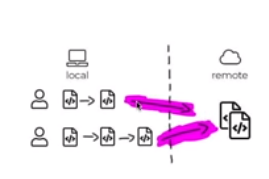
GitHub, GitLab, BitBucket = zdalne repozytoria plikow (remote).

Git umozliwia prace w trybie "offline".



Branching - rozgalezianie, czyli tworzenie nowych funkcji programu niezaleznie od siebie. Np. Na glownej galezi (master) utrzymujemy stabilna wersje aplikacji, natomiast zmiany prowadzimy na branchu pochodnym.



Remote repository (zdalne repozytorium) - pelni role kopii zapasowej projektu oraz stanowi podstawe pracy zespolowej. Oprocz tego kazdy pracownik posiada swoje lokalne repozytorium na ktorym pracuje. Git jest systemem rozporoszonym, co oznacza, ze w kazdej chwili mozemy przelaczac sie pomiedzy repozytoriami.

**Zapisywanie zmian:**

* Detekcja zmodyfikowanych plikow:

Machine generated alternative text:
zmodyf i kowa ny 
(modif ied) 
TRZY STANY PLIKOw 
Sledzony 
(staged) 
zatwierdzony 
.com itted) 

**Jak dziala to w praktyce?**

* <git checkout> resetuje caly proces

Machine generated alternative text:
katalog roboczy 
(working directory) 
git add. 
LOKALNE REPOZYTORIUM 
przechowalnia 
(stage area) 
git commit 
repozytorium 
(.git folder) 
git checkout 

**LOKALNE ZMIANY: korzystanie bez remote repository**

* git init   //inicjalizacja repo
* git status    //sprawdzenie aktualnego statusu
* git add .    //. -> dodaje wszystkie pliki do repo
* git status
* git commit -m "moj komentarz"     //zapisanie zmian do repo

**Przestrzeń robocza i stage:**

* Aktywnosc w naszym gicie (stan plikow)

* git clean -idf  //czysci pliki I katalogi, ktore nie sa sledzone - nie ma trackingu, pliki, ktore nigdy wczesniej nie zostaly dodane do repozytorium, dodajac 'd' usuwamy rowniez katalogi, 'i' - zapyta co chcemy usunac po id
* git reset   //odwrotnosc polecenia git add (usuwa pliki z kolejki oczekiwania)
* git rm    //usuwanie z repo (to dzialanie moze byc odwrocne za pomoca 'git reset'

Machine generated alternative text:
git clean 
usuwa pliki. ktOre nie zostaty dodane do indeksu Gita. 
git reset 
Przenosi pliki ze stage do working directory. Petni odwrotnq role ni2 git add 
git rm / git mv 
Usuwa / przenosi pliki w repozytorium. DziatajQ tak samo jak rm i rnv, i jednoczeSnie dodajQ zmiany na stage. 
git checkout 
Przywraca Stan pliku z indeksu gita. W praktyce przywraca wprowadzone zmiany. 
Mote bye v.ykorzystane tylko wtedy. gdy plik istnieje jui w indeksie. 

**PRZYWRACANIE ZMIAN:**

Opcja 1: Mozemy sprawdzic historie naszych commitow (tan odlaczanego 'head', wszystkie zmiany jakie wprowadzamy zostana usuniete)

* git log --oneline    //wyswietli cala historie commitow po 1linijce **+ nie modyfikuje historii commitow**

Aby wrocic do biezacej wersji wpisujemy

* git checkout master

Opcja 2: git revert: usuwa zmiany I tworzy nowy commit, nie modysikuje historii commitow.

* git revert 7ffg5     //**+ nie modyfikuje historii commitow**

Opcja 3: git reset: modyfikuje cala historie commitow (od punktu w czasie), dlatego nie powinnismy nanosic takich zmian po wykonaniu pusha (!)

* git  reset ff4ed    /**MODYFIKUJE historie commitow**

Machine generated alternative text:
git checkout 
Checkout to operacja polegajqca na przesunieciu wska2nika HEAD na wskazany commit W praktyce w katalogu roboczym 
zostaJQ wczytane pliki w wersji z wybranego commit•u, git checkout moie bye uZyte na poziomie commit•u lub pliku. 
W tym drugim przypadku zostanie wczytana wersja pliku z wybranego commit'u (domySlnie z HEAD) 
git revert 
Revert to operacja polegajqca na odwröceniu zmian z wybranego commit•u i zapisaniu ich jako nowy commit. 
Polecenie mote bye uruchomione vMacznie na poziomie commit•u. Z pomocq git revert m02emy bezpiecznie 
przywracaC zmiany ktOre znajdujQ sie ju2 w repozytorium publicznym. 
git reset 
Przyvvraca zmiany w repozytorium do wskazanego punktu z historii zmian, 
Waine: Naleiy zachowae szczegölnq ostroinoge podczas przywracania zmian z pomocq tego polecenia. 
PrzywrOcenie zmian oznacza modyfikacje historii commit'öw. Z tego powodu nie zaleca sie stosowania tego polecenia 
w przypadku commitOw wystanych do zdalnego repozytorium. 

**PRZEGLADANIE HISTORII W REPO:**

* git log  --oneline
* git log  --author="Adam"
* git log  --grep="''view' w komentarzu commita"
* git log  --oneline -5 //5 ostatnich w oneline
* git log  --oneline --index.html
* git log  --oneline --path --index.html   //wyswietli nam szczegoly w pliku (jakie byly zmiany)
* git log  --format   //mozliwosc wlasnego formatowania wyjscia loga
* git shortlog  //wyswietla info, kto aktualnie pracuje na jakim pliku
* git log --all --decorate --oneline --graph

**JAK PISAC KOMENTARZE?**

Machine generated alternative text:
7 zasad 
Praktyczny kurs GIT 
Dobre praktyki pisania komentarzy w repozytorium 
Podziel komentarz na tytut oraz treSC. oddzielajqc je pustym wierszem 
Ogranicz dtugoSC tytutu komentarza do 50 znaköw 
• Tytut rozpocznij Wiellq literq 
• Nie dodawaj kropki na kohcu tytutu 
IJ2ywaj trybu rozkazujqcego w treSci tytutu. np. Add e-mail field to login form zamiast New field in login form 
Ogranicz dtugoSC wiersza treSci do 72 znaköw 
Piszqc trest komentarza odpowiadaj na pytania co i dlaczego, Na pytanie jak m02na odpowiedziee przegl@dajqc 
wprowadzone zmiany. 

**STOS:**

Git stash wrzuca pliki na tymczasowy stos, aby umozliwic przelaczenie sie pomiedzy branchami. Odkladamy zmiany "na bok"

* git stash   //dolozenie na stos
* git checkout **<<branch>>** //teraz mozna bez problemow przepiac sie na innego brancha
* git stash pop   //wracamy do uprzedniego stanu przed wrzuceniem plikow na stos, czyli usuwamy ze stosu
* git stash list   //lista plikow-zmian na stosie

**BRANCH:**

Git trzyma wszystkie zmiany w postaci snapshotow, ktore tworzone sa podczas tworzenia commitow (stad tez hash commita/snapshota). Nie tworzymy kopii plikow, tylko wskazniki HEAD zmieni pozycje na wskazany przez nas hash snapshota. Idealny stan: wersja stabilna softu jest na <<master>>, my dewekopujemy na <<branch2>>, po akceptacji robimy merge z <<branch2>> do <<master>>.

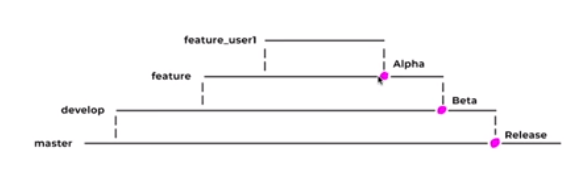
Machine generated alternative text:
HEAD 

* git branch <<dev2>>  //dodajemy brancha
* git checkout <<dev2>>    //przepinamy sie z mastera na inny branch (pamietaj o patencie ze stosem!)

**MERGOWANIE BRANCHY (Z dev2 DO master):**

* Przelaczamy sie na mastera
* git checkout <<master>>
* git merge <<dev2>>
* git log --oneline
* git log --all --decorate --oneline --graph
* git branch -D <<dev2>>    //usniecie niepotrzebnego brancha dev2

**JAK USTALAC BRANCHE?**



Machine generated alternative text:
Master branch 
Gtöwna gatQ2 projektu wykorzystyvvana do publikacji najnowszej wersji projektu na serwerze produkcyjnym. 
Master powinien zawsze zawierat stabilna wersje projektu. Moiliwoge wprowadzania zmian powinna bye ograniczona. 
Dev branch 
GatQ2 na ktÖrej znajduje sie wersja testowa projektu (alpha/beta), zwykle potqczona z serwerem testowym. 
Feature branch 
Na tej gaf€zi prowadzone SQ prace nad nowymi funkcjami. PO ukorhczeniu prac. zmiany trafiaja na branch dev 
User branch 
Gatezie u2ytkowniköw sfu2Q indywidualnej pracy czfonka zespofu nad zmianami. Ukonczone zmiany trafiaj@ na branch feature. 
Test / Bugfix branch 
W szczegOlnych przypadkach. konieczne jest szybkie wprowadzenie zmian. W tej sytuacji wykorzystywane sq branch'e typu bugfix. 
Przygotowane tutaj zmiany najczqSciej trafiajq bezpoSrednio na branch dev. 

**ZDALNE REPOZYTORIUM I FORK:**

* "origin" to alias zdalnego repozytorium (np. Gitlab/github) - repo zdalne moze byc rowniez na naszym komputerze, ale w innej lokalizacji.
* git push
* git pull
* git log --all --decorate --oneline --graph

Machine generated alternative text:
git remote add origin https://github.com/iceener/remote—repository.git 
git push —u origin master 

**FORK:** praca z repo zdalnym do ktorego nie mamy dostepu, laczymy sie do kopii

Fork wprowadza pewien schemat pracy & **PULL REQUEST** (z Forka to glownego mastera).

* Wlasciciel glownego repo
* Nowa osoba kopiuje (robi Fork) glownego repo
* Nowa osoba powinna stworzyc osobny branch, jesli chce wrzucic nowe elementy: **git branch feature**
* Wlasciciel glownego repo otrzyma Pull Request z opcja ACCEPT/DECLINE
* Po skonczonej pracy na nowo utworzonym branchu w forku, mozemy go posprzatac: **git branch -D feature & git push origin --delete feature**

**ROZWIAZYWANIE KONFLIKTOW & MERGE:**

* Recznie
* Automatycznie (z pomoca edytora)

Machine generated alternative text:
Merge conflict 
Rozwiqzywanie konfliktöw 
orancnF1 cornmlt: ZOCL3bS 
html > 
< h >overment. come / h 1 > 
< / body> 
/html> 
html> 
HEAD 
branch#2 
/ html > 
.html> 
< body > 
h 2 »overment< / h 2 > 
< / body > 
•/html> 

Lub przypadek drugi - w jednym branchu usuwamy plik, a w innym modyfikujemy:

* git add <<plik>>
* git rm  <<plik>>

Machine generated alternative text:
Deleted file conflict 
Rozwiqzywanie konfliktÖw 
-file deleted in HEAD and 
modified in [file), Version [file) of 
file left in tree.- 
zmodyfikowany 

Lub blad mergowania w PULL REQUEST:

Machine generated alternative text:
Pull request 
Rozwiqzywanie konfliktöw 
o 
Master branch 
[web editor] 
Pull request 
Feature branch 
[command linel 
1. pull remote branch 
2. fix conflict 
3. merge & push 

**GIT REBASE:**

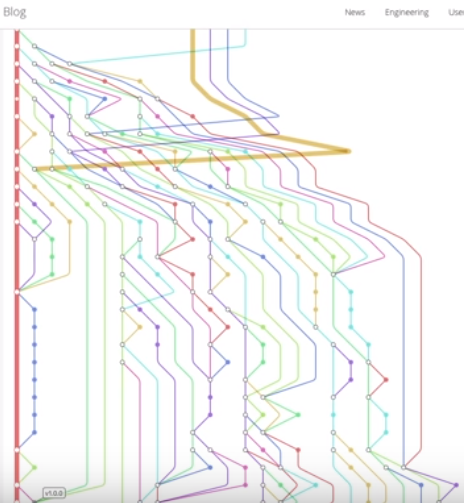
* git:(feature) git rebase master //uwaga, zmieniamy historie commitow!
* git:(feature) git rebase --skip //cofniecie zmian, moze byc problematyczne

* **UWAGA:** git rebase modyfikuje historie commitow! Mozna cofnac zmiany, ale jest to bardzo trudne. Nigdy nie robimy git rebase na remote (gitlab) bo mozemy nadpisac zmiany innych developerow!

Troche inny sposob mergowania repozytoriow - ktory zmienia historie commitow (a wraz znim polaczonych branchy). Commity z wczesniejszych branchy sa usuniete a ich zawartosc zostaje skopiowana do nowych commitow (nie przenosimy, tylko kopiujemy!).

W malych projektach zasadniczo wszystko ogarneimy mergowaniem repozytoriow, natomiast problem pojawia sie wtedy, kiedy nasz projekt jest bardzo rozbudowany I tworzozne jest wiele osobnych branchy przez developerow - przyklad (latwo stracic kontrole nad projektem):

Widok branchy (timeline) duzego projektu:



**REBASE - KROK #1** (branch **feature** zostal utworzony na podstawie kopii commita **9bbc4f0**)

Machine generated alternative text:
FEATURE 
000183b 

**REBASE KROK #2** (wykonanie rebase, czyli usniecie i skopiowanie brancha **feature** oraz ustawienie go (podlinkowanie) do najnowszego commita: **88a183b**).

Machine generated alternative text:
FEATuRE 

**REBASE KROK #3 (**finalnie po wykonaniu rebase mamy nadal pelna historie zmian wraz z ustawieniem mastera na najwoszym commicie: **f03ed04).**

Machine generated alternative text:
FEATuQE 
'01be9S 

**GIT - TAGI:**

Zakladki repozytorium. Kazdy commit ma swoj hash, tagami oznaczamy id commita np. taki, na ktorym zostaly zaoknczone pewne wazne prace z punktu widzenia projektu.

* git tag v1.0   //dodaje nowy tag dla aktualnego commita
* git tag   //listuje wszystkie tagi
* git tag -d v1.0   //usuwa tag
* git tag v.10 **2c6237d3** -a -m "Initial version 1.0"   //podpiecie konkretnego tagu pod commit **2c6237d3** wraz z informacja o autorze -a oraz wiadomoscia -m
* git push tags  //dopiero to polecenie zakomituje zmiany z lokalnego repo do zdalnego
* git push origin -d v1.0   //usuniecie tagu z repozytorium zdalnego



Na podstawie tagow mozna stworzyc take specjalny typ tagu: **RELEASE:**

**PRZYDATNA GRAFIKA OPISUJACA ACTION FLOW GIT:**

Machine generated alternative text:
index 
local 
repository 
remote 
repository 