun Dec 5 17:16:19 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l "app.kubernetes.io/name=demobook,app.kubernetes.io/instance=demochart" -o
  jsonpath="{.items[0].metadata.name}")
  export CONTAINER_PORT=$(kubectl get pod --namespace default $POD_NAME -o jsonpath="{.spec.containers[0].ports[0].containerPort}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl --namespace default port-forward $POD_NAME 8080:$CONTAINER_PORT
```

Rysunek 10.19. Polecenie `helm install`

Aby sprawdzić instalację pakietu za pomocą Helma, uruchom polecenie `kubectl get pods`, by wyświetlić listę utworzonych podów.

Poniższy zrzut ekranu przedstawia listę utworzonych podów:

```
PS Learning-DevOps-Second-Edition\CHAP10> kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
demochart-demobook-77458bfd9f-h4mwv 1/1     Running   0           104s
```

Rysunek 10.20. `kubectl get pods`

Uruchom także polecenie `helm ls`, aby wyświetlić listę zainstalowanych chartów Helma, jak pokazano na poniższym zrzucie ekranu:

```
PS Learning-DevOps-Second-Edition\CHAP10> helm ls
NAME      NAMESPACE   REVISION   UPDATED                     STATUS   CHART          APP VERSION
demochart default      1          2021-12-05 17:16:19.0370031 +0100 CET  deployed demobook-0.1.0 1.16.0
```

Rysunek 10.21. Lista pakietów Helma

Na powyższym zrzucie ekranu widzimy chart `demochart`, który jest zainstalowany w klastrze Kubernetesa.

W tej sekcji zapoznaliśmy się z instalacją i użytkowaniem Helma, który jest menedżerem pakietów dla Kubernetesa. Następnie dowiedzieliśmy się, jak zainstalować chart Helma z repozytorium Artifact Hub. Na koniec utworzyliśmy niestandardowy chart i zainstalowaliśmy go w klastrze Kubernetesa.

W następnej sekcji dowiemy się, jak opublikować niestandardowy chart Helma w prywatnym repozytorium — czyli ACR.

Publikowanie charta Helma w rejestrze prywatnym (ACR)

W poprzedniej sekcji omówiliśmy publiczne repozytorium Helma o nazwie Artifact Hub, które doskonale nadaje się do publicznych (lub społecznościowych) aplikacji lub narzędzi. Ale w przypadku aplikacji firmowych lepsze (i zalecane) jest posiadanie prywatnego rejestru Helma.

Na rynku istnieje wiele prywatnych rejestrów, takich jak Nexus lub Artifactory. Ta dokumentacja wyjaśnia, jak utworzyć prywatne repozytorium Helma: https://helm.sh/docs/topics/chart_repository/.

W tej sekcji powiemy, jak korzystać z prywatnego repozytorium Helma (czyli ACR, o którym dowiedzieliśmy się już w rozdziale 9., „Konteneryzacja aplikacji za pomocą Dockera”). Przed rozpoczęciem naszego przykładu zakładamy, że utworzyliśmy już listę ACR o nazwie demobookacr na platformie Azure.

Uwaga

Jeśli nie utworzyłeś jeszcze ACR, postępuj zgodnie z dokumentacją tutaj: <https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-azure-cli>.

Aby opublikować niestandardowy chart Helma w ACR, wykonaj następujące kroki:

1. Pierwszym krokiem jest utworzenie pakietu charta w formacie `tar.gz`, uruchamiając polecenie `helm package` w folderze, w którym znajduje się plik `chart.yaml`.

Uruchom następujące polecenie z `.` jako parametrem, wskazując, że jest to chart.

Plik `yaml` znajduje się w bieżącym folderze:

`helm package .`

Poniższy zrzut ekranu pokazuje wykonanie tego polecenia:

```
PS > .\Learning-DevOps-Second-Edition\CHAP10\demobook> helm package .
Successfully packaged chart and saved it to: .\Learning-DevOps-Second-Edition\CHAP10\demobook\demobook-0.1.0.tgz
```

Rysunek 10.22. Tworzenie pakietu Helma

Pakiet wykresów Helma jest tworzony pod nazwą *demobook-0.1.0.tgz*.

2. Następnie uwierzytelnij się w ACR za pomocą następującego skryptu PowerShell:

```
$env:HELM_EXPERIMENTAL_OCI=1
$USER_NAME="00000000-0000-0000-0000-000000000000"
$ACR_NAME="demobookacr"
az login
$PASSWORD=$(az acr login --name $ACR_NAME --expose-token --output tsv
↳--query accessToken)
helm registry login "$ACR_NAME.azurecr.io" --username $USER_NAME
↳--password "$PASSWORD"
```

Powyższy skrypt wykonuje te operacje:

- Ustawia zmienną środowiskową `HELM_EXPERIMENTAL_OCI` na 1. Więcej informacji na temat zmiennych środowiskowych można znaleźć w tej dokumentacji: <https://helm.sh/docs/topics/registries/>.
 - Ustawia zmienną `USER_NAME` na przypadkową wartość 000.
 - Ustawia zmienną `ACR_NAME` na nazwę ACR.
 - Ustawia dynamicznie zmienną `PASSWORD` za pomocą polecenia `az cli`.
 - Używa loginu rejestru Helma do uwierzytelniania w rejestrze ACR.
3. Na koniec przesyłamy chart Helma do rejestru ACR za pomocą następującego polecenia:

```
helm push .\demobook-0.1.0.tgz oci://$ACR_NAME.azurecr.io/helm
```

Po wykonaniu tego polecenia wysłany chart Helma znajdzie się w repozytorium *helm/demobook* w ACR.

4. Możemy sprawdzić, czy chart Helma jest poprawnie przesłany do ACR, uruchamiając następujące polecenie:

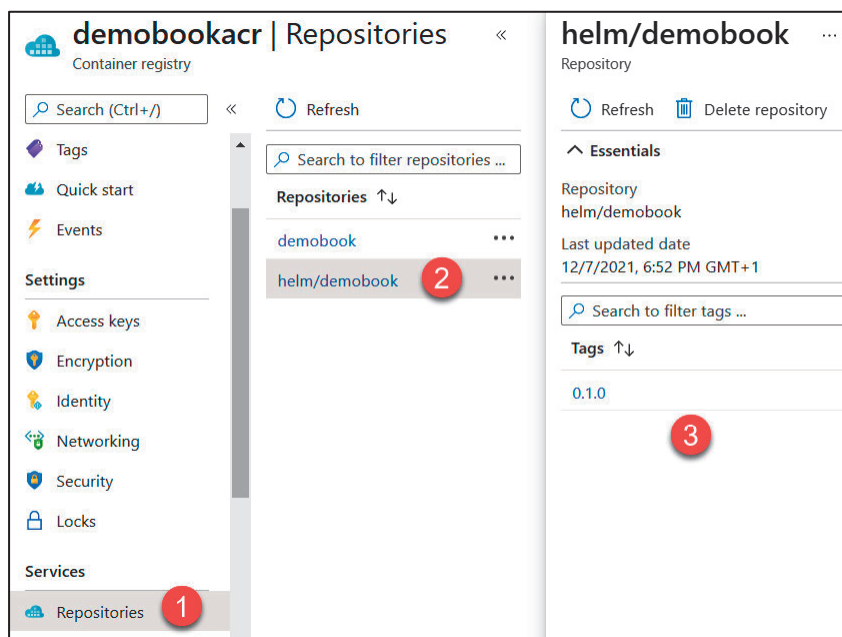
```
az acr repository show --name $ACR_NAME --repository helm/demobook
```

Wykonanie tego polecenia wyświetla szczegółowe informacje na temat charta *helm/demobook*.

W portalu Azure możemy zobaczyć repozytorium Helma, chart i tag, jak pokazano na rysunku 10.23.

Właśnie dowiedzieliśmy się, jak tworzyć i wysyłać niestandardowy chart Helma do ACR za pomocą wiersza poleceń Helma i poleceń `az cli`.

W następnej sekcji przyjrzymy się przykładowi zarządzanej usługi Kubernetesa hostowanej na platformie Azure o nazwie AKS.



Rysunek 10.23. Repozytorium Helma w ACR

Korzystanie z AKS

Produkcyjny klaster Kubernetesa często może być skomplikowany w instalacji i konfiguracji. Ten rodzaj instalacji wymaga dostępności serwerów, zasobów ludzkich posiadających wymagane umiejętności dotyczące zarządzania klastrem Kubernetesa, a zwłaszcza wdrożenia zaawansowanej polityki bezpieczeństwa w celu ochrony aplikacji.

Aby rozwiązać te problemy, dostawcy chmury oferują zarządzane usługi klastrowe Kubernetesa. Tak jest w przypadku Amazona, który oferuje **Elastic Kubernetes Service (EKS)**, Google z **Google Kubernetes Engine (GKE)** i wreszcie Azure z AKS. W tej sekcji proponuję omówienie AKS, jednocześnie podkreślając zalety zarządzanego klastra Kubernetesa.

AKS jest usługą platformy Azure, która umożliwia nam tworzenie prawdziwego klastra Kubernetesa i zarządzanie nim.

Zaletą tego zarządzanego klastra Kubernetesa jest to, że nie musimy się martwić o jego instalację sprzętową, a zarządzanie częścią główną odbywa się w całości przez platformę Azure, gdy węzły są instalowane na **maszynach wirtualnych (VM)**.

Korzystanie z tej usługi jest bezpłatne; naliczany jest koszt maszyn wirtualnych, na których zainstalowane są węzły.

Uwaga

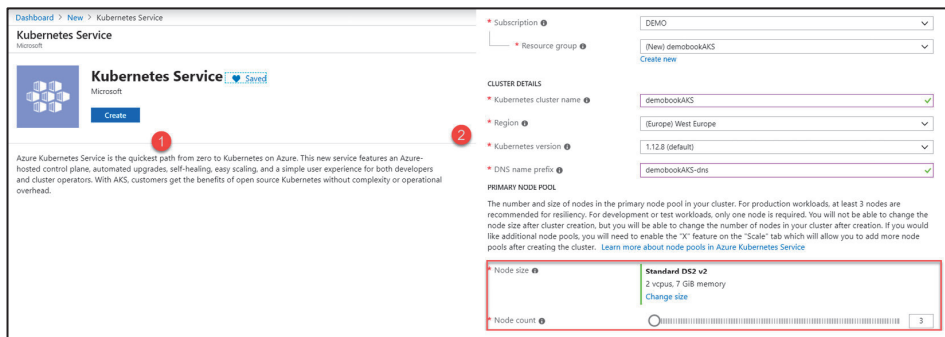
Aby dowiedzieć się więcej o korzyściach oferowanych przez AKS, zapoznaj się z dokumentacją pod adresem <https://docs.microsoft.com/en-us/azure/aks/intro-kubernetes>.

Przyjrzyjmy się teraz, jak utworzyć usługę AKS.

Tworzenie usługi AKS

Tworzenie klastra AKS na platformie Azure można wykonać na trzy różne sposoby, jak opisano tutaj:

- **Ręcznie, za pośrednictwem portalu Azure.** Standardowym sposobem utworzenia usługi AKS jest zrobienie tego za pośrednictwem portalu Azure, tworząc usługę Kubernetesa, a następnie wprowadzając podstawowe właściwości Azure — czyli typ i liczbę żądanych węzłów, jak pokazano na poniższym zrzucie ekranu:



Rysunek 10.24. Tworzenie AKS za pośrednictwem portalu Azure

- **Tworzenie za pomocą skryptu az cli.** Możesz również skorzystać ze skryptu az cli, by zautomatyzować tworzenie klastra AKS. Skrypt jest pokazany tutaj:

```
#Utwórz grupę zasobów
az group create --name Rg-AKS --lokalizacja westeurope
#Utwórz zasób AKS
az aks create --resource-group Rg-AKS --name demoBookAKS --node-count 2
--generate-ssh-keys --enable-addons monitoring
```

Właściwość node-count wskazuje liczbę węzłów, a właściwość enableaddons umożliwia nam monitorowanie usługi AKS.

- **Tworzenie za pomocą Terraform.** Możliwe jest również utworzenie usługi AKS za pomocą Terraform. Kompletny skrypt Terraform jest dostępny w dokumentacji platformy Azure pod adresem <https://docs.microsoft.com/en-us/azure/terraform/terraform-create-k8s-cluster-with-tf-and-aks>. Aby dowiedzieć się więcej o korzystaniu z Terraform, można o tym przeczytać w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”.

Po utworzeniu klastra AKS będziemy mogli skonfigurować plik *kubeconfig*, aby połączyć się z tym klastrem AKS.

Konfigurowanie pliku kubeconfig dla AKS

Aby skonfigurować plik *kubeconfig* używany przez *kubectl* do łączenia się z usługą AKS, użyjemy narzędzia *az cli*, wykonując następujące polecenia w terminalu:

```
az login
#Jeśli masz kilka subskrypcji platformy Azure
az account set --subscription <identyfikator subskrypcji>
az aks get-credentials --resource-group Rg-AKS --name demoBookAKS
```

To ostatnie polecenie przyjmuje jako parametry grupę zasobów i nazwę utworzonego klastra AKS. Rolą tego polecenia jest automatyczne utworzenie pliku *.kube\config*, który jest używany przez *kubectl* do połączenia z klastrem AKS, jak pokazano na poniższym zrzucie ekranu:

```
C:\> az aks get-credentials --resource-group Rg-AKS --name demoBookAKS
Merged "demoBookAKS" as current context in .kube\config
```

Rysunek 10.25. AKS pobiera dane uwierzytelniające za pośrednictwem polecenia *az cli*

Aby przetestować połączenie z AKS, możemy wykonać polecenie *kubectl get nodes*, które wyświetla liczbę węzłów skonfigurowanych podczas tworzenia klastra AKS, jak pokazano na poniższym zrzucie ekranu:

```
>kubectl get nodes
```

| NAME | STATUS | ROLES | AGE | VERSION |
|---------------------------|--------|-------|-----|---------|
| aks-nodepool11-41966373-0 | Ready | agent | 8m | v1.12.8 |
| aks-nodepool11-41966373-1 | Ready | agent | 8m | v1.12.8 |

Rysunek 10.26. Lista zwracana przez polecenie *kubectl get nodes*

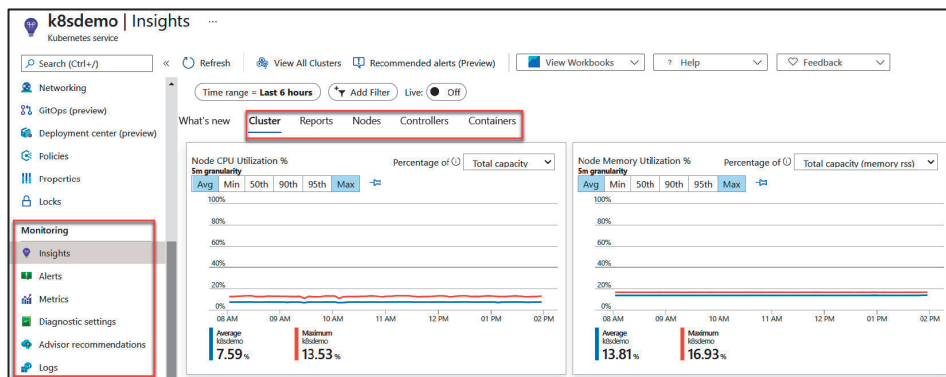
Wszystkie operacje, które widzieliśmy w sekcji „Pierwszy przykład wdrożenia aplikacji w Kubernetesie” w tym rozdziale, są identyczne, niezależnie od tego, czy wdrażasz aplikację za pomocą AKS, czy za pomocą *kubectl*.

Po zapoznaniu się z czynnościami podejmowanymi w celu utworzenia usługi AKS na platformie Azure przedstawimy zalety tego rozwiązania.

Zalety AKS

AKS to usługa Kubernetesa zarządzana na platformie Azure. Dzięki temu jest zintegrowana z Azure. Ma także kilka innych zalet:

- **Gotowość do użycia** — w AKS webowy pulpit nawigacyjny Kubernetesa jest natywnie zainstalowany, a dokumentacja zawarta pod adresem <https://docs.microsoft.com/en-us/azure/aks/kubernetes-dashboard> wyjaśnia, jak uzyskać do niego dostęp.
- **Zintegrowane usługi monitorowania** — AKS ma również wszystkie zintegrowane usługi monitorowania platformy Azure, w tym monitorowanie kontenerów, zarządzanie wydajnością klastra i zarządzanie dziennikami, jak pokazano na poniższym zrzucie ekranu:



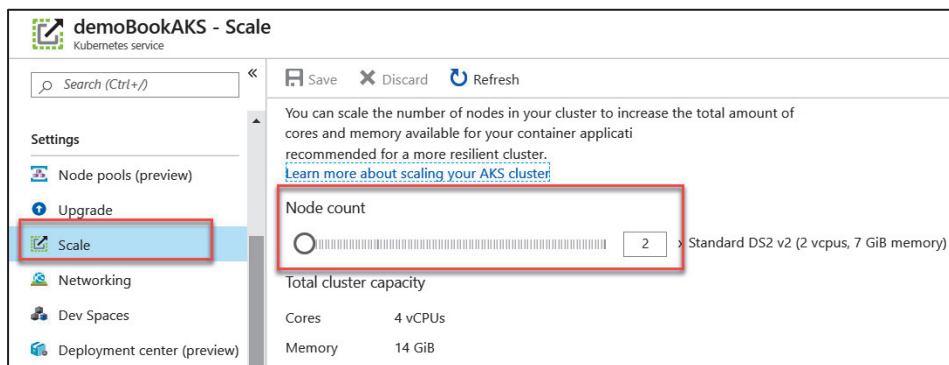
Rysunek 10.27. Monitorowanie AKS

- **Łatwość skalowania** — AKS umożliwia szybkie i bezpośrednie skalowanie liczby węzłów klastra za pośrednictwem portalu lub skryptów.

Jak widać na poniższym zrzucie ekranu, w portalu Azure wybieramy liczbę węzłów, a zmiana zaczyna obowiązywać od razu.

Jeśli posiadamy subskrypcję Azure i chcemy korzystać z Kubernetesa, instalacja jest intuicyjna i szybka. AKS ma wiele zalet, np. zintegrowane monitorowanie i skalowanie w Azure. Korzystanie z narzędzia `kubectl` nie wymaga żadnych zmian w porównaniu z lokalną instancją Kubernetesa.

W tej sekcji omówiliśmy AKS, która jest zarządzaną usługą Kubernetesa na platformie Azure. Następnie utworzyliśmy instancję AKS i skonfigurowaliśmy plik `kubeconfig`, aby



Rysunek 10.28. Skalowanie AKS

nawiązać z nią połączenie. Na koniec wymieniliśmy jej zalety, którymi są głównie zintegrowane monitorowanie i szybka skalowalność.

W następnej sekcji poznamy niektóre zasoby konieczne do wdrażania aplikacji w Kubernetesie przy użyciu potoku CI/CD z Azure Pipelines.

Tworzenie potoku CI/CD dla Kubernetesa za pomocą Azure Pipelines

Do tej pory widzieliśmy, jak używać `kubectl` do wdrażania aplikacji kontenerowej w lokalnym klastrze Kubernetesa lub w klastrze zdalnym z AKS.

W pierwszym wydaniu tej książki wyjaśniłem, jak zbudować kompletny potok w Azure DevOps, od utworzenia nowego obrazu platformy Docker wysłanego do Docker Huba aż do jego wdrożenia w klastrze AKS.

Od czasu pierwszej edycji wiele funkcji służących do wdrożenia w Kubernetesie zostało ulepszonych w różnych narzędziach CI/CD.

Dlatego w tym drugim wydaniu nie będę już wyjaśniał tego szczegółowo, ale udostępnię rozmaite materiały przydatne w mojej codziennej pracy:

- Pierwszy zasób to świetny, kompletny film, który wyjaśnia wszystkie szczegóły dotyczące wdrażania aplikacji w AKS w Azure DevOps. Można go znaleźć pod następującym linkiem:

<https://www.youtube.com/watch?v=K4uNl6JA7g8>

- Ten sam temat omawia przykład dotyczący Azure DevOps pod następującym linkiem:

<https://www.azuredevopslabs.com/labs/vstsextend/kubernetes/>

- Aby utworzyć potok Azure DevOps w formacie YAML do wdrożenia w Kubernetesie, przeczytaj oficjalną dokumentację tutaj:

<https://docs.microsoft.com/en-us/azure/devops/pipelines/ecosystems/kubernetes/aks-template?view=azure-devops>

- Poniższa dokumentacja omawia samouczek wykorzystujący Jenkinsa i Azure DevOps w celu wdrożenia aplikacji na Kubernetesie:

<https://docs.microsoft.com/en-us/azure/devops/pipelines/release/integrate-jenkins-pipelines-aks?view=azure-devops>

W tej sekcji omówiliśmy niektóre zasoby wymagane dla **kompleksowego** (ang. *end-to-end* — E2E) potoku DevOps CI/CD w celu wdrożenia aplikacji w klastrze Kubernetesa (w naszym przykładzie AKS) za pomocą Azure Pipelines.

W następnej sekcji nauczymy się różnych sposobów monitorowania aplikacji i metryk w Kubernetesie oraz dowiemy się więcej na temat narzędzi, których możemy do tego użyć.

Monitorowanie aplikacji i metryk w Kubernetesie

Kiedy wdrażamy aplikację w Kubernetesie, bardzo ważne jest — i uważam to za wymóg — posiadanie strategii monitorowania w celu sprawdzania i debugowania cyklu życia tych aplikacji, a także **kontrola procesora (CPU)** i **pamięci (RAM)**.

Omówimy teraz różne sposoby debugowania i monitorowania aplikacji w Kubernetesie.

Zacznijmy od podstawowego sposobu, jakim jest użycie wiersza poleceń `kubectl`.

Korzystanie z wiersza poleceń `kubectl`

Aby debugować aplikacje za pomocą wiersza poleceń `kubectl`, uruchom poniższe polecenia:

- Aby wyświetlić stan zasobów Kubernetesa, uruchom następujące polecenie:

```
kubectl get pods, svc
```

Dane wyjściowe tego polecenia pokazano na poniższym zrzucie ekranu:


```
PS C:\Users\mkrief> kubectl get pods,svc
```

| NAME | READY | STATUS | RESTARTS | AGE |
|---|-------|---------|---------------|------|
| pod/demochart-demobook-77458bfd9f-h4mwv | 1/1 | Running | 2 (3m27s ago) | 6d4h |
| pod/webapp-799697d7d6-gjv7x | 1/1 | Running | 2 (3m27s ago) | 6d7h |
| pod/webapp-799697d7d6-rd96r | 1/1 | Running | 2 (3m27s ago) | 6d7h |

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|----------------------------|-----------|----------------|-------------|--------------|------|
| service/demochart-demobook | ClusterIP | 10.111.122.236 | <none> | 80/TCP | 6d4h |
| service/kubernetes | ClusterIP | 10.96.0.1 | <none> | 443/TCP | 8d |
| service/webapp | NodePort | 10.110.12.30 | <none> | 80:31000/TCP | 6d6h |

Rysunek 10.29. kubectl pobiera zasoby

Za pomocą tego polecenia możemy sprawdzić, czy pody są uruchomione, i dowiedzieć się o statusach usług.

- Aby wyświetlić dzienniki aplikacji, uruchom polecenie `kubectl logs pod/<nazwa_poda>`, jak pokazano na poniższym zrzucie ekranu:

```
PS C:\Users\mkrief> kubectl logs pod/webapp-799697d7d6-gjv7x
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 10.1.0.35. Set the 'ServerName' directive globally to suppress this message
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 10.1.0.35. Set the 'ServerName' directive globally to suppress this message
[Sat Dec 11 20:57:07.523851 2021] [mpm_event:notice] [pid 1:tid 140521422701696] AH00489: Apache/2.4.41 (Unix) configured -- resuming normal operations
[Sat Dec 11 20:57:07.532170 2021] [core:notice] [pid 1:tid 140521422701696] AH00094: Command line: 'httpd -D FOREGROUND'
```

Rysunek 10.30. kubectl pobiera logi poda

Polecenie to wyświetla dane wyjściowe dzienników aplikacji.

Aby uzyskać więcej informacji na temat używania `kubectl` do debugowania aplikacji, zapoznaj się z poniższą dokumentacją: <https://kubernetes.io/docs/tasks/debug-application-cluster/debug-running-pod/>.

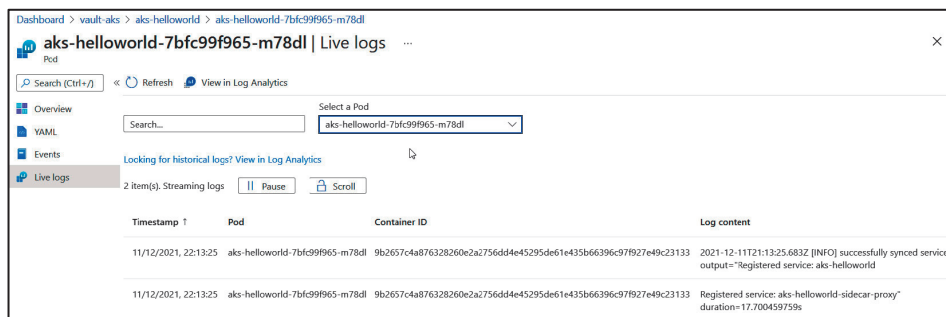
Za pomocą polecenia `kubectl` możemy zautomatyzować debugowanie aplikacji, ale konieczne jest poznanie wszystkich opcji wiersza poleceń.

Następnie omówimy debugowanie aplikacji za pomocą niektórych narzędzi.

Korzystanie z interfejsu webowego

Jak wyjaśniono w pierwszej części tego rozdziału, „Instalacja Kubernetesa”, aby wyświetlić w webowym interfejsie użytkownika (ang. *user interface* — UI) szczegóły wszystkich zasobów wdrożonych w Kubernetesie, możemy skorzystać z podstawowego pulpitu nawigacyjnego.

W przypadku usług Kubernetesa zarządzanych w chmurze, takich jak AKS dla Azure, EKS dla Amazon Web Services (AWS) i GKE dla Google Cloud Platform (GCP), możemy używać zintegrowanych i zarządzanych pulpitu nawigacyjnego. W przypadku AKS w Azure możemy łatwo wyświetlić wszystkie dzienniki podów. Oto przykładowy log podów w portalu Azure:



Rysunek 10.31. Logi w czasie rzeczywistym w AKS

Aby uzyskać więcej informacji na temat debugowania w AKS, przeczytaj dokumentację tutaj:

<https://docs.microsoft.com/en-us/azure/architecture/microservices/logging-monitoring>

Możemy również użyć narzędzi takich jak Octant lub Lens.

Korzystanie z narzędzi

Istnieje wiele narzędzi lub rozwiązań do monitorowania i wyświetlania wszystkich zasobów wdrożonych w klastrze Kubernetesa. Wśród nich są dwa darmowe narzędzia, z których często korzystam: Octant i Lens.

Octant

Octant to projekt webowy społeczności Vmware. Może być uruchamiany lokalnie lub w kontenerze Dockera. Służy do wizualizacji zasobów Kubernetesa i dzienników aplikacji.

Dokumentacja Octant jest dostępna tutaj:

<https://octant.dev/>

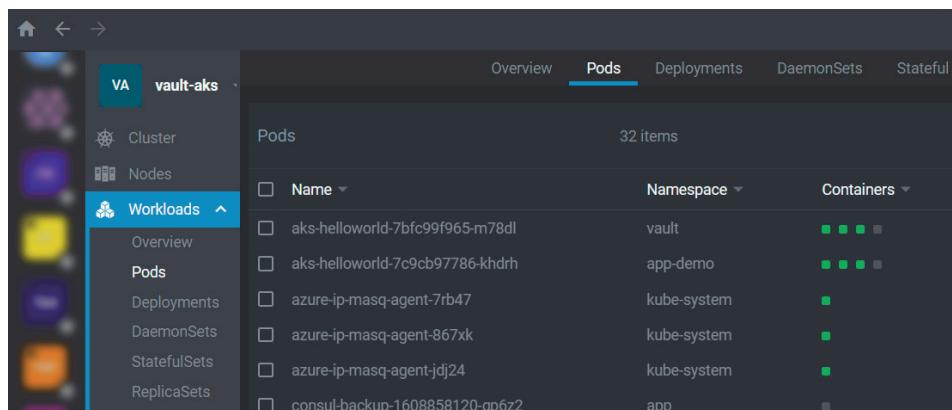
Kod źródłowy jest dostępny tutaj:

<https://github.com/vmware-tanzu/octant/blob/master/README.md>

Lens

Lens to również darmowe narzędzie, instalowane za pomocą pliku binarnego klienta. Dla mnie Lens jest najlepszym narzędziem do wizualizacji i debugowania aplikacji. Dokumentacja dostępna jest tutaj: <https://k8slens.dev/>.

Możesz zobaczyć przegląd pulpitu nawigacyjnego Lens na następującym zrzucie ekranu:



Rysunek 10.32. Pulpit nawigacyjny Lens

Po omówieniu narzędzi do debugowania aplikacji zobaczymy, jak wyświetlić metryki Kubernetesa.

Monitorowanie metryk Kubernetesa

Wszystkie opisane narzędzia doskonale nadają się do debugowania zasobów i aplikacji hostowanych w Kubernetesie. Ale to nie wystarczy — jeśli chodzi o monitorowanie, musimy również monitorować metryki, takie jak procesor i pamięć RAM używane przez aplikacje.

Za pomocą podstawowego polecenia `kubectl` możemy wyświetlić procesor i pamięć RAM używane przez węzły i pody:

```
PS C:\Users\mkrief> kubectl top nodes
NAME                                CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
aks-default-29977126-vmss000000    133m        7%    1875Mi          87%
aks-default-29977126-vmss000001    147m        7%    1986Mi          92%
aks-default-29977126-vmss000002    342m       18%    2017Mi          93%

PS C:\Users\mkrief> kubectl top pods -n vault
NAME                                CPU(cores)   MEMORY(bytes)
aks-helloworld-7bfc99f965-m78dl     15m          87Mi
consul-consul-connect-injector-webhook-deployment-79b4b98ctt844  4m           38Mi
consul-consul-controller-5bdb877dfd-4rlmw  2m           22Mi
consul-consul-gjtkq                  8m           32Mi
consul-consul-kmcl8                  8m           26Mi
consul-consul-server-0               14m          64Mi
consul-consul-tnlds                  9m           23Mi
consul-consul-webhook-cert-manager-556df5dbfd-k6wfp  3m           20Mi
vault-0                              4m           53Mi
vault-1                              6m           37Mi
vault-agent-injector-846f9f7bc6-76bjz   4m           13Mi
web-01enqv6avrgwqgscxzhcsz6rpk-f5984776d-lkzzq  0m            0Mi
```

Rysunek 10.33. `kubectl` pobiera metryki

Do najbardziej znanych rozwiązań należą narzędzia **Prometheus** i **Grafana**, które monitorują metryki Kubernetesa i dostarczają wiele modeli dashboardów.

Aby uzyskać więcej informacji, możesz przeczytać ten artykuł, który wyjaśnia monitorowanie w Kubernetesie za pomocą Prometheusa i Grafany:

<https://sysdig.com/blog/kubernetes-monitoring-prometheus/>

W tej sekcji omówiliśmy niektóre polecenia, narzędzia lub rozwiązania `kubectl`, takie jak pulpity nawigacyjne, Octant i Lens, służące do debugowania aplikacji w Kubernetesie.

Podsumowanie

W tym rozdziale zobaczyliśmy zaawansowane użycie kontenerów z wykorzystaniem Kubernetesa, czyli menedżera kontenerów.

Omówiliśmy różne opcje instalacji małego klastra na komputerze lokalnym za pomocą Docker Desktop. Następnie, korzystając z pliku YAML i polecenia `kubectl`, zrealizowaliśmy wdrożenie obrazu Dockera w naszym klastrze Kubernetesa w celu uruchomienia aplikacji internetowej.

Zainstalowaliśmy i skonfigurowaliśmy Helma, który jest menedżerem pakietów Kubernetesa. Następnie zastosowaliśmy tę wiedzę w praktyce na przykładzie wdrożenia charta w Kubernetesie.

Omówiliśmy również AKS, która jest usługą Kubernetesa zarządzaną przez platformę Azure. Przyjrzelśmy się jej tworzeniu i konfiguracji oraz niektórym linkom do zasobów, które wyjaśniają, jak wdrażać aplikacje za pomocą potoków CI/CD przy użyciu usługi Azure DevOps.

Rozdział zakończyliśmy omówieniem narzędzi do monitorowania Kubernetesa, takich jak polecenie `kubectl`, Lens, Prometheus i Grafana do debugowania metryk Kubernetesa.

Następny rozdział rozpoczyna nową część tej książki, która zajmuje się testowaniem aplikacji, a my zaczniemy od testowania **interfejsu programowania aplikacji** (ang. *application programming interface* — **API**) za pomocą programu Postman.

Pytania

1. Jaka jest rola Kubernetesa?
2. Gdzie znajduje się konfiguracja obiektów tworzonych w Kubernetesie?

3. Jak nazywa się narzędzie klienckie Kubernetesa?
4. Które polecenie pozwala nam zastosować wdrożenie w Kubernetesie?
5. Co to jest Helm?
6. Co to jest AKS?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o Kubernetesie, zapoznaj się z następującymi zasobami:

- *The DevOps 2.3 Toolkit* — <https://www.packtpub.com/business/devops-23-toolkit>.
- *Hands-On Kubernetes on Azure* — <https://www.packtpub.com/virtualization-and-cloud/hands-kubernetes-azure>.
- *Hands-On Kubernetes on Azure — Second Edition* — <https://www.packtpub.com/product/hands-on-kubernetes-on-azure-second-edition/9781800209671>.
- *Mastering Kubernetes — Third Edition* — <https://www.packtpub.com/product/mastering-kubernetes-third-edition/9781839211256>.

Testowanie aplikacji

W tej sekcji wyjaśniono kilka sposobów testowania interfejsów API za pomocą Postmana. Omawiamy także statyczną analizę kodu za pomocą SonarQube i testy wydajności za pomocą Postmana.

Część ta składa się z następujących rozdziałów:

- Rozdział 11., „Testowanie interfejsów API za pomocą Postmana”.
- Rozdział 12., „Statyczna analiza kodu za pomocą SonarQube”.
- Rozdział 13., „Testy bezpieczeństwa i wydajności”.



Testowanie interfejsów API za pomocą Postmana

Rozdział

11

W poprzednich rozdziałach omówiliśmy kulturę DevOps i pojęcie **infrastruktury jako kodu (IaC)** wykorzystujące narzędzia Terraform, Ansible i Packer. Następnie zobaczyliśmy, jak korzystać z menedżera kodu źródłowego Git wraz z implementacją potoku CI/CD za pomocą Jenkinsa i Azure Pipelines. Na koniec pokazaliśmy konteneryzację aplikacji za pomocą Dockera i ich wdrożenie w klastrze Kubernetesa.

Jeśli jesteś programistą, powinieneś zdać sobie sprawę, że używasz API na co dzień, albo po stronie klienta (gdzie korzystasz z API), albo jako dostawca API.

API, podobnie jak aplikacja, musi być testowalne, tzn. musi być możliwe testowanie różnych metod tego API w celu sprawdzenia, czy odpowiada bezbłędnie i czy odpowiedź API jest równa oczekiwanemu wynikowi.

Ponadto prawidłowe działanie interfejsu API jest o wiele bardziej krytyczne dla aplikacji, ponieważ ten interfejs API jest potencjalnie używany przez kilka aplikacji klienckich, a jeśli nie działa, będzie miał wpływ na wszystkie te aplikacje.

Typowe wyzwania związane z API polegają na tym, że musimy napisać skrypt lub opracować dedykowanego klienta aplikacji w celu przetestowania każdego API z osobna, lub zdefiniować workflow dla wielu wykonań API.

W tym rozdziale dowiemy się, jak przetestować API za pomocą specjalistycznego narzędzia o nazwie **Postman**.

Zbadamy wykorzystanie kolekcji i zmiennych, następnie napiszemy testy Postmana, a na koniec zobaczymy, jak zautomatyzować wykonywanie testów Postmana za pomocą Newmana w potoku CI/CD.

Ten rozdział obejmuje następujące tematy:

- tworzenie kolekcji Postmana,
- korzystanie ze środowisk i zmiennych,

- tworzenie testów Postmana,
- wykonywanie testów lokalnie,
- zrozumienie koncepcji Newmana,
- przygotowywanie kolekcji Postmana dla Newmana,
- korzystanie z wiersza poleceń Newmana,
- integracja Newmana z procesem potoku CI/CD.

Wymagania techniczne

W tym rozdziale użyjemy **Newmana**, który jest pakietem Node.js. Dlatego musimy wcześniej zainstalować na naszym komputerze pakiety Node.js i npm, które możemy pobrać pod adresem <https://nodejs.org/en/>.

W przypadku demonstracyjnych interfejsów API stosowanych w tym rozdziale użyjemy przykładu udostępnionego w internecie: <https://jsonplaceholder.typicode.com/>.

Repozytorium GitHuba, które zawiera kompletny kod źródłowy użyty w tym rozdziale, można znaleźć pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP11>.

Obejrzyj poniższy film z kanału Code in Action: <https://bit.ly/3s7239U>.

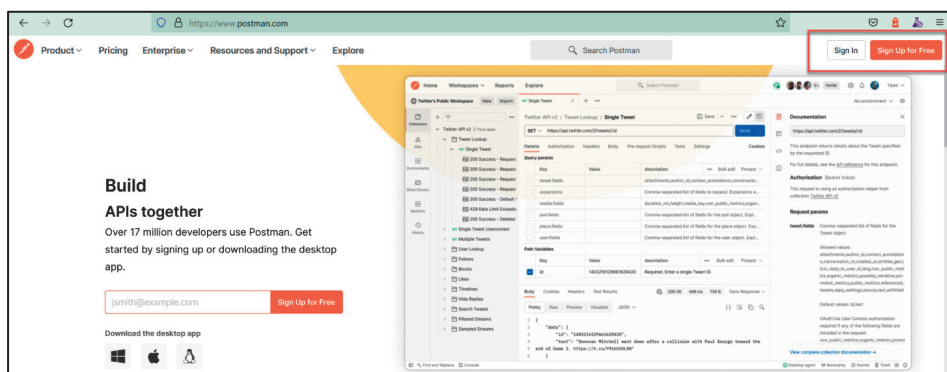
Tworzenie kolekcji żądań Postmana

Postman to darmowe narzędzie klienckie z interfejsem graficznym, które można zainstalować w dowolnym systemie operacyjnym. Jego rolą jest testowanie API za pomocą żądań, które zorganizujemy w kolekcje. Pozwala nam również zdynamizować testy API poprzez wykorzystanie zmiennych i implementację środowisk. Postman słynie z łatwości obsługi, a także z zaawansowanych funkcji, które oferuje.

W tej sekcji dowiemy się, jak utworzyć i zainstalować konto Postmana, następnie utworzymy kolekcję, która posłuży jako folder do uporządkowania naszych żądań, a na koniec utworzymy żądanie, które przetestuje przykładowe API.

Zanim skorzystamy z Postmana, musimy utworzyć konto. W tym celu przechodzimy na stronę <https://www.postman.com/> i klikamy przycisk *Sign Up for Free*. Następnie kliknij łącze *Create Account*, jak pokazano na rysunku 11.1.

Teraz możesz utworzyć dla siebie konto w serwisie Postmana, wypełniając formularz. Możesz je również utworzyć za pomocą swojego konta Google.



Rysunek 11.1. Rejestracja w serwisie Postmana

To konto będzie używane do synchronizowania danych między Twoim komputerem a kontem Postmana. W ten sposób dane będą dostępne na wszystkich Twoich stacjach roboczych.

Po utworzeniu konta przyjrzymy się, jak pobrać i zainstalować Postmana na komputerze lokalnym.

Instalacja Postmana

Po utworzeniu konta w serwisie Postmana osoby korzystające z systemu Windows mogą pobrać Postmana z adresu <https://www.getpostman.com/downloads/> i wybrać wersję do zainstalowania. Ci, którzy chcą zainstalować go w systemie Linux lub macOS, muszą po prostu kliknąć link odpowiadający ich systemowi operacyjnemu:



Rysunek 11.2. Pobieranie Postmana

Po pobraniu Postmana musimy go zainstalować, klikając pobrany plik dla systemu Windows lub dla innych systemów operacyjnych. Postępuj zgodnie z dokumentacją instalacji dostępną pod adresem https://learning.getpostman.com/docs/postman/launching_postman/installation_and_updates/.

Właśnie zobaczyliśmy, że instalacja Postmana jest bardzo prosta. Kolejnym krokiem jest utworzenie kolekcji, w której utworzymy żądanie.

Uwaga

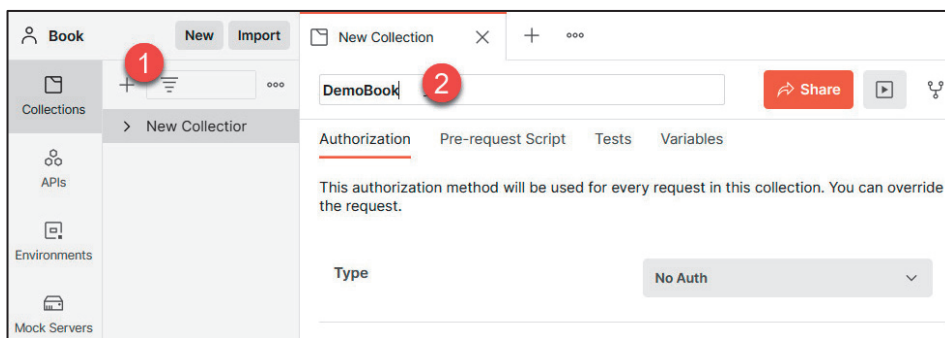
API, które przetestujemy w tym rozdziale, to demonstracyjne API, które jest udostępniane bezpłatnie na następującej stronie: <https://jsonplaceholder.typicode.com/>.

Tworzenie kolekcji

W Postmanie każde żądanie, które testujemy, musi zostać dodane do katalogu o nazwie *Collection*. Katalog ten zapewnia przechowywanie żądań i pozwala na lepszą organizację.

Utworzymy zatem kolekcję *DemoBook*, która będzie zawierała żądania do API. W tym celu wykonamy następujące zadania:

1. W programie Postman w panelu po lewej stronie kliknij *Collections* i przycisk +.
2. Po otwarciu zakładki wprowadzimy nazwę **DemoBook**. Te kroki tworzenia nowej kolekcji ilustruje poniższy zrzut ekranu:



Rysunek 11.3. Tworzenie kolekcji Postmana

Otrzymujemy kolekcję *Demobook*, która pojawia się w lewym panelu Postmana.

Jest ona również zsynchronizowana z naszym kontem w Postmanie i możemy uzyskać do niej dostęp pod adresem <https://web.postman.co/me/collections>.

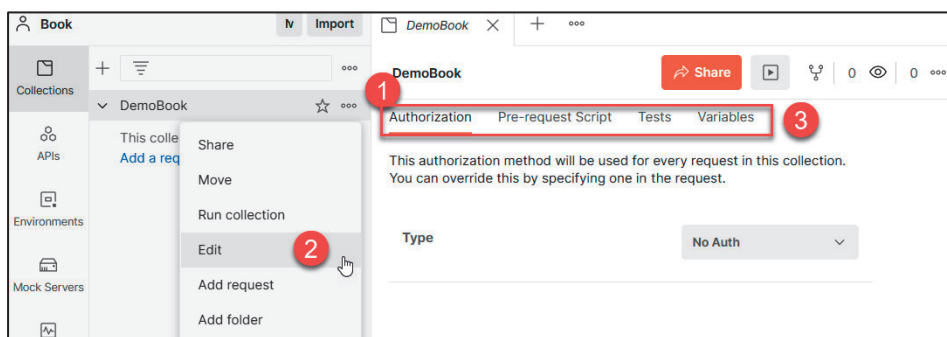
Ta kolekcja pozwoli nam uporządkować żądania naszych testów API. Będzie można zmieniać jej właściwości w celu zastosowania określonej konfiguracji do wszystkich żądań, które będą zawarte w tej kolekcji.

Właściwości te zawierają uwierzytelnianie żądań, testy do wykonania przed żądaniem i po żądaniu oraz zmienne wspólne dla wszystkich żądań w tej kolekcji.

Aby zmodyfikować ustawienia i właściwości tej kolekcji, wykonaj następujące czynności:

1. Kliknij przycisk ... w menu kontekstowym kolekcji.
2. Wybierz opcję *Edit*. Pojawi się formularz edycji, gdzie możemy zmienić wszystkie ustawienia, które będą miały zastosowanie do żądań w tej kolekcji.
3. Przełączaj się między wszystkimi zakładkami konfiguracyjnymi dla opcji edytowania autoryzacji, skryptów, testów lub zmiennych.

Poniższy zrzut ekranu przedstawia kroki podejmowane w celu modyfikacji właściwości kolekcji:



Rysunek 11.4. Edycja kolekcji Postmana

Omówiliśmy procedurę, którą należy wykonać, aby utworzyć kolekcję. Jest to pierwszy element Postmana. Pozwoli nam zorganizować nasze żądania testowania interfejsu API.

Utworzymy teraz żądanie, które wywoła i przetestuje poprawność działania naszego demonstracyjnego API.

Tworzenie pierwszego żądania

W Postmanie obiekt, który zawiera właściwości testowanego interfejsu API, nazywa się **żądaniem** (ang. *request*).

Żądanie to zawiera konfigurację samego interfejsu API, a także testy, które należy wykonać, by sprawdzić, czy działa on prawidłowo.

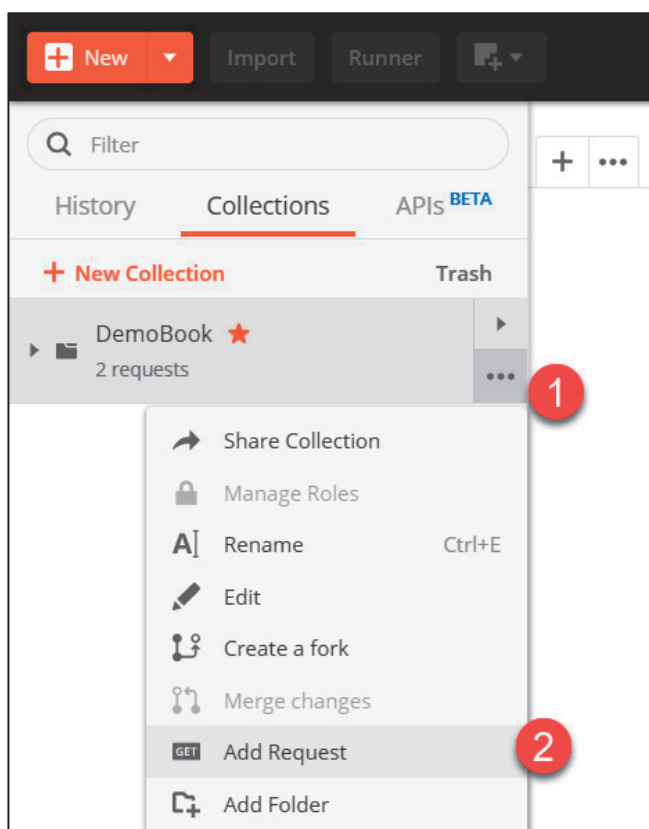
Główne parametry żądania są następujące:

- Adres URL API.
- Jego metoda: GET/POST/DELETE/PATCH.
- Jego właściwości uwierzytelniania.
- Jego klucze ciągu zapytania i żądanie treści.
- Testy, które mają być wykonane przed wykonaniem lub po wykonaniu API.

Tworzenie żądania odbywa się w dwóch krokach — utworzenie w kolekcji, a następnie jego konfiguracja.

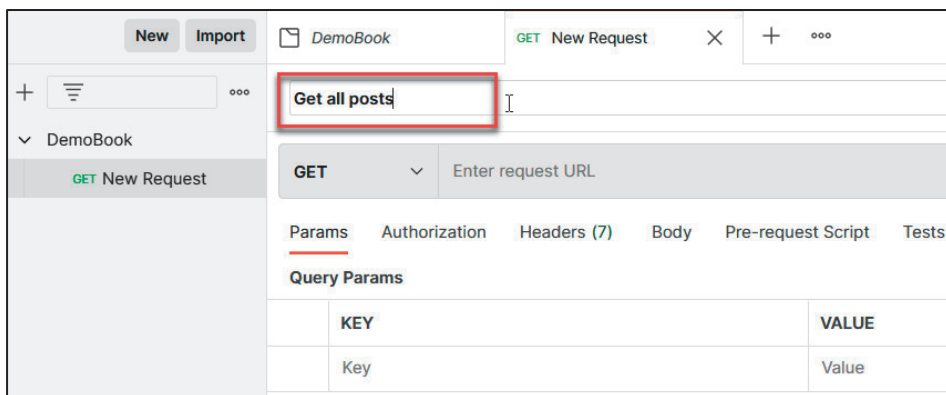
1. Tworzenie żądania. Aby utworzyć żądanie naszego API, należy wykonać następujące kroki:

- I. Przechodzimy do menu kontekstowego kolekcji *DemoBook* i klikamy opcję *Add Request*:



Rysunek 11.5. Dodaj żądanie w Postmanie

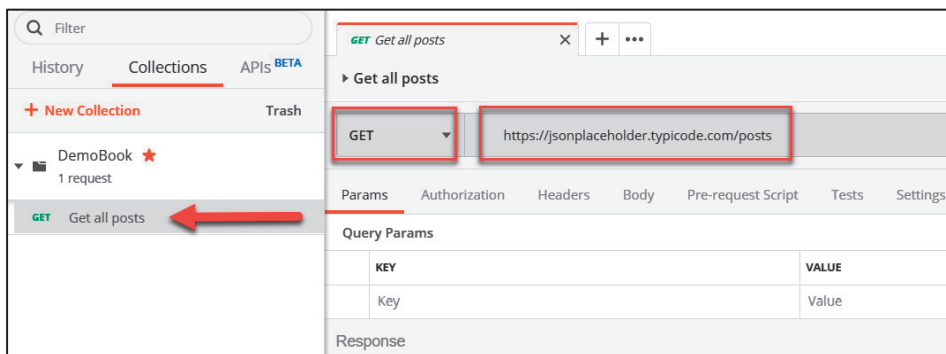
- II. Następnie w nowym tagu wprowadź nazwę żądania, **Get all posts**, jak pokazano na poniższym zrzucie ekranu:



Rysunek 11.6. Nowe żądanie w Postmanie

2. **Konfiguracja żądania.** Po utworzeniu żądania skonfigurujemy je, dodając do metody GET adres URL testowanego API, czyli *https://jsonplaceholder.typicode.com/posts*. Po wprowadzeniu adresu URL zapisujemy konfigurację żądania, klikając przycisk *Save*.

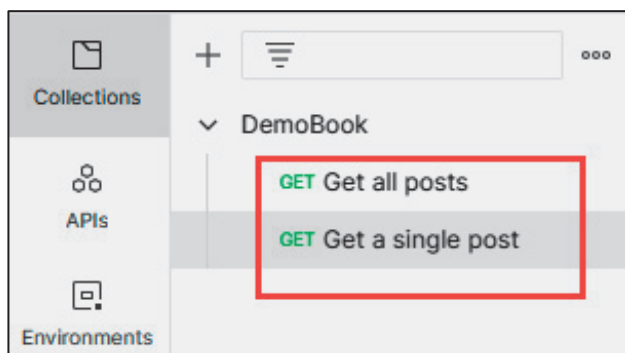
Poniższy zrzut ekranu przedstawia parametry tego żądania wraz z jego adresem URL i metodą:



Rysunek 11.7. Edycja żądania Postmana

Na koniec do naszej kolekcji dodamy drugie żądanie, które nazwiemy **Get a single post**. Będzie ono testowało inną metodę interfejsu API za pomocą adresu URL *https://jsonplaceholder.typicode.com/posts/<ID postu>*.

Poniższy zrzut ekranu przedstawia żądania naszej kolekcji:



Rysunek 11.8. Lista żądań Postmana

Uwaga

Należy pamiętać, że dokumentację Postmana dotyczącą tworzenia kolekcji można znaleźć na stronie https://learning.getpostman.com/docs/postman/collections/creating_collections/.

W tej sekcji dowiedzieliśmy się, jak tworzyć kolekcję w Postmanie, a także jak tworzyć żądania i ich konfiguracje.

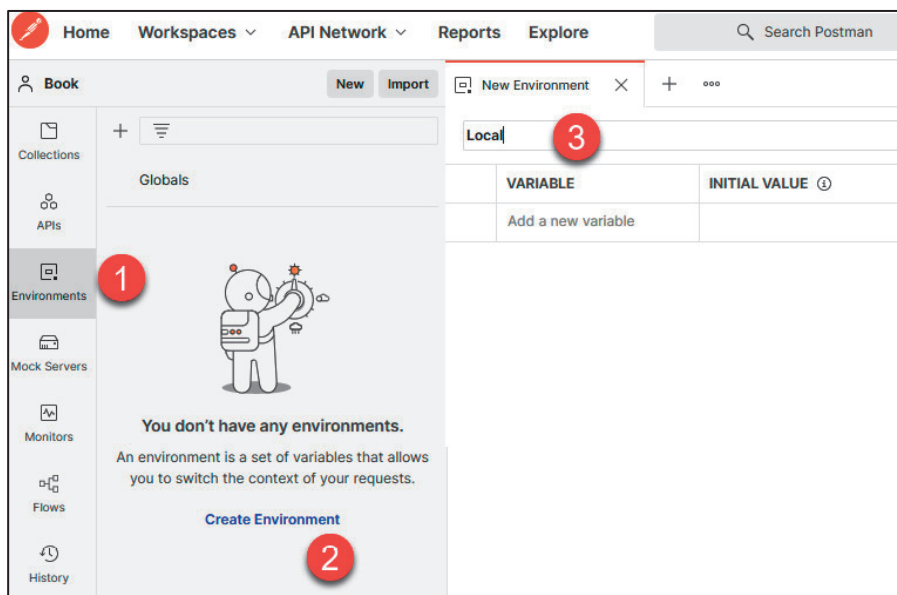
W kolejnym podrozdziale dowiemy się, jak zdynamizować nasze żądania za pomocą środowisk i zmiennych.

Wykorzystywanie środowisk i zmiennych do dynamizowania żądań

Kiedy chcemy przetestować API, musimy przetestować je na kilku środowiskach, aby uzyskać lepsze wyniki. Na przykład przetestujemy je na naszym lokalnym komputerze i w środowisku programistycznym, a następnie również w środowisku QA. Aby zoptymalizować czas implementacji testów i uniknąć zduplikowanych żądań w Postmanie, wprowadzimy zmienne do tego samego żądania. Dzięki temu umożliwimy testowanie we wszystkich środowiskach.

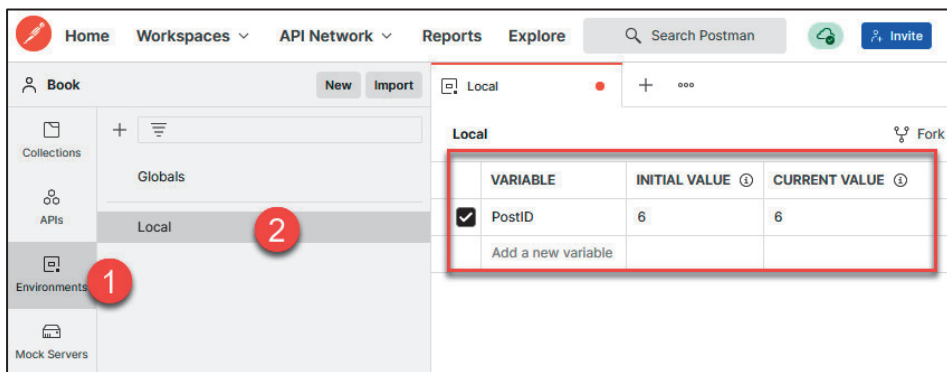
W kolejnych krokach poprawimy nasze żądania, tworząc środowisko i dwie zmienne; następnie zmodyfikujemy nasze żądania, aby korzystać z tych zmiennych.

1. W Postmanie zaczniemy od utworzenia **środowiska** (ang. *environment*), które nazywamy **Local**, jak pokazano na poniższym zrzucie ekranu:



Rysunek 11.9. Dodawanie środowiska w Postmanie

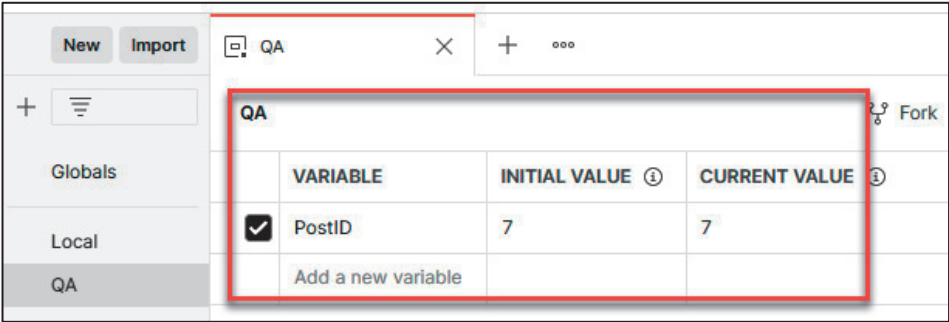
2. Następnie w środowisku **Local** wstawimy **zmienną** (ang. *variable*) o nazwie **PostID**, która będzie zawierała wartość do przekazania w adresie URL żądania. Poniższy zrzut ekranu przedstawia tworzenie zmiennej **PostID**:



Rysunek 11.10. Dodawanie zmiennej środowiskowej w Postmanie

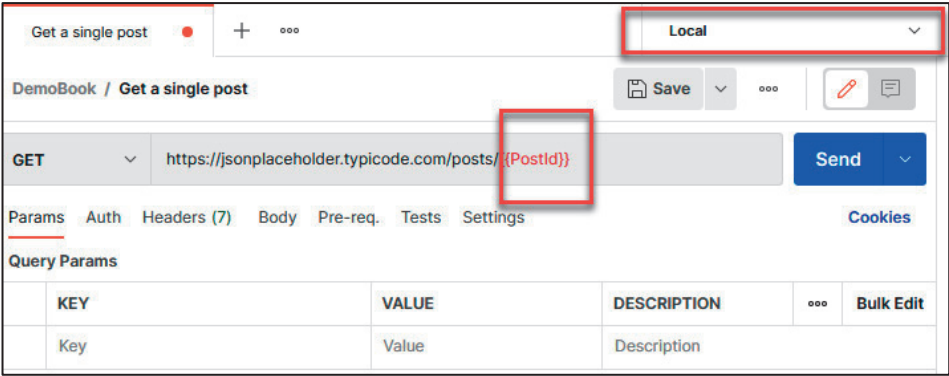
3. Zatem dla środowiska **Local** wartość zmiennej **PostID** wynosi 6. Aby mieć inną wartość dla innych środowisk, konieczne jest utworzenie innych środowisk

za pomocą tych samych kroków, które przed chwilą widzieliśmy, a następnie dodanie tych samych zmiennych (o tej samej nazwie) i odpowiadających im wartości. Pokazuje to ekran zmiennych dla środowiska QA:



Rysunek 11.11. Dodawanie drugiego środowiska w Postmanie

4. Na koniec zmodyfikujemy żądanie tak, aby korzystało ze zmiennej, którą właśnie zadeklarowaliśmy. W Postmanie użycie zmiennej odbywa się za pomocą wzorca `{{nazwa zmiennej}}`. Tak więc najpierw wybieramy żądane środowisko z listy rozwijanej w prawym górnym rogu. Następnie w żądaniu zamieniamy identyfikator posta na końcu adresu URL na `{{PostID}}`, jak pokazano na poniższym zrzucie ekranu:



Rysunek 11.12. Używanie zmiennej środowiskowej w Postmanie

Uwaga

Należy pamiętać, że dokumentacja środowisk i zmiennych Postmana jest dostępna pod adresem https://learning.getpostman.com/docs/postman/environments_and_globals/intro_to_environments_and_globals/.

W tej sekcji utworzyliśmy żądanie Postmana, które pozwoli nam przetestować API. Następnie uelastyczniliśmy jego wykonanie, tworząc środowisko, które zawiera zmienne używane również w żądaniach Postmana.

W następnej sekcji utworzymy testy, aby zweryfikować wynik API.

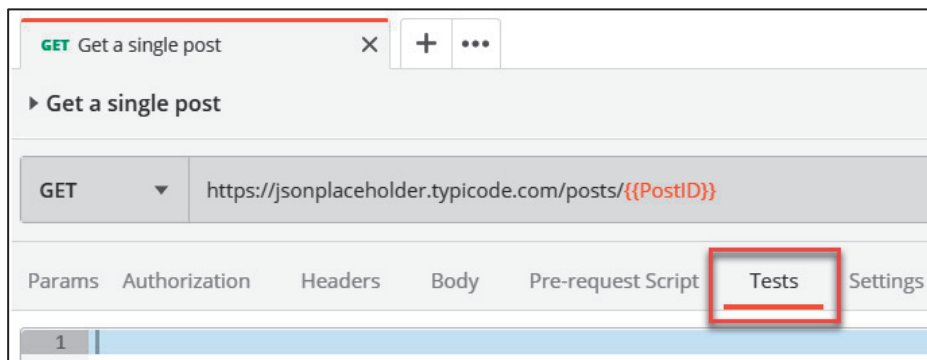
Tworzenie testów Postmana

Testowanie API polega na sprawdzeniu nie tylko, czy jego wywołanie zwraca kod powrotu 200, czyli czy API dobrze odpowiada, lecz także czy jego wynik odpowiada oczekiwanemu lub czy czas wykonania nie jest zbyt długi.

Rozważmy interfejs API, który zwraca odpowiedź w formacie JSON z kilkoma własnościami. W testach tego API konieczne będzie sprawdzenie, czy zwracany wynik jest tekstem w formacie JSON zawierającym oczekiwane własności, a tym bardziej zweryfikowanie wartości tych własności.

W Postmanie możliwe jest tworzenie testów, które za pomocą języka JavaScript zapewnią, że odpowiedź na żądanie będzie odpowiadała oczekiwanemu wynikowi pod względem czasu lub wykonania.

Testy Postmana są zapisywane w zakładce *Tests* żądania:



Rysunek 11.13. Zakładka Tests Postmana

Aby przetestować nasze żądanie, napiszemy kilka testów, które sprawdzają, czy:

- Kod zwrotny żądania to 200.
- Czas odpowiedzi na żądanie jest krótszy niż 400 ms.
- Odpowiedź nie jest pustym plikiem JSON.
- Zwrócona odpowiedź JSON zawiera właściwość `userId`, która jest równa 1.

Aby wykonać te testy, napiszemy kod w zakładce *Tests*.

Poniższy kod ilustruje sprawdzenie kodu powrotu żądania:

```
pm.test("Kod stanu to 200", function () {  
  pm.response.to.have.status(200);  
});
```

Poniższy kod ilustruje sprawdzenie czasu odpowiedzi, czy jest krótszy niż 400 ms:

```
pm.test("Czas odpowiedzi jest krótszy niż 400ms", function() {  
  pm.expect(pm.response.responseTime).to.be.below(400);  
});
```

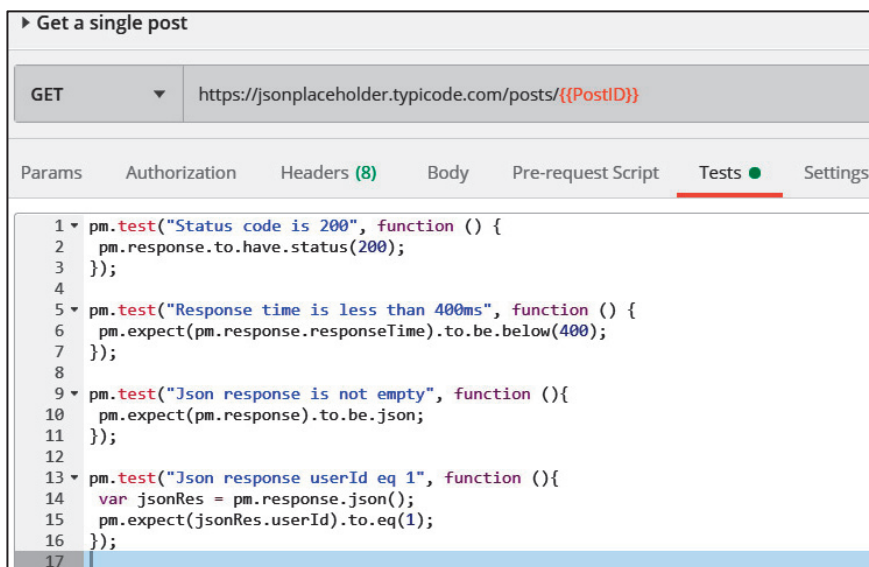
Poniższy kod ilustruje sprawdzenie, czy odpowiedź w formacie JSON nie jest pusta:

```
pm.test("Odpowiedź JSON nie jest pusta", function(){  
  pm.expect(pm.response).to.be.json;  
});
```

Poniższy kod ilustruje sprawdzenie, czy w odpowiedzi JSON właściwość `userId` jest równa 1:

```
pm.test("userId w odpowiedzi JSON ma wartość 1", function () {  
  var jsonRes = pm.response.json();  
  pm.expect(jsonRes.userId).to.eq(1);  
});
```

Zakładka *Tests* naszego żądania, która testuje nasze API, zawiera cały ten kod, jak pokazano na poniższym zrzucie ekranu:



Rysunek 11.14. Kod testów Postmana

Tworzenie żądania Postmana zakończyliśmy testem, który sprawdzi poprawność działania API na podstawie zwracanego kodu, wydajności i treści odpowiedzi.

Uwaga

Więcej informacji na temat testów i skryptów Postmana można znaleźć w dokumentacji na stronie https://learning.getpostman.com/docs/postman/scripts/intro_to_scripts.

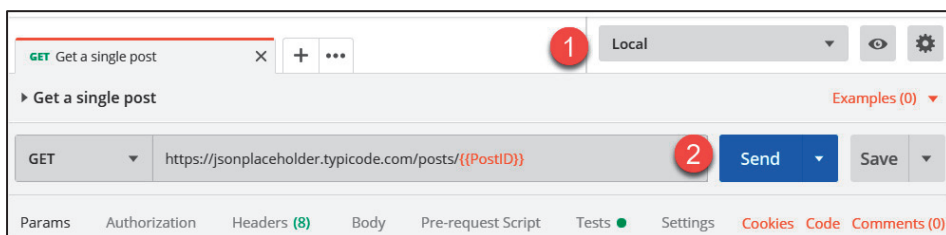
W tej sekcji właśnie zobaczyliśmy, jak w Postmanie tworzyć testy API, aby sprawdzić poprawność działania naszego API. Uruchomimy teraz nasze żądanie lokalnie, aby przetestować nasze API.

Wykonywanie lokalnych testów za pomocą żądań Postmana

Do tej pory utworzyliśmy kolekcję, w której dwa żądania zawierają parametry i testy naszych API, które mają zostać wykonane. Aby przetestować prawidłowe działanie interfejsów API z ich parametrami i testami, musimy teraz wykonać nasze żądania. Zauważ, że dopiero pod koniec tego wykonania będziemy wiedzieć, czy nasze API odpowiadają oczekiwaniom.

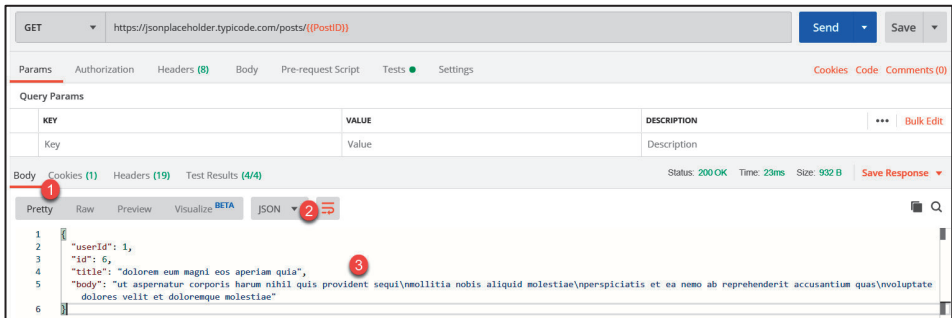
Aby wykonać żądanie Postmana, wykonaj następujące czynności:

1. Najpierw musisz wybrać żądane środowisko.
2. Kliknij przycisk żądania *Send*, jak pokazano na poniższym zrzucie ekranu:



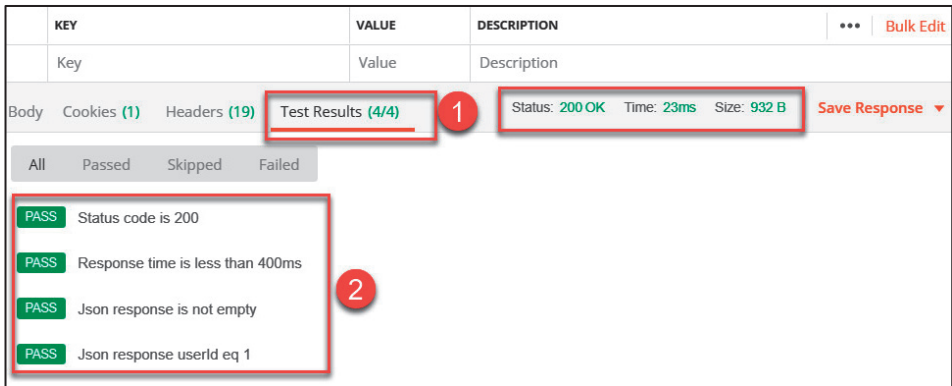
Rysunek 11.15. Rejestracja za pomocą Postmana

3. W zakładce *Body* możemy wtedy podejrzeć treść odpowiedzi na zapytanie, a jeśli chcemy wyświetlić ją w formacie **JSON**, możemy wybrać format wyświetlania. Poniższy zrzut ekranu przedstawia odpowiedź na żądanie wyświetlane w formacie JSON:



Rysunek 11.16. Treść odpowiedzi w Postmanie

4. Zakładka *Test Results* wyświetla wyniki wykonania testów, które wcześniej napisaliśmy. W naszym przypadku cztery testy zostały wykonane poprawnie — wszystkie są zielone, jak pokazano na poniższym rzucie ekranu:

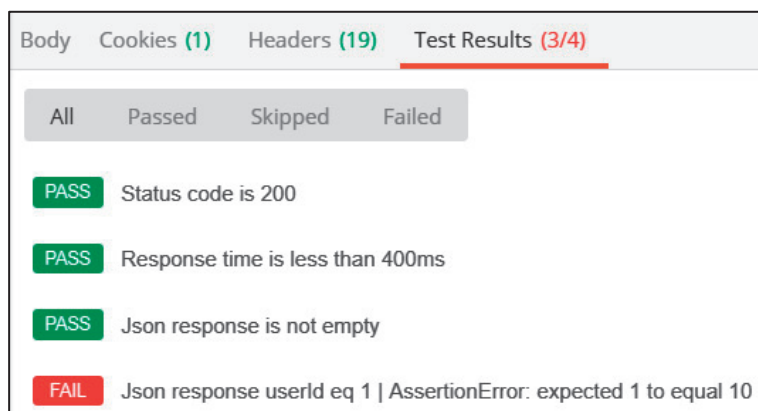


Rysunek 11.17. Wyniki testu Postmana

Na rzucie widzimy, że kod odpowiedzi na żądanie Postmana jest równy 200, co odpowiada kodowi pomyślnego wykonania żądania. Czas wykonania zapytania wynosi 23 ms, czyli jest poniżej progu (400 ms), który ustawiłem przykładowo.

W przypadku niepowodzenia jednego z testów zostanie on wyświetlony na czerwono, aby można go było wyraźnie zidentyfikować. Przykład nieudanego testu pokazano na rysunek 11.18.

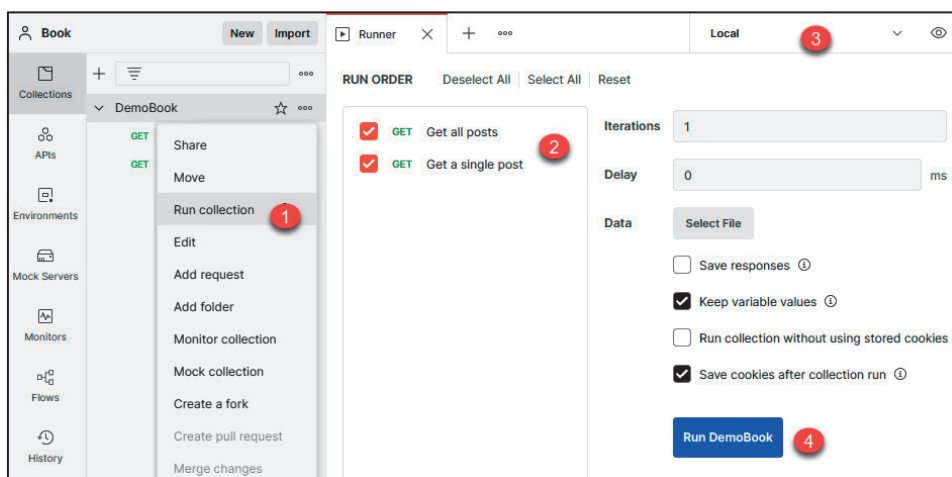
Właśnie widzieliśmy wykonanie żądania Postmana w celu przetestowania interfejsu API. Jednak to wykonanie dotyczy tylko bieżącego żądania. Jeśli chcemy wykonać wszystkie żądania Postmana w kolekcji, możemy skorzystać z narzędzia **Postman Collection Runner**.

**Rysunek 11.18. Test Postmana nie powiódł się**

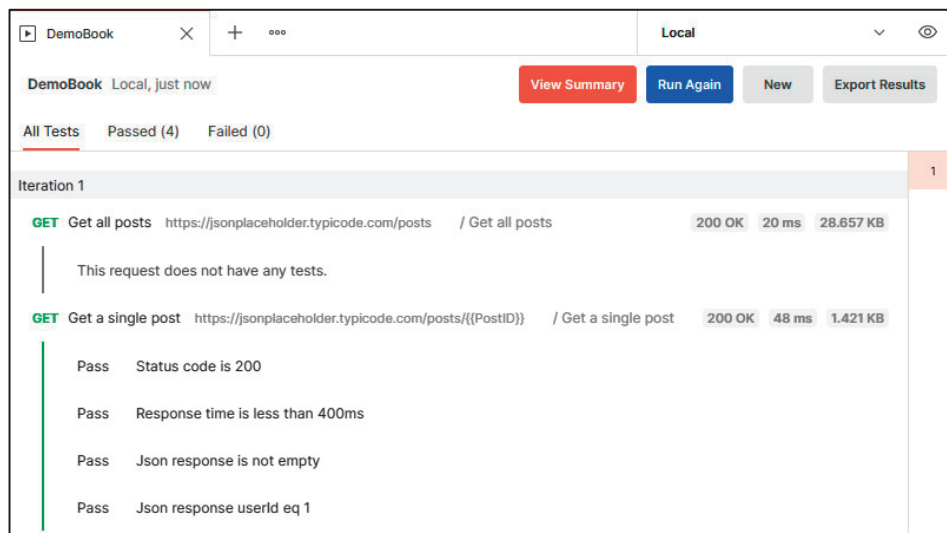
Postman Collection Runner to funkcja Postmana, która automatycznie wykonuje wszystkie żądania w kolekcji w kolejności, w jakiej zostały zorganizowane.

Więcej informacji na temat kolekcji można znaleźć na stronie https://learning.getpostman.com/docs/postman/collection_runs/starting_a_collection_run/.

Poniższe dwa zrzuty ekranu pokazują kroki wykonania *Runnera*, w których wybieramy kolekcję do wykonania, środowisko i liczbę iteracji. Aby rozpocząć jego wykonanie, klikamy przycisk *Run DemoBook*:

**Rysunek 11.19. Postman Runner**

W zakładce *Runner* możemy zobaczyć wynik wykonania testu wszystkich żądań w kolekcji, jak pokazano na poniższym zrzucie ekranu:



Rysunek 11.20. Wykonanie Postman Runnera

Uwaga

Dokumentację programu Postman Runner można znaleźć na stronie https://learning.getpostman.com/docs/postman/collection_runs/intro_to_collection_runs.

W tej sekcji dowiedzieliśmy się, jak wykonywać żądania Postmana w celu przetestowania interfejsu API zarówno za pomocą pojedynczego żądania, jak i za pomocą wszystkich żądań w kolekcji za pomocą Postman Runnera. W dalszej części przedstawimy Newmana, który pozwala nam zautomatyzować wykonywanie testów Postmana.

Zrozumienie koncepcji Newmana

Do tej pory w tym rozdziale mówiliśmy o używaniu Postmana do lokalnego testowania interfejsów API, które tworzymy lub wykorzystujemy. Lecz w testach jednostkowych, akceptacyjnych i integracyjnych ważne jest to, że są one zautomatyzowane, dzięki czemu mogą być wykonywane w ramach potoku CI/CD.

Postman jest narzędziem graficznym, które samo się nie automatyzuje, ale istnieje inne narzędzie o nazwie **Newman**, które automatyzuje testy napisane w Postmanie.

Newman to darmowe narzędzie wiersza poleceń, które ma wielką zaletę — automatyzuje testy, które są już napisane w Postmanie. Pozwala nam to zintegrować wykonanie testów API w skryptach lub procesach CI/CD.

Uwaga

Aby uruchomić API Postmana w Node.js lub przeglądarce, możemy również użyć innego narzędzia o nazwie Postman Sandbox. Jeśli chcesz uzyskać więcej informacji, zapoznaj się z repozytorium GitHuba tutaj: <https://github.com/postmanlabs/postman-sandbox>.

Dodatkowo oferuje możliwość generowania wyników testów raportów w różnych formatach (HTML, JUnit, JSON).

Jednak Newman nie pozwala nam na:

- Tworzenie lub konfigurowanie żądań Postmana; jak zobaczymy, żądania wykonywane przez Newmana będą eksportowane z Postmana.
- Wykonanie tylko jednego żądania znajdującego się w kolekcji — wykonuje wszystkie żądania w kolekcji.

Uwaga

Aby dowiedzieć się więcej na temat Newmana, możesz odwiedzić stronę narzędzia pod adresem <https://www.npmjs.com/package/newman>.

Aby korzystać z Newmana, będziemy potrzebować — zgodnie z sekcją „Wymagania techniczne” tego rozdziału — zainstalowanego Node.js i npm, które są dostępne pod adresem <https://nodejs.org/en/> (ten instalator instaluje oba narzędzia).

Następnie, aby zainstalować Newmana, musimy wykonać polecenie:

```
npm install -g newman
```

Poniższy zrzut ekranu przedstawia wykonanie tego polecenia:

```
PS C:\Users\mkrief> npm install -g newman
C:\Users\mkrief\AppData\Roaming\npm\newman -> C:\Users\mkrief\AppData\Roaming\npm\node_modules\newman\bin\newman.js
+ newman@5.3.0
added 129 packages from 189 contributors in 23.971s
```

Rysunek 11.21. Instalacja Newmana

Polecenie to instaluje pakiet npm (Newman i wszystkie jego zależności) globalnie, tzn. pakiet ten jest dostępny w obrębie całej maszyny.

Po zainstalowaniu możemy przetestować jego instalację, uruchamiając polecenie `newman --help`, które wyświetla argumenty i opcje, jak pokazano na rysunku 11.22.

W tej sekcji przedstawiliśmy Newmana, opowiadając o jego zaletach. Nauczyliśmy się także go instalować. W następnej sekcji wyeksportujemy kolekcję Postmana i środowisko potrzebne do współpracy z Newmanem.

```
>newman --help
Usage: newman [options] [command]

Options:
  -v, --version          output the version number
  -h, --help             output usage information

Commands:
  run [options] <collection>  URL or path to a Postman Collection.

To get available options for a command:
  newman [command] -h
```

Rysunek 11.22. Polecenie pomocy Newmana

Przygotowywanie kolekcji Postmana dla Newmana

Jak właśnie widzieliśmy, Newman jest narzędziem klienckim Postmana i aby działać, potrzebuje konfiguracji kolekcji, żądań i środowisk, które utworzyliśmy w Postmanie.

Dlatego przed uruchomieniem Newmana będziemy musieli wyeksportować kolekcję i środowiska Postmana, a ten eksport posłuży jako argument dla Newmana. Zaczniemy więc eksportować kolekcję *DemoBook*, którą utworzyliśmy w Postmanie.

Eksportowanie kolekcji

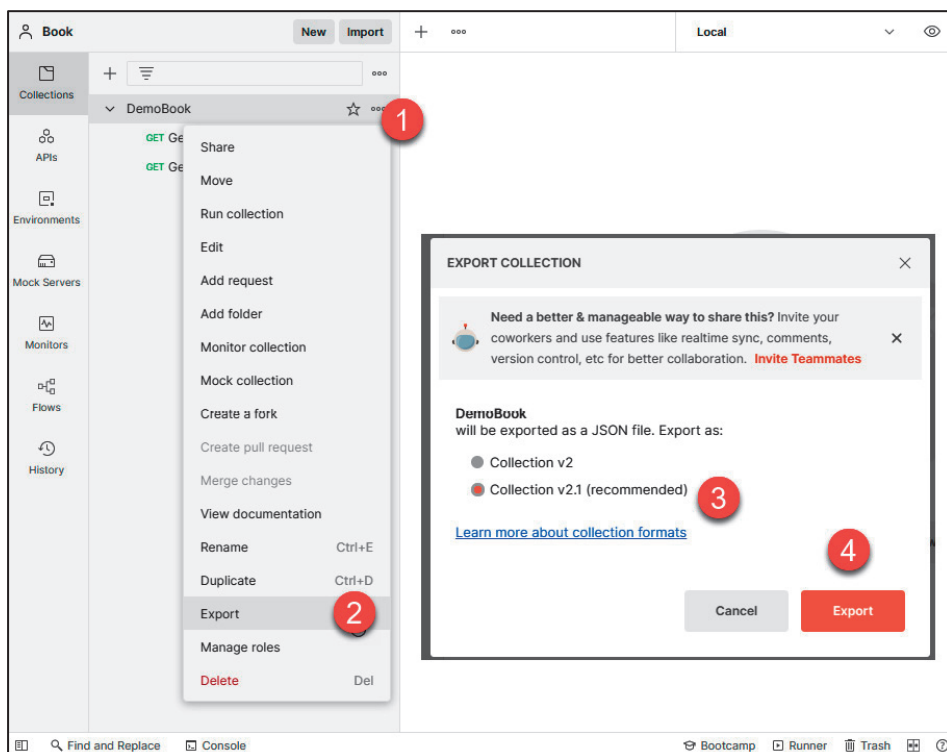
Eksportowanie kolekcji Postmana polega na uzyskaniu pliku JSON, który zawiera wszystkie ustawienia tej kolekcji i znajdujące się w niej żądania.

To właśnie z tego pliku JSON Newman będzie mógł uruchamiać te same testy API, które uruchamialiśmy z Postmana.

Aby wykonać ten eksport, wykonaj następujące czynności:

1. Przejdź do menu kontekstowego kolekcji, którą chcemy wyeksportować.
2. Wybierz akcję *Export*.
3. Następnie w oknie, które zostanie otwarte, usuń zaznaczenie pola *Collection v2.1 (recommended)*.
4. Na koniec zatwierdź operację, klikając przycisk *Export*.

Te kroki pokazano na poniższym zrzucie ekranu:



Rysunek 11.23. Eksport kolekcji Postmana

Kliknięcie przycisku *Export* powoduje eksport kolekcji do pliku w formacie JSON, *DemoBook.postman_collection.json*, który zapisujemy w utworzonym przez nas folderze, dedykowanym dla Newmana.

Po wyeksportowaniu kolekcji musimy również wyeksportować środowisko i zmienne, ponieważ od tego zależą żądania w naszej kolekcji.

Eksportowanie środowisk

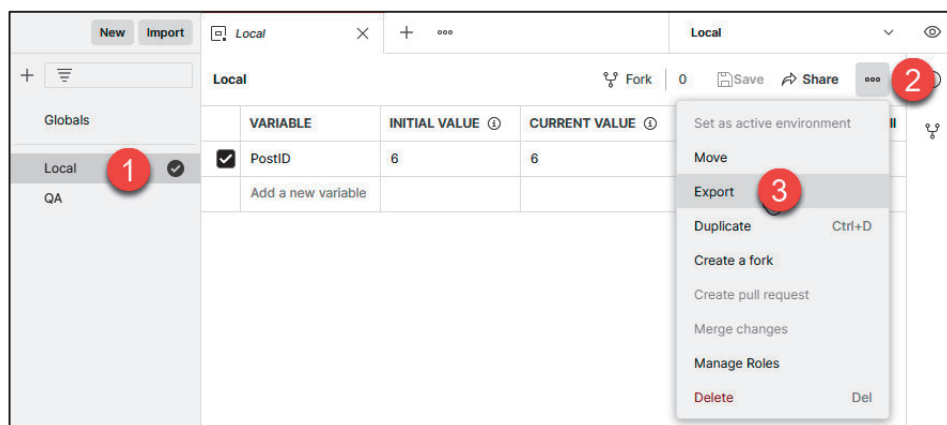
Jeżeli chodzi o konfigurację Newmana, moglibyśmy na tym poprzestać. Problem polega jednak na tym, że nasze żądania Postmana używają zmiennych skonfigurowanych w środowiskach.

Z tego powodu będziemy musieli również wyeksportować informacje z każdego środowiska w formacie JSON, abyśmy mogli przekazać je jako argument do narzędzia Newman.

Aby wyeksportować środowiska i ich zmienne, wykonaj następujące czynności:

1. Otwórz menu *ENVIRONMENTS* z lewego panelu w Postmanie.
2. Następnie kliknij przycisk *Export*, aby wyeksportować środowisko.

Te kroki są pokazane na poniższym zrzucie ekranu:



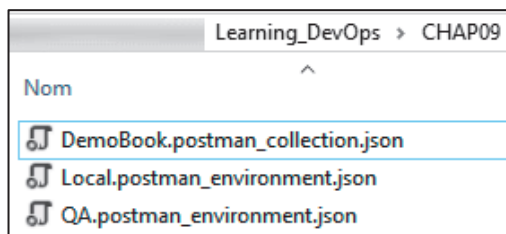
Rysunek 11.24. Eksport środowiska w Newmanie

Zatem dla każdego środowiska wyeksportujemy jego konfigurację do pliku JSON, który zapiszemy w tym samym folderze, do którego wyeksportowaliśmy kolekcję.

Wreszcie otrzymujemy folder na naszej maszynie, który zawiera trzy pliki Postmana w formacie JSON:

- Jeden plik JSON dla kolekcji.
- Jeden plik JSON dla środowiska lokalnego.
- Jeden plik JSON dla środowiska QA.

Poniższy zrzut ekranu przedstawia zawartość folderu lokalnego zawierającego eksporty programu Postman:



Rysunek 11.25. Folder plików eksportu programu Postman

Właśnie omówiliśmy eksport wszystkich konfiguracji naszych żądań Postmana, w tym kolekcji i środowisk, a teraz przyjrzymy się pracy wiersza poleceń Newmana.

Korzystanie z wiersza poleceń Newmana

Po wyeksportowaniu konfiguracji Postmana, którą widzieliśmy wcześniej, uruchomimy narzędzie Newman na naszej lokalnej maszynie.

Aby uruchomić Newmana, przejdź do terminala, a następnie do folderu, w którym znajdują się pliki konfiguracyjne JSON, i wykonaj następujące polecenie:

```
newman run DemoBook.postman_collection.json -e  
Local.postman_environment.json
```

Polecenie `newman run` pobiera plik JSON wyeksportowanej kolekcji jako argument i wartość parametru `-e`, który reprezentuje plik JSON wyeksportowanego środowiska.

Uwaga

Aby uzyskać więcej informacji na temat wszystkich argumentów tego polecenia, przeczytaj dokumentację pod adresem <https://www.npmjs.com/package/newman#newman-options>.

Newman wykona żądania Postmana z wyeksportowanej przez nas kolekcji. Wykorzysta również zmienne wyeksportowanego środowiska, a także wykona testy, które napisaliśmy w żądaniu.

Wynik jego wykonania, który jest dość szczegółowy, pokazano na rysunku 11.26.

Rysunek 11.27 pokazuje wynik w przypadku wystąpienia błędu w czasie testu.

Możemy zobaczyć szczegóły wykonania testu. Pokazany jest błąd i oczekiwania żądania.

W tej sekcji nauczyliśmy się, jak uruchomić Newmana na komputerze lokalnym, a teraz dowiemy się, jak Newman jest zintegrowany z potokiem CI/CD.

```
C:\Users\Mikael\Documents\Postman>newman run DemoBook.postman_collection.json -e Local.postman_environment.json
newman

DemoBook

→ Get all posts
GET https://jsonplaceholder.typicode.com/posts [200 OK, 27.64KB, 194ms]

→ Get a single post
GET https://jsonplaceholder.typicode.com/posts/6 [200 OK, 932B, 27ms]
✓ Status code is 200
✓ Response time is less than 400ms
✓ Json response is not empty
✓ Json response userId eq 1
```

| | executed | failed |
|----------------------|----------|--------|
| iterations | 1 | 0 |
| requests | 2 | 0 |
| test-scripts | 1 | 0 |
| prerequisite-scripts | 0 | 0 |
| assertions | 4 | 0 |

total run duration: 338ms

total data received: 27.16KB (approx)

average response time: 110ms [min: 27ms, max: 194ms, s.d.: 83ms]

Rysunek 11.26. Wykonanie Newmana

```
C:\Users\Mikael\Documents\Postman>newman run DemoBook.postman_collection.json -e qa.postman_environment.json
newman

DemoBook

→ Get all posts
GET https://jsonplaceholder.typicode.com/posts [200 OK, 27.64KB, 254ms]

→ Get a single post
GET https://jsonplaceholder.typicode.com/posts/7 [200 OK, 868B, 28ms]
✓ Status code is 200
✓ Response time is less than 400ms
✓ Json response is not empty
1. Json response userId eq 1
```

| | executed | failed |
|----------------------|----------|--------|
| iterations | 1 | 0 |
| requests | 2 | 0 |
| test-scripts | 1 | 0 |
| prerequisite-scripts | 0 | 0 |
| assertions | 4 | 1 |

total run duration: 412ms

total data received: 27.09KB (approx)

average response time: 141ms [min: 28ms, max: 254ms, s.d.: 113ms]

```
# failure detail
1. AssertionError      Json response userId eq 1
                        expected 1 to equal 10
                        at assertion:3 in test-script
                        inside "Get a single post"
```

Rysunek 11.27. Nieudane testy Newmana

Integracja Newmana z procesem potoku CI/CD

Newman to narzędzie automatyzujące wykonywanie żądań Postmana z linii poleceń, co pozwoli nam szybko zintegrować go z potokiem CI/CD.

Aby uprościć jego integrację w potoku, przechodzimy do katalogu zawierającego pliki JSON wyeksportowane z Postmana i tworzymy plik konfiguracyjny npm — *package.json*.

Będzie on zawierał następującą treść:

```
{
  "name": "postman",
  "version": "1.0.0",
  "description": "postmanrestapi",
  "scripts": {
    "testapilocal": "newman run
DemoBook.postman_collection.json -e
Local.postman_environment.json -r junit,cli --reporter
junit-export result-tests-local.xml",
    "testapiQA": "newman run
DemoBook.postman_collection.json -e
QA.postman_environment.json -r junit,cli --reporter-junit
export result-tests-qa.xml"
  },
  "devDependencies": {
    "newman": "^5.3.0"
  }
}
```

W sekcji `scripts` umieszczamy dwa skrypty, które zostaną wykonane za pomocą wiersza poleceń, który widzieliśmy w poprzedniej sekcji. Dodajemy argument `-r`, który umożliwia podgląd danych wyjściowych w formacie JUnit. W sekcji `DevDependencies` wskazujemy, że potrzebujemy pakietu Newmana.

Otóż to; mamy wszystkie pliki, które są niezbędne do zintegrowania wykonania Newmana z potokiem CI/CD.

Aby pokazać integrację Newmana z potokiem CI/CD, użyjemy Azure Pipelines — usługi Azure DevOps, którą widzieliśmy w rozdziale 7, „Ciągła integracja i ciągłe wdrażanie”, oraz w rozdziale 9, „Konteneryzacja aplikacji za pomocą Dockera”, a która ma tę zaletę, że posiada graficzną reprezentację potoku.

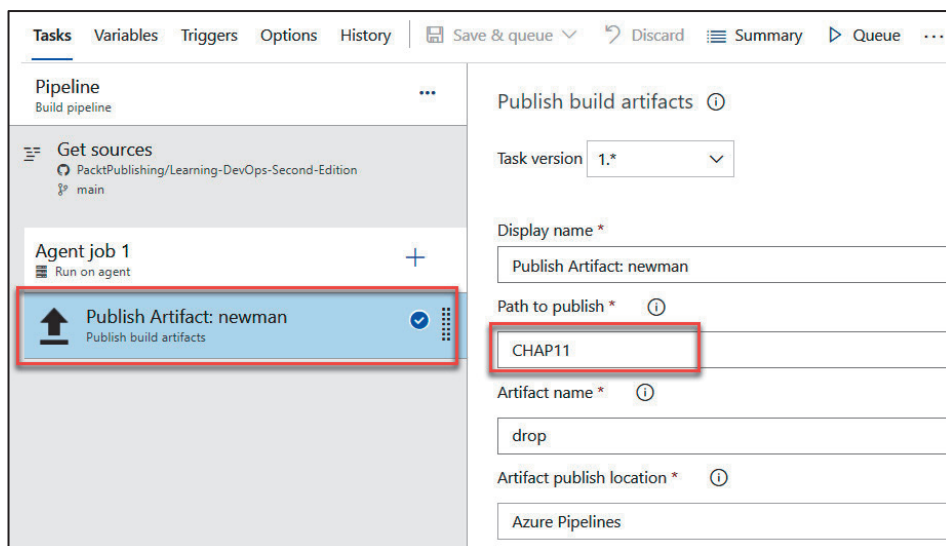
Warunkiem wstępnym dla potoku jest zatwierdzenie katalogu zawierającego pliki JSON z eksportem programu Postman i pliku *package.json* w systemie kontroli wersji.

W naszym przypadku skorzystamy z repozytorium GitHuba, które zawiera pełny kod źródłowy tego rozdziału: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP11>.

Budowa i udostępnianie konfiguracji

W Azure Pipelines zbudujemy i udostępnimy konfigurację, wykonując następujące czynności:

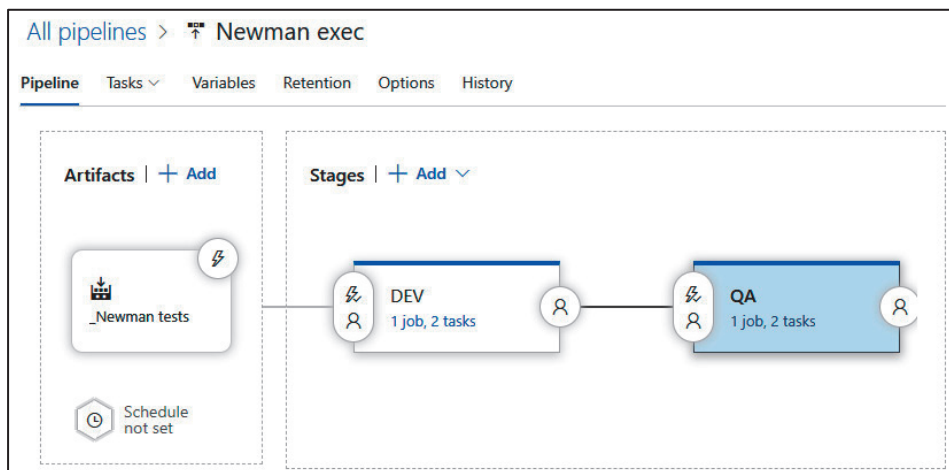
1. Tworzymy nową **definicję kompilacji**, która kopiuje pliki potrzebne do uruchomienia Newmana dla elementów kompilacji, jak pokazano na poniższym zrzucie ekranu:



Rysunek 11.28. Azure Pipelines publikuje pliki Newmana

Włączamy również opcję ciągłej integracji w zakładce *Triggers*. Następnie, aby uruchomić tę kompilację, zapisujemy tę definicję kompilacji i ustawiamy ją w kolejce.

2. Następnie tworzymy nową **definicję wydania**, która będzie odpowiedzialna za uruchamianie Newmana dla każdego środowiska. Ta wersja otrzyma artefakty kompilacji i będzie się składała z dwóch etapów, *DEV* i *QA*, jak pokazano na poniższym zrzucie ekranu:

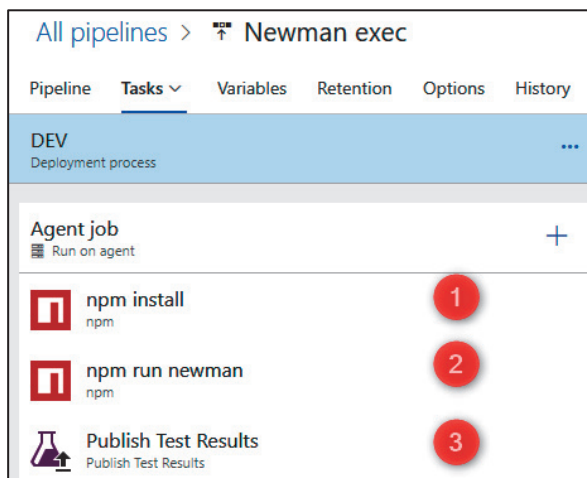


Rysunek 11.29. Tworzenie wydania w Azure Pipelines za pomocą Newmana

Dla każdego z tych etapów konfigurujemy trzy zadania, które przebiegają następująco, na podstawie pliku *package.json*:

1. Instalacja Newmana.
2. Uruchomienie Newmana.
3. Publikacja wyników testu w Azure Pipelines.

Poniższy zrzut ekranu przedstawia konfigurację zadań dla każdego etapu:

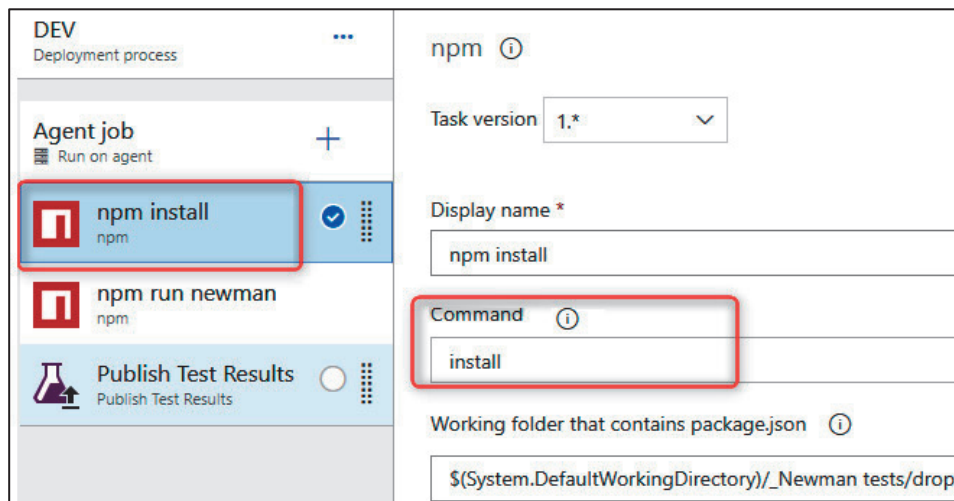


Rysunek 11.30. Kroki przygotowania wydania przez Newmana w Azure Pipelines

Przyjrzyjmy się szczegółom parametrów tych zadań.

npm install

Parametry zadania `npm install` są następujące:



The screenshot shows the configuration for the 'npm install' task in an Azure Pipelines agent job. On the left, a list of tasks includes 'npm install', 'npm run newman', and 'Publish Test Results'. The 'npm install' task is selected and highlighted with a red box. On the right, the configuration details for the 'npm' task are shown. The 'Task version' is set to '1.*'. The 'Display name' is 'npm install'. The 'Command' field is set to 'install' and is also highlighted with a red box. The 'Working folder that contains package.json' is set to '\$(System.DefaultWorkingDirectory)/_Newman tests/drop'.

| Task configuration details |
|--|
| Task name: npm install |
| Task version: 1.* |
| Display name: npm install |
| Command: install |
| Working folder that contains package.json: \$(System.DefaultWorkingDirectory)/_Newman tests/drop |

Rysunek 11.31. Instalacja npm w Azure Pipelines

W tym przypadku poleceniem, które ma zostać wykonane w katalogu zawierającym pliki artefaktów, jest `npm install`.

npm run newman

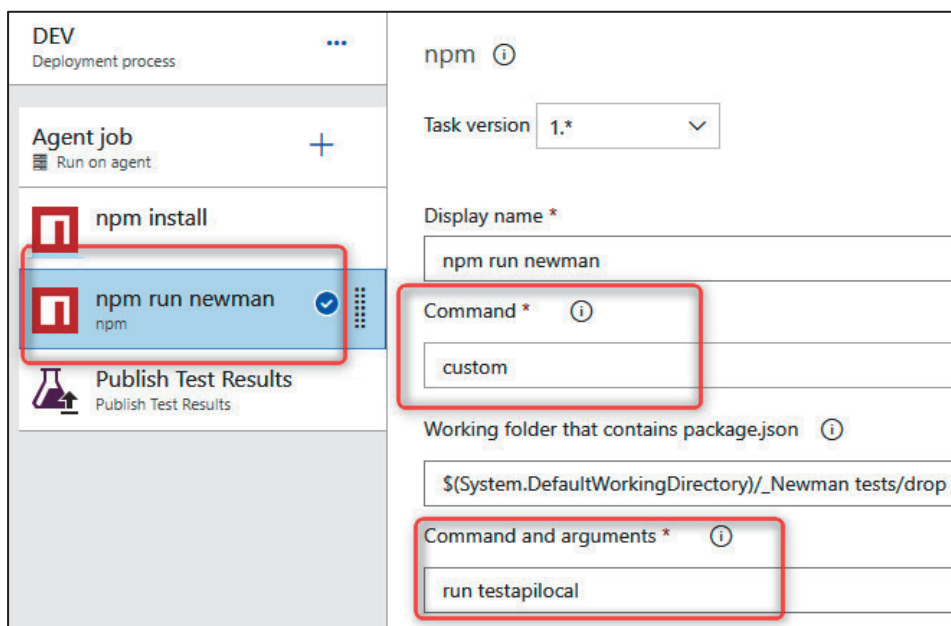
Parametry zadania `npm run newman` pokazuje rysunek 11.32.

W tym przypadku poleceniem, które ma zostać wykonane, jest `npm run testapi:local` w katalogu zawierającym pliki artefaktów. Polecenie `testapi:local` jest zdefiniowane w pliku `package.json` w sekcji skryptów (jak pokazano wcześniej) i uruchamia wiersz poleceń Newmana.

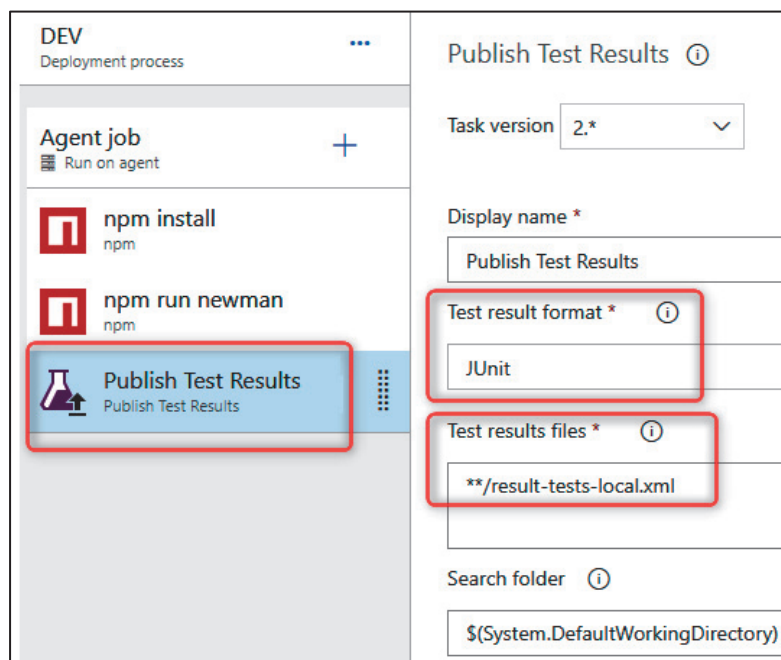
Publish Test Results

Parametry zadania *Publish Test Results*, które pozwala nam opublikować wyniki testów, pokazuje rysunek 11.33.

W parametrach tego zadania wskazujemy pliki raportowe JUnit XML, które są generowane przez Newmana, a w polu *Control Options* wybieramy opcję, że test ma zostać wykonany, nawet jeśli zadanie `npm run newman` się nie powiedzie.

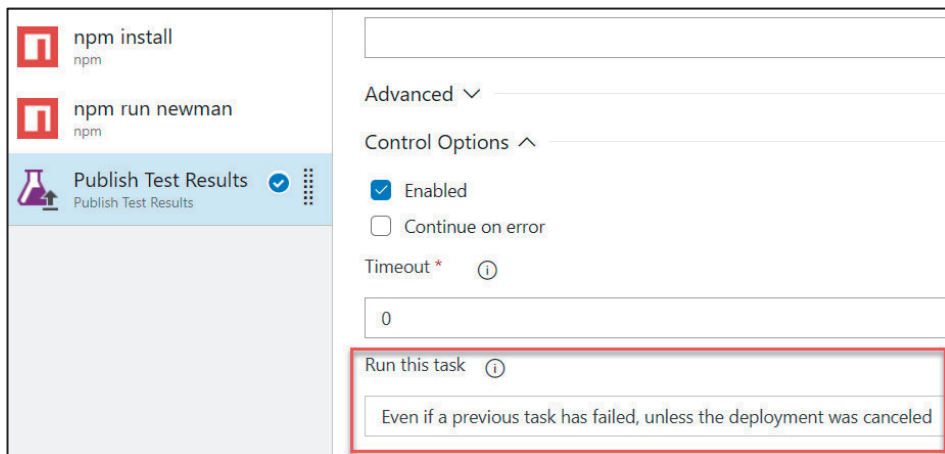


Rysunek 11.32. Uruchomienie Newmana w Azure Pipelines



Rysunek 11.33. Publikacja wyniku testów Newmana w Azure Pipelines

Poniższy zrzut ekranu pokazuje parametr *Control Options* dla uruchomienia zadania *Even if a previous task has failed, unless the deployment was canceled*:



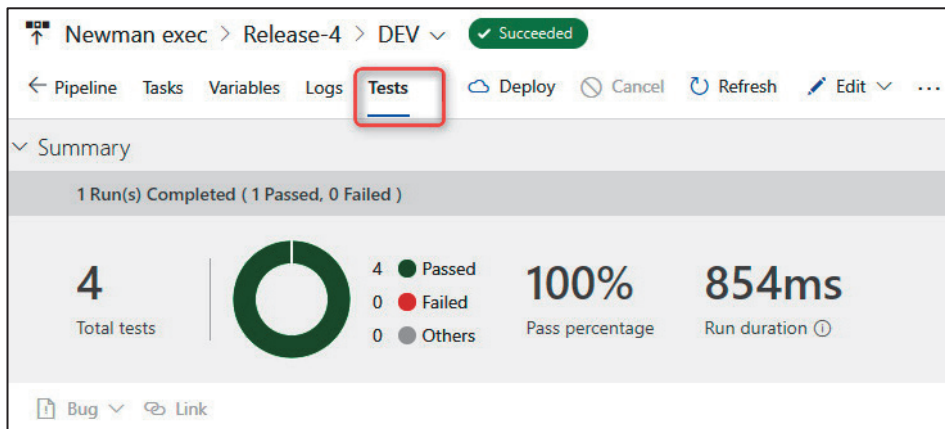
Rysunek 11.34. Opcja publikacji wyniku testów Newmana w Azure Pipelines

Konfiguracja potoku została zakończona — przejdziemy teraz do jego wykonania.

Wykonanie potoku

Po zakończeniu konfiguracji wydania aplikacji możemy ją uruchomić, a na koniec w zakładce *Tests* możemy zobaczyć raport z testów Newmana.

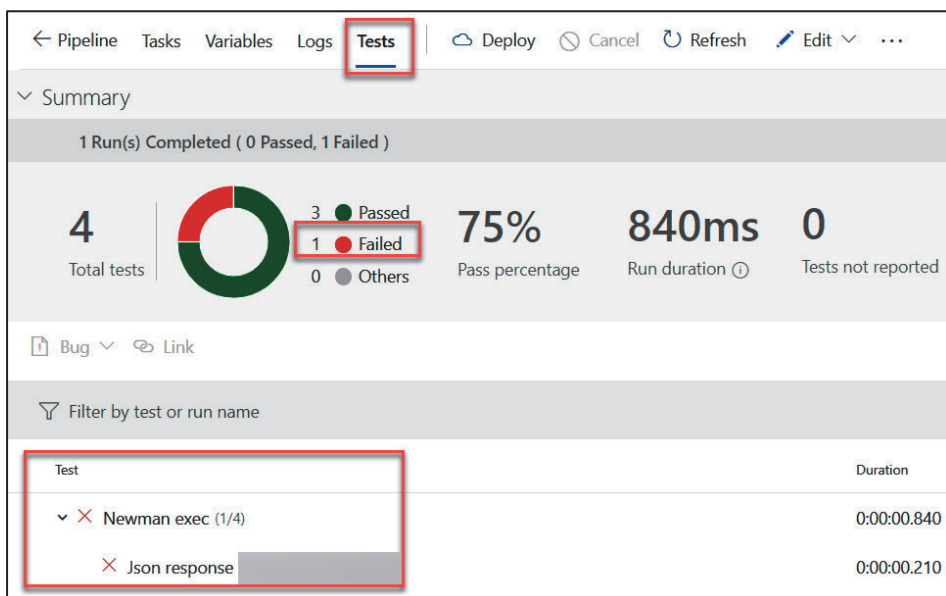
Poniższy zrzut ekranu przedstawia raportowanie testów Newmana w Azure Pipelines:



Rysunek 11.35. Widok wyników testów Newmana w usłudze Azure Pipelines

Wszystkie testy Postmana zakończyły się pomyślnie.

Oto zrzut ekranu przedstawiający raportowanie w przypadku niepowodzenia jednego z testów:



Rysunek 11.36. Wynik nieudanych testów Newmana w Azure Pipelines

Tak więc pomyślnie zintegrowaliśmy wykonanie Newmana, dzięki czemu mogliśmy zautomatyzować żądania naszego API, które skonfigurowaliśmy w Postmanie w potoku CI/CD.

Jeśli chodzi o integrację wykonania Newmana w **Jenkinsie**, przeczytaj dokumentację pod adresem https://learning.getpostman.com/docs/postman/collection_runs/integration_with_jenkins, a w przypadku integracji z **Travis CI** dokumentację można znaleźć pod adresem https://learning.getpostman.com/docs/postman/collection_runs/integration_with_travis.

W tej sekcji dowiedzieliśmy się, jak utworzyć i skonfigurować potok CI/CD w Azure Pipelines wykonujący testy Postmana, które zostały wyeksportowane dla Newmana.

Podsumowanie

W tym rozdziale przedstawiliśmy Postmana, który jest doskonałym narzędziem do testowania API. Utworzyliśmy konto w Postmanie i zainstalowaliśmy go lokalnie.

Następnie utworzyliśmy kolekcje i środowiska, w których utworzyliśmy żądania zawierające ustawienia naszych API, które mają zostać przetestowane.

Rozmawialiśmy również o automatyzacji tych testów za pomocą narzędzia wiersza poleceń Newman, z eksportem kolekcji i środowisk Postmana.

Wreszcie w ostatniej części rozdziału utworzyliśmy i wykonaliśmy potok CI/CD w Azure DevOps, który automatyzuje wykonywanie testów API w procesie DevOps.

W kolejnym rozdziale pozostaniemy przy tematyce testowania. Przyjrzymy się analizie kodu statycznego za pomocą dobrze znanego narzędzia o nazwie **SonarQube**.

Pytania

1. Jaki jest cel Postmana?
2. Jaki jest pierwszy element, który należy utworzyć w Postmanie?
3. Jak nazywa się element zawierający konfigurację API, która ma być testowana?
4. Które narzędzie w Postmanie pozwala nam wykonać wszystkie żądania kolekcji?
5. Które narzędzie pozwala nam zintegrować testy API Postmana z potokiem CI/CD?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o Postmanie, oto kilka dodatkowych zasobów:

- Centrum nauki Postmana — <https://learning.getpostman.com/>.
- Filmy i samouczki dotyczące Postmana — <https://www.getpostman.com/resources/videos-tutorials/>.

Statyczna analiza kodu za pomocą SonarQube

Rozdział

12

W poprzednim rozdziale przyjrzeliśmy się, jak testować funkcjonalność **interfejsu programowania aplikacji (API)** za pomocą Postmana (bezpłatnego narzędzia do testowania interfejsów API) oraz jak integrować i automatyzować te testy w potoku **ciągłej integracji/ciągłego wdrażania (CI/CD)** przy użyciu Newmana.

Testowanie funkcjonalności API lub aplikacji to dobra praktyka, gdy chcemy poprawić jej jakość. W firmie jakość aplikacji musi być brana pod uwagę przez wszystkich jej członków, ponieważ aplikacja, która przynosi użytkownikom wartość biznesową, zwiększa zyski firmy.

Jednak często zaniedbujemy testowanie jakości kodu aplikacji, ponieważ uważamy, że liczy się to, jak aplikacja działa, a nie jak jest zakodowana. Taki sposób myślenia jest dużym błędem, bo źle napisany kod może zawierać luki w zabezpieczeniach, a także powodować problemy z wydajnością. Ponadto jakość kodu wpływa na jego utrzymanie i skalowalność, gdyż kod, który jest zbyt złożony lub źle napisany, bywa trudny w utrzymaniu, a zatem naprawa może kosztować firmę więcej.

W tym rozdziale skupimy się na statycznej analizie kodu za pomocą dobrze znanego narzędzia o nazwie **SonarQube**. Przedstawimy jego krótki opis i powiemy, jak je zainstalować. Następnie użyjemy narzędzia SonarLint do lokalnej analizy kodu. Na koniec zintegrujemy SonarQube z potokiem CI/CD w Azure Pipelines.

W tym rozdziale omówimy następujące tematy:

- odkrywanie SonarQube,
- instalacja SonarQube,
- analiza w czasie rzeczywistym za pomocą SonarLint,
- wykonywanie SonarQube w procesie CI.

Wymagania techniczne

Aby korzystać z SonarQube i SonarLint, musimy zainstalować środowisko **Java Runtime Environment (JRE)**, które można znaleźć pod adresem <https://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html> (wymagane jest konto Oracle), na serwerze, na którym mamy SonarQube, oraz w lokalnym środowisku programistycznym, na którym mamy SonarLint.

Aby zintegrować SonarQube z potokiem Azure DevOps, musimy zainstalować następujące rozszerzenie w naszym środowisku Azure DevOps: <https://marketplace.visualstudio.com/items?itemName=SonarSource.sonarqube>.

Pełny kod źródłowy dla tego rozdziału jest dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP12>.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/3s96XTR>.

Odkrywanie SonarQube

SonarQube to narzędzie typu open source firmy SonarSource (<https://www.sonarsource.com/>) napisane w Javie. Pozwala nam przeprowadzać statyczną analizę kodu w celu weryfikacji jakości i bezpieczeństwa kodu aplikacji.

SonarQube jest przeznaczony dla zespołów programistów i zapewnia im pulpit nawigacyjny oraz raporty, które można tak dostosować, by programiści mogli zaprezentować stan jakości kodu swoich aplikacji.

Pozwala na analizę kodu statycznego w wielu językach (ponad 25), takich jak **PHP** (ang. *Hypertext Preprocessor*), Java, .NET, JavaScript, Python itd. Pełną listę można znaleźć na stronie <https://www.sonarqube.org/features/multi-languages/>.

Dodatkowo, poza analizą kodu związaną z kwestiami bezpieczeństwa, zapachem kodu i duplikacją kodu, SonarQube zapewnia również test pokrycia dla testów jednostkowych. Aby uzyskać więcej informacji na temat tych problemów, przeczytaj dokumentację: <https://docs.sonarqube.org/latest/user-guide/concepts/>.

Wreszcie SonarQube bardzo dobrze integruje się z potokami CI/CD, dzięki czemu może automatyzować analizę kodu podczas zatwierdzania kodu przez programistę. Zmniejsza to ryzyko wdrożenia aplikacji, która ma luki w zabezpieczeniach lub zbyt dużą złożoność kodu.

Z drugiej strony należy zauważyć, że SonarQube posiada wiele wtyczek, za które można zapłacić. Lista wtyczek jest dostępna tutaj: <https://www.sonarplugins.com/>.

Po tym jak przedstawiliśmy przegląd SonarQube, przyjrzymy się jego architekturze i komponentom. Na koniec przyjrzymy się różnym sposobom instalacji.

Instalacja SonarQube

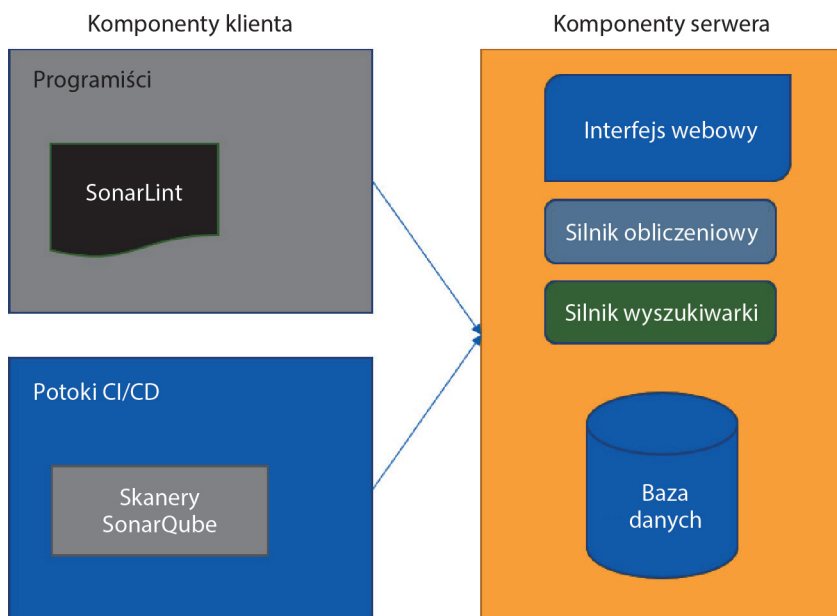
SonarQube to rozwiązanie lokalne (ang. *on-premises*), innymi słowy: musi być zainstalowane na serwerach lub **maszynach wirtualnych (VM)**. Ponadto SonarQube składa się z kilku komponentów, które będą analizować kod źródłowy aplikacji, pobierać i przechowywać dane z tej analizy oraz dostarczać raporty dotyczące jakości i bezpieczeństwa kodu.

Zanim zainstalujemy SonarQube, koniecznie przyjrzymy się jego architekturze i komponentom.

Przegląd architektury SonarQube

SonarQube jest narzędziem typu klient/serwer, co oznacza, że jego architektura składa się z artefaktów zarówno po stronie serwera, jak i po stronie klienta.

Uproszczoną architekturę SonarQube pokazano na poniższym diagramie:



Rysunek 12.1. Architektura SonarQube z komponentami klienta i serwera

Spójrzmy na komponenty pokazane na powyższym diagramie. Komponenty tworzące SonarQube po stronie serwera są wymienione tutaj:

- Baza danych SQL Server, MySQL, Oracle lub PostgreSQL zawierająca wszystkie dane analityczne.
- Aplikacja internetowa wyświetlająca pulpity nawigacyjne.
- Silnik obliczeniowy odpowiedzialny za pobieranie analiz i procesów oraz umieszczanie ich w bazie danych.
- Wyszukiwarka zbudowana za pomocą narzędzia Elasticsearch.

Komponenty po stronie klienta są wymienione tutaj:

- Skaner, który skanuje kod źródłowy aplikacji i przesyła dane do silnika obliczeniowego.
- Skaner jest zwykle instalowany na agentach kompilacji używanych do wykonywania potoków CI/CD.
- SonarLint to narzędzie instalowane na stacjach roboczych programistów, służące do analizy w czasie rzeczywistym.

Przyjrzymy się temu szczegółowo w dalszej części tego rozdziału.

Aby uzyskać więcej informacji, możemy zapoznać się z dokumentacją dotyczącą architektury i integracji SonarQube, którą można znaleźć pod adresem <https://docs.sonarqube.org/latest/user-guide/concepts/>.

Teraz, gdy przyjrzeliliśmy się jego architekturze i komponentom, nauczymy się, jak zainstalować SonarQube.

Instalacja SonarQube

SonarQube można zainstalować na różne sposoby — ręcznie lub instalując kontener Dockera z obrazu Sonar. Alternatywnie, jeśli mamy subskrypcję Azure, możemy skorzystać z maszyny wirtualnej SonarQube z Azure Marketplace. Przyjrzymy się bliżej każdej z tych opcji.

Ręczna instalacja SonarQube

Jeśli chcemy ręcznie zainstalować serwer SonarQube, musimy wziąć pod uwagę wymagania wstępne. Warunkiem wstępnym jest to, że Java musi być już zainstalowana na serwerze i że musimy sprawdzić konfigurację sprzętową pokazaną na stronie <https://docs.sonarqube.org/latest/requirements/requirements/>.

Następnie musimy ręcznie zainstalować komponenty serwera w następującej kolejności:

1. Zainstaluj bazę danych. Może to być MSSQL, Oracle, PostgreSQL lub MySQL.
2. Dla aplikacji webowej pobierz Community Edition SonarQube ze strony <https://www.sonarqube.org/downloads/> i rozpakuj pobrany plik ZIP.
3. W pliku `$SONARQUBE-HOME/conf/sonar.properties` skonfiguruj dostęp do bazy danych, którą zainstalowaliśmy w kroku 1., i ścieżkę magazynu dla Elasticsearch, zgodnie z poniższą dokumentacją: <https://docs.sonarqube.org/latest/setup/install-server/>.
4. Uruchom serwer WWW.

Aby poznać wszystkie szczegóły dotyczące tej instalacji zgodnie z wybraną bazą danych i naszym **systemem operacyjnym (OS)**, możemy zapoznać się z następującą dokumentacją: <https://docs.sonarqube.org/latest/setup/install-server/>.

Instalacja za pomocą Dockera

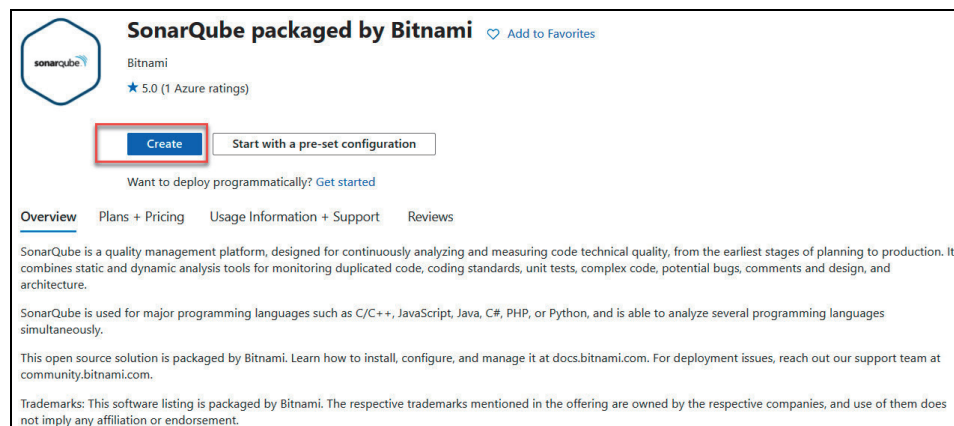
Jeśli chcemy zainstalować wersję SonarQube Community w celach testowych lub demonstracyjnych, możemy ją zainstalować za pomocą oficjalnego obrazu Dockera, który jest dostępny w Docker Hubie pod adresem https://hub.docker.com/_/sonarqube/.

Uważaj, ponieważ ten obraz wykorzystuje małą zintegrowaną bazę danych, która nie jest przeznaczona dla produkcji.

Instalacja na platformie Azure

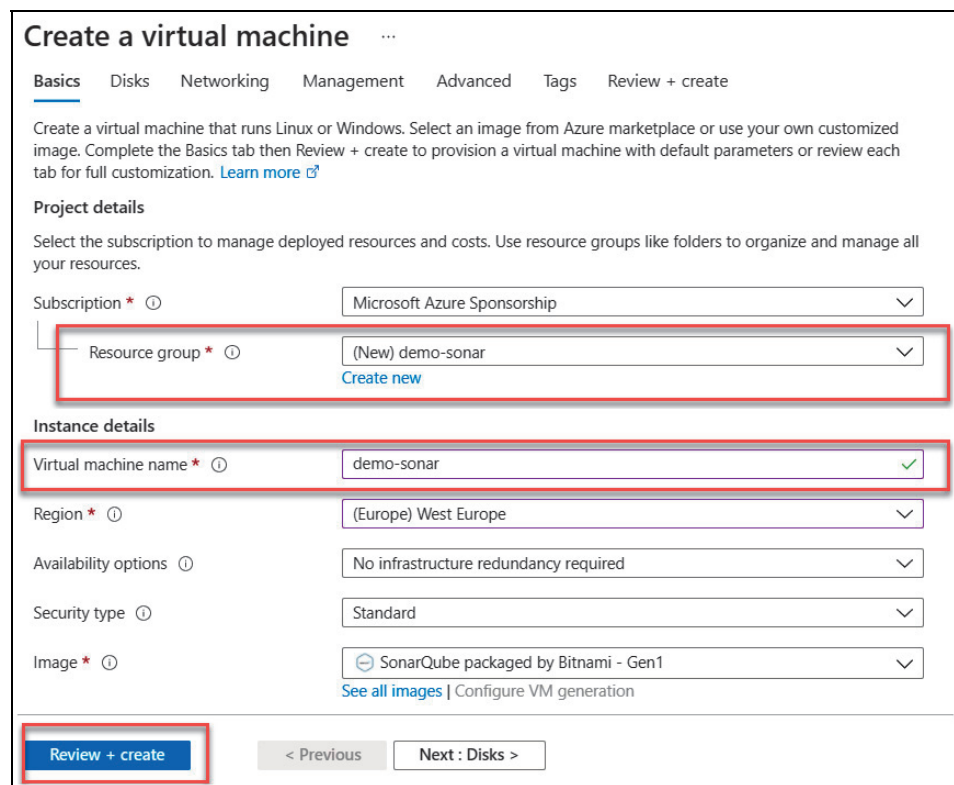
Jeśli mamy subskrypcję Azure, możemy szybko uzyskać dostęp do całego serwera SonarQube za pomocą maszyny wirtualnej SonarQube z Azure Marketplace. Wykonaj następujące kroki, aby utworzyć maszynę wirtualną SonarQube na platformie Azure:

1. W witrynie Azure Marketplace wyszukaj i wybierz obraz *SonarQube*. Rysunek 12.2 przedstawia stronę SonarQube w Marketplace.
2. Kliknij przycisk *Create*.
3. W formularzu maszyny wirtualnej na karcie *Basics* wybierz typ *Resource group* i podaj informacje w polu *Virtual machine name*, jak pokazano na rysunku 12.3.
4. Możemy również zmienić niektóre opcje VM w zakładkach *Disks* i *Networking*. Następnie zatwierdzamy te zmiany, klikając przycisk *Review + create*.
5. Pod koniec tworzenia zasobu w portalu Azure możemy zobaczyć status wdrożenia, który w tym przypadku jest pomyślny — rysunek 12.4.



The screenshot shows the Azure Marketplace page for 'SonarQube packaged by Bitnami'. The page includes the SonarQube logo, the Bitnami publisher name, and a 5.0 star rating from 1 Azure rating. There are two buttons: 'Create' (highlighted with a red box) and 'Start with a pre-set configuration'. Below these buttons is a link 'Want to deploy programmatically? Get started'. The page has tabs for 'Overview', 'Plans + Pricing', 'Usage Information + Support', and 'Reviews'. The 'Overview' tab is selected, showing a description of SonarQube as a quality management platform. It lists supported programming languages: C/C++, JavaScript, Java, C#, PHP, and Python. It also mentions that the solution is open source and packaged by Bitnami, with a link to the documentation at docs.bitnami.com. A disclaimer at the bottom states that trademarks are owned by their respective companies.

Rysunek 12.2. Azure SonarQube w Marketplace



The screenshot shows the 'Create a virtual machine' wizard in the Azure portal. The 'Basics' tab is selected. The wizard prompts the user to create a virtual machine running Linux or Windows. The 'Project details' section shows the 'Subscription' as 'Microsoft Azure Sponsorship' and the 'Resource group' as '(New) demo-sonar' (highlighted with a red box). Below this is a link 'Create new'. The 'Instance details' section shows the 'Virtual machine name' as 'demo-sonar' (highlighted with a red box), the 'Region' as '(Europe) West Europe', 'Availability options' as 'No infrastructure redundancy required', 'Security type' as 'Standard', and the 'Image' as 'SonarQube packaged by Bitnami - Gen1' (highlighted with a red box). At the bottom, there is a 'Review + create' button (highlighted with a red box), a '< Previous' button, and a 'Next : Disks >' button.

Rysunek 12.3. Tworzenie SonarQube w Azure

✓

Your deployment is complete

⊖

Deployment name: CreateVm-bitnami.sonarqube-6-4-2019072815... Start time: 7/28/2019, 3:40:26 PM
Subscription: [Microsoft Azure Sponsorship](#) Correlation ID: 6bfacfb4-78eb-4e07-8f79-8
Resource group: [demo-sonar](#)

^ Deployment details [\(Download\)](#)

| RESOURCE | TYPE | STATUS | OPERATION DETAILS |
|-----------------------------------|----------------------------|---------|-----------------------------------|
| ✓ demo-sonar | Microsoft.Compute/virt... | OK | Operation details |
| ✓ demo-sonar592 | Microsoft.Network/netw... | Created | Operation details |
| ✓ demosonardiag | Microsoft.Storage/stora... | OK | Operation details |
| ✓ demo-sonar-vnet | Microsoft.Network/virtu... | OK | Operation details |
| ✓ demo-sonar-nsg | Microsoft.Network/netw... | OK | Operation details |
| ✓ demo-sonar-ip | Microsoft.Network/publ... | OK | Operation details |

Rysunek 12.4. Wdrożenie SonarQube w Azure

6. Aby uzyskać dostęp do zainstalowanego serwera SonarQube, wyświetl szczegóły maszyny wirtualnej i uzyskaj wartość publicznego adresu IP, jak pokazano na poniższym zrzucie ekranu:

Dashboard > CreateVm-bitnami.sonarqube-6-4-20190728153635 - Overview > demo-sonar

demo-sonar

Virtual machine

Search (Ctrl+J)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Settings

Networking

Connect Start Restart Stop Capture Delete Refresh

Resource group (change) : [demo-sonar](#)

Status : Running

Location : West Europe

Subscription (change) : [Microsoft Azure Sponsorship](#)

Subscription ID : 8a7aace5-74aa-416f-b8e4-2c292b6304e5

Computer name : demo-sonar

Operating system : Linux (ubuntu 16.04)

Size : Standard D2 v2 (2 vcpus, 7 GiB memory)

Ephemeral OS disk : N/A

Public IP address : 168.1.1.1

Private IP address : 10.0.0.4

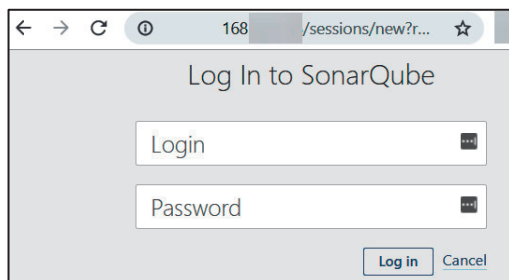
Virtual network/subnet : [demo-sonar-vnet/default](#)

DNS name : Configure

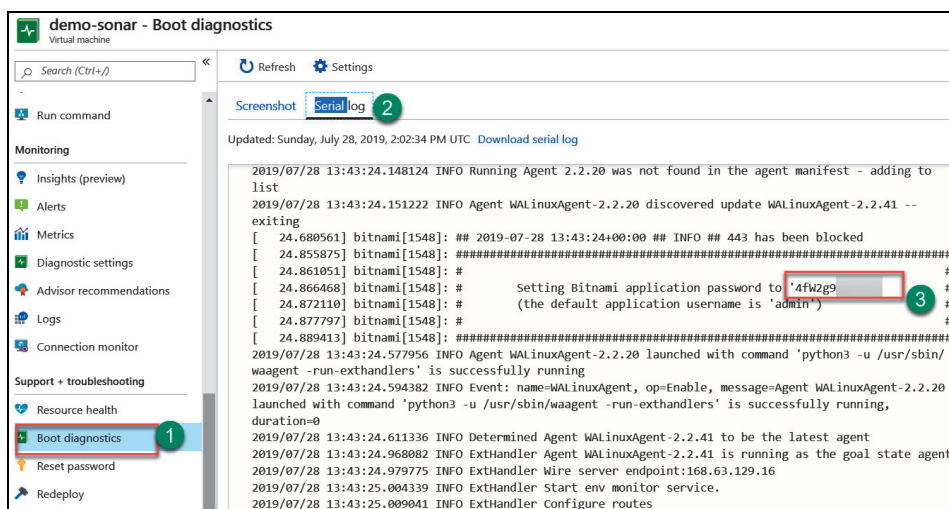
Rysunek 12.5. Adres IP SonarQube w Azure

Otwórz przeglądarkę internetową z tym adresem IP, podając go jako **URL** (ang. *Uniform Resource Locator*). Zostanie wyświetlona strona uwierzytelniania SonarQube, jak pokazano na rysunku 12.6.

Domyślny login to *admin*. Informacja na ten temat jest dostępna za pośrednictwem informacji diagnostycznych rozruchu maszyny wirtualnej, jak wskazano w dokumentacji (<https://docs.bitnami.com/azure/faq/get-started/find-credentials/>). Rysunek 12.7 pokazuje, jak przeprowadzić odzyskiwanie hasła za pomocą opcji *Boot diagnostics*.

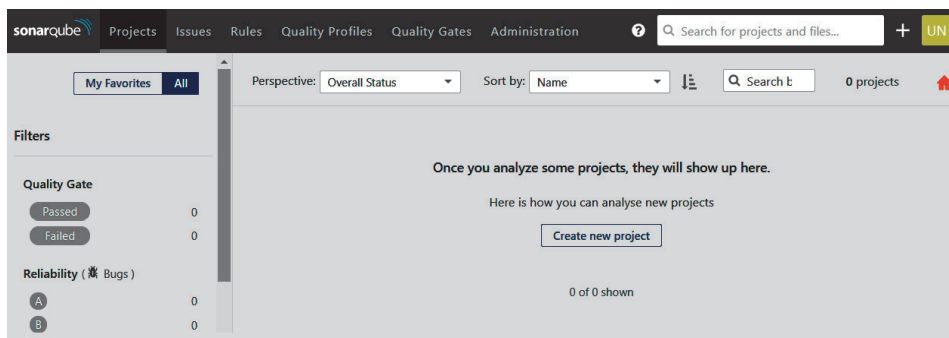


Rysunek 12.6. Ekran logowania SonarQube



Rysunek 12.7. Odzyskiwanie hasła SonarQube

Po uwierzytelnieniu możemy uzyskać dostęp do pulpitu nawigacyjnego SonarQube, który wygląda tak:



Rysunek 12.8. Strona główna SonarQube

Nauczyliśmy się, jak zainstalować SonarQube w maszynie wirtualnej, więc teraz nauczymy się instalować SonarQube na Kubernetesie.

Instalowanie SonarQube na Kubernetesie

Do instalacji SonarQube na klastrze Kubernetesa użyjemy pakietu *helm-chart* dostępnego tutaj: <https://github.com/SonarSource/helm-chart-sonarqube/tree/master/charts/sonarqube>.

Przed zainstalowaniem SonarQube na Kubernetesie ważne jest, aby zapoznać się z wymaganiami: <https://docs.sonarqube.org/latest/setup/sonarqube-on-kubernetes/>.

Aby zainstalować SonarQube na Kubernetesie, uruchom następujący skrypt:

```
helm repo add sonarqube https://SonarSource.github.io/helmchart-sonarqube
helm repo update
kubectl create namespace sonarqube
helm upgrade --install -n sonarqube sonarqube/sonarqube
```

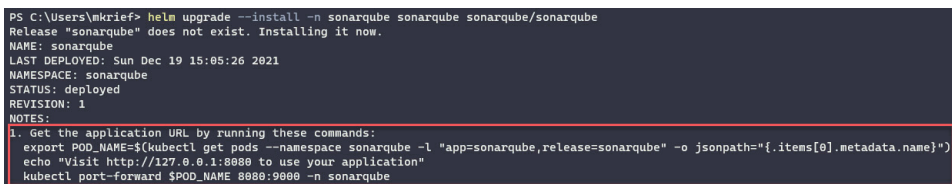
Pierwsza linia tego skryptu dodaje lokalnie repozytorium SonarQube do Helma.

Drugi wiersz aktualizuje to repozytorium.

Następnie skrypt tworzy przestrzeń nazw sonarqube.

Ostatnia linia instaluje chart Helma o nazwie sonarqube.

Pod koniec działania skryptu wykonanie wykresu Helma wyświetla skrypt służący do użycia w konsoli zainstalowanej instancji SonarQube, jak pokazano na poniższym zrzucie ekranu:



```
PS C:\Users\mkrief> helm upgrade --install -n sonarqube sonarqube sonarqube/sonarqube
Release "sonarqube" does not exist. Installing it now.
NAME: sonarqube
LAST DEPLOYED: Sun Dec 19 15:05:26 2021
NAMESPACE: sonarqube
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace sonarqube -l "app=sonarqube,release=sonarqube" -o jsonpath="{.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl port-forward $POD_NAME 8080:9080 -n sonarqube
```

Rysunek 12.9. Instalacja SonarQube na Kubernetesie z Helmem

Uwaga

Przed wykonaniem tego skryptu sprawdź, czy wszystkie pody działają, wykonując następujące polecenie:

```
kubectl get pods -n sonarqube
```

W tej sekcji przyjrzelśmy się architekturze SonarQube wraz ze szczegółami dotyczącymi komponentów klienta i serwera. Następnie przyjrzelśmy się różnym dostępnym metodom instalacji i konfiguracji SonarQube. W następnej sekcji zobaczymy, w jaki sposób programiści mogą przeprowadzać analizę kodu w czasie rzeczywistym za pomocą SonarLint, zanim zatwierdzą swój kod.

Analiza w czasie rzeczywistym za pomocą SonarLint

Deweloperzy korzystający z SonarQube w kontekście CI często napotykają problem zbyt długiego oczekiwania na wyniki analizy SonarQube. Muszą zatwierdzić swój kod i poczekać na koniec potoku CI, zanim otrzymają wyniki analizy kodu.

Aby rozwiązać ten problem, a tym samym poprawić codzienne życie programistów, SonarSource — edytor SonarQube — zapewnia inne narzędzie, **SonarLint**, które umożliwia analizę kodu w czasie rzeczywistym.

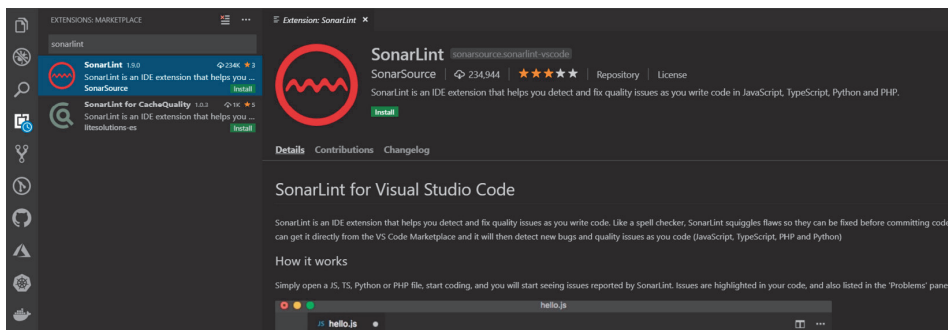
SonarLint to bezpłatne narzędzie typu open source (<https://www.sonarlint.org/>), które pobiera się w różny sposób w zależności od narzędzia programistycznego i języka programowania.

SonarLint jest dostępny dla **zintegrowanych środowisk programistycznych (IDE)**: Eclipse, IntelliJ IDEA, Visual Studio i **Visual Studio Code (VS Code)**.

W tej książce przyjrzymy się przykładowi korzystającemu z SonarLint w aplikacji napisanej w TypeScriptie przy użyciu VS Code IDE. Warunkiem wstępnym korzystania z SonarLint jest zainstalowanie środowiska JRE na lokalnym komputerze deweloper-skim. Można go pobrać ze strony <https://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html>.

Aby dowiedzieć się więcej o konkretnym zastosowaniu SonarLint, wykonaj następujące kroki:

1. W VS Code zainstaluj rozszerzenie SonarLint, przechodząc do następującej strony w witrynie Azure Marketplace — rysunek 12.10.
2. Następnie w VS Code, w ustawieniach użytkownika skonfiguruj rozszerzenie, podając ścieżkę instalacji JRE jak na rysunku 12.11.
3. W naszym projekcie utwórz folder *tsApp*. W tym folderze utwórz plik *app.ts* zawierający kod naszej aplikacji. Kod źródłowy jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP12/tsApp/app.ts>.

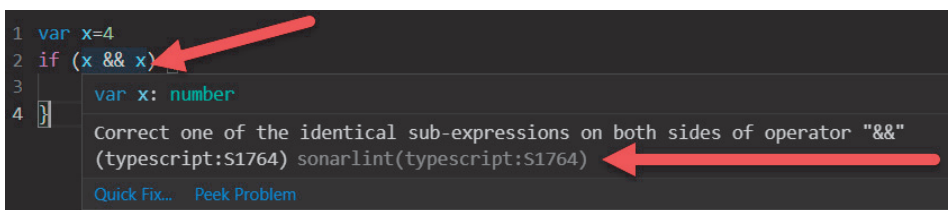


Rysunek 12.10. Rozszerzenie SonarLint w VS Code



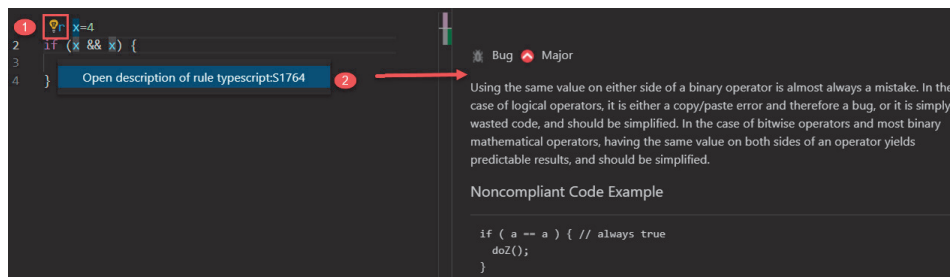
Rysunek 12.11. Konfiguracja rozszerzenia SonarLint w VS Code

4. Zauważ, że w tym przykładowym kodzie SonarLint stwierdza się, że kod jest nieprawidłowy, jak pokazano na poniższym zrzucie ekranu:



Rysunek 12.12. Przykładowe sprawdzenie kodu przez SonarLint

SonarLint pozwala nam dowiedzieć się więcej o tym błędzie, wyświetlając szczegółowe informacje dotyczące błędu i sposobu naprawy, jak pokazano na poniższym zrzucie ekranu:



Rysunek 12.13. Szczegóły na temat błędnego kodu prezentowane przez SonarLint

W ten sposób SonarLint i jego integracja z różnymi środowiskami IDE pozwalają nam wykrywać statyczne błędy kodu w czasie rzeczywistym tak szybko, jak to możliwe — tzn. w czasie, gdy programista pisze swój kod, i zanim zatwierdzi zmiany w systemie kontroli wersji kodu źródłowego.

W tej sekcji dowiedzieliśmy się, jak zainstalować SonarLint w VS Code i jak go używać do przeprowadzania analizy kodu w czasie rzeczywistym.

W następnej sekcji opowiemy, jak zintegrować analizę SonarQube z procesem CI w Azure Pipelines.

Wykonywanie SonarQube w procesie CI

Do tej pory w tym rozdziale dowiedzieliśmy się, jak zainstalować SonarQube, i widzieliśmy, jak programiści używają SonarLint na swoich lokalnych komputerach.

Teraz zobaczymy, jak przeprowadzić analizę kodu podczas procesu CI, aby się upewnić, że za każdym razem, gdy zostanie dokonane zatwierdzenie kodu, możemy sprawdzić kod aplikacji dostarczony przez wszystkich członków zespołu.

Aby zintegrować SonarQube z procesem CI, będziemy musieli wykonać następujące czynności:

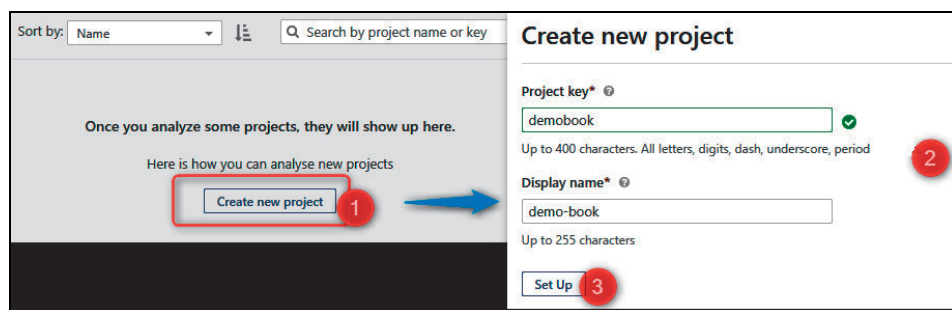
1. Skonfiguruj SonarQube, tworząc nowy projekt.
2. Utwórz i skonfiguruj kompilację CI w Azure Pipelines.

Zacznijmy od zbadania, jak utworzyć nowy projekt w SonarQube.

Konfigurowanie SonarQube

Konfiguracja SonarQube polega na utworzeniu nowego projektu i pobraniu tokena identyfikacyjnego. Aby utworzyć nowy projekt, wykonaj następujące kroki:

1. Kliknij łącze *Create new project* na pulpicie nawigacyjnym.
2. W formularzu wprowadź unikalny klucz **demobook** i nazwę dla tego projektu: **demo-book**.
3. Aby zatwierdzić ustawienia, kliknij przycisk *Set Up*. Utworzy się projekt. Kroki przedstawiono na poniższym zrzucie ekranu:



Rysunek 12.14. Tworzenie projektu SonarQube

Zaraz po utworzeniu projektu asystent SonarQube proponuje nam utworzenie tokena (unikatowego klucza), który posłuży do analizy.

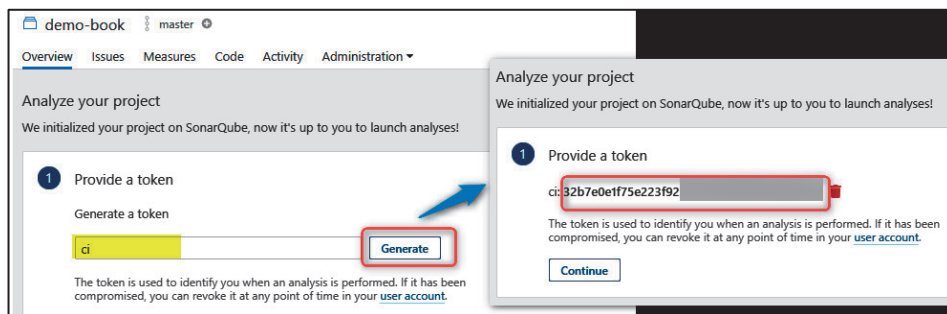
Aby wygenerować i utworzyć ten token, wykonaj następujące kroki:

1. W polu wejściowym wpisz żądaną nazwę tokena.
2. Następnie zatwierdź go, klikając przycisk *Generate*.
3. Unikalny klucz zostanie wyświetlony na ekranie. Klucz jest naszym tokenem i musimy go chronić. Rysunek 12.15 przedstawia kroki generowania tokena:

Konfiguracja SonarQube z naszym nowym projektem została zakończona. Teraz skonfigurujemy potok CI do przeprowadzenia analizy SonarQube.

Tworzenie potoku CI dla SonarQube w Azure Pipelines

Aby zilustrować integrację analizy SonarQube z potokiem CI, użyjemy Azure Pipelines, które szczegółowo omówiliśmy w rozdziale 7., „Ciągła integracja i ciągłe wdrażanie”.



Rysunek 12.15. Generowanie tokena SonarQube

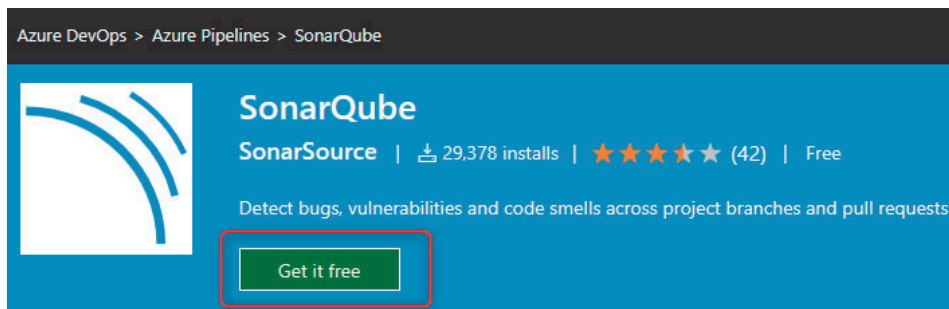
Aplikacja, której użyjemy jako przykładu, została napisana w Node.js. Jest to prosty kalkulator zawierający pewne metody, w tym metody testów jednostkowych.

Uwaga

Należy zauważyć, że celem tej sekcji nie jest omawianie kodu aplikacji, ale raczej potoku. Dostęp do kodu źródłowego aplikacji można uzyskać pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP12/AppDemo>.

Aby używać SonarQube w Azure Pipelines, musimy zainstalować **rozszerzenie SonarQube** w naszej organizacji Azure DevOps. Ten dodatek znajduje się pod adresem <https://marketplace.visualstudio.com/items?itemName=SonarSource.sonarqube>, jak opisano w sekcji wymagań technicznych tego rozdziału.

Poniższy zrzut ekranu przedstawia nagłówek i przycisk instalowania rozszerzenia w Visual Studio Marketplace:

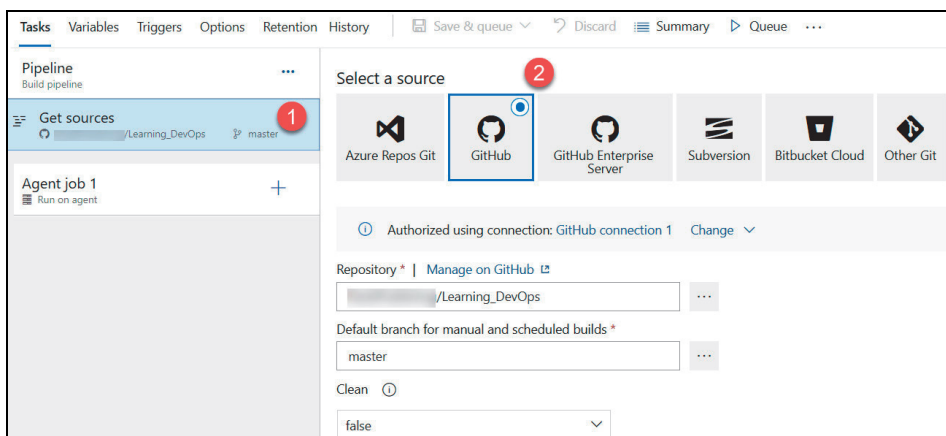


Rysunek 12.16. Rozszerzenie SonarQube w Azure DevOps

Po zainstalowaniu rozszerzenia możemy skonfigurować naszą kompilację CI.

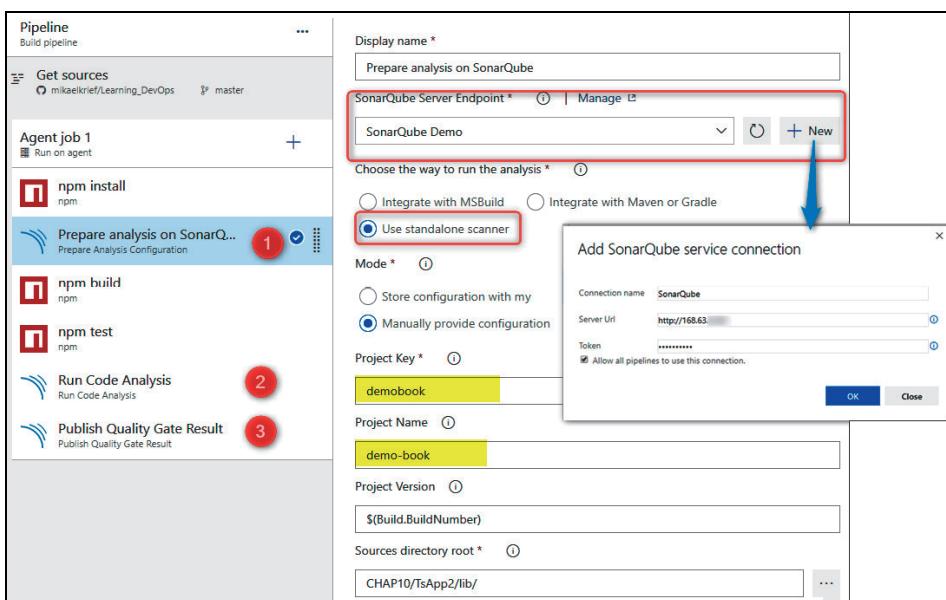
W Azure Pipelines utworzymy nową definicję kompilacji z następującą konfiguracją:

1. W zakładce *Get Sources* wybierz repozytorium i gałąź zawierającą kod źródłowy aplikacji, jak pokazano na poniższym zrzucie ekranu:



Rysunek 12.17. Repozytorium wybrane dla usługi Azure DevOps

2. Następnie w zakładce *Tasks* skonfiguruj harmonogram zadań, który przedstawia się następująco:



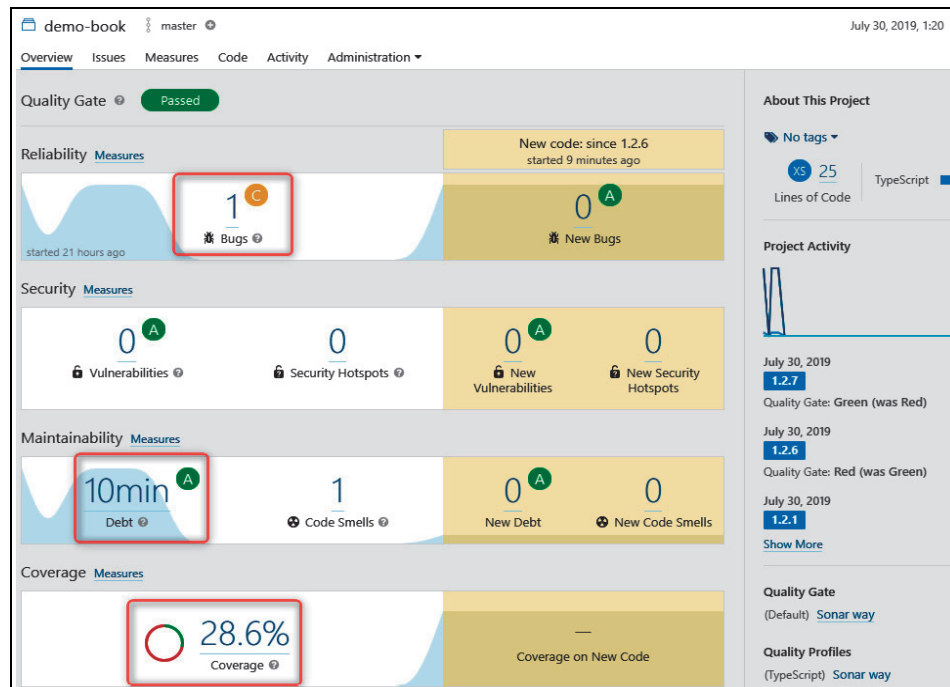
Rysunek 12.18. SonarQube w Azure Pipelines — przygotowywanie analizy

Oto szczegóły konfiguracji tych zadań:

1. Zadanie *Prepare analysis on SonarQube* obejmuje skonfigurowanie SonarQube za pomocą następujących elementów:
 - Usługa punktu końcowego, czyli połączenie z SonarQube wraz z jego adresem URL i tokenem, który wygenerowaliśmy wcześniej w konfiguracji SonarQube.
 - Klucz i nazwa projektu SonarQube.
 - Numer wersji analizy.
2. Następnie budujemy i wykonujemy testy jednostkowe aplikacji za pomocą poleceń `npm build` i `npm test`.
3. Zadanie *Run Code Analysis* pobiera wyniki testów, analizuje kod TypeScriptu naszej aplikacji i przesyła dane z analizy na serwer SonarQube.

Następnie zapisujemy konfigurację, rozpoczynamy wykonywanie kompilacji w procesie CI i czekamy, aż się zakończy.

Pulpit nawigacyjny SonarQube został zaktualizowany o analizę kodu, jak pokazano na poniższym zrzucie ekranu:



Rysunek 12.19. Analiza na pulpicie SonarQube

Tutaj możemy zobaczyć pomiary liczby błędów, ocenę trudności naprawy kodu, a także pokrycie kodu. Klikając każdy z tych fragmentów danych, możemy uzyskać dostęp do szczegółów elementu.

W tej sekcji zobaczyliśmy, jak zintegrować analizę SonarQube z potokiem CI, który dostarcza pulpit nawigacyjny wraz z wynikami i raportami analizy kodu każdego zatwierdzenia kodu.

Podsumowanie

W tym rozdziale zobaczyliśmy, jak analizować statyczny kod aplikacji za pomocą SonarQube. Ta analiza może wykrywać problemy ze składnią kodu i luki w kodzie, a także wskazywać pokrycie kodu zapewniane przez testy jednostkowe.

Następnie szczegółowo omówiliśmy użycie SonarLint, które pozwala programistom sprawdzać ich kod w czasie rzeczywistym podczas pisania kodu.

Na koniec przyjrzelśmy się konfiguracji SonarQube i jego integracji z procesem CI, aby zapewnić ciągłą analizę, która będzie uruchamiana przy każdym zatwierdzeniu kodu przez członka zespołu.

W następnym rozdziale przyjrzymy się niektórym praktykom bezpieczeństwa, przeprowadzając testy bezpieczeństwa za pomocą narzędzia **Zed Attack Proxy (ZAP)**, testy wydajności za pomocą Postmana i uruchamiając testy obciążenia za pomocą Azure DevOps.

Pytania

1. W jakim języku został opracowany SonarQube?
2. Jakie są wymagania dotyczące instalacji SonarQube?
3. Jaka jest rola SonarQube?
4. Jak nazywa się narzędzie, które umożliwia programistom analizę w czasie rzeczywistym?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o SonarQube, oto źródło, które Ci w tym pomoże:

- Dokumentacja SonarQube — <https://docs.sonarqube.org/latest>.

Testy bezpieczeństwa i wydajności

W rozdziale 11., „Testowanie interfejsów API za pomocą Postmana”, i rozdziale 12., „Statyczna analiza kodu za pomocą SonarQube”, omówiliśmy odpowiednio automatyzację testów API za pomocą Postmana i statyczną analizę kodu za pomocą SonarQube.

W tym rozdziale powiemy, jak przeprowadzić testy bezpieczeństwa i testy penetracyjne aplikacji internetowej przy użyciu narzędzia ZAP opartego na zaleceniach OWASP. Następnie wprowadzimy Postmana, abyśmy mogli przeprowadzać testy wydajności na interfejsach API.

Będziemy poruszać następujące tematy:

- stosowanie zabezpieczeń internetowych i testów penetracyjnych za pomocą narzędzia ZAP,
- przeprowadzanie testów wydajnościowych za pomocą Postmana.

Wymagania techniczne

Aby korzystać z ZAP-a, musimy zainstalować środowisko Java Runtime Environment (JRE), które jest dostępne pod adresem <https://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html> (wymagane jest konto Oracle).

W tym rozdziale wykorzystamy Postmana, o którym mówiliśmy w rozdziale 11., „Testowanie interfejsów API za pomocą Postmana”.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/3HbOgD0>.

Stosowanie zabezpieczeń internetowych i testów penetracyjnych za pomocą narzędzia ZAP

Obecnie bezpieczeństwo aplikacji musi się znajdować w centrum zainteresowania firm. Gdy tylko aplikacja internetowa (lub strona internetowa) zostanie upubliczniona w internecie, staje się kandydatem do złośliwych ataków. Ponadto należy pamiętać, że bezpieczeństwo aplikacji jest jeszcze ważniejsze, jeśli jest ona używana do przechowywania poufnych danych, takich jak konta bankowe lub dane osobowe.

Aby rozwiązać ten problem, powstał **Open Web Application Security Project (OWASP)** (https://www.owasp.org/index.php/Main_Page), ogólnosiwiatowa organizacja badająca kwestie bezpieczeństwa aplikacji. Celem tej organizacji jest publiczne zwracanie uwagi na problemy bezpieczeństwa i luki, które można napotkać w systemie aplikacji. Oprócz tych cennych informacji o zabezpieczeniach OWASP zapewnia zalecenia, rozwiązania i narzędzia do testowania i ochrony aplikacji.

Jednym z ważnych i przydatnych projektów i dokumentów dostarczanych przez OWASP jest lista 10 najistotniejszych problemów związanych z bezpieczeństwem aplikacji. Dokument ten jest dostępny pod adresem <https://owasp.org/www-project-top-ten/>. Jest bardzo szczegółowy i zawiera wyjaśnienia, przykłady oraz rozwiązania każdego problemu związanego z bezpieczeństwem. W tym dokumencie widać, że największą luką w zabezpieczeniach, na którą aplikacje są najbardziej narażone, jest luka polegająca na wstrzykiwaniu kodu (ang. *injection*), np. wstrzyknięciu SQL. Polega ona na nieuprawnionym dodaniu kodu lub żądań do aplikacji w celu zebrania, usunięcia lub uszkodzenia danych z aplikacji.

W chwili pisania tego tekstu 10 najlepszych technik ograniczania skutków OWASP to:

1. Bieżąca ocena ryzyka.
2. Używanie zautomatyzowanych i ręcznych środków do oceny ryzyka.
3. Używanie silnej **zapory aplikacji webowej** (ang. *Web Application Firewall* — **WAF**).
4. Upewnienie się, że framework służący do tworzenia stron webowych i praktyki kodowania mają wbudowane zabezpieczenia.
5. Wymuszenie **uwierzytelniania wieloskładnikowego** (ang. *multi-factor authentication* — **MFA**).
6. Szyfrowanie wszystkich danych.
7. Stosowanie wszystkich aktualizacji oprogramowania.

8. Sanityzacja aplikacji webowej.
9. Prowadzenie formalnych inicjatyw uświadamiających zagrożenia.
10. Przestrzeganie standardów OWASP.

Aby uzyskać więcej informacji na temat tych technik łagodzenia skutków zagrożeń, przeczytaj następujący artykuł: <https://www.indusface.com/blog/owasp-top-10-mitigation-techniques/>.

OWASP opisuje również inną znaną lukę w zabezpieczeniach, którą jest *cross-site scripting* (XSS). Polega ona na wykonaniu kodu HTML-a lub złośliwego kodu JavaScriptu w przeglądarce internetowej użytkownika.

Wyzwaniem dla firm jest możliwość zautomatyzowania testów bezpieczeństwa swoich aplikacji w celu ich ochrony i jak najszybszego podjęcia działań w przypadku wykrycia luki.

Istnieje wiele narzędzi do testowania bezpieczeństwa i penetracji. Kompletna lista jest dostępna na stronie https://www.owasp.org/index.php/Appendix_A:_Testing_Tools. Wśród nich znajduje się poznany w poprzednim rozdziale SonarQube, który umożliwia analizę kodu w celu wykrycia luk w zabezpieczeniach.

Kolejnym bardzo interesującym narzędziem z tej listy jest **Zed Attack Proxy (ZAP)** (https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project) opracowany przez społeczność OWASP.

Nauczmy się, jak używać ZAP-a do przeprowadzania testów bezpieczeństwa naszych aplikacji.

Korzystanie z ZAP-a w celu testowania bezpieczeństwa

ZAP to bezpłatne narzędzie graficzne o otwartym kodzie źródłowym, które umożliwia skanowanie stron internetowych i przeprowadzanie wielu testów bezpieczeństwa i penetracji.

W przeciwieństwie do SonarQube, który wprowadzie realizuje analizę bezpieczeństwa kodu źródłowego aplikacji, ale jej nie wykonuje, ZAP uruchamia aplikację i przeprowadza testy bezpieczeństwa.

Po uruchomieniu ZAP będzie działał jako proxy między użytkownikiem a aplikacją, skanując wszystkie adresy URL aplikacji, a następnie przeprowadzając na nich serię testów penetracyjnych. To dziś jedno z najczęściej używanych narzędzi do testowania aplikacji, ponieważ oprócz tego, że jest bezpłatne, udostępnia wiele bardzo ciekawych

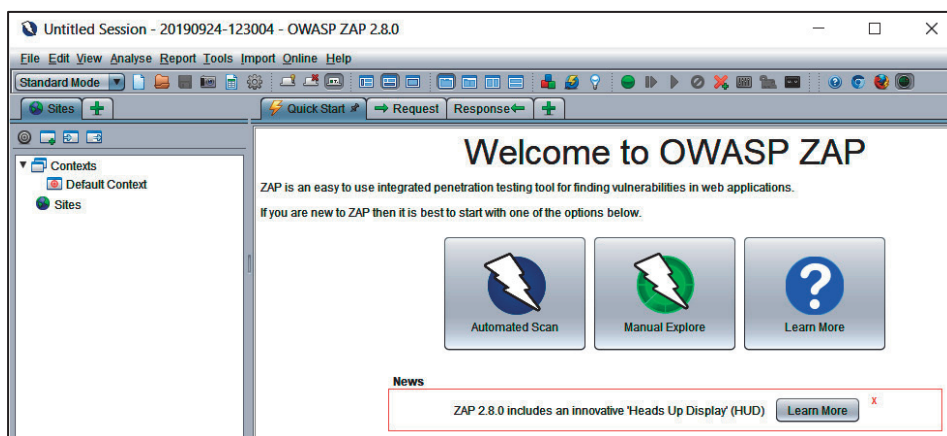
funkcji, takich jak możliwość konfiguracji testów penetracyjnych Ajax, a także zaawansowane konfiguracje testów. Ponadto bardzo dobrze integruje się z wieloma platformami potoków CI/CD. Wreszcie można nim sterować za pomocą REST API. Odpowiednia dokumentacja jest dostępna pod adresem <https://www.zaproxy.org/docs/api/>.

Proponuję zrobić małe laboratorium przy użyciu ZAP-a na publicznej stronie demonstracyjnej, która zawiera luki w zabezpieczeniach. Jak wspomnieliśmy w sekcji „Wymagania techniczne”, jednym z warunków korzystania z ZAP-a jest zainstalowanie Javy na maszynie, na której będą wykonywane testy. Może to być komputer lokalny lub agent kompilacji.

ZAP-a możemy pobrać ze strony <https://www.zaproxy.org/download/>; pobierz pakiet, który odpowiada Twojemu systemowi operacyjnemu.

Następnie zainstaluj ZAP-a, postępując zgodnie z procedurami instalacji oprogramowania w swoim systemie operacyjnym. Po zakończeniu instalacji możemy otworzyć go i uzyskać dostęp do jego interfejsu.

Poniższy zrzut ekranu pokazuje domyślny interfejs ZAP-a:



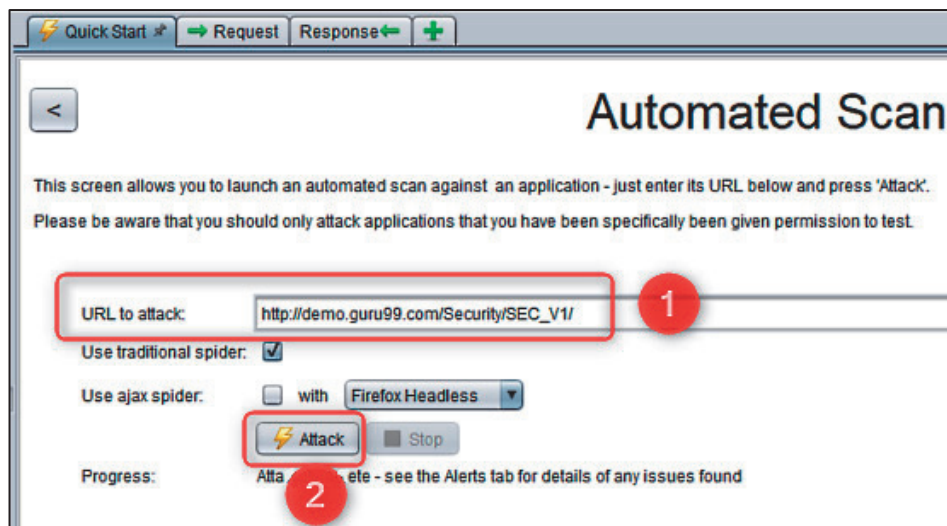
Rysunek 13.1. Narzędzie OWASP ZAP

Przeprowadzimy naszą pierwszą analizę bezpieczeństwa za pomocą ZAP-a, wykonując następujące kroki:

1. W prawym panelu kliknij przycisk *Automated Scan*, co spowoduje otwarcie formularza, gdzie możemy wpisać adres URL do przeskanowania.
2. W polu *URL to attack* wprowadź adres URL strony internetowej do analizy. W naszym przykładzie wprowadzimy adres URL strony demonstracyjnej: http://demo.guru99.com/Security/SEC_V1/.

3. Następnie, aby rozpocząć analizę, kliknij przycisk *Attack*.

Poniższy zrzut ekranu przedstawia w sposób wizualny poprzednie kroki:



Rysunek 13.2. OWASP ZAP — automatyczne skanowanie

Musimy poczekać na zakończenie analizy testów bezpieczeństwa witryny.

4. Zaraz po zakończeniu analizy w panelu w lewym dolnym rogu możemy zobaczyć, jakie wystąpiły problemy bezpieczeństwa.
5. Na koniec kliknięcie jednego z alertów wyświetla szczegóły problemu i informacje na temat jego rozwiązania.

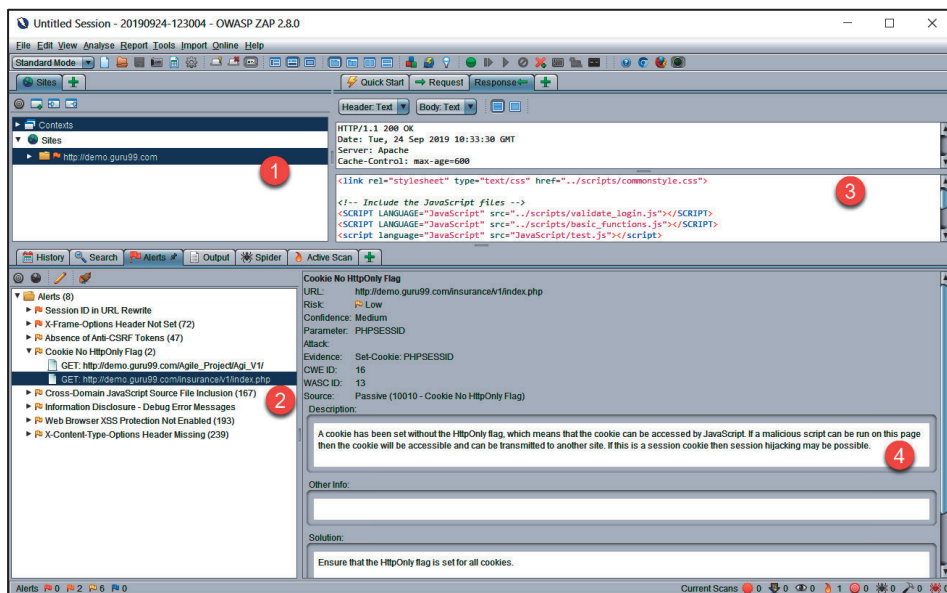
Rysunek 13.3 pokazuje wyniki analizy, o których właśnie wspomnieliśmy.

Nauczyliśmy się korzystać z ZAP-a, graficznego narzędzia służącego do bardzo szybkiej analizy luk w zabezpieczeniach witryny.

Przyjrzyjmy się teraz różnym sposobom automatyzacji wykonywania ZAP-a.

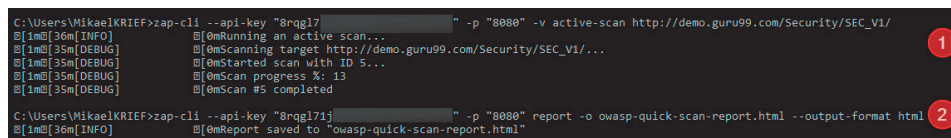
Sposoby automatyzacji wykonywania ZAP-a

Możemy również zautomatyzować tę analizę, instalując ZAP-a na serwerze agenta naszego potoku CI/CD, i skorzystać z narzędzia `zap-cli`, dostępnego pod adresem <https://github.com/Grunny/zap-cli>. Jest to narzędzie używane z linii poleceń i wywołuje interfejsy API ZAP.



Rysunek 13.3. Wynik skanowania OWASP ZAP

Poniższy zrzut ekranu pokazuje użycie `zap-cli` i wierszu poleceń do analizy naszej strony demonstracyjnej:

Rysunek 13.4. Wiersz poleceń dla `zap-cli`

W poprzednim wykonaniu używane są dwa polecenia:

- Pierwsze, `zap-cli active-scan`, analizuje stronę internetową, która została przekazana jako parametr polecenia.
- Drugie, `zap-cli report`, generuje raport wyników skanowania w formacie HTML.

Uwaga

W poprzednich poleceniach użyliśmy parametru `--api-key`. Aby pobrać wartość klucza API, przejdź do menu *Tools/Options/API* w Twojej instancji narzędzia ZAP.

Jeśli używamy Azure DevOps jako platformy potoku CI/CD, możemy skorzystać z zadania OWASP Zed Attack Proxy Scan w Visual Studio Marketplace, które jest dostępne pod adresem <https://marketplace.visualstudio.com/items?itemName=kasunkodagoda.owasp-zap-scan>. Jeśli mamy subskrypcję Azure, Azure Pipelines może również uruchamiać ZAP-a w kontenerze Dockera, hostowanym w instancji Azure Container, zgodnie z wyjaśnieniami i szczegółowymi informacjami na stronie <https://devblogs.microsoft.com/premier-developer/azure-devops-pipelines-leveraging-owasp-zap-in-the-release-pipeline/>.

Jeśli używamy Jenkinsa jako fabryki kompilacji, spójrz na następujący artykuł, który wyjaśnia, jak zintegrować i używać wtyczki ZAP-a podczas uruchamiania zadania: <https://www.breachlock.com/integrating-owasp-zap-in-devsecops-pipeline/>.

Właśnie nauczyliśmy się przeprowadzać testy bezpieczeństwa w naszych aplikacjach internetowych za pomocą ZAP-a, który jest rozwijany przez społeczność OWASP. Przyrzekliśmy się jego podstawowemu użyciu za pośrednictwem interfejsu graficznego i przeprowadziliśmy testy bezpieczeństwa na aplikacji demonstracyjnej. Następnie zobaczyliśmy, że możliwe jest również zautomatyzowanie jego wykonywania za pomocą `zap-cli`, abyśmy mogli zintegrować go z potokiem DevOps CI/CD.

Teraz nauczymy się przeprowadzać testy wydajności za pomocą Postmana.

Uruchamianie testów wydajności za pomocą Postmana

Wśród testów, które należy wykonać, aby zagwarantować jakość naszych aplikacji i zapewnić ich funkcjonalność, w tym analizę kodu i testy bezpieczeństwa, znajdują się również testy wydajnościowe. Celem testowania wydajności nie jest wykrywanie błędów w aplikacjach; ma na celu zapewnienie, że aplikacja (lub interfejs API) zareaguje w akceptowalnym przedziale czasowym, aby zapewnić dobre wrażenia dla użytkownika.

Wydajność aplikacji jest określana za pomocą metryk, takich jak:

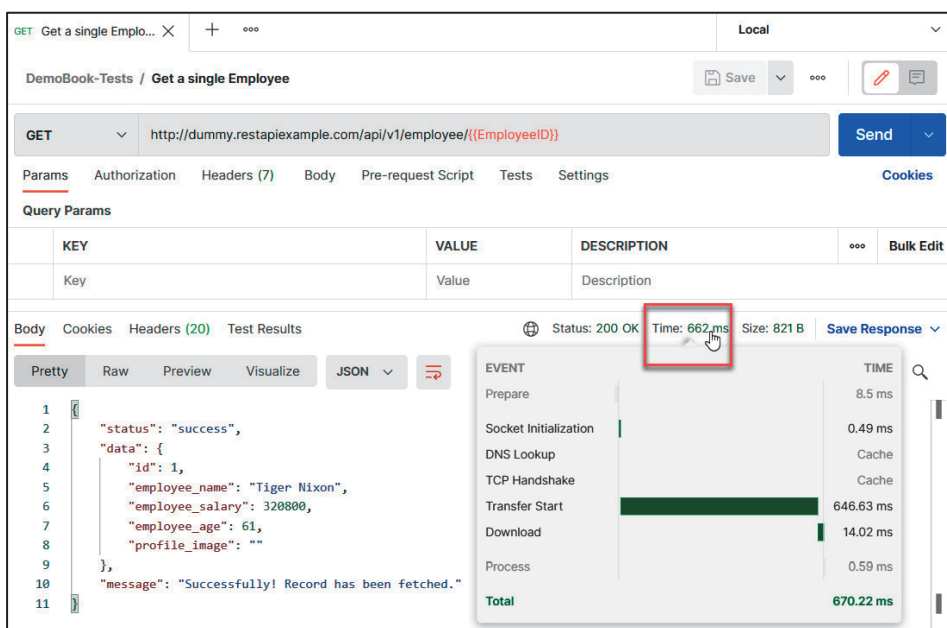
- czas reakcji,
- z jakich zasobów korzysta (procesor, pamięć RAM i sieć),
- wskaźniki błędów,
- liczba żądań na sekundę.

Testy wydajności dzielą się na kilka rodzajów, takich jak testy obciążenia, testy warunków skrajnych (ang. *stress test*) i testy skalowalności.

Dostępnych jest wiele narzędzi do przeprowadzania testów wydajności. Poniższy artykuł zawiera listę 15 najlepszych: <https://www.softwaretestinghelp.com/performance-testing-tools-load-testing-tools/>. Wśród narzędzi, które widzieliśmy już w tej książce, Postman nie jest dedykowany do testowania wydajności, zwłaszcza że koncentruje się głównie na interfejsach API, a nie na monolitycznych aplikacjach internetowych. Jednak Postman może dostarczyć wielu dobrych wskazówek na temat wydajności naszego interfejsu API.

Jego możliwości testowania API omówiliśmy szczegółowo w rozdziale 11., „Testowanie interfejsów API za pomocą Postmana”.

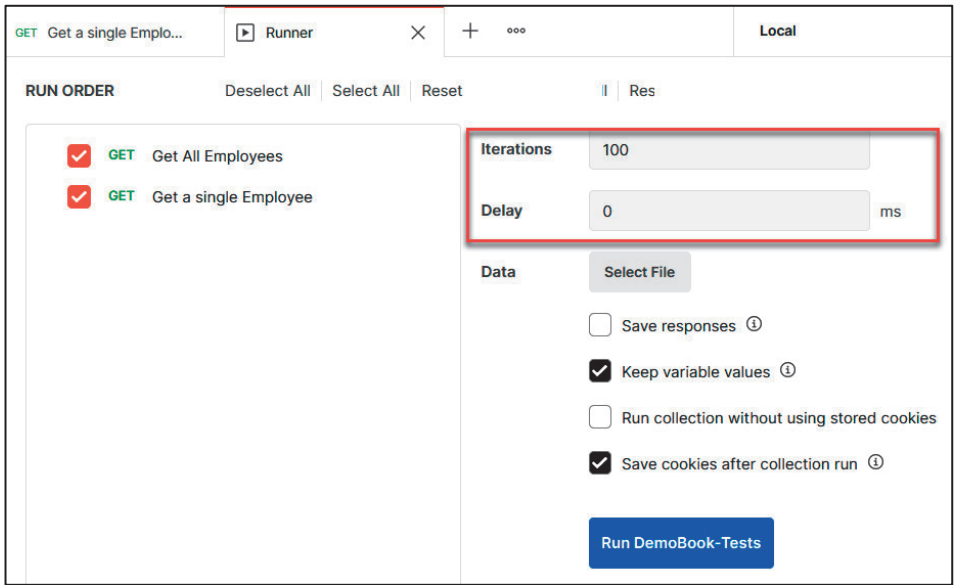
Gdy wykonujesz żądanie, które testuje interfejs API w sposób jednolity, Postman podaje czas wykonania tego API, jak pokazano na poniższym zrzucie ekranu:



Rysunek 13.5. Test wydajności Postmana

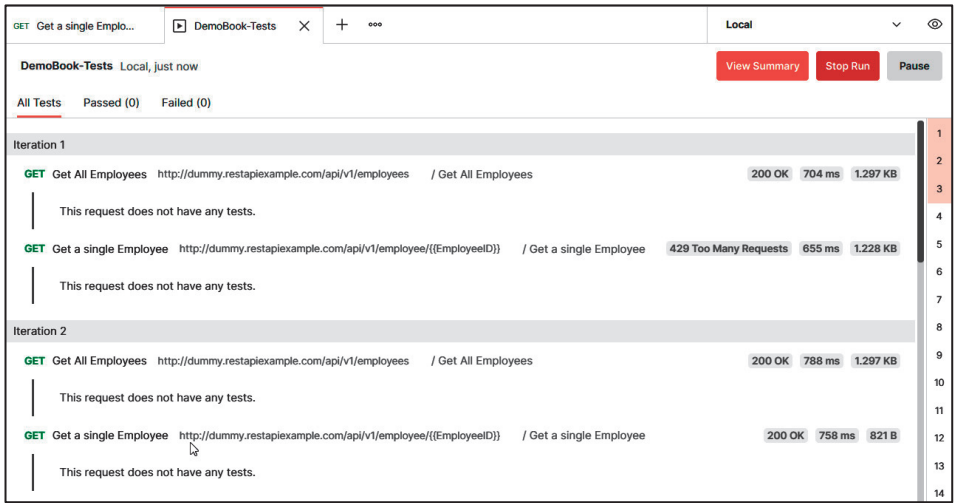
Ponadto za pomocą narzędzia **Collection Runner** dostępnego w Postmanie możliwe jest wykonanie wszystkich żądań kolekcji i wskazanie liczby iteracji, tzn. ile razy w pętli zostaną wykonane testy. Symulowane jest kilka połączeń, które wywołują interfejs API, przez co czas wykonania renderowany przez Postmana staje się bardzo interesujący.

Poniższy zrzut ekranu przedstawia konfigurację **Runnera** z kilkoma iteracjami parametrów wejściowych:



Rysunek 13.6. Test konfiguracji Runnera

Poniższy zrzut ekranu przedstawia wyniki Runnera:



Rysunek 13.7. Wyniki testu Runnera

Tutaj widzimy, że Runner wyświetla czas wykonania każdego żądania, co oznacza, że możemy zidentyfikować problemy z przeciążeniem w interfejsie API.

Teraz, gdy nauczyliśmy się przeprowadzać testy wydajnościowe w Postmanie, podsumujmy ten rozdział.

Podsumowanie

W tym rozdziale widzieliśmy, jak używać ZAP-a, narzędzia opracowanego przez społeczność OWASP w celu automatyzacji wykonywania testów bezpieczeństwa aplikacji internetowych. Zobaczyliśmy również, jak Postman dostarcza informacje o wydajności API.

W następnym rozdziale będziemy dalej rozmawiać o bezpieczeństwie i DevSecOps, ucząc się, jak automatyzować testowanie infrastruktury za pomocą InSpec, jak chronić tajemnice za pomocą HashiCorp Vault i jak używać zestawu Secure DevOps Kit dla platformy Azure do sprawdzania bezpieczeństwa infrastruktury Azure.

Pytania

Odpowiedz na poniższe pytania, aby sprawdzić swoją wiedzę na temat zagadnień omawianych w tym rozdziale:

1. Czy ZAP jest narzędziem analizującym kod źródłowy aplikacji?
2. Która metryka Postmana pozwala nam uzyskać informacje o wydajności?

Dalsza lektura

Aby dowiedzieć się więcej na temat tego, co zostało omówione w tym rozdziale, zapoznaj się z następującymi zasobami:

- *Learn Penetration Testing* — <https://www.packtpub.com/networking-and-servers/learn-penetration-testing>.
- Filmy Pluralsight na temat społeczności OWASP i narzędzia ZAP — <https://www.pluralsight.com/search?q=owasp>.

Więcej informacji na temat DevOps

Ta część wyjaśnia zaawansowane tematy związane z procesami DevOps dotyczące integracji bezpieczeństwa w DevOps (DevSecOps), niektóre techniki dotyczące metody wdrażania niebiesko-zielonego i zastosowanie DevOps w projekcie open source.

Składa się z następujących rozdziałów:

- Rozdział 14., „Bezpieczeństwo w procesie DevOps z wykorzystaniem DevSecOps”.
- Rozdział 15., „Skrócenie czasu przestoju wdrażania”.
- Rozdział 16., „DevOps dla projektów open source”.
- Rozdział 17., „Najlepsze praktyki DevOps”.



Bezpieczeństwo w procesie DevOps z wykorzystaniem DevSecOps

Rozdział

14

Do tej pory w tej książce szczegółowo omówiliśmy kulturę **DevOps** (ang. *development-operations*), a także narzędzia DevOps, które ułatwiają komunikację i współpracę między programistami a pracownikami operacyjnymi (ang. *information technology-operations* lub *ITOps*).

Jednak omawiając te zagadnienia, zauważyliśmy, że często brakuje bardzo ważnego aspektu, jakim jest bezpieczeństwo. Rzeczywiście, potoki **ciągłej integracji/ciągłego wdrażania (CI/CD)** i **praktyka infrastruktura jako kod (IaC)** umożliwiają szybsze wdrażanie infrastruktury i aplikacji, ale problem polega na tym, że aby wdrażać szybciej, nie uwzględniamy zespołów ds. bezpieczeństwa, co powoduje, że:

- Zespoły ds. bezpieczeństwa blokują lub spowalniają wdrażanie, co prowadzi do dłuższych cykli wdrażania.
- Problemy bezpieczeństwa są wykrywane bardzo późno w infrastrukturze i aplikacjach.

Dlatego od pewnego czasu bezpieczeństwo zostało włączone do kultury DevOps, stając się szerzej kulturą **DevSecOps** (ang. *development-security-operations*). Nie zawiera ona nic poza obszarem bezpieczeństwa. Ponieważ rozwijamy się w szybkim tempie, rozsądne jest uczynienie bezpieczeństwa częścią procesu, a nie rozszerzanie tego procesu.

Kultura lub podejście DevSecOps jest zatem połączeniem programistów i operacji z integracją bezpieczeństwa na jak najwcześniejszym etapie wdrażania i projektowania. Podejście DevSecOps to także automatyzacja procesów, weryfikacji zgodności i bezpieczeństwa w potokach CI/CD, by zagwarantować stałe bezpieczeństwo i nie spowalniać cykli wdrażania aplikacji.

Dziś kultura DevOps musi bezwzględnie integrować zespoły bezpieczeństwa, ale także wszystkie procesy bezpieczeństwa czy to w narzędziach, czy w infrastrukturze, czy w aplikacjach. Ma to na celu zapewnienie nie tylko lepszej jakości, lecz także bezpieczniejszych aplikacji.

W tym rozdziale skupimy się na podejściu DevSecOps. Najpierw zobaczymy, jak przetestować infrastrukturę platformy Azure za pomocą narzędzia InSpec firmy Chef. Następnie dowiemy się, jak chronić wszystkie tajemnice infrastruktury i aplikacji za pomocą narzędzia Vault firmy HashiCorp.

W tym rozdziale poruszone zostaną następujące tematy:

- testowanie infrastruktury Azure za pomocą InSpec,
- ochrona poufnych danych dzięki Vault od HashiCorp.

Wymagania techniczne

W tym rozdziale zobaczymy wykorzystanie InSpec, który wymaga zainstalowania **Ruby** w wersji 2.4 lub nowszej na lokalnej maszynie. Aby zainstalować Ruby zgodnie ze swoim **systemem operacyjnym (OS)**, przeczytaj tę dokumentację: <https://www.ruby-lang.org/en/documentation/installation/>.

W sekcji „Ochrona poufnych danych dzięki Vault od HashiCorp” omówimy integrację Vault i Terraform bez zagłębiania się w szczegóły Terraform, dlatego sugeruję najpierw przeczytanie rozdziału 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”.

Pełny kod źródłowy tego rozdziału jest dostępny tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP14>.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/3In9kb2>.

Testowanie infrastruktury Azure za pomocą InSpec

Jedną z ważnych praktyk kultury DevOps jest IaC, szczegółowo opisana w rozdziale 1., „Kultura DevOps i praktyki kodowania infrastruktury”, która polega na kodowaniu konfiguracji infrastruktury i jej automatycznym wdrażaniu za pośrednictwem potoków CI/CD. IaC umożliwia bardzo szybkie wdrażanie i udostępnianie infrastruktury

chmurowej. Jednak często pojawia się pytanie: *Czy automatycznie dostarczana infrastruktura spełnia wymogi zgodności funkcjonalnej i bezpieczeństwa?*

Aby odpowiedzieć na to pytanie, będziemy musieli napisać i zautomatyzować testy infrastruktury, które zweryfikują następujące elementy:

- Wdrożona infrastruktura dobrze odpowiada specyfikacji aplikacji i architekturze korporacyjnej.
- Polityka bezpieczeństwa firmy jest właściwie stosowana w infrastrukturze.

Testy te można napisać w dowolnym języku skryptowym, który współpracuje z naszym dostawcą chmury, a jeśli mamy subskrypcję platformy Azure, możemy skorzystać np. z **interfejsu wiersza polecenia platformy Azure (CLI)** lub z poleceń programu Azure PowerShell, aby zakodować testy naszych zasobów platformy Azure. Ponadto, jeśli używamy PowerShell, możemy użyć biblioteki Pester (<https://pester.dev/docs/quick-start/#what-is-pester>), która pozwala nam przeprowadzać testy PowerShell i w połączeniu z Azure PowerShell pozwala nam przeprowadzać testy zgodności infrastruktury.

Uwaga

Aby uzyskać przykład wykorzystania biblioteki Pester do testowania infrastruktury Azure, sugeruję przeczytanie tego artykułu: <https://dzone.com/articles/azure-security-audits-with-pester>. Zajrzyj również do tego wpisu na blogu: <https://dev.to/omiossec/unit-testing-in-powershell-introduction-to-pester-1de7>.

Problem z tymi narzędziami skryptowymi polega na tym, że wymagają one napisania dużej ilości kodu. Ponadto narzędzia te są dedykowane dla konkretnego dostawcy chmury i wymagają nauki nowego języka skryptowego.

Jednym z narzędzi IaC jest **InSpec** (<https://www.inspec.io/>), które wykonuje testy zgodności infrastruktury.

W tej sekcji szczegółowo przyjrzymy się wykorzystaniu InSpec do testowania zgodności infrastruktury Azure.

Omówienie InSpec

InSpec to otwartoźródłowe narzędzie napisane w języku Ruby, które działa w wierszu poleceń i jest tworzone przez jednego z głównych producentów narzędzi DevOps, **Chef**, którego strona internetowa to <https://www.chef.io/>. Umożliwia użytkownikom pisanie kodu w stylu deklaratywnym służącego do testowania zgodności systemu lub infrastruktury.

Aby korzystać z InSpec, nie trzeba uczyć się nowego języka skryptowego; powinniśmy mieć już wystarczającą wiedzę, aby utworzyć pożądany stan zasobów infrastruktury lub systemu, który chcemy przetestować.

Dzięki InSpec możemy testować zgodność zdalnych maszyn i danych, a od najnowszej wersji możliwe jest również testowanie infrastruktury chmurowej, takiej jak Azure, **Amazon Web Services (AWS)** i **Google Cloud Platform (GCP)**.

Po tym krótkim omówieniu programu InSpec przyjrzymy się, jak go pobrać i zainstalować.

Instalacja InSpec

W sekcji „Wymagania techniczne” dowiedzieliśmy się, że InSpec musi mieć **Ruby** (>2.4) zainstalowane na naszej maszynie.

InSpec można zainstalować ręcznie lub za pomocą skryptu, jak opisano tutaj:

- **Ręcznie.** Można to zrobić, pobierając pakiet odpowiadający naszemu systemowi operacyjnemu z <https://www.chef.io/downloads/tools/inspec>.
- **Za pomocą skryptu.** InSpec możemy zainstalować, wykonując w terminalu podane niżej polecenia.

W systemie Windows możemy pobrać i zainstalować InSpec za pomocą pakietu Chocolatey, dostępnego pod adresem <https://community.chocolatey.org/packages/inspec>:

```
choco install inspec -y
```

W systemie Linux użyj następującego skryptu:

```
curl https://omnitruck.chef.io/install.sh | sudo bash -s -- -P inspec
```

Rysunek 14.1 pokazuje instalację InSpec za pośrednictwem **pakietu Chocolatey**.

Uwaga

Aby uzyskać więcej informacji na temat instalacji InSpec dla wszystkich systemów operacyjnych, zapoznaj się z tą dokumentacją: <https://docs.chef.io/inspec/install/>.

Aby sprawdzić, czy InSpec został poprawnie zainstalowany i działa, uruchamiamy polecenie `inspec --version`, aby wyświetlić jego wersję, oraz polecenie `inspec`, aby wyświetlić listę dostępnych poleceń.

Rysunek 14.2 pokazuje wykonanie tych poleceń.


```
PS C:\WINDOWS\system32> choco install inspec
Chocolatey v0.10.15
Installing the following packages:
inspec
By installing you accept licenses for the packages.
Error retrieving packages from source 'http://srv-rd-packages.talentsoft.com/nuget/TalentsoftChoco':
Le nom distant n'a pas pu être résolu: 'srv-rd-packages.talentsoft.com'
Progress: Downloading inspec 4.46.13... 100%

inspec v4.46.13 [Approved]
inspec package files install completed. Performing other installation steps.
The package inspec wants to run 'chocolateyinstall.ps1'.
Note: If you don't run this script, the installation will fail.
Note: To confirm automatically next time, use '-y' or consider:
choco feature enable -n allowGlobalConfirmation
Do you want to run the script?([Y]es/[A]ll - yes to all/[N]o/[P]rint): A

Downloading inspec 64 bit
from 'https://packages.chef.io/files/stable/inspec/4.46.13/windows/2016/inspec-4.46.13-1-x64.msi'
Progress: 100% - Completed download of C:\Users\mkrief\AppData\Local\Temp\chocolatey\inspec\4.46.13\inspec-4.46.13-1-x64.msi (127.29 MB).
Download of inspec-4.46.13-1-x64.msi (127.29 MB) completed.
Hashes match.
Installing inspec...
inspec has been installed.
inspec may be able to be automatically uninstalled.
Environment Vars (like PATH) have changed. Close/reopen your shell to
see the changes (or in powershell/cmd.exe just type 'refreshenv').
The install of inspec was successful.
Software installed as 'msi', install location is likely default.

Chocolatey installed 1/1 packages.
See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).
```

Rysunek 14.1. Instalacja InSpec w systemie Windows

```
PS C:\Users\mkrief> inspec --version
4.46.13
PS C:\Users\mkrief> inspec
Commands:
  inspec archive PATH                # Archive a profile to...
  inspec artifact SUBCOMMAND        # Manage Chef InSpec A...
  inspec automate SUBCOMMAND or compliance SUBCOMMAND # Chef Automate commands
  inspec check PATH                 # verify all tests at ...
  inspec clear-cache                # clears the InSpec ca...
  inspec detect                     # detect the target OS
  inspec env                         # Output shell-appropri...
  inspec exec LOCATIONS             # Run all tests at LOC...
  inspec habitat SUBCOMMAND         # Manage Habitat with ...
  inspec help [COMMAND]             # Describe available c...
  inspec init SUBCOMMAND            # Generate InSpec code
  inspec json PATH                  # read all tests in PA...
  inspec plugin SUBCOMMAND          # Manage Chef InSpec a...
  inspec shell                       # open an interactive ...
  inspec supermarket SUBCOMMAND ... # Supermarket commands
  inspec vendor PATH                # Download all depende...
  inspec version                    # prints the version o...

Options:
  -l, [--log-level=LOG_LEVEL]      # Set the log level: info (default), debug, warn, error
  --log-locations=LOG_LOCATION    # Location to send diagnostic log messages to. (default: $stdout or InSpec::Log.error)
  --diagnose, [--no-diagnose]     # Show diagnostics (versions, configurations)
  --color, [--no-color]           # Use colors in output.
  --interactive, [--no-interactive] # Allow or disable user interaction
  --disable-user-plugging          # Disable loading all plugins that the user installed.
  --enable-telemetry, [--no-enable-telemetry] # Allow or disable telemetry
  --chef-license=CHEF_LICENSE     # Accept the license for this product and any contained products: accept, accept-no-persist, accept-sil
ent
```

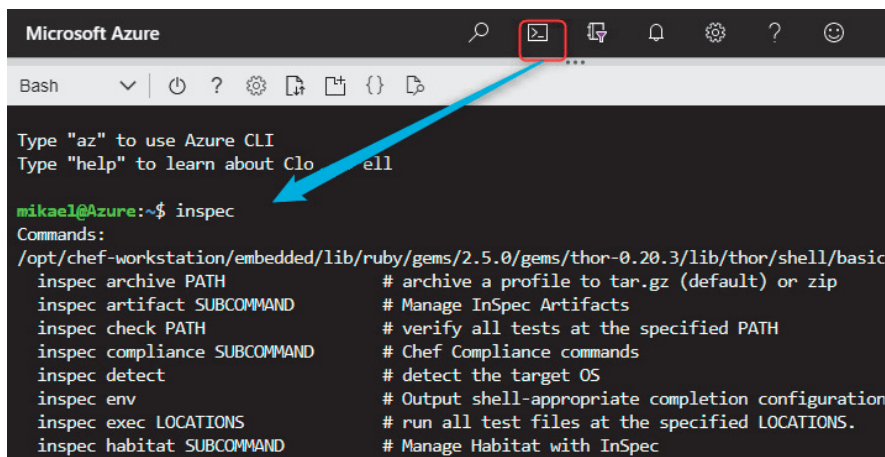
Rysunek 14.2. Sprawdzanie wersji i wyświetlanie opcji InSpec

Ponadto, jak w przypadku wielu narzędzi opisanych już w tej książce, InSpec został zintegrowany z pakietem narzędzi **Azure Cloud Shell**, jak pokazano na rysunku 14.3.

Właśnie poznaliśmy różne sposoby instalacji InSpec, a teraz zobaczymy konfigurację platformy Azure dla tego narzędzia.

Konfigurowanie platformy Azure dla InSpec

Przed przetestowaniem naszej infrastruktury platformy Azure musimy utworzyć jednostkę usługi platformy Azure mającą uprawnienia do odczytu zasobów platformy Azure, które będą testowane.



```

Microsoft Azure
Bash
Type "az" to use Azure CLI
Type "help" to learn about Cloud Shell

mikael@Azure:~$ inspec

Commands:
/opt/chef-workstation/embedded/lib/ruby/gems/2.5.0/gems/thor-0.20.3/lib/thor/shell/basic
inspec archive PATH                # archive a profile to tar.gz (default) or zip
inspec artifact SUBCOMMAND         # Manage InSpec Artifacts
inspec check PATH                  # verify all tests at the specified PATH
inspec compliance SUBCOMMAND       # Chef Compliance commands
inspec detect                      # detect the target OS
inspec env                         # Output shell-appropriate completion configuration
inspec exec LOCATIONS              # run all test files at the specified LOCATIONS.
inspec habitat SUBCOMMAND          # Manage Habitat with InSpec

```

Rysunek 14.3. InSpec w Azure Cloud Shell

Aby utworzyć jednostkę usługi platformy Azure, użyjemy tej samej procedury, która została już szczegółowo opisana w sekcji „Konfigurowanie Terraform dla platformy Azure” w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”.

Za pomocą narzędzia **Azure CLI** wykonujemy następującą komendę `az cli`:

```
az ad sp create-for-rbac -name="<nazwa SP> -role="Reader" -
scopes="/subscriptions/<identyfikator subskrypcji>"
```

Polecenie to wymaga następujących parametrów:

- `--name` to nazwa jednostki usługi platformy Azure, która ma zostać utworzona.
- `--scopes` to **identyfikator (ID) subskrypcji** (lub innych zakresów scope), w których będą obecne zasoby platformy Azure.
- `--role` to nazwa roli, którą jednostka usługi będzie miała w określonym zakresie zasobów.

Wykonanie tego poprzedniego polecenia zwraca następujące trzy informacje uwierzytelniające dotyczące utworzonej nazwy usługi:

- identyfikator klienta,
- tajny klucz klienta,
- identyfikator dzierżawcy.

Uwaga

Aby uzyskać więcej informacji na temat jednostek usługi, przeczytaj następującą dokumentację: <https://docs.microsoft.com/en-us/cli/azure/create-an-azure-service-principal-azure-cli?view=azure-cli-latest>.

Zobaczmy, jak korzystać z tych informacji uwierzytelniających podczas uruchamiania InSpec. Jednak przed jego wykonaniem potrzebujemy wystarczającej wiedzy, aby utworzyć testy InSpec. Zobaczmy, jak to zrobić.

Tworzenie testów InSpec

Po zainstalowaniu InSpec i skonfigurowaniu uwierzytelniania dla Azure możemy zacząć korzystać z InSpec. Utworzymy przykładowe testy, które sprawdzą, czy infrastruktura Azure udostępniona za pomocą Terraform w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”, jest zgodna ze specyfikacjami naszej infrastruktury Azure, która musi się składać z następujących elementów:

- Jedna grupa zasobów o nazwie bookRg.
- Jedna **sieć wirtualna (VNet)** z jedną podsiecią wewnątrz niej o nazwie book-subnet.
- Jedna maszyna wirtualna (VM) o nazwie bookvm.

Pierwszym krokiem w tworzeniu naszych testów InSpec jest utworzenie pliku profilu.

Tworzenie pliku profilu InSpec

Aby utworzyć plik profilu InSpec, wygenerujemy testową strukturę katalogów, a następnie zmodyfikujemy wygenerowany plik profilu.

Aby wykonać te kroki, tworzymy strukturę folderów testowych i profil InSpec, który definiuje niektóre metadane i konfigurację InSpec. Aby utworzyć strukturę i plik profilu InSpec, przejdź na swoim komputerze do wybranego katalogu i wykonaj następujące polecenie:

```
inspec init profile azuretests
```

Polecenie to inicjuje nowy profil, tworząc nowy folder, azuretests, który zawiera wszystkie artefakty potrzebne do testów InSpec:

- kontrole (testy),
- biblioteki,
- plik profilu, *inspec.yml*, z pewnymi domyślnymi metadanymi.

Następnie modyfikujemy plik profilu *inspec.yml* za pomocą pewnych osobistych metadanych i dodajemy łącze **Uniform Resource Locator (URL)** z pakietu zasobów InSpec-Azure do przykładowego kodu w następujący sposób:

```
name: azuretests
title: InSpec Profile
maintainer: Twoje imię i nazwisko
```

```
copyright: Twoje imię i nazwisko
copyright_email: you@example.com
license: All Rights Reserved
summary: An InSpec Compliance Profile
version: 0.1.0
inspec_version: '>= 4.6.9'
depends:
  - name: inspec-azure
    url: https://github.com/inspec/inspec-azure.git
```

W tym kodzie wprowadziliśmy pewne dane osobowe, takie jak nasze imię i nazwisko, oraz informacje o rodzaju licencji tego kodu. Następnie w ostatniej części tego pliku wskazaliśmy zależność adresu URL pakietu bibliotek zasobów InSpec-Azure.

W rzeczywistości od wersji 2.2.7 w InSpec możemy korzystać z zestawu bibliotek, które korzystają z Azure API, a tym samym umożliwiają nam dostęp do wszystkich zasobów Azure. Na podstawie tego zespół InSpec tworzy pakiet zasobów Azure, który zawiera wiele bibliotek do testowania szerokiej gamy zasobów Azure, m.in. takich jak użytkownicy Azure, Azure Monitor, Azure Networking (VNet i podsieci), Azure SQL Server, **Azure Virtual Machines (maszyny wirtualne Azure)**.

Uwaga

Aby uzyskać pełną listę dostępnych zasobów platformy Azure, zapoznaj się z dokumentacją InSpec dotyczącą pakietu zasobów InSpec dla platformy Azure: <https://www.inspec.io/docs/reference/resources/#azure-resources>.

Po wygenerowaniu naszych katalogów, które zawierają testy i aktualizacje plików profili, napiszemy nasze testy zgodności infrastruktury.

Tworzenie testów zgodności InSpec

Nawiązując do naszej przykładowej specyfikacji, utworzymy testy, które sprawdzą, czy nasza udostępniona infrastruktura zawiera grupę zasobów, maszynę wirtualną i podsieć.

Wszystkie testy, które będziemy tworzyć, znajdują się w folderze *controls* i są zapisane w języku Ruby (.rb) z bardzo prostym i czytelnym kodem.

Na początek napiszemy test sprawdzający istnienie grupy zasobów. W tym celu usuniemy z folderu *controls* plik *example.rb*, który jest przykładem testów dostarczonych w szablonach testów, i utworzymy nowy plik o nazwie *resourcegroup.rb*, który zawiera następującą treść kodu:

```
control 'rg' do
  describe azure_resource_groups do #odwołanie do biblioteki
                                     #azurerm_resource_groups
    its('names') { should include 'bookRg' } #asercja
  end
end
```

W tym deklaratywnym kodzie opisano pożądany stan zasobów i podjęto następujące działania:

1. Utworzenie kontrolki (lub testu) o nazwie `test_rg`.
2. W tej kontrolce utworzymy metodę typu `describe`, w której użyjemy biblioteki pakietów zasobów Azure `azure_resource_groups`, co pozwoli nam przetestować istnienie grupy zasobów.
3. W tej metodzie `describe` utworzymy testowe asercje, które sprawdzają, czy w subskrypcji Azure istnieje grupa zasobów `bookRg`.

Następnie będziemy kontynuować pisanie naszych testów dla maszyny wirtualnej i pod-sieci. W tym celu ręcznie tworzymy w katalogu `controls` plik `subnet.rb`, który zawiera następujący kod:

```
control "subnet" do
  describe azure_subnet(resource_group: 'bookRg', vnet: 'bookvnet',
    ↪name: 'book-subnet')
  do
    it { should exist }
    its('address_prefix') { should eq '10.0.10.0/24' }
  end
end
```

W tym kodzie korzystamy z biblioteki `azurerm_subnet`, która pozwala nam przetestować istnienie podsieci w sieci wirtualnej. W tym teście sprawdzamy, czy podsieć `book-↪subnet` istnieje w sieci wirtualnej `book-vnet` i czy ma zakres adresów IP `10.0.10.0/24`.

Na koniec tworzymy testy, które pozwalają nam sprawdzić naszą maszynę wirtualną — za pomocą następującego kodu w pliku `vm.rb`:

```
control 'vm' do
  describe azure_virtual_machine(resource_group: 'bookRg', name:
    'bookvm')
  do
    it { should exist }
    its('properties.location') { should eq 'westeurope' }
    its('properties.hardwareProfile.vmSize') { should eq
      'Standard_DS1_v2' }
    its('properties.storageProfile.osDisk.osType') { should eq
      'Linux' }
  end
end
```

```
}  
end  
end
```

W tym kodzie korzystamy z biblioteki `azurerms_virtual_machine` i sprawdzamy, czy maszyna wirtualna o nazwie `demovm` istnieje w grupie zasobów `bookRg`. Sprawdzamy również niektóre właściwości, takie jak region, typ systemu operacyjnego i rozmiar maszyny wirtualnej.

Zakończyliśmy pisanie testów InSpec, które będą używane do sprawdzania zgodności naszej infrastruktury Azure, a teraz zobaczymy wykonanie InSpec z tymi testami, które właśnie utworzyliśmy.

Wykonywanie InSpec

Aby wykonać InSpec, wykonamy następujące kroki:

1. Skonfigurujemy uwierzytelnianie InSpec na platformie Azure; w tym celu utworzymy zmienne środowiskowe z wartościami informacji o nazwie głównej usługi platformy Azure, które utworzyliśmy wcześniej w sekcji „Konfigurowanie platformy Azure dla InSpec”. Cztery zmienne środowiskowe i ich wartości są wymienione tutaj:
 - `AZURE_CLIENT_ID` z identyfikatorem klienta jednostki usługi.
 - `AZURE_CLIENT_SECRET` z tajnym kluczem klienta jednostki usługi.
 - `AZURE_TENANT_ID` z identyfikatorem dzierżawcy.
 - `AZURE_SUBSCRIPTION_ID` z identyfikatorem subskrypcji, która zawiera zasoby i której jednostka usługi ma uprawnienia do odczytu.

Oto przykład tworzenia tych zmiennych w systemie operacyjnym Linux:

```
export AZURE_SUBSCRIPTION_ID="<identyfikator subskrypcji>  
export AZURE_CLIENT_ID="<identyfikator klienta>  
export AZURE_CLIENT_SECRET="<tajny klucz klienta>  
export AZURE_TENANT_ID="<identyfikator dzierżawcy>"
```

2. Następnie w terminalu przejdziemy do katalogu zawierającego plik profilu, *inspec.yml*, i uruchomimy polecenie `inspec vendor`, aby pobrać wszystkie zależności i wygenerować plik blokady w katalogu *vendor*.
3. Teraz wykonaj następującą komendę `inspec`, aby sprawdzić, czy składnia testów jest poprawna:

```
inspec check .
```

Argumentem, jaki należy podać temu poleceniu, jest ścieżka do katalogu zawierającego plik *inspec.yml*. W tym poleceniu używamy `.` (kropki),

aby wskazać, że plik *inspec.yml* znajduje się w bieżącym katalogu. Poniższy zrzut ekranu pokazuje wynik wykonania tego polecenia:

```
PS > .\Learning-DevOps-Second-Edition\CHAP14\azuretests> inspec check
Location : .
Profile : azuretests
Controls : 3
Timestamp : 2021-12-26T16:21:45+01:00
Valid : true

No errors or warnings
```

Rysunek 14.4. Sprawdzanie profilu InSpec

4. Na koniec uruchamiamy InSpec, aby wykonać testy za pomocą polecenia *inspec exec* w następujący sposób:

```
inspec exec . -t azure://
```

Polecenie to przyjmuje jako argument ścieżkę do katalogu zawierającego plik *inspec.yml* (tutaj jest to kropka). Dodajemy również opcję *-t*, która przenosi wartość testów do celu, czyli do platformy Azure.

Poniższy zrzut ekranu pokazuje wynik wykonania tego polecenia:

```
PS > .\Learning-DevOps-Second-Edition\CHAP14\azuretests> inspec exec . -t azure://

Profile: InSpec Profile (azuretests)
Version: 0.1.0
Target: azure://8a7aac5e-74aa-416f-b8e4-2c292b6384e5

[PASS] rg: InspectTest RG
[PASS] Azure Resource Groups - api_version: 2021-04-01 latest: /resourcegroups/ /resourcegroups/ names is expected to include "booklog"
[PASS] subnet: InspectTest Subnet
[PASS] Azure Subnet - api_version: 2021-06-01 latest: booklog Microsoft.Network/virtualNetworks book-subnet is expected to exist
[PASS] Azure Subnet - api_version: 2021-06-01 latest: booklog Microsoft.Network/virtualNetworks book-subnet address_prefix is expected to eq "10.0.10.0/24"
[PASS] vm: InspectTest VM
[PASS] Azure Virtual Machine - api_version: 2021-11-01 latest: booklog Microsoft.Compute/virtualMachines bookvm is expected to exist
[PASS] Azure Virtual Machine - api_version: 2021-11-01 latest: booklog Microsoft.Compute/virtualMachines bookvm location is expected to eq "westeurope" [PASS] Azure Virtual Machi
me - api_version: 2021-11-01 latest: booklog Microsoft.Compute/virtualMachines bookvm properties.hardwareProfile.vhSize is expected to eq "Standard_DS1_v2"
[PASS] Azure Virtual Machine - api_version: 2021-11-01 latest: booklog Microsoft.Compute/virtualMachines bookvm properties.storageProfile.osDisk.osType is expected to eq "Linux"

Profile: Azure Resource Pack (inspec-azure)
Version: 1.94.1
Target: azure://8a7aac5e-74aa-416f-b8e4-2c292b6384e5

No tests executed.

Profile Summary: 3 successful controls, 0 control failures, 0 controls skipped
Test Summary: 7 successful, 0 failures, 0 skipped
```

Rysunek 14.5. Wykonanie testów za pomocą InSpec

Na podstawie tego wyniku widzimy, że wszystkie testy są udane, więc zgodność infrastruktury jest zachowana.

W tej sekcji wyjaśniliśmy, jak tworzyć testy InSpec i uruchamiać je, by zweryfikować zgodność naszej infrastruktury Azure.

InSpec to bardzo potężne narzędzie; umożliwia również przetestowanie konfiguracji maszyny wirtualnej. Dlatego zachęcam do zapoznania się z poniższą dokumentacją: <https://www.inspec.io/docs/>.

Właśnie zapoznaliśmy się z instalacją InSpec, następnie z tworzeniem testów InSpec, a na koniec dowiedzieliśmy się, jak korzystać z tego narzędzia za pomocą wiersza poleceń do testowania zgodności infrastruktury Azure.

W następnej sekcji przyjrzymy się innemu aspektowi bezpieczeństwa, czyli ochronie wrażliwych danych za pomocą menedżera Vault od firmy HashiCorp, służącego do zarządzania hasłami.

Ochrona poufnych danych dzięki Vault od HashiCorp

Dziś, gdy mówimy o bezpieczeństwie w systemach informatycznych, najbardziej wyciekającym tematem jest ochrona wrażliwych danych przekazywanych pomiędzy różnymi komponentami systemu. Te wrażliwe dane, które należy chronić, obejmują hasła dostępu do serwera, połączenia z bazą danych, tokeny uwierzytelniania **interfejsu programowania aplikacji (API)** i konta użytkowników aplikacji. Rzeczywiście, wiele ataków ma miejsce, ponieważ ten typ danych jest odszyfrowywany w kodzie źródłowym aplikacji lub w słabo chronionych plikach, które są udostępniane lokalnym stacjom roboczym. Do zabezpieczenia tych poufnych danych można użyć wielu znanych narzędzi, takich jak:

- KeyPass (<https://keepass.info/>).
- LastPass (<https://www.lastpass.com/>).
- Ansible Vault, którego wykorzystanie omówiliśmy w rozdziale 3., „Używanie Ansible do konfigurowania infrastruktury IaaS”.
- Vault firmy HashiCorp.

Ponadto dostawcy usług w chmurze oferują własne usługi ochrony poufnych danych, takie jak:

- Azure Key Vault — <https://azure.microsoft.com/en-us/services/key-vault/>.
- **Usługa zarządzania kluczami** (ang. *key management service* — **KMS**) dla Google Cloud Platform — <https://cloud.google.com/kms/>.
- Menedżer haseł AWS (ang. *AWS Secrets Manager*) — https://aws.amazon.com/secrets-manager/?nc1=h_ls.

Spośród wszystkich narzędzi, o których wspomnieliśmy, przyjrzymy się wykorzystaniu Vault firmy HashiCorp, który jest bezpłatny i otwartoźródłowy. Może być zainstalowany na dowolnym typie systemu operacyjnego, a także na Kubernetesie.

Oto główne cechy i zalety Vault:

- Pozwala na przechowywanie haseł zarówno statycznych, jak i dynamicznych.
- Zawiera system rotacji i unieważniania haseł.

- Umożliwia szyfrowanie i odszyfrowywanie danych bez konieczności ich przechowywania.
- Ma interfejs webowy, który umożliwia zarządzanie hasłami.
- Integruje się z wieloma systemami uwierzytelniania.
- Wszystkie hasła są przechowywane w jednym scentralizowanym narzędziu.
- Pozwala uniezależnić się od swojej architektury, będąc dostępnym u wszystkich głównych dostawców chmury, w Kubernetesie, a nawet w wewnętrznych centrach danych (on-premises).

Uwaga

Więcej informacji na temat funkcji programu Vault można znaleźć na stronie produktu pod adresem <https://www.vaultproject.io/docs/what-is-vault/index.html>.

Dzięki tym wszystkim bardzo interesującym funkcjom Vault jest narzędziem, jakie polecam każdej firmie, która chce chronić swoje poufne informacje i zintegrować swoje bezpieczeństwo z potokiem CI/CD. Oprócz tego, że jest bardzo wydajny w ochronie danych, Vault świetnie integruje się z potokami CI/CD. Te potoki będą mogły korzystać z chronionych danych podczas udostępniania infrastruktury i wdrażania aplikacji.

Po omówieniu programu Vault przystąpimy do instalacji programu Vault na komputerze lokalnym i użyjemy go do szyfrowania i odszyfrowywania danych. Omówimy również interfejs użytkownika (UI) Vault, a na koniec przedstawimy proces pobierania danych z Vault w Terraform.

Lokalna instalacja programu Vault

Decydując się na korzystanie z Vault, warto wiedzieć, że jest to narzędzie, które odpowiada za bezpieczeństwo Twojej wrażliwej infrastruktury i danych aplikacji. W praktyce Vault to nie tylko narzędzie, a przed zainstalowaniem go w środowisku produkcyjnym należy zrozumieć jego koncepcje i architekturę.

Uwaga

Aby dowiedzieć się więcej o topologiach architektury Vault, zapoznaj się z dokumentacją pod adresem <https://learn.hashicorp.com/vault/operations/ops-reference-architecture>.

Dlatego celem rozdziału nie jest zagłębianie się w szczegóły koncepcji i architektury programu Vault, ale wyjaśnienie instalacji i korzystania z programu Vault w trybie

programistycznym. Innymi słowy: zainstalujemy Vault na lokalnej stacji roboczej, aby mieć małą instancję używaną do testowania i programowania.

W tej książce szczegółowo opisaliśmy wykorzystanie narzędzi HashiCorp z Terraform i Packerem. Podobnie Vault można zainstalować ręcznie lub za pomocą skryptu w następujący sposób:

- Aby **ręcznie zainstalować program Vault**, wykonamy procedurę dokładnie taką samą jak w przypadku instalowania programów Terraform i Packer, dlatego należy wykonać następujące czynności:
 - A. Przejdź do strony pobierania: <https://www.vaultproject.io/downloads.html>.
 - B. Pobierz do wybranego folderu pakiet związany z Twoim systemem operacyjnym.
 - C. Rozpakuj pakiet i zaktualizuj zmienną środowiskową PATH, podając jej ścieżkę do tego folderu.
- Aby **automatycznie zainstalować Vault**, użyjemy skryptu, którego kod zależy od naszego systemu operacyjnego.

W systemie Linux w terminalu uruchom następujący skrypt (pobrany z <https://www.vaultproject.io/downloads>):

```
curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo  
apt-key add -sudo apt-add-repository "deb [arch=amd64]  
↳https://apt.releases.hashicorp.com $(lsb_release -cs) main"sudo apt-get  
↳update && sudo apt-get install vault
```

Ten skrypt wykonuje następujące czynności:

1. Rejestruje się w rejestrze pakietów HashiCorp.
2. Instaluje pakiet Vault.

W systemie **Windows**, aby zainstalować Vault za pomocą skryptu, użyjemy **Chocolatey**, czyli menedżera pakietów oprogramowania Windows (<https://chocolatey.org/>), wykonując następujące polecenie w terminalu:

```
choco install vault -y
```

Polecenie to pobiera i instaluje Vault za pomocą Chocolatey.

Oprócz tych skryptów, które umożliwiają instalację Vault na lokalnej stacji roboczej, HashiCorp zapewnia również kod Terraform, który pozwala na utworzenie kompletnej infrastruktury Vault dla różnych dostawców usług chmurowych.

Uwaga

Kod Terraform jest dostępny dla platformy Azure pod adresem <https://github.com/hashicorp/terraform-azurerm-vault>; dla AWS jest dostępny pod adresem <https://github.com/hashicorp/terraform-aws-vault>; a dla GCP jest dostępny pod adresem <https://github.com/hashicorp/terraform-google-vault>.

W przypadku **Kubernetesa** możemy skorzystać z charta Helma. Aby zainstalować Vault w instancji Kubernetesa, użyj następującego skryptu:

```
helm repo add hashicorp https://helm.releases.hashicorp.com
kubectl create namespace vault
helm install vault hashicorp/vault --namespace vault
```

W powyższym skrypcie wykonujemy następujące czynności:

1. Dodanie lokalnego repozytorium HashiCorp dla Helma.
2. Utworzenie przestrzeni nazw `vault`.
3. Instalacja Vault w przestrzeni nazw `vault` za pomocą charta Helma.

Aby uzyskać więcej informacji na temat instalacji Vault na Kubernetesie, przeczytaj samouczek tutaj: <https://learn.hashicorp.com/tutorials/vault/kubernetes-raft-deployment-guide?in=vault/kubernetes>.

Po zainstalowaniu programu Vault przetestujemy jego instalację, uruchamiając następujące polecenie w terminalu:

```
vault --version
```

Polecenie to wyświetla zainstalowaną wersję programu Vault. Możemy również wykonać polecenie `vault --help`, aby wyświetlić listę dostępnych poleceń.

Właśnie poznaliśmy różne sposoby instalacji Vault na komputerze lokalnym lub na Kubernetesie; następnym krokiem jest uruchomienie serwera Vault.

Uruchamianie serwera Vault

Vault to narzędzie typu klient/serwer, które składa się z komponentu klienta używanego przez programistów do tworzenia aplikacji i komponentu serwera odpowiedzialnego za ochronę danych w zdalnych zapleczech (ang. *backend*).

Uwaga

Vault obsługuje bardzo dużą liczbę backendów, których lista jest dostępna tutaj: <https://www.vaultproject.io/docs/configuration/storage/index.html>.

Po zainstalowaniu Vault lokalnie mamy dostęp tylko do części klienckiej i aby móc korzystać z Vault, uruchomimy komponent serwerowy.

Aby uruchomić komponent serwera Vault w trybie programistycznym, wykonamy to polecenie w drugim terminalu:

```
vault server -dev
```

Polecenie to uruchamia i konfiguruje serwer Vault z minimalną konfiguracją zawierającą token uwierzytelniania i domyślny backend, zwany *in-memory*, który przechowuje wszystkie tajne dane w pamięci serwera, jak pokazano na poniższym zrzucie ekranu:

```
minae@P-FYL2202:/mnt/c/Users/murlof$ vault server -dev
==> Vault server configuration:

  Api Address: http://127.0.0.1:8200
  Cgo: disabled
  Cluster Address: https://127.0.0.1:8201
  Go Version: go1.17.5
  Listener 1: tcp (Addr: "127.0.0.1:8200", cluster address: "127.0.0.1:8201", max_request_duration: "1m30s", max_request_size: "33554432", tls: "disabled")
  Log Level: info
  Lock: supported: true, enabled: false
  Recovery Mode: false
  Storage: inmem
  Version: Vault v1.9.2
  Version Sha: f4c6d873e2707c0d6853b5d9ffc77b0d297bfdbf

==> Vault server started! Log data will stream in below:

WARNING! dev mode is enabled! In this mode, Vault runs entirely "in-memory"
and starts unsealed with a single unseal key. The root token is already
authenticated to the CLI, so you can immediately begin using Vault.

You may need to set the following environment variable:

  $ export VAULT_ADDR='http://127.0.0.1:8200'

The unseal key and root token are displayed below in case you want to
seal/unseal the Vault or re-authenticate.

Unseal Key: 6LqjGz8nHv3iukdQJM98Rh8f8T/+Njvjoa6UR0m7ZCo=
Root Token: s.oSr2LL8mqxWwy9876pKrcIo

Development mode should NOT be used in production installations!
```

Rysunek 14.6. Uruchomienie programu Vault — tryb programistyczny

Ważna uwaga

Terminal musi pozostać otwarty, aby serwer Vault działał.

Ponadto, ponieważ jesteśmy w trybie programistycznym, zasoby zaplecza znajdują się tylko w pamięci. Po zatrzymaniu serwera Vault wszystkie tajne dane są usuwane z pamięci.

Następnie, jak wskazano podczas tego wykonania, wyeksportujemy zmienną środowiskową `VAULT_ADDR` za pomocą tego polecenia w innym terminalu:

```
export VAULT_ADDR='http://127.0.0.1:8200'
```

Na koniec, aby sprawdzić stan wykonania serwera Vault, wykonujemy następujące polecenie:

```
vault status
```

Oto dane wyjściowe polecenia, które wyświetlają właściwości serwera Vault:

```
root@mikrrief:/home/mikaelkrief# vault status
Key          Value
---          -
Seal Type    shamir
Initialized   true
Sealed       false
Total Shares  1
Threshold    1
Version      1.2.1
Cluster Name  vault-cluster-9e7d6ef4
Cluster ID    82bfd424-73ce-9c3d-1dd1-dae6d88bb604
HA Enabled    false
```

Rysunek 14.7. Status Vault

Uwaga

Aby dowiedzieć się więcej o serwerze Vault uruchomionym w trybie programistycznym, przeczytaj dokumentację pod adresem <https://www.vaultproject.io/docs/concepts/dev-server.html>.

Teraz, gdy Vault jest zainstalowany i serwer jest uruchomiony, zobaczymy, jak zapisywać dane w Vault, aby je chronić, i jak odczytać te dane, aby można było z nich korzystać w aplikacji innej firmy.

Zapisywanie haseł w Vault

Gdy chcesz zabezpieczyć wrażliwe dane, które będą wykorzystywane przez aplikację lub zasoby infrastruktury, pierwszym krokiem jest przechowywanie tych danych w wybranym przez firmę menedżerze haseł. Zobaczymy w praktyce, jak wyglądają kroki zapisywania danych w Vault.

Aby chronić dane w Vault, przechodzimy do terminala i wykonujemy następujące polecenie:

```
vault kv put secret/vmadmin vmpassword=admin123*
```

Poniższy zrzut ekranu pokazuje jego wykonanie.

Polecenie z operacją `put` tworzy nowe tajne dane typu klucz-wartość w pamięci z tytułem `vmadmin`. W tym przykładzie jest to konto administratora maszyny wirtualnej w ścieżce `secret/`.

W Vault wszystkie chronione dane są przechowywane w ścieżce odpowiadającej lokalizacji organizacyjnej w Vault. Domyślna ścieżka do Vault to `secret/`. Możliwe jest tworzenie

```
root@mkrrief:/home/mikaelkrief# vault kv put secret/vmadmin vmpassword=admin123*
Key                               Value
---                               -
created_time                      2019-08-13T13:56:14.5200652Z
deletion_time                    n/a
destroyed                        false
version                          1
```

Rysunek 14.8. Zapis hasła w Vault za pomocą polecenia vault kv put

niestandardowych ścieżek, które pozwolą na lepsze zarządzanie prawami i lepszą organizację według domeny, tematu lub aplikacji.

Jeśli chodzi o sekrety przechowywane w Vault, jedną z ich zalet jest możliwość przechowywania wielu danych w tym samym sekrecie; np. zaktualizujemy utworzone przez nas tajne dane za pomocą innego sekretu, którym jest administrator logowania maszyny wirtualnej.

W tym celu wykonamy następujące polecenie, które doda kolejny tajny klucz-wartość do tych samych danych Vault:

```
vault kv put secret/vmadmin vmpassword=admin123*  
vmadmin=bookadmin
```

Jak widać, w tym wykonaniu użyliśmy dokładnie tego samego polecenia z tym samym sekretem i dodaliśmy nowe dane klucz-wartość, czyli vmadmin.

Uwaga

Aby uzyskać więcej informacji na temat polecenia kv put, przeczytaj dokumentację pod adresem <https://www.vaultproject.io/docs/commands/kv/put.html>.

Nauczyliśmy się, jak używać poleceń do tworzenia sekretu w Vault i różnych jego zastosowań, a teraz przyjrzymy się poleceniu odczytywania tego sekretu w celu użycia go w aplikacji lub w zasobach infrastruktury.

Odczytywanie sekretów z Vault

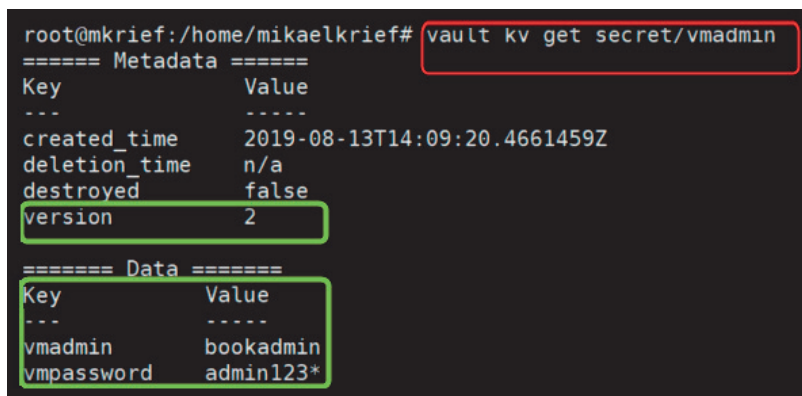
Po utworzeniu sekretów w Vault będziemy musieli je odczytać, aby użyć ich w naszych aplikacjach lub skryptach infrastruktury.

Aby odczytać klucz przechowywany w Vault, przechodzimy do terminala, by wykonać to polecenie:

```
vault kv get secret/vmadmin
```

W tym poleceniu używamy operacji `kv` z operatorem `get` i wskazujemy w parametrze pełną ścieżkę klucza, aby uzyskać chronioną wartość w naszym przykładzie `secret/vmadmin`.

Poniższy zrzut ekranu pokazuje wykonanie polecenia, a także jego wynik:



```
root@mkrif:/home/mikaelkrief# vault kv get secret/vmadmin
===== Metadata =====
Key          Value
---          -
created_time  2019-08-13T14:09:20.4661459Z
deletion_time n/a
destroyed     false
version       2
===== Data =====
Key          Value
---          -
vmadmin      bookadmin
vmpassword   admin123*
```

Rysunek 14.9. Pobranie sekretu za pomocą polecenia `vault kv get`

W danych wyjściowych tego polecenia zauważymy, że:

- Numer wersji sekretu wynosi tutaj 2, ponieważ dwukrotnie wykonaliśmy polecenie `kv put`, więc numer wersji był zwiększany przy każdym wykonaniu.
- Istnieją dwa elementy danych klucz-wartość, które chroniliśmy w sekrecie w poprzedniej sekcji „Zapisywanie haseł w Vault”.

Jeśli chcesz uzyskać dostęp do danych przechowywanych w tym sekrecie, ale z wcześniejszej wersji, możesz wykonać to samo polecenie, opcjonalnie określając żądany numer wersji, jak w tym przykładzie:

```
vault kv get -version=1 secret/vmadmin
```

Poniższy zrzut ekranu pokazuje jego wykonanie.

W sekcji `Data` możemy zobaczyć pierwszą wersję danych klucz-wartość, które mieliśmy podczas pierwszego wykonania polecenia `kv put`.

Uwaga

Aby uzyskać więcej informacji na temat polecenia `kv get`, zapoznaj się z dokumentacją pod adresem <https://www.vaultproject.io/docs/commands/kv/get.html>.

```
root@mkrif:/home/mikaelkrief# vault kv get -version=1 secret/vmadmin
===== Metadata =====
Key          Value
---          -
created_time  2019-08-13T13:56:14.5200652Z
deletion_time n/a
destroyed     false
version       1

===== Data =====
Key          Value
---          -
vmpassword   admin123*
root@mkrif:/home/mikaelkrief#
```

Rysunek 14.10. Pobranie sekretu według wersji

Właśnie widzieliśmy zastosowanie polecenia `kv get`, aby pobrać wszystkie lub określone wersje wartości sekretu; teraz pokrótce zobaczymy, jak używać interfejsu webowego Vault UI do lepszego zarządzania sekretami.

Korzystanie z interfejsu webowego (UI) programu Vault

Jedną z interesujących funkcji Vault jest to, że oprócz narzędzia po stronie klienta, które umożliwia wykonywanie wszystkich operacji na serwerze Vault, Vault zawiera interfejs webowy, który pozwala zarządzać sekretami, ale w sposób graficzny.

Aby otworzyć interfejs webowy Vault i użyć go do wizualizacji sekretów utworzonych za pomocą narzędzia klienckiego, musimy wykonać następujące kroki:

1. W przeglądarce wprowadź adres URL podany podczas uruchamiania serwera, czyli `http://127.0.0.1:8200/ui`, który jest domyślnym lokalnym adresem URL Vault.
2. W formularzu uwierzytelniania wprowadź token, który został podany w terminalu w informacjach o tokenie głównym (ang. *root token*), jak pokazano na rysunku 14.11.
3. Kliknij przycisk *Sign In*, aby się uwierzytelnić, jak pokazano na rysunku 14.12.
4. Strona główna interfejsu wyświetla listę ścieżek sekretów, zwanych *Secrets Engines*, zawierających sekrety, które zostały zapisane, jak pokazano na rysunku 14.13.


```

root@mkrif:/home/mikaelkrief# vault server -dev
==> Vault server configuration:

    Api Address: http://127.0.0.1:8200
      Cgo: disabled
    Cluster Address: https://127.0.0.1:8201
    Listener 1: tcp (addr: "127.0.0.1:8200", cluster address: "127.0.0.1:8201")
    Log Level: info
      Mlock: supported: true, enabled: false

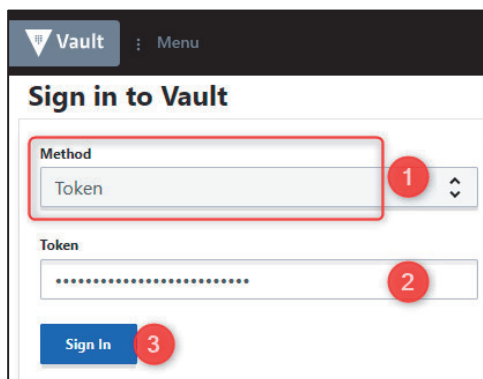
$ export VAULT_ADDR='http://127.0.0.1:8200'

The unseal key and root token are displayed below in case you want to
seal/unseal the Vault or re-authenticate.

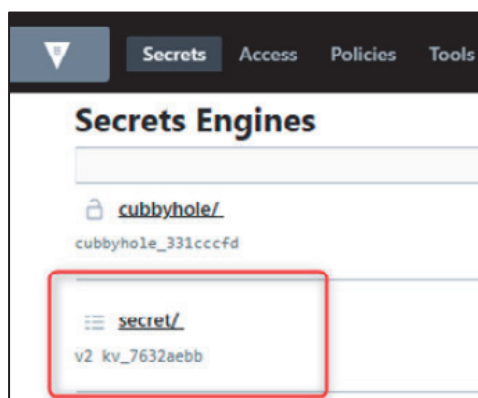
Unseal Key: fh7vj+J8LHHXr+AtTva4DU2Ru4kcPtqQuM9jKBo01BA=
Root Token: s.6MGUVmH1bnhD36aWf0Fb9oR4
Development mode should NOT be used in production installations!

```

Rysunek 14.11. Pobieranie tokena głównego

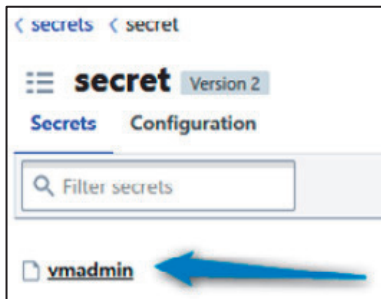


Rysunek 14.12. Logowanie do UI Vault



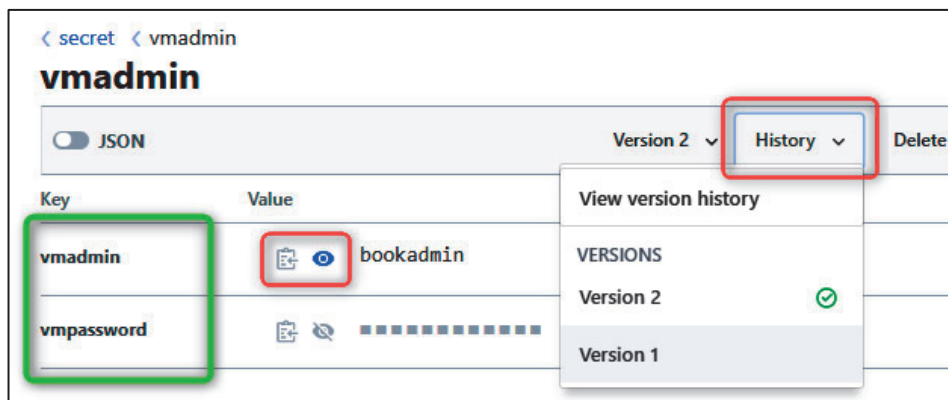
Rysunek 14.13. Secrets Engines

5. Klikając każdą ścieżkę, możesz zobaczyć listę zapisanych sekretów. Poniższy zrzut ekranu przedstawia stronę *Secrets Engines*, gdzie wyświetlany jest sekret, który utworzyliśmy w wierszu poleceń w poprzedniej sekcji:



Rysunek 14.14. Sekret w interfejsie webowym Vault

6. Klikając konkretny sekret, możesz uzyskać dostęp do listy danych, które chronimy, z możliwością przeglądania wartości każdego klucza w postaci jawnego tekstu. Możemy również wyświetlić historię zmian zawartości sekretu, wybierając żądaną wersję z rozwijanego menu *History*, jak pokazano na poniższym zrzucie ekranu:



Rysunek 14.15. Interfejs Vault — odczytywanie szczegółów sekretów

Interfejs webowy programu Vault umożliwia także wykonywanie wszystkich operacji na sekretach i zarządzanie innymi składnikami programu Vault.

Uwaga

Jeśli chcesz dowiedzieć się więcej o interfejsie webowym, przeczytaj ten artykuł: <https://www.hashicorp.com/resources/vault-oss-ui-introduction>.

Dowiedzieliśmy się, że interfejs webowy Vault jest bardzo dobrą alternatywą dla narzędzia klienckiego, które umożliwia przeglądanie elementów Vault i zarządzanie nimi. Po tym przeglądzie narzędzia Vault i jego operacji proponuję zapoznanie się z krótkim przykładem, który pokazuje, jak pobrać sekrety w kodzie Terraform.

Pobieranie sekretów Vault w Terraform

Jak już widzieliśmy w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”, bardzo ważne jest, aby chronić informacje o konfiguracji infrastruktury, które zapisujemy w kodzie Terraform. Jednym ze sposobów ochrony tych wrażliwych danych jest przechowywanie ich w menedżerze haseł takim jak Vault i dynamiczne odzyskiwanie ich bezpośrednio za pomocą Terraform.

Oto przykład kodu Terraform, który umożliwia pobranie hasła maszyny wirtualnej z Vault. Przykładowy kod Terraform składa się z trzech bloków, które są szczegółowo opisane w następujący sposób:

1. Najpierw używamy dostawcy Vault do skonfigurowania adresu URL, wykonując następujący kod:

```
provider "vault" {  
    address = "http://127.0.0.1:8200" #Local Vault Url  
}
```

Dostawca jest konfigurowany za pomocą konfiguracji serwera Vault i jego uwierzytelniania.

W naszym przypadku w kodzie Terraform konfigurujemy adres URL serwera Vault, a do uwierzytelnienia tokena użyjemy zmiennej środowiskowej w momencie uruchomienia Terraform.

Uwaga

Więcej informacji na temat dostawcy Vault i jego konfigurowania można znaleźć w następującej dokumentacji: <https://www.terraform.io/docs/providers/vault/index.html>.

2. Następnie dodajemy blok danych, `vault_generic_secret`, który służy do pobierania sekretu z serwera Vault. Kod jest zilustrowany w następującym fragmencie:

```
data "vault_generic_secret" "vmadmin_account" {  
    path = "secret/vmadmin"  
}
```

Ten blok danych pozwala nam pobrać (w trybie tylko do odczytu) treści sekretu przechowywanego w Vault. W tym miejscu prosimy Terraform o odzyskanie sekretu znajdującego się w ścieżce Vault, sekret/vmadmin, którą utworzyliśmy wcześniej w tej sekcji.

Uwaga

Aby uzyskać więcej informacji na temat danych vault_generic_secret i ich konfiguracji, zobacz następującą dokumentację: https://www.terraform.io/docs/providers/vault/d/generic_secret.html.

3. Na końcu dodajemy blok output, aby użyć odszyfrowanej wartości sekretu, w następujący sposób:

```
output "vmpassword" {  
  value = "${data.vault_generic_secret.vmadmin_account.data["vmpassword"]}"  
  sensitive = true  
}
```

Ten blok stanowi przykład wykorzystania sekretu.

Wyrażenie `data.vault_generic_secret.vmadmin_account.data["vmpassword"]` jest używane do uzyskania sekretu zwróconego przez poprzednio używany blok danych. W tablicy `data` dodajemy nazwy tylko tych kluczy, dla których potrzebujemy odzyskać zaszyfrowane wartości. Ponadto to wyjście jest uważane za wrażliwe, więc Terraform nie wyświetla wartości w postaci jawnego tekstu.

Uwaga

Pełny kod źródłowy Terraform jest również dostępny tutaj: https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP14/vault/terraform_usevault/main.tf.

Zakończyliśmy tworzenie kodu Terraform; teraz szybko go wykonamy, aby zobaczyć odzyskanie sekretu.

Uwaga

W celu wykonania kodu, który omówiliśmy już szczegółowo w rozdziale 2., „Udoskonalanie infrastruktury chmury za pomocą Terraform”, podamy tylko polecenia bez dalszych szczegółów.

Aby uruchomić Terraform, przechodzimy w terminalu do folderu zawierającego kod Terraform, a następnie postępujemy w takiej kolejności:

1. Wyeksportuj zmienną środowiskową `VAULT_TOKEN` z wartością tokena Vault. W przypadku trybu programistycznego ten token jest udostępniany w czasie startu serwera Vault.

Następujące polecenie pokazuje eksport tej zmiennej środowiskowej w systemie operacyjnym Linux:

```
export VAULT_TOKEN=xxxxxxxxxxxx
```

2. Następnie uruchomimy Terraform za pomocą tych poleceń:

```
terraform init
terraform plan
terraform apply
```

Oto krótki opis wykonania tych poleceń:

- Polecenie `terraform init` inicjuje kontekst i pobiera wszystkich niezbędnych dostawców.
- Polecenie `terraform plan` wyświetla podgląd wszystkich zmian, które zostaną wprowadzone przez Terraform.
- Polecenie `terraform apply` stosuje wszystkie zmiany w infrastrukturze i wyświetla wartości wyjściowe.

Uwaga

Aby poznać wszystkie szczegóły dotyczące głównych poleceń Terraform i cyklu życia Terraform, przeczytaj rozdział 2. tej książki, „Udostępnianie infrastruktury chmury za pomocą Terraform”.

Rysunek 14.16. pokazuje wykonanie polecenia `apply`.

Widzimy, że wartość w danych wyjściowych Terraform o nazwie `vmpassword` nie jest wyświetlana w postaci jawnego tekstu w terminalu.

3. Na koniec wyświetlamy dane wyjściowe w formacie *JSON* (ang. *JavaScript Object Notation*) za pomocą polecenia `terraform output` i opcji `-json` — rysunek 14.17.

Widzimy, że Terraform wyświetlił wartość klucza zawartego w sekrecie, który umieściliśmy w Vault. Tej wartości można teraz używać w przypadku wszelkich poufnych danych, które nie powinny być przechowywane w kodzie Terraform, np. takich jak hasła maszyn wirtualnych.

W tej sekcji przestudiowaliśmy użycie narzędzia Vault firmy HashiCorp, które jest menedżerem haseł. Widzieliśmy, że instalacja w różnych systemach operacyjnych może odbywać się ręcznie i automatycznie. Użyliśmy jego poleceń do ochrony i odczytu

```
PS Learning-DevOps-Second-Edition\CHAP14\vault\terraform_usevault> terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:

Terraform will perform the following actions:

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

Changes to Outputs:
  + vmpassword = (sensitive value)

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

  Enter a value: yes

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

Outputs:

vmpassword = <sensitive>
```

Rysunek 14.16. Wyświetlenie danych wrażliwych przez Terraform

```
PS Learning-DevOps-Second-Edition\CHAP14\vault\terraform_usevault> terraform output -json
{
  "vmpassword": {
    "sensitive": true,
    "type": "string",
    "value": "admin123*"
  }
}
```

Rysunek 14.17. Dane wyjściowe Terraform w formacie JSON

Jednak bądź ostrożny

Chronimy nasz kod Terraform, powierzając wszystkie poufne dane menedżerowi haseł. Nie należy jednak zapominać, że Terraform przechowuje wszystkie informacje — w tym dane i informacje wyjściowe — w pliku stanu Terraform. Dlatego bardzo ważne jest, aby go chronić, przechowując plik stanu Terraform w chronionym zdalnym zapleczu, jak widzieliśmy w sekcji „Ochrona pliku stanu za pomocą zdalnego zaplecza” w rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”.

chronionych danych. Następnie powiedzieliśmy, jak zarządzać sekretami w Vault za pomocą interfejsu webowego. Na koniec napisaliśmy i wykonaliśmy kod Terraform, który korzysta z dostawcy Vault i umożliwia nam odzyskanie sekretów przechowywanych na serwerze Vault.

Podsumowanie

Ten rozdział został poświęcony integracji bezpieczeństwa z praktykami DevOps. Przedstawiliśmy trzy narzędzia do weryfikacji i zabezpieczenia Twoich danych i infrastruktury chmurowej. Opisaliśmy, jak sprawdzić zgodność infrastruktury platformy Azure za pomocą narzędzia **InSpec** firmy Chef.

Aby to zrobić, zainstalowaliśmy InSpec, a następnie szczegółowo omówiliśmy testy InSpec. Użyliśmy jego poleceń do zweryfikowania zgodności infrastruktury Azure.

W ostatniej sekcji zobaczyliśmy, jak chronić poufne dane za pomocą Vault firmy HashiCorp. W tej sekcji przyjrzelśmy się szyfrowaniu i deszyfrowaniu danych w Vault i napisaliśmy kod Terraform, który będzie dynamicznie pobierał sekrety przechowywane w Vault.

W kolejnym rozdziale przedstawimy koncepcję **wdrażania niebiesko-zielonego** (ang. *blue-green deployment*) wraz z wzorcami ograniczania przestojów wdrażania. Następnie dowiemy się, jak ją zaimplementować w aplikacji, a także we wdrożeniu infrastruktury Azure.

Pytania

1. Jaka jest rola InSpec?
2. Które polecenie InSpec umożliwia wykonywanie testów InSpec?
3. Kto jest wydawcą Vault?
4. Które polecenie uruchamia program Vault w trybie deweloperskim?
5. Jeśli program Vault jest zainstalowany lokalnie, czy można go używać do produkcji?
6. Gdzie w trybie lokalnym są przechowywane zaszyfrowane dane Vault?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o DevSecOps w połączeniu z InSpec, Vault i **Secure DevOps Kit for Azure (AzSK)**, oto kilka zasobów:

- Dokumentacja InSpec — <https://www.inspec.io/>.
- Dokumentacja Vault — <https://www.vaultproject.io/docs/>.
- Nauka Vault — <https://learn.hashicorp.com/vault>.

Skrócenie czasu przestoju wdrażania

Do tej pory w książce omówiliśmy praktyki DevOps, takie jak **infrastruktura jako kod (IaC)**, **potoki ciągłej integracji/ciągłego wdrażania (CI/CD)** i automatyzację różnych typów testów.

W części 1., „DevOps i infrastruktura jako kod”, widzieliśmy, że te praktyki DevOps poprawiają jakość aplikacji, a tym samym zwiększają zysk finansowy firmy. Teraz zagłębimy się w praktyki DevOps, sprawdzając, jak zapewnić ciągłą dostępność aplikacji nawet podczas wdrożeń i jak częściej dostarczać nowe wersje tych aplikacji w środowisku produkcyjnym.

Często widzimy, że wdrożenia wymagają przerwania pracy aplikacji przez np. zmiany w infrastrukturze lub wyłączenia usług. Co więcej, widzimy również, że firmy nadal niechętnie dostarczają wyroby do produkcji. Nie są wyposażone w narzędzia umożliwiające testowanie aplikacji w środowisku produkcyjnym lub czekają na inne zależności.

W tym rozdziale przyjrzymy się kilku praktykom, które pomogą usprawnić procesy dostarczania aplikacji. Zaczniemy od sposobu skrócenia przestojów Twojej infrastruktury i aplikacji podczas wdrożeń Terraform. Następnie omówimy koncepcję i wzorce wdrożenia zielono-niebieskiego oraz sposób jego konfiguracji z niektórymi zasobami platformy Azure. Na koniec przedstawimy szczegóły implementacji flagi funkcjonalności (ang. *feature flag*) w Twojej aplikacji, co pozwoli Ci na modyfikację działania aplikacji bez konieczności jej ponownego wdrażania w produkcji.

Dowiesz się również, jak skonfigurować kod Terraform, aby skrócić czas przestoju aplikacji. Będziesz mógł konfigurować zasoby platformy Azure za pomocą zielono-niebieskiego wdrożenia i wdrażać flagi funkcjonalności w swoich aplikacjach za pomocą komponentu open source lub platformy LaunchDarkly.

W tym rozdziale omówimy następujące tematy:

- skrócenie przestojów we wdrażaniu dzięki Terraform,
- zrozumienie zielono-niebieskich koncepcji i wzorców wdrażania,
- stosowanie wdrożeń zielono-niebieskich na platformie Azure,
- wprowadzenie flag funkcjonalności,

- używanie frameworka open source dla flag funkcjonalności,
- korzystanie z narzędzia LaunchDarkly.

Wymagania techniczne

Aby zrozumieć koncepcje Terraform, które zostaną przedstawione w tym rozdziale, należy przeczytać rozdział 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”.

Przyjrzymy się przykładowi implementacji zielono-niebieskiego wdrożenia na platformie Azure. Jeśli nie masz subskrypcji platformy Azure, możesz utworzyć bezpłatne konto platformy Azure tutaj: <https://azure.microsoft.com/en-gb/free/>.

Następnie przyjrzymy się przykładowi użycia flag funkcjonalności w aplikacji ASP.NET Core. Aby skorzystać z naszego przykładu, musisz zainstalować bibliotekę .NET Core **Software Development Kit (SDK)**, którą można pobrać ze strony <https://dotnet.microsoft.com/download>.

Do edycji kodu wykorzystaliśmy darmowy edytor **Visual Studio Code (VS Code)**, który jest dostępny do pobrania tutaj: <https://code.visualstudio.com/>.

Pełny kod źródłowy tego rozdziału można znaleźć na stronie <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP15>.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/3hcvrVH>.

Skrócenie czasu przestojów we wdrażaniu dzięki Terraform

W rozdziale 2., „Udostępnianie infrastruktury chmury za pomocą Terraform”, szczegółowo omówiliśmy korzystanie z Terraform, przyglądając się jego poleceniom i cyklowi życia, a następnie zastosowaliśmy to w praktyce za pomocą implementacji na platformie Azure.

Jednym z problemów związanych z Terraform jest to, że w zależności od zmian w infrastrukturze, które należy wdrożyć, Terraform może automatycznie usuwać i odbudowywać niektóre zasoby.

Aby w pełni zrozumieć to zachowanie, przyjrzymy się wynikom następującego wykonania Terraform, które zainicjowało obsługę aplikacji Azure Web App na platformie Azure i zostało zmodyfikowane poprzez zmianę nazwy:

```

PS D:\Repos\Learning_Devops\CHAP13\terraform_webapps> terraform apply
azurerm_resource_group.rg-app: Refreshing state... [id=/subscriptions/8a7aace5-74aa-416f-b8e4-2c292b6304e5/resourceGroups/rgApp/providers/Microsoft.Web/serverfarms/planApp]
azurerm_app_service_plan.serviceplan-app: Refreshing state... [id=/subscriptions/8a7aace5-74aa-416f-b8e4-2c292b6304e5/resourceGroups/rgApp/providers/Microsoft.Web/sites/MyWebAppBook]
azurerm_app_service.webapp: Refreshing state... [id=/subscriptions/8a7aace5-74aa-416f-b8e4-2c292b6304e5/resourceGroups/rgApp/providers/Microsoft.Web/sites/MyWebAppBook/slots/StagedGreen]
azurerm_app_service_slot.test: Refreshing state... [id=/subscriptions/8a7aace5-74aa-416f-b8e4-2c292b6304e5/resourceGroups/rgApp/providers/Microsoft.Web/sites/MyWebAppBook/slots/StagedGreen]

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  ~ destroy and then create replacement
  ~ create replacement and then destroy

Terraform will perform the following actions:

# azurerm_app_service.webapp must be replaced
~ resource "azurerm_app_service" "webapp" {
  app_service_plan_id = "/subscriptions/8a7aace5-74aa-416f-b8e4-2c292b6304e5/resourceGroups/rgApp/providers/Microsoft.Web/serverfarms/planApp"
  app_settings = {
    "WEBSITE_RUN_FROM_PACKAGE" = {}
  }
  ~ client_affinity_enabled = true -> (known after apply)
  ~ client_cert_enabled = false -> (known after apply)
  ~ default_site_hostname = "mywebappbook.azurewebsites.net" -> (known after apply)
  ~ enabled_https_only = false -> (known after apply)
  ~ id = "/subscriptions/8a7aace5-74aa-416f-b8e4-2c292b6304e5/resourceGroups/rgApp/providers/Microsoft.Web/sites/MyWebAppBook" -> (known after apply)
  ~ location = "westeurope" -> (known after apply)
  ~ name = "MyWebAppBook1" -> (known after apply)
  ~ outbound_ip_addresses = "52.233.128.61, 52.233.135.180, 52.233.131.224, 52.233.128.247, 52.233.129.243" -> (known after apply)
  ~ possible_outbound_ip_addresses = "52.233.128.61, 52.233.135.180, 52.233.131.224, 52.233.128.247, 52.233.129.243, 52.233.130.50, 52.233.135.227" -> (known after apply)
  ~ resource_group_name = "rgApp" -> (known after apply)
  ~ site_credentials = [
    {
      password = "Mn2Gnkxvp57nRdr07HBHvt18bw8rncvvg6uTdZw5ETbPcpHxoh1k1F4"
      username = "MyWebAppBook"
    },
  ] -> (known after apply)
  ~ source_control = [
    {
      branch = "master"
      repo_url = ""
    },
  ] -> (known after apply)
  ~ tags = {} -> (known after apply)
}

```

Rysunek 15.1. Przestój Terraform

Widzimy, że Terraform usuwa aplikację webową, a następnie odbuduje ją z nową nazwą. Chociaż usuwanie i rekonstrukcja odbywają się automatycznie, aplikacja będzie niedostępna dla użytkowników.

Aby rozwiązać problem przestojów, możemy dodać do Terraform opcję `create_before_destroy` w następujący sposób:

```

resource "azurerm_app_service" "webapp" {
  name = "MyWebAppBook1" #new name
  location = "West Europe"
  resource_group_name = "${azurerm_resource_group.rg-app.name}"
  app_service_plan_id = "${azurerm_app_service_plan.serviceplan-app.id}"
  app_settings = {
    WEBSITE_RUN_FROM_PACKAGE = var.package_zip_url
  }
  lifecycle { create_before_destroy = true }
}

```

Dodając tę opcję, Terraform wykona następujące czynności:

1. Najpierw tworzy nową aplikację webową o nowej nazwie.
2. Podczas udostępniania nowej aplikacji używa ona *adresu URL* dla pakietu aplikacji w formacie ZIP, który jest podany we właściwości `app_settings`. Użyj `WEBSITE_RUN_FROM_PACKAGE`, aby uruchomić aplikację.
3. Następnie Terraform usuwa starą aplikację webową.

Użycie opcji Terraform `create_before_destroy` zapewni żywotność naszych aplikacji podczas wdrożeń.

Jednak bądź ostrożny, ponieważ ta opcja będzie przydatna tylko wtedy, gdy tworzony nowy zasób nie generuje opóźnień w działaniu, kiedy jest dostarczany, aby nie wystąpiła przerwa w działaniu usługi.

W naszym przykładzie aplikacji webowej skorzystaliśmy z właściwości `WEBSITE_RUN_FROM_PACKAGE`. W przypadku maszyny wirtualnej (VM) możemy użyć obrazu *maszyny wirtualnej* utworzonego przez *Packera*. Jak widzieliśmy w rozdziale 4., „Optymalizacja wdrażania infrastruktury za pomocą Packera”, Packer zawiera informacje dotyczące aplikacji VM, które zostały już zaktualizowane w obrazie VM.

Uwaga

Aby uzyskać więcej informacji na temat opcji `create_before_destroy`, zapoznaj się z następującą dokumentacją Terraform: <https://www.terraform.io/language/meta-arguments/lifecycle>.

Właśnie dowiedzieliśmy się, że dzięki Terraform i IaC możliwe jest skrócenie przestojów podczas wdrożeń w przypadku zmian zasobów.

Przyjrzymy się teraz koncepcjom i wzorcom praktyki zwanej **zielono-niebieskim wdrażaniem**, która pozwala nam z dużą pewnością wdrożyć i przetestować aplikację w środowisku produkcyjnym.

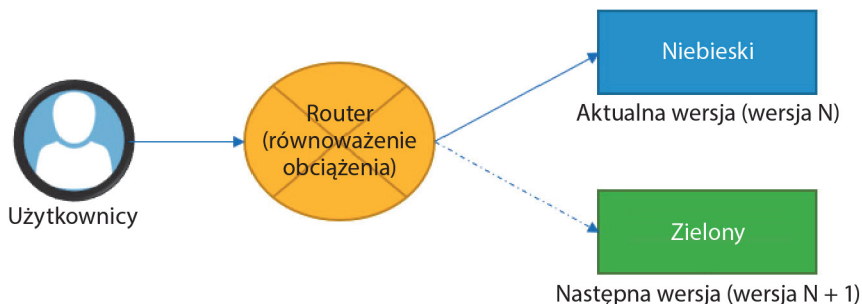
Zrozumienie zielono-niebieskich koncepcji i wzorców wdrażania

Wdrożenie zielono-niebieskie to praktyka, która pozwala nam wdrożyć nową wersję aplikacji w środowisku produkcyjnym bez wpływu na bieżącą wersję aplikacji. W tym podejściu architektura produkcyjna musi się składać z dwóch identycznych środowisk: jedno środowisko jest znane jako środowisko **niebieskie**, a drugie jako środowisko **zielone**.

Elementem umożliwiającym kierowanie ruchu z jednego środowiska do drugiego jest router, czyli system równoważenia obciążenia (ang. *load balancer*).

Poniższy diagram przedstawia uproszczony schemat zielono-niebieskiej architektury.

Jak widać, istnieją dwa identyczne środowiska: środowisko o nazwie **niebieski**, w którym znajduje się aktualna wersja aplikacji, i środowisko o nazwie **zielony**, które reprezentuje nową wersję lub następną wersję aplikacji. Widzimy też router,



Rysunek 15.2. Zielono-niebieska architektura

który przekierowuje żądania użytkowników albo do środowiska niebieskiego, albo do środowiska zielonego.

Teraz, gdy wprowadziliśmy zasadę wdrożenia zielono-niebieskiego, przyjrzymy się, jak wykorzystać ją w praktyce podczas wdrażania.

Korzystanie z wdrożenia zielono-niebieskiego w celu ulepszenia środowiska produkcyjnego

Podstawowy wzorzec wdrożenia zielono-niebieskiego wygląda następująco: kiedy wdrażamy nowe wersje aplikacji, aplikacja jest wdrażana w niebieskim środowisku (wersja N), a router jest konfigurowany w tym środowisku.

Podczas wdrażania kolejnej wersji (wersja $N + 1$) aplikacja zostanie wdrożona w zielonym środowisku, a router zostanie skonfigurowany w tym środowisku.

Niebieskie środowisko staje się nieużywane i bezczynne do czasu wdrożenia wersji $N + 2$. Przyda się również w przypadku szybkiego powrotu do wersji N .

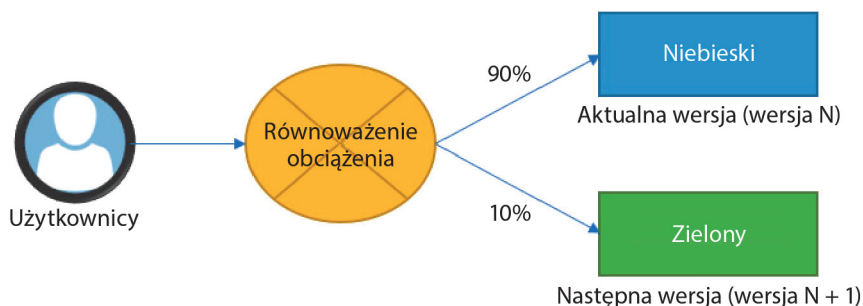
Tę praktykę rozmieszczania zielono-niebieskiego można również odrzucić w przypadku kilku wzorców — tzn. wzorców wdrożeń Canary release i Dark launch. Omówmy szczegółowo implementację każdego z tych wzorców. Zaczniemy od Canary release.

Opis wzorca Canary release

Technika Canary release jest bardzo podobna do wdrożenia zielono-niebieskiego. Nowa wersja aplikacji jest wdrażana w zielonym środowisku, ale tylko dla wąskiej, ograniczonej grupy użytkowników, którzy przetestują aplikację w rzeczywistych warunkach produkcyjnych.

Ta praktyka polega na skonfigurowaniu routera (lub modułu równoważenia obciążenia) tak, by wersja aplikacji była przekierowywana do obu środowisk. Na tym routerze stosujemy ograniczenia przekierowania grupy użytkowników, aby można było ją przekierować tylko do zielonego środowiska, które zawiera nową wersję.

Oto przykładowy diagram wzorca Canary release:

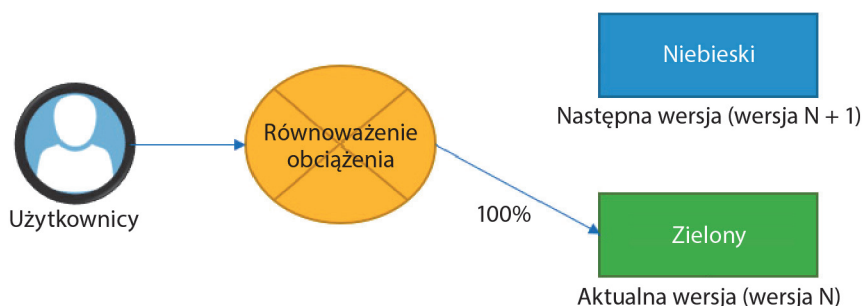


Rysunek 15.3. Zielono-niebieskie wdrożenie Canary release

Na powyższym diagramie router przekierowuje 90% użytkowników do środowiska niebieskiego i 10% użytkowników do środowiska zielonego, w którym znajduje się nowa wersja aplikacji.

Następnie, po przeprowadzeniu testów przez tę grupę użytkowników, router można w pełni skonfigurować w środowisku zielonym, pozostawiając wolne środowisko niebieskie do testowania kolejnej wersji ($N + 2$).

Jak pokazano na poniższym diagramie, router jest skonfigurowany tak, aby przekierowywać wszystkich użytkowników do zielonego środowiska:



Rysunek 15.4. Architektura zielono-niebieska z routerem

Ta technika wdrażania umożliwia zatem wdrożenie i przetestowanie aplikacji w rzeczywistym środowisku produkcyjnym bez konieczności wywierania wpływu na wszystkich użytkowników.

Przyjrzymy się praktycznej implementacji tego wzorca wdrażania zielono-niebieskiego na platformie Azure w dalszej części tego rozdziału, w sekcji „Stosowanie wdrożeń zielono-niebieskich na platformie Azure”. Ale zanim to nastąpi, przyjrzymy się innemu schematowi wdrażania — wzorcowi Dark launch.

Badanie wzorca Dark launch

Wzorzec Dark launch to kolejna praktyka związana z wdrażaniem zielono-niebieskim, która polega na wdrażaniu nowych funkcji w trybie ukrytym lub wyłączonym (tak, aby były niedostępne) w środowisku produkcyjnym. Następnie, gdy chcemy mieć dostęp do tych funkcji we wdrożonej aplikacji, możemy je aktywować na bieżąco bez konieczności ponownego wdrażania aplikacji.

W przeciwieństwie do wzorca Canary release, Dark launch nie jest wdrożeniem zielono-niebieskim, które zależy od infrastruktury, ale jest zaimplementowany w kodzie aplikacji. Aby skonfigurować Dark launch, konieczne jest zamknięcie kodu każdej funkcji aplikacji w elementach zwanych **flagami funkcjonalności** (ang. *feature flags*) (lub **przełącznikami funkcjonalności**, ang. *feature toggles*), które będą używane do zdalnego włączania lub wyłączania tych funkcji.

W kilku ostatnich sekcjach tego rozdziału przyjrzymy się implementacji i wykorzystaniu flag funkcjonalności w środowisku open source i na platformie chmurowej.

W tej sekcji przedstawiliśmy praktykę wdrażania zielono-niebieskiego wraz z jej koncepcjami i wzorcami, takimi jak wzorce Canary release i Dark launch. Mówiliśmy, że ta praktyka wymaga wprowadzenia zmian w infrastrukturze produkcyjnej, ponieważ składa się ona z dwóch instancji infrastruktury — jednej niebieskiej i jednej zielonej — oraz routera, który przekierowuje żądania użytkowników.

Teraz, gdy omówiliśmy wzorce wdrażania zielono-niebieskie, przyjrzymy się, jak wdrożyć je w praktyce w infrastrukturze chmury Azure.

Stosowanie wdrożeń zielono-niebieskich na platformie Azure

Teraz, gdy przyjrzeliliśmy się wdrożeniu zielono-niebieskiemu, przyjrzymy się, jak zastosować je w infrastrukturze platformy Azure przy użyciu dwóch typów komponentów: usługi App Service i usługi Azure Traffic Manager.

Zacznijmy od najbardziej podstawowego składnika — usługi App Service.

Używanie App Service z gniazdami

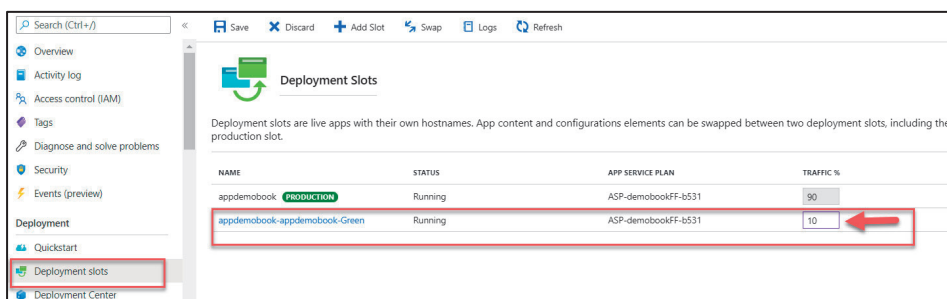
Jeśli mamy subskrypcję platformy Azure i chcemy korzystać z wdrożenia niebiesko-zielonego bez inwestowania dużego nakładu pracy, możemy użyć slotów App Service (Azure Web Apps lub Azure Functions).

W usłudze Azure App Services, takiej jak aplikacja webowa, możemy utworzyć drugie wystąpienie naszej aplikacji, tworząc dla niej slot (do 20 slotów, w zależności od planu usługi App Service). Jest to dodatkowa aplikacja webowa, ale jest dołączona do naszej głównej aplikacji.

Innymi słowy: główna aplikacja reprezentuje niebieskie środowisko, a slot reprezentuje zielone środowisko.

Aby korzystać z tej aplikacji i jej slotu w postaci architektury zielono-niebieskiej, wykonamy następujące kroki konfiguracyjne:

1. Po utworzeniu slotu aplikacji webowej nowa wersja aplikacji zostanie w nim wdrożona i możemy przypisać jej pewien procent ruchu, jak pokazano na poniższym zrzucie ekranu:

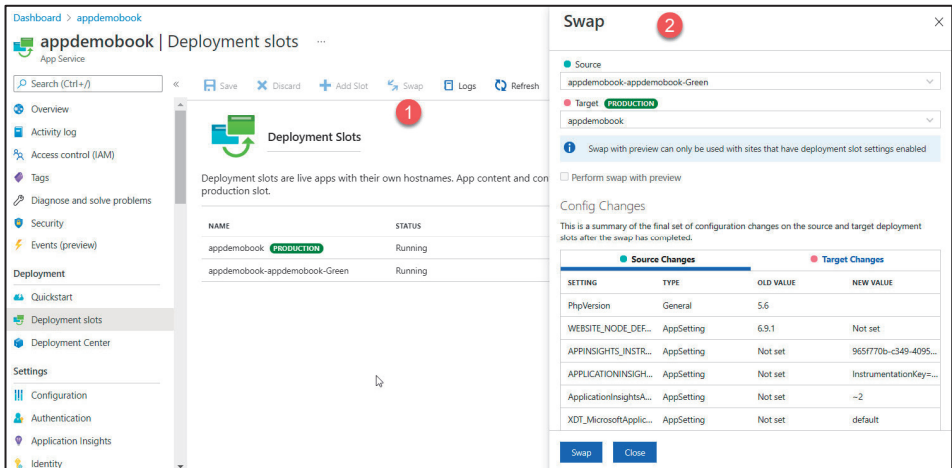


Rysunek 15.5. Sloty wdrażania na platformie Azure

Tutaj przypisaliśmy 10% ruchu do slotu aplikacji webowej, która zawiera zmiany w nowej wersji aplikacji.

2. Gdy tylko nowa wersja aplikacji zostanie przetestowana na slotcie, możemy zamienić slot na główną aplikację internetową (niebieskie środowisko), jak pokazano na poniższym zrzucie ekranu.

Dzięki tej zamianie aplikacja pobiera zawartość swojego slotu i odwrotnie.



Rysunek 15.6. Zamiana slotów

Aplikacja uzyskała teraz nową wersję ($N + 1$), a slot zawiera starszą wersję (N). W przypadku wystąpienia nagłego problemu możemy przywrócić poprzednią wersję aplikacji poprzez ponowne wykonanie zamiany.

Uwaga

Aby dowiedzieć się więcej o konfigurowaniu i używaniu slotów aplikacji, możesz przeczytać następującą dokumentację: <https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots>.

Dokładnie to widzieliśmy we wzorcu Canary release, który pozwala nam dystrybuować ruch produkcyjny dla grupy użytkowników, a także kierować aplikację do środowiska, w którym znajduje się wersja $N + 1$ aplikacji.

Teraz, gdy omówiliśmy korzystanie ze slotów, przyjrzymy się składnikowi usługi Azure Traffic Manager, który również umożliwia nam wdrożenie zielono-niebieskie.

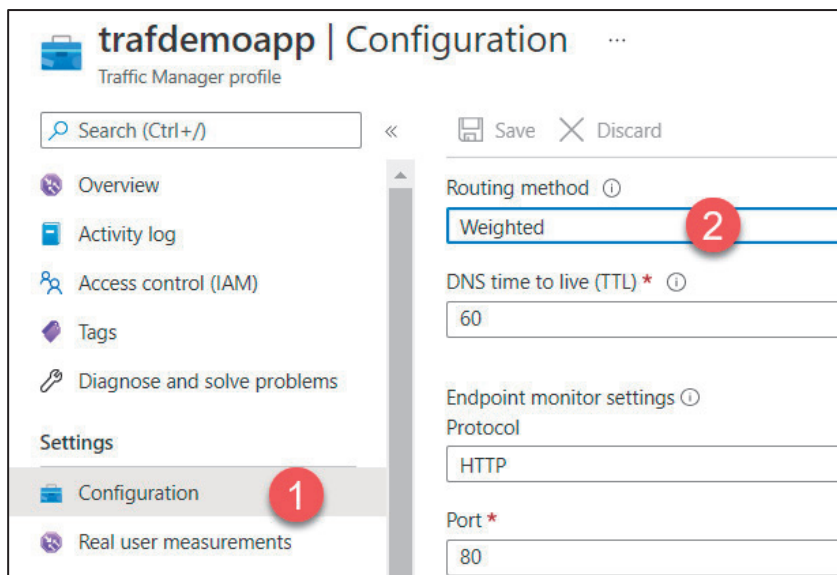
Korzystanie z usługi Azure Traffic Manager

Na platformie Azure dostępny jest komponent o nazwie **Azure Traffic Manager**, który pozwala nam zarządzać ruchem między kilkoma punktami końcowymi zasobów, takimi jak dwie aplikacje webowe.

Musimy posiadać dwie aplikacje: jedną dla niebieskiego środowiska, a drugą dla zielonego środowiska.

Następnie w ramach naszej subskrypcji platformy Azure musimy utworzyć usługę Azure Traffic Manager, którą skonfigurujemy, wykonując następujące czynności:

1. Najpierw w Traffic Managerze skonfigurujemy profil określający sposób kierowania ruchu. W naszym przypadku skonfigurujemy profil *Weighted* — czyli skonfigurujemy go zgodnie z wagą, którą przypiszemy w naszej aplikacji internetowej. Poniższy zrzut ekranu przedstawia konfigurację profilu według wagi:



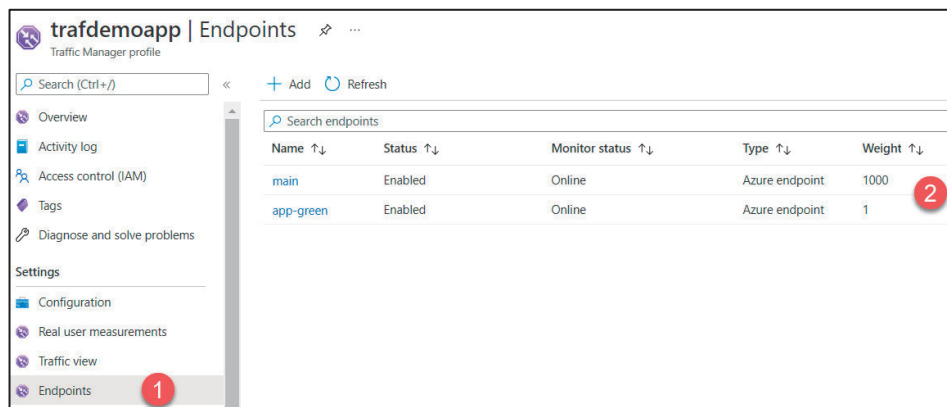
Rysunek 15.7. Konfiguracja usługi Azure Traffic Manager

Uwaga

Aby dowiedzieć się o innych opcjach konfiguracji profilu i sposobie ich działania, możesz przeczytać następującą dokumentację: <https://docs.microsoft.com/en-us/azure/traffic-manager/traffic-manager-routing-methods>.

2. Następnie zarejestrujemy punkty końcowe, które składają się na nasze dwie aplikacje. Dla każdego z nich skonfigurujemy wagę, jak pokazano na poniższym zrzucie ekranu.

Zatem punkt końcowy *main* (czyli niebieskie środowisko) ma maksymalną wagę, która odpowiada 100% ruchu.



Rysunek 15.8. Punkty końcowe usługi Azure Traffic Manager

Uwaga

Jeśli chcesz dowiedzieć się więcej o konfigurowaniu Traffic Managera, skorzystaj z tego samouczka: <https://docs.microsoft.com/en-us/azure/traffic-manager/tutorial-traffic-manager-weighted-endpoint-routing>.

Jeśli chodzi o sloty App Service, to za pomocą usługi Traffic Manager możemy dostosować tę wagę zgodnie z ruchem, jaki chcemy uzyskać w każdym punkcie końcowym, a następnie zastosować wdrożenie zielono-niebieskie.

Właśnie omówiliśmy implementację wdrożenia zielono-niebieskiego, a dokładniej wdrożenie Canary release w infrastrukturze Azure przy użyciu kilku rozwiązań:

- W przypadku pierwszego rozwiązania użyliśmy slotu dla aplikacji webowej i skonfigurowaliśmy dla niego procentowy ruch użytkowników.
- W przypadku drugiego rozwiązania wykorzystaliśmy i skonfigurowaliśmy zasób Azure Traffic Manager, który działa jako router między dwiema aplikacjami.

Przyjrzyjmy się teraz szczegółowo schematowi Dark launch, zaczynając od wprowadzenia flag funkcjonalności i ich implementacji.

Wprowadzenie flag funkcjonalności

Flagi funkcjonalności (zwane także przełącznikami funkcji) umożliwiają nam dynamiczne włączanie i wyłączanie funkcji aplikacji bez konieczności jej ponownego wdrażania.

W przeciwieństwie do wdrożenia zielono-niebieskiego z wzorcem Canary release, który jest koncepcją architektoniczną, flagi funkcjonalności są zaimplementowane w kodzie aplikacji. Ich implementacja odbywa się za pomocą prostej enkapsulacji przy użyciu warunkowych reguł `if`, jak pokazano w poniższym przykładzie kodu:

```
if(activateFeature("addTaxToOrder")==True) {  
    ordervalue = ordervalue + tax  
}else{  
    ordervalue = ordervalue  
}
```

W tym kodzie funkcja `activateFeature` pozwala nam dowiedzieć się, czy aplikacja powinna dodać podatek do zamówienia zgodnie z parametrem `addTaxToOrder`, który jest określony poza aplikacją (np. w bazie danych lub w pliku konfiguracyjnym).

Funkcje zawarte we flagach funkcjonalności mogą być niezbędne do działania aplikacji lub do celów wewnętrznych, takich jak aktywacja logowania lub monitorowanie.

Aktywacja i dezaktywacja funkcji może być kontrolowana przez administratora lub bezpośrednio przez użytkowników za pośrednictwem interfejsu graficznego.

Czas życia flagi funkcjonalności może być następujący:

- **Tymczasowy** — tylko aby przetestować funkcję. Po zatwierdzeniu zmiany przez użytkowników flaga funkcjonalności zostanie usunięta.
- **Ostateczny** — pozostawienie flagi funkcjonalności na długi czas.

Dzięki temu przy użyciu flag funkcjonalności nowa wersja aplikacji może zostać szybciej wdrożona na etapie produkcyjnym. Odbywa się to poprzez wyłączenie nowych funkcjonalności dla danej wersji aplikacji. Następnie ponownie aktywujemy te nowe funkcje dla określonej grupy użytkowników, takich jak testerzy, którzy będą testować te funkcje bezpośrednio w środowisku produkcyjnym.

Co więcej, jeśli zauważymy, że któraś z funkcjonalności aplikacji nie działa poprawnie, flagi mogą ją bardzo szybko wyłączyć bez konieczności ponownego instalowania przez nas reszty aplikacji.

Flagi funkcjonalności umożliwiają również przeprowadzanie testów A/B, czyli testowanie zachowania nowych funkcji przez określonych użytkowników i zbieranie ich opinii.

Jeśli chodzi o implementację flag funkcjonalności w aplikacji, istnieje kilka rozwiązań technicznych, jak opisano tutaj:

- Opracowujesz i utrzymujesz własny system flag funkcjonalności, który został dostosowany do Twoich potrzeb biznesowych. Rozwiązanie to będzie odpowiednie dla Twoich potrzeb, ale wymaga dużo czasu na opracowanie,

a także niezbędnych rozważań dotyczących specyfikacji architektury, takich jak korzystanie z bazy danych, bezpieczeństwo danych i buforowanie danych.

- Używasz narzędzia typu open source, które musisz zainstalować w swoim projekcie. Rozwiązanie to pozwala Ci zaoszczędzić czas poświęcony na programowanie, ale wymaga wyboru narzędzi, szczególnie w przypadku narzędzi otwartoźródłowych. Co więcej, wśród nich niewiele oferuje administrację portalem lub dashboardem, która pozwala na zdalne zarządzanie flagami funkcjonalności. Istnieje wiele frameworków i narzędzi typu open source, które wspierają flagi funkcjonalności. Przejdź pod adres <http://featureflags.io/resources/>, aby je znaleźć. Zapoznaj się również z poniższymi informacjami:
 - RimDev.FeatureFlags (<https://github.com/ritterim/RimDev.FeatureFlags>).
 - Flager (<https://github.com/checkr/flagr>).
 - Unleash (<https://github.com/Unleash/unleash>).
 - Togglz (<https://github.com/togglz/togglz>).
 - Flip (<https://github.com/pda/flip>).
- Możesz skorzystać z rozwiązania chmurowego (**platforma jako usługa**, ang. *platform as a service*, lub PaaS), które nie wymaga instalacji i ma zaplecze do zarządzania flagami funkcjonalności, ale większość z nich wymaga inwestycji finansowych w celu wykorzystania na dużą skalę w przedsiębiorstwie. Wśród tych rozwiązań możemy wymienić następujące:
 - LaunchDarkly (<https://launchdarkly.com/>).
 - Rollout (<https://app.rollout.io/signup>).
 - Featureflow (<https://www.featureflow.io/>).

W tej sekcji dowiedzieliśmy się, że użycie flag funkcjonalności jest praktyką programistyczną, która pozwala przetestować aplikację bezpośrednio na etapie produkcyjnym.

Wspomnieliśmy również o różnych rozwiązaniach wykorzystania flag i zilustrowaliśmy ich implementację. Omówimy jedną z ich implementacji za pomocą narzędzia open source znanego jako RimDev.FeatureFlags.

Używanie frameworka open source dla flag funkcjonalności

Jak widzieliśmy, istnieje duża liczba narzędzi lub platform typu open source, które pozwalają nam używać flag funkcjonalności w naszych aplikacjach.

W tej sekcji przyjrzymy się przykładowi implementacji flag funkcjonalności w aplikacji .NET (Core) przy użyciu prostego frameworka o nazwie `RimDev.FeatureFlags`.

`RimDev.FeatureFlags` to darmowy i otwartoźródłowy framework utworzony za pomocą platformy .NET (<https://github.com/ritterim/RimDev.FeatureFlags>). Jest budowany i dystrybuowany za pośrednictwem pakietu NuGet. Można go znaleźć tutaj: <https://www.nuget.org/packages/RimDev.AspNetCore.FeatureFlags>.

Aby przechowywać dane flag, `RimDev.FeatureFlags` używa bazy danych, która musi zostać wcześniej utworzona. Zaletą `RimDev.FeatureFlags` jest to, że po zaimplementowaniu w naszej aplikacji udostępnia on webowy **interfejs użytkownika (UI)**, który pozwala nam włączać i wyłączać flagi funkcjonalności.

Warunkiem początkowym dla tego przykładu jest zainicjowana aplikacja ASP.NET Core MVC. Użyjemy bazy danych SQL Server, która została utworzona do przechowywania danych flag funkcjonalności.

Aby zainicjować `RimDev.FeatureFlags` w tej aplikacji, wykonamy następujące kroki:

1. Pierwszy krok polega na odwołaniu się w naszej aplikacji do pakietu NuGet `RimDev.FeatureFlags` i zmodyfikowaniu (dowolnym edytorem tekstu) pliku `.csproj`, który znajduje się w katalogu głównym plików aplikacji i zawiera pewne parametry. Dodajemy element `PackageReference` w następujący sposób:

```
<ItemGroup>
...
  <PackageReference Include="RimDev.AspNetCore.
FeatureFlags" Version="2.1.3" />
</ItemGroup>
```

Alternatywnie, aby odwołać się do pakietu NuGet w istniejącym projekcie, możemy wykonać następujące polecenie w wierszu poleceń terminala:

```
dotnet add package RimDev.AspNetCore.FeatureFlags
```

2. Następnie przejdziemy do pliku konfiguracyjnego `appsettings.json`, aby skonfigurować parametry połączenia z wcześniej utworzoną bazą danych za pomocą następującego kodu:

```
"connectionStrings": {
  "localDb": "Data Source=<your database
server>;Database=FeatureFlags.AspNetCore;User ID=<your
user>;Password=<password data>"
}
```

3. W pliku `startup.cs`, który znajduje się w katalogu głównym plików aplikacji, dodamy konfigurację dla `RimDev.FeatureFlags` za pomocą tego bloku kodu:

```
private readonly FeatureFlagOptions options;
public Startup(IConfiguration configuration)
{
```

```

    Configuration = configuration;
    options = new FeatureFlagOptions().UseCachedSqlFeatureProvider
    ↪(Configuration.GetConnectionString("localDb"));
}

```

W powyższym fragmencie kodu zainicjowaliśmy opcje `RimDev.FeatureFlags` przy użyciu połączenia z bazą danych. Możemy skonfigurować ładowanie usługi za pomocą następującego kodu:

```

public void ConfigureServices(IServiceCollection
services)
{
    ...
    services.AddFeatureFlags(options);
}
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    ...
    app.UseFeatureFlags(options);
    app.UseFeatureFlagsUI(options);
}

```

Zaraz po uruchomieniu aplikacji `RimDev` załaduje dane flagi funkcjonalności do kontekstu aplikacji. Dzięki temu skonfigurowaliśmy `RimDev.FeatureFlags` w naszym projekcie.

Teraz utworzymy flagi funkcjonalności i użyjemy ich w aplikacji. W tym przykładzie utworzymy flagę o nazwie `ShowBoxHome`, która może, ale nie musi, wyświetlać obrazek na środku strony głównej naszej aplikacji. Przyjrzyjmy się, jak manipulować tymi flagami funkcjonalności w naszym projekcie:

1. Najpierw utworzymy flagi funkcjonalności, tworząc nową klasę zawierającą następujący kod:

```

using RimDev.AspNetCore.FeatureFlags;
namespace appFeatureFlags.Models{
    public class ShowBoxHome : Feature {
        public override string Description { get;
    } = "Show the home center box.";
    }
}

```

Ta klasa zawiera flagę `ShowBowHome`. Do niej dodawany jest opis.

2. Następnie w naszym kontrolerze wywołujemy klasę `ShowBoxHome` z następującym kodem:

```

public class HomeController : Controller {
    private readonly ShowBoxHome showboxHome;
    public HomeController (ShowBoxHome showboxHome){
        this.showboxHome = showboxHome;
    }
}

```

```

    public IActionResult Index() {
        return View(new HomeModel { ShowboxHome = this.showboxHome.Value });
    }
    ...
}

```

Kontroler otrzymuje wartości flag funkcjonalności zapisanych w bazie danych, które zostały załadowane podczas uruchamiania aplikacji.

3. Utworzymy również klasę `HomeModel`, która będzie zawierać listę wszystkich flag potrzebnych do obsługi strony głównej:

```

public class HomeModel
{
    public bool ShowBoxHome { get; set; }
}

```

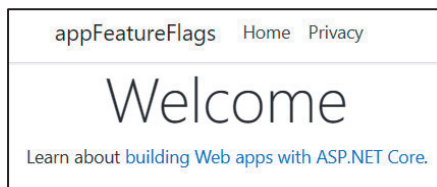
4. Na koniec w pliku `Views/Home/index.html` użyjemy tego modelu do wyświetlenia obrazka na środku strony głównej zależnie od wartości flagi funkcjonalności:

```

@if (Model.ShowBoxHome) {
    <div></div>
}

```

Gdy proces rozwoju aplikacji dobiegnie końca, wdróż i uruchom ją. Domyślnie na środku strony głównej nie ma obrazka, jak pokazano na poniższym rzucie ekranu:



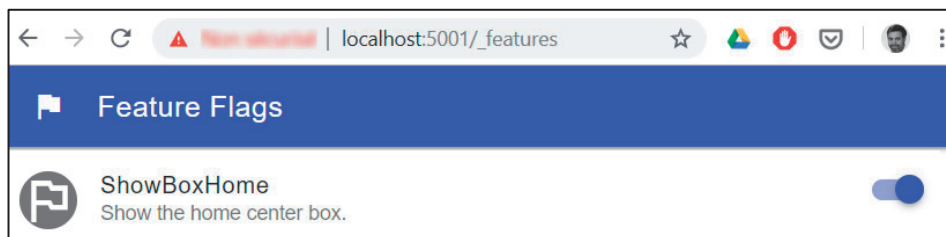
Rysunek 15.9. Aplikacja prezentująca wykorzystanie flagi funkcjonalności

Aby wyświetlić obrazek, musimy dynamicznie aktywować tę funkcjonalność:

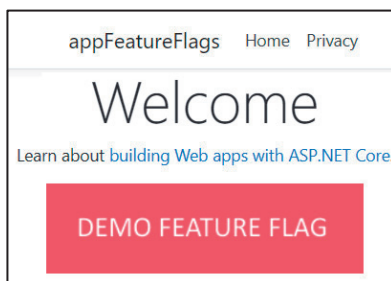
1. Przejdź do strony pod adresem `http://<twojawitryna>/_features`. Zobaczymy przełącznik o nazwie `ShowHomeBow`.
2. Aktywujemy flagi, włączając przełącznik, jak pokazano na rysunku 15.10.

Przeładuj stronę główną naszej aplikacji. Tutaj widzimy, że obraz jest wyświetlany na środku strony — rysunek 15.11.

Korzystając z frameworka `RimDev.FeatureFlags` i flag funkcjonalności, byliśmy w stanie włączyć lub wyłączyć funkcję naszej aplikacji bez konieczności jej ponownego wdrażania.



Rysunek 15.10. Prezentacja przełączania flagi funkcjonalności



Rysunek 15.11. Aplikacja prezentująca wykorzystanie flagi funkcjonalności

Pełny kod źródłowy tej aplikacji można znaleźć na stronie <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP15/appFF>.

Właśnie widzieliśmy, jak używać narzędzia typu open source do implementowania podstawowych flag funkcjonalności w aplikacji .NET Core. Zauważyliśmy, że za pomocą narzędzia open source możemy utworzyć bazę danych w naszej infrastrukturze. To użycie flag jest dość podstawowe, ponadto dostęp do interfejsu użytkownika do zarządzania nimi nie jest bezpieczny.

Wreszcie, podobnie jak w przypadku każdego narzędzia typu open source, ważne jest, aby sprawdzić, czy jest ono regularnie wspierane i aktualizowane przez jego twórców lub społeczność. Jednak korzystanie z narzędzi otwartoźródłowych do zarządzania flagami funkcjonalności pozostaje atrakcyjne i niedrogie w przypadku projektów dla małych firm.

Przyjrzyjmy się teraz kolejnemu rozwiązaniu narzędzia wykorzystującego flagi funkcjonalności, które polega na użyciu rozwiązania chmurowego PaaS. Przykładem takiego rozwiązania jest LaunchDarkly.

Korzystanie z narzędzia LaunchDarkly

W poprzedniej sekcji omówiliśmy używanie narzędzi open source wykorzystujących flagi funkcjonalności, które mogą być dobrym rozwiązaniem, ale wymagają pewnych składników infrastruktury i są zależne od języka programowania (w naszym przykładzie był to .NET Core).

Aby lepiej zarządzać flagami, możemy skorzystać z rozwiązania chmurowego, które nie wymaga implementacji architektury i zapewnia wiele funkcji wokół flag funkcjonalności.

Wśród tych rozwiązań (**oprogramowanie jako usługa** lub **SaaS**, ang. *software as a service*) znajduje się **LaunchDarkly** (<https://launchdarkly.com/>), które jest platformą SaaS składającą się z zaplecza do zarządzania flagami funkcjonalności i SDK. Umożliwia sterowanie flagami w naszych aplikacjach.

Biblioteki SDK LaunchDarkly są dostępne dla wielu języków programowania, takich jak .NET, JavaScript, Go i Java. Pełna lista SDK jest dostępna tutaj: <https://docs.launchdarkly.com/sdk>.

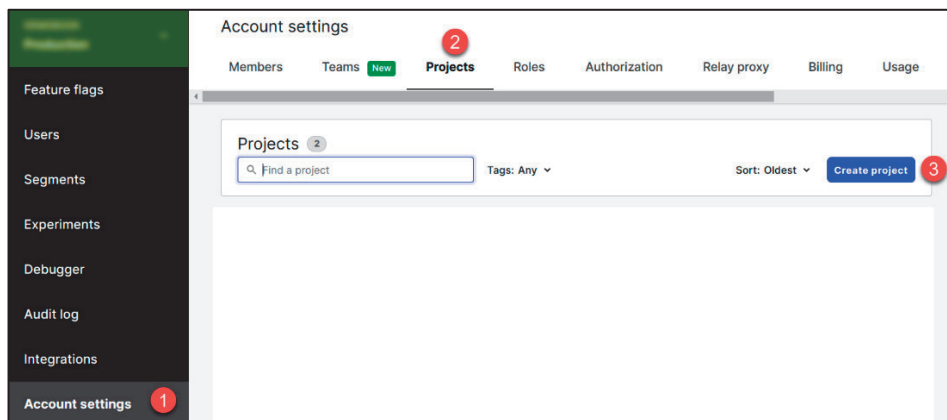
Oprócz klasycznej wersji zarządzania flagami funkcjonalności za pomocą `RimDev.FeatureFlags` LaunchDarkly umożliwia zarządzanie flagami przez użytkownika, a także zapewnia funkcje testowania A/B — w połączeniu z flagami funkcjonalności. Testy A/B mogą mierzyć wykorzystanie funkcji aplikacji za pomocą flag funkcjonalności.

Jednak LaunchDarkly jest rozwiązaniem płatnym (<https://launchdarkly.com/pricing/>). Na szczęście dostępna jest 30-dniowa wersja próbna. Dzięki temu możemy je przetestować. Przyjrzyj się, jak używać LaunchDarkly, aby móc zaimplementować flagi funkcjonalności w aplikacji .NET.

W tym celu zaczniemy od utworzenia następujących flag:

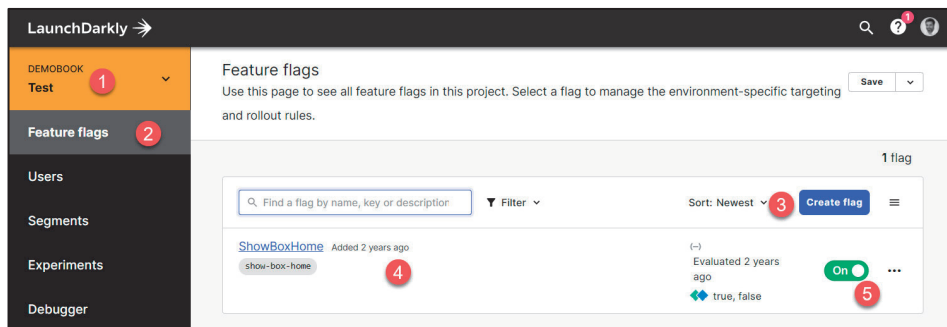
1. Najpierw zaloguj się na swoje konto LaunchDarkly, klikając przycisk *Sign In*, który znajduje się w górnym menu strony LaunchDarkly. Alternatywnie możesz przejść pod adres <https://app.launchdarkly.com/>.
2. Po połączeniu się z naszym kontem w sekcji *Account settings* możemy utworzyć nowy projekt o nazwie **DemoBook**, jak pokazano na rysunku 15.12.

Domyślnie w projekcie tworzone są dwa środowiska. Będziemy mogli tworzyć własne środowiska, a następnie testować flagi funkcjonalności w różnych środowiskach. Dodatkowo każde z tych środowisk ma unikalny klucz SDK, który będzie służył jako uwierzytelnienie dla SDK.



Rysunek 15.12. Tworzenie projektu LaunchDarkly

3. W środowisku o nazwie *Test* przechodzimy do menu *Feature flags*, klikamy przycisk *Create flag* i tworzymy flagę funkcjonalności o nazwie *ShowBoxHome*, jak pokazano na poniższym zrzucie ekranu:



Rysunek 15.13. Tworzenie flagi funkcjonalności LaunchDarkly

Po utworzeniu flagi możemy ją aktywować, klikając przełącznik *On/Off*.

Teraz, gdy mamy skonfigurowaną i utworzoną flagę w portalu LaunchDarkly, zobaczymy, jak wykorzystać SDK w kodzie aplikacji.

W LaunchDarkly zmiany wprowadzane do flag funkcjonalności są wykonywane przez użytkowników podłączonych do aplikacji. Oznacza to, że aplikacja musi zapewniać system uwierzytelniania.

Aby użyć zestawu SDK i uruchomić aplikację, wykonaj następujące kroki:

1. Pierwszym krokiem jest wybór SDK odpowiadającego językowi tworzenia aplikacji. Możemy to zrobić, przechodząc do <https://docs.launchdarkly.com/docs/getting-started-with-launchdarkly-sdks#section-supported-sdks>. W naszym przypadku mamy aplikację .NET, więc postępujemy zgodnie z tą procedurą: <https://docs.launchdarkly.com/docs/dotnet-sdk-reference>.
2. Teraz zintegrujemy odwołanie do pakietu NuGet *LaunchDarkly.ServerSdk* (<https://www.nuget.org/packages/LaunchDarkly.ServerSdk/>) w pliku *.csproj* naszej aplikacji, dodając go do pakietów referencyjnych:

```
<ItemGroup>
  <PackageReference Include="LaunchDarkly.ServerSdk" Version="6.3.1" />
  ...
</ItemGroup>
```

3. W kodzie .NET wykonujemy to w kontrolerze. Aby to zrobić, musimy zaimportować SDK za pomocą polecenia `using` w następujący sposób:


```
using LaunchDarkly.Client;
```
4. W kodzie kontrolera dodajemy połączenie z LaunchDarkly wywołane przez `FeatureFlag`. Kod jest zilustrowany w następującym fragmencie:

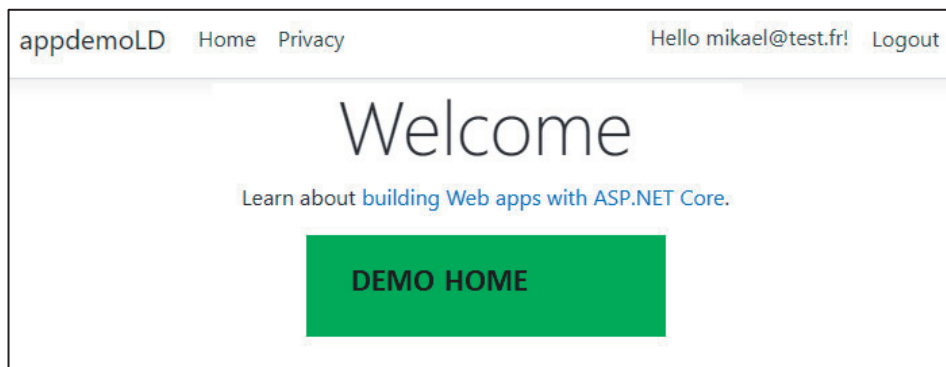
```
public IActionResult Index() {
    LdClient ldClient = new LdClient("sdk-eb0443dc-xxxx-xxxxx-xxx");
    User user = LaunchDarkly.Client.User.WithKey(User.Identity.Name);
    bool showBoxHome = ldClient.BoolVariation("show-box-home", user, false);
    return View(new HomeModel { ShowBoxHome = showBoxHome });
}
```

Aby połączyć się z LaunchDarkly, musimy użyć klucza SDK, który został dostarczony podczas tworzenia projektu. Następnie w powyższym kodzie dodajemy użytkownika połączony z aplikacją z flagą funkcjonalności, którą utworzyliśmy wcześniej w portalu.

5. W widoku *Home/Index.html* dodajemy następujący kod, aby wstawić warunek, który wyświetli obraz zależnie od wartości flagi funkcjonalności:

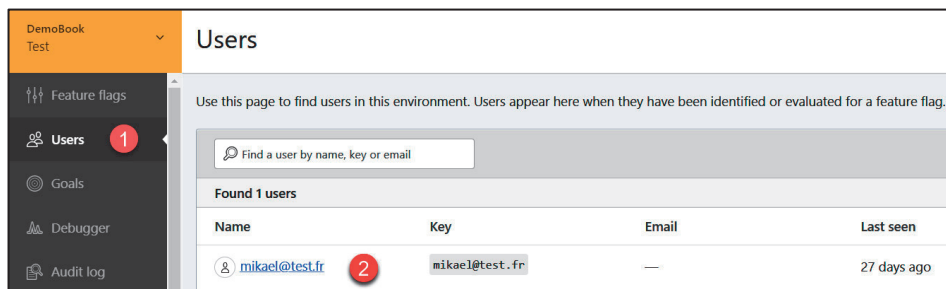
```
<div class="text-center">
  @if (Model.ShowBoxHome) {
    <div></div>
  }
</div>
```

6. Na końcu wdrażamy i uruchamiamy aplikację. Jak widać na poniższym zrzucie ekranu, strona główna pokazuje obrazek, ponieważ flagi funkcjonalności są domyślnie ustawione na `true`:



Rysunek 15.14. Aplikacja prezentująca wykorzystanie flagi funkcjonalności w LaunchDarkly

7. Następnie przechodzimy do portalu LaunchDarkly i modyfikujemy konfigurację flagi funkcjonalności dla bieżącego użytkownika na wartość `false`. Na stronie zarządzania *Users* wybieramy użytkownika, jak pokazano na poniższym zrzucie ekranu:



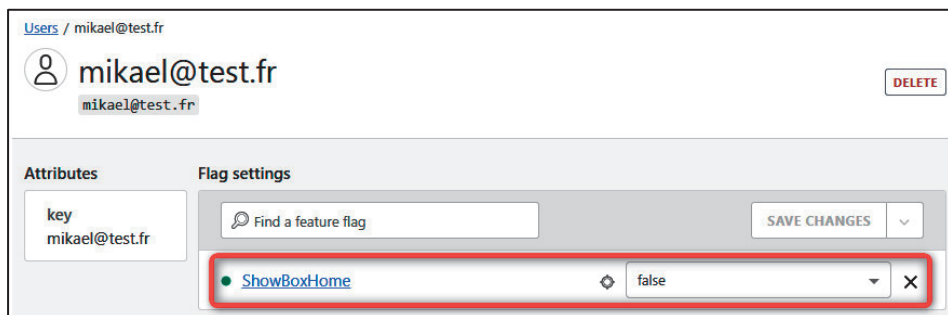
Rysunek 15.15. Użytkownik LaunchDarkly

Następnie aktualizujemy wartość flagi funkcji na `false`, jak pokazano na rysunku 15.16.

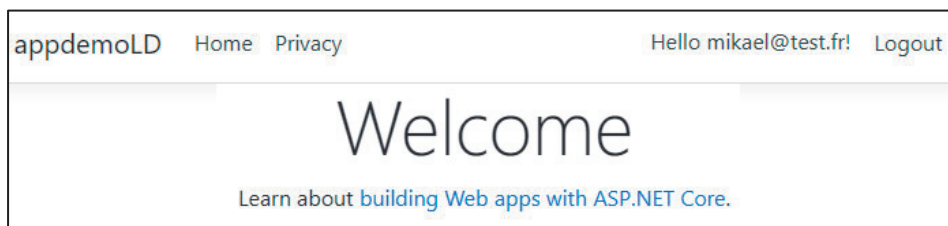
8. Po ponownym załadowaniu strony widzimy, że centralny obrazek nie jest już wyświetlany, jak pokazano na rysunku 15.17.

Jest to przykład wykorzystania narzędzia LaunchDarkly. Ma ono wiele innych ciekawych funkcji, takich jak system zarządzania użytkownikami, wykorzystanie funkcji z testami A/B, integracja z platformami CI/CD i raportowanie.

W tej sekcji omówiliśmy narzędzie LaunchDarkly, które jest platformą chmurową wykorzystującą flagi funkcjonalności. Zbadaliśmy jego implementację w aplikacji internetowej, tworząc flagi w portalu LaunchDarkly.



Rysunek 15.16. LaunchDarkly — ustawienia flagi funkcjonalności dla użytkownika



Rysunek 15.17. Aplikacja prezentująca flagi funkcjonalności w LaunchDarkly

Uwaga

Pełne źródła kodu dla tej aplikacji można znaleźć pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP15/appdemoLD>.

Następnie zmieniliśmy wartość flagi w kodzie aplikacji za pomocą biblioteki SDK dostarczonej przez LaunchDarkly. Wreszcie w portalu LaunchDarkly włączyliśmy/wyłączyliśmy funkcję dla użytkownika, który chce przetestować nową funkcjonalność aplikacji bez konieczności jej ponownego wdrażania.

Podsumowanie

W tym rozdziale skupiliśmy się na ulepszaniu wdrożeń produkcyjnych. Zaczęliśmy od zastosowania Terraform w celu skrócenia przestojów podczas udostępniania i usuwania zasobów.

Następnie skupiliśmy się na praktyce wdrażania zielono-niebieskiego i jego wzorcach, takich jak Canary release i Dark launch. Przyjrzelśmy się implementacji architektury wdrażania zielono-niebieskiego na platformie Azure przy użyciu usług App Service i Azure Traffic Manager.

Na koniec szczegółowo omówiliśmy implementację flag funkcjonalności w aplikacji .NET przy użyciu dwóch rodzajów narzędzi: `RimDev.FeatureFlags`, które jest narzędziem typu open source i oferuje podstawowy system flag funkcjonalności, i `LaunchDarkly`, które jest oparte na rozwiązaniu chmurowym. Nie jest bezpłatne, ale zapewnia kompletne i zaawansowane zarządzanie flagami funkcjonalności.

Następny rozdział jest poświęcony GitHubowi. Przyjrzymy się najlepszym praktykom w zakresie wkładu w projekty open source.

Pytania

1. Z jakiej opcji w Terraform możemy skorzystać, aby skrócić przestoje?
2. Z czego składa się zielono-niebieska infrastruktura wdrożeniowa?
3. Jakie są dwa zielono-niebieskie wzorce wdrażania omówione w tym rozdziale?
4. Jakie komponenty na platformie Azure pozwalają nam zastosować zielono-niebieską praktykę wdrożeniową?
5. Jaka jest rola flag funkcjonalności?
6. Czym jest narzędzie `RimDev.FeatureFlags`?
7. Które narzędzie stosujące flagi funkcjonalności omówione w tym rozdziale jest rozwiązaniem SaaS?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej na temat redukcji przestojów i praktyk wdrożeniowych, zapoznaj się z następującymi zasobami:

- *Zero Downtime Updates with HashiCorp Terraform* — <https://www.hashicorp.com/blog/zero-downtime-updates-with-terraform>.
- *BlueGreenDeployment* Martina Fowlera — <https://martinfowler.com/bliki/BlueGreenDeployment.html>.
- *Feature Toggles (aka Feature Flags)* Martina Fowlera — <https://martinfowler.com/articles/feature-toggles.html>.
- Przewodnik po flagach funkcjonalności — <http://featureflags.io/>.
- Przypadki użycia flagi funkcjonalności w LaunchDarkly — <https://launchdarkly.com/use-cases/>.

DevOps dla projektów open source

Rozdział

16

Jeszcze kilka lat temu praktyka open source, polegająca na publicznym udostępnianiu kodu źródłowego produktu, była zasadniczo używana tylko przez społeczność Linuksa. W miarę upływu czasu, wraz z pojawieniem się GitHuba, zaszło wiele zmian dotyczących open source. Microsoft utworzył wiele swoich produktów typu open source i jest również jednym z największych dostawców dla GitHuba.

Dzisiaj open source jest koniecznością w świecie deweloperskim i korporacyjnym, niezależnie od tego, czy chcemy korzystać z projektu, czy wnieść do niego swój wkład.

Jednak aplikacje open source nie zawsze są darmowe. Czasami obowiązują opłaty licencyjne za wtyczki, wsparcie lub funkcje dla firm. Ponadto, jeśli chodzi o wsparcie produktu, korzystanie z oprogramowania open source może czasami stwarzać trudności i pułapki.

W tej książce widzieliśmy wiele przypadków użycia narzędzi otwartoźródłowych, takich jak Terraform, Ansible, Packer, Vagrant, Jenkins i SonarQube. Jedną z wielkich zalet open source jest nie tylko korzystanie z produktów, ale także fakt, że możemy się przyczynić do ich rozwoju.

Aby brać udział w projekcie open source, musimy uczestniczyć w jego ewolucji poprzez omawianie problemów podczas jego używania lub zgłaszać sugestie dotyczące jego ulepszenia. Ponadto, jeśli jesteś programistą, możesz również modyfikować jego kod źródłowy.

Wreszcie jako programista lub członek zespołu operacyjnego możesz udostępnić społeczności Twój projekt open source.

W tym rozdziale omówimy temat wsparcia dla open source i wyjaśnimy, dlaczego ważne jest stosowanie praktyk DevOps we wszystkich takich projektach.

Wszystkie te praktyki, takie jak korzystanie z Gita, potoku CI/CD i analiza bezpieczeństwa, zostały już omówione w książce. W tym rozdziale skupimy się bardziej na tym, jak je zastosować w kontekście własnego projektu open source.

Zacznijmy od nauczenia się, jak udostępnić kod projektu w GitHubie i jak zainicjować wsparcie dla innego projektu. Opowiemy również, jak zarządzać żądaniami pobierania kodu (ang. *pull*), co jest jedną z najważniejszych cech wsparcia. Ponadto zobaczymy, jak wskazać zmiany wersji za pomocą informacji o wersji aplikacji (ang. *release notes*), i omówimy temat udostępniania plików binarnych w wydaniach GitHuba. Wyjaśnimy koncept GitHub Actions, który pozwala nam tworzyć potoki CI/CD w projektach open source hostowanych w GitHubie. Na koniec przyjrzymy się analizie kodu źródłowego projektów open source. Zrobimy to za pomocą SonarCloud, który jest używany do statycznej analizy kodu, oraz WhiteSource Bolt, który jest używany do analizowania luk w zabezpieczeniach pakietów zawartych w projekcie open source.

W tym rozdziale omówimy następujące tematy:

- przechowywanie kodu źródłowego w GitHubie,
- przyczynianie się do rozwoju projektów open source przy użyciu żądań pobierania,
- zarządzanie plikiem dziennika zmian i informacjami o wydaniu,
- udostępnianie plików binarnych w wydaniach GitHuba,
- wprowadzenie do GitHub Actions,
- analiza kodu za pomocą narzędzia SonarCloud,
- wykrywanie luk w zabezpieczeniach za pomocą narzędzia WhiteSource Bolt.

Wymagania techniczne

W tym rozdziale wykorzystamy GitHuba jako platformę repozytorium Gita do przechowywania naszego projektu open source. Dlatego będziesz potrzebować konta GitHuba, które możesz założyć bezpłatnie na stronie <https://github.com/>. Aby w pełni zrozumieć praktyki DevOps, które zostaną użyte w tym rozdziale, powinieneś dobrze zapoznać się z następującymi rozdziałami tej książki:

- Rozdział 6., „Zarządzanie kodem źródłowym za pomocą Gita”.
- Rozdział 7., „Ciągła integracja i ciągłe wdrażanie”.
- Rozdział 12., „Statyczna analiza kodu za pomocą SonarQube”.

Cały kod zawarty w tym rozdziale można znaleźć pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP16/>.

Obejrzyj poniższy film na kanale Code in Action: <https://bit.ly/3JJCb9V>.

Przechowywanie kodu źródłowego w GitHubie

Jeśli chcemy udostępnić jeden z naszych projektów jako open source, musimy wersjonować jego kod na platformie Gita, która udostępnia następujące właściwości:

- Repozytoria publiczne; czyli musimy mieć dostęp do kodu źródłowego zawartego w tym repozytorium, ale bez konieczności uwierzytelnienia na platformie Git.
- Funkcje i narzędzia do współpracy przy kodzie między różnymi członkami tej platformy.

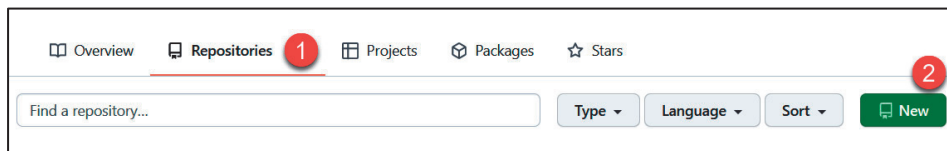
Dwie główne platformy pozwalają nam hostować narzędzia open source: GitLab, któremu przyjrzelśmy się w sekcji „Korzystanie z GitLab CI” w rozdziale 7., „Ciągła integracja i ciągłe wdrażanie”, oraz GitHub, który jest obecnie najczęściej używaną platformą dla projektów open source.

Nauczmy się, jak korzystać z GitHuba, abyśmy mogli hostować nasz projekt lub przyczynić się do rozwoju innego projektu.

Tworzenie nowego repozytorium na GitHubie

Jeśli chcemy hostować nasz projekt na GitHubie, musimy utworzyć repozytorium. Wykonaj następujące kroki, aby to zrobić:

1. Najpierw zaloguj się na swoje konto GitHuba lub utwórz nowe, jeśli jesteś nowym użytkownikiem, przechodząc pod adres <https://github.com/>.
2. Po zalogowaniu przejdź do zakładki *Repositories* w swoim koncie. Kliknij przycisk *New*, jak pokazano na poniższym zrzucie ekranu:



Rysunek 16.1. Dodawanie repozytorium

3. Utwórz nowy formularz dla repozytorium. Można go wypełnić, tak jak pokazano na poniższym zrzucie ekranu:

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner **Repository name ***

mikaelkrief / DemoApp ✓

Great repository names are short and memorable. Need inspiration? How about [cuddly-memory](#)?

Description (optional)

Demo for OSS project

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer.

Add .gitignore: **None** | Add a license: **None** ⓘ

Create repository

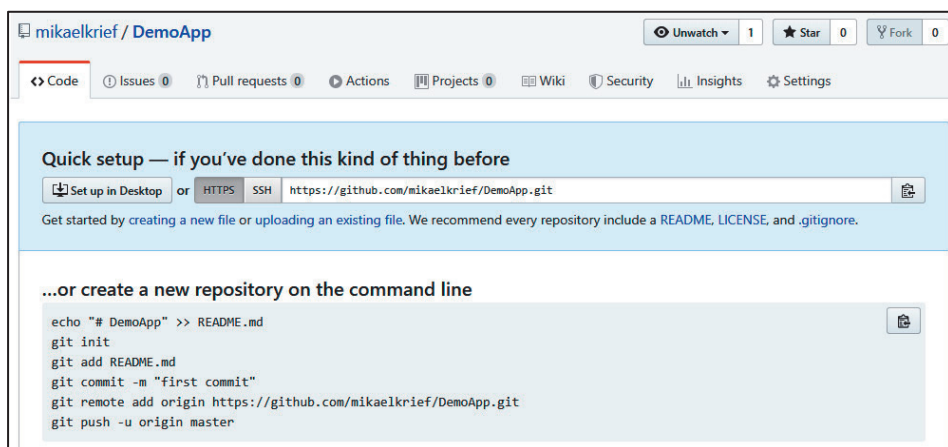
Rysunek 16.2. Szczegóły repozytorium GitHuba

Informacje, które należy wypełnić, aby utworzyć repozytorium, są następujące:

- Nazwa repozytorium.
- Opis (który jest opcjonalny).
- Musimy określić, czy repozytorium jest publiczne (ang. *Public*) (dostępne dla wszystkich, nawet jeśli nie ma uwierzytelnienia), czy prywatne (ang. *Private*) (dostępne tylko dla członków, którym dajemy dostęp).
- Możemy również zainicjować repozytorium pustym plikiem *README.md* lub plikiem *.gitignore*.

Następnie zatwierdź formularz, klikając przycisk *Create repository*.

4. Gdy tylko repozytorium zostanie utworzone, strona główna wyświetli pierwsze instrukcje, abyś mógł rozpocząć archiwizację kodu. Poniższy zrzut ekranu pokazuje część instrukcji dla nowego repozytorium GitHuba:



Rysunek 16.3. Repozytorium GitHuba — pierwsze instrukcje

Wszystko jest gotowe do archiwizacji kodu w GitHubie. Zrób to, korzystając z przepływu pracy i poleceń Gita, które omówiliśmy w rozdziale 6., „Zarządzanie kodem źródłowym za pomocą Gita”.

Ta procedura obowiązuje przy tworzeniu repozytorium w GitHubie. Teraz nauczymy się, jak współtworzyć projekt GitHuba, używając projektu z innego repozytorium.

Przyczynianie się do rozwoju projektu w GitHubie

Właśnie nauczyliśmy się tworzyć repozytorium na GitHubie. Jednak musimy wiedzieć, że domyślnie tylko właściciel repozytorium może modyfikować kod tego repozytorium.

Uwaga

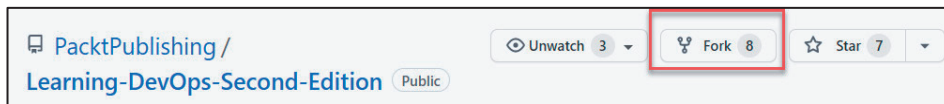
Możemy dodawać osoby jako współpracowników do tego repozytorium, by mogli wprowadzać zmiany w kodzie. Aby uzyskać więcej informacji na temat tej procedury, przejdź do <https://help.github.com/en/articles/inviting-collaborators-to-a-personal-repository>.

Zgodnie z tym nie mamy praw do modyfikacji kodu innego repozytorium.

Aby przyczynić się do rozwoju kodu innego repozytorium, będziemy musieli utworzyć **rozwidlenie** (ang. *fork*) początkowego repozytorium, nad którym chcemy pracować. Fork to powielenie początkowego repozytorium, które jest tworzone na naszym koncie GitHuba.

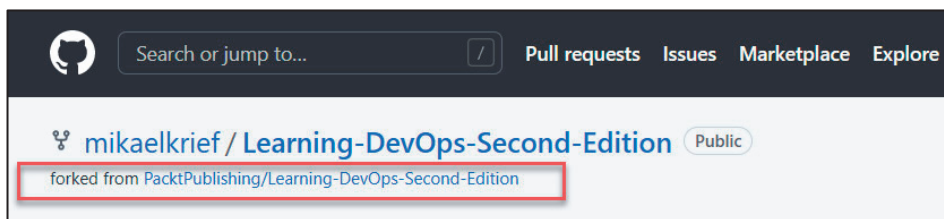
Wykonaj następujące kroki, aby się dowiedzieć, jak utworzyć rozwidlenie repozytorium:

1. Najpierw przejdź do początkowego repozytorium, do rozwoju którego chcesz się przyczynić. Następnie kliknij przycisk *Fork* u góry strony, jak pokazano na poniższym zrzucie ekranu:



Rysunek 16.4. GitHub — rozwidlanie repozytorium

2. Po kilku sekundach to repozytorium zostanie rozwidlone i zduplikowane wraz z całą zawartością na Twoim koncie. W ten sposób otrzymujesz nowe repozytorium na swoim koncie, które jest połączone z pierwotnym repozytorium, jak pokazano na poniższym zrzucie ekranu:



Rysunek 16.5. Link do rozwidlenia GitHuba

3. Teraz masz dokładną kopię repozytorium, które chcesz rozwijać, na swoim koncie GitHuba. Możesz dowolnie modyfikować kod i zatwierdzać zmiany, a wszystkie zostaną zarchiwizowane w Twoim repozytorium.

Jednak nawet jeśli istnieje powiązanie między początkowym repozytorium a rozwidleniem, kod każdego repozytorium jest całkowicie nieskorelowany i nie jest synchronizowany automatycznie.

W tej sekcji nauczyliśmy się, jak utworzyć repozytorium GitHuba lub rozwidlenie innego repozytorium, abyśmy mogli wnieść do niego swój wkład. Teraz nauczymy się proponować zmiany w kodzie i scalać nasz kod z innym repozytorium za pomocą żądania pobrania (ang. *pull request*).

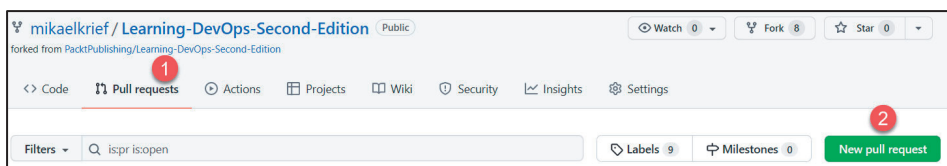
Przyczynianie się do rozwoju projektów open source przy użyciu żądań pobierania

Gdy chcemy współtworzyć projekt open source w GitHubie, musimy wprowadzić zmiany w kodzie źródłowym aplikacji, który znajduje się w repozytorium naszego konta GitHuba. Aby scalić te zmiany kodu z początkowym repozytorium, musimy wykonać operację **scalania** (ang. *merge*).

W GitHubie istnieje element zwany *pull request*, który pozwala nam wykonać operację scalania między repozytoriami. Oprócz wykonywania prostego i klasycznego scalania gałęzi kodu żądanie pobierania (ang. *pull request*) dodaje również zupełnie nowy aspekt współpracy, udostępniając funkcje, które pozwalają różnym współautorom omawiać zmiany w kodzie.

Nauczmy się, jak wykonać operację *pull request*:

1. Po dokonaniu zmian w źródle kodu w repozytorium na swoim koncie należy zarchiwizować te zmiany poprzez dokonanie zatwierdzenia (ang. *commit*). Wprowadzone zmiany są teraz gotowe do połączenia ze zdalnym repozytorium. Aby to zrobić, przejdź do swojego repozytorium, do zakładki *Pull requests* i kliknij przycisk *New pull request*, jak pokazano na poniższym zrzucie ekranu:

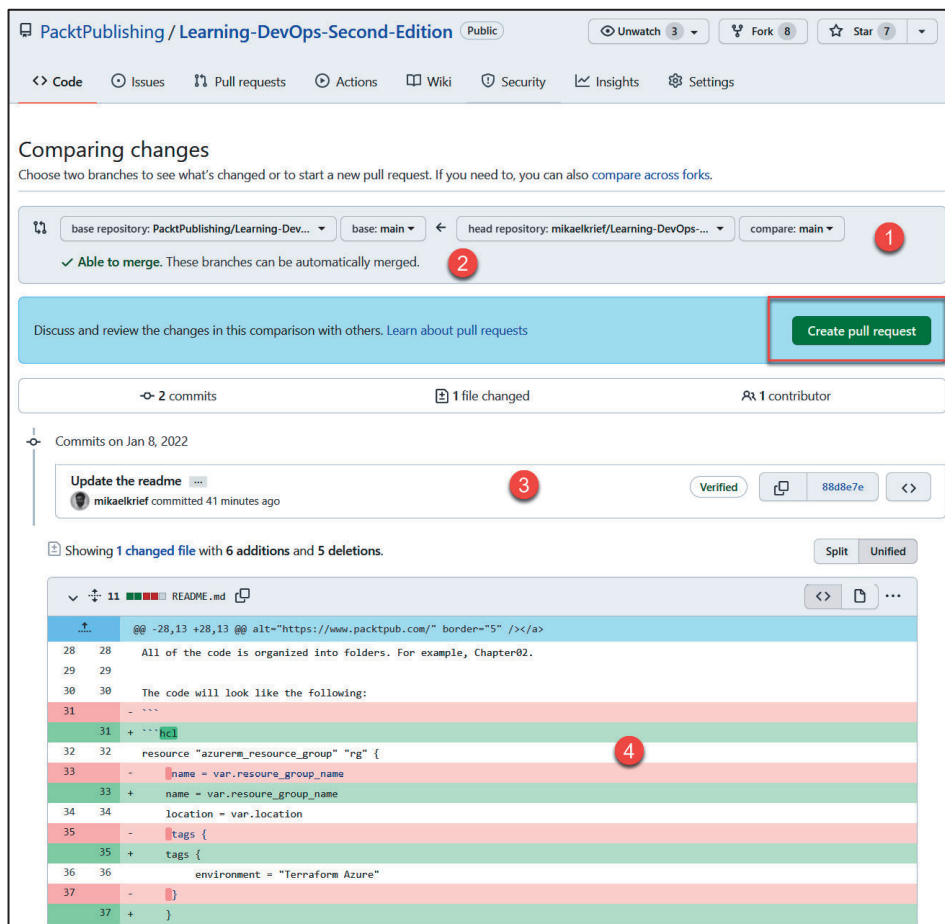


Rysunek 16.6. GitHub — New pull request

2. Strona, która się pojawi, zawiera wszystkie informacje dotyczące żądania pobierania, które zostanie utworzone, jak pokazano na poniższym zrzucie ekranu.

Na ekranie wyświetlane są następujące informacje:

- Źródłowe repozytorium/gałęzie i docelowe repozytorium/gałęzie.
- Wskaźnik pokazujący, czy występują konflikty kodu.
- Lista zatwierdzeń zawartych w tym żądaniu pobierania.
- Różnice w kodzie dla zmodyfikowanych plików.

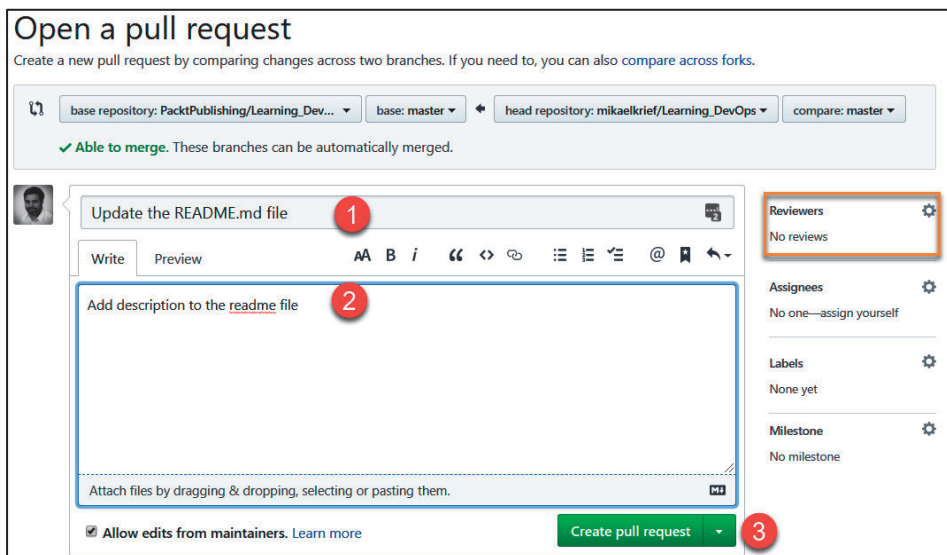


Rysunek 16.7. Szczegóły żądania pobierania w GitHubie

3. Aby zweryfikować utworzenie żądania pobierania, kliknij przycisk *Create pull request*.
4. Wprowadź nazwę i opis żądania pobierania w wyświetlonym formularzu, jak pokazano na rysunku 16.8.

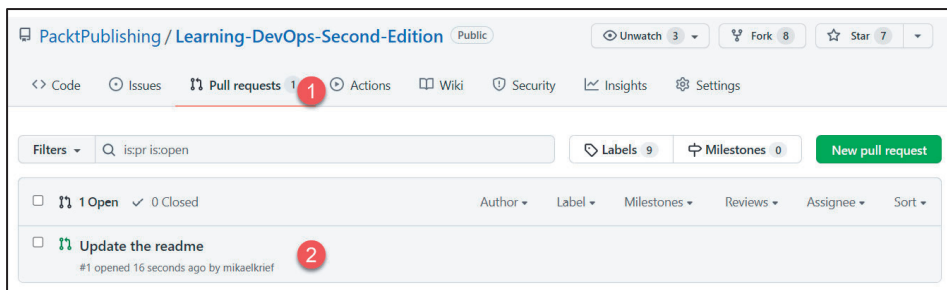
Ta informacja jest ważna, bo pomoże docelowemu właścicielowi repozytorium szybko zrozumieć cel zmiany kodu. Dodatkowo z prawego panelu można wybrać recenzentów, którzy zostaną powiadomieni e-mailem o wykonaniu operacji *pull request*. Dokona tego osoba odpowiedzialna za przeglądanie zmian w kodzie i ich zatwierdzanie lub odrzucanie.

5. Na końcu potwierdź utworzenie żądania pobierania, klikając przycisk *Create pull request*.



Rysunek 16.8. Tytuł i opis żądania pobierania w GitHubie

Po utworzeniu żądania właściciel oryginalnego repozytorium zobaczy, że nowe żądanie pobierania zostało otwarte (z podanym tytułem) w zakładce *Pull requests* w jego repozytorium. Może tam sprawdzić wszystkie szczegóły żądania.

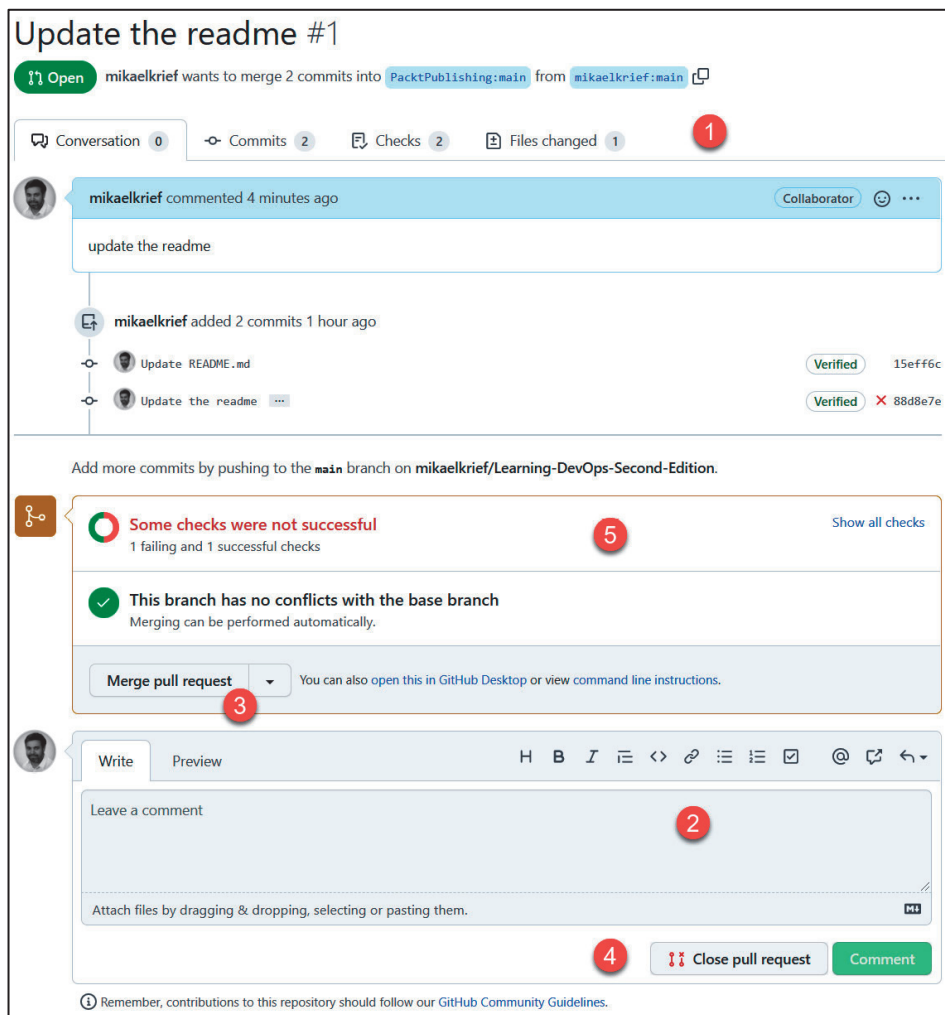


Rysunek 16.9. Lista żądań pobierania w GitHubie

Na poniższym zrzucie ekranu możemy zobaczyć różne opcje, które zostały zaproponowane dla tego żądania pobrania.

Poniżej przedstawiono różne operacje, które właściciel repozytorium może wykonać dla tego żądania pobrania:

- Klikając zakładkę *File changed*, recenzent może zobaczyć zmiany, które zostały wprowadzone w kodzie, i zostawić notatki na temat każdej ze zmodyfikowanych linii.



Rysunek 16.10. Weryfikacja żądania pobrania w GitHubie

- Recenzent lub właściciel repozytorium może zainicjować dyskusję na temat zmian w kodzie i kliknąć przycisk *Comment*, aby zatwierdzić swoje uwagi.
- Jeśli właściciel jest zadowolony ze zmian w kodzie, może kliknąć przycisk *Merge pull request*, aby przeprowadzić scalanie.
- Z drugiej strony, jeśli właściciel nie jest usatysfakcjonowany i odrzuca żądanie pobrania, może kliknąć przycisk *Close pull request* i żądanie zostaje zamknięte.
- Możemy również przeprowadzać pewne automatyczne kontrole.

Po scaleniu żądanie pobrania będzie miało status *Merged*, a kod oryginalnego repozytorium zostanie zaktualizowany o wprowadzone przez nas zmiany w kodzie.

W tej sekcji widzieliśmy, że dzięki żądaniu *pull request* mamy prosty sposób na wniesienie wkładu w projekt open source w GitHubie poprzez zaproponowanie zmian w kodzie. Następnie właściciel projektu może albo zaakceptować to żądanie ściągnięcia i scalić kod, albo odrzucić zmiany.

W następnej sekcji dowiemy się, jak zarządzać zmianami, które wprowadziliśmy w naszym projekcie, za pomocą pliku dziennika zmian.

Zarządzanie plikiem dziennika zmian i informacjami o wydaniu

Gdy udostępniamy projekt jako open source, dobrą praktyką jest informowanie użytkowników o zmianach, które są w nim stosowane, gdy tylko się pojawiają. Ten system rejestrowania zmian (znany też jako *release notes*) jest tym ważniejszy, jeśli oprócz kodu źródłowego nasze repozytorium udostępnia publicznie również plik binarny aplikacji, ponieważ użycie tego pliku binarnego jest uzależnione od jego różnych wersji i zmian kodu.

Logicznie rzecz biorąc, moglibyśmy znaleźć historię zmian w kodzie, przeglądając historię pobrań w Gicie. Byłoby to jednak zbyt żmudne i czasochłonne dla nowicjuszy. Z tych powodów zmianę historii wskażemy wersjami kodu w pliku tekstowym, który każdy może przeczytać. Ten plik nie ma ustalonej nomenklatury ani formalizmu, ale dla uproszczenia zdecydowaliśmy się nazwać go *CHANGELOG.md*.

Tak więc ten plik dziennika zmian jest plikiem tekstowym w formacie języka Markdown, który jest łatwy do edycji dzięki prostemu formatowaniu i jest umieszczony w katalogu głównym repozytorium. W tym pliku historia zmian jest przedstawiona w formie listy i nie zawiera zbyt wielu szczegółów dotyczących każdej zmiany.

Aby uzyskać lepszą przejrzystość, historia zostanie uporządkowana od najnowszych do najstarszych zmian, abyśmy mogli szybko uzyskać dostęp do tych aktualnych. Żeby dać wyobrażenie o kształcie pliku dziennika zmian, pokazujemy zrzut ekranu przedstawiający wyciąg z pliku dziennika zmian dostawcy Terraform dla platformy Azure — rysunek 16.11.

Ważną informacją, o której należy wspomnieć w tym pliku, jest historia wersji zmian wprowadzonych w tej aplikacji. Dla każdej wersji sporządzamy listę dostarczonych nowych funkcji, ulepszeń i poprawek błędów.

1.30.1 (June 07, 2019)

BUG FIXES:

- Ensuring the authorization header is set for calls to the User Assigned Identity API's (#3613)

1.30.0 (June 07, 2019)

FEATURES:

- New Data Source: `azurerm_redis_cache` (#3481)
- New Data Source: `azurerm_sql_server` (#3513)
- New Data Source: `azurerm_virtual_network_gateway_connection` (#3571)

IMPROVEMENTS:

- dependencies: upgrading to Go 1.12 (#3525)
- dependencies: upgrading the `storage` SDK to `2019-04-01` (#3578)
- Data Source `azurerm_app_service` - support windows containers (#3566)
- Data Source `azurerm_app_service_plan` - support windows containers (#3566)
- `azurerm_api_management` - rename `disable_triple_des_chipers` to `disable_triple_des_ciphers` (#3539)
- `azurerm_application_gateway` - support for the value `General` in the `rule_group_name` field within the `disabled_rule_group` block (#3533)
- `azurerm_app_service` - support for windows containers (#3566)
- `azurerm_app_service_plan` - support for the `maximum_elastic_worker_count` property (#3547)
- `azurerm_managed_disk` - support for the `create_option` of `Restore` (#3598)
- `azurerm_app_service_plan` - support for windows containers (#3566)

Rysunek 16.11. Przykład dziennika zmian w GitHubie

Uwaga

Pełna zawartość tego pliku jest dostępna tutaj: <https://github.com/terraform-providers/terraform-provider-azurerm/blob/master/CHANGELOG.md>.

Dla każdej zmiany jest bardzo krótki opis, a numer zatwierdzenia jest przypisany jako link, który pozwala nam zobaczyć wszystkie szczegóły zmian — poprzez kliknięcie go.

Uwaga

Aby uzyskać szczegółowe informacje na temat formatu pliku dziennika zmian, zapoznaj się z następującą dokumentacją: <https://keepachangelog.com/en/1.1.0/>.

Wreszcie w celu integracji z procesem DevOps możliwe jest również automatyczne generowanie pliku dziennika zmian przy użyciu zatwierdzeń i tagów.

Wiele skryptów i narzędzi umożliwia generowanie dziennika zmian, np. konta GitHuba (<https://github.com/conventional-changelog>). Jeśli jednak nie jesteś pewien, czy powinien napisać, czy wygenerować ten plik, oto bardzo interesujący artykuł wyjaśniający zalety i wady tych dwóch metod: <https://depfu.com/blog/changelogs-to-write-or-to-generate>.

W tej sekcji nauczyliśmy się, jak informować użytkowników i współpracowników o historii zmian w kodzie, które są wprowadzane w projektach open source, za pomocą pliku dziennika zmian. Następnie przyjrzelśmy się przydatnym informacjom, które powinniśmy umieścić w tym pliku dziennika, aby użytkownicy mogli dokładnie dowiedzieć się, jakie zmiany zachodzą w aplikacji między poszczególnymi wersjami.

W następnej sekcji nauczymy się, jak udostępniać pliki binarne w projekcie open source w wydaniach GitHuba.

Udostępnianie plików binarnych w wydaniach GitHuba

Celem projektu open source jest nie tylko odsłonięcie kodu źródłowego projektu, lecz także udostępnienie go użytkownikom publicznym. Dla każdej nowej wersji projektu (zwanej wydaniem) ten udział zawiera informację o wydaniu, a także plik binarny powstały w wyniku kompilacji projektu.

W ten sposób użytkownik, który chce korzystać z tej aplikacji, nie musi pobierać całego kodu źródłowego i go kompilować — wystarczy, że pobierze udostępniony plik binarny dla żądanej wersji i użyje go bezpośrednio.

Zauważ, że wydanie jest skojarzone z tagiem Gita, który służy do umieszczania etykiety w określonym punkcie w historii kodu źródłowego. Tag jest często używany do podania numeru wersji kodu źródłowego; np. tagiem może być wartość v1.0.1.

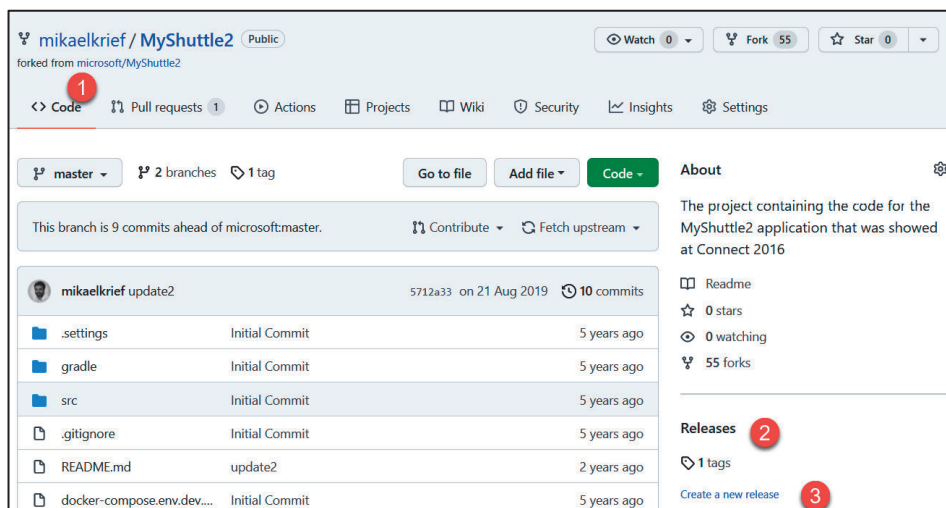
Uwaga

Aby dowiedzieć się więcej na temat używania tagów w Gicie, przeczytaj następującą dokumentację: <https://git-scm.com/book/en/v2/Git-Basics-Tagging>.

W GitHubie, w każdym repozytorium, istnieje możliwość publikowania wydań za pomocą tagów, które będą zawierały numer wersji (pozyskany z tagu Gita), opis określający listę zmian i pliki binarne aplikacji.

Po tym wprowadzeniu do wydań nauczmy się, jak utworzyć wydanie w GitHubie za pomocą jego interfejsu webowego:

1. Aby utworzyć wydanie, przejdź do repozytorium zawierającego kod aplikacji.
2. Kliknij link *Releases* w prawym panelu, który można znaleźć na karcie *Code*, jak pokazano na poniższym zrzucie ekranu:



Rysunek 16.12. GitHub — link do wydania

3. Na następnej stronie, która się pojawi, kliknij przycisk *Create a new release*, aby utworzyć nową wersję. Zostanie wyświetlony formularz z wypełnionymi informacjami na temat wydania, jak pokazano na rysunku 16.13.

W tym formularzu wprowadziliśmy następujące informacje:

- Etykieta powiązana z wydaniem.
- Tytuł wydania.
- Opis wydania, który może zawierać listę zmian (ang. *release notes*).
- Przesyłamy plik binarny aplikacji w formacie ZIP, który odpowiada tej wersji.
- Zaznaczamy również pole wyboru dotyczące tego, czy jest to przedpremierowa wersja.

Następnie potwierdź nową wersję, klikając przycisk *Publish release*.

4. Na koniec zostajemy przekierowani do listy wydań projektu, który właśnie utworzyliśmy, jak pokazano na rysunku 16.14.

The screenshot shows the GitHub 'New Release' interface. At the top, there are tabs for 'Releases' and 'Tags'. Below them is a dropdown menu for the tag, currently set to 'v1.0.0'. A red circle with the number '1' is next to this dropdown. Below the dropdown is a checkbox labeled 'Existing tag'. Underneath is a text input field for the 'First release', with a red circle and the number '2' next to it. Below the input field are two tabs: 'Write' and 'Preview'. The 'Write' tab is active, showing a rich text editor with a toolbar containing icons for bold, italic, link, list, and other formatting options. The text area contains the placeholder text 'This first release contain:' followed by two bullet points: '- feature 1' and '- feature 2'. A red circle with the number '3' is next to the text area. Below the text area is a dashed line indicating where to attach files. Below that is a section for attaching binaries, with a red circle and the number '4' next to it. At the bottom, there is a checkbox labeled 'This is a pre-release' with a red circle and the number '5' next to it. Below the checkbox is a button labeled 'Publish release' with a red circle and the number '6' next to it, and a button labeled 'Save draft'.

Rysunek 16.13. GitHub — tworzenie wydania

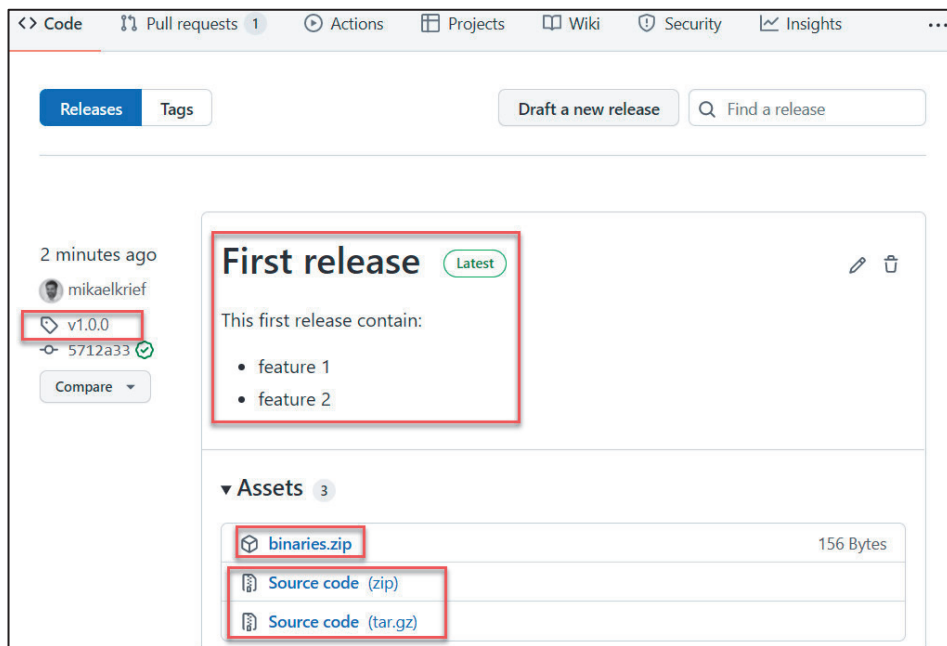
Na powyższym zrzucie ekranu widzimy tag (v1.0.0) powiązany z wydaniem, wprowadzone przez nas informacje i przesłany plik *binaries.zip*. Ponadto projekt GitHuba został automatycznie dodany do listy innych zasobów; tzn. pakiet (ZIP) zawierający kod źródłowy aplikacji skojarzonej z tym tagiem.

Właśnie dowiedzieliśmy się, że za pośrednictwem interfejsu webowego GitHuba możemy utworzyć wydanie, które pozwala nam udostępniać *release notes* i pliki binarne projektu wszystkim użytkownikom.

Uwaga

Możliwe jest również zintegrowanie wszystkich tych kroków z potokiem CI/CD za pomocą automatycznego skryptu korzystającego z różnych interfejsów API GitHuba. Dokumentację na ten temat można znaleźć na stronie <https://developer.github.com/v3/repos/releases/>.

W następnej sekcji utworzymy ten sam potok, ale przy użyciu GitHub Actions.



Rysunek 16.14. Szczegóły wydania w GitHubie

Wprowadzenie do GitHub Actions

GitHub integruje ze swoją platformą źródłową repozytorium kilka innych funkcji DevOps. Pozwala to na pełną integrację z kodem repozytorium.

W chwili pisania tego tekstu te nowe funkcje są następujące:

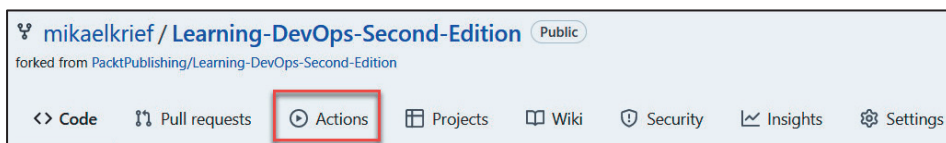
- Rejestr pakietów GitHuba — menedżer pakietów, którego dokumentację można znaleźć na stronie <https://github.com/features/package-registry>.
- GitHub Actions — menedżer potoków CI/CD, którego dokumentację można znaleźć na stronie <https://github.com/features/actions>.

W tej sekcji przedstawimy opis korzystania z usługi GitHub Actions, która umożliwia tworzenie potoków CI/CD bezpośrednio w GitHubie. Dzięki temu kod źródłowy hostowany w Twoim repozytorium GitHuba zostanie sprawdzony i wdrożony.

Na potrzeby tego przykładu utworzymy potok CI w usłudze GitHuba, który skompiluje i uruchomi testy dla naszej aplikacji Node.js. Zasoby na ten temat można znaleźć na stronie <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP16/appdemo>.

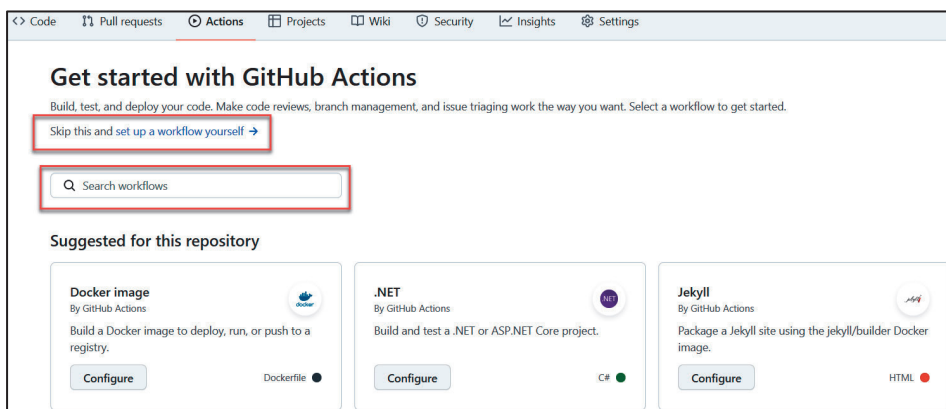
Wykonaj następujące kroki, aby utworzyć potok CI za pomocą GitHub Actions:

1. Przejdź do repozytorium zawierającego kod źródłowy do wdrożenia i kliknij zakładkę *Actions*, jak pokazano na poniższym zrzucie ekranu:



Rysunek 16.15. GitHub — zakładka Actions

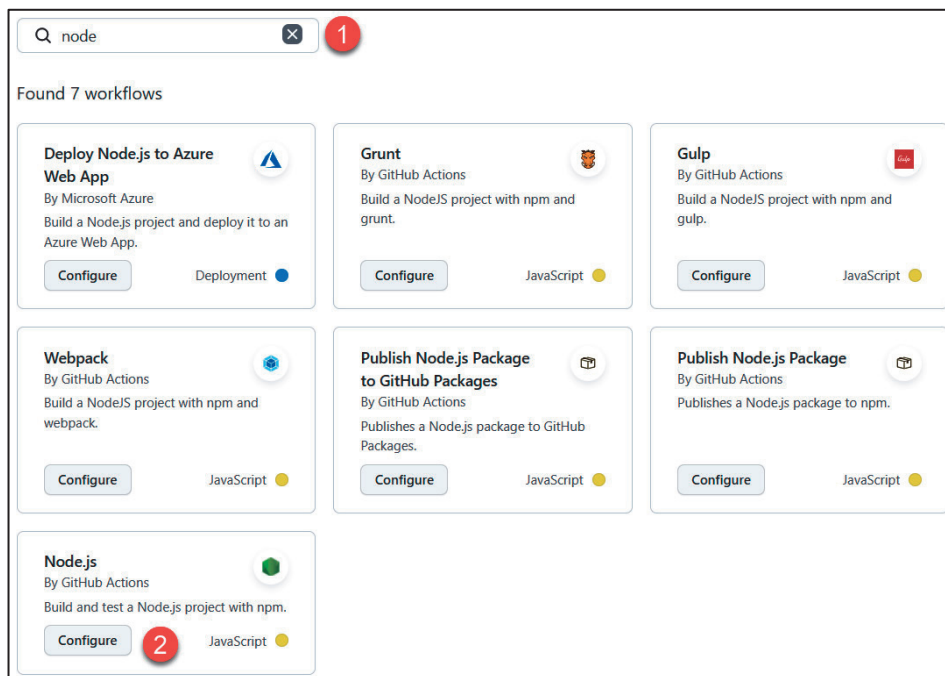
2. W tym momencie interfejs GitHuba zaproponuje szablony potoków, zwane *workflows*, zgodnie z różnymi językami programowania i systemem docelowym, takim jak VM lub Kubernetes. Możemy również utworzyć niestandardowy przepływ pracy, zaczynając od pustego szablonu. Poniższy zrzut ekranu pokazuje opcje tworzenia przepływu pracy (ang. *workflow*):



Rysunek 16.16. Szablony przepływów pracy GitHub Actions

Na powyższym zrzucie ekranu widzimy link o nazwie *set up a workflow yourself*, który służy do utworzenia niestandardowego przepływu pracy, a także pole tekstowe do wyszukiwania szablonów i sugerowane szablony.

3. Na potrzeby tego przykładu utwórzmy przepływ pracy z szablonu zaprojektowanego dla aplikacji Node.js. Znajdziemy go, wyszukując tekst *node* i klikając opcję *Configure* w polu szablonu Node.js.
4. GitHub wyświetli kod YAML przepływu pracy, który zostanie umieszczony w pliku *node.js.yml*. Ten plik zostanie automatycznie utworzony w drzewie



Rysunek 16.17. GitHub Actions — wybór szablonu Node.js

folderów `.github/workflows`. W tym kodzie YAML, który jest również dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/.github/workflows/node.js.yml>, możemy zobaczyć:

- Właściwość `runs-on`, która określa agenta Ubuntu dla potoku dostarczonego przez GitHuba.
- Listę kroków dotyczących użycia bloku akcji (`actions/checkout`), abyśmy mogli pobrać kod GitHuba, a następnie bloku skryptu (`npm`), który zostanie wykonany na agencie Ubuntu.

Przed zarchiwizowaniem tego pliku dodaj do niego krótki fragment kodu wskazujący ścieżkę wykonania skryptów, jak pokazano na rysunku 16.18.

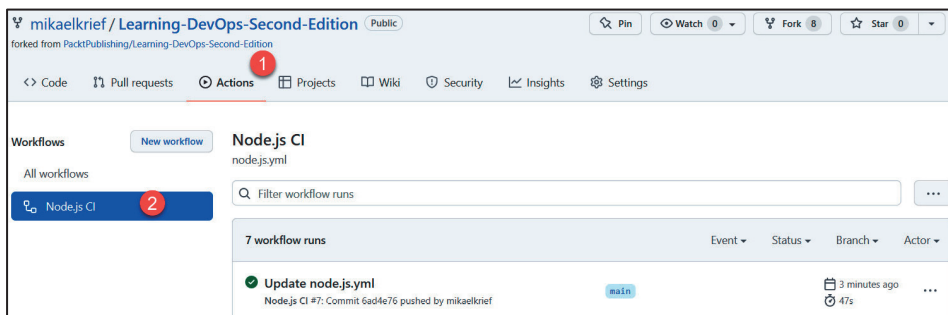
Pełny kod źródłowy tego przepływu pracy jest dostępny pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/.github/workflows/node.js.yml>.

5. Zatwierdź ten plik, klikając przycisk *Start commit* w górnej części edytora kodu. Po zatwierdzeniu plik będzie obecny w kodzie repozytorium i uruchomi nowy potok CI.


```
build:
  runs-on: ubuntu-latest
  strategy:
    matrix:
      node-version: [12.x, 14.x, 16.x]
      # See supported Node.js release schedule at https://nodejs.org/en/about/releases/
  steps:
    - uses: actions/checkout@v2
    - name: Use Node.js ${ matrix.node-version }
      uses: actions/setup-node@v2
      with:
        node-version: ${ matrix.node-version }
    - run: |
      cd CHAP16/appdemo
      npm ci
      npm run build --if-present
      npm test
```

Rysunek 16.18. Kod źródłowy przepływu pracy GitHub Actions

6. Na koniec wróćmy do zakładki *Actions*. Zobaczmy, że przepływ pracy został uruchomiony i zakończony:

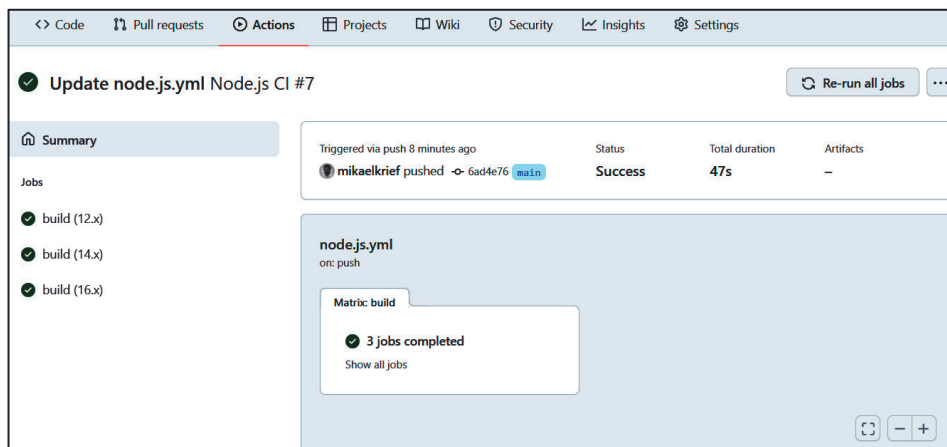


Rysunek 16.19. Uruchomiony przepływ pracy GitHub Actions

Tutaj możemy zobaczyć listę uruchomień dla tego przepływu pracy. Klikając linię uruchomienia, możemy zobaczyć szczegóły wykonania. Poniższy zrzut ekranu pokazuje szczegóły wykonania.

Wielką zaletą GitHub Actions jest to, że natywnie zapewnia bardzo obszerny katalog działań w swoim GitHub Marketplace (<https://github.com/marketplace?type=actions>), i to, że może również opracowywać i publikować akcje (<https://docs.github.com/en/actions/using-workflows/workflow-commands-for-github-actions>).

W tej sekcji omówiliśmy implementację potoku CI w GitHubie przy użyciu usługi GitHub Actions, która umożliwia integrację DevOps z projektami open source. Do tej pory



Rysunek 16.20. Szczegóły wykonania przepływu pracy GitHub Actions

w rozdziale skupialiśmy się na zarządzaniu kodem w GitHubie i wdrażaniu potoków CI/CD dla projektów open source z wykorzystaniem GitHub Actions.

W kolejnych sekcjach będziemy mówić o bezpieczeństwie otwartego kodu źródłowego. Zaczniemy od nauki analizy kodu za pomocą SonarCloud.

Analiza kodu za pomocą SonarCloud

W rozdziale 12., „Statyczna analiza kodu za pomocą SonarQube”, wyjaśniliśmy, jak ważne jest wdrożenie praktyk statycznej analizy kodu. W przypadku projektów otwartoźródłowych analiza kodu jest jeszcze ważniejsza, ponieważ kod źródłowy i jego pliki binarne są publikowane publicznie.

Jedną z ról open source jest dostarczanie kodu i komponentów, które można wykorzystać w aplikacjach korporacyjnych, więc ten kod musi być napisany poprawnie i bez żadnych problemów związanych z bezpieczeństwem.

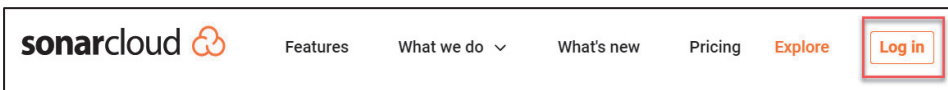
Wcześniej w tej książce omówiliśmy fakt, że SonarQube, z jego instalacjami i zastosowaniami, jest jednym z głównych narzędzi, które umożliwiają analizę kodu dla aplikacji korporacyjnych. Wymaga jednak posiadania infrastruktury lokalnej, co jest droższe dla firmy.

Do analizy kodu projektu open source można użyć narzędzia *SonarCloud* (<https://sonarcloud.io/>), które jest tym samym produktem co SonarQube, ale jest dostępne w rozwiązywaniu chmurowym, niewymagającym instalacji.

SonarCloud ma bezpłatny plan, który pozwala analizować kod projektów publicznych repozytoriów open source z GitHuba, BitBucketa, a nawet Azure Repos. Aby uzyskać więcej informacji na temat planów cenowych, przejdź do strony <https://sonarcloud.io/pricing>.

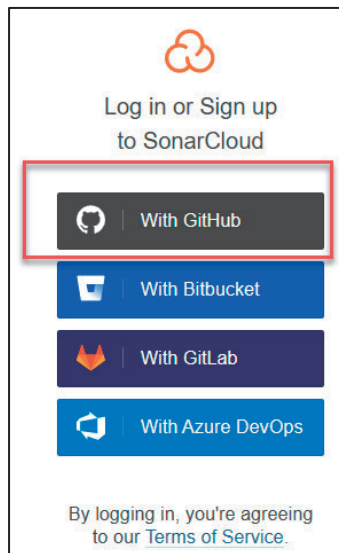
Teraz zobaczmy, jak szybko skonfigurować analizę kodu dla projektu open source hostowanego w serwisie GitHub. Przed wdrożeniem samej analizy połączymy się z naszym repozytorium GitHuba w SonarCloud. Aby to zrobić, musimy uzyskać dostęp do strony <https://sonarcloud.io/>, wykonując następujące kroki:

1. Na wspomnianej stronie głównej kliknij przycisk *Log in*:



Rysunek 16.21. SonarCloud — logowanie

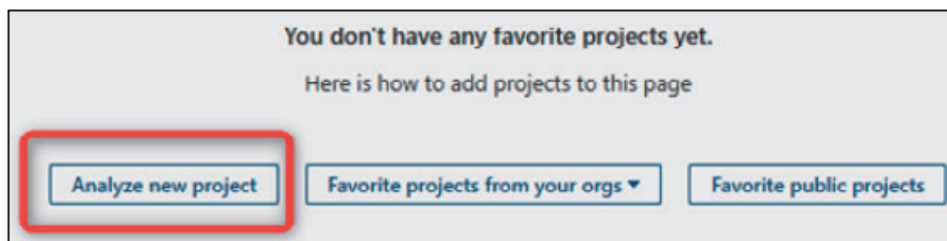
2. Następnie wybierz opcję logowania przy użyciu konta GitHuba, jak pokazano na poniższym zrzucie ekranu:



Rysunek 16.22. SonarCloud — logowanie za pomocą GitHuba

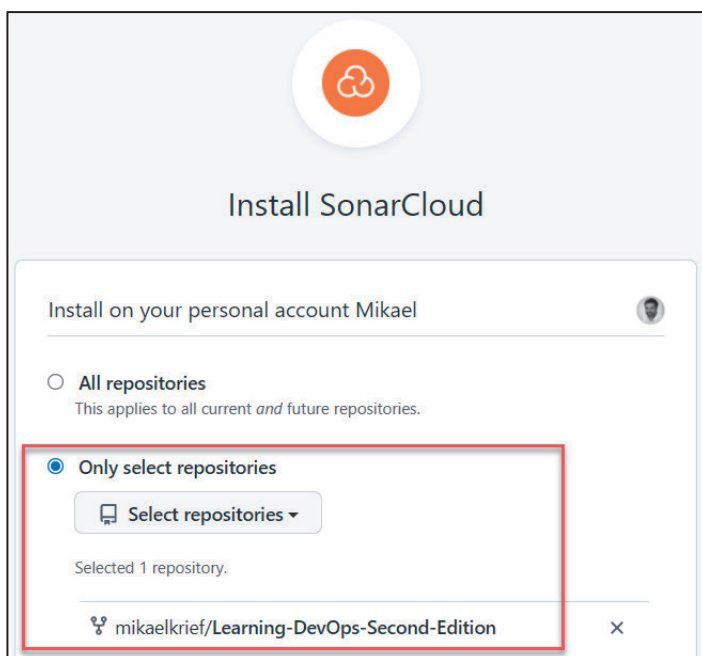
Teraz musimy skonfigurować SonarCloud, abyśmy mogli utworzyć projekt, który będzie zawierał analizę naszego projektu GitHuba. Projekt dostępny jest pod adresem <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP16/appdemo>. Aby to zrobić, wykonaj następujące kroki:

1. Po połączeniu się z SonarCloud za pomocą konta GitHuba kliknij przycisk *Analyze new project* na stronie głównej:



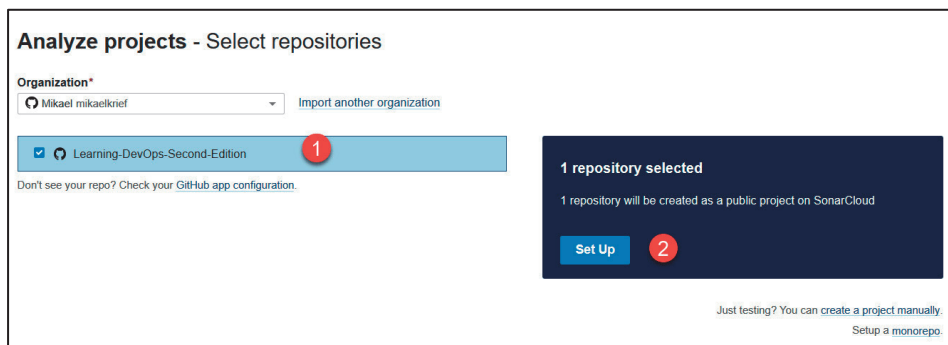
Rysunek 16.23. SonarCloud — analiza nowego projektu

2. SonarCloud proponuje kilka kroków, by wybrać docelowe repozytorium GitHuba do analizy. Najpierw wybierz opcję *Import an organization from GitHub*. Następnie wybierz opcję *Only select repositories* i repozytorium docelowe:



Rysunek 16.24. SonarCloud — wybór repozytorium

3. Następnie zaznacz swoje repozytorium i powiąż je ze swoją organizacją, klikając przycisk *Set Up*:

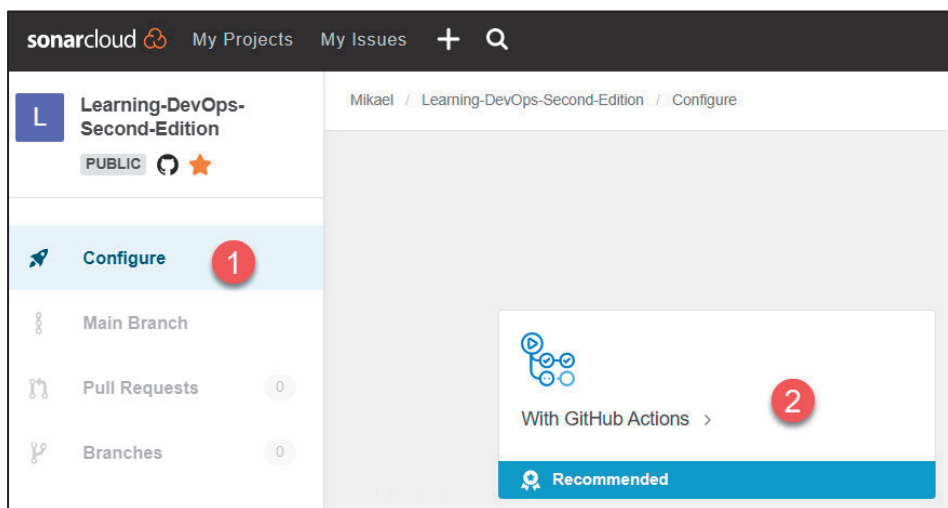


Rysunek 16.25. SonarCloud — konfigurowanie repozytorium

4. Zostanie wyświetlony dashboard analizy projektu i wskaże, że kod nie został jeszcze przeanalizowany.

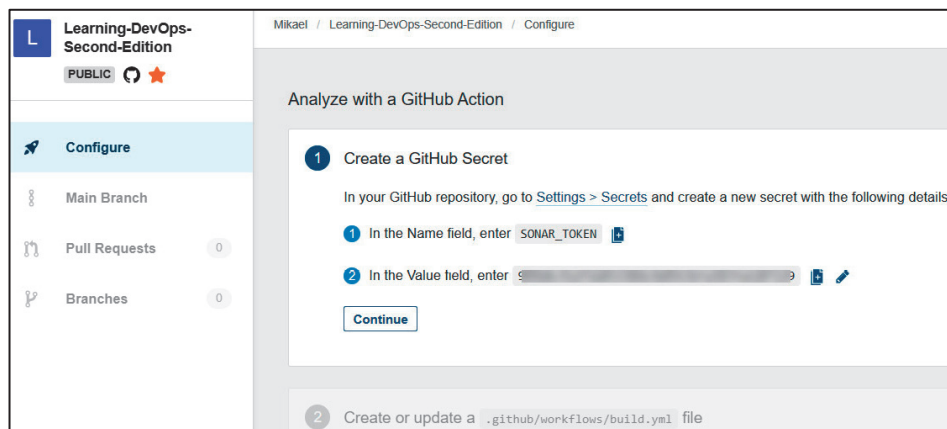
Dzięki temu udało nam się pomyślnie skonfigurować projekt SonarCloud. Teraz przeanalizujmy projekt w najbardziej podstawowy sposób, wykonując następujące kroki:

1. W projekcie SonarCloud w sekcji *Configure* wybierz zalecaną opcję *With GitHub Actions* (jak dowiedzieliśmy się w poprzedniej sekcji, „Wprowadzenie do GitHub Actions”):



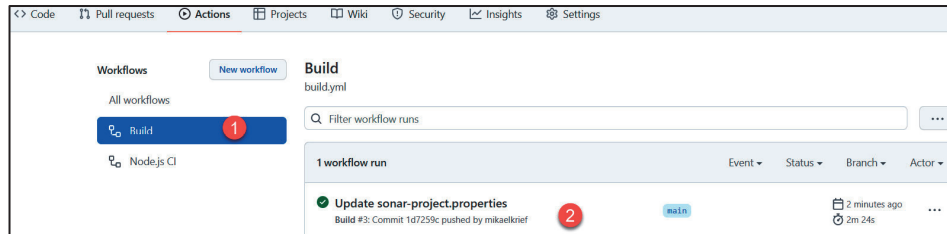
Rysunek 16.26. SonarCloud — używanie szablonu GitHub Actions

2. Następnie postępuj zgodnie ze wskazówkami dostarczonymi przez SonarCloud, aby utworzyć sekrety (za pomocą sekretu `SONAR_TOKEN`) w repozytorium GitHuba, i dodaj nowy plik z konfiguracją SonarCloud:



Rysunek 16.27. Instrukcje dotyczące GitHub Actions

3. Następnie zatwierdź nowy plik przepływu pracy GitHub Actions. Spowoduje to automatyczne uruchomienie analizy SonarCloud dla tego projektu.
4. Po kilku minutach wyświetli się wynik dwóch uruchomień. Najpierw zobaczysz, że przepływ pracy GitHub Actions powiódł się:



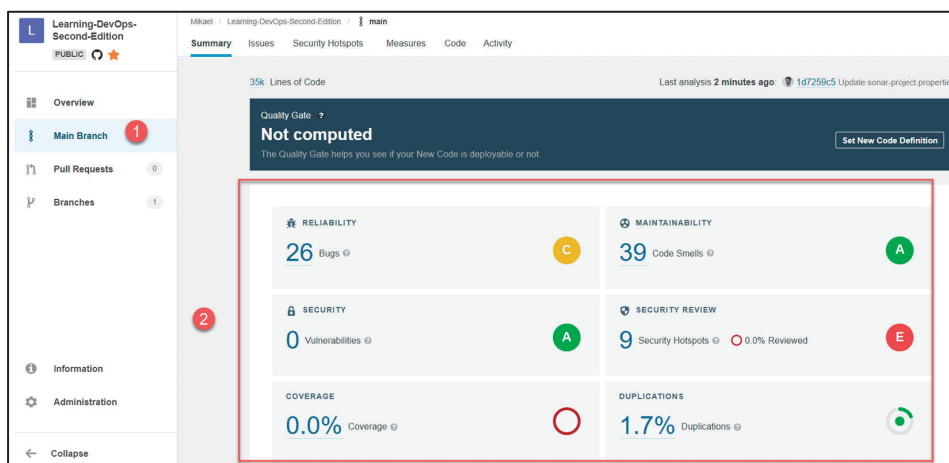
Rysunek 16.28. Analiza za pomocą GitHub Actions

Następnie zobaczysz analizę na pulpicie nawigacyjnym SonarCloud, jak pokazano na poniższym zrzucie ekranu.

Tak więc przy każdym nowym zatwierdzeniu kodu w tym repozytorium, bezpośrednio lub poprzez wykonanie scalenia za pomocą żądania pobrania, zostanie uruchomiona analiza kodu i zaktualizowany pulpit nawigacyjny SonarCloud.

Oczywiste jest, że naszym celem końcowym jest zintegrowanie analizy SonarCloud z potokiem CI/CD, więc oto kilka zasobów, które mogą w tym pomóc:

- Jeśli korzystasz z Azure DevOps, oto kompletny samouczek, który pomoże Ci zintegrować SonarCloud z potokiem: <https://docs.microsoft.com/en-us/labs/devops/sonarcloudlab/>.



Rysunek 16.29. Wyniki analizy SonarCloud

- Jeśli korzystasz z Travis CI, zapoznaj się z następującą dokumentacją: <https://docs.travis-ci.com/user/sonarcloud/>.

W tej sekcji dowiedzieliśmy się, jak skonfigurować SonarCloud, platformę chmurową, która analizuje kod statyczny. Zrobiliśmy to, by przeanalizować kod źródłowy projektu open source na GitHubie za pomocą procesu ciągłej integracji. Następnie przyjrzelśmy się wynikowi tej analizy na pulpicie nawigacyjnym.

W następnej sekcji przyjrzymy się aspektowi bezpieczeństwa kodu open source, czyli analizie luk w kodzie za pomocą narzędzia WhiteSource Bolt.

Wykrywanie luk w zabezpieczeniach za pomocą narzędzia WhiteSource Bolt

Ze względu na ich publiczną widoczność projekt lub komponenty open source są silnie narażone na wystąpienie luk w zabezpieczeniach, ponieważ łatwiej jest nieumyślnie wstrzyknąć do nich kod (pakiet lub jedną z jego zależności) zawierający lukę w zabezpieczeniach.

Oprócz statycznej analizy kodu źródłowego bardzo ważne jest ciągłe sprawdzanie bezpieczeństwa pakietów, do których się odwołujemy lub których używamy w naszych projektach open source.

Dostępnych jest wiele narzędzi, które możemy zastosować do analizy bezpieczeństwa pakietów referencyjnych w aplikacjach, takich jak SonaType AppScan (<https://www.>

sonatype.com/appscan), Snyk (<https://snyk.io/>) i WhiteSource Bolt (<https://bolt.whitesourcesoftware.com/>).

Uwaga

Aby uzyskać więcej informacji na temat narzędzi open source wykorzystywanych do skanowania luk w zabezpieczeniach, zapoznaj się z następującym artykułem, w którym wymieniono 13 narzędzi analizujących bezpieczeństwo zależności open source: <https://techbeacon.com/app-dev-testing/13-tools-checking-security-risk-open-source-dependencies>.

Spośród wszystkich tych narzędzi przyjrzymy się **WhiteSource Bolt** (<https://bolt.whitesourcesoftware.com/>), które jest dostępne w bezpłatnej ofercie. Może analizować kod pakietu wielu języków programowania i umożliwia bezpośrednią integrację z GitHubem i Azure DevOps.

Uwaga

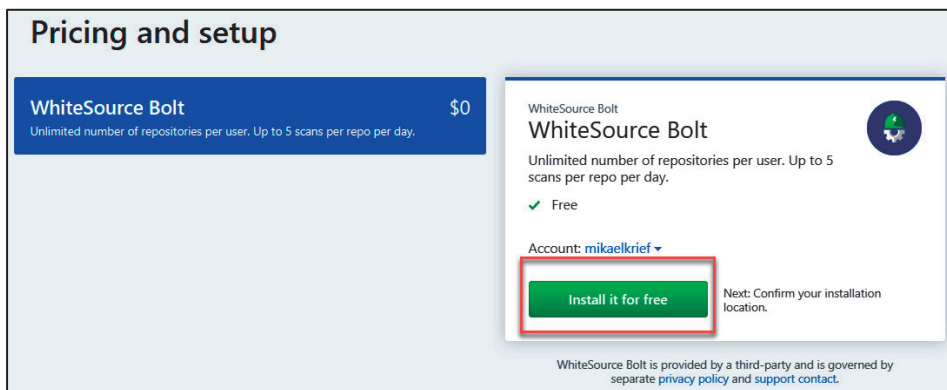
Pełna dokumentacja dotycząca integracji WhiteSource Bolt z GitHubem jest dostępna pod adresem <https://whitesource.atlassian.net/wiki/spaces/WD/pages/556007950/WhiteSource+Bolt+for+GitHub>.

W naszym przypadku wykorzystamy go bezpośrednio w GitHubie do analizy bezpieczeństwa aplikacji, której źródła są dostępne tutaj: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP16/appdemo>.

Aby przeprowadzić analizę bezpieczeństwa, musimy zainstalować i skonfigurować WhiteSource Bolt na naszym koncie GitHuba i uruchomić analizę kodu. Wykonaj następujące kroki:

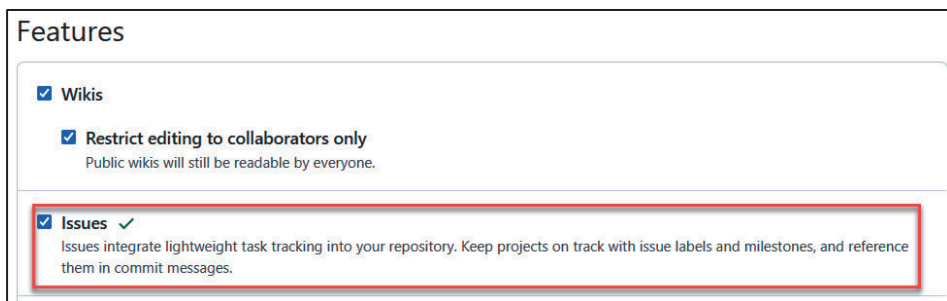
1. W przeglądarce internetowej przejdź do adresu <https://bolt.whitesourcesoftware.com/github/> i kliknij przycisk *GitHub APP*, aby zainstalować WhiteSource Bolt na swoim koncie GitHuba.
2. Zostaniesz przekierowany do aplikacji WhiteSource Bolt, którą znajdziesz w GitHub Marketplace (<https://github.com/marketplace/whitesource-bolt>). Aby zainstalować ją bezpłatnie, kliknij przycisk *Install it for free* na samym dole strony, jak pokazano na poniższym zrzucie ekranu.
3. Potwierdź zakup aplikacji za 0\$, klikając przycisk *Complete order and begin installation*. Na następnej stronie potwierdź, że WhiteSource Bolt został zainstalowany na Twoim koncie GitHuba.

Po zakończeniu instalacji zostaniesz przekierowany na stronę tworzenia konta WhiteSource Bolt, gdzie musisz podać swoje imię i nazwisko oraz kraj, a następnie zatwierdzić podane dane.



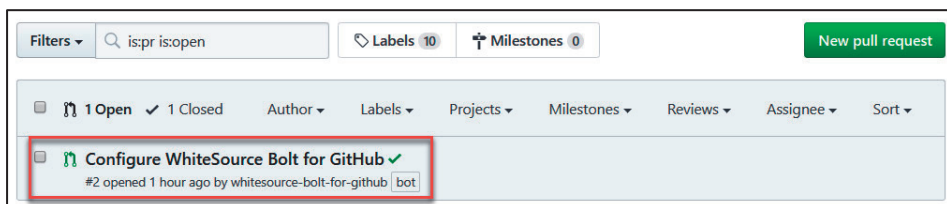
Rysunek 16.30. Instalacja WhiteSource Bolt

4. Teraz aktywuj funkcje *Issues* w GitHubie, przechodząc do zakładki *Settings* dla repozytorium, które zawiera kod do skanowania, i zaznaczając pole *Issues*, jak pokazano na poniższym zrzucie ekranu:



Rysunek 16.31. GitHub — aktywacja Issues

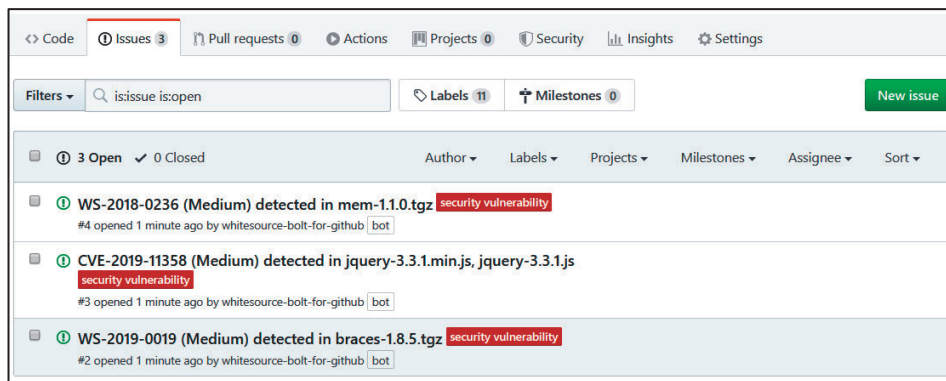
5. Aby skonfigurować analizę WhiteSource w tym repozytorium, musisz zweryfikować i scalić nowe żądanie pobrania, które zostało automatycznie utworzone podczas instalacji WhiteSource Bolt.



Rysunek 16.32. Konfiguracja WhiteSource Bolt

To żądanie *pull* dodaje plik *.whitesource*, który jest używany do konfiguracji w katalogu głównym repozytorium.

6. Na końcu wykonaj analizę kodu, zatwierdzając kod tej aplikacji w repozytorium GitHuba.
7. Po kilku minutach zobaczysz listę problemów bezpieczeństwa w zakładce repozytorium *Issues*.



Rysunek 16.33. Problemy wykryte podczas analizy WhiteSource w GitHubie

8. Aby poznać wszystkie szczegóły problemu związanego z bezpieczeństwem, po prostu kliknij żądany problem, przeczytaj go i weź pod uwagę informacje zawarte w opisie problemu — rysunek 16.34.

Będziemy musieli naprawić wszystkie te problemy i ponownie wykonać zatwierdzenia kodu, aby uruchomić nowe skanowanie WhiteSource Bolt i dostarczyć bezpieczną aplikację tym, którzy będą z niej korzystać.

W tej sekcji nauczyliśmy się analizować kod projektów open source za pomocą narzędzia WhiteSource Bolt. Zainstalowaliśmy je i uruchomiliśmy analizę kodu, która ujawniła problemy z bezpieczeństwem w naszej aplikacji.

Podsumowanie

Ten rozdział został poświęcony najlepszym praktykom DevOps, które można zastosować dla projektu open source, zwłaszcza dostępnego na GitHubie. W tym rozdziale nauczyliśmy się, jak współpracować nad otwartym kodem źródłowym, zaczynając od tworzenia repozytorium przy użyciu GitHuba i rozgałęzień. Potem nauczyliśmy się korzystać z żądań *pull* i udostępniać pliki binarne w wydaniach GitHuba.

CVE-2019-11358 (Medium) detected in jquery-3.3.1.min.js, jquery-3.3.1.js #3

Open whitesource-bolt-for-github [bot] opened this issue 1 minute ago · 0 comments

whitesource-bolt-... [bot] commented 1 minute ago

CVE-2019-11358 - Medium Severity Vulnerability

▼ Vulnerable Libraries - jquery-3.3.1.min.js, jquery-3.3.1.js

- ▶ jquery-3.3.1.min.js
- ▶ jquery-3.3.1.js

Found in HEAD commit: 64bf284ae4358b2a0bb8e5d7cf619e0050cb8ab0

▼ Vulnerability Details

jQuery before 3.4.0, as used in Drupal, Backdrop CMS, and other products, mishandles jQuery.extend(true, {}, ...) because of Object.prototype pollution. If an unsanitized source object contained an enumerable proto property, it could extend the native Object.prototype.

Publish Date: 2019-04-20

URL: CVE-2019-11358

- ▶ CVSS 3 Score Details (6.1)
- ▶ Suggested Fix

Assignees: No one—assign yourself

Labels: security vulnerability

Projects: None yet

Milestone: No milestone

Notifications: Customize

Subscribe

You're not receiving notifications from this thread.

0 participants

Lock conversation

Rysunek 16.34. WhiteSource Bolt — analiza szczegółów

Następnie wdrożyliśmy procesy ciągłej integracji za pomocą usługi GitHub Actions, która jest w pełni zintegrowana z GitHubem.

Na koniec nauczyliśmy się, jak analizować kod open source pod kątem statycznej analizy kodu za pomocą SonarCloud oraz pod kątem analizy podatności na zagrożenia za pomocą WhiteSource Bolt.

W następnym rozdziale podsumujemy wszystkie najlepsze praktyki DevOps, o których mówiliśmy w tej książce.

Pytania

1. Czy w GitHubie mogę modyfikować kod repozytorium innego użytkownika?
2. Który element w GitHubie pozwala nam łączyć zmiany kodu między dwoma repozytoriami?
3. Który element pozwala nam w prosty sposób wyświetlić historię zmian kodu w projekcie open source?

4. Która funkcja wspomniana w tym rozdziale pozwala nam udostępniać pliki binarne w GitHubie?
5. Jakie dwa narzędzia, którym przyjrzelśmy się w tym rozdziale, pozwalają nam analizować kod źródłowy projektu open source?
6. W której zakładce GitHuba wymienione są problemy bezpieczeństwa wykryte przez WhiteSource Bolt?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o stosowaniu praktyk DevOps w projektach open source, zajrzyj do książki *GitHub Essentials* Achilleasa Pipinellisa opublikowanej przez wydawnictwo Packt: <https://www.packtpub.com/in/web-development/github-essentials-second-edition>.

Najlepsze praktyki DevOps

Rozdział

17

Dotarliśmy do ostatniego rozdziału tej książki i wreszcie, po przeczytaniu ich wszystkich, prawdopodobnie zadajesz sobie pytanie: *Jakie są najlepsze praktyki, które należy zastosować, aby skutecznie wdrożyć kulturę rozwoju oprogramowania i operacji (DevOps)?*

Ten rozdział jest świetnym przeglądem dobrych praktyk DevOps, które już poznaliśmy, i pozwoli Ci przećwiczyć wszystkie elementy, które widzieliśmy w tej książce.

Omówimy najlepsze praktyki w zakresie automatyzacji, wyboru narzędzi, **infrastruktury jako kodu (IaC)**, architektury aplikacji i projektowania infrastruktury. Omówimy również dobre praktyki, które należy zastosować w zarządzaniu projektami, by ułatwić wdrożenie kultury i praktyk DevOps. Następnie dokonamy przeglądu najlepszych praktyk dotyczących potoków **ciągłej integracji/ciągłego wdrażania (CI/CD)**, automatyzacji testów i integracji zabezpieczeń z procesami DevOps.

Rozdział zakończymy opisem najlepszych praktyk monitorowania w kulturze DevOps.

Rozdział ten obejmuje następujące tematy:

- pełna automatyzacja,
- wybór odpowiedniego narzędzia,
- tworzenie całej konfiguracji za pomocą kodu,
- projektowanie architektury systemu,
- budowanie dobrego potoku CI/CD,
- testy integracyjne,
- przesunięcie bezpieczeństwa w lewo dzięki DevSecOps,
- monitorowanie systemu,
- ewoluujące zarządzanie projektami.

Pełna automatyzacja

Żeby wdrożyć praktyki DevOps w firmie, należy pamiętać o celu kultury DevOps: szybkie dostarczanie nowych wersji aplikacji w krótszych cyklach.

Aby to zrobić, pierwszą dobrą praktyką, którą należy zastosować, jest zautomatyzowanie wszystkich zadań związanych z wdrażaniem, testowaniem i zabezpieczaniem aplikacji i jej infrastruktury. Gdy zadanie jest wykonywane ręcznie, istnieje duże ryzyko błędu. Ręczne wykonywanie tych zadań wydłuża cykle wdrażania aplikacji.

Ponadto po zautomatyzowaniu tych zadań w skryptach można je łatwo zintegrować i wykonać w potokach CI/CD. Kolejną zaletą automatyzacji jest to, że programiści i zespół operacyjny mogą poświęcić więcej czasu i skoncentrować pracę na funkcjonalności swojego biznesu.

Ważne jest również to, aby na początku rozwoju projektu rozpocząć automatyzację procesu dostawy; dzięki temu możemy udzielać informacji zwrotnej szybciej i wcześniej.

Wreszcie automatyzacja umożliwia usprawnienie monitorowania wdrożeń poprzez śledzenie każdej akcji. Umożliwia również bardzo szybkie wykonanie kopii zapasowej i przywrócenie kodu w przypadku wystąpienia problemu.

Automatyzacja wdrożeń skróci zatem cykle wdrożeń, a zespoły mogą teraz pozwolić sobie na pracę w mniejszych iteracjach. W ten sposób **czas wprowadzenia na rynek** (ang. *time to market* — **TTM**) zostanie skrócony, a dodatkową korzyścią będą aplikacje lepszej jakości.

Jednak automatyzacja i orkiestracja wymagają wdrożenia narzędzi, a wybór tych narzędzi jest istotnym elementem w przypadku wdrażania kultury DevOps.

Wybór odpowiedniego narzędzia

Jednym z wyzwań stojących przed firmą, która chce zastosować kulturę DevOps, jest wybór narzędzi.

Wiele narzędzi jest płatnych, bezpłatnych i otwartoźródłowych. Umożliwiają wersjonowanie kodu źródłowego aplikacji, automatyzację procesów, wdrażanie potoków CI/CD oraz testowanie i monitorowanie aplikacji.

Wraz z tymi narzędziami dodawane są języki skryptowe, takie jak PowerShell, Bash i Python, które są również częścią zestawu narzędzi DevOps służących do integracji.

Często zadawane mi pytanie brzmi: jak wybrać odpowiednie narzędzia DevOps, które są przydatne dla mojej firmy i biznesu?

W rzeczywistości, aby odpowiedzieć na to pytanie, musimy pamiętać o definicji kultury DevOps podanej przez Donovana Browna, o której wspomniano w rozdziale 1., „Kultura DevOps i praktyki kodowania infrastruktury”, i przytoczono ją tutaj:

„DevOps to połączenie ludzi, procesów i produktów, które umożliwia ciągłe dostarczanie wartości naszym użytkownikom końcowym”.

Ważnym punktem tej definicji jest to, że kultura DevOps jest połączeniem Dev, Ops, procesów, a także narzędzi. Oznacza to, że używane narzędzia muszą być udostępniane i stosowane zarówno przez programistów, jak i operatorów oraz powinny być zintegrowane z tym samym procesem. Innymi słowy: wybór narzędzi zależy od zespołów i modelu firmy.

Konieczne jest również uwzględnienie systemu finansowego poprzez wybór narzędzi open source, które często są bezpłatne; łatwiej jest z nich korzystać na początku transformacji DevOps. Inaczej będzie w przypadku płatnych narzędzi, które z pewnością są bogatsze w funkcje i we wsparcie, ale wymagają znacznych inwestycji.

Jeśli chodzi o języki skryptowe, powiedziałbym, że wybór języka musi być dokonany zgodnie z wiedzą zespołów. Na przykład zespoły operacyjne, które są lepiej przeszkolone w zakresie systemów Linux, będą w stanie udoskonalić skrypty automatyzacji w Bash, a nie w PowerShell.

W tej książce zapoznaliśmy Cię z kilkoma narzędziami, z których jedno są otwarte i bezpłatne — takie jak Terraform, Packer, Vault i Ansible — a inne płatne — takie jak Azure DevOps (dla ponad pięciu użytkowników) lub LaunchDarkly. Zrobiliśmy to, aby pomóc Ci wybrać narzędzia, które najlepiej Ci odpowiadają.

Po tej refleksji nad doбором narzędzi przy wdrażaniu praktyki DevOps przyjrzymy się kolejnej dobrej praktyce, jaką jest umieszczanie wszystkiego w kodzie.

Tworzenie całej konfiguracji za pomocą kodu

W całej książce, zwłaszcza w pierwszych trzech rozdziałach poświęconych IaC, widzieliśmy, że zapisanie pożądanej konfiguracji infrastruktury w kodzie oferuje wiele korzyści dla produktywności zarówno zespołów, jak i firmy.

Dlatego bardzo dobrą praktyką jest umieszczanie w kodzie wszystkiego, co dotyczy konfiguracji infrastruktury. Widzieliśmy to w praktyce z wykorzystaniem narzędzi Terraform, Ansible i Packer. Istnieje jeszcze wiele innych narzędzi, które mogą być lepiej dostosowane do Twoich potrzeb i Twojej organizacji. Mówimy nie tylko o edytorach, ale także o wykorzystaniu plików JSON (ang. *JavaScript Object Notation*), skryptów Bash, PowerShell i Python. Kluczem jest posiadanie opisu infrastruktury w kodzie, który jest łatwy do odczytania przez człowieka, i narzędzi dostosowanych do użytkownika, jak omówiono w poprzedniej sekcji.

Co więcej, praktyka ta ewoluuje w innych obszarach, co widzieliśmy w rozdziale 14., „Bezpieczeństwo w procesie DevOps z wykorzystaniem DevSecOps”, z wykorzystaniem narzędzia InSpec, które pozwala nam opisać zasady zgodności infrastruktury w kodzie.

Widzieliśmy również tę praktykę IaC w kilku rozdziałach tej książki z tzw. **potokiem w postaci kodu** (ang. *Pipeline as Code* — **PaC**), z wykorzystaniem GitLab CI, GitHub Actions, a także platformy Azure Pipelines, która pozwala również na użycie formatu **YAML** (ang. *YAML Ain't Markup Language*) w trybie potokowym (tryb ten nie został omówiony w tej książce).

Umieszczanie dowolnej konfiguracji w kodzie jest kluczową praktyką kultury DevOps, którą należy brać pod uwagę od początku projektu zarówno dla zespołów operacyjnych, jak i deweloperskich. Dla deweloperów istnieją również dobre praktyki dotyczące projektowania architektury aplikacji i infrastruktury, co omówimy w następnej sekcji.

Projektowanie architektury systemu

Kilka lat temu wszystkie usługi tej samej aplikacji były zakodowane w tym samym bloku aplikacji. Taka architektura była zasadna, ponieważ aplikacja była zarządzana w modelu kaskadowym (<https://activecollab.com/blog/project-management/waterfall-project-management-methodology>), więc nowe wersje aplikacji były wdrażane w bardzo długich cyklach. Od tego czasu w praktykach inżynierii oprogramowania zaszło wiele zmian, poczynając od przyjęcia metodyki zwinnej i kultury DevOps, kończąc na nadejściu chmury. Ta ewolucja przyniosła wiele ulepszeń nie tylko w aplikacjach, ale także w ich infrastrukturze.

Aby jednak wykorzystać efektywną kulturę DevOps do wdrażania aplikacji w chmurze, podczas projektowania architektury oprogramowania, a także przy projektowaniu infrastruktury należy wziąć pod uwagę dobre praktyki.

Przede wszystkim architekci chmury muszą współpracować z programistami (lub architektami rozwiązań), by zapewnić, że opracowana aplikacja jest zgodna z różnymi

komponentami architektury i że architektura uwzględnia również rozmaite ograniczenia aplikacji.

Oprócz tej współpracy zespoły ds. bezpieczeństwa muszą zapewnić specyfikacje, które zostaną wdrożone przez programistów i architektów chmury.

Aby móc częściej wdrażać nową wersję aplikacji bez konieczności wpływania na wszystkie jej funkcje, dobrą praktyką jest rozdzielanie różnych obszarów aplikacji, najpierw opisując je za pomocą osobnych kodów, a potem, na późniejszym etapie, dzieląc pomiędzy różne zespoły. W ten sposób podzielony kod będzie znacznie łatwiejszy w utrzymaniu i skalowalny i będzie można go wdrożyć szybciej bez konieczności ponownego wdrażania wszystkiego.

Uwaga

Metoda rozdzielania kodu na kilka usług jest częścią wzorca architektury zwanego **mikroserwisami**; aby dowiedzieć się więcej, przeczytaj następujący obszerny artykuł: <https://microservices.io/patterns/microservices.html>.

Jednak po rozdzieleniu nadal istnieje potrzeba kontrolowania zależności w celu zaimplementowania potoku CI/CD, który uwzględnia wszystkie zależności aplikacji.

W rozdziale 15., „Skrócenie czasu przestoju wdrażania”, omówiliśmy inną dobrą praktykę, która pozwala na częstsze wdrażanie w środowisku produkcyjnym. Polega ona na hermetyzacji funkcjonalności aplikacji za pomocą flag funkcjonalności. Te flagi muszą być również brane pod uwagę przy projektowaniu aplikacji, ponieważ umożliwiają wdrożenie aplikacji na etapie produkcyjnym, gdyż pozwalają na dynamiczne włączanie/wyłączanie funkcjonalności bez potrzeby jej ponownego wdrażania.

Wreszcie w rozwoju aplikacji musi być jak najszybciej uwzględnione wdrożenie testów jednostkowych i mechanizmu rejestrowania, ponieważ pozwalają one na bardzo szybkie udostępnianie informacji zwrotnej o stanie aplikacji w jej cyklu wdrażania.

Kultura DevOps obejmuje implementację potoków CI/CD; jak właśnie widzieliśmy, wymaga to zmian w projekcie aplikacji, z wydzieleniem funkcjonalności w celu utworzenia mniej monolitycznych aplikacji, implementacji testów i dodania systemu logowania.

Po rozważeniu dobrych praktyk projektowania aplikacji przyjrzymy się kilku dobrym praktykom wdrażania potoków CI/CD.

Budowanie dobrego potoku CI/CD

W tej książce poświęciliśmy rozdział 7., „Ciągła integracja i ciągłe wdrażanie”, tworzeniu potoków CI/CD przy użyciu różnych narzędzi, takich jak GitLab CI, Jenkins i Azure Pipelines, w których wspomnieliśmy o warunkach wstępnych wdrożenia potoków CI/CD.

W rozdziale 16., „DevOps dla projektów open source”, omówiliśmy również proces CI/CD z kilkoma przykładami potoku CI dla projektów open source, takich jak GitHub Actions.

Budowanie dobrego potoku CI/CD jest rzeczywiście podstawową praktyką w kulturze DevOps i wraz z właściwym doбором narzędzi pozwala na szybsze wdrażanie i lepszą jakość aplikacji.

Jedną z najlepszych praktyk dla potoków CI/CD jest ich konfiguracja już na etapie uruchamiania projektu. Jest to szczególnie prawdziwe w przypadku potoku CI, który pozwoli na weryfikację kodu (przynajmniej kroku kompilacji) podczas pisania pierwszych linii kodu. Następnie, gdy pierwsze środowisko zostanie udostępnione, natychmiast utwórz potok wdrażania, który pozwoli na wdrożenie i przetestowanie aplikacji w tym środowisku. Pozostałe zadania procesu potokowego CI/CD, takie jak przeprowadzanie testów jednostkowych, można realizować w miarę postępu projektu.

Ponadto ważna jest optymalizacja procesów potoku CI/CD poprzez ich szybkie uruchamianie. Służą one do szybkiego przekazywania informacji zwrotnych członkom zespołu (zwłaszcza w przypadku CI), a także dzięki nim można uniknąć blokowania kolejki wykonywania innych potoków, które mogą się w niej znajdować.

Tak więc jeśli niektóre potoki działają zbyt długo, np. testy integracyjne (które mogą być czasochłonne), dobrym pomysłem może być zaplanowanie ich wykonywania w czasie z mniejszą aktywnością, np. w nocy.

Wreszcie ważna jest ochrona wrażliwych danych osadzonych w potokach CI/CD. Jeśli więc używasz narzędzia do zarządzania konfiguracją w swoich potokach, nie pozostawiaj informacji takich jak hasła, parametry połączenia i tokeny widocznych dla wszystkich użytkowników.

Aby chronić te dane, użyj scentralizowanych narzędzi do zarządzania kluczami, takich jak Vault, które widzieliśmy w rozdziale 14., „Bezpieczeństwo w procesie DevOps z wykorzystaniem DevSecOps”, lub użyj **Azure Key Vault (AKV)**, jeśli masz subskrypcję platformy Azure.

Oto niektóre z najlepszych praktyk wdrażania potoków CI/CD. Wspomnieliśmy o innych dobrych praktykach dotyczących potoków CI/CD, które możesz przestudiować w różnych rozdziałach tej książki i w innych książkach poświęconych kulturze DevOps.

W ramach kontynuacji dobrych praktyk dotyczących potoków CI/CD przejrzymy te dotyczące integracji testów z procesami DevOps.

Testy integracyjne

Testowanie jest w dzisiejszym świecie główną częścią procesu DevOps, ale także praktyk programistycznych. Owszem, możliwe jest posiadanie najlepszego potoku DevOps, który automatyzuje wszystkie fazy dostarczania, ale bez integracji testów traci on prawie całą swoją wydajność. Uważam, że minimalnym wymaganiem dla procesu DevOps jest zintegrowanie przynajmniej wykonania testów jednostkowych aplikacji. Ponadto te testy jednostkowe muszą być napisane od pierwszej linii kodu aplikacji przy użyciu praktyk testowych, takich jak **programowanie oparte na testach** (ang. *test-driven development* — **TDD**) (<https://hackernoon.com/introduction-to-test-driven-development-tdd-61a13bc92d92>) i **rozwój oprogramowania oparty na zachowaniu** (ang. *behavior-driven development* — **BDD**), dzięki czemu automatyczne wykonywanie tych testów może zostać zintegrowane z potokiem CI.

Jednak ważne jest, aby zintegrować inne rodzaje testów, takie jak testy funkcjonalne lub testy integracyjne, które pozwalają przetestować aplikację od początku do końca pod względem funkcjonowania z innymi komponentami jej ekosystemu.

Z pewnością prawdą jest, że przeprowadzenie tych testów może zająć trochę czasu; w takim przypadku istnieje możliwość zaplanowania ich wykonania w nocy. Ale właśnie te testy integracyjne zagwarantują jakość płynnego działania aplikacji na wszystkich etapach jej dostarczania, od wdrożenia po produkcję.

Niestety często spotyka się bardzo złą praktykę wyłączania przeprowadzenia testów w potoku CI w przypadku niepowodzenia ich wykonania. Ma to na celu uniknięcie zablokowania całego procesu CI, a tym samym szybsze dostarczanie w produkcji. Należy jednak pamiętać, że błędy wykryte przez testy jednostkowe, w tym te, które zostałyby wykryte przez wyłączone testy, zostaną w pewnym momencie znalezione na etapie produkcji, a naprawa nieudanego kodu zajmie więcej czasu, niż gdyby testy zostały włączone podczas CI.

W tej książce dowiedzieliśmy się również o innych rodzajach testów, takich jak testy analizy kodu lub testy bezpieczeństwa, których nie należy ignorować. Im szybciej zostaną one zintegrowane z potokami CI/CD, tym większa będzie wartość dodana dla utrzymania kodu i zabezpieczenia naszej aplikacji.

Podsumowując: nie powinieneś ignorować implementacji testów w swoich aplikacjach i ich integracji z potokami CI/CD, ponieważ gwarantują one jakość Twojej aplikacji.

Po dobrych praktykach dotyczących integracji testów proponuję zapoznać się z dobrymi praktykami dotyczącymi integracji bezpieczeństwa w procesach CI/CD.

Przesunięcie bezpieczeństwa w lewo dzięki DevSecOps

Jak mówiliśmy w rozdziale 14., „Bezpieczeństwo w procesie DevOps z wykorzystaniem DevSecOps”, analizy bezpieczeństwa i zgodności muszą być częścią procesów DevOps. Jednak w firmach często brakuje świadomości zespołów programistycznych na temat zasad bezpieczeństwa, przez co zabezpieczenia w procesach DevOps są wdrażane zbyt późno.

Aby zintegrować bezpieczeństwo z procesami, konieczne jest zatem podnoszenie świadomości programistów w zakresie aspektów bezpieczeństwa kodu aplikacji, a także ochrony konfiguracji potoków CI/CD.

Ponadto konieczne jest również wyeliminowanie bariery między DevOps a bezpieczeństwem poprzez częstszą integrację zespołów ds. bezpieczeństwa na różnych spotkaniach skupiających zespoły Dev i Ops, zapewniając w ten sposób lepszą spójność między programistami, zespołami operacyjnymi, a także komórką bezpieczeństwa. Jeśli chodzi o wybór narzędzi, nie używaj zbyt wielu różnych, ponieważ celem jest, aby te narzędzia były używane także przez programistów i zintegrowane z potokami CI/CD. Konieczne jest zatem wybranie kilku narzędzi, które są zautomatyzowane, nie wymagają dużej wiedzy na temat bezpieczeństwa i dostarczają raportów dla lepszej analizy.

Jeżeli nie wiesz, od czego zacząć, jeśli chodzi o analizę bezpieczeństwa aplikacji, pracuj z prostymi regułami bezpieczeństwa, które są uznawane przez społeczności takie jak **OWASP** (ang. *Open Web Application Security Project*) (https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project), którą widzieliśmy w rozdziale 13., „Testy bezpieczeństwa i wydajności”. Możesz skorzystać z narzędzia **Zed Attack Proxy (ZAP)**, które wykorzystuje 10 reguł do przeprowadzania testów bezpieczeństwa aplikacji webowej.

Oto kilka dobrych praktyk dotyczących integracji bezpieczeństwa z kulturą DevOps w celu osiągnięcia kultury DevSecOps. Przyjrzymy się teraz kilku dobrym praktykom monitorowania.

Monitorowanie systemu

Jednym z głównych elementów sukcesu kultury DevOps jest wdrożenie narzędzi, które będą na bieżąco monitorować stan systemu i aplikacji. Monitorowanie musi być realizowane na wszystkich poziomach systemu, angażując wszystkie zespoły, aby otrzymać aplikację z rzeczywistą wartością dodaną dla użytkownika końcowego.

Pierwszym elementem, który można monitorować, jest sama aplikacja — poprzez jak najszybsze wdrożenie systemu rejestrowania lub śledzenia, który będzie służył gromadzeniu informacji o korzystaniu z aplikacji. Następnie będziemy mierzyć i monitorować stan infrastruktury, czyli pamięć **RAM** (ang. *random-access memory*), wykorzystanie **procesorów** (ang. *central processing unit* — **CPU**) **maszyn wirtualnych** (ang. *virtual machines* — **VM**) czy przepustowość sieci. Ostatnim elementem, który należy monitorować, jest stan procesów DevOps. Dlatego ważne jest posiadanie metryk dotyczących wykonania potoków CI/CD, takich jak informacje o czasie wykonania potoków lub o liczbie potoków, które zostały wykonane pomyślnie lub nie. Na podstawie tych danych możemy np. określić szybkość wdrażania aplikacji.

Istnieje wiele narzędzi do monitorowania, takich jak Prometheus, Grafana, New Relic, Nagios i inne, które są zintegrowane z różnymi dostawcami chmury, takimi jak Azure Application Insights lub Azure Monitor Logs.

Jeśli chodzi o dobre praktyki monitorowania, powiedziałbym, że ważne jest ukierunkowanie na **kluczowe wskaźniki efektywności** (ang. *key performance indicators* — **KPI**), które są dla Ciebie niezbędne i łatwe do analizy. Nie ma sensu posiadać systemu monitorującego, który przechwytyuje wiele danych, lub aplikacji, która zapisuje wiele logów, ponieważ jest to zbyt czasochłonne, jeśli chodzi o analizę tych informacji. Ponadto, biorąc pod uwagę ilość przechwytywanych danych, musimy zadbać o ich przechowywanie. Należy ocenić czas przechowywania danych i skonsultować się z różnymi zespołami. Zbyt duża retencja może spowodować nasycenie pojemności maszyn wirtualnych lub wysokie koszty zarządzanych komponentów w chmurze, a przy zbyt małej retencji historia logów jest krótsza, przez co możesz stracić kontrolę nad wszelkimi problemami.

Wreszcie wybierając narzędzie, musisz się upewnić, że chroni ono wszystkie przechwytywane dane, że pulpity nawigacyjne dostarczane przez narzędzie są wystarczająco zrozumiałe dla wszystkich członków zespołu i że jest ono zintegrowane z procesem DevOps.

Wyjaśniliśmy, że monitorowanie to praktyka, którą należy zintegrować z kulturą DevOps, biorąc pod uwagę kilka punktów dobrych praktyk, które mogą poprawić komunikację między Dev i Ops oraz jakość produktu dla użytkowników końcowych.

Po zapoznaniu się z najlepszymi praktykami DevOps w zakresie automatyzacji, potoków CI/CD i monitorowania dokonamy przeglądu praktyk DevOps w zakresie zarządzania projektami i organizacji zespołu.

Ewoluuujące zarządzanie projektami

Wcześniej omawialiśmy kilka dobrych praktyk DevOps, które można zastosować w projektach, ale wszystko to można wdrożyć i zrealizować tylko poprzez zmianę sposobu zarządzania projektami i organizacji zespołów.

Oto kilka dobrych praktyk, które mogą ułatwić wdrożenie kultury DevOps w zarządzaniu projektami w firmach.

Przede wszystkim należy pamiętać, że kultura DevOps ma sens tylko przy wdrażaniu praktyk deweloperskich i dostarczających, które pozwolą na dostarczanie aplikacji w krótkich cyklach wdrożeniowych. Dlatego aby projekty miały zastosowanie, muszą być zarządzane w krótkich cyklach. Aby to osiągnąć, jedną z najbardziej odpowiednich metod zarządzania projektami do zastosowania kultury DevOps, która sprawdziła się w ostatnich latach, jest metoda zwinna (ang. *agile*), która wykorzystuje sprinty (krótkie cykle trwające od 2 do 3 tygodni) z przyrostowymi, iteracyjnymi wdrożeniami i ze ścisłą współpracą między deweloperami.

Kultura DevOps po prostu rozszerza metodologię zwinną, promując współpracę między kilkoma domenami (Dev/Ops/bezpieczeństwo/testerzy).

Uwaga

Aby dowiedzieć się więcej o metodzie zwinnej i jej różnych frameworkach (np. scrum i **programowanie ekstremalne XP** — ang. *Extreme Programming*), polecam stronę <http://agilemethodology.org/>, która zawiera dużo dokumentacji.

Ponadto by lepiej stosować implementacje DevOps, ważne jest, aby zmienić swoją organizację, nie mając już zespołów zorganizowanych według obszarów wiedzy, takich jak zespół programistów, inny zespół operacyjny i zespół testerów.

Problem z tym modelem organizacyjnym polega na tym, że zespoły są podzielone na sekcje, co skutkuje brakiem komunikacji (ang. *wall of confusion*). Oznacza to, że różne zespoły mają różne cele, co hamuje stosowanie dobrych praktyk dla kultury DevOps.

Jednym z modeli pozwalających na lepszą komunikację jest organizacja tzw. zespołu *feature team* z multidyscyplinarnymi zespołami projektowymi złożonymi z ludzi ze

wszystkich dziedzin. W zespole mamy programistów, personel operacyjny i testerów, a wszyscy ci ludzie pracują w tym samym celu.

Jeśli chcesz dowiedzieć się więcej o transformacji DevOps Microsoftu, sugeruję obejrzenie prezentacji Donovan Brown na stronie <https://www.agilealliance.org/resources/sessions/microsoft-devops-transformation-donovan-brown/>, która wyjaśnia, w jaki sposób Microsoft zmienił swoją organizację, aby dostosować się do kultury DevOps i stale ulepszać swoje produkty, biorąc pod uwagę potrzeby użytkowników.

Właśnie dowiedzieliśmy się, że wdrożenie kultury DevOps w firmach wymaga zmian organizacyjnych, w tym zwinnego zarządzania projektami i składu multidyscyplinarnych zespołów.

Podsumowanie

W ostatnim rozdziale przekonaliśmy się, że wdrożenie kultury DevOps w ramach projektów wymaga zastosowania najlepszych praktyk w zakresie automatyzacji wszystkich zadań, odpowiedniego doboru narzędzi, mniej monolitycznej architektury projektu i wdrożenia monitoringu.

Na dużą skalę dla organizacji zespołów i całej firmy widzieliśmy, że metoda zwinna, jak również multidyscyplinarne zespoły silnie przyczyniają się do wdrażania kultury DevOps.

Kończąc tę książkę, radzę wszystkim czytelnikom, którzy przyjmują praktyki DevOps, wdrażać je i monitorować w małych projektach oraz zaczynać od korzystania z narzędzi, które są im najbardziej znane i dostępne. Następnie, gdy Twój proces DevOps będzie działał prawidłowo, możesz rozszerzyć go na większe projekty.

Pytania

1. Jakie są zalety automatyzacji wdrażania?
2. Czy do wykonania IaC konieczne jest użycie Terraform?
3. Co należy zrobić, aby poprawić bezpieczeństwo w procesach DevOps?
4. Czy monitoring dotyczy tylko monitoringu stanu infrastruktury?
5. Jaką dobrą praktykę należy wdrożyć w architekturze aplikacji?
6. W jaki sposób tworzone są zespoły w organizacji DevOps?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej na temat najlepszych praktyk DevOps, oto kilka artykułów, które Ci w tym pomogą:

- *16 Best Practices Of CI/CD Pipeline To Speed Test Automation* — <https://www.lambdatest.com/blog/16-best-practices-of-ci-cd-pipeline-to-speed-test-automation/>.
- *How To Implement Continuous Testing In DevOps Like A Pro?* — <https://www.lambdatest.com/blog/how-to-implement-continuous-testing-in-devops-like-a-pro/>.
- *Secure DevOps* — <https://www.microsoft.com/en-us/securityengineering/devsecops>.
- *9 Pillars of Continuous Security Best Practices* — <https://devops.com/9-pillars-of-continuous-security-best-practices/>.
- *Top 5 Best Practices for DevOps Monitoring* — <https://devops.com/top-5-best-practices-devops-monitoring/>.
- *10 Pitfalls to Avoid when Implementing DevOps* — <https://opensource.com/article/19/9/pitfalls-avoid-devops>.

Odpowiedzi |

Rozdział 1. Kultura DevOps i praktyki kodowania infrastruktury

1. DevOps to skrót utworzony od słów *Development* i *Operations*.
2. DevOps to termin reprezentujący kulturę.
3. Trzy osie kultury DevOps to współpraca, proces i narzędzia.
4. Celem ciągłej integracji jest uzyskanie szybkiej informacji zwrotnej na temat jakości kodu archiwizowanego przez członków zespołu.
5. Różnica między ciągłym dostarczaniem a ciągłym wdrażaniem polega na tym, że w przypadku ciągłego dostarczania inicjowanie wdrożenia w środowisku produkcyjnym odbywa się ręcznie, podczas gdy w przypadku ciągłego wdrażania odbywa się to automatycznie.
6. Podejście *infrastruktura jako kod* polega na tworzeniu kodu zasobów składających się na infrastrukturę.

Rozdział 2. Udostępnianie infrastruktury chmury za pomocą Terraform

1. Językiem używanym przez Terraform jest HCL (ang. *HashiCorp Configuration Language*).
2. Rolą Terraform jest dostarczenie narzędzia zgodnego z filozofią *infrastruktura jako kod*.
3. Nie. Terraform nie jest narzędziem do tworzenia skryptów.
4. Komendą pozwalającą na wyświetlenie zainstalowanej wersji jest `terraform version`.
5. Nazwą obiektu platformy Azure, który łączy Terraform z platformą Azure, jest jednostka usługi platformy Azure (Azure Service Principal).

6. Trzy główne polecenia Terraform to `terraform init`, `terraform plan` i `terraform apply`.
7. Komenda Terraform, która pozwala nam usuwać zasoby, to `terraform destroy`.
8. Dodajemy opcję `--auto-approve` do polecenia `terraform apply`.
9. Celem pliku stanu Terraform jest zachowanie zasobów i ich właściwości przez cały czas wykonywania Terraform.
10. Nie, przechowywanie pliku stanu Terraform lokalnie nie jest dobrą praktyką; musi być przechowywany w chronionym zdalnym zapleczu.

Rozdział 3. Używanie Ansible do konfigurowania infrastruktury IaaS

1. Rolą Ansible opisaną szczegółowo w tym rozdziale jest automatyzacja konfiguracji maszyny wirtualnej.
2. Nie. Nie możemy zainstalować Ansible w systemie operacyjnym Windows.
3. Dwa artefakty omówione w tym rozdziale, których Ansible potrzebuje do działania, to plik inwentarza i `playbook`.
4. Opcjonalnym parametrem jest `--check`.
5. Nazwa narzędzia używanego do szyfrowania i deszyfrowania danych Ansible to Ansible Vault.
6. W przypadku korzystania z dynamicznej inwentaryzacji na platformie Azure skrypt jest oparty na tagach maszyn wirtualnych, które służą do zwracania listy maszyn wirtualnych.

Rozdział 4. Optymalizacja wdrażania infrastruktury za pomocą Packera

1. Istnieją dwa sposoby instalacji Packera: ręcznie lub za pomocą skryptu.
2. Obowiązkowymi sekcjami szablonu programu Packer, które są używane do tworzenia obrazu maszyny wirtualnej na platformie Azure, są `builders` i `provisioners`.

3. Polecenie używane do sprawdzania poprawności szablonu Packera to `packer validate`.
4. Polecenie używane do generowania obrazu Packera to `packer build`.

Rozdział 5. Tworzenie środowiska programistycznego z Vagrantem

1. Rolą Vagranta jest tworzenie lokalnego środowiska programistycznego.
2. Komenda Vagranta do tworzenia maszyny wirtualnej to `vagrant up`.
3. Komenda Vagranta służąca do łączenia się z maszyną wirtualną za pomocą SSH to `vagrant ssh`.

Rozdział 6. Zarządzanie kodem źródłowym za pomocą Gita

1. Git to rozproszony system kontroli wersji (ang. *distributed version control system* — DVCS).
2. Polecenie służące do inicjalizacji repozytorium to `git init`.
3. Artefaktem jest zatwierdzenie (ang. *commit*) polegające na zapisaniu części kodu.
4. Komendą pozwalającą na zapisanie kodu w lokalnym repozytorium jest `git commit`.
5. Komendą pozwalającą na wysłanie kodu do zdalnego repozytorium jest `git push`.
6. Komendą, która pozwala zaktualizować Twoje lokalne repozytorium na podstawie zdalnego repozytorium, jest `git pull`.
7. Gałąź (ang. *branch*) to mechanizm, który pozwala wyizolować kod.
8. Gitflow to model zarządzania gałęziami w Gicie.

Rozdział 7. Ciągła integracja i ciągłe wdrażanie

1. Warunkiem wstępnym skonfigurowania potoku CI jest posiadanie jego kodu w menedżerze kontroli kodu źródłowego.
2. Potok CI jest uruchamiany za każdym razem, gdy członek zespołu zatwierdzi/przekáže kod.
3. Menedżer pakietów to centralne repozytorium używane do centralizacji i udostępniania pakietów, bibliotek programistycznych, narzędzi lub oprogramowania.
4. Menedżer pakietów NuGet umożliwia przechowywanie bibliotek/frameworków .NET.
5. Usługa Azure Artifacts jest zintegrowana z usługą Azure DevOps.
6. Jest to narzędzie lokalne, które należy zainstalować na serwerze.
7. W Azure DevOps usługą zarządzającą potokami CI/CD jest Azure Pipelines.
8. Usługi GitLaba składają się z menedżera kodu źródłowego, menedżera potoku CI/CD i pulpitu do zarządzania projektami.
9. W GitLab CI potok CI jest zbudowany w pliku YAML o nazwie *.gitlab-ci.yml*.

Rozdział 8. Wdrażanie infrastruktury jako kodu za pomocą potoku CI/CD

1. Używanym narzędziem jest Azure Pipelines.
2. Kolejność dostarczania narzędzi: Packer, następnie Terraform i wreszcie Ansible.

Rozdział 9. Konteneryzacja aplikacji za pomocą Dockera

1. Docker Hub to publiczny rejestr obrazów platformy Docker.
2. Podstawowym elementem jest plik Dockerfile.
3. FROM to instrukcja.

4. Polecenie tworzenia obrazu Dockera to `docker build`.
5. Polecenie tworzenia instancji kontenera Dockera to `docker run`.
6. Komenda Dockera służąca do publikowania obrazu to `docker publish`.

Rozdział 10. Efektywne zarządzanie kontenerami za pomocą Kubernetesa

1. Rolą Kubernetesa jest zarządzanie kontenerami.
2. W Kubernetesie wszystkie obiekty są zapisywane w plikach specyfikacji YAML.
3. Narzędzie Kubernetesa CLI to **kubectl**.
4. Polecenie, które stosuje wdrożenie w K8s, to `kubectl apply`.
5. Helm jest menedżerem pakietów dla Kubernetesa.
6. Azure Kubernetes Services to zarządzany klaster Kubernetesa na platformie Azure.

Rozdział 11. Testowanie interfejsów API za pomocą Postmana

1. Postman to narzędzie umożliwiające wykonywanie testów API.
2. Pierwszym elementem, który jest tworzony, jest kolekcja.
3. W Postmanie konfiguracja API znajduje się w żądaniu.
4. Narzędzie Collection Runner umożliwia wykonanie wszystkich żądań w kolekcji.
5. Newman to narzędzie wiersza poleceń, które wykonuje testy Postmana w potoku CI/CD.

Rozdział 12. Statyczna analiza kodu za pomocą SonarQube

1. SonarQube jest tworzony w języku Java.
2. Aby zainstalować SonarQube, konieczne jest posiadanie zainstalowanej Javy.

3. SonarQube to narzędzie służące do statycznej analizy kodu.
4. SonarLint umożliwia programistom przeprowadzanie analizy kodu podczas tworzenia kodu.

Rozdział 13. Testy bezpieczeństwa i wydajności

1. ZAP nie jest narzędziem służącym do analizy kodu źródłowego aplikacji.
2. W Postmanie metryką wydajności jest czas wykonania każdego żądania.

Rozdział 14. Bezpieczeństwo w procesie DevOps z wykorzystaniem DevSecOps

1. Jego rolą jest testowanie zgodności systemu lub infrastruktury.
2. Polecenie to `inspec exec`.
3. Vault to narzędzie rozwijane przez HashiCorp.
4. Polecenie to `vault server -dev`.
5. Kiedy jest zainstalowany lokalnie, można go używać tylko do programowania i testowania.
6. W tym trybie dane są zapisywane w pamięci.

Rozdział 15. Skrócenie czasu przestoju wdrażania

1. Opcją Terraform, która służy do skrócenia przestoju, jest `create_before_destroy`.
2. Zielono-niebieska infrastruktura wdrożeniowa składa się z jednego niebieskiego i jednego zielonego środowiska oraz routera lub systemu równoważenia obciążenia (ang. *load balancer*).
3. Wzorce postępowania to *Canary release* i *Dark launch*.

4. Komponenty platformy Azure, które umożliwiają zielono-niebieskie wdrożenie, to sloty usług aplikacji i usługa Azure Traffic Manager.
5. Rolą flag funkcjonalności jest włączanie lub wyłączanie funkcji aplikacji bez konieczności jej ponownego wdrażania.
6. *Feature toggle* to prosta flaga funkcjonalności, framework typu open source dla aplikacji .NET.
7. LaunchDarkly to rozwiązanie SaaS, które uwalnia Cię od jakiegokolwiek instalacji.

Rozdział 16. DevOps dla projektów open source

1. Aby zmodyfikować kod innego repozytorium, musisz utworzyć rozwidlenie (ang. *fork*) tego repozytorium.
2. Elementem umożliwiającym scalenie kodu jest *pull request*.
3. Plik *CHANGELOG.md* umożliwia wyświetlenie informacji o wydaniu oprogramowania.
4. Wydanie aplikacji jest połączone z tagiem Gita.
5. Używane narzędzia to SonarCloud i WhiteSource Bolt.
6. Luki w zabezpieczeniach są wymienione w zakładce *Issues*.

Rozdział 17. Najlepsze praktyki DevOps

1. Automatyzacja wdrażania eliminuje błędy i skraca cykle wdrażania.
2. Można użyć dowolnego narzędzia, które pozwala na oskryptowanie konfiguracji infrastruktury.
3. Zespoły bezpieczeństwa muszą być zintegrowane z zespołami Dev i Ops.
4. Monitorowanie dotyczy aplikacji, infrastruktury i potoków CI/CD.
5. W aplikacji dobrą praktyką jest oddzielenie funkcji lub domen aplikacji, aby otrzymać kod łatwy do wdrożenia.
6. W strukturze organizacyjnej DevOps zespoły są multidyscyplinarne.

Skorowidz |

A

- ACI, Azure Container Instances, 285, 305
 - wdrażanie kontenera, 285
 - Docker Compose, 300
- ACR, Azure Container Registry, 283, 305
 - obraz Dockera, 285
 - repozytorium Helma, 325
- Active Directory, 55
- agent hostowany samodzielnie, 223
- AKS, Azure Kubernetes Service, 305, 325
 - konfigurowanie pliku kubeconfig, 327
 - logi w czasie rzeczywistym, 332
 - monitorowanie, 328
 - skalowanie, 329
 - tworzenie usługi, 326
 - zalety, 328
- Aminator, 40
- analiza kodu
 - statyczna, 369
 - w czasie rzeczywistym, 378
 - za pomocą
 - SonarCloud, 468
 - SonarLint, 378
 - SonarQube, 369
- Ansible
 - artefakty, 89
 - hosty, 89
 - instalowanie za pomocą skryptu, 86
 - integrowanie z Azure Cloud Shell, 88
 - inwentarz, 89
 - maszyn wirtualnych, 113
 - katalog roles, 99, 105
 - konfigurowanie, 90
 - hostów, 94
 - infrastruktury IaaS, 84
 - korzystanie z podglądu, 102
 - moduły, 98
 - opcja testowa pracy, 102
 - PLAY Recap, 101
 - playbook, 90
 - integracja z szablonem, 135
 - tworzenie, 134
 - ulepszanie, 99
 - uruchamianie, 97
 - wykonanie testu, 103
 - wykonywanie, 101, 102
 - plik inwentarza, 93
 - testowanie, 95
 - wykonanie, 114
 - tworzenie szablonów Packera, 133
 - uruchamianie w Azure Pipelines, 255
 - używanie zmiennych, 105
 - wyświetlenie konfiguracji, 92
 - wywołanie MySQL, 107
 - zwiększanie poziomu logowania, 104
- Ansible Vault, 410
 - ochrona danych, 104
 - odszyfrowanie pliku, 109
 - szyfrowanie pliku, 108
- API, 297, 339
- aplikacje mikrousług, 305
- App Service, 433
 - sloty wdrażania, 433
- Artifact Hub, 317
 - pakiet wordpress, 319
- automatyzacja wdrożeń, 480
- Azure
 - adres IP SonarQube, 375
 - brak grupy zasobów, 67
 - klucz dostępu do magazynu, 80
 - konfigurowanie
 - platformy, 403
 - Terraform, 55
 - lista rejestracji aplikacji, 57
 - odzyskiwanie hasła SonarQube, 376
 - strona logowania, 54, 59
 - tag role, 111

- Azure
 - tworzenie
 - jednostki SP, 56
 - obrazu platformy, 130
 - SonarQube, 374
 - udostępnianie zasoby, 72
 - uwierzytelnianie, 136
 - używanie
 - Ansible, 88
 - Packera, 123
 - wdrażanie
 - SonarQube, 375
 - zielono-niebieskie, 432
 - Azure Artifacts, 206, 217
 - Azure Boards, 217
 - Azure CLI, 404
 - Azure Cloud Shell, 53, 54, 403
 - Azure DevOps
 - rozszerzenie SonarQube, 382
 - Azure Key Vault, 410
 - Azure Pipelines, 217
 - definiowanie
 - wersji, 233
 - aplikacji, 229
 - dodawanie
 - artefaktu, 230
 - zadania, 225
 - dzienniki
 - Ansible, 259
 - potoku, 255
 - Terraform, 259
 - edycja nazwy
 - usługi aplikacji, 233
 - wersji, 232
 - instalowanie npm, 364
 - klonowanie, 232
 - konfigurowanie
 - kodu źródłowego, 224
 - potoku CI/CD, 287
 - puli agentów, 224
 - wersji, 231
 - lista zadań, 223
 - przeglądanie artefaktów, 228
 - stan wdrożenia, 234
 - sablon Azure App Service deployment, 229
 - testy
 - podsumowanie wykonania, 227
 - publikowanie wyników, 365, 366
 - tworzenie potoku, 254
 - CD, 228
 - CI, 219, 381
 - CI/CD, 329
 - YAML, 236
 - tworzenie
 - wersji, 234
 - wydania, 363
 - uruchamianie, 227
 - Ansible, 255
 - Newmana, 365
 - Packera, 252
 - potoku, 239
 - Terraform, 255
 - włączanie
 - CI, 226
 - CD, 230
 - wybór
 - pliku YAML, 237, 238
 - repozytorium, 221, 237
 - szablonu, 222
 - źródła GitHuba, 288
 - wykonanie zadania, 239
 - zakładka Variables, 222
 - zapisywanie i kolejkovanie potoku, 226
 - zarządzanie potokami CI/CD, 216
 - zmienne, 255, 259
 - Azure Repos, 164, 181, 217
 - dodawanie pliku, 188
 - formularz PR, 194
 - import repozytorium, 219
 - lista gałęzi, 192
 - menu, 183
 - menu Import, 218
 - tworzenie PR, 193
 - zatwierdzenie PR, 194
 - Azure Test Plans, 217
 - Azure Traffic Manager, 434
 - konfigurowanie usługi, 435
 - punkty końcowe, 436
- ## B
- BDD, behavior-driven development, 485
 - BDD, behavior-driven design, 26
 - bezpieczeństwo, 399

C

CD, continuous delivery, 25, 31, 202
 tworzenie potoku, 228
Chocolatey, 122, 175, 402
CI, continuous integration, 25, 29, 200
 korzystanie z GitHub Actions, 465
 menedżer
 konfiguracji, 203
 repozytorium, 202
 obszary pośrednie, 202
 serwer, 202
 tworzenie potoku, 219, 245
 dla SonarQube, 381
 wykonywanie SonarQube, 380
CI/CD, ciągła integracja/ciągłe wdrażanie, 163, 484
 korzystanie
 z Azure Pipelines, 216
 z GitLab CI, 240
 z Jenkinsa, 208
 z Newmana, 361
 menedżer pakietów, 203
 narzędzia, 203
 potok
 w pliku YAML, 234
 w trybie UI, 234
 przebieg pracy, 201
 tworzenie potoku
 dla kontenera, 287
 dla Kubernetesa, 329
 wdrażanie
 infrastruktury jako kodu, 251
 kontenera do ACI, 285
ciągła integracja, CI, 25, 29, 200
ciągłe dostarczanie, CD, 25, 31, 202
ciągłe wdrażanie, 33
Collection Runner, 393

D

demon Dockera, 267
DevOps, development-operations, 25, 399
 najlepsze praktyki, 479
DevSecOps, 486
Docker, 41, 265
 dokumentacja instalacji, 271
 instalowanie, 268
 kontener, 273

 lista obrazów, 280
 narzędzia wiersza poleceń, 294
 obraz, 273
 tworzenie, 276, 278
 w ACR, 285
 plik Dockerfile, 273
 testowanie kontenera, 279
 wolumin, 273
 wysyłanie obrazu
 do rejestru prywatnego, 283
 do rejestru publicznego, 279

Docker Compose, 296
 instalowanie, 297
 lista kontenerów, 300
 plik konfiguracyjny, 298
 wdrażanie kontenerów w ACI, 300
 wykonywanie, 299
Docker Desktop, 269
 komponenty Docker Compose, 300
 logowanie do Docker Huba, 271
 włączanie Kubernetesa, 308
Docker Hub, 267
 publikowanie obrazu Dockera, 279
 sekcja Explore, 268
 strona logowania, 268
 tworzenie konta, 267
 wyszukiwanie obrazu, 282
dostęp do magazynu Azure, 80
dynamiczny plik inwentarza, 256

E

ewolucja kultury, 44

F

flagi funkcjonalności, feature flags, 432, 436
 implementacja, 439
 prezentacja przełączania, 442
 tworzenie, 444
 w LaunchDarkly, 444, 447
format
 HCL, 140
 JSON, 351, 424
 YAML, 234, 297, 482
FQDN, fully qualified domain name, 93

G

- Git, 164
 - gałąź master, 177–180
 - gałęzie, 177
 - przełączanie, 191
 - scalanie, 195
 - tworzenie, 190, 192, 197
 - zarządzanie, 195
 - instalowanie, 166
 - klonowanie, 176
 - konfigurowanie, 176
 - narzędzie
 - GitKraken, 196
 - Sourcetree, 197
 - polecenia, 177
 - przepływ danych, 182
 - repozytorium, 176
 - konfigurowanie, 182
 - tworzenie, 182
 - wersjonowanie kodu w Azure Repos, 218
 - wyświetlanie wersji, 174
 - wzorec Gitflow, 195
 - zatwierdzanie, 176
- Gitflow, 195
- GitHub
 - aktywacja Issues, 475
 - dziennik zmian, 459, 460
 - funkcje i narzędzia, 451
 - konfigurowanie webhooka, 210
 - link do wydania, 462
 - lista żądań pobierania, 457
 - operacja scalania, merge, 455
 - repozytoria publiczne, 451
 - rozwidlanie repozytorium, 454
 - rozwijanie projektu, 453
 - tworzenie
 - repozytorium, 451
 - wydania, 463
 - udostępnianie plików binarnych, 461
 - żądanie pobierania, pull request, 455
- GitHub Actions, 464
 - kod źródłowy przepływu pracy, 467
 - szablony przepływów pracy, 465
 - tworzenie potoku CI, 465
 - wybór szablonu Node.js, 466
 - wykonanie przepływu pracy, 468
 - zarządzanie potokami CI/CD, 464

GitLab

- importowanie kodu, 244
 - konfigurowanie projektu, 243
 - rejestracja, 241
 - repozytorium, 245
 - szablon projektu, 243
 - tworzenie
 - nowego projektu, 242
 - potoku CI, 245
 - uruchamianie CI/CD, 244
 - uwierzytelnianie, 241
 - wykonanie potoku CI, 247
 - zarządzanie kodem źródłowym, 242
- GitLab CI, 240

H

- HashiCorp Packer, 40
- hasła
 - zapisywanie w Vault, 415
- Helm, 316
 - charty, 317
 - publikowanie, 323
 - tworzenie, 321
 - instalowanie, 316
 - aplikacji, 320
 - pakiety
 - lista, 320, 322
 - tworzenie, 323
 - wyszukiwanie, 318
- hosty, 89

I

- IaaS, 84
- IaC, Infrastructure as Code, 25, 35, 251
 - infrastruktura z kontenerami, 41
 - konfigurowanie serwera, 40
 - narzędzia, 36
 - typy
 - deklaratywne, 37
 - programowe, 38
 - skryptowe, 36
 - udostępnianie infrastruktury, 39
 - wdrażanie
 - infrastruktury, 39
 - w Kubernetesie, 42
- infrastruktura
 - IaaS, 84
 - jako kod, IaC, 25, 35, 251

InSpec, 400
 instalowanie, 402
 konfigurowanie platformy Azure, 403
 sprawdzanie profilu, 409
 testy
 tworzenie, 405
 wykonanie, 409
 zgodności, 406
instalacja
 Ansible, 86
 Docker Compose, 297
 Dockera, 268
 Gita, 166
 Helma, 316
 InSpec, 402
 Jenkinsa, 208
 Kubernetesa, 307
 Newmana, 355
 npm, 364
 Packera, 119
 Postmana, 341
 SonarQube, 372
 Terraform, 48
 Vagranta, 148
 Vault, 411
interfejs
 programowania aplikacji, API, 297, 339
 użytkownika, UI, 234
inwentarz, 89
 dynamiczny, 92, 256
 statyczny, 92
izolacja kodu, 190

J

Java Runtime Environment, JRE, 370
jednostka Azure SP, 55
Jenkins
 implementacja CI/CD, 208
 instalowanie, 208
 konfigurowanie, 208
 GitHuba, 213
 webhooka GitHuba, 211
 zadania CI, 211
 konsola zadań, 216
 w Azure Marketplace, 209
 wtyczka integracyjna GitHuba, 209
 zadania
 historia wykonywania, 215
 tworzenie, 212
 uruchamianie, 214
 wykonywanie, 215

język Markdown, 459
języki
 IaC, 36
 skryptowe, 36
JSON, 351, 424

K

KeyPass, 410
klient Dockera, 267
konfiguracja
 Ansible, 90
 Docker Desktop, 270
 dostawcy Terraform, 57
 Gita, 176
 infrastruktury IaaS, 84
 Jenkinsa, 208
 maszyn wirtualnych, 40, 84
 Terraform, 55
 webhooka GitHuba, 210
 zadania CI, 211
konteneryzacja aplikacji, 265
kontrola kodu źródłowego
 rozproszona, 165
 scentralizowana, 165
KPI, key performance indicators, 487
Kubernetes, 42
 architektura, 307
 instalowanie, 307
 pulpitu nawigacyjnego, 309
 lista zasobów, 312
 menedżer pakietów Helm, 316
 monitorowanie
 aplikacji, 330
 metryk, 330, 333
 narzędzie
 Grafana, 334
 Lens, 332
 Octant, 332
 Prometheus, 334
 pody, 307
 serwer główny, 306
 uwierzytelnianie, 311
 wdrażanie aplikacji, 312
 węzły, nodes, 306
 robocze, 307
 zarządzanie kontenerami, 305
kultura współpracy, 26

L

LastPass, 410
LaunchDarkly, 443
 flagi funkcjonalności, 446, 447
luki w zabezpieczeniach, 473

M

magazyn Azure, 80
maszyna wirtualna, VM, 84
 publiczny adres IP, 260
 tworzenie, 156
 uzyskiwanie połączenia, 158
 wykonywanie poleceń, 159
menedżer
 hasel AWS, 410
 kodu źródłowego, SCM, 30
 konfiguracji, 32, 203
 kontroli wersji, VCM, 163
 pakietów, 32, 203
 Helm, 316
 Nexusa, 206
 NuGet, 204
 poświadczeń Gita, 172
 potoków CI/CD, 464
 repozytorium, 202
metoda zwinna, agile, 488
moduły Ansible, 98
monitorowanie systemu, 487

N

narzędzia, 27
 do konfigurowania potoku CI/CD, 203
 Gitflow, 196
 IaC, 36
Newman, 354
 instalowanie, 355
 integracja z potokiem CI/CD, 361
 publikowanie wyniku testów, 366
 w Azure Pipelines
 tworzenie wydania, 363
 uruchamianie, 365
nginx, 260, 294
npm, 205
 instalowanie w Azure Pipelines, 364

O

obraz
 Dockera, 276
 maszyny wirtualnej, 138, 143
 Azure, 139
 Terraform, 295
ochrona danych, 410
oprogramowanie jako usługa, SaaS, 202
OWASP, 387, 486
OWASP ZAP, 389
 automatyczne skanowanie, 390
 wynik skanowania, 391

P

PaaS, Platform-as-a-Service, 39
PaC, Pipeline as Code, 246, 482
Packer
 dane wyjściowe, 139
 instalowanie
 ręczne, 119
 za pomocą skryptu, 120–123
 integrowanie z Azure Cloud Shell, 123
 obraz maszyny wirtualnej, 138, 139
 sprawdzanie instalacji, 124
 szablony
 dla maszyn wirtualnych, 125
 obraz platformy Azure, 130
 struktura, 125–130
 użycie Ansible, 133
 w formacie HCL, 140
 walidacja, 137
 zmienne szablonu, 137
tymczasowa
 grupa zasobów, 139
 maszyna wirtualna, 138
uruchamianie, 136
 w Azure Pipelines, 252
 uwierzytelnianie w Azure, 136
paczki, charts, 316
pakiety
 Azure Artifacts, 207
 NuGet, 205
 uniwersalne, 207
 wordpress, 319
piaskownica, sandbox, 58
platforma jako usługa, PaaS, 39

- playbook, 90
 - integracja z szablonem, 135
 - tworzenie, 97, 134
 - ulepszanie, 99
 - uruchamianie, 97
 - wykonywanie, 101, 102
- plik
 - Dockerfile, 273
 - instrukcje, 274, 275
 - dziennika zmian, 459
 - inwentarza Ansible
 - dynamiczny, 110
 - konfigurowanie hostów, 94
 - statyczny, 93
 - testowanie, 95
 - konfiguracyjny
 - Ansible, 91
 - Vagranta, 152, 154
 - kubeconfig, 311, 327
 - main.tf, 286
 - profilu InSpec, 405
 - Readme.md, 185
 - stanu infrastruktury, 77
- pliki
 - JSON, 356, 424
 - YAML, 234, 297, 313
- polecenia Gita, 177
- polecenie
 - ansible
 - help, 87
 - ping, 96
 - version, 87, 159
 - config, 91
 - playbook, 101
 - docker
 - build, 276, 288
 - compose up, 302
 - help, 272
 - image, 278
 - login, 280
 - ps, 278, 300
 - pull, 295
 - push, 281
 - run, 278, 295
 - tag, 280
 - compose up -d, 299
 - git
 - add, 179, 186
 - branch, 180, 191
 - checkout, 180
 - clone, 178, 188
 - commit, 179, 186, 187
 - init, 178, 184
 - merge, 180
 - pull, 172, 180, 189
 - push, 179, 187
 - remote add, 178, 185
 - status, 186
 - helm
 - create, 321
 - delete, 321
 - help, 317
 - install, 322
 - ls, 322
 - package, 323
 - search, 318
 - inspec exec, 409
 - kubectl, 309
 - apply, 314
 - get pods, 322
 - proxy, 310
 - login, 66
 - newman run, 359
 - packer
 - help, 124
 - version, 124
 - terraform
 - apply, 70, 296, 423
 - destroy, 73
 - fmt, 74
 - init, 68, 81, 295, 423
 - plan, 69, 75, 296, 423
 - validate, 75
 - vagrant
 - destroy, 159
 - help, 156
 - ssh, 158
 - up, 157
 - vault kv get, 417
 - zap-cli
 - active-scan, 391
 - report, 391
- połączenie SSL, 209
- Postman
 - dodawanie zmiennej środowiskowej, 347
 - edytowanie
 - kolekcji, 343
 - żądania, 345
 - eksportowanie
 - kolekcji, 356
 - środowisk, 357

- instalowanie, 341
- kod testów, 350
- konfigurowanie żądania, 345
- rejestracja, 351
- testowanie interfejsów API, 339
- tworzenie
 - kolekcji, 342
 - kolekcji żądań, 340
 - środowiska, 347
 - testów, 349
 - żądania, 343
- uruchamianie testów wydajności, 392
- używanie zmiennej środowiskowej, 348
- wyniki testu, 352
- zakładka Tests, 349

Postman Collection Runner, 352

praktyki, 43

- GitOps, 45

- IaC, 35

proces, 26

- CI/CD Terraform, 76

- ciągłego dostarczania, 33

- ciągłego wdrażania, 34

- ciągłej integracji, 31

- Gita, 181

projektowanie

- architektury systemu, 482

- oparte na testach, TDD, 26, 485

- oparte na zachowaniu, BDD, 26, 485

projekty open source, 449

Pulumi, 38

R

rejestr

- Dockera, 267

- prywatny, 283

repozytorium

- Gita, *Patrz także* Azure Repos

- adres URL, 185

- aktualizacja kodu, 189

- archiwizacja, 187

- inicjowanie, 184

- klonowanie, 188

- konfigurowanie, 182

- pobieranie aktualizacji, 189

- tworzenie, 182

- GitLaba, 245

- Helma, 317, 325

- na GitHubie, 451

- Nexusa OSS, 205

- NuGet, 205

rola, role, 99

rozwidlenie, fork, 453

S

SaaS, software-as-a-service, 202

SCM, Source Code Manager, 30

serwer

- CI, 202

- Vault, 413

SonarCloud

- analiza kodu, 468

SonarLint, 378

- analiza kodu, 378

SonarQube, 370

- analiza kodu, 369, 384

- architektura, 371

- instalowanie

- na Kubernetesie, 377

- na platformie Azure, 373

- ręczne, 372

- za pomocą Dockera, 373

- integrowanie z procesem CI, 380

- w Azure Pipelines, 383

SP, service principal, 55

SSL, Secure Sockets Layer, 209

system kontroli wersji, VCS, 163, *Patrz także* Git

szablon Packera, 252

- playbook Ansible, 135

- sekcja

- builders, 125

- provisioners, 127

- variables, 129

- sprawdzanie poprawności, 137

- tworzenie obrazu platformy Azure, 130

- użycie Ansible, 133

- w formacie HCL, 140

szyfrowanie pliku, 108

Ś

środowisko programistyczne, 147

T

TDD, test-driven design, 26
TDD, test-driven development, 485
Terraform, 38, 47

- cykl życia, 72
 - w procesie CI/CD, 75
- dane wyjściowe, 424
- definiowanie architektury Azure, 60
- dobre praktyki, 63
- formatowanie kodu, 74
- inicjalizowanie, 67
- instalowanie
 - ręczne, 49
 - za pomocą skryptu, 49–52
- integrowanie z Azure Cloud Shell, 53
- jednostka Azure SP, 55
- katalog konfiguracji, 68
- konfiguracja dostawcy, 57
- korzystanie z obrazów Packera, 143
- lista udostępnionych zasobów, 72
- plik stanu, 77
- pobieranie sekretów Vault, 421
- podgląd zmian, 69
- potwierdzenie zmian, 70
- przepływ pracy CI/CD, 76
- skracać czas przestoju, 427
- testowanie, 58
- udostępnianie zasobu ACI, 286
- uruchamianie w Azure Pipelines, 255
- usuwanie infrastruktury, 72
- walidacja konfiguracji, 74
- wdrażanie infrastruktury, 66
- wyświetlanie danych wrażliwych, 424

testowanie interfejsów API, 339
testy

- A/B, 443
- bezpieczeństwa, 386
- integracyjne, 485
- konfiguracji Runnera, 394
- penetracyjne, 387
- wydajności, 386, 392
- zgodności InSpec, 406

topologia IaC, 39
tworzenie

- charta, 321
- kodu Terraform, 286
- kolekcji Postmana, 340, 342
- kontenera obrazu, 278
- maszyny wirtualnej, 156
- obrazu

Dockera, 276

- platformy Azure, 130

pakietu Helma, 323
playbooka, 97

- Ansible, 134

pliku

- Dockerfile, 273
- inwentarza dynamicznego, 110
- inwentarza statycznego, 92
- konfiguracyjnego Vagranta, 152, 154

potoku, 254

- CD, 228
- CI, 219, 245, 381
- CI/CD, 287, 329
- YAML, 234, 236

projektu LaunchDarkly, 444
repozytorium

- Gita, 182
- na GitHubie, 451

szablonów Packera

- dla maszyn wirtualnych, 125
- przy użyciu Ansible, 133
- w formacie HCL, 140

środowiska programistycznego, 147
testów

- InSpec, 405, 406
- Postmana, 349

usługi AKS, 326
żądania, 343

U

UI, user interface, 234
usługa

- ACI, 285
- AKS, 326
- App Service, 432
- Azure
 - Active Directory, 55
 - AD SP, 131, 136
 - Pipelines, 216
 - Repos, 181
 - SP, 55
 - Traffic Manager, 432, 434
- Docker Hub, 267
- zarządzania kluczami, 410

usługi Azure DevOps, 217
uwierzycznianie

- w Azure, 136
- wieloskładnikowe, MFA, 387

V**Vagrant**

- inicjalizacja, 154
- instalowanie
 - ręczne, 148
 - za pomocą skryptu, 150, 151
- tworzenie
 - pliku konfiguracyjnego, 152, 154
 - środowiska programistycznego, 147
- weryfikacja konfiguracji, 156
- wyświetlanie poleceń, 157

Vagrant Box, 152**Vagrant CLI, 156**

- tworzenie maszyny wirtualnej, 156

Vagrant Cloud, 152

- boksy, 153, 154

Vault, 410

- instalowanie
 - automatyczne, 412
 - ręczne, 412
- interfejs webowy, 418
- ochrona poufnych danych, 410
- odczytywanie sekretów, 416
- status, 415
- uruchamianie serwera, 413
- zapisywanie haseł, 415

VCM, version control manager, 163**VCS, version control system, 163****Visual Studio Code, 378, 427**

- rozszerzenie SonarLint, 379

VSTS, Patrz Azure Pipelines**W****walidacja**

- kodu, 74
- szablonu Packera, 137

wdrażanie

- CI, 30
- infrastruktury
 - Azure, 60
 - jako kodu, 251
 - za pomocą Packera, 117
- kontenera do ACI, 285
- zielono-niebieskie, 430, 432

webhook GitHuba, 210**wersja aplikacji, release, 228****WhiteSource Bolt**

- instalowanie, 475
- konfigurowanie, 475
- wykryte problemy, 476
- wykrywanie luk w zabezpieczeniach, 473

wiersz poleceń

- kubectl, 330
- Newmana, 359
- zap-cli, 391

wybór narzędzi, 480**wysyłanie obrazu Dockera**

- do Docker Huba, 279
- do rejestru prywatnego, 283

wyszukiwanie

- pakietów Helma, 318
- pakietu wordpress, 319

wzorce wdrażania, 429**wzorzec**

- Canary release, 430
- Dark launch, 432
- Gitflow, 181, 195

Y**YAML, YAML Ain't Markup Language, 297, 482**

- tworzenie definicji potoku, 234

Z**ZAP, Zed Attack Proxy, 388, 486**

- testy penetracyjne, 387

zapora aplikacji webowej, WAF, 387**zarządzanie**

- kontenerami, 305
- projektami, 488

zdalne zaplecze, remote backend, 78**Ż****żądanie, request, 343**

- pobierania, pull request, 455

PROGRAM PARTNERSKI

— GRUPY HELION —



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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA
Helion 