Git

Spis treści

[Pierwsze kroki 3](#_Toc536733010)

[Wprowadzenie do kontroli wersji 3](#_Toc536733011)

[Lokalne systemy kontroli wersji 4](#_Toc536733012)

[Scentralizowane systemy kontroli wersji 5](#_Toc536733013)

[Rozproszone systemy kontroli wersji 6](#_Toc536733014)

[Krótka historia Git 6](#_Toc536733015)

[Podstawy Git 7](#_Toc536733016)

[Migawki, nie różnice 7](#_Toc536733017)

[Niemal każda operacja jest lokalna 9](#_Toc536733018)

[Git ma wbudowane mechanizmy spójności danych 9](#_Toc536733019)

[Standardowo Git wyłącznie dodaje nowe dane 10](#_Toc536733020)

[Instalacja Git 10](#_Toc536733021)

[Instalacja ze źródeł 10](#_Toc536733022)

[Instalacja w systemie Linux 11](#_Toc536733023)

[Instalacja na komputerze Mac 11](#_Toc536733024)

[Instalacja w systemie Windows 12](#_Toc536733025)

[Trzy stany 12](#_Toc536733026)

[Wstępna konfiguracja Git 14](#_Toc536733027)

[Twoja tożsamość 14](#_Toc536733028)

[Edytor 15](#_Toc536733029)

[Narzędzie obsługi różnic 15](#_Toc536733030)

[Sprawdzanie ustawień 15](#_Toc536733031)

[Uzyskiwanie pomocy 15](#_Toc536733032)

[Podsumowanie 16](#_Toc536733033)

[Podstawy Gita 16](#_Toc536733034)

[Pierwsze repozytorium Gita 16](#_Toc536733035)

[Inicjalizacja Gita w istniejącym katalogu 16](#_Toc536733036)

[Klonowanie istniejącego repozytorium 17](#_Toc536733037)

[Rejestrowanie zmian w repozytorium 18](#_Toc536733038)

[Sprawdzanie stanu twoich plików 18](#_Toc536733039)

[Śledzenie nowych plików 19](#_Toc536733040)

[Dodawanie zmodyfikowanych plików do poczekalni 20](#_Toc536733041)

[Ignorowanie plików 21](#_Toc536733042)

[Podgląd zmian w poczekalni i poza nią 22](#_Toc536733043)

[Zatwierdzanie zmian 24](#_Toc536733044)

[Pomijanie poczekalni 25](#_Toc536733045)

[Usuwanie plików 26](#_Toc536733046)

[Przenoszenie plików 27](#_Toc536733047)

[Podgląd historii rewizji 28](#_Toc536733048)

[Ograniczanie wyniku historii 32](#_Toc536733049)

[Wizualizacja historii w interfejsie graficznym 33](#_Toc536733050)

[Cofanie zmian 34](#_Toc536733051)

[Poprawka do ostatniej rewizji 34](#_Toc536733052)

[Usuwanie pliku z poczekalni 35](#_Toc536733053)

[Cofanie zmian w zmodyfikowanym pliku 36](#_Toc536733054)

[Praca ze zdalnym repozytorium 36](#_Toc536733055)

[Wyświetlanie zdalnych repozytoriów 37](#_Toc536733056)

[Dodawanie zdalnych repozytoriów 37](#_Toc536733057)

[Pobieranie i wciąganie zmian ze zdalnych repozytoriów (polecenia fetch i pull) 38](#_Toc536733058)

[Wypychanie zmian na zewnątrz 39](#_Toc536733059)

[Inspekcja zdalnych zmian 39](#_Toc536733060)

[Usuwanie i zmiana nazwy zdalnych repozytoriów 40](#_Toc536733061)

[Tagowanie (etykietowanie) 40](#_Toc536733062)

[Listowanie etykiet 41](#_Toc536733063)

[Tworzenie etykiet 41](#_Toc536733064)

[Etykiety opisane 41](#_Toc536733065)

[Podpisane etykiety 42](#_Toc536733066)

[Etykiety lekkie 42](#_Toc536733067)

[Weryfikowanie etykiet 43](#_Toc536733068)

[Etykietowanie historii 43](#_Toc536733069)

[Współdzielenie etykiet 44](#_Toc536733070)

[Sztuczki i kruczki 45](#_Toc536733071)

[Auto-uzupełnianie 45](#_Toc536733072)

[Aliasy 46](#_Toc536733073)

[Podsumowanie 47](#_Toc536733074)

[Gałęzie Gita 47](#_Toc536733075)

[Czym jest gałąź 47](#_Toc536733076)

[Podstawy rozgałęziania i scalania 53](#_Toc536733077)

[Podstawy rozgałęziania 54](#_Toc536733078)

[Podstawy scalania 58](#_Toc536733079)

[Podstawowe konflikty scalania 60](#_Toc536733080)

[Zarządzanie gałęziami 62](#_Toc536733081)

[Sposoby pracy z gałęziami 63](#_Toc536733082)

[Gałęzie długodystansowe 63](#_Toc536733083)

[Gałęzie tematyczne 64](#_Toc536733084)

[Gałęzie zdalne 66](#_Toc536733085)

[Wypychanie zmian 71](#_Toc536733086)

[Gałęzie śledzące 72](#_Toc536733087)

[Usuwanie zdalnych gałęzi 73](#_Toc536733088)

[Zmiana bazy 73](#_Toc536733089)

[Typowa zmiana bazy 73](#_Toc536733090)

[Ciekawsze operacje zmiany bazy 76](#_Toc536733091)

[Zagrożenia operacji zmiany bazy 78](#_Toc536733092)

[Podsumowanie 81](#_Toc536733093)

# 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

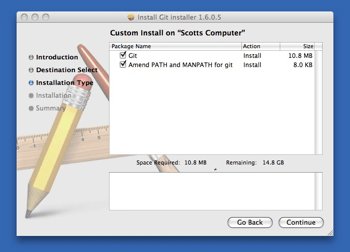
Jeśli korzystasz z dystrybucji opartej na Debianie (np. Ubuntu), użyj apt-get:

$ 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.

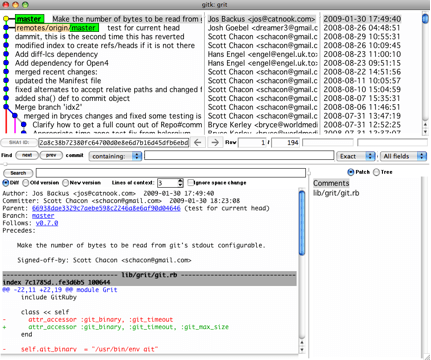


Figure 2-2. Graficzny interfejs programu gitk przedstawiający historię rewizji.

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

## Cofanie zmian

Każdą z wcześniej wprowadzonych zmian możesz cofnąć w dowolnym momencie. Poniżej przyjrzymy się kilku podstawowym funkcjom cofającym modyfikacje. Musisz być jednak ostrożny ponieważ nie zawsze można cofnąć niektóre z tych cofnięć [FIXME]. Jest to jedno z niewielu miejsc w Gitcie, w których należy być naprawdę ostrożnym, gdyż można stracić bezpowrotnie część pracy.

### Poprawka do ostatniej rewizji

Jeden z częstych przypadków to zbyt pochopne wykonanie rewizji i pominięcie w niej części plików, lub też pomyłka w notce do zmian. Jeśli chcesz poprawić wcześniejszą, błędną rewizję, wystarczy uruchomić git commit raz jeszcze, tym razem, z opcją --amend (popraw):

$ git commit --amend

Polecenie bierze zawartość poczekalni i zatwierdza jako dodatkowe zmiany. Jeśli niczego nie zmieniłeś od ostatniej rewizji (np. uruchomiłeś polecenie zaraz po poprzednim zatwierdzeniu zmian) wówczas twoja migawka się nie zmieni ale będziesz miał możliwość modyfikacji notki.

Jak zwykle zostanie uruchomiony edytor z załadowaną treścią poprzedniego komentarza. Edycja przebiega dokładnie tak samo jak zawsze, z tą różnicą, że na końcu zostanie nadpisana oryginalna treść notki.

Czas na przykład. Zatwierdziłeś zmiany a następnie zdałeś sobie sprawę, że zapomniałeś dodać do poczekalni pliku, który chciałeś oryginalnie umieścić w wykonanej rewizji. Wystarczy, że wykonasz następujące polecenie:

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ git commit --amend

Wszystkie trzy polecenia zakończą się jedną rewizją - druga operacja commit zastąpi wynik pierwszej.

### Usuwanie pliku z poczekalni

Następne dwie sekcje pokazują jak zarządzać poczekalnią i zmianami w katalogu roboczym. Dobra wiadomość jest taka, że polecenie używane do określenia stanu obu obszarów przypomina samo jak cofnąć wprowadzone w nich zmiany. Na przykład, powiedzmy, że zmieniłeś dwa pliki i chcesz teraz zatwierdzić je jako dwie osobne rewizje, ale odruchowo wpisałeś git add \* co spowodowało umieszczenie obu plików w poczekalni. Jak w takiej sytuacji usunąć stamtąd jeden z nich? Polecenie git status przypomni ci, że:

$ git add .

$ git status

# On branch master

# Changes to be committed:

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

#

# modified: README.txt

# modified: benchmarks.rb

#

Tekst znajdujący się zaraz pod nagłówkiem zmian do zatwierdzenia mówi "użyj git reset HEAD <plik>... żeby usunąć plik z poczekalni. Nie pozostaje więc nic innego jak zastosować się do porady i zastosować ją na pliku benchmarks.rb:

$ git reset HEAD benchmarks.rb

benchmarks.rb: locally modified

$ git status

# On branch master

# Changes to be committed:

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

#

# modified: README.txt

#

# Changes not staged for commit:

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

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: benchmarks.rb

#

Polecenie wygląda odrobinę dziwacznie, ale działa. Plik benchmarks.rb ciągle zawiera wprowadzone modyfikacje ale nie znajduje się już w poczekalni.

### Cofanie zmian w zmodyfikowanym pliku

Co jeśli okaże się, że nie chcesz jednak zatrzymać zmian wykonanych w pliku benchmarks.rb? W jaki sposób łatwo cofnąć wprowadzone modyfikacje czyli przywrócić plik do stanu w jakim był po ostatniej rewizji (lub początkowym sklonowaniu, lub jakkolwiek dostał się do katalogu roboczego)? Z pomocą przybywa raz jeszcze polecenie git status. W ostatnim przykładzie, pliki będące poza poczekalnią wyglądają następująco:

# Changes not staged for commit:

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

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: benchmarks.rb

#

Git konkretnie wskazuje jak pozbyć się dokonanych zmian (w każdym bądź razie robią to wersje Gita 1.6.1 i nowsze - jeśli posiadasz starszą, bardzo zalecamy aktualizację, która ułatwi ci korzystanie z programu). Zróbmy zatem co każe Git:

$ git checkout -- benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

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

#

# modified: README.txt

#

Możesz teraz przeczytać, że zmiany zostały cofnięte. Powinieneś sobie już także zdawać sprawę, że jest to dość niebezpieczne polecenie: wszelkie zmiany jakie wykonałeś w pliku przepadają - w rzeczy samej został on nadpisany poprzednią wersją. Nigdy nie używaj tego polecenia dopóki nie jesteś absolutnie pewny, że nie chcesz i nie potrzebujesz już danego pliku. Jeśli jedynie chcesz się go chwilowo pozbyć przyjrzymy się specjalnemu poleceniu schowka (stash) oraz gałęziom w kolejnych rozdziałach - są to generalnie znacznie lepsze sposoby.

Pamiętaj, że wszystko co zatwierdzasz do repozytorium Gita może zostać w niemalże dowolnym momencie odtworzone. Nawet rewizje, które znajdowały się w usuniętych gałęziach, albo rewizje nadpisane zatwierdzeniem poprawiającym --amend mogą być odtworzone (odzyskiwanie danych opisujemy w rozdziale 9). Jednakże, cokolwiek utraciłeś a nie było to nigdy wcześniej zatwierdzane do repozytorium, prawdopodobnie odeszło na zawsze.

## Praca ze zdalnym repozytorium

Żeby móc współpracować za pośrednictwem Gita z innymi ludźmi, w jakimkolwiek projekcie, musisz nauczyć się zarządzać zdalnymi repozytoriami. Zdalne repozytorium to wersja twojego projektu utrzymywana na serwerze dostępnym poprzez Internet lub inną sieć. Możesz mieć ich kilka, z których każde może być tylko do odczytu lub zarówno odczytu jak i zapisu. Współpraca w grupie zakłada zarządzanie zdalnymi repozytoriami oraz wypychanie zmian na zewnątrz i pobieranie ich w celu współdzielenia pracy/kodu. Zarządzanie zdalnymi repozytoriami obejmuje umiejętność dodawania zdalnych repozytoriów, usuwania ich jeśli nie są dłużej poprawne, zarządzania zdalnymi gałęziami oraz definiowania je jako śledzone lub nie, i inne. Zajmiemy się tym wszystkim w niniejszym rozdziale.

### Wyświetlanie zdalnych repozytoriów

Aby zobaczyć obecnie skonfigurowane serwery możesz uruchomić polecenie git remote. Pokazuje ono skrócone nazwy wszystkich określonych przez ciebie serwerów. Jeśli sklonowałeś swoje repozytorium, powinieneś przynajmniej zobaczyć origin (źródło) - nazwa domyślna którą Git nadaje serwerowi z którego klonujesz projekt:

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

Initialized empty Git repository in /private/tmp/ticgit/.git/

remote: Counting objects: 595, done.

remote: Compressing objects: 100% (269/269), done.

remote: Total 595 (delta 255), reused 589 (delta 253)

Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.

Resolving deltas: 100% (255/255), done.

$ cd ticgit

$ git remote

origin

Dodanie parametru -v spowoduje dodatkowo wyświetlenie przypisanego do skrótu, pełnego, zapamiętanego przez Gita, adresu URL:

$ git remote -v

origin git://github.com/schacon/ticgit.git

Jeśli posiadasz więcej niż jedno zdalne repozytorium polecenie wyświetli je wszystkie. Na przykład, moje repozytorium z Gritem wygląda następująco:

$ cd grit

$ git remote -v

bakkdoor git://github.com/bakkdoor/grit.git

cho45 git://github.com/cho45/grit.git

defunkt git://github.com/defunkt/grit.git

koke git://github.com/koke/grit.git

origin git@github.com:mojombo/grit.git

Oznacza to, że możesz szybko i łatwo pobrać zmiany z każdego z nich. Zauważ jednak, że tylko oryginalne źródło (origin) jest adresem URL SSH, więc jest jedynym do którego mogę wysyłać własne zmiany (w szczegółach zajmiemy się tym tematem w rozdziale 4).

### Dodawanie zdalnych repozytoriów

W poprzednich sekcjach jedynie wspomniałem o dodawaniu zdalnych repozytoriów, teraz pokażę jak to zrobić to samemu. Aby dodać zdalne repozytorium jako skrót, do którego z łatwością będziesz się mógł odnosić w przyszłości, uruchom polecenie git remote add [skrót] [url]:

$ git remote

origin

$ git remote add pb git://github.com/paulboone/ticgit.git

$ git remote -v

origin git://github.com/schacon/ticgit.git

pb git://github.com/paulboone/ticgit.git

Teraz możesz używać nazwy pb zamiast całego adresu URL. Na przykład, jeżeli chcesz pobrać wszystkie informacje, które posiada Paul, a których ty jeszcze nie masz, możesz uruchomić polecenie fetch wraz z parametrem pb:

$ git fetch pb

remote: Counting objects: 58, done.

remote: Compressing objects: 100% (41/41), done.

remote: Total 44 (delta 24), reused 1 (delta 0)

Unpacking objects: 100% (44/44), done.

From git://github.com/paulboone/ticgit

\* [new branch] master -> pb/master

\* [new branch] ticgit -> pb/ticgit

Główna gałąź (master) Paula jest dostępna lokalnie jako pb/master - możesz scalić ją do którejś z własnych gałęzi lub, jeśli chcesz, jedynie ją przejrzeć przełączając się do lokalnej gałęzi.

### Pobieranie i wciąganie zmian ze zdalnych repozytoriów (polecenia fetch i pull)

Jak przed chwilą zobaczyłeś, aby uzyskać dane ze zdalnego projektu wystarczy uruchomić:

$ git fetch [nazwa-zdalengo-repozytorium]

Polecenie to sięga do zdalnego projektu i pobiera z niego wszystkie dane, których jeszcze nie masz. Po tej operacji, powinieneś mieć już odnośniki do wszystkich zdalnych gałęzi, które możesz teraz scalić z własnymi plikami lub sprawdzić ich zawartość. (Gałęziami oraz ich obsługą zajmiemy się w szczegółach w rozdziale 3).

Po sklonowaniu repozytorium automatycznie zostanie dodany skrót o nazwie origin wskazujący na oryginalną lokalizację. Tak więc, git fetch origin pobierze każdą nową pracę jaka została wypchnięta na oryginalny serwer od momentu sklonowania go przez ciebie (lub ostatniego pobrania zmian). Warto zauważyć, że polecenie fetch pobiera dane do lokalnego repozytorium - nie scala jednak automatycznie zmian z żadnym z twoich plików roboczych jak i w żaden inny sposób tych plików nie modyfikuje. Musisz scalić wszystkie zmiany ręcznie, kiedy będziesz już do tego gotowy.

Jeśli twoja gałąź lokalna jest ustawiona tak, żeby śledzić zdalną gałąź (więcej informacji na ten temat znajdziesz w następnej sekcji, rozdziale 3), wystarczy użyć polecenia git pull, żeby automatycznie pobrać dane (fetch) i je scalić (merge) z lokalnymi plikami. Może być to dla ciebie wygodniejsze; domyślnie, polecenie git clone ustawia twoją lokalną gałąź główną master tak, aby śledziła zmiany w zdalnej gałęzi master na serwerze z którego sklonowałeś repozytorium (zakładając, że zdalne repozytorium posiada gałąź master). Uruchomienie git pull, ogólnie mówiąc, pobiera dane z serwera na bazie którego oryginalnie stworzyłeś swoje repozytorium i próbuje automatycznie scalić zmiany z kodem roboczym nad którym aktualnie, lokalnie pracujesz.

### Wypychanie zmian na zewnątrz

Jeśli doszedłeś z projektem do tego przyjemnego momentu, kiedy możesz i chcesz już podzielić się swoją pracą z innymi, wystarczy, że wypchniesz swoje zmiany na zewnątrz. Służące do tego polecenie jest proste git push [nazwa-zdalnego-repo] [nazwa-gałęzi]. Jeśli chcesz wypchnąć gałąź główną master na oryginalny serwer źródłowy origin (ponownie, klonowanie ustawia obie te nazwy - master i origin - domyślnie i automatycznie), możesz uruchomić następujące polecenie:

$ git push origin master

Polecenie zadziała tylko jeśli sklonowałeś repozytorium z serwera do którego masz prawo zapisu oraz jeśli nikt inny w międzyczasie nie wypchnął własnych zmian. Jeśli zarówno ty jak i inna osoba sklonowały dane w tym samym czasie, po czym ta druga osoba wypchnęła własne zmiany, a następnie ty próbujesz zrobić to samo z własnymi modyfikacjami, twoja próba zostanie od razu odrzucona. Będziesz musiał najpierw zespolić (pobrać i scalić) najnowsze zmiany ze zdalnego repozytorium zanim będziesz mógł wypchnąć własne. Więcej szczegółów na temat wypychania zmian dowiesz się z rozdziału 3.

### Inspekcja zdalnych zmian

Jeśli chcesz zobaczyć więcej informacji o konkretnym zdalnym repozytorium, użyj polecenia git remote show [nazwa-zdalnego-repo]. Uruchamiając je z konkretnym skrótem, jak np. origin, zobaczysz mniej więcej coś takiego:

$ git remote show origin

\* remote origin

URL: git://github.com/schacon/ticgit.git

Remote branch merged with 'git pull' while on branch master

master

Tracked remote branches

master

ticgit

Informacja zawiera adres URL zdalnego repozytorium oraz informacje o śledzonej gałęzi. Polecenie mówi także, że jeśli znajdujesz się w gałęzi master i uruchomisz polecenie git pull, zmiany ze zdalnego repozytorium zaraz po pobraniu automatycznie zostaną scalone z gałęzią master w twoim, lokalnym repozytorium. Polecenie listuje także wszystkie pobrane zdalne odnośniki.

Poniżej znajdziesz prosty przykład na który, pewnie w nieco innej wersji, ale sam się wkrótce natkniesz. Używając intensywnie Gita, możesz zobaczyć znacznie więcej informacji w wyniku działania polecenia git remote show:

$ git remote show origin

\* remote origin

URL: git@github.com:defunkt/github.git

Remote branch merged with 'git pull' while on branch issues

issues

Remote branch merged with 'git pull' while on branch master

master

New remote branches (next fetch will store in remotes/origin)

caching

Stale tracking branches (use 'git remote prune')

libwalker

walker2

Tracked remote branches

acl

apiv2

dashboard2

issues

master

postgres

Local branch pushed with 'git push'

master:master

Przywołane polecenie pokazuje która gałąź zostanie automatycznie wypchnięta po uruchomieniu git push na poszczególnych gałęziach. Zobaczysz także, których zdalnych gałęzi z serwera jeszcze nie posiadasz, które z nich już masz ale z kolei nie ma ich już na serwerze oraz gałęzie, które zostaną automatycznie scalone po uruchomieniu git pull.

### Usuwanie i zmiana nazwy zdalnych repozytoriów

Aby zmienić nazwę odnośnika, czyli skrótu przypisanego do repozytorium, w nowszych wersjach Gita możesz uruchomić git remote rename. Na przykład, aby zmienić nazwę pb na paul, wystarczy, że uruchomisz polecenie git remote rename w poniższy sposób:

$ git remote rename pb paul

$ git remote

origin

paul

Warto wspomnieć, że polecenie zmienia także nazwy zdalnych gałęzi. To co do tej pory było określane jako pb/master od teraz powinno być adresowane jako paul/master.

Jeśli z jakiegoś powodu chcesz usunąć odnośnik - przeniosłeś serwer czy dłużej nie korzystasz z konkretnego mirror-a, albo współpracownik nie udziela się już dłużej w projekcie - możesz skorzystać z git remote rm:

$ git remote rm paul

$ git remote

origin

## Tagowanie (etykietowanie)

Podobnie jak większość systemów kontroli wersji, Git posiada możliwość etykietowania konkretnych, ważnych miejsc w historii. Ogólnie, większość użytkowników korzysta z tej możliwości do zaznaczania ważnych wersji kodu (np. wersja 1.0, itd.). Z tego rozdziału dowiesz się jak wyświetlać dostępne etykiety, jak tworzyć nowe oraz jakie rodzaje tagów rozróżniamy.

### Listowanie etykiet

Wyświetlanie wszystkich dostępnych tagów w Gitcie jest bardzo proste. Wystarczy uruchomić git tag:

$ git tag

v0.1

v1.3

Polecenie wyświetla etykiety w porządku alfabetycznym; porządek w jakim się pojawią nie ma jednak faktycznego znaczenia.

Możesz także wyszukiwać etykiety za pomocą wzorca. Na przykład, repozytorium kodu źródłowego Gita zawiera ponad 240 tagów. Jeśli interesuje cię np. wyłącznie seria 1.4.2, możesz ją wyszukać w następujący sposób:

$ git tag -l 'v1.4.2.\*'

v1.4.2.1

v1.4.2.2

v1.4.2.3

v1.4.2.4

### Tworzenie etykiet

Git używa 2 głównych rodzajów etykiet: lekkich i opisanych. Pierwsze z nich - lekkie - zachowują się mniej więcej tak jak gałąź, która się nie zmienia - jest to tylko wskaźnik do konkretnej rewizji. Z kolei, etykiety opisane są przechowywane jako pełne obiekty w bazie danych Gita. Są one opatrywane sumą kontrolną, zawierają nazwisko osoby etykietującej, jej adres e-mail oraz datę; ponadto, posiadają notkę etykiety, oraz mogą być podpisywane i weryfikowane za pomocą GNU Privacy Guard (GPG). Ogólnie zaleca się aby przy tworzeniu etykiet opisanych uwzględniać wszystkie te informacje; a jeżeli potrzebujesz jedynie etykiety tymczasowej albo z innych powodów nie potrzebujesz tych wszystkich danych, możesz po prostu użyć etykiety lekkiej.

### Etykiety opisane

Tworzenie etykiety opisanej, jak większość rzeczy w Gitcie, jest proste. Wystarczy podać parametr -a podczas uruchamiania polecenia tag:

$ git tag -a v1.4 -m 'my version 1.4'

$ git tag

v0.1

v1.3

v1.4

Parametr -m określa notkę etykiety, która jest wraz z nią przechowywania. Jeśli nie podasz treści notki dla etykiety opisowej, Git uruchomi twój edytor tekstu gdzie będziesz mógł ją dodać.

Dane etykiety wraz z tagowaną rewizją możesz zobaczyć używając polecenia git show:

$ git show v1.4

tag v1.4

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

Date: Mon Feb 9 14:45:11 2009 -0800

my version 1.4

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

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

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

Jak widać została wyświetlona informacja o osobie etykietującej, data stworzenia etykiety, oraz notka poprzedzająca informacje o rewizji:

### Podpisane etykiety

Swoją etykietę możesz podpisać prywatnym kluczem używając GPG. Wystarczy w tym celu użyć parametru -s zamiast -a:

$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for

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

1024-bit DSA key, ID F721C45A, created 2009-02-09

Po uruchomieniu na etykiecie polecenia git show, zobaczysz, że został dołączony do niej podpis GPG:

$ git show v1.5

tag v1.5

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

Date: Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag

-----BEGIN PGP SIGNATURE-----

Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN

Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/

=WryJ

-----END PGP SIGNATURE-----

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

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

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

Nieco później, zobaczysz w jaki sposób można weryfikować podpisane etykiety.

### Etykiety lekkie

Innym sposobem na tagowanie rewizji są etykiety lekkie. Jest to w rzeczy samej suma kontrolna rewizji przechowywana w pliku - nie są przechowywane żadne inne, dodatkowe informacje. Aby stworzyć lekką etykietę, nie przekazuj do polecenia tag żadnego z parametrów -a, -s czy -m:

$ git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5

Uruchamiając teraz na etykiecie git show nie zobaczysz żadnych dodatkowych informacji. Polecenie wyświetli jedynie:

$ git show v1.4-lw

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

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

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

### Weryfikowanie etykiet

Do weryfikacji etykiety używa się polecenia git tag -v [nazwa-etykiety]. Polecenie używa GPG do zweryfikowania podpisu. Żeby mogło zadziałać poprawnie potrzebujesz oczywiście publicznego klucza osoby podpisującej w swoim keyring-u:

$ git tag -v v1.4.2.1

object 883653babd8ee7ea23e6a5c392bb739348b1eb61

type commit

tag v1.4.2.1

tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.

gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A

gpg: Good signature from "Junio C Hamano <junkio@cox.net>"

gpg: aka "[jpeg image of size 1513]"

Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A

Jeśli nie posiadasz klucza publicznego osoby podpisującej, otrzymasz następujący komunikat:

gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A

gpg: Can't check signature: public key not found

error: could not verify the tag 'v1.4.2.1'

### Etykietowanie historii

Możesz także etykietować historyczne rewizje. Załóżmy, że historia zmian wygląda następująco:

$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support

0d52aaab4479697da7686c15f77a3d64d9165190 one more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function

4682c3261057305bdd616e23b64b0857d832627b added a todo file

166ae0c4d3f420721acbb115cc33848dfcc2121a started write support

9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme

Teraz, załóżmy, że zapomniałeś oznaczyć projektu jako wersja 1.2, do której przeszedł on wraz z rewizją "updated rakefile". Możesz dodać etykietę już po fakcie. W tym celu wystarczy na końcu polecenia git tag podać sumę kontrolną lub jej część wskazującą na odpowiednią rewizję:

$ git tag -a v1.2 9fceb02

Aby sprawdzić czy etykieta została stworzona wpisz:

$ git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5

$ git show v1.2

tag v1.2

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

Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date: Sun Apr 27 20:43:35 2008 -0700

updated rakefile

...

### Współdzielenie etykiet

Domyślnie, polecenie git push nie przesyła twoich etykiet do zdalnego repozytorium. Będziesz musiał osobno wypchnąć na współdzielony serwer stworzone etykiety. Proces ten jest podobny do współdzielenia gałęzi i polega na uruchomieniu git push origin [nazwa-etykiety].

$ git push origin v1.5

Counting objects: 50, done.

Compressing objects: 100% (38/38), done.

Writing objects: 100% (44/44), 4.56 KiB, done.

Total 44 (delta 18), reused 8 (delta 1)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.5 -> v1.5

Jeśli masz mnóstwo tagów, którymi chciałbyś się podzielić z innymi, możesz je wszystkie wypchnąć jednocześnie dodając do git push opcję --tags. W ten sposób zostaną przesłane wszystkie tagi, których nie ma jeszcze na serwerze.

$ git push origin --tags

Counting objects: 50, done.

Compressing objects: 100% (38/38), done.

Writing objects: 100% (44/44), 4.56 KiB, done.

Total 44 (delta 18), reused 8 (delta 1)

To git@github.com:schacon/simplegit.git

\* [new tag] v0.1 -> v0.1

\* [new tag] v1.2 -> v1.2

\* [new tag] v1.4 -> v1.4

\* [new tag] v1.4-lw -> v1.4-lw

\* [new tag] v1.5 -> v1.5

Jeśli ktokolwiek inny sklonuje lub pobierze zmiany teraz z twojego repozytorium, dostanie także wszystkie twoje etykiety.

## Sztuczki i kruczki

Zanim zamkniemy ten rozdział, pokażemy kilka sztuczek, które uczynią twoją pracę prostszą, łatwiejszą i przyjemniejszą. Wielu ludzi używa Gita nie korzystając z przytoczonych tutaj porad, ty też nie musisz, ale przynajmniej powinieneś o nich wiedzieć.

### Auto-uzupełnianie

Jeśli używasz powłoki Bash, Git jest wyposażony w poręczny skrypt auto-uzupełniania. Pobierz kod źródłowy Gita i zajrzyj do katalogu contrib/completion. Powinieneś znaleźć tam plik o nazwie git-completion.bash. Skopiuj go do swojego katalogu domowego i dodaj do .bashrc następującą linijkę:

source ~/.git-completion.bash

Jeśli chcesz ustawić Gita tak, żeby automatycznie pozwalał na auto-uzupełnianie wszystkim użytkownikom, skopiuj wymieniony skrypt do katalogu /opt/local/etc/bash\_completion.d na systemach Mac, lub do /etc/bash\_completion.d/ w Linuxie. Jest to katalog skryptów ładowanych automatycznie przez Basha, dzięki czemu opcja zostanie włączona wszystkim użytkownikom.

Jeśli używasz Windows wraz z narzędziem Git Bash, które jest domyślnie instalowane wraz wraz z msysGit, auto-uzupełnianie powinno być pre-konfigurowane i dostępne od razu.

Wciśnij klawisz Tab podczas wpisywania polecenia Gita, a powinieneś ujrzeć zestaw podpowiedzi do wyboru:

$ git co<tab><tab>

commit config

W tym wypadku wpisanie git co i wciśnięcie Tab dwukrotnie podpowiada operacje commit oraz config. Dodanie kolejnej literki m i wciśnięcie Tab uzupełni automatycznie polecenie do git commit.

Podobnie jest z opcjami, co pewnie przyda ci się znacznie częściej. Na przykład jeżeli chcesz uruchomić polecenie git log i nie pamiętasz jednej z opcji, zacznij ją wpisywać i wciśnij Tab aby zobaczyć sugestie:

$ git log --s<tab>

--shortstat --since= --src-prefix= --stat --summary

Jest to bardzo przydatna możliwość pozwalająca na zaoszczędzenie mnóstwa czasu spędzonego na czytaniu dokumentacji.

### Aliasy

Git nie wydedukuje sam polecenia jeśli wpiszesz je częściowo i wciśniesz Enter. Jeśli nie chcesz w całości wpisywać całego tekstu polecenia możesz łatwo stworzyć dla niego alias używając git config. Oto kilka przykładów, które mogą ci się przydać:

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

Oznacza to, że na przykład, zamiast wpisywać git commit, wystarczy, że wpiszesz git ci. Z czasem zaczniesz też stosować także inne polecenia regularnie, nie wahaj się wówczas tworzyć sobie dla nich nowych aliasów.

Technika ta jest także bardzo przydatna do tworzenia poleceń, które uważasz, że powinny istnieć a których brakuje ci w zwięzłej formie. Na przykład, aby skorygować problem z intuicyjnością obsługi usuwania plików z poczekalni, możesz dodać do Gita własny, ułatwiający to alias:

$ git config --global alias.unstage 'reset HEAD --'

W ten sposób dwa poniższe polecenia są sobie równoważne:

$ git unstage fileA

$ git reset HEAD fileA

Od razu polecenie wygląda lepiej. Dość częstą praktyką jest także dodawanie polecenia last:

$ git config --global alias.last 'log -1 HEAD'

Możesz dzięki niemu łatwo zobaczyć ostatnią rewizję:

$ git last

commit 66938dae3329c7aebe598c2246a8e6af90d04646

Author: Josh Goebel <dreamer3@example.com>

Date: Tue Aug 26 19:48:51 2008 +0800

test for current head

Signed-off-by: Scott Chacon <schacon@example.com>

Jak można zauważyć, Git zastępuje nowe polecenie czymkolwiek co do niego przypiszesz. Jednakże, możesz chcieć także uruchomić zewnętrzne polecenie zamiast polecenia Gita. Rozpocznij je wówczas znakiem wykrzyknika !. Przydaje się to podczas tworzenia własnego narzędzia, które współpracuje z repozytorium Gita. Możemy pokazać to na przykładzie aliasu git visual uruchamiającego gitk:

$ git config --global alias.visual "!gitk"

## Podsumowanie

Umiesz już pracować z wszystkimi najważniejszymi, lokalnymi poleceniami Gita - tworzyć i klonować repozytoria, dokonywać zmian, umieszczać je w poczekalni i zatwierdzać do rewizji oraz przeglądać historię repozytorium. W dalszej kolejności zajmiemy się jedną z kluczowych możliwości Gita: modelem gałęzi.

# Gałęzie Gita

Prawie każdy system kontroli wersji posiada wsparcie dla gałęzi. Rozgałęzienie oznacza odbicie od głównego pnia linii rozwoju i kontynuację pracy bez wprowadzania tam bałaganu. W wielu narzędziach kontroli wersji jest to proces dość kosztowny, często wymagający utworzenia nowej kopii katalogu z kodem, co w przypadku dużych projektów może zająć sporo czasu.

Niektórzy uważają model gałęzi Gita za jego „killer feature” i z całą pewnością wyróżnia go spośród innych narzędzi tego typu. Co w nim specjalnego? Sposób, w jaki Git obsługuje gałęzie, jest niesamowicie lekki, przez co tworzenie nowych gałęzi jest niemalże natychmiastowe, a przełączanie się pomiędzy nimi trwa niewiele dłużej. W odróżnieniu od wielu innych systemów, Git zachęca do częstego rozgałęziania i scalania projektu, nawet kilkukrotnie w ciągu jednego dnia. Zrozumienie i opanowanie tego wyjątkowego i potężnego mechanizmu może dosłownie zmienić sposób, w jaki pracujesz.

## Czym jest gałąź

Żeby naprawdę zrozumieć sposób, w jaki Git obsługuje gałęzie, trzeba cofnąć się o krok i przyjrzeć temu, w jaki sposób Git przechowuje dane. Jak może pamiętasz z Rozdziału 1., Git nie przechowuje danych jako serii zmian i różnic, ale jako zestaw migawek.

Kiedy zatwierdzasz zmiany w Gicie, ten zapisuje obiekt zmian (commit), który z kolei zawiera wskaźnik na migawkę zawartości, która w danej chwili znajduje się w poczekalni, metadane autora i opisu oraz zero lub więcej wskaźników na zmiany, które były bezpośrednimi rodzicami zmiany właśnie zatwierdzanej: brak rodziców w przypadku pierwszej, jeden w przypadku zwykłej, oraz kilka w przypadku zmiany powstałej wskutek scalenia dwóch lub więcej gałęzi.

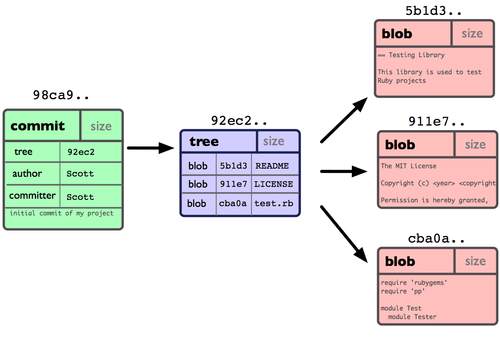
Aby lepiej to zobrazować, załóżmy, że posiadasz katalog zawierający trzy pliki, które umieszczasz w poczekalni, a następnie zatwierdzasz zmiany. Umieszczenie w poczekalni plików powoduje wyliczenie sumy kontrolnej każdego z nich (skrótu SHA-1 wspomnianego w Rozdziale 1.), zapisanie wersji plików w repozytorium (Git nazywa je blobami) i dodanie sumy kontrolnej do poczekalni:

$ git add README test.rb LICENSE

$ git commit -m 'Początkowa wersja mojego projektu'

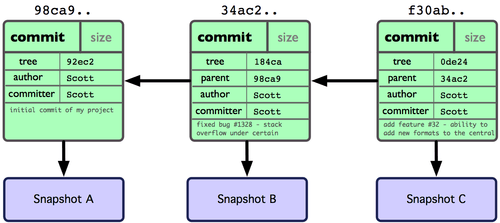
Kiedy zatwierdzasz zmiany przez uruchomienie polecenia git commit, Git liczy sumę kontrolną każdego podkatalogu (w tym wypadku tylko głównego katalogu projektu) i zapisuje te trzy obiekty w repozytorium. Następnie tworzy obiekt zestawu zmian (commit), zawierający metadane oraz wskaźnik na główne drzewo projektu, co w razie potrzeby umożliwi odtworzenie całej migawki.

Teraz repozytorium Gita zawiera już 5 obiektów: jeden blob dla zawartości każdego z trzech plików, jedno drzewo opisujące zawartość katalogu i określające, które pliki przechowywane są w których blobach, oraz jeden zestaw zmian ze wskaźnikiem na owo drzewo i wszystkimi metadanymi. Jeśli chodzi o ideę, dane w repozytorium Gita wyglądają jak na Rysunku 3-1.



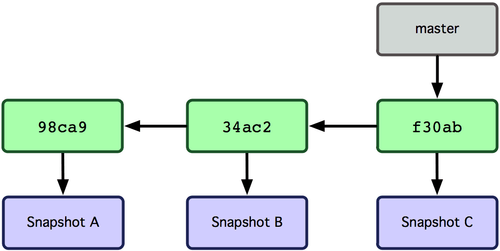
Rysunek 3-1. Dane repozytorium z jedną zatwierdzoną zmianą.

Jeśli dokonasz zmian i je również zatwierdzisz, kolejne zatwierdzenie zachowa wskaźnik do zestawu zmian, który został utworzony bezpośrednio przed właśnie dodawanym. Po dwóch kolejnych zatwierdzeniach, Twoja historia może wyglądać podobnie do przedstawionej na Rysunku 3-2:



Rysunek 3-2. Dane Gita dla wielu zestawów zmian.

Gałąź w Gicie jest po prostu lekkim, przesuwalnym wskaźnikiem na któryś z owych zestawów zmian. Domyślna nazwa gałęzi Gita to master. Kiedy zatwierdzasz pierwsze zmiany, otrzymujesz gałąź master, która wskazuje na ostatni zatwierdzony przez Ciebie zestaw. Z każdym zatwierdzeniem automatycznie przesuwa się ona do przodu.

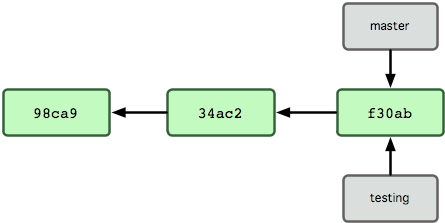


Rysunek 3-3. Gałąź wskazująca na dane zestawu zmian w historii.

Co się stanie, jeśli utworzysz nową gałąź? Cóż, utworzony zostanie nowy wskaźnik, który następnie będziesz mógł przesuwać. Powiedzmy, że tworzysz nową gałąź o nazwie testing. Zrobisz to za pomocą polecenia git branch:

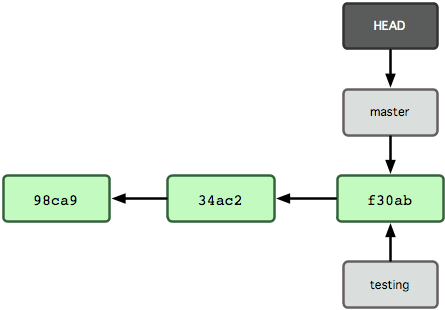
$ git branch testing

Polecenie to tworzy nowy wskaźnik na ten sam zestaw zmian, w którym aktualnie się znajdujesz (zobacz Rysunek 3-4).



Rysunek 3-4. Wiele gałęzi wskazujących na dane zestawów zmian w historii.

Skąd Git wie, na której gałęzi się aktualnie znajdujesz? Utrzymuje on specjalny wskaźnik o nazwie HEAD. Istotnym jest, że bardzo różni się on od koncepcji HEADa znanej z innych systemów kontroli wersji, do jakich mogłeś się już przyzwyczaić, na przykład Subversion czy CVS. W Gicie jest to wskaźnik na lokalną gałąź, na której właśnie się znajdujesz. W tym wypadku, wciąż jesteś na gałęzi master. Polecenie git branch utworzyło jedynie nową gałąź, ale nie przełączyło cię na nią (porównaj z Rysunkiem 3-5).

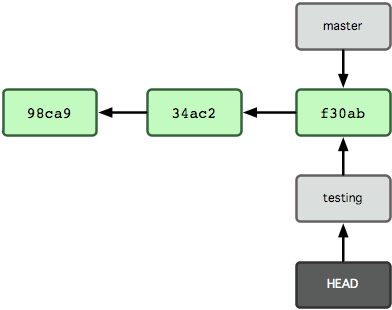


Rysunek 3-5. HEAD wskazuje na gałąź, na której się znajdujesz.

Aby przełączyć się na istniejącą gałąź, używasz polecenia git checkout. Przełączmy się zatem do nowo utworzonej gałęzi testing:

$ git checkout testing

HEAD zostaje zmieniony tak, by wskazywać na gałąź testing (zobacz Rysunek 3-6).



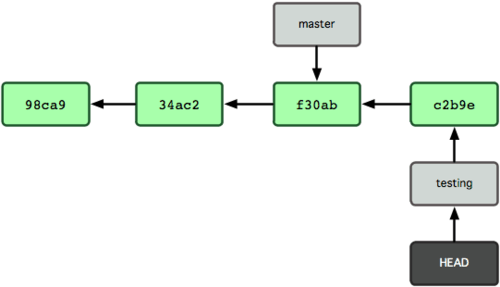
Rysunek 3-6. Po przełączaniu gałęzi, HEAD wskazuje inną gałąź.

Jakie ma to znaczenie? Zatwierdźmy nowe zmiany:

$ vim test.rb

$ git commit -a -m 'zmiana'

Rysunek 3-7 ilustruje wynik operacji.

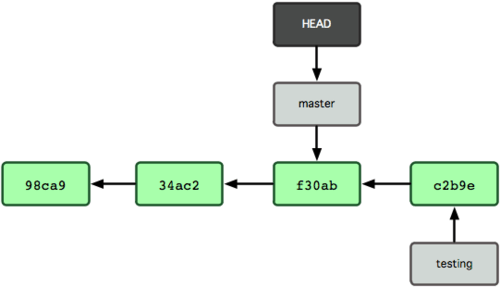


Rysunek 3-7. Gałąź wskazywana przez HEAD przesuwa się naprzód po każdym zatwierdzeniu zmian.

To interesujące, bo teraz Twoja gałąź testing przesunęła się do przodu, jednak gałąź master ciągle wskazuje ten sam zestaw zmian, co w momencie użycia git checkout do zmiany aktywnej gałęzi. Przełączmy się zatem z powrotem na gałąź master:

$ git checkout master

Rysunek 3-8 pokazuje wynik.



Rysunek 3-8. Po wykonaniu checkout, HEAD przesuwa się na inną gałąź.

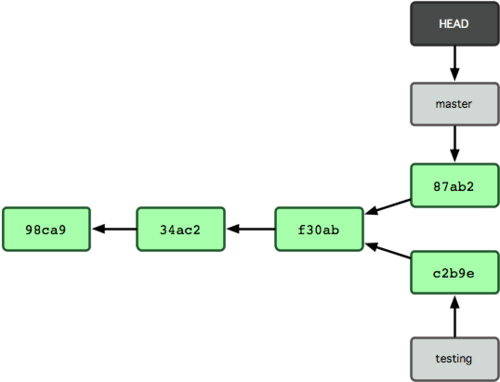
Polecenie dokonało dwóch rzeczy. Przesunęło wskaźnik HEAD z powrotem na gałąź master i przywróciło pliki w katalogu roboczym do stanu z migawki, na którą wskazuje master. Oznacza to również, że zmiany, które od tej pory wprowadzisz, będą rozwidlały się od starszej wersji projektu. W gruncie rzeczy cofa to tymczasowo pracę, jaką wykonałeś na gałęzi testing, byś mógł z dalszymi zmianami pójść w innym kierunku.

Wykonajmy teraz kilka zmian i zatwierdźmy je:

$ vim test.rb

$ git commit -a -m 'inna zmiana'

Teraz historia Twojego projektu została rozszczepiona (porównaj Rysunek 3-9). Stworzyłeś i przełączyłeś się na gałąź, wykonałeś na niej pracę, a następnie powróciłeś na gałąź główną i wykonałeś inną pracę. Oba zestawy zmian są od siebie odizolowane w odrębnych gałęziach: możesz przełączać się pomiędzy nimi oraz scalić je razem, kiedy będziesz na to gotowy. A wszystko to wykonałeś za pomocą dwóch prostych poleceń branch i checkout.



Rysunek 3-9. Rozwidlona historia gałęzi.

Ponieważ gałęzie w Gicie są tak naprawdę prostymi plikami, zawierającymi 40 znaków sumy kontrolnej SHA-1 zestawu zmian, na który wskazują, są one bardzo tanie w tworzeniu i usuwaniu. Stworzenie nowej gałęzi zajmuje dokładnie tyle czasu, co zapisanie 41 bajtów w pliku (40 znaków + znak nowej linii).

Wyraźnie kontrastuje to ze sposobem, w jaki gałęzie obsługuje większość narzędzi do kontroli wersji, gdzie z reguły w grę wchodzi kopiowanie wszystkich plików projektu do osobnego katalogu. Może to trwać kilkanaście sekund czy nawet minut, w zależności od rozmiarów projektu, podczas gdy w Gicie jest zawsze natychmiastowe. Co więcej, ponieważ wraz z każdym zestawem zmian zapamiętujemy jego rodziców, odnalezienie wspólnej bazy przed scaleniem jest automatycznie wykonywane za nas i samo w sobie jest niezwykle proste. Możliwości te pomagają zachęcić deweloperów do częstego tworzenia i wykorzystywania gałęzi.

Zobaczmy, dlaczego ty też powinieneś.

## Podstawy rozgałęziania i scalania

Zajmijmy się prostym przykładem rozgałęziania i scalania używając schematu, jakiego mógłbyś użyć w rzeczywistej pracy. W tym celu wykonasz następujące czynności:

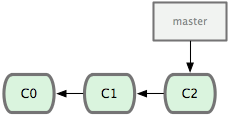
1. Wykonasz pracę nad stroną internetową.
2. Stworzysz gałąź dla nowej funkcji, nad którą pracujesz.
3. Wykonasz jakąś pracę w tej gałęzi.

Na tym etapie otrzymasz telefon, że inny problem jest obecnie priorytetem i potrzeba błyskawicznej poprawki. Oto, co robisz:

1. Powrócisz na gałąź produkcyjną.
2. Stworzysz nową gałąź, by dodać tam poprawkę.
3. Po przetestowaniu, scalisz gałąź z poprawką i wypchniesz zmiany na serwer produkcyjny.
4. Przełączysz się na powrót do gałęzi z nową funkcją i będziesz kontynuować pracę.

### Podstawy rozgałęziania

Na początek załóżmy, że pracujesz nad swoim projektem i masz już zatwierdzonych kilka zestawów zmian (patrz Rysunek 3-10).



Rysunek 3-10. Krótka i prosta historia zmian.

Zdecydowałeś się zająć problemem #53 z systemu śledzenia zgłoszeń, którego używa Twoja firma, czymkolwiek by on nie był. Dla ścisłości, Git nie jest powiązany z żadnym konkretnym systemem tego typu; tym niemniej ponieważ problem #53 to dość konkretny temat, utworzysz nową gałąź by się nim zająć. Aby utworzyć gałąź i jednocześnie się na nią przełączyć, możesz wykonać polecenie git checkout z przełącznikiem -b:

$ git checkout -b iss53

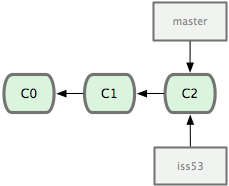
Switched to a new branch "iss53"

Jest to krótsza wersja:

$ git branch iss53

$ git checkout iss53

Rysunek 3-11 pokazuje wynik.

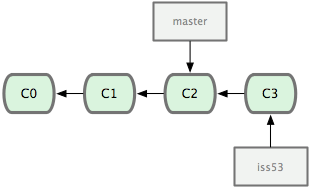


Rysunek 3-11. Tworzenie wskaźnika nowej gałęzi.

Pracujesz nad swoim serwisem WWW i zatwierdzasz kolejne zmiany. Każdorazowo naprzód przesuwa się także gałąź iss53, ponieważ jest aktywna (to znaczy, że wskazuje na nią wskaźnik HEAD; patrz Rysunek 2-12):

$ vim index.html

$ git commit -a -m 'nowa stopka [#53]'



Rysunek 3-12. Gałąź iss53 przesunęła się do przodu wraz z postępami w Twojej pracy.

Teraz właśnie otrzymujesz telefon, że na stronie wykryto błąd i musisz go natychmiast poprawić. Z Gitem nie musisz wprowadzać poprawki razem ze zmianami wykonanymi w ramach pracy nad iss35. Co więcej, nie będzie cię również kosztować wiele wysiłku przywrócenie katalogu roboczego do stanu sprzed tych zmian, tak, by nanieść poprawki na kod, który używany jest na serwerze produkcyjnym. Wszystko, co musisz teraz zrobić, to przełączyć się z powrotem na gałąź master.

Jednakże, nim to zrobisz, zauważ, że, jeśli Twój katalog roboczy lub poczekalnia zawierają niezatwierdzone zmiany, które są w konflikcie z gałęzią, do której chcesz się teraz przełączyć, Git nie pozwoli ci zmienić gałęzi. Przed przełączeniem gałęzi najlepiej jest doprowadzić katalog roboczy do czystego stanu. Istnieją sposoby pozwalające obejść to ograniczenie (mianowicie schowek oraz poprawianie zatwierdzonych już zmian) i zajmiemy się nimi później. Póki co zatwierdziłeś wszystkie swoje zmiany, więc możesz przełączyć się na swoją gałąź master:

$ git checkout master

Switched to branch "master"

W tym momencie Twój katalog roboczy projektu jest dokładnie w takim stanie, w jakim był zanim zacząłeś pracę nad problemem #53, więc możesz skoncentrować się na swojej poprawce. Jest to ważna informacja do zapamiętania: Git resetuje katalog roboczy, by wyglądał dokładnie jak migawka zestawu zmian wskazywanego przez aktywną gałąź. Automatycznie dodaje, usuwa i modyfikuje pliki, by upewnić się, że kopia robocza wygląda tak, jak po ostatnich zatwierdzonych w niej zmianach.

Masz jednak teraz do wykonania ważną poprawkę. Stwórzmy zatem gałąź, na której będziesz pracował do momentu poprawienia błędu (patrz Rysunek 3-13):

$ git checkout -b 'hotfix'

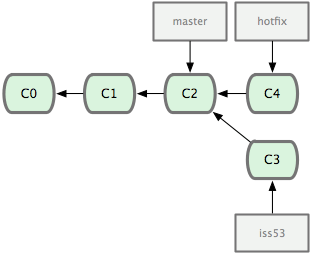
Switched to a new branch "hotfix"

$ vim index.html

$ git commit -a -m 'poprawiony adres e-mail'

[hotfix]: created 3a0874c: "poprawiony adres e-mail"

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



Rysunek 3-13. Gałąź hotfix bazująca na gałęzi master.

Możesz uruchomić swoje testy, upewnić się, że poprawka w gałęzi hotfix jest tym, czego potrzebujesz i scalić ją na powrót z gałęzią master, by następnie przenieść zmiany na serwer produkcyjny. Robi się to poleceniem git merge:

$ git checkout master

$ git merge hotfix

Updating f42c576..3a0874c

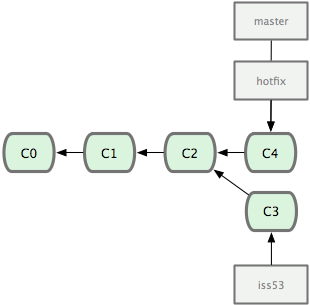
Fast forward

README | 1 -

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

Rezultat polecenia scalenia zawiera frazę „Fast forward”. Ponieważ zestaw zmian wskazywany przez scalaną gałąź był bezpośrednim rodzicem aktualnego zestawu zmian, Git przesuwa wskaźnik do przodu. Innymi słowy, jeśli próbujesz scalić zestaw zmian z innym, do którego dotrzeć można podążając wzdłuż historii tego pierwszego, Git upraszcza wszystko poprzez przesunięcie wskaźnika do przodu, ponieważ nie ma po drodze żadnych rozwidleń do scalenia — stąd nazwa „fast forward” („przewijanie”).

Twoja zmiana jest teraz częścią migawki zestawu zmian wskazywanego przez gałąź master i możesz zaktualizować kod na serwerze produkcyjnym (zobacz Rysunek 3-14).



Rysunek 3-14. Po scaleniu Twoja gałąź master wskazuje to samo miejsce, co gałąź hotfix.

Po tym, jak Twoje niezwykle istotne poprawki trafią na serwer, jesteś gotowy powrócić do uprzednio przerwanej pracy. Najpierw jednak usuniesz gałąź hotfix, gdyż nie jest już ci potrzebna — gałąź master wskazuje to samo miejsce. Możesz ją usunąć używając opcji -d polecenia git branch:

$ git branch -d hotfix

Deleted branch hotfix (3a0874c).

Teraz możesz przełączyć się z powrotem do gałęzi z rozpoczętą wcześniej pracą nad problemem #53 i kontynuować pracę (patrz Rysunek 3-15):

$ git checkout iss53

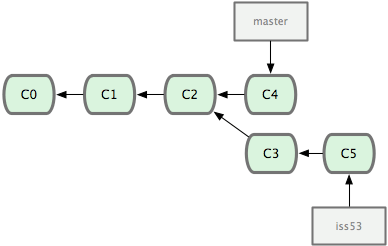
Switched to branch "iss53"

$ vim index.html

$ git commit -a -m 'skończona nowa stopka [#53]'

[iss53]: created ad82d7a: "skończona nowa stopka [#53]"

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



Rysunek 3-15. Twoja gałąź iss53 może przesuwać się do przodu niezależnie.

Warto tu zauważyć, że praca, jaką wykonałeś na gałęzi hotfix nie jest uwzględniona w plikach w gałęzi iss53. Jeśli jej potrzebujesz, możesz scalić zmiany z gałęzi master do gałęzi iss53, uruchamiając git merge master, możesz też zaczekać z integracją zmian na moment, kiedy zdecydujesz się przenieść zmiany z gałęzi iss53 z powrotem do gałęzi master.

### Podstawy scalania

Załóżmy, że zdecydowałeś, że praca nad problemem #53 dobiegła końca i jest gotowa, by scalić ją do gałęzi master. Aby to zrobić, scalisz zmiany z gałęzi iss53 tak samo, jak wcześniej zrobiłeś to z gałęzią hotfix. Wszystko, co musisz zrobić, to przełączyć się na gałąź, do której chcesz zmiany scalić, a następnie uruchomić polecenie git merge:

$ git checkout master

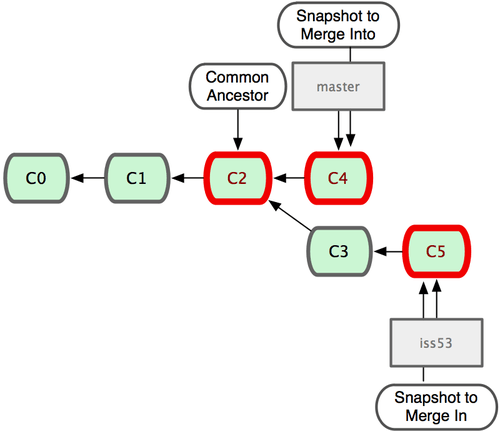
$ git merge iss53

Merge made by recursive.

README | 1 +

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

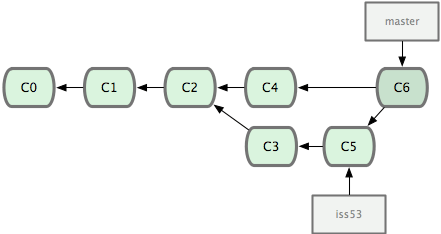
Wygląda to odrobinę inaczej, niż w przypadku wcześniejszego scalenia gałęzi hotfix. W tym wypadku Twoja historia rozwoju została rozszczepiona na wcześniejszym etapie. Ponieważ zestaw zmian z gałęzi, na której obecnie jesteś, nie jest bezpośrednim potomkiem gałęzi, którą scalasz, Git musi w końcu popracować. W tym przypadku Git przeprowadza scalenie trójstronne (ang. three-way merge), używając dwóch migawek wskazywanych przez końcówki gałęzi oraz ich wspólnego przodka. Rysunek 3-16 pokazuje trzy migawki, których w tym przypadku Git używa do scalania.



Rysunek 3-16. Git automatycznie odnajduje najlepszego wspólnego przodka, który będzie punktem wyjściowym do scalenia gałęzi.

Zamiast zwykłego przeniesienia wskaźnika gałęzi do przodu, Git tworzy nową migawkę, która jest wynikiem wspomnianego scalenia trójstronnego i automatycznie tworzy nowy zestaw zmian, wskazujący na ową migawkę (patrz Rysunek 3-17). Określane jest to mianem zmiany scalającej (ang. merge commit), która jest o tyle wyjątkowa, że posiada więcej niż jednego rodzica.

Warto zaznaczyć, że Git sam określa najlepszego wspólnego przodka do wykorzystania jako punkt wyjściowy scalenia; różni się to od zachowania CVS czy Subversion (przed wersją 1.5), gdzie osoba scalająca zmiany musi punkt wyjściowy scalania znaleźć samodzielnie. Czyni to scalanie w Gicie znacznie łatwiejszym, niż w przypadku tamtych systemów.



Rysunek 3-17. Git automatycznie tworzy nowy zestaw zmian zawierający scaloną pracę.

Teraz, kiedy Twoja praca jest już scalona, nie potrzebujesz dłużej gałęzi iss53. Możesz ją usunąć, a następnie ręcznie zamknąć zgłoszenie w swoim systemie śledzenia zadań:

$ git branch -d iss53

### Podstawowe konflikty scalania

Od czasu do czasu proces scalania nie przebiega tak gładko. Jeśli ten sam plik zmieniłeś w różny sposób w obu scalanych gałęziach, Git nie będzie w stanie scalić ich samodzielnie. Jeśli Twoja poprawka problemu #53 zmieniła tę samą część pliku, co zmiana w gałęzi hotfix, podczas scalania otrzymasz komunikat o konflikcie, wyglądający jak poniżej:

$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.

Git nie zatwierdził automatycznie zmiany scalającej. Wstrzymał on cały proces do czasu rozwiązania konfliktu przez Ciebie. Jeśli chcesz zobaczyć, które pliki pozostałe niescalone w dowolnym momencie po wystąpieniu konfliktu, możesz uruchomić git status:

[master\*]$ git status

index.html: needs merge

# On branch master

# Changes not staged for commit:

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

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# unmerged: index.html

#

Cokolwiek spowodowało konflikty i nie zostało automatycznie rozstrzygnięte, jest tutaj wymienione jako „unmerged” (niescalone). Git dodaje do problematycznych plików standardowe znaczniki rozwiązania konfliktu, możesz więc owe pliki otworzyć i samodzielnie rozwiązać konflikty. Twój plik zawiera teraz sekcję, która wygląda mniej więcej tak:

<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

please contact us at support@github.com

</div>

>>>>>>> iss53:index.html

Oznacza to, że wersja wskazywana przez HEAD (Twoja gałąź master, ponieważ tam właśnie byłeś podczas uruchamiania polecenia scalania) znajduje się w górnej części bloku (wszystko powyżej ======), a wersja z gałęzi iss53 to wszystko poniżej. Aby rozwiązać konflikt, musisz wybrać jedną lub druga wersję albo własnoręcznie połączyć zawartość obu. Dla przykładu możesz rozwiązać konflikt, zastępując cały blok poniższą zawartością:

<div id="footer">

please contact us at email.support@github.com

</div>

To rozwiązanie ma po trochu z obu części, całkowicie usunąłem także linie <<<<<<<, ======= i >>>>>>>. Po rozstrzygnięciu wszystkich takich sekcji w każdym z problematycznych plików, uruchom git add na każdym z nich, aby oznaczyć go jako rozwiązany. Przeniesienie do poczekalni oznacza w Gicie rozwiązanie konfliktu. Jeśli chcesz do rozwiązania tych problemów użyć narzędzia graficznego, możesz wydać polecenie git mergetool. Uruchomi ono odpowiednie narzędzie graficzne, które przeprowadzi cię przez wszystkie konflikty:

$ git mergetool

merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff

Merging the files: index.html

Normal merge conflict for 'index.html':

{local}: modified

{remote}: modified

Hit return to start merge resolution tool (opendiff):

Jeśli chcesz użyć narzędzia innego niż domyślne (Git w tym przypadku wybrał dla mnie opendiff, ponieważ pracuję na Maku), możesz zobaczyć wszystkie wspierane narzędzia wymienione na samej górze, zaraz za „merge tool candidates”. Wpisz nazwę narzędzia, którego wolałbyś użyć. W Rozdziale 7 dowiemy się, jak zmienić domyślną wartość dla twojego środowiska pracy.

Po opuszczeniu narzędzia do scalania, Git zapyta, czy wszystko przebiegło pomyślnie. Jeśli odpowiesz skryptowi, że tak właśnie było, plik zostanie umieszczony w poczekalni, by konflikt oznaczyć jako rozwiązany.

Możesz uruchomić polecenie git status ponownie, by upewnić się, że wszystkie konflikty zostały rozwiązane:

$ git status

# On branch master

# Changes to be committed:

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

#

# modified: index.html

#

Jeśli jesteś zadowolony i potwierdziłeś, że wszystkie problematyczne pliki zostały umieszczone w poczekalni, możesz wpisać git commit, by tym samym zatwierdzić zestaw zmian scalających. Jego domyślny opis wygląda jak poniżej:

Merge branch 'iss53'

Conflicts:

index.html

#

# It looks like you may be committing a MERGE.

# If this is not correct, please remove the file

# .git/MERGE\_HEAD

# and try again.

#

Możesz go zmodyfikować, dodając szczegółowy opis sposobu scalenia zmian, jeśli tylko uważasz, że taka informacja będzie pomocna innym, gdy przyjdzie im oglądać efekt scalenia w przyszłości — dlaczego zrobiłeś to w taki, a nie inny sposób, jeśli nie jest to oczywiste.

## Zarządzanie gałęziami

Teraz, kiedy już stworzyłeś, scaliłeś i usunąłeś pierwsze gałęzie, spójrzmy na dodatkowe narzędzia do zarządzania gałęziami, które przydadzą się, gdy będziesz już używać gałęzi w swojej codziennej pracy.

Polecenie git branch robi coś więcej, poza tworzeniem i usuwaniem gałęzi. Jeśli uruchomisz je bez argumentów, otrzymasz prostą listę istniejących gałęzi:

$ git branch

iss53

\* master

testing

Zauważ znak \*, którym poprzedzona została gałąź master: wskazuje on aktywną gałąź. Oznacza to, że jeżeli w tym momencie zatwierdzisz zmiany, wskaźnik gałęzi master zostanie przesunięty do przodu wraz z nowo zatwierdzonymi zmianami. Aby obejrzeć ostatni zatwierdzony zestaw zmian na każdej z gałęzi, możesz użyć polecenia git branch -v:

$ git branch -v

iss53 93b412c fix javascript issue

\* master 7a98805 Merge branch 'iss53'

testing 782fd34 add scott to the author list in the readmes

Kolejna przydatna opcja pozwalająca na sprawdzenie stanu gałęzi to przefiltrowanie tej listy w celu wyświetlenia gałęzi, które już zostały lub jeszcze nie zostały scalone do aktywnej gałęzi. Przydatne opcje --merged i --no-merged służą właśnie do tego celu i są dostępne w Gicie począwszy od wersji 1.5.6. Aby zobaczyć, które gałęzie zostały już scalone z bieżącą, uruchom polecenie git branch --merged:

$ git branch --merged

iss53

\* master

Ponieważ gałąź iss53 została już scalona, znalazła się ona na Twojej liście. Gałęzie znajdujące się na tej liście a niepoprzedzone znakiem \* można właściwie bez większego ryzyka usunąć poleceniem git branch -d; wykonana na nich praca została już scalona do innej gałęzi, więc niczego nie stracisz.

Aby zobaczyć wszystkie gałęzie zawierające zmiany, których jeszcze nie scaliłeś, możesz uruchomić polecenie git branch --no-merged:

$ git branch --no-merged

testing

Pokazuje to Twoją drugą gałąź. Ponieważ zawiera ona zmiany, które nie zostały jeszcze scalone, próba usunięcia jej poleceniem git branch -d nie powiedzie się:

$ git branch -d testing

error: The branch 'testing' is not an ancestor of your current HEAD.

If you are sure you want to delete it, run 'git branch -D testing'.

Jeśli naprawdę chcesz usunąć gałąź i stracić tę część pracy, możesz wymusić to opcją -D zgodnie z tym, co podpowiada komunikat na ekranie.

## Sposoby pracy z gałęziami

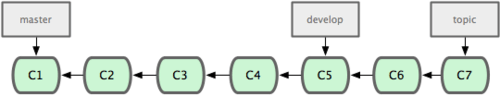
Teraz, kiedy poznałeś już podstawy gałęzi i scalania, co ze zdobytą wiedzą możesz i co powinieneś zrobić? W tej części zajmiemy się typowymi schematami pracy, które stają się dostępne dzięki tak lekkiemu modelowi gałęzi. Pozwoli ci to samemu zdecydować, czy warto stosować je w swoim cyklu rozwoju projektów.

### Gałęzie długodystansowe

Ponieważ Git używa prostego scalania trójstronnego, scalanie zmian z jednej gałęzi do drugiej kilkukrotnie w długim okresie czasu jest ogólnie łatwe. Oznacza to, że możesz utrzymywać kilka gałęzi, które są zawsze otwarte i których używasz dla różnych faz w cyklu rozwoju; możesz scalać zmiany regularnie z jednych gałęzi do innych.

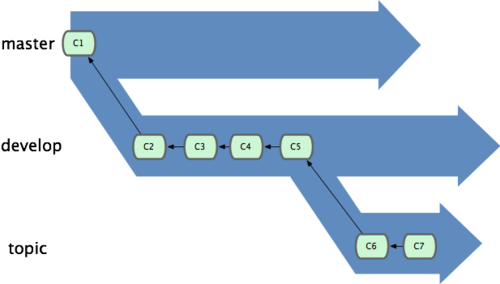
Wielu programistów pracuje z Gitem wykorzystując to podejście, trzymając w gałęzi master jedynie stabilny kod — możliwe, że jedynie kod, który już został albo w najbliższej przyszłości zostanie wydany. Równolegle utrzymują oni inną gałąź o nazwie develop lub next, na której pracują lub używają jej do stabilizacji przyszłych wersji — zawarta w niej praca nie musi być zawsze stabilna, lecz po stabilizacji może być scalona do gałęzi master. Taką gałąź wykorzystuje się także do wciągania zmian z gałęzi tematycznych (gałęzi krótkodystansowych, takich jak wcześniejsza iss53), kiedy są gotowe, aby przetestować je i upewnić się, że nie wprowadzają nowych błędów.

W rzeczywistości mówimy o wskaźnikach przesuwających się w przód po zatwierdzanych przez Ciebie zestawach zmian. Stabilne gałęzie znajdują się wcześniej w historii, a gałęzie robocze na jej końcu (patrz Rysunek 3-18).



Rysunek 3-18. Stabilniejsze gałęzie z reguły znajdują się wcześniej w historii zmian.

Ogólnie łatwiej jest myśleć o nich jak o silosach na zmiany, gdzie grupy zmian są promowane do stabilniejszych silosów, kiedy już zostaną przetestowane (Rysunek 3-19).



Rysunek 3-19. Może być ci łatwiej myśleć o swoich gałęziach jak o silosach.

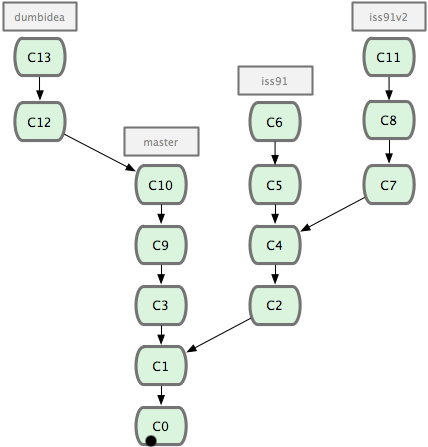
Możesz powielić ten schemat na kilka poziomów stabilności. Niektóre większe projekty posiadają dodatkowo gałąź proposed albo pu („proposed updates” — proponowane zmiany), scalającą gałęzie, które nie są jeszcze gotowe trafić do gałęzi next czy master. Zamysł jest taki, że twoje gałęzie reprezentują różne poziomy stabilności; kiedy osiągają wyższy stopień stabilności, są scalane do gałęzi powyżej. Podobnie jak poprzednio, posiadanie takich długodystansowych gałęzi nie jest konieczne, ale często bardzo pomocne, zwłaszcza jeśli pracujesz przy dużych, złożonych projektach.

### Gałęzie tematyczne

Gałęzie tematyczne, dla odmiany, przydadzą się w każdym projekcie, niezależnie od jego rozmiarów. Gałąź tematyczna to gałąź krótkodystansowa, którą tworzysz i używasz w celu stworzenia pojedynczej funkcji lub innych tego rodzaju zmian. Z całą pewnością nie jest to coś czego chciałbyś używać pracując z wieloma innymi systemami kontroli wersji, ponieważ scalanie i tworzenie nowych gałęzi jest w nich ogólnie mówiąc zbyt kosztowne. W Gicie tworzenie, praca wewnątrz jak i scalanie gałęzi kilkukrotnie w ciągu dnia jest powszechnie stosowane i naturalne.

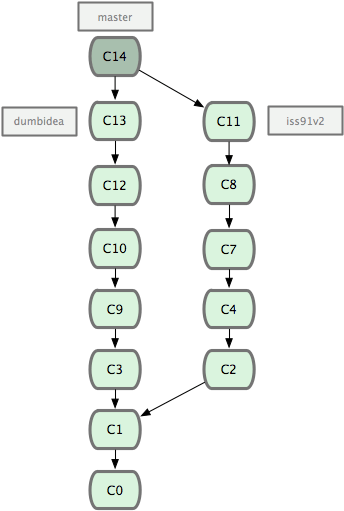
Widziałeś to w poprzedniej sekcji, kiedy pracowaliśmy z gałęziami iss53 i hotfix. Stworzyłeś wewnątrz nich kilka rewizji, po czym usunąłeś je zaraz po scaleniu zmian z gałęzią główną. Ta technika pozwala na szybkie i efektywne przełączanie kontekstu - ponieważ Twój kod jest wyizolowany w osobnych silosach, w których wszystkie zmiany są związane z pracą do jakiej została stworzona gałąź, znacznie łatwiej jest połapać się w kodzie podczas jego przeglądu, recenzowania i temu podobnych. Możesz przechowywać tam swoje zmiany przez kilka minut, dni, miesięcy i scalać je dopiero kiedy są gotowe, bez znaczenia w jakiej kolejności zostały stworzone oraz w jaki sposób przebiegała praca nad nimi.

Rozważ przykład wykonywania pewnego zadania (na gałęzi głównej), stworzenia gałęzi w celu rozwiązania konkretnego problemu (iss91), pracy na niej przez chwilę, stworzenia drugiej gałęzi w celu wypróbowania innego sposobu rozwiązania tego samego problemu (iss91v2), powrotu do gałęzi głównej i pracy z nią przez kolejną chwilę, a następnie stworzenia tam kolejnej gałęzi do sprawdzenia pomysłu, co do którego nie jesteś pewny, czy ma on sens (gałąź dumbidea). Twoja historia rewizji będzie wygląda mniej więcej tak:



Rysunek 3-20. Twoja historia rewizji zawierająca kilka gałęzi tematycznych.

Teraz, powiedzmy, że decydujesz się, że najbardziej podoba ci się drugie rozwiązanie Twojego problemu (iss91v2); zdecydowałeś się także pokazać gałąź dumbidea swoim współpracownikom i okazało się, że pomysł jest genialny. Możesz wyrzucić oryginalne rozwiązanie problemu znajdujące się w gałęzi iss91 (tracąc rewizje C5 i C6) i scalić dwie pozostałe gałęzie. Twoja historia będzie wyglądać tak, jak na Rysunku 3-21.



Rysunek 3-21. Historia zmian po scaleniu gałęzi dumbidea i iss91v2.

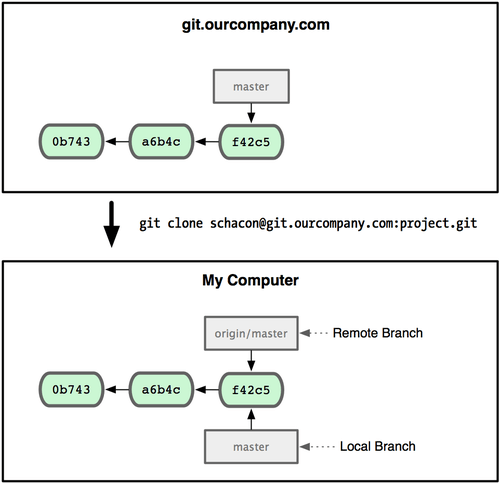
Ważne jest, żeby robiąc to wszystko pamiętać, że są to zupełnie lokalne gałęzie. Tworząc nowe gałęzie i scalając je później, robisz to wyłącznie w ramach własnego repozytorium - bez jakiejkolwiek komunikacji z serwerem.

## Gałęzie zdalne

Zdalne gałęzie są odnośnikami do stanu gałęzi w zdalnym repozytorium. Są to lokalne gałęzie, których nie można zmieniać; są one modyfikowane automatycznie za każdym razem, kiedy wykonujesz jakieś operacje zdalne. Zdalne gałęzie zachowują się jak zakładki przypominające ci, gdzie znajdowały się gałęzie w twoim zdalnym repozytorium ostatnim razem, kiedy się z nim łączyłeś.

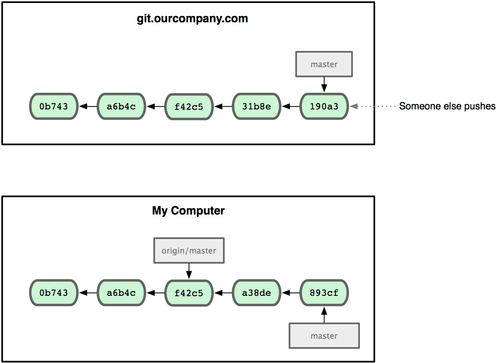
Ich nazwy przybierają następującą formę: (nazwa zdalnego repozytorium)/(nazwa gałęzi). Na przykład, gdybyś chciał zobaczyć, jak wygląda gałąź master w zdalnym repozytorium origin z chwili, kiedy po raz ostatni się z nim komunikowałeś, musiałbyś sprawdzić gałąź origin/master. Jeśli na przykład pracowałeś nad zmianą wraz z partnerem który wypchnął gałąź iss53, możesz mieć lokalną gałąź iss53, ale gałąź na serwerze będzie wskazywała rewizję znajdującą się pod origin/iss53.

Może być to nieco mylące, więc przyjrzyjmy się dokładniej przykładowi. Powiedzmy, że w swojej sieci masz serwer Git pod adresem git.ourcompany.com. Po sklonowaniu z niego repozytorium, Git automatycznie nazwie je jako origin, pobierze wszystkie dane, stworzy wskaźnik do miejsca gdzie znajduje się gałąź master i nazwie ją lokalnie origin/master; nie będziesz mógł jej przesuwać. Git da ci także do pracy Twoją własną gałąź master zaczynającą się w tym samym miejscu, co zdalna (zobacz Rysunek 3-22).



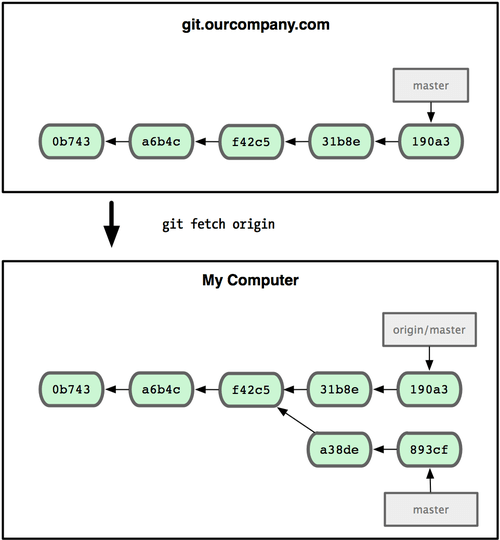
Rysunek 3-22. Po sklonowaniu otrzymasz własną gałąź główną oraz zdalną origin/master wskazującą na gałąź w zdalnym repozytorium.

Jeśli wykonasz jakąś pracę na gałęzi głównej, a w międzyczasie ktoś inny wypchnie zmiany na git.ourcompany.com i zaktualizuje jego gałąź główną, wówczas wasze historie przesuną się do przodu w różny sposób. Co więcej, dopóki nie skontaktujesz się z serwerem zdalnym, Twój wskaźnik origin/master nie przesunie się (Rysunek 3-23).



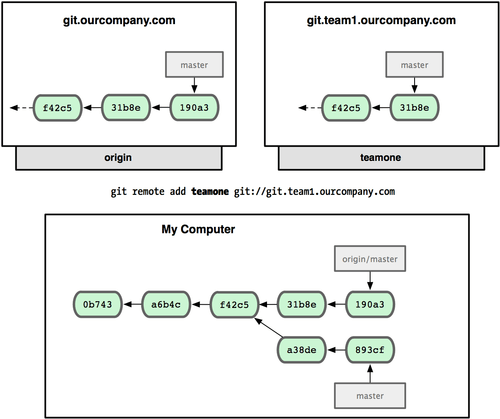
Rysunek 3-23. Kiedy pracujesz lokalnie, wypchnięcie przez kogoś zmian na serwer powoduje, że obie historie zaczynają przesuwać się do przodu w odmienny sposób.

Aby zsynchronizować zmiany uruchom polecenie git fetch origin. Polecenie to zajrzy na serwer, na który wskazuje nazwa origin (w tym wypadku git.ourcompany.com), pobierze z niego wszystkie dane, których jeszcze nie masz u siebie, i zaktualizuje Twoją lokalną bazę danych przesuwając jednocześnie wskaźnik origin/master do nowej, aktualniejszej pozycji (zobacz Rysunek 3-24).



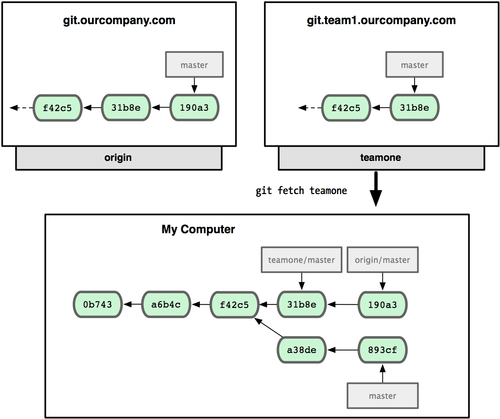
Rysunek 3-24. Polecenie git fetch aktualizuje zdalne referencje.

Aby zaprezentować fakt posiadania kilku zdalnych serwerów oraz stan ich zdalnych gałęzi, załóżmy, że posiadasz jeszcze jeden firmowy serwer Git, który jest używany wyłącznie przez jeden z twoich zespołów sprintowych. Jest to serwer dostępny pod adresem git.team1.ourcompany.com. Możesz go dodać do projektu, nad którym pracujesz, jako nowy zdalny odnośnik uruchamiając polecenie git remote add tak, jak pokazaliśmy to w rozdziale 2. Nazwij go teamone, dzięki czemu później będziesz używał tej nazwy zamiast pełnego adresu URL (rysunek 3-25).



Rysunek 3-25. Dodanie kolejnego zdalnego serwera.

Możesz teraz uruchomić polecenie git fetch teamone aby pobrać wszystko, co znajduje się na serwerze, a czego jeszcze nie posiadasz lokalnie. Ponieważ serwer ten zawiera podzbiór danych które zawiera serwer origin, Git nie pobiera niczego ale tworzy zdalną gałąź teamone/master wskazującą na rewizję dostępną w repozytorium teamone i jej gałęzi master (rysunek 3-26).



Rysunek 3-26. Dostajesz lokalny odnośnik do gałęzi master w repozytorium teamone.

### Wypychanie zmian

Jeśli chcesz podzielić się swoją gałęzią ze światem, musisz wypchnąć zmiany na zdalny serwer, na którym posiadasz prawa zapisu. twoje lokalne gałęzie nie są automatycznie synchronizowane z serwerem, na którym zapisujesz - musisz jawnie określić gałęzie, których zmianami chcesz się podzielić. W ten sposób możesz używać prywatnych gałęzi do pracy, której nie chcesz dzielić, i wypychać jedynie gałęzie tematyczne, w ramach których współpracujesz.

Jeśli posiadasz gałąź o nazwie serverfix, w której chcesz współpracować z innymi, możesz wypchnąć swoje zmiany w taki sam sposób jak wypychałeś je w przypadku pierwszej gałęzi. Uruchom git push (nazwa zdalnego repozytorium) (nazwa gałęzi):

$ git push origin serverfix

Counting objects: 20, done.

Compressing objects: 100% (14/14), done.

Writing objects: 100% (15/15), 1.74 KiB, done.

Total 15 (delta 5), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

\* [new branch] serverfix -> serverfix

Posłużyłem się pewnym skrótem. Git automatycznie sam rozwija nazwę serverfix do pełnej refs/heads/serverfix:refs/heads/serverfix, co oznacza "Weź moją lokalną gałąź serverfix i wypchnij zmiany, aktualizując zdalną gałąź serverfix". Zajmiemy się szczegółowo częścią refs/heads/ w rozdziale 9, ale ogólnie nie powinieneś się tym przejmować. Możesz także wykonać git push origin serverfix:serverfix co przyniesie ten sam efekt - dla Gita znaczy to "Weź moją gałąź serverfix i uaktualnij nią zdalną gałąź serverfix". Możesz używać tego formatu do wypychania lokalnych gałęzi do zdalnych o innej nazwie. Gdybyś nie chciał żeby gałąź na serwerze nazywała się serverfix mógłbyś uruchomić polecenie w formie git push origin serverfix:innanazwagałęzi co spowodowałoby wypchnięcie gałęzi serverfix do innanazwagałęzi w zdalnym repozytorium.

Następnym razem kiedy twoi współpracownicy pobiorą dane z serwera, uzyskają referencję do miejsca, w którym została zapisana Twoja wersja serverfix pod zdalną gałęzią origin/serverfix:

$ git fetch origin

remote: Counting objects: 20, done.

remote: Compressing objects: 100% (14/14), done.

remote: Total 15 (delta 5), reused 0 (delta 0)

Unpacking objects: 100% (15/15), done.

From git@github.com:schacon/simplegit

\* [new branch] serverfix -> origin/serverfix

Warto zauważyć, że kiedy podczas pobierania ściągasz nową, zdalną gałąź, nie uzyskujesz automatycznie lokalnej, edytowalnej jej wersji. Inaczej mówiąc, w tym przypadku, nie masz nowej gałęzi serverfix na której możesz od razu pracować - masz jedynie wskaźnik origin/serverfix którego nie można modyfikować.

Aby scalić pobraną pracę z bieżącą gałęzią roboczą uruchom polecenie git merge origin/serverfix. Jeśli potrzebujesz własnej gałęzi serverfix na której będziesz mógł pracować dalej, możesz ją stworzyć bazując na zdalnej gałęzi w następujący sposób:

$ git checkout -b serverfix origin/serverfix

Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.

Switched to a new branch "serverfix"

Otrzymasz lokalną gałąź, w której będziesz mógł rozpocząć pracę od momentu, w którym znajduje się ona w zdalnej gałązi origin/serverfix.

### Gałęzie śledzące

Przełączenie do lokalnej gałęzi ze zdalnej automatycznie tworzy coś, co określa się jako gałąź śledzącą. Gałęzie śledzące są gałęziami lokalnymi, które posiadają bezpośrednią relację z gałęzią zdalną. Jeśli znajdujesz się w gałęzi śledzącej, po wpisaniu git push Git automatycznie wie, na który serwer wypchnąć zmiany. Podobnie uruchomienie git pull w jednej z takich gałęzi pobiera wszystkie dane i odnośniki ze zdalnego repozytorium i automatycznie scala zmiany z gałęzi zdalnej do odpowiedniej gałęzi zdalnej.

Po sklonowaniu repozytorium automatycznie tworzona jest gałąź master, która śledzi origin/master. Z tego właśnie powodu polecenia git push i git pull działają od razu, bez dodatkowych argumentów. Jednakże, możesz skonfigurować inne gałęzie tak, żeby śledziły zdalne odpowiedniki. Prosty przypadek to przywołany już wcześniej przykład polecenia git checkout -b [gałąź] [nazwa zdalnego repozytorium]/[gałąź]. Jeśli pracujesz z Gitem nowszym niż 1.6.2, możesz także użyć skrótu --track:

$ git checkout --track origin/serverfix

Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.

Switched to a new branch "serverfix"

Żeby skonfigurować lokalną gałąź z inną nazwą niż zdalna, możesz korzystać z pierwszej wersji polecenia podając własną nazwę:

$ git checkout -b sf origin/serverfix

Branch sf set up to track remote branch refs/remotes/origin/serverfix.

Switched to a new branch "sf"

Teraz Twoja lokalna gałąź sf będzie pozawalała na automatyczne wypychanie zmian jak i ich pobieranie z origin/serverfix.

### Usuwanie zdalnych gałęzi

Załóżmy, że skończyłeś pracę ze zdalną gałęzią - powiedzmy, że ty i twoi współpracownicy zakończyliście pracę nad nową funkcją i scaliliście zmiany ze zdalną gałęzią główną master (czy gdziekolwiek indziej, gdzie znajduje się stabilna wersja kodu). Możesz usunąć zdalną gałąź używając raczej niezbyt intuicyjnej składni git push [nazwa zdalnego repozytorium] :[gałąź]. Aby np. usunąć z serwera gałąź serverfix uruchom polecenie:

$ git push origin :serverfix

To git@github.com:schacon/simplegit.git

- [deleted] serverfix

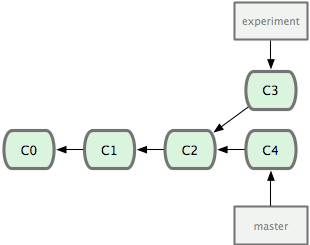
Bum. Nie ma już na serwerze tej gałęzi. Jeśli chcesz, zaznacz sobie tę stronę, ponieważ będziesz potrzebował tego polecenia, a najprawdopodobniej zapomnisz jego składni. Polecenie to można spróbować zapamiętać przypominając sobie składnię git push [nazwa zdalnego repozytorium] [gałąź lokalna]:[gałąź zdalna], którą omówiliśmy odrobinę wcześniej. Pozbywając się części [gałąź lokalna], mówisz mniej więcej "Weź nic z mojej strony i zrób z tego [gałąź zdalną]".

## Zmiana bazy

W Git istnieją dwa podstawowe sposoby integrowania zmian z jednej gałęzi do drugiej: scalanie (polecenie merge) oraz zmiana bazy (polecenie rebase). W tym rozdziale dowiesz się, czym jest zmiana bazy, jak ją przeprowadzić, dlaczego jest to świetne narzędzie i w jakich przypadkach lepiej się powstrzymać od jego wykorzystania.

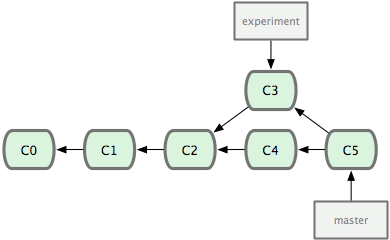
### Typowa zmiana bazy

Jeśli cofniesz się do poprzedniego przykładu z sekcji Scalanie (patrz Rysunek 3-27), zobaczysz, że rozszczepiłeś swoją pracę i wykonywałeś zmiany w dwóch różnych gałęziach.



Rysunek 3-27. Początkowa historia po rozszczepieniu.

Najprostszym sposobem, aby zintegrować gałęzie - jak już napisaliśmy - jest polecenie merge. Przeprowadza ono trójstronne scalanie pomiędzy dwoma ostatnimi migawkami gałęzi (C3 i C4) oraz ich ostatnim wspólnym przodkiem (C2), tworząc nową migawkę (oraz rewizję), tak jak widać to na rysunku 3-28.



Rysunek 3-28. Scalanie gałęzi integrujące rozszczepioną historię zmian.

Jednakże istnieje inny sposób: możesz stworzyć łatkę ze zmianami wprowadzonymi w C3 i zaaplikować ją na rewizję C4. W Gicie nazywa się to zmianą bazy (ang. rebase). Dzięki poleceniu rebase możesz wziąć wszystkie zmiany, które zostały zatwierdzone w jednej gałęzi i zaaplikować je w innej.

W tym wypadku, mógłbyś uruchomić następujące polecenie:

$ git checkout experiment

$ git rebase master

First, rewinding head to replay your work on top of it...

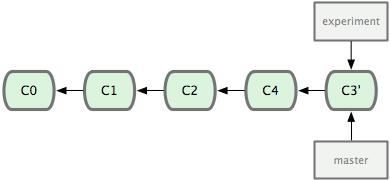
Applying: added staged command

Polecenie to działa przesuwając się do ostatniego wspólnego przodka obu gałęzi (tej w której się znajdujesz oraz tej do której robisz zmianę bazy), pobierając różnice opisujące kolejne zmiany (ang. diffs) wprowadzane przez kolejne rewizje w gałęzi w której się znajdujesz, zapisując je w tymczasowych plikach, następnie resetuje bieżącą gałąź do tej samej rewizji do której wykonujesz operację zmiany bazy, po czym aplikuje po kolei zapisane zmiany. Ilustruje to rysunek 3-29.



Rysunek 3-29. Zmiana bazy dla zmian wprowadzonych w C3 do C4.

W tym momencie możesz wrócić do gałęzi master i scalić zmiany wykonując proste przesunięcie wskaźnika (co przesunie wskaźnik master na koniec) (rysunek 3-30).



Rysunek 3-30. Przesunięcie gałęzi master po operacji zmiany bazy.

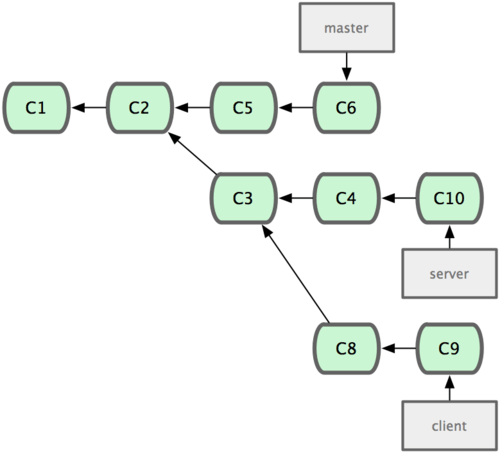
Teraz migawka wskazywana przez C3' jest dokładnie taka sama jak ta, na którą wskazuje C5 w przykładzie ze scalaniem. Nie ma różnicy w produkcie końcowym integracji. Zmiana bazy tworzy jednak czystszą historię. Jeśli przejrzysz historię gałęzi po operacji rebase, wygląda ona na liniową: wygląda jakby cała praca była wykonywana stopniowo, nawet jeśli oryginalnie odbywała się równolegle.

Warto korzystać z tej funkcji, by mieć pewność, że rewizje zaaplikują się w bezproblemowy sposób do zdalnej gałęzi - być może w projekcie w którym próbujesz się udzielać, a którym nie zarządzasz. W takim wypadku będziesz wykonywał swoją pracę we własnej gałęzi, a następnie zmieniał jej bazę na origin/master, jak tylko będziesz gotowy do przesłania własnych poprawek do głównego projektu. W ten sposób osoba utrzymująca projekt nie będzie musiała dodatkowo wykonywać integracji - jedynie prostolinijne scalenie lub czyste zastosowanie zmian.

Zauważ, że migawka wskazywana przez wynikową rewizję bez względu na to, czy jest to ostatnia rewizja po zmianie bazy lub ostatnia rewizja scalająca po operacji scalania, to taka sama migawka - różnica istnieje jedynie w historii. Zmiana bazy nanosi zmiany z jednej linii pracy do innej w kolejności, w jakiej były one wprowadzane, w odróżnieniu od scalania, które bierze dwie końcówki i integruje je ze sobą.

### Ciekawsze operacje zmiany bazy

Poleceniem rebase możesz także zastosować zmiany na innej gałęzi niż ta, której zmieniasz bazę. Dla przykładu - weź historię taką jak na rysunku 3-31. Utworzyłeś gałąź tematyczną (server), żeby dodać nowe funkcje do kodu serwerowego, po czym utworzyłeś rewizję. Następnie utworzyłeś gałąź, żeby wykonać zmiany w kliencie (client) i kilkukrotnie zatwierdziłeś zmiany. W końcu wróciłeś do gałęzi server i wykonałeś kilka kolejnych rewizji.

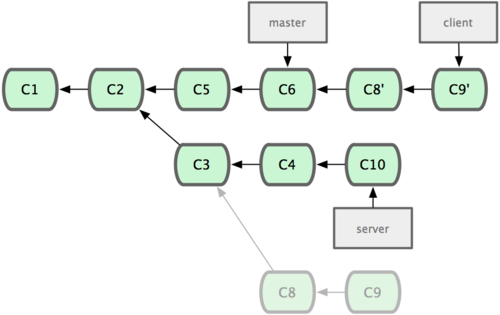


Rysunek 3-31. Historia z gałęzią tematyczną utworzoną na podstawie innej gałęzi tematycznej.

Załóżmy, że zdecydowałeś się scalić zmiany w kliencie do kodu głównego, ale chcesz się jeszcze wstrzymać ze zmianami po stronie serwera, dopóki nie zostaną one dokładniej przetestowane. Możesz wziąć zmiany w kodzie klienta, których nie ma w kodzie serwera (C8 i C9) i zastosować je na gałęzi głównej używając opcji --onto polecenia git rebase:

$ git rebase --onto master server client

Oznacza to mniej więcej "Przełącz się do gałęzi klienta, określ zmiany wprowadzone od wspólnego przodka gałęzi client i server, a następnie nanieś te zmiany na gałąź główną master. Jest to nieco skomplikowane, ale wynik (pokazany na rysunku 3-32) całkiem niezły.

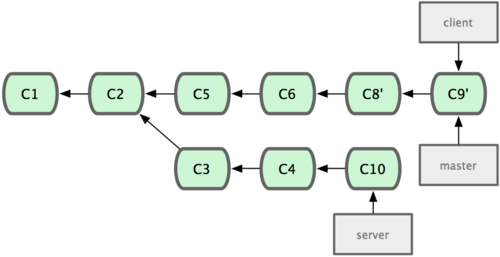


Rysunek 3-32. Zmiana bazy gałęzi tematycznej odbitej z innej gałęzi tematycznej.

Teraz możesz zwyczajnie przesunąć wskaźnik gałęzi głównej do przodu (rysunek 3-33):

$ git checkout master

$ git merge client

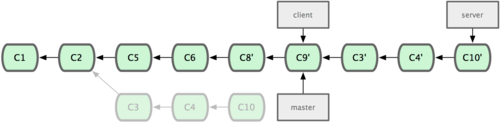


Rysunek 3-33. Przesunięcie do przodu gałęzi master w celu uwzględnienia zmian z gałęzi klienta.

Powiedzmy, że zdecydujesz się pobrać i scalić zmiany z gałęzi server. Możesz zmienić bazę gałęzi server na wskazywaną przez master bez konieczności przełączania się do gałęzi server używając git rebase [gałąź bazowa] [gałąź tematyczna] - w ten sposób zmiany z gałęzi server zostaną zaaplikowane do gałęzi bazowej master:

$ git rebase master server

Polecenie odtwarza zmiany z gałęzi server na gałęzi master tak, jak pokazuje to rysunek 3-34.



Rysunek 3-34. Zmiana bazy gałęzi serwer na koniec gałęzi głównej.

Następnie możesz przesunąć gałąź bazową (master):

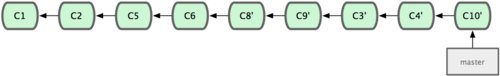
$ git checkout master

$ git merge server

Możesz teraz usunąć gałęzie client i server, ponieważ cała praca jest już zintegrowana i więcej ich nie potrzebujesz pozostawiając historię w stanie takim, jaki obrazuje rysunek 3-35:

$ git branch -d client

$ git branch -d server



Rysunek 3-35. Ostateczna historia rewizji.

### Zagrożenia operacji zmiany bazy

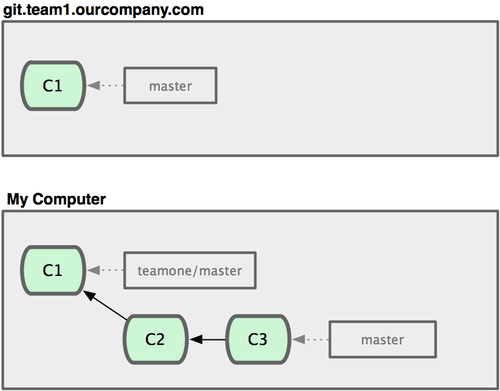
Błogosławieństwo, jakie daje możliwość zmiany bazy, ma swoją mroczną stronę. Można ją podsumować jednym zdaniem:

**Nie zmieniaj bazy rewizji, które wypchnąłeś już do publicznego repozytorium.**

Jeśli będziesz się stosował do tej reguły, wszystko będzie dobrze. W przeciwnym razie ludzie cię znienawidzą, a rodzina i przyjaciele zaczną omijać szerokim łukiem.

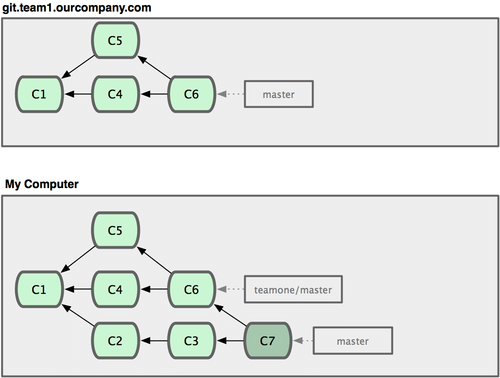
Stosując operację zmiany bazy porzucasz istniejące rewizje i tworzysz nowe, które są podobne, ale inne. Wypychasz gdzieś swoje zmiany, inni je pobierają, scalają i pracują na nich, a następnie nadpisujesz te zmiany poleceniem git rebase i wypychasz ponownie na serwer. Twoi współpracownicy będą musieli scalić swoją pracę raz jeszcze i zrobi się bałagan, kiedy spróbujesz pobrać i scalić ich zmiany z powrotem z twoimi.

Spójrzmy na przykład obrazujący, jak operacja zmiany bazy może spowodować problemy. Załóżmy, że sklonujesz repozytorium z centralnego serwera, a następnie wykonasz bazując na tym nowe zmiany. Twoja historia rewizji wygląda tak jak na rysunku 3-36.



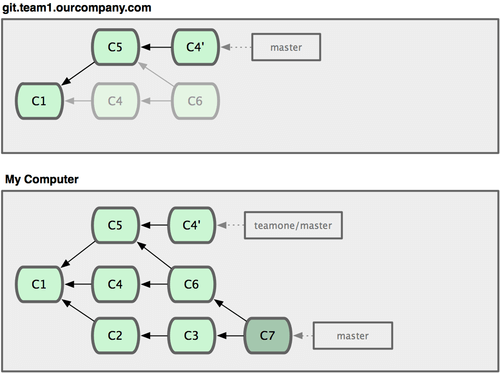
Rysunek 3-36. Sklonowane repozytorium i dokonane zmiany.

Teraz ktoś inny wykonuje inną pracę, która obejmuje scalenie, i wypycha ją na centralny serwer. Pobierasz zmiany, scalasz nową, zdalną gałąź z własną pracą, w wyniku czego historia wygląda mniej więcej tak, jak na rysunku 3-37.



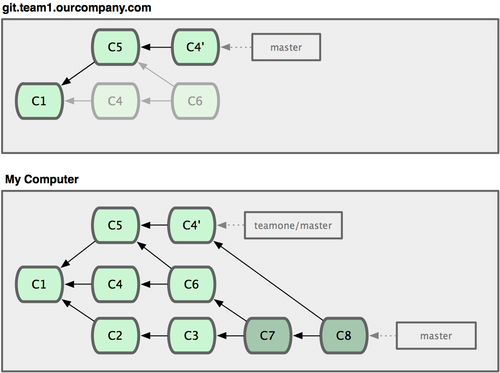
Rysunek 3-37. Pobranie kolejnych rewizji i scalenie ich z własnymi zmianami.

Następnie osoba, która wypchnęła scalone zmiany, rozmyśliła się i zdecydowała zamiast scalenia zmienić bazę swoich zmian; wykonuje git push --force, żeby zastąpić historię na serwerze. Następnie ty pobierasz dane z serwera ściągając nowe rewizje.



Rysunek 3-38. Ktoś wypycha rewizje po operacji zmiany bazy porzucając rewizje, na których ty oparłeś swoje zmiany.

W tym momencie musisz raz jeszcze scalać tę pracę mimo tego, że już to wcześniej raz zrobiłeś. Operacja zmiany bazy zmienia sumy kontrolne SHA-1 tych rewizji, więc dla Gita wyglądają one jak zupełnie nowe, choć w rzeczywistości masz już zmiany wprowadzone w C4 w swojej historii (rysunek 3-39).



Rysunek 3-39. Scalasz tą samą pracę raz jeszcze tworząc nową rewizję scalającą.

Musisz scalić swoją pracę w pewnym momencie po to, żeby dotrzymywać kroku innym programistom. Kiedy już to zrobisz, Twoja historia zmian będzie zawierać zarówno rewizje C4 jak i C4', które mają różne sumy SHA-1, ale zawierają te same zmiany i mają ten sam komentarz. Jeśli uruchomisz git log dla takiej historii, zobaczysz dwie rewizje mające tego samego autora, datę oraz komentarz, co będzie mylące. Co więcej, jeśli wypchniesz tę historię z powrotem na serwer, raz jeszcze wprowadzisz wszystkie rewizje powstałe w wyniku operacji zmiany bazy na serwer centralny, co może dalej mylić i denerwować ludzi.

Jeśli traktujesz zmianę bazy jako sposób na porządkowanie historii i sposób pracy z rewizjami przed wypchnięciem ich na serwer oraz jeśli zmieniasz bazę tylko tym rewizjom, które nigdy wcześniej nie były dostępne publicznie, wówczas wszystko będzie w porządku. Jeśli zaczniesz zmieniać bazę rewizjom, które były już publicznie dostępne, a ludzie mogą na nich bazować swoje zmiany, wówczas możesz wpaść w naprawdę frustrujące tarapaty.

## Podsumowanie

Omówiliśmy podstawy tworzenia gałęzi oraz scalania w Git. Powinieneś już z łatwością tworzyć gałęzie, przełączać się pomiędzy nimi i scalać zawarte w nich zmiany. Powinieneś także umieć współdzielić swoje gałęzie wypychając je na serwer, pracować z innymi w współdzielonych gałęziach oraz zmieniać bazę gałęziom, zanim zostaną udostępnione innym.