

Apress®

Git

dla profesjonalistów

Wydanie II

Scott Chacon, Ben Straub

Helion 

Tytuł oryginału: Pro Git, 2nd Edition

Tłumaczenie: Wojciech Moch

ISBN: 978-83-283-8731-7

First published in English under the title Pro Git by Scott Chacon and Ben Straub, edition: 2

Copyright © 2014 by Scott Chacon and Ben Straub

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature. APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

Polish edition copyright © 2022 by Helion S.A.

All rights reserved.

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/gitpr2>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

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

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

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



Spis treści

O autorach	15
Przedmowa Scotta Chacona	17
Przedmowa Bena Strauba	19
Rozdział 1. Zaczynamy	21
Systemy kontroli wersji	21
Lokalne systemy kontroli wersji	21
Scentralizowane systemy kontroli wersji	22
Rozproszone systemy kontroli wersji	23
Krótką historia Gita	24
Podstawy Gita	25
Migawki, a nie różnice	25
Niemal każda operacja wykonywana jest lokalnie	26
Git sprawdza spójność danych	27
Git wyłącznie dopisuje dane	27
Trzy stany	28
Wiersz poleceń	29
Instalowanie Gita	29
Instalowanie w systemie Linux	29
Instalowanie w systemie Mac	30
Instalowanie w systemach Windows	31
Pierwsze konfigurowanie Gita	32
Tożsamość	32
Edytor	33
Sprawdzanie ustawień	33
Szukanie pomocy	33
Podsumowanie	34

Rozdział 2. Podstawy Gita	35
Pobieranie repozytorium Gita	35
Inicjowanie repozytorium w istniejącym katalogu	35
Klonowanie istniejącego repozytorium	36
Zapisywanie zmian w repozytorium	37
Sprawdzanie stanu plików	37
Śledzenie nowych plików	38
Umieszczanie zmodyfikowanych plików w przechowalni	39
Skrócona informacja o statusie	40
Ignorowanie plików	41
Przeglądanie zmian w przechowalni i poza nią	42
Zatwierdzanie zmian	44
Pomijanie obszaru przechowywania	45
Usuwanie plików	46
Przenoszenie plików	47
Przeglądanie historii commitów	48
Ograniczanie wielkości wydruku	52
Cofanie zmian	54
Usuwanie pliku z obszaru przechowywania	54
Wycofywanie zmian w zmodyfikowanym pliku	56
Praca ze zdalnymi repozytoriami	56
Wyświetlanie listy repozytoriów	57
Dodawanie zdalnego repozytorium	58
Pobieranie i wypychanie zmian do zdalnych repozytoriów	58
Wypychanie zmian do zdalnego repozytorium	59
Sprawdzanie zdalnego serwera	59
Usuwanie zdalnych serwerów i zmienianie ich nazw	61
Używanie tagów	61
Wypisywanie listy tagów	61
Tworzenie tagów	62
Tagi opisane	62
Tagi lekkie	63
Tworzenie tagów dla starszych commitów	63
Publikowanie tagów	64
Aliasy w Gicie	65
Podsumowanie	66
Rozdział 3. Rozgałęzienia	67
Rozgałęzienia w skrócie	67
Tworzenie nowej gałęzi	69
Przełączanie między gałęziami	70
Podstawy rozgałęziania i złączania	73
Podstawy rozgałęziania	74
Podstawy złączania	77
Proste konflikty w złączeniach	78
Zarządzanie gałęziami	81

Sposoby pracy z gałęziami	82
Długie gałęzie	82
Gałęzie tematyczne	83
Zdalne gałęzie	85
Wypychanie zmian	89
Gałęzie śledzące	91
Pobieranie danych	92
Usuwanie zdalnych gałęzi	92
Przebazowanie	93
Podstawowe przebazowanie	93
Bardziej interesujące przykłady przebazowania	95
Niebezpieczeństwa związane z przebazowaniem	97
Przebazowanie po przebazowaniu	100
Przebazowanie a złączenie	101
Podsumowanie	102
Rozdział 4. Git na serwerze	103
Protokoły	104
Protokół lokalny	104
Protokoły HTTP	105
Protokół SSH	107
Protokół Git	108
Przenoszenie Gita na serwer	109
Umieszczanie czystego repozytorium na serwerze	110
Małe konfiguracje	111
Generowanie własnych publicznych kluczy SSH	112
Konfigurowanie serwera	113
Demon Gita	115
Smart HTTP	116
GitWeb	118
GitLab	120
Instalacja	120
Administrowanie	121
Użytkownicy	121
Grupy	122
Projekty	123
Hooki	123
Podstawowa obsługa	123
Współpraca	124
Opcje hostowania na zewnętrznych serwerach	124
Podsumowanie	125
Rozdział 5. Rozproszony Git	126
Rozproszone metody pracy	126
Metoda scentralizowana	126
Metoda z menedżerem integracji	127

Metoda z dyktatorem i porucznikami	128
Podsumowanie metod pracy	129
Współpraca nad projektem	129
Poradnik tworzenia commitów	130
Mały, prywatny zespół	132
Prywatny zespół zarządzany	138
Projekt publiczny i kopia projektu	142
Projekt publiczny i wiadomości e-mail	146
Podsumowanie	148
Opieka nad projektem	149
Praca z gałęziami tematycznymi	149
Nakładanie łatek otrzymanych w e-mailu	149
Nakładanie łatki przy użyciu polecenia am	150
Integrowanie otrzymanych prac	155
Metody pracy z wielkimi złączeniami	157
Metody pracy używające przebazowania i wybierania commitów	158
Rerere	159
Oznaczanie wydań	160
Generowanie numeru kompilacji	161
Przygotowanie wydania	162
Polecenie shortlog	162
Podsumowanie	163
Rozdział 6. GitHub	164
Tworzenie i konfigurowanie konta	164
Dostęp przy użyciu protokołu SSH	166
Awatar	167
Adresy e-mail	169
Uwierzytelnianie dwuskładnikowe	169
Praca nad projektem	170
Tworzenie kopii projektów	170
Praca z GitHubem	171
Tworzenie żądania pobrania	172
Iterowanie w żądaniu pobrania	175
Zaawansowane żądania pobrania	179
Składnia Markdown	184
Składnia Flavored Markdown	185
Opieka nad projektem	189
Tworzenie nowego repozytorium	189
Dodawanie współpracowników	191
Zarządzanie żądaniami pobrania	191
Powiadomienia i napomknięcia	197
Pliki specjalne	200
Administrowanie projektem	201

Zarządzanie organizacją	203
Podstawy organizacji	203
Zespoły	204
Protokoły dla audytów	205
Stosowanie skryptów	206
Hooks	206
API serwisu GitHub	209
Octokit	214
Podsumowanie	214
Rozdział 7. Narzędzia Gita	215
Wybieranie commitów	215
Pojedynczy commit	215
Skrócony skrót SHA-1	215
Referencje gałęzi	217
Skrócone nazwy w protokole referencji	217
Referencje przodków	218
Zakresy commitów	220
Interaktywny obszar przechowywania	222
Umieszczanie plików w przechowalni i wycofywanie ich	223
Umieszczanie łatek w obszarze przechowywania	225
Stos zmian i oczyszczanie	226
Odkładanie prac na stosie	226
Kreatywne używanie stosu	228
Cofanie zmian pobranych ze stosu	229
Tworzenie gałęzi ze stosu	230
Czyszczenie katalogu roboczego	230
Podpisywanie swoich prac	231
Wprowadzenie do GPG	232
Podpisywanie tagów	232
Sprawdzanie tagów	233
Podpisywanie commitów	233
Wszyscy muszą używać podpisów	235
Wyszukiwanie	235
Narzędzie git grep	235
Przeszukiwanie protokołu Gita	237
Przeszukiwanie wierszy w protokole	237
Modyfikowanie historii	238
Modyfikowanie ostatniego commitu	239
Zmianianie wielu komunikatów commitów	239
Zmiana kolejności commitów	241
Skracanie historii commitów	242
Dzielenie commitu	242
Rozwiązanie nuklearne: polecenie filter-branch	243

Polecenie reset	245
Trzy drzewa	245
Sposób pracy	247
Działanie operacji reset	251
Polecenie reset ze ścieżką	254
Łączenie commitów	256
Polecenie checkout	258
Podsumowanie	260
Zaawansowane złączenia	260
Konflikty przy złączeniach	261
Przerywanie złączenia	263
Ignorowanie znaków białych	263
Ręczne ponowne złączenie pliku	264
Kontrolowanie konfliktów	266
Protokół złączenia	268
Łączony format diff	269
Cofanie złączenia	271
Inne typy złączeń	274
Funkcja rerere	277
Debugowanie za pomocą Gita	282
Opisywanie plików	282
Szukanie binarne	283
Podmoduły	285
Rozpoczynanie pracy z podmodułami	286
Klonowanie projektu z podmodułami	288
Praca w projekcie z podmodułami	289
Publikowanie zmian w podmodule	294
Złączanie zmian w podmodule	295
Wskazówki dotyczące podmodułów	298
Łączenie	302
Podmienianie	305
Przechowywanie danych uwierzytelniających	311
Jak to działa?	312
Własna pamięć podręczna dla danych uwierzytelniających	315
Podsumowanie	316
Rozdział 8. Dostosowywanie Gita	317
Konfigurowanie Gita	317
Podstawowa konfiguracja klienta	318
Kolory w Gicie	321
Zewnętrzne narzędzia do złączania i wypisywania różnic	322
Formatowanie i znaki białe	325
Konfiguracja serwera	327
Atrybuty Gita	328
Pliki binarne	328
Rozwijanie słów kluczowych	331

Eksportowanie repozytorium	334
Strategie łączenia	335
Hooki	335
Instalowanie hooka	335
Hooki klienckie	336
Hooki związane z zatwierdzaniem commitów	336
Hooki związane ze stosowaniem e-maili	337
Pozostałe hooki	337
Hooki serwerowe	338
Przykładowa reguła wymuszana przez Git	339
Hook serwerowy	340
Wymuszanie stosowania określonego formatu komunikatów commitów	340
Kontrola dostępu za pomocą list ACL	342
Testowanie	344
Hooki klienckie	345
Podsumowanie	348
Rozdział 9. Git i inne systemy	349
Git jako klient	349
Git i Subversion	349
Git i Mercurial	360
Git i Perforce	368
Git i TFS	382
Migracja do Gita	391
Subversion	391
Mercurial	393
Perforce	395
TFS	397
Niestandardowy importer	398
Podsumowanie	405
Rozdział 10. Wewnętrzne mechanizmy Gita	406
Wewnętrzne i piękne	406
Obiekty Gita	408
Obiekty drzewa	409
Obiekty commitów	412
Przechowywanie obiektów	414
Referencje	416
Wskaźnik HEAD	417
Tagi	418
Zdalne repozytoria	419
Pliki spakowane	419
Specyfikacja referencji	422
Wypychanie specyfikacji referencji	424
Usuwanie referencji	424

Protokoły transferu	425
Protokół podstawowy	425
Protokół rozbudowany	427
Podsumowanie protokołów	430
Konserwacja i odzyskiwanie danych	430
Konserwowanie	430
Odzyskiwanie danych	431
Usuwanie obiektów	433
Zmienne środowiskowe	437
Zachowanie globalne	437
Lokalizacje repozytorium	438
Specyfikacje ścieżek	438
Zatwierdzanie zmian	439
Praca w sieci	439
Złączanie i wyznaczanie różnic	439
Debugowanie	440
Zmienne różne	442
Podsumowanie	443
Dodatek A. Git i inne środowiska	445
Interfejsy graficzne	445
gitk i git-gui	445
GitHub dla systemów macOS i Windows	448
Inne narzędzia graficzne	451
Git w Visual Studio	451
Git w Eclipse	452
Git w powłoce Bash	453
Git w powłoce Zsh	454
Git w PowerShell	455
Podsumowanie	456
Dodatek B. Git wbudowany w aplikacje	457
Używanie Gita w wierszu poleceń	457
Libgit2	458
Zaawansowane funkcje	460
Inne powiązania	462
LibGit2Sharp	462
Objective-Git	462
Pygit2	463
Więcej informacji	463
Dodatek C. Polecenia Gita	464
Konfiguracja	464
git config	464
git help	465

Pobieranie i tworzenie projektów	465
git init	465
git clone	465
Tworzenie migawek	466
git add	466
git status	467
git diff	467
git difftool	467
git commit	468
git reset	468
git rm	468
git mv	469
git clean	469
Gałęzie i złączanie	469
git branch	469
git checkout	470
git merge	470
git mergetool	470
git log	471
git stash	471
git tag	472
Udostępnianie i aktualizowanie projektów	472
git fetch	472
git pull	472
git push	473
git remote	473
git archive	474
git submodule	474
Kontrola i porównywanie	474
git show	474
git shortlog	474
git describe	475
Debugowanie	475
git bisect	475
git blame	475
git grep	475
Łatki	475
git cherry-pick	476
git rebase	476
git revert	476
E-mail	476
git apply	477
git am	477
git format-patch	477
git send-email	477
git request-pull	477

Zewnętrzne systemy	478
git svn	478
git fast-import	478
Administrowanie	478
git gc	478
git fsck	478
git reflog	478
git filter-branch	479
Polecenia wewnętrzne	479

ROZDZIAŁ 1.



Zaczynamy

W tym rozdziale zaczniemy pracować z systemem Git. Na początek omówimy podstawy działania systemów kontroli wersji, następnie przystąpimy do uruchamiania Gita w swoim systemie, a zakończymy odpowiednim skonfigurowaniem go i przygotowaniem do właściwej pracy. Po przeczytaniu tego rozdziału będziesz wiedzieć, dlaczego w ogóle używamy Gita i dlaczego warto z niego korzystać. Będziesz też w stanie przygotować sobie działający system.

Systemy kontroli wersji

Czym właściwie są „systemy kontroli wersji” i dlaczego musimy się nimi zajmować? Każdy z takich systemów zajmuje się zapamiętywaniem zmian wprowadzanych do pliku albo zbioru plików, co pozwala na późniejsze odwoływanie się do poszczególnych wersji. W przykładach prezentowanych w tej książce będziemy korzystać z plików zawierających kod źródłowy programów, choć w rzeczywistości można tu używać niemal dowolnego rodzaju plików, jakie znajdziesz w swoim komputerze.

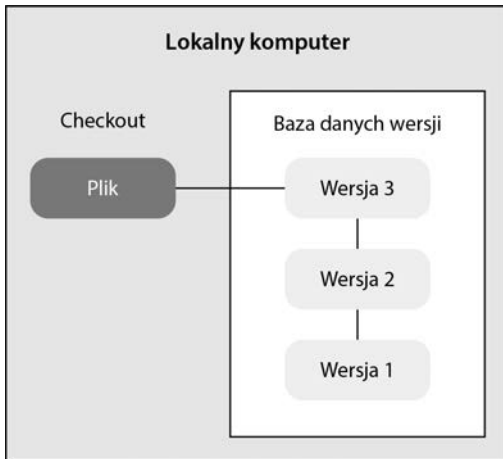
Jeżeli pracujesz jako grafik lub projektant stron WWW i chcesz przechowywać wszystkie wersje tworzonych obrazków lub stron (a na pewno warto to robić), zainteresuj się jednym z systemów kontroli wersji (ang. *Version Control System* — VCS). Każdy z tych systemów pozwala na przywrócenie wcześniejszych wersji pliku albo nawet całego projektu, umożliwia przeglądanie zmian wprowadzanych w kolejnych wersjach, sprawdzanie, kto ostatnio wprowadzał modyfikacje, które mogą powodować różne problemy i wiele innych. Kiedy korzystamy z systemu VCS, to nawet w przypadku utraty plików albo ich zepsucia zawsze możemy w łatwy sposób odzyskać poprawną postać. Co ważne, wszystkie te funkcje wymagają od nas tylko niewielkiego zaangażowania.

Lokalne systemy kontroli wersji

Wiele osób prowadzi własny system kontroli wersji polegający na kopiowaniu plików do innego katalogu (w bardziej zaawansowanych rozwiązaniach takie katalogi oznaczane są

datą i czasem). To popularne rozwiązanie, ponieważ jest bardzo proste, choć jednocześnie jest też niezwykle podatne na różnorakie błędy. Można przecież łatwo pomylić się przy kopiowaniu i przez przypadek zapisać niewłaściwy plik albo nadpisać istniejące już pliki.

Programiści już dawno poradzili sobie z tym problemem, tworząc lokalne systemy kontroli wersji, które budują prostą bazę danych przechowującą wszystkie zmiany wprowadzane do plików objętych kontrolą (rysunek 1.1).



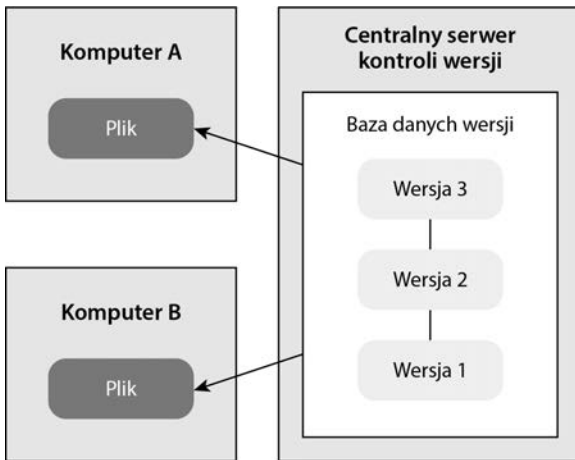
Rysunek 1.1. Lokalna kontrola wersji

Jednym z popularniejszych systemów kontroli wersji był system o nazwie RCS, który nadal można znaleźć na wielu komputerach. Nawet w popularnym systemie operacyjnym Mac OS X po zainstalowaniu narzędzi dla programistów dostępne jest polecenie `rcs`. System RCS przechowuje na dysku zestawy poprawek (czyli różnic między plikami) zapisane w specjalnym formacie. Zatem z jego pomocą można odtworzyć zawartość dowolnego pliku w wybranej wersji, po prostu nakładając kolejne poprawki na pierwotną postać pliku.

Scentralizowane systemy kontroli wersji

Kolejnym problemem, z jakim musimy się zmagać, jest konieczność współpracowania z innymi programistami, używającymi innych systemów i komputerów. Z tym problemem poradzono sobie za pomocą scentralizowanych systemów kontroli wersji (ang. *Centralized Version Control Systems* — CVCS). W tych systemach (należą do nich CVS, Subversion albo Perforce) istnieje specjalny serwer przechowujący wszystkie pliki poddawane wersjonowaniu, a dowolna liczba klientów może pobierać te pliki z serwera (rysunek 1.2). Przez wiele lat ten rodzaj systemów był standardem w rozwiązaniach kontroli wersji.

Ten rodzaj systemów ma wiele zalet, szczególnie w porównaniu z systemami lokalnymi. Przykładowo do pewnego stopnia każdy wie, nad czym pracują pozostali członkowie zespołu projektowego. Administratorzy mają możliwość dokładnego określenia uprawnień poszczególnych użytkowników, a samo administrowanie systemem CVCS jest znacznie łatwiejsze niż obsługiwanie lokalnych baz danych znajdujących się na wszystkich komputerach klienckich.



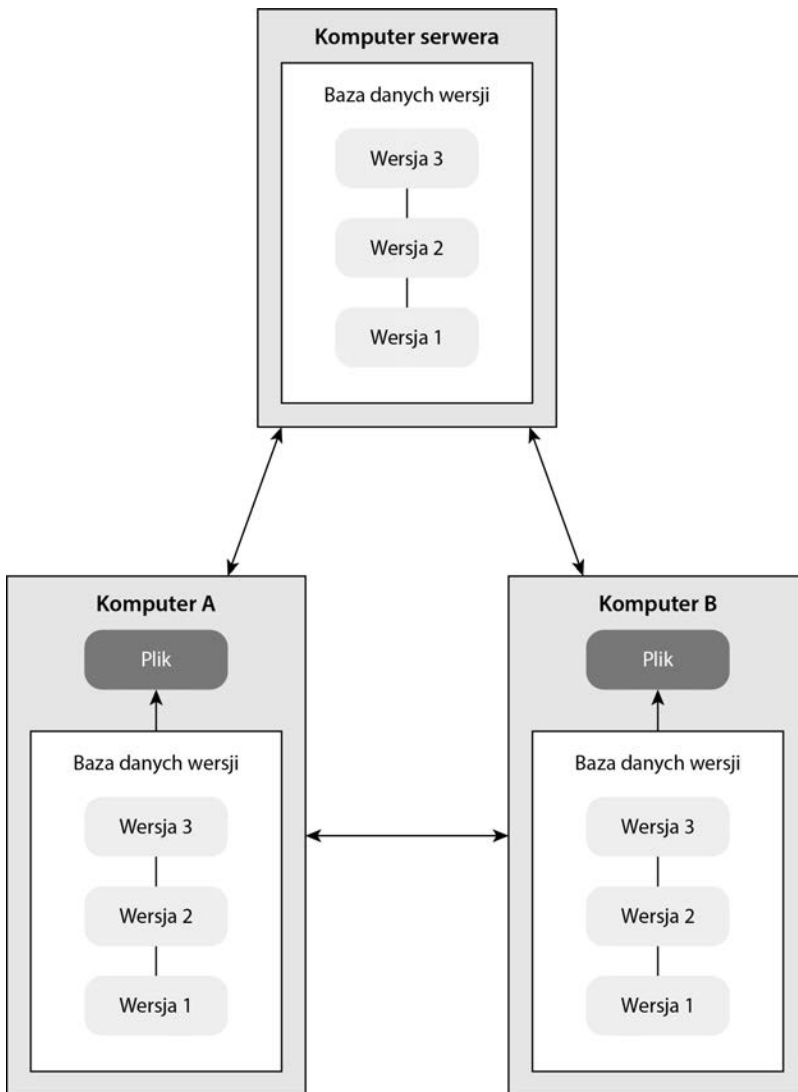
Rysunek 1.2. Scentralizowany system kontroli wersji

Niestety takie systemy mają też dość poważne wady. Najbardziej oczywistą jest pojedynczy punkt awarii, którym jest sam centralny serwer. Jeżeli serwer nie będzie działał przez godzinę, przez ten czas nikt nie będzie w stanie zapisać zmienionych przez siebie plików ani pobrać zmian wprowadzonych przez innych. Jeżeli awarii ulegnie dysk twardy centralnego serwera, a nikt nie zadbał o wykonywanie odpowiednich kopii zapasowych, tracimy wszystko — całą historię projektu. Pozostają jedynie pliki, jakie poszczególni użytkownicy mają na swoich lokalnych komputerach. Z podobnym problemem zmagają się też lokalne systemy kontroli wersji. Jeżeli historia projektu jest zapisana w jednym miejscu, istnieje poważne ryzyko utraty wszystkich informacji o tym projekcie.

Rozproszone systemy kontroli wersji

I tu przechodzimy do rozproszonych systemów kontroli wersji (ang. *Distributed Version Control Systems* — DVCS). W tym typie systemów (należą do nich Git, Mercurial, Bazaar albo Darcs) klienci nie pobierają jedynie ostatniej postaci każdego pliku, ale wykonują pełną kopię całego repozytorium. Jeżeli zatem dowolny serwer ulegnie awarii, całe repozytorium może zostać skopiowane z jednego z klientów. W ten sposób stan serwera może zostać łatwo odtworzony. Oznacza to jednak, że przy każdym pobieraniu danych z serwera musimy wykonać pełną kopię zapasową przechowywanych na nim informacji (rysunek 1.3).

Co ważne, każdy z tych systemów świetnie radzi sobie z koniecznością pracy z kilkoma zdalnymi repozytoriami, co oznacza, że możemy w różny sposób współpracować z niezależnymi zespołami w ramach jednego projektu. Ten rodzaj systemów umożliwia stosowanie kilku różnych schematów wykonywania pracy, co nie jest możliwe w scentralizowanych systemach.



Rysunek 1.3. Rozproszony system kontroli wersji

Krótką historia Gita

Jak wiele innych wartościowych rzeczy, Git powstał w wyniku kreatywnej destrukcji oraz wielkich kontrowersji.

Jądro systemu Linux jest bardzo rozbudowanym projektem oprogramowania o otwartych źródłach. Przez dużą część czasu istnienia tego projektu (lata 1991 – 2002) zmiany w oprogramowaniu były przesyłane w postaci poprawek oraz plików archiwów. W 2002 roku projekt jądra systemu zaczął korzystać z własnościowego, rozproszonego systemu kontroli wersji o nazwie BitKeeper.

W 2005 roku związki łączące społeczność zajmującą się rozwojem jądra Linuksa oraz firmą rozwijającą system BitKeeper zostały zerwane, a za korzystanie z narzędzia dostępnego dotychczas bez opłat zaczęto żądać zapłaty. W efekcie cała społeczność Linuksa (a szczególnie Linus Torvalds — twórca tego systemu) przystąpiła do tworzenia własnego narzędzia, wykorzystując doświadczenia wyniesione z pracy z systemem BitKeeper. Twórcy nowego systemu chcieli osiągnąć następujące cele:

- szybkość,
- prosty projekt,
- dobrą obsługę nieliniowego rozwoju (tysięcy równoległych rozgałęzień),
- całkowite rozproszenie,
- możliwość skutecznego obsługiwania wielkich projektów, takich jak projekt jądra Linuksa (z naciskiem na szybkość i wielkość danych).

Od powstania w 2005 roku Git cały czas ewoluuje i dojrzewa, starając się zachowywać początkowe założenia i umożliwiać jak najprostszą obsługę. Nadal jest niezwykle szybki, bardzo sprawny w pracy z wielkimi projektami i udostępnia niesamowity system tworzenia odgałęzień pozwalający na nieliniowy rozwój projektu (więcej na ten temat w rozdziale 3.).

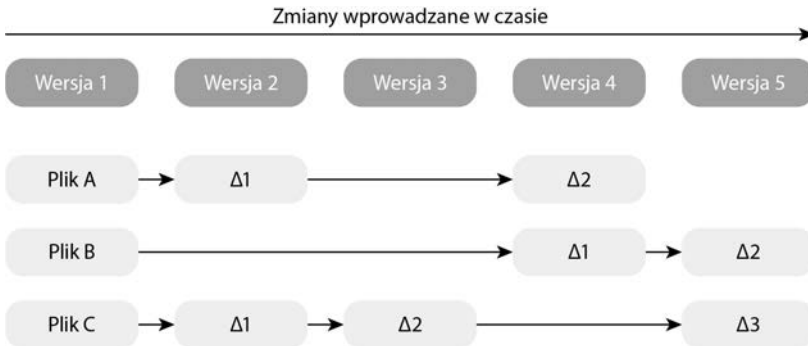
Podstawy Gita

Jak działa Git? To bardzo ważna część rozdziału, ponieważ musisz dowiedzieć się, czym jest Git i jakie mechanizmy działają u jego podstaw, a to bardzo ułatwi Ci skuteczne korzystanie z jego funkcji. Poznając Git, staraj się oczyścić umysł z informacji, jakie masz na temat innych systemów kontroli wersji, takich jak Subversion lub Perforce. Dzięki temu unikniesz małych nieporozumień podczas korzystania z nowego narzędzia. Git przechowuje i obsługuje informacje inaczej niż inne systemy, mimo że nie widać tego w interfejsie użytkownika. Poznanie tych różnic pozwala uniknąć nieporozumień w trakcie korzystania z systemu.

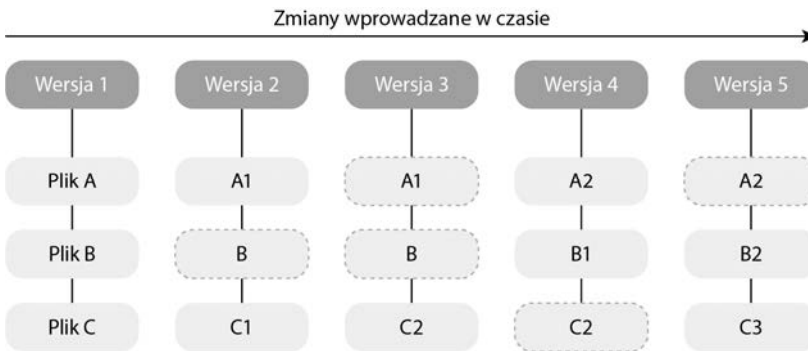
Migawki, a nie różnice

Najważniejszą różnicą między Gitem a pozostałymi systemami kontroli wersji (takimi jak Subversion i podobne) jest sposób, w jaki Git traktuje dane. Inne systemy przechowują informacje w postaci listy zmian wprowadzanych do plików. Systemy, takie jak Subversion, Perforce, Bazaar i tym podobne, wszystkie swoje dane zapisują w postaci pliku bazowego oraz zbioru zmian, jakie z czasem były wprowadzane do tego pliku (rysunek 1.4).

Git nie przechowuje i nie obsługuje danych w ten sam sposób. Traktuje raczej wszystkie dane jak zbiór migawek w miniaturowym systemie plików. Git przy wykonywaniu każdego zatwierdzenia zmian albo zapisywania stanu projektu wykonuje migawkę zawartości wszystkich plików projektu i zapisuje sobie referencję do niej. W ramach zwiększania wydajności pracy wszystkie niezmienione pliki nie są ponownie zapisywane w systemie, a w migawce zapisywane jest łącze do zapisanej wcześniej, takiej samej wersji pliku. Jak widać, Git traktuje przechowywane dane jak *strumień migawek* (rysunek 1.5).



Rysunek 1.4. Przechowywanie danych jako zmian wprowadzanych do bazowej wersji pliku



Rysunek 1.5. Przechowywanie danych jako migawek projektu tworzonych w czasie

To bardzo ważna cecha wyróżniająca Git spośród innych systemów kontroli wersji. Sprawia ona, że Git niemal w każdym aspekcie działa nieco inaczej niż większość systemów kontroli wersji poprzedniej generacji. Powoduje to, że Git jest raczej minisystemem plików obudowanym bardzo zaawansowanymi narzędziami, a nie tylko prostym systemem VCS. W rozdziale 3., podczas omawiania rozgałęzień, przyjrzymy się niektórym zaletom takiego traktowania danych, jakie zastosowano w Gicie.

Niemal każda operacja wykonywana jest lokalnie

Większość operacji w Gicie wymaga dostępu wyłącznie do lokalnych plików i zasobów. Zwykle nie potrzeba żadnych informacji z innych komputerów znajdujących się w sieci. Kto używał wcześniej scentralizowanych systemów kontroli wersji, w których niemal każda operacja obciążona jest opóźnieniami wynikającymi z komunikacji sieciowej, może pomyśleć, że bogowie prędkości obdarowali Git jakimiś nadzwyczajnymi zdolnościami. Ponieważ cała historia projektu znajduje się na Twoim dysku, większość operacji realizowana jest niemal natychmiastowo.

Aby na przykład przejrzeć historię projektu, Git nie musi pobierać jej z centralnego serwera i przygotowywać otrzymanych danych do wyświetlenia na ekranie. Wszystkie potrzebne informacje ma już gotowe w swojej lokalnej bazie danych, a to oznacza, że historię projektu

można zobaczyć właściwie od razu. Jeżeli chcesz zobaczyć zmiany wprowadzone między aktualną wersją pliku a jego wersją sprzed miesiąca, Git może pobrać z bazy wersję pliku sprzed miesiąca i od razu wyznaczyć różnice między tymi plikami. Nie musi prosić zdalnego serwera o wyliczenie tych różnic ani pobierać starej wersji pliku z serwera, aby samodzielnie wykonać porównanie.

Oznacza to również, że w przypadku braku połączenia z siecią albo z firmową siecią VPN nadal mamy możliwość wykonywania niemal wszystkich operacji. Podróżując samolotem albo koleją, nadal możesz pracować nad projektem i zatwierdzać swoje zmiany, aby przesłać je do głównego repozytorium po uzyskaniu połączenia z siecią. Możesz pracować w domu nawet wtedy, kiedy klient VPN nie działa tam poprawnie. W wielu innych systemach kontynuowanie pracy w takich warunkach jest albo niemożliwe, albo bardzo utrudnione. Przykładowo w systemie Perforce bez połączenia z serwerem nie można zrobić prawie nic, natomiast w systemach Subversion i CVS można — co prawda — edytować pliki, ale nie da się już wprowadzić tych zmian do bazy danych (ponieważ ta jest nieosiągalna). Może Ci się to wydawać nieistotnym szczegółem, ale szybko się przekonasz, jaka to wielka różnica w komforcie pracy.

Git sprawdza spójność danych

W Gicie wszystko przed zapisaniem w bazie danych otrzymuje swoją sumę kontrolną, z jej wykorzystaniem można się później odwoływać do poszczególnych obiektów. Oznacza to, że nie da się zmienić zawartości żadnego pliku lub katalogu w taki sposób, żeby Git nie otrzymał o tym informacji. Funkcje te są wbudowane w Git na bardzo niskim poziomie, stając się integralną częścią jego filozofii. Nie da się utracić przesyłanych informacji ani pobrać uszkodzonego pliku, ponieważ Git natychmiast wykrywa takie zdarzenia.

Do wyliczania sum kontrolnych w Gicie używa się algorytmu SHA-1. W efekcie powstaje 40-znakowy ciąg składający się ze znaków szesnastkowych (0 – 9 i a – f), wyliczany na podstawie zawartości pliku lub katalogu. Wyliczona wartość skrótu SHA-1 wygląda podobnie do poniższej:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Podczas pracy z Gitem zobaczysz te skróty w wielu miejscach, ponieważ są w Gicie używane często. Co więcej, Git zapisuje dane w swojej bazie, nie przypisując im nazw plików, ale identyfikuje je za pomocą wartości sumy kontrolnej.

Git wyłącznie dopisuje dane

Podczas wykonywania operacji za pomocą Gita zauważysz, że niemal wszystkie dopisują jedynie dane do jego bazy. Bardzo trudno zmusić system do wykonania operacji, której nie da się cofnąć, albo do usunięcia informacji. Jak w każdym innym systemie kontroli wersji, możliwe jest zniszczenie lub utracenie zmian, które nie zostały jeszcze zatwierdzone, ale po przesłaniu migawki do Gita bardzo trudno utracić dane, szczególnie w przypadku, gdy regularnie wypychasz swoje zmiany do innego repozytorium.

To bardzo uprzyjemnia pracę, ponieważ możemy dowolnie eksperymentować bez obawy, że zepsujemy coś w projekcie.

Trzy stany

A teraz skup się. To najważniejsza rzecz, jaką musisz zapamiętać na temat Gita, jeżeli chcesz, żeby późniejsza nauka nie sprawiała Ci trudności. Git definiuje trzy stany, w których mogą znajdować się pliki: zatwierdzony (ang. *committed*), zmodyfikowany (ang. *modified*) lub śledzony (ang. *staged*). Stan *zatwierdzony* oznacza, że dane są bezpiecznie zapisane w lokalnej bazie danych. Stan *zmodyfikowany* oznacza, że do pliku wprowadzone zostały zmiany, ale nie zostały jeszcze zatwierdzone w bazie. Stan *śledzony* oznacza, że aktualna wersja pliku została oznaczona jako część następczej zatwierdzanej migawki.

Te trzy stany przekładają się na trzy główne sekcje każdego projektu Gita: katalog Gita, katalog roboczy i obszar przechowywania (rysunek 1.6).



Rysunek 1.6. Katalog roboczy, obszar przechowywania i katalog Gita

Katalog Gita jest miejscem, w którym system przechowuje swoje metadane oraz bazę danych obiektów Twojego projektu. To najważniejsza część systemu Git, która jest kopiowana podczas klonowania repozytorium z innego komputera.

Katalog roboczy jest kopią jednej z wersji projektu. Pliki z tego katalogu są pobierane ze skompresowanej bazy danych znajdującej się w katalogu Gita i umieszczane we wskazanym miejscu na dysku, gdzie można poddawać je modyfikacjom.

Obszar przechowywania jest plikiem, który najczęściej znajduje się w katalogu Gita. W tym pliku zapisywane są informacje na temat elementów, które mają być częścią następczej operacji zatwierdzania. Czasami stosowany jest też termin „indeks”, ale najczęściej mówi się jednak o obszarze przechowywania lub po prostu o przechowalni.

Oto najprostszy sposób pracy z Gitem.

1. Zmieniasz zawartość plików w katalogu roboczym.
2. Oznaczasz pliki jako śledzone, czyli umieszczasz migawki ich zawartości w obszarze przechowywania.
3. Wykonujesz zatwierdzenie, w wyniku czego zawartość plików z obszaru przechowywania zapisywana jest na stałe w katalogu Gita.

Jeżeli jakaś wersja pliku znajduje się w katalogu Gita, to znaczy, że została zatwierdzona. Jeżeli plik został zmodyfikowany, ale też przeniesiony do obszaru przechowywania, to taki plik jest śledzony. Natomiast plik w katalogu roboczym z naniesionymi poprawkami, ale nie przeniesiony do obszaru przechowywania, jest plikiem zmodyfikowanym. W rozdziale 2. dokładniej omówimy te stany, podając metody ich wykorzystywania oraz sposób na pominięcie etapu śledzenia pliku.

Wiersz poleceń

Istnieje wiele różnych metod używania Gita. Do dyspozycji mamy oryginalne narzędzia działające w wierszu poleceń, ale istnieje też wiele rozwiązań z graficznym interfejsem użytkownika. W tej książce będziemy używać Gita w wierszu poleceń. Jedynie w wierszu poleceń można uruchomić wszystkie polecenia Gita, ponieważ większość narzędzi graficznych dla uproszczenia implementuje jedynie pewien podzbiór funkcji Gita. Jeżeli wiesz, jak posługiwać się narzędziem w wierszu poleceń, z pewnością łatwo domyślisz się, jak należy użyć go w programie z interfejsem graficznym. Niestety ta zależność nie działa w drugą stronę. Poza tym wybór klienta graficznego jest kwestią osobistych upodobań, ale każdy użytkownik będzie miał zainstalowane dokładnie te same narzędzia wiersza poleceń.

Oczekujemy zatem, że wiesz, jak należy otworzyć *Terminal* w komputerach Mac albo okno *Wiersza Poleceń* lub *PowerShell* w systemach Windows. Jeżeli nie wiesz, o czym tutaj mówimy, lepiej zatrzymaj się w tym miejscu i szybko poznaj obsługę swojego wiersza poleceń, aby odpowiednio interpretować podawane w tej książce przykłady oraz ich opisy.

Instalowanie Gita

Zanim zaczniesz używać Gita, musisz umieścić odpowiednie oprogramowanie na swoim komputerze. Nawet jeżeli masz zainstalowane narzędzia Gita, dobrym pomysłem jest zaktualizowanie ich do najnowszej wersji. Możesz to zrobić, instalując w systemie pakiet Gita albo stosując specjalny instalator. Można też pobrać kod źródłowy systemu i samodzielnie go skompilować.

-
- **Uwaga** Książkę tę pisaliśmy, używając Gita w wersji 2.0.0. Oznacza to, że większość prezentowanych tu narzędzi powinna działać nawet w bardzo starych wersjach Gita. Część z nich może jednak nie pracować zgodnie z opisem. Ponieważ Git świetnie radzi sobie z zachowywaniem zgodności wstecznej, podawane tu instrukcje będą działały również w wersjach nowszych niż 2.0.
-

Instalowanie w systemie Linux

Jeżeli chcesz zainstalować Git w systemie Linux za pomocą instalatora binarnego, najłatwiej będzie Ci skorzystać z narzędzi do zarządzania pakietami dostępnych w używanej przez Ciebie dystrybucji. Jeśli na przykład pracujesz z systemem Fedora, możesz zastosować narzędzie yum:

```
$ yum install git
```

Jeżeli jednak korzystasz z dystrybucji bazującej na Debianie, takiej jak Ubuntu, użyj narzędzia `apt-get`:

```
$ apt-get install git
```

Więcej instrukcji opisujących sposoby instalowania Gita w kilku różnych odmianach systemów uniksowych znajdziesz na stronach Gita, pod adresem <https://git-scm.com/download/linux>.

Instalowanie w systemie Mac

W systemie Mac można zainstalować Git na kilka różnych sposobów. Najprostszym z nich jest chyba zainstalowanie narzędzi wiersza poleceń Xcode. W systemie Mavericks (10.9) lub nowszych możesz po prostu spróbować uruchomienia polecenia `git` w oknie Terminal. Jeżeli Git nie jest jeszcze zainstalowany, system sam zapyta, czy chcesz go zainstalować.

Jeżeli chcesz korzystać z aktualniejszej wersji Gita, możesz zainstalować go za pomocą instalatora binarnego (rysunek 1.7). Instalator dla systemu OSX jest dostępny do pobrania na stronach Gita, pod adresem <https://git-scm.com/download/mac>.



Rysunek 1.7. Instalator Gita dla systemu OS X

Możesz też zainstalować go jako część instalacji GitHuba dla systemów Mac. W graficznym narzędziu Gita znajduje się też opcja zainstalowania wszystkich narzędzi wiersza poleceń. Odpowiedni instalator możesz pobrać ze stron projektu GitHub for Mac, pod adresem <https://mac.github.com>.

Instalowanie w systemach Windows

Istnieje tylko kilka sposobów instalowania Gita w systemach Windows. Najbardziej oficjalna kompilacja jest zawsze dostępna do pobrania na stronach Gita. Wystarczy wpisać w przeglądarce adres <https://git-scm.com/download/win>, a pobieranie powinno się rozpocząć automatycznie. Pamiętaj, że jest to projekt o nazwie Git for Windows (nazywany też msysGit), który jest niezależny od samego projektu Gita. Więcej informacji na jego temat znajdziesz pod adresem <https://msysgit.github.io/>.

Inną prostą metodą zainstalowania Gita w swoim systemie jest zainstalowanie w nim aplikacji GitHub for Windows. W jej instalatorze znajdują się też narzędzia wiersza poleceń Gita oraz specjalne GUI. Instalator dobrze współpracuje z powłoką PowerShell i właściwie konfiguruje mechanizm buforowania danych uwierzytelniających, jak również poprawnie definiuje znaki końca wiersza. Więcej informacji na ten temat podamy dalej w książce, a na razie wystarczy powiedzieć, że instalator przyjmuje tu właściwe ustawienia dla tych opcji. Instalator GitHub for Windows możesz pobrać z adresu <https://windows.github.com>.

Niektórzy mogą zdecydować się na zainstalowanie Gita, kompilując jego kod źródłowy, ponieważ w ten sposób uzyskują zawsze najnowszą wersję. Instalatory binarne zwykle powstają z małym opóźnieniem, choć nie ma to już większego znaczenia, ponieważ projekt Gita zdążył dojrzeć przez te wszystkie lata.

Jeżeli chcesz zainstalować Git z jego kodów źródłowych, musisz najpierw pobrać biblioteki, z których korzysta Git, czyli *curl*, *zlib*, *openssl*, *expat* i *libiconv*. Przykładowo w systemach używających programów yum (takich jak Fedora) lub apt-get (takich jak systemy bazujące na Debianie) możesz skorzystać z poniższych poleceń, aby zainstalować wszystkie niezbędne zależności:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

Po zainstalowaniu wszystkich zależności możesz przystąpić do pobierania paczki z kodem źródłowym oficjalnego wydania, która jest dostępna w kilku różnych miejscach. Możesz na przykład pobrać ją z witryny *kernel.org*, spod adresu <https://mirrors.edge.kernel.org/pub/software/scm/git/>, ale możesz też skorzystać z serwera lustrzanego GitHuba, pod adresem <https://github.com/git/git/releases>. Zwykle to na stronach GitHuba można znacznie łatwiej zorientować się, który pakiet jest najnowszy, ale na stronach *kernel.org* dostępne są sygnatury pakietów, co pozwala na skontrolowanie poprawności pobranych danych.

Sama kompilacja i instalacja wygląda tak:

```
$ tar -zxf git-1.9.1.tar.gz
$ cd git-1.9.1
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

Po wykonaniu takiej instalacji możesz też użyć Gita, aby z jego pomocą pobierać aktualizacje:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Pierwsze konfigurowanie Gita

Po zainstalowaniu Gita w swoim systemie warto wykonać kilka operacji, aby skonfigurować swoje środowisko. Na każdym komputerze trzeba te operacje wykonać tylko raz, ponieważ ustalona w ten sposób konfiguracja nie zmienia się po zainstalowaniu aktualizacji. Oczywiście opcje te można zmienić w dowolnym momencie, ponownie uruchamiając te same polecenia.

Git udostępnia specjalne narzędzie o nazwie `git config`, które umożliwia odczytywanie i ustalanie zmiennych konfiguracyjnych wpływających na działanie i wygląd Gita. Zmienne konfiguracyjne mogą być zapisywane w trzech różnych miejscach.

- Plik `/etc/gitconfig`. Przechowuje wartości konfiguracji obowiązujące wszystkich użytkowników systemu oraz wszystkie repozytoria. Jeżeli do polecenia `git config` dodasz opcję `--system`, konfiguracja będzie odczytywana i zapisywana do tego pliku.
- Plik `~/.gitconfig` lub `~/.config/git/config`. Przechowuje konfigurację właściwą dla wybranego użytkownika. Podając Gitowi opcję `--global`, nakazujemy mu korzystać z tego właśnie pliku.
- Plik `config` znajdujący się w katalogu Gita (czyli `.git/config`) związanym z dowolnym używanym właśnie repozytorium. Przechowuje konfigurację właściwą dla tego repozytorium.

Każdy następny poziom nadpisuje wartości z poprzednich poziomów, a zatem zapisy z pliku `.git/config` są ważniejsze niż te z pliku `/etc/config`.

W systemach Windows Git poszukuje pliku `.gitconfig` w katalogu głównym użytkownika (dla większości użytkowników będzie to katalog `C:\Users\NAZWA_UŻYTKOWNIKA`). Poszukuje też pliku `/etc/gitconfig`, choć ta ścieżka wyznaczana jest względem katalogu, w którym zainstalowano Git w systemie Windows.

Tożsamość

Pierwszą rzeczą, jaką trzeba zrobić po zainstalowaniu Gita, jest podanie mu swojej nazwy użytkownika oraz adresu e-mail. To bardzo ważne, ponieważ Git używa tych informacji w każdej operacji zatwierdzania, a zatem każda zmiana umieszczona w repozytorium będzie miała je dołączone w sposób uniemożliwiający późniejszą modyfikację.

```
$ git config --global user.name "Adam Nowak"
$ git config --global user.email adamnowak@przyklad.pl
```

W tym przypadku również możesz zdefiniować te ustawienia tylko raz, dodając do tych instrukcji opcję `--global`, ponieważ w ten sposób Git będzie zawsze korzystał z tych informacji przy wykonywaniu jakichkolwiek operacji w systemie. Jeżeli chcesz zmienić te wartości

i w wybranym projekcie używać innej nazwy użytkownika lub adresu e-mail, to w tym projekcie możesz uruchomić te same polecenia bez opcji `--global`.

Większość narzędzi z interfejsem graficznym prosi o podanie tych informacji zaraz po pierwszym uruchomieniu.

Edytor

Po zdefiniowaniu swojej tożsamości możesz przystąpić do konfigurowania domyślnego edytora, który będzie używany za każdym razem, gdy Git będzie prosił Cię o wpisanie wiadomości. Jeżeli ta opcja nie zostanie skonfigurowana, Git będzie korzystał z domyślnego edytora systemowego, którym najczęściej jest Vim. Jeżeli chcesz używać innego edytora tekstu, takiego jak Emacs, skorzystaj z poniższego polecenia:

```
$ git config --global core.editor emacs
```

-
- **Uwaga** Vim i Emacs są popularnymi edytorami tekstu często używanymi przez programistów w systemach uniksowych, takich jak Linux lub Mac. Jeżeli nie używasz tych edytorów albo pracujesz w systemach Windows, musisz poszukać specjalizowanych instrukcji konfigurowania edytora dla Gita.
-

Sprawdzanie ustawień

Jeżeli chcesz skontrolować swoje ustawienia, możesz użyć polecenia `git config --list`, aby wypisać wszystkie zapisy konfiguracyjne, które są w tym momencie dostępne dla Gita:

```
$ git config --list
user.name=Adam Nowak
user.email=adamnowak@przyklad.pl
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Niektóre klucze mogą pojawiać się wielokrotnie, ponieważ Git odczytuje wartości tych kluczy z różnych plików (na przykład z plików `/etc/gitconfig` i `~/.gitconfig`). W takich przypadkach Git będzie używał wartości z ostatniego odczytanego pliku.

Możesz też sprawdzić, jakiej wartości wybranego klucza używa aktualnie, wprowadzając polecenie `git config <klucz>`:

```
$ git config user.name
Adam Nowak
```

Szukanie pomocy

Jeżeli podczas używania Gita będzie Ci potrzebna pomoc, możesz na trzy różne sposoby wywołać stronę podręcznika `man` opisującą wybrane polecenie Gita:

```
$ git help <polecenie>  
$ git <polecenie> --help  
$ man git-<polecenie>
```

Aby na przykład wyświetlić stronę pomocy dla polecenia `config`, możesz wpisać:

```
$ git help config
```

To bardzo przydatne polecenia, ponieważ możesz z nich korzystać w dowolnym momencie, nawet bez połączenia z internetem. Jeżeli strony podręcznika `man` i ta książka okażą się niewystarczające i będziesz potrzebować rozmowy z ekspertem lub innym użytkownikiem, możesz skorzystać z kanałów IRC `#git` lub `#github` dostępnych na serwerze Freenode (irc.freenode.net). Na tych kanałach zwykle dyskutują setki osób, które zapewne doskonale wiedzą, jak używać Gita i chętnie pomogą Ci rozwiązać Twój problem.

Podsumowanie

Teraz znasz już podstawowe zasady działania Gita i wiesz, czym różni się od scentralizowanych systemów kontroli wersji, z których korzystałeś wcześniej. W Twoim systemie powinna być już zainstalowana działająca wersja Gita, w której zdefiniowałeś Twoją tożsamość. Nadszedł czas, żeby nauczyć się wykonywania podstawowych operacji w Gicie.

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 

GitHub: odkryj świat rozproszonej kontroli wersji!

W wypadku pracy zespołowej efektywny system kontroli wersji jest warunkiem sukcesu. Powszechnie stosowanym w tym zakresie rozwiązaniem jest Git, który umożliwia zarządzanie wersjami projektu, przeglądanie wszystkich zmian, a także przywracanie pliku do dowolnej wcześniejszej wersji. Git jest dynamicznie rozwijanym oprogramowaniem. W ostatnich latach wzbogacił się o wiele przydatnych narzędzi, wyjątkowe stały się też możliwości jego konfiguracji i dostosowania. Wciąż jednak pełne wykorzystanie potencjału Gita wymaga wprawy i dobrej znajomości specyfiki pracy tego narzędzia.

To drugie, zaktualizowane i uzupełnione wydanie popularnego podręcznika opracowanego przez specjalistów dla profesjonalistów. Dowiesz się z niego, czym jest rozproszona kontrola wersji i jakie daje korzyści. Poznasz poszczególne funkcjonalności Gita i nauczysz się je rozbudowywać tak, aby zaspokajały nawet najbardziej wyrafinowane potrzeby. W książce znalazły się również wyczerpujące opisy funkcji dostępnych w Git 2.0, a także omówienie systemu usług GitHub i jego zalet. Istotną zmianą w nowym wydaniu jest uwzględnienie w znaczącej większości prezentowanych przykładów protokołu HTTP w miejsce SSH — co stanowi odpowiedź na rosnącą popularność HTTP.

W książce między innymi:

- rodzaje systemów kontroli wersji i podstawy pracy z Gitem
- konfiguracja Gita i rozproszone metody pracy
- praca w zespole z systemem GitHub
- konflikty scalania i ich rozstrzyganie
- dostosowywanie Gita do szczególnych potrzeb
- Git i inne systemy, w tym Subversion, Mercurial i Perforce

Scott Chacon jest programistą, współzałożycielem serwisu GitHub i jednym z prowadzących stronę główną Gita (git-scm.com). Występował na wielu prestiżowych konferencjach na całym świecie i przeprowadził setki korporacyjnych szkoleń dla użytkowników Gita.

Ben Straub jest programistą, współtwórcą biblioteki libgit2 i międzynarodowym popularyzatorem Gita, a także nienasyconym czytelnikiem, odkrywcą i pasjonatem sztuki tworzenia doskonałego oprogramowania.

Helion	KOD KORZYŚCI Sięgnij po więcej! ▶	
 helion.pl	ISBN 978-83-283-8731-7	
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 250 98 63 helion@helion.pl	 9 788328 387317	
Cena: 109,00 zł		

Apress