O'REILLY®

Wydanie III

Kontrola wersji z systemem Git

Zaawansowane narzędzia i techniki do wspólnego projektowania oprogramowania

Prem Kumar Ponuthorai
Jon Loeliger

Tytuł oryginału: Version Control with Git: Powerful Tools and Techniques
for Collaborative Software Development, 3rd Edition

Tłumaczenie: Piotr Pilch

ISBN: 978-83-8322-647-7

© 2023 Helion S.A.

Authorized Polish translation of the English edition of *Version Control with Git, 3rd Edition*
ISBN 9781492091196 © 2023 Prem Kumar Ponuthorai.

This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by
any means, electronic or mechanical, including photocopying, recording or by any information
storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej
publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną,
fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym
powoduje naruszenie praw autorskich niniejszej publikacji.

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

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

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)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/konwe3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Przedmowa 11

CZĘŚĆ I. Filozofia działania systemu Git 17

1. Wprowadzenie do systemu Git 19
Komponenty systemu Git 19
Właściwości systemu Git 21
Wiersz poleceń systemu Git 22
Krótkie wprowadzenie do obsługi systemu Git 25
Przygotowanie do pracy z systemem Git 25
Korzystanie z repozytorium lokalnego 26
Korzystanie z repozytorium współużytkowanego 35
Pliki konfiguracyjne 36
Podsumowanie 39

2. Podstawowe zagadnienia 40
Repozytoria 40
Magazyn obiektów systemu Git 41
Indeks 43
Baza danych adresowana zawartością 44
Git monitoruje zawartość 45
Ścieżka nazw i zawartość 46
Pliki spakowane 47
Wizualizacja magazynu obiektów systemu Git 48
Wewnętrzne mechanizmy systemu Git: zagadnienia w praktyce 50
W obrębie katalogu .git 51
Obiekty blob i wartości mieszające 53
Obiekt drzewa i pliki 54
Uwaga dotycząca użycia algorytmu SHA1 w systemie Git 55

Kup ksi k Pole ksi k

Hierarchie drzew 57
Obiekty rewizji 58
Obiekty tagów 60
Podsumowanie 61

CZĘŚĆ II. Fundamenty systemu Git 63

3. Gałęzie ... 69
Powody stosowania gałęzi w systemie Git 69
Wytyczne dotyczące rozgałęziania 70
Nazwy gałęzi 70
Nazwy gałęzi — dozwolone i niedozwolone rzeczy 71
Zarządzanie gałęziami 72
Praca w obrębie gałęzi 72
Tworzenie gałęzi 75
Wyświetlanie listy nazw gałęzi 76
Wyświetlanie gałęzi oraz ich rewizji 76
Przełączanie gałęzi 79
Scalanie zmian z inną gałęzią 83
Tworzenie nowej gałęzi i przełączanie do niej 85
Odłączony wskaźnik HEAD 86
Usuwanie gałęzi 88
Podsumowanie 91

4. Rewizje ... 92
Rewizje: rejestrowane jednostki zmian 92
Atomowe zestawy zmian 93
Identyfikowanie rewizji 94
Bezwzględne nazwy rewizji 94
Odwołania i odwołania symboliczne 95
Względne nazwy rewizji 97
Historia rewizji 100
Wyświetlanie starych rewizji 100
Grafy rewizji 102
Zakresy rewizji 108
Podsumowanie 113

5. Zarządzanie plikami i indeks ... 114
Znaczenie indeksu 114
Klasyfikacje plików w systemie Git 116
Użycie polecenia git add 118

Kup książkę Poleć książkę

Uwagi dotyczące stosowania polecenia git commit 122
Użycie polecenia git commit --all 122
Tworzenie komunikatów dziennika rewizji 123
Użycie polecenia git rm 124
Użycie polecenia git mv 126
Uwaga dotycząca monitorowania operacji zmiany nazwy 128
Plik .gitignore 128
Podsumowanie 132

6. Scalenia 133
Scalanie: techniczny punkt widzenia 133
Przykłady scalania 134
Przygotowanie do operacji scalania 134
Scalanie dwóch gałęzi 134
Scalanie powodujące konflikt 137
Zajmowanie się konfliktami podczas scalania 140
Lokalizowanie plików powodujących konflikt 141
Inspekcja konfliktów 142
Jak system Git monitoruje konflikty? 147
Finalizowanie rozwiązywania konfliktu 149
Przerywanie lub ponawianie scalania 150
Strategie scalania 151
Scalenia zdegenerowane 154
Zwykłe scalenia 157
Scalenia specjalistyczne 159
Stosowanie strategii scalania 160
Sterowniki scalania 161
Sposób traktowania scaleń przez system Git 162
Scalenia i model obiektów systemu Git 162
Scalenia skoncentrowane 163
Dlaczego po prostu nie można scalać każdej zmiany po kolei? 164
Podsumowanie 164

7. Różnice 166
Postaci polecenia git diff 168
Prosty przykład użycia polecenia git diff 172
Analiza danych wyjściowych polecenia git diff 175
Polecenie git diff i zakresy rewizji 177
Użycie polecenia git diff z ograniczaniem ścieżki 179
Jak system Git ustala różnice? 181
Podsumowanie 182

Kup ksi k Pole ksi k

CZĘŚĆ III. Umiejętności na poziomie średnio zaawansowanym 183

8. **Znajdowanie rewizji** **185**
Użycie polecenia git bisect 185
Użycie polecenia git blame 191
Zastosowanie „kilofa” 192
Podsumowanie 193

9. **Modyfikowanie rewizji** **194**
Filozofia modyfikowania historii rewizji 195
Ostrzeżenie dotyczące modyfikowania historii 196
Użycie polecenia git revert 198
Modyfikowanie rewizji HEAD 199
Użycie polecenia git reset 201
Użycie polecenia git cherry-pick 209
Polecenia reset, revert i checkout 212
Zmiana bazy rewizji 213
Użycie polecenia git rebase -i 215
Porównanie operacji zmiany bazy i scalania 219
Podsumowanie 224

10. **Schowek i dziennik odwołań** **225**
Schowek 225
Przypadek użycia: przerwany przepływ pracy 225
Przypadek użycia: aktualizowanie efektów trwających prac lokalnych za pomocą zmian źródłowych 231
Przypadek użycia: przekształcanie zmian w schowku do postaci gałęzi 233
Dziennik odwołań 236
Podsumowanie 240

11. **Repozytoria zdalne** **242**
Część I: zagadnienia dotyczące repozytoriów 243
Repozytoria czyste i projektowe 243
Klony repozytoriów 245
Elementy zdalne 246
Gałęzie monitorujące 247
Odwoływanie się do innych repozytoriów 249
Odwoływanie się do repozytoriów zdalnych 249
Specyfikacja odwołań 251
Część II: przykład zastosowania repozytoriów zdalnych 254
Tworzenie autorytatywnego repozytorium 254
Tworzenie własnego elementu zdalnego origin 256

Kup ksi k Pole ksi k

Prace projektowe we własnym repozytorium 258
Przekazywanie dokonanych zmian 259
Dodawanie nowego projektanta 260
Uzyskiwanie aktualizacji repozytorium 262
Część III: zobrazowanie cyklu projektowego z użyciem repozytoriów zdalnych 268
Klonowanie repozytorium 268
Alternatywne historie 269
Operacje umieszczania bez „przewijania” 270
Uzyskiwanie alternatywnej historii 271
Scalanie historii 272
Konflikty podczas scalania 273
Przekazywanie scalonej historii 273
Część IV: konfiguracja elementu zdalnego 274
Zastosowanie polecenia git remote 275
Użycie polecenia git config 276
Ręczne edytowanie 277
Część V: użycie gałęzi monitorujących 277
Tworzenie gałęzi monitorujących 277
Z przodu i z tyłu 280
Dodawanie i usuwanie gałęzi zdalnych 282
Repozytoria czyste i polecenie git push 283
Podsumowanie 284

12. Zarządzanie repozytoriami ... 286
Publikowanie repozytoriów 287
Repozytoria z kontrolowanym dostępem 287
Repozytoria z anonimowym dostępem w trybie odczytu 289
Repozytoria z anonimowym dostępem w trybie zapisu 293
Rada dotycząca publikowania repozytoriów 293
Struktura repozytorium 294
Struktura repozytorium współużytkowanego 294
Struktura repozytorium rozproszonego 295
Funkcjonowanie w ramach procesu projektowania rozproszonego 296
Modyfikowanie publicznie dostępnej historii 296
Osobne kroki zatwierdzania i publikowania 297
Nie ma jednej prawdziwej historii 298
Znajomość własnego miejsca 299
Przepływy z repozytoriami źródłowymi i podrzędnymi 299
Role osób utrzymujących i projektantów 300
Interakcja między osobą utrzymującą a projektantem 301
Dualność roli 301

Kup ksi k Pole ksi k

Korzystanie z wielu repozytoriów 303
Twój własny obszar roboczy 303
Gdzie zainicjować swoje repozytorium? 304
Przekształcanie do postaci innego repozytorium źródłowego 305
Zastosowanie wielu repozytoriów źródłowych 307
Rozwidlanie projektów 309
Podsumowanie 311

CZĘŚĆ IV. Umiejętności na poziomie zaawansowanym 313

13. Poprawki 315
Dlaczego warto stosować poprawki? 316
Generowanie poprawek 317
Poprawki i sortowanie topologiczne 325
Wysyłanie poprawek w wiadomości pocztowej 326
Zastosowanie poprawek 329
Złe poprawki 336
Porównanie scalania i stosowania poprawek 337
Podsumowanie 338

14. „Haki” 339
Typy „haków” 339
Uwaga dotycząca stosowania „haków” 340
Instalowanie „haków” 341
Przykładowe „haki” 342
Tworzenie pierwszego „haka” 343
Dostępne „haki” 345
„Haki” powiązane z zatwierdzaniem 345
„Haki” powiązane z poprawkami 347
„Haki” powiązane z poleceniem push 347
Inne „haki” repozytorium lokalnego 349
Używać „haka” czy nie? 349
Podsumowanie 350

15. Podmoduły 351
Łącza gitlink 351
Podmoduły 354
Dlaczego podmoduły? 355
Korzystanie z podmodułów 355
Podmoduły i ponowne użycie danych uwierzytelniających 363

Kup książkę Poleć książkę

Poddrzewa systemu Git 364
Dodawanie projektu podrzędnego 365
Uzyskiwanie aktualizacji projektu podrzędnego 366
Modyfikowanie projektu podrzędnego w obrębie superprojektu 367
Porównanie wizualne poddrzew i podmodułów systemu Git 368
Podsumowanie 369

16. Zaawansowane modyfikacje 371
Interaktywne umieszczanie „kawałków” w przechowalni 371
„Czułe” polecenie git rev-list 382
Przełączanie oparte na dacie 382
Uzyskiwanie starej wersji pliku 385
Odzyskiwanie utraconej rewizji 387
Polecenie git fsck 387
Ponowne łączenie z utraconą rewizją 391
Użycie polecenia git filter-repo 392
Przykłady użycia polecenia git filter-repo 393
Podsumowanie 399

CZĘŚĆ V. Wskazówki i porady 401

17. Wskazówki, porady i techniki 403
Interaktywna zmiana bazy z użyciem nieuporządkowanego katalogu roboczego 403
Proces oczyszczania 404
Wskazówki dotyczące przywracania rewizji 407
Cofanie efektów operacji zmiany bazy w przypadku repozytorium źródłowego 407
Szybki przegląd zmian 409
Porządkowanie 410
Użycie polecenia git-grep do przeszukiwania repozytorium 410
Aktualizowanie i usuwanie odwołań 413
Monitorowanie przeniesionych plików 413
Czy byłeś tutaj wcześniej? 414
Migracja do systemu Git 415
Migracja z systemu kontroli wersji Git 416
Migracja z systemu kontroli wersji innego niż Git 419
Uwaga dotycząca korzystania z dużych repozytoriów 423
Git LFS 425
Repozytorium przed zastosowaniem rozszerzenia Git LFS i po nim 426
Instalowanie rozszerzenia Git LFS 427

Kup ksi k Pole ksi k

Monitorowanie dużych obiektów za pomocą rozszerzenia Git LFS 429
Przydatne techniki związane z rozszerzeniem Git LFS 432
Przekształcanie istniejących repozytoriów w celu użycia rozszerzenia Git LFS 434
Podsumowanie 436

18. Git i serwis GitHub ... 437
Coś o serwisie GitHub 438
Typy kont serwisu GitHub 439
Serwis GitHub w ekosystemie systemu Git 442
Udostępnianie repozytorium w serwisie GitHub 446
Widok repozytorium 448
Kod 451
Problemy 454
Żądania pobrania 456
Przepływ pracy GitHub Flow 465
Rozwiązywanie konfliktów podczas scalania w serwisie GitHub 469
Przepływy pracy procesu projektowania 475
Integracja z serwisem GitHub 478
Podsumowanie 481

A Historia systemu Git ... 483

B Instalacja systemu Git ... 490

Kup ksi k Pole ksi k

ROZDZIAŁ 1.
Wprowadzenie do systemu Git

Mówiąc wprost, system Git służy do monitorowania zawartości. Mając to na względzie, Git korzysta ze wspólnych zasad większości systemów kontroli wersji. To, co jednak decyduje o unikalności systemu Git wśród różnych dostępnych obecnie narzędzi, to fakt, że jest on rozproszonym systemem kontroli wersji. Różnica ta oznacza, że Git jest szybki i skalowalny, oferuje bogatą kolekcję zestawów poleceń zapewniających dostęp zarówno do operacji wysokiego poziomu, jak i operacji niskopoziomowych, a ponadto jest zoptymalizowany pod kątem operacji lokalnych.

W rozdziale poznasz fundamentalne zasady systemu Git, jego właściwości oraz podstawowe polecenia `git`. Poza tym zaznajomisz się z wybranymi, ogólnymi wytycznymi dotyczącymi tworzenia i dodawania zmian do repozytorium.

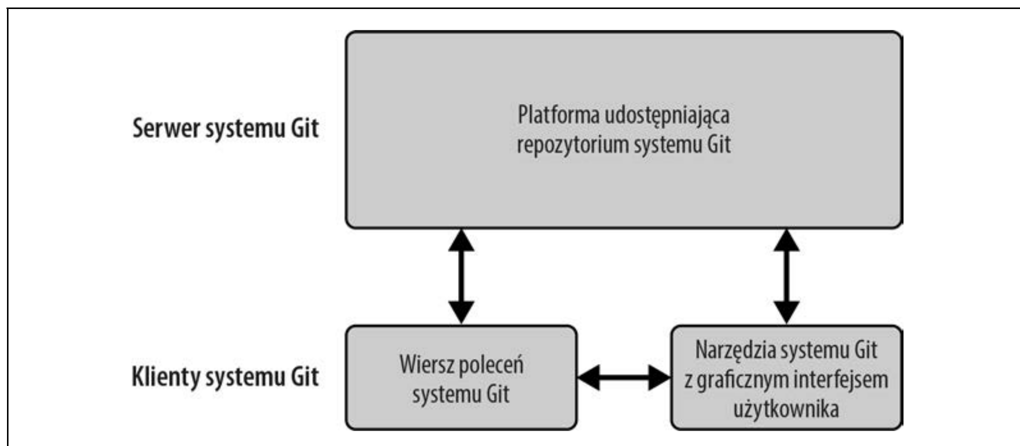
Szczególnie zalecamy poświęcić czas na opanowanie objaśnionych tutaj ważnych pojęć. Są to elementy konstrukcyjne systemu Git, które ułatwią zrozumienie średnio zaawansowanych i zaawansowanych technik zarządzania repozytorium Git w ramach codziennej pracy. Zaprezentowane fundamentalne pojęcia okażą się też pomocne w zwiększeniu tempa przyswajania wiedzy, gdy będziemy analizować wewnętrzne mechanizmy działania systemu Git w rozdziałach zebranych w części II („Podstawy systemu Git”), części III („Umiejętności na poziomie średnio zaawansowanym”) i części IV („Zaawansowane umiejętności”) książki.

Komponenty systemu Git

Zanim zagłębimy się w świat poleceń `git`, cofnijmy się o jeden krok i dokonajmy przeglądu komponentów tworzących ekosystem Git. Na rysunku 1.1 pokazano, w jaki sposób komponenty ze sobą współpracują.

Narzędzia systemu Git z graficznym interfejsem użytkownika pełnią rolę interfejsu wiersza poleceń Git, a niektóre z nich oferują rozszerzenia zapewniające integrację z popularnymi platformami udostępniającymi repozytorium systemu Git. Narzędzia klientów Git korzystają przeważnie z lokalnej kopii używanego repozytorium.

Kup ksi k Pole ksi k



Rysunek 1.1. Przegląd komponentów systemu Git

Gdy używasz systemu Git, typowa konfiguracja obejmuje jego serwer i klienty. Możesz zrezygnować z serwera, ale zwiększy to stopień złożoności metod utrzymywania repozytoriów i zarządzania nimi w przypadku udostępniania zmian rewizji w wariancie pracy grupowej, a ponadto utrudni zapewnienie spójności (powrócimy do tego w rozdziale 11.). Serwer i klienty systemu Git działają w następujący sposób:

Serwer Git

Serwer Git sprawia, że łatwiejsza staje się współpraca, ponieważ zapewnia on dostępność centralnego i pewnego „źródła prawdy" w odniesieniu do repozytoriów, z których będziesz korzystać. Taki serwer jest też miejscem, w którym są przechowywane zdalne repozytoria systemu Git. Zgodnie z powszechną praktyką repozytorium zawiera najbardziej aktualny i stabilny kod źródłowy realizowanych projektów. Masz możliwość zainstalować i skonfigurować własny serwer Git albo zrezygnować z tych działań i wybrać opcję udostępniania repozytoriów Git w obrębie zaufanych zewnętrznych witryn hostingowych, takich jak GitHub, GitLab i Bitbucket.

Klienty Git

Klienty systemu Git prowadzą interakcję z lokalnymi repozytoriami. Masz możliwość komunikacji z tymi klientami za pośrednictwem wiersza poleceń systemu Git lub jego narzędzi z graficznym interfejsem użytkownika. Po zainstalowaniu i skonfigurowaniu klienta Git będziesz w stanie uzyskać dostęp do zdalnych repozytoriów, używać lokalnej kopii repozytorium oraz kierować zmiany z powrotem do serwera Git. Jeśli dopiero zaczynasz poznawać system Git, zalecamy rozpoczęcie od wiersza poleceń. Zaznajom się najpierw z typowym podzbiorem poleceń `git`, niezbędnych do realizowania codziennych operacji, a następnie zacznij korzystać z wybranego narzędzia systemu Git z graficznym interfejsem użytkownika.

Zdecydowaliśmy się na takie podejście w pewnym stopniu dlatego, że narzędzia z graficznym interfejsem użytkownika oferują zwykle terminologię reprezentującą żądany wynik, który może nie być częścią standardowych poleceń systemu Git. Przykładem jest narzędzie z opcją `sync`, która

Kup ksi k Pole ksi k

powoduje maskowanie realizowanego w tle łączenia dwóch lub większej liczby poleceń `git` w celu osiągnięcia wymaganego rezultatu. Jeżeli z jakiegoś powodu użyłbyś w wierszu poleceń polecenia podrzędnego `sync`, możesz uzyskać następujące niejasne dane wynikowe:

```
$ git sync
git: 'sync' is not a git command. See 'git --help'.
The most similar command is
       svn
```

`git sync` nie jest poprawnym poleceniem podrzędnym polecenia `git`. Aby zapewnić, że lokalna kopia robocza repozytorium jest zsynchronizowana ze zmianami ze zdalnego repozytorium systemu Git, konieczne będzie użycie kombinacji następujących poleceń: `git fetch`, `git merge` oraz `git pull` lub `git push`.

Do dyspozycji jest udostępnionych mnóstwo narzędzi. Niektóre narzędzia systemu Git z graficznym interfejsem użytkownika są rozbudowane i mogą być rozszerzane z wykorzystaniem modelu dodatków, który zapewnia opcję łączenia ze sobą i stosowania składników udostępnionych w obrębie popularnych witryn hostingowych systemu Git, oferowanych przez zewnętrznych dostawców. Choć wygodnym rozwiązaniem może być poznawanie Git za pomocą narzędzia z graficznym interfejsem użytkownika, z myślą o przykładach i omówieniach kodu będziemy koncentrować się na narzędziu wiersza poleceń. Dzięki temu zapewnisz sobie solidną podstawową wiedzę i nabierzesz zręczności w korzystaniu z systemu Git.

Właściwości systemu Git

Po dokonaniu przeglądu komponentów systemu Git dowiedzmy się czegoś o jego właściwościach. Zrozumienie cech wyróżniających ten system pozwoli bez większego trudu przestawić się z rozumowania kategoriami scentralizowanej kontroli wersji na postrzeganie jej w wariancie rozproszonym. Lubimy określać to mianem „myślenia zgodnie z filozofią systemu Git”.

System Git przechowuje zmiany wersji jako migawki

Zdecydowanie pierwszym pojęciem do poznania jest to, w jaki sposób system Git przechowuje wiele wersji pliku, którym się zajmujesz. W przeciwieństwie do innych systemów kontroli wersji system Git nie monitoruje zmian wersji jako zestawu modyfikacji, powszechnie nazywanych **różnicami**. Git tworzy migawkę (ang. *snapshot*) zmian dotyczących stanu repozytorium w konkretnym momencie. W terminologii systemu Git jest to określane mianem **rewizji** (ang. *commit*). Potraktuj to, jak utrwalanie danej chwili, podobnie jak w fotografii.

System Git umożliwia projektowanie lokalne

W systemie Git korzystasz z kopii repozytorium znajdującego się na Twoim komputerze używanym do projektowania lokalnego. Kopia jest nazywana **repozytorium lokalnym** lub klonem repozytorium zdalnego zlokalizowanego na serwerze Git. Repozytorium lokalne będzie zapewniać w całości w jednym miejscu zasoby i migawki zmian wersji dokonanych dla tych zasobów. W systemie Git takie zbiory powiązanych migawek są określane mianem **historii rewizji repozytorium** lub w skrócie **historią repozytorium**. Umożliwia to pracę w środowisku bez nawiązanego połączenia, ponieważ Git nie wymaga stałego połączenia ze swoim

Kup ksi k Pole ksi k

serwerem, aby wprowadzone przez Ciebie zmiany objąć zakresem kontroli wersji. Naturalną konsekwencją tego jest możliwość uczestniczenia w dużych i złożonych projektach, realizowanych przez rozproszone zespoły bez negatywnego wpływu na efektywność i wydajność operacji kontroli wersji.

System Git jest autorytatywny

Określenie *autorytatywny* oznacza, że polecenia `git` są jednoznaczne. System Git oczekuje, aż dostarczysz mu instrukcje odnośnie do tego, co ma zrealizować i kiedy. Przykładowo, Git nie synchronizuje automatycznie zmian w repozytorium lokalnym z repozytorium zdalnym ani nie zapisuje samoczynnie migawki wersji w historii repozytorium lokalnego. Każde działanie wymaga użycia przez Ciebie wyraźnego polecenia lub instrukcji informującej system Git, co jest wymagane. Obejmuje to dodawanie nowych rewizji, poprawianie tych już istniejących, wypychanie zmian z repozytorium lokalnego do repozytorium zdalnego, a nawet uzyskiwanie z niego nowych zmian. Podsumowując, podejmowane działania muszą być świadome. Dotyczy to również informowania systemu Git o tym, jakie pliki zamierzasz monitorować, ponieważ nie obejmuje on automatycznie nowych plików zakresem kontroli wersji.

System Git zaprojektowano z myślą o wspieraniu projektowania nieliniowego

Git umożliwia wyobrażanie sobie opcji i eksperymentowanie z różnymi ich implementacjami w celu uzyskania realnych rozwiązań w ramach realizowanego projektu. Dzięki temu możesz się odłączyć i pracować równolegle względem głównej, stabilnej bazy kodu projektu. Taka metodologia, nazywana **rozgałęzianiem** (ang. *branching*), to bardzo rozpowszechniona praktyka zapewniająca integralność głównej linii projektowej, a ponadto zapobiegająca jakimkolwiek przypadkowym zmianom, które mogłyby jej zaszkodzić.

W systemie Git pojęcie rozgałęziania jest uważane za coś prostego i mało kosztownego, ponieważ w jego przypadku gałąź to tylko wskaźnik do najnowszej rewizji w zestawie połączonych rewizji. Dla każdej utworzonej gałęzi Git monitoruje zestaw rewizji dla danej gałęzi. Masz możliwość lokalnego przełączania się między gałęziami. Git przywraca następnie stan projektu do chwili, gdy została utworzona najbardziej aktualna migawka konkretnej gałęzi. Gdy postanowisz dokonać scalenia zmian z dowolnej gałęzi z główną linią projektową, system Git będzie w stanie połączyć zestawy rewizji dzięki zastosowaniu technik omawianych w rozdziale 6.

Ze względu na to, że Git oferuje wiele nowości, pamiętaj, że pojęcia i praktyki powiązane z innymi systemami kontroli wersji mogą się różnić lub wcale nie być możliwe do zastosowania w systemie Git.

Wiersz poleceń systemu Git

Interfejs wiersza poleceń systemu Git jest prosty w użyciu. Zaprojektowano go tak, byś miał pełną kontrolę nad swoim repozytorium. W związku z tym istnieje wiele sposobów na zrealizowanie tej samej rzeczy. Skupiając się na tym, jakie polecenia są ważne w Twojej codziennej pracy, możesz sobie ułatwić korzystanie z nich i dogłębniej się z nimi zaznajomić.

Kup ksi k Pole ksi k

Na początek wpisz po prostu polecenie **git version** lub **git --version**, aby stwierdzić, czy na używanym komputerze został już załadowany system Git. Powinieneś uzyskać następujące dane wynikowe:

```
$ git --version
git version 2.37.0
```

Jeśli nie zainstalowałeś na komputerze systemu Git, przed przejściem do następnego podrozdziału zajrzyj do dodatku B w celu uzyskania informacji, jak przeprowadzić instalację Git odpowiednio do posiadanej platformy systemu operacyjnego.

Po dokonaniu instalacji wpisz polecenie **git** bez żadnych argumentów. System Git wyświetli jego opcje oraz najpowszechniej używane polecenia podrzędne:

```
$ git
usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [--super-prefix=<path>] [--config-env=<name>=<envvar>]
           <command> [<args>]
These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   restore    Restore working tree files
   rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   diff       Show changes between commits, commit and working tree, etc
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status

grow, mark and tweak your common history
   branch     List, create, or delete branches
   commit     Record changes to the repository
   merge      Join two or more development histories together
   rebase     Reapply commits on top of another base tip
   reset      Reset current HEAD to the specified state
   switch     Switch branches
   tag        Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
   fetch      Download objects and refs from another repository
   pull       Fetch from and integrate with another repository or a local branch
   push       Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some((("git help command")))
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.
```

Aby uzyskać kompletną listę poleceń podrzędnych polecenia `git`, wpisz **`git help --all`**.

Na podstawie informacji o możliwościach użycia polecenia możesz stwierdzić, że z poleceniem `git` jest powiązana niewielka liczba opcji. Większość opcji reprezentowanych w opisie polecenia przez łańcuch `[<args>]` ma zastosowanie w przypadku konkretnych poleceń podrzędnych.

Przykładowo, opcja `--version` dotyczy polecenia `git` i zapewnia numer wersji:

```
$ git --version
git version 2.37.0
```

Z kolei opcja `--amend` to przykład opcji powiązanej z poleceniem podrzędnym `commit` polecenia `git`:

```
$ git commit --amend
```

Niektóre wywołania wymagają użycia obu form opcji (dodatkowe spacje wstawione w poniższym przykładzie wiersza poleceń służą wyłącznie do wizualnego oddzielenia polecenia podrzędnego od polecenia podstawowego i nie są wymagane):

```
$ git --git-dir=projekt.git    repack -d
```

Dla wygody dokumentacja każdego polecenia podrzędnego polecenia `git` jest dostępna po wpisaniu jednego z następujących poleceń: `git help` *`polecenie_podrzędne`*, `git --help` *`polecenie_podrzędne`*, `git` *`polecenie_podrzędne`* `--help` lub `man git-`*`polecenie_podrzędne`*.

Kompletna dokumentacja systemu Git jest dostępna w internecie pod adresem *https://mirrors.edge.kernel.org/pub/software/scm/git/docs/*.

Dotychczas system Git dostarczano jako zestaw wielu prostych, odrębnych i autonomicznych poleceń stworzonych zgodnie z filozofią systemu Unix, sprowadzającą się do budowania niewielkich, współdziałających ze sobą narzędzi. Każde polecenie miało nazwę zawierającą znak - (np. `git-commit` i `git-log`). W nowoczesnych instalacjach tego systemu nie są już jednak obsługiwane polecenia o nazwie zawierającej znak -. Zamiast tego korzysta się z pojedynczego programu `git` z określonym poleceniem podrzędnym.

Polecenia `git` rozpoznają zarówno „krótkie”, jak i „długie” opcje. Przykładowo, następujące polecenia `git commit` zadziałają jednakowo:

```
$ git commit -m "Popraw literówkę."
$ git commit --message="Popraw literówkę."
```

W przykładzie krótka forma `-m` uwzględnia jeden łącznik, natomiast długa forma `--message` zawiera dwa takie znaki (jest to spójne z rozszerzeniem długich opcji systemu GNU). Niektóre opcje istnieją wyłącznie w jednej postaci.

Używając wiele razy opcji -m, możesz utworzyć podsumowanie zatwierdzania i powiązany z tym szczegółowy komunikat:

```
$ git commit -m "Podsumowanie" -m "Szczegóły podsumowania"
```

I wreszcie możesz oddzielić opcje od listy argumentów z wykorzystaniem **konwencji polegającej na użyciu wyłącznie dwóch łączników**. Użyj ich na przykład w celu odróżnienia części sterującej wiersza poleceń od listy argumentów, takich jak nazwy plików:

```
$ git diff -w main origin -- tools/Makefile
```

Może być konieczne zastosowanie dwóch łączników w celu oddzielenia i wyraźnego zidentyfikowania nazw plików, aby nie zostały pomylone z inną częścią polecenia. Jeśli na przykład akurat użyto zarówno pliku, jak i tagu o nazwie *main.c*, konieczne będzie sprecyzowanie celu realizowanych operacji:

```
# Przełączenie na tag o nazwie "main.c"
$ git checkout main.c

# Przełączenie na plik o nazwie "main.c"
$ git checkout -- main.c
```

Krótkie wprowadzenie do obsługi systemu Git

Aby się przekonać, jak działa system Git, możesz utworzyć nowe repozytorium, dodać do niego pewną zawartość i monitorować kilka wersji. Repozytorium możesz utworzyć na dwa sposoby: albo wygenerować je od podstaw i wypełnić zawartością, albo skorzystać z istniejącego repozytorium przez **sklonowanie** go ze zdalnego serwera systemu Git.

Przygotowanie do pracy z systemem Git

Niezależnie od tego, czy tworzysz nowe repozytorium, czy używasz już istniejącego, po zainstalowaniu systemu Git na swoim lokalnym komputerze służącym do projektowania musisz przeprowadzić podstawowe, wstępne działania konfiguracyjne. Przypomina to ustawianie poprawnej daty, strefy czasowej i języka w przypadku nowego aparatu przed wykonaniem pierwszego zdjęcia.

Jako absolutnego minimum system Git wymaga podania swoich personaliów i adresu e-mail przed utworzeniem pierwszej rewizji w repozytorium. Podane dane określające tożsamość są później używane do identyfikacji **twórcy** rewizji wraz z innymi metadanymi migawki. Dane ustalające tożsamość możesz zapisać w pliku konfiguracyjnym za pomocą polecenia `git config`:

```
$ git config user.name "Jan Nowak"
$ git config user.email "jn@przyklad.pl"
```

Jeśli postanowisz nie uwzględniać danych tożsamości w pliku konfiguracyjnym, konieczne będzie określenie jej dla każdego polecenia podrzędnego `git commit` przez dołączenie na jego końcu argumentu `--author`:

```
$ git commit -m "komunikat dziennika" --author="Jan Nowak <jn@przyklad.pl>"
```

Pamiętaj, że jest to utrudniony wariant, który może szybko stać się uciążliwy.

Kup ksi k   Pole ksi k

Masz również możliwość określenia swojej tożsamości przez podanie personaliów i adresu e-mail odpowiednio z użyciem zmiennych środowiskowych `GIT_AUTHOR_NAME` i `GIT_AUTHOR_EMAIL`. Jeżeli te zmienne zostaną ustawione, nadpiszą one wszystkie ustawienia konfiguracji. W przypadku jednak specyfikacji określanych w wierszu poleceń system Git nadpisze wartości podane w pliku konfiguracyjnym i zmiennej środowiskowej.

Korzystanie z repozytorium lokalnego

Po określeniu danych tożsamości możesz zacząć korzystać z repozytorium. Rozpocznij od utworzenia nowego, pustego repozytorium na swoim lokalnym komputerze używanym do projektowania. Zaczniemy od czegoś prostego, a później będziemy zmierzać w stronę technik umożliwiających pracę z repozytorium współużytkowanym serwera Git.

Tworzenie początkowego repozytorium

Poprzez utworzenie repozytorium osobistej witryny internetowej przeprowadzimy modelowanie typowej sytuacji. Załóżmy, że zaczynasz od podstaw i zamierzasz umieścić zawartość swojego projektu w katalogu lokalnym *~/moja_witryna*, który dodajesz do repozytorium systemu Git.

Użyj następujących poleceń, aby utworzyć katalog i umieścić podstawową zawartość w pliku o nazwie *index.html*:

```
$ mkdir ~/moja_witryna
$ cd ~/moja_witryna
$ echo 'Moja wspaniała witryna!' > index.html
```

W celu przekształcenia katalogu *~/moja_witryna* w repozytorium Git uruchom polecenie `git init`. W tym przypadku zastosowano opcję `-b`, po której podano nazwę `main` gałęzi domyślnej:

```
$ git init -b main
Initialized empty Git repository in ../moja_witryna/.git/
```

Jeśli wolisz najpierw zainicjalizować puste repozytorium Git, a następnie dodać do niego pliki, możesz w tym celu uruchomić następujące polecenia:

```
$ git init -b main ~/moja_witryna

Initialized empty Git repository in ../moja_witryna/.git/

$ cd ~/moja_witryna
$ echo 'Moja wspaniała witryna!' > index.html
```

Możliwe jest zainicjalizowanie całkowicie pustego katalogu lub istniejącego katalogu wypełnionego plikami. W obu wariantach proces przekształcania katalogu w repozytorium systemu Git przebiega identycznie.

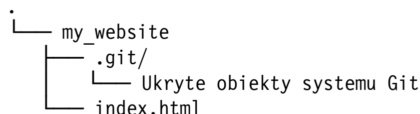
Polecenie `git init` tworzy ukryty katalog o nazwie *.git* na najwyższym poziomie projektu. Wszystkie informacje dotyczące rewizji wraz z dodatkowymi metadanymi, a także rozszerzenia systemu Git są przechowywane w tym katalogu najwyższego poziomu.

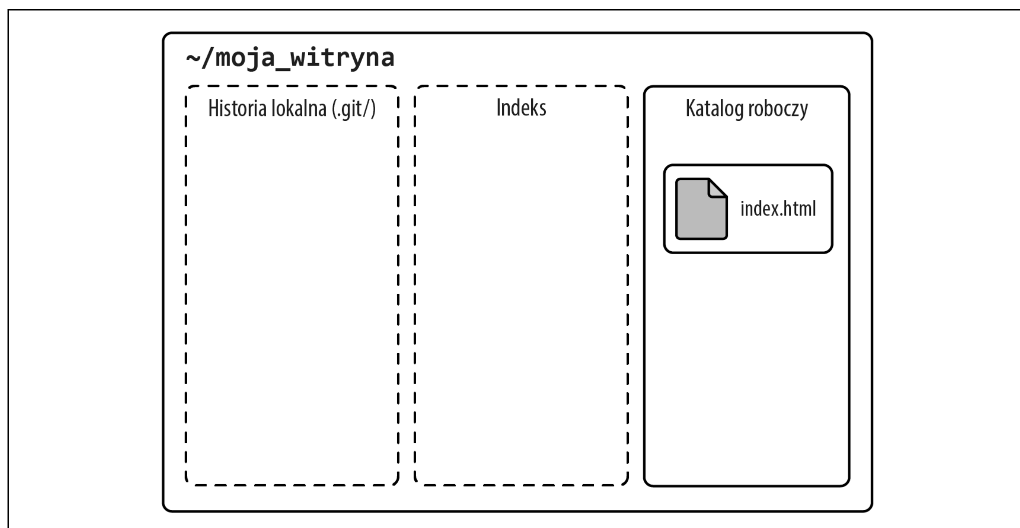
Kup ksi k Pole ksi k

System Git traktuje katalog *~/moja_witryna* jako **katalog roboczy**. Zawiera on bieżącą wersję plików Twojej witryny internetowej. Gdy dokonasz zmian w istniejących plikach lub dodasz do projektu nowe pliki, Git zarejestruje te zmiany w ukrytym folderze *.git*.

Z myślą o procesie uczenia się będziemy korzystać z dwóch katalogów wirtualnych o nazwach *Historia lokalna* i *Indeks*, aby zilustrować pojęcie inicjalizowania nowego repozytorium systemu Git. O tych dwóch katalogach będzie mowa odpowiednio w rozdziałach 4. i 5.

Na rysunku 1.2 pokazano to, co właśnie zostało objaśnione.





Rysunek 1.2. Początkowe repozytorium

Linie kreskowe otaczające nazwy *Historia lokalna* i *Indeks* reprezentują ukryte katalogi w obrębie folderu *.git*.

Dodawanie pliku do repozytorium

Do tej pory **utworzyłeś** jedynie nowe repozytorium Git. Inaczej mówiąc, jest ono puste. Choć plik *index.html* istnieje w katalogu *~/moja_witryna*, dla systemu Git jest to **katalog roboczy**, czyli reprezentacja czegoś w rodzaju notatnika lub katalogu, gdzie często modyfikujesz pliki.

Gdy po zakończeniu wprowadzania zmian w plikach chcesz je umieścić w repozytorium systemu Git, konieczne będzie przeprowadzenie tego bezpośrednio za pomocą polecenia `git add plik`:

```
$ git add index.html
```


Kup ksi k   Pole ksi k

Choć za pomocą polecenia `git add .` możesz pozwolić systemowi Git na dodanie wszystkich plików w katalogu oraz wszystkich podkatalogów, powoduje to umieszczenie ich wszystkich w przechowalni. Radzimy postępować świadomie odnośnie do tego, co planuje się wstawiać do przechowalni. Głównie ma to na celu uniknięcie uwzględnienia poufnych informacji lub niepożądanych plików w momencie tworzenia rewizji. Aby zapobiec dołączeniu takich informacji, możesz użyć pliku *.gitignore* omówionego w rozdziale 5.

Argument `.` w postaci pojedynczej kropki (w żargonie systemu Unix stosuje się też dla niej termin **dot**) pełni rolę skróconej reprezentacji bieżącego katalogu.

W przypadku użycia polecenia `git add` system Git „wie", że zamierzasz uwzględnić w repozytorium jako rewizję końcową iterację modyfikacji w pliku *index.html*. Jak dotąd jednak Git umieścił jedynie plik w przechowalni, co stanowi pośredni krok przed utworzeniem migawki w ramach operacji zatwierdzania (ang. *commit*).

Git oddziela kroki poleceń `add` i `commit`, aby uniknąć nieprzewidywalności przy jednoczesnym zapewnianiu elastyczności i szczegółowości związanej z metodą rejestrowania zmian. Wyobraź sobie, jak zakłócające pracę, niejasne i czasochłonne byłoby aktualizowanie repozytorium każdorazowo w momencie dodawania, usuwania lub modyfikowania pliku. Zamiast tego wiele powiązanych i tymczasowych kroków, takich jak dodawanie, może być wykonywanych **partiami**. Dzięki temu możliwe jest utrzymanie repozytorium w stabilnym i spójnym stanie. Metoda ta umożliwia też uzyskanie uzasadnienia, dlaczego kod jest modyfikowany. W rozdziale 4. bardziej zagłębimy się w to zagadnienie.

Zalecamy, aby przed utworzeniem rewizji dążyć do grupowania logicznych partii zmian. Taka rewizja jest określana mianem **atomowej**. Okaże się ona pomocna w sytuacjach, gdy w kolejnych rozdziałach będziesz musiał przeprowadzić pewne zaawansowane operacje systemu Git.

Uruchomienie polecenia `git status` ujawnia ten pośredni stan pliku *index.html*:

```
$ git status
On branch main
No commits yet
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
        new file: index.html
```

Polecenie informuje o tym, że nowy plik *index.html* zostanie dodany do repozytorium podczas następnej operacji zatwierdzania.

Po umieszczeniu pliku w przechowalni kolejnym logicznym krokiem jest zatwierdzenie pliku w repozytorium. Po wykonaniu tej operacji plik staje się częścią **historii rewizji repozytorium**. Dla uproszczenia będziemy w tym przypadku posługiwać się terminem **historia repo**. Każdorazowo podczas tworzenia rewizji system Git rejestruje jednocześnie kilka innych porcji metadanych, wśród których najważniejsze to komunikat dziennika rewizji i twórca zmiany.

W pełni kwalifikowana postać polecenia `git commit` powinna uwzględniać zwięzły i sensowny komunikat dziennika utworzony za pomocą używanego aktualnie języka. Komunikat ma opisywać zmianę wprowadzaną przez rewizję. Jest to bardzo pomocne, gdy trzeba przejrzeć historię

Kup ksi k   Pole ksi k

repo w celu zidentyfikowania konkretnej zmiany lub szybkiego ustalenia zmian w ramach rewizji bez konieczności zagłębiania się w szczegóły zmian. Zagadnienie to zostanie obszerniej omówione w rozdziałach 4. i 8.

Zatwierdźmy umieszczony w przechowalni plik *index.html* witryny internetowej:

```
$ git commit -m "Początkowa zawartość katalogu moja_witryna"

[main (root-commit) c149e12] początkowa zawartość katalogu moja_witryna
 1 file changed, 1 insertion(+)
 create mode 100644 index.html
```

Szczegóły dotyczące osoby, która tworzy rewizję, są uzyskiwane z przygotowanej wcześniej konfiguracji systemu Git.

W przykładowym kodzie podano argument `-m`, aby mieć możliwość zapewnienia komunikatu dziennika bezpośrednio w wierszu poleceń. Jeśli wolisz dostarczyć szczegółowy komunikat dziennika za pośrednictwem interaktywnej sesji edytora, również jest to możliwe. Niezbędne będzie skonfigurowanie systemu Git pod kątem uruchamiania Twojego ulubionego edytora w trakcie używania polecenia `git commit` (pomiń argument `-m`). Jeżeli jeszcze tego nie zrobiłeś, możesz ustawić zmienną środowiskową `$GIT_EDITOR` w następujący sposób:

```
# Dla powłoki bash lub zsh
$ export GIT_EDITOR=vim

# Dla powłoki tcsh
$ setenv GIT_EDITOR emacs
```

System Git będzie akceptować domyślny edytor tekstu określony w zmiennych środowiskowych `VISUAL` i `EDITOR`. Jeśli żaden edytor nie zostanie skonfigurowany, Git będzie się posiłkować edytorem `vi`.

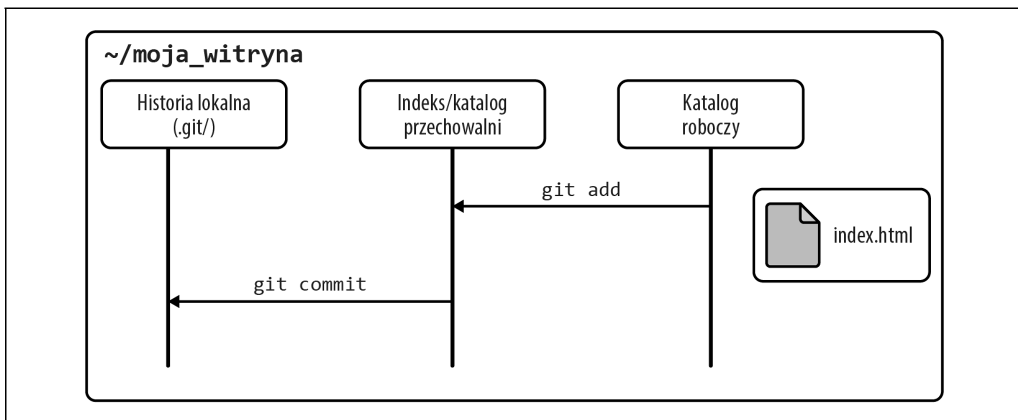
Po zatwierdzeniu pliku *index.html* w repozytorium uruchom polecenie `git status` w celu zaktualizowania bieżącego stanu repozytorium. W prezentowanym przykładzie po wykonaniu polecenie to powinno wskazywać, że nie ma żadnych oczekujących zmian do zatwierdzenia:

```
$ git status

On branch main
nothing to commit, working tree clean
```

System Git informuje też o tym, że **katalog roboczy** jest czysty. Oznacza to, że nie ma w nim żadnych nowych ani zmodyfikowanych plików, które różnią się od tego, co znajduje się w repozytorium.

Rysunek 1.3 ułatwi wizualizację wszystkich właśnie omówionych kroków.

Kup ksi k   Pole ksi k



Rysunek 1.3. Umieszczenie pliku w przechowalni i dodanie go do repozytorium

Różnica między poleceniami `git add` i `git commit` jest bardzo podobna jak w przypadku ustawiania grupy uczniów w preferowany sposób, aby wykonać idealne zdjęcie klasy: polecenie `git add` odpowiada za organizację, natomiast polecenie `git commit` za utworzenie migawki.

Tworzenie kolejnej rewizji

Wprowadźmy kilka modyfikacji w pliku *index.html* i utwórzmy historię repo w obrębie repozytorium.

Przekształć plik *index.html* we właściwy plik HTML i zatwierdź w nim zmianę:

```
$ cd ~/moja_witryna
# Edycja pliku index.html.

$ cat index.html
<html>
<body>
Moja witryna jest cudowna!
</body>
</html>

$ git commit index.html -m 'Przekształcenie w plik HTML'
[main 521edbe] 'Przekształcenie w plik HTML
1 file changed, 5 insertions(+), 1 deletion(-)
```

Jeśli jesteś już zaznajomiony z systemem Git, możesz się zastanawiać, dlaczego pominięto krok polecenia `git add index.html` przed zatwierdzeniem pliku. Wynika to z tego, że zawartość do zatwierdzenia może zostać określona w systemie Git na więcej niż jeden sposób.

Aby dowiedzieć się więcej o dostępnych opcjach, wprowadź polecenie **`git commit --help`**:

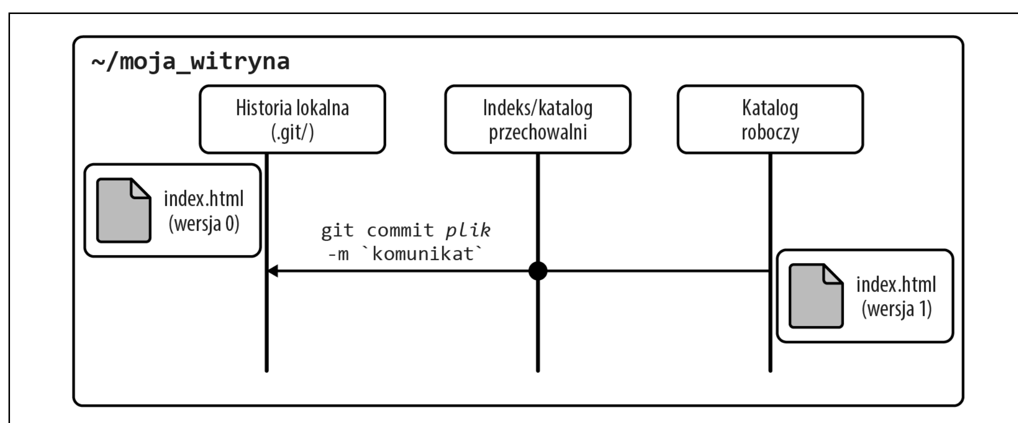
```
$ git commit --help

NAME
     git-commit - Record changes to the repository

SYNOPSIS
     git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
```


Kup ksi k   Pole ksi k

```
                          [--dry-run] [(-c | -C | --squash) <commit> | --fixup [(amend|reword):]
                          ↪<commit>)]
                          [-F <file> | -m <msg>] [--reset-author] [--allow-empty]
                          [--allow-empty-message] [--no-verify] [-e] [--author=<author>]
                          [--date=<date>] [--cleanup=<mode>] [--[no-]status]
                          [-i | -o] [--pathspec-from-file=<file> [--pathspec-file-nul]]
                          [(--trailer <token>[(=|:)<value>])...] [-S[<keyid>]]
                          [--] [<pathspec>...]
...
```

Szczegółowe objaśnienie różnych metod zatwierdzania jest też dostępne na stronach podręcznika wyświetlanych po wykonaniu polecenia `git commit --help`.

W prezentowanym przykładzie zdecydowano się na zatwierdzenie pliku *index.html* z użyciem dodatkowego argumentu `-m`, który umożliwia zapewnienie komunikatu wyjaśniającego zmiany w rewizji (`'Przekształcenie w plik HTML'`). Na rysunku 1.4 objaśniono właśnie przedstawioną metodę.



Rysunek 1.4. Umieszczenie pliku w przechowalni i dodanie zmian do monitorowanego pliku w repozytorium

Zauważ, że w przypadku zastosowanego tutaj polecenia `git commit index.html -m 'Przekształcenie w plik HTML'` nie jest pomijane umieszczanie pliku w przechowalni. System Git zajmuje się tym automatycznie w ramach operacji zatwierdzania.

Wyświetlanie rewizji

Gdy w historii repo znajduje się już więcej rewizji, możesz poddać je inspekcji na różne sposoby. Część poleceń `git` prezentuje sekwencję poszczególnych rewizji, część zapewnia podsumowanie konkretnej rewizji, a jeszcze inne polecenia wyświetlają wszystkie szczegóły dowolnej rewizji określonej w repozytorium.

Polecenie `git log` zapewnia historię kolejnych osobnych rewizji w obrębie repozytorium:

```
$ git log

commit 521edbe1dd2ec9c6f959c504d12615a751b5218f (HEAD -> main)
```


Kup ksi k   Pole ksi k

```
Author: Jan Nowak <jn@przyklad.pl>
Date:   Mon Jul 4 12:01:54 2022 +0200

     Przekształcenie w plik HTML

commit c149e12e89a9c035b9240e057b592ebfc9c88ea4
Author: Jan Nowak <jn@przyklad.pl>
Date:   Mon Jul 4 11:58:36 2022 +0200

     Początkowa zawartość katalogu moja_witryna
```

W powyższych danych wyjściowych polecenie `git log` zapewnia szczegółowe informacje z dziennika dla każdej rewizji w repozytorium. Na razie dysponujesz tylko dwiema rewizjami w historii repo, co ułatwia analizowanie tych danych. W przypadku repozytoriów z wieloma historiami rewizji taki standardowy widok może nie być pomocny podczas przeglądania długiej listy dokładnych informacji o rewizjach. W takich sytuacjach możesz użyć opcji `--oneline`, aby razem z komunikatem zatwierdzania wyświetlić identyfikator liczbowy rewizji:

```
$ git log --oneline

521edbe (HEAD -> main) Przekształcenie w plik HTML
c149e12 Początkowa zawartość katalogu moja_witryna
```

Pozycje dziennika rewizji są wyszczególniane w kolejności od najnowszych do najstarszych[1] (oryginalny plik). Każda pozycja zawiera personalia twórcy rewizji i adres e-mail, datę utworzenia rewizji, komunikat dziennika dotyczący zmiany, a także wewnętrzny numer identyfikacyjny rewizji. Identyfikator rewizji objaśniono w punkcie „Baza danych adresowana zawartością” rozdziału 2. Rewizje omówiono bardziej szczegółowo w rozdziale 4.

Aby uzyskać więcej informacji o konkretnej rewizji, użyj polecenia `git show` z podanym identyfikatorem rewizji:

```
$ git show c149e12e89a9c035b9240e057b592ebfc9c88ea4

commit c149e12e89a9c035b9240e057b592ebfc9c88ea4
Author: Jan Nowak <jn@przyklad.pl>
Date: Mon Jul 4 11:58:36 2022 +0200

Początkowa zawartość katalogu moja_witryna

diff --git a/index.html b/index.html
new file mode 100644
index 0000000..6331c71
--- /dev/null
+++ b/index.html
@@ -0,0 +1 @@
+Moja cudowna witryna!
```

Jeśli wykonasz polecenie `git show` bez podania numeru rewizji, zostaną pokazane po prostu szczegóły rewizji HEAD, którą w zaprezentowanym przykładzie jest najnowsza rewizja.

[1] Mówiąc wprost, rewizje nie są uporządkowane **chronologicznie**, lecz raczej z wykorzystaniem sortowania **topologicznego**.

Kup ksi k Pole ksi k

Polecenie `git log` udostępnia dzienniki rewizji pozwalające stwierdzić, w jaki sposób zmiany dotyczące każdej rewizji są uwzględniane w historii repo. W celu wyświetlenia zwięzłych, jednowierszowych podsumowań dla bieżącej gałęzi projektowej bez określania dla polecenia `git log --oneline` dodatkowych opcji filtrowania możesz skorzystać z alternatywnego rozwiązania polegającego na użyciu polecenia `git show-branch`:

```
$ git show-branch --more=10

[main] Przekształcenie w plik HTML
[main^] Początkowa zawartość katalogu moja_witryna
```

Łańcuch `--more=10` ujawnia maksymalnie dziesięć dodatkowych wersji, ale na razie istnieją tylko dwie, dlatego obie zostały pokazane (w tym przykładzie domyślnie zostałaby wyszczególniona wyłącznie najnowsza rewizja). `main` to domyślna nazwa gałęzi.

W rozdziale 3. bardziej szczegółowo będą omawiane gałęzie, a także ponownie skorzystasz z polecenia `git show-branch`.

Wyświetlanie różnic między rewizjami

Po uzyskaniu historii repo w wyniku dodania rewizji możesz już sprawdzić różnice między dwoma rewizjami pliku *index.html*. Konieczne będzie ponowne użycie identyfikatorów liczbowych rewizji i uruchomienie polecenia `git diff`:

```
$ git diff c149e12e89a9c035b9240e057b592ebfc9c88ea4 \
            521edbe1dd2ec9c6f959c504d12615a751b5218f

diff --git a/index.html b/index.html
index 6331c71..8cfcb90 100644
--- a/index.html
+++ b/index.html
@@ -1 +1,5 @@
-Moja cudowna witryna!

+<html>
+<body>
Moja witryna jest cudowna!
+</body>
+</html>
```

Dane wyjściowe przypominają to, co jest zwracane przez polecenie `git diff`. Zgodnie z konwencją identyfikator `c149e12e89a9c035b9240e057b592ebfc9c88ea4` pierwszej rewizji reprezentuje wcześniejszą zawartość pliku *index.html*, natomiast jego najnowsza zawartość jest określana przez identyfikator `521edbe1dd2ec9c6f959c504d12615a751b5218f` drugiej rewizji. Znak + poprzedza zatem każdy wiersz nowej zawartości po tym, jak użyto znaku - w celu wskazania usuniętej zawartości.

Nie obawiaj się długich wartości heksadecymalnych. System Git oferuje wiele łatwiejszych sposobów uruchamiania podobnych poleceń w krótszej postaci, dlatego możesz uniknąć długich i skomplikowanych identyfikatorów rewizji. Wystarczających jest zwykle pierwszych siedem znaków liczb szesnastkowych, tak jak w zamieszczonym wcześniej przykładzie użycia polecenia `git log --oneline`. Kwestią tą zajmiemy się szerzej w punkcie „Baza danych adresowana zawartością" rozdziału 2.

Kup ksi k Pole ksi k

Usuwanie plików z repozytorium oraz zmienianie ich nazw

Skoro już się dowiedziałeś, jak dodawać pliki do repozytorium systemu Git, przyjrzyjmy się kwestii usuwania z niego pliku. Operacja ta przebiega analogicznie do dodawania pliku, jednak używa się w tym celu polecenia `git rm`. Załóżmy, że w skład zawartości witryny internetowej wchodzi plik *adverts.html*, który planujesz usunąć. Możesz to zrobić w następujący sposób:

```
$ cd ~/moja_witryna
$ ls
index.html adverts.html

$ git rm adverts.html
rm 'adverts.html'

$ git commit -m "Usuwanie pliku adverts.html"
[main 97ff70a] Usuwanie pliku adverts.html
1 file changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 adverts.html
```

Podobnie jak dodawanie operacja usuwania również wymaga wykonania dwóch kroków, czyli wyrażenia zamiaru usunięcia pliku za pomocą polecenia `git rm`, które powoduje też **umieszczenie** pliku do usunięcia **w przechowalni** i potwierdzenia zmiany w repozytorium z użyciem polecenia `git commit`.

Tak jak w przypadku polecenia `git add` zastosowanie polecenia `git rm` nie powoduje bezpośrednio usunięcia pliku. Zamiast tego modyfikowane jest to, co jest monitorowane, czyli usuwanie lub dodawanie pliku.

Nazwę pliku możesz zmienić pośrednio za pomocą kombinacji poleceń `git rm` i `git add`. Ewentualnie operację tę możesz wykonać w szybszy i bezpośredni sposób z użyciem polecenia `git mv`. Oto przykład zastosowania pierwszego sposobu:

```
$ mv foo.html bar.html
$ git rm foo.html
rm 'foo.html'

$ git add bar.html
```

W przypadku przedstawionej sekwencji działań musisz na początku wykonać polecenie `mv foo.html bar.html`, aby zapobiec trwałemu usunięciu pliku *foo.html* z systemu plików przez polecenie `git rm`.

Oto identyczna operacja zrealizowana za pomocą polecenia `git mv`:

```
$ git mv foo.html bar.html
```

W obu wariantach zmiany znajdujące się w przechowalni muszą później zostać zatwierdzone:

```
$ git commit -m "Zmieniono nazwę foo na bar"
[main d1e37c8] Zmieniono nazwę foo na bar
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename foo.html => bar.html (100%)
```

System Git obsługuje operacje przenoszenia plików odmiennie niż większość podobnych systemów. W tym celu wykorzystuje mechanizm oparty na podobieństwie zawartości między dwoma wersjami plików. Szczegóły z tym związane opisano w rozdziale 5.

Kup ksi k Pole ksi k

Korzystanie z repozytorium współużytkowanego

Zainicjalizowałeś już nowe repozytorium i wprowadziłeś w nim zmiany. Wszystkie zmiany są dostępne wyłącznie w obrębie lokalnego komputera używanego do projektowania. To dobry przykład tego, jak możesz zarządzać projektem, który jest dostępny tylko dla Ciebie. W jaki jednak sposób współpracować z innymi osobami z użyciem repozytorium udostępnionym na serwerze Git? Wyjaśnijmy, jak możesz to zrealizować.

Tworzenie kopii lokalnej repozytorium

Masz możliwość utworzenia kompletnej kopii lub **klona** repozytorium za pomocą polecenia `git clone`. W ten właśnie sposób współpracujesz z innymi ludźmi – wprowadzając zmiany w tych samych plikach i utrzymując synchronizację ze zmianami z innych wersji jednego repozytorium.

Na potrzeby prezentowanego przykładu zacznijmy po prostu od utworzenia kopii istniejącego repozytorium. Dalej możliwe będzie zestawienie tego samego przykładu w sytuacji, gdyby repozytorium znajdowało się na zdalnym serwerze Git:

```
$ cd ~
$ git clone moja_witryna nowa_witryna
```

Choć te dwa repozytoria systemu Git zawierają obecnie dokładnie te same obiekty, pliki i katalogi, występują między nimi pewne subtelne różnice. Możesz je sprawdzić za pomocą następujących poleceń:

```
$ ls -lsa moja_witryna nowa_witryna
...
$ diff -r moja_witryna nowa_witryna
...
```

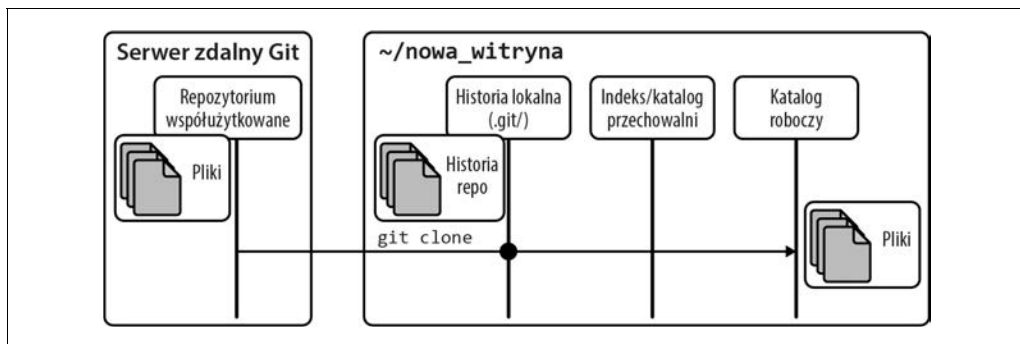
Użycie polecenia `git clone` w przypadku tego typu lokalnego systemu plików do utworzenia kopii repozytorium jest dość podobne do zastosowania polecenia `cp -a` lub `rsync`. Z kolei jeśli byłoby **klonowane** to samo repozytorium z serwera Git, składnia wyglądałaby następująco:

```
$ cd ~

$ git clone https://serwer-hostingowy-git.com/katalog/moja_witryna.git nowa_witryna
Cloning into 'nowa_witryna'...
remote: Enumerating objects: 2, done.
remote: Counting objects: 100% (2/2), done.
remote: Compressing objects: 100% (103/103), done.
remote: Total 125 (delta 45), reused 65 (delta 18), pack-reused 0
Receiving objects: 100% (125/125), 1.67 MiB | 4.03 MiB/s, done.
Resolving deltas: 100% (45/45), done.
```

Po sklonowaniu repozytorium możesz zmodyfikować sklonowaną wersję, utworzyć nowe rewizje, sprawdzić jego dzienniki, historię itp. Jest to kompletne repozytorium z pełną historią. Pamiętaj, że zmiany dokonywane w sklonowanym repozytorium nie będą automatycznie wypychane do oryginalnej kopii w obrębie repozytorium.

Zagadnienie to zilustrowano na rysunku 1.5.

Kup ksi k   Pole ksi k



Rysunek 1.5. Klonowanie repozytorium współużytkowanego

Staraj się nie rozpraszać niektórymi terminami widocznymi w danych wyjściowych. Na potrzeby określania nazwy repozytorium, które ma zostać sklonowane, Git obsługuje szerszy zestaw źródeł repozytoriów, w tym nazwy sieciowe. Te postacie nazw i ich zastosowanie zostaną objaśnione w rozdziale 11.

Pliki konfiguracyjne

Pliki konfiguracyjne systemu Git to wyłącznie proste pliki tekstowe w stylu plików *.ini*. Służą one do przechowywania preferencji i ustawień używanych przez wiele poleceń `git`. Część ustawień reprezentuje preferencje osobiste (np. określające, czy powinna zostać użyta zmienna `color.pager`), a część jest niezbędna do poprawnego działania repozytorium (np. zmienna `core.repositoryformatversion`). Jeszcze inne ustawienia modyfikują nieznacznie działanie poleceń `git` (np. zmienna `gc.auto`). Podobnie do innych narzędzi system Git obsługuje hierarchię plików konfiguracyjnych.

Hierarchia plików konfiguracyjnych

Na rysunku 1.6 przedstawiono hierarchię plików konfiguracyjnych systemu Git, począwszy od najmniej ważnych plików.

.git/config

W pliku znajdują się ustawienia konfiguracyjne powiązane z repozytorium, modyfikowane za pomocą opcji `--file` lub domyślne. Opcja `--local` umożliwia też wprowadzanie zmian w tym pliku. Zawarte w nim ustawienia mają największą ważność.

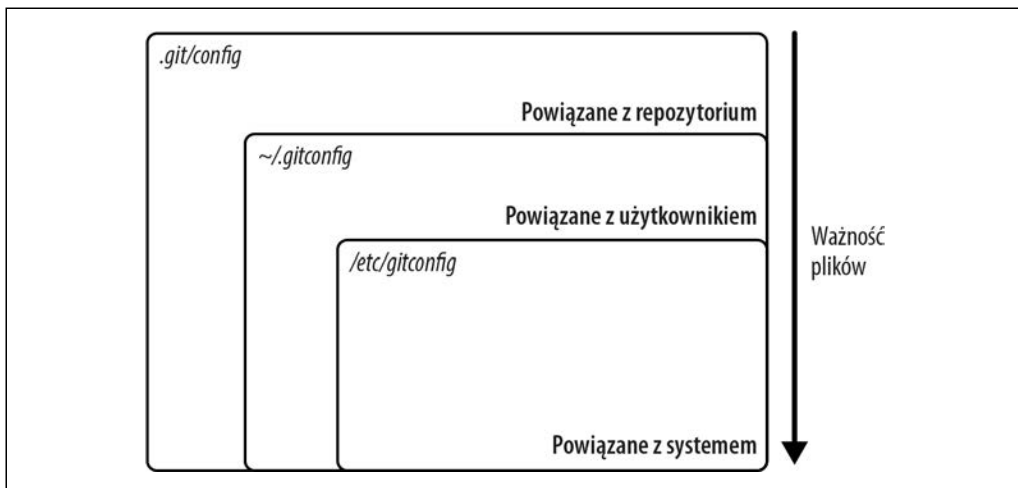
~/.gitconfig

W pliku umieszczono ustawienia konfiguracyjne powiązane z użytkownikiem, modyfikowane z użyciem opcji `--global`.

/etc/gitconfig

W pliku umiejscowiono ogólnosystemowe ustawienia konfiguracyjne modyfikowane za pomocą opcji `--system`, jeśli dysponujesz odpowiednimi uprawnieniami zapisu systemu Unix umożliwiającymi modyfikowanie pliku *gitconfig*. Ustawienia te mają najmniejszą ważność. Zależnie od używanej instalacji plik ustawień systemowych może znajdować się w innym miejscu (być może w katalogu */usr/local/etc/gitconfig*) lub może go wcale nie być.

Kup ksi k Pole ksi k



Rysunek 1.6. Hierarchia plików konfiguracyjnych systemu Git

Aby na przykład zapisać personalia autora i adres e-mail, które będą używane dla wszystkich rewizji tworzonych w przypadku wszystkich stosowanych repozytoriów, za pomocą polecenia `git config --global` skonfiguruj wartości zmiennych `user.name` i `user.email` w pliku *$HOME/.git config*:

```
$ git config --global user.name "Jan Nowak"
$ git config --global user.email "jn@przyklad.pl"
```

Jeśli musisz określić personalia oraz adres e-mail powiązane z repozytorium, które spowodują nadpisanie wartości opcji `--global`, po prostu pomiń ją lub w celu sprecyzowania użyj opcji `--local`:

```
$ git config user.name "Jan Nowak"
$ git config user.email "jn@specjalny-projekt.przyklad.pl"
```

Możesz użyć polecenia `git config -l` (lub z pełną postacią `--list`), aby wyświetlić jedną listę wartości ustawionych dla wszystkich zmiennych znalezionych w kompletnym zestawie plików konfiguracyjnych:

```
# Utworzenie zupełnie nowego, pustego repozytorium
$ mkdir /tmp/new
$ cd /tmp/new
$ git init

# Ustawienie wybranych wartości konfiguracyjnych
$ git config --global user.name "Jan Nowak"
$ git config --global user.email "jn@przyklad.pl"
$ git config user.email "jn@specjalny-projekt.przyklad.pl"

$ git config -l
user.name=Jan Nowak
user.email=jn@przyklad.pl
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
user.email=jn@specjalny-projekt.przyklad.pl
```


Kup ksi k Pole ksi k

Dodanie opcji `--show-scope` i `--show-origin` w trakcie wpisywania polecenia `git config -l` ułatwi wyświetlenie różnych źródeł konfiguracji! Przekonaj się o tym, uruchamiając w oknie terminala polecenie `git config -l --show-scope --show-origin`.

Ponieważ pliki konfiguracyjne to proste pliki tekstowe, możesz wyświetlić ich zawartość za pomocą polecenia `cat`, a także edytować je z wykorzystaniem ulubionego edytora tekstu:

```
# Sprawdzanie jedynie ustawień powiązanych z repozytorium

$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
ignorecase = true
    precomposeunicode = true

[user]
    email = jn@specjalny-projekt.przyklad.pl
```

Zależnie od typu używanego systemu operacyjnego zawartość tekstowego pliku konfiguracyjnego może być prezentowana z niewielkimi różnicami. Wiele spośród nich umożliwia skorzystanie z różnych właściwości systemu plików.

Jeśli musisz usunąć ustawienie z plików konfiguracyjnych, zastosuj opcję `--unset` razem z odpowiednią flagą tych plików:

```
$ git config --unset --global user.email
```

System Git zapewnia wiele opcji konfiguracyjnych i zmiennych środowiskowych, które często mają takie samo przeznaczenie. Przykładowo, możesz określić wartość dla edytora, który ma zostać użyty podczas tworzenia komunikatu dziennika rewizji. Zależnie od istniejącej konfiguracji w tym celu zostanie zastosowana jedna z następujących rzeczy:

- zmienna środowiskowa `GIT_EDITOR`,
- opcja konfiguracyjna `core.editor`,
- zmienna środowiskowa `VISUAL`,
- zmienna środowiskowa `EDITOR`,
- polecenie `vi`.

Istnieje więcej niż kilkaset parametrów konfiguracyjnych. Nie będziemy Cię nimi zanudzać, ale w dalszej części książki zwrócimy uwagę na te, które są istotne. Bardziej rozbudowana lista (lecz nadal niekompletna) jest dostępna na stronie podręcznika wyświetlanej po użyciu polecenia `git config`.

Kompletna lista wszystkich poleceń `git` jest dostępna w internecie (*https://git-scm.com/docs*).

Kup ksi k   Pole ksi k

Konfigurowanie aliasu

Aliasy systemu Git umożliwiają zastąpienie prostymi i łatwymi do zapamiętania aliasami powszechnych, lecz złożonych poleceń `git`, które często wprowadzasz. Dzięki aliasom nie trzeba również zapamiętywać ani wpisywać tych długich poleceń. Ponadto eliminuje się w ten sposób frustracje związane z literówkami:

```
$ git config --global alias.show-graph \
        'log --graph --abbrev-commit --pretty=oneline'
```

W powyższym przykładzie utworzono alias `show-graph` i udostępniono go do użycia w dowolnym definiowanym repozytorium. W przypadku zastosowania polecenia `git show-graph` zapewni ono te same dane wyjściowe, które są uzyskiwane po wprowadzeniu długiego polecenia `git log` z tymi wszystkimi opcjami.

Podsumowanie

Z pewnością będziesz mieć mnóstwo pytań bez odpowiedzi związanych z zasadami działania systemu Git, nawet po tym wszystkim, czego się dotychczas dowiedziałeś. Przykładowo: jak Git przechowuje każdą wersję pliku? Co tak naprawdę tworzy rewizję? Skąd się wzięły te dziwne numery rewizji? Dlaczego używa się nazwy `main`? I czy gałąź jest tym, czym **według mnie** jest? Są to dobre pytania. Zaprezentowane dotąd informacje wprowadzają w zakres operacji, jakie będziesz często wykonywać w ramach swoich projektów. Szczegółowe odpowiedzi na Twoje pytania zostaną przedstawione w części II książki.

W następnym rozdziale pojawią się definicje części terminologii, zostaniesz wprowadzony w niektóre zagadnienia związane z systemem Git, a także zostanie zbudowany fundament dla wiedzy przekazywanej w reszcie książki.

Kup ksi k

Pole ksi k

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

Bądź na bieżąco, kontroluj wersje projektu!

Jeśli pracujesz w zespole i prowadzisz złożone projekty, dobrze wiesz, czym się kończy brak zarządzania wersjami. Dzięki Gitowi możesz zapomnieć o tych problemach. Ten niezawodny rozproszony system kontroli wersji cechuje się szybkością i skalowalnością, zapewnia bogatą kolekcję zestawów poleceń, jest też zoptymalizowany pod kątem operacji lokalnych. Aby docenić Gita, musisz jedynie nabrać wprawy, pewności siebie — i dobrze go poznać.

Ta książka, napisana z myślą o inżynierach oprogramowania, jest trzecim, gruntownie zaktualizowanym wydaniem praktycznego przewodnika, który szybko przeprowadzi Cię od podstaw aż do zaawansowanych technik pracy z Gitem. Po zapoznaniu się z filozofią systemu i jego najważniejszymi funkcjami płynnie przejdziesz do takich zagadnień jak modyfikowanie drzew, korzystanie z dziennika odwołań i schowka. Znajdziesz tu również kilka przydatnych scenariuszy projektowych i sporo ciekawych wskazówek i porad. W efekcie nauczysz się korzystać z elastyczności Gita i w praktyczny sposób zarządzać procesem tworzenia kodu z zastosowaniem wielu różnych metod.

W książce:

- rozproszone systemy kontroli wersji
- typowe zastosowania Gita i jego podstawowe funkcje
- metody zarządzania scaleniami, konfliktami, poprawkami i różnicami
- zaawansowane techniki, takie jak zmiana bazy i haki
- korzystanie z serwisu GitHub

Prem Kumar Ponuthorai planuje i wdraża produkty serwisu GitHub. Jest inżynierem oprogramowania, prowadzi warsztaty i szkolenia dotyczące użytkowania systemu Git.

Jon Loeliger jest inżynierem oprogramowania. Angażuje się w projekty open source, takie jak Linux, U-Boot i Git. Napisał kilka artykułów poświęconych systemowi Git do czasopisma „Linux Magazine”.

Autorzy książki przechodzą w niej stopniowo od najbardziej podstawowych zagadnień ze świata systemu Git do zaawansowanych zastosowań.

Jeff King, projektant oprogramowania *open source*

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-8322-647-7

Cena: 119,00 zł