

OKIEM EKSPERTA

Linux

Podręcznik dewelopera

Rzeczowy przewodnik
po wierszu poleceń
i innych narzędziach

David Cohen, Christian Sturm

Helion

<packt>

Tytuł oryginału: The Software Developer's Guide to Linux: A practical,
no-nonsense guide to using the Linux command line and utilities
as a software developer

Tłumaczenie: Lech Lachowski (rozdz. 1 – 16), Robert Górczyński (wprowadzenie, rozdz. 17)

ISBN: 978-83-289-1755-2

Copyright © Packt Publishing 2024. First published in the English language
under the title 'The Software Developer's Guide to Linux – (9781804616925)'

Polish edition copyright © 2025 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 jakiegokolwiek postaci jest zabronione.
Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie
książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie
praw autorskich niniejszej publikacji.

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

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

https://helion.pl/user/opinie/lipode_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](#)

Spis treści |

O autorach	13
O korektorach merytorycznych	14
Wprowadzenie	15
 ROZDZIAŁ 1	
Jak działa wiersz poleceń?	21
Na początku był... REPL	21
Składnia wiersza poleceń (odczyt)	23
Wiersz poleceń i powłoka	24
Skąd powłoka wie, co uruchomić? (ewaluowanie)	24
Krótka definicja interfejsu POSIX	26
Podstawowe umiejętności wiersza poleceń	27
Podstawy systemu plików Uniksa	27
Bezwzględne i względne ścieżki plików	27
Rozglądanie się po systemie plików — nawigacja w wierszu poleceń	29
Poruszanie się po systemie plików	31
Odczytywanie plików	32
Wprowadzanie zmian	33
Uzyskiwanie pomocy	35
Autouzupełnianie powłoki	36
Podsumowanie	37
 ROZDZIAŁ 2	
Praca z procesami	39
Podstawy procesów	40
Z czego składa się proces Linuksa?	41
Identyfikator procesu (PID)	42
Efektywny identyfikator użytkownika (EUID) i efektywny identyfikator grupy (EGID)	42
Zmienne środowiskowe	42
Katalog roboczy	43

Praktyczne polecenia do pracy z procesami systemu Linux	43
Zaawansowane koncepcje i narzędzia związane z procesami	45
Sygnały	45
Isof — wyświetlanie uchwytów plików otwartych przez proces	48
Dziedziczenie	50
Przegląd — przykładowa sesja rozwiązywania problemów	50
Podsumowanie	52

ROZDZIAŁ 3

Zarządzanie usługami za pomocą usługi systemd	53
Podstawy	54
init	55
Procesy i usługi	55
Polecenia systemctl	55
Sprawdzanie statusu usługi	56
Uruchamianie usługi	57
Zatrzymywanie usługi	57
Restartowanie usługi	57
Ponowne załadowywanie usługi	57
Enable i disable	58
Kilka słów na temat Dockera	59
Podsumowanie	59

ROZDZIAŁ 4

Korzystanie z historii powłoki	60
Historia powłoki	60
Pliki konfiguracyjne powłoki	61
Pliki historii	61
Przeszukiwanie historii powłoki	62
Wyjątki	62
Wykonywanie poprzednich poleceń za pomocą !	63
Ponowne uruchamianie polecenia z tymi samymi argumentami	63
Dołączanie do polecenia jakiegoś polecenia z historii	63
Przeskakiwanie na początek lub koniec bieżącej linii	64
Podsumowanie	64

ROZDZIAŁ 5

Wprowadzenie do plików	65
Pliki w Linuksie — absolutne podstawy	66
Pliki tekstowe	66
Co to jest plik binarny?	66
Znaki zakończenia linii	67
Drzewo systemu plików	67
Podstawowe operacje systemu plików	68
ls	68
pwd	69
cd	69
touch	70
less	70
tail	71
mv	71
cp	72
mkdir	72
rm	72
Edycja plików	72
Typy plików	73
Dowiązania symboliczne	75
Dowiązania twarde	75
Polecenie file	76
Zaawansowane operacje na plikach	76
Wyszukiwanie zawartości pliku za pomocą narzędzia grep	77
Wyszukiwanie plików za pomocą narzędzia find	77
Zaawansowane zagadnienia systemu plików	80
FUSE — jeszcze więcej zabawy z systemem plików Uniksa	81
Podsumowanie	82

ROZDZIAŁ 6

Edycja plików w wierszu poleceń	83
Nano	84
Instalowanie nano	84
Ściągawka z nano	84
Vi(m)	85
Polecenia vi/vima	86
Tryby	86
Wskazówki dotyczące nauki edytora vi(m)	88
Wiązania vima w innym oprogramowaniu	90

Edytowanie pliku, do którego nie masz uprawnień	90
Ustawianie preferowanego edytora	90
Podsumowanie	91

ROZDZIAŁ 7

Użytkownicy i grupy	92
Czym jest użytkownik?	92
Root kontra reszta świata	93
Polecenie sudo	93
Czym jest grupa?	95
Miniprojekt: zarządzanie użytkownikami i grupami	95
Tworzenie użytkownika	95
Tworzenie grupy	96
Modyfikowanie użytkownika	97
Zagadnienia zaawansowane, czyli czym tak naprawdę jest użytkownik?	98
Metadane i atrybuty użytkownika	98
Kilka słów na temat skryptów	100
Podsumowanie	100

ROZDZIAŁ 8

Własność i uprawnienia	102
Odszyfrowywanie długiego listingu	102
Atrybuty pliku	103
Typ pliku	103
Uprawnienia	103
Liczba dowiązań twardych	104
Własność użytkownika	104
Własność grupy	104
Rozmiar pliku	104
Czas modyfikacji	105
Nazwa pliku	105
Własność	105
Uprawnienia	105
Zapisywanie uprawnień za pomocą liczb (ósemkowych)	106
Typowe uprawnienia	107
Zmiana własności (chown) i uprawnień (chmod)	108
chown	108
chmod	108
Podsumowanie	109

ROZDZIAŁ 9**Zarządzanie zainstalowanym oprogramowaniem 111**

Praca z pakietami oprogramowania	112
Aktualizowanie lokalnej pamięci podręcznej stanem repozytorium	113
Wyszukiwanie pakietu	113
Instalowanie pakietu	114
Uaktualnianie wszystkich pakietów, które mają dostępne aktualizacje	114
Usuwanie pakietu (i jego wszelkich zależności, pod warunkiem że nie są wykorzystywane przez inne pakiety)	115
Kwerendowanie zainstalowanych pakietów	115
Wymagana ostrożność — curl bash	116
Kompilowanie zewnętrznego oprogramowania ze źródła	117
Przykład: kompilowanie i instalowanie narzędzia http	118
Podsumowanie	120

ROZDZIAŁ 10**Konfigurowanie oprogramowania 121**

Hierarchia konfiguracji	121
Argumenty wiersza poleceń	123
Zmienne środowiskowe	124
Pliki konfiguracyjne	126
Konfiguracja na poziomie systemu w katalogu /etc/	126
Konfiguracja na poziomie użytkownika w katalogu ~/.config	127
Jednostki systemd	127
Tworzenie własnej usługi	128
Kilka zdań na temat konfiguracji w Dockerze	129
Podsumowanie	130

ROZDZIAŁ 11**Potoki i przekierowanie 131**

Deskryptory plików	132
Do czego odwołują się te deskryptory plików?	132
Przekierowywanie wejścia i wyjścia (praca z deskryptorami plików dla zabawy i potencjalnych korzyści)	133
Przekierowywanie danych wejściowych — <	133
Przekierowywanie danych wyjściowych — >	134
Przekierowywanie błędów za pomocą 2>	135
Łączenie poleceń za pomocą potoków ()	136
Polecenia z wieloma potokami	136

Narzędzia CLI, które należy znać	137
cut	138
sort	138
uniq	139
wc	140
head	140
tail	141
tee	141
awk	141
sed	142
Praktyczne wzorce potoków	142
„Top X” z licznikiem	142
curl bash	143
Filtrowanie i wyszukiwanie za pomocą narzędzia grep	145
grep i tail do monitorowania dzienników	146
find i xargs do wykonywania operacji na grupach plików	146
sort, uniq i odwrotne sortowanie liczbowe do przeprowadzania analizy danych	146
awk i sort do przeformatowywania danych i przetwarzania opartego na polach	147
sed i tee do edytowania i tworzenia kopii zapasowych	148
Zagadnienia zaawansowane: sprawdzanie deskryptorów plików	149
Podsumowanie	151

ROZDZIAŁ 12

Automatyzacja zadań za pomocą skryptów powłoki	152
Dlaczego potrzebujesz podstaw pisania skryptów powłoki Bash?	153
Podstawy	153
Zmienne	153
Pobieranie	154
Porównanie Basha z innymi powłokami	154
Shebangi i wykonywalne pliki tekstowe	155
Typowe ustawienia powłoki Bash (opcje i argumenty)	156
/usr/bin/env	157
Znaki specjalne i znaki ucieczki	157
Podstawianie poleceń	158
Testowanie	158
Operatory testowe	159
[[testowanie plików i łańcuchów znaków]]	159
((testowanie arytmetyczne))	160

Wyrażenia warunkowe: if/then/else	161
if-else	161
Pętle	161
Pętle w stylu C	161
for...in	162
While	162
Eksportowanie zmiennych	162
Funkcje	163
Preferuj zmienne lokalne	163
Przekierowanie wejścia i wyjścia	164
< — przekierowanie wejścia	164
> i >> — przekierowanie wyjścia	164
Zastosowanie 2>&1 do przekierowywania STDERR i STDOUT	165
Składnia interpolacji zmiennej — \${}	165
Ograniczenia skryptów powłoki	166
Podsumowanie	166
Źródła	167

ROZDZIAŁ 13

Bezpieczny dostęp zdalny za pomocą SSH	168
Elementarz kryptografii klucza publicznego	169
Szyfrowanie komunikatów	169
Podpisywanie komunikatów	169
Klucze SSH	170
Wyjątki od zasad	171
Logowanie i uwierzytelnianie	171
Projekt praktyczny: konfigurowanie logowania opartego na kluczu na zdalnym serwerze	172
Krok 1. Otwórz terminal na kliencie SSH (nie na serwerze)	172
Krok 2. Wygeneruj parę kluczy	172
Krok 3. Skopiuj klucz publiczny na serwer	173
Krok 4. Przetestuj to!	173
Konwersja kluczy SSH2 na format OpenSSH	173
Co chcemy osiągnąć?	174
Jak przekonwertować klucz w formacie SSH2 na OpenSSH?	174
Na odwrót: konwersja kluczy OpenSSH na format SSH2	175
Agent SSH	175
Typowe błędy SSH i argument -v (verbose)	176
Przesyłanie plików	177
SFTP	178
SCP	178
Przydatne przykłady	179

Tunele	180
Przekierowanie lokalne	181
Serwer pośredni (proxy)	181
Plik konfiguracyjny	182
Podsumowanie	183

ROZDZIAŁ 14

Kontrola wersji za pomocą Gita

Trochę informacji na temat Gita	184
Czym jest rozproszony system kontroli wersji?	185
Podstawy Gita	185
Pierwsza konfiguracja	186
Inicjalizowanie nowego repozytorium Gita	186
Wprowadzanie i obserwowanie zmian	186
Przechowywanie i zatwierdzanie zmian	186
Opcjonalne dodawanie zdalnego repozytorium Gita	187
Wysyłanie i pobieranie	187
Klonowanie repozytorium	187
Terminologia	188
Repozytorium	188
Gałąź	188
Tag	189
Scalanie	189
Konflikt scalania	190
Stash	190
Pull request	190
Cherry-picking	190
Bisecting	191
Rebasing	192
Najlepsze praktyki dotyczące komunikatów commitów	193
Dobre komunikaty o commitach	194
Graficzne interfejsy użytkownika	195
Przydatne aliasy powłoki	195
GitHub dla ubogich	195
Uwagi wstępne	196
1. Łączenie z serwerem	196
2. Instalowanie Gita	196
3. Inicjalizowanie repozytorium	196
4. Klonowanie repozytorium	197
5. Dokonaj edycji projektu i wyślij zmiany	197
Podsumowanie	198

ROZDZIAŁ 15**Konteneryzacja aplikacji za pomocą Dockera 199**

Dlaczego kontenery działają jako pakiety?	200
Wymagania wstępne — instalacja Dockera	201
Przyspieszony kurs Dockera	201
Tworzenie obrazów za pomocą pliku Dockerfile	203
Polecenia kontenera	206
docker run	206
docker ps	207
docker exec	208
docker stop	208
Projekt Dockera: kontener aplikacji utworzonej w języku Python i frameworku Flask	208
1. Konfigurowanie aplikacji	208
2. Tworzenie obrazu Dockera	210
3. Uruchomienie kontenera z obrazu	210
Porównanie kontenerów i maszyn wirtualnych	211
Krótką uwagę na temat repozytoriów obrazów Dockera	212
Bolesne lekcje dotyczące kontenerów	213
Rozmiar obrazu	213
Standardowa biblioteka C	213
Laptop nie jest środowiskiem produkcyjnym — zewnętrzne zależności	214
Teoria kontenerów: przestrzeń nazw	214
Jak przeprowadzać operacje na kontenerach?	215
Podsumowanie	215

ROZDZIAŁ 16**Monitorowanie dzienników aplikacji 217**

Wprowadzenie do rejestrowania	218
Rejestrowanie w Linuksie bywa... dziwne	218
Wysyłanie komunikatów dziennika	219
journald usługi systemd	219
Przykładowe polecenia journalctl	220
Śledzenie aktywnych dzienników dla jednostki	220
Filtrowanie według czasu	220
Filtrowanie pod kątem określonego poziomu dziennika	221
Sprawdzanie dzienników z poprzedniego rozruchu	221
Komunikaty jądra	221
Rejestrowanie w kontenerach Dockera	221

Podstawy dziennika syslog	222
Kategorie rejestrowania	222
Poziomy dotkliwości	224
Konfiguracja i implementacje	224
Wskazówki dotyczące rejestrowania	224
Słowa kluczowe podczas korzystania z logowania ustrukturyzowanego	224
Poziomy dotkliwości	225
Rejestrowanie scentralizowane	225
Podsumowanie	227

ROZDZIAŁ 17

Mechanizm równoważenia obciążenia i HTTP	228
Podstawowa terminologia	229
Brama	229
Upstream	230
Najczęściej pojawiające się błędne przekonania na temat protokołu HTTP	230
Kody stanu HTTP	230
Nagłówki HTTP	233
Wersje protokołu HTTP	234
Mechanizm równoważenia obciążenia	237
CORS	243
Podsumowanie	244

O autorach

David Cohen od ponad 15 lat pracuje jako administrator systemów Linux, inżynier oprogramowania, inżynier infrastruktury, inżynier platformy, inżynier niezawodności, inżynier zapewnienia bezpieczeństwa, programista aplikacji internetowych oraz na kilku innych stanowiskach. W czasie wolnym prowadzi w serwisie YouTube kanał *tutoriallinux*, poprzez który nauczył tysiące osób podstaw pracy z systemem Linux, programowania i podejścia DevOps. Od 2019 roku pracuje w firmie Hashicorp — najpierw jako inżynier niezawodności (SRE), następnie jako architekt referencyjny, a obecnie jako inżynier oprogramowania.

Aleyna, dziękuję Ci za bezwarunkowe wsparcie przez ostatnie kilka lat, gdy byłem zajęty przygotowaniem i pisanem tej książki. Bez Ciebie to byłby kolejny z moich obiecujących, ale niedokończonych projektów, który trafiłby do zapomnianego katalogu „Archiwum”. Dziękuję również Christianowi, który od ponad dekady jest moim przyjacielem i partnerem w praktycznie każdym pomysle technicznym, jaki przyszedł mi do głowy, odkąd się poznaliśmy. Ogromne podziękowania składam także moim przyjaciołom i kolegom z firmy Hashicorp oraz z innych, w których pracowałem przez ostatnie ponad 15 lat. Dzięki nim stałem się lepszym inżynierem i znalazłem zachętę do realizacji projektów takich jak niniejsza książka.

Christian Sturm jest konsultantem oprogramowania i architektury systemów. W ciągu ostatniej dekady pracował na różnych stanowiskach technicznych. Zajmował się programowaniem aplikacji na potrzeby frontendu i backendu w małych i dużych firmach, takich jak zoomsquare i Plutonium Labs. Ponadto jest aktywnym współpracownikiem wielu projektów otwartoźródłowych. Posiada dużą wiedzę z zakresu systemów operacyjnych, protokołów sieciowych, zapewnienia bezpieczeństwa oraz systemów zarządzania bazami danych.

O korektorach merytorycznych

Mario Splivalo jest konsultantem zajmującym się bazami danych istniejącymi w nowoczesnych architekturach opartych na chmurze. Pomaga również firmom w projektowaniu infrastruktury za pomocą narzędzi IaaS takich jak Terraform i AWS Cloudformation. Przez pięć lat Mario pracował w firmie Canonical jako inżynier OpenStack.

Jego fascynacja komputerami rozpoczęła się w czasie, gdy na rynku komputerów domowych dominował Commodore 64. Pierwsze kroki w informatyce postawił, używając języka programowania BASIC w komputerze C64 należącym do jego taty, a następnie szybko przeszedł do języka assembler. Stopniowo przeniósł się na platformę PC i zajął się w programowaniu, projektowaniu systemów i administrowaniu bazami danych. Na początku obecnego stulecia zaczął używać systemu Linux (najpierw dystrybucji Knoppix, a później Ubuntu i nigdy nie żałował tej decyzji), kontynuując pracę jako administrator baz danych, programista i administrator systemu.

Nathan Chancellor pochodzi z Arizony i jest niezależnym konsultantem pracującym nad jądrem systemu Linux. Jako programista koncentruje się na poprawieniu zgodności między jądrem Linuksa i zestawem narzędzi LLVM. Używa Linuksa od 2016 roku, a od 2018 roku jest to jego główny system operacyjny wykorzystywany w programowaniu. Wybrane przez niego dystrybucje to Arch Linux i Fedora.

Wprowadzenie |

Większość inżynierów oprogramowania nie zna systemów Unix, pomimo że są one wszechobecne w świecie inżynierii oprogramowania. Od programisty oczekuje się pracy z systemami z rodziny Unix działającymi w środowisku roboczym (macOS), procesach w trakcie tworzenia oprogramowania (kontenery Dockera), narzędziach kompilacji i automatyzacji (potok ciągłej integracji i GitHub), środowiskach produkcyjnych (serwery Linuksa i kontenery) itd.

Umiejętność pracy z powłoką Linuksa może pomóc programistom wyjść poza to, czego się od nich oczekuje, umożliwiając im także:

- oszczędność czasu dzięki wiedzy, kiedy można skorzystać z wbudowanych narzędzi systemu Unix zamiast tworzyć składające się z setek wierszy kodu skrypty bądź programy pomocnicze;
- pomoc podczas debugowania skomplikowanych awarii w środowiskach produkcyjnych, co często obejmuje pracę z serwerami Linuksa i ich interfejsami dla aplikacji;
- pomoc młodszy inżynierom;
- znacznie lepsze zrozumienie, jak tworzone przez nich oprogramowanie wpasowuje się w większy ekosystem i stos sieciowy.

Mamy nadzieję, że teoria, przykłady i projekty zamieszczone w tej książce pomogą Ci podnieść na wyższy poziom umiejętności w zakresie programowania w Linuksie.

Dla kogo przeznaczona jest ta książka?

Ta książka jest przeznaczona dla programistów oprogramowania, którzy dopiero poznają Linuksa i powłokę lub którzy wypadli z obiegu i chcą szybko odświeżyć swoje umiejętności. Jeżeli nadal czujesz się niepewnie, siadając przed powłoką Linuksa w serwerze produkcyjnym o drugiej nad ranem, to ta książka jest właśnie dla Ciebie. Jeżeli chcesz w krótkim czasie zdobyć umiejętności w zakresie Linuksa, które mogą pomóc Ci w karierze, to ta książka jest właśnie dla Ciebie. Jeżeli kieruje Tobą ciekawość i chcesz zobaczyć, jak można poprawić efektywność codziennej pracy programistycznej poprzez wykorzystanie magii w powłoce, to ta książka jest również dla Ciebie.

Czego nie znajdziesz w tej książce?

Jednym ze sposobów, w jakie staraliśmy się spełnić swoją wizję wyjątkowo użytecznej książki, było zachowanie dużej ostrożności w zakresie zamieszczonego w niej materiału. Postanowiliśmy wyeliminować wszystko, co nie jest niezbędne w pracy programisty albo w celu poznania na podstawowym poziomie systemu Linux i jego najważniejszych abstrakcji. Innymi słowy niniejsza książka okazuje się tak przydatna dlatego, że *pominięliśmy w niej wiele tematów*.

Ta książka nie jest pełnym kursem systemu Linux. Nie jest także przeznaczona dla Czytelników pracujących jako inżynierowie systemu Linux bądź programiści zajmujący się rozwijaniem jądra Linuksa. Z tego powodu jej objętość nie sięga ponad 750 stron, więc można ją przeczytać w ciągu zaledwie kilku dni, być może w trakcie spokojniejszego sprintu w pracy.

Co znajdziesz w tej książce?

Rozdział 1. „Jak działa wiersz poleceń?” przedstawia sposób działania interfejsu wiersza poleceń w Linuksie, wyjaśnia, czym jest powłoka, a następnie pomaga w zdobyciu podstawowych umiejętności w zakresie pracy z systemem Linux. Na początku poznasz nieco teorii, a potem zaczniesz zaznajamiać się z zagadnieniami związanymi z wierszem poleceń, wyszukiwaniem i pracą z plikami, a także dowiesz się, gdzie szukać pomocy w razie problemów. Ten rozdział jest przeznaczony dla początkujących programistów, którzy dzięki niemu zdobędą najważniejsze umiejętności w zakresie pracy w powłoce. Nawet jeśli zakończysz lekturę książki na tym rozdziale, i tak znajdziesz się w lepszej sytuacji niż przed sięgnięciem po nią.

Rozdział 2. „Praca z procesami” zawiera wprowadzenie do tematu procesów w Linuksie. Następnie przedstawiamy użyteczne i praktyczne umiejętności przydatne podczas pracy z procesami w interfejsie wiersza poleceń. Dokładnie poznasz kilka aspektów, które najczęściej okazują się źródłem problemów związanych z procesami napotkanych podczas pracy na stanowisku programisty oprogramowania (np. uprawnienia), a także znajdziesz podpowiedzi pomocne w usuwaniu takich problemów. Ten rozdział to preludium do bardziej zaawansowanych tematów, które pojawią się w dalszej części książki.

Rozdział 3. „Zarządzanie usługami za pomocą usługi systemd” zawiera materiał oparty na informacjach dotyczących procesów, które zostały zaprezentowane w poprzednim rozdziale. W tym rozdziale pojawi się dodatkowa warstwa abstrakcji, czyli usługa systemd. Wyjaśnimy, na czym polega działanie procesu init w systemie operacyjnym i dlaczego jest on tak ważny. Następnie zaprezentujemy praktyczne polecenia, które trzeba znać podczas pracy z usługami w Linuksie.

Rozdział 4. „Korzystanie z historii powłoki” jest krótki i zawiera wybrane sztuczki, których opanowanie pozwoli zwiększyć szybkość i efektywność pracy w interfejsie wiersza poleceń. Sztuczki te wiążą się z używaniem skrótów i wykorzystaniem historii powłoki w celu uniknięcia zbędnych naciśnięć klawiszy.

Rozdział 5. „Wprowadzenie do plików” zawiera wprowadzenie do plików, czyli podstawowej abstrakcji pozwalającej na zrozumienie systemu Linux. Poznasz **standard FHS** (ang. *filesystem hierarchy standard*), przypominający mapę, dzięki której można orientować się w rozmieszczeniu plików i katalogów w systemie Unix. Następnie przedstawiamy praktyczne polecenia przeznaczone do pracy z plikami i katalogami w Linuksie, m.in. wybrane specjalne typy plików, których prawdopodobnie nie znasz. Ponadto poruszymy temat wyszukiwania plików i treści w nich, co okazuje się jedną z najcenniejszych umiejętności, jakie może posiadać programista.

Rozdział 6. „Edycja plików w wierszu poleceń” zawiera wprowadzenie do dwóch edytorów tekstowych, nano i vim. Przedstawimy podstawy pracy z tymi edytorami na potrzeby edytowania plików z poziomu interfejsu wiersza poleceń. Ponadto znajdziesz tutaj omówienie najczęściej popełnianych podczas edycji tekstu błędów i sposobów ich unikania.

Rozdział 7. „Użytkownicy i grupy” wyjaśnia koncepcje użytkowników i grup jako podstawy dla modelu zapewnienia bezpieczeństwa w systemie Unix oraz kontrolowania dostępu do zasobów takich jak pliki i procesy. Ponadto przedstawiamy praktyczne polecenia do tworzenia i modyfikowania użytkowników i grup.

Rozdział 8. „Własność i uprawnienia” bazuje na poprzednim rozdziale i wyjaśnia sposób działania kontroli dostępu do zasobów w Linuksie. Na podstawie analizy długiego listingu zamieściliśmy tutaj informacje dotyczące własności i uprawnień. Ponadto w rozdziale znalazło się omówienie najczęściej stosowanych uprawnień plików i katalogów, zwykle napotykanych w produkcyjnych systemach Linuksa. Przedstawiliśmy też polecenia Linuksa przeznaczone do modyfikowania informacji dotyczących własności i uprawnień plików.

Rozdział 9. „Zarządzanie zainstalowanym oprogramowaniem” wyjaśnia kwestie związane z instalowaniem oprogramowania w różnych dystrybucjach Linuksa (a nawet w systemie macOS). Na początku omawiamy menedżery pakietów, które są preferowanym sposobem instalowania oprogramowania w Linuksie. Przy okazji zamieściliśmy najważniejsze informacje teoretyczne i polecenia praktyczne przeznaczone do operacji zarządzania pakietami, które są często przeprowadzane przez programistów oprogramowania. Następnie omówiliśmy kilka innych metod, m.in. pobieranie skryptów instalacyjnych, a także mającą długą tradycję w systemie Unix i niemalże rzemieślniczą metodę samodzielnej kompilacji oprogramowania w środowisku lokalnym, przeprowadzanej na podstawie kodu źródłowego (to tylko brzmi przerażająco).

Rozdział 10. „Konfigurowanie oprogramowania” bazuje na poprzednim rozdziale, dotyczącym instalowania oprogramowania, i zawiera informacje związane z jego konfigurowaniem w Linuksie. Wymienimy miejsca, w których większość oprogramowania będzie szukała plików konfiguracyjnych (tzw. hierarchia konfiguracji). Ta wiedza nie tylko okazuje się przydatna podczas prowadzonych późno w nocy sesji debugowania, ale również może naprawdę pomóc w tworzeniu lepszego oprogramowania. Omówimy argumenty powłoki, zmienne środowiskowe i pliki konfiguracyjne. Ponadto wyjaśnimy sposoby działania tych elementów w niestandardowych środowiskach Linuksa, takich jak kontenery Dockera. W tym rozdziale znajduje się nawet niewielki projekt dodatkowy, w ramach którego pokazujemy, jak można program niestandardowy zamienić we własną usługę działającą w ramach usługi systemd.

Rozdział 11. „Potoki i przekierowanie” pokazuje możliwości niesamowitej funkcjonalności systemu Unix, jaką jest zdolność łączenia istniejących programów w niestandardowe rozwiązanie za pomocą potoków. W tym rozdziale znajduje się omówienie niezbędnej teorii i praktycznych umiejętności, które trzeba opanować: deskryptorów plików oraz przekierowania wejścia-wyjścia. Ponadto przedstawiliśmy wykorzystanie potoków do tworzenia skomplikowanych poleceń. Zaprezentowaliśmy również ważne narzędzia interfejsu wiersza poleceń i związane z potokami praktyczne wzorce, które okażą się użyteczne jeszcze przez długi czas po zakończeniu lektury książki.

Rozdział 12. „Automatyzacja zadań za pomocą skryptów powłoki” ma charakter krótkiego wprowadzenia do tworzenia skryptów powłoki Bash. Zakładamy, że Czytelnik jest programistą, więc rozdział zawiera jedynie krótkie wprowadzenie pokazujące podstawowe możliwości powłoki Bash, bez poświęcania zbyt dużej ilości miejsca na wyjaśnianie podstaw programowania. Zaprezentowaliśmy podstawową składnię powłoki Bash, najlepsze praktyki w zakresie tworzenia skryptów, a także informacje o poważnych pułapkach, których należy unikać.

Rozdział 13. „Bezpieczny dostęp zdalny za pomocą SSH” wyjaśnia protokół Secure Shell (SSH) oraz związane z nim narzędzia działające z poziomu interfejsu wiersza poleceń. Przedstawiliśmy podstawy dotyczące **kryptografii klucza publicznego** (ang. *public-key cryptography*, PKI), których znajomość zawsze będzie przydatna dla programistów. Zaprezentowaliśmy również procedury tworzenia kluczy SSH i bezpiecznego logowania się do zdalnych systemów poprzez sieć. Jest to wiedza pozwalająca zdobyć doświadczenie w zakresie kopiowania plików przez sieć, używania protokołu SSH do tworzenia tymczasowych proxy bądź VPN. W tym rozdziale zamieściliśmy także przykłady różnych innych zadań, obejmujących przenoszenie danych poprzez zaszyfrowany tunel SSH.

Rozdział 14. „Kontrola wersji za pomocą Gita” pokazuje, w jaki sposób można używać doskonale znanego programistom narzędzia Git z poziomu interfejsu wiersza poleceń zamiast poprzez zintegrowane środowisko programistyczne bądź klienta graficznego. Pokrótkę przedstawiliśmy podstawową teorię kryjącą się za systemem kontroli wersji Git oraz polecenia stosowane podczas pracy z nim w powłoce. Ponadto omówiliśmy dwie oferujące potężne możliwości funkcje, bisecting i rebasing, oraz wybrane spośród najlepszych praktyk i użyteczne aliasy powłoki. W podrozdziale „GitHub dla ubogich” znalazł się również mały, lecz użyteczny projekt, który umożliwia wypróbowanie zdobytych dotychczas umiejętności w zakresie pracy w Linuksie.

Rozdział 15. „Konteneryzacja aplikacji za pomocą Dockera” przedstawia podstawową teorię i praktyczne umiejętności, które ułatwiają programistom pracę z oprogramowaniem konteneryzacji Docker. Omówiliśmy problemy rozwiązywane przez Dockera, najważniejsze koncepcje kryjące się za tym narzędziem, a także podstawowe przepływy pracy i polecenia. Na przykładzie konteneryzacji rzeczywistej aplikacji wyjaśniamy, jak można tworzyć własne obrazy. Ponieważ zagadnienia zostały przedstawione z perspektywy systemu Linux i tworzenia oprogramowania, materiał ten pozwala dobrze poznać sposób działania konteneryzacji oraz pokazuje, czym różni się ona od maszyn wirtualnych.

Rozdział 16. „Monitorowanie dzienników aplikacji” pokazuje ogólnie koncepcję rejestrowania danych w systemach Unix i Linux. Wyjaśniliśmy, jak (i gdzie) za pomocą usługi systemd są zbierane dzienniki zdarzeń w większości nowoczesnych systemów Linux. Pokazaliśmy również, jak działają bardziej tradycyjne podejścia (w rzeczywistości spotkasz się

z obydwoma podejściami). Przedstawiliśmy praktyczne polecenia umożliwiające wyszukiwanie i przeglądanie dzienników zdarzeń. Ponadto zamieściliśmy informacje o sposobach użycia dzienników zdarzeń w większych infrastrukturach.

Rozdział 17. „Mechanizm równoważenia obciążenia i HTTP” zawiera omówienie przeznaczonych dla programistów podstaw protokołu HTTP. Szczególny nacisk położyliśmy na złożoności pojawiające się podczas pracy z usługami HTTP w większych infrastrukturach. Wyprostowaliśmy niektóre z błędnych przekonań na temat kodów stanu HTTP, nagłówków HTTP i wersji HTTP, a także sposobów, w jakie powinny one być obsługiwane przez aplikacje. W tym rozdziale znajduje się wprowadzenie do rzeczywistego działania mechanizmu równoważenia obciążenia i proxy. Pokazaliśmy, że z ich powodu debugowanie wdrożonej aplikacji odbywa się znacznie inaczej niż w przypadku rozwiązywania problemów z wersją aplikacji pozostającą w komputerze programisty. Wiele zdobytych wcześniej umiejętności okazuje się przydatnych podczas lektury tego rozdziału. Ponadto zostało wprowadzone nowe narzędzie, polecenie `curl`, które pomaga w rozwiązywaniu wielu problemów związanych z protokołem HTTP.

Jak zmaksymalizować korzyści z lektury tej książki?

Jeżeli potrafisz samodzielnie skorzystać z powłoki Linuksa, np. poprzez zainstalowanie dystrybucji Ubuntu w maszynie wirtualnej bądź uruchomienie jej za pomocą kontenera Dockera, to będziesz w stanie wykonać wszystkie przykłady zamieszczone w tej książce.

Jednak wcale nie musisz się tak wysilać — Windows zawiera podsystem WSL, natomiast macOS bazuje na systemie operacyjnym Unix, dlatego niemalże wszystkie praktyczne polecenia zamieszczone w książce (z wyjątkiem tych, które są przeznaczone jedynie dla Linuksa) będą działały od razu, bez konieczności podejmowania dodatkowych kroków. Aby zapewnić sobie jak najlepsze wrażenia, warto skorzystać z systemu operacyjnego Linux.

Materiał zamieszczony w tym rozdziale wymaga jedynie podstawowych umiejętności, które posiada każdy programista oprogramowania, takich jak edytowanie tekstu, praca z plikami i katalogami, instalowanie oprogramowania i używanie środowiska programistycznego, jak również ogólnego pojęcia, czym jest system operacyjny. Wszystkiego innego dowiesz się z tej książki.

Przyjęte konwencje

W książce zastosowaliśmy wiele różnych stylów tekstu pozwalających rozróżnić poszczególne rodzaje informacji.

KodWTeKście — ten styl wskazuje fragmenty kodu w teKście, nazwy tabel baz danych i dane wejściowe wprowadzone przez użytkownika. Oto przykład: „Ten kod powoduje utworzenie w pamięci egzemplarza o nazwie `text`, w którym jest przechowywany komunikat `Witaj, świecie!`”.

Kursywa — tym stylem oznaczyliśmy nazwy plików i katalogów, rozszerzenia plików, ścieżki dostępu, adresy URL, a także adresy w serwisie Twitter. Tak oznaczone są również słowa widoczne na ekranie, w menu lub w oknach dialogowych. Oto przykład: „Główny plik konfiguracyjny znajduje się w pliku */etc/nginx.conf*, a dodatkowe pliki konfiguracyjne pochodzą z kolejnych plików z katalogu */etc/nginx/conf.d/* */etc/nginx/conf.d/*”.

Blok kodu w powłoce przedstawia się następująco:

```
/home/steve/Desktop# ls
anotherfile documents somefile.txt stuff
/home/steve/Desktop# cd documents/
/home/steve/Desktop/documents# ls
contract.txt
```

Nowe pojęcia i ważne słowa są zapisywane pogrubioną czcionką.

Uwaga

Ostrzeżenia, wskazówki i ważne informacje pojawiają się w takich ramkach.

Jak działa wiersz poleceń?

Rozdział 1

Zanim zagłębimy się w praktyczne polecenia Linuksa, musisz mieć podstawową wiedzę na temat sposobu działania wiersza poleceń. Ten rozdział pozwoli Ci ją zdobyć.

Początkujący programiści poznają podstawowe umiejętności potrzebne do rozpoczęcia pracy w wierszu poleceń Linuksa. Dla tych, którzy mają nieco więcej doświadczenia, nadal pozostają do odkrycia pewne niuanse, takie jak różnica między powłoką a wierszem poleceń. Warto znać tę różnicę!

W tym rozdziale omawiamy następujące zagadnienia:

- podstawowa idea interfejsu wiersza poleceń (ang. *command-line interface* — CLI);
- forma poleceń;
- działanie argumentów poleceń i ich wygląd podczas wpisywania poleceń i przeglądania dokumentacji;
- wprowadzenie do powłoki i wyjaśnienie, czym różni się ona od wiersza poleceń;
- podstawowe reguły wykorzystywane przez powłokę do wyszukiwania poleceń.

Zacniemy od podstawowej idei interfejsu wiersza poleceń. Wyjaśnimy działanie CLI i przeanalizujemy krótki przykład.

Na początku był... REPL

Interfejs wiersza poleceń (ang. *command-line interface* — CLI) to środowisko tekstowe do interakcji z komputerem, które wykonuje następujące czynności:

1. Odczytuje (ang. *read*) dane wejściowe wprowadzone przez użytkownika.
2. Ewaluuje (ang. *evaluate*), czyli przetwarza, te dane wejściowe.
3. W odpowiedzi wypisuje (ang. *print*) na ekranie dane wyjściowe.
4. Wraca (ang. *loop*) do punktu 1., aby powtórzyć ten proces.

Przyjrzyjmy się na poziomie praktycznym temu, co dzieje się na każdym z tych kroków. Wykorzystamy do tego celu polecenie `ls` (ang. *list*), które omówimy dalej w tym rozdziale. Na razie wystarczy wiedzieć, że polecenie `ls` wyświetla zawartość katalogu.

Krok	Opis
1. Odczyt danych wejściowych	Użytkownik wpisuje polecenie <code>ls</code> i naciska przycisk <i>Enter</i> .
2. Ewaluowanie polecenia	Powłoka wyszukuje plik binarny narzędzia <code>ls</code> i nakazuje komputerowi jego wykonanie.
3. Wypisanie danych wyjściowych	Polecenie <code>ls</code> emituje tekst — nazwy wszystkich znalezionych plików i katalogów — a powłoka wypisuje te dane wyjściowe w oknie terminala.
4. Powrót do punktu 1. (powtórzenie procesu)	Po zakończeniu programów wywołanych przez polecenie proces jest powtarzany przez przyjęcie od użytkownika kolejnych danych wejściowych.

Jeśli ponownie przeczytasz kroki 1. – 4., zauważysz, że pierwsze litery ich angielskich określeń tworzą skrótowiec REPL. Stosuje się go powszechnie do określenia tego rodzaju pętli odczyt – ewaluacja – wypisanie – powrót (ang. *Read-Eval-Print-Loop*) w językach takich jak Lisp, których twórcy wymyślili i udoskonalili ten przepływ pracy.

Powyższe instrukcje REPL można przełożyć na kod:

```
while (true) { // Pętla
    print(eval(read()))
}
```

W większości języków programowania rzeczywiście można za pomocą zaledwie kilku linii kodu utworzyć środowisko REPL zdolne do wykonywania podstawowych obliczeń. Oto jednoliniowy program „powłoki” napisany w Perlu:

```
perl -e 'while (<){print eval, "\n"}'
1+2
3
```

Zapisujemy tutaj kod jako parametr, wypisując wynik ewaluacji dopóty, dopóki są jakieś dane wejściowe do odczytania. Na koniec dodajemy znak nowej linii i kończymy działanie programu.

Ten program jest niewielki, ale wystarczy, aby zaimplementować interaktywną pętlę *Read-Eval-Print Loop* (REPL) w środowisku wiersza poleceń, czyli w **powłoce** (ang. *shell*). Powłoki używane w Linuksie i Uniksie są znacznie bardziej złożone niż ta minipowłoka Perla, ale rządzące nimi zasady są takie same.

Chodzi o to, że jako programista prawdopodobnie korzystałeś już ze środowisk REPL, nie zdając sobie z tego sprawy, ponieważ prawie wszystkie nowoczesne języki skryptowe są w nie wyposażone. Zasadniczo wiersz poleceń Linuksa (lub systemu macOS czy innego systemu uniksowego) działa jak „interaktywna powłoka”, którą zapewniają języki interpretowane. Dlatego nawet jeśli nie jesteś zaznajomiony ze środowiskiem REPL języka Lisp, powyższy fragment Perla powinien przypominać Ci bardzo podstawową powłokę Ruby lub Pythona.

Skoro rozumiesz już podstawy działania interfejsów wiersza poleceń, z których będziesz korzystać w systemie Linux, możesz wypróbować swoje pierwsze polecenia. W tym celu musisz poznać prawidłową składnię wiersza poleceń.

Składnia wiersza poleceń (odczyt)

Wszystkie środowiska REPL zaczynają od odczytania danych wejściowych. W wierszu poleceń Linuksa polecenia odczytywane przez powłokę muszą mieć poprawną składnię. Te polecenia przyjmują następującą podstawową formę:

nazwa_polecenia opcje

W kategoriach programistycznych nazwę polecenia można potraktować jak nazwę funkcji, a opcje jak dowolną liczbę argumentów, które zostaną przekazane tej funkcji. Jest to ważne, ponieważ nie ma jednej ustalonej składni dla wszystkich opcji — każde polecenie definiuje, jakie parametry będzie przyjmować. Z tego powodu poza sprawdzeniem, czy polecenie mapuje się na plik wykonywalny, powłoka niewiele może zrobić, aby zweryfikować poprawność polecenia.

Uwaga

Terminy „program” i „polecenie” są w tym rozdziale używane zamiennie. Między tymi pojęciami istnieje bardzo niewielka różnica, ponieważ niektóre polecenia wbudowane powłoki są zdefiniowane w jej kodzie i dlatego technicznie nie są oddzielnymi programami, ale nie musisz się tym przejmować — pozostaw to rozróżnienie uniksowym mędrcom.

Przyjrzyjmy się bardziej złożonym wariacjom na temat składni *polecenie* [*opcje*], z którymi będziesz się często spotykać:

polecenie [-flagi,] [--przykład=foobar] [nawet_więcej_opcji ...]

Jest to konwencjonalny format używany w dokumentacji pomocy, takiej jak strony podręcznika programu (ang. *manpages*) zawarte w większości środowisk linuxowych, i jest dość prosty:

- *polecenie* to uruchamiany program;
- elementy w nawiasach są opcjonalne, a nawiasy z wielokropkami ([xyz ...]) informują, że można tu przekazać zero lub więcej argumentów;
- *-flagi* oznaczają dowolną prawidłową opcję (w języku uniksowym „flagę”) dla tego programu, np. *-debug* lub *-foobar*.

Niektóre programy przyjmują również krótkie i długie wersje parametru, oznaczane zwykle pojedynczymi lub podwójnymi minusami, np. *-l* i *--long* mogą robić to samo. Nie dotyczy to jednak wszystkich poleceń; tego rodzaju zachowanie wymaga, aby twórca polecenia zaimplementował krótkie i długie argumenty, które ustawiają ten sam parametr.

Nie każde polecenie będzie implementować wszystkie te sposoby przekazywania konfiguracji podczas jego wywoływania, ale są to najczęściej spotykane formy.

Domyślnie spacja oznacza koniec argumentu, dlatego podobnie jak w większości języków programowania, ciąg argumentów zawierający spacje musi być ujęty w pojedyncze lub podwójne cudzysłowy. Więcej informacji na ten temat znajdziesz w rozdziale 12. „Automatyzacja zadań za pomocą skryptów powłoki”.

Za chwilę prześledzimy proces interpretowania przez powłokę polecenia wydanego przy użyciu tej składni, ale najpierw chcemy jasno zdefiniować różnicę między dwoma stosowanymi czasem zamiennie terminami, których używamy w tym rozdziale: wiersz poleceń i powłoka.

Wiersz poleceń i powłoka

W tej książce posługujemy się określeniem „środowisko wiersza poleceń”. Definiujemy je jako dowolne środowisko tekstowe, które działa jak rodzaj środowiska REPL i służy w szczególności do interakcji z systemem operacyjnym, interpreterem języka programowania, bazą danych itp. Środowisko lub interfejs wiersza poleceń opisuje ogólną ideę interakcji z systemem.

Istnieje jednak bardziej konkretny termin, którego będziemy tu używać: powłoka.

Powłoka to specjalny program, który implementuje to środowisko wiersza poleceń i umożliwia wydawanie poleceń tekstowych. Technicznie rzecz biorąc, istnieje wiele różnych powłok, które zapewniają ten sam rodzaj środowiska wiersza poleceń opartego na koncepcji REPL i często są przeznaczone do bardzo różnych rzeczy:

- Bash to popularne środowisko powłoki do interakcji z systemami operacyjnymi Linux i Unix.
- Popularne bazy danych, takie jak Postgres, MySQL i Redis, zapewniają programistom powłokę do interakcji i uruchamiania poleceń.
- Większość języków interpretowanych udostępnia środowisko powłoki w celu przyspieszenia rozwoju oprogramowania. W tych przypadkach prawidłowe polecenia są po prostu instrukcjami języka programowania. Przykłady to `irb` dla Ruby, interaktywna powłoka Pythona itp.
- Zsh (powłoka Z) to alternatywna powłoka systemu operacyjnego (taka jak Bash), którą można zobaczyć na komputerach niektórych programistów, jeśli dostosowali oni swoje środowiska.

Kiedy w tej książce piszemy o **powłoce**, mamy na myśli powłokę uniksową (zazwyczaj Bash), która jest interfejsem wiersza poleceń zaprojektowanym specjalnie w celu umożliwienia interakcji z podstawowym systemem operacyjnym Linux lub Unix.

Skąd powłoka wie, co uruchomić? (ewaluowanie)

Po **odczytaniu** polecenia, powłoka musi je **ewaluować**, wykonując program, pobierając informacje lub robiąc coś innego, co jest przydatne dla użytkownika.

Uwaga

Tak szczegółowy opis działania powłok może początkowo wydawać się nużący, ale obiecujemy, że wiedza ta przyda Ci się, gdy trzeba będzie rozwiązać problem z brakującym lub mającym nieprawidłowe uprawnienia programem.

Po wpisaniu przykładowego polecenia `foobar -opcja1 test.txt` w powłoce takiej jak Bash i naciśnięciu przycisku *Enter* dzieje się kilka rzeczy:

1. Jeśli polecenie ma określoną ścieżkę, zostaje ona użyta. Ścieżka może mieć różne formy:
 - Pełna ścieżka, taka jak `/usr/bin/foobar` w poleceniu `/usr/bin/foobar -opcja1 test.txt`.
 - Ścieżka względna, taka jak bieżący katalog roboczy w poleceniu `./foobaropcja1 test.txt` (• oznacza bieżący katalog, co omówimy w punkcie „Bezwzględne i względne ścieżki plików”; to polecenie zasadniczo instruuje „wykonaj plik *foobar*, który znajduje się w moim bieżącym katalogu”).
 - Ścieżka może być oparta na zmiennych i symbolach:
 - ◆ zmiennych środowiskowych powłoki, np. `$HOME/foobar`,
 - ◆ symbolach dostarczanych przez powłokę, np. `~/foobar` (~ oznacza katalog domowy danego użytkownika).
2. Jeśli polecenie nie ma określonej ścieżki, powłoka sprawdza, czy potrafi zinterpretować, co oznacza *foobar*:
 - Może to być wbudowane polecenie powłoki.
 - Może to być **alias**, który jest sposobem konfigurowania makr lub skrótów dla poleceń.
3. Jeśli powłoka nie potrafi zinterpretować polecenia, zazwyczaj sprawdza zmienną środowiskową `$PATH`, która zawiera kilka różnych lokalizacji do szukania poleceń: `/bin`, `/usr/bin`, `/sbin` itd. Użytkownicy mogą do tej listy `$PATH` dodawać lokalizacje, a różne programy będą modyfikować `$PATH` — z tego mechanizmu często korzystają menedżery wersji dla języków skryptowych, wirtualne środowiska Pythona i inne programy. Powłoka sprawdza miejsca określone w zmiennej `$PATH` w kolejności, w jakiej znajduje je w tej zmiennej, aby zweryfikować, czy któreś z nich zawiera plik wykonywalny o nazwie *foobar*.

Jeśli powłoka nadal nic nie znajdzie, zwróci błąd typu bash: `foobar: command not found`.

Natomiast jeżeli w którymś momencie powłoka rzeczywiście znajdzie plik wykonywalny o nazwie *foobar*, wykona go i przekaże jako argumenty `-opcja1` i `test.txt` (w tej kolejności).

W tym momencie powłoka wie, jakiego programu użyć do ewaluacji polecenia, i robi to. Gdy polecenie jest ewaluowane, wszelkie dane wyjściowe są wypisywane użytkownikowi, co kończy trzeci krok procesu REPL. Teraz pozostaje tylko zapętlenie z powrotem do początku i rozpoczęcie procesu od nowa przez przyjęcie od użytkownika kolejnego polecenia jako danych wejściowych.

Powłoka stara się jak najlepiej odgadnąć, który program chce uruchomić użytkownik, używając ogólnego procesu opisanego powyżej, aby usunąć niejednoznaczność. Jednak niejednoznaczność może być zła i prowadzić do nieporozumień lub błędów. Podczas rozwiązywania problemów często będziesz chciał dowiedzieć się, które polecenie jest naprawdę uruchamiane. W tym celu można użyć polecenia `which <polecenie>`, które wyświetli pełną ścieżkę (albo alias lub uruchamiany skrypt) i poinformuje, czy to polecenie jest wbudowanym poleceniem powłoki. W zależności od systemu polecenie `which` może nie być dostępne. W takich sytuacjach można zamiast z niego skorzystać z polecenia `command -v`. Jest to odpowiednik z interfejsu POSIX, który omówimy w następnym punkcie:

```
bash-3.2$ which ls
/bin/ls
bash-3.2$ command -v ls
/bin/ls
```

Krótką definicja interfejsu POSIX

Wikipedia informuje, że „**POSIX** (ang. *Portable Operating System Interface for UNIX*), czyli przenośny interfejs dla systemu operacyjnego Unix, to rodzina standardów określonych przez stowarzyszenie IEEE Computer Society w celu utrzymania kompatybilności między systemami operacyjnymi”. Praktycznie rzecz biorąc, jest to próba zdefiniowania wspólnych standardów dla systemów uniksowych, które w przeciwnym razie mogą mieć bardzo różne zestawy dostępnych podstawowych poleceń.

POSIX zasadniczo przyjmuje takie założenia jak „każdy system operacyjny zgodny z interfejsem POSIX powinien mieć polecenie listowania katalogów o nazwie `ls`”; w takim przypadku „każdy system operacyjny zgodny z interfejsem POSIX powinien mieć sposób na sprawdzenie, czy istnieje pasujący plik wykonywalny dla danej nazwy polecenia”.

Jeśli Twoje skrypty muszą być przenośne między systemami operacyjnymi Unix, dobrym pomysłem jest ograniczenie się do poleceń interfejsu POSIX. Jednak nadal nie stanowi to pełnej gwarancji kompatybilności — wiele niezwykle popularnych dystrybucji Linuksa odchodzi od interfejsu POSIX na wiele sposobów, z których większości nie zauważysz, dopóki nie staną Ci na drodze.

Zrozumienie interfejsu POSIX jest ostatnią cegiełką w fundamencie, którego potrzebujesz, aby rozpocząć praktyczną pracę w wierszu poleceń. Do tej pory wyjaśniliśmy już:

- czym jest REPL i jak ten podstawowy proces odwzorowuje sposób działania wszystkich nowoczesnych powłok;
- podstawową składnię poleceń, których będziesz używać podczas pracy z Linuksem.

Wiedziałeś, w jaki sposób powłoka decyduje o tym, jak przyjąć polecenie wejściowe i dokonać jego poprawnej ewaluacji. Przedstawiliśmy ważną terminologię, z którą będziesz się spotykać na co dzień, taką jak powłoka, interfejs wiersza poleceń, POSIX i kilka innych pojęć — jeśli nauczysz się ich teraz, przyniesie Ci to korzyści w dłuższej perspektywie. Wyposażony w tę wiedzę, jesteś gotowy, aby przejść od teorii do praktyki. W następnym podrozdziale omówimy kontekst charakterystyczny dla systemu Linux, w którym będziesz się

poruszać podczas uruchamiania poleceń. Poznasz absolutne podstawy systemu plików Linuksa i dowiesz się, jak działają różne rodzaje ścieżek. Resztę rozdziału poświęcimy uruchamianiu poleceń Linuksa!

Podstawowe umiejętności wiersza poleceń

Aby efektywnie pracować z Linuksem, musisz znać absolutne podstawy: jak zbudowany jest system, jak wyszukiwać i poruszać się w systemie oraz jak czytać i edytować pliki. W tym podrozdziale omówimy te zagadnienia i przedstawimy podstawy poruszania się po systemie Linux.

W kolejnych rozdziałach będziemy zgłębiać każde z tych zagadnień i poleceń, ale chcemy mieć pewność, że po lekturze tego podrozdziału będziesz mieć minimalny zestaw praktycznych umiejętności.

Podstawy systemu plików Uniksa

W graficznych interfejsach użytkownika **katalogi** (w systemie macOS zwane *folderami*) są reprezentowane przez ikony. Być może jesteś przyzwyczajony do widoku w katalogu głównym schludnych rzędów takich ikon jak *Desktop*, *Documents*, *Videos* itd. Dwukrotne kliknięcie ikony katalogu otwiera nowe okno z nowym widokiem z wnętrza tego katalogu.

Kiedy używamy terminu „system plików”, mamy na myśli zbiór katalogów i plików, które organizują wszystkie dane w systemie. Bazowa koncepcja jest dokładnie taka sama w środowisku wiersza poleceń — po prostu wygląda nieco inaczej.

Zamiast wielu okien i ikon wszystko jest w postaci tekstowej, a zawartość katalogów jest wyświetlana tylko wtedy, gdy o to poprosisz. Jednak pliki i katalogi nadal działają dokładnie tak, jak w interfejsie graficznym.

Początkowo utrzymywanie w głowie wyglądu systemu plików podczas nawigowania po nim wydaje się trudne, ale gdy już się do tego przyzwyczaisz, często będzie to bardziej efektywny sposób pracy. Po kilku dniach większość programistów zaczyna dobrze radzić sobie z takim systemem i weryfikuje widok tylko od czasu do czasu.

Bezwzględne i względne ścieżki plików

Kiedy początkujący programiści pracują z Linuksem, często mają problemy z rozróżnieniem między **ścieżką bezwzględną** a **ścieżką względną**. Niezrozumienie tej prostej kwestii skutkuje frustrującą ilością czasu zmarnowanego na wpatrywanie się w takie oto komunikaty o błędach:

No such file or directory

Ponieważ zrozumienie zagadnienia ścieżek jest warunkiem wstępnym dla prawie każdego uruchamianego polecenia Linuksa, omówimy je w pierwszej kolejności.

Ścieżka bezwzględna to pełna ścieżka do dowolnego pliku w systemie plików rozpoczynająca się od katalogu głównego. Można to rozpoznać, ponieważ zaczyna się od ukośnika (/), który odnosi się do katalogu głównego (samej góry, czyli początku, systemu plików, zawierającego wszystkie inne pliki i katalogi).

Oto kilka przykładów ścieżek bezwzględnych:

- `/home/dave/Desktop`,
- `/var/lib/floobkit/`,
- `/usr/bin/sudo`.

Ścieżki bezwzględne są jak pełny zestaw wskazówek dojazdu, zakręt po zakręcie, od znanego punktu początkowego (Twojego mieszkania, a w przypadku systemu Unix katalogu głównego).

Ścieżkę bezwzględną można natychmiast rozpoznać po tym, że zaczyna się od znaku /. Niezależnie od tego, w którym miejscu systemu plików się znajdujesz, ścieżki bezwzględne będą działać, ponieważ są to pełne, unikatowe adresy obiektów plików.

Ścieżka względna jest ścieżką częściową i przyjmujemy założenie, że zaczyna się od **aktualnej lokalizacji** zamiast od katalogu głównego. Ścieżkę bezwzględną można rozpoznać po tym, że *nie* zaczyna się od znaku /.

Ścieżki względne są jak wskazówki dojazdu, które jako punkt odniesienia wykorzystują bieżącą lokalizację. Jeśli zjedziesz z drogi, bo się zgubisz, i będziesz potrzebować nowych wskazówek, będą one musiały zaczynać się od Twojej **aktualnej lokalizacji**, a nie adresu domowego.

W rezultacie ścieżki względne są często wygodniejsze do wpisywania: jeśli znajdujesz się już w katalogu `/home/Desktop`, łatwiej jest odwołać się do pliku poprzez `mydocument.txt` niż poprzez `/home/Desktop/mydocument.txt` (nawet jeżeli oba sposoby są poprawne, biorąc pod uwagę Twoją lokalizację w systemie plików). Rzeczywista różnica pojawia się po zmianie katalogu. Po przeniesieniu katalogu z `/home/Desktop` do `/home` ścieżka bezwzględna nadal będzie odwoływać się do tego samego pliku, podczas gdy ścieżka względna już nie (teraz wpisanie `mydocument.txt` będzie odwoływać się do `/home/mydocument.txt`).

Wyobraźmy sobie taką częściową strukturę katalogów — w naszym przykładzie będzie to drzewo katalogów `/home/dave/Desktop`:

```
Desktop
├── anotherfile
├── documents
│   └── contract.txt
├── somefile.txt
└── stuff
    ├── nothing
    └── important
```

Znajdujesz się w katalogu `Desktop`, czyli Twój bieżący katalog (który możesz zobaczyć, gdy uruchomisz polecenie `pwd`) to `/home/dave/Desktop`.

Oto kilka przykładów względnych ścieżek do plików w katalogu *Desktop*:

- *anotherfile*,
- *documents/contract.txt*,
- *stuff/important*.

Oto bezwzględne ścieżki do tych samych plików:

- */home/dave/Desktop/anotherfile*,
- */home/dave/Desktop/documents/contract.txt*,
- */home/dave/Desktop/stuff/important*.

Zwróć uwagę, że ścieżka względna jest po prostu ścieżką bezwzględną z uciętą na początku ścieżką do bieżącego katalogu roboczego.

Przegląd bezwzględnych i względnych nazw ścieżek

Przypomnijmy sobie nasz przykład:

```
Desktop
├── anotherfile
├── documents
│   └── contract.txt
├── somefile.txt
└── stuff
    ├── nothing
    └── important
```

Teraz wyobraź sobie, że jesteś w środowisku powłoki, a Twoim bieżącym katalogiem roboczym jest *Desktop*. Chcesz wyświetlić plik *contract.txt*. Jak odwołać się do tego pliku? Masz dwie możliwości:

1. `ls /home/dave/Desktop/documents/contract.txt` — jest to ścieżka bezwzględna, która działa z dowolnego miejsca.
2. `ls documents/contract.txt` — jest to ścieżka względna do tego pliku z bieżącego katalogu.

Otwieranie terminala

W systemach Ubuntu Linux i macOS można przejść do wiersza poleceń przez otwarcie aplikacji *Terminal*.

Rozglądanie się po systemie plików — nawigacja w wierszu poleceń

Pierwszą rzeczą, którą zechcesz zrobić jako początkujący po otwarciu powłoki, jest rozejrzenie się po systemie. W tym podrozdziale omówimy najważniejsze polecenia służące do poruszania się po środowisku Linuksa i przeglądania go za pomocą okna powłoki.

Przejdźmy więc do kilku podstawowych poleceń Linuksa!

pwd — wypisywanie katalogu roboczego

Polecenie `pwd` oznacza wypisywanie katalogu roboczego (ang. *print working directory*). Po wpisaniu go w terminalu powłoka wyświetli katalog, w którym aktualnie się znajdujesz. Uniksowy system plików jest często porównywany do drzewa, ale na razie możesz potraktować go jako nieuporządkowany pulpit z wieloma katalogami w środku. Jeśli każdy katalog jest jak pokój, `pwd` pozwala zobaczyć, który pokój aktualnie jest odwiedzany przez środowisko wiersza poleceń.

Nowe sesje powłoki zazwyczaj rozpoczynają się w katalogu głównym użytkownika. Jeżeli korzystasz z Linuksa, będzie to wyglądać mniej więcej tak:

```
→ ~ pwd
/home/dave
```

Natomiast jeśli korzystasz z innej wersji systemu Unix, może to wyglądać nieco inaczej. Oto co można zobaczyć w systemie macOS:

```
→ ~ pwd
/Users/dave
```

Niezależnie od miejsca w systemie plików można odwoływać się do plików we wszystkich katalogach (zobacz punkt „Bezwzględne i względne ścieżki plików”), ale czasami przemieszczanie się ułatwia sprawę. Szczegółami struktury systemu plików zajmiemy się dalej.

ls — wylistowanie

Polecenie `ls` pozwala „wylistować” (wyświetlić listę) pliki w katalogu. Jeśli uruchomisz to polecenie bez żadnych argumentów, wyświetli ono tylko listę plików i katalogów znajdujących się w bieżącym katalogu. Natomiast gdy przekażesz mu jako argument ścieżkę do jakiegoś katalogu, sprawdzi, co znajduje się w tym katalogu, i wyświetli to:

```
ls /var/log
```

Wylistowanie przyjmuje również argumenty („flagi”). Istnieje wiele flag, ale najczęściej używane to `-l` (ang. *long*), czyli wylistowanie długie, i `-h` (ang. *human-readable*), czyli wersja czytelna dla człowieka.

```
ls -l -h
```

```
# To samo; możesz łączyć flagi
ls -lh
```

```
# Listuje konkretny katalog
ls -lh /usr/local/
```

Wylistowanie długie generuje następujący format danych wyjściowych:

```
-rw-r--r--  1 dcohen wheel  0 Jul  5 09:27 foobar.txt
```

Przeanalizujmy go kolumna po kolumnie:

- `-rw-r--r--` — typ pliku (pierwszy znak) i uprawnienia (trzy grupy po trzy bity, które reprezentują kolejno uprawnienia dla użytkownika będącego właścicielem, grupy będącej właścicielem i wszystkich innych osób w systemie).

- 1 — liczba referencji (dowiązań twardych) do tego pliku.
- dcohen — użytkownik, który jest właścicielem tego pliku.
- wheel — grupa, która jest właścicielem tego pliku.
- 0 — ilość miejsca na dysku wykorzystywanego przez plik (ten jest pusty). Flaga -h zmienia te dane wyjściowe z domyślnej liczby bajtów na coś czytelnego dla człowieka, co oznacza, że w razie potrzeby pokaże megabajty lub gigabajty.
- Jul 5 09:27 — czas modyfikacji pliku.
- foobar.txt — nazwa pliku.

Zrozumienie tych danych wyjściowych wymaga omówienia zagadnień takich jak użytkownicy, grupy i uprawnienia — zajmiemy się tym w rozdziale 7. „Użytkownicy i grupy”.

Poruszanie się po systemie plików

Skoro omówiliśmy już najbardziej podstawowe polecenia Linuksa, zajmijmy się tym, jak w środowisku wiersza poleceń nawigować do pożądanego miejsca.

cd — zmiana katalogu

Polecenie `cd` pozwala zmienić katalog (ang. *change directory*) na dowolne miejsce w systemie plików. Jeśli przywołamy wcześniejszą metaforę pokojów, jest to odpowiednik teleportacji z bieżącego pomieszczenia do innego.

Po pomyślnej zmianie katalogu polecenie `pwd` pokaże nową (zaktualizowaną) lokalizację:

```
bash-3.2$ cd /etc/ssl
```

```
bash-3.2$ pwd
/etc/ssl
```

```
bash-3.2$ ls
README cert.pem certs misc openssl.cnf private
```

```
bash-3.2$ cd certs
```

```
bash-3.2$ pwd
/etc/ssl/certs
```

find — wyszukiwanie plików

Polecenie `find` umożliwia wyszukiwanie plików. Jest to jedno z niewielu poleceń, w których nie stosuje się konwencji opcji długich (np. `--name`). Zamiast tego jego flagi są określane za pomocą pojedynczej kreski. Oto przykład:

```
bash-3.2$ find / -type d -name home
/home
...
```

Powyższe polecenie przeszuka / (cały system) w celu odnalezienia katalogu (-type d) o nazwie *home*. Pamiętaj, że jeśli nie wykonujesz tego polecenia jako wszechmocny użytkownik *root* (administrator), nie będzie ono miało uprawnień do wyświetlania zawartości wielu katalogów, więc oprócz tego, co zostało znalezione, otrzymasz takie dane wyjściowe jak `find: '/root': Permission denied`.

Innym częstym przypadkiem użycia jest wykonywanie poleceń na podstawie danych wyjściowych z polecenia `find`:

```
bash-3.2$ find . -exec echo {} \;  
.  
./foobar
```

Spowoduje to uruchomienie polecenia `echo`, gdzie w miejsce nawiasów klamrowych (`{}`) zostaną podstawione wszelkie znalezione pliki. Wynikowe dane wyjściowe będą przypominać wywołanie polecenia `ls`.

Jeśli zamiast uruchamiać `echo` dla każdego znalezionej pliku chcemy przekazać te pliki jako argumenty do `echo`, możemy zastąpić `\` znakiem `+`:

```
bash-3.2$ find . -exec echo {} +;  
. ./foobar
```

Polecenie `find` ma o wiele więcej flag. Jakie dokładnie? Zależy to od wersji polecenia `find`, z którą dostarczany jest system operacyjny.

Oto kilka typowych przypadków użycia:

- `find -iname foobar` — wyszukuje `foobar`, ale nie rozróżnia wielkości liter.
- `find -name "foobar*"` — wyszukuje pliki zaczynające się od `foobar`.
- `find -name "*foobar"` — wyszukuje pliki kończące się na `foobar`.

Odczytywanie plików

Skoro omówiliśmy już, jak znajdować pożądane pliki, zobaczmy, jak odczytać zawartość pliku w wierszu poleceń.

less — przeglądanie pliku strona po stronie

Polecenie `less` pozwala odczytać plik po jednej „stronie” (na podstawie rozmiaru okna terminala).

```
less jakiś_plik.txt
```

Uruchomienie `less` spowoduje otwarcie pliku i umożliwi jego przewijanie po jednej linii (klawisze strzałek góra i dół) lub stronie (spacja).

Aby móc wyszukiwać wewnątrz pliku, należy wpisać `/`, a następnie wyszukiwany łańcuch znaków i nacisnąć przycisk *Enter*. Po dopasowaniach nawiguje się za pomocą *n* (następne) i *Shift+n* (poprzednie).

Aby zakończyć, wpisz `q`.

Wprowadzanie zmian

Potrafisz już znajdować i odczytywać pliki, więc przyjrzyjmy się, jak je zmieniać lub tworzyć nowe.

touch — tworzenie pustego pliku lub aktualizowanie czasu modyfikacji istniejącego

Polecenie **touch** tworzy plik i dlatego jako argumentu wymaga ścieżki do pliku. Jeśli podana ścieżka jeszcze nie istnieje, tworzony jest pusty plik w tej ścieżce (przy założeniu, że masz do tego uprawnienia).

Jeżeli w podanej ścieżce plik już istnieje, jego znaczniki czasu dostępu i modyfikacji są aktualizowane na bieżący czas. Gdy chcesz jedynie zaktualizować czas dostępu *lub* czas modyfikacji, możesz użyć odpowiednio flag **-a** lub **-m**.

mkdir — tworzenie katalogu

Polecenie **mkdir** jako argumentu wymaga ścieżki pliku i używa jej do utworzenia (ang. *make*) katalogu:

```
bash-3.2$ mkdir foobar
bash-3.2$ ls
foobar
```

Opcjonalnie możesz podać dodatkowe argumenty, jeżeli chcesz utworzyć wiele katalogów:

```
ash-3.2$ mkdir foo bar baz
bash-3.2$ ls
foo
bar
baz
```

Jeśli chcesz utworzyć wiele katalogów zagnieżdżonych jeden w drugim (lub po prostu chcesz upewnić się, że wszystkie istnieją), możesz użyć flagi **-p**:

```
bash-3.2$ mkdir -p /var/log/myapp/error
bash-3.2$ ls /var/log/myapp
error
```

Nawet jeśli katalog `/var/log/myapp` wcześniej nie istniał, uruchomienie **mkdir** z flagą **-p** sprawi, że zostanie on utworzony przed utworzeniem w nim katalogu `/var/log/myapp/error`. Natomiast jeżeli katalog w ścieżce podanej poleceniu **mkdir -p** już istnieje, flaga **-p** nie zaszkodzi mu w żaden sposób, więc można je bezpiecznie uruchamiać wiele razy z rzędu (idempotentność). Czyni to flagę **-p** standardem dla użycia w skryptach.

rmdir — usuwanie pustych katalogów

Polecenie **rmdir** usuwa puste katalogi. Aby zadziało, katalogi muszą być puste, co oznacza, że jest to polecenie stosunkowo bezpieczne do uruchomienia. Większość użytkowników Linuksa używa zamiast niego **rm**, ponieważ potrafi ono robić to samo.

rm — usuwanie plików i katalogów

Aby usunąć plik, należy użyć polecenia `rm`:

```
rm nazwa_pliku
```

W praktyce większość użytkowników stosuje polecenie `rm` również do usuwania katalogów, ponieważ w przeciwieństwie do `rmdir` działa ono dla katalogów, które *nie* są puste. Aby zastosować to polecenie **rekurencyjnie** (do wszystkich katalogów zawartych w tym, który usuwasz), będziesz potrzebować flag `-r` i `-f` w celu wymuszenia (ang. *force*) usuwania bez potwierdzenia dla każdego pliku i katalogu:

```
rm -rf /ścieżka/do/katalogu
```

Uwaga

Podczas korzystania z polecenia `rm -rf` zachowaj szczególną ostrożność, ponieważ Linux pozwoli Ci usunąć katalogi, które są kluczowe dla działania systemu. Na przykład `rm -rf /` wskazuje poleceniu `rm`, że chcesz usunąć katalog główny, który zawiera cały system.

Niektóre dystrybucje Linuksa i uniksowe systemy operacyjne obchodzą to w kreatywny sposób (Ubuntu jest dostarczane z wersją polecenia `rm`, która ma opcję `--no-preserve-root` jako sposób na zapytanie, „czy jesteś pewny, że chcesz to zrobić?”, a w Solarisie celowo użyto luźnej interpretacji tego, co `rm` powinno robić, aby uniknąć usunięcia katalogu głównego). W praktyce zabezpieczenia te można łatwo obejść. Zachowaj ostrożność, gdy używasz polecenia `rm` i wklejasz polecenia do powłoki z internetu!

mv — przenoszenie plików i katalogów lub zmienianie ich nazw

Polecenie `mv` jest sprytne, ponieważ może robić dwie różne rzeczy przy użyciu tej samej składni: przenosić (ang. *move*) pliki z jednego katalogu do drugiego albo zmieniać nazwę pliku, zachowując go w tym samym katalogu.

Najpierw utworzymy plik przy użyciu polecenia `touch`:

```
bash-3.2$ touch foobar.txt
bash-3.2$ ls
foobar.txt
```

Następnie zmienimy nazwę pliku bez jego przenoszenia:

```
bash-3.2$ mv foobar.txt foobarbaz.txt
bash-3.2$ ls
foobarbaz.txt
```

Należy pamiętać, że powyższe polecenie nadpisze każdy istniejący plik o nazwie *foobarbaz.txt* (jeśli taki istnieje), więc podczas zmiany nazwy trzeba zachować ostrożność.

Aby przenieść plik do nowego katalogu, utworzymy nowy katalog, a następnie przeniesiemy do niego plik:

```
bash-3.2$ mkdir targetdir
bash-3.2$ mv foobarbaz.txt targetdir/
```

```
bash-3.2$ ls targetdir/
foobarbaz.txt
```

Te operacje można również łączyć. Jeśli chcesz przenieść plik do innego katalogu i jednocześnie zmienić jego nazwę, możesz to zrobić w ten sposób:

```
bash-3.2$ mv foobarbaz.txt targetdir/renamed.txt
bash-3.2$ ls targetdir/
renamed.txt
```

Uzyskiwanie pomocy

Wszystkie środowiska, z wyjątkiem tych najbardziej minimalnych, są zwykle dostarczane ze stronami podręcznika (ang. *manpages*), będącymi dokumentacją, której można użyć do nauki (lub zapamiętania), jak korzystać z dostępnych programów wiersza poleceń.

Aby uzyskać informacje o określonym poleceniu, wpisz **man** *\$NAZWAPOLECENIA*. Na przykład wpisanie **man ls** wyświetli coś takiego:

```
LS(1)                                     General Commands
Manual                                   LS(1)

NAME
    ls - list directory contents

SYNOPSIS
    ls [-@ABCFGHILOPRSTUWabcdeghiklmnopqrstuvwxyz1%,] [--color=when]
    ↪[-D format] [file ...]

DESCRIPTION
    For each operand that names a file of a type other than directory,
    ↪ls displays its name as well as any requested, associated information.
    ↪For each operand that names a file of type directory, ls displays the
    ↪names of files contained within that directory, as well as any
    ↪requested, associated information.
    If no operands are given, the contents of the current directory are
    ↪displayed. If more than one operand is given, non-directory operands
    ↪are displayed first; directory and non-directory operands are sorted
    ↪separately and in lexicographical order.
    The following options are available:
    -@    Display extended attribute keys and sizes in long (-l) output.
    -A    Include directory entries whose names begin with a dot ('.')
    ↪except for . and ... Automatically set for the super-user unless -I
    ↪is specified.
```

Ponieważ strony podręcznika są automatycznie otwierane w aplikacji pager, do przewijania, wyszukiwania i zamykania stosuje się te same skróty co w przypadku polecenia **less**.

Należy pamiętać, że **man** jest starym narzędziem, które przypomina prawdziwy podręcznik, z różnymi sekcjami (rozdziałami), obejmującymi poszczególne zagadnienia. W powyższym przykładzie (1) w **ls(1)** wskazuje, która sekcja podręcznika jest wyświetlana.

Czasami strona podręcznika o tej samej nazwie będzie istniała w różnych sekcjach. Aby określić sekcję, należy dodać numer przed nazwą polecenia. Na przykład w celu otrzymania tej samej strony podręcznika co powyżej można uruchomić polecenie `man 1 ls`.

Oto sekcje dotyczące większości uniksopodobnych systemów operacyjnych:

1. — polecenia ogólne, czyli polecenia, które zwykle uruchamia się w wierszu poleceń;
2. — wywołania systemowe;
3. — funkcje biblioteki, obejmujące bibliotekę standardową języka C;
4. — pliki specjalne (zazwyczaj urządzenia znajdujące się w `/dev`) i sterowniki;
5. — formaty plików i konwencje (obejmuje to pliki konfiguracyjne);
6. — gry i wygaszacze ekranu;
7. — różne;
8. — polecenia i demony administracyjne systemu.

Tak więc jeśli chcesz zagłębić się w jeden z tematów, które omawiamy w tej książce, prawdopodobnie zaczniesz od sekcji 1., 5. i 8. podręcznika.

Jeżeli nie masz pewności co do nazwy strony podręcznika, której szukasz, to aby ją znaleźć, możesz użyć polecenia `apropos <słowo_kluczowe>` lub `man -k <słowo_kluczowe>`. Spowoduje to wypisanie listy wszystkich stron podręcznika zawierających określone słowo kluczowe.

Autouzupełnianie powłoki

Jeśli jesteś w interaktywnej sesji powłoki (tzn. nie wykonujesz skryptu ani nie tworzysz pliku `Dockerfile`), możesz użyć **autouzupełniania powłoki**, znanego również jako **autouzupełnianie Tab**, aby konstruować polecenia za pomocą mniejszej liczby naciśnięć przycisków i z mniejszym ryzykiem literówek.

Aby skorzystać z autouzupełniania powłoki, zacznij wpisywać nazwę pliku lub katalogu i naciśnij `Tab`. Powłoka będzie stopniowo zawężać wybór, wyświetlając możliwe dopasowania poniżej wpisywanej linii. Gdy na podstawie tego, co wpisałeś, pozostanie tylko jeden wybór, powłoka automatycznie uzupełni to polecenie lub argument i możesz nacisnąć `Enter`. Prześledźmy to na przykładzie.

Jeśli znajdujesz się w katalogu `home` w systemie Linux, może to wyglądać następująco:

```
→ ~ pwd
/home/dave
```

```
→ ~ ls
Desktop
Documents
Downloads
Library
```

```
Movies
Music
Pictures
Public
code
go
```

Jeżeli chcesz przenieść się do katalogu *Documents*, użyj polecenia `cd` (zmień katalog):

```
→ ~ cd Documents
```

Najpierw wpisz `cd D` i naciśnij *Tab*:

```
→ ~ cd D
Desktop/  Documents/  Downloads/
```

Zobaczysz, że powłoka zawęziła dziesięć możliwych wyborów do trzech. Wpisz kolejną literę i ponownie naciśnij *Tab*. Zobaczysz, że pasują tylko dwa elementy:

```
→ ~ cd Do
Documents/  Downloads/
```

Wpisanie kolejnej litery, *c*, zawęzi wybór do jednej opcji, a kolejne naciśnięcie *Tab* automatycznie uzupełni nazwę katalogu:

```
→ ~ cd Documents/
```

Gdy tylko zakończy się autouzupełnianie dla określonej nazwy katalogu, możesz wykonać polecenie jak zwykle za pomocą przycisku *Enter* lub kontynuować autouzupełnianie w tym katalogu. Na przykład ponowne naciśnięcie w tym momencie przycisku *Tab* ponownie uruchomi proces autouzupełniania w katalogu *Documents*, w wyniku czego pozostanie prefiks *Documents/*, a po prawej stronie ukośnika zostaną dodane odpowiednie elementy. Bieżący katalog roboczy powłoki nie zmieni się, dopóki nie podasz prawidłowej ścieżki i nie naciśniesz przycisku *Enter*.

Na przestrzeni czasu ta mała sztuczka pozwoli Ci uniknąć *sporo* wpisywania. Zacznij jej używać jak najszybciej!

Podsumowanie

W tym rozdziale przedstawiliśmy podstawową teorię, którą musisz znać, aby efektywnie pracować w wierszu poleceń. Pokazaliśmy praktyczne przykłady składni wiersza poleceń i wyjaśniliśmy podstawy przyjmowania argumentów przez większość poleceń.

Wprowadziliśmy również pojęcie powłoki i omówiliśmy sposób wyszukiwania plików wykonywalnych po wpisaniu polecenia i naciśnięciu przycisku *Enter*. To zaskakujące, ale wielu zaawansowanych użytkowników nie rozumie w pełni tych dwóch pojęć, co utrudnia im szybkie i wydajne korzystanie ze środowisk wiersza poleceń.

Omówiliśmy także najważniejsze podstawowe polecenia służące do poruszania się po systemie w wierszu poleceń. Będziesz używać ich prawie za każdym razem, gdy będziesz pracować w systemie Linux — stanowią one absolutne podstawy, które każdy musi opanować,

zanim przejdzie dalej. Znasz już nawet swoją pierwszą sztuczkę, która pozwala zaoszczędzić czas, czyli autouzupełnianie powłoki.

Jeśli śledzisz przykłady i wypróbujesz je w prawdziwym systemie Linux (a *powinieneś!*), to zanim przejdiesz do następnego rozdziału, poćwicz przez kilka minut to, czego dotychczas się nauczyłeś. Będziemy opierać się na tej wiedzy przez resztę książki.

Praca z procesami

Jako programista jesteś już intuicyjnie zaznajomiony z procesami. Są to owoce Twojej pracy: po napisaniu i zdebugowaniu kodu Twój program w końcu się wykonuje i przekształca w piękny proces systemu operacyjnego!

W systemie Linux proces może być długo działającą aplikacją, szybkim poleceniem powłoki, takim jak `ls`, lub czymkolwiek, co uruchomi jądro w celu wykonania jakiejś pracy w systemie. Wszystkie zadania wykonywane w Linuksie uruchamiają jakiś proces. Przeglądarka internetowa, edytor tekstu, skaner luk w zabezpieczeniach, a nawet odczytywanie plików i wydawanie poleceń, które poznałeś do tej pory, tworzą proces.

Należy dobrze zrozumieć model procesów Linuksa, ponieważ od zapewnianej przez niego warstwy abstrakcji — linuksowego procesu — zależą wszystkie polecenia i narzędzia, których będziesz używać do zarządzania procesami. Zniknęły szczegóły, do których przywykłeś z perspektywy programisty: zmienne, funkcje i wątki zostały zhermetyzowane w postaci „procesu”. Pozostał inny, zewnętrzny zestaw pokręteł do manipulowania i mierników do sprawdzania: identyfikator procesu, status, wykorzystanie zasobów i wszystkie inne atrybuty procesu, które omówimy w tym rozdziale.

Najpierw przyjrzymy się bliżej samej abstrakcji procesu, a następnie przejdziemy do przydatnych, praktycznych rzeczy, które można robić z procesami Linuksa. Gdy będziemy zajmować się aspektami praktycznymi, zatrzymamy się na chwilę, aby przyjrzeć się kilku kwestiom będącym częstym źródłem problemów, takim jak uprawnienia. Przedstawimy też kilka heurystyk służących do rozwiązywania problemów z procesami.

W tym rozdziale omawiamy następujące zagadnienia:

- proces w systemie Linux i sposoby wyświetlania procesów aktualnie uruchomionych w systemie;
- atrybuty procesu, dzięki którym wiadomo, jakie informacje można zebrać podczas rozwiązywania problemów;
- popularne polecenia do przeglądania i wyszukiwania procesów;
- bardziej zaawansowane tematy, które mogą się przydać programistom piszącym programy wykonywane jako procesy systemu Linux: sygnały i komunikacja międzyprocesowa, wirtualny system plików `/proc`, wyświetlanie otwartych uchwytów plików za pomocą polecenia `ls -l` oraz sposób tworzenia procesów w systemie Linux.

Zapewnimy również praktyczny przegląd wszystkiego, czego się nauczyłeś w przykładowej sesji rozwiązywania problemów, która wykorzystuje teorię i polecenia omówione w tym rozdziale. Przejdźmy teraz do tego, czym dokładnie jest proces w systemie Linux.

Podstawy procesów

Kiedy piszemy o „procesie” w Linuksie, odnosimy się do wewnętrznego modelu systemu operacyjnego, który opisuje, czym dokładnie *jest* uruchomiony program. Linux potrzebuje ogólnej warstwy abstrakcji działającej dla *wszystkich* programów, która może hermetyzować elementy istotne dla systemu operacyjnego. Tą abstrakcją jest proces, który umożliwia systemowi operacyjnemu śledzenie niektórych ważnych kontekstów dotyczących wykonywanych programów. Do tych kontekstów należą:

- wykorzystanie pamięci;
- wykorzystanie czasu procesora;
- wykorzystanie innych zasobów systemowych (dostęp do dysku, wykorzystanie sieci);
- komunikacja między procesami;
- powiązane procesy uruchamiane przez program, na przykład uruchamianie polecenia powłoki.

Listę wszystkich procesów systemowych (przynajmniej tych, które użytkownik może zobaczyć) można uzyskać przez uruchomienie programu `ps` z flagami `aux` (zobacz rysunek 2.1).

```
root@localhost:~# ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.2	0.3	167308	12816	?	Ss	18:55	0:01	/sbin/init
root	2	0.0	0.0	0	0	?	S	18:55	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	I<	18:55	0:00	[rcu_gp]
root	4	0.0	0.0	0	0	?	I<	18:55	0:00	[rcu_par_gp]
root	5	0.0	0.0	0	0	?	I<	18:55	0:00	[slub_flushwq]
root	6	0.0	0.0	0	0	?	I<	18:55	0:00	[netns]
root	8	0.0	0.0	0	0	?	I<	18:55	0:00	[kworker/0:0H-eve
root	10	0.0	0.0	0	0	?	I<	18:55	0:00	[mm_percpu_wq]
root	11	0.0	0.0	0	0	?	S	18:55	0:00	[rcu_tasks_rude_]
root	12	0.0	0.0	0	0	?	S	18:55	0:00	[rcu_tasks_trace]
root	13	0.0	0.0	0	0	?	S	18:55	0:00	[ksoftirqd/0]
root	14	0.0	0.0	0	0	?	I	18:55	0:00	[rcu_sched]
root	15	0.0	0.0	0	0	?	S	18:55	0:00	[migration/0]
root	16	0.0	0.0	0	0	?	S	18:55	0:00	[idle_inject/0]
root	17	0.0	0.0	0	0	?	I	18:55	0:00	[kworker/0:1-cgro
root	18	0.0	0.0	0	0	?	S	18:55	0:00	[cpuhp/0]
root	19	0.0	0.0	0	0	?	S	18:55	0:00	[cpuhp/1]
root	20	0.0	0.0	0	0	?	S	18:55	0:00	[idle_inject/1]
root	21	0.0	0.0	0	0	?	S	18:55	0:00	[migration/1]
root	22	0.0	0.0	0	0	?	S	18:55	0:00	[ksoftirqd/1]
root	23	0.0	0.0	0	0	?	I	18:55	0:00	[kworker/1:0-even

Rysunek 2.1. Lista procesów systemowych

W tym rozdziale omówimy atrybuty najbardziej istotne dla pracy programisty.

Z czego składa się proces Linuksa?

Z perspektywy systemu operacyjnego proces jest po prostu strukturą danych, która ułatwia dostęp do określonych informacji. Należą do nich:

- **Identyfikator procesu (PID)** w powyższych danych wyjściowych z polecenia `ps`). PID 1 to inicjalizator (ang. *init*) systemu — pierwotny proces nadrzędny dla wszystkich pozostałych procesów, który uruchamia system. Uruchomienie go to jedna z pierwszych rzeczy, które robi jądro po rozpoczęciu wykonywania. Gdy proces jest tworzony, otrzymuje następny w kolejności sekwencyjnej dostępny identyfikator procesu. Ponieważ *init* jest tak ważny dla normalnego funkcjonowania systemu operacyjnego, nie może zostać zakończony, nawet przez użytkownika `root`. Poszczególne uniksowe systemy operacyjne stosują różne systemy inicjalizujące, na przykład większość dystrybucji Linuksa korzysta z `systemd`, podczas gdy `macOS` wykorzystuje `launchd`, a wiele innych Uniksów używa `SysV`. Niezależnie od konkretnej implementacji będziemy odnosić się do tego procesu za pośrednictwem nazwy roli, którą pełni: „*init*”.

Uwaga

W kontenerach procesy mają przestrzenie nazw — w „prawdziwym” środowisku wszystkie procesy kontenera mogą mieć PID 3210, podczas gdy ten pojedynczy PID będzie mapował się na wiele procesów (1..*n*, gdzie *n* to liczba uruchomionych procesów w kontenerze). Widać to z zewnątrz, ale nie wewnątrz kontenera.

- **PID procesu nadrzędnego** (ang. *Parent Process PID* — PPID). Każdy proces jest tworzony przez proces nadrzędny. Jeśli proces nadrzędny zostanie zakończony za „życia” swojego procesu podrzędnego, ten drugi staje się „sierotą”. Osierocone procesy są ponownie przypisywane do *init* (PID 1).
- **Status (STAT)** w powyższych danych wyjściowych z polecenia `ps`). Przegląd można wyświetlić za pomocą polecenia `man ps`:
 - D — nieprzerywalnie uśpiony (zwykle dotyczy operacji we-wy);
 - I — beczynny wątek jądra;
 - R — uruchomiony lub możliwy do uruchomienia (w kolejce uruchamiania);
 - S — przerywalnie uśpiony (oczekujący na zakończenie zdarzenia);
 - T — zatrzymany przez sygnał kontroli zadania;
 - t — zatrzymany przez debugger podczas śledzenia;
 - X — zakończony (nigdy nie powinien być widziany);
 - Z — niedziałający proces („zombie”), zakończony, ale nie odzyskany przez swój proces nadrzędny.
- **Status priorytetu** (*niceness* — czy dany proces pozwala innym procesom uzyskiwać priorytet wyższy od swojego?).

- **Właściciel** procesu (USER w powyższych danych wyjściowych z polecenia ps); efektywny identyfikator użytkownika.
- Efektywny **identyfikator grupy** (ang. *Effective Group ID* — EGID), który jest używany.
- **Mapa adresowa** przestrzeni pamięci procesu.
- Użycie zasobów — otwarte pliki, porty sieciowe i inne zasoby wykorzystywane przez proces (**VSZ** i **RSS** dla użycia pamięci w powyższych danych wyjściowych z polecenia ps).

(Zaczerpnięte z: *Unix i Linux. Przewodnik administratora systemów. Wydanie V*).

Przyjrzyjmy się bliżej kilku atrybutom procesu, które są najważniejsze dla programistów i osób okazjonalnie zajmujących się rozwiązywaniem problemów.

Identyfikator procesu (PID)

Każdy proces jest jednoznacznie identyfikowany przez swoje ID, które jest unikatową liczbą całkowitą przypisywaną mu podczas jego uruchamiania. Podobnie jak relacyjna baza danych z identyfikatorami, które jednoznacznie identyfikują każdy wiersz danych, system operacyjny Linux śledzi każdy proces za pomocą jego PID.

PID jest zdecydowanie najbardziej użyteczną etykietą wykorzystywaną w interakcjach z procesami.

Efektywny identyfikator użytkownika (EUID) i efektywny identyfikator grupy (EGID)

Określają one użytkownika i grupę, dla których uruchamiany jest proces. Razem uprawnienia użytkownika i grupy określają, co proces może robić w systemie.

Jak przekonasz się podczas lektury rozdziału 5. „Wprowadzenie do plików”, pliki mają ustawioną własność użytkownika i grupy, co określa, do kogo mają zastosowanie ich uprawnienia. Jeśli przyjmiemy, że własność i uprawnienia pliku są zasadniczo blokadą, proces z odpowiednimi uprawnieniami użytkownika lub grupy jest jak klucz, który otwiera tę blokadę i umożliwia dostęp do pliku. Zgłębijmy tę kwestię później, gdy będziemy omawiać uprawnienia.

Zmienne środowiskowe

Prawdopodobnie używałeś już zmiennych środowiskowych w swoich aplikacjach — umożliwiają one środowisku systemu operacyjnego, które uruchamia dany proces, przekazywanie wymaganych przez niego danych. Zwykle obejmuje to m.in. dyrektywy konfiguracyjne (LOG_DEBUG=1) i tajne klucze (AWS_SECRET_KEY), a każdy język programowania ma jakiś sposób na odczytanie ich z kontekstu programu.

Na przykład poniższy skrypt Pythona pobiera katalog domowy użytkownika ze zmiennej środowiskowej `HOME`, a następnie go wypisuje:

```
import os
home_dir = os.environ['HOME']
print("Katalog główny dla tego użytkownika to", home_dir)
```

W moim przypadku uruchomienie tego programu w środowisku REPL python3 na komputerze z systemem Linux powoduje wyświetlenie następujących danych wyjściowych:

```
Katalog główny dla tego użytkownika to /home/dcohen
```

Katalog roboczy

Proces ma bieżący katalog roboczy, podobnie jak powłoka (która i tak jest tylko procesem). Wpisanie `pwd` w powłoce powoduje wyświetlenie bieżącego katalogu roboczego, a każdy proces ma katalog roboczy. Katalog roboczy procesu może ulec zmianie, więc nie należy się na nim zbytnio opierać.

Na tym kończy się nasz przegląd istotnych atrybutów procesów. W następnym podrozdziale odejdziemy od teorii i przyjrzymy się kilku poleceniom, których można użyć, aby od razu rozpocząć pracę z procesami.

Praktyczne polecenia do pracy z procesami systemu Linux

Oto niektóre z najczęściej używanych poleceń:

- `ps` — wyświetla procesy w systemie (zobacz rysunek 2.2); przykład tego polecenia został przedstawiony wcześniej w tym rozdziale. Flagi modyfikują, które atrybuty procesu są wyświetlane jako kolumny. Polecenie to jest stosowane z filtrami, aby kontrolować ilość otrzymywanych danych wyjściowych, na przykład w celu ograniczenia danych wyjściowych tylko do pierwszych 10 linii (`ps aux | head -n 10`). Jeszcze kilka przydatnych sztuczek:
 - polecenie `ps -eLf` wyświetla informacje o wątkach dla procesów;
 - polecenie `ps -ejH` jest przydatne do wizualnego wyświetlania relacji między procesami nadrzędnymi i podrzędnymi (procesy podrzędne są wcięte pod procesami nadrzędnymi).

```
root@40086047ef36:/# ps -eLf
UID          PID  PPID  LWP  C  NLWP  STIME  TTY          TIME CMD
root          1      0      1  0      1  22:30  pts/0        00:00:00 /bin/bash
root          9      1      9  0      1  22:31  pts/0        00:00:00 ps -eLf
root@40086047ef36:/# ps -ejH
      PID  PGID  SID  TTY          TIME CMD
      1      1      1  pts/0        00:00:00 bash
     10     10      1  pts/0        00:00:00 ps
```

Rysunek 2.2. Przykłady danych wyjściowych z polecenia `ps` z flagami

- **pgrep** — znajduje identyfikatory procesów na podstawie nazwy. Może korzystać z wyrażeń regularnych (zobacz rysunek 2.3).

```
root@localhost:~# pgrep nginx
1589
1592
1593
root@localhost:~# pgrep ^n
6
584
1589
1592
1593
root@localhost:~# █
```

Rysunek 2.3. Przykłady danych wyjściowych z polecenia pgrep z flagami

- **top** — interaktywny program, który sprawdza wszystkie procesy (domyślnie raz na sekundę) i wyświetla posortowaną listę wykorzystania zasobów (można skonfigurować, według czego będzie sortować). Wyświetla również całkowite wykorzystanie zasobów systemowych. Aby zakończyć działanie tego polecenia, należy nacisnąć *Q* lub *Ctrl+C*. Dalej w tym rozdziale pokażemy przykład danych wyjściowych tego polecenia.
- **iotop** — działa podobnie jak **top**, ale dla operacji we-wy dysku. Niezwykle przydatne do znajdowania procesów wymagających dużej liczby operacji we-wy. Nie jest domyślnie instalowane we wszystkich systemach, ale jest dostępne za pośrednictwem większości menedżerów pakietów (zobacz rysunek 2.4).

```
Total DISK READ:      0.00 B/s | Total DISK WRITE:      0.00 B/s
Current DISK READ:    0.00 B/s | Current DISK WRITE:    0.00 B/s
```

TID	PRIO	USER	DISK READ	DISK WRITE	SWAPIN	IO>	COMMAND
1	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		init
2	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[kthreadd]
3	be/0	root	0.00 B/s	0.00 B/s	?unavailable?		[rcu_gp]
4	be/0	root	0.00 B/s	0.00 B/s	?unavailable?		[rcu_par_gp]
5	be/0	root	0.00 B/s	0.00 B/s	?unavailable?		[slub_flushwq]
6	be/0	root	0.00 B/s	0.00 B/s	?unavailable?		[netns]
8	be/0	root	0.00 B/s	0.00 B/s	?unavailable?		[kworker~_highpri]
10	be/0	root	0.00 B/s	0.00 B/s	?unavailable?		[mm_percpu_wq]
11	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[rcu_tasks_rude_]
12	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[rcu_tasks_trace]
13	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[ksoftirqd/0]
14	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[rcu_sched]
15	rt/4	root	0.00 B/s	0.00 B/s	?unavailable?		[migration/0]
16	rt/4	root	0.00 B/s	0.00 B/s	?unavailable?		[idle_inject/0]
18	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[cpuhp/0]
19	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[cpuhp/1]
20	rt/4	root	0.00 B/s	0.00 B/s	?unavailable?		[idle_inject/1]
21	rt/4	root	0.00 B/s	0.00 B/s	?unavailable?		[migration/1]
22	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[ksoftirqd/1]
23	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[kworker~ercpu_wq]

```
keys: any: refresh g: quit i: ionice o: active p: procs a: accum
sort: r: asc left: SWAPIN right: COMMAND home: TID end: COMMAND
CONFIG_TASK_DELAY_ACCT not enabled in kernel, cannot determine SWAPIN and IO %
```

Rysunek 2.4. Przykład danych wyjściowych z polecenia iotop

- **nethogs** — działa podobnie jak **top**, ale dla operacji we-wy sieci. Grupuje wykorzystanie sieci według procesów, co jest niezwykle wygodne. Dostępne za pośrednictwem większości menedżerów pakietów (zobacz rysunek 2.5).

NetHogs version 0.8.6-3

PID	USER	PROGRAM	DEV	SENT	RECEIVED
2064	root	curl	eth0	0.580	8.730 KB/sec
719	root	sshd: root@pts/0	eth0	11.067	0.412 KB/sec
?	root	45.33.83.163:445-201.91..		0.021	0.024 KB/sec
?	root	45.33.83.163:6379-47.57..		0.011	0.014 KB/sec
?	root	45.33.83.163:7777-139.1..		0.000	0.000 KB/sec
2062	root	curl	eth0	0.000	0.000 KB/sec
?	root	45.33.83.163:9010-79.12..		0.000	0.000 KB/sec
2060	root	curl	eth0	0.000	0.000 KB/sec
?	root	45.33.83.163:50153-66.7..		0.000	0.000 KB/sec
?	root	45.33.83.163:36198-35.2..		0.000	0.000 KB/sec
?	root	45.33.83.163:37464-35.2..		0.000	0.000 KB/sec
?	root	45.33.83.163:2443-66.29..		0.000	0.000 KB/sec
?	root	45.33.83.163:57240-23.2..		0.000	0.000 KB/sec
?	root	unknown TCP		0.000	0.000 KB/sec
TOTAL				11.679	9.181 KB/sec

Rysunek 2.5. Przykład danych wyjściowych z polecenia **nethogs**

- **kill** — umożliwia użytkownikom wysyłanie sygnałów do procesów, zwykle w celu ich zatrzymania lub ponownego odczytania ich plików konfiguracyjnych. Sygnały i użycie polecenia **kill** omówimy dalej w tym rozdziale.

Zaawansowane koncepcje i narzędzia związane z procesami

Zaczynamy zaawansowaną część tego rozdziału. Chociaż aby skutecznie pracować z procesami Linuksa, nie musisz opanować wszystkich koncepcji omówionych w tym rozdziale, mogą one być niezwykle pomocne. Jeśli masz kilka dodatkowych minut, zalecamy przynajmniej zapoznanie się z każdym z nich.

Sygnały

W jaki sposób system **ct1** instruuje serwer internetowy, aby ponownie odczytał pliki konfiguracyjne? Jak można poprosić o czyste zamknięcie procesu? I jak można natychmiast zamknąć nieprawidłowo działający proces, ponieważ zakłóca on działanie aplikacji produkcyjnej?

W systemach Unix i Linux wszystko to odbywa się za pośrednictwem sygnałów. Sygnały to komunikaty liczbowe, które mogą być przesyłane między programami. Są sposobem na komunikowanie się procesów między sobą i z systemem operacyjnym, umożliwiając procesom wysyłanie i odbieranie określonych komunikatów.

Komunikaty te mogą być wykorzystywane do przekazywania procesowi różnych informacji, na przykład wskazujących, że miało miejsce określone zdarzenie albo wymagane jest konkretne działanie lub odpowiedź.

Praktyczne zastosowania sygnałów

Przyjrzyjmy się kilku przykładom praktycznych wartości, których przekazywanie umożliwia mechanizm sygnałów. Sygnały mogą być stosowane do implementowania komunikacji między procesami, na przykład jeden proces może wysłać sygnał do drugiego procesu, informując go, że zakończył wykonywanie określonego zadania, i teraz ten drugi proces może rozpocząć swoją pracę. Pozwala to procesom koordynować wzajemne działania oraz płynnie i wydajnie współpracować, podobnie jak ma to miejsce w przypadku wątków wykonawczych w językach programowania (ale bez powiązanego z tym udostępniania pamięci).

Kolejnym powszechnym zastosowaniem sygnałów procesów jest obsługa błędów programu. Można na przykład zaprojektować, by proces przechwytywał sygnał SIGSEGV, który wskazuje błąd segmentacji. Kiedy proces otrzyma ten sygnał, może go przechwycić, a następnie przed czystym zakończeniem wykonywania podjąć działania w celu zarejestrowania błędu, zrobienia zrzutu jądra do celów debugowania lub wyczyszczenia wszelkich używanych zasobów.

Sygnały procesów można również wykorzystać do zaimplementowania czystego zamykania systemu. Na przykład gdy system się wyłącza, do wszystkich procesów może zostać wysłany sygnał, aby za pośrednictwem przechwytywania sygnałów dać im szansę na zapisanie stanu i wyczyszczenie wszelkich zasobów, których używały.

Przechwytywanie

Wiele sygnałów może zostać przechwyconych (ang. *trapping*) przez odbierające je procesy — zasadniczo jest to ta sama koncepcja co wychwytywanie i obsługa błędów w języku programowania.

Jeśli proces odbierający ma funkcję obsługi wysyłanego sygnału, ta funkcja jest uruchamiana. W ten sposób programy ponownie odczytują swoją konfigurację bez restartowania, kończą zapisywanie w bazie danych i zamykają uchwyty plików po otrzymaniu sygnału zamknięcia.

Polecenie kill

Jednak nie tylko procesy komunikują się za pośrednictwem sygnałów: przerażająco (i z technicznego punktu widzenia błędnie) nazwany program `kill` pozwala użytkownikom wysyłać sygnały również do procesów.

Jednym z najczęstszych zastosowań procesów wysyłanych przez użytkownika za pomocą polecenia `kill` jest przerwanie działania procesu, który przestał odpowiadać. Jeśli proces utknął na przykład w nieskończonej pętli, można wysłać sygnał „kill”, aby zmusić go do zatrzymania.

Polecenie `kill` umożliwia wysłanie sygnału do procesu przez określenie jego PID. Jeśli proces, który chcemy zakończyć, ma PID 2600, uruchamiamy następujące polecenie:

```
kill 2600
```

To polecenie wysła do procesu sygnał 15 (SIGTERM, czyli „zakończenie”; ang. *terminate*), a proces będzie miał wtedy szansę przechwycić ten sygnał i zakończyć działanie w czyisty sposób.

Uwaga

Jak pokazaliśmy w tabeli standardowych numerów sygnałów na rysunku 2.6, domyślnym sygnałem, który wysyła `kill`, jest „terminate” (sygnał 15), a nie „kill” (SIGKILL to 9). Program `kill` służy nie tylko do zamykania procesów, ale także do wysyłania wszelkiego rodzaju sygnałów. Ta mylna nazwa to jedna z osobliwości Uniksa i Linuksa, do których się przyzwyczaisz.

Jeśli nie chcesz wysyłać domyślnego sygnału 15, możesz określić pożądany sygnał za pomocą dywizu; aby do tego samego procesu wysłać SIGHUP, należałoby uruchomić następujące polecenie:

```
kill -1 2600
```

Uruchomienie polecenia `man signal` spowoduje wyświetlenie listy sygnałów, które można wysyłać (zobacz rysunek 2.6).

	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see <code>fcntl(2)</code>)
24	SIGXCPU	terminate process	cpu time limit exceeded (see <code>setrlimit(2)</code>)
25	SIGXFSZ	terminate process	file size limit exceeded (see <code>setrlimit(2)</code>)
26	SIGVTALRM	terminate process	virtual time alarm (see <code>setitimer(2)</code>)
27	SIGPROF	terminate process	profiling timer alarm (see <code>setitimer(2)</code>)
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

Rysunek 2.6. Przykład danych wyjściowych z polecenia `man signal`

Warto się zapoznać z kilkoma z nich (czasami przydaje się to w rozmowach o pracę na stanowisku inżyniera):

- SIGHUP (1 — „zawieszenie”) — interpretowany przez wiele aplikacji (takich jak nginx) jako „ponownie odczytaj konfigurację, ponieważ wprowadziłem w niej zmiany”.
- SIGINT (2 — „przerwanie”) — często interpretowane tak samo jak SIGTERM, czyli „proszę o czyste zamknięcie”.
- SIGTERM (15 — „zakończenie”) — wysyła prośbę o zamknięcie procesu.
- SIGUSR1 (30) i SIGUSR2 (31) — czasami używane do przesyłania komunikatów zdefiniowanych przez aplikację. Na przykład SIGUSR1 prosi nginx o ponowne otwarcie plików dziennika, w których zapisujesz, co jest przydatne, jeśli właśnie je zmieniłeś.
- SIGKILL (9) — nie może być przechwytywany ani obsługiwany przez procesy. Jeśli ten sygnał zostanie wysłany do programu, system operacyjny natychmiast zakończy ten program. Nie jest wykonywany żaden kod czyszczący, taki jak czyszczenie zapisów lub bezpieczne zamykanie, więc jest to zazwyczaj ostateczność, ponieważ może prowadzić do uszkodzenia danych.

Jeśli chcesz trochę głębiej zbadać Linuksa, przejrzyj katalog `/proc`. Zdecydowanie wykracza to poza podstawy, ale jest to katalog zawierający poddrzewo systemu plików dla każdego procesu, w którym podczas odczytywania tych plików wyszukiwane są aktualne informacje o procesach.

`/proc`

W praktyce ta wiedza może się przydać podczas rozwiązywania problemów, gdy zidentyfikujesz niewłaściwie zachowujący się (lub tajemniczy) proces i będziesz chciał w czasie rzeczywistym dokładnie wiedzieć, co on robi.

Możesz dowiedzieć się wiele o procesie, przeglądając jego podkatalog `/proc` i wyszukując informacje na jego temat w internecie.

Wiele narzędzi, które prezentujemy w tym rozdziale, w rzeczywistości używa podkatalogu `/proc` do zbierania informacji o procesach i pokazuje tylko podzbiór tego, co można tam znaleźć. Jeśli chcesz zobaczyć *wszystko* i samodzielnie przeprowadzić filtrowanie informacji, zajrzyj do `/proc`.

Isof — wyświetlanie uchwytów plików otwartych przez proces

Polecenie `lsof` wyświetla listę wszystkich plików, które dany proces otworzył do odczytu i zapisu. Jest to przydatne, ponieważ wystarczy tylko jeden mały błąd, aby z programu wyciekły uchwyt plików (wewnętrzne referencje do plików, do których program żądał dostępu). Może to prowadzić do problemów z wykorzystaniem zasobów, uszkodzenia plików i wielu innych dziwnych zachowań.

Na szczęście uzyskanie listy plików otwartych przez proces jest dość proste. Wystarczy uruchomić polecenie `lsdf` i przekazać mu flagę `-p` z PID (zwykle należy je uruchomić z uprawnieniami użytkownika `root`). Spowoduje to zwrócenie listy plików, które otworzył dany proces (w tym przypadku z PID 1589) (zobacz rysunek 2.7):

~ `lsdf -p 1589`

```
root@localhost:~# lsdf -p 1589
COMMAND PID USER  FD  TYPE          DEVICE  SIZE/OFF      NODE NAME
nginx   1589 root   cwd  DIR           8,0     4096         2 /
nginx   1589 root  rtd  DIR           8,0     4096         2 /
nginx   1589 root  txt  REG          8,0    1240136    78096 /usr/sbin/nginx
nginx   1589 root  mem  REG          8,0    184904    262503 /usr/lib/nginx/modules/nginx_stream_module.so
nginx   1589 root  mem  REG          8,0    112264    262494 /usr/lib/nginx/modules/nginx_mail_module.so
nginx   1589 root  mem  REG          8,0    149760     867 /usr/lib/x86_64-linux-gnu/libgpg-error.so.0.32.1
nginx   1589 root  mem  REG          8,0    125488    1035 /usr/lib/x86_64-linux-gnu/libgcc_s.so.1
nginx   1589 root  mem  REG          8,0    2252096    837 /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.30
nginx   1589 root  mem  REG          8,0    29476472    8427 /usr/lib/x86_64-linux-gnu/libcudata.so.70.1
nginx   1589 root  mem  REG          8,0    1296312     840 /usr/lib/x86_64-linux-gnu/libcrypt.so.2.0.3.4
nginx   1589 root  mem  REG          8,0    2062664    8432 /usr/lib/x86_64-linux-gnu/libcuc.so.70.1
nginx   1589 root  mem  REG          8,0     96416    17716 /usr/lib/x86_64-linux-gnu/libxslt.so.0.8.20
nginx   1589 root  mem  REG          8,0    264632    17717 /usr/lib/x86_64-linux-gnu/libxslt.so.1.1.34
nginx   1589 root  mem  REG          8,0    1967384    5818 /usr/lib/x86_64-linux-gnu/libxml2.so.2.9.13
nginx   1589 root  mem  REG          8,0    27672    262485 /usr/lib/nginx/modules/nginx_http_xslt_filter_module.so
```

Rysunek 2.7. Przykład listy plików otwieranych przez proces 1589 wyświetlonej za pomocą polecenia `lsdf -p 1589`

Na rysunku 2.7 pokazaliśmy dane wyjściowe dla procesu serwera WWW `nginx`. Pierwsza linia wskazuje bieżący katalog roboczy procesu — w tym przypadku katalog główny (`/`). Można również zobaczyć, że serwer ma otwarte uchwyty plików na własnej wersji binarnej (`/usr/sbin/nginx`) i różnych bibliotek z lokalizacji `/usr/lib/`.

Dalej w tym rozdziale możesz napotkać kilka ciekawszych ścieżek plików (zobacz rysunek 2.8).

```
nginx   1589 root   DEL   REG          0,1          11 /dev/zero
nginx   1589 root    0u   CHR          1,3          0t0      5 /dev/null
nginx   1589 root    1u   CHR          1,3          0t0      5 /dev/null
nginx   1589 root    2w   REG          8,0          76 262197 /var/log/nginx/error.log
nginx   1589 root    3u  unix 0xffff8da007507300 0t0 23207 type=STREAM
nginx   1589 root    4w   REG          8,0          2910 262196 /var/log/nginx/access.log
nginx   1589 root    5w   REG          8,0          76 262197 /var/log/nginx/error.log
nginx   1589 root    6u  IPv4          23965        0t0      TCP *:http (LISTEN)
nginx   1589 root    7u  IPv6          23966        0t0      TCP *:http (LISTEN)
nginx   1589 root    8u  unix 0xffff8da007505540 0t0 23208 type=STREAM
nginx   1589 root    9u  unix 0xffff8da007504000 0t0 29795 type=STREAM
nginx   1589 root   10u  unix 0xffff8da007505dc0 0t0 29796 type=STREAM
```

Rysunek 2.8. Dalszy ciąg listy plików otwartych przez proces 1589

Ta lista zawiera pliki dziennika, w których zapisuje `nginx`, a ponadto znajdziesz na niej pliki gniazd (Uniksa, IPv4 i IPv6), z których odczytuje i w których zapisuje. W systemach Unix i Linux gniazda sieciowe są specjalnym rodzajem pliku, co ułatwia korzystanie z tego samego podstawowego zestawu narzędzi w wielu różnych przypadkach użycia — narzędzia pracujące z plikami są niezwykle wydajne w środowisku, w którym prawie wszystko jest reprezentowane jako plik.

Dziedziczenie

Z wyjątkiem pierwszego procesu, init (PID 1), wszystkie procesy są tworzone przez proces nadrzędny, który zasadniczo tworzy kopię samego siebie, a następnie „rozwidla” (rozdziela) tę kopię. Gdy proces jest rozwidlany, zazwyczaj dziedziczy uprawnienia, zmienne środowiskowe i inne atrybuty procesu nadrzędnego.

Chociaż temu domyślnemu zachowaniu można zapobiec i można je zmienić, stanowi to pewne ryzyko związane z bezpieczeństwem: oprogramowanie uruchamiane ręcznie otrzymuje uprawnienia bieżącego użytkownika (lub nawet uprawnienia użytkownika root, jeśli używasz sudo). Te uprawnienia dziedziczą wszystkie procesy podrzędne, które mogą zostać utworzone przez ten proces, np. podczas instalacji, kompilacji itd.

Wyobraź sobie proces serwera WWW, który został uruchomiony z uprawnieniami roota (aby mógł łączyć się z portem sieciowym) i zmiennymi środowiskowymi zawierającymi klucze uwierzytelniania w chmurze (aby mógł pobierać dane z chmury). Kiedy ten główny proces rozwidla się na proces podrzędny, który nie wymaga uprawnień roota ani poufnych zmiennych środowiskowych, przekazanie ich procesowi potomnemu jest niepotrzebnym ryzykiem bezpieczeństwa. W rezultacie zmniejszanie uprawnień i usuwanie zmiennych środowiskowych jest powszechnym wzorcem w usługach tworzących procesy podrzędne.

Z punktu widzenia bezpieczeństwa należy pamiętać o tym, aby zapobiegać sytuacjom, w których mogą wyciekać informacje, takie jak hasła lub dostęp do poufnych plików. Chociaż zagłębienie się w szczegóły dotyczące sposobów unikania tego problemu wykracza poza zakres niniejszej książki, warto mieć tego świadomość, jeśli piszesz oprogramowanie, które będzie działać w systemach Linux.

Przegląd — przykładowa sesja rozwiązywania problemów

Przyjrzyjmy się przykładowej sesji rozwiązywania problemów. Wiemy tylko, że jeden konkretny serwer linuksowy działa bardzo wolno.

Na początek chcemy zobaczyć, co dzieje się w systemie. Z lektury tego rozdziału wiesz, że przez uruchomienie interaktywnego polecenia `top` możesz zobaczyć podgląd na żywo procesów działających w systemie (zobacz rysunek 2.9). Wypróbujmy je teraz.

Domyślnie polecenie `top` sortuje procesy według użycia procesora, dlatego aby znaleźć problematyczny proces, wystarczy przyjrzeć się pierwszemu procesowi z listy. I faktycznie ten pierwszy proces z listy wykorzystuje 94% dostępnego czasu przetwarzania jednego procesora.

W wyniku uruchomienia polecenia `top` otrzymaliśmy kilka przydatnych informacji:

- Problemem jest właśnie użycie procesora, a nie innego rodzaju zasobów.
- Problematyczny proces to PID 1763, a uruchamiane polecenie (wymienione w kolumnie `COMMAND`) to `gzip`, które jest programem kompresji.

```
top - 19:06:27 up 10 min, 2 users, load average: 0.08, 0.03, 0.01
Tasks: 108 total, 3 running, 105 sleeping, 0 stopped, 0 zombie
%Cpu(s): 46.8 us, 2.0 sy, 0.0 ni, 48.8 id, 2.3 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 3924.0 total, 3260.3 free, 145.5 used, 518.2 buff/cache
MiB Swap: 512.0 total, 512.0 free, 0.0 used, 3552.6 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1763	root	20	0	10168	7820	1436	R	94.7	0.2	0:03.14	bzip2
1761	root	20	0	7324	3100	2872	S	3.0	0.1	0:00.11	tar
1	root	20	0	167308	12816	8300	S	0.0	0.3	0:01.82	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_par_gp
5	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	slub_flushwq
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	netns
8	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:0H-events_highpri
10	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_wq
11	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_tasks_rude_
12	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_tasks_trace
13	root	20	0	0	0	0	S	0.0	0.0	0:00.02	ksoftirqd/0
14	root	20	0	0	0	0	R	0.0	0.0	0:00.03	rcu_sched
15	root	rt	0	0	0	0	S	0.0	0.0	0:00.00	migration/0
16	root	-51	0	0	0	0	S	0.0	0.0	0:00.00	idle_inject/0
17	root	20	0	0	0	0	I	0.0	0.0	0:00.33	kworker/0:1-mm_percpu_wq
18	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/0
19	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/1

Rysunek 2.9. Przykład danych wyjściowych z polecenia top

Ustalamy, że proces `bzip2` nie musi tu działać, i postanawiamy go zatrzymać. Za pomocą polecenia `kill` wysyłamy żądanie zakończenia tego procesu:

```
kill 1763
```

Po odczekaniu kilku sekund sprawdzamy, czy uruchomiony jest ten (lub jakikolwiek inny) proces `bzip2`:

```
pgrep bzip2
```

Niestety widzimy, że to samo PID nadal działa. Czas wziąć się za to na poważnie:

```
kill -9 1763
```

Nakazuje to systemowi operacyjnemu zamknięcie procesu bez umożliwienia mu przechwycenia (i potencjalnie zignorowania) sygnału. `SIGKILL` (sygnał 9) zamyka proces bezwarunkowo.

Ponieważ zamknęliśmy problematyczny proces, serwer znów działa płynnie i możemy spróbować wysledzić programistę, który uznał, że dobrym pomysłem jest kompresowanie dużych katalogów źródłowych na tym komputerze.

W tym przykładzie zastosowaliśmy najpopularniejszy istniejący schemat rozwiązywania problemów z systemami:

1. Przyjrzelismy się wykorzystaniu zasobów (w tym przykładzie za pomocą polecenia `top`). Może to być dowolne z pozostałych omówionych narzędzi, w zależności od tego, który zasób się wyczerpuje.
2. Znaleźliśmy PID do zbadania.

3. Podjęliśmy działania związane z tym procesem. W tym przykładzie dalsze dochodzenie nie było konieczne i wysłaliśmy sygnał z żądaniem zamknięcia (15, czyli SIGTERM).

Podsumowanie

W tym rozdziale przyjrzelśmy się uważnie warstwie abstrakcji, którą Linux opakowuje wykonywanie programów — procesowi. Wyjaśniliśmy, jakie wspólne komponenty mają wszystkie procesy, i omówiliśmy podstawowe polecenia do znajdowania i sprawdzania uruchomionych procesów. Dzięki tym narzędziom będziesz w stanie zidentyfikować, kiedy proces zachowuje się niewłaściwie, a co ważniejsze, *który* to proces.

Zarządzanie usługami za pomocą usługi systemd

Rozdział

3

W poprzednim rozdziale pokazaliśmy, jak działają procesy w systemie Linux. Teraz nadszedł czas przyjrzeć się, w jaki sposób te procesy są opakowywane w dodatkową warstwę abstrakcji: **usługę systemd**.

Polecenia, które omówiliśmy do tej pory, takie jak `ls`, `mv`, `rm` i `ps`, działają na pierwszym planie, dołączone do sesji powłoki. Uruchamiamy je, programy te wykonują swoją pracę, a potem kończą działanie. Jednak nie wszystkie programy działają w ten sposób.

Usługi, często nazywane również demonami, to długo działające procesy, które pracują w tle. Mogą to być bazy danych i serwery WWW, ale także zwykłe usługi systemowe, takie jak menedżer sieci, środowisko pulpitu itd. Te długo działające usługi w tle są zazwyczaj uruchamiane i kontrolowane za pośrednictwem systemu **init**, takiego jak **systemd**.

init odnosi się tutaj do pierwszego procesu uruchamianego przez jądro systemu operacyjnego, a zadaniem tego procesu jest zadbanie o uruchomienie innych procesów.

Usługi **systemd** są kontrolowane za pomocą narzędzia wiersza poleceń o nazwie `systemctl`. Będziemy go używać do uruchamiania i zatrzymywania usług, na przykład do ponownego uruchomienia usługi, która zachowuje się nieprawidłowo, lub do ponownego załadowania usługi, której konfiguracja uległa zmianie.

Jeśli czytasz tę książkę wyrywkowo i nie zapoznałeś się jeszcze z poprzednim rozdziałem, również możesz zaczerpnąć wiele informacji z niniejszego. Na razie potraktuj proces jak dowolne uruchomione polecenie, aplikację lub usługę. Kiedy zechcesz bliżej poznać działanie procesów, możesz przeczytać rozdział 2. „Praca z procesami”.

W tym rozdziale omawiamy następujące zagadnienia:

- polecenie `systemctl`, które służy do interakcji z usługami **systemd**;
- zadania systemu **init** i jak tę rolę spełnia **systemd**;
- zarządzanie usługami za pomocą polecenia `systemctl`;
- kilka wskazówek dotyczących pracy w środowiskach kontenerowych (takich jak kontenery Docker), które zwykle nie mają solidnej warstwy zarządzania usługami, jaką opiszemy w tym rozdziale.

Uwaga

Ten rozdział dotyczy tylko Linuksa, natomiast macOS i Windows (a nawet inne systemy uniksowe) zarządzają procesami przy użyciu innych narzędzi. Tak naprawdę nawet poszczególne dystrybucje Linuksa używają różnych narzędzi, jednak systemd jest najczęściej stosowanym. Chociaż koncepcje są podobne, wiedza o tym, jak nowoczesne środowiska Linux zarządzają usługami, jest najbardziej przydatna dla programistów.

Podstawy

Usługi linuksowe to procesy działające w tle w systemie Linux w celu wykonywania określonych zadań. Przypominają usługi Windowsa i demony w systemie macOS.

Większość niekontenerowych środowisk linuksowych do zarządzania usługami wykorzystuje systemd. Do interakcji z systemd będziesz używać dwóch narzędzi:

- `systemctl` — kontroluje usługi (w nomenklaturze systemd zwane jednostkami);
- `journalctl` — umożliwia pracę z dziennikami systemowymi.

W tym rozdziale omówimy `systemctl`, a narzędziem `journalctl` zajmiemy się w rozdziale 16. „Monitorowanie dzienników aplikacji”.

systemd to menedżer systemu i usług dla Linuksa, zapewniający standardowy sposób zarządzania usługami. Obecnie jest szeroko stosowany jako domyślny system init dla większości dystrybucji Linuksa. Wiele dystrybucji korzystało wcześniej z systemów init SysV, które pochodzą z Uniksa i są nadal używane przez wiele nowoczesnych uniksowych systemów operacyjnych. Inne dystrybucje, takie jak Alpine i Gentoo Linux, jako swoich systemów init używają OpenRC. Istnieje wiele innych systemów init, jednak zdecydowana większość dystrybucji Linuksa korzysta teraz z systemd. Za pomocą systemd można uruchamiać, zatrzymywać, restartować, włączać (ustawiać na uruchamianie przy rozruchu) i wyłączać usługi oraz sprawdzać ich status. Usługi są definiowane przez **pliki jednostek**, które dokładnie określają, w jaki sposób poszczególne usługi powinny być zarządzane przez systemd.

Aby zarządzać usługami za pomocą systemd, możesz używać następujących podstawowych poleceń (omówimy każde z nich dalej w tym rozdziale):

- `systemctl start <usługa>` — uruchamia usługę;
- `systemctl stop <usługa>` — zatrzymuje usługę;
- `systemctl restart <usługa>` — restartuje usługę;
- `systemctl status <usługa>` — wyświetla aktualny status usługi.

Pamiętaj, że tylko użytkownicy z uprawnieniami roota (np. używający `sudo`) mogą zarządzać usługami systemowymi za pomocą systemd.

init

Zróbmy krótką dygresję, aby zdefiniować powszechnie stosowany termin, z którym często będziesz się spotykać. W systemie Linux **init** (skrót od angielskiego słowa *initialization*, oznaczającego inicjalizowanie) jest pierwszym procesem uruchamianym po zbootowaniu systemu. Nic dziwnego, że można go znaleźć w PID 1 — **init** jest odpowiedzialny za zarządzanie procesem rozruchowym i uruchamianie wszystkich innych procesów i usług, które zostały skonfigurowane do działania w systemie. Ponadto „przygarnia” osierocone procesy (te, których pierwotny proces nadrzędny został zakończony) i utrzymuje je jako własne procesy potomne, aby zapewnić ich prawidłowe zachowanie.

Jak to zwykle bywa w świecie Linuksa, istnieje kilka różnych, wzajemnie wykluczających się programów, które mogą pełnić tę rolę. Wszystkie są określane jako **systemy init**, co jest ogólną nazwą każdego oprogramowania, które może pełnić tę ważną rolę bootstrapowania (ładowania początkowego), inicjalizowania i koordynowania. Jak wspomnieliśmy wcześniej, istnieje kilka systemów **init** dostępnych dla Linuksa, w tym **System V init (SysV)**, **OpenRC** i **systemd**. Większość nowoczesnych systemów Linux przeszła na **systemd**, dlatego właśnie nim zajmujemy się w tym rozdziale.

Używany przez Ciebie system **init** będzie określał sposób definiowania usług i zarządzania nimi, dlatego pamiętaj, że wszystko, co omawiamy w tym rozdziale, dotyczy tylko **systemd**.

Procesy i usługi

Przyjrzyjmy się subtelnej różnicy między procesami a usługami. Usługę możesz potraktować jak opakowanie dla elementu oprogramowania, ułatwiające zarządzanie nim jako uruchomionym procesem.

Usługa dodaje wygodne funkcjonalności do sposobu, w jaki program (i wynikowy proces tworzony przez ten program) jest obsługiwany przez system. Pozwala na przykład definiować zależności między różnymi procesami, kontrolować kolejność uruchamiania, dodawać zmienne środowiskowe, z którymi uruchamiany jest proces, ograniczać wykorzystanie zasobów, kontrolować uprawnienia oraz robić wiele innych przydatnych rzeczy. Mówiąc w skrócie, usługa zapewnia prostą nazwę, za pomocą której można odwoływać się do programu. Tworzenie własnych usług omówimy w rozdziale 10. „Konfigurowanie oprogramowania”.

Przez resztę tego rozdziału będziemy trzymać się zarządzania istniejącymi usługami.

Polecenia **systemctl**

Do zarządzania usługami zdefiniowanymi w systemie będziesz używać narzędzia **systemctl**. W przykładach będziemy wykorzystywać usługę **foobar** (która tak naprawdę nie istnieje) jako symbol zastępczy dla dowolnej usługi, którą możesz zarządzać.

Sprawdzanie statusu usługi

Polecenie `systemctl status <usługa>` pozwala sprawdzać stan usługi. Otrzymujesz z niego szereg danych, które są przydatne do wszelkiego rodzaju zadań związanych z rozwiązywaniem problemów. Na rysunku 3.1 pokazaliśmy, jak wyglądają dane wyjściowe dla usługi serwera WWW nginx.

```
root@localhost:~# systemctl status nginx
● nginx.service - A high performance web server and a reverse proxy server
   Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2023-02-06 21:47:43 UTC; 3min 54s ago
     Docs: man:nginx(8)
  Process: 1503 ExecStartPre=/usr/sbin/nginx -t -q -g daemon on; master_process on; (code=exited, status=0/SUCCESS)
  Process: 1504 ExecStart=/usr/sbin/nginx -g daemon on; master_process on; (code=exited, status=0/SUCCESS)
 Main PID: 1598 (nginx)
    Tasks: 3 (limit: 4575)
   Memory: 5.4M
      CPU: 20ms
   CGroup: /system.slice/nginx.service
           └─1598 "nginx: master process /usr/sbin/nginx -g daemon on; master_process on;"
             └─1601 "nginx: worker process"
               └─1602 "nginx: worker process"

Feb 06 21:47:43 localhost systemd[1]: Starting A high performance web server and a reverse proxy server...
Feb 06 21:47:43 localhost systemd[1]: Started A high performance web server and a reverse proxy server.
```

Rysunek 3.1. Dane wyjściowe dla usługi serwera WWW nginx

Przeanalizujmy wiersz po wierszu informacje, które można znaleźć w obszernych danych wyjściowych generowanych przez to polecenie:

- **Nazwa usługi** — nazwa danej usługi zgodnie z definicją z pliku jednostki.
- **Stan załadowania** — informuje, czy plik jednostki danej usługi został pomyślnie załadowany i czy usługa jest gotowa do uruchomienia.
- **Stan aktywności** — aktualny stan usługi i czas, przez jaki usługa znajduje się w tym stanie, niezależnie od tego, czy jest uruchomiona, nieaktywna czy zakończyła się niepowodzeniem.
- **Dokumentacja** — strona główna, na której można znaleźć odpowiednią dokumentację, jeśli została zainstalowana.
- **Główny PID i procesy potomne** — ID procesu (PID) dla procesu głównego związanego z daną usługą, z dodatkowymi wpisami dla wszystkich uruchomionych procesów podrzędnych.
- **Wykorzystanie zasobów** — czas wykorzystania pamięci RAM i procesora.
- **CGroup** — szczegóły dotyczące grupy kontrolnej, do której należy ten proces.
- **Podgląd dziennika** — kilka linii dziennika z danych wyjściowych usługi, aby zapewnić wyobrażenie o tym, co się dzieje.

Te informacje zawierają szczegółowy przegląd usługi i jej statusu i mogą być przydatne do debugowania problemów lub sprawdzania kondycji usługi.

Jeśli usługa zakończyła się niepowodzeniem, dane wyjściowe zazwyczaj zawierają szczegółowe informacje o tym, dlaczego tak się stało, takie jak kod wyjścia lub opis błędu.

Uruchamianie usługi

```
systemctl start foobar
```

To polecenie powoduje uruchomienie usługi. Jeżeli usługa była już uruchomiona, polecenie nie będzie miało wpływu.

Zatrzymywanie usługi

```
systemctl stop foobar
```

To polecenie powoduje zatrzymanie usługi. Jeżeli usługa nie działa, polecenie nie powinno mieć żadnego wpływu.

Restartowanie usługi

```
systemctl restart foobar
```

To polecenie powoduje zatrzymanie, a następnie uruchomienie usługi. Jest to równoznaczne z wykonaniem poniższych poleceń:

```
systemctl stop foobar  
systemctl start foobar
```

Uwaga

Zachowaj ostrożność przy stosowaniu tego polecenia: jeśli od czasu uruchomienia usługi jej plik konfiguracyjny na dysku się zmienił i zawiera błąd, który uniemożliwia pomyślne uruchomienie tego programu, to polecenie restart zatrzyma uruchomioną usługę, ale nie będzie w stanie uruchomić jej ponownie.

To logiczne, ale potencjalnie niepożądane zachowanie dało się we znaki na przestrzeni lat wielu programistom, więc przed zrestartowaniem upewnij się, że konfiguracja usługi jest nadal prawidłowa. Wiele popularnych programów ma wbudowaną walidację konfiguracji, na przykład aby przetestować konfigurację na dysku dla serwera nginx, można uruchomić następujące polecenie:

```
nginx -t
```

Ponowne załadowywanie usługi

```
systemctl reload foobar
```

Nie wszystkie usługi obsługują to podpolecenie — jego implementacja zależy od osoby tworzącej konfigurację usługi. Jeśli usługa ma opcję przeładowania, jest to na ogół bezpieczniejsze niż polecenie restart.

Zazwyczaj polecenie `reload` wykonuje następujące działania:

- ponownie sprawdza konfigurację na dysku, aby upewnić się, że jest poprawna;
- ponownie wczytuje tę konfigurację do pamięci bez przerywania uruchomionego procesu, jeśli to możliwe;
- uruchamia ponownie proces dopiero po sprawdzeniu poprawności konfiguracji i upewnieniu się, że po zatrzymaniu proces rozpocznie się pomyślnie.

Podobnie jak wiele rzeczy w Linuksie, jest to raczej konwencja niż rygorystyczny wymóg, więc możesz natknąć się na oprogramowanie, które:

- nie implementuje podpolecenia `reload`;
- nie implementuje niektórych funkcji bezpieczeństwa omówionych powyżej (walidacja konfiguracji itp.);
- robi coś innego za pomocą `reload`, ponieważ programista lub twórca pakietu uznał, że to dobry pomysł.

Ogólnie rzecz biorąc, podczas aktualizacji pliku konfiguracyjnego dla aplikacji, zwłaszcza w środowiskach produkcyjnych, należy korzystać raczej z polecenia `reload` niż `restart`.

Enable i disable

Polecenie `systemctl enable foobar` konfiguruje usługę `foobar` tak, aby uruchamiała się automatycznie podczas rozruchu. Jeśli usługa `foobar` jest skonfigurowana do automatycznego uruchamiania, polecenie `systemctl disable foobar` powoduje pozbycie się tej konfiguracji i zmianę `foobar` w usługę zarządzaną ręcznie.

Kluczowa różnica polega na tym, że polecenia `start` i `stop` mają natychmiastowy skutek — zapewniają, że usługa jest uruchamiana (lub zatrzymywana) *w tej chwili*; polecenia `enable` i `disable` dotyczą przyszłych startów systemu. Nie mają one jednak wpływu na status „działa” danej usługi w momencie uruchamiania polecenia.

Jednym z typowych błędów popełnianych przez programistów jest założenie, że `enable` uruchomi usługę. Tak się nie stanie. Jeśli chcesz uruchomić serwer WWW `nginx` w danym momencie i sprawić, aby był on uruchamiany automatycznie przy każdym ponownym uruchomieniu maszyny wirtualnej, musisz wykonać dwa polecenia:

```
systemctl start nginx
systemctl enable nginx
```

Z tego powodu polecenia `enable` i `disable` zawierają opcjonalną flagę, która dodatkowo uruchamia (w przypadku `enable`) lub zatrzymuje (w przypadku `disable`) usługę. Równoważne dwóm powyższym poleceniom jest następujące polecenie:

```
systemctl enable --now nginx
```

Kilka słów na temat Dockera

Chociaż `systemctl` jest powszechnym narzędziem do zarządzania usługami w tradycyjnych systemach Linux, zasadniczo nie jest stosowane w kontenerach Dockera ze względu na izolowany i niezależny charakter kontenerów.

Kontenery Dockera z założenia mają uruchamiać pojedynczy proces i dlatego nie wymagają skomplikowanej fazy rozruchu ani zarządzania procesami. Kontener sam w sobie *jest* procesem i nie ma dostępu do systemu `init` systemu hosta (w tym do narzędzia `systemd`).

Mimo że jest możliwe posiadanie dostępu do tych poleceń w kontenerze Dockera, zwykle nie jest pożądane używanie w nim jakiegokolwiek systemu zarządzania usługami.

Kontenery Dockera zasadniczo powinny zawierać jedną aplikację i po uruchomieniu inicjować pojedynczy proces. W tym celu nie jest potrzebne żadne zarządzanie usługami — uruchomiony kontener to pakiet usług, a kontener Dockera *jest* w istocie procesem.

Nie zalecamy konfiguracji Dockera, która obejmuje wiele procesów lub znaczące wewnętrzne zarządzanie usługami, więc nie będziemy się tym zajmować.

Podsumowanie

W tym rozdziale omówiliśmy sposób zarządzania usługami w systemie Linux i wprowadziliśmy praktyczne polecenia, których będziesz używać do sterowania nimi. Przedstawiliśmy teorię niezbędną, aby zrozumieć całą terminologię napotykaną w systemie: czym jest `init`, jaka jest rola `systemd` w systemach Linux i jakich poleceń potrzeba, aby wchodzić w interakcję z tym narzędziem.

W następnym rozdziale pokażemy kilka przydatnych sztuczek do interakcji z powłoką i historią poleceń, dzięki czemu zaoszczędzisz czas i poczujesz się jak uniksowy magik ze swojego ulubionego filmu (czyli zwiększysz szybkość i wydajność w swojej codziennej pracy).

Korzystanie z historii powłoki

Aby sprawnie się posługiwać wierszem poleceń, musisz używać go regularnie. Nie ma drogi na skróty, ale jest kilka wartościowych sztuczek, których możesz się nauczyć na wczesnym etapie, co pozwoli Ci zaoszczędzić czas i uniknąć frustracji. Im wcześniej wykształcisz sobie pamięć mięśniową w tym zakresie, tym lepiej.

W tym rozdziale pokażemy, jak wykorzystać historię powłoki, aby uniknąć żmudnego ponownego wpisywania uruchomionych już raz poleceń. Zobaczysz również, jak dostosować zachowanie lub wygląd powłoki za pomocą jej pliku konfiguracyjnego. Na koniec zaprezentujemy najbardziej przydatne skróty do edytowania i modyfikowania poleceń w wierszu poleceń. Podsumowując, lektura tego rozdziału szybko pozwoli Ci zacząć płynnie poruszać się w wierszu poleceń.

W tym rozdziale omawiamy następujące zagadnienia:

- historia powłoki;
- wykonywanie poprzednich poleceń za pomocą !;
- przeskakiwanie do początku lub końca linii.

Zacznijmy od przyjrzenia się historii powłoki.

Historia powłoki

Większość powłok przechowuje historię wykonanych poleceń. Oznacza to, że możesz zobaczyć każde uruchomione z powodzeniem polecenie przez naciskanie przycisków strzałek: *strzałka w górę* pozwala cofnąć się o jedno polecenie, a *strzałka w dół* przejść o jedno polecenie do przodu. Przewijanie historii powłoki w ten sposób może być bardzo przydatne, zwłaszcza jeśli okaże się, że często ponownie uruchamiasz podobne polecenia.

Pamiętaj, że możesz również edytować znalezione polecenia w następujący sposób: użyj *strzałki w lewo* i *strzałki w prawo*, aby przejść do linii tekstu danego polecenia, i po prostu wpisać żądane zmiany w celu edytowania polecenia.

Edytowane polecenie jest dodawane do końca historii powłoki (tak naprawdę nie modyfikuje zapisanej w historii linii).

Wszystkie te sztuczki pozwalają łatwo cofać się i ponownie wykonywać lub modyfikować poprzednie polecenia.

Pliki konfiguracyjne powłoki

Niektóre omawiane w tym rozdziale sztuczki wymagają wprowadzania zmian w pliku konfiguracyjnym powłoki. Przepływ pracy jest zwykle następujący:

1. Zmień żadaną opcję w pliku konfiguracyjnym powłoki.
2. Zapisz plik.
3. Otwórz nową sesję powłoki, aby zobaczyć zmiany.
4. W przypadku istniejących sesji powłoki ponownie wczytaj plik konfiguracyjny powłoki przez uruchomienie polecenia `source` (wykonaj): `source ~/ścieżka/do/pliku/konfiguracyjnego`.

Oto lokalizacje dla najpopularniejszych powłok:

Popularne powłoki	Lokalizacje
Bash	<code>~/.bashrc</code> dla sesji interaktywnych, takich jak ta, którą otrzymujesz po otwarciu nowego okna terminala w środowisku graficznym. Prawie zawsze jest to niezbędne, jeśli zmieniasz konfigurację na swojej maszynie roboczej. Określenie „interaktywna” odnosi się tutaj do sytuacji, w których jako użytkownik korzystasz z powłoki w terminalu, a nie do uruchomionego skryptu (na przykład do skryptu automatycznie wywoływanego przez zadanie crona). „Interaktywnej powłoki” używasz, gdy znajdujesz się w jakiejś formie terminala, w której musisz ręcznie wprowadzać i wpisywać polecenia. <code>~/.bash_profile</code> dla powłok logowania — może to być logowanie lokalne lub to, co otrzymujesz przy logowaniu się przez SSH. Również w tym przypadku porównujemy to do instancji powłoki używanych podczas uruchamiania skryptu.
Zsh (powłoka Z)	<code>~/.zshrc</code>

Pliki historii

Różne powłoki przechowują pliki historii w różnych miejscach, a większość z nich można skonfigurować tak, aby zmieniać lokalizację. Domyślnie prawie zawsze używasz powłoki Bash, która domyślnie przechowuje swój plik historii w lokalizacji `~/.bash_history`.

Jeśli kiedyś nie będziesz mieć pewności, gdzie znaleźć plik historii powłoki, wiele powłok ma opcję konfiguracji o nazwie `HISTFILE`, która zawiera lokalizację pliku historii.

Tutaj sprawdzamy, gdzie znajduje się plik historii podczas uruchamiania systemu operacyjnego zsh:

```
% echo $HISTFILE
/home/dcohen/.zsh_history
```

Bash ma dwie opcje konfiguracji, które zapobiegają nieograniczonemu rozrastaniu się pliku historii, aby umożliwić zarządzanie jego rozmiarem i szybkie wyszukiwanie historii:

- HISTSIZE — kontroluje maksymalną ilość historii przechowywanej w pamięci;
- HISTFILESIZE — steruje maksymalnym rozmiarem pliku historii zapisywanym między sesjami powłoki.

Jeśli chcesz, aby Bash przechowywał więcej historii, zwiększ wartości dla obu tych opcji w pliku konfiguracyjnym powłoki.

W tym celu otwórz plik konfiguracyjny powłoki (np. `~/.bashrc`) i ustaw te zmienne, dołączając na końcu pliku następujące linie:

```
export HISTSIZE=1000
export HISTFILESIZE=5000
```

Przeszukiwanie historii powłoki

Często będziesz szukał polecenia, które uruchomiłeś tydzień (lub miesiąc) temu. To polecenie prawdopodobnie znajdzie się dalej w Twojej historii i klikanie *strzałki w górę* setki razy, aby się tam dostać, byłoby stratą czasu. Jeśli masz przynajmniej *jakikolwiek* pojęcie o tym, czego szukasz, potrzebujesz sztuczki interaktywnego przeszukiwania historii powłoki. Oto jak przeszukać historię powłoki:

1. Naciśnij kombinację klawiszy *Ctrl+R*, aby wywołać reverse-i-search (wyszukiwanie odwrotne).
2. Wpisz część polecenia, którego szukasz.
3. Powłoka spróbuje dopasować wpisane znaki do historii poleceń i znaleźć najbliższe, najnowsze dopasowanie.
4. Aby przejść przez historię, wielokrotnie naciskaj *Ctrl+R*. Naciśnij *Enter*, aby wybrać polecenie, lub *Esc*, aby wyjść z tego trybu.
5. Jeśli przypadkowo przeskoczysz za daleko wstecz i pominiesz żądane polecenie, wciśnięcie kombinacji przycisków *Ctrl+Shift+R* wyszukuje następnego najnowszego dopasowania.

Wyjątki

Istnieją pewne wyjątki dla tej funkcjonalności zależnie od powłoki i konfiguracji.

Niektóre powłoki zapominają polecenia, których uruchomienie zakończyło się z błędem (z niezerowym kodem wyjścia). Wiele powłok zapomina również polecenia zaczynające się od znaku spacji — nie są one dodawane do historii. Jednak w obu scenariuszach zazwyczaj nadal możesz dotrzeć do wpisu historii, jeśli od razu, bez wykonywania żadnych innych poleceń, wrócisz za pomocą *strzałki w górę*.

Wykonywanie poprzednich poleceń za pomocą !

Wykonywanie poprzednich poleceń odbywa się za pomocą wykrzykników. Istnieją różne sposoby wykorzystania tej sztuczki, którym teraz się przyjrzymy.

Ponowne uruchamianie polecenia z tymi samymi argumentami

Polecenie ! spowoduje wykonanie ostatniego polecenia z poprzednimi argumentami. Na przykład wpisanie polecenia ! ssh spowoduje wyszukiwanie wstecz w celu znalezienia ostatniego uruchomionego polecenia ssh i wykonania go z tymi samymi argumentami. Możesz stosować je do ponownego uruchamiania poleceń, których często używasz z tymi samymi argumentami, na przykład do szybkiego ponownego połączenia się z serwerem SSH, z którym łączysz się każdego dnia.

Dołączanie do polecenia jakiegoś polecenia z historii

Polecenie !! spowoduje wykonanie ostatniego uruchomionego polecenia, ale z innym poleceniem przed nim. Może to brzmieć dziwnie, ale jest *bardzo* przydatne w sytuacjach, w których przypadkowo uruchomiłeś polecenie wymagające uprawnień roota, ale nie poprzedziłeś go poleceniem sudo.

```
apt-get install nginx # fails with a permission error
sudo !!
# To jest polecenie, które uruchamia:
sudo apt-get install nginx
```

Po tym jak poprzednie polecenie nie powiedzie się z powodu braku uprawnień, wystarczy uruchomić sudo !!, które spowoduje ponowne uruchomienie tego polecenia z sudo na początku.

Uwaga

Ze względów bezpieczeństwa nie rób z tego automatycznego nawyku, lecz zawsze upewnij się, że wiesz, dlaczego polecenie potrzebuje więcej uprawnień, i zadaj sobie pytanie, czy ufasz mu na tyle, aby dać mu pozwolenie na robienie dosłownie wszystkiego w Twoim systemie. Nieostrożne i nadmierne używanie sudo może łatwo coś popsuć lub umożliwić atakującemu zdobycie punktu oparcia w systemie.

Przeskakiwanie na początek lub koniec bieżącej linii

Nierzadko zdarza się, że podczas edycji trzeba przeskoczyć do początku linii, na przykład aby poprawić pisownię polecenia lub dodać wymagany argument. W tym celu naciśnij kombinację klawiszy *Ctrl+A*.

Analogicznie, aby przeskoczyć z powrotem na koniec linii, użyj kombinacji *Ctrl+E*. Te dwa skróty przydadzą się dość często.

Podsumowanie

Praca w środowisku powłoki Linuksa wymaga dużo pisania. Nawet najmniejsze usprawnienia w zakresie szybkości i dokładności konstruowania i edytowania poleceń mogą spowodować, że podstawowe zadania nie będą trwać całe wieki, a Ty poczujesz się jak prawdziwy uniksowy magik.

Sztuczki, którymi podzieliliśmy się w tym rozdziale, to niektóre z najbardziej przydatnych i najczęściej używanych przez nas w codziennej pracy skrótów. Połączenie nowych umiejętności przeszukiwania historii poleceń ze skrótami edycji i modyfikacji poleceń będzie miało ogromny pozytywny wpływ na komfort, wydajność i szybkość pracy w wierszu poleceń.

Wprowadzenie

do plików

Rozdział 5

W systemie Linux wszystko jest — lub może być — reprezentowane jako plik. Pliki są zorganizowane w system plików, który jest po prostu hierarchią plików i katalogów (katalogi są specjalnym rodzajem plików). Gdy jesteś programistą, prawie wszystko, co robisz w Linuksie, wymaga wiedzy na temat plików: pisanie i kopiowanie kodu źródłowego, tworzenie obrazów Dockera, rejestrowanie dzienników aplikacji, konfigurowanie zależności itd.

W tym rozdziale omówimy szczegóły dotyczące plików w systemie Linux. Poznasz różnicę między plikami tekstowymi a plikami binarnymi, które są najpopularniejszymi typami zawartości plików, z jakimi będziesz pracować. Zanim zagłębimy się w praktyczne polecenia potrzebne do tworzenia, modyfikowania, przenoszenia i edytowania plików, pokażemy Ci, w jaki sposób są one w Linuksie rozplanowywane i organizowane w drzewo systemu plików. Następnie uzupełnimy tę podstawową wiedzę praktycznym wprowadzeniem do edycji plików, korzystając z najpopularniejszych dostępnych edytorów tekstu dla wiersza poleceń.

Jednak w tym rozdziale nie poprzestaniemy na podstawach. Pliki linuksowe to jeden z tematów, przy których opłaca się (czasami dosłownie) zanurzyć nieco głębiej w zaawansowane zagadnienia. W końcu praca z plikami jest jedną z podstawowych rzeczy, które będziesz robić w Linuksie jako programista: pisanie i czytanie kodu źródłowego i plików konfiguracyjnych, wyszukiwanie określonych zawartości plików, kopiowanie i przenoszenie plików dziennika itp. Im lepiej opanujesz te podstawy, tym bardziej kompetentny będziesz jako wszechstronny programista, który nie musi w kółko wyszukiwać w internecie informacji na temat podstawowych poleceń Linuksa i może uniknąć żenującego utknięcia w edytorze tekstu wiersza poleceń podczas rozwiązywania problemów w trakcie połączenia za pomocą Zooma z innymi programistami.

Najpierw omówimy wyszukiwanie plików w drzewie systemu plików i znajdowanie określonych treści lub wzorców w poszczególnych plikach. Następnie przedstawimy pliki specjalne i alternatywne systemy plików, z którymi prawdopodobnie się zetkniesz, oraz wyjaśnimy to, co musisz wiedzieć, aby skutecznie z nimi pracować.

W tym rozdziale omawiamy następujące zagadnienia:

- typy plików i ich przeznaczenie;
- najważniejsze rodzaje danych plików;
- system plików Linuksa i polecenia do pracy z nim;

- podstawy edycji plików;
- niektóre typowe problemy i sposoby ich unikania.

Ten rozdział zawiera omówienie wielu tematów będących jednymi z najważniejszych podstaw dla zdobywania pozostałych umiejętności związanych z Linuksem. Zanim przejdiesz dalej, upewnij się, że zrozumiałeś każdą część — po pierwszej lekturze nie musisz pamiętać wszystkiego, ale podczas pracy z tym rozdziałem spróbuj zdobyć jak najwięcej praktycznego doświadczenia we własnym środowisku Linuksa. Znajomość tych kwestii owocuje wymiernymi korzyściami podczas rozwiązywania problemów w świecie rzeczywistym lub odbywania rozmów kwalifikacyjnych w sprawie pracy.

Pliki w Linuksie — absolutne podstawy

Aby podzielić obszerny temat plików w systemie Linux na mniejsze fragmenty, omówimy kilka absolutnych podstaw, z którymi prawdopodobnie radzisz już sobie w sposób intuicyjny: zwykłe pliki tekstowe i pliki binarne. Przyjrzymy się również praktycznemu błędowi, który może wystąpić podczas przenoszenia plików Windowsa do systemu Unix lub odwrotnie.

Pliki tekstowe

Jedną z najprostszych form plików tekstowych, jakie napotkasz, jest zwykły plik tekstowy. Choć niegdyś były to pliki ASCII, teraz są zazwyczaj zakodowane w formacie UTF-8. Możesz natknąć się na inne kodowania plików, ale jest to rzadkie, ponieważ są one ogólnie uważane za przestarzałe.

Co to jest plik binarny?

Unix nie rozróżnia plików binarnych i tekstowych, tak jak robi to wiele innych systemów operacyjnych. Wszystkie pliki mogą być przesyłane strumieniowo przez potoki i edytowane, a ponadto można dodawać do nich zawartość. Plik to po prostu plik. Gdy plik zostanie ustawiony jako wykonywalny, Unix postara się go wykonać — może zrobić to z powodzeniem w przypadku formatu **ELF** (ang. *Executable and Linkable Format*), który obecnie jest chyba najczęściej używanym formatem wykonywalnym, albo wykonywanie może się nie powieść, na przykład podczas próby wykonania pliku obrazu lub audio.

Ten prosty mechanizm otwiera niesamowite możliwości. Na przykład zanim pliki wykonywalne zostaną zdekompresowane i zapisane z powrotem, mogą być przesyłane potokowo przez narzędzie do kompresji, a następnie przez tunel sieciowy (taki jak SSH) — wszystko w jednym poleceniu, bez żadnych plików tymczasowych.

Oznacza to jednak również, że należy zachować ostrożność, aby zapobiec sytuacji, w której losowe pliki, na przykład przesłane lub zmodyfikowane przez użytkowników strony internetowej (w tym pliki dziennika!), mogłyby zostać wykonane. Może to bowiem prowadzić do poważnych problemów związanych z bezpieczeństwem.

Znaki zakończenia linii

Podczas gdy pliki Uniksa, zwłaszcza pliki tekstowe, działają podobnie do plików w innych systemach operacyjnych, warto wspomnieć, że między innymi Windows (i DOS) używa innego znaku zakończenia linii, co może powodować błędy w wielu programach korzystających z tych plików tekstowych. Chociaż dotyczy to tylko plików, które zostały utworzone w jednego rodzaju systemie, a następnie skopiowane do innego rodzaju systemu (na przykład przenoszenie pliku z Linuksa do DOS-a), warto o tym wiedzieć.

Przyczyny różnych znaków zakończeń linii są historyczne, a wiele narzędzi (na przykład Git i różne edytory tekstu) automatycznie poradzi sobie z tymi różnicami. Jednak w rzadkich przypadkach może być konieczne ręczne przekonwertowanie plików. Służą do tego popularne polecenia, takie jak `dos2unix`, ale w większości uniksowych systemów operacyjnych muszą one być instalowane ręcznie.

Istnieją jednak pewne metody konwersji plików przy użyciu bardziej tradycyjnych narzędzi:

- użycie `sed`: `sed 's/^m$//' oryginalny_plik_dosowy > plik_uniksowy;`
- użycie `tr`: `tr -d '\r' < oryginalny_plik_dosowy > plik_uniksowy;`
- konwertowanie w miejscu przy użyciu `perl`: `perl -pi -e 's/\r\n/\n/g' oryginalny_plik.`

Skoro omówiliśmy już kluczowe podstawowe pojęcia, które musisz znać, aby zrozumieć pliki w systemach uniksowych, przyjrzyjmy się kontekstowi, w którym wszystkie te pliki faktycznie istnieją: systemowi plików Linuksa.

Drzewo systemu plików

Standard hierarchii systemu plików (ang. *Filesystem Hierarchy Standard* — FHS) opisuje konwencjonalny układ katalogów systemów uniksopodobnych. Linux jest zgodny z tym standardem, co zasadniczo czyni go „oficjalną strukturą folderów Linuksa”. FHS jest znormalizowaną strukturą drzewa, w której każdy plik i katalog wywodzi się z katalogu głównego (nazwanego po prostu „/”). Ta hierarchia ma kluczowe znaczenie: chociaż istnieje miejsce, w którym użytkownicy końcowi mogą szaleć z własną strukturą katalogów, każdy podkatalog w / (katalogu głównym) ma określony cel.

Poznanie podstawowego układu tej hierarchii systemu plików nie zajmuje dużo czasu, a jeśli poświęcisz kilka minut, rozwiniesz umiejętność rozpoznawania, gdzie przechowywane są różnego rodzaju elementy, takie jak pliki binarne aplikacji, dzienniki, pliki danych czy urządzenia zewnętrzne, do których Twój kod potrzebuje dostępu. Innymi słowy, ułatwia to zarówno rozwój oprogramowania, jak i rozwiązywanie problemów, ponieważ kiedy wiesz, gdzie *powinny* znajdować się poszczególne elementy, mniej czasu tracisz na szukanie różnych rzeczy podczas analizowania incydentu. Ponadto ta wiedza jest wymagana, jeśli chodzi o pisanie własnych skryptów i wykonywanie lekkich zadań administracyjnych, których oczekuje się od starszego programisty.

Oto niektóre z ważnych miejsc w systemie plików, do których odwołania będziesz często napotykać lub których będziesz musiał używać samodzielnie:

- `/etc` — znajdują się tutaj pliki konfiguracyjne systemu i oprogramowania, zorganizowane w wiele podkatalogów;
- `/bin` i `/sbin` — znajdują się tutaj systemowe pliki binarne (nie grzeb w nich!);
- `/usr/bin` i `/usr/local/bin` — znajdują się tutaj zainstalowane oprogramowanie i własne pliki binarne, aby każdy w systemie mógł je zobaczyć i wykonać;
- `/var/log` i `/var/lib` — `/var` zawiera zmienne dane, czyli elementy, które są podatne na zmiany podczas pracy systemu, takie jak dzienniki aplikacji (`/var/log`) i biblioteki dynamiczne (`/var/lib`), pliki oraz inne stany uruchamianych aplikacji;
- `/var/lib/systemd` — jedno z kilku w systemie plików miejsc, które zawierają konfigurację `systemd`;
- `/etc/systemd/system` — dobre miejsce na niestandardowe systemowe pliki jednostkowe, jeśli tworzysz usługi;
- `/dev` — specjalny system plików używany do reprezentowania urządzeń sprzętowych;
- `/proc` — specjalny system plików używany do kwerendowania lub zmieniania stanu systemu.

Podstawowe operacje systemu plików

Pora zgłębić podstawowe polecenia Uniksa, których jako programista będziesz używać każdego dnia. Ten zestaw poleceń pozwoli Ci wykonywać wiele podstawowych zadań wiersza poleceń niezbędnych w dowolnym systemie, z którym wchodzisz w interakcję. Gdy nauczysz się tych poleceń i przećwiczysz ich stosowanie w tym rozdziale, będziesz mógł wykonywać kilka istotnych czynności. Oto one:

- śledzenie dzienników aplikacji w czasie rzeczywistym;
- naprawianie uszkodzonych plików konfiguracyjnych, aby przywrócić działanie aplikacji;
- przechodzenie z jednego katalogu do drugiego w repozytorium Gita na lokalnym komputerze programistycznym z systemem macOS.

Zacznijmy od wyświetlania zawartości katalogu. Upewnij się, że jesteś zalogowany do systemu Linux lub Unix (może być Ubuntu lub macOS) i masz otwartą aplikację Terminal, gotową do pracy z przykładami.

ls

Wyświetla listę plików lub katalogów. To polecenie jest podobne do „otwierania folderu” w graficznym interfejsie użytkownika. Wyświetla zawartość podanego katalogu. Domyślnie używa bieżącego katalogu:

```
/home/steve# ls
my_document.txt
```

W tym przykładzie bieżącą lokalizacją powłoki jest katalog `/home/steve`, który zawiera jeden plik (`my_document.txt`).

Możesz jednak poprosić `ls` o wyświetlenie dowolnej ścieżki katalogu w systemie, przekazując ten katalog jako argument:

```
/home/steve# ls /var/log/
alternatives.log apt bootstrap.log btmp dpkg.log faillog lastlog
utmp
```

Aby uzyskać bardziej uporządkowane dane wyjściowe, możesz dodać opcję `-l`. Zapewni ona „długą listę”, co oznacza jeden plik lub katalog na wiersz, wraz z dodatkowymi informacjami:

```
# ls -l /var/log/
total 296
-rw-r--r-- 1 root root 4686 Jun 24 02:31 alternatives.log
drwxr-xr-x 2 root root 4096 Jun 24 02:31 apt
-rw-r--r-- 1 root root 64547 Jun 24 02:06 bootstrap.log
-rw-rw---- 1 root utmp 0 Jun 24 02:06 btmp
-rw-r--r-- 1 root root 177139 Jun 24 02:31 dpkg.log
-rw-r--r-- 1 root root 32032 Oct 28 14:26 faillog
-rw-rw-r-- 1 root utmp 296296 Oct 28 14:26 lastlog
-rw-rw-r-- 1 root utmp 0 Jun 24 02:06 utmp
```

Krótko mówiąc, polecenie `ls` to sposób na „rozglądanie się” po systemie plików Uniksa.

pwd

Jest to skrót od angielskiego określenia *print working directory*, które oznacza wypisanie katalogu roboczego. Pokazuje, w którym miejscu systemu plików się znajdujesz w kontekście bieżącej sesji powłoki. Jeśli jesteś zalogowany w systemie Linux jako użytkownik Steve i znajdujesz się w swoim katalogu domowym, możesz się spodziewać, że `pwd` wypisze coś takiego:

```
pwd
/home/steve
```

cd

Zmienia bieżący katalog roboczy w powłoce. Polecenia uruchamiane po użyciu tego polecenia będą wykonywane z perspektywy nowej, zmienionej lokalizacji w systemie plików.

Oto przykładowy katalog:

```
Desktop
|—— anotherfile
|—— documents
```

```
|   └── contract.txt
|   └── somefile.txt
|   └── stuff
|       ├── nothing
|       └── important
```

Jeśli będziesz znajdować się w katalogu *Desktop*, ale potem przejdziesz do katalogu *documents* za pomocą polecenia `cd documents`, to gdy użyjesz polecenia `ls`, otrzymasz inną listę z tej nowej perspektywy. Zobaczmy to w działaniu:

```
/home/steve/Desktop# ls
anotherfile documents somefile.txt stuff
/home/steve/Desktop# cd documents/
/home/steve/Desktop/documents# ls
contract.txt
```

Teraz, gdy możemy rozejrzeć się po otoczeniu (`ls`), poruszać się po systemie plików (`cd`) i dowiedzieć się, gdzie jesteśmy (`pwd`), przejdźmy do rzeczywistego wpływu na system plików przez tworzenie i modyfikowanie plików.

touch

Tę operację zapisujemy jako `touch filepath`.

W zależności od tego, czy podana ścieżka plików już istnieje, polecenie to wykona jedną z dwóch rzeczy:

1. Jeśli podany plik nie istnieje jeszcze w tej ścieżce, `touch` go utworzy:

```
→ /tmp touch filepath
→ /tmp ls -l filepath
-rw-r--r-- 1 dcohen wheel 0 Aug 7 16:02 filepath
```
2. Jeśli podany plik *istnieje* w tej ścieżce, `touch` zaktualizuje czasy dostępu i modyfikacji dla tego pliku:

```
→ /tmp touch filepath
→ /tmp ls -l filepath
-rw-r--r-- 1 dcohen wheel 0 Aug 7 16:03 filepath
```

Zwróć uwagę, że jedyną rzeczą, która się zmieniła, jest czas modyfikacji pokazany na długiej liście.

less

Polecenie `less` to tzw. pager — program, który umożliwia przeglądanie zawartości pliku po jednym ekranie (stronie; ang. *page*):

```
less /etc/hosts
```

Jest to polecenie interaktywne — gdy użyjesz go do przeglądania pliku, będziesz mógł wykonywać następujące czynności:

- przewijanie w górę lub w dół linia po linii za pomocą kółka myszy lub klawiszy strzałek;
- przewijanie w dół o całą stronę za pomocą **SPACJI**;
- wyszukiwanie za pomocą / (wprowadź wzór wyszukiwania) **RETURN**;
- przechodzenie do następnego dopasowania za pomocą przycisku **n**;
- używanie przycisku **q** do zamykania programu.

Poćwicz korzystanie z tego polecenia przez kilka minut, a szybko je opanujesz.

tail

Polecenie **tail** służy do wyświetlania kilku ostatnich linii pliku:

```
tail /jakiś/plik
```

Opcja **-f** (ang. *follow*) dla polecenia **tail** jest bardzo przydatna w przypadku przesyłania dzienników na żywo do terminala:

```
tail -f /var/log/some.log
```

Użyj przycisku **q**, aby zakończyć działanie polecenia **tail**.

mv

Polecenie **mv** (ang. *move*) służy do przenoszenia plików i zmieniania ich nazwy.

Przenoszenie plików

Wyobraź sobie, że masz plik o nazwie *somefile.txt*:

```
→ Desktop ls -alh somefile.txt  
-rw-r--r-- 1 dcohen wheel 0B Aug 7 11:02 somefile.txt
```

O ile znajdujesz się w tym samym katalogu co dany plik, możesz przenieść go do katalogu */var/log* bez zmiany nazwy w ten sposób:

```
mv somefile.txt /var/log/
```

Zmienianie nazw plików

Teraz chcesz zmienić nazwę tego pliku na *foobar*:

```
mv /var/log/somefile /var/log/foobar
```

To wszystko!

cp

Do kopiowania plików i katalogów należy zastosować polecenie `cp`.

Polecenie `cp plik miejsce_docelowe` powoduje skopiowanie pliku o nazwie *plik* do ścieżki docelowej określonej przez *miejsce_docelowe*. Najczęściej używaną opcją jest `-r` lub `-recursive` — jeśli kopiujesz katalog, spowoduje ona skopiowanie również wszystkiego, co znajduje się w środku.

```
cp -r /home/dave /storage/userbackups/
```

mkdir

To polecenie tworzy nowy, pusty katalog o nazwie *nazwa_katalogu*:

```
mkdir nazwa_katalogu
```

Przydatną opcją jest `-p`, która umożliwia tworzenie zagnieżdżonych katalogów w jednym poleceniu. Na przykład jeśli chcesz utworzyć katalog *Documents* zawierający katalog *school*, który z kolei zawiera katalog *reports*, możesz uruchomić następujące polecenie:

```
mkdir -p Documents/school/reports
```

rm

Polecenie `rm` usuwa (kasuje) pliki i katalogi:

- `rm nazwa_pliku` usuwa plik o nazwie *nazwa_pliku*;
- `rm -r nazwa_katalogu` usuwa katalog o nazwie *nazwa_katalogu* oraz rekurencyjnie każdy znajdujący się w nim plik i katalog.

Istnieje osobne polecenie usuwania pustych katalogów, `rmdir`, ale zwykle jest używane tylko w skryptach, w których programiści starają się ograniczyć potencjalne szkody wynikające z niezamierzonego usunięcia.

Edycja plików

Niezależnie od tego, czy chodzi o aktualizację plików konfiguracyjnych, tworzenie nowych usług Linuksa czy robienie notatek podczas sesji rozwiązywania problemów, praca w systemie Linux czasami wymaga edycji plików w wierszu poleceń. Edycję plików w wierszu poleceń omówimy szczegółowo w rozdziale 6. „Edycja plików w wierszu poleceń”, ale tutaj przedstawimy bardzo krótki opis tego zagadnienia.

Jeśli jesteś ograniczony do środowiska zawierającego tylko wiersz poleceń, możesz użyć kilku edytorów tekstu wiersza poleceń:

- **nano** — prawie zawsze zainstalowany lub dostępny; łatwy w użyciu.
- **vi** — zainstalowany prawie wszędzie; trzeba się do niego przyzwyczaić.
- **vim** — łatwy do zainstalowania wszędzie; bardziej funkcjonalny niż `vi`.

Jeśli żaden z nich nie jest zainstalowany, możesz je zainstalować za pośrednictwem menedżera pakietów. Jeżeli używasz na przykład Linuksa Ubuntu, będzie to polecenie takie jak `sudo apt-get install nano` (lub `vim` zamiast `nano`). Polecenia zarządzania pakietami omówimy szerzej w rozdziale 9. „Zarządzanie zainstalowanym oprogramowaniem”. Niezależnie od tego, który edytor wybierzesz, plik możesz edytować przez wpisanie w wierszu poleceń [`$EDYTOR nazwa_pliku`], na przykład:

```
vi nazwa_pliku
vim /jakiś/plik
nano /kolejny/plik
```

- Jeśli plik istnieje, będziesz mógł go edytować w swoim edytorze.
- Jeśli plik nie istnieje, ale istnieje katalog, utworzysz nowy plik w tej ścieżce przy pierwszym zapisaniu go w edytorze.
- Jeśli katalog nie istnieje, możesz edytować plik, ale edytor nie będzie w stanie zapisać go w systemie plików bez kilku dodatkowych kroków.

Praktyczne umiejętności związane z edycją plików w wierszu poleceń Linuksa opiszemy szczegółowo w rozdziale 6. „Edycja plików w wierszu poleceń”. Jeżeli przed zakończeniem tego rozdziału będziesz musiał edytować jakiś plik, wpisz po prostu `nano /ścieżka/do/pliku/` i postępuj zgodnie z wyświetlanymi na ekranie podpowiedziami, aby zapisać zmiany i zamknąć edytor. W międzyczasie przyjrzymy się różnym rodzajom plików, które jako programista napotkasz w systemie Linux.

Typy plików

Omówiliśmy już „standardowe” pliki, takie jak zwykłe pliki tekstowe oraz dane binarne w plikach obrazów i programach wykonywalnych. Istnieje jednak jeszcze kilka innych typów plików i musisz wiedzieć, jak je rozpoznawać i pracować z nimi w systemie Linux. Niezależnie od tego, czy szukasz pamięci USB lub klawiatury podłączonych właśnie do komputera, tworzysz łącze wskazujące na plik czy sprawdzasz gniazda sieciowe, które otworzył proces internetowy, warto dowiedzieć się trochę o tym *wszystkim*.

Oto wszystkie typy plików linuksowych i ich przeznaczenie:

- **Zwykły plik** — jest to najpopularniejszy typ pliku, zawierający dane tekstowe lub binarne. Jako inżynier oprogramowania będziesz napotykać zwykłe pliki w prawie każdym zadaniu programistycznym, niezależnie od tego, czy będziesz pisać kod, edytować pliki konfiguracyjne czy wykonywać programy. Typowym przykładem, który można zobaczyć w danych wyjściowych długiej listy, może być plik kodu źródłowego, taki jak ten:

```
-rw-r--r-- 1 dave dave 210 Jan 04 09:30 main.c
```

- **Katalog** — katalogi to specjalne pliki, które służą do organizowania innych plików i katalogów. Jeśli kiedykolwiek korzystałeś z systemu Windows lub macOS, jesteś już zaznajomiony z katalogami (nazywanymi w tych systemach folderami). Zawierają one inne pliki i katalogi. Na długiej liście katalog taki jak `/etc` będzie wyglądał następująco:

```
drwxr-xr-x 5 root root 4096 Jan 04 09:21 /etc
```

- **Plik specjalny blokowy** — ten specjalny typ pliku zapewnia buforowany dostęp do urządzeń sprzętowych, co czyni go szczególnie przydatnym w przypadku urządzeń takich jak dyski twarde, gdzie dostęp do danych jest uzyskiwany w dużych blokach o stałym rozmiarze. Rzadko będziesz z nimi pracować bezpośrednio, z wyjątkiem przypadków montowania systemów plików. Przykładem może być partycja dysku twardego, wyświetlana na liście w następujący sposób:

```
brw-rw---- 1 root disk 8, 2 Jan 19 11:00 sda2
```

Reprezentuje to urządzenie blokowe z uprawnieniami do odczytu i zapisu dla właściciela i grupy.

- **Plik specjalny znakowy** — podobnie jak pliki blokowe pliki znakowe zapewniają niebuforowany, surowy dostęp do urządzeń sprzętowych, ale są przeznaczone dla urządzeń, w których dane nie opierają się na blokach, takich jak klawiatury lub myszy. Nigdy nie będziesz musiał się nimi przejmować, chociaż czasami możesz ich używać w trakcie pracy (na przykład `/dev/urandom`, `/dev/null` lub `/dev/zero`). Urządzenie znakowe, takie jak terminal, może pojawić się na długiej liście w taki sposób:

```
crw-rw-rw- 1 root tty 5, 1 Jan 19 22:00 /dev/tty1
```

- **Plik specjalny FIFO („potok nazwany”)** — potoki nazwane (nie należy mylić ich z często używanymi w powłokach potokami anonimowymi) są wykorzystywane do komunikacji międzyprocesowej. Prawie nigdy nie będziesz musiał mieć z nimi do czynienia, chociaż będziesz używać ich anonimowych kuzynów, aby zostać uniksowym magikiem, o czym więcej przeczytasz w rozdziale 11. „Potoki i przekierowanie”. Nieczęsto będziesz się z nimi spotykać, ale jednym z przykładów jest plik potoku nazwanego, który może wyglądać następująco:

```
prw-r--r-- 1 user user 0 Jan 21 10:00 mynamedpipe
```

- **Linki (dowiązania)** — są rodzajem skrótu do innego pliku. Istnieją dwa rodzaje linków, a mianowicie dowiązania twarde i symboliczne (miękkie). Prawie nigdy nie będziesz musiał zajmować się dowiązaniami twardymi, ale możesz korzystać z dowiązań symbolicznych, aby tworzyć wygodne ścieżki do często używanych plików lub sprawiać, by do tego samego pliku prowadziło wiele ścieżek. Omówimy je dalej w tym rozdziale. Dowiązanie symboliczne może być wyświetlane tak:

```
lrwxrwxrwx 1 user user 7 Jan 21 10:30 versions/latest -> bin/app-3.1
```

Ten przykład pokazuje dowiązanie o nazwie `latest`, które wskazuje na plik o nazwie `app-3.1`.

- **Gniazda** — gniazda uniksowe są używane do IPC, podobnie jak pliki potoków. Pliki gniazd możesz napotkać podczas rozwiązywania problemów z usługami, które muszą się ze sobą komunikować („Dlaczego nginx nie może połączyć się z mojego serwera aplikacji?”). Plik gniazda, w tym przypadku gniazda używane przez nginx i php-fpm do komunikacji w celu uruchomienia aplikacji WordPress, może wyglądać następująco:

```
srwxrwx--- 1 root socket 0 Jan 23 11:31 /run/wordpress.sock
```

Ta lista obejmuje dodatkowe, specjalne typy plików, które możesz napotkać, i daje Ci pewne wyobrażenie o tym, w jakich sytuacjach (i dlaczego) możesz się z nimi zetknąć w swojej pracy. Aby pomóc Ci zbudować przydatne umiejętności praktyczne, przyjrzymy się dokładnie niektórym z tych plików. Zaczniemy od zdobycia prawdziwych doświadczeń z najpopularniejszymi spośród tych specjalnych typów plików: dowiązaniami.

Dowiązania symboliczne

Dowiązania symboliczne, często nazywane symlinkami lub dowiązaniami miękkimi, to typ pliku, który służący jako odwołanie do innego pliku lub katalogu. W przeciwieństwie do dowiązania twardego dowiązanie symboliczne może wskazywać plik lub katalog w różnych systemach plików i zachowuje oddzielny i-węzeł od pliku lub katalogu, do którego się odwołuje.

Dowiązanie symboliczne możesz utworzyć przy użyciu tej podstawowej składni:

```
ln -s document.txt /ścieżka/do/utworzenia/dowiązania
```

`ln` (pisane od małej litery) to polecenie „link”.

Na przykład jeśli masz w bieżącym katalogu plik o nazwie *file1.txt* i chcesz utworzyć do niego dowiązanie symboliczne o nazwie `link1`, użyj takiego polecenia:

```
ln -s file1.txt link1
```

Gdy wyświetlisz teraz długą listę swojego katalogu za pomocą polecenia `ls -l`, zobaczysz `link1` wymieniony jako dowiązanie dla pliku *file1.txt*:

```
ls -l
total 0
-rw-r--r-- 1 root root 0 Oct 28 16:08 file1.txt
lrwxrwxrwx 1 root root 9 Oct 29 17:20 link1 -> file1.txt
```

Przy próbie uzyskania dostępu do `link1`, na przykład podczas wypisywania zawartości pliku za pomocą polecenia `cat link1`, system automatycznie wyłuska to dowiązanie i podaje zawartość pliku *file1.txt*. Jeśli plik *file1.txt* zostanie przeniesiony, usunięty lub zmieniony, dowiązanie symboliczne nie zaktualizuje się automatycznie i będzie wskazywać na nieistniejący plik (uszkodzone dowiązanie).

Dowiązania symboliczne są szczególnie przydatne do tworzenia skrótów, porządkowania plików i katalogów oraz utrzymywania elastycznych i logicznych struktur systemu plików.

Dowiązania twarde

Dowiązanie twarde to dodatkowa nazwa pliku istniejącego w tym samym systemie plików, w efekcie działająca jako alias. Zarówno oryginalny plik, jak i dowiązanie twarde mają ten sam i-węzeł, co oznacza, że zmiany w jednym są odzwierciedlane w drugim. W przeciwieństwie do dowiązań symbolicznych dowiązania twarde nie mogą przekraczać granic systemu plików ani łączyć się z katalogami. Jeżeli oryginalny plik zostanie

usunięty, dowiązanie twarde nadal będzie przechowywać dane. Aby utworzyć dowiązanie twarde o nazwie `link1` do pliku *file1.txt*, należy użyć następującego polecenia:

```
ln file1.txt link1.
```

Polecenie file

Polecenie `file` jest narzędziem umożliwiającym sprawdzenie typu pliku. Podstawowe użycie tego polecenia jest proste: wpisz **file**, a następnie podaj nazwę pliku, na przykład:

```
file mysecret.txt
```

Dane wyjściowe z tego polecenia mogą wyglądać tak: `myscret.txt: ASCII text`, co będzie wskazywać, że *myscret.txt* jest zwykłym plikiem tekstowym.

Jeśli masz plik binarny, taki jak skompilowany program o nazwie `mybinary`, uruchomienie `file mybinary` może wygenerować następujące dane wyjściowe: `mybinary: ELF 64-bit LSB executable`, które będą wskazywać, że ten program jest binarnym plikiem wykonywalnym.

W przypadku katalogu, takiego jak `/home/user`, uruchomienie polecenia `file /home/user` prawdopodobnie zwróci `/home/user: directory`, wskazując, że `/home/user` jest katalogiem.

Polecenie `file` jest potężnym narzędziem, umożliwiającym szybkie rozpoznawanie typów plików, z którymi pracujesz, zwłaszcza w przypadku nieznanych typów.

Jeśli masz ochotę poćwiczyć, użyj polecenia `file`, aby sprawdzić następujące pliki:

- `file /bin/sh`,
- `file /dev/zero`,
- `file /dev/urandom`,
- `file /dev/sda1`,
- `file ~/.bashrc`,
- `file /bin/ls`,
- `file /home`,
- `file /proc/1/cwd`.

Zaawansowane operacje na plikach

Podczas pracy z plikami w uniksowych systemach operacyjnych często trzeba wykonywać operacje na nich, z nimi lub z ich zawartością, ale bez bezpośredniego modyfikowania ich w edytorze. Oto przykładowe operacje, które możesz wykonywać:

- przeszukiwanie pliku, aby sprawdzić, czy zawiera on daną treść;
- identyfikowanie partii plików, które zostały zmodyfikowane w określonym czasie;
- bezpieczne przenoszenie pliku do innego systemu zamiast kopiowania go za pomocą polecenia `mv` na komputerze lokalnym.

Możesz nawet połączyć wszystkie te trzy operacje w jedną! Ta wiedza może się naprawdę przydać podczas rozwiązywania problemów (wyszukiwanie konkretnego identyfikatora żądania lub kodu błędu w dzienniku), tworzenia oprogramowania (znajdowanie niedawno zmodyfikowanych plików kodu źródłowego) lub przeprowadzania testów (kopiowanie zaktualizowanego kodu źródłowego aplikacji do systemu testowego).

Oto krótki przegląd tego rodzaju operacji na plikach, który powinien dać Ci wyobrażenie o narzędziach i poleceniach, których będziesz używać do ich wykonywania.

Wyszukiwanie zawartości pliku za pomocą narzędzia grep

Dopasowywanie tekstu odbywa się tradycyjnie za pomocą narzędzia grep. Na swoim osobistym lub służbowym laptopie możesz zainstalować `ag` lub `rg` (np. `sudo apt-get install silversearcher-ag`), które są bardziej przyjaznymi dla programistów i szybszymi wersjami tej koncepcji, ale w systemach produkcyjnych zawsze będziesz mieć do dyspozycji `grep`.

Oto składnia wyszukiwania wzorca `wzorzec_wyszukiwania` w pliku `ścieżka/do/pliku`:

```
grep "wzorzec_wyszukiwania" ścieżka/do/pliku
```

Możesz oczywiście wyszukiwać w ten sposób literały łańcuchów znaków, ale `grep` jest tak wszechstronnym narzędziem dlatego, że do wyszukiwania wzorców pozwala używać wyrażeń regularnych (ang. *regex*). Poniższe polecenie zwróci linie zaczynające się od `startswith`:

```
grep ^startswith /jakiś/plik
```

To polecenie zwróci linie, które kończą się na `endswith`:

```
grep endswith$ /jakiś/plik
```

Wyrażenia regularne są niezwykle przydatne, a każdy programista i użytkownik Linuksa powinien znać ich podstawy.

Narzędzie `grep` możesz stosować także do wyszukiwania rekurencyjnego w katalogu, czyli przeszukiwania wszystkich plików we wszystkich katalogach, które zawiera dany katalog:

```
root@c7f1417df8d2:/tmp# grep -r -i "hello world" /tmp
/tmp/secret/dontlook.key:hello world
/tmp/hi.txt:hello world
/tmp/hi.txt:HeLlO WoRlD! You found me!
```

A co, jeśli nie chcemy wyszukiwać łańcuchów znaków w pliku, lecz chcemy tylko znaleźć określone pliki?

Wyszukiwanie plików za pomocą narzędzia find

Narzędzie `find` pozwala wyszukiwać pliki i katalogi według nazwy, czasu modyfikacji lub innych atrybutów. Zasadniczo jest to wyszukiwanie wszerz drzewa systemu plików i jest bardzo przydatne w określonych przypadkach. Oto kilka przykładów:

- wyszukiwanie wszystkich plików dziennika aplikacji, które zostały utworzone lub zmodyfikowane w ostatnim dniu;
- identyfikowanie wszystkich plików testowych kodu źródłowego z nazwami kończącymi się na `_test.go`;
- lokalizowanie w celu usunięcia wszystkich plików `php.ini` pozostawionych przez programistę odbywającego staż.

W poniższych przykładach `/ścieżka/wyszukiwania` jest częścią systemu plików, którą chcemy przeszukać. Jeśli chcesz przeszukać bieżący katalog i wszystkie jego podkatalogi, możesz użyć znaku kropki (`.`), na przykład `find . -name 'file.txt'`:

- Wyszukiwanie plików według rozszerzenia:
`find /ścieżka/wyszukiwania -name '*.ext'`
- Wyszukiwanie plików pasujących do wielu wzorców ścieżki lub nazwy:
`find /ścieżka/wyszukiwania -path '**/ścieżka/**/*.*ext' -or -name '*wzorzec*'`
- Wyszukiwanie katalogów o danej nazwie w trybie bez rozróżniania wielkości znaków:
`find /ścieżka/wyszukiwania -type d -iname '*lib*'`
- Wyszukiwanie plików pasujących do danego wzorca z wyłączeniem określonych ścieżek:
`find /ścieżka/wyszukiwania -name '*.py' -not -path '*/site-packages/*'`
- Wyszukiwanie plików z danego przedziału rozmiarów:
`find /ścieżka/wyszukiwania -size +500k -size -10M`

Kopiowanie plików między hostami lokalnymi i zdalnymi za pomocą narzędzia rsync

`rsync` to niezwykle przydatne narzędzie, które kopiuje pliki i katalogi między hostami. Działa podobnie jak `cp`, ale dodatkowo umożliwia kopiowanie między hostami, z których jeden lub oba są zdalne.

Narzędzie `rsync` jest zasadniczo kombinacją `cp` (do kopiowania danych) i `ssh` (do bezpiecznego, zaszyfrowanego transportu). Jeśli nie znasz `ssh`, musisz dowiedzieć się, jak działa to narzędzie (i skonfigurować własne klucze i dostęp SSH), zanim wypróbujesz polecenia `rsync`.

Oto kilka przykładowych wywołań, które zamieszczamy dzięki uprzejmości twórców projektu `tl;dr`:

- Przenoszenie pliku z hosta lokalnego na zdalny:
`rsync ścieżka/do/pliku_lokalnego host_zdalny:ścieżka/do/katalogu_zdalnego`
- Przenoszenie pliku z hosta zdalnego na lokalny:
`rsync host_zdalny:ścieżka/do/zdalnego_pliku ścieżka/do/katalogu_lokalnego`
- Przenoszenie pliku w trybie zarchiwizowanym (ang. *archive*; w celu zachowania atrybutów) i skompresowanym (ang. *zipped*) ze szczegółowymi

(ang. *verbose*) i czytelnymi dla człowieka (ang. *human-readable*) danymi wyjściowymi z wyświetlaniem postępu (ang. *Progress*) — od pierwszych liter angielskich określeń pochodzą nazwy atrybutów `-azvhP`:

```
rsync -azvhP ścieżka/do/katalogu_lokalnego  
host_zdalny:ścieżka/do/katalogu_zdalnego
```

Polecenia z ostatniego przykładu jeden z autorów książki używał setki razy do tworzenia szybkich, zautomatyzowanych kopii zapasowych.

Łączenie poleceń `find`, `grep` i `rsync`

Łączeniu poleceń za pomocą znaku potoku (`|`) przyjrzymy się szczegółowo w rozdziale 11. „Potoki i przekierowanie”.

Jeśli chcesz połączyć przykłady z poprzedniego podpunktu, aby wykonać na przykład kopię zapasową wszystkich plików z katalogu `/tmp`, które zostały zmodyfikowane w ostatnim tygodniu, wystarczy jedno sprytnie polecenie:

```
find /tmp -type f -mtime -7 -exec grep -l "hello world" {} \; | xargs -I  
_ backupscript.sh _ backup@backupserver.local:/backups_
```

Najpierw uruchamiamy `find`, szukając plików, których czas ostatniej modyfikacji jest krótszy niż 7 dni. Używamy flagi `-exec` polecenia `find` w celu wykonania polecenia `grep` z flagą `-l`, która zwraca nazwę pasującego pliku. Następnie wprowadzamy nazwy plików do polecenia `xargs`, które wykonuje określoną operację dla każdej linii danych wejściowych otrzymanych z poprzedniego polecenia. W tym przypadku ta operacja polega na uruchomieniu wymyślnego skryptu tworzenia kopii zapasowej dla każdego pasującego pliku w wymyślonej ścieżce docelowej, w której ktoś mógłby chcieć utworzyć kopię zapasową danego pliku.

Przy założeniu, że mamy te same pliki co wcześniej, przy omawianiu polecenia `grep`, to dziwnie wyglądające polecenie uruchomi za nas dwa polecenia:

```
backupscript.sh /tmp/secret/dontlook.key  
backup@backupserver.local:/backups/tmp/secret/dontlook.key  
backupscript.sh /tmp/hi.txt backup@backupserver.local:/backups/tmp/hi.txt
```

Robi dokładnie to, co chcieliśmy: uruchamia skrypt kopii zapasowej *tylko* dla dwóch plików zawierających łańcuch znaków *hello world*, na którym nam zależy, i zmodyfikowanych w ciągu ostatnich 7 dni.

Chociaż złożenie takiego polecenia z pewnością może zająć kilka minut (i wymagać trochę wyszukiwania w internecie), na dłuższą metę polecenie to może zaoszczędzić Ci wiele godzin. Taka jest potęga środowiska wiersza poleceń w połączeniu z małymi, skoncentrowanymi narzędziami uniksowymi, które możesz dowolnie łączyć.

Więcej informacji o uniksowych potokach i narzędziu `xargs` znajdziesz w rozdziale 11. „Potoki i przekierowanie”, ale zaprezentowaliśmy ten przykład, aby dać Ci wyobrażenie o tym, jak będziesz łączyć wszystkie te proste polecenia podczas ich poznawania.

Zaawansowane zagadnienia systemu plików

Wprowadziliśmy już różne typy plików Linuksa i zdobyłeś pewne doświadczenie w pracy z najpopularniejszymi typami plików. Omówimy teraz mniej popularne zagadnienia z zakresu systemu plików, których znajomość przyda Ci się podczas pracy w systemach Linux.

Oto przykładowe sytuacje, w których wykorzystasz tę wiedzę:

- rozwiązywanie problemów z pierwszą aplikacją Dockera, która ma zamontowane woluminy pamięci masowej;
- praca nad aplikacją, która komunikuje się z kontrolerami przemysłowymi, kamerami lub innym zewnętrznym sprzętem;
- pisanie kodu aplikacji, który wymaga dostępu do funkcji losowości w celu bezpiecznego generowania haseł lub tokenów API; jednym ze specjalnych typów plików, z którymi będziesz się stykać, są **urządzenia blokowe** — przypominają one pewną formę dysku, gdzie dane są pobierane i odczytywane w blokach.

Klasyczne urządzenia dyskowe są urządzeniami blokowymi i zazwyczaj znajdziesz je dołączone do systemu plików w następujących lokalizacjach:

- `/dev/hdX`,
- `/dev/sdX`,
- `/dev/nvmeN`.

X i *N* są tutaj indeksami alfabetycznymi lub liczbowymi odpowiednich dysków, np. `/dev/sda` lub `/dev/nvme0`. **Partycje** wyglądają jak dyski, ale z dołączoną dodatkową cyfrą lub znakiem, tak jak `/dev/sda0`, czyli pierwsza partycja na pierwszym dysku.

Zwróć uwagę, że nawet gdy system operacyjny wykryje nowy dysk twardy i dołączy go (i wszelkie wykryte partycje) w jednej z tych lokalizacji, nadal będziesz musiał „zamontować” system plików znajdujący się na dysku za pomocą polecenia `mount`. Nie jest to szczególnie powszechna czynność dla programistów, więc nie będziemy szerzej jej omawiać.

Istnieją również specjalne **urządzenia programowe**. Obejmują one zakres od `/dev/null` (mogłeś widzieć przekazywanie do niego danych wyjściowych w postaci *jakieś polecenie* > `/dev/null`) do `/dev/random` i `/dev/urandom`, które dostarczają losowe bajty. Stąd wybrany przez Ciebie język programowania najprawdopodobniej będzie pobierał swoje kryptograficznie zabezpieczone liczby losowe.

Kolejnym katalogiem jest `/proc`, czyli system plików spopularyzowany przez system operacyjny Plan 9, ale przewidziany we wczesnych czasach Uniksa. Jak sama nazwa wskazuje, został on utworzony, aby reprezentować procesy jako pliki. Katalog `/proc` zawiera katalogi noszące nazwy po identyfikatorach procesów; przechowują one pliki, które mogą być wykorzystywane do odczytywania stanu tych procesów. Szczególnie w systemie

Linux został on rozszerzony o różne inne interfejsy, w tym sposoby konfigurowania sterowników jądra, odczytu informacji sprzętowych i wyjść czujników, a nawet interakcji z BIOS-em i UEFI.

FUSE — jeszcze więcej zabawy z systemem plików Uniksa

Jak już pokazaliśmy, w Uniksie wiele rzeczy można interpretować jako pliki. Ta filozofia polega na tym, że edycja plików jest powszechna, więc polecenia i języki programowania zdolne do interakcji z plikami zapewniają łatwy do zrozumienia interfejs. **FUSE** (ang. *Filesystem in Userspace*), czyli system plików w przestrzeni użytkownika, to API, które umożliwia każdemu implementowanie nowych uniksowych systemów plików bez konieczności bycia programistą jądra. Innymi słowy, ponieważ wiele rzeczy może komunikować się z plikami, warto mieć możliwość „podrobienia” API plików Uniksa dla tych rzeczy, które nie są rodzajem normalnych, lokalnie przechowywanych danych, jakimi powinien być plik. Jeśli brzmi to trochę dziwnie, zapoznaj się z niektórymi projektami napisanymi przez użytkowników za pomocą systemu FUSE. FUSE jest wykorzystywany do implementacji wielu klasycznych sterowników systemu plików, takich jak NTFS, dzięki któremu można odczytać stare systemy plików Windowsa na komputerze z systemem Linux. Jednak ze względu na elastyczność i możliwość uzyskiwania dostępności do FUSE istnieją również dość dziwaczne systemy plików, które zostały zaimplementowane w ten sposób:

- sshfs umożliwia na przykład lokalne zamontowanie katalogu na innym komputerze dostępnym za pośrednictwem SSH;
- inne systemy plików FUSE umożliwiają montowanie zdalnej pamięci masowej w chmurze (takiej jak S3 firmy Amazon) jako katalogu lokalnego;
- niektóre, jeszcze mniej zrozumiałe systemy pozwalają zamontować Wikipedię jako katalog plików lub reprezentować jako systemy plików protokoły, np. IRC, i usługi, takie jak pogodowe API.

FUSE jest tak przydatny, że znalazł się w wielu uniksopodobnych systemach operacyjnych oprócz Linuksa i jest teraz dostępny nawet w systemie Windows. Warto o tym wiedzieć, nie tylko dlatego, że jest to nowatorskie wykorzystanie warstwy abstrakcji dla plików w Uniksie, ale też dlatego, że może to być niezwykle przydatne, gdy ma się do czynienia z informacjami przechowywanymi gdzieś bez klasycznego API, które dana aplikacja może wykorzystać w warstwie aplikacji. Prawdopodobnie każdy język programowania, którego będziesz używać, ma standardową bibliotekę pozwalającą komunikować się z plikami w systemie plików Uniksa, a FUSE to sposób na utworzenie tego interfejsu dla prawie każdego rodzaju informacji.

Podsumowanie

Ten rozdział był intensywną podróżą przez podstawy i kilka bardziej zaawansowanych zagadnień związanych z plikami i systemem plików w Linuksie. Wyjaśniliśmy różnicę między plikami tekstowymi a plikami binarnymi, pokazaliśmy, jak układa się drzewo systemu plików w Linuksie, i omówiliśmy wszystkie podstawowe polecenia potrzebne do pracy z plikami. Jeśli byłeś skrupulatny, poświęciłeś też trochę czasu we własnym środowisku Linuksa, ćwicząc ważne umiejętności edycji plików wiersza poleceń, które tutaj opisaliśmy.

Po omówieniu podstaw przeskoczyliśmy do najbardziej kluczowych tematów pośrednich i zaawansowanych, których znajomość będzie Ci potrzebna. Pokazaliśmy, jak znajdować pliki i przeszukiwać ich zawartość, oraz daliśmy Ci wyobrażenie na temat plików specjalnych i systemów plików.

Wszystko to razem wyposaża Cię w najważniejsze umiejętności i wiedzę, których potrzebujesz, aby używać Linuksa do rozwiązywania rzeczywistych problemów.

Edycja plików w wierszu poleceń

Rozdział 6

Edycja tekstu w wierszu poleceń jest często niezbędnym wymogiem ze względu na ograniczenia systemów produkcyjnych, które zwykle nie mają graficznego interfejsu użytkownika. Jednak pomijając kwestię systemów produkcyjnych, płynne edytowanie tekstu w wierszu poleceń i tak ma wiele zalet — nawet jeśli dysponujesz graficznymi edytorami tekstu lub **zintegrowanymi środowiskami programistycznymi** (ang. *Integrated Development Environment* — IDE).

Na przykład wiele w pełni funkcjonalnych edytorów tekstu i środowisk IDE obsługuje omówione w tym rozdziale wzorce, które umożliwiają przeniesienie uzyskanej szybkości i wydajności na inne narzędzia. Skrótów, które pokażemy w tym rozdziale, możesz używać do wszelkiego rodzaju zadań, od szybkiego wyszukiwania i zastępowania tekstu po poprawianie błędnie napisanego słowa w środku długiego polecenia powłoki.

Możesz nawet znaleźć podobne skróty wbudowane w Twoje ulubione narzędzia (czasami za pośrednictwem wtyczek) — wystarczy na przykład tylko kilka wyszukiwań w internecie, aby odkryć klienty e-mailowe, wtyczki przeglądarki i aplikacje internetowe obsługujące skróty klawiaturowe vim, którym przyjrzymy się dalej w tym rozdziale.

Nauka myślenia w kategoriach efektywnych wzorców, których wymagają te minimalistyczne interfejsy tekstowe, może pomóc Ci znaleźć bardziej wydajne sposoby na wykonywanie codziennych zadań, zapobiegając marnowaniu czasu na żmudne klikanie graficznych menu lub kreatorów, gdy możesz wykonać to samo zadanie za pomocą zaledwie kilku naciśnień przycisków klawiatury.

Uwaga

Wiele bardzo minimalistycznych (lub wysoce bezpiecznych) środowisk cechuje tendencja do usuwania edytorów tekstu z obrazów produkcyjnych, chociaż nie poprawia to bezpieczeństwa (aby w mgnieniu oka zaimprovizować działający edytor tekstu, wystarczy zaledwie narzędzia `cat`, `echo`, `mv` i przekierowywanie wejścia i wyjścia). Podobnie jak hakerzy na całym świecie zapewne w końcu sam zainstalujesz nano lub vim w licznych kontenerach Dockera.

Podczas lektury tego rozdziału poznasz podstawy dwóch edytorów tekstu: jednego, który naszym zdaniem jest najłatwiejszy do rozpoczęcia pracy (`nano`), i drugiego, który według nas jest najlepszą długoterminową inwestycją w rozwój kariery (`vim`). Przedstawimy podstawowy kontekst edycji tekstu w wierszu poleceń w Linuksie, przyjrzymy się

bliżej edytorom nano i vim, a na koniec omówimy, jak unikać najczęstszych błędów związanych z edycją tekstu. Pokażemy Ci również, jak dostosować powłokę, aby automatycznie używała preferowanego edytora.

Nano

Nano to niewielki i łatwy w użyciu edytor tekstu CLI. Jedną z jego cech — można nawet powiedzieć, że główną — jest to, iż ma on wyraźnie widoczną ściągawkę ze skrótami klawiaturowymi przypiętą u dołu ekranu, podczas gdy Ty beztrudnie edytujesz tekst w terminalu. Jest to szczególnie przydatne, jeśli pracujesz w stresie i nie jesteś przyzwyczajony do edytowania tekstu w wierszu poleceń.

Nano dobrze się sprawdza w nagłych przypadkach, ale nie jest zainstalowany w bardziej minimalistycznych środowiskach (takich jak kontenery Dockera lub produkcyjne maszyny wirtualne). Należy pamiętać, że nano ma również tendencję do automatycznego tworzenia kopii zapasowych plików (*~yourfile.txt*), potencjalnie zanieczyszczając w ten sposób system plików.

Instalowanie nano

We wszystkich popularnych dystrybucjach Linuksa, z których prawdopodobnie będziesz korzystać, nazwa pakietu dla nano to nano — aby go zainstalować, użyj menedżera pakietów preferowanego dla danego systemu operacyjnego (w tym przypadku instalujemy go w Ubuntu):

```
apt-get install nano
```

Ściągawka z nano

Oficjalną aktualną ściągawkę dla nano możesz znaleźć na stronie <https://www.nano-editor.org/dist/latest/cheatsheet.html>.

Oto niektóre z najbardziej przydatnych poleceń:

Obsługa plików

- *Ctrl+S* — zapisuje bieżący plik;
- *Ctrl+O* — oferuje zapisanie pliku („zapisz jako”);
- *Ctrl+R* — wstawia plik do bieżącego pliku;
- *Ctrl+X* — zamyka bufor i umożliwia wyjście z nano.

Edytowanie

- *Ctrl+K* — wycina bieżącą linię i kopiuje do schowka;
- *Ctrl+U* — wkleja zawartość schowka;

- *Alt+3* — komentuje lub odkomentowuje linię lub blok tekstu;
- *Alt+U* — cofa ostatnią czynność;
- *Alt+E* — powtarza ostatnią cofniętą czynność.

Wyszukiwanie i zastępowanie

- *Ctrl+Q* — rozpoczyna wyszukiwanie wstecz;
- *Ctrl+W* — rozpoczyna wyszukiwanie do przodu;
- *Alt+Q* — wyszukuje następne wystąpienie wstecz;
- *Alt+W* — wyszukuje następne wystąpienie do przodu;
- *Alt+R* — rozpoczyna sesję zastępowania.

Vi(m)

Vi (często nazywany również *ex-vi* lub *nvi*) to edytor tekstu w wierszu poleceń. Vim (*vi* i*m*proved) natomiast to rozszerzona wersja, której wiele osób używa jako całego IDE. Edytory *vi* i *vim* mają te same podstawowe polecenia i powiązania klawiaturowe, więc jeśli nauczysz się podstaw, doskonale sobie poradzisz bez względu na to, do jak przestarzałego lub nowoczesnego systemu się zalogujesz.

Musimy jednak uczciwie ostrzec, że *vim* jest skomplikowany i ma stosunkowo stromą krzywą uczenia się. Aby go opanować, prawdopodobnie będziesz musiał poświęcić kilka tygodni wolnego czasu na naukę i eksperymentowanie — jest to porównywalne z konfigurowaniem pierwszego serwera linuksowego lub pisanie pierwszego 500-liniowego programu.

Wspaniałą rzeczą w nauce edytora *vim* jest to, że możesz go używać lokalnie na laptopie lub zdalnie na serwerze, który nie ma GUI, a wrażenie wydajności podczas edytowania w obu przypadkach jest takie samo. Aby skutecznie z korzystać z tego narzędzia, ważna jest zarówno zmiana sposobu myślenia, jak i zrozumienie jego słownictwa (**poleceń** i **trybów**, którym się przyjrzymy).

Jak w przypadku nabywania każdej innej umiejętności, aby naprawdę dobrze zrozumieć to narzędzie i zacząć pewnie się nim posługiwać, niezbędne są ćwiczenia, a następnie regularna praktyka. Prawdopodobnie początkowo przyjdzie Ci przezwyciężyć trudności i będziesz czuć się nieco zagubiony, ale nie pozwól się zniechęcić.

Vim jest edytorem modalnym, co oznacza, że te same przyciski wykonują różne czynności w zależności od tego, w którym trybie się znajdujesz. Gdy jesteś na przykład w trybie wstawiania, naciśnięcia przycisków są zapisywane w edytowanym pliku (lub buforze) — podobnie jak w IDE lub programie Microsoft Word. Jednak w trybie normalnym naciśnięcie tych samych liter na klawiaturze spowoduje wykonanie odpowiedniego powiązania przycisków. Po oswojeniu się z tą koncepcją **edycji modalnej** reszta nauki edytora *vi/vim* to już tylko praktyka.

Na przykład jeśli uruchomisz vima i dwukrotnie wpiszesz małą literę *i*, pierwsze *i* spowoduje wejście w tryb wstawiania, a dopiero drugie faktycznie zapisze znak *i* w oknie edycji (buforze), za pomocą którego będziesz edytować plik. Jeżeli wydaje Ci się to zagmatwane, nie przejmuj się. Nawet jeśli nigdy nie zdecydujesz się używać vima jako zwykłego IDE, po lekturze tego rozdziału poczujesz się dużo pewniej w zakresie znajomości jego podstaw.

Uwaga

Warto również wspomnieć, że gdy piszemy ten rozdział, zaczyna zyskiwać popularność inny, podobny do vima edytor, a mianowicie nvim (neovim). Większość tego, co ma zastosowanie do vima, dotyczy również nvima, nie musisz się więc przejmować, od którego zacząć. Główne różnice dotyczą tworzenia wtyczek, dlatego nic nie stracisz, jeśli w przyszłości zdecydujesz się przejść z vima na neovim, tak jak my.

Polecenia vi/vima

Oto kilka podstawowych poleceń edytora vi(m) (przed użyciem któregokolwiek z nich naciśnij *Escape*, aby znaleźć się w trybie normalnym):

Tryby

- *v* — przejście do trybu wizualnego. Ta funkcjonalność istnieje tylko w vimie (*vi* jej nie ma) i na początku może być nadużywana, ponieważ znają ją osoby korzystające wcześniej z innych edytorów.
- *ESC* — wyjście z dowolnego trybu, w którym się znajdujesz, i przejście do trybu normalnego, gdzie można wydawać polecenia.

Tryb poleceń

Do trybu poleceń można wejść z trybu normalnego (naciśnij *Escape*) przez wpisanie dwukropka (:). Dla jasności dodaliśmy dwukropki do przedstawionych poniżej poleceń.

Polecenia pomocnicze

- *:set number* — pokazuje numery linii;
- *:set paste* — jest pomocne, jeśli zamierzasz wkleić coś do vima i nie chcesz, aby powodowało to zmianę oryginalnych wcięć (możesz to wyłączyć poleceniem *:set nopaste*).

Wychodzenie z edytora

- *:q* — wyjście;
- *:q!* — wyjście bez zapisywania (wymusza zamknięcie);
- *:w* — zapisanie pliku;
- *:wq* — zapisanie i wyjście;
- *:wqa* — tylko w vimie; pomaga, gdy otwartych jest wiele paneli, takich jak wtyczka otwierająca w panelu bocznym przeglądarkę plików.

Tryb normalny

Tryb normalny to tryb, w którym znajdujesz się po uruchomieniu vima i przed wpisaniem czegokolwiek. Zawsze możesz wrócić do trybu normalnego, naciskając przycisk *Escape*.

Nawigacja

- *k* — przesunięcie w górę;
- *j* — przesunięcie w dół;
- *l* — przesunięcie w prawo;
- *h* — przesunięcie w lewo.

Można też używać klawiszy strzałek, ale pomocne może być myślenie w kategoriach skrótów vi, zamiast posługiwania się nim jak zwykłym edytorem. Odkryliśmy, że podczas ćwiczeń pomaga trzymanie się klawiszy nawigacji edytora vi.

- *w* — następne słowo;
- *b* — początek bieżącego lub poprzedniego słowa;
- *^* lub *O* — przejście do początku linii;
- *\$* — przejście do końca linii;
- *gg* — przejście do początku pliku;
- *G* — przejście do końca pliku.

Edytowanie

- *i* — przejście do trybu wstawiania (wpisywania rzeczywistego tekstu);
I — wstawianie na początku linii;
- *a* — wstawianie tekstu, który jest dołączany za kursorem; *A* — dołączanie na końcu bieżącej linii;
- *o* — otwieranie nowej linii (*O* otwiera nową linię przed bieżącą);
- */* — wyszukiwanie na podstawie wzorca (działają tutaj wyrażenia regularne; użyj *Enter* do wyszukiwania, a *n* i *Shift+n*, aby przejść do przodu lub do tyłu przez wyniki wyszukiwania);
- *dd* — usuwanie (i wycinanie) bieżącej linii;
- *y* — kopiowanie (ang. *yank*) zaznaczonego tekstu;
- *yy* — kopiowanie bieżącej linii;
- *p* — wstawianie (wklejanie) tekstu za kursorem;
- *u* — cofanie ostatniej zmiany;
- *Ctrl+R* — powtórzenie;
- *nX* — gdzie *n* jest liczbą, a *X* jest poleceniem, czyli polecenie *X* jest wykonywane *n* razy, na przykład *3dd* usunie trzy linie.

Wskazówki dotyczące nauki edytora vi(m)

W ciągu kilku tygodni wykonywania normalnych zadań edycyjnych możesz poczuć się całkiem pewnie w pracy z vimem. Jednak nie zawsze łatwo jest zacząć. Oto nasze wskazówki, które pomogą Ci łatwiej przez to przebrnąć.

Użyj vimtutora

Vim jest wyposażony we wbudowany samouczek. Jeśli chcesz zacząć korzystać z vima, warto zacząć od tego samouczka. Wystarczy uruchomić vimtutora w wierszu poleceń, aby otworzyć vima i samouczek.

Myślenie w kategoriach mnemotechnik

Podczas korzystania z edytora vi(m) do edycji plików staraj się „budować zdania” za pomocą poznanych poleceń. Na przykład *d2w* oznacza *delete two words*, czyli „usuń dwa słowa”. Chociaż wspomnieliśmy o używaniu odpowiednich słów z powyższej listy poleceń, każdy preferuje określone sposoby zapamiętywania, więc możesz z powodzeniem tworzyć własne słownictwo.

Unikaj używania klawiszy strzałek

Unikaj używania klawiszy strzałek i rozważ wyłączenie tej funkcji. Dzięki temu nie potraktujesz vima jako kolejnego edytora i skrócisz czas potrzebny do uzyskania biegłości w posługiwaniu się standardowymi powiązaniem przycisków. Chociaż na początku możesz czuć dyskomfort, po kilku sesjach przyzwyczaisz się do podstawowych powiązań przycisków vima.

Unikaj używania myszy

Chociaż vima można używać z myszką do wizualnego zaznaczania, warto oprzeć się tej pokusie i trenować pamięć roboczą skrótów klawiaturowych. W przeciwnym razie będziesz przełączać się między tymi opcjami i nie będziesz czuć się pewnie, gdy nadejdzie kluczowy moment, na przykład podczas rozwiązywania problemów na zdalnym serwerze, który o trzeciej rano nie ma danych wejściowych myszy.

Nie używaj gvima

Mimo że gvim (Graphical/GUI vim) może być przydatny, korzystanie z jego skrótów graficznych nie jest dobrym pomysłem, gdy nie jest dostępny odpowiedni terminal. Zaletą edytora vi(m) jest to, że pozwala skutecznie operować tekstem za pomocą klawiatury, kiedy nie masz dostępnego środowiska graficznego — jak w przypadku wielu serwerów linuxowych, których problemy będziesz rozwiązywać po przeczytaniu tej książki!

Unikaj rozpoczynania z rozbudowaną konfiguracją lub wtyczkami

Typowym błędem początkującego jest rozpoczęcie od czyjejś konfiguracji vima. Chociaż na początku wydają się przydatne, mocno spersonalizowane ustawienia vima mogą przeszkadzać, gdy próbujesz nauczyć się podstawowych pojęć. Rozbudowane konfiguracje

nie sprawią magicznie, że będziesz bardziej produktywny, zwłaszcza na początku. Gdy zyskasz większą biegłość, napiszesz własną konfigurację.

Kolejną rzeczą, której należy unikać, jest nadmierne korzystanie z wtyczek, szczególnie na wczesnym etapie. Czasami wtyczki psują różne rzeczy, co może stać się obciążeniem i powodować więcej problemów. Gdy dopiero zaczynasz, rozwiązywanie problemów z wtyczką vima nie jest czymś, z czym chciałbyś mieć do czynienia. Zewnętrzne pliki konfiguracyjne i wtyczki mogą być niezwykle przydatne, ale mogą również stać się kulą u nogi: gdy nagle zostaniesz wrzucony do środowiska poza maszyną programistyczną, nie będziesz mógł polegać na tych wszystkich fantazyjnych dodatkach. Jeśli jesteś w dużym stopniu zależny od niestandardowego przepływu pracy, nawet podstawowe zmiany mogą stać się trudne i frustrujące, zwłaszcza pod wpływem stresu.

Rozsądniejszym podejściem jest rozpoczęcie od minimalnej konfiguracji i dodawanie tylko niewielkich elementów, które w pełni rozumiesz (i których na pewno będziesz potrzebować). Znacznie lepiej spożytkujesz swój czas, korzystając z tego edytora w prawdziwych projektach, ponieważ pozwoli Ci to zachować w pamięci roboczej najważniejsze skróty vima. Zajmie to trochę czasu, ale ostatecznie zaczniesz używać ich bez myślenia o rzeczywistych poleceniach — będziesz raczej posługiwać się własnymi mnemotechnikami.

Oto przykład minimalnej konfiguracji vima, która może być pomocna na początku. Możesz ją zmienić wedle uznania lub wybrać jedynie te elementy, które wydają Ci się przydatne.

Umieść to w swoim pliku `$HOME/.vimrc`:

```
" Likwiduje kompatybilność z vi, ponieważ chcemy korzystać z zalet vima
set nocompatible
```

```
" Włącza podświetlanie składni
syntax on
```

```
" Znacząco zwiększa pojemność historii poleceń
set history=10000
```

```
" Wcięcia ma być oparte na poprzedniej linii
set autoindent
```

```
" Wyszukiwania mają zawijać się na końcu pliku
set wrapscan
```

```
" Pokazuje bieżący tryb w wierszu poleceń
set showmode
```

```
" Wyświetla częściowe polecenia w ostatniej linii
set showcmd
```

```
" Podświetla wyszukiwania
set hlsearch
```

```
" Używaj wyszukiwania bez uwzględnienia wielkości znaków
set ignorecase
```

" Przy stosowaniu wielkich liter nie używaj wyszukiwania bez uwzględnienia wielkości znaków
set smartcase

" Wyświetla pozycję kursora na dole
set ruler

Wiązania vima w innym oprogramowaniu

Jeśli zaczniesz Ci się podobać działanie vima i stanie się ono Twoim preferowanym sposobem pracy, wiedz, że wiele edytorów tekstu i środowisk IDE ma opcje i wtyczki, które pozwalają przejść do trybu wprowadzania danych charakterystycznego dla vima. Istnieją nawet przeglądarki internetowe ze sposobem wprowadzania danych w stylu vima!

Jeśli interesuje Cię to zagadnienie lub chcesz powtórzyć sobie materiał na temat vima przedstawiony w tym rozdziale, na stronie <https://www.youtube.com/watch?v=ggSyF41SVFr4> możesz znaleźć samouczek wideo, który obejmuje niektóre z najważniejszych tematów omówionych w tym rozdziale (a nawet niektóre dodatkowe funkcjonalności vima!).

Edytowanie pliku, do którego nie masz uprawnień

Niezależnie od tego, jakiego edytora używasz, czasami będziesz chciał edytować plik, dla którego Twój użytkownik nie ma uprawnień zapisu. Jeżeli jesteś na przykład zwykłym użytkownikiem i chcesz edytować plik `/etc/hosts` (plik należący do użytkownika root i zapisywany tylko przez niego), musisz zostać rootem lub użyć polecenia `sudo`. Więcej informacji na ten temat znajdziesz w rozdziale 7. „Użytkownicy i grupy”.

Chociaż do edycji plików z uprawnieniami roota może być stosowane polecenie takie jak `sudo $EDITOR /etc/hosts`, lepszym podejściem do wykonania polecenia edycji jako root jest wykorzystanie polecenia `sudoedit`:

- `sudoedit /etc/hosts`,
- `EDITOR=nano sudoedit /etc/hosts`,
- `EDITOR=vi sudoedit /etc/host`.

W pierwszym przykładzie zostanie użyty dowolny edytor ustawiony w zmiennej środowiskowej `EDITOR`, podczas gdy pozostałe dwa polecenia przekazują (lub nadpisują) zmienną środowiskową `EDITOR` jako część polecenia.

Ustawianie preferowanego edytora

Linux i właściwie wszystkie systemy unixopodobne pozwalają ustawić preferowany edytor za pomocą zmiennej środowiskowej `EDITOR`. Większość programów wiersza poleceń, która uruchamia edytor dla określonych zadań, np. `git` podczas zatwierdzania

`commitu` lub `vi` `sudo` przy edycji pliku `sudoers`, użyj tej zmiennej, aby określić, który edytor otworzyć. Możesz ustawić dla zmiennej `EDITOR` ścieżkę do dowolnego edytora, nawet graficznego (pod warunkiem że system ma zainstalowany graficzny interfejs użytkownika):

```
bash-3.2$ echo $EDITOR
nano
bash-3.2$ export EDITOR=vim
```

Zwróć uwagę, że powyższe polecenie interaktywnej powłoki będzie działać tylko do momentu zamknięcia bieżącej sesji powłoki. Aby zachować to ustawienie w powłoce Bash, dodaj je do swojego pliku `~/.bashrc`. Więcej informacji na ten temat znajdziesz w rozdziale 4. „Korzystanie z historii powłoki”.

Podsumowanie

W tym rozdziale wyjaśniliśmy, jak edytować pliki tekstowe w wierszu poleceń. Najpierw wprowadziliśmy najprostszy sposób na rozpoczęcie pracy (`nano`), a następnie pokazaliśmy, jak zacząć doskonalić umiejętności — znajomość edytorów `vi` i `vim` oraz ich powiązań przycisków, które są obsługiwane w niezwykle szerokim zestawie oprogramowania, zaowocuje podczas kariery programisty.

Aby rozpocząć edytowanie w wierszu poleceń, możesz skorzystać ze ściągawek przedstawionych w tym rozdziale, ale po kilku dniach ćwiczeń będziesz gotowy, aby nauczyć się dodatkowych skrótów i poleceń `vima`. W tym celu najlepiej korzystać ze wszelkich możliwych źródeł: `vimtutora`, ściągawek online i filmików z YouTube’a. Bardzo podoba nam się również książka *Practical Vim, 2nd Edition* Drew Neila.

Pewność i biegłość w edycji tekstu w wierszu poleceń to jeden z najlepszych sposobów, aby poczuć się jak profesjonalista. Rozwijaj zatem te umiejętności!

W tym rozdziale przyjrzymy się dwóm elementom, których Linux używa do zarządzania zasobami i zapewniania bezpieczeństwa: użytkownikom i grupom. Po wyjaśnieniu podstaw i omówieniu bardzo wyjątkowego użytkownika o nazwie *root*, pokażemy Ci, w jaki sposób koncepcja grup użytkowników Linuksa dodaje wygodną warstwę nad warstwą abstrakcji użytkownika.

Po omówieniu niezbędnej teorii przejdziemy bezpośrednio do praktycznych poleceń do tworzenia i modyfikowania użytkowników i grup. Zobaczysz, *czym tak naprawdę jest użytkownik Linuksa* (wskazówka: to tylko trzy linie zwykłego tekstu) — wiedza ta z pewnością przyda Ci się podczas rozmowy o pracę.

W tym rozdziale omawiamy następujące zagadnienia:

- użytkownicy i ich przeznaczenie;
- różnica między użytkownikiem *root* i normalnymi użytkownikami oraz sposoby przełączania się między nimi;
- tworzenie i modyfikowanie użytkowników i grup;
- metadane użytkowników.

Czym jest użytkownik?

W kontekście Uniksa użytkownik jest nazwaną encją, która może wykonywać różne działania w systemie. Użytkownicy mogą uruchamiać i posiadać procesy, mogą posiadać pliki i katalogi oraz mieć do nich różne uprawnienia, a ponadto mogą wykonywać tylko określone czynności i korzystać jedynie ze wskazanych zasobów w systemie. Praktycznie użytkownik to „osoba”, za pomocą której się logujesz i która jest właścicielem uruchamianych przez Ciebie procesów lub edytowanych plików.

Słowo „użytkownik” jest oczywiście metaforą prawdziwej osoby z kontem użytkownika, hasłem itd. Jednak w prawdziwych systemach większość użytkowników nie reprezentuje konkretnych ludzi. Są to konta maszyny przeznaczone do grupowania zasobów, takich jak procesy i pliki, w celach zapewnienia bezpieczeństwa lub organizacji.

Istnieje jednak znacznie ważniejsze rozróżnienie niż to, czy konto jest przeznaczone do interaktywnego korzystania przez operatora. Istnieją dokładnie dwa rodzaje użytkowników i zanim przejdziemy do praktycznych umiejętności zarządzania użytkownikami, musimy przyrzeć się temu rozróżnieniu.

Root kontra reszta świata

Niektóre polecenia niosą niepożądane konsekwencje. Na przykład `fdisk` może wyczyścić partycje dysku lub w inny sposób zmodyfikować sprzęt. `iptables` może otworzyć port sieciowy i pozwolić atakującemu wykorzystać lukę w zabezpieczeniach. Nawet użycie nieszkodliwego polecenia `echo` do wysłania wartości w niewłaściwe miejsce w systemie plików może zmienić konfigurację systemu operacyjnego w subtelny, lecz bardzo szkodliwy sposób.

Aby tego uniknąć, środowisko uniksowe, w którym działa interfejs wiersza poleceń, ma wbudowane pewne zabezpieczenia. W każdym systemie uniksowym istnieje „superużytkownik” o nazwie `root`. W rezultacie podstawowy model bezpieczeństwa jest następujący:

- Najpierw jest `root`. Ten użytkownik jest odpowiednikiem administratora w innych systemach i jest użytkownikiem o największej liczbie uprawnień — `root` może robić prawie wszystko.
- Potem są wszyscy inni. Użytkownicy inni niż `root` mają ograniczone uprawnienia — nie mogą uruchamiać procesów ani edytować plików, które mogą mieć wpływ na cały system, ale mogą uruchamiać własne (nieposiadające wysokich uprawnień) aplikacje i edytować własne pliki.

Aby nie dochodziło do problemów, tylko użytkownik `root` może wykonywać polecenia, które zmieniają ważne aspekty systemu. Ponieważ nawet pozornie nieszkodliwe polecenia, jeśli są uruchamiane z określonymi argumentami, mogą spowodować spustoszenie, możesz potrzebować uprawnień użytkownika `root` nawet do edycji pliku tekstowego.

Polecenie `sudo`

Ponieważ byłoby niedogodnością logowanie się jako oddzielny użytkownik za każdym razem, gdy chcemy zrobić coś potencjalnie niebezpiecznego w systemie, istnieje polecenie `sudo`. Poprzedzanie innego polecenia poleceniem `sudo` (od ang. *substitute user (and) do* — „zamień użytkownika i zrób”) pozwala wykonać to inne polecenie *jako użytkownik root*. Gdy polecenie zakończy wykonywanie i działanie, następne polecenie zostanie zinterpretowane jako pochodzące od zwykłego użytkownika (innego niż `root`).

Możesz zobaczyć to zachowanie, uruchamiając dwa polecenia. Najpierw uruchom polecenie `whoami`, które wypisuje bieżącego użytkownika:

```
whoami
```

W tym przypadku zalogowany jest użytkownik `dave`, więc to polecenie wypisze takie dane wyjściowe:

```
dave
```

Teraz przed tym samym poleceniem dodaj `sudo`:

```
sudo whoami
```

Mimo że nadal jesteś zalogowany jako użytkownik inny niż root, Twój **efektywny** identyfikator użytkownika zmienił się na czas trwania pojedynczego polecenia z powodu sudo:

root

Przyjrzyjmy się bardziej praktycznemu przykładowi, w którym chcemy uruchomić pojedynczą akcję jako root, a następnie kontynuować uruchamianie innych poleceń jako zwykły użytkownik:

```
sudo systemctl start nginx
```

<wracasz do działania jako zwykły użytkownik>

To pierwsze polecenie uruchamia serwer internetowy nginx (przy założeniu, że pakiet nginx jest zainstalowany), co może zrobić tylko root. Wszelkie późniejsze polecenia są ponownie wykonywane jako zwykły użytkownik.

Jest to powszechny przepływ pracy, który zapewnia bezpieczeństwo — większość czasu spędzasz w systemie, pracując jako zwykły użytkownik, który nie może zarządzać całym systemem za pomocą jednego polecenia. Kiedy potrzebujesz uprawnień użytkownika root, wywołujesz je z dodaniem prefiksu *tylko do tych poleceń, które tego wymagają*. To miła psychologiczna bariera zapobiegająca przypadkowemu narozrabianiu w systemie.

Wzorec ten jest używany do różnych potencjalnie niebezpiecznych działań w systemie, takich jak edytowanie plików konfiguracyjnych na poziomie systemu, tworzenie katalogów poza katalogiem domowym użytkownika (co omówimy dalej w tym rozdziale) i wiele innych:

- `sudo mkdir /var/log/foobar,`
- `sudo vim /etc/hosts,`
- `sudo mount /dev/sdb1.`

Polecenia sudo możesz użyć, aby uzyskać długotrwałą sesję powłoki roota, jeśli planujesz uruchamiać wiele poleceń jako root (albo jeśli rozwiązujesz problem z czymś, co działa jako root, lub symulujesz środowisko, w którym byłby wykonywany skrypt `cloud-init`):

sudo -i

Spowoduje to uruchomienie **interaktywnej** sesji powłoki z uprawnieniami użytkownika root. Zachowaj jednak ostrożność! W takim przypadku nic nie stanie Ci na przeszkodzie, aby zniszczyć system w wyniku błędu lub źle wpisanego polecenia.

Podczas gdy sudo domyślnie zastępuje bieżącego użytkownika użytkownikiem root, korzystając z opcji `-u`, możesz również zmienić go na innego użytkownika, na przykład:

```
sudo -u myuser vim /home/myuser/.bashrc
```

Spowoduje to otwarcie pliku `/home/myuser/.bashrc` w edytorze vim jako użytkownik myuser.

Dokładne uprawnienia poszczególnych użytkowników (lub grup) można zdefiniować w pliku `/etc/sudo.conf`. Nigdy nie należy edytować tego pliku bezpośrednio, lecz użyć polecenia `visudo`.

Czym jest grupa?

Grupy są dodatkowym podstawowym elementem, który umożliwia udostępnianie uprawnień zestawowi użytkowników. Są często wykorzystywane do uzyskania funkcjonalności zestawu uprawnień lub profilu. W systemie Linux często występuje na przykład grupa zwana *sudoers*, a w systemie macOS można napotkać grupę *wheel*. Zgodnie z konwencją użytkownicy, którzy są członkami grup *sudoers* lub *wheel* w tych systemach, mogą stosować *sudo* do wykonywania poleceń z uprawnieniami użytkownika *root*. Pod względem funkcjonalnym jest to analogiczne do dodawania użytkownika do grupy *Administrators* w systemie Windows.

Skoro grupy są przydatne do zarządzania tym, kto może uruchamiać polecenie *sudo*, mogą posłużyć także do grupowania użytkowników i zarządzania innymi rodzajami uprawnień.

Miniprojekt: zarządzanie użytkownikami i grupami

Wyobraźmy sobie, że chcemy umożliwić każdemu użytkownikowi będącemu programistą w naszej firmie odczytywanie określonego pliku — nazwijmy go *document.txt*. W tym celu możemy utworzyć grupę *developers* i dodać do niej wszystkich naszych użytkowników będących programistami.

Następnie gdy ustawiamy własność i uprawnienia dla pliku *document.txt*, możemy odwołać się do grupy programistów zamiast śledzić indywidualnie każdego, kto może być członkiem tej grupy.

Tworzenie użytkownika

W systemie Linux, który ma zainstalowane polecenie *adduser*, możemy wykorzystać je do **interaktywnego** utworzenia użytkownika o nazwie *dave*. Jeżeli nie masz zainstalowanego tego polecenia, ten pakiet ma zwykle nazwę *useradd* (więcej informacji na temat instalowania pakietów znajdziesz w rozdziale 9. „Zarządzanie zainstalowanym oprogramowaniem”).

Uruchomienie tego polecenia z nazwą użytkownika jako jedynym argumentem przypomina proces kreatora dla tworzenia użytkownika. Zwróć uwagę, że stosujemy tutaj *sudo*, ponieważ tylko *root* może dodawać i usuwać użytkowników:

```
$ sudo adduser steve
Adding user `steve' ...
Adding new group `steve' (1000) ...
Adding new user `steve' (1000) with group `steve' ...
Creating home directory `/home/steve' ...
Copying files from `/etc/skel' ...
New password:
```

```
Retype new password:
passwd: password updated successfully
Changing the user information for steve
Enter the new value, or press ENTER for the default
Full Name []: Steve
Room Number []:
Work Phone []:
Home Phone []:
Other []:
Is the information correct? [Y/n] y
```

W tym listingu pogrubiliśmy fragmenty, które wymagały z naszej strony interakcji użytkownika, a mianowicie ustawienia hasła, imienia i nazwiska oraz potwierdzenia, że chcemy utworzyć tego użytkownika w systemie.

To dobry sposób na dodanie jednego lub dwóch użytkowników, ale co, jeśli pracujesz na serwerze testowym Linuksa, który potrzebuje indywidualnych kont dla 300 największych klientów? Wówczas będziesz potrzebować nieinteraktywnego polecenia `useradd`, które pozwala określić atrybuty użytkownika jako argumenty dla pojedynczego polecenia. Ułatwia to pisanie skryptów w celu wprowadzania zmian związanych z użytkownikami (zobacz naszą uwagę na temat możliwości skryptowych dalej w tym rozdziale):

```
useradd --home-dir /home/dave --create-home --shell /bin/zsh -g dave -G
↳sudoers dave
```

To polecenie wykonuje również następujące czynności:

- ustawia i tworzy katalog domowy użytkownika (`--home-dir i --create-home`);
- ustawia niestandardową powłokę (`--shell`);
- ustawia podstawową grupę użytkownika na `dave` (choć może to być również coś w rodzaju `employees`) za pomocą opcji `-g`;
- dodaje dodatkowe członkostwo grupy do grupy `sudoers` (możesz przekazywać tutaj wiele nazw grup oddzielonych przecinkami).

I to wszystko — jeśli polecenie zakończy się powodzeniem, nowy użytkownik zostanie utworzony!

Jeszcze jednak nie skończyliśmy — ten użytkownik będzie pracował nad nową, ściśle tajną aplikacją `tutoriallinux`, utwórzmy więc grupę dla tego projektu i dodajmy do niej naszego nowego użytkownika.

Tworzenie grupy

Aby utworzyć nową grupę o nazwie `tutoriallinux`, zastosuj polecenie `groupadd`:

```
groupadd tutoriallinux
```

Spowoduje to utworzenie w systemie nowej grupy przez dodanie linii do pliku konfiguracyjnego `grupy/etc/`, który jest zapisem wszystkich grup istniejących w systemie Unix. Utworzenie grupy możesz zweryfikować, „grepując” (wyszukując) nazwę grupy w tym pliku:

```
# grep tutorialinux /etc/group
tutorialinux:x:1001:
```

Widać, że grupa o nazwie `tutorialinux` istnieje i ma **ID grupy (GID)** 1001.

Nie będziemy się zagłębiać w to, co oznacza tutaj znak `x` — wystarczy wiedzieć, że ten plik składa się z jednej linii dla każdej grupy, z wartościami oddzielonymi dwukropkiem. Zawsze będą interesowały Cię tylko takie elementy, jak nazwa grupy (pierwsza kolumna), identyfikator grupy (trzecia kolumna) i członkowie (ostatnia kolumna, która w tym przykładowym pliku jest pusta).

Modyfikowanie użytkownika

Podobnie jak `useradd` pozwala ustawić pożądane metadane użytkownika podczas jego tworzenia, polecenia `usermod` i `gpasswd` umożliwiają modyfikowanie wszystkich aspektów istniejącego użytkownika. Dodajmy do nowej grupy `tutorialinux` utworzonego wcześniej użytkownika `dave`, aby mógł pracować nad plikami projektu, które mogą zobaczyć lub modyfikować tylko członkowie grupy.

Dodawanie użytkownika do grupy

Aby zmienić podstawową grupę użytkownika, należy użyć polecenia `sudo usermod -g nazwa_grupy nazwa_użytkownika`.

Jednak nie do końca tego tutaj chcemy: użytkownik `dave` powinien nadal być w tytułowej grupie `dave`; chcemy, aby `dave` był *również* członkiem grupy `tutorialinux`. Aby dodać użytkownika do grupy *bez* ustawiania jej jako głównej grupy tego użytkownika, użyj opcji `-aG` (*dodaj do dodatkowych grup*):

```
sudo usermod -aG tutorialinux dave
```

Jeśli ponownie sprawdzisz plik `/etc/group`, zobaczysz, że użytkownik `dave` jest teraz członkiem trzech grup: `dave`, `sudoers` i `tutorialinux`:

```
grep dave /etc/group

sudoers:x:27:dave
dave:x:1000:
tutorialinux:x:1001:dave
```

Korzystając z poleceń poznanych w poprzednim rozdziale, służących do modyfikowania własności plików i uprawnień, możesz teraz kontrolować dostęp każdego członka grupy `tutorialinux` do określonych plików i katalogów.

Teraz, gdy któryś z użytkowników zakończy pracę nad projektem `tutorialinux`, możesz wycofać i cofnąć jego dostęp bez konieczności modyfikowania indywidualnych uprawnień do plików i katalogów.

Usuwanie użytkownika z grupy

Aby usunąć użytkownika z grupy, możesz użyć polecenia `gpasswd` w następujący sposób:

```
gpasswd -delete nazwa_użytkownika nazwa_grupy
```

Usuwanie użytkownika

Aby całkowicie usunąć użytkownika Linuksa, użyj polecenia `userdel`:

```
userdel -r nazwa_konta
```

Jeśli chcesz zachować katalog domowy tego użytkownika, pomiń flagę `(-r/--remove)`.

Usuwanie grupy

Istnieje również polecenie `groupdel`, służące do usuwania grup, które przestały być potrzebne:

```
groupdel nazwa_grupy
```

Zagadnienia zawansowane, czyli czym tak naprawdę jest użytkownik?

Jeśli chodzi o użytkowników i grupy, można wyraźnie dostrzec to, co jest wspaniałe w systemach Unix i Linux: nie ma tu zbyt wiele magii.

Użytkownik Linuksa to tak naprawdę tylko **identyfikator użytkownika (UID)**, który jest prostą liczbową reprezentacją użytkownika (32-bitową liczbą całkowitą bez znaku). UID użytkownika root jest równy 0. Wszyscy pozostali użytkownicy mają UID większy od 0. To samo dotyczy grup.

Informacje te nie są przechowywane w żadnej tajnej lokalizacji, formacie binarnym czy zastrzeżonej strukturze danych, z którą może pracować tylko system operacyjny: użytkownicy i grupy są definiowane w plikach tekstowych, które tradycyjnie można modyfikować za pomocą kilku prostych poleceń, które tutaj omówiliśmy.

Ta prostota i brak magii oznaczają, że zwykli śmiertelnicy (tacy jak spanikowany programista, który niewiele zapamiętał z tego rozdziału) mogą szybko poznać stan użytkowników i grup w uruchomionym systemie podczas rozwiązywania problemów z błędami aplikacji mogącymi wynikać z nieprawidłowo przygotowanego środowiska hosta, w którym brakuje niezbędnego użytkownika aplikacji. Przydaje się to również w trakcie rozmowy o pracę na stanowisku programisty, gdy rozmowa schodzi na temat inżynierii systemów.

Zatem w celu ugruntowania Twojej wiedzy na temat tego zagadnienia w następnym punkcie przedstawimy kilka bardziej przydatnych kwestii dotyczących tego, co dzieje się pod maską, gdy tworzymy użytkowników i grupy oraz nimi zarządzamy.

Metadane i atrybuty użytkownika

Użytkownicy zdefiniowani tylko przez liczbę nie są szczególnie przydatni — dodatkowe metadane dodadzą nieco pikanterii. Na przykład konto używane przez jednego z autorów do codziennej pracy na komputerze z systemem Linux lub macOS, które ma UID 502, może również mieć następujące atrybuty:

- przyjazna nazwa użytkownika (dave);
- własna grupa (dave);
- członkostwo w różnych grupach (staff, developer i wheel);
- powłoka logowania (bash, zsh itd.);
- katalog domowy (/Users/dave/ w systemie macOS lub /home/dave/ w systemie Linux).

Możesz uzyskać informacje o swoim bieżącym użytkowniku, uruchamiając polecenie `id`:

```
# id
uid=0(root) gid=0(root) groups=0(root)
```

Domyślnie wszystkie te dodatkowe informacje o użytkowniku definiuje i zawiera kilka plików:

- */etc/passwd* zawiera nazwę użytkownika, UID, GID, katalog domowy i powłokę logowania, a wszystkie te informacje są rozdzielone dwukropkami i znajdują się w jednej linii dla każdego użytkownika:

```
root@localhost:~# cat /etc/passwd
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sy
```

- */etc/shadow* zawiera zaszyfrowane, „posolone” hasła użytkownika i jest czytelny tylko dla użytkownika root:

```
root@localhost:~# cat /etc/shadow
```

```
root:$6$SPevRPxD94AYwtmF$I0p9k15dnaN8FW8RUDDQ1i fLPp9pJ3btgJcMfI
QEs1kT.ZNjDfX66XB0cP0BZzkRcG0b3Rwq6qTsDQ0jiZNh/:19251:0:99999:7:::
daemon*:19251:0:99999:7:::
bin*:19251:0:99999:7:::
sys*:19251:0:99999:7:::
sync*:19251:0:99999:7:::
```

- */etc/group* to odpowiednik */etc/passwd*, ale dla grup zamiast użytkowników (ten plik widziałeś wcześniej w tym rozdziale):

Opcjonalnie katalog domowy, taki jak */home/dave*:

```
# /etc/passwd username:password:UID:GID:comment:home:shell
```

Ostrzeżenie

Chociaż warto wiedzieć, co zawierają te pliki, NIGDY NIE NALEŻY edytować żadnego z nich ręcznie. Do tworzenia, usuwania i modyfikowania użytkowników i grup systemu używaj narzędzi, które stosowaliśmy wcześniej w tym rozdziale.

Miejmy nadzieję, że ten szybki teoretyczny przegląd ruchomych elementów — a dokładniej statycznych plików tekstowych — które tworzą użytkowników i grupy, był przydatny. Nie tylko chcemy mieć pewność, że będziesz się orientować, jak to działa, ale także mamy nadzieję, że w tym punkcie rozdziału pokazaliśmy Ci, że tak naprawdę jest to *prosty* mechanizm. Nie ma w tym żadnej magii! Możesz mieć pewność, że przy rozwiązywaniu problemu nie przegapisz niczego, gdy aplikacja nie zostanie uruchomiona lub użytkownik nie będzie miał uprawnień do przeglądania konkretnego pliku z określonymi uprawnieniami własności grupy.

Kilka słów na temat skryptów

Wspominaliśmy wcześniej, że należy preferować automatyczne narzędzia, takie jak `useradd`, zamiast interaktywnych narzędzi opartych na kreatorach, do których należy na przykład `adduser` — nawet jeśli te automatyczne narzędzia są bardziej złożone lub trudniejsze do opanowania. Być może zapytasz: „Dlaczego nie użyć narzędzia graficznego zamiast tych trudnych do zapamiętania poleceń CLI?”.

Jedną z rzeczy, których chcemy Cię nauczyć poprzez tę książkę, jest ogólne preferowanie nieinteraktywnych poleceń.

Ponieważ polecenia te nie opierają się na danych wprowadzanych przez użytkownika w czasie rzeczywistym, można używać ich w skryptach: utworzenie stu użytkowników jest prawie tak proste, jak utworzenie jednego. Jest to naprawdę przydatne, gdy masz do czynienia z rzeczywistymi problemami, takimi jak tworzenie obrazów Dockera, wielokrotne przygotowywanie środowisk produkcyjnych lub pisanie skryptów konfiguracyjnych `cloud-init` dla instancji chmury.

Dla programisty powinno to brzmieć prawdziwie: automatyzacja sprawia, że wszystko staje się bardziej powtarzalne, bezpieczne i szybkie. Jeśli nauczysz się polecenia, które nie jest interaktywne, możesz je wykorzystać jako część większej zautomatyzowanej całości, zamiast wykonywać kroki, które wymagają podatnego na błędy, czasochłonnego i ryzykownego ręcznego działania.

Podsumowanie

W tym rozdziale omówiliśmy podstawy dotyczące tego, w jaki sposób Linux wykorzystuje abstrakcje użytkowników i grup do zarządzania procesami, plikami i innych zasobami w systemie oraz do ich kontrolowania. Co równie ważne, objaśniliśmy podstawowe polecenia potrzebne do tworzenia użytkowników i grup oraz zarządzania nimi w prawdziwym systemie. Dowiedziałeś się, jaka jest istotna różnica między rootem a pozostałymi, zwykłymi użytkownikami systemu.

Następnie wykonaliśmy praktyczne ćwiczenie, w którym utworzyliśmy użytkownika, dodaliśmy grupę do systemu, zmodyfikowaliśmy tego użytkownika oraz wyczyściliśmy wszystkie utworzone zasoby.

Na koniec pokazaliśmy, że nie ma w tym żadnej magii: to tylko zwykłe pliki tekstowe, które definiują użytkowników i grupy w systemie Unix. To dobra wiadomość, gdyż ułatwi Ci to życie jako programiście, gdy będziesz wykonywać następujące czynności:

- tworzenie obrazu Dockera w celu uruchomienia aplikacji jako określonego użytkownika innego niż root;
- konfigurowanie długotrwałej instancji chmury z loginami i wspólną grupą dla Twojego zespołu zajmującego się nauką o danych;
- minimalizowanie negatywnych skutków popełnienia błędu we własnym lokalnym środowisku testowym;
- rozwiązywanie problemów z błędami użytkowników i grup w aplikacjach, na przykład w aplikacji internetowej, która potrzebuje uprawnień roota do otwarcia sekretnego pliku lub wykonywania uprzywilejowanych działań w systemie.

W następnym rozdziale wykorzystamy całą tę wiedzę, aby zagłębić się w sposób działania modelu bezpieczeństwa Uniksa, przyglądając się własności i uprawnieniom.

Własność i uprawnienia

W tym rozdziale omówimy, w jaki sposób użytkownicy i grupy są łączy się z własnością i uprawnieniami w celu utworzenia podstawowego modelu bezpieczeństwa Linuksa. Ta kombinacja podstawowych elementów służy do kontrolowania dostępu do prawie wszystkiego w systemie Linux — procesów, plików, gniazd sieciowych, urządzeń itp.

Najpierw przedstawimy wszystkie istotne informacje o plikach, które otrzymujesz z długiego listingu (oczywiście z naciskiem na uprawnienia). Następnie omówimy typowe uprawnienia, które napotkasz w produkcyjnych systemach Linuksa, a na koniec pokażemy wszystkie linuksowe polecenia, których będziesz używać do ustawiania i modyfikowania uprawnień do plików.

W tym rozdziale omawiamy następujące zagadnienia:

- odszyfrowywanie danych wyjściowych z długiego listingu;
- atrybuty pliku;
- sposób działania własności i uprawnień;
- wspólny punkt zaczepienia, czyli format uprawnień „ósemkowych”;
- praktyczne polecenia dotyczące zmiany własności i uprawnień.

Odszyfrowywanie długiego listingu

Zajmijmy się tym tematem długiego listingu.

Gdy poruszasz się po systemie, czasami nie wystarczy zobaczenie jedynie nazw plików i katalogów. Jeśli chcesz uzyskać więcej informacji na temat plików, które widzisz, użyj polecenia `ls` z jego „długą” opcją: `ls -l`.

Oto przykład danych wyjściowych po uruchomieniu tego polecenia w katalogu systemowym `/lib`. Otwórz terminal i wpisz `ls -l /lib/`:

```
# ls -l /lib/
total 56
drwxr-xr-x 10 root root 4096 Mar 8 02:12 aarch64-linux-gnu
drwxr-xr-x  5 root root 4096 Mar 8 02:12 apt
drwxr-xr-x  3 root root 4096 Mar 8 02:08 dpkg
drwxr-xr-x  2 root root 4096 Mar 8 02:12 init
```

```
lrwxrwxrwx 1 root root 39 Jul 6 2022 ld-linux-aarch64.so.1 ->
aarch64-linux-gnu/ld-linux-aarch64.so.1
drwxr-xr-x 3 root root 4096 Mar 4 2022 locale
drwxr-xr-x 3 root root 4096 Mar 8 02:12 lsb
drwxr-xr-x 3 root root 4096 Aug 29 2021 mime
-rw-r--r-- 1 root root 386 Feb 16 2023 os-release
drwxr-xr-x 2 root root 4096 Mar 8 02:12 sysctl.d
drwxr-xr-x 3 root root 4096 Apr 18 2022 systemd
drwxr-xr-x 16 root root 4096 Jan 17 2022 terminfo
drwxr-xr-x 2 root root 4096 Mar 8 02:12 tmpfiles.d
drwxr-xr-x 3 root root 4096 Mar 8 02:12 udev
drwxr-xr-x 3 root root 4096 Mar 8 02:12 usrmerge
```

Widzimy tu wiele ciekawych informacji, przeanalizujmy je zatem pole po polu.

Atrybuty pliku

W pierwszym polu wyświetlane są atrybuty pliku: typ i uprawnienia. Innymi słowy, to pole pokazuje, na jaki typ pliku patrzymy i jakie są jego uprawnienia dostępu. Domyślnie te informacje są wyświetlane w trybie symbolicznym, w przeciwieństwie do trybu numerycznego, który można wyświetlić za pomocą opcji `-n`.

Typ pliku

```
-rw-r--r-- 1 root root 386 Feb 16 2023 os-release
```

Pierwszy znak wskazuje tutaj typ pliku. W powyższym listingu znak `-` wskazuje zwykły plik. Linie rozpoczynające się od `l` wskazują na **dowiązanie symboliczne**, będące specjalnym plikiem, który nie ma własnej zawartości i jedynie wskazuje jedynie inną lokalizację w systemie plików. Możesz potraktować to jak skrót Windowsa lub alias pliku w systemie macOS.

Inne popularne typy plików to `d`, który wskazuje katalog, lub `c`, który oznacza plik znakowy — ten ostatni znajdziesz głównie w katalogu `/dev`, reprezentującym sprzętowe urządzenia wejściowe, takie jak klawiatury. Więcej informacji na temat typów plików znajdziesz w rozdziale 5. „Wprowadzenie do plików”.

Uprawnienia

```
-rw-r--r-- 1 root root 386 Feb 16 2023 os-release
```

Są to „bity uprawnień”, określające, którzy użytkownicy i które grupy w systemie mogą odczytywać, zapisywać i wykonywać dany plik. Zajmiemy się tym szczegółowo w dalszej części rozdziału, w podrozdziale „Uprawnienia”.

Liczba dowiązań twardych

```
-rw-r--r-- 1 root root 386 Feb 16 2023 os-release
```

Następne pole wskazuje liczbę dowiązań twardych. Dowiązania twarde to specjalne wskaźniki łączące nazwy plików z rzeczywistym plikiem. Tak więc w większości sytuacji będzie to 1 dla plików. W przeciwieństwie do dowiązania symbolicznego (symlinku), które wskazuje na ścieżkę pliku, dowiązanie twarde wskazuje na rzeczywisty plik. Jeśli przesuniesz plik, na który wskazuje dowiązanie symboliczne, przestanie ono być ważne; dowiązanie twarde będzie nadal wskazywać na plik, nawet jeśli plik ten zostanie przesunięty, przemianowany lub w inny sposób zmodyfikowany.

Być może zauważyłeś, że podczas gdy większość plików ma tylko jedno dowiązanie wskazujące na nie, katalogi mają bardzo różną liczbę dowiązań w tej kolumnie. Dzieje się tak, ponieważ każdy plik i katalog w tym katalogu tworzy referencję do niego. Nawet pusty katalog zaczyna się od dwóch dowiązań, „.” (skrót do „tego katalogu”) i „..” (skrót do „katalogu powyżej tego katalogu”).

Własność użytkownika

```
-rw-r--r-- 1 root root 386 Feb 16 2023 os-release
```

Trzecie pole pokazuje, który użytkownik jest właścicielem pliku — wszystkie pliki w naszym przykładzie są własnością roota. Uruchomienie `ls` w trybie liczbowym — `ls -ln` — pokaże również liczbową nazwę użytkownika zamiast przyjaznej nazwy, którą tutaj widzisz.

Własność grupy

```
-rw-r--r-- 1 root root 386 Feb 16 2023 os-release
```

Poniższe pole pokazuje grupę właścicieli, którą tutaj również okazuje się root.

Rozmiar pliku

```
-rw-r--r-- 1 root root 386 Feb 16 2023 os-release
```

Jak się domyślasz, następne pole wyświetla rozmiar pliku. Gdy nie określono żadnych dodatkowych znaczników, jest on wyświetlany w bajtach. Aby był bardziej czytelny, można użyć flagi `-h` („czytelne dla człowieka”).

Zapewne nie uszło Twojej uwadze, że wszystkie katalogi mają ten sam rozmiar pliku, 4096:

```
drwxr-xr-x 1 root root 4096 Apr 18 2022 systemd
```

Pliki mogą zajmować tyle miejsca, ile potrzebują, ale pamięć katalogowa jest przydzielana w oddzielnych blokach systemu plików. Ponieważ najmniejszy rozmiar bloku w większości systemów plików to 4096, katalogi podadzą ich rozmiar jako 4096.

Ta królicza nora sięga głębiej, ale uważamy, że informacje te nie będą wystarczająco przydatne w codziennej pracy inżyniera oprogramowania, aby omawianie ich tutaj było uzasadnione. Jeśli nadal jesteś ciekawy i chcesz zgłębić temat, poczytaj o i-nodach Linuksa.

Czas modyfikacji

```
-rw-r--r-- 1 root root 386 Feb 16 2023 os-release
```

Następnie widzimy znacznik czasu ostatniej modyfikacji pliku.

Nazwa pliku

```
-rw-r--r-- 1 root root 386 Feb 16 2023 os-release
```

Na końcu znajduje się nazwa pliku, będąca jedyną rzeczą, którą możesz zobaczyć, jeśli wykonasz polecenie zwykłego listingu zamiast długiego (`ls` zamiast `ls -l`). Zwykle będzie to po prostu zwykła nazwa pliku lub katalogu, z wyjątkiem przypadku dowiązania symbolicznego (symlinku) — wówczas zobaczysz nazwę dowiązania symbolicznego i ścieżkę pliku, do którego się ona odnosi.

Własność

Aby zmienić właściciela plików lub katalogów, użyj polecenia `chown` (ang. *change owner* — zmień właściciela). Składnia to `chown użytkownik: grupa ścieżka`, gdzie *użytkownik* jest nazwą użytkownika, a *grupa* jest nazwą grupy, podczas gdy *ścieżka* jest pełną lub względną częścią pliku lub katalogu.

Możesz pominąć dwukropek i grupę, aby zmienić użytkownika właściciela i pozostawić własność grupy. Oczywiście użytkownik próbujący zmienić uprawnienia do pliku musi mieć do tego uprawnienia, więc w większości sytuacji polecenie to będzie uruchamiane jako użytkownik `root`.

W poniższym fragmencie polecenia znajduje się długa lista plików należących do `roota`, których właściciel został zmieniony za pomocą polecenia `chown`:

```
bash-3.2$ ls -l mysecret.txt
-rw-r--r-- 1 root staff 0 Apr 12 15:39 mysecret.txt
bash-3.2$ sudo chown dave mysecret.txt
bash-3.2$ ls -l mysecret.txt
-rw-r--r-- 1 dave staff 0 Apr 12 15:39 mysecret.txt
```

Uprawnienia

Oto pojedynczy plik z naszej poprzedniej listy plików `ls -l`. Dostosowaliśmy uprawnienia, aby ten przykład był bardziej obrazowy:

```
rw-r-xr-x 1 root root 386 Aug 2 13:14 os-release
```

Przyjrzyj się w szczególności uprawnieniom:

```
rwxr-xr-x
```

Są one tutaj wyświetlane w trzech grupach po trzy osoby. Dla ułatwienia wyobraź sobie, że dzielą się one na trzy grupy:

```
rwX r-x r-x
```

Każda z tych trójek reprezentuje uprawnienia do odczytu (r), zapisu (w) i wykonania (x) dla określonego zestawu użytkowników na podstawie informacji o własności tego pliku dla użytkowników i grup. Jeśli zamiast litery widzisz znak -, to działanie (dla zbioru użytkowników, których dotyczy) jest niedozwolone. Przyjrzyjmy się temu bliżej:

1. Pierwsze trzy bity reprezentują uprawnienia właściciela pliku. W takim przypadku właściciel pliku (root) może odczytywać, zapisywać i wykonywać plik — `rwX`.
2. Kolejne trzy bity reprezentują uprawnienia właściciela grupy pliku, którym w tym przypadku jest również root. Uprawnienia w tym miejscu to `r-x`, czyli odczyt i wykonanie (bez zapisu!). Ale ponieważ root jest również użytkownikiem-właścicielem pliku, te uprawnienia mają pierwszeństwo, a więc root może zapisywać w tym pliku. Powodem, dla którego często widzisz takie uprawnienia, jest to, że właściciel grupy *musi* być ustawiony w plikach, a jeśli nie chcesz udostępniać pliku innym grupom, możesz po prostu użyć tutaj grupy właściciela. Uprawnienia grup dla pliku są zwykle niższe niż uprawnienia właściciela.
3. Ostatnie trzy bity reprezentują uprawnienia, które wszyscy inni użytkownicy systemu mają do tego pliku (czyli „cały świat”). Jest to prawie zawsze najbardziej restrykcyjny zestaw uprawnień, ponieważ większość plików nie musi być udostępniana nikomu poza właścicielem (a czasami osobnemu właścicielowi grupy). W tym przypadku mamy do czynienia ze współdzielonym plikiem biblioteki, który musi być dostępny dla wszystkich użytkowników systemu, więc uprawnienia to `r-x` (odczyt i wykonywanie, a zapis niedozwolony).

Zapisywanie uprawnień za pomocą liczb (ósemkowych)

„Odczyt”, „zapis” i „wykonywanie” to terminy, które ułatwiają zrozumienie uprawnień, ale istnieje inny ważny sposób reprezentowania uprawnień w Linuksie i Uniksie: za pomocą liczb ósemkowych.

Jako programista jesteś prawdopodobnie zaznajomiony z niedziesiątymi systemami liczbowymi. Jednym z nich jest system ósemkowy, oparty na liczbie 8, w przeciwieństwie do stosowanego przez nas zwykle systemu dziesiętnego, opartego na liczbie 10, oraz do innego systemu niedziesiątnego, mianowicie dwójkowego (binarnego), którego używają komputery.

Ponieważ istnieje tylko osiem możliwych stanów dla każdej trzybitowej kombinacji uprawnień, ósemka jest doskonale wydajnym systemem do ich reprezentowania.

Ponownie pomyśl o naszych 9 bitach jako 3-bitowych fragmentach. Każdy 3-bitowy fragment uprawnień może reprezentować liczbę ósemkową, do której wyrażenia wymagane są 3 bity (tabela 8.1). Te 9 bitów daje nam dokładnie tyle miejsca, ile potrzebujemy, aby reprezentować trzy liczby ósemkowe: jedną dla uprawnień użytkowników, drugą dla uprawnień grup i trzecią dla pozostałych uprawnień (zewnętrznych).

Tabela 8.1. Uprawnienia ósemkowe

Liczba ósemkowa	Liczba binarna	Znaczenie
0	0	Brak uprawnień (-)
1	1	Uprawnienia wykonywania (-x)
2	10	Uprawnienia zapisu (-w-)
3	11	Uprawnienia zapisu i wykonywania (-wx)
4	100	Uprawnienia odczytu (r-)
5	101	Uprawnienia odczytu i wykonywania (r-x)
6	110	Uprawnienia odczytu i zapisu (rw-)
7	111	Uprawnienia odczytu, zapisu i wykonywania (rwx)

Przekonasz się, iż jest to tak ułożone, że dodawanie ósemkowe działa: dodanie „odczytu” (4) i „wykonywania” (1) razem daje „odczyt i wykonywanie” (5).

Może się to wydawać dziwne i arbitralne (i takie jest!), ale szybko to zrozumiesz. Podczas pracy będziesz głównie używać uprawnień 7 (wszystkich), 6 (odczyt i zapis), 5 (odczyt i wykonywanie), 4 (odczyt) i 0 (brak uprawnień).

Typowe uprawnienia

Oto najczęstsze uprawnienia, z którymi będziesz się spotykać:

- `-rw-r--r--` (644) — właściciel może odczytywać i zapisywać, a wszyscy pozostali mogą tylko odczytywać.
- `-rwxr-xr-x` (755) — właściciel może robić wszystko, a wszyscy pozostali mogą odczytywać lub wykonywać plik. Jest to powszechnie stosowany zestaw uprawnień dla plików wykonywalnych, takich jak skrypty i pliki binarne, oraz domyślnie dla katalogów.
- `-rw-----` (600) — tylko właściciel może odczytywać i zapisywać, nikt inny nie ma żadnych uprawnień do pliku. Jest to powszechnie stosowany zestaw uprawnień w przypadku tajnych kluczy, plików zawierających hasła i innych poufnych informacji. SSH nie będzie na przykład używać kluczy, do których uprawnienia mają grupa lub wszyscy użytkownicy, dopóki nie zmienisz ich uprawnień w celu ich utajnienia.

Zmiana własności (chown) i uprawnień (chmod)

Do zmieniania własności i uprawnień plików będziesz używać dwóch poleceń: `chown` i `chmod`.

chown

Polecenie `chown` (ang. *change owner*) służy do zmiany właściciela i grupy pliku. Tego polecenia używa się w następujący sposób:

```
chown [OPCJA]... [WŁAŚCICIEL][:[GRUPA]] PLIK...
```

Mamy na przykład taki plik:

```
$ ls -lh testfile
-rw-r--r-- 1 dave dave 10 Aug 14 16:18 testfile
```

Zmiana właściciela

Zmieńmy właściciela na `chris` (zakładając, że `chris` jest użytkownikiem w systemie):

```
$ chown chris testfile
$ ls -lh testfile
-rw-r--r-- 1 chris dave 10 Aug 14 16:18 testfile
```

Zmiana właściciela i grupy

Zmieniliśmy właściciela, ale gdybyśmy chcieli zmienić również grupę, moglibyśmy to zrobić za pomocą poniższego polecenia:

```
$ chown chris:staff testfile
$ ls -lh testfile
-rw-r--r-- 1 chris staff 10 Aug 14 16:18 testfile
```

Rekurencyjne zmienianie właściciela i grupy

Jednym z typowych zadań jest zmiana właściciela i grupy dla wszystkich plików w danym katalogu. Możesz to zrobić za pomocą opcji `-R` lub `--recursive`:

```
$ chown -R dave:staff /home/dave
```

Spowoduje to rekurencyjne ustawienie własności dla `/home/dave/` oraz każdego znajdującego się w nim pliku i katalogu.

chmod

Polecenie `chmod` (ang. *change mode*) służy do zmiany uprawnień pliku. Możesz używać tutaj uprawnień zwykłych lub ósemkowych:

```
chmod [OPCJA]... TRYB[,TRYB]... PLIK...
chmod [OPCJA]... TRYB-ÓSEMKOWY PLIK...
```

Opcje możesz podawać w formie `ugo{+,-}rwx`:

1. `ugo` (ang. *user, group, other* — użytkownik, grupa, pozostali) — jeśli nie określisz inaczej, przyjmuje się założenie, że uprawnienia dotyczą wszystkich tych encji.
2. `+` dodaje uprawnienia, a `-` je usuwa.
3. `rwx` (odczyt, zapis, wykonywanie) — dowolna z tych liter lub wszystkie.

Aby dodać na przykład uprawnienia wykonywania dla użytkownika, który jest właścicielem pliku, należy wykonać następujące polecenie:

```
$ chmod u+x testfile
$ /tmp ls -lh testfile
-rwxr--r-- 1 dave dave 10 Aug 14 16:18 testfile
```

W celu dodania uprawnień do zapisu i wykonywania dla grupy i wszystkich pozostałych użytkowników wystarczy wpisać:

```
$ chmod go+wx testfile
$ /tmp ls -lh testfile
-rwxrwxrwx 1 dave dave 10 Aug 14 16:18 testfile
```

Ups, tak naprawdę chcieliśmy usunąć wszystkie uprawnienia dla pozostałych użytkowników:

```
$ chmod o-rwx testfile
$ /tmp ls -lh testfile
-rwxrwx--- 1 dave dave 10 Aug 14 16:18 testfile
```

Formatu ósemkowego dla uprawnień można używać w taki sposób:

```
$ chmod 744 testfile
$ /tmp ls -lh testfile
-rwxr--r-- 1 dave dave 10 Aug 14 16:18 testfile
```

Niech ten plik będzie tylko do odczytu:

```
$ chmod 400 testfile
$ /tmp ls -lh testfile
-r----- 1 dave dave 10 Aug 14 16:18 testfile
```

Korzystanie z referencji

Zarówno polecenie `chown`, jak i `chmod` pozwalają używać argumentu `-reference` — za jego pomocą możemy przekazać plik, z którego własność lub uprawnienia zostaną skopiowane.

Podsumowanie

W tym rozdziale omówiliśmy wszystko, co musisz wiedzieć, aby rozwiązywać najczęstsze problemy z uprawnieniami systemu Linux: wiesz już, jak wyświetlać uprawnienia plików i jak je modyfikować. Co ważniejsze, pokazaliśmy, jak traktować uprawnienia i jak są one powiązane z użytkownikami i grupami Linuksa, a z tym zagadnieniem wiele osób ma problemy.

Upewnij się, że dobrze rozumiesz materiał przedstawiony w tym rozdziale, ponieważ ogromny odsetek problemów, którymi będziesz się zajmować w trakcie swojej kariery, będzie dotyczyć własności i uprawnień. Na szczęście większość tych problemów wynika ze zwykłej niewiedzy, a Ty już doskonale rozumiesz to zagadnienie. Możesz zatem teraz z powodzeniem rozwiązać jakiś problem!

Zarządzanie zainstalowanym oprogramowaniem

Rozdział

9

Ponieważ pracujesz w różnych środowiskach linuksowych lub uniksowych, musisz dodawać lub usuwać oprogramowanie. Zwykle odbywa się to za pośrednictwem menedżerów pakietów, chociaż czasami trzeba korzystać z innych metod.

Prawdopodobnie znasz narzędzia, które zarządzają bibliotekami w Twoim środowisku programistycznym, takie jak `m.in. - npm, gem, pip, go get, maven i gradle`. Wszystkie te menedżery pakietów działają na tych samych zasadach, które mają zastosowanie w systemach Linux i Unix.

Menedżery pakietów oprogramowania tworzą warstwę abstrakcji dla wielu plików konfiguracyjnych i binarnych, które składają się na oprogramowanie, i pozwalają na pracę z jednym, schludnym pakietem. Powinno to być dla Ciebie znajome, jeśli masz doświadczenie w pracy systemami Windows (instalatory `.exe` i `.msi`) lub macOS (instalatory `.dmg`).

Ponadto większość menedżerów pakietów Linuksa dodaje do tego procesu warstwę bezpieczeństwa poprzez:

- użycie bezpiecznej warstwy transportowej (TLS) do pobierania;
- użycie podpisu kryptograficznego na samych pakietach, aby udowodnić, że autorzy są co najmniej tymi, za których się podają (niezależnie od tego, czy w sposób dorozumiany im ufasz, czy nie).

Różne dystrybucje Linuksa zapoczątkowały również koncepcje przeszukiwalnych repozytoriów pakietów, które pomagają użytkownikom znaleźć oprogramowanie do pobrania, co stało się inspiracją dla istniejących obecnie sklepów z aplikacjami Apple'a i Microsoftu.

W tym rozdziale omawiamy następujące zagadnienia:

- czym są menedżery pakietów;
- najbardziej popularne menedżery pakietów;
- najważniejsze operacje zarządzania pakietami i poszczególne polecenia potrzebne do ich wykonywania, „przetłumaczone” na menedżery pakietów — to 90% tego, co musisz wiedzieć i stosować w praktyce;
- popularna procedura pobierania i wykonywania niestandardowych skryptów instalacyjnych;

- krótkie praktyczne wprowadzenie do samodzielnego lokalnego kompilowania i instalowania oprogramowania.

Jeśli szukasz instrukcji dotyczących instalowania oprogramowania w Dockerze, zapoznaj się z rozdziałem 15. „Konteneryzacja aplikacji za pomocą Dockera”.

Jako programista będziesz sięgać po polecenia zarządzania pakietami przede wszystkim wtedy, gdy będziesz chciał robić kilka typowych rzeczy. Oto one:

- instalowanie nowych pakietów oprogramowania, na przykład zależności, których dostępności Twoja aplikacja oczekuje w swoim środowisku wykonawczym;
- sprawdzanie zainstalowanych pakietów (na przykład czy serwer WWW nginx jest już zainstalowany w danym systemie);
- aktualizowanie obecnie zainstalowanego zestawu pakietów, aby zawsze dysponować najnowszymi dostępnymi wersjami (jest to powszechne w przypadku usuwania wykrytych luk w zabezpieczeniach lub zapewniania, aby wszystkie funkcje oprogramowania były najnowsze);
- usuwanie pakietu, czyli odinstalowywanie go z systemu.

Przejdźmy do tego, jak osiągnąć te praktyczne cele, i zobaczmy odpowiednie polecenia, z których będziesz korzystać.

Praca z pakietami oprogramowania

Zanim omówimy praktyczne polecenia, powinieneś wiedzieć, że różnią się one w zależności od dystrybucji Uniksa (lub Linuksa), z której korzystasz. Poszczególne dystrybucje Linuksa używają różnych menedżerów pakietów i chociaż ich składnie irytująco się różnią, wszystkie działają niemalże identycznie. Oto typowe menedżery pakietów:

- homebrew (macOS);
- apt (można go znaleźć w systemach opartych na Ubuntu i Debianie, nawet tych minimalnych, które nie mają zainstalowanych wszystkich narzędzi);
- pacman (Arch);
- apk (Alpine).

W kolejnych praktycznych punktach tego podrozdziału przedstawimy ogólny cel, który chcemy osiągnąć (na przykład zainstalowanie konkretnego pakietu), a następnie pokażemy polecenia, które wykonują to zadanie, korzystając z popularnych menedżerów pakietów, które właśnie wymieniliśmy.

Aktualizowanie lokalnej pamięci podręcznej stanem repozytorium

Przed zainstalowaniem lub usunięciem pakietów należy upewnić się, że ich lokalna pamięć podręczna (systemowy zapis pakietów dostępnych w internecie) jest aktualna. Jeśli chcesz zainstalować na przykład serwer WWW `nginx`, ale ostatni raz ktoś aktualizował lokalną pamięć podręczną pakietów miesiąc temu, możesz nieumyślnie zainstalować nieaktualną wersję z zeszłego miesiąca, zamiast najnowszej wersji z tego tygodnia.

Aby zaktualizować pamięć podręczną, wybierz swój menedżer pakietów z poniższej tabeli i uruchom odpowiednie polecenie.

Menedżer pakietów	Polecenia
<code>homebrew</code>	<code>brew update</code>
<code>apt</code>	<code>apt update</code>
<code>pacman</code>	<code>pacman -Sy</code>
<code>apk</code>	<code>apk update</code>

Twoja lokalna pamięć podręczna dostępnych pakietów zostanie zaktualizowana i będziesz mógł przejść do ekscytującej pracy polegającej na wyszukiwaniu i instalowaniu pakietów.

Wyszukiwanie pakietu

Nie wszystkie pakiety mają taką samą nazwę jak oprogramowanie, które zawierają. Firefox może być dostępny w pakiecie `firefox`, ale możesz być rozczarowany, jeśli spróbujesz zainstalować `ag` (nazwa pakietu to `silversearcher-ag`). Opis pakietu, którego instalację rozważasz, możesz wyszukać, korzystając z poleceń menedżera pakietów pokazanych w poniższej tabeli.

Menedżer pakietów	Polecenia
<code>homebrew</code>	<code>brew search \$NAZWA_PAKIETU</code>
<code>apt</code>	<code>apt-cache search \$NAZWA_PAKIETU</code>
<code>pacman</code>	<code>pacman -Ss \$NAZWA_PAKIETU</code>
<code>apk</code>	<code>apk search \$NAZWA_PAKIETU</code>

Jest to dobry sposób, aby potwierdzić, że otrzymujesz to, czego oczekujesz, ale może być stosowany do rozszerzenia poszukiwań i wyszukiwania ogólnego oprogramowania związanego z danym problemem. Na przykład w systemie Ubuntu za pomocą polecenia `apt-cache search grep` można wyszukać narzędzia podobne do `grep`. Zostanie wyświetlony każdy pakiet zawierający `grep` w nazwie lub opisie.

Instalowanie pakietu

Wreszcie najważniejsze! Teraz, gdy pamięć podręczna repozytorium jest aktualna i wiemy dokładnie, który pakiet chcemy zainstalować, uruchommy polecenie `install`.

Menedżer pakietów	Polecenia
homebrew	<code>brew install \$NAZWA_PAKIETU</code>
apt	<code>apt install \$NAZWA_PAKIETU</code>
pacman	<code>pacman -Sy \$NAZWA_PAKIETU</code>
apk	<code>apk add \$NAZWA_PAKIETU</code>

Jeśli menedżer pakietów poprosi Cię o potwierdzenie, odpowiedz na monit, a pakiet zostanie zainstalowany (wraz z innymi pakietami jako jego zależnościami).

Ponieważ niektóre z tych poleceń przed zainstalowaniem pakietu wyświetlają monit z prośbą o potwierdzenie, mogą one zablokować skrypt. Kiedy automatyzujesz zadania za pomocą skryptów i musisz zainstalować pakiet, przeczytaj odpowiednią stronę podręcznika menedżera pakietów, aby dowiedzieć się, jak instalować pakiety w sposób nieinteraktywny. Jest to często wykonywane za pomocą zmiennej środowiskowej lub dodatkowego argumentu polecenia.

Uaktualnianie wszystkich pakietów, które mają dostępne aktualizacje

W długo pracującym systemie będziesz chcieć od czasu do czasu uaktualnić zainstalowane pakiety do ich najnowszych wersji. Pozwala to naprawić znane luki w zabezpieczeniach, uzyskać najnowsze funkcje i zapobiec rozbieżności stanu różnych systemów z powodu tego, że zostały uruchomione kilka miesięcy przed innym systemem lub po nim.

Menedżer pakietów	Polecenia
homebrew	<code>brew upgrade</code>
apt	<code>apt dist-upgrade</code>
pacman	<code>pacman -Syu</code>
apk	<code>apk upgrade</code>

Te polecenia również będą monitować o potwierdzenie, dlatego jeśli używasz ich w skryptach, skorzystaj z tej samej porady, która dotyczy dodawania opcji, aby uczynić ten proces nieinteraktywnym. Na przykład `apt -y dist-upgrade` nie będzie czekać na ręczne potwierdzenie i po prostu wykona aktualizację.

Usuwanie pakietu (i jego wszelkich zależności, pod warunkiem że nie są wykorzystywane przez inne pakiety)

Bywa, że z różnych powodów chcesz odinstalować pakiet: bo jedynie go testowałeś, wymagania aplikacji uległy zmianie albo wiadomo, że jest podatny na ataki i w najbliższej przyszłości nie ma widoków na poprawki. Wszystkie menedżery pakietów mają polecenie do usuwania zainstalowanego pakietu.

Menedżer pakietów	Polecenia
homebrew	<code>brew remove \$NAZWA_PAKIETU</code>
apt	<code>apt remove \$NAZWA_PAKIETU</code>
pacman	<code>pacman -Rs \$NAZWA_PAKIETU</code>
apk	<code>apk del \$NAZWA_PAKIETU</code>

Jedną z rzeczy, które należy zrobić przed usunięciem lub po usunięciu pakietu, jest sprawdzenie, czy pakiet jest w ogóle zainstalowany. Przyjrzyjmy się teraz, jak to zrobić.

Kwerendowanie zainstalowanych pakietów

Jeśli chcesz wyświetlić listę wszystkich pakietów, które są obecnie zainstalowane w systemie, możesz to zrobić za pomocą jednego polecenia.

Menedżer pakietów	Polecenia
homebrew	<code>brew list</code>
apt	<code>dpkg -l</code>
pacman	<code>pacman -Qi</code>
apk	<code>apk info</code>

Ponieważ ta lista obejmuje często setki lub tysiące pakietów, jest trochę nieporęczna. Należy zawęzić dane wyjściowe, przekierowując je do polecenia wyszukiwania, takiego jak `grep`, aby znaleźć dokładnie to, czego szukasz:

```
dpkg -l | grep silversearcher
ii silversearcher-ag 2.2.0+git20200805-1 arm64 very fast grep-like program,
↳ alternative to ack-grep
```

Jeśli nie do końca rozumiesz, w jaki sposób używamy tego potoku do podawania danych wyjściowych `dpkg` do polecenia `grep`, zajrzyj do rozdziału 1. „Jak działa wiersz poleceń?”, aby uzyskać informacje na temat podstaw łączenia poleceń za pomocą znaku potoku (`|`). Temu zagadnieniu przyjrzymy się także znacznie bliżej w rozdziale 11. „Potoki i przekierowanie”.

Skoro omówiliśmy już podstawowe polecenia zarządzania pakietami, pora zademonstrować kilka wzorców przydatnych, gdy nie będzie dostępnego gotowego pakietu dla oprogramowania, które chcesz zainstalować.

Wymagana ostrożność — curl | bash

Czasami nie znajdziesz gotowego pakietu. I nie ma w tym nic złego! Wiele źródeł internetowych — nawet godnych zaufania i popularnych, takich jak homebrew dla systemu macOS — zaleca proces instalacji wiersza poleceń, który wygląda tak:

```
curl $JAKIŚ_URL | bash
```

Wykorzystuje on polecenie `curl` do pobierania zawartości z sieci, a następnie stosuje tę zawartość jako dane wejściowe (`|`, znak potoku, który omówiliśmy w rozdziale 1. „Jak działa wiersz poleceń?”) do uruchomienia powłoki Bash. Kiedy to zrobisz, zasadniczo uruchomisz skrypt w sieci, a nie jako plik lokalny. Może to być bardzo wygodny sposób instalacji oprogramowania, ale upewnij się *bezwzględnie*, że pochodzi z wiarygodnego źródła.

Zalecamy, by zawsze przynajmniej *zajrzeć* do źródła skryptu, które można zobaczyć w przeglądarce, odwiedzając adres URL skryptu dla tego polecenia (reprezentowany w poniższym przykładzie jako `$JAKIŚ_URL`); ewentualnie możesz podzielić polecenie `curl $URL | bash` na kilka poleceń, co pozwoli Ci wykonać następujące działania:

- pobranie skryptu;
- przeczytanie go w lokalnym edytorze tekstu, aby sprawdzić, czy nie robi nic złośliwego, i opcjonalnie edycja skryptu, aby spełniał Twoje wymagania;
- uruchomienie skryptu po zweryfikowaniu, że robi tylko to, co powinien.

Aby podzielić coś takiego jak ten wzorec `curl $URL | bash` na wiele poleceń, wykonaj następujące kroki:

```
# Pobranie instalatora i nazwanie wynikowego pliku installer.sh
curl $JAKIŚ_URL -o installer.sh
# Odczyt i ewentualna modyfikacja przy użyciu edytora tekstu, takiego jak vim
vim installer.sh
# Uczynienie skryptu wykonywalnym i uruchomienie go
chmod +x installer.sh
./installer.sh
```

Dzieląc polecenie na wiele kroków, zamiast pobierać niezauwany skrypt i natychmiast go uruchamiać, dajemy sobie czas na sprawdzenie kodu, który mamy zamiar wykonać, i zweryfikowanie, czy jest on bezpieczny. Efekt końcowy jest taki sam (skrypt instalacyjny działa), ale takie podejście daje znacznie większą kontrolę i nie wymaga ślepego zaufania.

Istnieje inny sposób na zainstalowanie oprogramowania w systemie, nawet jeśli nie ma wcześniej napisanego skryptu instalacyjnego.

Kompilowanie zewnętrznego oprogramowania ze źródła

Jest to najbardziej czasochłonny sposób instalacji oprogramowania w systemie: ręczna kompilacja i instalacja. Nie ma on wielu zalet menedżera pakietów, takich jak szybkość, powtarzalność, łatwość zarządzania zainstalowanym oprogramowaniem oraz walidacja kryptograficzna instalowanego oprogramowania binarnego.

Jednak jest to najbardziej niezawodny sposób na zainstalowanie oprogramowania, bez rzeczywistych zewnętrznych zależności, z wyjątkiem podstawowych narzędzi programowych (kompilatora, linkera i programu make dla skryptu), które jako programista zapewne już znasz.

Oto sytuacje, w których będziesz ręcznie kompilować i instalować oprogramowanie:

- W menedżerze pakietów nie ma wstępnie zapakowanej wersji oprogramowania. Na przykład jeżeli używasz minimalnej dystrybucji kontenerów (np. Alpine), w menedżerze pakietów możesz nie znaleźć tego, czego potrzebujesz. W takim przypadku możesz skompilować własny plik binarny ze źródła i w ten sposób wprowadzić go do obrazu kontenera.
- Musisz dodać do kontenera Dockera własne (lub inne niestandardowe) oprogramowanie.
- Potrzebujesz absolutnie najnowszej, najnowocześniejszej wersji oprogramowania, dla którego wersja pakietowa jeszcze nie istnieje. Może to mieć miejsce w przypadku wolniejszych projektów, które nie zawsze mają nowe pakiety, lub w sytuacjach, gdy poprawka dla krytycznej luki w zabezpieczeniach musi zostać wdrożona *natychmiast*, zanim przejdzie proces pakowania.

Proces ten obejmuje kilka kroków, z niewielkimi zmianami w zależności od oprogramowania:

1. Uruchomienie polecenia `curl` lub `wget`, aby pobrać archiwum skompresowanego oprogramowania.
2. Uruchomienie polecenia `tar xzf nazwa_pobierania` lub `unzip nazwa_pobierania`, aby utworzyć archiwum i zdekompresować pobrany katalog kodu źródłowego.
3. Zmiana katalogu na pobrany katalog źródłowy i odczytanie wszystkich dołączonych plików *README*. W tym miejscu znajdziesz informacje o dokładnym procesie, który musisz wykonać, aby skompilować oprogramowanie, a także o wszelkich odchyleniach od norm, które tutaj opisujemy.
4. Uruchomienie `./configure`, następnie `make`, a potem `sudo make install`, aby zbudować i zainstalować plik binarny.

Podobnie jak w przypadku innych sposobów instalacji oprogramowania, ręcznie lub za pośrednictwem menedżera pakietów, należy pamiętać, że *zgodnie z założeniem* `configure` i `make` wykonują dowolny kod. Oznacza to, że uruchomienie polecenia `make install` jako root spowoduje, że cały ten kod zostanie uruchomiony w uprawnieniach roota. To powinno Cię niepokoić. Upewnij się zatem, że kod źródłowy oprogramowania jest godny zaufania i że pobierasz go z wiarygodnego źródła.

Przykład: kompilowanie i instalowanie narzędzia htop

Aby uruchomić ten prawdziwy przykład, będziemy pobierać, kompilować i instalować htop, który jest niewielkim, lecz niezwykle użytecznym narzędziem do monitorowania systemu (podobnym do wbudowanego polecenia top, ale znacznie lepszym). Dla przypomnienia: jest on łatwo dostępny za pośrednictwem niemal każdego menedżera pakietów dystrybucji Linuksa, ale zamierzamy udawać, że jest to trudny do znalezienia, niestandardowy program, który nie jest szeroko rozpowszechniany przez menedżery pakietów.

System, w którym to robimy, to serwer Ubuntu 22.04 Linux, więc jeśli chcesz przeciwieć ten przykład bez samodzielnego rozwiązywania problemów, skorzystaj z tego samego systemu.

Najpierw sprawdzamy najnowsze wydanie z oficjalnego repozytorium GitHuba na stronie <https://github.com/htop-dev/htop/releases> — w chwili gdy piszemy ten rozdział, jest to wersja 3.2.2.

Następnie tworzymy katalog dla tej kompilacji, aby zachować porządek — polecam coś w rodzaju `/tmp`, który przechowuje pliki tymczasowe, a jego zawartość jest usuwana przy każdym uruchomieniu systemu:

```
mkdir /tmp/htopbuild && cd /tmp/htopbuild
```

W ten sposób możemy usunąć wszystko po zakończeniu kompilacji, aby zapobiec zaśmiecaniu systemu niepotrzebnymi plikami ze starych kompilacji. Teraz jesteśmy gotowi, aby zacząć ten proces.

Instalowanie wymagań wstępnych

Najpierw musimy zainstalować podstawowy łańcuch narzędzi programistycznych języka C (kompilator, linker, make i inne narzędzia — wszystko, czego potrzebujemy do skompilowania kodu w C w systemie Linux). W Ubuntu osiąga się to poprzez zainstalowanie **metapakietu** `build-essential`, który jest rodzajem aliasu dla wielu innych pakietów:

```
sudo apt install build-essential
```

Zainstalujemy również kilka innych narzędzi, których będziemy używać: `wget` do pobierania plików z sieci i bibliotekę `ncurses-dev`, wykorzystywaną przez htop do tworzenia responsywnego interfejsu wiersza poleceń:

```
sudo apt install wget libncurses-dev
```

Pobieranie, weryfikowanie i rozpakowywanie kodu źródłowego

Najpierw pobierzemy kod źródłowy i zweryfikujemy jego podpis kryptograficzny, aby upewnić się, że jest to prawdziwa wersja podpisana kluczem programisty:

```
wget https://github.com/htop-dev/htop/releases/download/3.2.2/  
↳ htop-3.2.2.tar.xz
```

Daje nam to skompresowany, zarchiwizowany katalog kodu źródłowego, który będziemy kompilować do postaci binarnej.

Teraz upewnijmy się, że mamy wydanie uznane przez dewelopera, poprzez sprawdzenie podpisu, którym jest po prostu skrót sha256 kodu źródłowego.

Pobieramy plik zawierający oczekiwany skrót dla tej wersji i wypisujemy go w terminalu:

```
wget https://github.com/htop-dev/htop/releases/download/3.2.2/  
↳ htop-3.2.2.tar.xz.sha256  
cat htop-3.2.2.tar.xz.sha256
```

Jeżeli pracujesz z tą samą wersją co użyta w tym przykładzie, zobaczysz ten oto skrót:

```
bac9e9ab7198256b8802d2e3b327a54804dc2a19b77a5f103645b11c12473dc8  
↳ htop-3.2.2.tar.xz
```

Teraz wypiszemy skrót kodu źródłowego, który pobraliśmy, aby za pomocą narzędzia sha256sum sprawdzić, czy te skróty są zgodne:

```
sha256sum htop-3.2.2.tar.xz  
bac9e9ab7198256b8802d2e3b327a54804dc2a19b77a5f103645b11c12473dc8 htop-  
3.2.2.tar.xz
```

Wspaniale! Teraz wiemy, że oprogramowanie, które posiadamy, jest takie samo jak oficjalne wydanie, które zamierzaliśmy pobrać. Rozpakujmy katalog kodu źródłowego i przeńśmy się do niego:

```
tar xf htop-3.2.2.tar.xz  
cd htop-3.2.2
```

Jeśli chcesz, możesz przeczytać plik *README* (ogólne informacje o programie) oraz plik *INSTALL* (instrukcje, jak skompilować i zainstalować program).

Teraz możemy rozpocząć konfigurowanie i kompilowanie tego oprogramowania!

Konfigurowanie i kompilowanie narzędzia htop

W katalogu kodu źródłowego uruchamiamy skrypt `./configure`. Powoduje on zainstalowanie wszelkich zależności wymaganych do kompilacji (współdzielonych bibliotek, narzędzi itp.) i skonfigurowanie elementów dla kompilacji, którą mamy zamiar wykonać:

```
./configure
```

Podczas uruchamiania tego skryptu zostaną wygenerowane dane wyjściowe, a także nastąpi sprawdzenie różnych zależności i zweryfikowanie, czy środowisko ma wszystko, co potrzebne do przeprowadzenia kompilacji.

Jeśli ten skrypt powoduje błędy, przeczytaj uważnie komunikat — zazwyczaj jasno informuje on, co jest nie tak (np. brakująca biblioteka lub problematyczne ustawienie systemu operacyjnego). Po naprawieniu wszelkich problemów uruchom skrypt ponownie. Jeśli tym razem uruchomi się pomyślnie, możesz skompilować plik binarny htop:

```
make
```

Spowoduje to ponowne wygenerowanie sporych ilości danych wyjściowych podczas uruchamiania skryptu kompilacji. Jeżeli narzędzie `makefiles` stanowi dla Ciebie nowość,

wyjaśniamy, że jest ono niezwykle użyteczne do automatyzacji i szeroko stosowane przez programistów. Oto doskonały samouczek na ten temat: <https://makefiletutorial.com/>.

Po zakończeniu kompilacji możemy zainstalować utworzony przez nas właśnie plik binarny `htop` (będzie się on znajdował w głównym katalogu źródłowym o nazwie `htop`). Zazwyczaj można to zrobić w zautomatyzowany sposób:

```
sudo make install
```

Polecenie `sudo` jest wymagane, ponieważ przenosisz skompilowany plik binarny do chronionej (należącej do `roota`) lokalizacji. Następnie możesz sprawdzić, czy `htop` jest zainstalowany i działa — w tym celu wpisz:

```
htop
```

Powinieneś zobaczyć piękny interfejs użytkownika oparty na terminalu (dzięki bibliotece `ncurses`), pokazujący bieżące obciążenie procesora, wykorzystanie pamięci i listę procesów systemu.

W przypadku programów, które nie są dostarczane z pełnoprawnym poleceniem `install`, pamiętaj, że w Linuksie nie ma żadnej magii i po prostu zainstaluj plik binarny, przenosząc go do katalogu `/usr/local/bin/`, gdzie przechowywane są lokalnie skompilowane pliki binarne:

```
mv htop /usr/local/bin/
```

Widząc, jak prosty może być ten proces, masz teraz całą wiedzę, której potrzebujesz, aby iść do przodu i kompilować!

Podsumowanie

W tym rozdziale przedstawiliśmy podstawy zarządzania oprogramowaniem zainstalowanym w środowisku Linuksa. Przyjrzelśmy się przede wszystkim, jak to osiągnąć w prosty sposób, czyli przez zarządzanie oprogramowaniem za pośrednictwem menedżerów pakietów, które najprawdopodobniej napotkasz na swojej drodze. Chociaż to pierwsze podejście powinno spełniać większość Twoich potrzeb, pokazaliśmy Ci także procedury, które będziesz musiał zastosować w niektórych sytuacjach — dokładną weryfikację pobranego oprogramowania, a następnie użycie niestandardowych skryptów instalacyjnych lub przeprowadzenie ręcznej kompilacji i instalacji.

Mamy nadzieję, że śledziłeś praktyczny przykład kompilacji i wypróbowałeś monitor systemu `htop`. Na szczęście `htop` jest dostępny wszędzie za pośrednictwem menedżerów pakietów — jest to naprawdę przydatne narzędzie, które wielu administratorów systemów uważa za bezcenne w przypadku długo działających systemów produkcyjnych.

Teraz już znasz ogólne koncepcje i praktyczne polecenia, których będziesz potrzebować, aby efektywnie korzystać z wielu systemów Unix i Linux, zarówno w środowiskach programistycznych, jak i produkcyjnych.

Konfigurowanie oprogramowania

Rozdział 10

Prędzej czy później każdy musi skonfigurować jakieś oprogramowanie w systemie Linux. I chociaż dostępnych jest wiele sposobów, by to zrobić, na szczęście istnieje ogólny wzorec, którym można się posłużyć w celu uzyskania pożądaných wyników. W systemie Linux jest to szczególnie powszechne; ponieważ większość standardowych narzędzi w tym systemie jest zgodna z filozofią „niewielkich, ostrych narzędzi”, mamy do dyspozycji wiele małych, lecz potężnych programów, które zapewniają elastyczność dzięki obsłudze rozbudowanej konfiguracji.

W tym rozdziale przyjrzymy się hierarchii konfiguracji, z której korzystają dobrze zaprojektowane programy. Niezależnie od tego, czy musisz sprawdzić stronę podręcznika, aby znaleźć argument wiersza poleceń dla pojedynczego polecenia, czy chcesz ustawić zmienną środowiskową, która dotyczy wszystkich poleceń uruchamianych w powłocie, dowiesz się, jak to zrobić.

Potem pokażemy ogólną hierarchię konfiguracji, z której korzysta prawie całe oprogramowanie uniksowe, dzięki czemu zawsze będziesz wiedzieć, gdzie sprawdzić, dlaczego program nie zachowuje się zgodnie z oczekiwaniami opartymi na podanej mu konfiguracji.

Na koniec zobaczysz, jak ta konfiguracja przekłada się na programy zarządzane przez systemd, czyli najpopularniejsze narzędzie do zarządzania usługami w systemie Linux.

W tym rozdziale omawiamy następujące zagadnienia:

- hierarchia konfiguracji;
- argumenty wiersza poleceń;
- zmienne środowiskowe;
- pliki konfiguracyjne;
- konfiguracja w Dockerze.

Hierarchia konfiguracji

Jedną z pierwszych rzeczy, które będziesz robić podczas uruchamiania programów w systemie Linux, będzie dostosowywanie ich do konkretnych potrzeb. I już to robiłeś: przekazując argumenty do poleceń takich jak `ls` czy `grep`, zmieniałeś sposób zachowania tych programów.

Prawdopodobnie masz pewne wyobrażenie o tym, jak powinno to działać, ponieważ całe życie korzystasz z oprogramowania. Na przykład może się wydawać naturalne, że podawanie argumentów wiersza poleceń nadpisuje domyślne ustawienia programu: `ls -l` generuje dane wyjściowe, które różnią się od domyślnego wyjścia z polecenia `ls`.

Spróbujmy teraz nieco bardziej zgłębić tę kwestię i sprawdźmy, czy możemy zmapo- wać pewne heurystyki dotyczące tego, jak *zazwyczaj* działa konfiguracja w środowisku Uniksa. Jedną z norm, do których stosuje się większość standardowych programów wiersza poleceń systemu Unix, jest specyficzna hierarchia konfiguracji, w której wcześniejsze wartości są nadpisywane późniejszymi. Jeśli napisałeś oprogramowanie obsługujące konfigurację użytkownika, być może wcześniej utworzyłeś hierarchię priorytetów. Może ona wyglądać tak:

1. Ustawienie dla wartości konfigurowalnych wbudowanych ustawień domyślnych.
2. Sprawdzenie wartości przekazywanych przez pliki konfiguracyjne powoduje nadpisanie tych wartości domyślnych.
3. Sprawdzenie zmiennych środowiskowych (określanych jako `env vars`) powoduje nadpisanie plików konfiguracyjnych i wcześniejszych wartości.
4. Sprawdzenie argumentów interfejsu wiersza polecenia (określanych jako *CLI args*) i w razie potrzeby zaktualizowanie wartości powoduje nadpisanie wcześniejszych wartości.

Każdy kolejny poziom jest bliższy użytkownikowi uruchamiającemu oprogramowanie w określonym momencie, a więc każdy kolejny poziom ma pierwszeństwo przed poprzednim.

Jeżeli oprogramowanie wykryje na przykład sprzeczne wartości w pliku konfiguracyjnym i argumentach interfejsu wiersza poleceń, z którymi program został uruchomiony, powinno preferować wartości z argumentów wiersza poleceń. **Innymi słowy, wartości znalezione bliżej wywołania programu „przysłaniają”** (w sensie nadpisywania lub zastępowania) **wartości znajdujące się dalej od wykonywania**. Wartość argumentu CLI zastępuje wartość pliku konfiguracyjnego, ponieważ plik konfiguracyjny znajduje się dalej od punktu wywołania niż argumenty przekazywane do programu podczas uruchamiania. To powinno mieć intuicyjny sens: nie możesz polegać na oprogramowaniu, jeśli ignoruje flagi wiersza poleceń na korzyść domyślnych ustawień programu. Polecenie `ls -l` nie powinno generować takich samych danych wyjściowych co `ls`.

Większość oprogramowania w systemie Linux przestrzega tej hierarchii, gdy istnieje wiele sposobów konfiguracji danego programu. Należy pamiętać, że nie wszystkie programy wykorzystują wszystkie ścieżki konfiguracyjne, które pokażemy tutaj jako przykłady, i nie każde oprogramowanie dokładnie respektuje tę kolejność konfiguracji.

Przjrzyjmy się jeszcze raz tej hierarchii, ale tym razem w kontekście praktycznych, konkretnych przykładów programu serwera WWW **nginx**. Prawdopodobnie w pewnym momencie swojej kariery będziesz pracować z serwerem **nginx**, jest to bowiem jeden z najpopularniejszych serwerów WWW, używany do obsługi wszelkiego rodzaju dynamicznych aplikacji internetowych. Zobaczmy, jak poszczególne części hierarchii priorytetów, którą właśnie omówiliśmy, mapują się na praktyczną konfigurację serwera **nginx**:

1. **Wbudowane ustawienia domyślne** — nginx ma domyślnie zakodowanego na stałe użytkownika nobody i po uruchomieniu działa jako ten użytkownik.
2. **Globalne pliki konfiguracyjne** mogą to zmienić dla wszystkich procesów serwera nginx, dlatego w lokalizacji `/etc/nginx/nginx.conf` często można znaleźć globalny plik konfiguracyjny tego programu z wartością `"user www;"`, która instruuje nginx, aby uruchamiał się jako użytkownik WWW.
3. **Pliki konfiguracyjne na poziomie użytkownika** to zazwyczaj „pliki kropkowe” (pliki o nazwie zaczynającej się od znaku kropki (`.`), który wyklucza je ze zwykłych listingów `ls`) znajdujące się w katalogu `/home` użytkownika. Na przykład `/home/dave/.bashrc` jest lokalizacją dla konfiguracji Basha charakterystycznej dla użytkownika. Serwer nginx to długo działający proces, który zwykle nie jest uruchamiany jako zwykły użytkownik Linuksa, ale ma coś takiego: poszczególne witryny są często konfigurowane we własnych, oddzielnych plikach konfiguracyjnych przechowywanych w lokalizacji `/etc/nginx/conf.d/yourwebsite.conf`. Zazwyczaj dziedziczą one wartości z konfiguracji globalnej z poprzedniego poziomu.
4. **Zmienne środowiskowe** — nginx pobiera informacje o strefie czasowej ze zmiennej środowiskowej o nazwie `TZ`.
5. **Argumenty wiersza poleceń** są określane podczas uruchamiania oprogramowania ręcznie lub automatycznie (na przykład za pomocą zadań `crona` lub plików jednostkowych). Podczas debugowania problemu przyjrzyj się tym możliwym zewnętrznym źródłom argumentów wiersza poleceń — są one częstymi winowajcami, gdy widzisz rozbieżność między zachowaniem programu a plikami konfiguracyjnymi. Serwer nginx przyjmuje różne argumenty wiersza poleceń, które modyfikują jego zachowanie: od nadpisania plików konfiguracyjnych, których będzie używał, po całkowite uniemożliwienie jego działania jako serwera WWW i zamiast tego sygnalizowanie zatrzymania lub ponownego załadowania uruchomionego już procesu nginx.

Skoro zobaczyłeś zarówno teoretyczne, jak i praktyczne aspekty interakcji tej hierarchii konfiguracji ze wszystkimi programami uruchamianymi w systemie Linux i wszystkimi programami, które możesz dla niego *napisać*, przeanalizujmy krok po kroku hierarchię konfiguracji, aby przyjrzeć się bliżej każdemu poziomowi. Zaczniemy od najbardziej bezpośredniej i potężnej formy konfiguracji, która nadpisuje wszystko inne, czyli od przekazywania argumentów wiersza poleceń w momencie wywoływania programu.

Argumenty wiersza poleceń

Znasz już najczęstszy sposób konfigurowania programów: argumenty wiersza poleceń. Konfiguruje one program w momencie, gdy jest on wywoływany jako polecenie powłoki.

Aby znaleźć prawidłowe argumenty wiersza poleceń dla programu, zacznij od strony podręcznika dla danego polecenia. Z wyjątkiem najbardziej minimalnych systemów oprogramowanie uniksowe jest dostarczane ze stronami podręcznika, które dokumentują

większość programów, objaśniają dostępne flagi i — zwykle na końcu — wymieniają inne rodzaje metod konfiguracji, takie jak pliki konfiguracyjne.

Na początek przyjrzyjmy się zawartości strony podręcznika dla polecenia `find`:

```
man find
FIND(1)
General Commands Manual
FIND(1)
```

NAME

`find` - walk a file hierarchy

SYNOPSIS

```
find [-H | -L | -P] [-EXdsx] [-f path] path ... [expression]
find [-H | -L | -P] [-EXdsx] -f path [path ...] [expression]
```

DESCRIPTION

The `find` utility recursively descends the directory tree for each path listed, evaluating an expression (composed of the "primaries" and "operands" listed below) in terms of each file in the tree.

The options are as follows:

- E Interpret regular expressions followed by `-regex` and `-iregex` primaries as extended (modern) regular expressions rather than basic regular expressions (BRE's). The `re_format(7)` manual page fully describes both formats.
- H Cause the file information and file type (see `stat(2)`) returned for each symbolic link specified on the command line to be those of the file referenced by the link, not the link itself. If the referenced file does not exist, the file information and type will be for the link itself. File information of all symbolic links not on the command line is that of the link itself.

Widać, że większość tej strony podręcznika dokumentuje różne argumenty wiersza poleceń, które są dostępne podczas uruchamiania polecenia `find`. Od pierwszego rozdziału używaliśmy wielu argumentów wiersza poleceń, więc wszystko powinno być znajome.

Przyjrzyjmy się teraz następnemu, nieco bardziej odległemu rodzajowi konfiguracji: zmiennym środowiskowym.

Zmienne środowiskowe

Chociaż argument wiersza poleceń jest potężny, ma zastosowanie tylko do pojedynczego wywołania programu, którego jest częścią. Po wpisaniu `ls -l` tylko jedno polecenie `ls` będzie miało długie dane wyjściowe. A jeśli chcesz, aby wartość konfiguracji utrzymywała się przez wiele wywołań polecenia? Jest to przydatne, kiedy piszesz na przykład skrypt, który instaluje pakiety w kilku różnych punktach, i chcesz ustawić opcję konfiguracji *raz*, zamiast dodawać ją w kółko jako argument wiersza poleceń za każdym razem, gdy uruchamiasz polecenie instalacji pakietu. Właśnie w takich sytuacjach przydają się zmienne środowiskowe.

Jako programista piszący wszelkiego rodzaju oprogramowanie prawdopodobnie masz świadomość istnienia zmiennych środowiskowych: wartości powłoki analogicznych do zmiennych w dowolnym innym języku programowania. Różnią się one od argumentów wiersza poleceń, ponieważ działają o jeden poziom wyżej. Zmienne środowiskowe zapewniają większą dzwignię: po ustawieniu w powłoce zmiennej konfiguracyjnej ma ona zastosowanie do wszystkich wywołań programu wykonanych w tej sesji powłoki. Ustawiasz ją raz, a program, który szuka zmiennej środowiskowej, będzie ją respektował za każdym razem, gdy zostanie uruchomiony, aż zmienna zostanie zmodyfikowana lub zakończysz sesję powłoki.

Uwaga

Zmienne środowiskowe omówimy szerzej w rozdziale 12. „Automatyzacja zadań za pomocą skryptów powłoki”, ale w tym podrozdziale przedstawimy podstawy.

Większość standardowych środowisk uniksowych używa zmiennych środowiskowych jako sposobu określania typowych konfiguracji, które są istotne dla wielu różnych programów, a nie tylko dla jednego. Na przykład zmienne środowiskowe śledzą, gdzie można znaleźć katalog `/home` użytkownika (`$HOME`), jaki jest bieżący katalog roboczy (`$PWD`), która powłoka powinna być domyślnie używana (`$SHELL`), gdzie szukać plików wykonywalnych odpowiadających poleceniom otrzymywanym za pośrednictwem CLI (`$PATH`) itd.

Możesz je sprawdzić od razu — wartość konkretnej zmiennej środowiskowej możesz zobaczyć przez wypisanie jej za pomocą polecenia `echo`:

```
$ echo $SHELL
/bin/zsh
```

Możesz też użyć polecenia `env`, aby zobaczyć wszystkie aktualnie ustawione zmienne środowiskowe:

```
$ env
...
# Wiele linii danych wyjściowych, po jednej dla każdej z Twoich zmiennych środowiskowych
...
```

Aby ustawić zmienną środowiskową w bieżącej powłoce, użyj znaku równości (=) do przypisania (po obu stronach znaku równości ma nie być spacji):

```
MYVAR=fruitloops
```

Ustawiłeś ją dla swojej obecnej powłoki:

```
$ echo $MYVAR
fruitloops
```

Aby zachować tę zmienną dla wszystkich uruchamianych podpowłok (na przykład podczas uruchamiania skryptu), należy użyć wbudowanego narzędzia `export`:

```
export MYVAR=fruitloops
```

Więcej informacji na ten temat znajdziesz w rozdziale 12. „Automatyzacja zadań za pomocą skryptów powłoki”, ale powyższe polecenie przybliża zakres tego, czego będziesz potrzebować do przekazywania konfiguracji zmiennej środowiskowej do większości programów, z którymi będziesz pracować.

Wróćmy do przykładu `find`: jeśli przewiniesz stronę podręcznika `find` wystarczająco daleko, czego przedsmak pokazaliśmy w poprzednim podrozdziale, zobaczysz sekcję zatytułowaną `ENVIRONMENT`:

ENVIRONMENT

The `LANG`, `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_MESSAGES` and `LC_TIME`
→ environment variables affect the execution of the `find` utility
→ as described in `environ(7)`.

Jest to inny poziom konfiguracji — zamiast być przekazywane jako argumenty poleceń w czasie wykonywania, są to dyrektywy konfiguracyjne, które mogą być odczytywane ze zmiennych środowiskowych powłoki.

Dlaczego program powinien traktować zmienne środowiskowe inaczej niż argumenty? Przeanalizujmy to: argument wiersza poleceń `-H` jest niewiarygodnie specyficzny, ponieważ jest zdefiniowany na poziomie wywoływania poleceń. W rezultacie ma zastosowanie tylko do polecenia uruchamianego w danym momencie.

Natomiast zmienne środowiskowe są mniej specyficzne. Są definiowane na poziomie powłoki i dlatego są dostępne dla wszystkich poleceń uruchamianych z tej powłoki.

Przejdźmy dalej po hierarchii konfiguracji: jeśli wartość nie jest ustawiana w czasie wykonywania za pomocą argumentu wiersza poleceń lub jako zmienna środowiskowa w sesji powłoki, z której poziomu uruchamiany jest program, skąd pochodzi konfiguracja?

Pliki konfiguracyjne

Następnym miejscem, w którym program szuka konfiguracji, są pliki konfiguracyjne. **Lokalizacja**, w której program będzie szukał konfiguracji, może się znacznie różnić, ale istnieje kilka standardowych miejsc, które można przeszukać.

Konfiguracja na poziomie systemu w katalogu `/etc/`

Dobrym miejscem na początek jest przede wszystkim katalog `/etc/`. Pokazaliśmy go już wcześniej, w rozdziale 5. „Wprowadzenie do plików”. Powszechnym wyborem katalogu dla oprogramowania w celu przechowywania konfiguracji dla całego systemu jest `/etc/nazwa_programu`, gdzie fragment `nazwa_programu` należy zastąpić rzeczywistą nazwą programu, który chcesz skonfigurować. Dla wielu programów to wystarczy. Na przykład serwer WWW `nginx` jest programem na poziomie systemu: różni użytkownicy nie uruchamiają zwykle własnych instancji serwerów WWW na jednym komputerze, więc konfiguracja dla całego systemu jest wystarczająca.

Mimo to konfiguracja dużych lub złożonych programów może być nadal podzielona w katalogu `/etc/nazwa_programu`. Dobrym tego przykładem jest `nginx`: jego główny plik konfiguracyjny znajduje się w pliku `/etc/nginx.conf`, a dodatkowe pliki konfiguracyjne pochodzą z kolejnych plików z katalogu `/etc/nginx/conf.d/` `/etc/nginx/conf.d/`.

Konfiguracja na poziomie użytkownika w katalogu `~/.config`

W przypadku programów, które mają istotną konfigurację dla każdego użytkownika (np. edytory tekstu, narzędzia programistyczne, gry), używany jest katalog `~/.config`, znajdujący się w katalogu domowym użytkownika. Przypomnijmy z rozdziału 1. „Jak działa wiersz poleceń?”, że znak `~` jest skrótem dla katalogu domowego bieżącego użytkownika, a ponadto katalogi, których nazwy zaczynają się od znaku kropki (`.`), są pomijane w wyjściu polecenia `ls`, chyba że prześlemy mu flagę `-a`. Katalog `~/.config` jest częścią standardu katalogu bazowego XDG, którego omówienie znajdziesz na stronie https://wiki.archlinux.org/title/XDG_Base_Directory.

Na przykład nasza konfiguracja `neovima` znacznie różni się od konfiguracji innych programistów, jednak jeden plik binarny `neovima` w systemie może obsługiwać setki programistów pracujących na tym samym komputerze jednocześnie, wywołanie `neovima` przez każdego programistę wykorzystuje bowiem jego charakterystyczne dla użytkownika pliki konfiguracyjne przechowywane w lokalizacji `~/.config/nvim/`. I nie ma w tym nic złego!

Możesz sobie wyobrazić pandemonium, które powstałoby, gdyby było tylko jedno systemowe miejsce do skonfigurowania tego programu w katalogu `/etc/` — każdy programista przed uruchomieniem edytora `neovim` musiałby ustawiać niezliczone zmienne środowiskowe lub wywoływać polecenie edytora z mnóstwem flag wiersza poleceń.

Skoro omówiliśmy już klasyczne źródła konfiguracji dla programów uniksowych, przyjrzyjmy się jednej charakterystycznej dla Linuksa komplikacji, o której należy wiedzieć: w jaki sposób konfiguracja za pomocą plików środowiskowych i argumentów CLI jest zarządzana dla programów kontrolowanych przez `systemd`?

Jednostki `systemd`

W większości dystrybucji Linuksa najważniejszą pracę wykonuje — oprócz kontenerów `Dockera` — `systemd`. Omówiliśmy już podstawy usługi `systemd` (zobacz rozdział 3. „Zarządzanie usługami za pomocą usługi `systemd`”), a w tym podrozdziale przyjrzymy się, jak `systemd` zarządza konfiguracją programów.

Najpierw szybki przegląd na wypadek, gdyby rozdział 3. wydawał Ci się strasznie odległy: w środowisku Linuksa zarządzanym przez `systemd` usługi są pakowane w pliki jednostek `systemd`, które opakowują i kontrolują rzeczywisty wykonywalny plik binarny, jego argumenty, polecenia używane do uruchamiania, restartowania, zatrzymywania jednostki itd.

Jak już wspomnieliśmy, istnieje wiele typów jednostek `systemd`, ale interesuje nas tutaj `service`.

Omówiliśmy już fakt, że pliki jednostkowe mogą istnieć w kilku różnych katalogach, w zależności od ich przeznaczenia, ale Twoje niestandardowe jednostki `systemd` będą zazwyczaj przechowywane w katalogu `/etc/systemd/system`.

Aby zrozumieć, w jaki sposób jednostka systemd pozwala wpływać na warstwy hierarchii konfiguracji, które omówiliśmy w tym rozdziale, utwórzmy usługę zarządzaną przez systemd, pisząc własną jednostkę systemd dla wymyślnego programu o nazwie `yourprogram`.

Tworzenie własnej usługi

Jako programista możesz mieć potrzebę opakowania pisanego programu w usługę, która może być łatwiej zarządzana niż ręcznie (interaktywnie) wywoływany program. Jest to niezwykle przydatne samo w sobie, ale w tym rozdziale zagłębimy się w dodatkową kontrolę, jaką dają jednostki systemd nad tym, jak i gdzie program jest konfigurowany. Przejdźmy proces tworzenia usługi przez opakowanie pliku binarnego w plik jednostki systemd.

Najpierw upewnij się, że plik wykonywalny został skopiowany do miejsca, które znajduje się w domyślnej zmiennej `$PATH`: `/usr/local/bin/yourprogram`. Najlepiej skorzystaj z ręcznie skompilowanego programu, takiego jak plik binarny `htop`, który utworzyliśmy w rozdziale 9. „Zarządzanie zainstalowanym oprogramowaniem”, i zastąp nim wymyślny program `yourprogram`.

Teraz utwórz następujący plik jednostki systemd w katalogu `/etc/systemd/system/yourprogram.service`:

```
[Unit]
Description=Your program description.
After=network-online.target

[Service]
Type=exec
ExecStart=/usr/local/bin/yourprogram -clioption=1 -clioption2
EnvironmentFile=-/etc/yourprogram/prod_defaults
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

Czy w tym pliku jednostki możesz znaleźć dwie linie związane z konfiguracją?

Widać, że linia `ExecStart` określa, w jaki sposób program jest wywoływany, gdy ktoś uruchomi tę usługę systemd. Używamy pliku jednostki systemd do przekazywania argumentów wiersza poleceń do programu, aby za każdym razem gdy ktoś uruchomi usługę, program został uruchomiony z dokładnie tymi opcjami, które chcemy. Ilekroć ktoś uruchomi `systemctl start yourprogram`, `yourprogram` zostanie wywołany z opcjami `-clioption=1` i `-clioption2`.

Druga linia, `EnvironmentFile`, określa ścieżkę pliku dla systemd w celu sprawdzenia, gdzie może on oczekiwać ustawienia zmiennych środowiskowych istotnych dla tego programu. Ten plik będzie parsowany przez powłokę, której systemd używa do uruchamiania pliku binarnego; powinien on zawierać przypisania zmiennych powłoki, takie jak:

```
# Zmienne środowiskowe dla yourprogram
ENV=production
DB_HOST=localhost
DB_PORT=5432
```

Niech systemd ponownie odczyta pliki konfiguracyjne, abyśmy mieli pewność, że widzi nową usługę Unit, którą zdefiniowaliśmy:

```
$ sudo systemctl daemon-reload
```

Teraz możesz zarządzać tym jak każdą inną usługą systemd:

```
systemctl start yourprogram
systemctl status yourprogram
systemctl stop yourprogram
systemctl enable yourprogram
systemctl disable yourprogram
```

Wiesz, że za każdym razem gdy będziesz uruchamiać tę usługę, plik środowiskowy w `/etc/yourprogram/prod_defaults` będzie używany jako źródło zmiennych środowiskowych, a linia `ExecStart` przekaże określone opcje CLI.

Pokazaliśmy tutaj niezwykle prostą usługę, aby wyjaśnić Ci, w jaki sposób systemd jest używany do sterowania konfiguracją programu, ale istnieje *wiele* innych dyrektyw konfiguracyjnych, które możesz tutaj przekazywać. Jeśli musisz zająć się bardziej złożoną usługą, poświęć trochę czasu na zapoznanie się z dokumentacją jednostki systemd (<https://www.freedesktop.org/software/systemd/man/latest/systemd.unit.html#%5BUnit%5D%20>).

Kilka zdań na temat konfiguracji w Dockerze

Wcześniej w tym rozdziale wspomnieliśmy, że Docker jest często wyjątkiem, jeśli chodzi o konfigurację. Ponieważ kontenery Dockera są o wiele bardziej minimalnym środowiskiem, nie mają zbyt wielu dodatkowych plików binarnych, usług i plików konfiguracyjnych, które znajdziesz w tradycyjnym systemie Unix. Jednak większość oprogramowania tworzonego przez programistów działa teraz w kontenerach, w przeciwieństwie do tradycyjnych, pełnych środowisk systemu operacyjnego, dlatego chcemy tutaj omówić pewne podstawy, aby dać Ci wyobrażenie na temat tego, czym różni się konfiguracja w kontenerach Dockera. Kontenery Dockera omówimy szerzej w rozdziale 15. „Konteneryzacja aplikacji za pomocą Dockera”.

W środowisku kontenerowym — niezależnie od tego, czy jest to środowisko Dockera, czy innych kontenerów — mamy do czynienia ze znacznie zminimalizowanym środowiskiem. Istnieje bardzo niewiele zainstalowanych programów i narzędzi, drastycznie ograniczony `init` zamiast usługi `systemd` oraz znacznie mniejszy system plików, który nie ma wielu katalogów, o których tu wspominaliśmy.

Jednak zasada hierarchii konfiguracji nadal ma zastosowanie. Większość aplikacji kontenerowych oczekuje uzyskania konfiguracji z kilku miejsc. Oto one:

- plik konfiguracyjny znajdujący się gdzieś w systemie plików kontenerów, często dynamicznie tworzony przez planistę kontenerów tuż przed uruchomieniem kontenera;
- zmienne środowiskowe przekazywane przez planistę kontenera lub przez uruchamiającego go operatora;
- argumenty wiersza poleceń.

Chociaż jest to uproszczona wersja hierarchii konfiguracji, zauważysz, że jest zasadniczo identyczna z tą, którą zbadaliśmy w pełnych, niekontenerowych systemach Linuksa.

Kontenery omówimy szczegółowo w rozdziale 15. „Konteneryzacja aplikacji za pomocą Dockera”.

Podsumowanie

W tym rozdziale przedstawiliśmy przegląd hierarchii konfiguracji Linuksa i jej zastosowania do programów, z których będziesz korzystać (i które będziesz pisać) w swojej codziennej pracy. Omówiliśmy argumenty wiersza poleceń, zmienne środowiskowe i wszystkie inne elementy wpasowujące się w większą hierarchię, z której programy pobierają konfigurację.

W przykładzie utworzyliśmy usługę `systemd`, która opakowuje program i pozwala zarządzać jego konfiguracją w bardziej ujednolicony sposób.

Potoki i przekierowanie

Rozdział 11

W tym rozdziale pokażemy Ci, jak wykorzystać jedną z najpotężniejszych koncepcji obliczeniowych, jaką są potoki! Potoki mogą być używane do łączenia poleceń, tworząc złożone, dostosowane przepływy, które spełniają określone zadanie. Po lekturze tego rozdziału będziesz w stanie zrozumieć (lub skomponować) coś takiego:

```
history | awk '{print $2}' | sort | uniq -c | sort -rn | head -n 10
```

Powoduje to wypisanie listy 10 najczęściej używanych poleceń powłoki — na naszym komputerze zostały wygenerowane następujące dane wyjściowe:

```
1000 git
115 ls
102 go
 83 gpo (alias, który ustawiłem do wysłania lokalnej gałęzi Gita
    ↳do oryginalnej)
 68 make
 65 cd
 59 docker
 42 vagrant
 35 G00S=linux
 30 echo
```

Aby zrozumieć potoki, musisz najpierw zrozumieć deskryptory plików i przekierowanie operacji we-wy, więc od tego zaczniemy. Niektóre informacje zawarte w tym rozdziale są dość skompilowane, więc nie spiesz się i wypróbuj wszystkie przykłady, aby wszystko dobrze zrozumieć. Czas, który zainwestujesz teraz w naukę tych pojęć, pozwoli Ci zaoszczędzić wiele godzin w całej Twojej karierze programisty.

W tym rozdziale omawiamy następujące zagadnienia:

- deskryptory plików;
- łączenie poleceń za pomocą potoków (|);
- narzędzia CLI;
- praktyczne wzorce potoków;
- sprawdzanie deskryptorów plików.

Deskryptory plików

Zapewne z własnego doświadczenia w inżynierii oprogramowania znasz uchwyty plików (zwane również **deskryptorami plików**). Jeśli nie, zalecamy zapoznanie się z rozdziałem 5. „Wprowadzenie do plików”. Krótko mówiąc, jeśli Twój program musi odczytać lub zapisać plik w systemie operacyjnym, otwarcie tego pliku daje do niego „uchwyt” — wskaźnik, czyli inaczej mówiąc, referencję do tego obiektu pliku.

Ponieważ system operacyjny pośredniczy w dostępie do zasobów systemowych, takich jak pliki, śledzi, do których uchwytów (deskryptorów) plików program aktywnie się odwołuje.

Ale nawet jeśli proces nie korzysta z żadnego pliku w systemie operacyjnym, ma otwarte pewne uchwyty plików. W uniksowych systemach operacyjnych każdy proces ma co najmniej trzy deskryptory plików:

- `stdin` — standardowy strumień danych wejściowych lub `fd 0` („zerowy deskryptor pliku”);
- `stdout` — standardowy strumień danych wyjściowych lub `fd 1` („pierwszy deskryptor pliku”);
- `stderr` — standardowy strumień błędów lub `fd 2` („drugi deskryptor pliku”).

Te pierwsze trzy deskryptory plików działają jako standardowe kanały komunikacji między procesami. W rezultacie istnieją one w tej samej kolejności dla każdego procesu tworzonego w systemie. Pierwszy zawsze wskazuje na plik, który będzie używany do odczytu danych wejściowych. Drugi wskazuje na plik, który będzie używany do zapisu danych wyjściowych. A trzeci odwołuje się do pliku, który otrzyma dane wyjściowe na temat błędu.

Opcjonalnie, po tych pierwszych trzech standardowych deskryptorach, może istnieć dowolna liczba innych deskryptorów (uchwytów), w zależności od tego, co robi dany program. Twój proces może mieć następujące elementy:

- pliki, z którymi pracuje;
- gniazda, z których odczytuje lub w których zapisuje (gniazda uniksowe lub TCP zostały napisane dla obsługi sieci);
- urządzenia takie jak klawiatury lub dyski, z których korzysta.

Do czego odwołują się te deskryptory plików?

Teraz wiesz już, do czego służą te deskryptory plików z perspektywy procesów:

- `0 (STDIN)` — stąd są pobierane dane wejściowe;
- `1 (STDOUT)` — tutaj umieszczane są normalne dane wyjściowe;
- `2 (STDERR)` — tutaj umieszczane są dane wyjściowe błędów.

Ale jeśli przyjrzymy się z bliska pojedynczemu procesowi, na które pliki tak naprawdę wskazują te deskryptory plików? Skąd pochodzą dane wejściowe i gdzie zapisywane są dane wyjściowe i błędy?

Jako przykładu użyjmy procesu powłoki Bash: domyślnie pobiera on dane wejściowe (STDIN) z terminala (który jest reprezentowany przez plik w systemie plików). Bash wypisuje dane wyjściowe i błędy w tym samym terminalu. Zasadniczo cała sesja powłoki polega na wykonywaniu operacji odczytu i zapisu w jednym pliku. Więcej informacji na ten temat znajdziesz w rozdziale 12. „Automatyzacja zadań za pomocą skryptów powłoki”.

Przyjrzyjmy się dokładnie temu rodzajowi przekierowania wejścia i wyjścia.

Przekierowywanie wejścia i wyjścia (praca z deskryptorami plików dla zabawy i potencjalnych korzyści)

Ta wiedza przydaje się dość często podczas wykonywania rzeczywistych zadań programistycznych: gdy chcesz uniknąć wpisywania dużej ilości danych wejściowych i zamiast tego pobierać je z pliku, gdy chcesz rejestrować dane wyjściowe programu i w wielu innych sytuacjach. Podczas tworzenia procesu można kontrolować, gdzie wskazują jego trzy standardowe deskryptory plików. Można dzięki temu wiele osiągnąć.

Przekierowywanie danych wejściowych — <

Symbol < (mniejsze niż) pozwala kontrolować, skąd proces pobiera dane wejściowe. Z pewnością jesteś na przykład przyzwyczajony do podawania powłoce Bash danych wejściowych za pomocą klawiatury, po jednym poleceniu. Zamiast tego spróbujmy podać tej powłoce dane wejściowe z pliku!

Załóżmy, że mamy plik *commands.txt* z następującą zawartością (używamy tutaj *cat* do wypisania zawartości przykładowego pliku):

```
# cat commands.txt
pwd
echo "witajcie, przyjaciele"
echo $SHELL
cd /tmp
pwd
```

Jeśli chodzi o Bash, są to prawidłowe polecenia powłoki, zamierzamy więc uruchomić nowy proces powłoki i użyć tego pliku jako standardowych danych wejściowych:

```
# bash < commands.txt
/tmp/gopsinspect
witajcie, przyjaciele
/bin/bash
/tmp
```

Zamiast monitować nas o wprowadzenie danych i czekać, aż je podamy, Bash odczytuje i wykonuje po jednej linii: odczytuje dane wejściowe z pliku, dopóki nie natknie się na znak nowej linii (`\n`), i wykonuje polecenie tak, jakbyśmy wcisnęli przycisk *Return*.

W tym przykładzie standardowe dane wyjściowe wracają do naszego terminala, gdzie możemy je odczytać. Spróbujmy to teraz zmienić.

Przekierowywanie danych wyjściowych — >

Chcemy przekierować STDOUT (deskryptor pliku 1) do pliku zamiast do terminala, rejestrując dane wyjściowe każdego polecenia zamiast wypisywać je w terminalu w czasie rzeczywistym:

```
# bash < commands.txt > output.log
```

Zauważ, że w terminalu nie ma teraz żadnych widocznych danych wyjściowych, ponieważ znak > przekierował je do pliku *output.log*. Użyj polecenia *cat*, aby wypisać ten plik dziennika i potwierdzić, że zawiera on oczekiwane dane wyjściowe:

```
# cat output.log
/tmp/gopsinspect
witajcie, przyjaciele
/bin/bash
/tmp
```

Co ciekawe, zauważysz, że ponieważ deskryptor pliku 1 jest standardowym wyjściem, zapis > jest równoważny z zapisem 1>. Rzadko spotkasz się z zapisem 1, gdyż przyjmuje się założenie, że standardowe wyjście jest przekierowywane. Innymi słowy poniższe zapisy są równoważne:

```
date > mydate.log
równoważne z napisaniem
date 1> mydate.log
```

Użycie znaków >>, aby dołączać dane wyjściowe bez nadpisywania

W poprzednim przykładzie utworzyliśmy plik dziennika, przekierowując wyjście polecenia za pomocą znaku >. Jeśli uruchomisz ten przykład kilka razy, zauważysz, że plik dziennika w ogóle się nie rozrasta. Za każdym razem gdy przekierujesz dane wyjściowe do pliku za pomocą > *nazwa_pliku*, wszystko w tym pliku zostanie nadpisane.

Aby tego uniknąć — tak jak w przypadku długotrwałego pliku dziennika, który zbiera dane wyjściowe z więcej niż jednego procesu lub polecenia — użyj znaków >> (dołącz). Spowoduje to dołączanie danych do pliku wyjściowego, zamiast nadpisywania za każdym razem całej jego zawartości.

Skrypty powłoki Bash omówimy szczegółowo dalej, a na razie przedstawimy krótki skrypt, który zapisuje znacznik bieżącego czasu w pliku dziennika raz na sekundę:

```
while true; do
    date >> /tmp/date.log
    sleep 1
done
```

W tym przykładowym skrypcie tworzymy nieskończoną pętlę (`while true; do [...]`), która uruchamia polecenie `date`. Przekierowuje ona dane wyjściowe tego polecenia do pliku `/tmp/date.log` za pomocą znaków `>>`, które dołączają dane wyjściowe do pliku (znak `>` za każdym razem nadpisałby plik). Następnie skrypt śpi przez sekundę i zaczyna się od początku.

Uruchomienie polecenia `date` jeden raz generuje następujące dane wyjściowe:

```
→ ~ date
Sat Jan 6 16:39:37 EST 2024
```

Z drugiej strony uruchomienie tego skryptu nie robi na początku nic widocznego, dane wyjściowe są bowiem przekierowywane do pliku. Oto jak to wygląda, kiedy wklejamy ten mały skrypt do terminala, pozwalamy mu działać przez chwilę, przerywamy jego działanie za pomocą `Ctrl+C`, a następnie wypisujemy zawartość utworzonego pliku:

```
→ ~ while true; do
  date >> /tmp/date.log
  sleep 1
done
^C%
→ ~ cat /tmp/date.log
Sat Jan 6 16:44:01 EST 2024
Sat Jan 6 16:44:02 EST 2024
Sat Jan 6 16:44:03 EST 2024
[ ... ]
Sat Jan 6 16:44:08 EST 2024
```

Tego rodzaju prostego przekierowania wyjściowego będziesz używać we wszystkich codziennych sytuacjach, takich jak tworzenie pliku dziennika ad hoc dla szybkiego debugowania skryptu.

Przekierowywanie błędów za pomocą `2>`

Wiele programów wiersza poleceń, które mają dużo oczekiwanych wyników, generuje również sporadyczne błędy — wyobraź sobie polecenie `find`, które napotyka sporadyczne błędy „odmowy uprawnień” dla katalogów, do których nie możesz zajrzeć.

Chociaż tego rodzaju błędy są rzadkie i oczekiwane, nie chcesz, aby były zmieszane ze wszystkim innym, wpływając negatywnie na Twoją wydajność. Staje się to szczególnie ważne, gdy nie używasz narzędzi wiersza poleceń interaktywnie, lecz raczej piszesz krótkie skrypty lub większe programy, które przetwarzają wyniki uruchamianych poleceń.

Pokazaliśmy już, jak przekierowywać standardowe dane wejściowe (`fd 0`) i standardowe dane wyjściowe (`fd 1`). Przyjrzyjmy się teraz, jak przekierować standardowy strumień błędów (`fd 2`) za pomocą składni `2>` (deskryptora 2 dla przekierowania pliku):

```
find /etc/ -name php.ini > /tmp/phpinis.log 2>/dev/null
```

To polecenie wyszukuje wszystkie pliki o nazwie `php.ini` w drzewie katalogu `/etc`. Znalezione pliki (STDOUT polecenia `find`) są zapisywane w pliku `/tmp/phpinis.log`, a wszelkie napotkane błędy są ignorowane przez wysłanie ich do specjalnego pliku o nazwie `/dev/null`.

Wskazówka

`/dev/null` to specjalny obiekt podobny do pliku, który zwraca zera, gdy próbujesz go odczytać, i ignoruje wszystko, co zostało w nim zapisane — jest używany jako coś w rodzaju wysypiska śmieci dla danych wyjściowych, które inżynierowie chcą wyciszyć lub zignorować. Przekonasz się, że jest on dość często wykorzystywany w skryptach.

Skoro znasz już przekierowanie wejścia i wyjścia, przyjrzyjmy się potokom, które łączą obie te koncepcje: przekierowują wyjście jednego polecenia do wejścia drugiego.

Łączenie poleceń za pomocą potoków (|)

Nauczyłeś się przekierowywać każdy z trzech standardowych deskryptorów plików do różnych lokalizacji i przekonałeś się, dlaczego często jest to przydatne. Ale co, jeśli zamiast przekierowywać wejścia i wyjścia między plikami plików, chcesz połączyć ze sobą *wiele programów*?

W wierszu poleceń można użyć znaku potoku (`|`), aby połączyć wyjście jednego programu z wejściem drugiego. Jest to niezwykle potężny paradygmat, który jest często używany w systemach Unix i Linux do tworzenia niestandardowych poleceń sortowania, filtrowania i przetwarzania:

```
echo -e "jakiś tekst \n znaleziono skarb \n więcej jakiegoś tekstu" | grep skarb
```

Jeśli wkleisz to do swojej powłoki, wypisany zostanie tekst `znaleziono skarb`. Oto co się stało:

1. Zostało uruchomione pierwsze polecenie `echo` i wygenerowało dane wyjściowe, które widzisz między podwójnymi cudzysłowami (znaki nowej linii sprawiają, że jest to 3-liniowy łańcuch znaków).
2. Znak potoku przesyłał strumieniowo te dane wyjściowe (deskryptor pliku 1) do wejścia następnego polecenia (deskryptor pliku 0) `grep`. Dane wyjściowe `grep` zostały teraz podłączone do wyjścia z poprzedniego polecenia.
3. Następnie polecenie `grep` przejrzało każdą oddzieloną nową linię i znalazło dopasowanie do `skarb` w drugiej linii. `grep` wypisało tę drugą linię do standardowego wyjścia.

Polecenia z wieloma potokami

Oto dość ekstremalny przykład, który pokazaliśmy na początku rozdziału:

```
history | awk '{print $2}' | sort | uniq -c | sort -rn | head -n 10 > \n/tmp/top10commands
```

Każdy potok w tym złożonym poleceniu pobiera wynik poprzedniego polecenia (STDOUT) i używa go jako wejścia (STDIN) dla następnego polecenia.

Przekazywanie za pomocą potoku wyjścia jednego polecenia do wejścia drugiego umożliwia tego rodzaju przepływy pracy, filtrując i sortując strumienie danych między tymi poleceniami bez konieczności pisania niestandardowego oprogramowania. To, że nie ma programu o nazwie `top10commands`, nie oznacza, iż nie można szybko poskładać takiego polecenia z istniejących standardowych poleceń, takich jak te.

Odczytywanie (i budowanie) złożonych poleceń wielopotokowych

Bez względu na to, jak skomplikowane lub magiczne będą się wydawać niektóre z połączonych potokami poleceń, wszystkie zostały zbudowane w ten sam sposób: po jednym poleceniu w danym momencie. Niezależnie od tego, czy próbujesz odczytywać tak złożoną serię poleceń, czy tworzyć własne, proces jest taki sam:

1. Weź pierwsze polecenie i upewnij się, że rozumiesz, co robi na podstawowym poziomie. Przejrzyj stronę podręcznika lub inną dokumentację, jeśli nie jesteś zaznajomiony z tym poleceniem.
2. Uruchom polecenie i sprawdź jego dane wyjściowe.
3. Dodaj potok i następujące po nim polecenie.
4. Powtarzaj czynności od *kroku 1.* do momentu, aż zrozumiesz działanie wszystkich poleceń.

Zobaczysz, że gdy zastosujesz ten proces, będziesz w stanie opanować nawet najbardziej przerażające powłokowo-potokowe zawiłości. Zawsze pamiętaj, że masz do czynienia tylko ze strumieniem danych, który przepływa przez potoki od polecenia do polecenia, a po drodze jest kształtowany, modyfikowany, filtrowany, przekierowywany i przekształcany.

Omówimy to szczegółowo w rozdziale 12. „Automatyzacja zadań za pomocą skryptów powłoki”, ale postaraj się szanować innych programistów, którzy muszą czytać Twój kod: ogranicz swoje instrukcje do dwóch lub trzech potoków i używaj dobrze nazwanych zmiennych do przechowywania pośrednich wyników w celu ułatwienia odczytu, jeśli pozwalają na to ograniczenia pamięci.

Skoro wiesz już, jak podstawowe deskryptory plików są udostępniane jako łatwe w użyciu przekierowania wejścia i wyjścia, przyjrzyjmy się kilku prawdziwym przykładom użytecznych kombinacji programów, które opierają się na tej komponowalności wbudowanej w system Unix.

Narzędzia CLI, które należy znać

Zanim przejdziemy do dziwacznych kombinacji, które pokazaliśmy na początku rozdziału, przyjrzyjmy się kilku z najczęściej używanych narzędzi Uniksa, które służą do filtrowania, sortowania i sklejanego strumieni danych, które będziesz tworzyć w wierszu poleceń.

cut

Narzędzie `cut` pobiera separator (`-d`) i na jego podstawie dzieli dane wyjściowe, podobnie jak `String.Split()` lub `String.Fields()` w wielu językach programowania. Następnie należy za pomocą opcji `-f`, na przykład `f1` dla pierwszego pola, wybrać, które pole (element listy) ma generować dane wyjściowe.

Jeśli podasz poleceniu `cut` więcej niż jedną linię wejścia, powtórzy ona tę samą operację na wszystkich liniach:

```
echo "this is a space-delimited line" | cut -d " " -f4
space-delimited
```

Możesz również zobaczyć, jak działa użycie różnych separatorów w poleceniu `cut` — w poniższym przykładzie zamiast spacji wykorzystamy znak łącznika:

```
→ ~ echo "this is a space-delimited line" | cut -d "-" -f1
this is a space
→ ~ echo "this is a space-delimited line" | cut -d "-" -f2
delimited line
```

Widać, że zmienia to również liczbę dostępnych pól — w tym przypadku są dwa pola, ponieważ w tekście jest tylko jeden łącznik. Próba wypisania czwartego pola za pomocą `f4`, jak w poprzednim przykładzie, daje po prostu pustą linię.

Aby na komputerze z systemem macOS uzyskać przyjazne nazwy dla wszystkich użytkowników root, można użyć następujących opcji:

```
# grep root /etc/passwd | cut -d ":" -f5
System Administrator
System Services
CVMS Root
```

sort

Polecenie `sort` przeprowadza sortowanie poszczególnych linii alfabetycznie lub numerycznie.

Sortowanie odwrotne za pomocą opcji `-r` jest często przydatne w przypadku danych liczbowych (`-n`). Często będziesz używać razem opcji `-rn` (zobacz punkt „»Top X« z licznikiem” w podrozdziale „Praktyczne wzorce potoków”).

Flaga `-h` może być bardzo przydatna do sortowania na podstawie czytelnych dla człowieka danych wyjściowych wielu innych poleceń pokazanych w poniższym listingu:

```
# du -h | sort -rh
1.6M  .
1.3M  ../git
1.2M  ../git/objects
60K   ../git/hooks
28K   ../git/objects/d8
```

uniq

To polecenie usuwa zduplikowane linie. W celu oczekiwanego działania wymaga posortowanych danych, w przeciwnym razie sprawdza tylko, czy każda linia nie jest duplikatem poprzedniej:

```
# cat /tmp/uniq
one
two
one
one
one
seven
one
```

Domyślne zachowanie nie jest prawdopodobnie tym, czego byśmy oczekiwali:

```
# uniq /tmp/uniq
one
two
one
seven
one
```

uniq pomija wystąpienia, gdy znajdują się jedno po drugim, ale pozostawia je, kiedy są oddzielone innym tekstem. Teraz to samo z posortowanymi danymi:

```
# sort /tmp/uniq | uniq
one
seven
two
```

Zliczanie

Polecenie uniq ma również użyteczną opcję zliczania, którą można włączyć za pomocą argumentu -c. Warto tutaj powtórzyć to samo zastrzeżenie co przy posortowanych danych wejściowych — na przykład dla pliku o następującej zawartości:

```
arch
alpine
arch
arch
```

Po uruchomieniu przez uniq -c wygenerowany zostanie następujący wynik:

```
$ uniq -c /tmp/sort1.txt
1 arch
1 alpine
2 arch
```

Nie tego oczekiwałyby większość użytkowników: w tym pliku są trzy wystąpienia arch, ale uniq pokazuje dwa oddzielne liczniki dla tego samego słowa. Aby uzyskać oczekiwane zachowanie (polecenie uniq powinno zwrócić wyjście, które nie zawiera żadnych zduplikowanych linii), dane wejściowe muszą być posortowane.

Dla początkujących użytkowników jest to irytujące, ale całkowicie zgodne z filozofią Uniksa: narzędzia powinny być niewielkie i ostre i nie powinny powielać wzajemnie swoich funkcjonalności. Jeśli piszesz narzędzie sortowania, powinno ono tylko sortować, a jeśli piszesz narzędzie ujednolicające, może ono opierać się na sortowaniu, które jest wykonywane przez inne narzędzia, aby zapewnić ekstremalnie konserwatywne (i spójne) wykorzystanie pamięci.

Tutaj przed użyciem `uniq` najpierw sortujemy, co daje nam oczekiwane dane wyjściowe:

```
$ sort /tmp/sort1.txt | uniq -c
 1 alpine
 3 arch
```

Zwróć uwagę, że to sortowanie odbywa się w kolejności rosnącej, a nie tego chcielibyśmy dla listy poleceń `top X`, którą pokazaliśmy na początku rozdziału. Aby rozwiązać ten problem, przeprowadzamy sortowanie „liczbowe odwrotne” (`-rn`) tej numerowanej listy (ponieważ każda linia zaczyna się teraz od liczby, a dzięki `uniq -c` jest to łatwe do zrobienia). Oto przykład tego w działaniu dla pliku z wieloma duplikatami:

```
$ sort /tmp/sortme.txt | uniq -c | sort -rn
 6 ubuntu
 4 alpine
 3 gentoo
 2 yellow dog
 2 arch
 1 suse
 1 mandrake
```

WC

Za pomocą tego polecenia można policzyć wejściowe słowa, linie, znaki i bajty. Można również liczyć słowa rozdzielane spacjami za pomocą opcji `-w`:

```
# echo "foo bar baz" | wc -w
3
```

Liczenie linii jest niezwykle powszechne w następującym formacie:

```
# wc -l < /etc/passwd
123
```

head

Polecenie `head` zwraca pierwsze linie strumienia lub pliku — domyślnie 10 linii. Liczbę linii możesz określić za pomocą opcji `-n`:

```
# head -n 2 /etc/passwd
##
# User Database
```

tail

Jest to przeciwieństwo polecenia `head`: zwraca linie z końca pliku lub strumienia. Przyjmuje flagę `-n` podobnie jak `head`.

Polecenie `tail` może być również używane interaktywnie do śledzenia pliku dziennika, nawet jeśli do tego pliku są strumieniowane (zapisywane) nowe dane. Bardzo często będziesz je napotykać podczas rozwiązywania problemów:

```
tail -f /var/log/nginx/access.log
```

tee

Czasami jedna kopia danych ze standardowego wejścia nie wystarczy. Polecenie `tee` kopiuje standardowe wejście do standardowego wyjścia, zapisując jednocześnie kopię w pliku. Bardzo lubimy stosować `tee` w dwóch konkretnych przypadkach.

Pierwszy to debugowanie i rejestrowanie: kiedy uruchamiamy skrypty lub programy, które generują dane wyjściowe, polecenie `tee` może być używane do wyświetlania wyjścia na ekranie i rejestrowania go w pliku w celu późniejszej analizy. Wykorzystujemy tutaj polecenie `echo`, ale prawdopodobnie przed pierwszym potokiem wywołasz tu swój własny program:

```
# echo "Hello" | tee /tmp/greetings.txt
Hello

# cat /tmp/greetings.txt
Hello
```

Drugi przypadek użycia, w którym `tee` jest przydatne, to kopiowanie danych z potoków, takich jak `te`, które uczymy się konstruować w tym rozdziale. Możesz użyć `tee`, aby przechwycić ten przepływ w dowolnym punkcie potoku i zapisać lub sprawdzić wyniki pośrednie bez zakłócania działania potoku.

Oto wcześniejszy przykład „10 najlepszych poleceń”, ale z poleceniem `tee` wstawionym przed ograniczeniem wyników do 10. Zapisuje to pełne wyniki w pliku tymczasowym przed ich skróceniem:

```
history | awk '{print $2}' | sort | uniq -c | sort -rn | tee
/tmp/all_commands.txt | head -n 10
```

Jeśli chcesz teraz zobaczyć wszystkie polecenia, a nie tylko najlepsze 10, możesz po prostu użyć narzędzia `cut` lub `less`, aby sprawdzić plik `/tmp/all_commands.txt`.

awk

Polecenie `awk` jest często stosowane jedynie do kolumn danych, ale w rzeczywistości jest to cały język.

Można na przykład pobrać drugą kolumnę z każdej linii w następujący sposób:

```
# echo "two columns" | awk '{print $2}'
columns
```

sed

Polecenie `sed` jest edytorem strumieni z mnóstwem opcji. Najczęściej jest używane do zastępowania znaków w strumieniach lub plikach.

Wyobraź sobie, że masz plik o nazwie *somefile.txt*:

```
# cat /tmp/sensitive.txt
Nopasswords
not_a_password_either
sillypasswordtimes
password
ok this works
```

Jeśli chcesz zredagować *tylko* wiersz, który zawiera `password`, wykonaj następujące polecenie:

```
sed 's/^password$/REDACTED/' /tmp/sensitive.txt
nopasswords
not_a_password_either
sillypasswordtimes
REDACTED
ok this works
```

W tym przykładzie użyliśmy pliku zamiast strumienia wejściowego pochodzącego z innego polecenia. Domyślnie nie spowoduje to modyfikacji oryginalnego pliku. Jeśli *chcesz* zmodyfikować plik wejściowy, użyj opcji `-i` (ang. *in-place*), czyli modyfikacji w miejscu.

Skoro zapoznałeś się z już z potokami i niektórymi z najczęściej wykorzystywanych narzędzi wiersza poleceń, zestawimy razem te elementy konstrukcyjne i pokażemy kilka praktycznych wzorców, których możesz używać, aby ułatwić sobie codzienną pracę z wierszem poleceń.

Praktyczne wzorce potoków

Jak wspomnieliśmy wcześniej, dłuższe polecenia wielopotokowe buduje się iteracyjnie — po jednym. Istnieje jednak kilka przydatnych wzorców, które są często wielokrotnie wykorzystywane.

„Top X” z licznikiem

Ten wzorec sortuje dane wejściowe w porządku malejącym według liczby wystąpień. Widziałeś to w pierwotnym przykładzie z tego rozdziału, który wyświetlał najczęściej używane polecenia powłoki z pliku historii Basha.

Oto ten wzorec:

```
jakieś_wejście | sort | uniq -c | sort -rn | head -n 3
```

Możemy zauważyć następujące szczegóły dotyczące tego wzorca:

- Dane wejściowe są sortowane alfabetycznie, a następnie przekazywane do polecenia `uniq -c`, które do działania wymaga posortowania danych wejściowych.
- Narzędzie `uniq -c` eliminuje duplikaty, ale dodaje licznik (`-c`), wskazujący, ile duplikatów znaleziono dla każdego wpisu.
- Polecenie `sort` jest uruchamiane ponownie, tym razem jako sortowanie odwrotne (`-r` i `-n`), które sortuje unikatowe zliczenia z wejścia i przekazuje linie posortowane w odwrotnej kolejności (od najwyższej liczby).
- Polecenie `head` przyjmuje ten ranking i przycina go do trzech linii (`-n 3`), dając trzy pierwsze łańcuchy znaków z oryginalnych danych wejściowych wraz częstotliwością ich występowania.

Ten wzorzec może się przydać, gdy trzeba poznać najczęstsze przeglądarki użytkowników wyświetlające Twoją stronę, adresy IP najgorszych hakerów, którzy próbują sondować i wykorzystać Twoją stronę, lub w każdej innej sytuacji, w której użyteczna jest posortowana, rankingowa lista.

curl | bash

Wzorzec `curl | bash` jest popularnym skrótem stosowanym w Linuksie do pobierania i wykonywania skryptów bezpośrednio z internetu. Metoda ta łączy w sobie dwa potężne narzędzia wiersza poleceń: `curl`, który pobiera zawartość z adresu URL, i `bash`, czyli interpreter powłoki, wykonujący pobrany skrypt. Ten wzorzec pozwala zaoszczędzić mnóstwo czasu i umożliwia programistom szybkie wdrażanie aplikacji lub uruchamianie skryptów bez ich ręcznego pobierania, a następnie wykonywania.

Zainstalujmy na przykład blokujący reklamy serwer DNS Pi-hole, korzystając z tego wzorca:

```
curl -sSL https://install.pi-hole.net | bash
```

Przeanalizujmy to krok po kroku:

1. `curl -sSL https://install.pi-hole.net` — powoduje pobranie skryptu instalacji Pi-hole, który jest hostowany pod podanym adresem URL. Przekazujemy dwie opcje:
 - `-sS` — tryb cichy daje nieprzetworzoną odpowiedź z serwera, ale pokazuje błędy w przypadku ich wystąpienia;
 - `-L` — stosowanie przekierowań.
2. `|` — symbol potoku przekazuje wyjście z poprzedniego polecenia (`curl`) jako wejście do następnego polecenia (`bash`).
3. `bash` — wykonuje skrypt pobrany przez `curl`.

Jest to niezwykle przydatny wzorzec do automatyzacji takich rzeczy jak wdrożenia kodu albo instalacja lub konfiguracja lokalnego środowiska. Należy jednak zachować szczególną ostrożność, aby skrypt, który pobierasz i wykonujesz, nie był złośliwy. Ślepe uruchamianie skryptów z internetu jest bardzo złą praktyką.

Względy bezpieczeństwa dotyczące `curl` | `sudo` | `bash`

Za każdym razem gdy pozwalasz zewnętrznej stronie, aby uruchamiała kod na Twoim komputerze, przehandlowujesz dla wygody pewne względy bezpieczeństwa. Pod tym względem używanie `curl` | `sudo` | `bash` do instalowania czegoś za pośrednictwem skryptu hostowanego na zaufanym serwerze nie różni się zbytnio od korzystania z menedżera pakietów. Większość menedżerów pakietów (z wyjątkiem `nix`) nie ma również szczególnie imponującego projektu dotyczącego zabezpieczeń, ale zasadniczo zapewnia rozsądny zestaw funkcji bezpieczeństwa. Gdy stosujesz `curl` | `sudo` | `bash` do instalowania jakiegось skryptu, rezygnujesz z tych wszystkich funkcjonalności bezpieczeństwa:

- Nie ma takiego pakietu, którego sumę kontrolną można sprawdzić i który można kryptograficznie podpisać, aby mieć pewność, że mamy poprawną i oficjalną wersję.
- Nie ma ograniczeń — ani egzekwowania — z których serwerów pobierane są pakiety, i nie wiemy, jak bezpieczne są te serwery: nie masz sposobu na zidentyfikowanie zainfekowanego serwera, na którym znajdują się złośliwe skrypty instalacyjne.
- Same skrypty są tylko kodem uruchamianym z uprawnieniami roota na Twoim komputerze, więc mogą zrobić wszystko, co Ty możesz zrobić na swoim komputerze, to zaś może wyjść na dobre lub na złe. Szczercze mówiąc, wiele popularnych menedżerów pakietów również ma ten problem.

Z tych wszystkich powodów zwróć uwagę na nasze ostrzeżenie, aby rozdzielić polecenie `curl` na osobny krok i przed uruchomieniem `sudo bash` w celu wykonania pobranego skryptu instalacyjnego najpierw go przeczytać. Oto najważniejsze rzeczy, na które należy zwrócić uwagę:

- Upewnij się, że serwer lub domena, z których pobierasz skrypt, są godne zaufania; powinna to być strona renomowanego programisty lub zaufana zewnętrzna platforma hostingowa kodu.
- Upewnij się, że do pobierania za pomocą `curl` używasz szyfrowanego protokołu HTTPS (tzn. adres URL powinien zaczynać się od `https://`).
- Przeczytaj uważnie skrypt, aby zobaczyć, które polecenia uruchamia i skąd pobiera dodatkowy kod lub pliki wykonywalne. Jeśli pobiera dodatkowe skrypty lub pliki wykonywalne, przyjrzyj się również im.

Ustaliliśmy, iż `curl` | `sudo` | `bash` nie jest szczególnie bezpieczną metodą instalacji oprogramowania. Przestrzeganie tych wytycznych może pomóc Ci zachować większe bezpieczeństwo, jeśli — jak większość z nas — pewnego dnia ulegniesz pokusie i zastosujesz tę metodę instalacji dla konkretnego oprogramowania (na przykład za pomocą `homebrew` w systemie `macOS`).

Przyjrzyjmy się teraz innemu typowemu wzorcowi: filtrowaniu i wyszukiwaniu przy użyciu narzędzia `grep`.

Filtrowanie i wyszukiwanie za pomocą narzędzia grep

Kiedy uruchamiasz polecenia, które generują dużo danych wyjściowych, najlepszą praktyką jest filtrowanie tych danych w celu uzyskania pożądanych informacji. Najpopularniejszym służącym do tego narzędziem jest `grep` i możesz potraktować je jako wysoce konfigurowalną funkcję wyszukiwania tekstu lub dopasowywania łańcuchów znaków. Oto przykład, jak może wyglądać filtrowanie.

Wyobraź sobie, że musisz znaleźć katalog roboczy procesu Linuksa. W tym celu można zastosować narzędzie `ls -l`:

```
→ ~ ls -l | grep cwd
vagrant 3243 dcohen cwd   DIR           1,4      192
51689680 /Users/dcohen/code/my_vagrant_testenv
```

Oto krótki opis tego, co się tutaj dzieje:

1. Przy użyciu `ls -l` otrzymujemy listę otwartych uchwytów plików dla określonego procesu (PID 3243).
2. Następnie przekazujemy wyniki (`|`) do narzędzia `grep` i używamy go do wyszukiwania wyników dla łańcucha znaków `cwd`. Jest tylko jedna linia wyników, która zawiera łańcuch znaków `cwd`, więc jest to jedyna linia wypisywana przez `grep` w terminalu.

Ten wzorec jest przydatny zawsze, gdy masz *dużo* danych wejściowych, ale potrzebujesz tylko podzbioru tych danych, które można zidentyfikować za pomocą określonego łańcucha znaków. Polecenie `grep` działa na liniach tekstu wejściowego, więc jest niezwykle pomocne przy zbieraniu różnych danych. Mogą to być:

- dzienniki zawierające obserwowany adres IP;
- wystąpienia nazwy użytkownika w strumieniu danych przekazywanych za pomocą potoku;
- linie pasujące do wzorca (`grep` może wykorzystywać wyrażenia regularne i przyjmować wzorce łańcuchów znaków oprócz literałów łańcuchów znaków wyszukiwania).

Polecenie `grep` to duże i potężne narzędzie, z którego będziesz korzystał prawie każdego dnia. Więcej informacji na temat `grep` znajdziesz na stronie podręcznika, gdy wpiszesz `man grep`.

Pokazaliśmy już narzędzie `grep` używane dla plików (na przykład `grep searchstring hello.txt`), ale jest to również nieoceniony składnik filtrujący w poleceniach potokowych. Przyjrzyjmy się teraz praktycznemu przykładowi.

grep i tail do monitorowania dzienników

Kiedy przeglądasz dzienniki produkcyjne, aby dowiedzieć się, co jest nie tak, często chcesz zobaczyć tylko dzienniki zawierające określone słowa kluczowe lub łańcuchy znaków wyszukiwania. W tym celu uruchom coś takiego:

```
tail -f /var/log/webapp/too_many_logs.log | grep "twójRegexWyszukiwania"
```

Ten wzorzec stale monitoruje plik dziennika pod kątem nowych wpisów, które pasują do *twójRegexWyszukiwania*, dzięki czemu możesz zobaczyć tylko dzienniki potrzebne do wykonania aktualnego zadania.

find i xargs do wykonywania operacji na grupach plików

xargs jest potężnym narzędziem, które umożliwia iterację (czyli wykonywanie pętli for) wewnątrz pojedynczego polecenia. Domyślnie xargs pobiera każdy otrzymywany fragment danych wejściowych (rozdzielony spacją, tabulatorem, nową linią i końcem pliku) i wykonuje określony program, używając tego fragmentu jako wejścia. Na przykład jeśli chcesz wyszukać określoną zawartość *tylko* w plikach zwróconych przez określone zapytanie find, możesz uruchomić następujące polecenie:

```
find . -type f -name "*.txt" | xargs grep "termin_wyszukiwania"
```

Polecenie to wyszukuje wszystkie pliki, których nazwy kończą się na .txt, a następnie używa xargs do zastosowania polecenia grep indywidualnie do każdego pliku. Ten wzorzec jest przydatny do wyszukiwania lub modyfikowania wielu plików jednocześnie. Musimy Cię uprzedzić, że xargs jest potężnym — i *dużym* — programem, zdolnym do wykonywania wielu rzeczy (w tym interpolacji łańcucha znaków do wykonywanego przez Ciebie polecenia). Nie możemy opisać tutaj tego wszystkiego, dlatego zapoznaj się ze stroną podręcznika i przeszukaj internet, aby znaleźć przykłady, jeśli jesteś w sytuacji, w której tego rodzaju funkcjonalność może Ci pomóc.

sort, uniq i odwrotne sortowanie liczbowe do przeprowadzania analizy danych

Jest to użyteczny wzorzec, który pokazaliśmy na początku rozdziału, gdzie używaliśmy go do filtrowania obszernej historii poleceń, aby uzyskać listę „najpopularniejszych X poleceń uruchamianych w tym systemie”. Podstawowy wzór jest następujący:

```
(strumień wejściowy) | sort | uniq -c | sort -rn
```

Ten przydatny do analizy danych wzorzec sortuje dane ze strumienia wejściowego, deduplikuje je podczas liczenia unikatowych wystąpień, a następnie wykonuje odwrotne sortowanie liczbowe, aby zwrócić zdeduplikowane dane, z najczęściej używanymi liniami na początku listy.

Często przycina się go za pomocą `| head -n $LICZBA`, aby uzyskać tylko określoną liczbę wyników:

```
history | awk '{print $2}' | sort | uniq -c | sort -rn | head -n 10
```

Używamy tutaj `history`, aby pobrać całą historię poleceń powłoki. Daje nam to wiele linii:

```
12 brew install --cask emacs
```

Interesuje nas tylko najczęściej używane polecenie (w tym przypadku `brew`), korzystamy więc z `awk` w celu pobrania drugiej kolumny.

Następnie sortujemy, aby duplikaty tego samego polecenia występowały obok siebie w strumieniu.

Potem usuwamy te duplikaty za pomocą `uniq`, dodając do każdego pozostałego wystąpienia licznik wystąpień. Teraz sortujemy ponownie — tym razem wykorzystując `-rn` do odwróconego sortowania liczbowego, co daje nam efekt „top X”. Na koniec wybieramy pierwsze 10 linii za pomocą polecenia `head`.

Powoduje to wypisanie wspomnianej listy 10 najczęściej używanych poleceń powłoki — na naszym komputerze wygląda to tak:

```
1000 git
115 ls
102 go
 83 gpo (alias, który ustawiłem do wysłania lokalnej gałęzi Gita do oryginalnej)
 68 make
 65 cd
 59 docker
 42 vagrant
 35 GOOS=linux
 30 echo
```

awk i sort do przeformatowywania danych i przetwarzania opartego na polach

`awk` to coś więcej niż program — to język przetwarzania strumieniowego. Jeśli pracujesz ze strumieniami danych w systemie Unix, to spędzenie kilku dni na nauce podstaw może zaoszczędzić tygodnie podczas pracy. Dobrym początkiem jest jednak użycie składni `$#` w celu odwołania się do kolumn rozdzielanych znakami niedrukowalnymi w każdej linii strumienia danych.

Przyjrzyjmy się przykładowi podania strumienia danych w następujący sposób:

```
Foo bar baz
Some data is nice
```

Gdy narzędzie `awk` widzi znak `$1`, interpretuje to jako „pierwszą kolumnę” lub w tym przypadku `Foo` w linii 1 i `some` w linii 2. `$2` to druga kolumna (`bar`, `data`) itd. Jest to niezwykle powszechna funkcja do wykorzystania podczas pracy z danymi, które są zbyt skomplikowane dla prostych poleceń `cut`:

```
cat file.txt | awk '{print $2, $1}'
```

Wygeneruje to następujące dane wyjściowe:

```
bar Foo
data Some
```

W tym przypadku dla każdego pliku wypisuje kolumnę 2 przed kolumną 1 i ignoruje wszystkie pozostałe dane z każdej linii. Jest to często wykorzystywane do przeformatowywania i porządkowania danych na podstawie określonych pól.

sed i tee do edytowania i tworzenia kopii zapasowych

Polecenie `sed` (ang. *Stream Editor*), czyli edytor strumienia, jest stosowane, gdy chcemy przekształcić strumień danych. Robisz to kilkanaście razy dziennie w edytorze tekstu, kiedy wyszukujesz i zastępujesz jakiś symbol. To polecenie jest zasadniczo wersją wiersza polecenia tej funkcjonalności: przekształca wszystkie wystąpienia `old` z pliku `file.txt` w nowy łańcuch znaków i zapisuje wynikowy strumień w nowym pliku `file.txt.changed`. Robi to bez wprowadzania zmian w oryginalnym pliku `file.txt`:

```
sed 's/old/new/g' file.txt | tee file.txt.changed
```

Chociaż edycja zawartości pliku jest łatwą demonstracją tej koncepcji, `sed` jest niezwykle przydatne do przekształcania danych strumieniowych, gdy przepływają z wyjścia jednego polecenia do wejścia drugiego:

```
(strumień wejściowy) | sed 's/old/new/g' | (następne polecenie)
```

ps, grep, awk, xargs i kill do zarządzania procesami

Chociaż `pgrep` jest dobrym narzędziem do wysyłania sygnałów do wszystkich procesów, których nazwa pasuje do wzorca, czasami jest po prostu niedostępny w systemie. Używając poniższego zestawu połączonych potokami poleceń, możesz łączyć podobne funkcjonalności (i uzyskać znacznie bardziej szczegółowe informacje docelowe, a nie tylko nazwy):

```
ps aux | grep "nazwa_procesu" | awk '{print $2}' | xargs kill
```

Polecenie `ps` rozpoczyna od listy uruchomionych procesów, które `grep` filtruje, aby uzyskać tylko te, które zawierają szukany wzorzec. Narzędzie `awk` pobiera drugą kolumnę (identyfikator procesu) dla każdej pasującej linii, a następnie podaje wszystkie dopasowane linie poleceniu `xargs` (naszej quasi pętli `for`), która wykonuje `kill` dla każdego PID. Powoduje to wysłanie sygnału `SIGTERM` do każdego dopasowanego procesu i (miejmy nadzieję) zatrzymuje go.

tar i gzip do tworzenia kopii zapasowych i kompresowania

Chociaż wiele narzędzi ma flagi, które pozwalają zrobić obie rzeczy, łączenie archiwizacji i kompresji jest kolejnym przypadkiem użycia, który ma sens. Daje to dodatkową elastyczność dodawania kolejnych poleceń do łańcucha. Jeśli chcesz na przykład dodać szyfrowanie, wystarczy tylko jedno dodatkowe polecenie w potoku:

```
tar cvf - /path/to/directory | gzip > backup.tar.gz
```

Powoduje to utworzenie skompresowanego archiwum katalogu, powszechnie używanego do tworzenia kopii zapasowych i przechowywania plików. Tego rodzaju wzorca używają też większe polecenia:

```
ssh user@mysql-server "mysqldump --add-drop-table database_name | gzip -9c" |  
↪gzip -d | mysql
```

Jest to szczególnie fajny przykład, który umożliwia zalogowanie się do serwera bazy danych za pomocą SSH, zrzuca bazę danych, kompresuje ten strumień danych, przesyła go z powrotem do lokalnej maszyny przez SSH, ponownie go dekompresuje, a ostatecznie zrzuca do lokalnego serwera MySQL.

Twoim celem nie musi być pisanie tak złożonych poleceń (lub podobnie skomplikowanych, które pokazaliśmy w tym rozdziale), ale jeśli będziesz potrafić w mgnieniu oka złożyć coś takiego, może to wydostać Cię z niektórych skrajnych tarapatów programistycznych. Mamy nadzieję, że w tym podrozdziale wykazaliśmy, iż zrozumienie podstaw przekierowywania wejścia i wyjścia, które umożliwiają systemy Unix — za pośrednictwem `<`, `>`, `>>`, `|` i ogólnie deskryptorów plików — to w zasadzie supermoce. Używaj ich mądrze.

Zagadnienia zaawansowane: sprawdzanie deskryptorów plików

W systemie Linux można łatwo *zobaczyć*, gdzie wskazują deskryptory plików procesu. W tym celu będziemy używać odrobinę magicznego wirtualnego systemu plików */proc*.

Procfs (ang. *proc virtual filesystem*), czyli wirtualny system plików *proc*, to wyłącznie linuksowa warstwa abstrakcji, która reprezentuje stan jądra i procesu w postaci plików. Dane w tych plikach pochodzą bezpośrednio z jądra systemu operacyjnego i istnieją tylko podczas ich odczytywania. Samo wylistowanie katalogu */proc* pokaże Ci wiele plików — oto niektóre z ważniejszych, zaczerpnięte ze strony wiki Arch Linux:

/proc/cpuinfo — informacja o CPU

/proc/meminfo — informacja o pamięci fizycznej

/proc/vmstats — informacja o pamięci wirtualnej

/proc/mounts — informacja o montowaniach

/proc/filesystems — informacja o systemach plików, które zostały skompilowane w jądrze i których moduły jądra są aktualnie załadowane

/proc/uptime — czas działania systemu

/proc/cmdline — wiersz poleceń jądra

Jeśli chodzi o deskryptory plików, najciekawsze dla nas jest coś, czego nie widać na powyższej liście: */proc* zawiera katalog dla każdego procesu uruchomionego na komputerze; katalog taki ma w nazwie **ID procesu (PID)**.

W katalogu `/proc` procesu deskryptory plików tego procesu są reprezentowane jako dowiązania symboliczne w katalogu o nazwie `fd`. Kiedy wyświetlisz długą listę dla tego katalogu `/proc/$PID/fd`, zobaczysz, że `1` jest pierwszym znakiem na długiej liście, który oznacza specjalny plik `link`, jak zapewne pamiętasz z rozdziału 5. „Wprowadzenie do plików”.

Praktycznie rzecz biorąc, `/proc/1/` to katalog `init` `proc` procesu, a deskryptory plików `init` można wyświetlić, wyświetlając długą listę dla `/proc/1/fd`.

Przyjrzyjmy się deskryptorom plików dla interaktywnego procesu powłoki Bash, działającej na naszym komputerze, której `ps aux | grep bash` mówi, że mamy PID 9:

```
root@server:/# ls -alh /proc/9/fd
total 0
dr-x----- 2 root root 0 Sep  1 19:16 .
dr-xr-xr-x 9 root root 0 Sep  1 19:16 ..
lrwx----- 1 root root 64 Sep  1 19:16 0 -> /dev/pts/1
lrwx----- 1 root root 64 Sep  1 19:16 1 -> /dev/pts/1
lrwx----- 1 root root 64 Sep  1 19:16 2 -> /dev/pts/1
lrwx----- 1 root root 64 Sep  5 00:46 255 -> /dev/pts/1
```

Zauważysz, że jest to interaktywna sesja powłoki: jej standardowe wejście pochodzi z wirtualnego terminala (`/dev/pts/1`), a standardowe strumienie błędów i wyjścia wracają do tego samego terminala. To się zgadza.

Rzucmy okiem na edytor tekstu taki jak `vim`, który zachowuje się podobnie do terminala — wejście i wyjście odbywa się za pośrednictwem terminala. Istnieje jednak dodatkowa komplikacja, która polega na tym, że edytory tekstu zazwyczaj mają otwarty co najmniej jeden plik do zapisania. Jak to wygląda?

W tym przykładzie uruchamiam edytor tekstu `vim` i edytuję plik w katalogu `/tmp`. Znajdźmy identyfikator procesu dla `vim`, skoro wiemy, do którego katalogu `/proc` zajrzeć:

```
root@server:/# ps aux | grep vim
root      453  0.0  0.1 17232   9216 pts/1    S+   15:57   0:00 vim
/tmp/hello.txt
root      458  0.0  0.0  2884   1536 pts/0    S+   15:58   0:00 grep
--color=auto vim
```

I oto jest: proces 453. Nie daj się zwieść poleceniu `grep`, które zawiera również `vim` w swoich argumentach. Skoro mamy PID, przyjrzyjmy się deskryptorom plików `vima`:

```
root@server:/# ls -l /proc/453/fd
total 0
lrwx----- 1 root root 64 Jan  7 15:58 0 -> /dev/pts/1
lrwx----- 1 root root 64 Jan  7 15:58 1 -> /dev/pts/1
lrwx----- 1 root root 64 Jan  7 15:58 2 -> /dev/pts/1
lrwx----- 1 root root 64 Jan  7 15:58 3 -> /tmp/.hello.txt.swp
```

Widzimy, że `stdin` (0), `stdout` (1) i `stderr` (2) wskazują na urządzenie terminala, podobnie jak na powłokę. Widzimy również, że edytor ma otwarty plik, z deskryptorem pliku 3, dowiązanym do pliku, który edytuje `vim`. Gdy proces otwiera dodatkowe pliki, tworzone są nowe deskryptory plików i można je tutaj wyświetlić.

Poza tym, że jest to interesujące samo w sobie, może się przydać, gdy programy zachowują się nieprawidłowo z powodu błędów lub gdy próbujesz prześledzić, co robi potencjalnie szkodliwy program. Narzędzie `procfs` jest dość interesujące i przydatne, jeżeli poświęcisz trochę czasu na jego naukę: na początek wpisz po prostu `man proc` lub przeczytaj stronę wiki Arch Linux <https://wiki.archlinux.org/title/Procfs>, aby uzyskać łagodniejsze wprowadzenie do tego zagadnienia.

Podsumowanie

W tym rozdziale zebraliśmy wszystkie omówione wcześniej praktyczne umiejętności i teorię, aby odblokować jedną z najpotężniejszych funkcjonalności systemów Unix i Linux: przesyłanie danych przez wiele poleceń za pomocą potoków i przekierowywania wejścia i wyjścia.

Zaczęliśmy od pokazania, w jaki sposób system operacyjny udostępnia podstawowe elementy, takie jak deskryptory plików, a potem przedstawiliśmy praktyczne zastosowania przekierowywania danych wejściowych i wyjściowych. Następnie omówiliśmy potoki, które są prawdopodobnie jedną z najbardziej przydatnych funkcji Linuksa i innych uniksowych systemów operacyjnych. Po omówieniu niezbędnej teorii i pokazaniu kilku użytecznych przykładów przyjrzelśmy się najpopularniejszym narzędziom pomocniczym, których programiści używają do okrajania i przycinania strumieni danych gromadzonych za pomocą potoków. Na koniec przedstawiliśmy kilka najczęstszych i najbardziej przydatnych wzorców i kombinacji programów używanych w prawdziwym świecie.

Materiał omówiony w tym rozdziale jest podstawą większości zaawansowanego użycia wiersza poleceń, z którym będziesz mieć styczność w codziennej pracy. Znasz już podstawowe koncepcje, narzędzia i wzorce, które napotkasz w środowisku naturalnym. Dzięki temu łatwiej Ci będzie zacząć budować niestandardowe polecenia stosowane w rozwoju oprogramowania, rozwiązywaniu problemów i automatyzacji zadań.

Aby rozwijać swoje umiejętności, wykorzystaj w codziennej pracy to, czego dowiedziałeś się podczas lektury tego rozdziału. Użyj go jako odniesienia dla wzorców do wypróbowywania i uczenia się nowych narzędzi i poleceń, które możesz dodawać do własnych receptur i stosować do filtrowania danych lub operowania nimi w wierszu poleceń. Już niedługo poczujesz się jak magik.

Automatyzacja zadań za pomocą skryptów powłoki

Zdarza się, że trzeba powtarzać kilka tych samych poleceń w kółko, niekiedy z jedynie niewielkimi zmianami. W końcu masz już dość i mówisz: „Napiszę do tego skrypt”. Jako magik wiersza poleceń wykonaj w tym celu następujące czynności:

1. Uruchom polecenie `tail -n 20 ~/.bash_history > myscript.sh`, aby utworzyć plik zawierający ostatnie 20 uruchomionych przez Ciebie poleceń powłoki Bash;
2. Następnie uruchom `bash myscript.sh`, aby wykonać ten skrypt.

Chociaż nie jest to zalecana procedura (dojdziemy do tego dalej w tym rozdziale), jest to całkowicie poprawny sposób na tworzenie i uruchamianie skryptu powłoki Bash.

Ten rozdział jest kursem dotyczącym awarii skryptów Basha. Jak każdy kurs awarii w programowaniu, jest całkowicie bezużyteczny, jeśli nie śledzisz przykładów, nie wpisujesz całego kodu samodzielnie i nie uruchamiasz go w swoim środowisku Linuksa.

Pokażemy Ci podzbiór składni Basha, która jest uważana za nowoczesną i najlepszą praktykę, a ponadto podamy wiele wskazówek z naszego z trudem wypracowanego doświadczenia na przestrzeni lat, wskazując typowe pułapki i trudne przypadki.

Bash nie jest naszym ulubionym językiem, ale czasami jest to właściwe narzędzie dla problemu, z którym się borykasz. Postaramy się również pokazać Ci, dlaczego tak jest.

W tym rozdziale omawiamy następujące zagadnienia:

- podstawy skryptów powłoki Bash;
- porównanie Basha z innymi powłokami;
- shebangi i wykonywalne pliki tekstowe;
- testowanie;
- tryby warunkowe.

Dlaczego potrzebujesz podstaw pisania skryptów powłoki Bash?

Skrypty powłoki są niezbędnym narzędziem każdego programisty, bo nawet jeśli nie będziesz pisać skryptów na co dzień, z pewnością będziesz je czytać. W tym rozdziale omówimy podstawy, które musisz znać, aby czuć się pewnie w temacie skryptów. Oto kilka przykładów:

- Kilka lat temu ktoś napisał skrypt powłoki. Twoim zadaniem może być na przykład sprawdzenie, czy można ponownie wykorzystać skrypty automatyzacji, które Steve napisał przed odejściem do Google’a.
- Widzisz możliwość napisania własnego skryptu powłoki, gdy masz zadanie, które już rozwiązały istniejące programy powłoki (filtrowanie, wyszukiwanie, sortowanie i podawanie wyników z jednego programu do drugiego).
- Chcesz dokładnie kontrolować to, co jest umieszczane w poszczególnych warstwach Dockera podczas tworzenia obrazu.
- Musisz koordynować inne oprogramowanie w kontekście systemu operacyjnego serwera Linux: kolejność uruchamiania, sprawdzanie błędów, wcześniejsze przerywanie działania między kolejnymi programami itd.

Istnieje mnóstwo przypadków użycia, w których skrypt powłoki jest odpowiednim rozwiązaniem dla danego problemu. Dzięki lekturze tego rozdziału będziesz mieć umiejętności potrzebne do pisania takich niestandardowych skryptów.

Podstawy

Powłoki Bash można się nauczyć jak każdego innego języka programowania. Ma ona swoje środowisko (uniksove lub linuksowe), standardową bibliotekę (dowolny zainstalowany w systemie program oparty na CLI), zmienne, przepływ sterowania (pętle, testowanie i iteracja), interpolację, kilka wbudowanych struktur danych (tablice, łańcuchy znaków i wartości logiczne) itd.

W tej książce przyjęliśmy założenie, że jesteś programistą i dlatego wiesz, jak programować, więc zamiast uczyć Cię tych standardowych funkcjonalności języka programowania, pokażemy Ci po prostu, jak wyglądają one w powłoce Bash; damy też kilka porad dotyczących idiomatycznego użycia (lub powszechnego niewłaściwego użycia).

Zmienne

Jak każdy język programowania, Bash ma zmienne, które mogą być puste lub mieć ustaloną określoną wartość. Nieustawione zmienne są po prostu „puste”, a Bash chętnie będzie ich używać bez wywoływania stanu paniki, chyba że ustawisz opcję `-u` (błąd przy nieustawionych zmiennych) za pomocą polecenia `set -u`.

Ustawienia

Aby ustawić zmienną, należy użyć znaku równości.

Polecenie `FOOBAR=NICE` spowoduje ustawienie wartości `NICE` dla zmiennej `FOOBAR`.

W powłoce Bash nie ma żadnych typów — jest ona tak bardzo nietypowana, jak tylko może być język programowania.

Sam symbol zmiennej może zawierać litery, cyfry i podkreślenia, ale nie może zaczynać się od cyfry.

Powszechną praktyką jest zapisywanie nazw zmiennych środowiskowych wielkimi literami, a małymi literami zmiennych w skryptach Basha. Do oddzielania poszczególnych słów używa się podkreślenia. Nazwa może zawierać cyfrę, ale nie powinna się od niej rozpoczynać. Podobnie jak w przypadku innych języków dobrą praktyką jest, aby nazwy wskazywały, do czego służą zmienne i czy są one stałe, a także stosowanie nazw w liczbie mnogiej dla tablic:

- niedozwolone: `%foo&bar=bad;`
- niedozwolone: `2foo_bar=bad;`
- dozwolone, ale niezbyt dobra praktyka: `foo_BAR123=still_very_bad;`
- dobra zmienna środowiskowa: `PORT=443;`
- dobrze: `local_var=512;`
- dobra zmienna środowiskowa: `FOO_BAR123=good;`
- dobra zmienna lokalnej tablicy: `words=(foo bar baz).`

Pobieranie

Aby użyć zmiennej, należy odwołać się do niej za pomocą znaku `$`:

```
$ echo $FOOBAR
nice
```

Porównanie Basha z innymi powłokami

Istnieje ogromna różnorodność programów powłoki dla środowisk uniksopodobnych; można powiedzieć, że jednym z głównych powodów popularności Uniksa jest fakt, iż zawsze było to środowisko, w którym zasadniczo nie ma żadnych barier dla skryptów i automatyzacji.

W tym rozdziale pokażemy, jak pisać własne skrypty w powłoce Bash. Spora część zawartego tu materiału będzie działać również w innych powłokach (na przykład takich jak `ksh` i inne typowe minimalistyczne powłoki, które znajdziesz w katalogu `/bin/sh`), ale skupimy się na Bashu.

Jeśli piszesz skrypt powłoki, Bash zapewnia idealną równowagę między szeroką dostępnością a zestawem funkcjonalności językowych, który jest wystarczająco obszerny, aby wygodnie pisać niewielkie programy.

Shebangi i wykonywalne pliki tekstowe

W systemach uniksopodobnych skrypt to po prostu wykonywalny plik tekstowy. System operacyjny (w Linuksie często nazywany jądrem) analizuje pierwszą linię, aby określić, do którego interpretera należy wprowadzić zawartość pliku.

Pierwsza linia to tak zwany shebang (lub hashbang) i składa się ze znaków kratki i wykrzyknika (`#!`), po których następuje ścieżka do interpretera używanego do wykonania kodu pliku. Oto przykładowa linia shebang:

```
#!/usr/bin/env bash
```

Kiedy jądra systemów uniksopodobnych uruchamiają plik z ustawionym bitem trybu wykonywalnego, przyglądają się pierwszym bajtom. Mogą one zawierać magiczną liczbę. Liczba ta może być częścią plików binarnych lub jakimś czytelnym dla człowieka znakiem, jak w shebangu. Jądro wykorzystuje te informacje, aby zweryfikować, czy istnieje właściwy sposób ich wykonania. Zapobiega to na przykład sytuacjom, w których jądro próbuje wykonać plik obrazu i ulega awarii. W zależności od systemu jądro lub powłoka upewniają się, że następujące po shebangu polecenie jest wykonywalne. Program `env` uruchomi polecenie i weźmie pod uwagę zmienną środowiskową `PATH`, aby znaleźć i wykonać `bash`.

Chociaż w większości języków skryptowych znak kratki oznacza komentarz i w związku z tym jest ignorowany przez interpreter, ten specjalny komentarz na początku pliku informuje system operacyjny, które polecenie należy uruchomić, aby zinterpretować resztę pliku. Oto kilka typowych przykładów, z którymi się zetkniesz:

- `#!/bin/sh` — użycie tego konkretnego programu powłoki z tej konkretnej lokalizacji systemu plików;
- `#!/usr/bin/python3` — użycie tego konkretnego pliku binarnego Pythona;
- `#!/usr/bin/env python` — użycie programu `env`, aby określić, które pliki binarne Pythona stosować w tym środowisku (różne systemy mogą mieć różne wersje tego samego programu zainstalowane w różnych ścieżkach).

Chociaż będziesz napotykać wszystkie te warianty, najlepszą opcją jest zawsze korzystanie z `/usr/bin/env` w celu zapewnienia przenośności. Wyjątkiem jest tutaj `/bin/sh`, ponieważ każdy system kompatybilny z interfejsem POSIX musi mieć w tej lokalizacji powłokę kompatybilną z tym interfejsem.

Typowe ustawienia powłoki Bash (opcje i argumenty)

Ponieważ linia shebang jest wykonywana jako polecenie, można przekazywać również argumenty. I chociaż dobrym pomysłem może być zachowywanie prostoty, częstym motywem jest przekazywanie powłokom dodatkowych argumentów, zwłaszcza powłoce Bash, która jest często wykorzystywana do pisania dużych skryptów, gdyż ma dodatkowe funkcjonalności w porównaniu z mniejszymi powłokami znajdującymi się zwykle w katalogu `/bin/sh`.

W skryptach Basha często zobaczysz przekazywane argumenty, takie jak `-e`, `-u`, `-x` czy `-o pipefail`. Te argumenty można znaleźć w samej linii shebang:

```
#!/usr/bin/env bash -euxo pipefail
```

Można je wpisać także w następnym poleceniu przy użyciu polecenia `set` powłoki Bash, które ustawia argumenty lub opcje:

```
#!/usr/bin/env bash
```

```
set -eu -o pipefail
```

Ustawienie tych opcji sprawia, że Bash zachowuje się nieco bardziej jak języki programowania, do których jesteś przyzwyczajony. Oto konsekwencje zastosowania tych opcji:

- natychmiastowe zakończenie działania, jeśli nie powiedzie się którykolwiek z elementów potoku poleceń;
- traktowanie nieustawionych zmiennych jako krytycznych błędów.

Oto analiza dokumentacji dla tych opcji wraz z użyteczną opcją debugowania (`-x`) jako bonusem:

- `-o pipefail` — podczas korzystania z potoków zapewnia, że błędy występujące w potoku będą przekazywane. Jeśli wystąpi więcej niż jeden błąd, zostanie użyty znajdujący się najbardziej po prawej.
- `-e` — jeśli wystąpi błąd lub polecenie się nie powiedzie, zapewni to natychmiastowe zamknięcie skryptu powłoki.
- `-u` — wyrzuci błąd, jeżeli użyte zostaną jakieś nieustawione zmienne.

Do debugowania przydatna jest opcja `-x`:

- `-x` — umożliwi śledzenie. Oznacza to, że przed wykonaniem każde polecenie jest zapisywane w standardowym strumieniu błędów.

Wszystkie argumenty, z wyjątkiem `-o pipefail`, można znaleźć w większości powłok Uniksa. Więcej informacji na temat opcji dostępnych w powłoce Bash znajdziesz na stronie podręcznika: <https://manpages.org/bash>.

/usr/bin/env

Oto coś, o czym należy pamiętać: `/bin/sh` jest standardową ścieżką w interfejsie POSIX prowadzącą do dowolnej powłoki zgodnej z tym interfejsem. Możesz być pewien, że znajdziesz ją w *każdym* systemie linuksowym lub uniksowym. Zazwyczaj nie jest to `bash`, lecz bardziej minimalistyczna powłoka, oferująca tylko tyle funkcjonalności, aby spełnić wymagania standardu POSIX, które pozwalają pisać bardzo przenośne skrypty powłoki. Dla wszystkich innych powłok i interpreterów, których Twój skrypt może wymagać, w każdym pozostałym przypadku najlepiej jest skorzystać z prefiksu `#!/usr/bin/env`. Zagwarantuje to użycie poprawnej ścieżki ze zmiennej `PATH` i zapobiegnie wystąpieniu błędu *command not found* (nie znaleziono polecenia), gdy plik binarny nie będzie znajdował się w `/usr/bin/`.

Istnieją różne scenariusze, w których `/usr/bin/bash` lub `/bin/bash` nie będą właściwymi ścieżkami. Oto przykłady:

- Menedżer pakietów lub skrypty konfiguracyjne charakterystyczne dla danej firmy często instalują interpreter w innym miejscu, niż znajduje się on w Twoim systemie programistycznym.
- Jeśli ktoś instaluje oprogramowanie ręcznie, aby na przykład obejść lub odtworzyć błąd, często umieszcza wynikowy plik binarny w lokalizacji `/usr/local`.
- Wirtualne środowiska różnych języków skryptowych umieszczają pliki binarne w podkatalogu każdego *projektu lub repozytorium* kodu źródłowego.
- Osoby instalujące interpreter bez uprawnień administratora, na przykład w swoim katalogu domowym.
- Osoby korzystające z różnych wersji menedżera wersji dla interpretera (`rvm`, `nvm` itd.).
- Różne uniksopodobne systemy operacyjne i niektóre dystrybucje Linuksa nie instalują zewnętrznych pakietów w katalogu `/usr/`.

Chociaż wiele osób nie może sobie wyobrazić, że ich skrypty kiedykolwiek skończą w tak niestandardowym miejscu, są duże szanse, iż w końcu się na to natkniesz. Zamiast ryzykować zepsuciem oprogramowania w takich przypadkach, dobrze jest nabrać nawyku pisanie w skryptach `/usr/bin/env bash` (lub jakiegokolwiek interpretera, dla którego napisany jest kod). Dzięki temu żaden programista — w tym Ty o trzeciej nad ranem — nie będzie musiał przeglądać plików źródłowych, wyszukiwać lub wprowadzać w nich zmian oraz rozwiązywać problemów, gdy się rozpadną z powodu niewielkiej zmiany w środowisku.

Znaki specjalne i znaki ucieczki

Jednym ze znaków specjalnych, z których będziesz często korzystać, jest symbol kratki (`#`). Wszystko, co znajduje się w linii poprzedzonej tym symbolem, jest komentarzem ignorowanym przez interpreter.

Pozostałe znaki w Bashu mają specjalne znaczenie i gdy używa się ich jako fragmentu wartości zmiennej, trzeba stosować do nich znak ucieczki (czyli znak modyfikacji) w postaci wstecznego ukośnika (`\`). Oto niektóre z tych znaków:

- znaki cytowania (" i ');
- nawiasy klamrowe, kwadratowe i okrągłe ({, }, [,] i (,));
- znaki mniejszości i większości (< i >);
- tylda (~);
- gwiazdka (znak glob w Bashu) (*);
- ampersand (&);
- znak zapytania (?);
- typowe operatory (!, =, | itd.).

Ich modyfikację (ucieczkę) przeprowadza się tak, jak w większości innych języków programowania:

```
$ F00="jaa\$\""
```

Podstawianie poleceń

Jedną z zalet i głównych zastosowań skryptów powłoki jest to, że każde polecenie jest łatwo dostępne. Bardzo częstym przykładem jest podstawianie poleceń. Jest ono przydatne, gdy chcesz użyć wyjścia z co najmniej jednego polecenia. Można to zrobić za pomocą podstawiania poleceń:

```
echo "Teraz jest $(date)"
```

Powoduje to wykonanie tych poleceń — w tym przypadku jedynie `date`, ale może to być również złożone wyrażenie, które połączyłeś za pomocą potoków. Innym sposobem na osiągnięcie tego samego celu jest użycie grawisów. Poniższy przykład będzie miał ten sam wynik:

```
echo "Teraz jest `date`"
```

Testowanie

Pokazane tutaj polecenia testowe są zwykle używane wraz z instrukcjami przepływu sterowania `if-else`. Zarówno funkcja testowania łańcuchów znaków (`[{}`), jak i funkcja testowania arytmetycznego (`(())`) zwracają 0, jeśli test ewaluuje do wartości `true`, lub 1, gdy test ewaluuje do wartości `false`. Wynika to z zerowego kodu wyjścia wskazującego powodzenie i różni się w stosunku do większości znanych Ci języków programowania, które zazwyczaj ewaluują wartość 0 jako `false`. W Bashu nie ma natywnego typu danych `boolean`; liczby całkowite 0 i 1 są używane w kontekstach boolowskich, takich jak ten. Czasami zmienne `true` i `false` są inicjalizowane i używane w całym skrypcie.

Operatory testowe

Oto kilka podstawowych operatorów logicznych, których możesz użyć do konstruowania instrukcji w Bashu (zasadniczo zgodnie z przyzwyczajeniem z innych języków):

- `!` — not (negacja),
- `&&` — and (i),
- `||` — or (lub).

Operatory te mogą być stosowane zarówno z łańcuchami znaków, jak i typami testów arytmetycznych:

- `==` — jest równy,
- `!=` — nie jest równy.

[[testowanie plików i łańcuchów znaków]]

Złożone polecenie `[[` umożliwia wykonywanie (i łączenie) porównań łańcuchów znaków. Jak wspomnieliśmy wcześniej, Bash nie ma żadnego rodzaju ścisłych typów danych, które znasz z innych języków programowania, dlatego nazywamy je porównaniami łańcuchów znaków, ponieważ jest to znajoma programistom koncepcja.

Jeśli katalog domowy użytkownika nie istnieje, utwórz go:

```
if [[ ! -d $HOME ]]; then
    echo "Tworzenie katalogu domowego: ${HOME}..."
    mkdir -p $HOME
    echo "done"
fi
```

Ten znak `!` jest negacją Basha, możesz więc przeczytać linię tego przykładu tak: *jeśli nie ma katalogu \$HOME, to...*

Oto nieco bardziej skomplikowany przykład. Jeżeli katalog domowy użytkownika nie istnieje *lub* jeśli zmienna `ALWAYSCREATE` ma ustawioną wartość `yes`, utwórz katalog domowy:

```
ALWAYSCREATE=yes

if ! [[ -d $HOME ]] || [[ $ALWAYSCREATE == yes ]]; then
    echo "Tworzenie katalogu domowego: ${HOME}..."
    mkdir -p $HOME
    echo "done"
fi
```

Przydatne operatory do testowania łańcuchów znaków

- `-z` — brak ustawienia (stosowany do zmiennych);
- `-n` — wartość niezerowa (set używane dla zmiennych);
- `==` — lewy operand, który pasuje do wyrażenia regularnego (prawego operandu), na przykład `[[foobar == f*bar]]`.

Przydatne operatory do testowania plików

- `d` — katalog;
- `e` — istnieje;
- `-f` — zwykły plik;
- `-S` — plik gniazda;
- `-W` — możliwy do zapisania z perspektywy tego procesu Basha.

((testowanie arytmetyczne))

Wyrażenie arytmetyczne ewaluowane w teście w podwójnych nawiasach `(())` ustawia wartość wyjściową testu na 1, jeżeli wyrażenie ewaluuje do wartości 0, w przeciwnym razie zwraca status wyjścia 0. To sprawia, że testowanie jest dość intuicyjne dzięki użyciu operatorów, które znasz praktycznie z każdego innego języka programowania:

- `>` `>=` — większe niż i większe niż lub równe;
- `<` `<=` — mniejsze niż i mniejsze niż lub równe;
- `==` — test równości.

`(($SOME_NUMBER == 24))` jest dość prostym testem arytmetycznym. Zobaczmy, jak się zachowuje.

Dla liczby 24:

```
→ SOME_NUMBER=24
→ (( $SOME_NUMBER == 24 ))
→ echo $?
0
```

Polecenie `echo $?` wypisuje status wyjścia poprzedniego polecenia, co pozwala zobaczyć, jaka była rzeczywista ewaluacja tego testu arytmetycznego. W przypadku innych wartości, w tym wartości nieliczbowych, wygląda to tak:

```
→ SOME_NUMBER=foobar
→ (( $SOME_NUMBER == 24 ))
→ echo $?
1
```

Jeśli wartość `$SOME_NUMBER` jest nieustawiona (na przykład `[[-z $SOME_NUMBER]]`):

```
→ unset SOME_NUMBER
→ (( $SOME_NUMBER == 24 ))
zsh: bad math expression: operand expected at `== 24 '
```

Podsumujmy:

- `(($SOME_NUMBER == 24))` zostanie ewaluowane do wartości 0, jeśli zmienna `SOME_NUMBER` będzie miała ustawioną wartość 24.
- Jeżeli zmienna `$SOME_NUMBER` będzie miała ustawioną wartość *inną niż 24* (włącznie z wartością nieliczbową), test zostanie ewaluowany do wartości 1.

- Jeśli zmienna `$SOME_NUMBER` będzie *nieustawiona*, pojawi się błąd, ponieważ test arytmetyczny nie będzie miał lewego operandu do wykorzystania w celu porównania.

Wyrażenia warunkowe: `if/then/else`

Instrukcję `if` powłoki Bash można zwykle napotkać w takiej postaci:

```
if [[ $TEST ]]; then $INSTRUKCJE else $INNE_INSTRUKCJE fi
```

Zanim przyjrzymy się przykładom, zapamiętaj kilka kwestii dotyczących tej formy:

- `if` rozpoczyna, a `fi` kończy blok `if`.
- `;` rozdziela instrukcje w Bashu; dodaj zaraz po teście.
- `[[i]]` ograniczają wyrażenie testowe.
- Klauzula `else` jest opcjonalna.

Oto jak wygląda instrukcja `if` w Bashu:

```
if [[ -e "example.txt" ]]; then  
    echo "Plik istnieje!"  
fi
```

`if-else`

Jeśli chcesz dodać klauzulę `else` do tej struktury, możesz to zrobić!

```
if [[ -e "example.txt" ]]; then  
    echo "Plik istnieje!"  
else  
    echo "Plik nie istnieje!"  
fi
```

Pętle

Ogólny format pętli w Bashu jest następujący: `for/do/done`. Obsługują one również instrukcje `break` i `continue`, z których pierwsza powoduje wyjście z pętli, a druga przejście do następnej iteracji.

Pętle w stylu C

Bash obsługuje pętle w stylu języka C, z wyrażeniem inicjalizującym, wyrażeniem warunkowym i wyrażeniem zliczającym:

```
for (( i=0; i<=9; i++ ))  
do  
    echo "Zmienna i pętli to aktualnie $i"  
done
```

for...in

Spójrzmy na iterację za pomocą pętli `for...in`. Spróbuj uruchomić w swojej powłoce następujące elementy:

```
for i in 1 2 3 4 5
do
    echo $i
done
```

Oto pętla z przepływem sterowania wewnątrz:

```
for os in FreeBSD Linux NetBSD "macOS" DragonflyBSD
do
    echo "Sprawdzanie ${os}..."
    if [[ "$os" == 'NetBSD' ]]; then
        echo "(Jestem przekonany, że dałoby się to uruchomić na moim toasterze)"
    fi
    sleep 1
done
```

While

Kolejną typową strukturą sterowania, którą możesz znać z innych języków programowania, jest pętla `while`. W Bashu działa bardzo podobnie. Aby wyjść z pętli, można użyć instrukcji `break`.

Poniższy skrypt odczytuje plik `lines.txt` linia po linii, dopóki nie napotka linii `STOP`. Ostatnia linia pokazuje również, jak można przekazać plik do pętli. Polecenie `read` zajmuje się przetwarzaniem tego pliku linia po linii:

```
file="lines.txt"

while read line; do
    if [[ $line == "STOP" ]]; then
        echo "Napotkano STOP. Wychodzenie z pętli."
        break
    fi

    echo "Przetwarzanie: $line"
    # Tu można dodać dodatkowe polecenie do przetwarzania $line
done < "$file"
```

Eksportowanie zmiennych

Wyeksportowanie zmiennej przez poprzedzenie jej prefiksem `export` sprawia, że wszelkie podpowłoki uruchomione przez proces skryptu również będą miały dostęp do wartości tej zmiennej. Jest to sposób na zagwarantowanie, by zmienna była propagowana w dół do wszelkich przyszłych podzakresów (lub podprzestrzeni nazw) zakresu zmiennej powłoki lub przestrzeni nazw bieżącej powłoki.

Ustaw zmienną w powłoce:

```
MYDIR=$HOME
```

Utwórz i uruchom ten skrypt (uwaga: ta operacja się nie powiedzie!):

```
#!/usr/bin/env bash
```

```
LISTING=$(ls "${MYDIR}/Documents")  
echo $LISTING
```

Zobaczysz błąd `ls: /Documents: No such file or directory` (brak takiego pliku lub katalogu), ponieważ uruchomienie tego skryptu utworzyło podpowłokę, która nie miała dostępu do niewyeksportowanych zmiennych z jej powłoki nadrzędnej (powłoki, która utworzyła tę podpowłokę, innymi słowy powłoki interaktywnej). Umożliwienie dostępu do zmiennych podpowłokom musi być dokonane bezpośrednio, za pomocą słowa kluczowego `export`:

```
export MYDIR=$HOME
```

Gdy wyeksportujesz tę zmienną, uruchom ponownie ten przykładowy skrypt, a zobaczysz, że może on teraz uzyskać dostęp do zmiennej `MYDIR`.

Funkcje

Ogólnie zalecamy, abyś do czasu, gdy będziesz potrzebować funkcji Basha, znalazł inny język, w którym będziesz mógł napisać swój rozwijający się program. Czasami jednak Bash jest właściwym językiem dla danego problemu i chcemy pokazać Ci absolutne podstawy z zaleceniami dotyczącymi ich użycia.

Zdefiniuj funkcję za pomocą słowa kluczowego `function`:

```
function my_great_function {  
    $EXPRESSIONS  
}
```

Funkcje wywołuj przez wpisywanie ich nazw:

```
my_great_function
```

Preferuj zmienne lokalne

Bash działa z mniej lub bardziej globalnym zakresem — dokładniej: zakresem na (pod)powłokę. Wiele nowoczesnych języków programowania daje oddzielny zakres funkcji do pracy, więc stan funkcji nie zanieczyszcza globalnego stanu po wyjściu funkcji.

Używanie zmiennych `local` w funkcjach uchroni Cię przed tym i zalecamy ich stosowanie:

```
#!/usr/bin/env bash
```

```
important_var=somevalue
```

```
function local_var_example() {
```

```
    local important_var="zmień to lokalnie, nie przejmuj się"
    echo "local_var_example: ${important_var}"
}

function bad_example() {
    important_var="to mutuje zmienną lokalną, ponieważ jestem zły i powinienem
    ↪czuć się źle"
    echo "bad_example: ${important_var}"
}

echo "przed funkcjami: ${important_var}"
local_var_example
echo
echo "po local_var_example: ${important_var}"
echo
bad_example
echo "po bad_example: ${important_var}"

exit 0
```

Uruchom ten kod samodzielnie i zobacz, jaką różnicę robi użycie lokalnych vars.

Przekierowanie wejścia i wyjścia

Kiedy uruchamiasz skrypty, często chcesz przekierować ich wyniki:

- do innego programu (za pomocą potoku (|) — więcej informacji na ten temat znajdziesz w rozdziale 11. „Potoki i przekierowanie”);
- do zwykłego pliku (np. pliku dziennika);
- do specjalnej lokalizacji, takiej jak `/dev/null`, która może działać jako coś w rodzaju czarnej dziury dla niepotrzebnych już danych.

Dalej przedstawiamy najczęstsze, oprócz potoków, sztuczki przekierowywania wejścia lub wyjścia, które można napotkać w kodzie.

< — przekierowanie wejścia

Jest często używane do pobierania danych wejściowych z pliku zamiast z powłoki generującej proces:

```
grep foobar < stuff.txt
```

> i >> — przekierowanie wyjścia

Symbol `>` będzie przesyłał strumieniowo dane wyjściowe do dowolnego miejsca, w które go skierujesz, zastępując wszystko, co już się tam znajduje, jeśli jest to zwykły plik:

```
ps aux | grep foo > /var/log/foo_overwrite.log
```

Za każdym razem gdy to uruchomisz, wyjście z `ps aux | grep foo` zostanie zapisane w pliku `/var/log/foo_overwrite.log` i zastąpi jego istniejącą zawartość.

Użycie zamiast tego opcji `>>` spowoduje *dołączenie* danych do pliku wyjściowego i pozostawienie jego zawartości nienaruszonej (zwykle właśnie tego chcemy dla plików dziennika):

```
echo $(date && cat /proc/stat) >> /var/log/kernelstate.log
```

Zastosowanie `2>&1` do przekierowywania `STDERR` i `STDOUT`

Czasami chcemy przekierować do pliku zarówno standardowe wyjście, jak i standardowy strumień błędów:

```
consul agent -dev >> /var/log/consul.log 2>&1 &
```

To polecenie uruchamia rozwiązanie Consul firmy Hashicorp w trybie programistycznym i sprawia, że proces działa w tle (symbol `&` na końcu), przekierowując standardowe dane wyjściowe do pliku dziennika. Opcja `2>&1` instruuje powłokę Bash, aby „przekierowała deskryptor pliku 2 (`STDERR`) do tego samego miejsca co 1 (`STDOUT`)” — w tym przypadku jest to plik `/var/log/consul.log`.

Znasz już deskryptory plików: `STDIN`, `STDOUT` i `STDERR`. A co zrobić, jeśli chcesz jedynie przekierować standardowy strumień błędów do *innego* pliku niż standardowe wyjście?

Składnia interpolacji zmiennej — `${}`

Aby osiągnąć to, co w większości języków programowania jest znane jako interpolacja łańcuchów znaków (podstawienie wartości zmiennej pod fragment łańcucha znaków), możesz użyć interpolacji zmiennej Basha, czyli `${}`.

Spróbujmy:

```
MYNAME=dave
echo "Nie mogę tego zrobić, ${MYNAME}."
```

W Bashu istnieją także inne metody interpolacji zmiennych, ale jest to nasz ulubiony sposób, ponieważ charakteryzuje się najmniejszym ryzykiem zepsucia programu ze względu na nieoczekiwane ukształtowanie wejścia (spacje, znaki specjalne itp.).

Skorzystaj z tej składni, jeśli zamierzasz użyć zmiennej jako wartości przypominającej łańcuch znaków — nawet jeśli ta zmienna jest sama i tak naprawdę nie musi być interpolowana do innego łańcucha znaków:

```
NAME="${MYNAME}"
```

Zapobieganie to wielu dziwnym błędom i zachowaniom Basha, więc jest to dobry nawyk, który warto sobie wyrobić.

Uwaga

Podczas pracy z interpolacją zmiennych prawie zawsze trzeba uruchomić Bash z opcją `-u` (wywołując ją z `-u` albo używając `set -euo pipefail` na początku skryptów, jak zalecamy). Zapobiegnie to konieczności sprawdzania wartości zerowych przed użyciem danej zmiennej.

Ograniczenia skryptów powłoki

Bash ma niezliczone funkcjonalności i wielu z nich tutaj nie omawiamy. Jeżeli chcesz zgłębić zagadnienie języka i środowiska powłoki Bash, istnieje wiele książek i mnóstwo darmowych zasobów w internecie. Strona podręcznika Basha (`man bash`) to również dobry początek, nawet teraz, gdy już nieco się orientujesz w tym temacie.

Zapewne w trakcie swojej kariery napotkasz wiele skryptów Basha. Najprawdopodobniej poświęcisz jednak więcej czasu na czytanie i rozszyfrowywanie istniejących skryptów niż na pisanie nowych, obszernych programów Basha. Bash jest doskonałym narzędziem w przypadku małych problemów i zadań systemowych, nadających się do rozwiązania za pomocą istniejącego oprogramowania, które trzeba jedynie połączyć w określoną receptę. Często jest to jednak *fatalny* pomysł w przypadku dużych problemów, które wymagają więcej niż połączenie standardowych programów linuxowych i unixowych.

Oto stwierdzenia prawdziwe w przypadku Basha:

- Małe jest lepsze niż duże.
- Czytelne jest lepsze niż sprytne.
- Lepiej zachować bezpieczeństwo niż potem żałować.

W miarę rozrastania się skryptów Basha ich zastępowanie narzędziami napisanymi w innym języku programowania (często w Pythonie) nie jest niczym niezwykłym. Nie chodzi nam o kwestionowanie roli Basha! Idealnie wypełniania on swoją niszę, dlatego jest tak rozpowszechniony. Jeśli od czasu do czasu przestaniesz zadawać sobie pytanie, czy skrypt Basha jest nadal właściwym rozwiązaniem danego problemu, nie będzie w tym nic złego.

Podsumowanie

Ten rozdział był kursem szybkiego pisania skryptów Basha. Jest w nim mnóstwo informacji, ale obejmuje wszystkie podstawy, których potrzebujesz, aby stać się skutecznym programistą skryptów Basha. Jeśli musisz, popracuj z nim więcej. Oprócz składni omówiliśmy najlepsze naszym zdaniem praktyki, które sprawiają, że pisanie w świecie rzeczywistym czytelnych, łatwych w utrzymaniu skryptów staje się prostsze (lub przynajmniej możliwe).

Ćwicz, ćwicz i jeszcze raz ćwicz — najlepiej na rzeczywistych problemach, a nie tylko na wymyślonych przykładach. Nie ma szybszego sposobu, aby stać się w czymś dobrym.

Źródła

Bash test and comparison functions (wykorzystane dla opcji tablic `[[` i `()`), <https://developer.ibm.com/tutorials/l-bash-test/> [dostęp: 25 września 2022].

Bezpieczny dostęp zdalny za pomocą SSH

SSH (ang. *Secure Shell Protocol*) to jak szwajcarski scyzoryk — narzędzie do wszystkiego: do tworzenia bezpiecznych połączeń i tunelowego przesyłania przez nie danych. Podczas swojej kariery będziesz używać SSH po trochu do wszystkiego. Oto kilka przykładów:

- bezpieczne logowanie do zdalnego systemu;
- klonowanie prywatnego repozytorium Gita;
- przesyłanie plików z laptopa na serwer lub między serwerami;
- mapowanie usługi internetowej uruchomionej za VPN-em na lokalny port w laptopie, aby mogli z niej korzystać użytkownicy sieci domowej;
- różne inne zadania, które obejmują tunelowanie ruchu lub wysyłanie plików przez wiele połączeń sieciowych.

W tym rozdziale przedstawimy wszystko, czego będziesz potrzebować, aby nabrać biegłości w podstawowych kwestiach. Dowiesz się, na czym polega kryptografia klucza publicznego, jest ona bowiem niezbędna, aby zrozumieć działanie i stosowanie tego rodzaju narzędzi. Będziesz tworzyć klucze SSH i wykorzystywać je do logowania się na zdalnym serwerze. Aby pomóc Ci ugruntować podstawy, przygotowaliśmy nawet mały projekt, w którym skonfigurujesz loginy oparte na kluczach — będą one wymagane dla zdalnego hosta.

Dla przypadków, w których nieuchronnie możesz mieć problemy podczas korzystania z SSH, zebraliśmy niektóre z najpowszechniejszych błędów, z jakimi możesz się zetknąć. Dowiesz się, co wskazują najczęstsze komunikaty o błędach i jak korzystać z wbudowanej opcji „debugowania” SSH, aby rozwiązywać związane z nimi problemy.

W tym rozdziale omawiamy następujące zagadnienia:

- elementarz kryptografii klucza publicznego;
- szyfrowanie komunikatów;
- podpisywanie komunikatów;
- klucze SSH;
- konwersja kluczy SSH2 do formatu OpenSSH;
- przesyłanie plików;

- tunelowanie SSH;
- plik konfiguracyjny.

Przejdźmy od razu do podstaw tego, co musisz wiedzieć na temat kryptografii klucza publicznego, aby reszta tego rozdziału nie stanowiła dla Ciebie czarnej magii.

Elementarz kryptografii klucza publicznego

Być może w swojej karierze zawodowej natknąłeś się już na zagadnienie kryptografii klucza publicznego. Chociaż kryptografia jest osobną dziedziną, a niniejsza książka nie jest o kryptografii, ważne jest zrozumienie podstaw. Na szczęście podstawowe pojęcia są bardzo proste, a ich znajomość jest wystarczająca w większości przypadków. Ten podrozdział będzie możliwie jak najkrótszy, a potem przejdziemy do poleceń potrzebnych do konfigurowania i korzystania z bezpiecznego dostępu za pomocą SSH.

Kryptografia klucza publicznego to system, który używa dwóch oddzielnych kluczy: publicznego i prywatnego. Razem tworzą one parę kluczy. Jak wskazują ich nazwy, klucz publiczny można udostępnić wszystkim, natomiast klucz prywatny koniecznie musi pozostać prywatny.

Szyfrowanie komunikatów

Po utworzeniu pary kluczy każdy komunikat zaszyfrowany przy użyciu klucza publicznego z tej pary może zostać odszyfrowany za pomocą odpowiedniego klucza prywatnego.

Wyobraź sobie, że Alicja chce wysłać Robertowi zaszyfrowaną wiadomość. W tym celu Alicja pobiera klucz publiczny Roberta i używa go do zaszyfrowania wiadomości. Następnie wysyła Robertowi tę zaszyfrowaną wiadomość. Ponieważ Robert jest właścicielem odpowiedniego klucza prywatnego, będzie w stanie odszyfrować i odczytać tę wiadomość. Jeśli zaś ktoś inny zobaczy zaszyfrowaną wiadomość, nie będzie mógł jej odszyfrować i odczytać, gdyż tylko Robert ma odpowiedni klucz prywatny.

Z tego powodu ważne jest, aby *nigdy* nie udostępniać swojego klucza prywatnego innym osobom. Byłoby to naruszeniem bezpieczeństwa.

Podpisywanie komunikatów

Istnieje jeszcze inny sposób użycia tych dwóch kluczy: można je wykorzystywać do podpisywania komunikatów. „Podpisywanie” komunikatów to sposób na kryptograficzne udowodnienie, że dany komunikat został naprawdę napisane przez osobę posiadającą klucz. Komunikat zaszyfrowany kluczem publicznym może zostać odszyfrowany kluczem prywatnym, ale działa to również w drugą stronę: komunikat *zaszyfrowany* kluczem prywatnym może zostać *odszyfrowany* za pomocą klucza publicznego.

Kiedy Alicja chce podpisać wiadomość, klucz prywatny może zostać wykorzystany do zaszyfrowania wiadomości (czyli kryptograficznego zabezpieczenia skrótu wiadomości). Każda osoba posiadająca klucz publiczny Alicji może go użyć do odszyfrowania wiadomości, a jeśli to zadziała, dana osoba będzie wiedziała, że wiadomość musiała zostać zaszyfrowana kluczem prywatnym Alicji.

Oba mechanizmy, szyfrowanie i podpisywanie, są często stosowane razem. Ponadto same podpisy są nierzadko używane do zapewnienia bezpieczeństwa (przez weryfikację autorstwa) podczas pobierania oprogramowania, na przykład za pośrednictwem menedżerów pakietów lub sklepów z aplikacjami.

Warto zauważyć, że te podstawowe mechanizmy szyfrowania i podpisywania z wykorzystaniem kluczy publicznych są używane do niemal wszystkiego — od szyfrowania wiadomości e-mail, przez zabezpieczanie ruchu internetowego HTTPS, po tysiące innych rzeczy we współczesnym świecie. Innymi słowy, Alicja i Robert nie muszą być ludźmi — mogą to być komputery, usługi itd.

Znasz już podstawowe kryptograficzne elementy konstrukcyjne, które SSH wykorzystuje do zabezpieczania zdalnego dostępu, i jesteś gotowy do praktycznego wykorzystania tej fantazyjnej technologii!

Klucze SSH

Jedną z pierwszych rzeczy, które prawdopodobnie zrobisz, jeśli chodzi o SSH, jest utworzenie własnej pary kluczy. Umożliwi to uwierzytelnianie się na serwerze SSH. Oto klasyczne polecenie do tworzenia pary kluczy:

```
ssh-keygen -t ed25519 -C 'Jan Kowalski <jan.kowalski@example.org>'
```

Spowoduje to utworzenie pary kluczy ed25519 (jest to nowoczesny algorytm kryptografii krzywej eliptycznej) z zastosowaniem Jan Kowalski <jan.kowalski@example.org> jako komentarza. Komentarze nie różnią się od tych, które znasz z języków programowania, ponieważ mogą być dowolnym łańcuchem znaków i nie będą zakłócać niczego na poziomie funkcjonalnym. W przypadku SSH komentarz zostanie dołączony do klucza publicznego, co ułatwi rozróżnienie kluczy podczas przesyłania ich na serwer, na przykład w pliku *authorized_keys*. Dalej w tym rozdziale zajmiemy się dokładniej plikami *authorized_keys* i wyjaśnimy, jak z nich korzystać, aby skonfigurować bezproblemowy, bezpieczny dostęp do zdalnych serwerów.

Po uruchomieniu tego polecenia OpenSSH zada Ci kilka pytań o to, gdzie chcesz przechowywać tworzone pliki kluczy i jakiego hasła chcesz użyć do szyfrowania klucza prywatnego. Ponieważ będzie to klucz wykorzystywany do dostępu do systemów zdalnych, ustaw silne hasło.

Uwaga

Domyślnie klucze zostaną umieszczone w katalogu `~/.ssh`.

Gdy masz już parę kluczy, powtórzmy najważniejszą praktyczną kwestię: nigdy, prze-nigdy nie udostępniaj klucza prywatnego, gdyż pozwoliłoby to innej osobie podszywać się pod Ciebie. Żadna usługa nie powinna prosić Cię o udostępnienie klucza prywatnego. Klucz publiczny, który ma być udostępniany i można go bezpiecznie upublicznić, będzie zawierał przyrostek *.pub*, dzięki któremu będzie można go odróżnić.

Na szczęście zawartość tych plików wygląda zupełnie inaczej, więc jeśli będziesz mieć wątpliwości, możesz zajrzeć do nich, aby je rozróżnić:

- Klucz publiczny ma format `<algorytm> <klucz> <komentarz>`.
- Klucz prywatny zaczyna się od linii takiej jak `-----BEGIN OPENSSH PRIVATE KEY-----`, po której następują właściwy kluz i podobna linia końcowa.

Pliki te należy zapisać w bezpiecznej kopii zapasowej i nigdy ich nie nadpisywać. W tym celu można skorzystać z menedżera haseł. Wiele menedżerów haseł ma nawet określoną opcję przechowywania plików kluczy prywatnych lub tekstu ogólnego.

Wyjątki od zasad

Do użytku osobistego — na przykład na laptopie — należy zaszyfrować swój klucz prywatny, określając hasło podczas tworzenia klucza, a potem nigdy nie udostępniać hasła, jak już wspomnieliśmy wcześniej.

Istnieją jednak sytuacje, w których można zrobić wyjątek od tych zasad, szczególnie podczas konfigurowania kluczy dla zautomatyzowanych systemów. Jeśli chcesz, aby serwer kompilacji uwierzytlił się na GitHubie przed sprawdzaniem bazy kodu, prawdopodobnie użyjesz pary kluczy, w których klucz prywatny *nie jest* zaszyfrowany (chyba że chcesz ręcznie wpisywać to hasło za każdym razem, gdy uruchamiana jest usługa kompilacji).

Uwierzytelnianie i szyfrowanie między maszynami to świetne powody, dla których warto korzystać z par kluczy kryptograficznych. Pamiętaj tylko, aby zawsze tworzyć jednorazowe pary kluczy przeznaczone specjalnie do takich zadań i nie używać ich ponownie ani nie udostępniać innym maszynom lub usługom.

Logowanie i uwierzytelnianie

Logowanie do systemu zdalnego przy użyciu uwierzytelniania opartego na SSH wygląda mniej więcej tak:

```
ssh user@example.org
```

W tym przypadku `user` to nazwa użytkownika, z którego uprawnieniami chcesz się zalogować, a w miejsce `example.org` należy podstawić zdalne urządzenie, z którym chcesz się połączyć. Często jest to tylko adres IP zamiast w pełni kwalifikowanej nazwy domenowej.

Jeśli logujesz się za pomocą klucza SSH lub musisz określić konkretny klucz (tożsamość lub `-i`), wygląda to następująco:

```
ssh -i ~/.ssh/id_ecdsa user@example.org
```

Podczas uzyskiwania dostępu SSH do serwera, z którym nigdy wcześniej się nie łączyłeś, wyświetlony zostanie odcisk palca (fingerprint) tego zdalnego serwera. Pozwala to upewnić się, że serwer, z którym się komunikujesz, jest w rzeczywistości tym, z którym zamierzasz się połączyć, i że nie ma akurat miejsca atak typu „człowiek pośrodku” (ang. *man-in-the-middle attack*). Powinieneś upewnić się, że odcisk palca jest poprawny.

Po wpisaniu `yes` (tak) i oznaczeniu tego odcisku palca jako zaufanego zostanie on zapisany w pliku. Jeśli kiedykolwiek się to zmieni — na przykład gdy ktoś skonfiguruje złośliwy serwer pod tym samym adresem IP, z którego korzystał wcześniej Twój zaufany serwer — zostaniesz powiadomiony przez klienta SSH, a uwierzytelnienie nie będzie możliwe.

Po oznaczeniu serwera jako zaufanego lokalny klient i serwer będą negocjować, której formy uwierzytelniania należy użyć. OpenSSH oferuje szeroką gamę opcji, z dwiema najczęstszymi, obejmującymi hasło lub parę kluczy. W zależności od tego, która z nich zostanie wybrana, OpenSSH poprosi Cię o podanie hasła (lub hasła do odszyfrowania klucza prywatnego). Po pomyślnym zakończeniu procesu uwierzytelniania użytkownik zostanie zalogowany.

Projekt praktyczny: konfigurowanie logowania opartego na kluczu na zdalnym serwerze

W tym projekcie zakładamy, że masz dostęp do długo działającego serwera linuksowego, na którym chcesz zezwolić na logowanie oparte na kluczu. Wykonaj opisane dalej czynności.

Krok 1. Otwórz terminal na kliencie SSH (nie na serwerze)

Do pozostałych kroków będziesz używać lokalnego środowiska wiersza poleceń.

Krok 2. Wygeneruj parę kluczy

Jeśli masz już skonfigurowaną parę kluczy, ponieważ śledziłeś wcześniejsze przykłady z tego rozdziału, bardzo dobrze! Możesz pominąć ten krok.

Jeżeli nie masz jeszcze pary kluczy SSH, utwórz ją — wpisz poniższe polecenie i naciśnij przycisk *Enter*:

```
ssh-keygen -t ed25519
```

```
# Ten klucz publiczny nie może być czytelny dla każdego  
chmod 600 ~/.ssh/id_ed25519
```

Jak wspomnieliśmy wcześniej, zdecydowanie zalecamy dodanie hasła w celu zwiększenia bezpieczeństwa.

Krok 3. Skopiuj klucz publiczny na serwer

Po wygenerowaniu kluczy musisz umieścić klucz publiczny na serwerze. Klucz publiczny zwykle ma rozszerzenie *.pub* i domyślnie znajduje się w katalogu *~/.ssh*.

Możesz skopiować go ręcznie do pliku *authorized_keys* zdalnego użytkownika (plik ten zawiera wszystkie autoryzowane klucze publiczne dla tego użytkownika, po jednym kluczu na linię) lub skondensować wszystkie te działania w jednym poleceniu przy użyciu programu *ssh-copy-id*:

```
ssh-copy-id username@example.org
```

Zastąp *username* swoim użytkownikiem na serwerze zdalnym i *remote_server_address* adresem IP serwera lub nazwą domenową.

To polecenie spowoduje wyświetlenie monitu o podanie hasła użytkownika na zdalnym serwerze. Po wprowadzeniu hasła klucz publiczny zostanie dołączony do pliku *~/.ssh/authorized_keys* w katalogu domowym użytkownika zdalnego. Umożliwia to logowanie się i wykonywanie poleceń na komputerze zdalnym bez konieczności podawania hasła.

Krok 4. Przetestuj to!

Teraz spróbuj zalogować się do serwera:

```
ssh username@example.org
```

Jeśli wszystko zadziała, zostaniesz zalogowany bez pytania o hasło (chyba że ustawiłeś hasło dla swojego klucza SSH). Zamiast hasła w postaci krótkiego łańcucha znaków, który może zostać odgadnięty przez atakującego, używasz teraz do uwierzytelnienia *znacznie* bezpieczniejszego klucza kryptograficznego.

Witamy w cudownym świecie bezpiecznego, bezhasłowego dostępu SSH!

Konwersja kluczy SSH2 na format OpenSSH

Jeśli nie korzystasz z systemu operacyjnego opartego na Uniksie, często będziesz napotykać format klucza publicznego SSH2. Najbardziej bodaj znanym oprogramowaniem używającym tego formatu jest PuTTY i wiele osób korzystających z systemu Windows używa go do łączenia się przez SSH. Aby połączyć się z serwerem **SFTP** (ang. *SSH File Transfer Protocol*), repozytorium Gita lub innym systemem używającym formatu klucza OpenSSH, musisz przekonwertować klucz publiczny SSH2 na format OpenSSH. Oto jak to zrobić.

Co chcemy osiągnąć?

Zaczynamy od klucza publicznego w formacie SSH2, który wygląda tak:

```
---- BEGIN SSH2 PUBLIC KEY ----
Comment: "rsa-key-20160402"
AAAAB3NzaC1yc2EAAAABJQAAAQEAiL0jjDdFqK/kYThqKt7ThrjABTPWvXmB3URI
pGKCP/jZ1SuCUP30c+IxuFeXSIMvVIYeW2PZAJXQGtn60XzPHr+M0NoGcPAvzZf2
u57aX3YKaL93cZSBHR97H+XhcYdrM7ATwfjMDgfgj7+VTvW4nI46Z+qjxmYifc8u
VELo1g1TDHWY789ggcdvy92oGjBOVUgMEywrOP+LS0DgG4dmkoUBWGP9dvYcPZDU
F4q0XY9ZHhvyPWEZ3o2vETTrEjr9QHYwgjmFfJn2VFND/4qeDDH0mS1DgE0fQcZ
Im+XU0n9eVsv//dAPSY/yMJXf8d0ZSm+VS29QShMjA4R+7yh5WhsIhouBRno2PpE
Vvb37Xwe3V6U3o9UnQ3ADtL75DbrZ5beNWcmKz1J7jVX5QzHSBAnePbBx/fyeP/f
144xPtJWB3jW/kXjtPyWjpzGndaPQ0WgXkbF8fvIuB3NJTTcZ7PeIKnLaMIzT5XN
CR+xobvdC8J9d6k84/q/1aJKF3G8KbRGPnwnoVg1cwWFez+dzqo2ypcTtv/20yAm
z86EvuohZoWrtowVwZLCoyxdq093ymeJgHAn2bsIWY00DtXovxAJqPgk3dxM1f9P
AEQwc1bG+Z/Gc1Fd8DncgxyhKSQzLsfWroTnIn8wsnmhPJtaZWNUt5Bja8Ghnx0
9g6nhbk=
---- END SSH2 PUBLIC KEY ----
```

Celem jest przekształcenie go w klucz publiczny OpenSSH w następujący sposób:

```
ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAAQEAiL0jjDdFqK/
kYThqKt7ThrjABTPWvXmB3URIPgK
CP/
jZ1SuCUP30c+IxuFeXSIMvVIYeW2PZAJXQGtn60XzPHr+M0NoGcPAvzZf2u57aX3YKaL93cZSBHR
97H+XhcYdrM7ATwfjMDgfgj7+VTvW4nI46Z+qjxmYifc8uVELo1g1TDHWY789ggcdvy92oG
jBOVUgMEywrOP+LS0DgG4dmkoUBWGP9dvYcPZDUF4q0XY9ZHhvyPWEZ3o2vETTrEjr9QHYwgjmFf
Jn2VFND/4qeDDH0mS1DgE0fQcZIm+XU0n9eVsv//dAPSY/
yMJXf8d0ZSm+VS29QShMjA4R+7yh5Wh
sIhouBRno2PpEVvb37Xwe3V6U3o9UnQ3ADtL75DbrZ5beNWcmKz1J7jVX5QzHSBAnePbBx/
fyeP/
f144xPtJWB3jW/kXjtPyWjpzGndaPQ0WgXkbF8fvIuB3NJTTcZ7PeIKnLaMIzT5XNCR+xobvdC8
J9d6k84/q/
1aJKF3G8KbRGPnwnoVg1cwWFez+dzqo2ypcTtv/20yAmz86EvuohZoWrtowVwZLCoyxdq093yme
JgHAn2bsIWY00DtXovxAJqPgk3dxM1f9PAEQwc1bG+Z/Gc1Fd8DncgxyhKSQzLsfWroTn
In8wsnmhPJtaZWNUt5Bja8Ghnx09g6nhbk=
```

Jak przekonwertować klucz w formacie SSH2 na OpenSSH?

Polecenie `ssh-keygen`, którego użyliśmy do utworzenia nowego klucza, może również przekonwertować go za pomocą tego bardzo prostego polecenia:

```
ssh-keygen -i -f ssh2.pub > openssh.pub
```

Polecenie to spowoduje pobranie klucza z pliku `ssh2.pub` i zapisanie go w pliku `openssh.pub`.

Jeśli chcesz przyjrzeć się, jak wygląda klucz OpenSSH, lub przygotować go do skopiowania i wklejenia, nie musisz przekierowywać stdout do pliku (to samo polecenie co powyżej, bez ostatniej części):

```
ssh-keygen -i -f ssh2.pub
```

Spowoduje to wyświetlenie klucza publicznego w formacie OpenSSH.

Bardziej praktycznym przykładem może być przekonwertowanie i dołączenie klucza współpracownika do pliku *authorized_keys* serwera. Można to zrobić za pomocą poniższego polecenia:

```
ssh-keygen -i -f coworker.pub >> ~/.ssh/authorized_keys
```

Wtedy współpracownik korzystający z odpowiedniego klucza prywatnego będzie mógł zalogować się do systemu jako użytkownik, który uruchamia to polecenie.

Na odwrót: konwersja kluczy OpenSSH na format SSH2

Możliwe jest również przekształcanie kluczy OpenSSH na klucze SSH2. Wystarczy użyć flagi *-e* (dla eksportu) zamiast *-i* (dla importu):

```
ssh-keygen -e -f openssh.pub > ssh2.pub
```

Agent SSH

Jeśli często logujesz się do serwerów za pomocą kluczy SSH, wpisywanie hasła klucza prywatnego za każdym razem, gdy łączysz się (lub ponownie łączysz) z hostem, może być irytujące. Agent SSH umożliwia przechowywanie tożsamości (klucza prywatnego) dla sesji lokalnej — innymi słowy, pozwala odszyfrować klucz prywatny raz, a następnie zachować go w pamięci do czasu wylogowania lub rozpoczęcia nowej sesji powłoki. Oznacza to, że dodajesz tożsamość (parę kluczy) raz i możesz z niej korzystać bez ponownego odszyfrowywania klucza prywatnego.

Uwaga

Agent SSH nie zawsze działa w lokalnej sesji powłoki — różne środowiska IDE, menedżery okien, menedżery pulpitu i menedżery haseł również mogą uruchamiać agenta. Będziesz wiedzieć, że tak się dzieje, gdy będziesz musiał tylko raz wprowadzić hasło dla tożsamości.

Aby dodać klucz do agenta, wystarczy użyć polecenia *ssh-add* — argument jest ścieżką do klucza prywatnego dla tej tożsamości:

```
ssh-add ~/.ssh/id_ecdsa
```

Zalecamy wyrobienie sobie nawyku korzystania z opcji *-t*, która dodaje limit czasowy przechowywania odszyfrowanych kluczy w pamięci. Poniższe polecenie jest takie samo jak

poprzednie, z wyjątkiem ustawienia limitu czasu 30 sekund, po którym agent usunie klucze z pamięci:

```
ssh-add -t 30 ~/.ssh/id_ecdsa
```

Aby zobaczyć, które klucze zostały dodane do agenta, użyj następującego polecenia:

```
ssh-add -L
```

Jeśli chcesz usunąć wszystkie tożsamości z agenta SSH, użyj opcji `-D`:

```
ssh-add -D
```

Uwaga

Jeśli dodałeś do agenta więcej niż trzy tożsamości, być może podczas logowania za pośrednictwem SSH będziesz musiał określić `-i $TWOJA_TOŻSAMOŚĆ`. Dzieje się tak dlatego, że większość serwerów jest skonfigurowana do odrzucania loginów po trzech nieprawidłowych próbach, a podczas logowania SSH będzie próbować każdego klucza przechowywanego w agencie, jednego po drugim. Jeśli pierwszy klucz nie zadziała, to spróbuje drugiego itd. Jeżeli serwer przerwie logowanie po trzech próbach, nigdy nie dojdiesz do czwartego klucza w agencie.

Za pomocą opcji `-A` możesz włączyć przekazywanie w agencie podczas korzystania z SSH, aby zalogować się do zdalnych maszyn (rób to jednak sporadycznie i ostrożnie, z powodów, które za chwilę wyjaśnimy):

```
ssh -A user@remotehost
```

Po zalogowaniu się do `remotehost` za pomocą powyższego polecenia SSH przekaże klucze do tego hosta, dzięki czemu będzie można ich używać do przeskakiwania do dodatkowych hostów z tego miejsca. Dlatego zalecamy korzystać z tej możliwości oszczędnie, ponieważ pozwala zainfekowanym hostom zobaczyć Twoje klucze prywatne. Pamiętaj, że bezwzględnie należy dbać o prywatność i przechowywać klucze najlepiej na swoim komputerze, a w ostateczności w menedżerze haseł. Innymi słowy, jeśli `remotehost` zostanie zhakowany, klucze SSH będą zagrożone.

Jeszcze jedna, ostatnia uwaga na temat bezpieczeństwa: niektóre środowiska pulpitu Linuksa, takie jak GNOME lub MATE, będą przechowywać klucz SSH w pamięci w nieskończoność, gdy już raz go użyjesz i odszyfrujesz. Dzieje się to domyślnie i jest zagrożeniem dla bezpieczeństwa.

Typowe błędy SSH i argument `-v` (verbose)

Flaga `-v` w poleceniu SSH włącza tryb verbose, który wypisuje szczegółowy dziennik procesu połączenia krok po kroku. Ta funkcja jest szczególnie przydatna w diagnozowaniu typowych problemów, takich jak błędy uwierzytelniania, limity czasu połączenia i niedopasowanie kluczy. Używa się tego w ten sposób:

```
ssh -v username@example.org
```

Otrzymasz informacje krok po kroku na temat każdego etapu uzgadniania i połączenia SSH, co ułatwi identyfikowanie i rozwiązywanie wszelkich problemów, które mogą się pojawić.

Oto kilka z częstych błędów, w których zdiagnozowaniu pomocne mogą być szczegółowe dane wyjściowe:

- **Odmowa uprawnień (klucz publiczny/hasło)** — oznacza, że serwer odrzucił próbę logowania. Szczegółowy dziennik pokaże, które klucze zostały wypróbowane, co pomoże określić, czy użyto prawidłowego klucza i czy w ogóle go zaoferowano. Jest to *niezwykle* powszechny problem, gdy w kliencie są przechowywane więcej niż trzy pary kluczy, a serwer zezwala tylko na trzy próby.
- **Przekroczono limit czasu połączenia** — jeśli połączenie trwa zbyt długo, może wystąpić problem z siecią, nieprawidłowym adresem IP lub portem. Flaga -v pokaże, w którym miejscu utknął proces, co pomoże zrozumieć, czy klient w ogóle skomunikował się z serwerem.
- **Odmowa połączenia** — zwykle oznacza, że SSH nie jest uruchomiony (lub osiągalny) na porcie docelowym na serwerze. Szczegółowe dane wyjściowe wyraźnie wskażą, że próba połączenia została odrzucona, co pomoże skupić się na regułach zapory sieciowej lub ustawieniach serwera SSH.
- **Weryfikacja klucza hosta nie powiodła się** — klucz serwera nie pasuje do klucza zapisanego w pliku *known_hosts* systemu. Flaga -v pokaże niedopasowane klucze, a wtedy można skupić się na ustaleniu **dlaczego** (na przykład czy istnieje nowy serwer pod tym adresem IP lub nazwą hosta).
- **Nie można rozpoznać nazwy hosta** — często jest to związane z problemami z systemem DNS lub siecią.
- **Brak trasy do hosta** — sugeruje to problemy z siecią, które mogą mieć związek z firewallem lub nieprawidłowym routinguem.
- **Zbyt wiele niepowodzeń uwierzytelniania** — osiągnięta została maksymalna liczba prób uwierzytelniania. W trybie szczegółowym zostaną wyświetlone wszystkie wypróbowane metody, które mogą obejmować niechciane lub nieoczekiwane oferty kluczy.
- **Błędy ładowania klucza** — zwykle wskazują na problem z formatem lub uprawnieniami klucza SSH. Flaga -v identyfikuje klucz, który klient SSH próbuje załadować, umożliwiając sprawdzenie problemów z formatowaniem lub uprawnieniami.

Użycie flagi -v pomoże Ci zrozumieć, co dokładnie jest nie tak i jak możesz to naprawić. Przynajmniej pomoże Ci to rozpocząć poszukiwania we właściwym kierunku.

Przesyłanie plików

W tym podrozdziale zbadamy polecenia *sftp* i *scp* służące do przesyłania plików. Dzięki przeanalizowaniu kilku przykładów z wykorzystaniem tych poleceń można zrozumieć, jak należy obsługiwać pliki w większości sytuacji. Omówimy jednak również transfer plików bez SFTP lub SCP, w przypadku gdy są one wyłączone na serwerze.

SFTP

OpenSSH jest często używany jako sposób logowania się do zdalnych systemów, pozwala jednak również na przesyłanie plików niezależnie od sesji logowania. Zwykle jest to realizowane za pośrednictwem podsystemu SFTP. Chociaż SFTP przypomina **FTP** (ang. *File Transfer Protocol*), jest to w rzeczywistości całkowicie niestandardowy protokół. Podobnie jak FTP umożliwia on uwierzytelnionym użytkownikom przesyłanie plików do serwerów zdalnych i pobieranie ich z tych serwerów. W przeciwieństwie do FTP, który jest niezabezpieczony, uwierzytelnianie i przesyłanie plików SFTP jest bezpieczne i w pełni zaszyfrowane.

Istnieje wiele klientów FTP, które obsługują również SFTP. Jednym ze słynnych przykładów jest Filezilla, która ma doskonały graficzny interfejs użytkownika. Ponieważ jednak niniejsza książka traktuje o wierszu poleceń Linuksa, zawrzemy tu podstawowe informacje o tym, jak używać SFTP w wierszu poleceń.

Uwierzytelnianie jest prawie identyczne jak w przypadku ssh:

```
sftp user@example.org
```

Następnie wyświetlany jest interfejs w stylu FTP. Przyjmuje on prostsze lub zmodyfikowane wersje niektórych poleceń powłoki omówionych już w poprzednich rozdziałach, a ponadto wprowadza kilka nowych. Oto najczęściej używane polecenia:

- **help** — wyświetla listę wszystkich poleceń i krótkie podsumowanie;
- **ls** — wyświetla zawartość katalogu zdalnego;
- **lls** — wyszczególnia zawartość katalogu lokalnego;
- **cd** — umożliwia zmianę katalogu zdalnego;
- **lcd** — umożliwia zmianę katalogu lokalnego;
- **pwd** — wyświetla bieżący zdalny katalog;
- **lpwd** — wyświetla bieżący lokalny katalog;
- **get** — pobiera plik z serwera zdalnego;
- **put** — przesyła plik na zdalny serwer;
- **chmod** — zmienia uprawnienia zdalnego pliku lub katalogu;
- **chown** — zmienia właściciela zdalnego pliku lub katalogu;
- **quit, exit, bye** — zamyka SFTP (*Ctrl+D* również działa).

SCP

Polecenie scp jest często bardziej praktycznym sposobem przesyłania i pobierania plików i katalogów niż sftp. Chociaż kiedyś było niezależne od SFTP, obecnie korzysta z podsystemu SFTP. Ma działać jako zamiennik polecenia cp, z tą różnicą, że umożliwia kopiowanie na zdalne serwery i w odwrotną stronę.

Polecenia mają następujący format:

```
scp $ŹRÓDŁO:ścieżka_pliku $MIEJSCE_DOCELOWE:ścieżka_pliku
```

`$ŹRÓDŁO`, `$MIEJSCE_DOCELOWE` lub obie te lokalizacje mogą być systemami zdalnymi; `scp` wykorzystuje składnię SSH `user@example.org`, którą już znasz. Oto jak wygląda to w praktyce:

```
scp user@example.org:/home/użytkownik/mój_zdalny_plik
/home/użytkownik/mój_lokalny_plik
```

Spowoduje to wykonanie następujących czynności:

1. Połączenie się z `example.org`.
2. Uwierzytelnienie się jako `user` (zostaniesz poproszony o hasło, chyba że używasz kluczy SSH i agenta SSH).
3. Skopiowanie pliku do lokalnej ścieżki `/home/użytkownik/mój_lokalny_plik`.

Jak widać, kolejność argumentów jest taka sama jak w przypadku linuksowego polecenia `cp` (kopiuj).

Odwrócenie argumentów źródła i miejsca docelowego — przesyłanie pliku zamiast jego pobierania — wygląda zgodnie z oczekiwaniami:

```
scp /home/użytkownik/mój_lokalny_plik user@example.org:/home/użytkownik/
mój_zdalny_plik
```

Podobnie jak w przypadku `cp` można określać ścieżki względne. Poniższe polecenie skopiuje zdalny plik do bieżącego (lokalnego) katalogu:

```
scp user@example.org:/home/użytkownik/mój_zdalny_plik .
```

Podobnie jak w przypadku polecenia `cp` możliwe jest również rekurencyjne kopiowanie całego katalogu przy użyciu flagi `-r` (w niektórych systemach również `-R`):

```
scp -r user@example.org:/home/użytkownik/katalog katalog_lokalny
```

Przydatne przykłady

`ssh` i `scp`, tak jak wszystkie polecenia i narzędzia, mogą być stosowane także w skryptach. Na przykład można użyć `ssh` do szybkiego utworzenia kopii zapasowej bazy danych:

```
ssh username@example.org "pg_dump nazwaBD | gzip -c" > database_backup.sql.gz
```

Jeśli korzystasz ze sprytnych trików powłoki, takich jak ten w skrypcie, zadбай o ostrzeżenie o błędach, w przeciwnym bowiem razie proces może niepostrzeżenie utknąć, gdy napotka problem. W sytuacjach programistycznych to polecenie może być naprawdę przydatne.

Bez SFTP lub SCP

W niektórych rzadkich przypadkach może się okazać, że SFTP został wyłączony na serwerze i można zalogować się tylko do interaktywnej powłoki. Chcesz przesłać plik do zdalnego systemu i chociaż istnieje możliwość, by po prostu otworzyć plik w edytorze, nie zawsze jest ona dostępna (na przykład w przypadku całych katalogów lub plików binarnych). Oto kilka sztuczek, których można użyć, aby osiągnąć swój cel (wszystkie wykorzystują potok Uniksa w połączeniu z sesją SSH).

Najprostszym przypadkiem jest pobranie pliku ze zdalnego serwera:

```
ssh user@example.org "cat /ścieżka/do/pliku" > lokalny_plik_docelowy
```

To polecenie wykona następujące czynności:

1. Zaloguje się do serwera.
2. Uruchomi polecenie `cat /ścieżka/do/pliku` na zdalnym serwerze, co spowoduje przesłanie zawartości tego pliku do stdout.
3. Strumień stdout może zostać następnie lokalnie przekazany do wybranego przez nas pliku.

Podobnie jak większość prawidłowo zachowującego się oprogramowania uniksowego błędy i inne potencjalnie zakłócające dane wyjściowe będą wysyłane do STDERR, więc nie musisz się na przykład obawiać, że monit o hasło stanie na drodze i uszkodzi zawartość pliku przesyłanego potokiem przez tunel SSH.

Przesyłanie katalogów i kompresja .tar.gz

Zróbmy coś innego i prześlijmy cały katalog. Ponieważ może to być dość dużo danych, skompresujemy je również za pomocą programu `tar`. Jest to polecenie, które konwertuje wiele plików i (lub) katalogów w jeden plik („archiwum”).

Często po archiwizacji dodajemy krok kompresji. Prawdopodobnie widziałeś na przykład pliki z rozszerzeniami `tar.gz` lub `tar.bz2`. Oznacza to, że pliki zostały najpierw zarchiwizowane do postaci jednego pliku za pomocą `tar`, a następnie skompresowane za pomocą narzędzia `gzip` lub `bzip2`:

```
tar czf - /home/użytkownik/katalog_do_przesłania | ssh user@example.org  
↪"tar -xvzf -C /home/użytkownik/"
```

Tutaj najpierw archiwizujemy `/home/użytkownik/katalog` za pomocą programu `tar` i przekształcamy wynik w strumień bajtów.

Znak `-` to plik docelowy. Gdy zamiast przysyłać strumieniowo do innego programu, zechcesz natychmiast go zapisać, będzie to wyglądać mniej więcej tak: `/home/użytkownik/↪katalog.tar.gz`. Jak w przypadku wielu poleceń uniksowych dywiz symbolizuje, że dane powinny być zamiast tego zapisane do standardowego wyjścia (stdout).

Wynikowy strumień zostanie następnie przekazany do procesu `ssh`, który użyje go jako danych wejściowych (stdin) dla polecenia `tar` na zdalnym systemie, a polecenie to zdekompresuje i zarchiwizuje ten strumień, zapisując wynikowy katalog w `/home/użytkownik/↪katalog_do_przesłania`.

Tunele

Tunelowanie SSH służy do przesyłania danych przez połączenie SSH. W tym podrozdziale przyjrzymy się dwóm metodom tunelowania: przekierowaniu lokalnemu i użyciu serwera pośredniego.

Przekierowanie lokalne

SSH może tworzyć bezpieczne, szyfrowane tunele do zdalnych systemów. Funkcjonalność ta jest podobna do tego, co zapewnia VPN, i umożliwia dostęp do usług dostępnych z systemu zdalnego.

To potężna funkcjonalność, a dzięki SSH można z niej łatwo skorzystać. Wystarczy podczas ustanawiania sesji SSH podać dodatkowy argument `-L` z miejscem docelowym i lokalnym portem do powiązania.

Wyobraźmy sobie zdalny system z uruchomionym serwerem HTTP na porcie 8080. Chcesz uzyskać do niego dostęp na swoim laptopie, na porcie 3000. Oto jak to osiągnąć za pomocą prostego polecenia:

```
ssh -L 3000:localhost:8080 user@example.org
```

Możesz teraz otworzyć przeglądarkę i wpisać adres `http://localhost:3000/`, aby uzyskać dostęp do serwera WWW tak, jakbyś otworzył tę przeglądarkę na zdalnym systemie i wpisał adres `http://localhost:8080`.

Serwer pośredni (proxy)

To samo będzie miało zastosowanie, gdybyś chciał uzyskać dostęp do serwera, do którego nie masz dostępu, ale serwer zdalny ma. Wyobraź sobie, że *app.example.org* to miejsce, w którym działa Twoja aplikacja internetowa. Ta aplikacja łączy się z serwerem bazy danych, takim jak PostgreSQL, pod adresem *db.example.org*, który jest dostępny tylko z sieci wewnętrznej *www.example.org*. Podobnie jak większość produkcyjnych baz danych jest ona chroniona przez firewall, który uniemożliwia bezpośrednie połączenia z zewnątrz.

Z perspektywy sieci wygląda to następująco:

```
(localhost) —> (app.example.org) —> (db.example.org)
```

Aby połączyć się z tą bazą danych, możesz użyć lokalnego klienta `psql` Postgresa. W tym celu należy utworzyć taki tunel:

```
ssh -L 5000:db.example.org:5432 user@app.example.org
```

Spowoduje to otwarcie nowej sesji SSH do *app.example.org*, połączenie się z tego miejsca z serwerem bazy danych i zmapowanie *db.example.org:5432* na *localhost:5000*.

Teraz uruchomienie na laptopie polecenia `psql --port=5000 --host=localhost nazwa_bazy_danych` połączy Cię z bazą danych Postgres pod adresem *db.example.org:5432*, przeskakując z *app.example.org*, aby uzyskać dostęp.

Plik konfiguracyjny

Istnieje możliwość określenia konfiguracji hosta w pliku `.ssh/config`. Może to być pomocne w rozmaitych sytuacjach, ponieważ pozwala określić różne elementy. Oto niektóre z nich:

- niestandardowe (przyjazne) nazwy dla hostów,
- domyślny użytkownik,
- port,
- tunele do otwarcia przed połączeniem,
- pliki tożsamości (klucze).

Uwaga

Jeśli serwer, z którym się łączysz, ma stały adres IP, warto określić go w pliku konfiguracyjnym SSH, aby uniknąć polegania na systemie DNS lub sieci CDN w sytuacji odzyskiwania sprawności po awarii.

Pliki konfiguracyjne SSH nie są szczególnie skomplikowane, więc pokażemy tutaj przykład, który wykorzystuje wiele dostępnych funkcjonalności:

```
# Ustawienie wartości domyślnych dla wszystkich hostów przy użyciu znaku glob (*)Host *
ServerAliveInterval 30      # Co 30 sekund sprawdzanie, czy połączenie jest aktywne
ForwardAgent yes            # Przekierowanie agenta SSH do zdalnego hosta
Compression yes             # Włączenie kompresji
IdentityFile ~/.ssh/id_rsa  # Domyślny klucz tożsamości
```

```
# Konkretnie ustawienia dla hosta "example1.com"
```

```
Host example1
  HostName example1.com      # Rzeczywista nazwa hosta do łączenia
  User john                  # Domyślna nazwa użytkownika dla tego hosta
  Port 22                    # Port SSH (domyślny to 22)
  IdentityFile ~/.ssh/id_ecdsa # Inny plik tożsamości dla tego hosta
```

```
# Ustawienia dla kolejnego konkretnego hosta "example2.com"
```

```
Host example2
  HostName example2.com
  User jane
  Port 22000                 # Inny port SSH dla tego hosta
  IdentityFile ~/.ssh/id_ed25519
```

```
# Użycie hosta przeskoku w celu połączenia się z hostem za firewallem
```

```
Host target-behind-fw
  HostName 192.168.0.2        # Prywatny adres IP docelowego hosta
  User alice
  Port 22
  ProxyJump jump-host        # Użycie 'jump-host' jako hosta przeskoku
```

Konfiguracja dla hosta przeskoku

Host jump-host

HostName jump.example.com *# Publiczny adres IP lub domena hosta przeskoku*

User jumpuser

Port 22

IdentityFile ~/.ssh/jump_key

Użycie SOCKS proxy do łączenia się z hostem

Host proxy-host

HostName proxy-target.com *# Rzeczywista nazwa hosta do łączenia się*

User proxyuser

Port 22

ProxyCommand nc -X 5 -x localhost:1080 %h %p *# Użycie SOCKS proxy na porcie lokalnym 1080*

Podsumowanie

OpenSSH jest bardzo wszechstronnym narzędziem i mamy nadzieję, że wprowadzenie, które przedstawiliśmy w tym rozdziale, zmotywowało Cię do eksperymentowania i dalszej nauki.

Omówiliśmy podstawy działania kryptografii klucza publicznego oraz wszystko, co jest niezbędne, aby móc zrozumieć działanie tego rodzaju narzędzi i ich zastosowanie. Pokazaliśmy, jak tworzyć klucze SSH i używać ich do zdalnych sesji powłoki.

Mamy nadzieję, że zdobyłeś również praktyczne doświadczenie, śledząc przykłady i konfigurując logowanie oparte na kluczach dla zdalnego hosta, z którym często pracujesz. Jeśli ten zdalny host działa na **AWS** (ang. *Amazon Web Services*) lub innej platformie, która korzysta z kluczy *.pem*, nauczyłeś się konwertować formaty kluczy (ta sztuczka z pewnością zrobi wrażenie na Twoich współpracownikach).

Nawet jeśli nie natknąłeś się na nie osobiście, omówiliśmy niektóre z najczęstszych błędów SSH, które napotkaliśmy w praktyce, i pokazaliśmy, jak je wyśledzić za pomocą opcji *-v*.

Omówiliśmy nawet użycie SSH poza zdalnymi interaktywnymi sesjami powłoki — do przesyłania plików, tunelowania ruchu sieciowego i konfigurowania niestandardowych ustawień dla różnych serwerów. Co zaskakujące, OpenSSH potrafi jeszcze więcej. Oto kilka przykładów:

- szyfrowanie i podpisywanie plików za pomocą *ssh-keygen*;
- dodawanie uwierzytelniania dwuskładnikowego za pomocą FIDO/U2F i przechowywanie klucza na urządzeniu zewnętrznym;
- wymuszanie uruchamiania określonych poleceń po zalogowaniu, co zarówno ogranicza wektor ataku, jak i pozwala SSH, aby działał jako interfejs do usługi.

Projekt OpenSSH zapewnia doskonałą dokumentację na swoich stronach podręcznika i na stronie internetowej. Jeśli masz problem, który wymaga bezpiecznych połączeń między maszynami, i potrzebujesz technologii, która została sprawdzona w boju, warto przetestować OpenSSH. A teraz idź i szyfruj!

Kontrola wersji za pomocą Gita

Git to rozproszony system kontroli wersji (ang. *distributed version control system* — DVCS), który w ciągu ostatnich dwóch dekad stał się najczęściej używanym systemem kontroli wersji na świecie. Chociaż jest bardzo prawdopodobne, że znasz już podstawy korzystania z Gita, możesz nie być zaznajomiony z typowymi wzorcami wiersza poleceń lub niektórymi z jego rzadziej używanych (ale potężnych!) funkcjonalności. Omówimy je tutaj. Ten rozdział obejmuje również podstawową wiedzę, dzięki której powszechnie używane terminy Gita nabiorą większego sensu, a często przywoływane koncepcje staną się jasne.

W tym rozdziale omawiamy następujące zagadnienia:

- podstawy Gita i rozproszonej kontroli wersji;
- pierwsza konfiguracja Gita;
- podstawowe polecenia Gita;
- powszechna terminologia Gita;
- dwie potężne i nieco bardziej zaawansowane koncepcje Gita: bisecting i rebasing;
- najlepsze praktyki Gita, szczególnie w zakresie efektywnego korzystania z komunikatów commitu;
- przydatne aliasy powłoki Gita, które pozwalają wyeliminować dużo pisania;
- narzędzia graficznego interfejsu użytkownika, których można używać do interakcji z Gitem.

Na koniec, w podrozdziale „GitHub dla ubogich”, przedstawimy mały, ale bardzo przydatny projekt, który możesz wykonać, aby poćwiczyć i ugruntować nabyte do tej pory umiejętności w zakresie Linuksa. Mamy nadzieję, że wypróbujesz ten projekt — jeśli to zrobisz, Twoja biegłość w posługiwaniu się wierszem poleceń znacznie się poprawi.

Trochę informacji na temat Gita

Git to system DVCS opracowany przez Linusa Torvaldsa, twórcę jądra systemu Linux. Początki Gita sięgają 2005 r., kiedy to doszło do zerwania relacji pomiędzy społecznością jądra Linuksa a autorskim rozproszonym systemem kontroli wersji o nazwie *BitKeeper*.

W odpowiedzi na to Torvalds starał się stworzyć darmowy system DVCS o otwartym kodzie źródłowym, który spełniałby potrzeby procesu rozwoju jądra Linuksa. W ciągu zaledwie kilku dni opracował koncepcję i położył podwaliny pod Gita.

Traktujący priorytetowo wydajność, bezpieczeństwo, elastyczność i nieliniarny rozwój (wspierający tysiące równoległych gałęzi) Git szybko zyskał popularność w społeczności programistów. Jego konstrukcja, w której położono nacisk na szybkość, integralność danych i wsparcie dla rozproszonych przepływów pracy, sprawiła, że stał się ulubionym systemem wśród programistów, co z kolei uczyniło go standardem kontroli wersji w branży oprogramowania.

Czym jest rozproszony system kontroli wersji?

Tradycyjne systemy kontroli wersji, takie jak na przykład **współbieżny system kontroli wersji** (ang. *concurrent versions system* — CVS), wykorzystują centralny serwer, który przez cały czas utrzymuje jeden, spójny stan repozytorium. Systemy te pozwalały programistom wysyłać i pobierać kod oraz umożliwiały korzystanie z gałęzi, tagów i innych znanych mechanizmów. Istotne jest to, że te systemy kontroli wersji zostały zaprojektowane z myślą o centralnym serwerze.

Git i inne systemy DVCS, takie jak **Mercurial** i **Fossil**, stosują odmienne podejście. Każdy programista ma własne kompletne repozytorium. Inni programiści, zamiast przechodzić przez centralny serwer, pobierają zmiany z repozytoriów innych osób. W przypadku projektu Linuksa istnieją setki niezależnych repozytoriów używanych przez programistów. Gdy programista uznaje, że stan jednego z tych repozytoriów jest gotowy, prosi o pobranie zmian do jądra głównego. Stąd właśnie pochodzi termin **pull request**, czyli żądanie pobrania.

Podczas gdy GitHub, GitLab, sourcehut i inne systemy zapewniają scentralizowany hosting dla Gita, dbając o takie kwestie jak autoryzacja użytkowników i zapewniając wiele innych funkcjonalności związanych z rozwojem projektów oprogramowania, Git radzi sobie dobrze bez żadnego z nich i zapewnia w tym celu wiele mechanizmów. Możliwe jest nawet wysyłanie i odbieranie poprawek i grup commitów za pomocą poczty elektronicznej, bez opuszczania wiersza poleceń i Gita. Ułatwia to współpracę, nawet jeśli współautor ma tylko adres e-mail, na który może wysłać poprawkę.

Podstawy Gita

Oto krótkie przypomnienie najważniejszych podstaw wiersza poleceń Gita. Choć nie użyliśmy formy instrukcji krok po kroku, napisaliśmy je tak, aby umożliwić Ci śledzenie przykładów, jeśli zechcesz poćwiczyć.

Pierwsza konfiguracja

Przed wszystkim, jeżeli uruchamiasz Gita po raz pierwszy, zapewne zechcesz ustawić kilka globalnych opcji konfiguracyjnych.

Ustaw domyślną nazwę gałęzi jako `main`:

```
git config --global init.defaultBranch main
```

Następnie skonfiguruj domyślną nazwę i adres e-mail (dołączane do wszystkich commitów):

```
git config --global user.email "ty@example.com"
git config --global user.name "Twoja nazwa"
```

Teraz możesz zainicjalizować nowe repozytorium Gita.

Inicjalizowanie nowego repozytorium Gita

Utwórz katalog i wejdź do niego:

```
mkdir my-repo
cd my-repo
```

Teraz poinstruj Gita, że chcesz zainicjalizować ten katalog jako nowe repozytorium Gita:

```
git init
```

Wprowadzanie i obserwowanie zmian

Utwórz plik z prostą zawartością i pokaż wynikową zmianę wykrytą przez Gita:

```
echo "Hello World" >> README
git status
```

Przechowywanie i zatwierdzanie zmian

Umieść w przechowalni (ang. *stage*) dokonaną zmianę, która ma zostać zatwierdzona, i zaobserwuj, jak zmieniły się dane wyjściowe z polecenia `git status`:

```
git add README
git status
```

Pokaż zawartość umieszczoną w przechowalni:

```
git diff --staged
```

Zatwierdź (ang. *commit*) zmiany z przechowalni:

```
git commit -m 'Add README file'
```

Jest to skrócona forma polecenia `commit`, która bezpośrednio określa komunikat (`-m`). Istnieje interaktywna wersja polecenia `commit`, a mianowicie `git commit`.

Interaktywna wersja tego polecenia (bez opcji `-m`) otworzy edytor tekstu określony w zmiennej środowiskowej `EDITOR` powłoki, a po zapisaniu pliku i wyjściu z edytora — czyli po powrocie z polecenia `$EDITOR — commit` zostanie zapisany.

Opcjonalne dodawanie zdalnego repozytorium Gita

Poniższe polecenie spowoduje dodanie zdalnego repozytorium z lokalną nazwą `origin`, do którego Git może wysyłać i z którego może pobierać. Przypomina ono polecenie logowania SSH, które omówiliśmy w rozdziale 13. „Bezpieczny dostęp zdalny za pomocą SSH”, ponieważ dokładnie to samo będzie robił Git w tym przypadku. Git obsługuje również inne protokoły, takie jak HTTPS.

```
git remote add origin git@example.org:repo-path
```

To tylko przykład, ale gdy będziesz pracować z prawdziwym repozytorium — na przykład takim, które istnieje w GitHubie — zmienisz nazwę hosta i `repo-path`, aby odpowiadały repozytorium, które chcesz dodać. GitHub i inne narzędzia do hostowania źródeł mają czytelną dokumentację, w której znajdziesz informacje, jak to zrobić dla hostowanych tam repozytoriów.

Wysyłanie i pobieranie

Wysyłanie (ang. *push*) zmian z bieżącej gałęzi do zdalnego repozytorium Gita:

```
git push -u origin HEAD
```

Pobieranie (ang. *pull*) zmian ze zdalnej gałęzi:

```
git pull
```

Klonowanie repozytorium

Sklonujmy zdalne repozytorium — cały kod dla opracowanego przez nas kursu Linuksa opartego na projekcie:

```
git clone https://github.com/groovemonkey/hands_on_linux-self_hosted_
wordpress_for_linux_beginners
```

Spowoduje to pobranie historii Gita dla bazy kodu i ustawienie podanego adresu URL dla pochodzenia zdalnego repozytorium. Następnie możesz pracować nad bazą kodu, używając wszystkich poleceń Gita, które znasz już z tego rozdziału.

Tak jak poprzednio, możesz sprawdzić status repozytorium:

```
git status
```

Chociaż polecenie to jest zwykle używane między innymi do sprawdzania, co zostało zmodyfikowane, dostarcza również informacje o trwających scalaniach, pokazuje pliki, których dotyczy konflikt scalania, oraz pomaga podczas bisectingu kodu. Gdy nie masz pewności, co się dzieje, warto sprawdzić `git status` — istnieje prawdopodobieństwo,

że znajdujesz się w specjalnym stanie Gita, z którego chcesz wyjść albo zatrzymać jego kontynuowanie.

Skoro omówiliśmy najczęściej stosowane polecenia, wyjaśnimy terminologię, która sprawia problemy osobom początkującym w pracy z Gitem.

Terminologia

Bardzo pomocne może być podstawowe zrozumienie słownictwa Gita. Gdy inne oprogramowanie stosuje te terminy w innym znaczeniu, znajomość ich znaczenia w przypadku Gita umożliwi dużo pewniejszą pracę, na przykład podczas rozwiązywania problemów i odczytywania komunikatów o błędach.

Oto przegląd najpopularniejszych terminów i ich znaczenia.

Repozytorium

Jest to zasadniczo „projekt”, czyli katalog główny kodu (ten, w którym znajduje się katalog *.git*), który jest zarządzany i śledzony przez kontrolę wersji. Repozytorium przechowuje kod źródłowy, jego historię i zmiany.

Gołe repozytorium

To pojęcie ma podobne znaczenie, tylko że kod nie jest klonowany. Odpowiada temu, co zawiera katalog *.git*. Na serwerach hostujących repozytoria, takich jak GitHub, GitLab, sourcehut lub firmowe instancje Gogsa bądź Gitei, gołe repozytorium znajduje się zwykle w katalogu *nazwa-projektu.git*, zawierającym tylko to, co można zobaczyć w sklonowanym repozytorium znajdującym się w katalogu *nazwa-projektu/.git*.

Gałąź

Jeśli wyobrazić sobie pierwszy commit jako załączek nowego repozytorium, projekt składa się z różnych gałęzi. Istnieje gałąź główna (ang. *main*; opisana dalej w rozdziale) i często co najmniej jedna gałąź boczna, zawierająca kod dla innego kierunku, w którym zmierza projekt.

Mogą to być gałęzie podstawowej wersji, które mają zastosowane poprawki błędów, ale nie zostaną scalone z powrotem z gałęzią główną. Mogą to być eksperymenty, które mogą zostać scalone z gałęzią główną, ale nie muszą. Mogą to być też nowe funkcjonalności lub gałęzie dla poprawek błędów, które są nadal w fazie rozwoju, ale zostaną scalone, gdy tylko będą gotowe. Możliwości są nieograniczone.

Gałąź *main/master*

Gałąź główna (ang. *main*) to domyślna gałąź, która będzie używana podczas inicjalizowania lub klonowania repozytorium. W zależności od projektu zwykle zawiera najnowszy kod w fazie rozwoju lub najnowszy stabilny kod.

HEAD

Jest to najnowszy commit w gałęzi. Jest również czasami określany jako „końcówka” gałęzi. W wierszu poleceń HEAD jest też powszechnie używane w połączeniu z commitami względnymi.

Na przykład HEAD~2 odnosi się do dwóch commitów wstecz, dlatego poniższe polecenie wyświetli dziennik z dwoma ostatnimi commitami:

```
git log HEAD~2
```

W skryptach i codziennym użyciu to polecenie można stosować jako alternatywę dla bieżącej gałęzi, ponieważ jest jej końcówką.

Tag

Tagi (znaczniki) służą do oznaczania konkretnych commitów, na przykład w celu utworzenia określonej wersji bazy kodu (i późniejszego odwoływania się do niej).

Płytkie klonowanie

Zazwyczaj określenie „płytkie” odnosi się do klonowania, które nie zawiera historii lub zawiera jej bardzo niewiele. Płytkie klonowania są stosowane, gdy Git jest wykorzystywany tylko jako środek do uzyskania kodu, a nie pełnego repozytorium i jego historii. Może to jednak uniemożliwić działanie określonych poleceń i narzędzi, które zależą od większej ilości historii.

Scalanie

Scalanie to proces integracji kodu z jednej gałęzi z kodem z drugiej. Może mieć miejsce w różnych scenariuszach, takich jak scalanie gałęzi funkcjonalności z gałęzią główną, pobieranie zmian z gałęzi zdalnej, pobieranie kodu ze stashu (schowka) Gita itd. Scalanie może się odbywać w sposób w pełni zautomatyzowany. Czasami scalenie może wymagać ręcznego działania, jak w przypadku konfliktów scalania.

Commit scalający

Jest to commit wynikający ze scalenia kodu. Podczas scalania kodu samo scalenie staje się commitem. W przypadku konfliktu scalania commit scalający będzie zawierał zmiany rozwiązujące ten konflikt. Chociaż jest to technicznie możliwe, dodawanie do takiego commitu jakichkolwiek innych zmian (takich jak dodatkowe poprawki błędów) nie jest dobrym pomysłem. Commity scalające powinny zawierać tylko te zmiany, które są potrzebne do wykonania tego konkretnego scalenia.

Bezkonfliktowe scalenia, które Git obsługuje automatycznie, często są zatwierdzane bez żadnych ręcznych zmian w kodzie lub komunikatów.

Konflikt scalania

Gdy Git nie jest pewien, jak scalić przychodzący kod, spowoduje to konflikt scalania, który należy rozwiązać ręcznie, zwykle za pomocą narzędzia do scalania. Takie konflikty mogą wystąpić, kiedy kod jest pobierany lub stosowany ze stashu, podczas scalania gałęzi albo w trakcie jakiegokolwiek innej czynności, która działa na aktualnie klonowany kod. Konflikty scalania muszą zostać rozwiązane, a następnie zatwierdzone. Polecenie `git status` zazwyczaj podpowie Ci, jak postępować.

Stash

Czasami konieczne jest odłożenie zmian na później. Git zapewnia do tego mechanizm zwany stashem (schowkiem). Stash ma strukturę przypominającą stos, co ułatwia przyrostowe wprowadzanie zmian w kolejności przez zdejmowanie ich ze stashu za pomocą polecenia `git stash`.

Pull request

Git to rozproszony system kontroli wersji, co oznacza, że każdy programista ma własne pełne repozytorium, a zatem może pracować niezależnie od innych programistów pracujących nad tym samym projektem.

Wyobraźmy sobie programistę Stefana, który wprowadza zmiany w kodzie w swoim repozytorium. Chce, aby inna programistka, Sara, zintegrowała te zmiany z bazą kodu przed nadchodzącym wydaniem oprogramowania. Stefan prosi Sarę o pobranie tych zmian do jej własnego repozytorium — jak pokazaliśmy wcześniej w tym rozdziale, stąd pochodzi termin „żądanie pobrania” (ang. *pull request*).

Ponieważ wiele firm i projektów nie wykorzystuje Gita jako systemu DVCS, preferując centralne, autorytatywne repozytorium kodu, z którego wszyscy programiści pobierają kod i do którego go przesyłają, termin „żądanie pobrania” jest obecnie używany zwykle do opisanego prośby o dodanie kodu do tego autorytatywnego repozytorium (lub czasami po prostu do głównej gałęzi repozytorium).

Uwaga

Ponieważ koncepcja ta odbiega od zdecentralizowanej natury Gita, nie ma dla niej natywnego słowa Gita. Różne produkty implementujące ten przepływ pracy (aktualizowanie autorytatywnej, centralnej wersji bazy kodu) mają odmienne nazwy: GitHub nazywa to *pull request*, Launchpad *merge proposal*, a GitLab *merge request*.

Cherry-picking

Czasami sensowne jest uzyskanie tylko pojedynczych zmian (commitów) z innej gałęzi. Typowym przykładem są poprawki błędów w gałęzi rozwoju oprogramowania, takiej jak gałąź funkcjonalności, która powinna zostać dodana do gałęzi stabilnej w celu jej

wydania. Można to zrobić za pomocą funkcji cherry-picking („zrywanie wisienek”). W przeciwieństwie do scaleń, w których scalana jest cała gałąź, cherry-picking pozwala określić poszczególne commity do dodania.

Bisecting

Polecenie `git bisect` to sposób na szybkie znalezienie commitu powodującego zmianę. Zwykle jest używane do identyfikowania, który commit wprowadził określony błąd. W tym celu należy określić znany „zły” i znany „dobry” commit. Zły commit zawiera błąd, a dobry jest prawidłowy. Git wyświetli commity, których można użyć do przetestowania błędu. Oto przykład:

```
git bisect start
git bisect bad
git bisect good a0634a0
```

Bisecting: 675 revisions left to test after this (roughly 10 steps)

Pierwsza linia rozpoczyna bisecting. W drugiej informujemy Gita, że obecna wersja jest zła, więc zawiera błąd. Ponieważ wiemy, że commit `a0634a0` jest dobry, określamy to w trzeciej linii. Oczywiście nie musi to być wyłącznie commit, może też być tag lub gałąź. Git poinformuje wtedy, ile wersji należy jeszcze sprawdzić.

Następnie należy przetestować kod pod kątem błędu, który próbujemy znaleźć. Jeśli błąd jest obecny, wpisujemy `git bisect bad`, w przeciwnym razie `git bisect good`. Potem powtarzamy te czynności. Ostatecznie otrzymujemy dokładnie ten commit, który wprowadził błąd.

Jeśli chcesz wyjść z tego trybu i powrócić do wcześniejszego stanu, wpisz `git bisect reset`.

„Dobre” i „złe” mogą nie być najlepszymi określeniami, gdy próbujesz znaleźć inny rodzaj zmiany zachowania. Można więc zamiast nich użyć określeń „stare” i „nowe”, aby znaleźć commit wprowadzający nowe zachowanie. Należy pamiętać, że nie można mieszać tych określeń. Mamy więc *dobrze* i *złe* albo *stare* i *nowe*.

Istnieją również sposoby na przyspieszenie tego procesu, takie jak określanie plików lub katalogów, jeśli wiesz, gdzie dane zachowanie zostało wprowadzone. Jeżeli wiesz, że zmiana musi być związana z zawartością *jakiś/katalog* lub *jakiś/inny/katalog*, możesz zawęzić wyszukiwanie w ten sposób:

```
git bisect start -- jakiś/katalog jakiś/inny/katalog
```

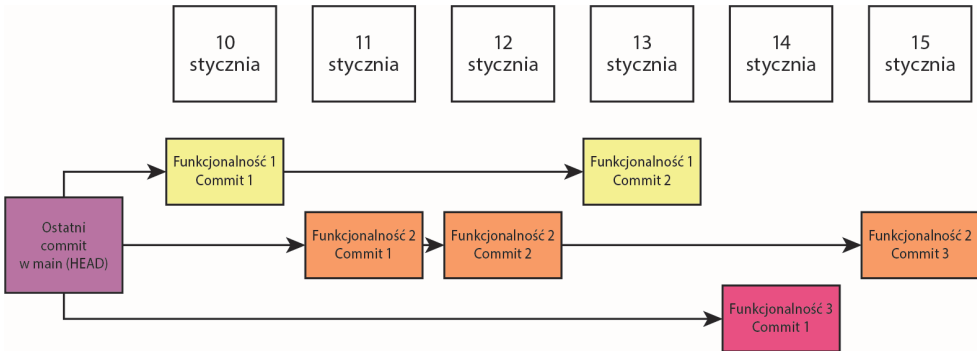
Git uwzględni tylko te commity, które wprowadzają zmiany w podanych ścieżkach.

Istnieje jeszcze więcej sposobów na przyspieszenie tego procesu, takich jak określenie wielu dobrych commitów lub nawet przekazanie skryptu testowego, który w zależności od kodu wyjścia automatycznie znajdzie commit. Przyda się również rzut oka na `man git-bisect`, jeżeli trzeba przeanalizować wiele commitów.

Rebasing

Polecenie `git rebase` jest powszechnym sposobem na utrzymanie łatwego śledzenia historii commitów przez odtwarzanie danego zestawu zmian (jak gałąź funkcjonalności) na nowym commicie bazowym, zamiast na commicie bazowym, w którym zmiany faktycznie nastąpiły.

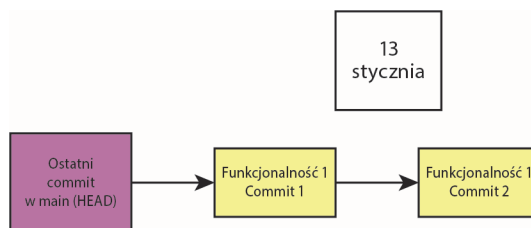
Ponieważ rozwój oprogramowania jest zwykle rozproszony, możesz mieć „prawdziwą” historię commitów, taką jak ta, którą pokazaliśmy na rysunku 14.1.



Rysunek 14.1. „Prawdziwa” historia commitów

Posiadanie wielu historii gałęzi funkcjonalności przeplatających się w historii jest często bardziej mylące niż użyteczne, dlatego rebasing Gita jest używany do usprawnienia commitów tych funkcjonalności podczas ich scalania.

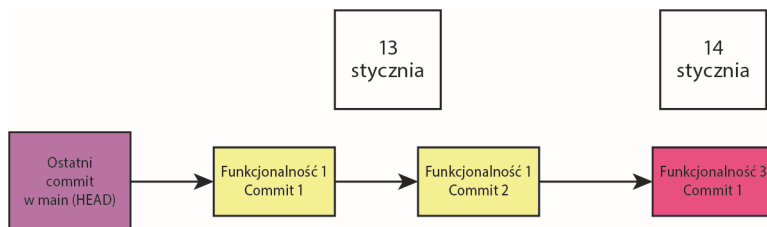
Funkcjonalność 1 jest scalana jako pierwsza, więc korzysta z oryginalnego commitu bazowego. Katalog projektu wygląda teraz tak, jak pokazaliśmy na rysunku 14.2.



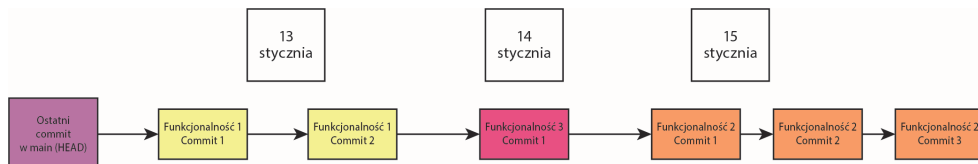
Rysunek 14.2. Funkcjonalność 1 rebase’owana (scalona) 13 stycznia

Funkcjonalność 3 jest kolejną, która zostanie rebase’owana i scalona, historia wygląda więc teraz tak, jaki widać na rysunku 14.3.

Wreszcie, rebase’owana jest funkcjonalność 2, w wyniku czego jej bazowy commit zmienia się na commit funkcjonalności 3 z 14 stycznia. Teraz mamy ładną, uproszczoną historię, którą pokazaliśmy na rysunku 14.4.



Rysunek 14.3. Funkcjonalność 3 rebase'owana (scalona) 14 stycznia



Rysunek 14.4. Funkcjonalność 2 rebase'owana (scalona) 15 stycznia

GitHub i inne scentralizowane hosty repozytoriów Gita mają funkcje, które automatyzują ten proces podczas scalania, więc rzadko trzeba przeprowadzać rebasing ręcznie w wierszu poleceń. Poniżej pokazaliśmy jednak, jak wygląda ta procedura:

1. Utwórz nową gałąź i dodaj commit:

```
git checkout -b dave/myfeature
git commit -m "wprowadziłem kilka zmian"
```

2. Zakładając, że gałąź bazowa nosi nazwę `main`, a od czasu rozpoczęcia rozwijania tej gałęzi dodano kilka commitów, możesz teraz „rebase’ować na `main`”:

```
git rebase main
```

Spowoduje to zmodyfikowanie historii Gita, aby przeprowadzić rebasing commitów gałęzi na najnowszym commicie `main`, jak pokazaliśmy wcześniej na rysunku 14.4. Ponieważ zmieniasz istniejącą historię, może to wymagać wymuszenia wysłania do autorytatywnego repozytorium (takiego jak repozytorium GitHuba), co może spowodować konflikty u innych użytkowników. Należy o tym pamiętać podczas rebasingu.

Skoro omówiliśmy już kilka kluczowych terminów i koncepcji, które napotkasz podczas korzystania z Gita, możemy przyrzeć się kilku dobrym praktykom dotyczącym pisania skutecznych komunikatów commitów.

Najlepsze praktyki dotyczące komunikatów commitów

Stosowanie ogólnej zasady „jedna zmiana na commit i jeden commit na zmianę” to sposób na zachowanie użyteczności commitów i historii Gita.

Istnieje wiele sytuacji, w których możesz pracować tylko nad jedną dużą zmianą, ale także dodawać do kodu kilka drobnych (niepowiązanych) poprawek i ulepszeń. Te niepowiązane ze sobą zmiany powinny być jednak wprowadzane oddzielnie. Dobrym pomysłem jest skoncentrowanie poszczególnych commitów na jednej konkretnej rzeczy, którą chcesz osiągnąć: drobna poprawka, poprawienie literówki, zmiana stylu, dodanie (jednej) funkcjonalności itd. Nawet jeśli ostatecznie wprowadzisz wiele powiązanych ze sobą zmian jednocześnie, nadal może mieć sens podzielenie ich później na wiele commitów. Częstsze zatwierdzanie może znacznie ułatwić ten proces.

Istnieje wiele powodów dla stosowania tej zasady. Jednym z najbardziej praktycznych jest to, że gdy commity są małe, poszczególne zmiany można łatwo wybrać za pomocą cherry-pickingu lub cofnąć, gdy okaże się to konieczne (nawet jeśli nigdy nie spodziewałeś się tego podczas wprowadzania zmiany w kodzie). Posiadanie małych, ściśle skoncentrowanych commitów jest również pomocne, gdy ktoś używa `git blame`, aby zrozumieć jakąś zmianę.

Dobre komunikaty o commitach

Czasami niejasne sugestie, takie jak na przykład „zachowaj krótki komunikat commitu, gdy używasz `git commit`”, mogą być mylące i trudne do prześledzenia. Dla kontekstu warto wyjaśnić, w jaki sposób powinno się używać Gita. Git, który jest systemem DVCS, pozwala wysyłać poprawki jako wiadomości e-mail. W rezultacie same wiadomości o commitach w pewnym sensie przybierają formę e-maili. Pierwsza linia jest uważana za linię tematu, zawierającą krótki przegląd tego, co zostało zrobione, po czym następuje pusta linia i bardziej szczegółowe podsumowanie tego, co zostało zmienione.

Ponieważ jest to bardzo elastyczny schemat, istnieją pewne powszechnie przyjęte zasady. Jak wszystkie tego typu zasady, mogą one zostać zmienione w zależności od projektu lub organizacji, ale oto co znajdziesz w wielu uznanych projektach otwartoźródłowych:

- Pierwsza linia powinna być krótka. Powinno to być podsumowanie składające się z maksymalnie 72 znaków.
- Pierwsza linia powinna zaczynać się czasownikiem w trybie rozkazującym (np. Dodaj . . . , Napraw . . .).
- Temat powinien być zapisany wielkimi literami.
- Jeśli potrzebujesz więcej, dodaj pustą linię i pełne podsumowanie.
- Użyj miejsca na treść, aby wyjaśnić, *dłaczego* dokonałeś tej zmiany. Może to być bardzo pomocne dla każdego przyszłego czytelnika używającego `git blame`.
- Pamiętaj, aby opisać, w jaki sposób doszedłeś do swoich wniosków (lub implementacji) i dlaczego jest to istotne. Jest to szczególnie ważne w przypadku złożonych commitów i commitów, które mogą nie mieć oczywistego sensu dla kogoś, kto patrzy tylko na kod. Może to być ogromna pomoc podczas śledzenia błędów, późniejszego usuwania przestarzałego kodu, przepisywania systemów lub po prostu próby zrozumienia kodu.
- Zastanów się, czy część tego, co piszesz w komunikacie o commicie, nie byłoby lepiej dodać jako komentarze do kodu.

- Wyobraźmy sobie osobę przeprowadzającą inspekcję lub przyszłego czytelnika bez żadnego kontekstu. Upewnij się, że zmiany w kodzie są zrozumiałe.

Mając na uwadze te porady, będziesz w stanie tworzyć jasne i dobrze sformułowane komunikaty. W następnym podrozdziale przyjrzymy się kilku kolejnym poradom dotyczącym łatwego i skutecznego korzystania z Gita.

Graficzne interfejsy użytkownika

Chociaż ta książka koncentruje się na rozwijaniu umiejętności pracy z wierszem poleceń, warto wspomnieć, że dostępne są pewne narzędzia graficzne, które niekiedy mogą ułatwić interakcję z Gitem.

Tig i gitk to przykłady graficznych przeglądarek repozytoriów, które zapewniają interfejs Gita podobny do tego, który zapewnia wiele środowisk IDE. Aby je wypróbować, wystarczy przejść do repozytorium za pomocą polecenia `cd` i uruchomić `gitk` lub `tig`. Prawdopodobnie będziesz musiał zainstalować te narzędzia za pomocą menedżera pakietów, ponieważ wiele wersji Uniksa (w tym popularne dystrybucje Linuksa i systemu macOS) nie ma ich zainstalowanych domyślnie.

Przydatne aliasy powłoki

Oto kilka przydatnych aliasów powłoki dla popularnych poleceń Gita. Możesz dodać je do pliku `~/.bash_aliases` (o ile używasz Basha):

```
alias gpo='git push origin $(git branch | grep "*" | cut -d " " -f2)'  
alias gp='git pull'  
alias gs='git status'  
alias gd='git diff'  
alias gds='git diff --staged'
```

Jeśli wpisujesz `git status` dziesiątki razy każdego dnia, dodanie aliasu umożliwiającego wpisanie zamiast tego `gs` może być ogromnym usprawnieniem. Możesz zmienić to na coś jeszcze wygodniejszego — od tego jest właśnie personalizacja!

Teraz przybliżmy nieco temat i zobaczmy, jak można praktycznie wykorzystać całą tę wiedzę podczas budowania niewielkiego projektu serwera linuksowego: własnego prywatnego serwera Gita.

GitHub dla ubogich

W tym podrozdziale pokażemy, jak skonfigurować dla siebie zdalne repozytorium Gita. Potrzebne są tylko konto SSH na komputerze zdalnym i plik binarny Gita na komputerze lokalnym (np. samo polecenie Gita). Jeśli Git jest już zainstalowany na zdalnym komputerze, nie będzie nawet potrzebny dostęp z uprawnieniami roota.

Jest to fajny projekt, który sprawi, że ugruntujesz wiedzę z zakresu podstawowych koncepcji systemu operacyjnego związanych z Gitem. Ta konfiguracja niekoniecznie jest sugerowana do użytku w środowisku produkcyjnym; raczej pokaże Ci, że nie ma absolutnie żadnej magii, jeśli chodzi o Gita. Jak wszystko inne w Linuksie, to tylko pliki (w tym przypadku zdalne pliki i tunel SSH).

Uwagi wstępne

W zależności od tego, czy masz dostęp roota i czy chcesz udostępniać repozytorium innym, możesz rozważyć utworzenie określonego użytkownika dla współdzielonej usługi Gita. Jest to całkowicie opcjonalne.

Do uwierzytelnienia użyjemy konta SSH, więc jeśli udostępnisz repozytorium Git jakiejś osobie, będzie ona miała na zdalnym komputerze takie same uprawnienia jak ten użytkownik, którego użyjesz. Sensowne może być utworzenie dla tego projektu osobnego użytkownika na zdalnej maszynie (o nazwie `git`), jeśli nie ufasz w pełni innym programistom, którzy będą mieli dostęp do tego konta.

Ten projekt zakłada, że jesteś w stanie skonfigurować SSH i połączyć się z serwerem — jeśli nie pamiętasz wszystkich szczegółów, wróć do rozdziału 13. „Bezpieczny dostęp zdalny za pomocą SSH”.

1. Łączenie z serwerem

Połącz się z serwerem, używając konta, do którego ma należeć repozytorium (np. `git` lub własnego użytkownika):

```
ssh myuser@example.com
```

2. Instalowanie Gita

Najpierw sprawdź, czy Git jest już zainstalowany — w tym celu spróbuj go uruchomić:

```
git version
```

Polecenie to powinno wyświetlić numer wersji Gita zainstalowanej na serwerze. Jeżeli otrzymasz komunikat typu `command not found` (nie znaleziono polecenia, będzie to oznaczać, że Git nie jest zainstalowany w Twoim systemie).

Aby zainstalować Gita, wystarczy użyć systemowego menedżera pakietów w celu zainstalowania pakietu `git`. W Ubuntu należy uruchomić następujące polecenie:

```
apt-get install git
```

3. Inicjalizowanie repozytorium

Teraz możesz zainicjalizować nowe gołe repozytorium Gita. W tym przypadku nazwiemy je `my-project`. Możesz je utworzyć w dowolnym miejscu. Dla uproszczenia przyjmiemy, że jest to katalog domowy użytkownika:

```
git init --bare my-project.git
```

Spowoduje to utworzenie katalogu o nazwie *my-project.git*. Nie jest to plik, lecz struktura katalogów, którą Git traktuje jako repozytorium. Nie będziemy tutaj wchodzić w szczegóły i prawdopodobnie minie sporo czasu, zanim będziesz musiał coś zmienić.

Możesz wierzyć lub nie, ale to już wszystko, co musisz zrobić!

Możesz teraz rozłączyć się z serwerem (*Ctrl+D*, jeśli jesteś połączony przez SSH).

4. Klonowanie repozytorium

Pomimo że jest ono całkowicie puste, możesz już sklonować repozytorium. Po rozłączeniu się z serwerem uruchom następujące polecenie z komputera lokalnego:

```
git clone myuser@example.com:my-project.git
```

Jak wspomnieliśmy wcześniej, przyjęliśmy założenie, że repozytorium zostało utworzone w katalogu domowym użytkownika *myuser*. Jeżeli zaczynasz od nazwy hosta *example.com* (może to być również adres IP serwera, jeśli nie skonfigurowano DNS-u), ścieżka jest względna wobec katalogu domowego użytkownika. Jeśli chcesz określić pełną (bezwzględną) ścieżkę, po prostu zacznij od ukośnika. Innymi słowy, użycie polecenia `git clone myuser@example.com:/home/myuser/my-project.git` doprowadziłoby do sklonowania tego katalogu.

Git ostrzeże Cię, że sklonowałeś puste repozytorium. Ponieważ jednak właśnie tego oczekujemy, nie ma powodu do zmartwień.

5. Dokonaj edycji projektu i wyślij zmiany

Możemy teraz przejść do sklonowanego katalogu i rozpocząć pracę nad projektem:

```
cd my-project
echo "Mój osobisty projekt" >> README
git add README
git commit -m 'początkowy commit'
```

Gdy masz już pierwszy commit, możemy go wysłać. Pierwszy push request ma drobne zastrzeżenie, o którym należy pamiętać: ponieważ repozytorium jest nadal całkowicie puste, nie zna jeszcze żadnych gałęzi, nawet gałęzi głównej. Git poinformuje Cię o tym, jeśli tylko uruchomisz `git push`. Upewnij się więc, że podczas pierwszego wysyłania do nowego repozytorium przekazujesz Gitowi gałąź:

```
git push origin master
```

To wszystko!

Teraz Ty lub ktoś inny z dostępem do Twojego konta SSH może klonować to repozytorium, wysłać do niego kod i go pobierać. Możesz nawet ustawić zaczepy i zrobić inne fajne rzeczy. Git to bardzo potężne narzędzie z ogromną różnorodnością funkcjonalności. Przyzwyczajanie się do nich może zająć trochę czasu, więc potraktuj to jako punkt wyjścia.

Możliwości są nieograniczone i zawsze cieszymy się, gdy słyszymy, że ktoś wykorzystuje Gita do interesujących lub unikatowych przypadków użycia. Miłej zabawy!

Podsumowanie

W tym rozdziale omówiliśmy podstawowe pojęcia, polecenia i przepływy pracy, które są potrzebne do efektywnego korzystania z Gita. Niektóre z często używanych zaawansowanych funkcjonalności i terminów powinny być teraz jaśniejsze, a ponadto przekazaliśmy kilka porad dotyczących „miękkich” umiejętności w zakresie Gita, takich jak pisanie dobrych komunikatów commitów.

Alias powłoki, które pokazaliśmy, pozwalają zaoszczędzić setki naciśnięć klawiszy w ciągu jednego dnia programowania. Mamy nadzieję, że aliasy okażą się tak samo przydatne dla Ciebie, jak są dla nas, i że będziesz używać ich każdego dnia dla wszystkich trudnych do zapamiętania lub wpisania poleceń.

Mamy również nadzieję, że śledziłeś projekt *GitHub dla ubogich*! Uruchomienie tych poleceń zajmuje tylko kilka minut, ale jeśli poświęcisz popołudnie i naprawdę je wypróbujesz (wynajmiesz na kilka godzin maszynę wirtualną z Linuksem, skonfigurujesz tam zdalne repozytorium i wyślesz kilka przykładowych commitów), poczujesz, jak skuteczne mogą być Twoje nowo zdobyte linuksowe umiejętności, gdy połączysz je, by rozwiązywać rzeczywiste problemy.

Konteneryzacja aplikacji za pomocą Dockera

Rozdział

15

W ciągu ostatniej dekady konteneryzacja z wykorzystaniem Dockera stała się rodzajem domyślnego formatu pakowania dla aplikacji internetowych i nowoczesnych mikrouслуг. W kontenerze program znajduje się w bardzo lekkich, odizolowanych warstwach abstrakcji powłoki linuksowego systemu plików, procesów, użytkowników i sieci, które są bezpiecznie oddzielone od środowiska hosta. Obrazy kontenerów są również niezwykle przenośne: można je łatwo przenosić z laptopa programisty do środowiska testowego lub przejściowego, a następnie na serwer produkcyjny. Rozwiązuje to wiele problemów, które nękały oprogramowanie i infrastrukturę w ciągu ostatnich kilku dekad.

W pewnym sensie kontenery są dość podobne do pakietów Linuksa, które umiesz już instalować z repozytoriów. Obraz kontenera to zasadniczo skompresowane archiwum (np. plik *.tar.gz*) aplikacji wraz ze wszystkimi plikami konfiguracyjnymi i zależnościami, których dana aplikacja potrzebuje. Ten mały pakiet — obraz — może być wykonywany przez Dockera. Rewolucyjne w takim kontenerze jest to, że zgrabnie przechowuje wszystko razem w jednym artefakcie i może działać w dowolnym systemie Linux, który ma zainstalowane środowisko uruchomieniowe kontenera (takie jak Docker).

Rozdział ten z łatwością mógłby stanowić samodzielną książkę, Docker i kontenery linuksowe to bowiem dość obszerne tematy. Jednak, podobnie jak w przypadku wszystkich innych zagadnień omówionych w tej książce, skupiamy się tylko na podstawowej teorii i praktycznych umiejętnościach, które są niezbędne do wygodnej interakcji w aplikacjach z infrastrukturą opartą na Dockerze.

W tym rozdziale omawiamy następujące zagadnienia:

- rozwój aplikacji i problemy operacyjne rozwiązywane przez kontenery;
- czym są kontenery i w jaki sposób przypominają pakiety Linuksa;
- różnica między obrazami a kontenerami Dockera;
- wszystkie praktyczne podstawy korzystania z Dockera w procesie tworzenia oprogramowania;
- tworzenie własnych obrazów kontenerów za pomocą plików Dockerfiles (skonteneryzujesz prawdziwą aplikację internetową Pythona);

- kilka bardziej zaawansowanych tematów, takich jak różnice między maszynami wirtualnymi a kontenerami oraz sposób, w jaki Linux tworzy warstwę abstrakcji kontenerów za pomocą przestrzeni nazw;
- kilka wskazówek, sztuczek i najlepszych praktyk dotyczących kontenerów.

Przejdźmy do rzeczy.

Dlaczego kontenery działają jako pakiety?

Docker stał się standardowym narzędziem do pakowania oprogramowania w sytuacji, gdy celem jest dołączenie systemu, o którym wiadomo, że jest działającą konfiguracją. Zazwyczaj kontener Dockera zawiera zarówno oprogramowanie, które chcesz uruchomić, jak i cały, choć często okrojony, system Linux jako środowisko wykonawcze. To środowisko wykonawcze zapewnia biblioteki i narzędzia, a także kilka innych rzeczy, takich jak podstawowa konfiguracja systemu, dzięki czemu może funkcjonować jako samodzielna jednostka, niezależna od systemu, na którym działa kontener. Głównym celem jest, aby aplikacja mogła być z powodzeniem uruchamiana na maszynie programisty, w środowiskach produkcyjnych i testowych oraz w innych miejscach bez konieczności przejmowania się takimi szczegółami jak wersje systemu operacyjnego i zainstalowane biblioteki.

Należy pamiętać, że system operacyjny i biblioteki nie znikają. Błędy w bibliotekach mogą nadal istnieć, a wszelkie spakowane zależności powinny zostać zaktualizowane między innymi ze względów bezpieczeństwa. Jednak konsumenci spakowanego oprogramowania, np. użytkownicy końcowi, operatorzy systemu oraz wszelkie oprogramowanie do orkiestracji, otrzymują teraz wspólny pakiet i nie muszą przejmować się zależnościami systemowymi. Chociaż szczegóły dotyczące sposobu uruchamiania i konfigurowania oprogramowania nadal zależą od oprogramowania, sposób jego wykonywania (w kontenerze) jest w pewnym stopniu znormalizowany.

Możemy podsumować, że każda konkretna konfiguracja środowiska, taka jak instalacja zależności, jest teraz opisana jako część pliku `Dockerfile`, a po utworzeniu działającego obrazu kontenera bez określonej konfiguracji oczekuje się, że kontener będzie działał w dowolnym systemie zdolnym do uruchomienia Dockera lub w szerszym ujęciu obrazów **OCI**.

Uwaga

OCI (ang. *Open Container Initiative*), czyli otwarta inicjatywa kontenerowa, zapewnia standardy określające takie kwestie jak format obrazu i wykonywanie kontenerów Linuksa. Czasami termin ten jest używany jako synonim Dockera w tym sensie, że programista może użyć Dockera do utworzenia obrazu, ale do wykonania w orkiestratorze Docker może w ogóle nie być wykorzystywany.

Nie ma lepszego sposobu na rozpoczęcie pracy niż zainstalowanie Dockera na komputerze i wypróbowanie kilku poleceń, więc zrobmy to.

Wymagania wstępne — instalacja Dockera

Najpierw pobierz i zainstaluj platformę Docker Desktop. Instrukcje na ten temat znajdziesz na stronie <https://docs.docker.com/get-docker/>.

Na stronie <https://docs.docker.com/getstarted/> znajduje się również doskonały oficjalny samouczek, lecz zalecamy, abyś zaczął z jego lekturą do czasu, aż skończysz pracę z tym rozdziałem. Omówimy niektóre podstawy, ale z mniejszym naciskiem na określone flagi wiersza poleceń, a z większym na to, jak będziesz używać tych poleceń i przepływów pracy jako programista aplikacji.

Skoro masz już zainstalowanego Dockera, możemy uruchomić nasz pierwszy kontener!

Przyspieszony kurs Dockera

Obraz Dockera jest „pakietem” z naszej metafory — jest to statyczny artefakt, który jest zapisywany, przechowywany i przenoszony. Staje się **kontenerem**, gdy jest wykonywany na komputerze. Jest to ważne, ponieważ czasami można się spotkać z nieprawidłowym użyciem tych terminów. Obrazy Dockera są niezmienną bazą, na podstawie której uruchamiany jest kontener — działający proces z przestrzenią nazw. Obrazy są wstępnie zbudowanym szablonem, z którego w czasie wykonywania generowane są kontenery.

Są zaprojektowane w taki sposób, aby były niemutowalne: jeśli pobierzesz obraz serwera WWW nginx, a następnie go uruchomisz, wszelkie zmiany wprowadzone w wynikowym kontenerze nie wpłyną w ogóle na obraz bazowy. Przeszkadza to większości programistów przyzwyczajonych do długo działających maszyn wirtualnych, które są dostarczane raz, a następnie uruchamiane i zatrzymywane wiele razy, ale przez cały czas zachowują swój wewnętrzny stan.

Kontenery Dockera są inne. W idealnym przypadku są one zaprojektowane tak, aby były krótkotrwałe i bezstanowe, podczas gdy obrazy, z których są tworzone, działają jako długotrwały szablon, który może posłużyć do tworzenia nieskończonej liczby kontenerów w wielu różnych środowiskach wykonawczych.

Poniżej znajduje się przykład podstawowego przepływu pracy Dockera, który ma na celu zapoznanie Cię z jego najważniejszymi poleceniami. Nie staraj się zapamiętywać tych poleceń, omówimy je szczegółowo dalej w tym rozdziale. Na razie wyjaśnimy tylko, co dzieje się na każdym kroku, abyś mógł przyzwyczaić się do tego, co zobaczysz na ekranie podczas następnego popisu mikrouslug.

Najpierw uruchommy kontener serwera nginx (`docker run`) i interaktywnie (`-it`) uruchommy w nim powłokę Bash (`/bin/bash`):

```
→ ~ docker run -it nginx /bin/bash
```

Spowoduje to wyświetlenie znaku zachęty powłoki dla unikatowego kontenera, który został uruchomiony z obrazu `nginx`. Powłoka Bash kontenera jest teraz połączona z naszym terminalem:

```
root@e96107c9a58e:/#
```

Napiszmy plik o nazwie `test.txt` i sprawdźmy, czy istnieje:

```
root@e96107c9a58e:/# echo "Jestem niemutowalny" >> test.txt
root@e96107c9a58e:/# cat test.txt
Jestem niemutowalny
```

Z powłoki możemy wyjść za pomocą zwykłego polecenia `Ctrl+d`:

```
root@e96107c9a58e:/#
exit
```

Kontener kończy działanie i wracamy do naszej zwykłej powłoki. W tym miejscu dla większości początkujących użytkowników Dockera jest to niejasne, więc uruchommy ponownie pierwsze polecenie, aby jeszcze raz uruchomić kontener z obrazu `nginx` Dockera i sprawdźmy nasz plik:

```
→ ~ docker run -it nginx /bin/bash
root@c3b4d95ab9e6:/# cat test.txt
cat: test.txt: No such file or directory
```

TRZEBA ZGŁOSIĆ BŁĄD! DOCKER JEST USZKODZONY!

W rzeczywistości tak nie jest. To nie jest ten sam kontener, w którym zapisaliśmy plik `test.txt`. Jeśli przyglądałeś się uważnie, zapewne zauważyłeś, że nazwa hosta w wierszu powłoki w drugim kontenerze była inna. Dzieje się tak, ponieważ każde polecenie `docker run` uruchamia nowy kontener z określonego obrazu. Kontenery są zaprojektowane tak, aby działać, kończyć działanie i znikać na zawsze.

Jednak oryginalny kontener nadal jest na liście:

```
→ ~ docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED
STATUS        PORTS     NAMES
c3b4d95ab9e6   nginx    "/docker-entrypoint...." 14 minutes ago
Exited (1) 6 minutes ago    agitated_hofstadter
e96107c9a58e   nginx    "/docker-entrypoint...." 14 minutes ago
Exited (0) 14 minutes ago    nervous_gould
```

Aby się go pozbyć, możesz użyć polecenia `docker rm` z identyfikatorem kontenera, który chcesz usunąć:

```
→ ~ docker rm c3b4d95ab9e6
c3b4d95ab9e6
```

Możliwe jest uruchomienie zatrzymanego kontenera za pomocą polecenia `docker start`:

```
→ ~ docker start e96107c9a58e
e96107c9a58e
```

W tym momencie kontener będzie widoczny na liście procesów Dockera:

```
→ ~ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
PORTS         NAMES
e96107c9a58e   nginx         "/docker-entrypoint...." 18 minutes ago Up 1
second        80/tcp       nervous_gould
```

Następnie można użyć narzędzia `docker exec`, aby wykonać polecenie w tym kontenerze, ponownie uruchamiając i łącząc się z programem powłoki Bash za pomocą `-it`. Stamtąd można wyświetlić stan systemu plików (*test.txt*), który zmodyfikowaliśmy:

```
→ ~ docker exec -it e96107c9a58e /bin/bash
root@e96107c9a58e:/# cat test.txt
Jestem niemutowalny
```

Jednak utrzymywanie kontenerów przez długi czas — modyfikowanie ich stanu oraz zatrzymywanie i ponowne uruchamianie zamiast uruchamiania za każdym razem nowych kontenerów z obrazu — jest odradzane, gdyż prowadzi do wielu tych samych błędów, które pomagają rozwiązać Docker.

Uniknijmy wszystkich tych błędów i usuńmy siłą ten działający kontener raz na zawsze:

```
→ ~ docker rm -f e96107c9a58e
e96107c9a58e
```

Można było również uruchomić `docker stop`, a następnie `docker rm`, ale wymuszenie usunięcia za pomocą `docker rm -f` zatrzyma i wykasuje działający kontener za jednym zamachem.

Możesz się przekonać, że Docker zachęca do stosowania niemutowalnych kontenerów, aby stan nie zaczynał różnić się od obrazu, który jest „źródłem prawdy” dla początkowego środowiska aplikacji. Jeśli chcesz wprowadzić zmiany w obrazie, nie możesz tego zrobić bezpośrednio — obrazy są niemutowalne.

Zmiany powinny być **bezpośrednie i intencjonalne**, co jest ważne dla tworzenia i uruchamiania niezawodnego oprogramowania. Jak to zrobić? Zaczynamy od oryginalnego obrazu, wprowadzamy zmiany w kontrolowany i powtarzalny sposób (nie za pomocą połączenia SSH z serwerem i podejmowania prób uruchamiania tych poleceń), a następnie zapisujemy je, tworząc nowy obraz. Tu do gry wkracza plik `Dockerfile`.

Tworzenie obrazów za pomocą pliku `Dockerfile`

Jeśli kiedykolwiek będziesz musiał zbudować nowy obraz Dockera lub zmodyfikować istniejący — na przykład dla rozwijanej aplikacji internetowej — będziesz intensywnie korzystać z plików `Dockerfile` (oficjalną dokumentację plików `Dockerfile` znajdziesz na stronie <https://docs.docker.com/engine/reference/builder/>).

Duży odsetek nowego oprogramowania będzie już miał dostępny oficjalny (lub przynajmniej otwartoźródłowy, zewnętrzny) plik Dockerfile. Nawet jeśli przed jego użyciem będziesz musiał dokonać pewnych dostosowań, dobrym miejscem do szukania przykładów jest dokumentacja używanego oprogramowania lub frameworku. Te przykłady zwykle trudniej uszkodzić niż własny niestandardowy plik Dockera, gdy zostaną wydane główne aktualizacje dla spakowanego oprogramowania.

Ponadto niektóre frameworki lub środowiska programistyczne, na przykład Spring Boot (Java), mogą generować obrazy Dockera w ramach procesu kompilacji.

Tak więc nawet jeśli istnieje szansa, że nigdy nie przyjdzie Ci samodzielnie dotykać plików Dockerfile, jest ona niewielka, dlatego warto mieć podstawowe pojęcie o tym, jak one działają.

Przyjrzyjmy się bardzo prostemu plikowi Dockerfile, pochodzącemu z otwartoźródłowego projektu serwera HTTP echo (<https://github.com/hashicorp/http-echo>). Powoduje on utworzenie obrazu Dockera, który pakuje plik binarny Go działający jako prosty serwer WWW:

```
FROM alpine

ADD "https://curl.haxx.se/ca/cacert.pem" "/etc/ssl/certs/ca-certificates.crt"
ADD "./pkg/linux_amd64/http-echo" "/"
RUN apk add curl
ENTRYPOINT ["/http-echo"]
```

Zasadniczo tworzy on nowy obraz kontenera przez wykonanie kilku czynności:

1. Użycie obrazu alpine Linuksa jako podstawy do budowania.
2. Pobranie kilku certyfikatów i dodanie ich do warstwy obrazu (zasadniczo dodanie czegoś do systemu plików kontenera wynikowego).
3. Skopiowanie pliku binarnego *http-echo* z katalogu kompilacji do obrazu kontenera.
4. Uruchomienie polecenia instalacji pakietu alpine w celu zainstalowania programu curl.
5. Zdefiniowanie pliku wykonywalnego lub polecenia, które jest uruchamiane po uruchomieniu kontenera zainicjalizowanego z tego obrazu.

Każdy z tych kroków jest wywoływany przez (pisaną wielkimi literami) **instrukcję**, którą parser pliku Dockerfile wie, jak wykonać. Ten konkretny plik Dockerfile wykorzystuje tylko podzbiór dostępnych instrukcji (FROM, ADD, RUN i ENTRYPOINT).

Oto pełny zestaw instrukcji dostępnych podczas tworzenia nowych obrazów Dockera za pomocą pliku Dockerfile:

- ARG — deklaruje argument czasu kompilacji; w zasadzie zmienną, która będzie używana później w kompilacji.
- ENV — zmienne środowiskowe do ustawienia podczas kompilacji, które pozostaną w uruchomionym środowisku kontenera (*nie* tylko podczas kompilacji!). Przyjmuje format *klucz=wartość*.

- FROM — obraz bazowy.
- CMD — zapewnia domyślne polecenie (lub domyślne argumenty ENTRYPOINT) dla działania kontenera po jego uruchomieniu. Można je napisać, ale powinno być zawarte w pliku Dockerfile. Dozwolone jest tylko jedno domyślne polecenie CMD na plik Dockerfile, a jeśli jest więcej niż jedno CMD, liczy się tylko ostatnie.
- ADD — elastyczna instrukcja, która kopiuje pliki i katalogi, dodając je do systemu plików obrazu. Może być również używana do kopiowania plików spoza obrazu lub ze zdalnych adresów URL (za pośrednictwem protokołu HTTP), a także do wykonywania złożonych czynności, takich jak rozszerzanie, dekompresja, dezarchiwizacja itd. Widziałeś ją jako zamiennik dla polecenia curl w powyższym przykładowym pliku Dockerfile do pobrania pliku certyfikatu CA.
- COPY — kopiuje pliki i katalogi (mniej skomplikowane, magiczne i potężne niż ADD).
- LABEL — dodaje metadane obrazu w formacie *klucz=wartość*.
- EXPOSE — informuje konsumenty tego obrazu o protokołach sieciowych i portach, na których będzie nasłuchiwał ten kontener.
- ENTRYPOINT — informuje kontener, jakie polecenie ma zostać uruchomione podczas jego startu. Aby kontener mógł odbierać sygnały spoza procesu kontenera i odpowiadać na nie, należy użyć **postaci wykonywalnej** (ENTRYPOINT ["plik wykonywalny", "parametr1", "parametr2"]).
- RUN *polecenie arg1 arg2* — uruchamia *polecenie* z argumentami *arg1* i *arg2* w powłoce obrazu:
 - RUN ["polecenie", "arg1", "arg2", "argN"] — to samo co powyżej, ale przydatne, aby uniknąć zniekształcania łańcuchów znaków powłoki.
 - Każda instrukcja RUN jest wykonywana w nowej warstwie obrazu (nie zagłębialmy się tutaj w temat warstw, ale warto o tym wiedzieć).
 - RUN --mount — może posłużyć do tymczasowego montowania systemów plików w kontenerze podczas kompilacji, bez kopiowania samych plików do warstwy obrazu.
 - RUN --network i RUN --security — pozwalają zarządzać odpowiednio kontekstem sieciowym i kontenerami z wysokimi uprawnieniami.
- WORKDIR — ustawia katalog roboczy dla instrukcji, które następują w pliku Dockerfile. Odpowiednik cd w systemach operacyjnych typu Unix.
- SHELL — nadpisuje domyślną powłokę używaną do interpretowania poleceń podczas kompilacji Dockera. Polecenia muszą używać postaci wykonywalnej.
- STOPSIGNAL — ustawia sygnał wywołania systemowego, który ten kontener powinien interpretować jako sygnał wyjścia. Domyślnie jest to SIGTERM, jak w przypadku każdego innego procesu Linuksa.
- VOLUME — definiuje woluminy, które będą montowane z hosta.

- **USER** — zmienia użytkownika (kontener), który od tego momentu będzie wykorzystywany do poleceń kompilacji (może być stosowana wielokrotnie do przełączania użytkowników podczas kompilacji).
- **ONBUILD** — definiuje instrukcję, która jest inicjowana, gdy ten obraz jest używany jako podstawa dla innej kompilacji.
- **HEALTHCHECK** — funkcje sprawdzania kondycji, które prawdopodobnie nie będą używane, ponieważ harmonogram kontenera ma własne funkcje sprawdzania kondycji.

Przejdziemy do praktycznego, kompleksowego przykładu, jak powiązać to wszystko z małym projektem, ale najpierw wróćmy do poleceń, które właśnie zastosowaliśmy, i przyjrzyjmy im się bliżej.

Polecenia kontenera

Wyjaśnimy teraz niektóre z bardziej skomplikowanych, ale ważnych poleceń oraz wywołań poleceń, które można napotkać podczas pracy z Dockerem.

docker run

Przyjrzyjmy się bardziej złożonemu wywołaniu polecenia `docker run`, którego użyliśmy wcześniej:

```
docker run --rm --name mywebcontainer -p 80:80 -v  
/tmp:/usr/share/nginx/html:ro -d nginx
```

- **--rm** — czyści (usuwa) dany kontener po zakończeniu jego działania.
- **--name mywebcontainer** — nadaje temu kontenerowi przyjazną nazwę: `mywebcontainer`.
- **-p 80:80** — mapuje port 80 hosta na port 80 w kontenerze. Numer portu po lewej stronie znajduje się na „zewnątrz” (w środowisku, w którym działa kontener), a numer portu po prawej reprezentuje port „wewnętrzny” (kontenera). Na przykład `-p 4000:80` zmapuje port kontenera 80 na `localhost:4000`.
- **-v /tmp:/usr/share/nginx/html:ro** — montuje wolumin — katalog `/tmp` środowiska hosta zostanie zamontowany w kontenerze pod adresem `/usr/share/nginx/html`; `:ro` sprawia, że będzie to montowanie tylko do odczytu (zamontowane pliki nie mogą być modyfikowane z kontenera).
- **-d** — uruchamia kontener w trybie odłączonym (działanie w tle).
- **nginx** — obraz, na którego podstawie ma zostać uruchomiony kontener.

Jeśli chcesz zobaczyć jakiś kod w HTML pod adresem `http://localhost:80`, możesz dodać plik `index.html` do katalogu `/tmp`:

```
cat <<EOF > /tmp/index.html  
<!doctype html>  
<h1>Hello World</h1>
```

```
<p>To jest mój kontener</p>
</html>
EOF
```

Ponieważ katalog `/tmp` jest mapowany na katalog `/usr/share/nginx/html` kontenera (gdzie `nginx` będzie szukał plików HTML), `nginx` natychmiast rozpozna i rozpocznie serwowanie tego pliku.

Woluminy są mechanizmem, za pomocą którego aplikacje stanowe mogą być nadal uruchamiane przy użyciu kontenerów bezstanowych.

docker image list

Aby zobaczyć, które obrazy zostały pobrane lokalnie, możesz uruchomić `run docker images` (lub `docker image list`, jeśli wolisz).

Jeśli tworzysz i wykorzystujesz wiele obrazów Dockera, ta lista może być długa!

```
$ docker image list
```

REPOSITORY			TAG
IMAGE ID	CREATED	SIZE	
nginx			latest
51086ed63d8c	10 days ago	142MB	
vault			latest
22fdc6314051	2 months ago	207MB	
golang			1.19-alpine
d0f5238dcb8b	2 months ago	352MB	

docker ps

Polecenie `docker ps` przypomina nieco polecenie `ps` z systemu Linux. Pozwala zobaczyć, które kontenery są uruchomione na hoście, wraz z pewnym kontekstem, takim jak ich identyfikator, polecenie, które uruchamiają, czas utworzenia i czas pracy, mapowanie portów itd.

Uruchom poniższe polecenie:

```
$ docker ps
```

Wygeneruje ono następujące dane wyjściowe:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
2aca849eef73	nginx	"/docker-entrypoint..."	About a minute ago
Up About a minute	0.0.0.0:80->80/tcp	mywebcontainer	

docker exec

Podczas tworzenia obrazu kontenera często zdarza się „wskakiwać” do kontenera i uruchamiać polecenia. Aby uruchomić interaktywną powłokę w uruchomionym kontenerze, zastosuj polecenie `docker exec`:

```
docker exec -it mywebcontainer /bin/bash
```

W przypadku naszego wcześniej uruchomionego kontenera `nginx` spowoduje to uruchomienie powłoki `Bash` w środowisku kontenera. Wszelkie wprowadzone zmiany stanu (tworzenie plików, ustawianie jądra itp.) zostaną utracone po zatrzymaniu kontenera — następne uruchomienie polecenia `docker run` uruchomi nowy kontener z tego samego stanu obrazu bazowego.

docker stop

Aby zatrzymać kontener, uruchom `docker stop $NAZWA_KONTENERA` — możesz również użyć identyfikatora kontenera, jeśli nie ma on przyjaznej nazwy:

```
docker stop mywebcontainer
```

Jeżeli kontener został uruchomiony z opcją `--rm`, tak jak kontener `nginx`, zostanie on usunięty, a jego stan (jeśli różnił się od obrazu bazowego) zostanie utracony.

Jeśli kontener nie został uruchomiony za pomocą `--rm`, jego stan pozostanie w systemie plików i będzie można uruchomić go ponownie za pomocą polecenia `docker start $NAZWA_KONTENERA`. Jego stan zostanie zachowany.

Projekt Dockera: kontener aplikacji utworzonej w języku Python i frameworku Flask

Skonteneryzujemy małą usługę sieciową napisaną w Pythonie, która korzysta z frameworku `Flask`. Jest to niezwykle powszechny wzorzec, a Python dobrze nadaje się do konteneryzacji, ponieważ pakowanie i zarządzanie zależnościami jest w wielu projektach tworzonych w Pythonie niechlujne. Wszystkie pliki będziesz tworzyć samodzielnie — spróbuj użyć edytora tekstu wiersza poleceń, aby poćwiczyć!

1. Konfigurowanie aplikacji

Najpierw utwórz nowy katalog i wejdź do niego:

```
mkdir dockerpy && cd dockerpy
```

Utwórz niewielką aplikację internetową w języku Python. W tym przykładzie używamy `vima`, ale możesz korzystać z dowolnego edytora:

```
vim echo_server.py
```

Wklej do pliku następujący kod:

```
from flask import Flask, request
import os

app = Flask(__name__)

@app.route('/')
def echo():
    return {
        "method": request.method,
        "headers": dict(request.headers),
        "args": request.args
    }

@app.route('/health')
def health():
    return {"status": "healthy"}

if __name__ == "__main__":
    env_port = os.environ.get("PORT", 8080)
    app.run(host='0.0.0.0', port=env_port)
```

To cała aplikacja internetowa — odczytuje niektóre informacje z przychodzącego żądania i wykorzystuje je do odesłania odpowiedzi klientowi.

Zapisz i zamknij plik (*esc*, *:x*).

Utwórz plik o nazwie *requirements.txt*, zawierający tylko następującą linię:

```
Flask>=3.0.0
```

Następnie utwórz plik *Dockerfile*:

```
vim Dockerfile
```

Wprowadź następujący kod:

```
# Użycie oficjalnego obrazu bazowego Pythona
FROM python:3.12-slim

# Ustawienie katalogu roboczego w kontenerze
WORKDIR /app

# Kopiowanie lisy zależności do kontenera
COPY requirements.txt .

# Instalowanie zależności Pythona
RUN pip install --no-cache-dir -r requirements.txt

# Kopiowanie skryptu do kontenera
COPY echo_server.py .

# Ustawienie kontroli kondycji, aby wyłączyła kontener, jeśli nie nasłuchuje
# na wewnętrznym porcie
HEALTHCHECK --interval=30s --timeout=5s \
  CMD curl --fail http://localhost:8080/health || exit 1
```

```
# Udostępnianie potu dla aplikacji  
EXPOSE 8080
```

```
ENV PORT=8080  
CMD ["python", "echo_server.py"]
```

Na razie to wszystko, czego potrzebujesz. Katalog *dockerpy* powinien teraz zawierać trzy pliki:

- Dockerfile,
- *echo_server.py*,
- *requirements.txt*.

2. Tworzenie obrazu Dockera

Skompiluj nowy obraz Dockera za pomocą polecenia `docker build command`. Opcja `-t` służy do otagowania kontenera nazwą:

```
docker build -t dockerpy .
```

Zwróć uwagę na znak `.` na końcu, który informuje Dockera, aby używał bieżącego katalogu jako kontekstu kompilacji.

3. Uruchomienie kontenera z obrazu

Polecenie `docker run` zostało już użyte wcześniej w tym rozdziale. Zastosuj je, aby uruchomić kontener z nowo utworzonego obrazu:

```
docker run --rm -d -p 8080:8080 --name my-dockerpy dockerpy  
(to polecenie wypisze ID Twojego nowego kontenera)
```

Jest tu kilka nowych argumentów:

- `--rm` — nakazuje Dockerowi usunięcie kontenera po jego zamknięciu. Zapobiega to pozostawianiu starych kontenerów w systemie plików, co można było zaobserwować w pierwszych przykładach w tym rozdziale.
- `-d` — nakazuje Dockerowi demonizację kontenera. Zapobiega to dołączaniu go do terminala na pierwszym planie.
- `-p` — ustawia mapowanie portów: lewa strona dwukropka to port kontenera, natomiast prawa strona to port hosta, na który zostanie zmapowany. Jeśli aplikacja kontenera działałaby na porcie 1234 i chcielibyśmy, aby mapowała się na port hosta 80, wyglądałoby to tak: `-p 1234:80`.
- `--name` — taguje kontener nazwą, dzięki czemu można go łatwo znaleźć w danych wyjściowych z polecenia `docker ps`.

Teraz masz już uruchomioną aplikację w kontenerze i możesz uzyskać do niej dostęp w przeglądarce lub w wierszu poleceń. Użyjmy polecenia `curl`, aby połączyć się i wysłać żądanie do usługi internetowej:

```
curl localhost:8080
{"args": {}, "headers": {"Accept": "*/*", "Host": "localhost:8080",
"User-Agent": "curl/8.1.2"}, "method": "GET"}
```

Dla tych, którzy cierpieli z powodu niemożliwych do odtworzenia koszmarów zależności (z czego słyną m.in. Python i Ruby), powinno to być jak objawienie. Cała złożoność, którą wcześniej trzeba było przeciągać wraz z aplikacją — od lokalnych środowisk programistycznych, przez CI i testy oraz środowiska przejściowe, po środowiska produkcyjne — jest teraz skondensowana w jednym artefakcie, który będzie miał tę samą wartość bez względu na to, gdzie go uruchomisz.

Jednym z poleceń, którego nie używaliśmy wcześniej, jest `docker exec`. Pozwala ono wykonać polecenie w uruchomionym kontenerze. Jest to przydatne, jeśli z jakiegoś powodu absolutnie konieczne jest sprawdzenie lub zmodyfikowanie uruchomionego kontenera:

```
docker exec -it my-dockerpy /bin/sh
```

Polecenie to powoduje uruchomienie powłoki `/bin/sh` i dołączenie do niej w kontenerze (większość kontenerów produkcyjnych będzie miała tylko minimalną powłokę `/bin/sh` i nie będzie dostarczana z czymś tak w pełni funkcjonalnym jak `Bash`).

Zatrzymajmy serwer za pomocą ostatniego polecenia, które tutaj omówimy, czyli `docker kill`:

```
docker kill my-dockerpy
```

Powoduje ono wysłanie do procesu `SIGKILL` (sygnału 9) — w przeciwieństwie do `SIGTERM` (sygnał 15) — i zatrzymuje go natychmiast, nie dając mu szansy na płynne zamknięcie.

Porównanie kontenerów i maszyn wirtualnych

Znasz już przepływ pracy, którego będziesz używać do tworzenia obrazów Dockera i pracy z nimi. Warto jednak poznać podstawowe różnice między kontenerami a maszynami wirtualnymi. Wiedza ta może mieć znaczenie podczas rozwiązywania problemów operacyjnych, a ponadto zagadnienie to często pojawia się w rozmowie kwalifikacyjnej, aby ocenić rozumienie zasad leżących u podstaw konteneryzacji.

Maszyny wirtualne (ang. *Virtual Machine* — VM) umożliwiają uruchamianie kompletnych systemów operacyjnych, takich jak Linux, Windows lub DragonFly BSD, w innym systemie operacyjnym hosta. Maszyny wirtualne działają niezależnie od systemu hosta. Uruchomienie Dockera w systemie macOS będzie transparentnie wykorzystywać maszynę wirtualną do zapewnienia systemu operacyjnego Linux, który jest potrzebny Dockerowi.

W rezultacie maszyna wirtualna uruchamia pełny system operacyjny, na przykład Linux, który z kolei korzysta z systemu init, takiego jak `systemd`. Dzięki temu można zarządzać usługami i procesami dokładnie tak, jakby maszyna wirtualna była maszyną fizyczną.

Jeśli chodzi o codzienne użytkowanie, wszystko, co odnosi się do maszyny fizycznej, ma również zastosowanie do maszyn wirtualnych. Jednak nie w ten sposób zwykle używane są kontenery.

Kontenery Dockera zazwyczaj zawierają pojedyncze aplikacje, a często tylko jeden proces. Jeśli w kontenerze znajduje się wiele procesów, jest to przeważnie spowodowane aplikacją wieloprotocową, która uruchomiła procesy potomne (najczęściej robią to serwery internetowe lub programy uruchamiające polecenia). Ponieważ powszechnie przyjętą najlepszą praktyką jest, aby kontener uruchamiał tylko jeden proces i kończył działanie, gdy tylko ten proces się zakończy, jakiegokolwiek rodzaju wewnętrznego nadzoru i zarządzania procesami byłby tutaj marnotrawstwem.

Zamiast tego można zauważyć, że zadania zwykle wykonywane przez system init systemu operacyjnego zostają przeniesione poza środowisko uruchomieniowe kontenera, do zewnętrznych systemów **zarządzających** kontenerami, takich jak Kubernetes, Nomad itd.

W tym nowym modelu kontenery są tym, czym kiedyś były procesy systemu operacyjnego, a orkiestratory kontenerów odgrywają różne role systemu operacyjnego i harmonogramu.

W kontenerze Dockera PID1, który jest systemem init w pełnym systemie operacyjnym Linux, jest tym, czym jest CMD lub ENTRYPOINT. Zazwyczaj jest to główny proces uruchomionego oprogramowania. Z reguły oczekuje się, że kontener uruchomi jeden proces. Chociaż istnieją scenariusze, w których ludzie celowo uruchamiają swoje kontenery w inny sposób, uruchomienie jednego procesu i zatrzymanie kontenera po zatrzymaniu procesu jest nadal oczekiwanym zachowaniem. Szczególnie w przypadku zwykłej konteneryzacji usługi, która ma być uruchomiona w środowisku produkcyjnym, należy postępować zgodnie z tym podejściem. Istnieją wyjątki od tej reguły, zwłaszcza przy uruchamianiu oprogramowania sprzed upowszechnienia się kontenerów Dockera, ale w takich przypadkach prawdopodobnie będziesz mieć tego świadomość i często będziesz się opierać na utworzonych w tym celu kontenerach.

Krótką uwaga na temat repozytoriów obrazów Dockera

W tym rozdziale sporo pracowaliśmy z obrazem nginx. Ale skąd dokładnie pochodzi ten obraz? Domyślnie Docker pobiera obrazy z Docker Hub (<https://hub.docker.com/>), który jest centralnym repozytorium publicznych obrazów Dockera. Docker Hub działa jak repozytorium pakietów Linuksa, które zawiera załadowane obrazy Docker gotowe do użycia. Można tam znaleźć większość popularnego oprogramowania serwerowego, które można pobierać i wykorzystywać tak łatwo, jak w przypadku nginx.

Nie wszystkie aplikacje są jednak publiczne i normalne jest używanie prywatnych repozytoriów do przechowywania obrazów Dockera. Istnieje stale zmieniająca się lista dostawców repozytoriów obrazów Dockera, więc nie będziemy ich tutaj wymieniać, ale wystarczy zrozumieć, że wszyscy działają tak samo jak Docker Hub.

Bolesne lekcje dotyczące kontenerów

Rozpoczynając tworzenie własnych kontenerów, można uniknąć wielu problemów, jeśli będzie się pamiętać o najlepszych praktykach omówionych w oficjalnej dokumentacji Dockera, którą możesz znaleźć na stronie https://docs.docker.com/get-started/09_image_best/.

W związku z tym przygotowaliśmy krótką listę najbardziej rażących błędów konteneryzacji, jakie napotkaliśmy w swojej pracy, oraz sposobów zapobiegania im. Ten podrozdział jest wynikiem wielu nieprzespanych nocy, przestojów serwerów i uczenia się na własnych błędach.

Rozmiar obrazu

Zacznij od minimalnych obrazów, takich jak Scratch lub Alpine. Aby wdrożyć większość aplikacji, dobrym pomysłem jest unikanie dużych obrazów i dystrybucji, takich jak Ubuntu. Gdy wymagane są zależności kompilacji, zaleca się ich usunięcie lub użycie pośrednich kontenerów kompilacji podczas kompilowania większych (wielokontenerowych) projektów.

Małe, minimalne obrazy nie tylko oznaczają szybsze pobieranie i mniejsze zużycie zasobów, ale także znacznie ułatwiają zarządzanie. Jeśli obraz nie zawiera oprogramowania i bibliotek, których nie potrzebujesz, oznacza to mniej aktualizacji, mniejszą powierzchnię do ataku dla hakerów oraz mniej hałaśliwych ostrzeżeń ze skanerów bezpieczeństwa kontenerów.

Standardowa biblioteka C

Należy pamiętać, z której **standardowej biblioteki C** (znanej również jako *libc*) korzystamy. Wiele dystrybucji Linuksa używa `glibc`, a niektóre, jak Alpine Linux, korzystają z `musl` lub innych. Biblioteki i wszelkie wynikowe pliki binarne mogą nie być między nimi kompatybilne. Na przykład w przypadku Alpine może być konieczne samodzielne skompilowanie mniej popularnych narzędzi. Jeśli Twoje projekty zależą od pewnych bibliotek dostępnych za pośrednictwem pakietów w obrazie bazowym, możesz napotkać niezgodności. Oczywiście aktualizacja, downgrading wersji lub całkowita zmiana obrazów bazowych może powodować podobne problemy.

Ponieważ jednak Alpine i `musl` stale zyskują na popularności, tego rodzaju problemy stają się coraz mniej prawdopodobne (a przynajmniej coraz bardziej możliwe jest ich wygooglowanie!). Jeśli nie masz zależności od żadnych bibliotek C, zwykle nie będzie to problemem. Ponadto statyczna kompilacja kodu może sprawić, że będziesz bardziej niezależny od systemu bazowego, a tym samym od obrazu bazowego.

Laptop nie jest środowiskiem produkcyjnym — zewnętrzne zależności

Nie polegaj na lokalnych montowaniach lub innych lokalnych kontenerach. Środowisko dla wdrożonego kontenera będzie prawdopodobnie bardzo różnić się od środowiska Twojego laptopa. To, że na laptopie masz kontener bazy danych obok kontenera aplikacji internetowej, nie oznacza, iż kontenery te będą zaplanowane na tych samych maszynach w środowisku produkcyjnym.

To samo dotyczy woluminów danych — te zewnętrzne w stosunku do kontenera punkty styku są miejscem, w którym trzeba przeprowadzić planowanie ze swoimi współpracownikami z zespołu Ops lub DevOps. Prawdopodobnie będziesz podpięty pod wykrywanie usług, sprawdzanie kondycji i współdzielone woluminy za pośrednictwem planisty lub innego narzędzia DevOps.

Teoria kontenerów: przestrzeń nazw

Jeśli zastanawiasz się, jak działa magia kontenerów, lub po prostu martwisz się, że pewnego dnia będziesz musiał rozwiązywać problemy w środowisku kontenerowym pod presją, warto zapoznać się z koncepcją przestrzeni nazw. Możesz pominąć tę sekcję, jeśli nie interesuje Cię, jak zbudowana jest warstwa abstrakcji kontenerów w systemie Linux.

Przestrzeń nazw to przeciążony termin, używany do wielu rzeczy w różnych niszach technologicznych. W kontekście kontenerów Linuksa ideę przestrzeni nazw najlepiej wyjaśnić za pośrednictwem narzędzia `chroot` (ang. *change root*), które oznacza zmianę ścieżki `roota`. `chroot` jest starym narzędziem dla uniksowych i uniksopodobnych systemów operacyjnych, które umożliwia użytkownikowi zmianę `roota` (ścieżki `/`) systemu plików.

Użycie tego narzędzia jest naprawdę bardzo proste: `chroot /jakaś/ścieżka` ustawi jako nowe `/` cokolwiek, co znajduje się w lokalizacji `/jakaś/ścieżka`. Oprócz umożliwienia instalatorom systemów operacyjnych zmiany na system, który jest aktualnie instalowany w celu uruchamiania poleceń, pozwala to również na podstawowe zastosowanie przestrzeni nazw. Różne programy i konfiguracje różnych dystrybucji Linuksa wykorzystują to w celu zwiększenia bezpieczeństwa, ponieważ użycie `chroot` zasadniczo wyklucza części systemu plików z aktualnie czytelnego zakresu — sprawia, że wszystko poza nowym `rootem` jest niedostępne. Tak więc jeśli atakujący użyje eksploita, który pozwala na zdalne wykonywanie kodu na serwerze WWW działającym w środowisku `chroot`, system i wszystkie pliki poza tym katalogiem pozostaną nienaruszone.

W ciągu ostatniej dekady techniczne podstawy wykorzystywane do implementacji kontenerów w Linuksie i innych systemach operacyjnych uległy znacznym zmianom i prawdopodobnie nadal będą się zmieniać. Na szczęście niskopoziomowa implementacja nie jest kluczowa dla Ciebie jako inżyniera oprogramowania, który jest głównie konsumentem konteneryzacji, w przeciwieństwie do operatora lub implementatora tej technologii.

Abstrakcja kontenera opiera się na kilku podstawowych technologiach:

- Zastosowanie przestrzeni nazw w systemie plików (na przykład za pomocą `chroot`).
- Przestrzenie nazw dla użytkowników i procesów, dzięki czemu procesy spoza kontenera są niewidoczne z jego wnętrza. Innymi słowy, `root` i `PID 5` w kontenerze byłyby odpowiednio mapowane na użytkownika z niskimi uprawnieniami i inny identyfikator procesu poza przestrzenią nazw kontenera.
- Technologie grupowania i uwzględniania zasobów, takie jak `cgroups`.
- Wirtualizacja sieci lub przestrzenie nazw, aby kontener nie mógł uzyskać bezpośredniego dostępu do interfejsu sieciowego, ale jednocześnie można było obsługiwać nakładanie się numerów portów. Na przykład można uruchomić dwa różne kontenery, które udostępniają port 8080; nie pojawi się błąd, że port jest już używany, ponieważ stosy sieciowe kontenerów są od siebie niezależne.

Jak przeprowadzać operacje na kontenerach?

Chociaż nie jest to książka przeznaczona dla administratorów systemów ani inżynierów niezawodności witryn, warto znać podstawowy kontekst, w którym uruchamiane są kontenery. Główną ideą jest to, że kontenery są w dużej mierze bezstanowymi „funkcjami”, które przetwarzają dane wejściowe (żądania internetowe lub komunikaty HTTP z innych usług) i generują dane wyjściowe (odpowiedzi internetowe, efekty uboczne i dzienniki przesyłane strumieniowo do `STDOUT`). W dobrze działającym środowisku operacyjnym kontenery można traktować jako analogię procesów w systemie Linux lub funkcji w programowaniu.

Kontenery są zwykle „rozplanowywane” na hostach przez zewnętrzną warstwę narzędziową, taką jak Kubernetes i Nomad. Jeśli kontenery są jak procesy, to pełnią one rolę harmonogramu systemu operacyjnego (całość jest systemem rozproszonym, a nie pojedynczym hostem).

Dane wyjściowe kontenera są zwykle przechwytywane przez to samo narzędzie i przekierowywane do rozwiązań rejestrowania, takich jak Logstash, Graylog i Datadog. Wskaźniki ze wszystkich uruchomionych kontenerów można wyodrębnić i wprowadzić do narzędzi takich jak Prometheus w celu analizowania i rozwiązywania problemów.

Podsumowanie

W tym rozdziale opisaliśmy najważniejsze zagadnienia, w których należy się orientować, pracując z Dockerem i kontenerami. Chociaż poszczególne technologie mogą się zmieniać (który planista kontenera jest w modzie lub jak najlepiej obsługiwać strumieniowanie dzienników), staraliśmy się skupić na podstawowej teorii i umiejętnościach, które powinien posiadać każdy nowoczesny programista.

Mamy nadzieję, że z tego rozdziału zapamiętasz kilka głównych koncepcji. Przede wszystkim spodziewamy się, że zaczniesz intuicyjnie rozumieć problemy, które rozwiązuje konteneryzacja, głównie przez kontrolowanie złożoności i pakowanie zależności w jeden artefakt.

Należy ponadto pamiętać o różnicy między obrazami a kontenerami i poćwiczyć tworzenie własnych plików Dockera od podstaw, korzystając z oficjalnej dokumentacji.

Znajomość kilku bardziej zaawansowanych tematów, takich jak różnice między maszynami wirtualnymi i kontenerami oraz działanie przestrzeni nazw, przyda Ci się podczas rozwiązywania problemów lub rozmowy kwalifikacyjnej. W tym przypadku przydadzą się również omówione przez nas najlepsze praktyki.

Na koniec, aby utrwalić zdobytą wiedzę, zalecamy przećwiczenie tych umiejętności poprzez konteneryzację jednej z własnych aplikacji. Wiele się nauczysz i znacznie łatwiej będzie Ci zacząć, gdy wszystkie informacje, które zawarliśmy w tym rozdziale, będą jeszcze świeże.

Monitorowanie dzienników aplikacji

Rozdział 16

Witamy w świecie rejestrowania w systemie Linux! Dla twórcy oprogramowania zrozumienie rejestrowania w Linuksie, zwłaszcza za pomocą narzędzi takich jak `systemd` i `journald`, jest kluczowe. Oto zestawienie tego, co musisz wiedzieć.

Dzienniki są zapisem zdarzeń zachodzących w aplikacji lub systemie operacyjnym. Jest to format elastyczny i unikatowy dla każdej aplikacji, ale w nowoczesnych systemach sposób przetwarzania, przechowywania i pobierania dzienników jest bardziej ujednolicony. Dla programisty zrozumienie dzienników jest niezbędne, ponieważ dzienniki, do których można uzyskać dostęp w systemie Linux, zapewniają wgląd w zachowanie systemu operacyjnego i wszystkich działających w nim aplikacji. Będziesz wykorzystywać tę wiedzę do zrozumienia błędów, śledzenia wydajności aplikacji i debugowania. Dzienniki są pierwszą linią obrony podczas rozwiązywania problemów, więc przygotuj się, by dobrze zrozumieć ich działanie.

Omówimy rejestrowanie w systemach Unix i Linux oraz pokażemy najczęstsze sposoby interakcji programistów z dziennikami.

W tym rozdziale omawiamy następujące zagadnienia:

- sposób emitowania dzienników przez system i działające w nim aplikacje;
- miejsce gromadzenia dzienników w większości nowoczesnych systemów linuksowych;
- trochę wiedzy historycznej na temat tego, jak rejestrowanie działało w przeszłości, co nadal przydaje się w wielu systemach produkcyjnych, z którymi będziesz mieć styczność;
- wyszukiwanie i przeglądanie dzienników podczas rozwiązywania problemów z aplikacją;
- sposób scentralizowania dzienników w firmach i sytuacje, w których usługi są wdrażane w środowiskach chmurowych.

Podamy również kilka wskazówek, jak w pełni wykorzystać rejestrowanie ustrukturyzowane, unikając typowych pułapek, w które często wpadają programiści.

Wprowadzenie do rejestrowania

Jak wspomnieliśmy we wstępie, **dzienniki** to komunikaty informacyjne — zapis (rejestr) zdarzeń zachodzących w aplikacji lub systemie operacyjnym. Podobnie jak w przypadku wielu pojęć uniksowych dla dzienników istnieje kilka rygorystycznych reguł: jeśli napiszesz dwuliniowy skrypt, który zapisuje znacznik czasu w pliku tekstowym, może to być liczone jako dziennik. Niektóre dzienniki są prostymi tekstowymi łańcuchami znaków wysyłanymi do dobrze znanych lokalizacji plików w systemie, a inne są wysoce ustrukturyzowanymi danymi binarnymi zarządzanymi wyłącznie przez demona, takiego jak `systemd`.

Jako programista prawdopodobnie znasz **poziomy dziennik**, które są etykietami wskazującymi pilność zdarzeń zachodzących w oprogramowaniu. Pomyśl o komunikatach `error`, `info` i `debug`, które z pewnością widziałeś, przewijając dane w terminalu podczas tworzenia oprogramowania. Omówimy te typowe poziomy dzienników dalej w tym rozdziale, ale na razie należy znać trzy główne **źródła** dzienników w nowoczesnym, w pełni funkcjonalnym środowisku Linuksa: **system**, **usługa** i **aplikacja** nieusługowa. Źródło dziennika może dostarczyć ważnych informacji kontekstowych na temat tego, co dzieje się w określonym komunikacie dziennika.

Dzienniki systemowe to dzienniki wysyłane przez sam system operacyjny (jądro). Obejmują one błędy oraz komunikaty o zdarzeniach sprzętowych, zużyciu zasobów i limitach, konfiguracji i bezpieczeństwie, a także godne uwagi zmiany stanu systemu.

Dzienniki usług są emitowane przez usługi działające w systemie. W Linuksie są one w szczególności emitowane przez usługi zarządzane przez system inicjujący `systemd`, który przeprowadza rejestrowanie za pośrednictwem usługi o nazwie `journald`. Dają one wgląd w kondycję i status różnych usług.

W systemach, które prawdopodobnie napotkasz, dzienniki systemowe i dzienniki usług są połączone w `journald`. Z tego rozdziału dowiesz się wszystkiego o `systemd` i `journald` (oraz `journalctl`).

Aplikacje niezarządzane przez systemd są wyjątkami, które zazwyczaj nie rejestrują za pośrednictwem `journald`. Trzeba wyszukiwać ich pliki dziennika na podstawie dokumentacji dla każdej aplikacji, chociaż dobrze zachowujące się aplikacje zwykle zapisują własne pliki dziennika w katalogu takim jak `/var/log/$NAZWA_APLIKACJI/`, gdzie `$NAZWA_APLIKACJI` jest nazwą aplikacji.

W miarę lektury tego rozdziału znaczenie zrozumienia poleceń `journald` i `journalctl` stanie się oczywiste. Zanim jednak do tego przejdziemy, powinniśmy zwrócić uwagę na kilka szczegółów dotyczących rejestrowania w Linuksie.

Rejestrowanie w Linuksie bywa... dziwne

Przekonałeś się już, że systemy uniksopodobne są niezwykle elastyczne. Jeśli nie podoba Ci się domyślny sposób, w jaki robione są różne rzeczy, możesz zerwać z konwencją i skonfigurować wszystko tak, aby działało, jak chcesz.

Jest to również ogromna wada podczas nauki podstaw systemów Unix i Linux. Wiele rzeczy — od konfiguracji oprogramowania po domyślne ustawienia użytkownika —

można zdefiniować na wiele różnych sposobów, a jedyną możliwością, aby dowiedzieć się, jaka konwencja jest stosowana w nowym środowisku, jest zadanie pytania (a czasem rozwiązywanie problemów).

W żadnej dziedzinie to stwierdzenie nie jest prawdziwsze niż w przypadku rejestrowania, które zostało szczególnie dotknięte ostatnimi zmianami w sposobie przetwarzania danych przez firmy. Rejestrowanie było przeprowadzane w określony sposób przez dziesięciolecia, kiedy większość firm bezpośrednio kupowała i konfigurowała długotrwałe serwery fizyczne z zainstalowanym na nich jednym systemem operacyjnym i zarządzała nimi. Wraz z przeniesieniem obciążeń roboczych do chmury i przejściem na wiele systemów operacyjnych przypadających na jedną maszynę fizyczną (maszyny wirtualne), a nawet na wiele środowisk na jeden system operacyjny (kontenery), zmieniły się również tradycyjne koncepcje działania rejestrowania.

Mamy na myśli to, że jeśli chodzi o rejestrowanie dzienników w nowej pracy lub nowym zespole, pytanie nie brzmi: „Jak rejestruje się w Linuksie?”, ale raczej: „Jak obecnie rejestruje się tutaj?”. To naprawdę zależy od decyzji podjętych przez twórców oprogramowania, z którego korzystasz. Warto o tym pamiętać podczas lektury tego rozdziału.

Wysyłanie komunikatów dziennika

Podczas gdy w większości sytuacji usługi rejestrują za pośrednictwem biblioteki lub po prostu zapisując w `stdout`, systemy uniksowe udostępniają polecenie, które rejestruje na serwerze `syslog`. Więcej na ten temat dowiesz się dalej w tym rozdziale. Ponieważ zarówno `syslogd`, jak i `systemd` zapewniają serwer `syslog` bez względu na rodzaj używanego systemu, istnieje zunifikowane polecenie wysyłania komunikatów dziennika:

logger Witaj, świecie!

To polecenie spowoduje zarejestrowanie *Witaj, świecie!*. Polecenie `logger` ma wiele opcji i może być cennym narzędziem podczas debugowania wszelkich problemów, gdy chcesz rejestrować w skrypcie powłoki, lub przy wyjaśnianiu, jak działa rejestrowanie.

journald usługi systemd

Gdy system używa `systemd`, rejestrowanie przejmuje `journald`. Podczas rozwiązywania problemów z maszyną z systemem Linux z uruchomionym `systemd` jest to pierwsze miejsce, w którym należy szukać dzienników. Domyślnie `journald` przechwytuje wszystkie dane wyjściowe z nadzorowanych procesów. Wszystko, co jest wyemitowane do strumienia `stderr`, jest traktowane jako błąd. Tak więc o ile oprogramowanie nie jest skonfigurowane lub zakodowane na stałe, aby rejestrować dzienniki w lokalizacji, którą nie jest `stderr` lub `stdout`, dzienniki znajdziesz w `journald` usługi `systemd`.

Dzienniki rejestrowane w `journald` mogą być sprawdzane za pomocą polecenia `journalctl`. Zapewnia ono środki do kwerendowania opartego na poszczególnych usługach, czasie i restartach systemu, a ponadto pozwala na użycie opcji podobnych do polecenia `tail`. Przejdźmy do rzeczy i poćwiczmy użycie polecenia `journalctl`.

Przykładowe polecenia journalctl

Podstawy pracy z narzędziem `journalctl` są proste. Wyobraź sobie, że rozwiązujesz problemy z aplikacją. Co musisz być w stanie zrobić z jej dziennikami?

Przed wszystkim należy znaleźć i wyświetlić bieżący zestaw dzienników. Zapewni Ci to polecenie `journalctl`, ale szybko zdasz sobie sprawę, że tak naprawdę nie chcesz *wszystkich* dzienników, lecz tylko te najnowsze. Przefiltrujmy więc dzienniki za pomocą flagi `-n`.

Aby zobaczyć w `journald` ostatnie 100 komunikatów dziennika, wykonaj to polecenie:

```
journalctl -n 100
```

Spowoduje ono wypisanie ostatnich 100 linii zarejestrowanych w systemie. Zauważysz, że przypomina polecenie `tail`, które wyjaśniliśmy wcześniej. Jeśli śledziłeś przykład, te linie będą prawdopodobnie zawierać powyższy komunikat *Witaj, świecie!*.

Śledzenie aktywnych dzienników dla jednostki

Możesz także przeglądać dzienniki w czasie rzeczywistym. Na przykład śledzenie dzienników aplikacji podczas uruchamiania może pomóc dokładnie zobaczyć, kiedy coś idzie nie tak:

```
journalctl -fu nazwa_jednostki
```

Flaga `-f` oznacza „podążanie” (ang. *follow*), a flaga `-u` oznacza „jednostkę” (ang. *unit*) — jednostkę systemową (lub usługę), dla której chcesz filtrować dzienniki.

Filtrowanie według czasu

Nawet po przefiltrowaniu do konkretnej jednostki, która Cię interesuje, nadal może Cię przytłaczać liczba znalezionych dzienników. Filtrowanie według czasu może być tutaj przydatne, zwłaszcza gdy chcesz przejrzeć dzienniki aplikacji pod kątem znanego problemu zewnętrznego (takiego jak awaria, błąd itp.) od momentu jego wystąpienia. W tym celu należy użyć `--since` (od) i `--until` (do):

```
journalctl --since "2021-01-01 00:00:00"
```

Możesz również użyć skrótu, takiego jak `today` (dzisiaj):

```
journalctl --until today
```

Opcję `--until` możesz także wykorzystać, aby ustawić czas zakończenia filtra i połączyć ją z inną opcją, by uzyskać dość konkretne dzienniki. Oto przykład:

```
journalctl --since "2021-01-01 00:00:00" --until "1 hour ago"
```

Uwaga

Jednym z zastrzeżeń dotyczących przeglądania i filtrowania dzienników według czasu jest to, że prawie zawsze będziesz chciał użyć dla polecenia `journalctl` opcji `--utc`, która wyświetla znaczniki czasu w UTC. Kiedy pomagasz zespołowi operacyjnemu w rozwiązywaniu problemów z awarią, odbywa się to niemal wyłącznie przy użyciu czasu UTC, aby uniknąć zamieszania związanego ze strefą czasową.

Istnieją dodatkowe filtry dla takich kwestii jak identyfikator użytkownika lub grupy.

Filtrowanie pod kątem określonego poziomu dziennika

Jeśli wiesz, że szukasz błędu, możesz sprawić, by polecenie `journalctl` pokazywało tylko błędy (lub dowolny z innych poziomów dziennika wymienionych na stronie https://wiki.archlinux.org/title/Systemd/Journal#Priority_level w kolejności malejącej krytyczności: `emerg`, `alert`, `crit`, `err`, `warning`, `notice`, `info`, `debug`):

```
journalctl -p err
```

Sprawdzanie dzienników z poprzedniego rozruchu

Czasami sprawy przybierają naprawdę szalony obrót, a błąd powoduje restart systemu. W takich przypadkach będziesz chciał zobaczyć dzienniki z poprzedniego uruchomienia systemu. Wszystkie dostępne bootowania możesz wyświetlić za pomocą polecenia `--list-boots` w następujący sposób:

```
journalctl --list-boots
```

Następnie wybierasz konkretny rozruch z listy za pomocą argumentu `-b`. W tym przypadku chcemy ten, który jest oznaczony jako 2:

```
journalctl -b -2
```

Sama flaga `-b` oznacza „bieżący rozruch systemu”.

Komunikaty jądra

We wstępie wspomnieliśmy, że komunikaty dziennika na poziomie systemu są wysyłane przez system operacyjny (w żargonie Linuksa „jądro”). Aby zobaczyć tylko te komunikaty, należy użyć flagi `--k` (lub ze względów historycznych `--dmesg`).

Rejestrowanie w kontenerach Dockera

W kontenerze Dockera najczęstszym sposobem traktowania dzienników jest założenie, że główny proces kontenera jest tym, z którego chcemy uzyskać dane wyjściowe, i że rejestruje on do standardowego wyjścia (`stdout`). Orkiestratory kontenerów (czyli narzędzia takie jak Kubernetes i Nomad), a także różne usługi w chmurze, które są odpowiedzialne za wykonywanie kontenerów, przyjmują, że `stdout` jest miejscem, do którego trafiają odpowiednie dzienniki, i stosownie je przekazują, w zależności od konfiguracji. Omówimy to nieco szerzej w podrozdziale „Rejestrowanie scentralizowane”.

Podstawy dziennika syslog

W porównaniu do rejestrowania za pomocą `systemd/journal`d, które omówiliśmy wcześniej, syslog może wydawać się nieco archaiczny. Ma on *bogatą historię* — chociaż istnieje od lat 80., nadal jest użytecznym, elastycznym i szeroko stosowanym narzędziem do rejestrowania. Co ważniejsze, prawie na pewno natkniesz się na niego w prawdziwych systemach produkcyjnych, więc warto znać podstawy, aby uniknąć zaskoczenia podczas awarii, w której czas ma kluczowe znaczenie.

W systemach uniksowych rejestrowanie w syslogu jest często równoważne rejestrowaniu w pliku `/var/log`, przy czym większość komunikatów trafia zazwyczaj do `/var/log/messages`. Należy jednak pamiętać, że nie wszystko, co znajduje się w `/var/log`, musiało przejść przez syslog. Różne elementy oprogramowania implementują również własny sposób zapisywania plików dziennika, całkowicie pomijając demona syslog.

Działa to w ten sposób, że syslog przyjmuje wszystkie wysyłane do niego dzienniki i w zależności od różnych parametrów, takich jak wymienione poniżej kategorie rejestrowania, zapisuje dane wyjściowe w pliku. W praktycznie wszystkich systemach domyślnie jest to katalog `/var/log/messages`. Jeśli śledziłeś przykłady i Twój system korzysta z syslogu, w tym miejscu znajdziesz również wcześniejszy komunikat *Witaj, świecie!*.

Syslog to znormalizowany protokół rejestrowania. Chociaż w chwili gdy piszemy ten rozdział, `syslogd` zajmuje się głównie liniami dziennika, obecny standard (RFC 5424) pozwala również na rejestrowanie ustrukturyzowane. Ponieważ jednak nie jest on powszechnie obsługiwany, omówimy tylko pokrótce jego podstawowe koncepcje jako protokołu rejestrowania opartego na rejestrowaniu linii i komunikatów. Jeśli w ogóle nie korzystasz z syslogu, możesz pominąć tę sekcję.

Jak wspomnieliśmy wcześniej, syslog jest protokołem. O ile jest najczęściej używany w oprogramowaniu takim jak bazy danych, które potrzebują rejestrować lokalnie, o tyle konfiguracje produkcyjne zwykle mają scentralizowany serwer rejestrowania, do którego wysyłane są dzienniki. Chociaż syslog jest tylko protokołem, różne oprogramowanie (np. PostgreSQL, nginx) może za jego pośrednictwem emitować dzienniki, a inne oprogramowanie związane z rejestrowaniem (m.in. Logstash, Loki, `syslogd` i `syslog-ng`) może pozyskiwać jego dzienniki. Przeważnie używa on portu 514 (UDP) lub portu 6514 (TCP).

Kategorie rejestrowania

Ponieważ syslog jest bardzo starym protokołem z lat 80., niektóre z jego koncepcji mogą wyglądać archaicznie. Dlatego do określania typu komunikatu dziennika wykorzystuje **predefiniowane kategorie rejestrowania** (ang. *facilities*). Każda z tych kategorii ma swój kod:

- 0: kern — komunikaty jądra.
- 1: user — komunikaty na poziomie użytkownika. Są często wykorzystywane przez procesy.

- 2: mail — system mailowy. Przydatny dla serwerów pocztowych, SMTP, IMAP i POP3. Rejestrowane są tu zazwyczaj demony i oprogramowanie związane ze spamem.
- 3: daemon — demony systemowe. Rejestrowane są tutaj demony, zwłaszcza związane z systemem operacyjnym (takie jak te dla NTP).
- 4: auth — komunikaty bezpieczeństwa (uwierzytelniania). Zazwyczaj można tu znaleźć próby logowania, na przykład lokalnie, przez SSH, ale także do różnych innych usług.
- 5: syslog — komunikaty generowane wewnętrznie przez syslogd. Są to komunikaty związane z samym syslogiem.
- 6: lpr — podsystem wypisywania liniowego. Dzienniki związane z wypisywaniem.
- 7: news — podsystem komunikatów sieciowych. Jest to rozwiązanie historyczne i zwykle nie jest już używane.
- 8: uucp — podsystem UUCP. Jest to rozwiązanie historyczne i zwykle nie jest już używane.
- 9: cron — podsystem crona. Dzienniki związane z zadaniami crona. Mogą być bardzo przydatne do debugowania jego zadań.
- 10: authpriv — komunikaty bezpieczeństwa (uwierzytelniania). Są podobne do autoryzacji, ale zwykle są rejestrowane w bardziej ograniczonym zestawie miejsc docelowych. Większość oprogramowania Linuksa rejestruje się tutaj zamiast w auth.
- 11: ftp — demon FTP. Komunikaty raczej historyczne, dzienniki dla serwerów FTP.
- 12: ntp — podsystem NTP. Dzienniki dla protokołu **NTP** (ang. *Network Time Protocol*), czyli synchronizacji zegara.
- 13: security — audyt dziennika. Zdarzenia związane z bezpieczeństwem.
- 14: console — alert dziennika. Komunikaty związane z „konsolą lokalną”.
- 15: solaris-cron — demon zegara.
- 16 – 23: od local0 do local7 — komunikaty wykorzystywane lokalnie, co oznacza lokalne oprogramowanie. Na przykład PostgreSQL podczas rejestrowania w syslogu domyślnie zapisuje w local0.

W wielu systemach w `/var/log/` można znaleźć pliki, których nazwy przypominają te komunikaty. Tak więc jeśli chcesz debugować na przykład zadanie crona w systemie, który nie korzysta z journald, prawdopodobnie dane wyjściowe znajdziesz w `/var/log/cron`, `/var/cron/log`, `/var/log/messages` lub podobnej lokalizacji.

Należy pamiętać, że to, co dokładnie jest rejestrowane w każdym komunikacie, nie jest znormalizowane. Prawdopodobnie napotkasz sytuacje, w których różne systemy operacyjne lub podobne oprogramowanie mogą różnić się co do tego, w którym komunikacie rejestrować określone zdarzenia.

Poziomy dotkliwości

Jest to koncepcja, którą prawdopodobnie dobrze znasz. Komunikaty mają jeden z ośmiu różnych poziomów dotkliwości:

- 0: emerg — nagły wypadek,
- 1: alert — alert,
- 2: crit — sytuacja krytyczna,
- 3: err — błąd,
- 4: warning — ostrzeżenie,
- 5: notice — powiadomienie,
- 6: info — komunikat informacyjny,
- 7: debug — debugowanie.

Podobnie jak kategorie rejestrowania poziomy dotkliwości zależą od oprogramowania.

Konfiguracja i implementacje

Syslog ma wiele implementacji. Zazwyczaj umożliwiają one konfigurowanie filtrowania oraz zapisywanie i przekazywanie komunikatów na podstawie kategorii rejestrowania i poziomów dotkliwości. Niektóre systemy są dostarczane z usługą o nazwie `syslogd`, `rsyslog` lub `syslog-ng`, którą można skonfigurować w `/etc/syslog.conf` lub `/etc/syslogng/`. Loki, Logstash i inne rozproszone narzędzia do zarządzania dziennikami mają swoje własne sposoby konfigurowania rejestrowania, zwykle w trójelementowej strukturze, z jednym miejscem do definiowania danych wejściowych, drugim do filtrowania i przekształcania, a trzecim do przechowywania lub przekazywania danych wyjściowych.

Wskazówki dotyczące rejestrowania

Każdy rejestruje nieco inaczej, a to, co jest uważane za najlepsze praktyki, może się różnić w zależności od projektów i czasu. Jest jednak kilka rzeczy, o których należy pamiętać.

Słowa kluczowe podczas korzystania z logowania ustrukturyzowanego

Korzystając z dowolnego rodzaju ustrukturyzowanego rejestrowania, staraj się udostępniać wspólne słowa kluczowe, takie jak żądania i identyfikatory użytkowników, jednocześnie unikając sprzecznych słów kluczowych używanych do podobnych, ale nie takich samych rzeczy. W zależności od bazy danych można również napotkać problemy z typami, na przykład w sytuacji, gdy `user` może być kluczem dla liczby całkowitej, łańcucha znaków lub własnej zagnieżdżonej struktury, takiej jak obiekt JSON.

Czasami możliwe jest uniknięcie nakładania się przez tworzenie przestrzeni nazw dla poszczególnych usług i utrzymywanie listy kluczy „stosowanych globalnie” wraz z ich definicjami.

Poziomy dotkliwości

Podczas tworzenia oprogramowania warto posiadać wewnętrzny dokument wyjaśniający, jakie znaczenie mają poszczególne poziomy dotkliwości. Pozwala to uniknąć sytuacji, w których nieudane próby logowania do publicznie dostępnej usługi lub kody błędów 404 z robotów internetowych żądających przestarzałych stron internetowych będą podnosić alarm i budzić kolegę z pracy w środku nocy. Ale nawet jeśli do tego nie dojdzie, taki dokument może znacznie ułatwiać debugowanie i uświadamiać skalę problemów.

Z tego powodu dobrym pomysłem jest wyraźne rozróżnienie między kilkoma kwestiami. Oto one:

- sytuacje, które *mogą* wskazywać na problem;
- sytuacje, które nie powinny się zdarzyć, ale *mogą się zdarzyć*;
- sytuacje, które wyraźnie wskazują na błąd lub poważniejszy problem.

Dzienniki z reguły stają się bardziej złożone w miarę rozwoju oprogramowania i dodawania usług, więc od samego początku warto poświęcić czas na określenie, co należy rejestrować i kiedy.

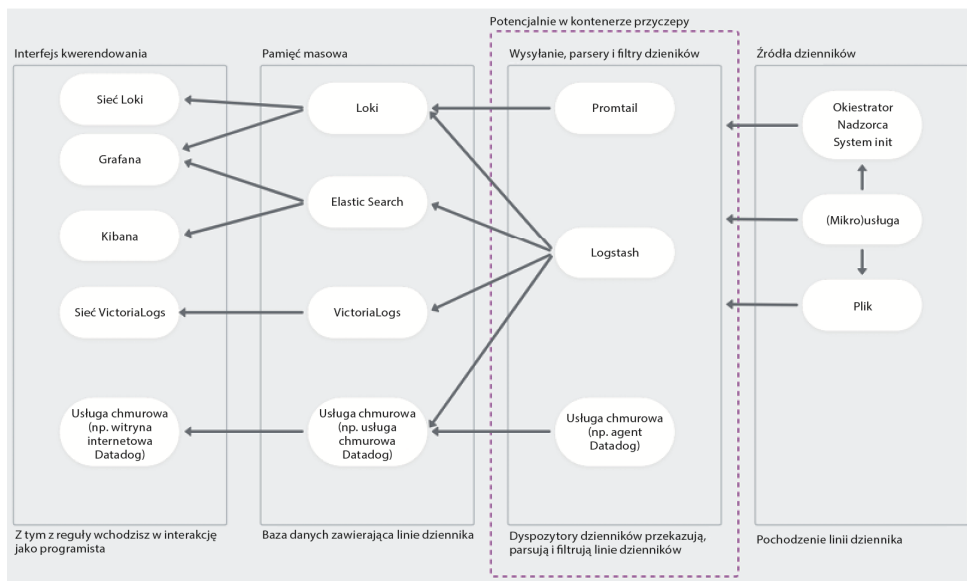
Rejestrowanie scentralizowane

W środowisku korporacyjnym typowe jest centralizowanie dzienników. Ułatwia to łączenie kropek podczas debugowania problemów. Oznacza również, że w aplikacji rozproszonej nie każdy dziennik na każdej maszynie fizycznej bądź wirtualnej lub w kontenerze musi być sprawdzany indywidualnie. Te scentralizowane usługi rejestrowania zwykle ułatwiają i przyspieszają wyszukiwanie dużych liczb dzienników, zwłaszcza gdy firma korzysta z rejestrowania ustrukturyzowanego, a usługi są zgodne z ujednoczoną strukturą dziennika.

Te usługi rejestrowania są ich własnymi produktami, takimi jak `rsyslog`, `Loki`, `stos ELK` (Elastic Search, Logstash i Kibana) oraz `Graylog`, albo usługami zarządzanymi. Mogą to być na przykład hostowane warianty usług, o których właśnie wspomnieliśmy, albo rozwiązania do rejestrowania w chmurze, takie jak pakiet operacyjny Google’a (wcześniej znany jako `Stackdriver`), `AWS CloudWatch` lub `Azure Monitor`. Istnieje wiele podobieństw między tymi systemami, ponieważ zapewniają mechanizmy „wysyłania” dzienników z plików lub za pośrednictwem jakiegoś rodzaju API, filtrowania i restrukturyzacji, aby w końcu zapisać je w ostatecznej pamięci masowej, gotowe do kwerendowania.

W architekturach mikrousług powszechne jest przekazywanie kontekstu, takiego jak identyfikator żądania, dzięki czemu żądanie wysłane przez klienta można łatwo prześledzić za pośrednictwem różnych usług, co jest niezbędne do debugowania architektur obejmujących wiele usług.

Jak wspomnieliśmy, większość tych systemów ma mechanizmy wysyłania dzienników do centralnego serwera lub klastra dzienników. Noszą one nazwy takie jak Logstash (L w stosie ELK) i Promtail (używane z Loki) i zazwyczaj zapewniają wiele sposobów pozyskiwania dzienników. Na przykład można je skonfigurować do utworzenia serwera HTTP, działania jako serwer syslog, korzystania z `journald`, odczytywania z innych usług wysyłania dzienników, korzystania z API w chmurze lub po prostu przycinania plików. Są one uruchamiane jako dodatkowe demony w systemach, jako kapsuły w Kubernetesie lub część konfiguracji Nomada. Ponieważ mają na celu umożliwienie centralizacji dzienników bez względu na to, jakie oprogramowanie jest używane, zwykle umożliwiają stosowanie różnych danych wejściowe dzienników i są z reguły bardzo elastyczne w kwestii konfiguracji — umożliwiają na przykład tworzenie hierarchii przez przekazywanie między poszczególnymi usługami wysyłania dzienników. W orkiestratorach kontenerów, takich jak Kubernetes i Nomad, jest to często realizowane za pomocą „przyczepy kontenera”, która działa obok kontenerów aplikacji i przechwytuje dzienniki ze wszystkich kontenerów w danej kapsule, alokacji lub węźle przed wysłaniem ich do miejsca docelowego (zobacz rysunek 16.1).



Rysunek 16.1. Orkiestracja kontenera z przyczepą

Jak widać, chociaż istnieje wiele technologii i produktów związanych z rejestrowaniem, wszystkie one należą do co najmniej jednej z opisanych kategorii, więc gdy scentralizujesz logowanie w swoim środowisku, powinno to dać Ci wyobrażenie o tym, jak poszczególne elementy ze sobą współpracują.

Podsumowanie

W nowoczesnych środowiskach produkcyjnych rejestrowanie może być ruchomym celem. Nauka zagadnień opisanych w tym rozdziale i eksperymentowanie z nimi powinny dać Ci solidne podstawy. Mamy nadzieję, że zapoznanie się z syslogiem i poleceniem `journalctl` zapewni Ci również elementarne zrozumienie i perspektywę historyczną, które ułatwią Ci pracę nad tym, jaki będzie rzeczywisty mechanizm przyszłego rozwiązania w postaci logowania jako usługi (ang. *logging as a service*).

Sądzymy, że umiejętności zdobyte podczas pracy z tym rozdziałem dadzą Ci praktyczną, wymierną przewagę, jeśli chodzi o projektowanie, debugowanie i optymalizację tworzonych i wdrażanych aplikacji. Jak zdążyłeś się przekonać, opanowanie podstaw narzędzia `journal` pozwala szybko diagnozować i wskazywać problemy, niezależnie od tego, czy są one związane z konkretną aplikacją, czy z jej szerszym kontekstem systemu Linux. Posiadanie wiedzy na temat alternatywnych i historycznych podejść do rejestrowania w systemie Linux pomoże Ci w rozwiązywaniu problemów z systemami (lub ludźmi), które nie były aktualizowane od dłuższego czasu.

Nie chodzi tylko o rozwiązywanie problemów, ale także o ułatwianie życia i usprawnianie pracy. Ponadto jest to umiejętność, która pozwoli Ci się wyróżnić. Krótko mówiąc, znajomość dzienników Linuksa sprawi, że będziesz mądrzejszym i bardziej efektywnym programistą.

Mechanizm równoważenia obciążenia i HTTP

W tym rozdziale przyjmiemy nieco inne podejście. Z jednej strony przedstawimy krótkie wprowadzenie do **protokołu HTTP** (ang. *hypertext transfer protocol*) i skoncentrujemy się przy tym na pewnych błędnych przekonaniach, w które popada wielu programistów aplikacji internetowych. Z drugiej strony postaramy się zachować pragmatyzm i omówimy jedno ze standardowych narzędzi HTTP, które jest dostępne w powłoce i oferuje naprawdę potężne możliwości — polecenie `curl`. W szczególności przedstawimy podstawy pracy z tym narzędziem w kontekście jego użycia do rozwiązywania najczęściej występujących problemów z aplikacjami internetowymi.

Przyjęliśmy założenie, że jeśli jesteś programistą aplikacji internetowych, to znasz już protokół HTTP. Zatem celem niniejszego rozdziału nie jest przedstawienie zupełnych podstaw tego protokołu, ale przypomnienie niektórych kwestii z nim związanych, jeżeli chcesz przypomnieć sobie nieco wiadomości na ten temat. Natomiast jeśli HTTP *jest* dla Ciebie nowością, w internecie znajdziesz wiele doskonałych źródeł wyjaśniających podstawy pracy z tym protokołem. Jako świetne źródło informacji na ten temat polecamy portal MDN firmy Mozilla.

W tym rozdziale skoncentrujemy się na najczęściej pojawiających się błędnych przekonaniach oraz pułapkach związanych z protokołem HTTP i sposobami jego rzeczywistego wykorzystania, które często mogą zaskakiwać. Te błędne przekonania z reguły wiążą się z faktem, że jako programista tworzysz aplikacje internetowe w bardzo prostym środowisku lokalnym, a następnie uruchamiasz je w skomplikowanych środowiskach produkcyjnych, które wyglądają zupełnie inaczej na przykład od laptopa, w którym te aplikacje zostały utworzone i przetestowane.

Różnica między tym, z czym aplikacja współpracuje w środowisku programistycznym znajdującym się w komputerze lokalnym programisty, a infrastrukturą, do której została wdrożona w środowisku roboczym bądź produkcyjnym, jest źródłem wielu nieporozumień oraz trudnych do wychwycenia błędów.

W niniejszym rozdziale przedstawimy najważniejsze z tych różnic. Zatem dowiesz się nieco na temat bramy, usługi upstream oraz innych koncepcji używanych podczas współpracy z warstwą infrastruktury w nowoczesnej witrynie bądź aplikacji internetowej. Następnie przejdziemy do wybranych spośród najczęściej popełnianych błędów związanych z protokołem HTTP, które prowadzą do powstawania trudnych do debugowania problemów dotyczących nagłówków, kodów stanu itd. Przedstawimy niektóre nowoczesne

funkcje zapewnienia bezpieczeństwa, które zostały dodane do HTTP, np. **CORS** (ang. *cross-origin resource sharing*), a także omówimy pokrótce historię protokołu HTTP i wersje, z którymi prawdopodobnie się spotkasz. Poza tym dowiesz się również nieco o mechanizmie równoważenia obciążenia i sposobie jego działania. Opanowanie tych podstaw pozwoli uniknąć przyjęcia niepoprawnego modelu ścieżki żądań klienta, co okazuje się częstym źródłem problemów projektowych w zakresie aplikacji i infrastruktury.

W tym rozdziale omawiamy następujące zagadnienia:

- podstawowa terminologia, którą trzeba poznać, aby można było zrozumieć kwestie związane z bardziej skomplikowaną infrastrukturą sieciową, jaka zostanie przedstawiona w dalszej części rozdziału;
- najczęściej pojawiające się błędne przekonania dotyczące kodów stanu HTTP, które jeśli zostaną dobrze zrozumiane, mogą pomóc w tworzeniu bardziej przejrzystego i poprawniej działającego kodu źródłowego przeznaczonego do obsługi kodów stanu;
- nagłówki HTTP i powiązane z nimi problemy, które możesz napotkać we własnych aplikacjach internetowych;
- różne wersje protokołu HTTP używane w rzeczywistych rozwiązaniach;
- sposoby działania mechanizmu równoważenia obciążenia i wyjaśnienie, dlaczego jako programista musisz go poznać, nawet jeśli nigdy nie zamierzasz zajmować się infrastrukturą aplikacji;
- sposoby rozwiązywania za pomocą działającego w powłoce polecenia `curl` problemów związanych z tymi wszystkimi zagadnieniami.

Jedynym wymaganiem podczas lektury tego rozdziału jest podstawowa znajomość sposobu działania żądań HTTP oraz narzędzi programistycznych istniejących dla aplikacji internetowych (np. trzeba wiedzieć, jak korzystać z konsoli w przeglądarce internetowej oraz z innych narzędzi programistycznych w celu debugowania prostych problemów związanych z HTTP).

Rozpocniemy od podstawowej terminologii, która okaże się przydatna, gdy przystąpimy do rozwiązywania problemów.

Podstawowa terminologia

W dalszej części rozdziału będziemy używać określeń, których możesz nie znać. Dlatego pokrótce je tutaj wyjaśnimy.

Brama

Obecnie brama (ang. *gateway*) to przeważnie proxy odwrotne HTTP lub mechanizm równoważenia obciążenia, a najczęściej połączenie obu tych technologii. To może być serwer HTTP, taki jak `nginx` lub `Apache`, fizyczny mechanizm równoważenia obciążenia w klasycznym znaczeniu albo działający w chmurze wariant tej samej koncepcji.

Może to być również **sieć CDN** (ang. *content delivery network*). Dlatego w przypadku otrzymania kodu stanu HTTP wskazującego na błąd związany z bramą będzie to jedno z tych urządzeń bramy bądź aplikacji, z którym się komunikujesz.

Upstream

Upstream to usługa otrzymująca dane od aplikacji. W większości przypadków będzie to rzeczywista aplikacja bądź usługa, np. utworzona przez Ciebie usługa HTTP. Warto pamiętać o możliwości łączenia kaskadowego lub warstwowego, więc między pierwszym proxy i aplikacją internetową może znajdować się jeszcze jakieś proxy pośrednie. Na przykład w wielu infrastrukturach chmury znajduje się mechanizm równoważenia obciążenia, który zajmuje się obsługą i filtrowaniem przychodzącego ruchu sieciowego (*ingress*). Dopiero za nim znajduje się mechanizm równoważenia obciążenia aplikacji, który faktycznie analizuje ruch sieciowy HTTP i przekazuje go do odpowiedniej puli serwera aplikacji.

Po przedstawieniu odrobiny technologii wykraczającej poza HTTP masz wiedzę, która będzie potrzebna w dalszej części rozdziału. W następnym podrozdziale nieco dokładniej omówimy kilka najczęściej błędnie rozumianych aspektów protokołu HTTP i pokażemy użycie polecenia `curl` w interfejsie wiersza poleceń do rozwiązywania najpowszechniejszych problemów.

Najczęściej pojawiające się błędne przekonania na temat protokołu HTTP

W trakcie opracowywania aplikacji internetowych i API HTTP dobrze jest wiedzieć o kilku szczegółach, które są pomijane przez wielu programistów. Przedstawimy kluczowe obszary, w których ta dodatkowa wiedza może przynieść wiele korzyści pod względem niezawodności tworzonych aplikacji. Zaprezentowane w tym rozdziale umiejętności w zakresie użycia polecenia `curl` pozwolą Ci rozpocząć debugowanie z poziomu powłoki dość niejednoznacznych problemów w stylu „witryna nie działa”.

Kody stanu HTTP

W kolejnych podpunktach omówimy wybrane kody stanu HTTP, które będziesz napotykać w codziennej pracy. Przedstawimy także związane z tymi kodami ważne informacje i mity.

Nie należy sprawdzać jedynie pod kątem kodu stanu 200 OK

Powszechnym sposobem sprawdzania pod kątem błędów jest jedynie ustalenie, czy zwrócony kod stanu żądania ma wartość 200 lub inną w zakresie 2xx, ponieważ to pozwala stwierdzić, czy wykonanie żądania zakończyło się pomyślnie. Jednak istnieją jeszcze pewne kwestie, o których warto pamiętać podczas wykonywania tego rodzaju operacji.

Zakres 2xx (obejmujący kody stanu z przedziału od 200 do 299) ma wskazywać na powodzenie operacji. Wiele API zwraca kod stanu *204 Brak treści* w celu wskazania, że operacja zakończyła się pomyślnie. Najczęściej ma to miejsce w sytuacjach, gdy API zwykle zwraca zasób, który został utworzony bądź zmodyfikowany. Jednak w określonych scenariuszach, na przykład podczas wykonywania operacji DELETE lub gdy prowadziło to do marnowania zasobów, taki kod stanu nie jest zwracany.

Sprawdzenie, czy kod stanu odpowiedzi mieści się w zakresie 2xx, może wydawać się odpowiednim rozwiązaniem w przypadku niektórych aplikacji. Trzeba jednak pamiętać, że logika aplikacji w stylu „jeżeli kod stanu jest inny niż 200, to zarejestruj błąd” jest niepoprawna. Kod stanu z zakresu 1xx i 3xx również nie wskazuje błędu.

Rzadko można się spotkać z kodem stanu z zakresu 1xx, o ile nie jest on oczekiwany, ponieważ większość najczęściej występujących sytuacji wykorzystujących zakres 1xx wiąże się z zadaniami typu przełączenia do WebSockets, a to zdecydowanie nie jest błąd. Z kolei kod stanu z zakresu 3xx jest zwracany dość często i ma informować klienta o przekierowaniu. Wprawdzie to może wskazywać na konieczność podjęcia jakiejś akcji — prawdopodobnie uaktualnienia ścieżki dostępu do określonej treści, która została przeniesiona — ale to zdecydowanie nie zalicza się do kategorii błędu.

Jeden kod stanu z zakresu 3xx pojawiający się znacznie częściej w środowisku produkcyjnym niż w programistycznym to *304 Nie zmodyfikowano*. Można go łatwo przeoczyć na etapie prac programistycznych, może on również pojawiać się ze względu na zmiany w infrastrukturze lub uaktualnienia biblioteki, które usprawniają działanie bądź wprowadzają nowy sposób działania związany z buforowaniem. Ten kod stanu jest również używany, gdy klient, np. przeglądarka internetowa lub biblioteka HTTP, wykonuje żądanie z nagłówkiem *If-Not-Modified*, aby wykorzystać zalety wynikające z buforowania.

Z wymienionych powodów wydaje się rozsądne, aby jedynie kody stanu z zakresu 4xx uznawać za potencjalny błąd, zamiast przyjmować tylko kody stanu z zakresu 2xx za oznakę powodzenia operacji. Takie rozwiązanie nie przeszkadza zastosowaniu w aplikacji eleganckiej logiki: sprawdzenie, czy kod stanu jest większy lub równy 400, okazuje się tak samo zwarte jak sprawdzenie, czy kod stanu znajduje się w zakresie 2xx.

404 Nie znaleziono

Trzeba pamiętać o bardzo ważnej kwestii: kod stanu *Nie znaleziono* może oznaczać co innego w zależności od aplikacji. Ten kod może zostać zwrócony nie tylko przez serwer lub bramę, ale także przez aplikację. Może wskazywać, że trasa nie istnieje, choć równie dobrze może oznaczać, iż określony zasób (np. post lub komentarz) nie istnieje z konkretnego powodu (np. został usunięty).

Dlatego kod 404 jest często elementem normalnej odpowiedzi zwróconej przez całkowicie sprawną aplikację, która działa tak, jak została zaprojektowana. W niektórych sytuacjach od tego może również zależeć działanie klienta, np. sprawdzenie, czy coś nie istnieje, zanim konkretny zasób zostanie utworzony lub w trakcie wskazywania użytkownikowi, że dany zasób bądź jego nazwa są już zajęte.

Innymi słowy sam kod stanu 404 (bez żadnego kontekstu odnośnie do aplikacji i żądania) nie jest wystarczający do wskazania problemu. Jak już wyjaśniliśmy, na wielu poziomach i warstwach może nawet oznaczać powodzenie. Czasami można tego uniknąć

poprzez nieużywanie tego zakresu na warstwie aplikacji i sygnalizowanie w inny sposób różnych sytuacji typu „nie znaleziono”, np. poprzez zwracanie kodu stanu 200. Właściwe podejście w tym zakresie będzie zależało zarówno od aplikacji, jak i od zespołu bądź uzgodnionego standardu, a także stylu architekuralnego użytego w aplikacji.

502 Zła brama

Ten kod stanu oznacza, że brama nie potrafi zinterpretować danych otrzymanych z usługi upstream. Innymi słowy odpowiedź udzielona na żądanie przekazane przez bramę nie była kompletną i poprawną odpowiedzią HTTP. Zwykle wskazuje to na problem związany z usługą upstream.

503 Usługa niedostępna

Kod stanu 503 często oznacza, że usługa upstream jest nieosiągalna na porcie, do którego sprawdzenia została skonfigurowana brama. W praktyce to oznacza, że aplikacja internetowa mogła ulec awarii, nie nasłuchuje żądań HTTP, nasłuchuje żądań na niewłaściwym porcie, jest blokowana przez regułę zapory sieciowej lub nieprawidłową regułę trasy oraz mnóstwo innych powodów.

504 Limit czasu bramy

Gdy brama nawiązuje połączenie z usługą upstream, w pewnym momencie czas tego połączenia zostanie przekroczony. To bardzo ważny aspekt, ponieważ wiszący proces będzie zużywał zasoby po obu stronach: bramy i usługi. Zwykle takie przekroczenie czasu oczekiwania następuje tylko wtedy, gdy wystąpiło coś nieoczekiwanego albo jeśli nie zostały zapisane lub odczytane żadne bajty.

Jeżeli usługa upstream zajmuje się obsługą długo przetwarzanego żądania, co wiąże się na przykład z oczekiwaniem na wynik obliczeń, to jedną z możliwości jest wydłużenie czasu wykonywania żądania, zanim dojdzie do przekroczenia czasu oczekiwania. Jednak przeważnie zaleca się zastosowanie innego podejścia, na przykład zdefiniowanie punktu końcowego jako asynchronicznego lub wcześniejsze rozpoczęcie zapisywania bajtów (jak w przypadku strumieniowania danych).

Powód takiego podejścia jest prosty: dopóki nie dojdzie do przekroczenia czasu oczekiwania, zasoby będą używane, a strona wykonująca żądanie nie wie, czy aplikacja udzieli na nie odpowiedzi. Dlatego jeśli usługa sieciowa działa nieprawidłowo, to informacje na ten temat nie zostaną przekazane ani bramie, ani klientowi. To może prowadzić do sytuacji, w której błędnie funkcjonująca usługa będzie przez bramę uznawana za wciąż działającą, choć w takim przypadku brama powinna szybko wykryć źródło problemu i uruchomić inny egzemplarz usługi.

Wprowadzenie do polecenia curl — sprawdzanie kodu stanu odpowiedzi HTTP

Jeżeli chcesz poznać tylko jedno działające w powłoce narzędzie, które miałoby Ci pomagać w rozwiązywaniu problemów związanych z protokołem HTTP, będzie to polecenie curl. W miarę omawiania kolejnych obszarów protokołu HTTP, które warto nieco dokładniej

poznać, w tekście pojawią się sekcje zawierające przykłady praktycznych poleceń `curl` przeznaczonych do rozwiązywania najczęstszych problemów związanych z omówionymi aspektami HTTP.

Najprostszy przykład wywołania polecenia `curl` przedstawia się następująco:

```
curl https://tutoriallinux.com/
```

Jego działanie przypomina skopiowanie adresu URL i jego wklejenie w pasku adresu przeglądarki internetowej. Jednak w omawianym przypadku następuje wyeliminowanie konieczności użycia przeglądarki internetowej, a polecenie `curl` bezpośrednio w powłoce wyświetla udzieloną przez serwer WWW odpowiedź na żądanie. Nie ulega wątpliwości, że to nie jest najbardziej ekscytujący sposób przeglądania internetu. W następnym kroku debugowania warto więc zrobić coś znacznie bardziej użytecznego, czyli sprawdzić kod stanu odpowiedzi HTTP.

Za pomocą polecenia `curl` można sprawdzić, czy serwer WWW działa i udziela odpowiedzi na żądania:

```
curl -IsS https://tutoriallinux.com/ | head -n 1  
HTTP/2 200
```

Na podstawie wygenerowanych danych wyjściowych wiemy, że serwer WWW działa, a wykonanie trasy / spowodowało udzielenie odpowiedzi z kodem stanu 200. Podane zostały również informacje o wersji protokołu HTTP (tutaj jest to HTTP/2), do czego jeszcze powrócimy w dalszej części rozdziału. Polecenie `curl` wykonuje żądanie HEAD (opcja `-I` lub `--head`), a także wyłącza wyświetlanie postępu poczynionego przez polecenie `curl` w trakcie operacji i wygenerowanych komunikatów błędów (opcja `-s` lub `--silent`).

Jednak w przypadku problemów komunikaty błędów zostaną wyświetlone dzięki użyciu opcji `-S` (`--show-error`):

```
curl -IsS https://tutoriajsdkfksjdghfkjsdhdflinux.com/ | head -n 1  
curl: (6) Could not resolve host: tutoriajsdkfksjdghfkjsdhdflinux.com
```

Ponadto pozbywamy się większości nagłówków i sprawdzamy jedynie kod stanu, który znajduje się w pierwszym wierszu danych wyjściowych (`| head -n 1`).

Jednak w trakcie debugowania często chcemy widzieć nagłówki. Warto zapoznać się z kilkoma niedoskonałościami protokołu HTTP dotyczącymi nagłówków, a następnie użyć polecenia `curl` do ich przeanalizowania.

Nagłówki HTTP

Wielkość znaków w nazwach nagłówków ma znaczenie

Wielkość znaków w nazwach nagłówków HTTP ma znaczenie. Część oprogramowania wykorzystuje tę cechę nagłówków, w wyniku czego pewne bramy mogą modyfikować i „normalizować” te nagłówki. Na szczęście podczas tworzenia aplikacji internetowych programiści rzadko muszą pracować bezpośrednio z niezmodyfikowanymi wartościami nagłówków. Zamiast tego korzystają z bibliotek internetowych, które zapewniają abstrakcję większości złożoności i zajmują się wszelkimi związanymi z tym szczegółami.

Jednak mimo to należy się upewnić, że tak właśnie jest, oraz normalizować nagłówki, na przykład poprzez używanie małych znaków w nazwach nagłówków definiowanych przez aplikację. Dzięki temu można również zapobiegać sytuacjom, w których nagłówki odpowiedzi zostaną przypadkowo dodane wielokrotnie, ale z nazwami stosującymi różną wielkość znaków.

Nagłówki niestandardowe

Gdy na potrzeby aplikacji tworzysz niestandardowe nagłówki HTTP, zwróć uwagę na prefiks, który zdecydujesz się używać. Swego czasu powszechne było stosowanie prefiksu X- dla nagłówków niestandardowych, np. X-My-Header. Obecnie ta praktyka jest uznawana za błędną (zapoznaj się z dokumentem RFC 6648, w którym to wyjaśniono). Zamiast tego rozsądne będzie utworzenie własnego prefiksu, np. w postaci nazwy projektu, produktu, firmy bądź skrótu tych danych. W ten sposób można zapobiec sytuacji, w której nagłówek zostanie ponownie użyty przez innego programistę, który pomyli go z oficjalnym nagłówkiem standardu HTTP.

Wyświetlanie nagłówków HTTP za pomocą polecenia curl

Opcja -I użyta we wcześniejszym przykładzie polecenia curl jest użyteczna podczas przeglądania nagłówków odpowiedzi udzielonej na żądanie. Dane nagłówków mogą wskazać na problemy związane z buforowaniem, błędnym określeniem typu treści itd. W kolejnym przykładzie możesz zobaczyć, jakie informacje w nagłówkach przekazuje serwer tutoriallinux:

```
curl -I https://tutoriallinux.com/
```

```
HTTP/2 200
server: nginx/1.24.0
date: Sat, 21 Oct 2023 16:37:12 GMT
content-type: text/html; charset=UTF-8
vary: Accept-Encoding
x-powered-by: PHP/8.2.11
link: <https://tutoriallinux.com/wp-json/>; rel="https://api.w.org/"
strict-transport-security: max-age=31536000; includeSubdomains; preload
```

W tym momencie nie występują żadne problemy związane ze sprawdzanym serwerem. Mimo to wyświetlone nagłówki podsuwają kilka pomysłów na usprawnienie konfiguracji nginx: z perspektywy zapewnienia bezpieczeństwa ujawnianie nazw oprogramowania i numerów wersji jest zwykle kiepskim rozwiązaniem. Użytkownicy otrzymujący dane HTML lub JSON z tego serwera nie muszą wiedzieć, że jego działanie jest wspomagane przez PHP.

Wersje protokołu HTTP

W celu wyjaśnienia niektórych nowszych funkcjonalności HTTP warto pokrótce przedstawić historię tego protokołu. HTTP jest dostępny od bardzo dawna i odkąd się pojawił, wiele się zmieniło, zwłaszcza w ostatnich latach, gdy aplikacje internetowe zyskiwały na popularności. Podstawowe koncepcje i fundamenty protokołu pozostały niezmienione

od chwili jego opracowania. Jednak zmianie uległy pewne sztuczki, optymalizacje i sposób działania. Znajomość wersji protokołu może pomóc w debugowaniu lub uchronić przed problemami oraz zminimalizować niepotrzebne lub nieintuicyjne optymalizacje i obejścia problemów.

HTTP/0.9

Istnieje znikome prawdopodobieństwo, że jeszcze napotkasz tę wersję protokołu HTTP. To najstarsza wersja, jaką można sobie wyobrazić. Pozwala ona wykonywać żądania GET do serwera i otrzymywać *treść* żądania HTTP. Nie są wysyłane ani otrzymywane żadne nagłówki, dotyczy to również numeru wersji i kodu stanu.

HTTP/1.0 i HTTP/1.1

Protokół HTTP/1.0, a szczególnie HTTP/1.1, jest znacznie bliższy temu, co obecnie jest uznawane za HTTP. Podczas gdy w HTTP/1.0 dodano numer wersji i nagłówki, wydanie HTTP/1.1 utorowało drogę dla metod i ogromnej liczby rozszerzeń, zwykle w postaci nagłówków.

W HTTP/1.1 dodano również obsługę potokowania (jest używane domyślnie). To oznacza, że wiele żądań może być przekazywanych za pomocą tego samego połączenia TCP. Kolejnym powszechnie używanym dodatkiem jest nagłówek Host, umożliwiający przypisanie wielu nazw hostów temu samemu serwerowi lub adresowi IP.

Na przykład serwer WWW może być teraz skonfigurowany do udzielania odpowiedzi na żądania kierowane do <http://example.org/>, <http://www.example.org/>, <http://forum.example.org/> i <http://blog.example.org/>, nie wymagając przy tym oddzielnego adresu IP dla poszczególnych domen.

W tej wersji protokołu zapewniono również obsługę wielu rozszerzeń: buforowania, kompresji, różnych schematów uwierzytelniania, negocjacji treści, a nawet WebSockets. Wszystkie one są do dzisiaj powszechnie używane.

HTTP/2

Opublikowano wiele artykułów wychwalających zalety protokołu HTTP/2. Można go uznać za ogromny i jednocześnie kontrowersyjny krok w nowym kierunku dla HTTP. Podczas gdy HTTP/1.1 był protokołem tekstowym pozwalającym każdemu tworzyć pełne i poprawne żądania z poziomu powłoki bądź edytora testowego, HTTP/2 jest protokołem binarnym obsługującym również strumienie, czyli mechanizm przeznaczony do tworzenia lekkiego wariantu połączenia TCP.

Format binarny i kompresja nagłówków oznaczają, że do komunikacji z serwerem HTTP (lub klientem) są obecnie wymagane specjalne narzędzia. Jednak ogólne koncepcje pozostały takie same jak we wcześniejszych wersjach HTTP. Zatem tylko programista internetowy będzie w stanie zauważyć różnice w konkretnych sytuacjach.

Protokół HTTP/2 wprowadził wiele nowych funkcjonalności, przy czym sporo z nich jest rzadko używanych w aplikacjach internetowych przeznaczonych dla użytkowników i nawet mogą one nie być zaimplementowane w przeglądarkach internetowych.

Wprawdzie nie jest to oficjalny wymóg standardu, ale HTTP/2 w przeglądarkach internetowych jest zwykle ograniczony do obsługi jedynie protokołu HTTPS.

W większości sytuacji aplikacje internetowe będą czerpały korzyści z usprawnień takich jak używanie pojedynczego połączenia TCP z wieloma strumieniami, zwłaszcza gdy wiele żądań, np. dotyczących plików statycznych i technologii AJAX, jest wykonywanych równolegle. Z tego powodu pewne optymalizacje, np. arkusze tzw. *sprite'ów* bądź łączenie wielu plików w jeden, okazują się zbędne. Gdy niektóre z tych optymalizacji prowadzą do przekazywania zbędnych danych, ostateczny wynik ich zastosowania może nawet okazać się odwrotny do zamierzonego.

Część aplikacji opracowanych z myślą o HTTP/1.1 może wymagać zmian podczas uaktualniania do HTTP/2, ponieważ rozwiązania typu zachowanie aktywnych połączeń mogą mieć nieprzewidywalne efekty uboczne. Z tych i innych powodów dobrym rozwiązaniem będzie przetestowanie aplikacji internetowych przed ich skonwertowaniem do HTTP/2. Zdarzają się nawet sytuacje, w których przejście aplikacji do protokołu HTTP/2 powoduje wydłużenie czasu wczytywania stron lub zwiększenie poziomu użycia zasobów.

Dlatego dobrym pomysłem będzie przeprowadzanie rzeczywistych testów i monitorowanie w celu sprawdzenia różnic między protokołami HTTP. Ponieważ wiele korzyści, jakie zapewnia HTTP/2, wiąże się z faktycznym użyciem przeglądarki internetowej, zwykły test z poziomu powłoki może nie dostarczać takich samych wyników, jakie uzyska rzeczywisty użytkownik korzystający z aplikacji. Często popełnianym błędem jest na przykład nieuwzględnianie oferowanej przez HTTP/1.1 funkcjonalności potokowania.

Jednak w większości rzeczywistych sytuacji protokół HTTP/2 zaoferuje większe korzyści. W przypadku wewnętrznych API HTTP bądź mikrousług wiele firm decyduje się pozostać przy HTTP/1.1 lub użyć gRPC.

HTTP/3 i QUIC

Protokół HTTP/3 został opracowany na bazie HTTP/2 i przenosi jego koncepcje do opartego na UDP protokołu o nazwie QUIC (zamiast TCP używanego przez wszystkie pozostałe wydania HTTP).

Podobnie jak poprzednia wersja HTTP także HTTP/3 używa strumieni jako lekkiej alternatywy dla nawiązywania nowego połączenia TCP. Jednak inaczej niż w HTTP/2 nie odbywa się to poprzez inicjowanie strumieni w istniejącym połączeniu TCP, lecz raczej poprzez użycie QUIC, czyli protokołu opracowanego z myślą o obsłudze tego rodzaju strumieni.

W powszechnie spotykanych przypadkach użycia TCP zastosowanie protokołu QUIC wiąże się z wieloma korzyściami. Na przykład ponieważ jest to protokół oparty na UDP (ang. *user datagram protocol*) — który jest działającą na niższym poziomie i szybszą alternatywą TCP — QUIC zapobiega sytuacjom, w których całe połączenie zostaje wstrzymane z powodu niedostarczenia (jeszcze) jednego pakietu, nawet jeśli ten pakiet jest przeznaczony dla innego strumienia. Protokół QUIC został również zoptymalizowany pod kątem szybkiego nawiązywania połączenia początkowego, z uwzględnieniem szyfrowania TLS w celu zapewnienia bezpieczeństwa połączeniu między klientem i serwerem. Protokół QUIC został opracowany z myślą o rozszerzalności i obsłudze przyszłych wersji. Krótco po jego standaryzacji wiele takich rozszerzeń było gotowych do rozpoczęcia procesu standaryzacji.

Ponieważ protokół HTTP/3 bazuje na UDP i został opracowany z myślą o zapobieżeniu zjawisku określanemu jako **protocol ossification**, wiele tradycyjnych form węzłów pośrednich i bram stało się zbędnych.

Uwaga

Zjawisko *protocol ossification* zachodzi, gdy węzły pośrednie (lub cokolwiek innego, co współpracuje z protokołem) wymagają od protokołu utrzymania określonej postaci, a tym samym utrudnione jest kontynuowanie rozwoju i modyfikacji tego protokołu (np. poprzez dodawanie rozszerzeń).

Warto przekonać się, jak te podstawowe koncepcje HTTP wpisują się w większą infrastrukturę, używaną do uruchamiania większości Twoich aplikacji internetowych. W tego rodzaju architekturze rzadko będzie to pojedynczy serwer WWW i klient (taki jak przeglądarka internetowa lub polecenie `curl`) — zwykle mamy wiele warstw, na których odbywa się komunikacja HTTP, a proste problemy stają się złożone i trudne do rozwiązania.

Jak zwykle przedstawiliśmy najważniejsze koncepcje, które trzeba zrozumieć, a także kilka praktycznych odpowiedzi z zakresu rozwiązywania problemów w bardziej zaawansowanej infrastrukturze internetowej za pomocą polecenia `curl` powłoki.

Mechanizm równoważenia obciążenia

Mechanizm równoważenia obciążenia to rozwiązanie pozwalające rozłożyć obciążenie spowodowane przez usługę między wiele egzemplarzy tej usługi. Wprawdzie zdecydowanie nie ogranicza się to do HTTP i usług sieciowych, ale protokół HTTP to jeden z najczęściej pojawiających się obecnie kontekstów, w których działa mechanizm równoważenia obciążenia.

Trzeba koniecznie zrozumieć sposób działania mechanizmu równoważenia obciążenia dla aplikacji internetowych, ponieważ ma on wpływ na to, jak błędy i problemy pojawiają się w środowisku produkcyjnym. Na przykład w lokalnym środowisku programistycznym z reguły pracuje się z jednym klientem (jest to przeglądarka internetowa bądź inny konsument API) oraz z jednym serwerem (aplikacja internetowa lub opracowywana usługa). Natomiast w rzeczywistości często mamy wiele warstw serwerów znajdujących się między klientem i aplikacją, a każdy z nich prowadzi komunikację i przekazuje ruch sieciowy HTTP oraz potencjalnie wprowadza do przepływu pracy własne problemy lub błędy.

Materiał zamieszczony w tym podrozdziale ma pomóc w ogólnym zrozumieniu poszczególnych elementów aplikacji, które stają się częścią aplikacji postrzeganej jako całość, nawet jeśli nie są one częścią tworzonego przez Ciebie kodu źródłowego aplikacji.

Mechanizm równoważenia obciążenia HTTP jest zwykle otrzymywany poprzez umieszczenie warstwy infrastruktury przed aplikacją, aby zapewnić obsługę proxy HTTP. Z reguły to będzie jeden z następujących elementów:

1. Usługa bramy, taka jak serwer HTTP obsługujący aplikację internetową (np. `nginx` lub `Apache`).
2. Oddzielna usługa przeznaczona do tego celu (np. `HAProxy` lub `relayd`).

3. Usługa chmury (np. Load Balancer w GCP, ELB w AWS).
4. Sprzętowe urządzenie do równoważenia obciążenia.

Czasami inżynierowie decydują się na użycie usługi niestandardowej albo na rozwiązanie oparte na DNS, zwłaszcza w kontekście regionalnego mechanizmu równoważenia obciążenia, który jest często stosowany jako warstwa dodatkowa oprócz jednej z innych wymienionych wcześniej metod. Mechanizmy służące do koordynacji kontenerów i odkrywania usług również są zwykle dostarczane jako kolejne rozwiązania w zakresie równoważenia obciążenia.

Pojawienie się w rozwiązaniu mechanizmu równoważenia obciążenia wymaga zrozumienia kilku kolejnych koncepcji. Szczególnie dotyczy to mechanizmu równoważenia obciążenia faktycznie zajmującego się mapowaniem żądań pochodzących od klientów na serwery zapewniające działanie egzemplarzy pieczołowicie tworzonej przez Ciebie aplikacji internetowej.

Zarządzanie sesją i ciasteczkami staje się skomplikowane, ponieważ w przypadku długo działających sesji nie ma już gwarancji, że za każdym razem zostanie użyty ten sam serwer. Nawet awaria jednego serwera w puli aplikacji spowoduje problem — czy użytkownicy będą mogli w niezakłócony sposób kontynuować działanie, czy utracą dane? Czy jako inżynier zajmujący się rozwiązywaniem problemów z własną aplikacją internetową nie będziesz w stanie odtworzyć danego problemu, ponieważ występuje on jedynie w jednym z dziesiątek bądź setek serwerów?

Zrozumienie sposobu działania nowoczesnego mechanizmu równoważenia obciążenia ma istotne znaczenie dla uniknięcia błędnych projektów aplikacji lub trudności na etapie rozwiązywania problemów. W kolejnych punktach poznasz podstawowy model mentalny, który możesz wykorzystać, aby uniknąć wspomnianych problemów.

Sesje trwałe, ciasteczka i tworzenie deterministycznych skrótów

Podczas konfigurowania mechanizmu równoważenia obciążenia dla usług HTTP jedno z pierwszych pytań, na które trzeba sobie odpowiedzieć, dotyczy konieczności używania sesji trwałych (ang. *sticky sessions*). Takie sesje to rodzaj mechanizmu pozwalającego powiązać klienta z określonym serwerem aplikacji na czas trwania sesji. Często są one wymagane w przypadku aplikacji przechowujących informacje o stanie w serwerze aplikacji.

To jest jeden z powodów, dla których najlepszą praktyką jest opracowywanie aplikacji „bezstanowych”, w których przypadku informacje o stanie są zapisywane na współdzielonej warstwie danych. W takich aplikacjach nie ma znaczenia, czy pierwsze żądanie klienta zostanie obsłużone przez inny serwer niż drugie. Na szczęście w dzisiejszym świecie, zwłaszcza w przypadku korzystania z infrastruktury opartej na chmurze, sesje trwałe zwykle są niepotrzebne. Jednak o tego rodzaju sesjach należy pamiętać, w szczególności podczas rozwiązywania problemów, które w niemalże magiczny sposób pojawiają się jedynie w środowiskach produkcyjnych korzystających z mechanizmu równoważenia obciążenia.

Wprawdzie istnieje wiele sposobów tworzenia sesji trwałych w HTTP, ale najczęściej spotykany bazuje na ciasteczkach. W tym celu można wykorzystać ciasteczka aplikacji (np. ciasteczka sesji), o których istnieniu wie mechanizm równoważenia obciążenia, albo z oddzielnych ciasteczek dostarczanych przez ten mechanizm.

Implementacja sesji trwałych poprzez przechowywanie w mechanizmie równoważenia obciążenia dodatkowych informacji o stanie powoduje własne problemy. Jeżeli taki mechanizm będzie musiał przechowywać wewnętrzne mapowanie adresów IP na serwery aplikacji, to co się stanie, gdy mechanizm równoważenia obciążenia ulegnie awarii i zostanie zastąpiony nowym egzemplarzem, a tym samym utraci przechowywane informacje o stanie? Jak widzisz, problem stanu został przeniesiony z serwera aplikacji do mechanizmu równoważenia obciążenia i pozostaje mieć nadzieję, że nic złego się tam nie stanie. Jednak zgodnie z przysłowiem sama nadzieja nie jest najlepszą strategią.

Sprytnym rozwiązaniem umożliwiającym zachowanie sesji trwałych bez konieczności radzenia sobie z problemami dotyczącymi przechowywania informacji o stanie w mechanizmie równoważenia obciążenia jest wykorzystanie tzw. wartości skrótów dla adresów IP. W takim przypadku należy utworzyć wartość skrótu dla adresu IP klienta i użyć go do mapowania żądań adresów IP na egzemplarze usługi. O ile adres IP klienta pozostanie taki sam, sesja będzie „trwale” przypisana do konkretnego egzemplarza aplikacji.

W takim przypadku jeden lub więcej mechanizmów równoważenia obciążenia może w sposób deterministyczny dopasowywać adresy IP do serwerów aplikacji, bez konieczności odwoływania się do współdzielonego stanu. Serwery mogą być dowolnie tworzone i usuwane, a każdy nowy serwer będzie podejmował dokładnie te same decyzje w zakresie dopasowywania adresów IP. Jest to możliwe, ponieważ do tworzenia wartości skrótu jest używany algorytm, który zawsze dopasuje dany adres IP do tego samego serwera aplikacji.

Cykliczny mechanizm równoważenia obciążenia

Jeżeli sesje trwałe nie są wymagane, najczęściej spotyka się cykliczny mechanizm równoważenia obciążenia, który polega na tym, że każde kolejne połączenie lub żądanie jest kierowane do następnego egzemplarza. W kategoriach matematycznych będzie to oznaczało, że egzemplarz jest wybierany na podstawie wzoru $\text{request_count} \% \text{instance_count}$ (tutaj $\%$ oznacza modulo, czyli resztę z dzielenia).

Inne mechanizmy

W tym momencie znasz ogólny sposób działania mechanizmu równoważenia obciążenia HTTP w rzeczywistych rozwiązaniach. Oczywiście istnieje wiele innych dostępnych mechanizmów, np. rozkładających obciążenie na podstawie poziomu użycia zasobów. Jednak należy zachować ostrożność i dokładnie zrozumieć efekt ich bardziej złożonej działalności — wiele „sprytnie” działających algorytmów równoważenia obciążenia ma swoje poważne wady.

Na przykład mechanizm równoważenia obciążenia działający na podstawie poziomu użycia zasobów może mieć problemy z obsługą krótkotrwałych skoków obciążenia, które mogą prowadzić do nierównego wykorzystania egzemplarzy (jeden nadmiernie, podczas gdy inny w znikomym stopniu). Tak się dzieje, ponieważ w przypadku rzeczywistego obciążenia usług takie skoki się zdarzają, pomiar zaś może się odbyć w niewłaściwym czasie i tym samym nawet nie zarejestruje skoku.

Dodawanie kolejnej warstwy przeznaczonej do wyrównywania efektów owych skoków może prowadzić do kolejnych problemów, np. wielu takich skoków jeden na drugim. Jeżeli

zdecydujesz się odejść od utartych i sprawdzonych rozwiązań w zakresie mechanizmu równoważenia obciążenia, upewnij się, że Twój zespół dokładnie przeanalizował pod względem technicznym architekturę nowego rozwiązania, kontekst konkretnej aplikacji i sposób jej używania.

Wysoka dostępność

Wprawdzie podstawowym zadaniem mechanizmu równoważenia obciążenia może być szybkie udzielanie odpowiedzi, ale poza tym zajmuje się on również monitorowaniem dostępności usług. Może to być sprawdzanie stanu w celu potwierdzenia, że serwery, do których są wykonywane żądania, są w pełni funkcjonalne. Zatem mechanizm równoważenia obciążenia jest również sposobem na zapewnienie wysokiej dostępności i często również wewnętrznym elementem architektury bez przestoju, w której usługa może zostać zastąpiona nową (np. po wdrożeniu jej nowej wersji) w sposób niezauważalny dla klientów.

Ten efekt można osiągnąć poprzez umożliwienie eleganckiego zamykania egzemplarzy, kiedy to połączenia z klientami nie są po prostu zrywane, lecz pozostają aktywne aż do chwili ich pełnego przetworzenia, podczas gdy nowe połączenia się kierowane tylko do uaktualnionych egzemplarzy. Po zakończeniu ostatniej sesji w nieaktualnym egzemplarzu może on zostać w pełni zamknięty.

Sprawdzenie informacji o stanie pozwala mechanizmowi równoważenia obciążenia ustalić, czy usługa jest w pełni sprawna. Najprostsze sprawdzenie to oczywiście określenie możliwości nawiązania połączenia z daną usługą. Jednak w architekturze mikrousług, jeśli zależność zewnętrzna (np. inna usługa) będzie niedostępna, może to uniemożliwić innej usłudze poprawne udzielanie odpowiedzi na żądania. Można to sprawdzić również za pomocą oddzielnego punktu końcowego przeznaczonego dla informacji o stanie.

Wiele zespołów aplikacji i infrastruktury uzgodniło stosowanie trasy w stylu */healthcheck*, której kod stanu będzie wskazywał, czy żądania mogą być kierowane do danej usługi. W przypadku bardziej skomplikowanych środowisk tego rodzaju trasa może nawet wskazywać, *które* rodzaje żądań można wykonywać do danego egzemplarza.

Gdy doświadczeni programiści aplikacji i zespoły platformy/SRE współpracują ze sobą, trasy sprawdzania informacji o stanie mogą być również utworzone w sposób sygnalizujący sytuacje wymagające działania ze strony infrastruktury, np. egzemplarz znajduje się w skrajnie kiepskim stanie i wymaga zastąpienia. Jeżeli tego rodzaju trasy są bardzo dobrze opracowane, zwykle udzielają odpowiedzi razem z dodatkowym kontekstem i informacjami dotyczącymi problemu, co powinno ułatwić proces debugowania.

Kiedy infrastruktura obsługująca aplikację internetową rozrasta się i staje coraz bardziej skomplikowana, wówczas liczba potencjalnych problemów wzrasta wykładniczo i w dużym stopniu zależy od konkretnej architektury i aplikacji. Jednym z problemów, które stają się bardziej prawdopodobne wraz ze wzrostem liczby warstw proxy i poziomu złożoności infrastruktury internetowej, są pętle przekierowania i ogólnie błędy związane z przekierowaniami.

Na szczęście w powłoce mamy polecenie `curl`, które świetnie nadaje się do rozwiązywania tego rodzaju problemów.

Użycie polecenia curl do rozwiązywania problemów z przekierowaniami

Jak wspomnieliśmy, przekierowania mogą być symptomem błędów, problemów oraz bardziej ogólnie nieoczekiwanego sposobu działania aplikacji internetowej i otaczającej jej architektury. Do sprawdzenia przekierowań można użyć opcji `-L` (`--location`) polecenia `curl`:

```
curl -IL http://www.tutoriallinux.com/
HTTP/1.1 301 Moved Permanently
Server: nginx/1.24.0
Date: Sun, 22 Oct 2023 22:58:02 GMT
Content-Type: text/html
Content-Length: 169
Connection: keep-alive
Location: https://tutoriallinux.com/

HTTP/2 200
server: nginx/1.24.0
date: Sun, 22 Oct 2023 22:58:02 GMT
content-type: text/html; charset=UTF-8
vary: Accept-Encoding
x-powered-by: PHP/8.2.11
link: <https://tutoriallinux.com/wp-json/>; rel="https://api.w.org/"
strict-transport-security: max-age=31536000; includeSubdomains; preload
```

Jak widać, serwer udzielił odpowiedzi z kodem stanu 301 (trwale przeniesiony), a poprawny adres `https://tutoriallinux.com/.curl` sprawdza przekierowanie i wykonuje żądanie pod nowy adres, którego wynikiem jest kod stanu 200 (OK).

Przedstawione przekierowanie działa zgodnie z oczekiwaniami. Polecenie `curl` można wykorzystać także do zadań związanych z identyfikowaniem pętli przekierowań w aplikacji bądź podczas rozwiązywania problemów związanych z buforowaniem i routingiem w obejmujących wiele warstw konfiguracjach mechanizmu równoważenia obciążenia.

Jednak czasami trzeba zejść jeszcze głębiej i wysłać dane do aplikacji internetowej, aby przeprowadzić proces debugowania. W takich przypadkach polecenie `curl` również może pomóc.

Użycie polecenia curl jako narzędzia testowania API

Posiadanie w głowie gotowego polecenia `curl` podczas testowania API okazuje się przydatne znacznie częściej, niż można sądzić. Zwłaszcza podczas pracy z API JSON akceptującym dane żądania HTTP POST jest to powszechnie stosowany sposób wysyłania danych testowych do punktu końcowego, aby dane zwrócone przez backend były zgodne z oczekiwaniami.

```
curl --header "Content-Type: application/json" \
--request POST \
--data '{"some":"JSON","goes":"here"}' \
http://localhost:4000/api/v1/endpoint
```

Przedstawione tutaj polecenie używa kilku opcji, które należy zapamiętać. Pierwsza, `--header (-H)`, pozwala określić ciąg tekstowy nagłówka przeznaczonego do zdefiniowania. (Istnieje możliwość zdefiniowania wielu nagłówków poprzez powtórzenie tego argumentu). Druga, `--request (-X)`, umożliwia określenie typu żądania HTTP. (Domyślnie polecenie `curl` wykonuje żądania GET, choć można to zmienić za pomocą wymienionej opcji). Gdy używasz metody HTTP POST lub PATCH, jak w omawianym przykładzie, chcesz przekazać dane. Wówczas trzecia opcja, `--data (-d)`, pozwala wskazać dane przeznaczone do wysłania.

Używając opcji `--data`, pamiętaj o ważnej roli, jaką odgrywa tutaj znak sterujący powłoki. Zatem w przypadku bardziej skomplikowanych danych łatwiejsze okaże się korzystanie z opcji `--data` w pokazany tutaj sposób:

```
curl -X POST --data "@my/data/file" https://localhost/api/v1/endpoint
```

Nie wolno zapomnieć o poprzedzeniu znakiem `@` ścieżki dostępu do pliku. Jeżeli chcesz przekazać skomplikowane dane, zapoznaj się również z opcjami `--data-raw`, `--data-binary` i `--data-urlencode`. Konieczne może się okazać wysłanie także dodatkowych nagłówków, w zależności od oczekiwań aplikacji internetowej.

W ten sposób pokazaliśmy, jak można zastosować bardziej interaktywne podejście podczas rozwiązywania problemów z aplikacją internetową za pomocą polecenia `curl` przekazującego jej niestandardowe dane. Chcemy w tym miejscu zaprezentować jeszcze jedną sztuczkę związaną z tym poleceniem. Szyfrowanie TLS (ang. *transport layer security*) stosowane podczas szyfrowania nowoczesnego ruchu sieciowego w HTTPS niekoniecznie jest „błędnie rozumianym” aspektem aplikacji internetowej, lecz raczej częstym punktem awarii, w której usunięciu może pomóc polecenie `curl`.

Użycie polecenia `curl` do zaakceptowania i wyświetlenia nieprawidłowych certyfikatów TLS

Polecenie `curl` obsługuje opcję `--insecure`, która pozwala zaakceptować nieprawidłowy certyfikat TLS z serwera i kontynuować obsługę żądania. Taka możliwość okazuje się użyteczna podczas rozwiązywania problemów związanych z błędnie skonfigurowanymi serwerami.

```
curl --insecure -v https://www.tutoriallinux.org/
```

Opcja `--insecure (-k)` powoduje, że polecenie `curl` działa tak, jakby certyfikat TLS był prawidłowy, nawet jeśli tak nie jest. Oczywiście to wiąże się z pewnym ryzykiem i ta możliwość powinna być stosowana jedynie podczas rozwiązywania problemów. Pozwala ona kontynuować działanie przez polecenie `curl` nawet w sytuacji, gdy weryfikacja certyfikatu TLS zakończyła się niepowodzeniem (bez opcji `--insecure` w takim przypadku polecenie `curl` przerwałoby przetwarzanie żądania).

W następnym podrozdziale omówimy ostatni element HTTP, który warto poznać nieco dokładniej, jeśli chcesz zajmować się czymś więcej niż tylko tworzeniem aplikacji internetowych lub rozwiązywaniem związanych z nimi problemów: technologię CORS.

CORS

CORS (ang. *cross-origin resource sharing*) to wyszukany sposób na określenie, że zasoby — takie jak obrazy, wideo, dokumenty HTML, skrypty JavaScriptu, a nawet odpowiedzi **AJAX** (ang. *asynchronous javascript and xml*) — będą pochodziły z hosta o zupełnie innej nazwie. Pozwala to zapobiegać sytuacjom, w których zasoby są wczytywane z serwerów zewnętrznych. Przeglądarka internetowa najpierw poprosi o potwierdzenie takiej operacji. Jest to czasami określane mianem żądania *pre-flight*.

Pre-flight to żądanie **OPTIONS** oczekujące odpowiedzi, która zawiera nagłówki HTTP informujące, czy żądanie jest dozwolone. Tego rodzaju odpowiedź zwykle zawiera kod stanu odpowiedzi 204 (brak treści) oraz składa się jedynie z nagłówków. Jeżeli te nagłówki nie zostaną znalezione lub nie wskazują, czy żądanie jest dozwolone, wówczas do zasobu nie będą wykonywane kolejne żądania.

Zapoznaj się teraz z przykładem takiej wymiany informacji.

Przeglądarka internetowa wykonuje żądanie do serwera obsługującego domenę *https://www.example.org/* i pyta, czy możliwe jest wykonywanie żądań **POST** do */api/test* w *api.example.org*:

```
OPTIONS /api/test
Origin: https://www.example.org
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-Custom-Header, Content-Type
```

Jeżeli te żądania są dozwolone, udzielona odpowiedź będzie miała postać podobną do tutaj pokazanej:

```
HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://www.example.org
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-Custom-Header, Content-Type
Access-Control-Allow-Max-Age: 3600
```

Skoro ta odpowiedź wskazuje na zgodę dla takich żądań, przeglądarka może wykonać pierwotne żądanie, które teraz będzie autoryzowane:

```
POST /api/test
Origin: https://www.example.org
Content-Type: application/json
X-Custom-Header: foobarbaz
```

Natomiast gdy te żądania są niedozwolone, to na zapytanie o zgodę na nie zostanie udzielona odpowiedź z kodem stanu błędu, ponieważ to sugerowałoby odrzucenie żądania, a po prostu brakuje oczekiwanego nagłówka **Access-Control-Allow-Origin**. W takim przypadku klient wie, że żądanie jest nieautoryzowane i powoduje wygenerowanie komunikatu błędu.

Tego rodzaju błędy można zobaczyć w konsoli narzędzi programistycznych przeglądarki internetowej. Mają one postać podobną do tutaj przedstawionej:

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at https://not-allowed. (Reason: something).
```

Zamieściliśmy tutaj to krótkie wprowadzenie do technologii CORS, ponieważ jest to bardzo ważny temat dla programistów aplikacji internetowych. Wprawdzie ta technologia nie jest ściśle związana z powłoką, ale dla programisty jest oczywiste, że musi on rozumieć te koncepcje i sprawdzić klienta internetowego pod kątem tego typu błędów. Dokładniejsze informacje na temat tej technologii znajdziesz w artykule w portalu MDN opublikowanym pod adresem <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>.

Podsumowanie

W tym rozdziale przedstawiliśmy to, co należy wiedzieć, aby uniknąć najczęściej spotykanych błędnych przekonań, błędów i frustrujących problemów projektowych, które mogą się pojawić, gdy aplikacja internetowa opuści laptopa programisty i poprzez skomplikowaną infrastrukturę rozpocznie działanie z użytkownikami. Wspomnieliśmy o elementach infrastruktury takich jak brama i usługa upstream, które ułatwiają dostęp do aplikacji.

Omówiliśmy również wybrane z najczęściej popełnianych błędów przez programistów pracujących z protokołem HTTP. Ta wiedza powinna Ci pomóc w uniknięciu trudnych do wykrycia problemów związanych z nagłówkami, niepoprawnymi lub niejednoznacznymi kodami stanu itd. Wspomnieliśmy o technologii **CORS** i wyjaśniliśmy, jak protokół HTTP ewoluował do swojej obecnej postaci.

Najważniejsze naszym zdaniem było to, że pokazaliśmy Ci, jak możesz podnieść swoje umiejętności jako programisty poprzez znajomość polecenia powłoki `curl` i wykorzystanie go w połączeniu z teoretyczną znajomością protokołu HTTP.

Dzięki wiedzy zdobytej w tym rozdziale możesz szybko i skutecznie rozwiązywać problemy związane z aplikacją internetową, niezależnie od tego, czy będzie to zidentyfikowanie pętli przekierowania w uszkodzonej witrynie WordPressa, znalezienie subtelного błędu związanego z buforowaniem poprzez przeanalizowanie nagłówków zwróconych przez aplikację Ruby-on-Rails, czy też replikowanie błędu (i zweryfikowanie sposobu jego usunięcia) o czwartej nad ranem poprzez przekazywanie za pomocą żądania HTTP POST danych JSON do serwera programistycznego.

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 