

Systemy kontroli wersji



Czyli wprowadzenie do Gita i serwisów na nim bazujących

Git vs Mercurial

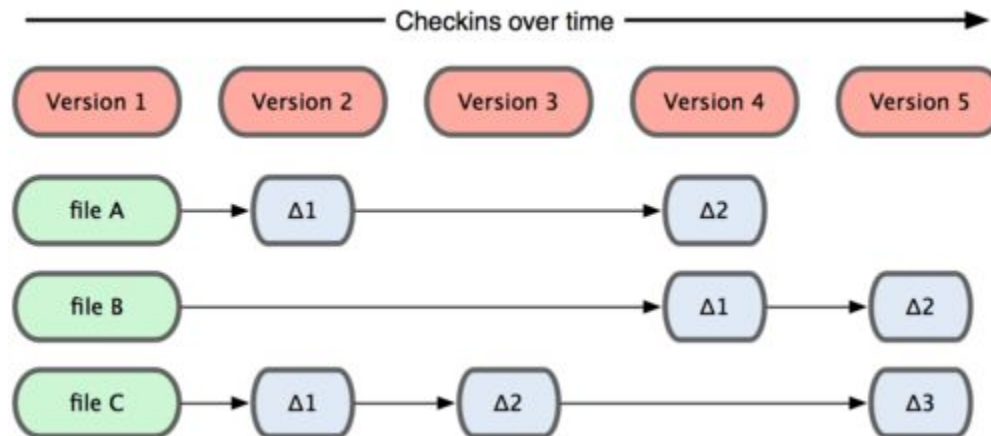
- 👍 rewrite history / modyfikowanie commitów
- 👍 szybszy przy operacjach sieciowych
- 👍 lepszy mechanizm branchy
- 👍 rozszerzenia mogą być pisane w różnych językach
- 👍 indeksy
- 👍 możliwość konwertowania repozytorium na Mercurialowe i z powrotem bez utraty danych
- 👍 dobry do dużych projektów
- 👍 łatwiejszy do opanowania
- 👍 bookmarks dzielą jedną przestrzeń nazw
- 👍 lepiej sobie radzi z merge'ami
- 👍 lepsza użyteczność GUI
- 👍 hg help bardziej pomocne niż git help
- 👍 branche wizualnie lepsze do zrozumienia
- 👍 revsets do naśladowania indeksów w razie potrzeby
- 👍 wbudowany webserver

Git vs Mercurial

- 🔊 nadmiernie skomplikowany
 - 🔊 wolniejszy na Windows
 - 🔊 długie polecenia
 - 🔊 niezbyt “user-friendly”
 - 🔊 rozszerzenia mają ograniczenia
 - 🔊 indeksowanie powoduje dodatkowy krok
 - 🔊 rebase może mocno namieszać
- 🔊 rozszerzenia tylko w pythonie
 - 🔊 łączenie funkcjonalności może być problematyczne przy użyciu różnych rozszerzeń
 - 🔊 nie można przekonwertować na repozytorium Git i z powrotem bez utraty danych
 - 🔊 rollback powoduje tylko cofnięcie ostatniego commita, potrzebne rozszerzenia by osiągnąć to co w Git jest dostępne “prosto z pudełka”
 - 🔊 brak “partial checkout”

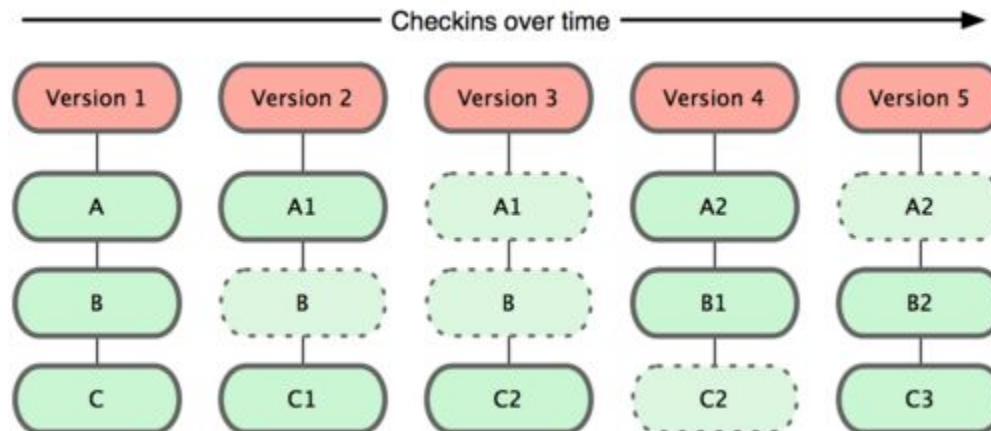
Jak działa Git?

Podstawową różnicą pomiędzy Gitem a każdym innym systemem kontroli wersji jest podejście Gita do przechowywanych danych. Większość pozostałych systemów przechowuje informacje jako listę zmian na plikach. Systemy te traktują przechowywane informacje jako zbiór plików i zmian dokonanych na każdym z nich w okresie czasu.



Jak działa Git?

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



Zaczynamy

Stworzenie pustego repozytorium - git init

```
~/Dev/tutorial ➤ git init  
Initialized empty Git repository in /Users/jakubdanielczyk/Dev/tutorial/.git/  
~/Dev/tutorial ➤ ↻ master |
```

Stworzyliśmy puste repozytorium i zostaliśmy przełączeni na główną gałąź naszego projektu, czyli master. Od teraz możemy już dokonywać zmian a git będzie to wszystko za nas śledził.

Status

Podejrzenie zmian od ostatniego commita - git status

```
~/Dev/tutorial ➤ master ➤ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
~/Dev/tutorial ➤ master ➤ |
```

No commits yet, ponieważ nic jeszcze nie zacommitowaliśmy.

Nothing to commit - znak, że nic się w repozytorium nie zmieniło od ostatniego commita, lub od stworzenia repozytorium w przypadku gdy nie ma jeszcze żadnych commitów

Git - dodawanie plików

Zmiany w repozytorium: dodanie/usunięcie plików, zmiany w plikach

```
~/Dev/tutorial ➤ master ➤ touch hello.txt
~/Dev/tutorial ➤ master ➤ git status
On branch master

No commits yet

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

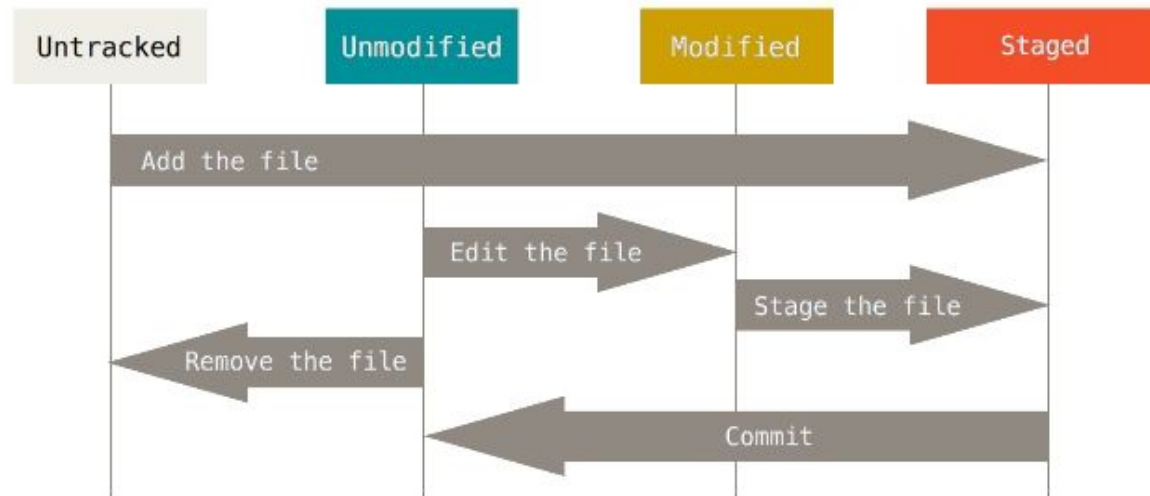
        hello.txt

nothing added to commit but untracked files present (use "git add" to track)
~/Dev/tutorial ➤ master ➤ |
```

Do repozytorium dodaliśmy nowy, pusty plik hello.txt. Po wywołaniu polecenia git status zostajemy poinformowani o tym, że jest jeden, nieśledzony plik (untracked).

Git - cykl życia pliku

Plik w gicie można znajdować się w czterech stanach.



Untracked - gdy dodajemy nowy plik do repozytorium jest on w tym stanie.

Unmodified - od ostatniego commita nie dokonano żadnych zmian w tym pliku.

Modified - dokonane zostały zmiany w tym pliku, które powinny być zacommitowane.

Staged - plik został dodany do “puli” plików, które zostaną zacommitowane z następnym użyciem komendy `git commit`.

Git add

Git add przenosi plik do stanu 'staged'. Innymi słowy, wrzuca plik do "puli" plików gotowych na commit (poczekalni).

git add <nazwaPliku>

```
~/Dev/tutorial ➤ master ➤ git add hello.txt
~/Dev/tutorial ➤ master + ➤ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   hello.txt

~/Dev/tutorial ➤ master + ➤ |
```

jeśli chcemy dodać wszystkie pliki na raz możemy użyć **git add -A** lub **git add .**

Git commit

Git commit tworzy commita, czyli zapisuje wszystkie zmiany wrzucone wcześniej do poczekalni przez **git add**. Po stworzeniu commita **git status** poinformuje nas, że nie ma żadnych zmian od ostatniego commita, więc możemy kontynuować pracę.

git commit -m “wiadomośćCommita”

```
x ~/Dev/tutorial ➤ ↻ master + ➤ git commit -m "pierwszy commit"
[master (root-commit) 886d9ca] pierwszy commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 hello.txt
~/Dev/tutorial ➤ ↻ master ➤ git status
On branch master
nothing to commit, working tree clean
~/Dev/tutorial ➤ ↻ master ➤ |
```

możemy również wykorzystać po prostu **git commit** wtedy do wpisania wiadomości otworzy nam się tekstowy edytor vim (lub emacs, w zależności od konfiguracji). Jest on jednak bardzo nieprzyjemny w obsłudze, więc nie polecam.

Co się stało?

Git log to po prostu historia commitów

git log

```
commit 886d9ca115a6ef03ed3be231675f5fe7c2f90b4b (HEAD -> master)
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Tue Feb 19 20:07:57 2019 +0100

    pierwszy commit
(END)
```

aby wyjść z tego widoku należy nacisnąć 'q'.

Mamy tu dostęp do takich informacji jak: autor, data utworzenia, wiadomość oraz commit hash (unikalny identyfikator commita).

Zmiany, zmiany, zmiany

Git diff pozwala nam na podejrzenie zmian dokonanych od ostatniego commita, lub pomiędzy dwoma commitami

git diff <hashPierwszegoCommita> <hashDrugiegoCommita>

```
~/Dev/tutorial master echo "hello world" >> hello.txt
~/Dev/tutorial master ● git diff

diff --git a/hello.txt b/hello.txt
index e69de29..3b18e51 100644
--- a/hello.txt
+++ b/hello.txt
@@ -0,0 +1 @@
+hello world
(END)
```

aby wyjść z tego widoku należy nacisnąć 'q'.

użycie samego **git diff** pokazuje zmiany dokonane od ostatniego commita.

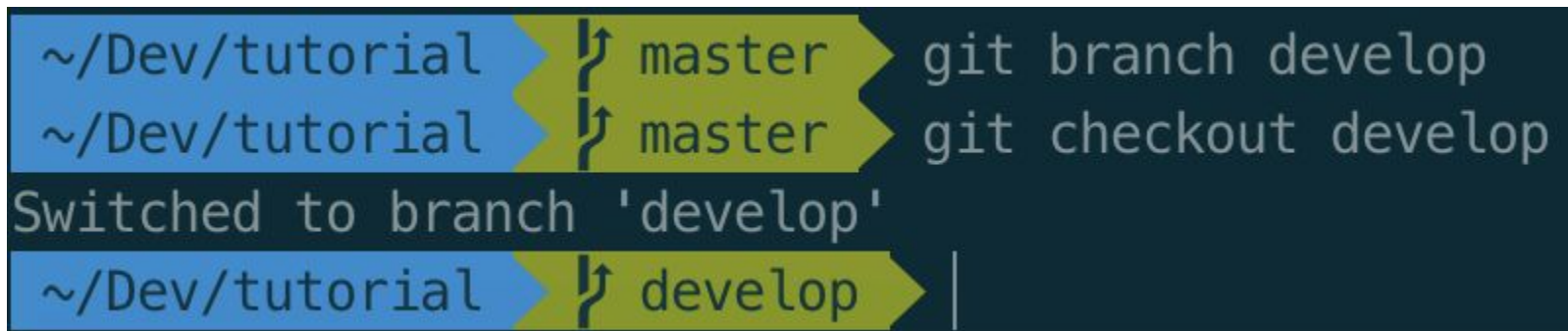
Git branch, git checkout

Git branch pozwala nam na stworzenie nowej gałęzi.

git branch <nazwaGałęzi>

Do przełączenia na wybraną gałąź służy git checkout

git checkout <nazwaGałęzi>



The image is a screenshot of a terminal window with a dark background. It shows the execution of two Git commands. The first command, 'git branch develop', is shown with its output: a new branch 'develop' is created from 'master'. The second command, 'git checkout develop', is shown with its output: 'Switched to branch 'develop''. The terminal text is as follows:

```
~/Dev/tutorial ➤ master ➤ git branch develop  
~/Dev/tutorial ➤ master ➤ git checkout develop  
Switched to branch 'develop'  
~/Dev/tutorial ➤ develop ➤ |
```

Do wyświetlenia listy gałęzi służy **git branch**

Trochę więcej o commitach

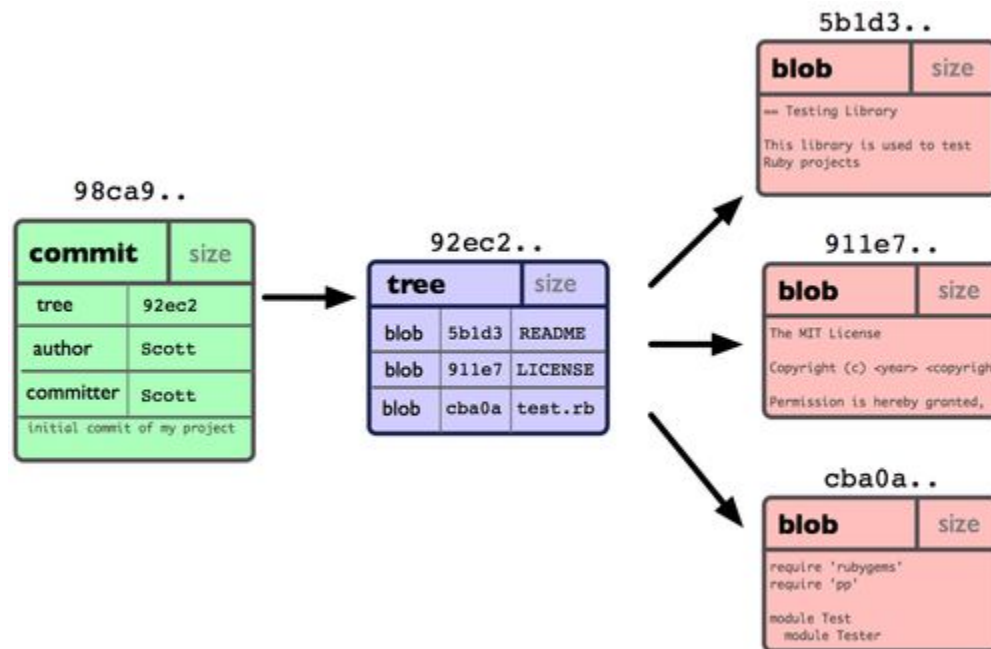
Kiedy zatwierdzamy zmiany w Gicie, ten zapisuje obiekt zmian (commit), który z kolei zawiera wskaźnik na migawkę zawartości, która w danej chwili znajduje się w poczekalni (czyli plików dodanych przez **git add**); metadane autora i opisu oraz zero lub więcej wskaźników na zmiany, które były bezpośrednimi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

Aby lepiej to zobrazować, załóżmy, że posiadamy katalog zawierający trzy pliki, które umieszczamy w poczekalni (**git add**). Umieszczenie w poczekalni plików powoduje wyliczenie sumy kontrolnej każdego z nich (skrótów SHA-1), zapisanie wersji plików w repozytorium (Git nazywa je blobami) i dodanie sumy kontrolnej do poczekalni.

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

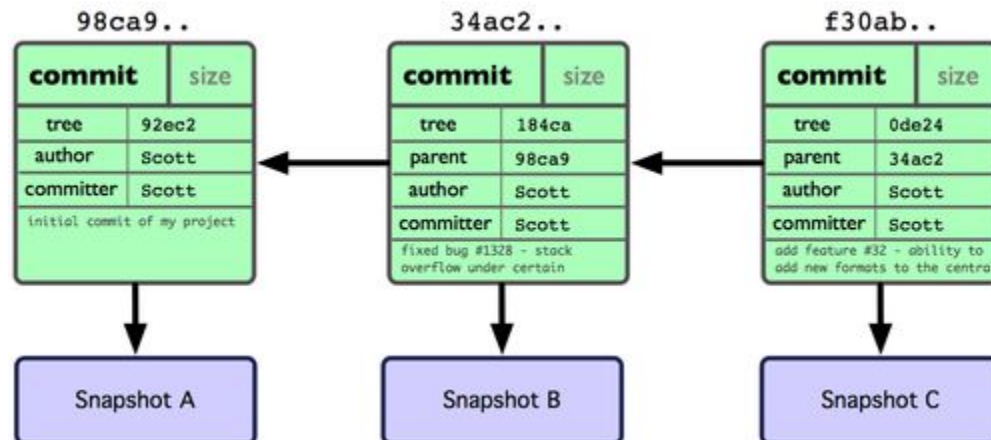
Teraz repozytorium Gita zawiera już 5 obiektów: jeden blob dla zawartości każdego z trzech plików, jedno drzewo opisujące zawartość katalogu i określające, które pliki przechowywane są w których blobach, oraz jeden zestaw zmian ze wskaźnikiem na owo drzewo i wszystkimi metadanymi.

Trochę więcej o commitach



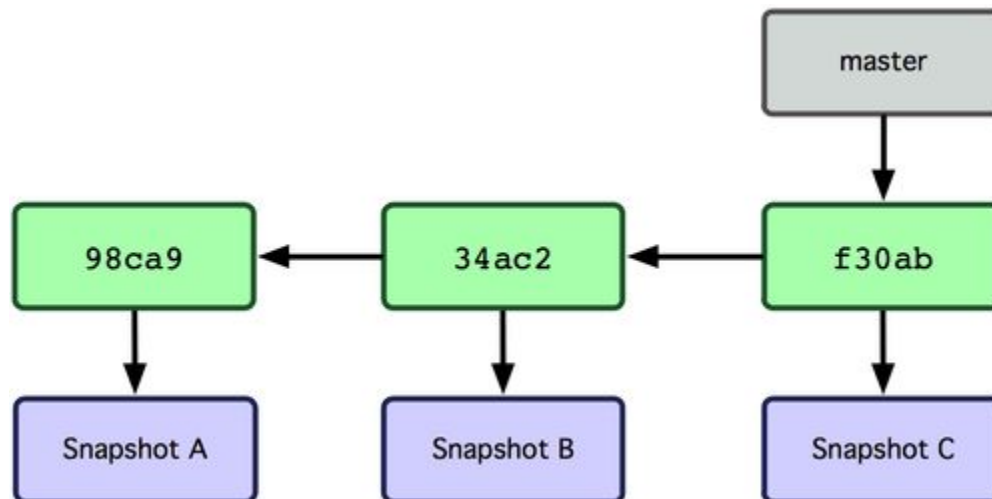
Trochę więcej o commitach

Jeśli dokonamy zmian i je również zacommitujemy, kolejny commit zachowa wskaźnik do poprzedniego.



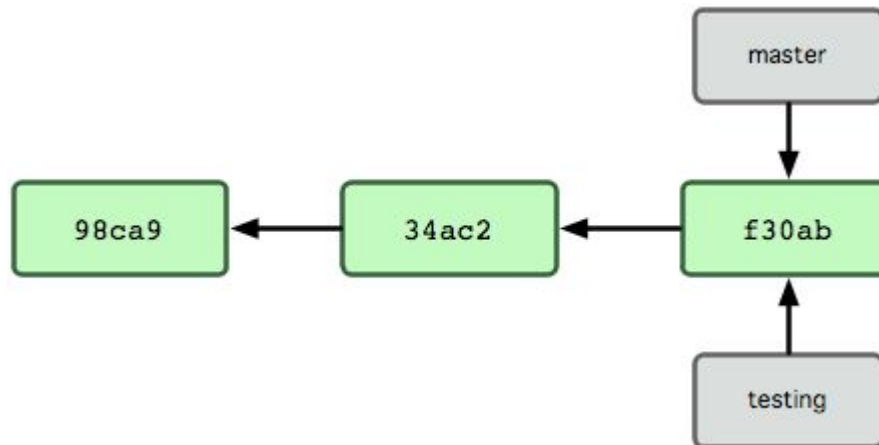
Czym jest gałąź?

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



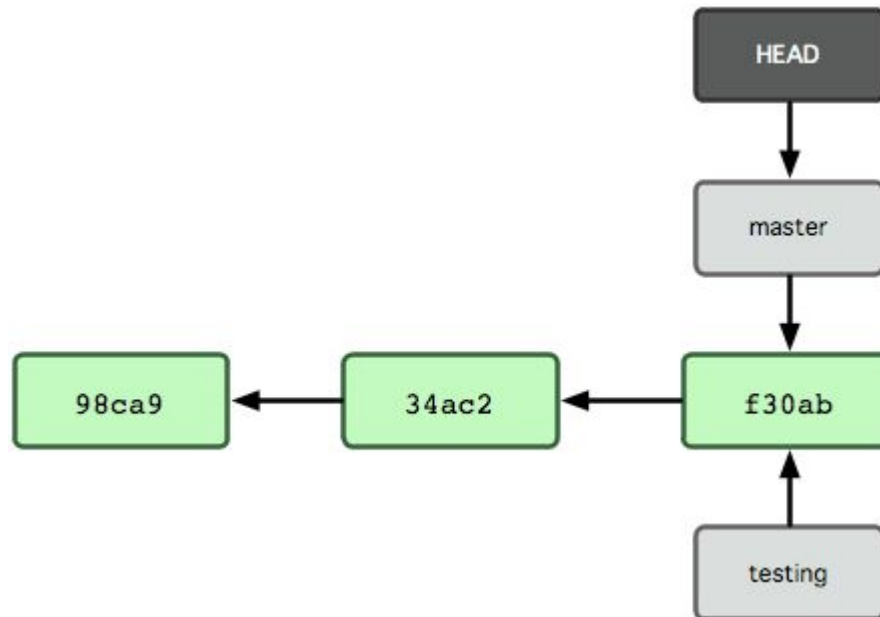
Czym jest gałąź?

Co się stanie, jeśli utworzymy nową gałąź? Utworzony zostanie nowy wskaźnik, który następnie będziemy mogli przesuwać. Powiedzmy, że tworzymy nową gałąź o nazwie testing (**git branch testing**). Utworzony wskaźnik (gałąź) będzie wskazywał na ten sam zestaw zmian, w którym aktualnie się znajdujemy.



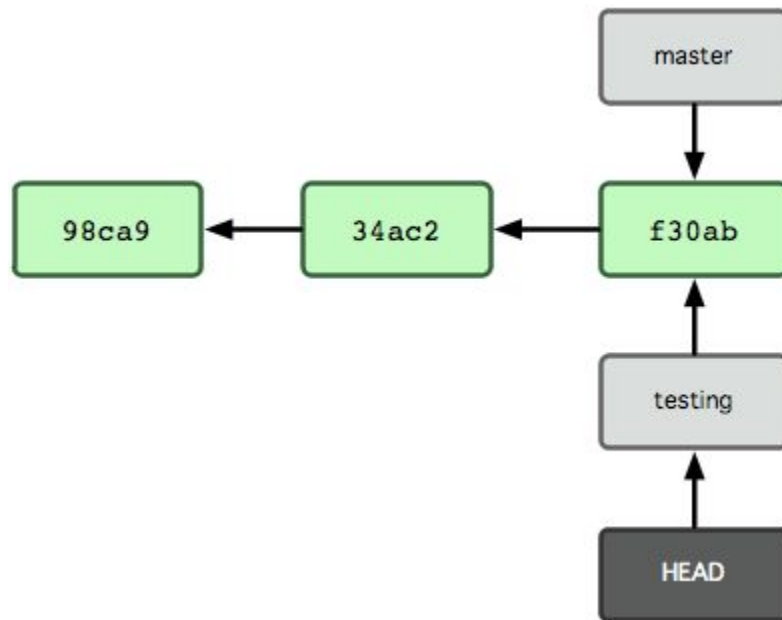
Czym jest gałąź?

Skąd Git wie, na której gałęzi się aktualnie znajdujemy? Utrzymuje on specjalny wskaźnik o nazwie HEAD. W Gicie jest to wskaźnik na lokalną gałąź, na której właśnie się znajdujemy. W tym wypadku, wciąż jesteś na gałęzi master. Polecenie git branch utworzyło jedynie nową gałąź, ale nie przełączyło nas na nią.



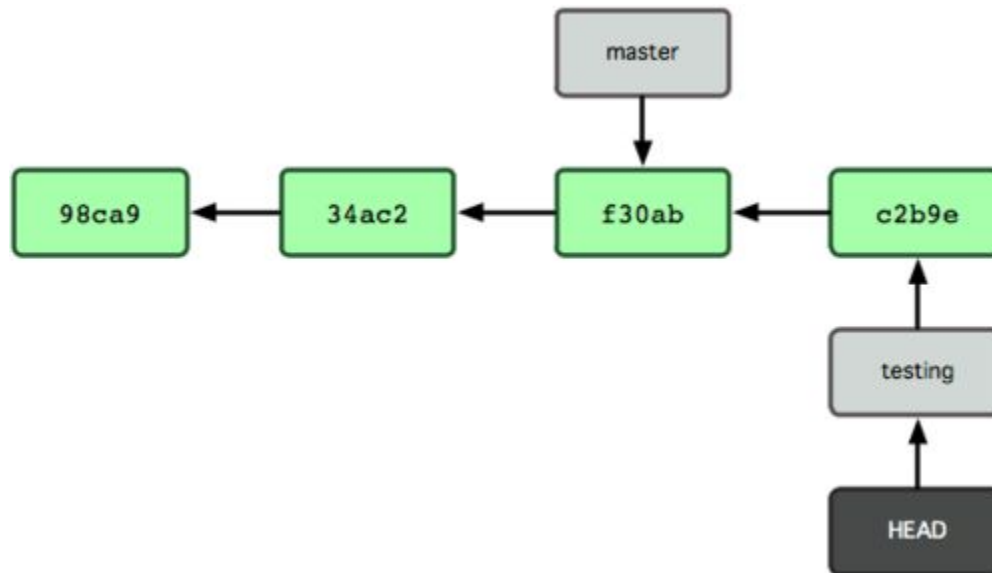
Czym jest gałąź?

Aby przełączyć się na inną gałąź używamy polecenia **git checkout**. A więc chcąc przełączyć się na gałąź testing, użyjemy **git checkout testing**. HEAD zostaje zmieniony tak, aby wskazywać na gałąź testing.



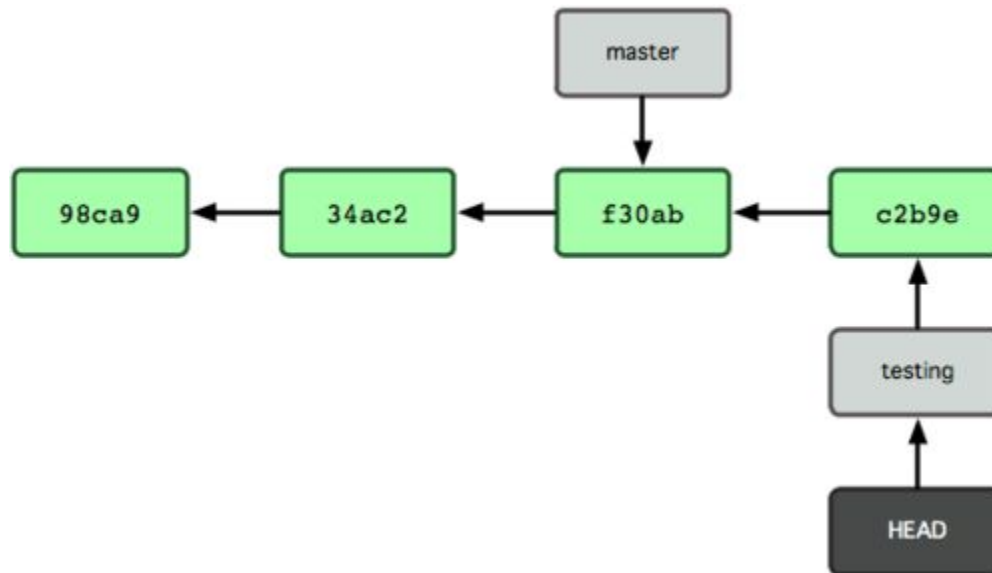
Czym jest gałąź?

Jakie ma to znaczenie? Oto co się stanie, gdy dodamy jakieś zmiany i je zacommitujemy.



Czym jest gałąź?

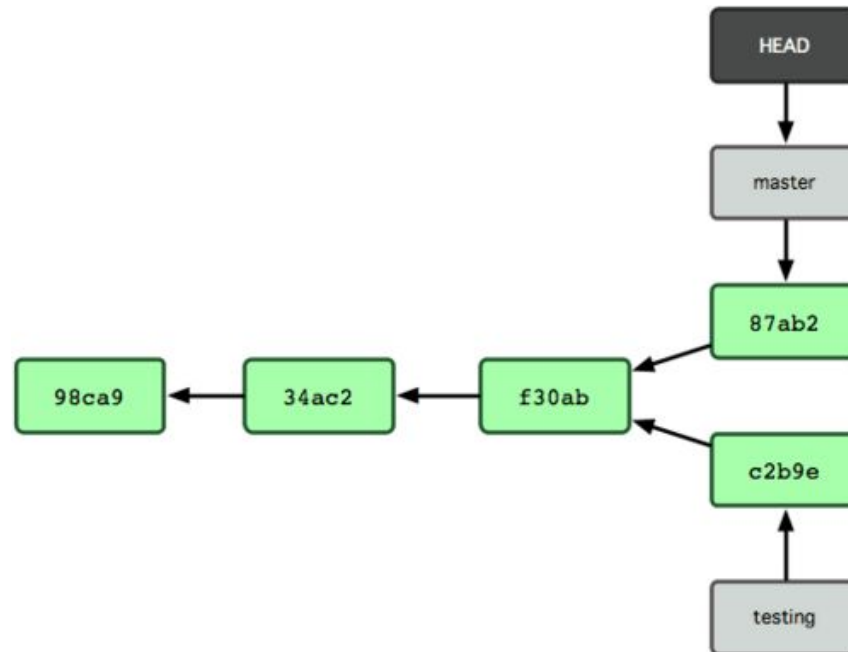
Teraz gałąź testing przesunęła się do przodu, jednak gałąź master ciągle wskazuje ten sam zestaw zmian, co w momencie użycia `git checkout` do zmiany aktywnej gałęzi. Przełączmy się zatem z powrotem na gałąź master. (**`git checkout master`**)



Czym jest gałąź?

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

Co się stanie po wykonaniu kilku zmian i ich zatwierdzeniu (commit)? Historia naszego projektu zostanie rozszczepiona. Oba zestawy zmian są od siebie odizolowane w odrębnych gałęziach: możemy przełączać się pomiędzy nimi oraz scalić je razem, kiedy będziemy na to gotowi.



Po co te gałęzie?

Założmy, że mamy aplikację pogody, która na ekranie wyświetla nazwę miasta i temperaturę w tym mieście. Decydujemy się na dodanie nowej, złożonej funkcjonalności. Na przykład widoku mapy, po której moglibyśmy się poruszać i zobaczyć temperaturę w różnych miejscach.

Zaczynamy więc pracę na gałęzi master. Stworzyliśmy już kilka commitów, po czym okazuje się, że w aplikacji został wykryty krytyczny błąd, który potrzebuję natychmiastowej naprawy. W tej sytuacji nie pozostaje nam nic innego jak oderwać się od tworzenia nowej funkcjonalności i naprawić błąd. Z racji tego, że robimy to na jednej gałęzi (master), historia naszych commitów będzie wyglądała mniej więcej tak:

commitFunkcjonalności -> commitFunkcjonalności -> commitNaprawyBłędu -> commitFunkcjonalności

Pojawia się teraz jeden poważny problem: Jeśli chcemy udostępnić nową wersję aplikacji z naprawionym błędem, będzie ona zawierała niedokończoną nową funkcjonalność. Chcemy raczej unikać takich sytuacji, więc z pomocą przychodzą gałęzie.

Wystarczy wycofać się o kilka commitów, do momentu gdzie nie zaczęliśmy jeszcze pracy nad nową funkcjonalnością, stworzyć nową gałąź i tam pracować nad naprawą błędu. W ten sposób otrzymujemy obecną wersję udostępnionej aplikacji tylko z naprawionym błędem. Żadnych niedokończonych rzeczy.

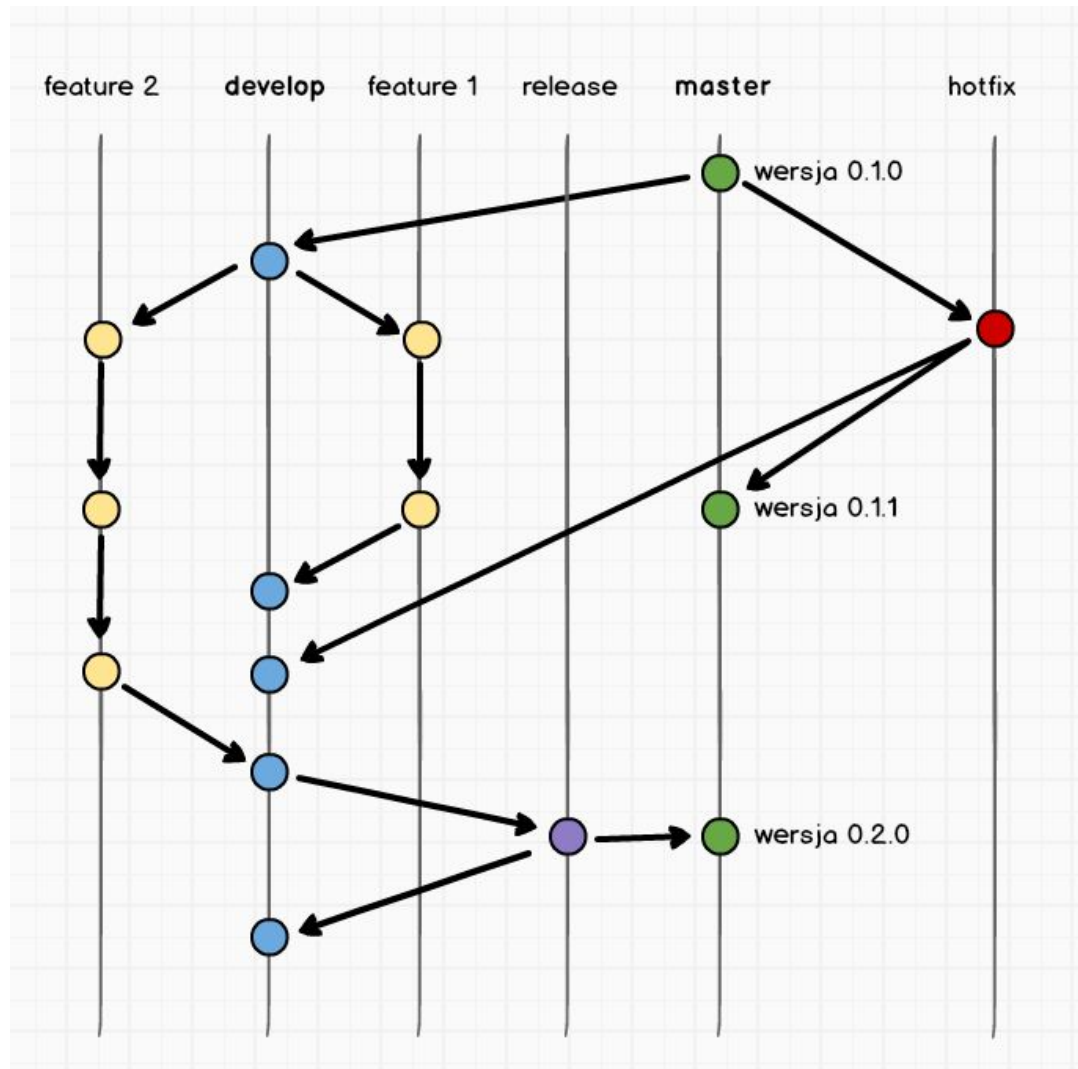
To oczywiście tylko jeden z powodów, dla których powinniśmy używać gałęzi. Jest ich o wiele więcej.

GitFlow

GitFlow to model rozgałęziania projektu stworzony przez Vincenta Driessena. Składa się on z pięciu głównych typów gałęzi:

1. Master - główna gałąź projektu, do której będą scalane inne gałęzie i tylko w jednym przypadku: zamiarze udostępnienia nowej wersji projektu.
2. Develop - gałąź developerska wychodząca z master, do której scalane są kolejne nowe funkcjonalności. Scalana do Release.
3. Feature - gałąź wychodząca z Developa, na której powinna być tworzona nowa funkcjonalność. Po skończeniu pracy powinna być scalona z Developem i usunięta.
4. Release - gałąź wychodząca z Mastera, do której będzie scalany Develop w zamiarze udostępnienia nowej wersji. W tym miejscu można dokonać zmian koniecznych do udostępnienia nowej wersji (np. inkrementacja wersji aplikacji). Później scalana z Masterem.
5. Hotfix - gałąź wychodząca z Mastera, służy tylko do naprawy krytycznych błędów. Po naprawie jest scalana bezpośrednio z Masterem, aby jak najszybciej udostępnić nową wersję z poprawką.

Git Flow



Git Flow

A po co?

- Zrównoleglenie pracy - bardzo łatwo można pracować jednocześnie nad kilkoma różnymi nowymi funkcjonalnościami oraz poprawkami.
- Współpraca - każdy programista może pracować jednocześnie nad inną funkcjonalnością i wzajemnie sobie nie przeszkadzają
- Wsparcie dla awaryjnych poprawek - w łatwy sposób można naprawić krytyczny błąd nie przeszkadzając innym w pracy nad resztą funkcjonalności
- Łatwe przełączanie między nowymi funkcjonalnościami, a starą wersją - Z racji faktu, że na Developie zawsze są najnowsze skończone funkcjonalności, a na Masterze obecna udostępniona wersja aplikacji, możemy szybko przełączać się między tymi gałęziami aby zobaczyć jak na przykład aplikacja zmieniła się od ostatniej wersji.

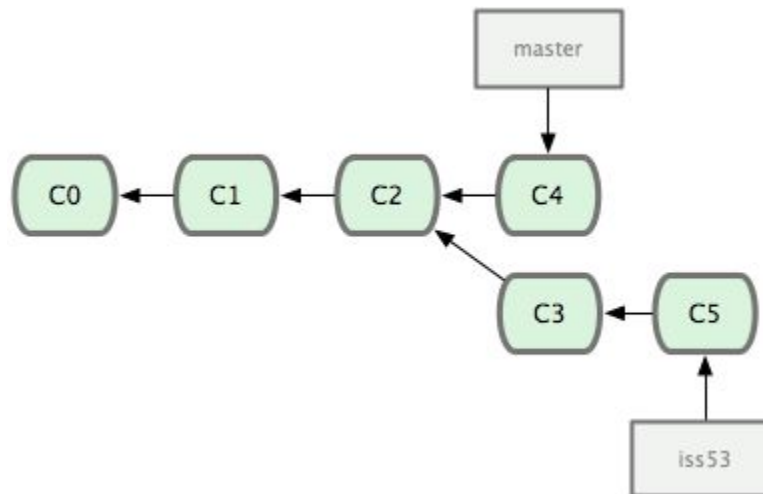
Scalanie gałęzi

Do połączenia gałęzi w jedną służy polecenie `git merge`

`git merge -m <nazwaGałęzi>`

Polecenie działa w ten sposób, że do gałęzi na której obecnie się znajdujemy dołączy zmiany z gałęzi, której nazwę umieściliśmy po poleceniu **`git merge`**. Podobnie jak w przypadku `git commit`, możemy pominąć przełącznik `-m`, ale wtedy otworzy nam się `vim`.

Przykład: Posiadamy repozytorium o następującej strukturze.



Scalanie gałęzi

Zakładamy, że skończyliśmy już pracować na gałęzi iss53 i chcemy ją scalić z powrotem z gałęzią master. Aby tego dokonać, musimy przełączyć się na gałąź master i wykonać merge:

```
~/Dev/nwta ➤ ⤵ iss53 ➤ git checkout master  
Switched to branch 'master'  
Your branch is up to date with 'origin/master'.  
~/Dev/nwta ➤ ⤵ master ➤ git merge iss53  
Merge made by the 'recursive' strategy.  
hello3.txt | 0  
hello4.txt | 0  
2 files changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 hello3.txt  
create mode 100644 hello4.txt  
~/Dev/nwta ➤ ⤵ master ➤ |
```

Scalanie gałęzi

Po użyciu polecenia **git log** (historia commitów) zobaczymy coś takiego:

```
commit ce79db7b3947612fa79b32b13320265bb08256ab (HEAD -> master)
Merge: c4107f8 d01d3bc
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 12:11:34 2019 +0100

    Merge branch 'iss53'

commit d01d3bc9b98c84fafa44678271692e39b48337d4 (iss53)
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 12:10:49 2019 +0100

    Trzeci commit na iss53

commit e832ece0c266e3cd69251d530180b7be98531b88
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 12:10:37 2019 +0100

    Drugi commit na iss53

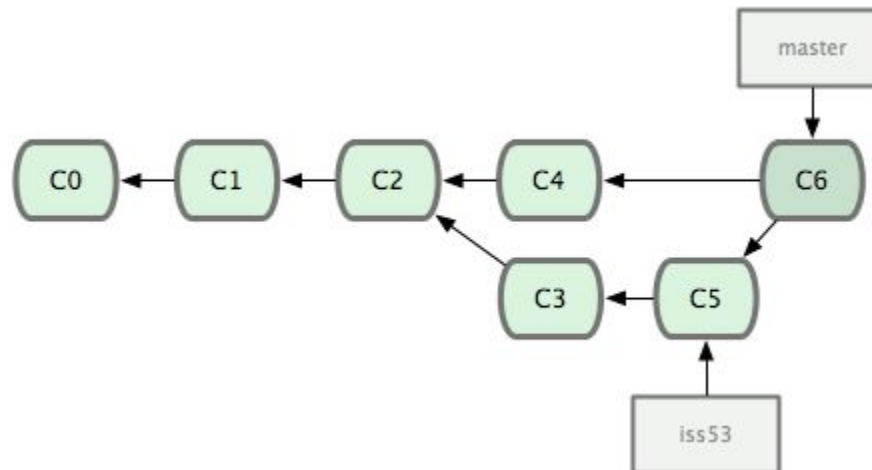
commit c4107f8f515f225c90fd4ca47bc96fa4d8417b0a (origin/master, origin/HEAD)
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 12:10:17 2019 +0100

    Drugi commit na masterze
```

Jak widać, git scalał dwie gałęzie poprzez przeniesienie commitów z iss52 na gałąź master, oraz stworzenie nowego commita scalającego (merge commit).

Scalanie gałęzi

A oto jak wygląda struktura naszego repozytorium po takim scaleniu:



Gałąź iss53 nie jest nam już dłużej potrzebna, a więc można ją usunąć (pamiętajmy, że to tylko wskaźnik) używając polecenia git branch z przełącznikiem -d

git branch -d <nazwaGałęzi>

Konflikty

Zdarzają się sytuacje, gdy scalanie gałęzi nie przebiega tak gładko. Jeśli ten sam plik został zmieniony w różny sposób na dwóch gałęziach, Git nie będzie w stanie scalić ich samodzielnie i zwróci komunikat o konflikcie.

Stworzyłem nową gałąź (test1), w której dodałem plik text.txt i wpisałem do niego tekst “witaj świecie”. Następnie przepiąłem się na gałąź master i dodałem ten sam plik, lecz do niego wpisałem tekst “hello world”. O to co się dzieje próbując scalić dwie gałęzie.

```
~/Dev/tutorial ➤ ↶ master ➤ git merge test1
Auto-merging text.txt
CONFLICT (add/add): Merge conflict in text.txt
Automatic merge failed; fix conflicts and then commit the result.
✖ ➤ ~/Dev/tutorial ➤ ↶ master ●+ >M< ➤ |
```

Konflikty

Otrzymaliśmy komunikat o konflikcie. Taki konflikt musimy naprawić ręcznie. A więc otwieramy ten plik (nie ma znaczenia, czy otworzymy go w konsoli, czy na przykład notatnikiem).

```
<<<<<< HEAD  
hello world  
=====  
witaj swiecie  
>>>>>> test1
```

Zawartość pliku nieco się zmieniła. O co chodzi?

Wszystko to, co znajduje się pomiędzy HEAD a znakami równości, to zawartość tego pliku w gałęzi, w której obecnie się znajdujemy, czyli w naszym wypadku master. Wszystko to, co od znaków równości do znaków większości, to zawartość pliku w gałęzi test1. Teraz od nas zależy decyzja, co chcemy zostawić, a co usunąć. Możemy pozostawić zawartość z obu gałęzi. To, co musimy jednak usunąć, to linia pierwsza, trzecia i piąta, czyli to co wygenerował za nas git.

```
hello world  
witaj swiecie
```

Konflikty

Po usunięciu konfliktu (bo to właśnie zrobiliśmy), należy dokonać standardowej procedury dodawania i zatwierdzania zmian, czyli:

```
x ~/Dev/tutorial ↗ master ●+ >M< ➤ gst
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

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

      both added:      text.txt

no changes added to commit (use "git add" and/or "git commit -a")
~/Dev/tutorial ↗ master ●+ >M< ➤ git add .
~/Dev/tutorial ↗ master + >M< ➤ git commit -m "Rozwiazywanie konfliktow"
[master dc7b7d9] Rozwiazywanie konfliktow
~/Dev/tutorial ↗ master ➤ |
```

Cherry-pick, czyli wisienka na torcie

Jeśli mamy kilka gałęzi i chcemy przenieść zmiany dokonane w commicie z jednej gałęzi na drugiej, używamy cherry pick

```
git cherry-pick <commitHash>
```

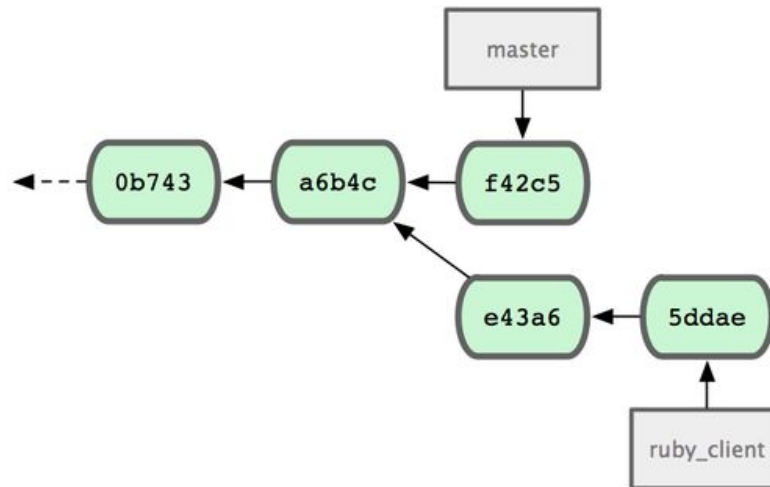
tego polecenia używamy na gałęzi, do której chcemy dołączyć zmiany z commita podanego w parametrze.

Operacja ta tworzy na gałęzi, na którą wskazuje HEAD nowy commit, który zawiera wszystkie zmiany z commita, którego hash znajduje się w parametrze (włącznie z wiadomością commita).

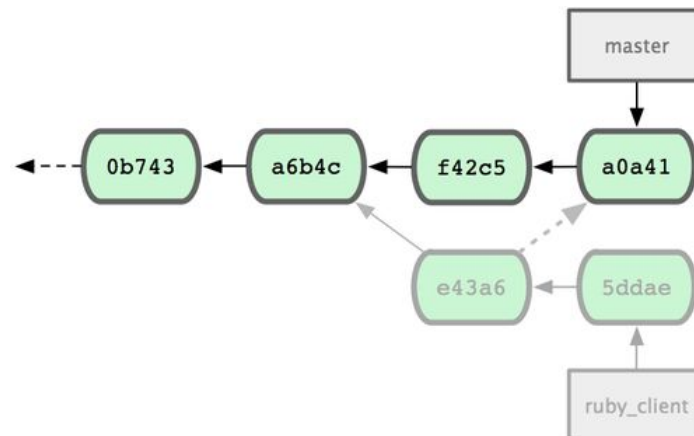
Operacja ta najczęściej jest używana w przypadku, gdy na przykład na gałęzi Develop (odsyłam do GitFlow) znajduje się jedna gotowa funkcjonalność, którą pilnie chcemy udostępnić w nowej wersji, a pozostałe w kolejnej. Wtedy możemy z gałęzi Release użyć cherry-picka na commita z konkretną funkcjonalnością, a w przyszłości - gdy reszta funkcjonalności będzie skończona - **git merge**.

Cherry-pick, czyli wisienka na torcie

Mamy taką strukturę:

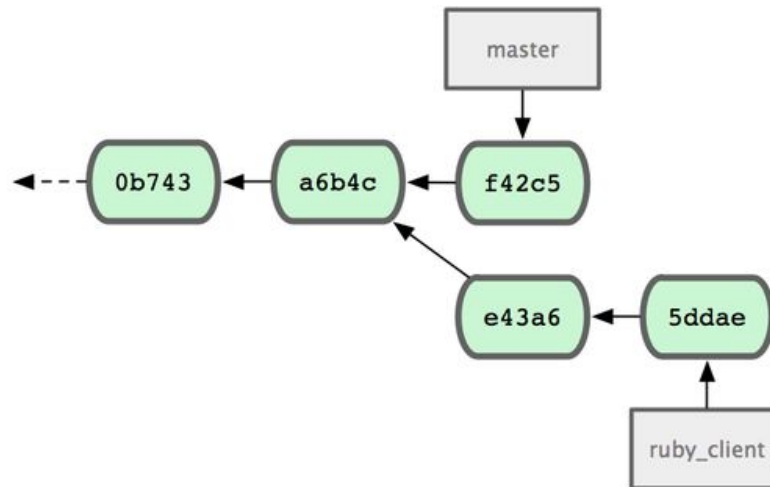


I cherry-pickujemy commit e43a6 z mastera:

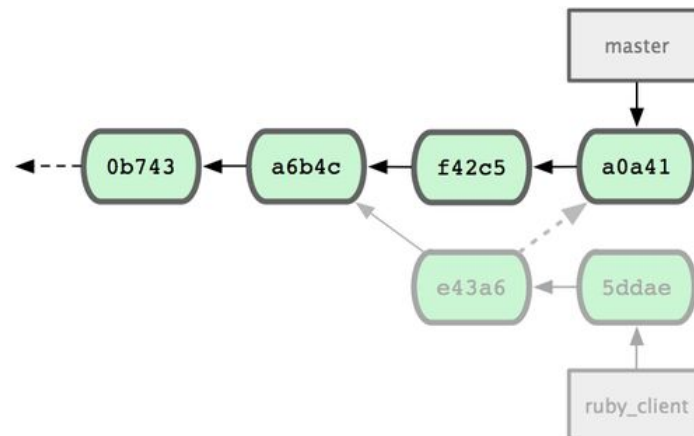


Cherry-pick, czyli wisienka na torcie

Mamy taką strukturę:



I cherry-pickujemy commit e43a6 z mastera:



Git rebase

Git rebase w wielkim uproszczeniu służy do przeniesienia wszystkich commitów z jednej gałęzi na drugą.

git rebase <nazwaGałęzi>

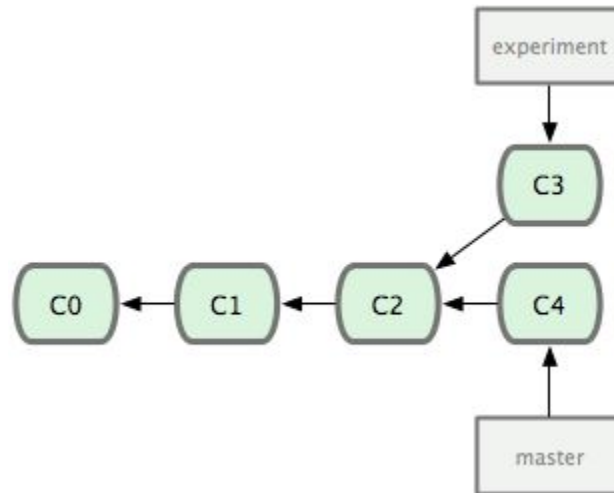
w przypadku git rebase, używamy tego polecenia z gałęzi którą chcemy przenieść (odwrotnie niż w przypadku **git merge**, czy **git cherry-pick**), a w parametrze umieszczamy nazwę gałęzi, na którą chcemy się przenieść.

Git rebase i git merge są do siebie podobne, ale różni je kilka rzeczy:

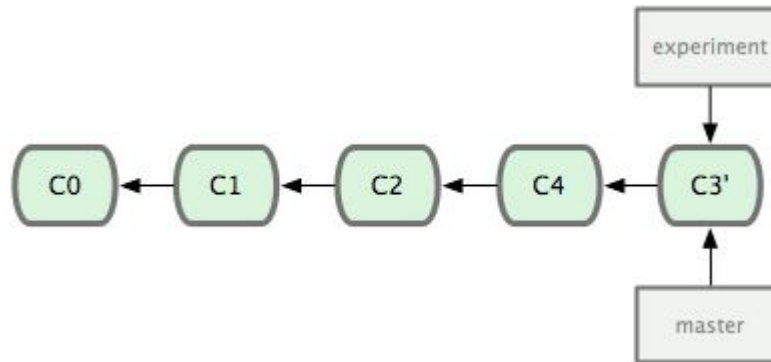
- merge tworzy merge commit, czego nie robi rebase
- rebase przenosi commity na inny branch, a więc w efekcie tworzy się jeden branch o bardzo dużej ilości commitów
- merge nie przenosi commitów, tworzy jedynie merge commit, który ma referencje do kilku rodziców (parent). Jest tyle rodziców ile gałęzi scaliliśmy. Rodzicem jest ostatni commit w każdej scalonej gałęzi.
- historia commitów po rebase jest liniowa, to znaczy widzimy zmiany gałąź po gałęzi, jakby były robione po kolei. historia po merge jest posortowana w kolejności (czyli według daty).
- commity przeniesione po rebase mają nowe hashe, bo zmienia im się parent, co nie występuje po merge, bo commity nie są przenoszone.

Git rebase

Przykład:

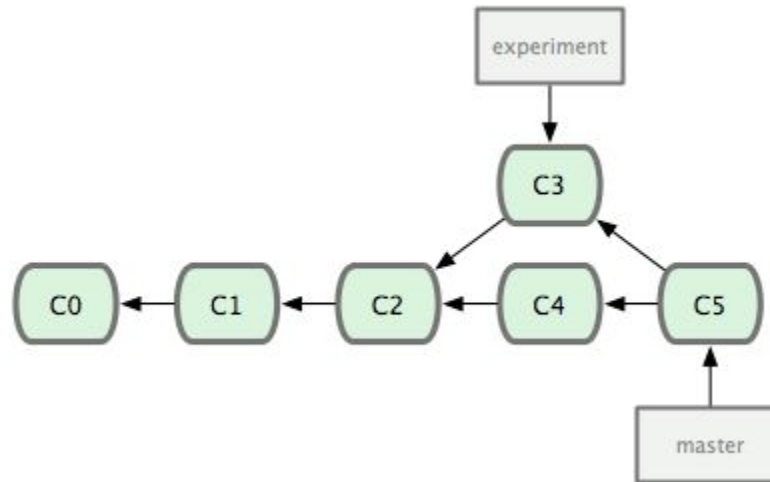


Po rebase:

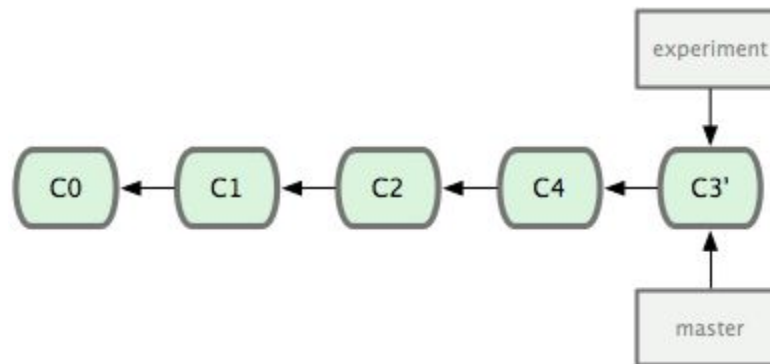


rebase vs merge

Po merge



Po rebase:



rebase vs merge

historia commitów po git merge:

```
commit 1a6f2f5eb0e5d46ccac822b5c43db722ff8b92b (HEAD -> master, tutorial1)
Merge: e68eeb6 190f490
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 13:40:07 2019 +0100

    Merge branch 'tutorial1'

commit 190f490084c393a4f7bb2759c907186f64b2d1c2
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 13:39:39 2019 +0100

    Trzeci commit tutorial

commit e68eeb66462cbcacf0475d26f66c7535af5d2aca
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 13:39:25 2019 +0100

    Pierwszy commit masterrr

commit ee16ba3064e0d966c7e9ad58101b4de60268be13
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 13:39:07 2019 +0100

    Drugi commit tutorial

commit e89c3b8cc2b624e7d8c5c14e9a954412698badc6
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 13:38:53 2019 +0100

    Pierwszycommittutorial
```

rebase vs merge

historia commitów po git rebase:

```
commit 84ad66b1b34b8e3f54ab22a8d43d1efdde2ca2d0 (HEAD -> master, tutorial3)
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 13:50:22 2019 +0100

    Tutorial3 2

commit 29d381f95ce69d0d0e724a0b6cbc79419632da5d
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 13:49:45 2019 +0100

    Tutorial3 1

commit 20c21e4df83ddf3c11c1b7aa2acaeefdbbee2e534
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 13:50:39 2019 +0100

    Master 2

commit 9bd353815084556a2c752f3f28be523f4e13a08b
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 13:50:02 2019 +0100

    Master 1

commit 59f42322ae6910e211f64d89b8a2c49eecb36de7 (tutorial2)
Author: Jakub Danielczyk <jakub.danielczyk@iteo.com>
Date: Thu Feb 21 13:33:53 2019 +0100

    tutorial2
```

Oznaczmy wersję

Git pozwala na tagowanie (etykietowanie) commitów. Aby to zrobić wystarczy użyć polecenia `git tag`:

`git tag <nazwaEtykiety>`

stosuje się to najczęściej aby oznaczyć nową wersję aplikacji.

Do miejsca, w którym jest etykieta bardzo łatwo można się przenieść, stosując `git checkout`

```
~/Dev/tutorial ➤ master ➤ git tag v1.0.0  
~/Dev/tutorial ➤ master ➤ git checkout v1.0.0
```

Otagowany zostanie ten commit, na który wskazuje HEAD w momencie wywołania polecenia `git tag`.

Zdalne repozytoria

Bardzo często zdarza się, że kilka osób pracuje nad jednym kodem. Wtedy wspólny kod zazwyczaj jest trzymany na serwerze, do którego mamy dostęp przez serwisy takie jak: Github, Gitlab, czy Bitbucket. Aby pobrać repozytorium z serwera na nasz komputer, należy użyć polecenia `git clone` w miejscu, w którym chcemy je zapisać.

`git clone <linkDoRepozytorium>`

Repozytorium pobrane w ten sposób zostaje zapisane jako **origin**. Można to sprawdzić poleceniem **`git remote -v`**

```
~/Dev ➤ git clone https://github.com/oleklamza/nwta.git
Cloning into 'nwta'...
remote: Enumerating objects: 45, done.
remote: Counting objects: 100% (45/45), done.
remote: Compressing objects: 100% (29/29), done.
remote: Total 45 (delta 12), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (45/45), done.
~/Dev ➤ cd nwta
~/Dev/nwta ➤ master ➤ git remote -v
origin https://github.com/oleklamza/nwta.git (fetch)
origin https://github.com/oleklamza/nwta.git (push)
~/Dev/nwta ➤ master ➤ |
```

Zdalne repozytoria

Skąd wziąć link do repozytorium? - Github

The screenshot shows the GitHub interface for the repository 'oleklamza / nwtA'. At the top, there are buttons for 'Watch' (1), 'Star' (0), and 'Fork' (6). Below these are tabs for 'Code', 'Issues' (0), 'Pull requests' (0), 'Projects' (0), 'Wiki', and 'Insights'. A message states 'No description, website, or topics provided.' Below this, statistics show '17 commits', '1 branch', '0 releases', and '7 contributors'. A row of buttons includes 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and a green 'Clone or download' button. A dropdown menu is open from the 'Clone or download' button, showing 'Clone with HTTPS' (with a red '2' next to it), 'Use SSH', and the URL 'https://github.com/oleklamza/nwtA.git' (circled in black). At the bottom of the dropdown are 'Open in Desktop' and 'Download ZIP' buttons. The repository content area shows a 'Merge pull request #7 from barkep/patch-2' and a file 'README.md' with an 'Update README.md' button.

oleklamza / nwtA

Watch 1 Star 0 Fork 6

<> Code Issues 0 Pull requests 0 Projects 0 Wiki Insights

No description, website, or topics provided.

17 commits 1 branch 0 releases 7 contributors 1

Branch: master New pull request Create new file Upload files Find file Clone or download

oleklamza Merge pull request #7 from barkep/patch-2

README.md Update README.md

README.md

Clone with HTTPS 2 Use Git or checkout with SVN using the web URL.
https://github.com/oleklamza/nwtA.git

Open in Desktop Download ZIP

Zdalne repozytoria

Mając już powiązanie ze zdalnym repozytorium, możemy “wypchnąć” obecny stan kodu w danej gałęzi na serwer, lub pobrać kod w danej gałęzi z serwera na lokalną maszynę

git push <nazwaZdalnegoRepo> <nazwaGałęzi>

git pull <nazwaZdalnegoRepo> <nazwaGałęzi>



```
~/Dev/nwta ➤ master ➤ git push origin master
remote: Permission to oleklamza/nwta.git denied to JakubDanielczyk.
fatal: unable to access 'https://github.com/oleklamza/nwta.git/': The requested
URL returned error: 403
✗ ~/Dev/nwta ➤ master |
```

Jeśli nie jesteśmy twórcami repozytorium na serwerze musimy zostać przez twórcę dodani do grupy collaborators.

Zdalne repozytoria

Dodawanie do collaborators - Github

The screenshot shows the GitHub interface for the repository **Limbou / nwtA**, which is forked from **oleklamza/nwtA**. The repository has 0 watches, 0 stars, and 6 forks. The navigation bar includes links for Code, Pull requests (0), Projects (0), Wiki, Insights, and **Settings** (marked with a red '1' and a black circle). On the left sidebar, the **Collaborators** option is highlighted with a red '2' and a black circle. The main content area is titled **Collaborators** with the subtitle 'Push access to the repository'. It contains a message: 'This repository doesn't have any collaborators yet. Use the form below to add a collaborator.' Below this is a search section titled 'Search by username, full name or email address' with a note: 'You'll only be able to find a GitHub user by their email address if they've chosen to list it publicly. Otherwise, use their username instead.' At the bottom of the search section is a text input field and an 'Add collaborator' button.

Limbou / nwtA
forked from oleklamza/nwtA

Watch 0 Star 0 Fork 6

Code Pull requests 0 Projects 0 Wiki Insights **Settings**

Options
Collaborators
Branches
Webhooks
Notifications
Integrations & services
Deploy keys

Moderation
Interaction limits

Collaborators Push access to the repository

This repository doesn't have any collaborators yet. Use the form below to add a collaborator.

Search by username, full name or email address
You'll only be able to find a GitHub user by their email address if they've chosen to list it publicly. Otherwise, use their username instead.

Add collaborator

Zdalne repozytoria

Pushowanie i pullowanie zmian

```
~/Dev/nwta ➤ ↶ master ➤ git push origin master  
Everything up-to-date  
~/Dev/nwta ➤ ↶ master ➤ git pull origin master  
From github.com:Limbou/nwta  
* branch                master      -> FETCH_HEAD  
Already up to date.  
~/Dev/nwta ➤ ↶ master ➤ |
```

Zdalne repozytoria

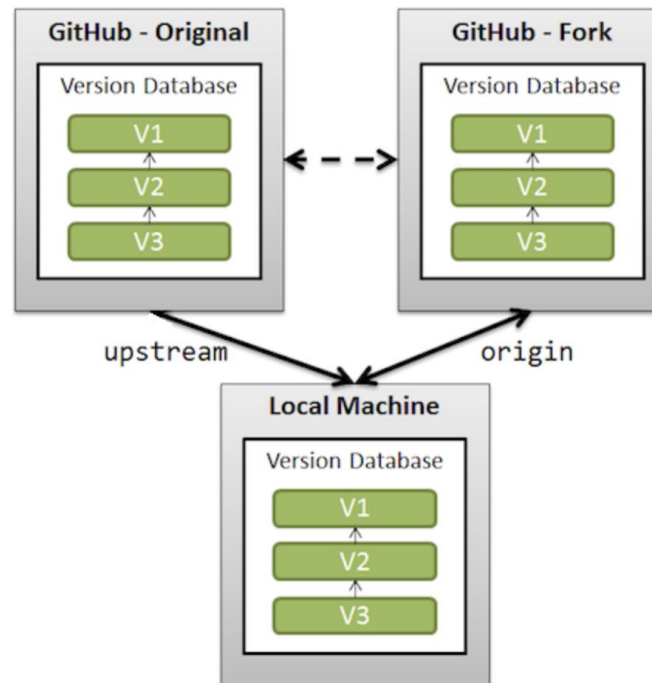
Wyobraźmy sobie jednak taką sytuację: Stworzyłem bibliotekę do komunikacji z pozaziemskim życiem. Chcąc podzielić się nią z innymi programistami, wrzucam ją na Githuba, jako publiczne repozytorium - każdy może je pobrać (git clone) i wykorzystać. Po jakimś czasie biblioteka zyskuje wielką popularność i inni ludzie starają się ją udoskonalić. Pewnego dnia 50 osób znajduje różne, niezwiązane se sobą błędy, które wymagają naprawy. Aby mogli swoje wersje kodu z poprawkami wrzucić na moje repozytorium na Githubie, muszę ich wszystkich z osobna dodać do grupy Collaborators. To rozwiązanie ma 2 wady:

1. Założyłem, że osób jest 50. A co jeśli byłoby ich 200? Dodawanie każdego z osobna to wielka strata czasu.
2. Po dodaniu użytkownika może on dowolnie modyfikować repozytorium. Ktoś mógłby na przykład usunąć wszystkie pliki i wypchnąć te zmiany na serwer (git push). Spowoduje to utratę wszystkich danych. Jedynym ratunkiem jest repozytorium trzymane na moim komputerze.

Zdalne repozytoria

Jest jednak rozwiązanie tego problemu: Fork oraz Pull request.

Fork to inaczej **git clone** po stronie serwera. Serwer kopiuje główne repozytorium i oznacza nas jako twórców tej kopii. Dzięki temu mamy na serwerze aktualną (w momencie forka) wersję kodu, którą możemy dowolnie modyfikować. Taki fork powinniśmy skopiować na nasz komputer (ponownie - **git clone**) i rozpocząć pracę.



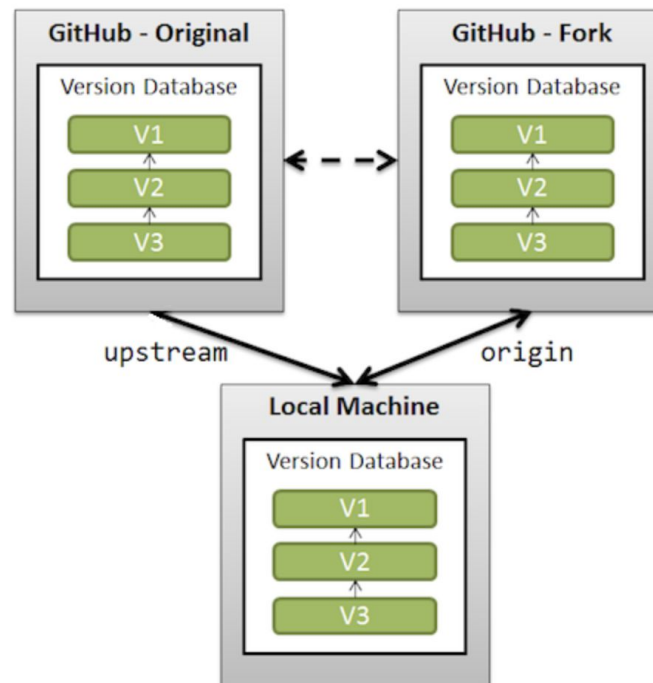
Zdalne repozytoria

Od tego momentu stosuje się nazewnictwo:

Główne, wspólne repozytorium - upstream

Osobista kopia głównego repozytorium na serwerze - origin

Klonując forka git automatycznie przypisze go nam jako origin (tak jak stało się to na poprzednich slajdach)



Zdalne repozytoria

Aby mieć dostęp z poziomu naszego komputera do upstreama, należy go dodać jako zdalne repozytorium.

```
git remote add upstream <linkDoRepozytorium>
```

```
~/Dev/nwta ➤ master ➤ git remote add upstream https://github.com/oleklamza/nwta.git
~/Dev/nwta ➤ master ➤ git remote -v
origin  git@github.com:Limbou/nwta.git (fetch)
origin  git@github.com:Limbou/nwta.git (push)
upstream      https://github.com/oleklamza/nwta.git (fetch)
upstream      https://github.com/oleklamza/nwta.git (push)
~/Dev/nwta ➤ master ➤ |
```

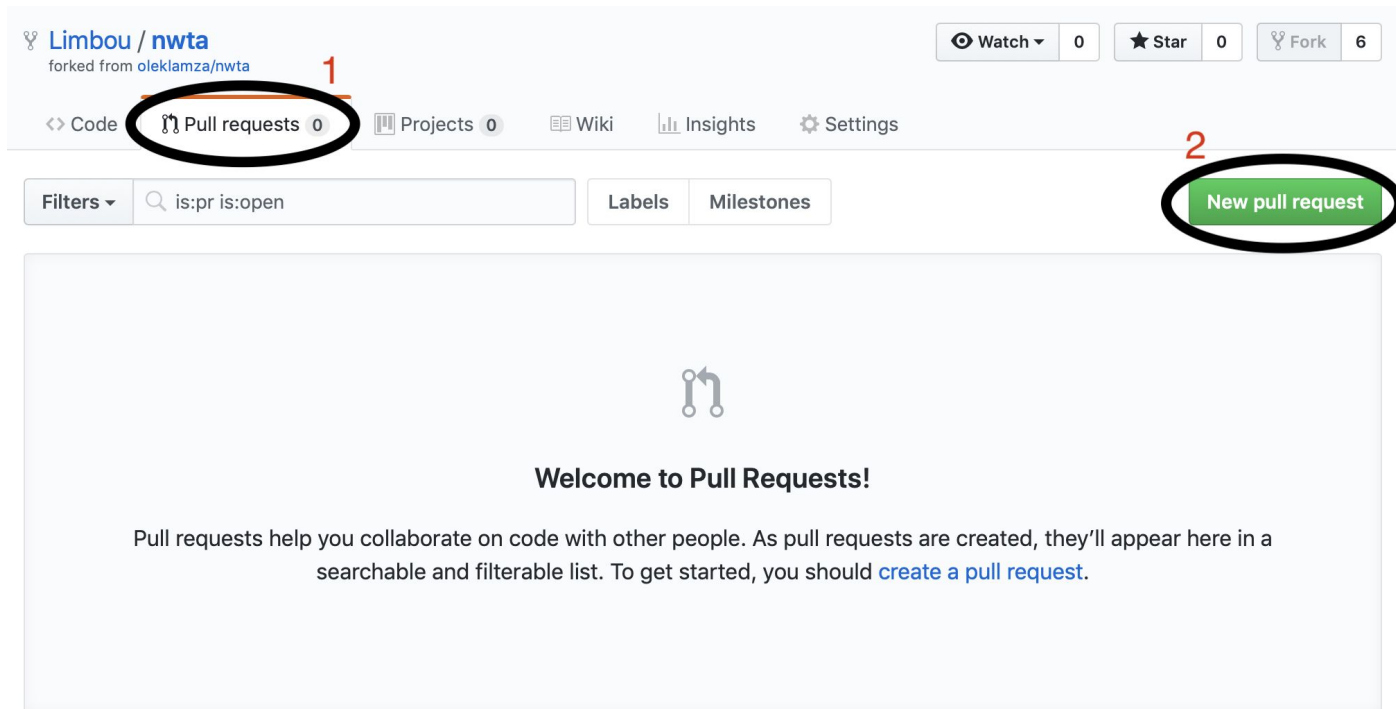
Od teraz chcąc pobrać najnowsze zmiany, pobieramy je z upstreama.

Zmiany zacommitowane przez nas pchniemy na origin.

Zdalne repozytoria

Jak przenieść zmiany z origina na upstream?

Po tym jak wypchnęliśmy zmiany na origin, trzeba stworzyć tzw. Pull request (merge request).



Zdalne repozytoria

Jak przenieść zmiany z origina na upstream?

Po tym jak wypchnęliśmy zmiany na origin, trzeba stworzyć tzw. Pull request (merge request).

The screenshot shows the GitHub interface for a pull request. At the top, the repository name 'oleklamza / nwta' is displayed with 1 watch, 0 stars, and 6 forks. Below this is a navigation bar with 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', and 'Insights'. The main heading is 'Comparing changes', followed by a subtext: 'Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).' Below this is a comparison bar showing 'base repository: oleklamza/nwta', 'base: master', 'head repository: Limbou/nwta', and 'compare: master'. A green checkmark indicates 'Able to merge. These branches can be automatically merged.' A prominent green button labeled 'Create pull request' is circled in black. Below the button, it says 'Discuss and review the changes in this comparison with others.' Further down, a summary bar shows '1 commit', '1 file changed', '0 commit comments', and '1 contributor'. The commit history section shows 'Commits on Feb 19, 2019' with a commit by 'JakubDanielczyk' titled 'pierwszy commit' with hash 'edc105b'. At the bottom, it says 'Showing 1 changed file with 0 additions and 0 deletions.' and shows a diff for 'hello.txt' with 'No changes.' and buttons for 'Copy path' and 'View file'.

oleklamza / nwta

Watch 1 Star 0 Fork 6

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

base repository: oleklamza/nwta base: master head repository: Limbou/nwta compare: master

✓ Able to merge. These branches can be automatically merged.

Create pull request Discuss and review the changes in this comparison with others.

1 commit 1 file changed 0 commit comments 1 contributor

Commits on Feb 19, 2019

JakubDanielczyk pierwszy commit edc105b

Showing 1 changed file with 0 additions and 0 deletions.


Unified Split

0 hello.txt Copy path View file

No changes.

Zdalne repozytoria

Po utworzeniu pull requesta, twórca głównego repozytorium może wcisnąć przycisk 'merge', aby zaakceptować zmiany i scalić je z obecnym kodem.

**This branch has no conflicts with the base branch**
Merging can be performed automatically.

Merge pull request ▼

You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Dzięki temu rozwiązaniu nie trzeba dodawać użytkowników do Collaborators, oraz nie ma ryzyka pojawienia się niechcianych zmian w głównym repozytorium.

GitHub vs BitBucket vs GitLab

Każdy z tych serwisów oferuje podstawowe narzędzia takie jak:

- Pull/Merge request
- Code review
- Inline editing
- Issue tracking
- Markdown
- Zarządzanie uprawnieniami
- Hosting stron statycznych
- Fork / Clone
- CI / DI

Kiedy wybrać GitHub?

Jeśli chcemy pracować nad projektem open source, najlepszym wyborem będzie GitHub, ponieważ jest to serwis o największym zasięgu (popularność). Nie będzie to jednak dobry wybór jeśli zależy nam na prywatnym repozytorium. Oczywiście można to zmienić wykupując odpowiedni pakiet.

Kiedy wybrać BitBucket?

BitBucket może być dobrym wyborem jeśli korzystamy z innych produktów Atlassian takich jak: Jira, Confluence i Bamboo z powodu ułatwionej integracji. Sprawa prywatnych repozytoriów wygląda tu już znacznie lepiej. Dodatkowym atutem jest możliwość integracji z trello oraz możliwość używania Mercurial'a.

Kiedy wybrać GitLab?

Warto wybrać GitLab jeśli zależy nam na pracy w prywatnym repozytorium w większym zespole. Darmowa wersja umożliwiającą hostowanie na własnym serwerze może okazać się dodatkowym atutem.

Ciekawe materiały

Do nauki gita w przeglądarce z graficznym przedstawieniem gałęzi:

<https://learngitbranching.js.org>

Dokumentacja gita: <https://git-scm.com/book/pl/v1/Pierwsze-kroki-Podstawy-Git>