Zaawansowane praktyki programistyczne

Lab 3 21.04.2018

Cezary Hołub

Wrocławska Wyższa Szkoła Informatyki Stosowanej

Wykład 3: Git - praca z repozytorium zdalnym

Git - zdalne repozytorium, praca w grupie

- Jak już wiemy w pracy z rozproszonym systemem kontroli wersji wszystkie repozytoria są równe, a wskazanie repozytorium, z którego na przykład budujemy wersję lub uruchamiany testy jest tylko kwestią umowy

Trzy kluczowe komendy:

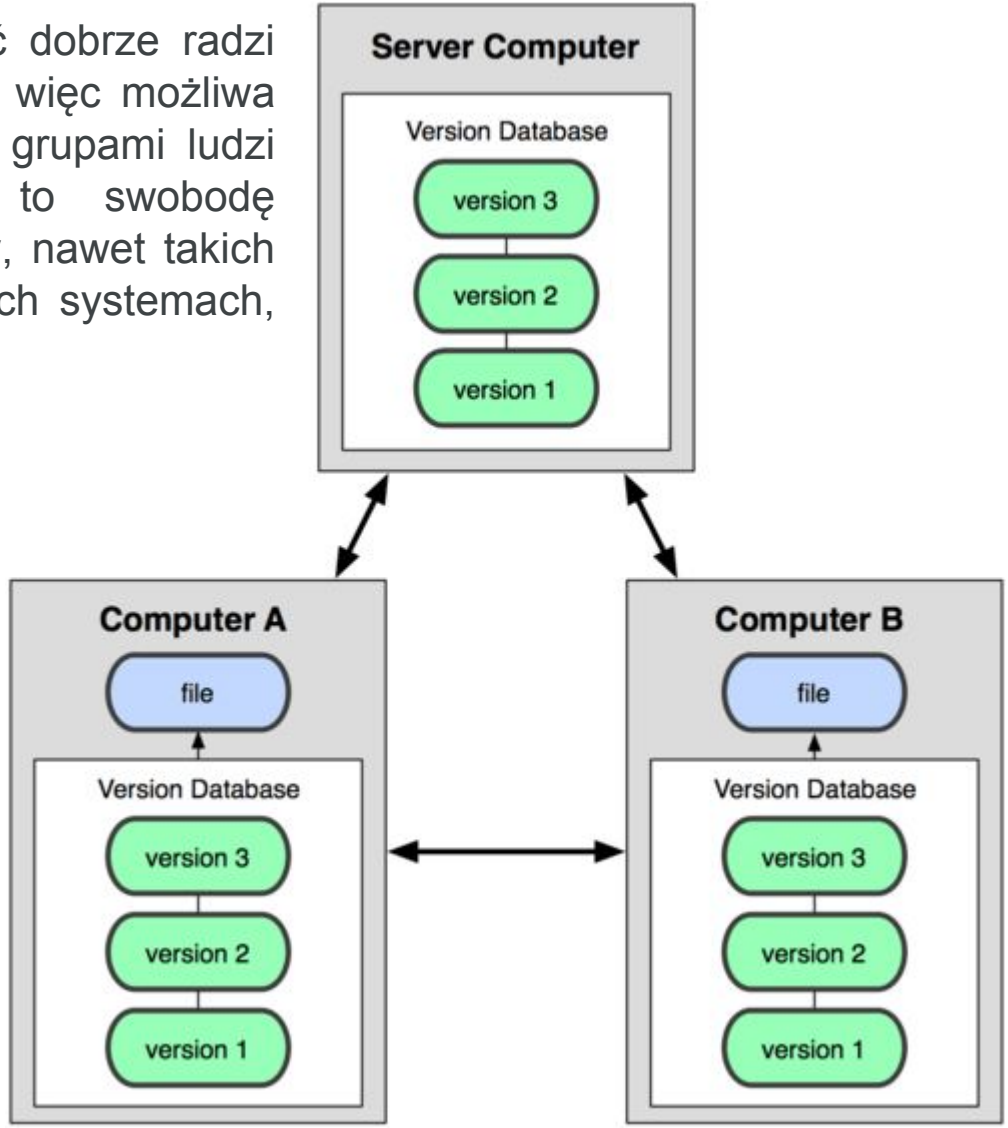
- **Clone**
- **Push**
- **Pull**

Rozproszone systemy kontroli wersji

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

Rozproszone systemy kontroli wersji c.d.

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



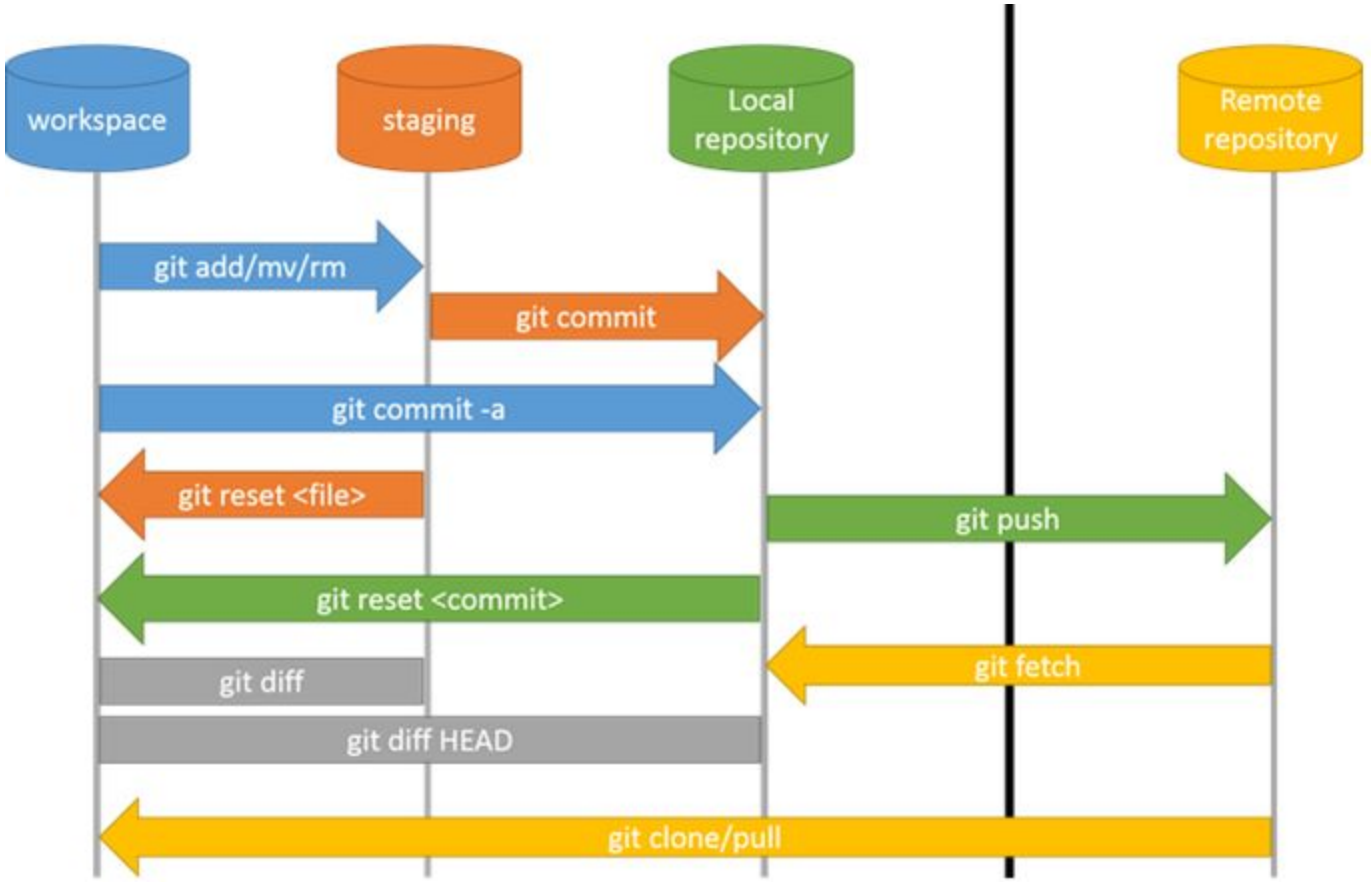
Co zyskujemy dzięki rozproszeniu?

Przede wszystkim pracujemy lokalnie – tworzymy branche wtedy kiedy my ich potrzebujemy, nazywamy je tak jak chcemy (nie ma konfliktów nazw często spotykanych w repozytoriach centralnych), commitujemy zmiany kiedy uznamy jakąś część pracy za wykonaną. **Pomimo pracy lokalnej jesteśmy cały czas pod kontrolą wersji** – możemy zarządzać nawet małymi lokalnymi zmianami, które potencjalnie mogą zawierać błędy.

Zarządzamy zmianami a nie wersjami. Wbrew pozorom to bardzo duża zmiana w podejściu do zarządzania kodem – zamiast zastanawiać się jak uaktualnić jedną wersję tak, żeby była taka sama jak inna – zarządzamy zmianami – mówimy więc ‘daj mi swój zestaw zmian’. Przy takim podejściu mergowanie zmian staje się codziennością w związku z czym musi być wydajne i bezproblemowe i takie właśnie jest. Dzięki temu pośrednio zyskujemy też silne wsparcie dla nieliniowego rozwoju projektu.

Kompletny projekt wraz z historią zmian istnieje w wielu miejscach. Podczas zakładania (klonowania) projektu dostajemy pełną kopię repozytorium (wraz z historią zmian). Dzięki temu nie mamy jednego newralgicznego punktu, ale te same źródła w wielu miejscach.

Git - zdalne repozytorium, praca w grupie



Git - zdalne repozytorium, praca w grupie

- Jak już wiemy w pracy z rozproszonym systemem kontroli wersji wszystkie repozytoria są równe, a wskazanie repozytorium, z którego na przykład budujemy wersję lub uruchamiany testy jest tylko kwestią umowy

- **`$git remote -`** Wyświetla listę repozytoriów zdalnych.

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

- **`$ git remote -v`**

- Jeśli posiadasz więcej niż jedno zdalne repozytorium polecenie wyświetli je wszystkie.

Clone

- **`$git clone git://serwer.com/repo.git repo`** - Klonuje repozytorium zdalne do katalogu repo

Przykład:
Załóżmy, że pracujemy w dwuosobowym zespole (user1, user2). User2 tworzy swoje lokalne repozytorium przez sklonowanie repozytorium należącego do user1:

```
$git clone git-repo-user1 git-repo-user2
Cloning into git-repo-user2…
done.
```

Następnie user2 edytuje lokalnie jakieś pliki umieszcza je w swoim repozytorium:

```
$cd git-repo-user2
$git commit –a –m 'Zmiany user2'
```

Pull

- **`$git pull -`** Pobiera zmiany z odpowiednich gałęzi zdalnych do gałęzi śledzących te gałęzie zdalne. Próbuje automatycznie zmergować zmiany. Może się to nie udać i doprowadzić do powstania konfliktu.

Przykład c.d.

Gdy user2 skończy, może powiadomić usera1 o możliwości zaciągnięcia zmian – wtedy user1 może dociągnąć zmiany z repozytorium usera2 za pomocą polecenia pull:

```
$git pull /home/atena/git-repo-user2 master
```

Polecenie pull powoduje pobranie zmian ze wskazanego repozytorium i zmergowanie ich z naszym kodem (fetch + merge). Wskazane jest zacommitowanie wszystkich lokalnych zmian przed wykonaniem operacji pull – pozwoli to na bezproblemowe przeprowadzenie merge'a.

Fetch

- **`$git fetch -`** Działa jak **`git pull`** z tą różnicą, że nie wmergowuje automatycznie zmian do lokalnych gałęzi. Po wykonaniu **`git fetch`** można samemu zmergować gałąź zdalną z odpowiednią gałęzią lokalną używając np. **`git merge repo/B`**.

- Istnieje możliwość ściągnięcia zmian z innego repozytorium bez ich mergowania – służy do tego polecenie fetch:

- ```
  $git fetch /home/atena/git-repo-user2 master
  $git log –p HEAD..FETCH_HEAD
  $git merge FETCH_HEAD
  ```
  ```

Alias do zdalnego repozytorium

- Dla ułatwienia pracy można nadać alias zdalnemu repozytorium, z którym pracujemy:
- **`$git remote add user2 /home/atena/git-repo-user2`**

Push

- **`$git push repo`** - Zapisuje śledzone gałęzie do zdalnego repozytorium **`repo`**.

Przykład c.d.

Z poziomu swojego repozytorium możemy też ‘wepchnąć’ zmiany do repozytorium zdalnego – praca z Gitem przypomina wtedy pracę z repozytorium centralnym. Służy do tego polecenie push:

```
$git push user2 master
```

Push c.d.

- Co ważne – Git domyślnie nie pozwoli na wepchnięcie zmian do zdalnego repozytorium jeśli wskazany branch został w nim zacheckoutowany. Dostaniemy wtedy taki komunikat:

- ```
  error: refusing to update checked out branch: refs/heads/master
  error: By default, updating the current branch in a non-bare repository
  error: is denied, because it will make the index and work tree inconsistent
  error: with what you pushed, and will require 'git reset --hard' to match
  error: the work tree to HEAD.
  ```

- Możemy wtedy zrobić dwie rzeczy – albo w zdalnym repozytorium przejść (checkout) na inny branch lub przekształcić zdalne repozytorium w repozytorium centralne (bare), czyli takie, które zawiera jedynie katalog .git i nie zawiera żadnych zacheckoutowanych plików

- `git config --bool core.bare true`
  ```

Rebase

- Rebase to w uproszczeniu przesunięcie miejsca utworzenia brancha. Załóżmy, że mamy w projekcie branch główny oraz jakiś dodatkowy z funkcjonalnością przygotowywaną dla konkretnego klienta. Jeśli chcemy aby branch dedykowany dla klienta zawierał wszystkie zmiany wrzucane do brancha głównego musimy dokonywać merge'y.

- Git daje nam możliwość wykonania operacji rebase, która spowoduje przesunięcie miejsca utworzenia brancha:

- ```
  $git checkout clientX_branch
  $git rebase origin
  ```

- Po takiej operacji Git zapisze wszystkie commity od momentu utworzenia brancha jako patche w katalogu .git/rebase, zaktualizuje brancha tak, żeby wskazywał aktualną wersję w branchu głównym i na koniec zaaplikuje odłożone patche. Dzięki temu historia zmian będzie serią commitów bez merge'y.
  ```

Operacje na gałęziach

- **`git branch`** - Wypisuje gałęzie w lokalnym repozytorium.

- **`git branch B`** - Tworzy nową gałąź o nazwie B. Utworzona gałąź wskazuje na bieżącą wersję. Bieżąca gałąź nie jest zmieniana, tzn. po wykonaniu komendy nadal pracujemy w gałęzi w której wcześniej byliśmy, a nie w nowo utworzonej. Właściwie częściej stosuje się **`git checkout -b B`** aby od razu zmienić bieżącą gałąź na nowo utworzoną. Gałąź tworzona jest lokalnie i nie będzie automatycznie uwzględniona przy wykonywaniu **`git push`**. Aby wypchnąć ją do repozytorium zdalnego należy użyć **`git push -u repo B`**, gdzie repo to nazwa repozytorium (zwykle origin). Opcja **`-u`** powoduje, że później bezargumentowe **`git push`** i **`git pull`** będą uwzględniać tę gałąź, tzn. gałąź B będzie śledzić odpowiednią gałąź w repozytorium zdalnym repo.

Operacje na gałęziach c.d.

- **`git branch -d B`** - Usuwa gałąź B, tzn. usuwa sam wskaźnik B a nie wersję na którą ta gałąź wskazuje. Wersja ta może być wciąż dostępna z innych gałęzi.

- **`git checkout B`** - Zmienia bieżącą gałąź na B, o ile B jest nazwą gałęzi. HEAD także jest ustawiane na wersję wskazywaną przez B.

- **`git checkout -b B`** - Mniej więcej to samo co:
  - **`git branch B`**
  - **`git checkout B`**

- **`git checkout rev`** - Jeśli rev jest specyfikacją wersji (np. HEAD^, HEAD~2), to HEAD zostaje ustawione na rev bez zmiany bieżącej gałęzi. W ten sposób można znaleźć się w stanie z obciętą głową.

Operacje na gałęziach - mergowanie

- **`git merge A B C`** - Tworzy nową wersję poprzez włączenie wersji wskazywanych przez gałęzie A, B i C do bieżącej wersji. Zwykle chcemy wmergować tylko jedną gałąź, ale można i kilka na raz. Przy mergowaniu może pojawić się konflikt, czyli niekompatybilne zmiany w tym samym pliku w różnych wersjach. Wtedy GIT nie tworzy automatycznie nowej wersji, tylko wypisuje, że wystąpił konflikt. W katalogu roboczym znajdują się wtedy pliki które udało się zmergować oraz skonfliktowane pliki. Musimy ręcznie rozwiązać konflikty, po czym samemu zacommitować nową wersję. Żeby zobaczyć jakie pliki są w konflikcie używamy **`git status`**. W każdym ze skonfliktowanych plików w katalogu roboczym będą miejsca w rodzaju:

  <<<<<<< HEAD
  Kod z wersji HEAD
  =======
  Kod z gałęzi A
  >>>>>>> A

- Trzeba te miejsca ręcznie poprawić i potem dodać poprawiony plik do indeksu przez
  `git add`. Na koniec gdy rozwiążemy wszystkie konflikty robimy
  `git commit.`

Operacje na zdalnym repozytorium

- **`git remote`** - Wyświetla listę repozytoriów zdalnych.
- **`git remote add repo URL`** - Dodaje repozytorium zdalne o adresie URL.
- **`git push`** - Zapisuje do gałęzi zdalnych w domyślnym repozytorium zdalnym (zwykle origin) zmiany z gałęzi je śledzących w repozytorium lokalnym. Właściwie to nie do końca prawda, ale dla uproszczenia można przyjąć, że tak jest.
- **`git push repo`** - Zapisuje śledzone gałęzie do zdalnego repozytorium repo.
- **`git push repo B`** - Zapisuje w repozytorium zdalnym gałąź B.
- **`git push -u repo B`** - Jak wyżej, ale dodatkowo sprawia, że gałąź lokalna B będzie śledzić jej zdalny odpowiednik (repo/B). Ten wariant komendy push powinien być używany gdy po raz pierwszy zapisujemy lokalnie utworzoną gałąź w repozytorium zdalnym.

Operacje na zdalnym repozytorium c.d.

`git pull` - Pobiera zmiany z odpowiednich gałęzi zdalnych do gałęzi śledzących te gałęzie zdalne. Próbuje automatycznie wmergować zmiany. Może się to nie udać i doprowadzić do powstania konfliktu.

`git pull repo` - Pobiera śledzone gałęzie ze zdalnego repozytorium repo.

`git pull repo B` - Pobiera gałąź B ze zdalnego repozytorium repo i zapisuje ją w lokalnej gałęzi B.

Operacje na zdalnym repozytorium c.d.

- **`git fetch`** - Działa jak git pull z tą różnicą, że nie wmergowuje automatycznie zmian do lokalnych gałęzi. Po wykonaniu **`git fetch (git fetch repo, git fetch repo B)`** można samemu zmergować gałąź zdalną z odpowiednią gałęzią lokalną używając np. git merge repo/B.

- **`git branch -r`** - Wypisuje gałęzie zdalne.

- **`git checkout -b B repo/B`** - Tworzy lokalnie gałąź B śledzącą gałąź zdalną repo/B. Zmienia gałąź bieżącą na B. Należy pamiętać, że pull/fetch nie tworzą lokalnie nowych gałęzi które są w repozytorium zdalnym, ale nie ma ich w lokalnym. Żeby taką gałąź stworzyć trzeba użyć powyższej komendy.

- **`git branch -u repo/B B`** - Po wykonaniu tej komendy gałąź B będzie śledzić gałąź zdalną repo/B. Przydaje się jeśli zapomnimy dać **`-u`** przy **`git push`**. W starszych wersjach GITa ta komenda ma postać: **`git branch --set-upstream B repo/B`**.

Inspekcja zdalnych zmian

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

- ```
  $ git remote show origin
  * remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch
  master
  master
  Tracked remote branches
  master
  ticgit
  ```

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

Git - podstawowe pojęcia w pigułce

- **katalog roboczy** (ang. *working directory, working tree*) - Katalog w którym dokonujemy zmian. Zawiera wycheckoutowaną zawartość ostatniej wersji w bieżącej gałęzi (czyli bieżącej wersji, HEAD), plus zmiany których dokonaliśmy.

- **index** (ang. *staging area*) - Trzyma zmiany które zostaną zacommitowane do nowej wersji (w lokalnym repozytorium) przy wykonaniu **`git commit`**. Zmiany z katalogu roboczego można dodać do indeksu przez **`git add`**.

- **repozytorium lokalne** - Nasza lokalna kopia repozytorium. Znajduje się w podkatalogu .git katalogu roboczego. Ogólna zasada jest taka, że wszystkie operacje są wykonywane na lokalnym repozytorium. Jeśli chcemy nasze zmiany upublicznić, to trzeba je explicite ,,wypchnąć" do repozytorium zdalnego używając **`git push`**.

Git - podstawowe pojęcia w pigułce c.d.

- **repozytorium zdalne** - Może być więcej niż jedno. Domyślne zwykle nazywa się **origin**. Żeby zmiany z repozytorium lokalnego znalazły się w zdalnym trzeba wykonać **`git push`**. Aby pobrać zmiany z repozytorium zdalnego do lokalnego używamy **`git pull`**. Należy mieć na uwadze, że obiekty w repozytorium zdalnym wcale nie muszą odpowiadać obiektom w lokalnym repozytorium. Mogą np. być w repozytorium zdalnym gałęzie których nie ma w lokalnym i vice versa. Można zapisywać zmiany do wielu repozytoriów zdalnych, które to repozytoria mogą się różnić między sobą np. istnieniem określonych gałęzi. Można zapisywać zmiany tylko z niektórych gałęzi w repozytorium lokalnym.

- **wersja** (ang. *commit*) - Wersja plików zapisana w repozytorium. Wersje tworzą graf acykliczny skierowany. Wersja A jest bezpośrednim następnikiem (syn, *child*) wersji B jeśli A powstała bezpośrednio poprzez zmianę wersji B. Wersje można łączyć (**`git merge`**), więc dana wersja może mieć więcej niż jednego bezpośredniego poprzednika (ojciec, *parent*). Tworzenie nowej wersji (w repozytorium lokalnym) zawierającej zmiany zapisane w indeksie dokonuje się przez **`git commit`**. Do zmiany plików w katalogu roboczym na pliki z danej wersji służy **`git checkout`**.

Git - podstawowe pojęcia w pigułce c.d.

- **gałąź** (ang. *branch*) - Liniowo uporządkowany zbiór wierzchołków w grafie wersji. Można myśleć o gałęzi jako o wskaźniku na jakąś wersję w grafie wersji, tzn. utożsamiać ją z ostatnią wersją w gałęzi. Zwykle istnieje bieżąca gałąź, której wskaźnik jest przesuwany razem z HEAD przy wykonywaniu operacji git commit, git reset, itp. Stan w którym nie ma bieżącej gałęzi nazywa się detached head.

- **gałąź zdalna** (ang. *remote branch*) - Gałąź w zdalnym repozytorium.

- **gałąź śledząca** (ang. *tracking branch*) - Gałąź w repozytorium lokalnym śledząca (track) gałąź zdalną. Z grubsza rzecz biorąc, jeśli gałąź A w repozytorium lokalnym śledzi gałąź repo/A w jakimś repozytorium zdalnym repo, to zmiany w repo/A będą pobierane do A przez **`git pull`**, a zmiany w A będą zapisywane do repo/A przez **`git push.`**

Git - podstawowe pojęcia w pigułce c.d.

- **HEAD** - Wskazuje na bieżącą wersję z lokalnego repozytorium. W katalogu roboczym znajdują się pliki z wersji wskazywanej przez HEAD zmienione o modyfikacje które wykonaliśmy.

- **ORIG_HEAD** - Poprzednia wartość HEAD, sprzed wykonania którejś z operacji zmieniających HEAD (**`git commit`**, **`git merge`**, **`git pull`**, **`git checkout`**, **`git reset`**, itd).

- **Master** - Główna, domyślna gałąź. Po stworzeniu nowego repozytorium jest to gałąź bieżąca. Typowy sposób pracy z GITem wygląda tak, że dla każdego większego zadania tworzymy nową gałąź, pracujemy w tej gałęzi, a potem jak mamy już gotową funkcjonalność to mergujemy tę gałąź do master.

Git - podstawowe pojęcia w pigułce c.d.

- **Revision** - Można w uproszczeniu powiedzieć, że revision jest specyfikacją wersji, tzn. określa o którą wersję nam chodzi. Zwrot ,,revision" pojawia się czasem w dokumentacji komend. Przydatne sposoby specyfikowania wersji:
  - rev^ - Pierwszy ojciec wersji rev. Pamiętajmy, że wersja może mieć wielu ojców jeśli np. powstała przez połączenie (merge) kilku wersji.
  - rev^N - N-ty ojciec wersji rev.
  - rev~N - Wersja N wersji przed wersją rev, gdzie wybieramy zawsze pierwszego ojca jeśli jest więcej niż jeden. Np. rev~3 to to samo co rev^^^, co jest tym samym co rev^1^1^1.

- Można te sposoby łączyć, np. HEAD~2^2 oznacza drugiego ojca dziadka wersji wskazywanej przez HEAD.

Schemat pracy z projektem (w skrócie)

Schemat zależy od sytuacji projektu. Proponowany podstawowy schemat pracy :

- **`mkdir git_repo`** # tworzymy katalog w którym będzie nasze repozytorium plików, możemy nazwać jak chcemy
- **`cd git_repo`** # przechodzimy do tego katalogu
- **`git init`** # inicjalizujemy bazę repozytorium (katalog .git)

- wprowadzenie zmian w pliku lub plikach
- sprawdzamy jakie pliki zostały zmienione : **`git status`**
- sprawdzamy co zostało zmienione w pliku : **`git diff [file]`**
- zatwierdzamy zmianę : **`git commit -a -m [message]`**

Teraz wgrywamy do tego katalogu pliki lub tworzymy nowe.

- **`git add .`** # dodajemy wszystkie pliki do śledzenia z aktualnego katalogu, kropka oznacza właśnie katalog aktualny
- **`git status`** # sprawdzamy status naszego repozytorium, to polecenie pokaże które pliki zmienione, a które nowe
- **`git commit -a`** # zatwierdzamy zmiany czyli wysyłamy pliki do bazy repozytorium

Schemat pracy z projektem c.d.

Samo polecenie **`git commit`** zapisze tylko pliki, które zostały dodane poleceniem **`git add`**. Dlatego dodajemy **`-a`**, żeby nie dodawać ręcznie każdego zmienionego pliku. Nowe pliki trzeba jednak dodać przez **`git add`**.

- **`git log`** # sprawdzamy historię zatwierdzeń
- **`git whatchanged -p`** # historia zmian razem z diff
- **`git whatchanged --pretty=oneline`** # wyświetla tylko nazwy zmienionych plików

Podstawowa ogłsuga Gita (w skrócie)

- Większość operacji wykonuje się przez "**`git polecenie`**".
- Tworzenie gałęzi to "**`git checkout -b`**".
- "**`git branch`**" powinno być używane tylko do wylistowania i usuwania gałęzi.
- Współdzielisz swoją pracę przez "**`git fetch`**" (pobranie) i "**`git push`**" (wysłanie). To są przeciwieństwa.
- "**`git pull`**" może również zrobić "**`git fetch`**" ale to jest opcjonalne.
- git nie dodaje pustych katalogów do repozytorium. To wynika z koncepcji, że git śledzi zawartość plików a nie pliki.
- Aby uzyskać pomoc na temat jakiegoś polecenia wpisujemy "**`git help polecenie`**" lub "**`git --help`**".

Git/Subversion

Porównanie podstawowych operacji

Polecenie	**git**	**svn**
Utworzenie repozytorium	`git init`	`svnadmin create repo`
Wyświetlenie pliku	`git show rev:path/to/file`	`svn cat url`
Dodanie plików do repozytorium	`git add .; git commit`	`svn import file://repo`
Co się zmieniło (w stylu svn)	`git whatchanged --pretty=oneline`	`svn log`

Git/Subversion c.d.

svn	git
`svn commit`	`git commit`
`svn add / rm / mv / mkdir`	`git add / rm / mv / mkdir`
`svn status / log / diff`	`git status / log / diff`
`svn import`	`git clone`
`svn update`	`git pull`
`svn merge`	`git merge / rebase`
`svn switch`	`git checkout`
`svn cp <trunk> <tag>`	`git tag`
`svn cp <trunk> <gałąź>`	`git branch`

Git/Subversion c.d.

- trunk = master
- lokalne repozytorium jest gałęzią
- git clone
  - klonuje zdalne repozytorium
  - Wyjątek: zdalny master = lokalny origin
  - ~ svn checkout
- git pull
  - Pobiera uaktualnienia ze zdalnego repozytorium
  - ~ svn update
- git push
  - Wysyła obiekty do zdalnego repozytorium
  - ~ svn commit