

# 2.1 Podstawy Gita - Pierwsze repozytorium 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 LICENSE

$ 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 https://github.com/libgit2/libgit2

Tworzony jest katalog o nazwie „libgit2”, 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 libgit2, zobaczysz wewnątrz pliki projektu, gotowe do użycia i pracy z nimi. Jeśli chcesz sklonować repozytorium do katalogu o nazwie innej niż libgit2, możesz to zrobić podając w wierszu poleceń kolejną opcję:

$ git clone https://github.com/libgit2/libgit2 mylibgit

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

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.

# 2.2 Podstawy Gita - Rejestrowanie zmian w repozytorium

## **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.



Figure 8. Cykl stanów 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:

$ echo 'My Project' > 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 CONTRIBUTING.md, 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)

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

modified: CONTRIBUTING.md

Plik CONTRIBUTING.md 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 CONTRIBUTING.md w poczekalni, a następnie ponownie wykonaj git status:

$ git add CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

new file: README

modified: CONTRIBUTING.md

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 CONTRIBUTING.md jeszcze przed zatwierdzeniem. Otwierasz go zatem, wprowadzasz zmianę i jesteś gotowy do zatwierdzenia. Uruchom jednak git status raz jeszcze:

$ vim CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

new file: README

modified: CONTRIBUTING.md

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: CONTRIBUTING.md

Co do licha? Plik CONTRIBUTING.md 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 CONTRIBUTING.md 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 CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

new file: README

modified: CONTRIBUTING.md

### **Zwięzły stan**

Rezultat polecenia git status jest dość szczegółowy, ale też zbyt rozległy. Git posiada też opcję zwięzłego stanu, więc możesz zobaczyć swoje zmiany w bardziej zwartej postaci. Jeśli wykonasz git status -s lub git status --short uzyskasz znacznie uproszczony wynik tego polecania.

$ git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

Nowe nieśledzone pliki mają obok siebie ??, nowe pliki dodane do poczekalni A, zmodyfikowane pliki mają natomiast M. Mamy tutaj tylko dwie kolumny - lewa wskazuje na to czy plik jest w poczekalni, a prawa czy jest zmieniony. Przykład dla powyższego rezultatu, plik README został zmodyfikowany w katalogu roboczym ale nie został dodany do poczekalni, podczas gdy lib/simplegit.rb został zmodyfikowany i dodany do poczekalni. Plik Rakefile został zmodyfikowany, dodany do poczekalni i zmodyfikowany ponownie, więc jego zmiany są jednocześnie w poczekalni i poza nią.

### **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).

Możesz użyć dwóch gwiazdek aby dopasować katalogi zagnieżdżone; a/\*\*/z would match a/z, a/b/z, a/b/c/z i tak dalej.

Poniżej znajdziesz kolejny przykład pliku .gitignore:

# no .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the root TODO file, not subdir/TODO

/TODO

# ignore all files in the build/ directory

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .txt files in the doc/ directory

doc/\*\*/\*.txt

|  |  |
| --- | --- |
| **Wskazówka** | Github zarządza dość obszerną listą przykładowych plików .gitignore dla wielu projektów i języków <https://github.com/github/gitignore> jeśli chcesz mieć punkt wyjścia dla swojego projektu. |

### **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 CONTRIBUTING.md, 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)

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

modified: CONTRIBUTING.md

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

$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if you patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

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 --staged. To polecenie porówna zmiany z poczekalni z ostatnią zmianą:

$ git diff --staged

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README

@@ -0,0 +1 @@

+My Project

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 CONTRIBUTING.md, a następnie zmodyfikujesz go ponownie, możesz użyć git status, by obejrzeć zmiany znajdujące się w poczekalni, jak i te poza nią:

$ git add CONTRIBUTING.md

$ echo 'test line' >> CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

modified: CONTRIBUTING.md

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: CONTRIBUTING.md

Teraz możesz użyć git diff, by zobaczyć zmiany spoza poczekalni

$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 643e24f..87f08c8 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -119,3 +119,4 @@ at the

## Starter Projects

See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).

+# test line

oraz git diff --cached, aby zobaczyć zmiany wysłane do poczekalni(--staged i --cached działają identycznie):

$ git diff --cached

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if you patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

|  |  |
| --- | --- |
| **Uwaga** | **Git Diff jako narzędzie zewnętrzne**  Będziemy kontynuować używanie polecenia git diff na wiele sposobów w dalszej części książki. Jest jeszcze inny sposób aby przyjrzeć się tym zmianom jeżeli preferujesz graficzny lub zewnętrzny program do porównywania. Jeśli wykonasz git difftool zamiast git diff możesz zobaczyć te różnice w programach takich jak Araxis, emerge, vimdiff i wielu innych. Uruchom git difftool --tool-help aby zobaczyć, które z nich są dostępne w twoim systemie. |

### **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:

# new file: README

# modified: CONTRIBUTING.md

#

~

~

~

".git/COMMIT\_EDITMSG" 9L, 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 463dc4f] Story 182: Fix benchmarks for speed

2 files changed, 2 insertions(+)

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:

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

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

modified: CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")

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

[master 83e38c7] added new benchmarks

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

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

### **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 PROJECTS.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

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

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

deleted: PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")

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

$ git rm PROJECTS.md

rm 'PROJECTS.md'

$ git status

On branch master

Changes to be committed:

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

deleted: PROJECTS.md

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

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 ~.

### **Zmiana nazw 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.md README

$ git status

On branch master

Changes to be committed:

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

renamed: README.md -> README

Jest to równoważne z uruchomieniem poleceń:

$ mv README.md README

$ git rm README.md

$ 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.

# 2.3 Podstawy Gita - Podgląd historii rewizji

## **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 https://github.com/schacon/simplegit-progit

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

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.platform = Gem::Platform::RUBY

s.name = "simplegit"

- s.version = "0.1.0"

+ s.version = "0.1.1"

s.author = "Scott Chacon"

s.email = "schacon@gee-mail.com"

s.summary = "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

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

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

removed unnecessary test

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 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

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

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

removed unnecessary test

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

1 file changed, 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(+)

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

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, 6 years ago : changed the version number

085bb3b - Scott Chacon, 6 years ago : removed unnecessary test

a11bef0 - Scott Chacon, 6 years ago : first commit

[Useful options for git log --pretty=format](https://git-scm.com/book/pl/v2/ch00/pretty_format) pokazuje najprzydatniejsze opcje akceptowane przez format.

| Table 1. Useful options for git log --pretty=format | |
| --- | --- |
| **Opcja** | **Opis** |
| %H | hash commita |
| %h | skrócony hash commita |
| %T | hash drzewa |
| %t | skrócony hash drzewa |
| %P | hash commita nadrzędnego |
| %p | skrócony hash commita nadrzędnego |
| %an | Nazwa autora |
| %ae | E-mail autora |
| %ad | Data autora (odnosi się do --date=option) |
| %ar | Data autora, względna |
| %cn | Nazwa zatwierdzającego zmiany |
| %ce | E-mail zatwierdzającego zmiany |
| %cd | Data zatwierdzającego zmiany |
| %cr | Data zatwierdzającego zmiany, względna |
| %s | Wiadomość |

Pewnie zastanawiasz się jaka jest różnica pomiędzy **autorem** a **zatwierdzającym\_zmiany**. 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 [Distributed Git](https://git-scm.com/book/pl/v2/ch00/_distributed_git).

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:

$ 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

Ten rodzaj grafu będzie bardziej interesujący kiedy przebrniemy przez tworzenie gałęzi i ich scalanie w następnym rozdziale.

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.

| Table 2. Najczęściej używane opcje git log | |
| --- | --- |
| **Option** | **Description** |
| -p | Show the patch introduced with each commit. |
| --stat | Show statistics for files modified in each commit. |
| --shortstat | Display only the changed/insertions/deletions line from the --stat command. |
| --name-only | Show the list of files modified after the commit information. |
| --name-status | Show the list of files affected with added/modified/deleted information as well. |
| --abbrev-commit | Show only the first few characters of the SHA-1 checksum instead of all 40. |
| --relative-date | Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format. |
| --graph | Display an ASCII graph of the branch and merge history beside the log output. |
| --pretty | Show commits in an alternate format. Options include oneline, short, full, fuller, and format (where you specify your own 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).

Inną przydatną opcją jest -S, która przymuje ciąg i pokazuje tylko te rewizje w których dodano lub usunięto ten ciąg. Na przykład jeżeli chcesz znaleźć ostatnią rewizję, w której dodano lub usunięto odwłoanie do określonej funkcji, możesz wywołać:

$ git log -Sfunction\_name

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 [Opcje ograniczające rezultat git log](https://git-scm.com/book/pl/v2/ch00/limit_options) znajduje się ta jak i kilka innych często używanych opcji.

| Table 3. Opcje ograniczające rezultat git log | |
| --- | --- |
| **Option** | **Description** |
| -(n) | Show only the last n commits |
| --since, --after | Limit the commits to those made after the specified date. |
| --until, --before | Limit the commits to those made before the specified date. |
| --author | Only show commits in which the author entry matches the specified string. |
| --committer | Only show commits in which the committer entry matches the specified string. |
| --grep | Only show commits with a commit message containing the string |
| -S | Only show commits adding or removing code matching the string |

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 attributes are in use

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

f563754 - demonstrate breakage of detached checkout with symbolic link HEAD

d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths

51a94af - Fix "checkout --track -b newbranch" on detached HEAD

b0ad11e - pull: allow "git pull origin $something:$current\_branch" into an unborn branch

Z prawie 40000 rewizji w historii kodu Gita, podane polecenie wyłowiło jedynie 6 spełniających zadane kryteria.

# 2.4 Podstawy Gita - Cofanie zmian

## **Cofanie zmian**

W każdym momencie może zajść potrzeba cofnięcia jakiejś zmiany. Poniżej przyjrzymy się kilku podstawowym funkcjom cofającym modyfikacje. Bądź ostrożny, ponieważ nie wszystkie cofnięcia można odwrócić. Jest to jedno z niewielu miejsc Gita, w których należy być naprawdę ostrożnym, gdyż można stracić bezpowrotnie część pracy.

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)

renamed: README.md -> README

modified: CONTRIBUTING.md

Tekst znajdujący się zaraz pod nagłówkiem “Changes to be committed” 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 CONTRIBUTING.md:

$ git reset HEAD CONTRIBUTING.md

Unstaged changes after reset:

M CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

renamed: README.md -> README

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: CONTRIBUTING.md

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

|  |  |
| --- | --- |
| **Uwaga** | W chwili gdy wywołasz polecenie git reset z opcją --hard **może** być ono niebezpieczne, ale w tym przypadku plik z katalogu roboczego pozostaje nietknięty. Wywołanie git reset bez opcji jest bezpieczne - dotyczy tylko poczekalni. |

W tej chwili ta "magiczna inwokacja" jest wszystkim co powineneś wiedzieć na temat polecenia git reset. Zagłębimy się w szczegóły co robi polecenie reset oraz jak używać go do robienia naprawdę ciekawych rzeczy w rozdziale [Reset Demystified](https://git-scm.com/book/pl/v2/ch00/_git_reset).

### **Cofanie zmian w zmodyfikowanym pliku**

Co jeśli okaże się, że nie chcesz jednak zatrzymać zmian wykonanych w pliku CONTRIBUTING.md? 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: CONTRIBUTING.md

Git konkretnie wskazuje jak pozbyć się dokonanych zmian. Zróbmy zatem co każe Git:

$ git checkout -- CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

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

renamed: README.md -> README

Możesz teraz przeczytać, że zmiany zostały cofnięte.

|  |  |
| --- | --- |
| **Ważne** | Powinieneś sobie już także zdawać sprawę, że polecenie git checkout -- [file] jest dość niebezpieczne: 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 [Gałęzie Gita](https://git-scm.com/book/pl/v2/ch00/_git_branching) - 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 [Data Recovery](https://git-scm.com/book/pl/v2/ch00/_data_recovery)). Jednakże, cokolwiek utraciłeś a nie było to nigdy wcześniej zatwierdzane do repozytorium, prawdopodobnie odeszło na zawsze.

# 2.5 Podstawy Gita - Praca ze zdalnym repozytorium

## **Praca ze zdalnym repozytorium**

Aby móc współpracować w jakimkolwiek projekcie opartym na git, 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 https://github.com/schacon/ticgit

Cloning into 'ticgit'...

remote: Reusing existing pack: 1857, done.

remote: Total 1857 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.

Resolving deltas: 100% (772/772), done.

Checking connectivity... 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 https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

Jeśli posiadasz więcej niż jedno zdalne repozytorium polecenie wyświetli je wszystkie. Na przykład, kilka zdalnych repozytoriów dla kilku współpracowników może wyglądać tak.

$ cd grit

$ git remote -v

bakkdoor https://github.com/bakkdoor/grit (fetch)

bakkdoor https://github.com/bakkdoor/grit (push)

cho45 https://github.com/cho45/grit (fetch)

cho45 https://github.com/cho45/grit (push)

defunkt https://github.com/defunkt/grit (fetch)

defunkt https://github.com/defunkt/grit (push)

koke git://github.com/koke/grit.git (fetch)

koke git://github.com/koke/grit.git (push)

origin git@github.com:mojombo/grit.git (fetch)

origin git@github.com:mojombo/grit.git (push)

Oznacza to, że możesz szybko i łatwo pobrać zmiany z każdego z nich. Dodatkowo możemy mieć prawo do wysyłania(push) do jednego lub wielu z nich, jednak nie omawiamy tego w tej chwili.

Zauważ, że repozytoria te korzystają z różnych protokołów. W szczegółach zajmiemy się tym tematem w rozdziale [Uruchomienie Git na serwerze](https://git-scm.com/book/pl/v2/ch00/_git_on_the_server).

### **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 https://github.com/paulboone/ticgit

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

pb https://github.com/paulboone/ticgit (fetch)

pb https://github.com/paulboone/ticgit (push)

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 git fetch pb:

$ git fetch pb

remote: Counting objects: 43, done.

remote: Compressing objects: 100% (36/36), done.

remote: Total 43 (delta 10), reused 31 (delta 5)

Unpacking objects: 100% (43/43), done.

From https://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. (Czym są gałęzie i jak ich używać omwawiamy w [Gałęzie Gita](https://git-scm.com/book/pl/v2/ch00/_git_branching)).

### **Pobieranie i łączenie zmian ze zdalnych repozytoriów**

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

$ git fetch [remote-name]

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.

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, [Gałęzie Gita](https://git-scm.com/book/pl/v2/ch00/_git_branching)), 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.

### **Wysyłanie zmian do repozytoriów zdalnych**

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(rejected). 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 [Gałęzie Gita](https://git-scm.com/book/pl/v2/ch00/_git_branching).

### **Przegląd zdalnych repozytoriów**

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

Fetch URL: https://github.com/schacon/ticgit

Push URL: https://github.com/schacon/ticgit

HEAD branch: master

Remote branches:

master tracked

dev-branch tracked

Local branch configured for 'git pull':

master merges with remote master

Local ref configured for 'git push':

master pushes to master (up to date)

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: https://github.com/my-org/complex-project

Fetch URL: https://github.com/my-org/complex-project

Push URL: https://github.com/my-org/complex-project

HEAD branch: master

Remote branches:

master tracked

dev-branch tracked

markdown-strip tracked

issue-43 new (next fetch will store in remotes/origin)

issue-45 new (next fetch will store in remotes/origin)

refs/remotes/origin/issue-11 stale (use 'git remote prune' to remove)

Local branches configured for 'git pull':

dev-branch merges with remote dev-branch

master merges with remote master

Local refs configured for 'git push':

dev-branch pushes to dev-branch (up to date)

markdown-strip pushes to markdown-strip (up to date)

master pushes to master (up to date)

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

# 2.6 Podstawy Gita - Tagowanie

## **Tagowanie**

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 tagów**

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 500 tagów. Jeśli interesuje cię np. wyłącznie seria 1.8.5, możesz ją wyszukać w następujący sposób:

$ git tag -l 'v1.8.5\*'

v1.8.5

v1.8.5-rc0

v1.8.5-rc1

v1.8.5-rc2

v1.8.5-rc3

v1.8.5.1

v1.8.5.2

v1.8.5.3

v1.8.5.4

v1.8.5.5

### **Tworzenie tagów**

Git używa 2 głównych rodzajów tagów: 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 tagów 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: Ben Straub <ben@straub.cc>

Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949

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

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

changed the version number

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

### **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 ca82a6dff817ec66f44342007202690a93763949

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

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

changed the version number

### **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

Możesz sprawdzić czy utworzyłeś tag:

$ 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: 14, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (12/12), done.

Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.

Total 14 (delta 3), reused 0 (delta 0)

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: 1, done.

Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.

Total 1 (delta 0), reused 0 (delta 0)

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

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

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

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

### **Przełączanie tagów**

Nie możesz tak naprawdę przełączyć się do tagu, ponieważ nie można ich przemieszczać. Jeżeli chcesz umieścić wersję wyglądającą jak określony tag w swoim katalogu roboczym możesz utworzyć nową gałąź, która na nim bazuje poprzez git checkout -b [branchname] [tagname]:

$ git checkout -b version2 v2.0.0

Switched to a new branch 'version2'

Oczywiście kiedy to zrobisz i zatwierdzisz zmiany gałąź version2 i tag v2.0.0 będą się nieco różnić, ponieważ będzie przesuwała się do przodu wraz z nowymi zmianami, więc bądź ostrożny.

# 2.7 Podstawy Gita - Aliasy

## **Aliasy**

Przed zakończeniem tego rozdziału, przedstawimy jeszcze jedną wskazówkę, która uczyni pracę z gitem prostszą, łatwiejszą i przyjazną: aliasy.

Nie odnosiliśmy się do nich ani nie zakładaliśmy, że skorzystasz z nich w dalszej części książki, ale powinieneś wiedzieć jak ich używać.

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"

# 2.8 Podstawy Gita - Podsumowanie

## **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.