Git

Spis treści

[Pierwsze kroki 2](#_Toc536556083)

[Wprowadzenie do kontroli wersji 2](#_Toc536556084)

[Lokalne systemy kontroli wersji 2](#_Toc536556085)

[Scentralizowane systemy kontroli wersji 3](#_Toc536556086)

[Rozproszone systemy kontroli wersji 4](#_Toc536556087)

[Krótka historia Git 5](#_Toc536556088)

[Podstawy Git 6](#_Toc536556089)

[Migawki, nie różnice 6](#_Toc536556090)

[Niemal każda operacja jest lokalna 7](#_Toc536556091)

[Git ma wbudowane mechanizmy spójności danych 8](#_Toc536556092)

[Standardowo Git wyłącznie dodaje nowe dane 8](#_Toc536556093)

[Instalacja Git 8](#_Toc536556094)

[Instalacja ze źródeł 9](#_Toc536556095)

[Instalacja w systemie Linux 9](#_Toc536556096)

[Instalacja na komputerze Mac 10](#_Toc536556097)

[Instalacja w systemie Windows 10](#_Toc536556098)

[Trzy stany 11](#_Toc536556099)

[Wstępna konfiguracja Git 12](#_Toc536556100)

[Twoja tożsamość 12](#_Toc536556101)

[Edytor 13](#_Toc536556102)

[Narzędzie obsługi różnic 13](#_Toc536556103)

[Sprawdzanie ustawień 13](#_Toc536556104)

[Uzyskiwanie pomocy 14](#_Toc536556105)

[Podsumowanie 14](#_Toc536556106)

[Podstawy Gita 14](#_Toc536556107)

[Pierwsze repozytorium Gita 14](#_Toc536556108)

[Inicjalizacja Gita w istniejącym katalogu 15](#_Toc536556109)

[Klonowanie istniejącego repozytorium 15](#_Toc536556110)

[Rejestrowanie zmian w repozytorium 16](#_Toc536556111)

[Sprawdzanie stanu twoich plików 17](#_Toc536556112)

[Śledzenie nowych plików 18](#_Toc536556113)

[Dodawanie zmodyfikowanych plików do poczekalni 18](#_Toc536556114)

[Ignorowanie plików 20](#_Toc536556115)

[Podgląd zmian w poczekalni i poza nią 20](#_Toc536556116)

[Zatwierdzanie zmian 23](#_Toc536556117)

[Pomijanie poczekalni 24](#_Toc536556118)

[Usuwanie plików 24](#_Toc536556119)

[Przenoszenie plików 25](#_Toc536556120)

# Pierwsze kroki

Ten rozdział poświęcony jest pierwszym krokom z Git. Rozpoczyna się krótkim wprowadzeniem do narzędzi kontroli wersji, następnie przechodzi do instalacji i początkowej konfiguracji Git. Po przeczytaniu tego rozdziału powinieneś rozumieć w jakim celu Git został stworzony, dlaczego warto z niego korzystać oraz być przygotowany do używania go.

## Wprowadzenie do kontroli wersji

Czym jest kontrola wersji i dlaczego powinieneś się nią przejmować? System kontroli wersji śledzi wszystkie zmiany dokonywane na pliku (lub plikach) i umożliwia przywołanie dowolnej wcześniejszej wersji. Przykłady w tej książce będą śledziły zmiany w kodzie źródłowym, niemniej w ten sam sposób można kontrolować praktycznie dowolny typ plików.

Jeśli jesteś grafikiem lub projektantem WWW i chcesz zachować każdą wersję pliku graficznego lub układu witryny WWW (co jest wysoce prawdopodobne), to używanie systemu kontroli wersji (VCS-Version Control System) jest bardzo rozsądnym rozwiązaniem. Pozwala on przywrócić plik(i) do wcześniejszej wersji, odtworzyć stan całego projektu, porównać wprowadzone zmiany, dowiedzieć się kto jako ostatnio zmodyfikował część projektu powodującą problemy, kto i kiedy wprowadził daną modyfikację. Oprócz tego używanie VCS oznacza, że nawet jeśli popełnisz błąd lub stracisz część danych, naprawa i odzyskanie ich powinno być łatwe. Co więcej, wszystko to można uzyskać całkiem niewielkim kosztem.

### Lokalne systemy kontroli wersji

Dla wielu ludzi preferowaną metodą kontroli wersji jest kopiowanie plików do innego katalogu (może nawet oznaczonego datą, jeśli są sprytni). Takie podejście jest bardzo częste ponieważ jest wyjątkowo proste, niemniej jest także bardzo podatne na błędy. Zbyt łatwo zapomnieć w jakim jest się katalogu i przypadkowo zmodyfikować błędny plik lub skopiować nie te dane.

Aby poradzić sobie z takimi problemami, programiści już dość dawno temu stworzyli lokalne systemy kontroli wersji, które składały się z prostej bazy danych w której przechowywane były wszystkie zmiany dokonane na śledzonych plikach (por. Rysunek 1-1).



Rysunek 1-1. Diagram lokalnego systemu kontroli wersji.

Jednym z najbardziej popularnych narzędzi VCS był system rcs, który wciąż jest obecny na wielu dzisiejszych komputerach. Nawet w popularnym systemie operacyjnym Mac OS X rcs jest dostępny po zainstalowaniu Narzędzi Programistycznych (Developer Tools). Program ten działa zapisując, w specjalnym formacie na dysku, dane różnicowe (to jest zawierające jedynie różnice pomiędzy plikami) z każdej dokonanej modyfikacji. Używając tych danych jest w stanie przywołać stan pliku z dowolnego momentu.

### Scentralizowane systemy kontroli wersji

Kolejnym poważnym problemem z którym można się spotkać jest potrzeba współpracy w rozwoju projektu z odrębnych systemów. Aby poradzić sobie z tym problemem stworzono scentralizowane systemy kontroli wersji (CVCS - Centralized Version Control System). Systemy takie jak CVS, Subversion czy Perforce składają się z jednego serwera, który zawiera wszystkie pliki poddane kontroli wersji, oraz klientów którzy mogą się z nim łączyć i uzyskać dostęp do najnowszych wersji plików. Przez wiele lat był to standardowy model kontroli wersji (por. Rysunek 1-2).



Rysunek 1-2. Diagram scentralizowanego systemu kontroli wersji.

Taki schemat posiada wiele zalet, szczególnie w porównaniu z VCS. Dla przykładu każdy może się zorientować co robią inni uczestnicy projektu. Administratorzy mają dokładną kontrolę nad uprawnieniami poszczególnych użytkowników. Co więcej systemy CVCS są także dużo łatwiejsze w zarządzaniu niż lokalne bazy danych u każdego z klientów.

Niemniej systemy te mają także poważne wady. Najbardziej oczywistą jest problem awarii centralnego serwera. Jeśli serwer przestanie działać na przykład na godzinę, to przez tę godzinę nikt nie będzie miał możliwości współpracy nad projektem, ani nawet zapisania zmian nad którymi pracował. Jeśli dysk twardy na którym przechowywana jest centralna baza danych zostanie uszkodzony, a nie tworzono żadnych kopii zapasowych, to można stracić absolutnie wszystko - całą historię projektu, może oprócz pojedynczych jego części zapisanych na osobistych komputerach niektórych użytkowników. Lokalne VCS mają ten sam problem - zawsze gdy cała historia projektu jest przechowywana tylko w jednym miejscu, istnieje ryzyko utraty większości danych.

### Rozproszone systemy kontroli wersji

W ten sposób dochodzimy do rozproszonych systemów kontroli wersji (DVCS - Distributed Version Control System). W systemach DVCS (takich jak Git, Mercurial, Bazaar lub Darcs) klienci nie dostają dostępu jedynie do najnowszych wersji plików, ale w pełni kopiują całe repozytorium. Gdy jeden z serwerów, używanych przez te systemy do współpracy, ulegnie awarii, repozytorium każdego klienta może zostać po prostu skopiowane na ten serwer w celu przywrócenia go do pracy (por. Rysunek 1-3).



Rysunek 1-3. Diagram rozproszonego systemu kontroli wersji.

Co więcej, wiele z tych systemów dość dobrze radzi sobie z kilkoma zdalnymi repozytoriami, więc możliwa jest jednoczesna współpraca z różnymi grupami ludzi nad tym samym projektem. Daje to swobodę wykorzystania różnych schematów pracy, nawet takich które nie są możliwe w scentralizowanych systemach, na przykład modeli hierarchicznych.

## Krótka historia Git

Jak z wieloma dobrymi rzeczami w życiu Git zaczął od odrobiny twórczej destrukcji oraz zażartych kontrowersji. Jądro Linuksa jest dość dużym projektem otwartego oprogramowania (ang. open source). Przez większą część życia tego projektu (1991-2002), zmiany w źródle były przekazywane jako łaty (ang. patches) i zarchiwizowane pliki. W roku 2002 projekt jądra Linuksa zaczął używać systemu DVCS BitKeeper.

W 2005 roku relacje pomiędzy wspólnotą rozwijającą jądro Linuksa, a firmą która stworzyła BitKeepera znacznie się pogorszyły, a pozwolenie na nieodpłatne używanie systemu zostało cofnięte. To skłoniło programistów pracujących nad jądrem (a w szczególności Linusa Torvaldsa, twórcę Linuksa) do stworzenia własnego systemu na podstawie wiedzy wyniesionej z używania BitKeepera. Do celów tego nowego systemu należały:

* Szybkość
* Prosta konstrukcja
* Silne wsparcie dla nieliniowego rozwoju (tysięcy równoległych gałęzi)
* Pełne rozproszenie
* Wydajna obsługa dużych projektów, takich jak jądro Linuksa (szybkość i rozmiar danych)

Od swoich narodzin w 2005 roku, Git ewoluował i ustabilizował się jako narzędzie łatwe w użyciu, jednocześnie zachowując wyżej wymienione cechy. Jest niewiarygodnie szybki, bardzo wydajny przy pracy z dużymi projektami i posiada niezwykły system gałęzi do nieliniowego rozwoju (patrz Rozdział 3).

## Podstawy Git

Czym jest w skrócie Git? To jest bardzo istotna sekcja tej książki, ponieważ jeśli zrozumiesz czym jest Git i podstawy jego działania to efektywne używanie go powinno być dużo prostsze. Podczas uczenia się Git staraj się nie myśleć o tym co wiesz o innych systemach VCS, takich jak Subversion czy Perforce; pozwoli Ci to uniknąć subtelnych błędów przy używaniu tego narzędzia. Git przechowuje i traktuje informacje kompletnie inaczej niż te pozostałe systemy, mimo że interfejs użytkownika jest dość zbliżony. Rozumienie tych różnic powinno pomóc Ci w unikaniu błędów przy korzystaniu z Git.

### Migawki, nie różnice

Podstawową różnicą pomiędzy Git, a każdym innym systemem VCS (włączając w to Subversion) jest podejście Git do przechowywanych danych. Większość pozostałych systemów przechowuje informacje jako listę zmian na plikach. Systemy te (CVS, Subversion, Perforce, Bazaar i inne) traktują przechowywane informacje jako zbiór plików i zmian dokonanych na każdym z nich w okresie czasu. Obrazuje to Rysunek 1-4.



Rysunek 1-4. Inne system przechowują dane w postaci zmian do podstawowej wersji każdego z plików.

Git podchodzi do przechowywania danych w odmienny sposób. Traktuje on dane podobnie jak zestaw migawek (ang. snapshots) małego systemu plików. Za każdym razem jak tworzysz commit lub zapisujesz stan projektu, Git tworzy obraz przedstawiający to jak wyglądają wszystkie pliki w danym momencie i przechowuje referencję do tej migawki. W celu uzyskania dobrej wydajności, jeśli dany plik nie został zmieniony, Git nie zapisuje ponownie tego pliku, a tylko referencję do jego poprzedniej, identycznej wersji, która jest już zapisana. Git myśli o danych w sposób podobny do przedstawionego na Rysunku 1-5.



Rysunek 1-5. Git przechowuje dane jako migawki projektu w okresie czasu.

To jest istotna różnica pomiędzy Git i prawie wszystkimi innymi systemami VCS. Jej konsekwencją jest to, że Git rewiduje prawie wszystkie aspekty kontroli wersji, które pozostałe systemy po prostu kopiowały z poprzednich generacji. Powoduje także, że Git jest bardziej podobny do mini systemu plików ze zbudowanymi na nim potężnymi narzędziami, niż do zwykłego systemu VCS. Odkryjemy niektóre z zalet które zyskuje się poprzez myślenie o danych w ten sposób, gdy w trzecim rozdziale będziemy omawiać tworzenie gałęzi w Git.

### Niemal każda operacja jest lokalna

Większość operacji w Git do działania wymaga jedynie dostępu do lokalnych plików i zasobów, lub inaczej – nie są potrzebne żadne dane przechowywane na innym komputerze w sieci. Jeśli jesteś przyzwyczajony do systemów CVCS, w których większość operacji posiada narzut związany z dostępem sieciowym, ten aspekt Git sprawi, że uwierzysz w bogów szybkości, którzy musieli obdarzyć Git nieziemskimi mocami. Ponieważ kompletna historia projektu znajduje się w całości na Twoim dysku, odnosi się wrażenie, że większość operacji działa niemal natychmiast.

Przykładowo, w celu przeglądu historii projektu, Git nie musi łączyć się z serwerem, aby pobrać historyczne dane - zwyczajnie odczytuje je wprost z lokalnej bazy danych. Oznacza to, że dostęp do historii jest niemal natychmiastowy. Jeśli chcesz przejrzeć zmiany wprowadzone pomiędzy bieżącą wersją pliku, a jego stanem sprzed miesiąca, Git może odnaleźć wersję pliku sprzed miesiąca i dokonać lokalnego porównania. Nie musi w tym celu prosić serwera o wygenerowanie różnicy, czy też o udostępnienie wcześniejszej wersji pliku.

Oznacza to również, że można zrobić prawie wszystko będąc poza zasięgiem sieci lub firmowego VPNa. Jeśli masz ochotę popracować w samolocie lub pociągu, możesz bez problemu zatwierdzać kolejne zmiany, by w momencie połączenia z siecią przesłać komplet zmian na serwer. Jeśli pracujesz w domu, a klient VPN odmawia współpracy, nie musisz czekać z pilnymi zmianami. W wielu innych systemach taki sposób pracy jest albo niemożliwy, albo co najmniej uciążliwy. Przykładowo w Perforce, nie możesz wiele zdziałać bez połączenia z serwerem; w Subversion, albo CVS możesz edytować pliki, ale nie masz możliwości zatwierdzania zmian w repozytorium (ponieważ nie masz do niego dostępu). Może nie wydaje się to wielkim problemem, ale zdziwisz się pewnie jak wielką stanowi to różnicę w sposobie pracy.

### Git ma wbudowane mechanizmy spójności danych

Dla każdego obiektu Git wyliczana jest suma kontrolna przed jego zapisem i na podstawie tej sumy można od tej pory odwoływać się do danego obiektu. Oznacza to, że nie ma możliwości zmiany zawartości żadnego pliku, czy katalogu bez reakcji ze strony Git. Ta cecha wynika z wbudowanych, niskopoziomowych mechanizmów Git i stanowi integralną część jego filozofii. Nie ma szansy na utratę informacji, czy uszkodzenie zawartości pliku podczas przesyłania lub pobierania danych, bez możliwości wykrycia takiej sytuacji przez Git.

Mechanizmem, który wykorzystuje Git do wyznaczenia sumy kontrolnej jest tzw. skrót SHA-1. Jest to 40-znakowy łańcuch składający się z liczb szesnastkowych (0–9 oraz a–f), wyliczany na podstawie zawartości pliku lub struktury katalogu. Skrót SHA-1 wygląda mniej więcej tak:

24b9da6552252987aa493b52f8696cd6d3b00373

Pracując z Git będziesz miał styczność z takimi skrótami w wielu miejscach, ponieważ są one wykorzystywane cały czas. W rzeczywistości Git przechowuje wszystko nie pod postacią plików i ich nazw, ale we własnej bazie danych, w której kluczami są skróty SHA-1, a wartościami - zawartości plików, czy struktur katalogów.

### Standardowo Git wyłącznie dodaje nowe dane

Wykonując pracę z Git, niemal zawsze jedynie dodajemy dane do bazy danych Git. Bardzo trudno jest zmusić system do zrobienia czegoś, z czego nie można się następnie wycofać, albo sprawić, by niejawnie skasował jakieś dane. Podobnie jak w innych systemach VCS, można stracić lub nadpisać zmiany, które nie zostały jeszcze zatwierdzone; ale po zatwierdzeniu migawki do Git, bardzo trudno jest stracić te zmiany, zwłaszcza jeśli regularnie pchasz własną bazę danych Git do innego repozytorium.

Ta cecha sprawia, że praca z Git jest czystą przyjemnością, ponieważ wiemy, że możemy eksperymentować bez ryzyka zepsucia czegokolwiek. Więcej szczegółów na temat sposobu przechowywania danych przez Git oraz na temat mechanizmów odzyskiwania danych, które wydają się być utracone, znajduje się w rozdziale 9, "Mechanizmy wewnętrzne".

## Instalacja Git

Czas rozpocząć pracę z Git. Pierwszym krokiem jest instalacja. Można ją przeprowadzić na różne sposoby; po pierwsze można zainstalować Git ze źródeł, po drugie - można skorzystać z pakietu binarnego dla konkretnej platformy.

### Instalacja ze źródeł

Jeśli masz taką możliwość, korzystne jest zainstalowanie Git ze źródeł, ponieważ w ten sposób dostajesz najnowszą wersję. Każda wersja Git zawiera zwykle użyteczne zmiany w interfejsie, zatem chęć skorzystania z najnowszych funkcji stanowi zwykle najlepszy powód by skompilować samodzielnie własną wersję Git. Jest to istotne także z tego powodu, że wiele dystrybucji Linuksa posiada stare wersje pakietów; zatem jeśli nie korzystasz z najświeższej dystrybucji, albo nie aktualizujesz jej nowszymi pakietami, instalacja ze źródeł to najlepsza metoda.

Aby zainstalować Git, potrzebne są następujące biblioteki: curl, zlib, openssl, expat oraz libiconv. Przykładowo, jeśli korzystasz z systemu, który posiada narzędzie yum (np. Fedora) lub apt-get (np. system oparty na Debianie), możesz skorzystać z następujących poleceń w celu instalacji 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

Gdy wszystkie wymagane zależności zostaną zainstalowane, możesz pobrać najnowszą wersję Git ze strony:

http://git-scm.com/download

A następnie skompilować i zainstalować Git:

$ tar -zxf git-1.6.0.5.tar.gz

$ cd git-1.6.0.5

$ make prefix=/usr/local all

$ sudo make prefix=/usr/local install

Po instalacji masz również możliwość pobrania Git za pomocą samego Git:

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

### Instalacja w systemie Linux

Jeśli chcesz zainstalować Git w systemie Linux z wykorzystaniem pakietów binarnych, możesz to zrobić w standardowy sposób przy użyciu narzędzi zarządzania pakietami, specyficznych dla danej dystrybucji. Jeśli korzystasz z Fedory, możesz użyć narzędzia yum:

$ yum install git-core

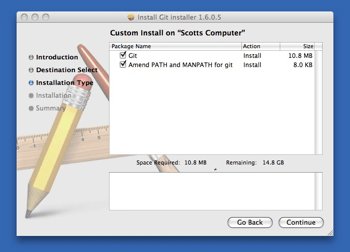
Jeśli korzystasz z dystrybucji opartej na Debianie (np. Ubuntu), użyj apt-get:

$ apt-get install git

### Instalacja na komputerze Mac

Istnieją dwa proste sposoby instalacji Git na komputerze Mac. Najprostszym z nich jest użycie graficznego instalatora, którego można pobrać z witryny SourceForge (patrz Ekran 1-7):

http://sourceforge.net/projects/git-osx-installer/



Rysunek 1-7. Instalator Git dla OS X.

Innym prostym sposobem jest instalacja Git z wykorzystaniem MacPorts (http://www.macports.org). Jeśli masz zainstalowane MacPorts, zainstaluj Git za pomocą

$ sudo port install git-core +svn +doc +bash\_completion +gitweb

Nie musisz instalować wszystkich dodatków, ale dobrym pomysłem jest dołączenie +svn w razie konieczności skorzystania z Git podczas pracy z repozytoriami Subversion (patrz Rozdział 8).

### Instalacja w systemie Windows

Instalacja Git w systemie Windows jest bardzo prosta. Projekt msysGit posiada jedną z najprostszych procedur instalacji. Po prostu pobierz program instalatora z witryny GitHub i uruchom go:

http://msysgit.github.com/

Po instalacji masz dostęp zarówno do wersji konsolowej, uruchamianej z linii poleceń (w tym do klienta SSH, który przyda się jeszcze później) oraz do standardowego GUI.

### Trzy stany

**Teraz uwaga**. To jedna z najważniejszych spraw do zapamiętania jeśli chodzi o pracę z Git, jeśli dalszy proces nauki ma przebiegać sprawnie. Git posiada trzy stany, w których mogą znajdować się pliki: zatwierdzony, zmodyfikowany i śledzony. Zatwierdzony oznacza, że dane zostały bezpiecznie zachowane w Twojej lokalnej bazie danych. Zmodyfikowany oznacza, że plik został zmieniony, ale zmiany nie zostały wprowadzone do bazy danych. Śledzony - oznacza, że zmodyfikowany plik został przeznaczony do zatwierdzenia w bieżącej postaci w następnej operacji commit.

Z powyższego wynikają trzy główne sekcje projektu Git: katalog Git, katalog roboczy i przechowalnia (ang. staging area).



Rysunek 1-6. Katalog roboczy, przechowalnia, katalog git.

Katalog Git jest miejscem, w którym Git przechowuje własne metadane oraz obiektową bazę danych Twojego projektu. To najważniejsza część Git i to właśnie ten katalog jest kopiowany podczas klonowania repozytorium z innego komputera.

Katalog roboczy stanowi obraz jednej wersji projektu. Zawartość tego katalogu pobierana jest ze skompresowanej bazy danych zawartej w katalogu Git i umieszczana na dysku w miejscu, w którym można ją odczytać lub zmodyfikować.

Przechowalnia to prosty plik, zwykle przechowywany w katalogu Git, który zawiera informacje o tym, czego dotyczyć będzie następna operacja commit. Czasami można spotkać się z określeniem indeks, ale ostatnio przyjęło się odwoływać do niego właśnie jako przechowalnia.

Podstawowy sposób pracy z Git wygląda mniej więcej tak:

1. Dokonujesz modyfikacji plików w katalogu roboczym.
2. Oznaczasz zmodyfikowane pliki jako śledzone, dodając ich bieżący stan (migawkę) do przechowalni.
3. Dokonujesz zatwierdzenia (commit), podczas którego zawartość plików z przechowalni zapisywana jest jako migawka projektu w katalogu Git.

Jeśli jakaś wersja pliku znajduje się w katalogu git, uznaje się ją jako zatwierdzoną. Jeśli plik jest zmodyfikowany, ale został dodany do przechowalni, plik jest śledzony. Jeśli zaś plik jest zmodyfikowany od czasu ostatniego pobrania, ale nie został dodany do przechowalni, plik jest w stanie zmodyfikowanym. W rozdziale 2 dowiesz się więcej o wszystkich tych stanach oraz o tym jak wykorzystać je do ułatwienia sobie pracy lub jak zupełnie pominąć przechowalnię.

## Wstępna konfiguracja Git

Teraz, gdy Git jest już zainstalowany w Twoim systemie, istotne jest wykonanie pewnych czynności konfiguracyjnych. Wystarczy to zrobić raz; konfiguracja będzie obowiązywać także po aktualizacji Git. Ustawienia można zmienić w dowolnym momencie jeszcze raz wykonując odpowiednie polecenia.

Git posiada narzędzie zwane git config, które pozwala odczytać, bądź zmodyfikować zmienne, które kontrolują wszystkie aspekty działania i zachowania Git. Zmienne te mogą być przechowywane w trzech różnych miejscach:

* plik /etc/gitconfig: Zawiera wartości zmiennych widoczne dla każdego użytkownika w systemie oraz dla każdego z ich repozytoriów. Jeśli dodasz opcję --system do polecenia git config, odczytane bądź zapisane zostaną zmienne z tej właśnie lokalizacji.
* plik ~/.gitconfig: Lokalizacja specyficzna dla danego użytkownika. Za pomocą opcji --global można uzyskać dostęp do tych właśnie zmiennych.
* plik konfiguracyjny w katalogu git (tzn. .git/config) bieżącego repozytorium: zawiera konfigurację charakterystyczną dla tego konkretnego repozytorium. Każdy poziom ma priorytet wyższy niż poziom poprzedni, zatem wartości zmiennych z pliku .git/config przesłaniają wartości zmiennych z pliku /etc/gitconfig.

W systemie Windows, Git poszukuje pliku .gitconfig w katalogu %HOME% (C:\Documents and Settings\%USERNAME% w większości przypadków). Sprawdza również istnienie pliku /etc/gitconfig, choć w tym wypadku katalog ten jest katalogiem względnym do katalogu instalacji MSysGit.

### Twoja tożsamość

Pierwszą rzeczą, którą warto wykonać po instalacji Git jest konfiguracja własnej nazwy użytkownika oraz adresu e-mail. Jest to ważne, ponieważ każda operacja zatwierdzenia w Git korzysta z tych informacji, które stają się integralną częścią zatwierdzeń przesyłanych i pobieranych później do i z serwera:

$ git config --global user.name "Jan Nowak"

$ git config --global user.email jannowak@example.com

Jeśli skorzystasz z opcji --global wystarczy, że taka konfiguracja zostanie dokonana jednorazowo. Git skorzysta z niej podczas każdej operacji wykonywanej przez Ciebie w danym systemie. Jeśli zaistnieje potrzeba zmiany tych informacji dla konkretnego projektu, można skorzystać z git config bez opcji --global.

### Edytor

Teraz, gdy ustaliłeś swą tożsamość, możesz skonfigurować domyślny edytor tekstu, który zostanie uruchomiony, gdy Git będzie wymagał wprowadzenia jakiejś informacji tekstowej. Domyślnie Git skorzysta z domyślnego edytora systemowego, którym zazwyczaj jest Vi lub Vim. Jeśli wolisz korzystać z innego edytora, np. z Emacsa, uruchom następujące polecenie:

$ git config --global core.editor emacs

### Narzędzie obsługi różnic

Warto również skonfigurować domyślne narzędzie do rozstrzygania różnic i problemów podczas edycji konfliktów powstałych w czasie operacji łączenia (ang. merge). Jeśli chcesz wykorzystywać w tym celu narzędzie vimdiff, użyj polecenia:

$ git config --global merge.tool vimdiff

Git zna narzędzia kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, oraz opendiff. Możesz również użyć własnego narzędzia; rozdział 7 zawiera więcej informacji na ten temat.

### Sprawdzanie ustawień

Jeśli chcesz sprawdzić bieżące ustawienia, wykonaj polecenie git config --list. Git wyświetli pełną konfigurację:

$ git config --list

user.name=Scott Chacon

user.email=schacon@gmail.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

Niektóre zmienne mogą pojawić się wiele razy, ponieważ Git odczytuje konfigurację z różnych plików (choćby z /etc/gitconfig oraz ~/.gitconfig). W takim wypadku Git korzysta z ostatniej wartości dla każdej unikalnej zmiennej, którą znajdzie.

Można również sprawdzić jaka jest rzeczywista wartość zmiennej o konkretnej nazwie za pomocą polecenia git config {zmienna}:

$ git config user.name

Scott Chacon

## Uzyskiwanie pomocy

Jeśli kiedykolwiek będziesz potrzebować pomocy podczas pracy z Git, istnieją trzy sposoby wyświetlenia strony podręcznika dla każdego z poleceń Git:

$ git help <polecenie>

$ git <polecenie> --help

$ man git-<polecenie>

Przykładowo, pomoc dotyczącą konfiguracji można uzyskać wpisując:

$ git help config

Polecenia te mają tę przyjemną cechę, że można z nich korzystać w każdej chwili, nawet bez połączenia z Internetem. Jeśli standardowy podręcznik oraz niniejsza książka to za mało i potrzebna jest pomoc osobista, zawsze możesz sprawdzić kanał #git lub #github na serwerze IRC Freenode (irc.freenode.net). Kanały te są nieustannie oblegane przez setki osób, które mają duże doświadczenie z pracą z Git i często chętnie udzielają pomocy.

## Podsumowanie

W tym momencie powinieneś posiadać podstawowy pogląd na to czym jest Git i czym różni się od scentralizowanych systemów kontroli wersji, do których być może jesteś przyzwyczajony. Powinieneś również mieć dostęp do działającej wersji Git na własnym komputerze, której konfiguracja została zainicjowana Twoimi danymi personalnymi. Nadszedł czas by poznać podstawy pracy z Git.

# Podstawy Gita

Jeśli chcesz ograniczyć się do czytania jednego rozdziału, dobrze trafiłeś. Niniejszy rozdział obejmuje wszystkie podstawowe polecenia, które musisz znać, aby wykonać przeważającą część zadań, z którymi przyjdzie ci spędzić czas podczas pracy z Gitem. Po zapoznaniu się z rozdziałem powinieneś umieć samodzielnie tworzyć i konfigurować repozytoria, rozpoczynać i kończyć śledzenie plików, umieszczać zmiany w poczekalni oraz je zatwierdzać. Pokażemy ci także, jak skonfigurować Gita tak, aby ignorował pewne pliki oraz całe ich grupy według zadanego wzorca, szybko i łatwo cofać błędne zmiany, przeglądać historię swojego projektu, podglądać zmiany pomiędzy rewizjami, oraz jak wypychać je na serwer i stamtąd pobierać.

## Pierwsze repozytorium Gita

Projekt Gita możesz rozpocząć w dwojaki sposób. Pierwsza metoda używa istniejącego projektu lub katalogu i importuje go do Gita. Druga polega na sklonowaniu istniejącego repozytorium z innego serwera.

### Inicjalizacja Gita w istniejącym katalogu

Jeśli chcesz rozpocząć śledzenie zmian w plikach istniejącego projektu, musisz przejść do katalogu projektu i wykonać polecenie

$ git init

To polecenie stworzy nowy podkatalog o nazwie .git, zawierający wszystkie niezbędne pliki — szkielet repozytorium Gita. W tym momencie żadna część twojego projektu nie jest jeszcze śledzona. (Zajrzyj do Rozdziału 9. aby dowiedzieć się, jakie dokładnie pliki są przechowywane w podkatalogu .git, który właśnie utworzyłeś).

Aby rozpocząć kontrolę wersji istniejących plików (w przeciwieństwie do pustego katalogu), najprawdopodobniej powinieneś rozpocząć ich śledzenie i utworzyć początkową rewizję. Możesz tego dokonać kilkoma poleceniami add (dodaj) wybierając pojedyncze pliki, które chcesz śledzić, a następnie zatwierdzając zmiany poleceniem commit:

$ git add \*.c

$ git add README

$ git commit -m 'initial project version'

Za chwilę zobaczymy dokładnie, co wymienione polecenia robią. W tym jednak momencie masz już własne repozytorium Gita, śledzące wybrane pliki i zawierające pierwsze zatwierdzone zmiany (początkową rewizję).

### Klonowanie istniejącego repozytorium

Jeżeli chcesz uzyskać kopię istniejącego już repozytorium Gita — na przykład projektu, w którym chciałbyś zacząć się udzielać i wprowadzać własne zmiany — polecenie, którego potrzebujesz to clone. Jeżeli znasz już inne systemy kontroli wersji, jak np. Subversion, zauważysz z pewnością, że w przypadku Gita używane polecenie to clone a nie checkout. Jest to istotne rozróżnienie — Git pobiera kopię niemalże wszystkich danych posiadanych przez serwer. Po wykonaniu polecenia git clone zostanie pobrana każda rewizja, każdego pliku w historii projektu. W praktyce nawet jeśli dysk serwera zostanie uszkodzony, możesz użyć któregokolwiek z dostępnych klonów aby przywrócić serwer do stanu w jakim był w momencie klonowania (możesz utracić pewne hooki skonfigurowane na serwerze i tym podobne, ale wszystkie poddane kontroli wersji pliki będą spójne — zajrzyj do Rozdziału 4. aby poznać więcej szczegółów).

Repozytorium klonujesz używając polecenia git clone [URL]. Na przykład jeśli chcesz sklonować bibliotekę Rubiego do Gita o nazwie Grit, możesz to zrobić wywołując:

$ git clone git://github.com/schacon/grit.git

Tworzony jest katalog o nazwie „grit”, następnie wewnątrz niego inicjowany jest podkatalog .git, pobierane są wszystkie dane z repozytorium, a kopia robocza przełączona zostaje na ostatnią wersję. Jeśli wejdziesz do świeżo utworzonego katalogu grit, zobaczysz wewnątrz pliki projektu, gotowe do użycia i pracy z nimi. Jeśli chcesz sklonować repozytorium do katalogu o nazwie innej niż grit, możesz to zrobić podając w wierszu poleceń kolejną opcję:

$ git clone git://github.com/schacon/grit.git mygrit

Powyższe polecenie robi dokładnie to samo, co poprzednia, ale wszystkie pliki umieszcza w katalogu mygrit.

Git oferuje do wyboru zestaw różnych protokołów transmisji. Poprzedni przykład używa protokołu git://, ale możesz także spotkać http(s):// lub uzytkownik@serwer:/sciezka.git, używające protokołu SSH. W Rozdziale 4. omówimy wszystkie dostępne możliwości konfiguracji dostępu do repozytorium Gita na serwerze oraz zalety i wady każdej z nich.

## Rejestrowanie zmian w repozytorium

Posiadasz już repozytorium Gita i ostatnią wersję lub kopię roboczą wybranego projektu. Za każdym razem, kiedy po naniesieniu zmian projekt osiągnie stan, który chcesz zapamiętać, musisz nowe wersje plików zatwierdzić w swoim repozytorium.

Pamiętaj, że każdy plik w twoim katalogu roboczym może być w jednym z dwóch stanów: śledzony lub nieśledzony. Śledzone pliki to te, które znalazły się w ostatniej migawce; mogą być niezmodyfikowane, zmodyfikowane lub oczekiwać w poczekalni. Nieśledzone pliki to cała reszta — są to jakiekolwiek pliki w twoim katalogu roboczym, które nie znalazły się w ostatniej migawce i nie znajdują się w poczekalni, gotowe do zatwierdzenia. Początkowo, kiedy klonujesz repozytorium, wszystkie twoje pliki będą śledzone i niezmodyfikowane, ponieważ dopiero co zostały wybrane i nie zmieniałeś jeszcze niczego.

Kiedy zmieniasz pliki, Git rozpoznaje je jako zmodyfikowane, ponieważ różnią się od ostatniej zatwierdzonej zmiany. Zmodyfikowane pliki umieszczasz w poczekalni, a następnie zatwierdzasz oczekujące tam zmiany i tak powtarza się cały cykl. Przedstawia go Diagram 2-1.



Rysunek 2-1. Cykl życia stanu twoich plików.

### Sprawdzanie stanu twoich plików

Podstawowe narzędzie używane do sprawdzenia stanu plików to polecenie git status. Jeśli uruchomisz je bezpośrednio po sklonowaniu repozytorium, zobaczysz wynik podobny do poniższego:

$ git status

# On branch master

nothing to commit, working directory clean

Oznacza to, że posiadasz czysty katalog roboczy — innymi słowy nie zawiera on śledzonych i zmodyfikowanych plików. Git nie widzi także żadnych plików nieśledzonych, w przeciwnym wypadku wyświetliłby ich listę. W końcu polecenie pokazuje również gałąź, na której aktualnie pracujesz. Póki co, jest to zawsze master, wartość domyślna; nie martw się tym jednak teraz. Następny rozdział w szczegółach omawia gałęzie oraz odniesienia.

Powiedzmy, że dodajesz do repozytorium nowy, prosty plik README. Jeżeli nie istniał on wcześniej, po uruchomieniu git status zobaczysz go jako plik nieśledzony, jak poniżej:

$ vim README

$ git status

# On branch master

# Untracked files:

# (use "git add <file>..." to include in what will be committed)

#

# README

nothing added to commit but untracked files present (use "git add" to track)

Widać, że twój nowy plik README nie jest jeszcze śledzony, ponieważ znajduje się pod nagłówkiem „Untracked files” (Nieśledzone pliki) w informacji o stanie. Nieśledzony oznacza, że Git widzi plik, którego nie miałeś w poprzedniej migawce (zatwierdzonej kopii); Git nie zacznie umieszczać go w przyszłych migawkach, dopóki sam mu tego nie polecisz. Zachowuje się tak, by uchronić cię od przypadkowego umieszczenia w migawkach wyników działania programu lub innych plików, których nie miałeś zamiaru tam dodawać. W tym przypadku chcesz, aby README został uwzględniony, więc zacznijmy go śledzić.

### Śledzenie nowych plików

Aby rozpocząć śledzenie nowego pliku, użyj polecenia git add. Aby zacząć śledzić plik README, możesz wykonać:

$ git add README

Jeśli uruchomisz teraz ponownie polecenie status, zobaczysz, że twój plik README jest już śledzony i znalazł się w poczekalni:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

#

Widać, że jest w poczekalni, ponieważ znajduje się pod nagłówkiem „Changes to be commited“ (Zmiany do zatwierdzenia). Jeśli zatwierdzisz zmiany w tym momencie, jako migawka w historii zostanie zapisana wersja pliku z momentu wydania polecenia git add. Być może pamiętasz, że po uruchomieniu git init wydałeś polecenie git add (pliki) — miało to na celu rozpoczęcie ich śledzenia. Polecenie git add bierze jako parametr ścieżkę do pliku lub katalogu; jeśli jest to katalog, polecenie dodaje wszystkie pliki z tego katalogu i podkatalogów.

### Dodawanie zmodyfikowanych plików do poczekalni

Zmodyfikujmy teraz plik, który był już śledzony. Jeśli zmienisz śledzony wcześniej plik o nazwie benchmarks.rb, a następnie uruchomisz polecenie status, zobaczysz coś podobnego:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: benchmarks.rb

#

Plik benchmarks.rb pojawia się w sekcji „Changes not staged for commit“ (Zmienione ale nie zaktualizowane), co oznacza, że śledzony plik został zmodyfikowany, ale zmiany nie trafiły jeszcze do poczekalni. Aby je tam wysłać, uruchom polecenie git add (jest to wielozadaniowe polecenie — używa się go do rozpoczynania śledzenia nowych plików, umieszczania ich w poczekalni, oraz innych zadań, takich jak oznaczanie rozwiązanych konfliktów scalania). Uruchom zatem git add by umieścić benchmarks.rb w poczekalni, a następnie ponownie wykonaj git status:

$ git add benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

# modified: benchmarks.rb

#

Oba pliki znajdują się już w poczekalni i zostaną uwzględnione podczas kolejnego zatwierdzenia zmian. Załóżmy, że w tym momencie przypomniałeś sobie o dodatkowej małej zmianie, którą koniecznie chcesz wprowadzić do pliku benchmarks.rb jeszcze przed zatwierdzeniem. Otwierasz go zatem, wprowadzasz zmianę i jesteś gotowy do zatwierdzenia. Uruchom jednak git status raz jeszcze:

$ vim benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

# modified: benchmarks.rb

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: benchmarks.rb

#

Co do licha? Plik benchmarks.rb widnieje teraz jednocześnie w poczekalni i poza nią. Jak to możliwe? Okazuje się, że Git umieszcza plik w poczekalni dokładnie z taką zawartością, jak w momencie uruchomienia polecenia git add. Jeśli w tej chwili zatwierdzisz zmiany, zostanie użyta wersja benchmarks.rb dokładnie z momentu uruchomienia polecenia git add, nie zaś ta, którą widzisz w katalogu roboczym w momencie wydania polecenia git commit. Jeśli modyfikujesz plik po uruchomieniu git add, musisz ponownie użyć git add, aby najnowsze zmiany zostały umieszczone w poczekalni:

$ git add benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

# modified: benchmarks.rb

#

### Ignorowanie plików

Często spotkasz się z klasą plików, w przypadku których nie chcesz, by Git automatycznie dodawał je do repozytorium, czy nawet pokazywał je jako nieśledzone. Są to ogólnie pliki generowane automatycznie, takie jak dzienniki zdarzeń, czy pliki tworzone w czasie budowania projektu. W takich wypadkach tworzysz plik zawierający listę wzorców do nich pasujących i nazywasz go .gitignore. Poniżej znajdziesz przykładowy plik .gitignore:

$ cat .gitignore

\*.[oa]

\*~

Pierwsza linia mówi Gitowi, by ignorował pliki kończące się na .o lub .a — pliki obiektów i archiwa, które mogą być produktem kompilacji kodu. Druga linia mówi Gitowi, żeby pomijał również wszystkie pliki, które nazwy kończą się tyldą (~), której to używa wiele edytorów tekstu, takich jak Emacs, do oznaczania plików tymczasowych. Możesz też dołączyć katalog log, tmp lub pid, automatycznie wygenerowaną dokumentację itp. Zajęcie się plikiem .gitignore jeszcze przed przystąpieniem do pracy jest zwykle dobrym pomysłem i pozwoli ci uniknąć przypadkowego dodania do repozytorium Git niechcianych plików.

Zasady przetwarzania wyrażeń, które możesz umieścić w pliku .gitignore są następujące:

* Puste linie lub linie rozpoczynające się od # są ignorowane.
* Działają standardowe wyrażenia glob.
* Możesz zakończyć wyrażenie znakiem ukośnika (/) aby sprecyzować, że chodzi o katalog.
* Możesz negować wyrażenia rozpoczynając je wykrzyknikiem (!).

Wyrażenia glob są jak uproszczone wyrażenia regularne, używane przez powłokę. Gwiazdka (\*) dopasowuje zero lub więcej znaków; [abc] dopasowuje dowolny znak znajdujący się wewnątrz nawiasu kwadratowego (w tym przypadku a, b lub c); znak zapytania (?) dopasowuje pojedynczy znak; nawias kwadratowy zawierający znaki rozdzielone myślnikiem ([0-9]) dopasowuje dowolny znajdujący się pomiędzy nimi znak (w tym przypadku od 0 do 9).

Poniżej znajdziesz kolejny przykład pliku .gitignore:

# komentarz — ta linia jest ignorowana

# żadnych plików .a

\*.a

# ale uwzględniaj lib.a, pomimo ignorowania .a w linijce powyżej

!lib.a

# ignoruj plik TODO w katalogu głównym, ale nie podkatalog/TODO

/TODO

# ignoruj wszystkie pliki znajdujące się w katalogu build/

build/

# ignoruj doc/notatki.txt, ale nie doc/server/arch.txt

doc/\*.txt

### Podgląd zmian w poczekalni i poza nią

Jeśli polecenie git status jest dla ciebie zbyt nieprecyzyjne — chcesz wiedzieć, co dokładnie zmieniłeś, nie zaś, które pliki zostały zmienione — możesz użyć polecenia git diff. W szczegółach zajmiemy się nim później; prawdopodobnie najczęściej będziesz używał go aby uzyskać odpowiedź na dwa pytania: Co zmieniłeś, ale jeszcze nie trafiło do poczekalni? Oraz, co znajduje się już w poczekalni, a co za chwilę zostanie zatwierdzone? Choć git status bardzo ogólnie odpowiada na oba te pytania, git diff pokazuje, które dokładnie linie zostały dodane, a które usunięte — w postaci łatki.

Powiedzmy, że zmieniłeś i ponownie dodałeś do poczekalni plik README, a następnie zmodyfikowałeś plik benchmarks.rb, jednak bez umieszczania go wśród oczekujących. Jeśli uruchomisz teraz polecenie status, zobaczysz coś podobnego:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: benchmarks.rb

#

Aby zobaczyć, co zmieniłeś ale nie wysłałeś do poczekalni, wpisz git diff bez żadnych argumentów:

$ git diff

diff --git a/benchmarks.rb b/benchmarks.rb

index 3cb747f..da65585 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -36,6 +36,10 @@ def main

@commit.parents[0].parents[0].parents[0]

end

+ run\_code(x, 'commits 1') do

+ git.commits.size

+ end

+

run\_code(x, 'commits 2') do

log = git.commits('master', 15)

log.size

Powyższe polecenie porównuje zawartość katalogu roboczego z tym, co znajduje się w poczekalni. Wynik pokaże ci te zmiany, które nie trafiły jeszcze do poczekalni.

Jeśli chcesz zobaczyć zawartość poczekalni, która trafi do repozytorium z najbliższym zatwierdzeniem, możesz użyć polecenia git diff --cached. (Git w wersji 1.6.1 i późniejszych pozawala użyć polecenia git diff --staged, które może być łatwiejsze do zapamiętania). To polecenie porówna zmiany z poczekalni z ostatnią migawką:

$ git diff --cached

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README2

@@ -0,0 +1,5 @@

+grit

+ by Tom Preston-Werner, Chris Wanstrath

+ http://github.com/mojombo/grit

+

+Grit is a Ruby library for extracting information from a Git repository

Istotnym jest, że samo polecenie git diff nie pokazuje wszystkich zmian dokonanych od ostatniego zatwierdzenia — ­jedynie te, które nie trafiły do poczekalni. Może być to nieco mylące, ponieważ jeżeli wszystkie twoje zmiany są już w poczekalni, wynik git diff będzie pusty.

Jeszcze jeden przykład — jeżeli wyślesz do poczekalni plik benchmarks.rb, a następnie zmodyfikujesz go ponownie, możesz użyć git status, by obejrzeć zmiany znajdujące się w poczekalni, jak i te poza nią:

$ git add benchmarks.rb

$ echo '# test line' >> benchmarks.rb

$ git status

# On branch master

#

# Changes to be committed:

#

# modified: benchmarks.rb

#

# Changes not staged for commit:

#

# modified: benchmarks.rb

#

Teraz możesz użyć git diff, by zobaczyć zmiany spoza poczekalni

$ git diff

diff --git a/benchmarks.rb b/benchmarks.rb

index e445e28..86b2f7c 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -127,3 +127,4 @@ end

main()

##pp Grit::GitRuby.cache\_client.stats

+# test line

oraz git diff --cached, aby zobaczyć zmiany wysłane dotąd do poczekalni:

$ git diff --cached

diff --git a/benchmarks.rb b/benchmarks.rb

index 3cb747f..e445e28 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -36,6 +36,10 @@ def main

@commit.parents[0].parents[0].parents[0]

end

+ run\_code(x, 'commits 1') do

+ git.commits.size

+ end

+

run\_code(x, 'commits 2') do

log = git.commits('master', 15)

log.size

### Zatwierdzanie zmian

Teraz, kiedy twoja poczekalnia zawiera dokładnie to, co powinna, możesz zatwierdzić swoje zmiany. Pamiętaj, że wszystko czego nie ma jeszcze w poczekalni — każdy plik, który utworzyłeś lub zmodyfikowałeś, a na którym później nie uruchomiłeś polecenia git add — nie zostanie uwzględnione wśród zatwierdzanych zmian. Pozostanie wyłącznie w postaci zmodyfikowanych plików na twoim dysku.

W tym wypadku, kiedy ostatnio uruchamiałeś git status, zobaczyłeś, że wszystkie twoje zmiany są już w poczekalni, więc jesteś gotowy do ich zatwierdzenia. Najprostszy sposób zatwierdzenia zmian to wpisanie git commit:

$ git commit

Zostanie uruchomiony wybrany przez ciebie edytor tekstu. (Wybiera się go za pośrednictwem zmiennej środowiskową $EDITOR — zazwyczaj jest to vim lub emacs, możesz jednak wybrać własną aplikację używając polecenia git config --global core.editor, które poznałeś w Rozdziale 1.).

Edytor zostanie otwarty z następującym tekstem (poniższy przykład pokazuje ekran Vima):

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

# modified: benchmarks.rb

~

~

~

".git/COMMIT\_EDITMSG" 10L, 283C

Jak widzisz, domyślny opis zmian zawiera aktualny wynik polecenia git status w postaci komentarza oraz jedną pustą linię na samej górze. Możesz usunąć komentarze i wpisać własny opis, lub pozostawić je, co pomoże zapamiętać zakres zatwierdzonych zmian. (Aby uzyskać jeszcze precyzyjniejsze przypomnienie, możesz przekazać do git commit parametr -v. Jeśli to zrobisz, do komentarza trafią również poszczególne zmodyfikowane wiersze, pokazując, co dokładnie zrobiłeś.). Po opuszczeniu edytora, Git stworzy nową migawkę opatrzoną twoim opisem zmian (uprzednio usuwając z niego komentarze i podsumowanie zmian).

Alternatywnie opis rewizji możesz podać już wydając polecenie commit, poprzedzając go flagą -m, jak poniżej:

$ git commit -m "Story 182: Fix benchmarks for speed"

[master]: created 463dc4f: "Fix benchmarks for speed"

2 files changed, 3 insertions(+), 0 deletions(-)

create mode 100644 README

Właśnie zatwierdziłeś swoje pierwsze zmiany! Sama operacja rewizji zwróciła dodatkowo garść informacji, między innymi, gałąź do której dorzuciłeś zmiany (master), ich sumę kontrolną SHA-1 (463dc4f), ilość zmienionych plików oraz statystyki dodanych i usuniętych linii kodu.

Pamiętaj, że operacja commit zapamiętała migawkę zmian z poczekalni. Wszystko czego nie dodałeś do poczekalni, ciągle czeka zmienione w swoim miejscu - możesz to uwzględnić przy następnym zatwierdzaniu zmian. Każdorazowe wywołanie polecenia git commit powoduje zapamiętanie migawki projektu, którą możesz następnie odtworzyć albo porównać do innej migawki.

### Pomijanie poczekalni

Chociaż poczekalnia może być niesamowicie przydatna przy ustalaniu rewizji dokładnie takich, jakimi chcesz je mieć później w historii, czasami możesz uznać ją za odrobinę zbyt skomplikowaną aniżeli wymaga tego twoja praca. Jeśli chcesz pominąć poczekalnię, Git udostępnia prosty skrót. Po dodaniu do składni polecenia git commit opcji -a każdy zmieniony plik, który jest już śledzony, automatycznie trafi do poczekalni, dzięki czemu pominiesz część git add:

$ git status

# On branch master

#

# Changes not staged for commit:

#

# modified: benchmarks.rb

#

$ git commit -a -m 'added new benchmarks'

[master 83e38c7] added new benchmarks

1 files changed, 5 insertions(+), 0 deletions(-)

Zauważ, że w tym wypadku przed zatwierdzeniem zmian i wykonaniem rewizji nie musiałeś uruchamiać git add na pliku banchmark.rb.

### Usuwanie plików

Aby usunąć plik z Gita, należy go najpierw wyrzucić ze zbioru plików śledzonych, a następnie zatwierdzić zmiany. Służy do tego polecenie git rm, które dodatkowo usuwa plik z katalogu roboczego. Nie zobaczysz go już zatem w sekcji plików nieśledzonych przy następnej okazji.

Jeżeli po prostu usuniesz plik z katalogu roboczego i wykonasz polecenie git status zobaczysz go w sekcji "Zmienione ale nie zaktualizowane" (Changes not staged for commit) (czyli, poza poczekalnią):

$ rm grit.gemspec

$ git status

# On branch master

#

# Changes not staged for commit:

# (use "git add/rm <file>..." to update what will be committed)

#

# deleted: grit.gemspec

#

W dalszej kolejności, uruchomienie git rm doda do poczekalni operację usunięcia pliku:

$ git rm grit.gemspec

rm 'grit.gemspec'

$ git status

# On branch master

#

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# deleted: grit.gemspec

#

Przy kolejnej rewizji, plik zniknie i nie będzie dłużej śledzony. Jeśli zmodyfikowałeś go wcześniej i dodałeś już do indeksu oczekujących zmian, musisz wymusić usunięcie opcją -f. Spowodowane jest to wymogami bezpieczeństwa, aby uchronić cię przed usunięciem danych, które nie zostały jeszcze zapamiętane w żadnej migawce i które później nie będą mogły być odtworzone z repozytorium Gita.

Kolejną przydatną funkcją jest możliwość zachowywania plików w drzewie roboczym ale usuwania ich z poczekalni. Innymi słowy, możesz chcieć trzymać plik na dysku ale nie chcieć, żeby Git go dalej śledził. Jest to szczególnie przydatne w sytuacji kiedy zapomniałeś dodać czegoś do .gitignore i przez przypadek umieściłeś w poczekalni np. duży plik dziennika lub garść plików .a. Wystarczy wówczas wywołać polecenie rm wraz opcją --cached:

$ git rm --cached readme.txt

Do polecenia git -rm możesz przekazywać pliki, katalogi lub wyrażenia glob - możesz na przykład napisać coś takiego:

$ git rm log/\\*.log

Zwróć uwagę na odwrotny ukośnik (\) na początku \*. Jest on niezbędny gdyż Git dodatkowo do tego co robi powłoka, sam ewaluuje sobie nazwy plików. Przywołane polecenie usuwa wszystkie pliki z rozszerzeniem .log, znajdujące się w katalogu log/. Możesz także wywołać następujące polecenie:

$ git rm \\*~

Usuwa ona wszystkie pliki, które kończą się tyldą ~.

### Przenoszenie plików

W odróżnieniu do wielu innych systemów kontroli wersji, Git nie śledzi bezpośrednio przesunięć plików. Nie przechowuje on żadnych metadanych, które mogłyby mu pomóc w rozpoznawaniu operacji zmiany nazwy śledzonych plików. Jednakże, Git jest całkiem sprytny jeżeli chodzi o rozpoznawanie tego po fakcie - zajmiemy się tym tematem odrobinę dalej.

Nieco mylący jest fakt, że Git posiada polecenie mv. Służy ono do zmiany nazwy pliku w repozytorium, np.

$ git mv file\_from file\_to

W rzeczywistości, uruchomienie takiego polecenia spowoduje, że Git zapamięta w poczekalni operację zmiany nazwy - możesz to sprawdzić wyświetlając wynik operacji status:

$ git mv README.txt README

$ git status

# On branch master

# Your branch is ahead of 'origin/master' by 1 commit.

#

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# renamed: README.txt -> README

#

Jest to równoważne z uruchomieniem poleceń:

$ mv README.txt README

$ git rm README.txt

$ git add README

Git rozpozna w tym przypadku, że jest to operacja zmiany nazwy - nie ma zatem znaczenia, czy zmienisz ją w ten czy opisany wcześniej (mv) sposób. Jedyna realna różnica polega na tym, że mv to jedno polecenie zamiast trzech - kwestia wygody. Co ważniejsze, samą nazwę możesz zmienić dowolnym narzędziem a resztą zajmą się już polecenia add i rm których musisz użyć przed zatwierdzeniem zmian.

## Podgląd historii rewizji

Po kilku rewizjach, lub w przypadku sklonowanego repozytorium zawierającego już własną historię, przyjdzie czas, że będziesz chciał spojrzeć w przeszłość i sprawdzić dokonane zmiany. Najprostszym, a zarazem najsilniejszym, służącym do tego narzędziem jest git log.

Poniższe przykłady operują na moim, bardzo prostym, demonstracyjnym projekcie o nazwie simplegit. Aby go pobrać uruchom:

git clone git://github.com/schacon/simplegit-progit.git

Jeśli teraz uruchomisz na sklonowanym repozytorium polecenie git log, uzyskasz mniej więcej coś takiego:

$ git log

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

Domyślnie, polecenie git log uruchomione bez argumentów, listuje zmiany zatwierdzone w tym repozytorium w odwrotnej kolejności chronologicznej, czyli pokazując najnowsze zmiany w pierwszej kolejności. Jak widzisz polecenie wyświetliło zmiany wraz z ich sumą kontrolną SHA-1, nazwiskiem oraz e-mailem autora, datą zapisu oraz notką zmiany.

Duża liczba opcji polecenia git log oraz ich różnorodność pozwalają na dokładne wybranie interesujących nas informacji. Za chwilę przedstawimy najważniejsze i najczęściej używane spośród nich.

Jedną z najprzydatniejszych opcji jest -p. Pokazuje ona różnice wprowadzone z każdą rewizją. Dodatkowo możesz użyć opcji -2 aby ograniczyć zbiór do dwóch ostatnich wpisów:

$ git log -p -2

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

diff --git a/Rakefile b/Rakefile

index a874b73..8f94139 100644

--- a/Rakefile

+++ b/Rakefile

@@ -5,7 +5,7 @@ require 'rake/gempackagetask'

spec = Gem::Specification.new do |s|

- s.version = "0.1.0"

+ s.version = "0.1.1"

s.author = "Scott Chacon"

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index a0a60ae..47c6340 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -18,8 +18,3 @@ class SimpleGit

end

end

-

-if $0 == \_\_FILE\_\_

- git = SimpleGit.new

- puts git.show

-end

\ No newline at end of file

Opcja spowodowała wyświetlenie tych samych informacji z tą różnicą, że bezpośrednio po każdym wpisie został pokazywany tzw. diff, czyli różnica. Jest to szczególnie przydatne podczas recenzowania kodu albo szybkiego przeglądania zmian dokonanych przez twojego współpracownika. Dodatkowo możesz skorzystać z całej serii opcji podsumowujących wynik działania git log. Na przykład, aby zobaczyć skrócone statystyki każdej z zatwierdzonych zmian, użyj opcji --stat:

$ git log --stat

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

Rakefile | 2 +-

1 files changed, 1 insertions(+), 1 deletions(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

lib/simplegit.rb | 5 -----

1 files changed, 0 insertions(+), 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

README | 6 ++++++

Rakefile | 23 +++++++++++++++++++++++

lib/simplegit.rb | 25 +++++++++++++++++++++++++

3 files changed, 54 insertions(+), 0 deletions(-)

Jak widzisz, --stat wyświetlił pod każdym wpisem historii listę zmodyfikowanych plików, liczbę zmienionych plików oraz liczbę dodanych i usuniętych linii. Dodatkowo, opcja dołożyła podobne podsumowanie wszystkich informacji na samym końcu wyniku. Kolejnym bardzo przydatnym parametrem jest --pretty. Pokazuje on wynik polecenia log w nowym, innym niż domyślny formacie. Możesz skorzystać z kilku pre-definiowanych wariantów. Opcja oneline wyświetla każdą zatwierdzoną zmianę w pojedynczej linii, co szczególnie przydaje się podczas wyszukiwania w całym gąszczu zmian. Dodatkowo, short, full oraz fuller pokazują wynik w mniej więcej tym samym formacie ale odpowiednio z odrobiną więcej lub mniej informacji:

$ git log --pretty=oneline

ca82a6dff817ec66f44342007202690a93763949 changed the version number

085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code

a11bef06a3f659402fe7563abf99ad00de2209e6 first commit

Najbardziej interesująca jest tutaj jednak opcja format. Pozwala ona określić własny wygląd i format informacji wyświetlanych poleceniem log. Funkcja przydaje się szczególnie podczas generowania tychże informacji do dalszego, maszynowego przetwarzania - ponieważ sam definiujesz ściśle format, wiesz, że nie zmieni się on wraz z kolejnymi wersjami Gita:

$ git log --pretty=format:"%h - %an, %ar : %s"

ca82a6d - Scott Chacon, 11 months ago : changed the version number

085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code

a11bef0 - Scott Chacon, 11 months ago : first commit

Tabela 2-1 pokazuje najprzydatniejsze opcje akceptowane przez format.

| **Opcja** | **Opis** |
| --- | --- |
| %H | Suma kontrolna zmiany |
| %h | Skrócona suma kontrolna zmiany |
| %T | Suma kontrolna drzewa |
| %t | Skrócona suma kontrolna drzewa |
| %P | Sumy kontrolne rodziców |
| %p | Skrócone sumy kontrolne rodziców |
| %an | Nazwisko autora |
| %ae | Adres e-mail autora |
| %ad | Data autora (format respektuje opcję -date=) |
| %ar | Względna data autora |
| %cn | Nazwisko zatwierdzającego zmiany |
| %ce | Adres e-mail zatwierdzającego zmiany |
| %cd | Data zatwierdzającego zmiany |
| %cr | Data zatwierdzającego zmiany, względna |
| %s | Temat |

Pewnie zastanawiasz się jaka jest różnica pomiędzy autorem a zatwierdzającymzmiany\_. Autor to osoba, która oryginalnie stworzyła pracę a zatwierdzający zmiany to osoba, która ostatnia wprowadziła modyfikacje do drzewa. Jeśli zatem wysyłasz do projektu łatkę a następnie któryś z jego członków nanosi ją na projekt, oboje zastajecie zapisani w historii - ty jako autor, a członek zespołu jako osoba zatwierdzająca. Powiemy więcej o tym rozróżnieniu w rozdziale 5.

Wspomniana już wcześniej opcja oneline jest szczególnie przydatna w parze z z inną, a mianowicie, --graph. Tworzy ona mały, śliczny graf ASCII pokazujący historię gałęzi oraz scaleń, co w pełnej krasie można zobaczyć na kopii repozytorium Grita:

$ git log --pretty=format:"%h %s" --graph

\* 2d3acf9 ignore errors from SIGCHLD on trap

\* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit

|\

| \* 420eac9 Added a method for getting the current branch.

\* | 30e367c timeout code and tests

\* | 5a09431 add timeout protection to grit

\* | e1193f8 support for heads with slashes in them

|/

\* d6016bc require time for xmlschema

\* 11d191e Merge branch 'defunkt' into local

Są to jedynie podstawowe opcje formatowania wyjścia polecenia git log - jest ich znacznie więcej. Tabela 2-2 uwzględnia zarówno te które już poznałeś oraz inne, często wykorzystywane, wraz ze opisem każdej z nich.

| **Opcja** | **Opis** |
| --- | --- |
| -p | Pokaż pod każdą zmianą powiązaną łatkę |
| --stat | Pokaż pod każdą zmianą statystyki zmodyfikowanych plików |
| --shortstat | Pokaż wyłącznie zmienione/wstawione/usunięte linie z polecenia --stat |
| --name-only | Pokaż pod każdą zmianą listę zmodyfikowanych plików |
| --name-status | Pokaż listę plików o dodanych/zmodyfikowanych/usuniętych informacjach. |
| --abbrev-commit | Pokaż tylko pierwsze kilka znaków (zamiast 40-tu) sumy kontrolnej SHA-1. |
| --relative-date | Pokaż datę w formacie względnym (np. 2 tygodnie temu) |
| --graph | Pokaż graf ASCII gałęzi oraz historię scaleń obok wyniku. |
| --pretty | Pokaż zatwierdzone zmiany w poprawionym formacie. Dostępne opcje obejmują oneline, short, full, fuller oraz format (gdzie określa własny format) |

### Ograniczanie wyniku historii

Jako dodatek do opcji formatowania, git log przyjmuje także zestaw parametrów ograniczających wynik do określonego podzbioru. Jeden z takich parametrów pokazaliśmy już wcześniej: opcja -2, która spowodowała pokazanie jedynie dwóch ostatnich rewizji. Oczywiście, możesz podać ich dowolną liczbę - -<n>, gdzie n jest liczbą całkowitą. Na co dzień raczej nie będziesz używał jej zbyt często, ponieważ Git domyślnie przekazuje wynik do narzędzia stronicującego, w skutek czego i tak jednocześnie widzisz tylko jedną jego stronę.

Inaczej jest z w przypadku opcji ograniczania w czasie takich jak --since (od) oraz --until (do) które są wyjątkowo przydatne. Na przykład, poniższe polecenie pobiera listę zmian dokonanych w ciągu ostatnich dwóch tygodni:

$ git log --since=2.weeks

Polecenie to obsługuje mnóstwo formatów - możesz uściślić konkretną datę (np. "2008-01-15") lub podać datę względną jak np. 2 lata 1 dzień 3 minuty temu.

Możesz także odfiltrować listę pozostawiając jedynie rewizje spełniające odpowiednie kryteria wyszukiwania. Opcja --author pozwala wybierać po konkretnym autorze, a opcja --grep na wyszukiwanie po słowach kluczowych zawartych w notkach zmian. (Zauważ, że jeżeli potrzebujesz określić zarówno autora jak i słowa kluczowe, musisz dodać opcję --all-match - w przeciwnym razie polecenie dopasuje jedynie wg jednego z kryteriów).

Ostatnią, szczególnie przydatną opcją, akceptowaną przez git log jako filtr, jest ścieżka. Możesz dzięki niej ograniczyć wynik wyłącznie do rewizji, które modyfikują podane pliki. Jest to zawsze ostatnia w kolejności opcja i musi być poprzedzona podwójnym myślnikiem --, tak żeby oddzielić ścieżki od pozostałych opcji.

W tabeli 2-3 znajduje się ta jak i kilka innych często używanych opcji.

| **Opcja** | **Opis** |
| --- | --- |
| -(n) | Pokaż tylko ostatnie n rewizji. |
| --since, --after | Ogranicza rewizje do tych wykonanych po określonej dacie. |
| --until, --before | Ogranicza rewizje do tych wykonanych przed określoną datą. |
| --author | Pokazuje rewizje, których wpis autora pasuje do podanego. |
| --committer | Pokazuje jedynie te rewizje w których osoba zatwierdzająca zmiany pasuje do podanej. |

Na przykład, żeby zobaczyć wyłącznie rewizje modyfikujące pliki testowe w historii plików źródłowych Git-a zatwierdzonych przez Junio Hamano, ale nie zespolonych w październiku 2008, możesz użyć następującego polecenia:

$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \

--before="2008-11-01" --no-merges -- t/

5610e3b - Fix testcase failure when extended attribute

acd3b9e - Enhance hold\_lock\_file\_for\_{update,append}()

f563754 - demonstrate breakage of detached checkout wi

d1a43f2 - reset --hard/read-tree --reset -u: remove un

51a94af - Fix "checkout --track -b newbranch" on detac

b0ad11e - pull: allow "git pull origin $something:$cur

Z prawie 20000 rewizji w historii kodu Gita, podane polecenie wyłowiło jedynie 6 spełniających zadane kryteria.

### Wizualizacja historii w interfejsie graficznym

Do wyświetlania historii rewizji możesz także użyć narzędzi okienkowych - być może spodoba ci się na przykład napisany w Tcl/Tk program o nazwie gitk, który jest dystrybuowany wraz z Gitem. Gitk to proste narzędzie do wizualizacji wyniku polecenia git log i akceptuje ono prawie wszystkie, wcześniej wymienione, opcje filtrowania. Po uruchomieniu gitk z linii poleceń powinieneś zobaczyć okno podobne do widocznego na ekranie 2-2.

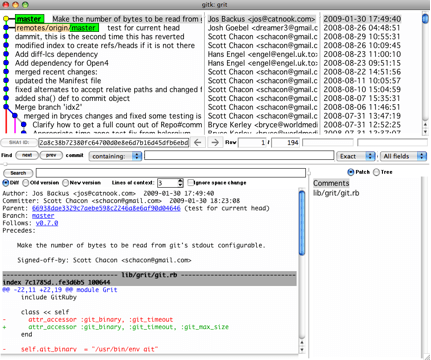


Figure 2-2. Graficzny interfejs programu gitk przedstawiający historię rewizji.

Historia wraz z grafem przodków znajduje się w górnej połówce okna. W dolnej części znajdziesz przeglądarkę różnic pokazującą zmiany wnoszone przez wybraną rewizję.