Systemy kontroli wersji

System Kontroli Wersji (VCS, Version Control System) rejestruje zmiany w pliku lub zestawie plików w czasie, dzięki czemu można później przywrócić określone wersje

System kontroli wersji pozwala:

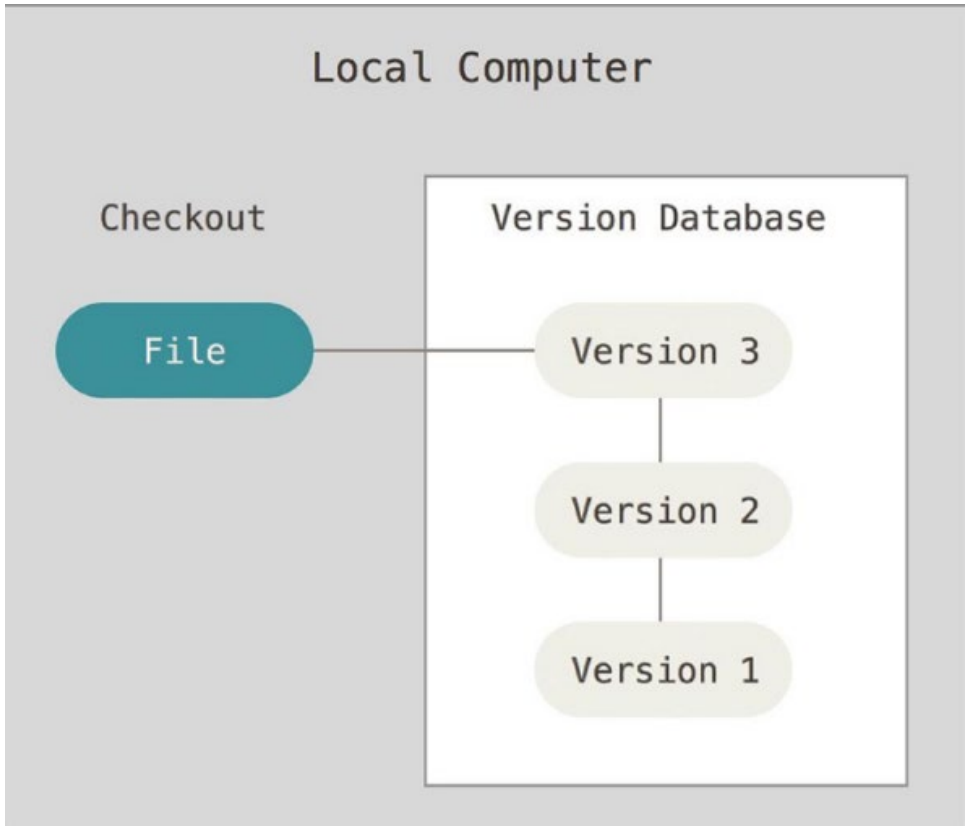
- przywrócić pliki do poprzedniego stanu
- przywrócić cały projekt do poprzedniego stanu
- porównać zmiany w czasie
- zobaczyć, kto ostatnio zmodyfikował coś, co może powodować problem
- kto wprowadził problem i kiedy, oraz więcej…

Korzystanie z VCS oznacza również, że jeśli coś zepsujesz lub utracisz pliki, możesz je łatwo odzyskać.

Kontrola wersji – kopiowanie, RCS

Metoda kontroli wersji (stosowana przez wielu ludzi):

- kopiowanie plików do innego katalogu (np. ze znacznikami czasu)
- zaleta: proste, wada: niezwykle podatne na błędy



Lokalne VCS, z prostą bazę danych, która zachowywałaby wszystkie zmiany w plikach pod kontrolą wersji.

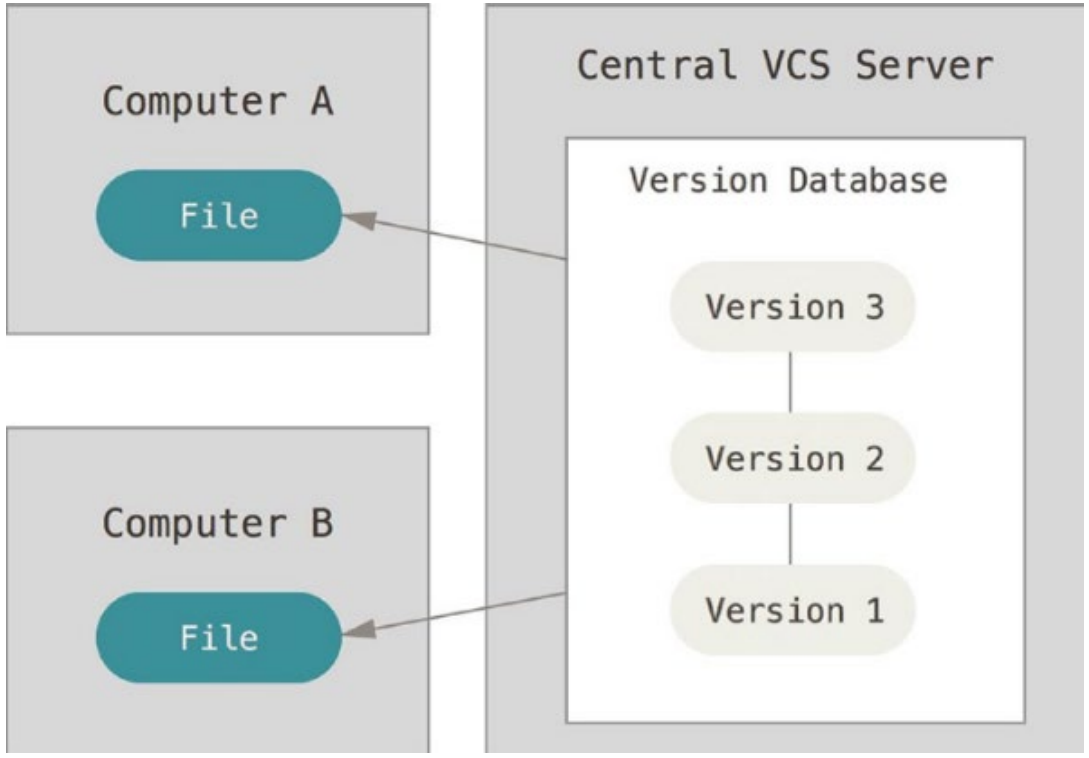
RCS (Revision Control System), nadal dystrybuowany z wieloma komputerami (Unix, Apple OS)

- działa poprzez utrzymywanie zestawów poprawek (czyli różnic między plikami) w specjalnym formacie na dysku
- może następnie odtworzyć, jak wyglądał dowolny plik w dowolnym momencie, dodając wszystkie poprawki

Kontrola wersji – CVCS systemy centralne

Współpraca z programistami z innych systemów:

- scentralizowane systemy kontroli wersji (CVCS) – CVS, Subversion i Perforce
- mają pojedynczy serwer, który zawiera wszystkie wersje plików i wielu klientów, którzy pobierają pliki z tego centralnego miejsca. Od wielu lat jest to standard kontroli wersji.



Zalety (zwłaszcza w lokalnych systemach VCS)

- każdy wie do pewnego stopnia co robią wszyscy inni w projekcie
- administratorzy mają precyzyjną kontrolę nad tym, co mogą robić
- znacznie łatwiej jest zarządzać CVCS, niż zajmować się lokalnymi bazami danych na każdym kliencie.

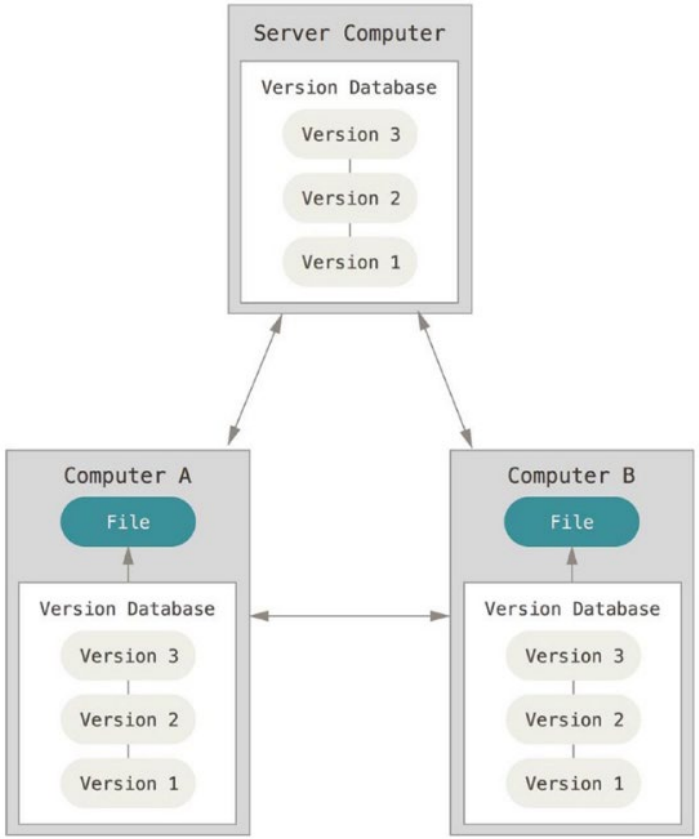
Wady

- pojedynczy punkt awarii, który scentralizowany serwer reprezentuje – gdy wyłączony, nikt nie może współpracować ani zapisywać zmian wersji
- jeśli dysk twardy, na którym znajduje się centralna baza danych, zostanie uszkodzony, (i brak kopii zapasowych), stracisz wszystko - całą historię projektu, z wyjątkiem pojedynczych migawek, które ludzie mają na swoich komputerach lokalnych

Kontrola wersji – DVCS systemy rozproszone

Rozproszone systemy kontroli wersji (DVCS – Distributed VCS)

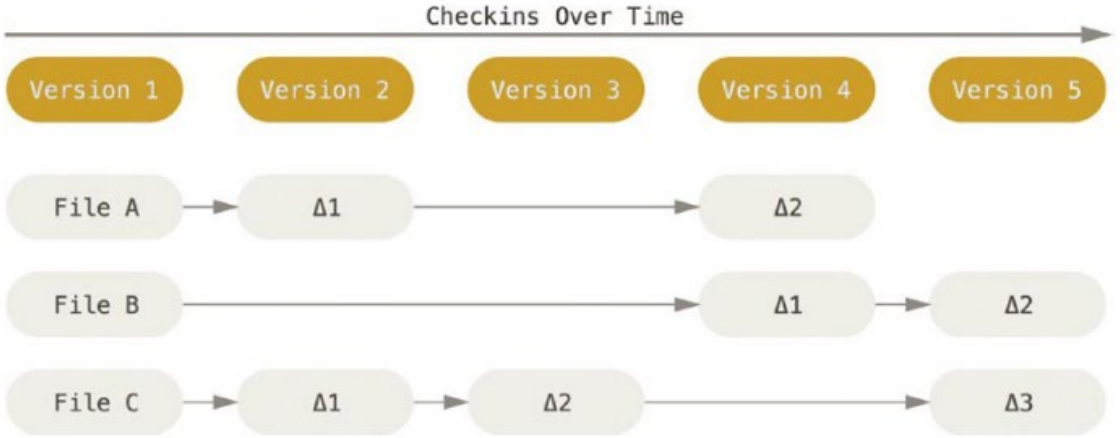
- Git, Mercurial, Bazaar lub Darcs
- nie tylko najnowsza migawka plików - w pełni odzwierciedlają repozytorium
- każda realizacja zamówienia jest w rzeczywistości pełną kopią zapasową wszystkich danych

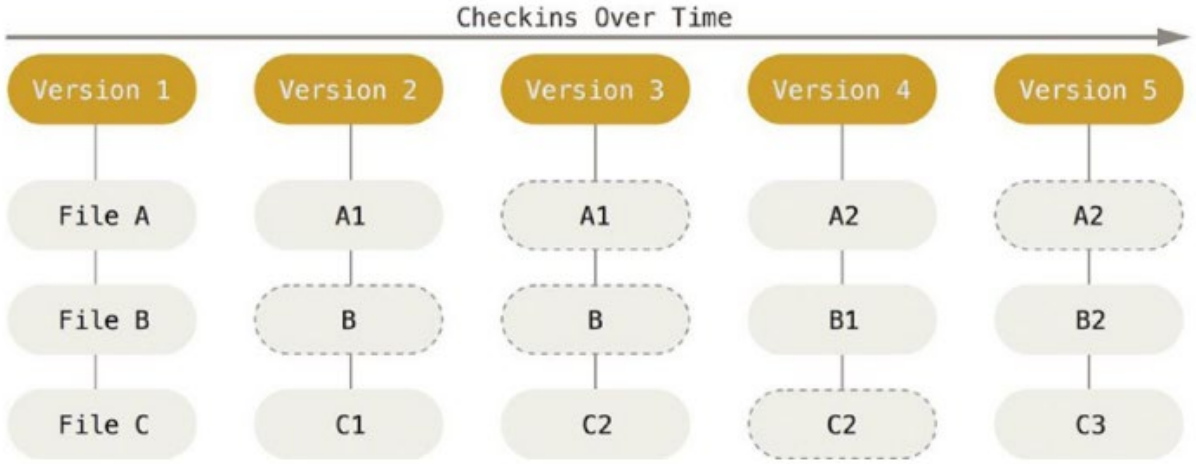


- w przypadku awarii serwera, dowolne repozytoria klientów mogą zostać skopiowane z powrotem na serwer, aby go przywrócić
- współpraca z wieloma zdalnymi repozytoriami, umożliwiająca współpracę z różnymi grupami ludzi jednocześnie
- można skonfigurować wiele typów organizacji pracy (niedostępne w systemach centralnych)

Kontrola wersji – metodyka

Większość systemów przechowuje informacje jako listę zmian opartych na plikach. Systemy te (CVS, Subversion, Perforce, Bazaar itd.) traktują przechowywaną informację jako zbiór plików, oraz zmiany dokonane w każdym pliku w czasie.





Za każdym razem, gdy zatwierdzasz lub zapisujesz stan swojego projektu w Git, robi on zdjęcie tego, jak wyglądają wszystkie w danym momencie i zapisuje odniesienie do tej migawki.
Jeśli plik się nie zmienił, Git nie zapisuje pliku ponownie, tylko link do poprzedniego identycznego pliku, który już zapisał.
Git traktuje dane jak strumień kolejnych migawek.

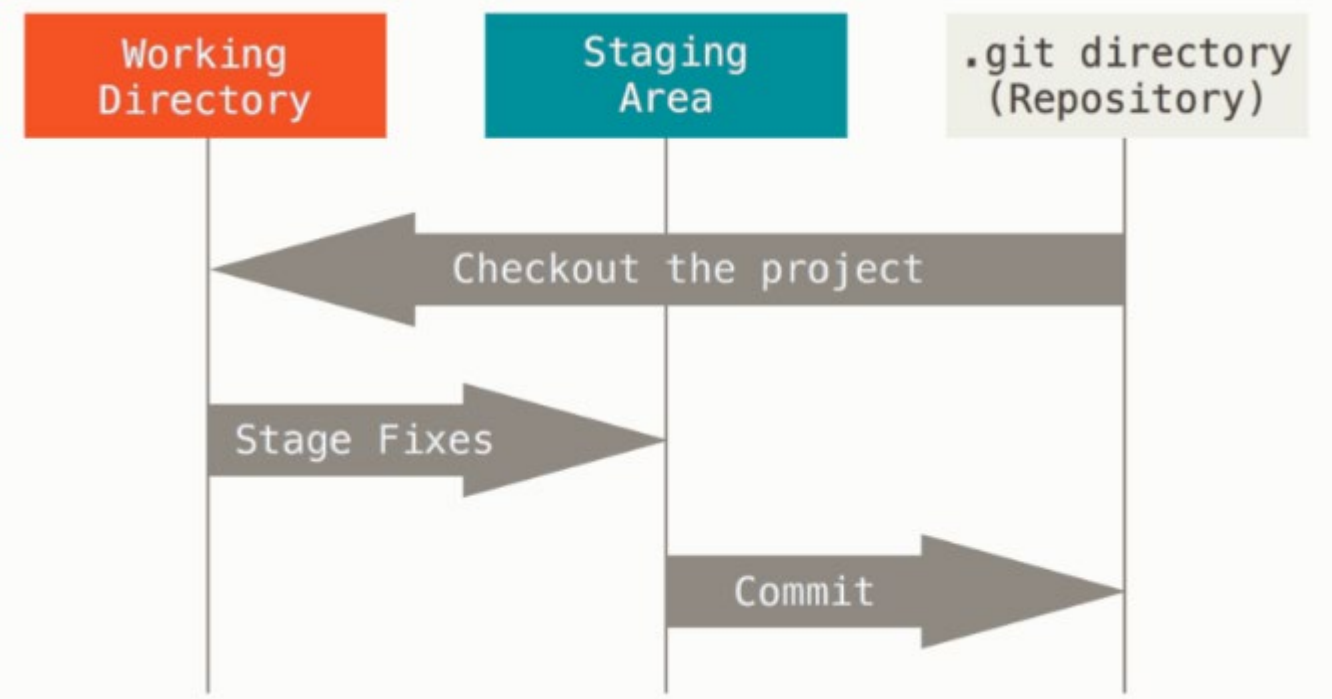
Git – lokalne repozytorium

- większość operacji – działania na lokalnych plikach
- przeglądanie historii projektu – nie trzeba łączyć się z serwerem, odczyt z lokalnej bazy danych, można pracować i zatwierdzać zmiany nie mając sieci
- mechanizmy spójności (każdy obiekt ma zapisaną sumę kontrolną)
  - suma kontrolna oparta o skrót SHA-1, 40-znakowy łańcuch (liczb hex), wyliczany na podstawie zawartości pliku lub struktury katalogu
  - Git przechowuje wszystko w swojej bazie danych, w której kluczami są skróty SHA-1, a wartościami - zawartości plików, czy struktur katalogów
  - każdy błąd (utrata, uszkodzenie) są wykrywane przez Git
  - Git dodaje dane do bazy, można śmiało eksperymentować bez lęku o utratę czy zepsucie czegoś

Git – trzy stany plików

- zatwierdzony (comitted), zmodyfikowany (modified), śledzony (staged)
- **comitted** – dane zostały bezpiecznie zachowane w lokalnej bazie danych
- **modified** – plik został zmieniony, ale zmian nie wprowadzono do bazy danych
- **staged** – zmodyfikowany plik został przeznaczony do zatwierdzenia w bieżącej postaci w następnej operacji commit

Trzy sekcje Git: **katalog Git** (Git directory), **katalog roboczy** (working directory), **przechowalnia** (staging area)



Git – stany plików i sposób pracy

- katalog Git (.git) – miejsce na własne metadane oraz obiektową bazę danych, ten katalog jest kopiowany podczas klonowania repozytorium z innego komputera
- katalog roboczy – obraz jednej wersji projektu, jego zawartość jest pobierana z bazy danych Git i umieszczana w pożądanym miejscu (z możliwością czytania i pisania)
- przechowalnia – plik (zwykle w katalogu .git) z informacją, czego będzie dotyczyć następna operacja commit

Praca z Git

- modyfikujesz pliki w katalogu roboczym
- oznaczasz zmodyfikowane pliki jako śledzone, dodajesz ich bieżący stan (snapshot, migawkę) do przechowalni
- zatwierdzasz (commit), wtedy zawartość plików z przechowalni jest zapisana jako migawka projektu w katalogu Git

Instalacja: (program, opis na stronie) **https://git-scm.com/**

Konfiguracja Git

Narzędzie: `git config` może zapisać konfigurację w trzech miejscach

- plik /etc/gitconfig globalna informacja dla wszystkich repozytoriów
  (git config z opcją `--system`, musisz mieć uprawnienia)
- plik ~/.gitconfig specyficzna dla danego użytkownika
  (git config z opcją `--global`)
- plik w katalogu Git, .git/config, konfiguracja bieżącego repozytorium
  (git config z opcją `--local`, najpierw musi istnieć repozytorium)

Tożsamość

```
$ git config --global user.name "Adam Nowak"
$ git config --global user.email adam.nowak@pwsz-ns.edu.pl
```

Edytor

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

Konfiguracja Git – sprawdzanie ustawień

Wyświetlenie konfiguracji

```
$ git config --list
```

Niektóre zmienne mogą się pojawić wiele razy (Git odczytuje informację z różnych plików, np. z /etc/gitconfig oraz ~/.gitconfig, korzystając z ostatniej wartości danej zmiennej, którą znajduje). Można też sprawdzić wybraną zmienną, git config { zmienna }

```
$ git config user.name
```

Uzyskiwanie pomocy

Podpowiedzi działają lokalnie, w każdej chwili:

`$ git help <verb>` (na przykład: `git help config`)

`$ git <verb> --help` (na przykład: `git config --help`)

`$ man git-<verb>` (na przykład: `man git-config`)

Pierwsze repozytorium Git-a

Rozpoczęcie śledzenia zmian w plikach istniejącego projektu

przejdź do katalogu projektu i wykonaj polecenie:

```
$ git init
Initialized empty Git repository in /u/przygoda/LESSON/.git/
```

Powstaje szkielet repozytorium Git-a, ale żadna część projektu nie jest jeszcze śledzona.
Dodajmy kilka plików (AAA.txt, BBB.txt, LICENCE) do śledzenia:

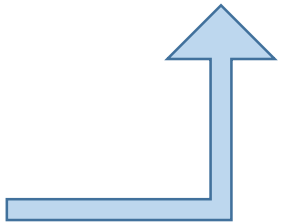
```
$ git add *.txt
$ git add LICENCE
```

Za pomocą **`git add`** można dodać również cały podkatalog.

i zawierdźmy zmiany:

```
$ git commit –m "pierwsza wersja"
```

Uwaga! GitHub to nie jest jedyna opcja na przetrzymywania zdalnego repozytorium. Inne, równie popularne, to **GitLab**, **Bitbucket**, **SourceForge**. Można też postawić **własny serwer** (działający jak GitLab)



Klonowanie istniejącego repozytorium: `git clone`, na przykład:

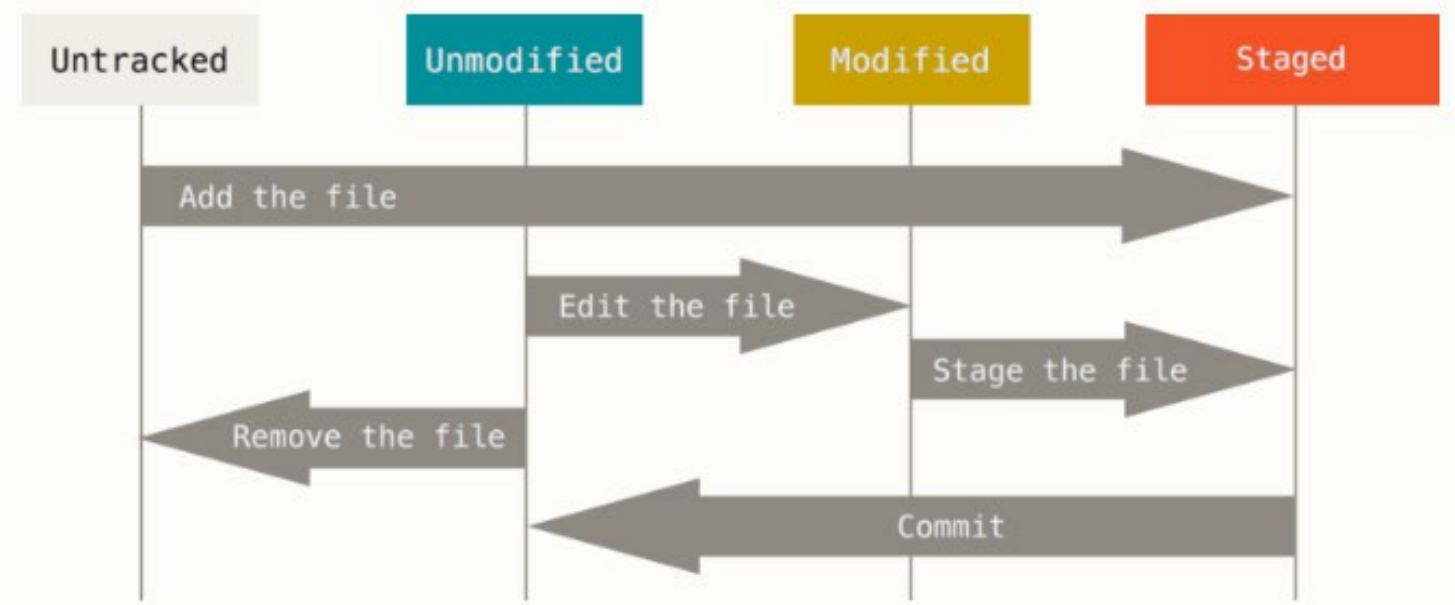
```
$ git clone https://github.com/pwszns/TString <new_name>
```

Powstanie podkatalog TString (wraz z podkatalogiem .git) lub (jeśli podano) new_name

Rejestrowanie zmian

Cykl śledzenia i zatwierdzania

- stan każdego pliku w katalogu roboczym: 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 katalogu roboczym, które nie znalazły się w ostatniej migawce i nie znajdują się w poczekalni, gotowe do zatwierdzenia
- po sklonowaniu repozytorium, wszystkie twoje pliki będą śledzone i niezmodyfikowane



Sprawdzanie stanu

Sprawdzanie stanu plików

```
$ git status
```

Bezpośrednio po wykonaniu `clone` lub `commit`, zobaczymy (na razie mamy gałąź "master "):

```
On branch master
nothing to commit, working directory clean
```

Utwórzmy nowy plik README i sprawdźmy `git status`. Git informuje o nieśledzonym pliku:

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

Dodajmy plik do śledzenia (`git add README`) i sprawdźmy:

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

Zmodyfikujmy któryś plik (np. AAA.txt) i sprawdźmy status:

Changes not staged for commit: (→ Zmienione ale nie zaktualizowane)

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

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

Sprawdzanie stanu

Uruchommy: `git add AAA.txt` i sprawdźmy:

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        modified:   AAA.txt
```

`git add` (wielozadaniowe polecenie) – używa się go do rozpoczynania śledzenia nowych plików, umieszczania ich w poczekalni, oraz innych zadań, takich jak oznaczanie rozwiązanych konfliktów scalania

Zmieńmy teraz ponownie AAA.txt i sprawdźmy status. Okaże się, że AAA.txt jest teraz zarówno w poczekalni jak i poza nią! Git dodał AAA.txt do poczekalni (git add), kolejne zmiany to kolejna wersja. Aby ją dodać do poczekalni, należy ponowić git add.

Skrócona wersja sprawdzania stanu: `git status --short` lub `git status --s`

?? nieśledzone pliki

A nowe pliki dodane do poczekalni

M zmodyfikowane pliki

Mamy też dwie kolumny: lewa wskazuje na to czy plik jest w poczekalni (staged) a prawa czy jest zmieniony:

MM AAA.txt

?? README

Przykład z książki:

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

Ignorowanie plików

Pewne pliki chcemy wykluczyć z raportu (śledzenia bądź nieśledzenia)

W tym celu tworzymy plik `.gitignore` wpisując do niego maski plików, które chcemy pominąć:

```
*.[oa]
*~
```

← ignorowanie plików kończących się na .o lub .a

← ignorowanie plików kończących się ~

Zasady przetwarzania wyrażeń z pliku `.gitignore`:

- puste linie lub linie rozpoczynające się od **#** są ignorowane
- działają standardowe wyrażenia `glob`
- zakończenie wyrażenie znakiem ukośnika (/) oznacza katalog
- negowanie wyrażeń rozpoczynanych wykrzyknikiem (!)

Wyrażenia `glob` (patterns) są jak uproszczone wyrażenia regularne (regular expressions), używane przez powłokę. **Maski**: * zero lub więcej znaków, [abcd] dowolny znak w nawiasie, ? pojedynczy dowolny znak, [0-9] dowolny znak z przedziału 0 do 9. Można użyć ** do dopasowania zagnieżdżonych katalogów: a/**/z oznacza np. a/z, a/b/z, a/b/c/z

Przykład:

```
# bez plików o rozszerzeniu .a
*.a
# za wyjątkiem pliku lib.a, który ma być śledzony
!lib.a
# ignoruj plik TODO w głównym drzewie, nie subdir/TODO
/TODO
# ignoruj wszystkie pliki w katalogu build/
build/
# ignoruj doc/*.txt, ale nie np. doc/server/*.txt
doc/*.txt
# ignoruj wszystkie pliki .txt w (pod)katalogu doc/
doc/**/*.txt
```

Wiele gotowych plików `.gitignore` dla różnych rodzajów projektów znajdziesz na stronie: **https://github.com/github/gitignore**

Podgląd zmian (katalog, poczekalnia, repo)

$ git diff

wyświetli zmiany w plikach nie wysłane do poczekalni, np.

```
$ git diff
diff --git a/AAA.txt b/AAA.txt
index c2d99b1..ecd51af 100644
--- a/AAA.txt
+++ b/AAA.txt
@@ -1,2 +1,3 @@
 This is AAA file.
 Pierwsza zmiana.
+Druga zmiana.
```

`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 zmiany są już w poczekalni, wynik git diff będzie pusty.

$ git diff --staged

$ git diff --cached

wyświetli zmiany w plikach w poczekalni, które zostaną po zatwierdzeniu wysłane do repozytorium:

```
git diff --staged
diff --git a/AAA.txt b/AAA.txt
index e86d377..c2d99b1 100644
--- a/AAA.txt
+++ b/AAA.txt
@@ -1 +1,2 @@
 This is AAA file.
+Pierwsza zmiana.
```

$ git difftool

$ git difftool --staged

graficzne wyświetlenie różnic.
Można użyć różnych programów, gdy nie jest skonfigurowane, otrzymamy:

```
$ git difftool

This message is displayed because 'diff.tool' is not configured.
See 'git difftool --tool-help' or 'git help config' for more details.
'git difftool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld kompare gvimdiff diffuse
diffmerge ecmerge p4merge araxis bc3 codecompare emerge
vimdiff

Viewing (1/1): 'AAA.txt'
Launch 'kdiff3' [Y/n]:
```

$ git difftool --tool-help

'git difftool --tool=<tool>' may be set to one of the following:
Wybierz twój ulubiony program!

Zatwierdzanie zmian

$ git commit otworzy edytor:

```
# 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:
#>.....modified:   AAA.txt
#
# Changes not staged for commit:
#>.....modified:   AAA.txt
#
```

Można odkomentować i/lub coś dopisać.
Jeśli nic nie zrobimy, commit zostanie odrzucony: Aborting commit due to empty commit message.
Po zatwierdzeniu linie zakomentowane zostaną usunięte.

$ git commit -v

jeśli chcemy podejrzeć zmiany.
Po opuszczeniu edytora, Git stworzy nową migawkę opatrzoną twoim opisem zmian (uprzednio usuwając z niego komentarze i podsumowanie zmian).

$ git commit -m "Dodana linia"

dodanie alternatywnego opisu rewizji razem z poleceniem.

```
$ git commit -m "Druga linia dodana"
[master b90c8fd] Druga linia dodana
 1 file changed, 1 insertion(+)
```

Sama operacja rewizji zwróciła dodatkowo garść informacji, między innymi, gałąź do której dodano zmiany (master), ich sumę kontrolną SHA-1 (b90c8fd), ilość zmienionych plików oraz statystyki dodanych i usuniętych linii kodu.
Każde wywołanie git commit powoduje zapamiętanie migawki projektu, którą można następnie odtworzyć albo porównać z inną migawką.

Pomijanie poczekalni:

$ git commit -a -m "Plik B aktulizacja"

opcja -a pozwala na pominięcie git add, każdy zmieniony plik, który jest już śledzony, automatycznie trafi do poczekalni.

```
[master 4731266] Plik B aktualizacja
 1 file changed, 1 insertion(+)
```

Usuwanie plików, zmiana nazwy

$ git rm

usuwa ze zbioru plików śledzonych i z katalogu roboczego. Jeśli najpierw usuniesz plik (rm BBB.txt) z katalogu, to zobaczysz go w sekcji "Zmienione ale nie zaktualizowane" (Changes not staged for commit). **W wersji skróconej**:

```
$ git status -s
 D BBB.txt
```

`git rm BBB.txt` doda wtedy do poczekalni operację usunięcia pliku. W obu powyższych przypadkach, po kolejnej rewizji, plik zniknie i nie będzie dłużej śledzony.

Jeśli plik do usunięcia został wcześniej zmodyfikowany i dodany do poczekalni, np.

```
$ git rm CCC.txt
error: the following file has changes staged in the index:
    CCC.txt
(use --cached to keep the file, or -f to force removal)
```

to konieczna jest opcja -f, by chronić przed usunięciem danych, które nie zostały jeszcze zapamiętane w żadnej migawce:

```
$ git rm –f CCC.txt
```

Do polecenia `git rm` można podać pliki, katalogi, wzory masek (glob), np.

```
$ git rm log/\*.log
$ git rm \*~
```

dodatkowy ukośnik, bo Git wykonuje własne rozwinięcie, oprócz tego z powłoki

---

Usunięcie plików ze śledzenia, ale bez fizycznego usunięcia w katalogu roboczym:

$ git rm --cached LICENCE

```
$ git rm --cached LICENCE
rm 'LICENCE'
```

użyteczne, gdy przekazaliśmy do śledzenia niechcący niewłaściwe pliki.

Zmiana nazwy pliku w repozytorium
(przekazana do poczekalni - zatwierdzenia):

```
$ git mv AAA.txt AAAA.txt
```

Równoważne jest to trzem operacjom:

```
$ mv AAA.txt AAAA.txt
$ git rm AAA.txt
$ git add AAAA.txt
```

Historia rewizji

$ git log

```
commit d005e7d6347961d8f3e48e1c43c2c593245a45fa
Author: Witold Przygoda <wprzygoda@pwsz-ns.edu.pl>
Date:   Mon Mar 18 00:21:59 2019 +0100

    Raz usuwam, raz dodaje

commit 473126658d34883d442f2fad0afd820b0a472c44
Author: Witold Przygoda <wprzygoda@pwsz-ns.edu.pl>
Date:   Sun Mar 17 23:51:23 2019 +0100

    Plik B aktualizacja
```

bez argumentów, listuje zmiany zatwierdzone w repozytorium w odwrotnej kolejności chronologicznej (najnowsze zmiany w pierwszej kolejności). Polecenie wyświetliło zmiany wraz z ich sumą kontrolną SHA-1, nazwiskiem oraz e-mailem autora, datą zapisu oraz notką zmiany.

$ git log -p -1

opcja `-p` pokazuje różnice wprowadzone z każdą rewizją. Opcja `-1` (ogólnie `-N`) ogranicza zbiór do jednego (N) ostatnich wpisów.

$ git log --stat

dodatkowo, pokazuje skrócone statystyki zatwierdzonych zmian. Pod każdym wpisem jest historia listy zmodyfikowanych plików, liczba zmienionych plików oraz liczba dodanych i usuniętych linii.

```
commit d005e7d6347961d8f3e48e1c43c2c593245a45fa
Author: Witold Przygoda <wprzygoda@pwsz-ns.edu.pl>
Date:   Mon Mar 18 00:21:59 2019 +0100

    Raz usuwam, raz dodaje

diff --git a/BBB.txt b/BBB.txt
deleted file mode 100644
index c2d0fa5..0000000
--- a/BBB.txt
+++ /dev/null
@@ -1,2 +0,0 @@
-This is BBB file.
-Zmiana pliku B.
diff --git a/CCC.txt b/CCC.txt
new file mode 100644
index 0000000..e5a22d4
--- /dev/null
+++ b/CCC.txt
@@ -0,0 +1 @@
+Kolejny CCC plik.
```

$ git log --pretty=<opt> (opt = oneline, short, full, fuller, …)

pokazuje wynik polecenia `git log` w określonym formacie (np. `oneline` – w jednej linii).

Historia rewizji – formatowanie

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

opcja `format` pozwala określić własny wygląd i format informacji wyświetlanych poleceniem `log`. Funkcja przydaje się szczególnie podczas generowania informacji do dalszego, maszynowego przetwarzania.

```
d005e7d - Witold Przygoda, 8 hours ago : Raz usuwam, raz dodaje
4731266 - Witold Przygoda, 9 hours ago : Plik B aktualizacja
b90c8fd - Witold Przygoda, 9 hours ago : Druga linia dodana
14e077c - Witold Przygoda, 9 hours ago : Pierwsza poprawka.
7b713f1 - Witold Przygoda, 11 hours ago : pierwsza wersja
```

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 (w --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

```
$ git log --pretty=oneline --graph
```

opcja `--graph` tworzy mały graf ASCII pokazujący historię gałęzi oraz scaleń (przykład książkowy):

```
$ 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 ważny przy listowaniu tworzenia gałęzi i ich scalania.

Historia rewizji – opcje formatowania

Podsumowanie najpopularniejszych opcji formatowania:

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).

Ograniczenie wyniku historii. Opcje ograniczania w czasie: `--since` (od) oraz `--until` (do). Na przykład, poniższe polecenie pobiera listę zmian dokonanych w ciągu ostatnich dwóch tygodni:

```
$ git log --since=2.weeks
```

Polecenie to obsługuje wiele formatów - można uściślić konkretną datę (np. "2019-03-18") lub podać datę względną jak np. 3 lata 1 dzień 4 minuty temu.

Inne opcje. Opcja `--author` pozwala wybierać po konkretnym autorze, opcja `--grep` na wyszukiwanie po słowach kluczowych zawartych w notkach zmian. Jeżeli potrzeba określić zarówno autora jak i słowa kluczowe, konieczne jest dodanie opcji `--all-match`

Historia rewizji – opcje formatowania

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

Np., żeby zobaczyć wyłącznie rewizje modyfikujące pliki testowe w historii plików źródłowych Git-a zatwierdzonych przez gitster, 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/
```

`$ git log -Sname`

przyjmuje ciąg `name` i pokazuje tylko te rewizje w których **dodano** lub **usunięto** ten ciąg (nie opis). Na przykład jeżeli chcemy znaleźć ostatnią rewizję, w której dodano lub usunięto odwołanie do określonej funkcji, możemy 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.

Git – cofanie zmian

$ git commit --amend

naprawia ewentualną pomyłkę poprzedniego `commit`. Polecenie bierze zawartość poczekalni i zatwierdza jako dodatkowe zmiany. Jeśli nic się nie zmieniło od ostatniej rewizji, to bieżąca migawka się nie zmieni, ale będzie

```
$ 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.

Wycofanie plików z poczekalni (np. dodanych przez pomyłkę):

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   ABA.txt
        new file:   ABB.txt
        new file:   ABC.txt
```

$ git reset HEAD <file>

gdzie `<file>` to nazwa albo maska wyboru plików.

Jeśli dokonaliśmy zmian w katalogu roboczym:

```
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:   ABA.txt
```

i chcemy przywrócić poprzedni stan pliku

$ git checkout -- <file>

to: jeśli w poczekalni jest któraś wersja pliku, to ona nadpisuje wersję z katalogu roboczego; jeśli nie ma w poczekalni, to przywrócona jest wersja z repozytorium. Uwaga: plik w katalogu zostaje nadpisany!

Wszystko co zatwierdzono do repozytorium Git-a może zostać w niemalże dowolnym momencie odtworzone. Nawet rewizje, które znajdowały się w usuniętych gałęziach, albo rewizje nadpisane zatwierdzeniem poprawiającym `--amend` mogą być odtworzone. Jednakże, gdy coś nie było nigdy wcześniej zatwierdzane do repozytorium, a zostało utracone (zmazane, nadpisane), jest nie do odzyskania.

Git – zdalne repozytorium

Zdalne repozytorium to wersja twojego projektu utrzymywana na serwerze dostępnym poprzez Internet lub inną sieć. Aby zobaczyć obecnie skonfigurowane serwery:

```
$ git remote
```

Sklonujmy raz jeszcze jakieś repozytorium, np.

```
git clone https://github.com/pwszns/TString
```

i w katalogu TString komenda: `git remote` da odpowiedź:

```
origin
```

, a komenda: `git remote -v` pokaże:

```
origin  https://github.com/pwszns/TString (fetch)
origin  https://github.com/pwszns/TString (push)
```

Jeśli posiadasz więcej niż jedno zdalne repozytorium polecenie wyświetli je wszystkie (przykład książkowy):

```
$ 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)
```

Można szybko i łatwo pobrać zmiany z każdego z nich. Dodatkowo możemy mieć prawo do wysyłania (push) do jednego lub wielu z nich. Repozytoria te korzystają z różnych protokołów.

```
$ git remote add [<opt>] <name> <url>
```

gdzie `<name>` to skrócona nazwa, zaś `<url>` to ścieżka do zdalnego repozytorium, np.

```
$ git remote add TS https://github.com/pwszns/TString
```

Pobranie zmian ze zdalnego projektu:

```
$ git fetch TS
```

```
warning: no common commits
remote: Enumerating objects: 26, done.
remote: Total 26 (delta 0), reused 0 (delta 0), pack-reused 26
Unpacking objects: 100% (26/26), done.
From https://github.com/pwszns/TString
 * [new branch]      master     -> TS/master
```

Tu główna gałąź jest dostępna jako `TS/master`

Po sklonowaniu repozytorium automatycznie zostanie dodany skrót o nazwie `origin` wskazujący na oryginalną lokalizację.
`$ git fetch origin` (może być z opcją `-v`)
pobierze każdą nową pracę jaka została wypchnięta na oryginalny serwer od momentu sklonowania lub ostatniego pobrania zmian.

```
From https://github.com/pwszns/TString
 = [up to date]      master     -> origin/master
```

Polecenie `fetch` pobiera dane do lokalnego repozytorium - nie scala jednak automatycznie zmian z żadnym z plików roboczych jak i w żaden inny sposób tych plików nie modyfikuje. Wszystkie zmiany muszą być scalone ręcznie.

Git – ściąganie i wysyłanie zmian

$ git pull (może być z opcją -v)

Jeśli gałąź lokalna jest ustawiona tak, żeby śledzić zdalną gałąź, wystarczy użyć polecenia `git pull`, żeby automatycznie pobrać dane (`fetch`) i je scalić (`merge`) z lokalnymi plikami.

(`git clone` ustawia lokalną gałąź główną `master` tak aby śledziła zmiany w zdalnej gałęzi master na serwerze z którego sklonowano repozytorium).

$ git push <repo> <branch>

Wysyłanie zmian na zdalny serwer (repozytorium) `<repo>` do gałęzi `<branch>`. Wysłanie gałęzi `master` na oryginalny serwer źródłowy (ustawiony przez `git clone`):

$ git push origin master

```
Username for 'https://github.com': pwszns
Password for 'https://pwszns@github.com':
Everything up-to-date
```

$ git remote show <repo>

(`git remote show origin`):

```
* remote origin
  Fetch URL: https://github.com/pwszns/TString
  Push  URL: https://github.com/pwszns/TString
  HEAD branch: master
  Remote branch:
    master 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)
```

aby zobaczyć więcej informacji o konkretnym zdalnym repozytorium.

$ git remote rename <oldname> <newname>

np. $ `git remote rename TS TSTR`

zmienia nazwę odnośnika, czyli skrótu przypisanego do repozytorium.

Polecenie zmienia także nazwy zdalnych gałęzi. To co do tej pory było określane jako `TS/master` od teraz powinno być adresowane jako `TSRT/master`.

Dany odnośnik można też usunąć:

$ `git remote rm <nazwa>`

(przykład książkowy)

```
$ 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)
```

Git – etykiety (tagowanie)

Git posiada możliwość etykietowania (dowolną nazwą) konkretnych wersji kodu (np. wersja 1.0, itd.). Wyświetlanie:

```
$ git tag
$ git tag -l '1.8.*'
```

Szukanie wg wzorca:

```
$ git tag
v0.1
v1.3
```

Etykiety lekkie (tylko nazwa, nie są przechowywane żadne inne, dodatkowe informacje):

$ git tag <nazwa>$_{opcjonalnie}$ np. $ git tag ver0.1

Wyświetlenie: $ git show <nazwa>$_{opcjonalnie}$

```
commit 49c627d6264abec224b2c4758184d7e78577f316
Author: Witold Przygoda <wprzygoda@pwsz-ns.edu.pl>
Date:   Mon Mar 25 00:05:09 2019 +0100

    ABA pierwszy raz

diff --git a/ABA.txt b/ABA.txt
new file mode 100644
index 0000000..ea7ca0d
--- /dev/null
+++ b/ABA.txt
@@ -0,0 +1 @@
+Pierwszy
```

Etykiety opisane (przechowywane jako pełne obiekty w bazie danych Git-a, opatrywane sumą kontrolną, zawierają nazwisko osoby etykietującej, jej adres e-mail oraz datę, posiadają notkę etykiety, oraz mogą być podpisywane i weryfikowane za pomocą GNU Privacy Guard (GPG)):

```
$ git tag -a 0.12-lw -m 'wersja long'
```

Wtedy: $ git show 0.12-lw

```
tag 0.12-lw
Tagger: Witold Przygoda <wprzygoda@pwsz-ns.edu.pl>
Date:   Mon Mar 25 08:35:34 2019 +0100

wersja long
```

Jeśli pominąć opis (opcję) -m otworzy się domyślny edytor z prośbą o wpisanie.

Git – tagowanie rewizji, przesyłanie

Można **dodać etykiety** do poprzednich wydań. Np. mamy:

$ `git log --pretty=oneline`

```
49c627d6264abec224b2c4758184d7e78577f316 ABA pierwszy raz
c25185325a68fe0bf32757ff85a180410f0da600 final solution
a420f9d80d2ba8d567c38fd037f993ef353bc622 Te pliki sa ok
a38a9b83648e6530a3c21c0fc2123b7780a945e0 Pierwsze wazne zmiany w plikach.
9b321b0bab9d1b2375db855f6af8f99c9131c1e4 pierwsza wersja kodu
```

W tym celu wystarczy na końcu polecenia git tag podać sumę kontrolną lub jej część wskazującą na odpowiednią rewizję:

$ `git tag -a verOK a420f9`

(można teraz sprawdzić: `git tag`, `git show tagOK`)

Przesyłanie etykiet. Domyślnie, polecenie `git push` nie przesyła etykiet do zdalnego repozytorium. Trzeba je osobno wypchnąć na współdzielony serwer. Proces ten jest podobny do współdzielenia gałęzi i polega na uruchomieniu:

$ `git push origin <nazwa>`

Można wysłać wiele etykiet naraz:

$ `git push origin --tags`

W ten sposób zostaną przesłane wszystkie tagi, których nie ma jeszcze na serwerze.

Z określonej etykiety można utworzyć nową gałąź:

$ `git checkout -b <nazwa> <etykieta>`

(`<nazwa>` to nowa gałąź, `<etykieta>` to tak oznaczona rewizja).

Git – aliasy (skróty)

Wygodne jest nadanie skrótowych nazw (**alias**es) najczęściej wykonywanym operacjom:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

→ zamiast `git commit` można napisać `git ci`

Można też przemianować (na bardziej intuicyjne):

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

i zamiast `git reset HEAD plikA` można napisać:

```
$ git unstage fileA
```

Dodanie skrótu `last` ułatwi zobaczenie ostatniej rewizji:

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

```
$ git last
```

```
commit 49c627d6264abec224b2c4758184d7e78577f316
Author: Witold Przygoda <wprzygoda@pwsz-ns.edu.pl>
Date:   Mon Mar 25 00:05:09 2019 +0100

    ABA pierwszy raz
```

Można też uruchomić (za pomocą skrótu) zewnętrzne narzędzie, współpracujące z Git-em. To zewnętrzne polecenie trzeba poprzedzić znakiem **!** Na przykład, uruchomienie graficznej nakładki **`gitk`** za pomocą skrótu `git visual`, po ustawieniu:

```
$ git config --global alias.visual '!gitk'
```

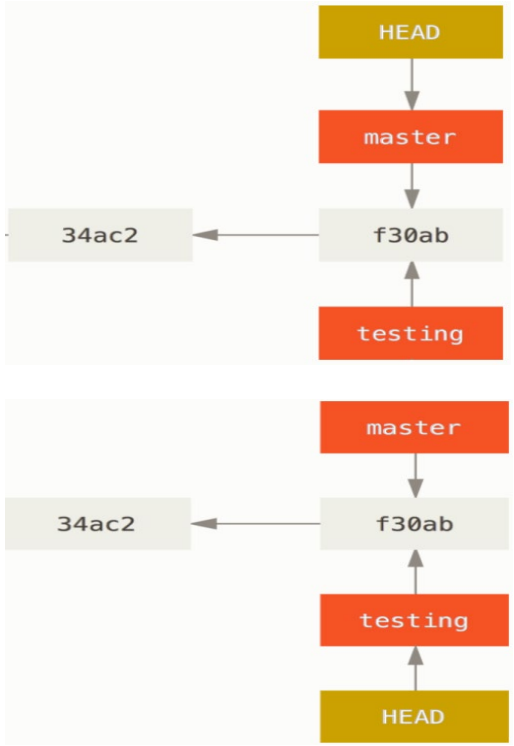
(uwaga: w pojedynczym cudzysłowie)

Git – gałęzie (branches)

Domyślna gałąź Git-a (tworzona podczas `git init`) to **master** (jest to gałąź taka sama jak wszystkie inne). Gałąź master wskazuje na ostatni zatwierdzony zestaw. Z każdym zatwierdzeniem automatycznie przesuwa się ona do przodu. Gałąź w Git jest lekkim, przesuwalnym wskaźnikiem, pokazującym na któryś z zestawów zmian. Wskaźnika o nazwie HEAD to lokalny wskaźnik pokazujący na aktywną w danej chwili gałąź.

Tworzenie nowej gałęzi (ale bez przełączenia na nią):
$ `git branch <nazwa>`, np. `git branch testing`
Wyświetlenie gdzie znajduje się wskaźnik HEAD:
$ `git log --oneline --decorate`

```
0859eae (HEAD, tag: ver1.0, tag: 0.123, testing, master) trzecia wersja teraz dobra
f9c65ec (tag: ver0.1) poczatkowa wersja
```



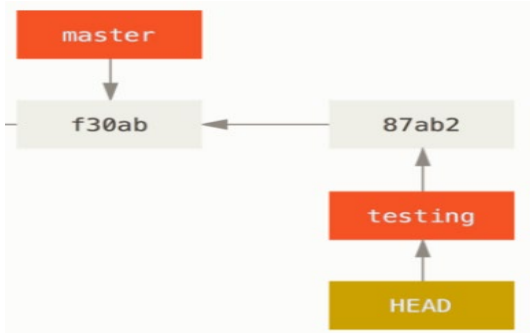
Git – gałęzie (branches)

Przełączenie na inną gałąź:

$ `git checkout <nazwa>`, np. `git checkout testing`

Wykonując teraz jakąś zmianę oraz robiąc commit:

```
b153e6a (HEAD, testing) Test
0859eae (tag: ver1.0, tag: 0.123, master) trzecia wersja teraz dobra
f9c65ec (tag: ver0.1) poczatkowa wersja
```

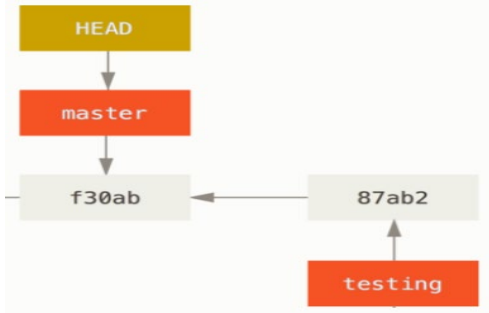


Gałąź wskazywana przez HEAD przesuwa się naprzód po każdym zatwierdzeniu zmian

Przełączenie z powrotem na gałąź master:

$ `git checkout master`

Wskaźnik HEAD jest z powrotem na gałęzi master i **przywrócone zostały pliki w katalogu roboczym do stanu z migawki, na którą wskazuje master**.

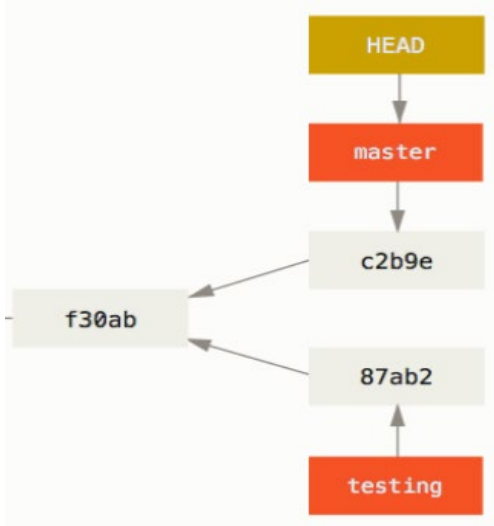


$ `git log --oneline --decorate --graph --all`

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Przełączanie gałęzi zmienia pliki w katalogu roboczym. Jeśli przełączysz się na starszą gałąź twój katalog roboczy zostanie cofnięty tak aby wyglądał jak w chwili zatwierdzenia ostatniej zmiany na danej gałęzi.

Po kolejnych zatwierdzonych zmianach, gałęzie mogą wyglądać następująco:



Git – gałęzie (rozgałęzianie, scalanie)

Tworzenie nowej gałęzi (`iss53`) i przełączenie na nią:

```
$ git checkout -b iss53
```

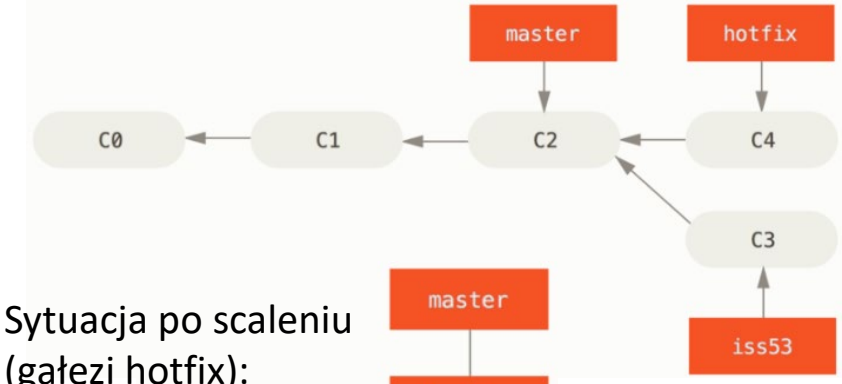
To samo za pomocą dwóch komend:

```
$ git branch iss53
$ git checkout iss53
```

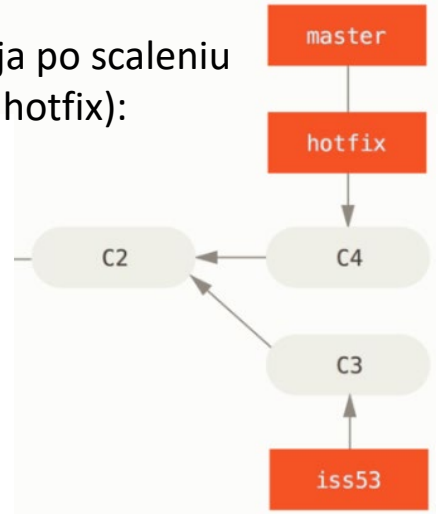
Wykonujemy jakieś zmiany, następnie commit.

Wracamy do gałęzi master. Ponownie, tworzymy nową gałąź, np.

`$ git checkout -b hotfix`, wykonajmy zmiany i commit. Sytuacja wygląda następująco:



Sytuacja po scaleniu (gałęzi hotfix):



Jeśli chcemy scalić zmiany z gałęzi hotfix do gałęzi master:

```
$ git checkout master
$ git merge hotfix
```

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

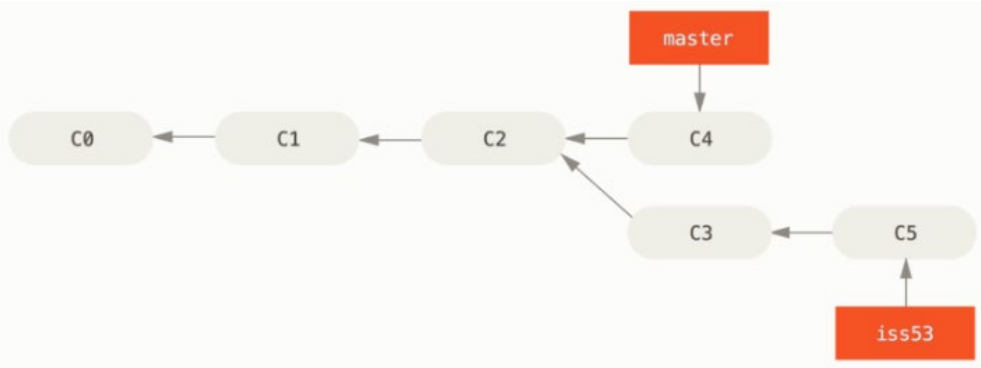
Zestaw zmian wskazywany przez scalaną gałąź był bezpośrednim rodzicem aktualnego zestawu zmian, Git przesuwa wskaźnik do przodu. Generalnie, jeśli scalamy zestaw zmian z innym, do którego dotrzeć można podążając wzdłuż historii tego pierwszego, Git upraszcza wszystko poprzez przesunięcie wskaźnika do przodu, ponieważ nie ma po drodze żadnych rozwidleń do scalenia — stąd "fast forward".

Niepotrzebną gałąź można usunąć:

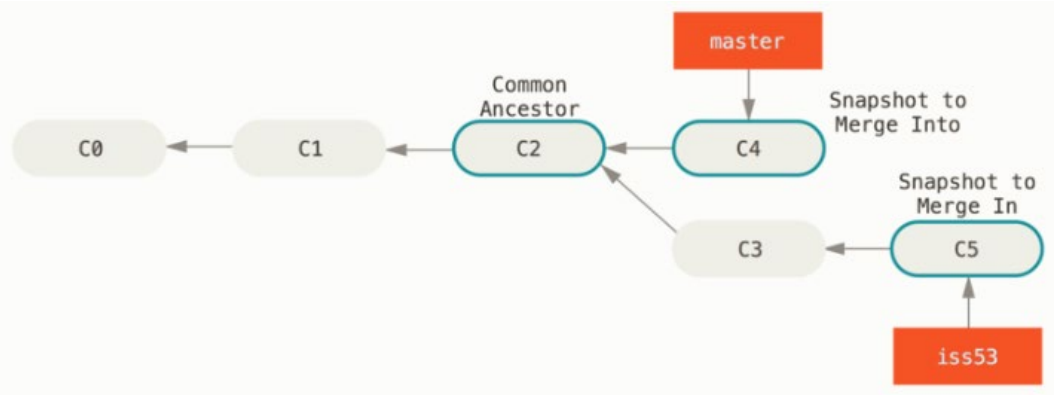
```
$ git branch -d hotfix
```

Git – gałęzie (rozgałęzianie, scalanie)

Kontynuacja pracy na gałęzi `iss53`:



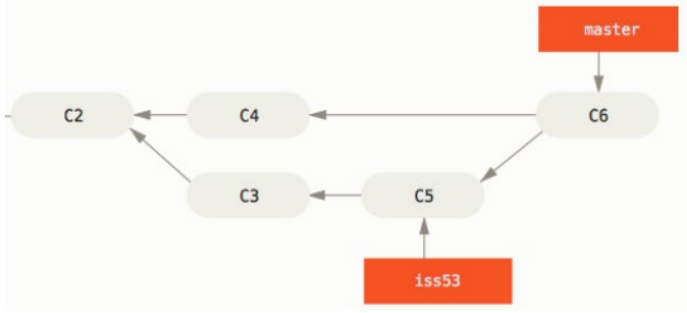
Można też scalić zmiany z gałęzi master do gałęzi iss53:
`$ git merge master`
albo odwrotnie, poczekać aż zmiany w iss53 się skończą i zechcemy je połączyć z gałęzią master. **Wtedy**:




```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

W tym wypadku historia rozwoju została rozszczepiona na wcześniejszym etapie. Ponieważ zestaw zmian z gałęzi master nie jest bezpośrednim potomkiem gałęzi scalanej (iss53), Git przeprowadza scalenie trójstronne (ang. **three-way merge**), używając dwóch migawek wskazywanych przez końcówki gałęzi oraz ich wspólnego przodka.

Git sam określa najlepszego wspólnego przodka do wykorzystania jako punkt wyjściowy scalenia.



Git – gałęzie (konflikty scalania)

Jeśli ten sam plik zmieniono w różny sposób w obu scalanych gałęziach, Git nie będzie w stanie scalić ich samodzielnie. Jeśli np. poprawka problemu #53 zmieniła tę samą część pliku, co zmiana w gałęzi hotfix, podczas scalania otrzymamy komunikat o konflikcie, typu:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Sprawdzenie, które pliki pozostałe niescalone w dowolnym momencie po wystąpieniu konfliktu:
$ `git status`

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

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

Git dodaje do problematycznych plików standardowe znaczniki rozwiązania konfliktu, trzeba te pliki otworzyć i samodzielnie rozwiązać konflikty. Np. plik może zawierać teraz taki fragment:

```
<<<<<<< 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
```

Wersja wskazywana przez HEAD (gałąź `master`, ponieważ tam byliśmy podczas uruchamiania polecenia scalania) znajduje się w górnej części bloku (wszystko powyżej ======), a wersja z gałęzi `iss53` to wszystko poniżej.
Aby rozwiązać konflikt, trzeba własnoręcznie zmodyfikować zawartość oraz usunąć <<<<<<<, ======= i >>>>>>>.

Po rozstrzygnięciu wszystkich takich sekcji w każdym z problematycznych plików, wykonać `git add` na każdym z nich, aby oznaczyć go jako rozwiązany.
Przeniesienie do poczekalni oznacza w Git rozwiązanie konfliktu.

Git – gałęzie (graficzne scalanie)

Rozwiązanie konfliktów scalania za pomocą narzędzia graficznego:

```
$ git mergetool
```

```
This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecom
pare emerge vimdiff
```

Odpowiednie narzędzie graficzne przeprowadzi przez wszystkie konflikty.

```
$ git mergetool --tool-help
$ git mergetool --tool=<tool>
```

Po opuszczeniu narzędzia do scalania, Git zapyta, czy wszystko przebiegło pomyślnie. Po zatwierdzeniu plik zostanie umieszczony w poczekalni, by konflikt oznaczyć jako rozwiązany. Można sprawdzić (`git status`), by upewnić się, że wszystkie konflikty zostały rozwiązane:

Aby potwierdzić, że wszystkie problematyczne pliki zostały umieszczone w poczekalni, można wpisać `git commit`, by tym samym zatwierdzić zestaw zmian scalających.

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

Git – branch (kilka informacji)

Wyświetla listę aktualnych gałęzi:

$ git branch

```
$ git branch
  iss53
* master
  testing
```

Informacja o ostatnich rewizjach:

$ 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
```

próba usunięcia niescalonej gałęzi, zawierającej zmiany, kończy się ostrzeżeniem:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Wymuszenie usunięcia gałęzi (i utrata wprowadzonych zmian:

$ git branch -D <nazwa>

Filtrowanie gałęzi, które

- zostały scalone z aktywną gałęzią:

$ git branch --merged

```
$ git branch --merged
  iss53
* master
```

gałęzie tu wypisane bez * można usunąć (git branch -d <nazwa>), bo zostały już scalone

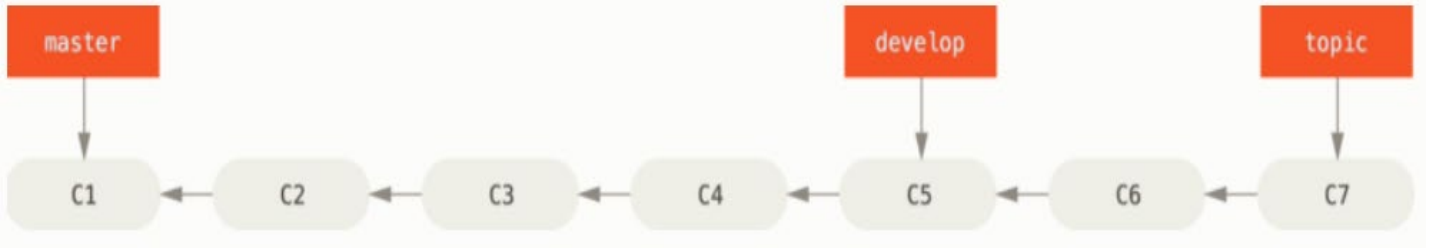
- nie zostały scalone z aktywną gałęzią:

$ git branch --no-merged

```
$ git branch --no-merged
  testing
```

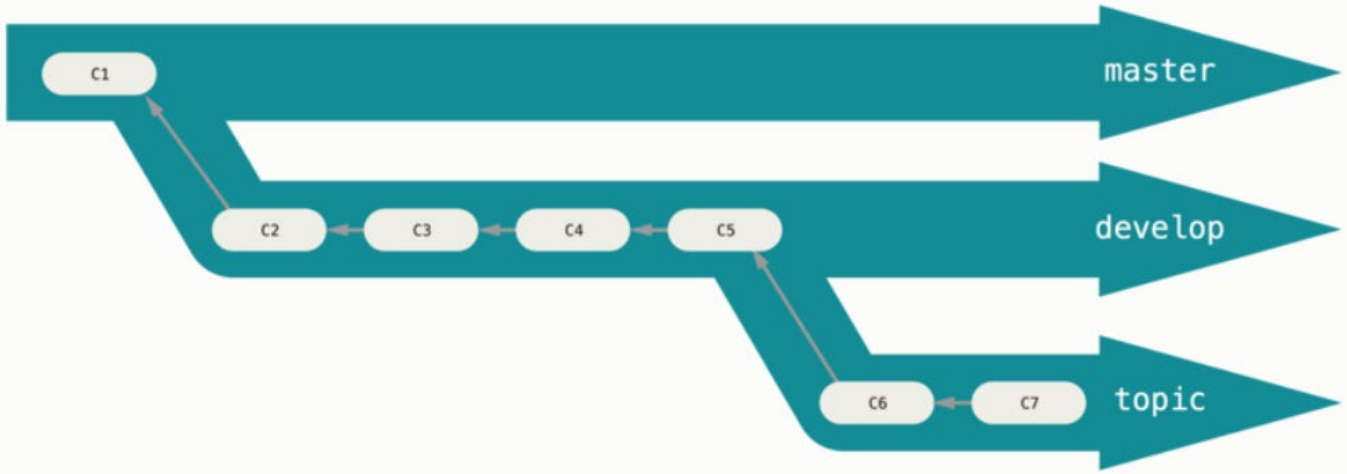
Git – branching workflows

Strategia: utrzymywanie wielu gałęzi celem zachowywania różnych poziomów rozwoju projektu (np. develop, feature, hotfix, master, release…).
W rzeczywistości mówimy o wskaźnikach przesuwających się wraz z kolejnymi rewizjami. Stabilne gałęzie znajdują się dalej w linii w historii zatwierdzania, a gałęzie z kodem zmienianym są dalej w historii.



Gałęzie są na różnych poziomach stabilności. Kiedy osiągną bardziej stabilny poziom, zostaną połączone w gałąź nad nimi:

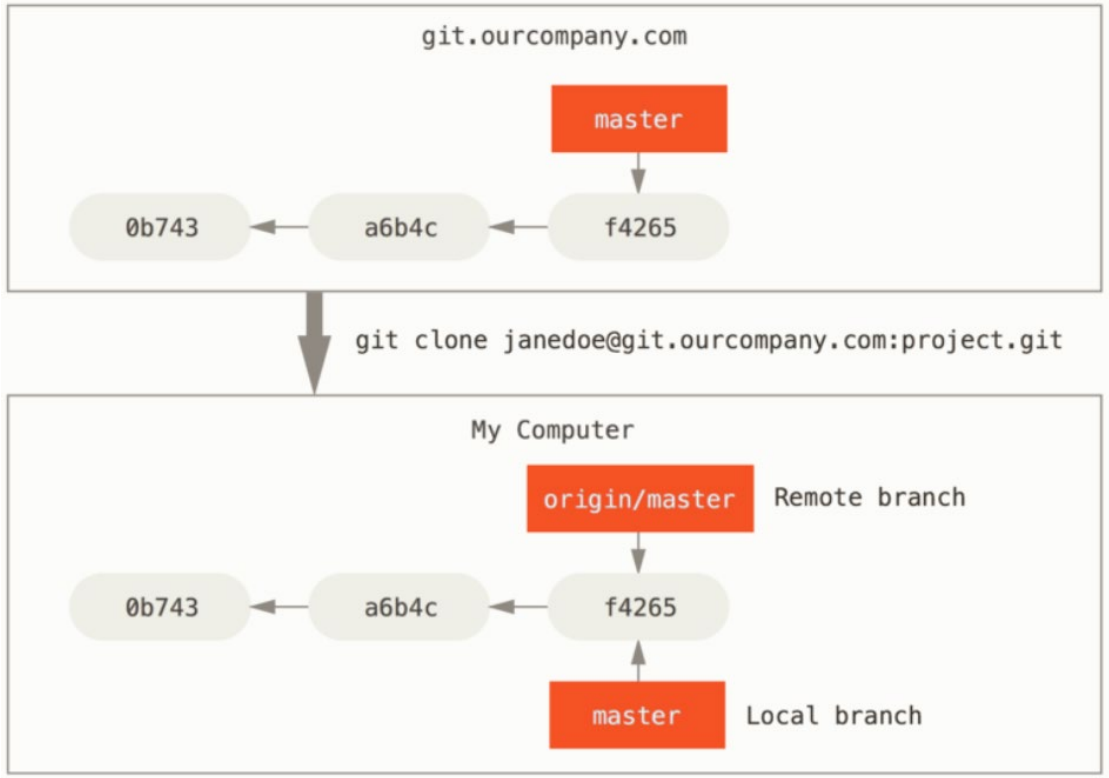
Gałęzie tematyczne - przydatne w projektach dowolnej wielkości. Gałąź tematyczna to krótkotrwała gałąź, którą tworzysz i używasz do pojedynczej konkretnej funkcji lub powiązanej pracy. W Git często tworzy się, przetwarza, łączy i usuwa gałęzie kilka razy dziennie.



Git – zdalne gałęzie

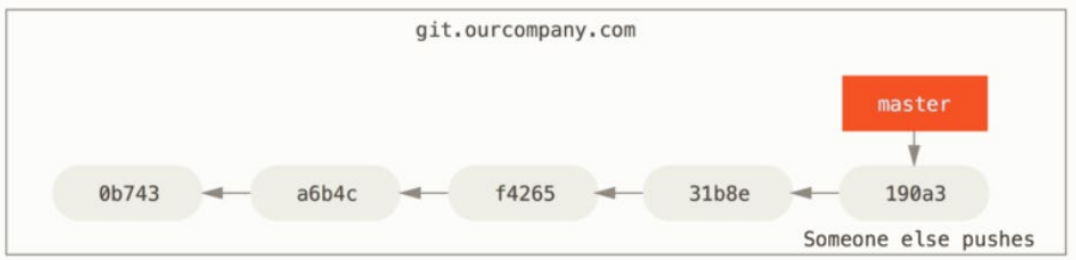
Zdalne gałęzie (**remote branches**) są odniesieniami (wskaźnikami) do stanu gałęzi w zdalnych repozytoriach. Są one automatycznie przenoszone za każdym razem, gdy wykonuje się jakąkolwiek komunikację sieciową. Zdalne gałęzie działają jako zakładki przypominające, gdzie były gałęzie w zdalnych repozytoriach podczas ostatniego połączenia z nimi.
Postać: **(remote)/(branch)**, np. origin/master, origin/iss53

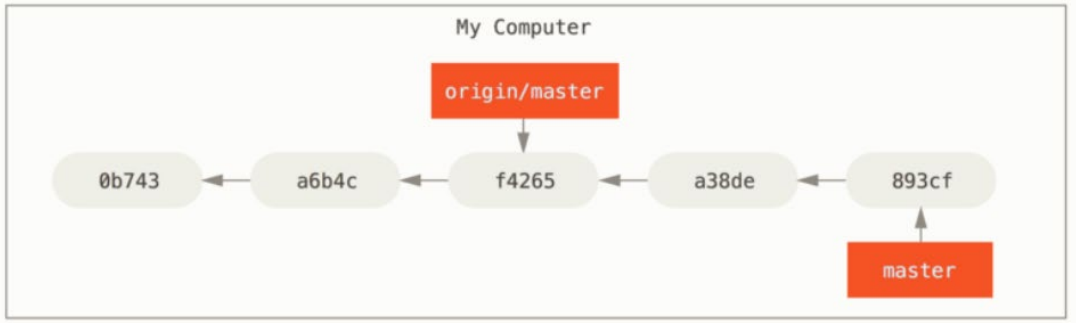
Powiedzmy, że masz serwer Git w swojej sieci na git.ourcompany.com. Jeśli sklonujesz z niego repozytorium, Git automatycznie nada mu nazwę pochodzenia, ściągnie wszystkie jego dane, utworzy wskaźnik do miejsca, w którym znajduje się jego gałąź główna, i lokalnie nazwie origin / master. Git ma także własną lokalną gałąź główną (master), która zaczyna się w tym samym miejscu, co główna gałąź pochodzenia (origin/master). Nazwa "origin" nie jest jakaś szczególna, po prostu domyślnie nadawana na określenie źródła po użyciu `git clone`



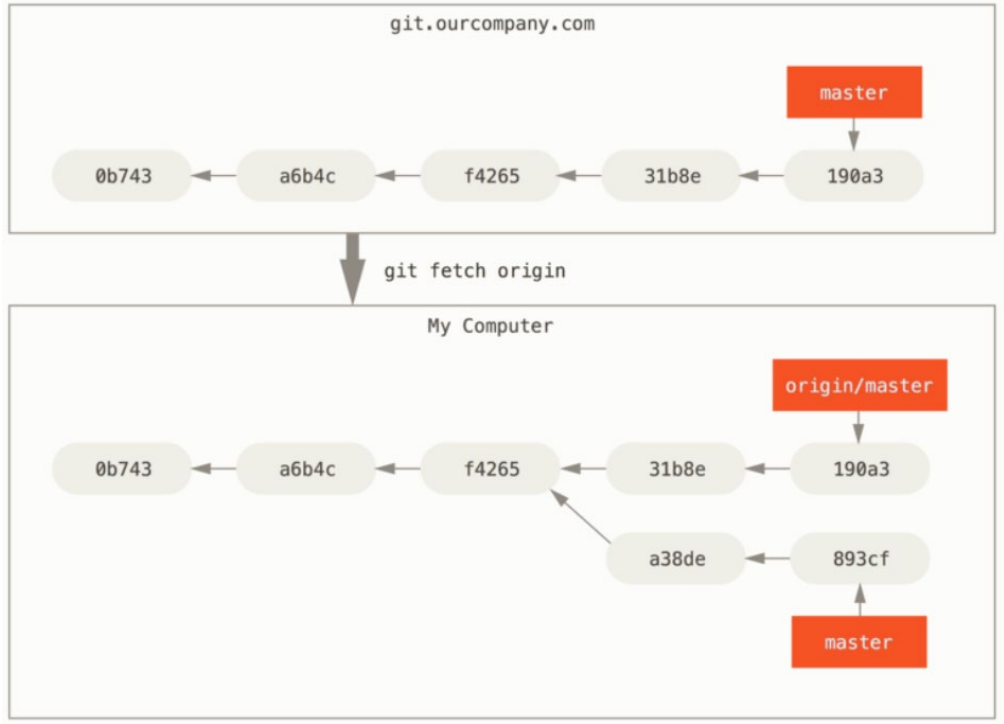
Git – zdalne gałęzie

Jeśli wykonujesz jakąś pracę na lokalnej gałęzi master, a w międzyczasie ktoś inny wysyła kod na git.ourcompany.com i aktualizuje gałąź master, to twoja historia (lokalna) zmienia się w inny sposób. Ponadto, dopóki nie łączysz się z serwerem ze zdalnym repozytorium, wskaźnik origin/master nie zmienia się.



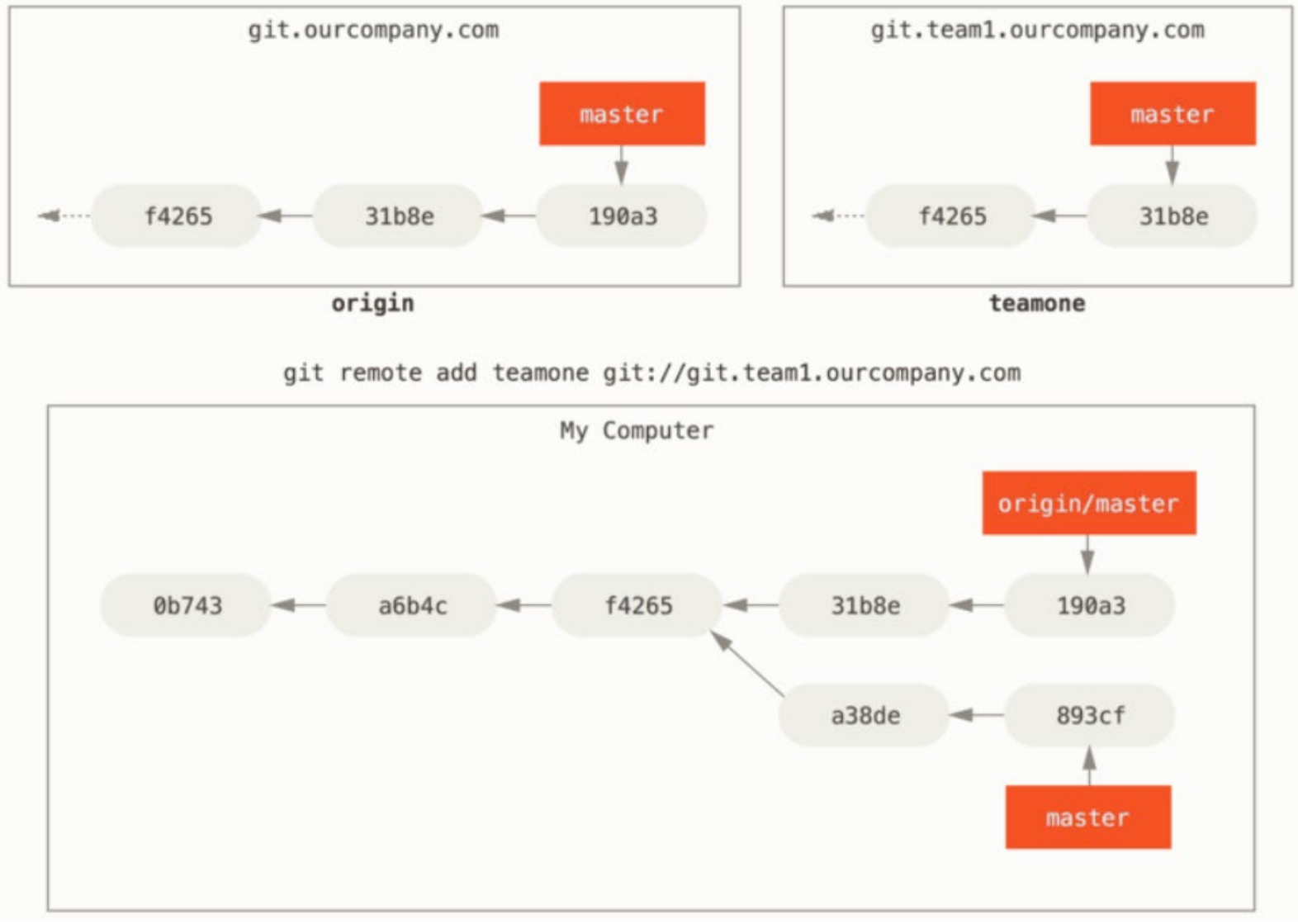


Synchronizacja pracy: `$ git fetch origin`
To polecenie sprawdza źródłowy serwer (w tym przypadku jest to git.ourcompany.com), pobiera z niego dane, których nie ma lokalnie, i aktualizuje lokalną bazę danych, przenosząc wskaźnik origin/master na jego nową, aktualną pozycję.



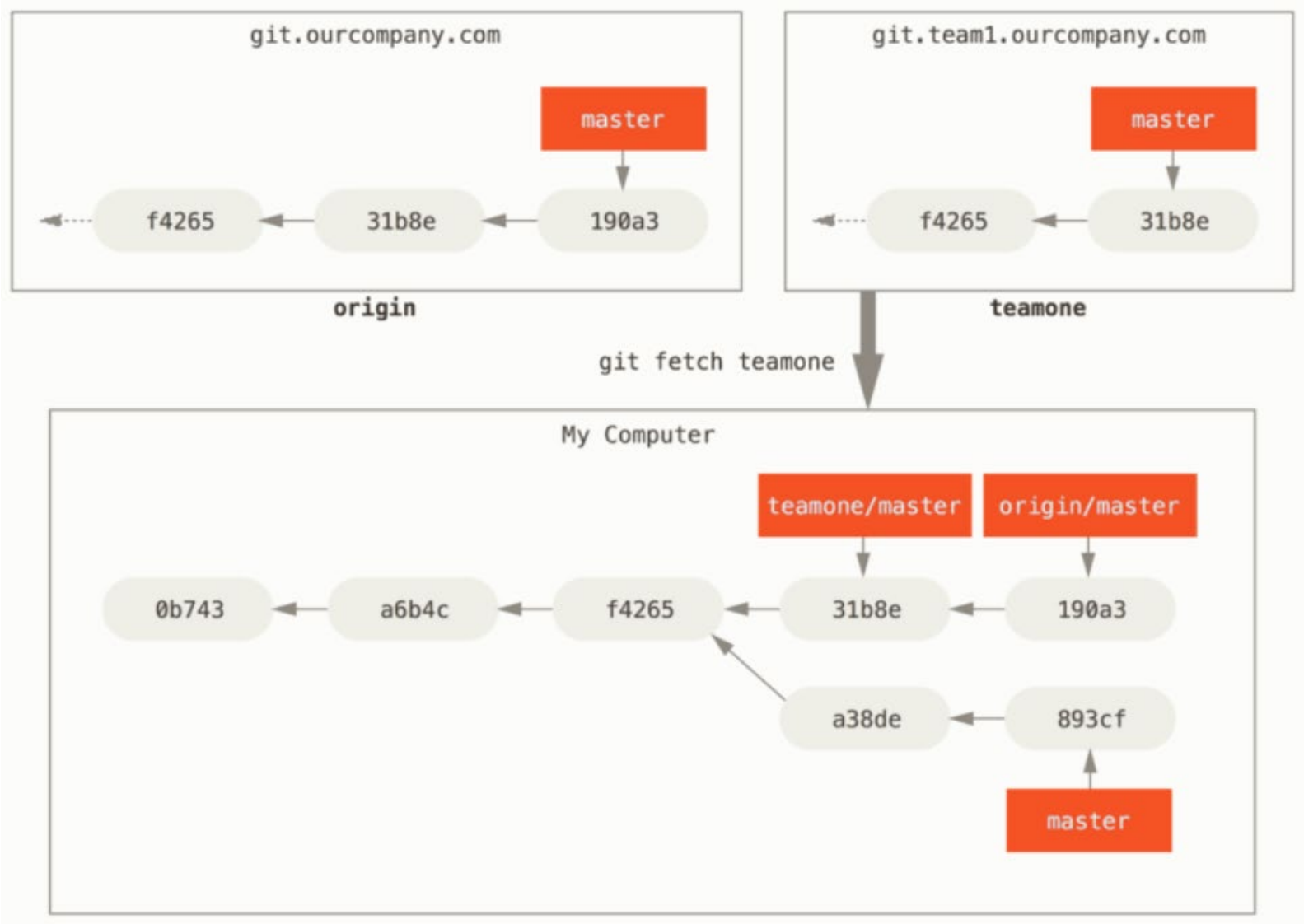
Git – gałęzie i więcej serwerów

Dany projekt (jego części) może być umieszczony na wielu zdalnych serwerach. Załóżmy, że jest inny wewnętrzny serwer Git, który jest używany tylko do rozwoju przez jeden z zespołów. Ten serwer znajduje się pod adresem git.team1.ourcompany.com. Możesz dodać go jako nowe zdalne odwołanie do projektu, nad którym obecnie pracujesz, uruchamiając polecenie git remote add. Nazwiemy ten zdalny serwer teamone (będzie to krótka nazwa dla całego adresu URL).



Git – gałęzie i więcej serwerów

Teraz można uruchomić:
$ `git fetch teamone`
aby pobrać zawartość zdalnego serwera `teamone`, której jeszcze nie ma lokalnie. Ponieważ ten serwer ma pewien wspólny podzbiór danych, które ma również serwer `origin`, Git nie pobiera danych, ale ustawia zdalną gałąź o nazwie `teamone/master`, aby wskazywał na repozytorium, które ma `teamone` jako gałąź główną.



Git – wypychanie (push) gałęzi

Udostępnienie gałęzi innym wymaga wypchnięcia jej (push) na zewnętrzny serwer, na którym masz prawa zapisu. Lokalne gałęzie nie są automatycznie synchronizowane ze zdalnymi, trzeba to zrobić bezpośrednio. Oczywiście, można mieć w lokalnym repozytorium prywatne gałęzie, a wysyłać tylko te, które chcemy współdzielić (udostępnić):

$ `git push (remote) (branch)`, np.

$ `git push origin serverfix`

to samo:

$ `git push origin serverfix:serverfix`

jeśli chcemy inną nazwę zdalnej gałęzi:

$ `git push origin serverfix:innanazwa`

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

Następnym razem, gdy ktoś pobierze repozytorium z serwera, otrzyma również wersję zdalnej gałęzi `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Git – wypychanie (push) gałęzi

Po pobraniu (`fetch`), które sprowadza nowe zdalne gałęzie, nie mamy automatycznie ich lokalnych, edytowalnych kopii. Np. nie masz nowej gałęzi serverfix - jest tylko wskaźnik `origin/serverfix`, którego nie można zmodyfikować.

Aby połączyć tę zdalną gałąź z lokalną gałęzią gotową do pracy, należy:

```
$ git merge origin/serverfix
```

Jeśli chcesz mieć własną gałąź serverfix, na której możesz pracować, możesz oprzeć ją na zdalnej gałęzi:

```
$ git checkout -b serverfix origin/serverfix
```