

Git

Git

System kontroli wersji - program zapisujący zmiany zachodzące w plikach (wersje), dzięki czemu możemy przejrzeć ich historię i jak trzeba, cofnąć zmiany. Wszystkie te zmiany są zapisywane w repozytorium projektu.

Git to najpopularniejszy taki system. Służy do zarządzania kodem źródłowym aplikacji. Te zmiany do Commity, każda taka zmiana zawiera informację o tym, kto ją wprowadził, kiedy i co zrobił.

Drugą dobrą funkcją Gita jest rozgałęzianie (branching), umożliwia to rozwój nowych funkcji aplikacji niezależnie od siebie. W trakcie trwania rozgałęziania, na głównej gałęzi dalej może być produkcyjna wersja aplikacji. Główna gałąź to master branch.

Git umożliwia przechowywanie kodu w chmurze. Jest to zdalne repozytorium. Każdy programista ma lokalne repozytorium (u siebie na kompie) i można jest zawsze połączyć ze zdalnym.

Rozpoczęcie pracy i zapisywanie zmian

Lokalne repozytorium.

Są 3 główne lokalizacje w Lokalnym repozytorium:

- katalog roboczy (working directory) - są tu główne pliki związane z projektem
- przechowalnia (stage area) - tu trafiają pliki, które po zatwierdzeniu mają trafić do repozytorium
- repozytorium (.git folder) - folder gita, w którym przechowywane są informacje o naszym projekcie.

Najpierw zmiany robimy w katalogu roboczym. Potem poleceniem `git add` . dodajemy zmiany do przechowalni. Następnym krokiem jest polecenie `git commit`, które zapisuje zmiany w repozytorium. Na koniec następuje zresetowanie cyklu (`git checkout`) i cykl może się zacząć od początku.

Ten proces tworzy nam 3 rodzaje plików: zmodyfikowany (modified) -> `git add` -> śledzony (staged) -> `git commit` -> zatwierdzony (committed).

Aby zacząć używać gita w terminalu trzeba zainicjować w danym folderze repozytorium:

`git init` - inicjacja repo, teraz git będzie śledził już zmiany

`git status` - wyświetlenie statusu repozytorium. Po stworzeniu mogą być już w repozytorium jakieś pliki, które nie są związane z naszym projektem i chcemy je ignorować (są to często pliki związane ze środowiskiem uruchomienia aplikacji), w tym celu trzeba je dodać do pliku `.gitignore` (często go samemu trzeba stworzyć na początku i podawać tam pliki ze ścieżkami).

`git add plik/folder/.`(kropka dodaje wszystko) - dodajemy zmodyfikowane pliki do przechowalni. Teraz `git status` pokaże nam pliki, które oczekują na dodanie do repo.

git commit -m "komentarz" - zapisanie plików do repo. Każdy commit powinien zawierać komentarz opisujący zmiany. Komentarz dodajemy flagą -m. Teraz git status powie, że nie mamy nic do zapisywania.

Jeśli mamy plik oczekujący na commita i w tym momencie go zmodyfikujemy jeszcze raz to git status nam pokaże, że mamy dwie wersje tego pliku (jedną w przechowalni czekającą na zapisanie, a drugą w katalogu roboczym czekającą na dodanie). Teraz dobrze jest zrobić jeszcze raz commita i będziemy mieć już jedną ostateczną wersję tego pliku czekającą na zapisanie do repozytorium.

Przestrzeń robocza i stage

git clean - usuwanie nieśledzonych plików i katalogów (takich, które nigdy nie były zapisane w repozytorium i nie zostały dodane do indexu gita) z przestrzeni roboczej.

git clean -nd - uruchomienie trybu testowego usuwania plików, jako rezultat wyświetli nam się lista plików i katalogów (flaga d), które normalny clean by usunął.

git clean -idf - flaga i to tryb interaktywny, git będzie mnie pytał, co chce usuwać, d to uwzględnienie katalogów, f to faktyczne usuwanie. W trybie interaktywnym pod opcją 4 możemy wybierać, które pliki chcemy usunąć.

git clean -xn - usuwa pliki, które są dodane do ignorowania.

git reset - usuwanie pliku z kolejki oczekiwania i powrót do przestrzeni roboczej, aby wprowadzić kolejne zmiany. Jest to odwrotna rola do git add.

git checkout -- plik - w przestrzeni roboczej usunięcie wszystkich zmian dla danego pliku. Powróci on do stanu z ostatniego zapisu. Jest to przywracanie stanu pliku z indexu gita.

git rm plik/katalog - usuwanie plików bezpośrednio z repozytorium, git status pokaże teraz usunięte pliki. Teraz trzeba zrobić jeszcze git commit i plik zostanie usunięty.

Aby odwrócić usuwanie, jeszcze przed commitem możemy wykonać polecenia git reset i git checkout plik/katalog. To przywróci nam ostatnią wersję pliku do repozytorium.

git mv - przenoszenie plików w repozytorium z miejsca na miejsce. Po każdym przeniesieniu i usunięciu (git rm i git mv) pliki zostają przeniesione do stage.

Przywracanie zmian w Git

Przywracanie zmian polega na 'cofaniu się w czasie'. Służą do tego 3 polecenia:

git checkout - pozwala na przeniesienie danej wersji projektu do katalogu roboczego. Polecenie może być użyte na poziomie commitu i pliku (możemy przywrócić cały projekt lub tylko plik).

git log --oneline - wyświetlenie historii w pełnej wersji. Mamy tutaj po kolei wszystkie commity z ich ID oraz komentarzem:

```
6d15e93 (HEAD -> master) Added console log to app.js
6e38f3a Created JS files
925760e Added header on home page
8918c80 Initialized repository
(END)
```

Teraz możemy wrócić do danej wersji projektu poleceniem checkout:

`git checkout ID_commitu`

Po tym poleceniu wyskoczy informacja, że jesteśmy w trybie "Detached head", oznacza to, że możemy przeglądać projekt i testowo wprowadzać zmiany, ale nie zostaną one zapisane. Jeśli jednak chcemy to zapisać to musimy utworzyć nową gałąź:

`git checkout -b nowa_gałąź`

`git checkout master` - powrót do bieżącej wersji katalogu roboczego z przed pierwszego checkouta.

Wszystkie te operacje zmieniają pozycje wskaźnika HEAD (screenshot). Wskazuje nam on, jaką aktualnie wersję projektu (jakiego commita) widzimy.

`git checkout ID_commitu ścieżka/plik` - przywrócenie tylko jednego pliku z danego commitu. Możemy z niego teraz zrobić nowego commita i go zapisać jako najnowszy.

`git revert` - odwrócenie zmian z wybranego commitu i zapisanie ich jako nowy commit. Możemy za pomocą tego polecenia przywracać zmiany, znajdujące się na publicznym repo, bo `git revert` nie usuwa i nie modyfikuje historii zmian (`git log`).

`git revert ID_commitu` - odwrócenie zmian z danego commita. To odwrócenie zostanie zapisane jako najnowszy commit, a stan z przed uruchomienia komendy, będzie poprzednią wersją projektu. Polecenie to odwraca zmiany tylko z wybranego commitu, jeśli np będzie to odwrócenie commitu nr 4, a teraz jesteśmy w 7 to zmiany z 5 i 6 będą nadal na miejscu.

`git reset` - przywracanie zmian w repozytorium do danego punktu w historii zmian. Polecenie to modyfikuje historię Gita, więc nie powinno się z tego korzystać w publicznym repozytorium. Polecenie to powinno być używane na commitach, które nie są opublikowane poleceniem `git push`.

Polecenie działa w 3 trybach: `--mixed` (wszystkie zmiany wprowadzone do commitu, do którego się cofamy zostaną zapisane w katalogu roboczym. Czyli wszystkie nowsze commity zostaną usunięte z historii, ale ich zmiany będą w katalogu roboczym), `--soft` (to samo co `mixed`, ale zmiany z nowszych commitów zostaną dodane od razu na stage, a nie do katalogu roboczego, więc można od razu z nich zrobić commit), `--hard` (zmiany zostaną całkowicie usunięte wraz z commitami)

`git reset --tryb ID_commitu`

Przeglądanie historii - `git log`

`git log` - wyświetlenie informacji o tym, kto i kiedy wprowadził jaki commit. Polecenie `git log` ma wiele przydatnych opcji, które ułatwiają przeglądanie historii.

—`oneline` - wyświetlenie skondensowanej historii z ID_commitów i komentarzami

—`author="username"` - wyświetlenie commitów zrobionych przez daną osobę.

Należy pamiętać, że opcje możemy łączyć np. `git log --oneline --author="username"`.

—`grep="tekst"` - wyświetlenie commitów, które mają szukany tekst w komentarzu

`git log --oneline --patch -- ścieżka/plik` - wyświetlenie commitów, w których był modyfikowany dany plik ze szczegółową informacją o zmianach (flaga `patch`. Inną opcją jest flaga `--summary` wyświetlająca tę informację w sposób skrócony lub `--stat` wyświetlająca statystyki zmian plików).

—`format` pozwala samemu modyfikować to, jak ma wyglądać historia zmian np: `git log --format="%h %an %s %cr"` - wyświetlenie skróconego hasha commitu (ID), autora, komentarz i informację o tym kiedy został dodany.

git shortlog - pokazanie historii zmian z podziałem na użytkowników (dobrze przedstawia, kto pracuje nad jakimi plikami)

Komentarze w Gicie

Dobre praktyki:

- Podzielenie komentarza na tytuł i treść, oddzielając pustym wierszem. W tym przypadku dobrze jest robiąc commita nie dodawać flagi -m i wtedy Git uruchomi drugie okno do wpisania komentarza, wtedy możemy łatwo dodawać nowe linie i puste wiersze.
- Tytuł powinien być zwięzły (maks 50 znaków)
- Dobrze jest pisać komentarze w trybie rozkazującym ze szczegółami, bo łatwiej zrozumieć je np w przypadku odwracania. Np "Add email field to login form"
- Należy raczej pisać co zostało zrobione i dlaczego, a niekoniecznie jak, bo to zobaczymy otwierając commita.

Git stash

Jeśli robimy modyfikację jakiegoś pliku, ale okazuje się, że robimy ją na masterze, a chcemy zrobić na innym branchu, to nie możemy od razu przełączyć się git checkoutem na inny branch, bo trzeba będzie najpierw zapisać zmiany. Inną opcją w tej sytuacji jest polecenie git stash, które wysyła zmiany na tzw stos, jest to kolejka, która może być dodana do każdego brancha. Teraz mogę zmienić branch bez problemu i poleceniem git statsh pop wczytać zmiany ze stosu do przestrzeni roboczej innego brancha. Dzięki temu zmiany robione najpierw na masterze będą zrobione tylko w innym branchu.

git stash push -m "tekst" - pozwala dodać komentarz do zmian wprowadzanych do stosu.

git stash list - wyświetlenie wszystkich zmian czekających w stosie wraz z komentarzami, jest tu też identyfikator zmian.

Ważną rzeczą jest to, że do stosu możemy domyślnie dodawać tylko śledzone już pliki (z nowym to nie zadziała). Jeśli chcemy to obejść, trzeba wysłać plik do stage poleceniem git add, lub dodać do git stash flagę -u:

git stash push -m "tekst" -u

Jeśli mamy kilka czekających zmian w stosie to możemy je przywracać pojedynczo lub wszystkie na raz.

git stash apply ID_zmiany - zmiana będzie przywrócona do katalogu roboczego, ale zostanie dodatkowo na stosie jej kopia.

git stash pop ID_zmiany - zmiana będzie przywrócona do katalogu roboczego i usunięta ze stosu. Jak nie podamy ID_zmiany to domyślnie zostaną przywrócone ostatnie wprowadzone zmiany.

git stash drop ID_zmiany - usunięcie danej zmiany ze stosu całkowicie

git stash clear - wyczyszczenie całego stosu, wszystkie zmiany zostaną całkowicie usunięte.

git stash branch nazwa_brancha - tworzenie nowego brancha ze zmian będących na stosie

Branch

Git przechowuje zmiany w postaci snapshotów (nowy tworzony jest wraz z kolejnym commitem). Każdy snapshot ma swój unikatowy hash (id), które pozwala go zidentyfikować. Branch to po prostu rodzaj wskaźnika wskazujący na konkretny snapshot. Utworzenie nowego

brancha to stworzenie nowego wskaźnika (nie jest tu tworzona żadna kopia i nie ma dodatkowych użytych zasobów). O tym, która wersja projektu jest aktualnie wczytana do katalogu roboczego decyduje dodatkowy wskaźnik HEAD. Jeśli w danym momencie zmienimy branch, to zmieni się położenie wskaźnika HEAD, ale dalej będziemy na tej samej wersji projektu. Dopiero jak zacznę wprowadzać zmiany i zapiszę je będąc dalej np w branchu develop to powstanie unikalny snapshot, jeśli wrócę do branchu master, to otrzymam wersję projektu bez tych ostatnich zmian. Najlepiej w branchu master mieć wersję produkcyjną projektu, a np na branchu develop robić testy. Jak będziemy chcieli wprowadzić zmiany z develop na master to trzeba będzie połączyć ze sobą te dwa branche.

git branch nazwa_brancha - tworzenie nowego brancha

git branch - podejrzanie wszystkich dostępnych branchy projektu

git checkout nazwa_brancha - przełączenie się w danym momencie na konkretnego brancha. Jeśli chcemy połączyć ze sobą branch develop z masterem to musimy być na masterze i w poleceniu przekazać nazwę brancha, który chcemy dodać. Dzięki temu finalnym wynikiem będzie master:

git merge nazwa_brancha_dodawanego - w tym momencie łączenia branchy powstanie nam też nowy commit.

Po połączeniu dwóch branchy do mastera najlepiej jest już usunąć ten niepotrzebny branch.

git branch -D nazwa_brancha - usuwanie danego brancha

Dobre praktyki:

- master branch - główny branch projektu, tu powinna być jego stabilna wersja, zwykle połączony ze środowiskiem produkcyjnym

- dobrze mieć branch developerski wykorzystywany do testowania naszej aplikacji, zwykle połączony ze środowiskiem testowym

- feature branch - można stworzyć też branch do pracy nad nowymi funkcjami

- user branch - gałęzie każdego członka zespołu wykorzystywane do indywidualnej pracy

Zdalne repozytorium i fork

Repozytorium lokalne to takie, które mamy na prywatnym komputerze, a repozytorium zdalne to takie, które mamy na jakimś serwerze, czy innym komputerze np. publiczny GitHub. Przy łączeniu repozytorium zdalnego z lokalnym, lokalne otrzyma adres tzw origin, który będzie wskazywał na konkretne zdalne repozytorium.

Łączenie repozytorium lokalnego z Githubem wykonujemy poleceniem:

git remote add origin... (jest to do skopiowania na GitHubie).

git push - wysyłanie zmian z lokalnego na zdalne repozytorium. W tym przypadku dobrze jest dodawać flagę u, dzięki której możemy wskazać zdalne repozytorium oraz branch, na który wysyłamy zmiany np:

git push -u origin master - po takim uruchomieniu komendy, branch z którego to uruchamiamy (tu master) będzie zapamiętany jako domyślny i następnym razem wystarczy zrobić tylko git push. Tutaj precyzujemy także, że zmiany wyślemy do brancha master na zdalnym repozytorium.

git fetch - pobranie zmian ze zdalnego repozytorium na lokalne. Jeśli ktoś inny zrobił jakieś commity, to zostaną one też pobrane na nasze lokalne repozytorium.

Po ich pobraniu wskaźnik HEAD będzie jednak cały czas ustawiony na nasz lokalny master, a nie na ten nowy pobrany, aby to uaktualnić i przesunąć HEAD na nowy master robimy:

git merge origin/master - synchronizacja zdalnego i lokalnego repozytorium. Dobrą praktyką jest przed rozpoczęciem pracy w lokalnym repozytorium, upewnić się, że na zdalnym nie ma jakichś nowych zmian. Należy pamiętać, że origin to alias naszego repo, więc zamiast niego może być inna nazwa jeśli chcemy synchronizować jakieś inne repozytorium zdalne z naszym lokalnym.

Fork to kopiowanie zdalnego repozytorium, do którego nie mamy dostępu (nie możemy tam robić commitów, więc możemy je skopiować do siebie i połączyć z naszym lokalnym, aby móc na nim pracować).

Często jest tak, że nie możemy np modyfikować repozytorium master na produkcji, więc kopiujemy je do siebie. Kopiowanie odbywa się przy użyciu polecenia:

```
git clone link_do_repo
```

Jeśli chcemy sprecyzować w linku jakim kontem łączymy się do repozytorium to w linku na początku podajemy: <https://username@github.com/.....>

Tam modyfikujemy, potem wysyłamy zmiany na nasze prywatne zdalne repo poleceniem git push i na koniec wykonujemy pull request do głównego repozytorium. Tu precyzujemy na jaki branch w głównym repo chcemy wysłać zmiany. Jeśli zostanie on zaakceptowany, to trzeba zmiany wejść w życie na prodzie. Po akceptacji możemy pousuwać nieużywane branche na naszym zdalnym i lokalnym repozytorium.

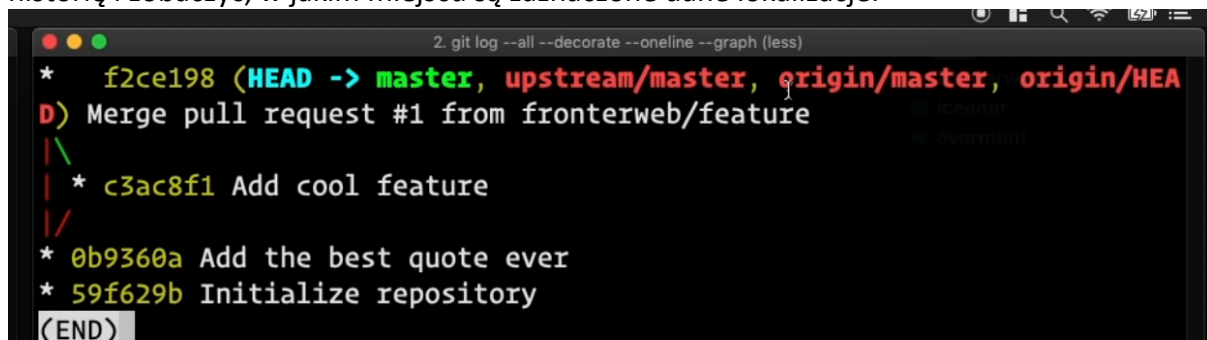
Jeśli chcemy podglądać zdalnie główne repozytorium to trzeba też je dodać do swojego zdalnego (można mieć spokojnie wiele dodanych).

```
git remote add alias_repo link_do_repo
```

```
git remote - wyświetla wszystkie połączone zdalne repozytoria.
```

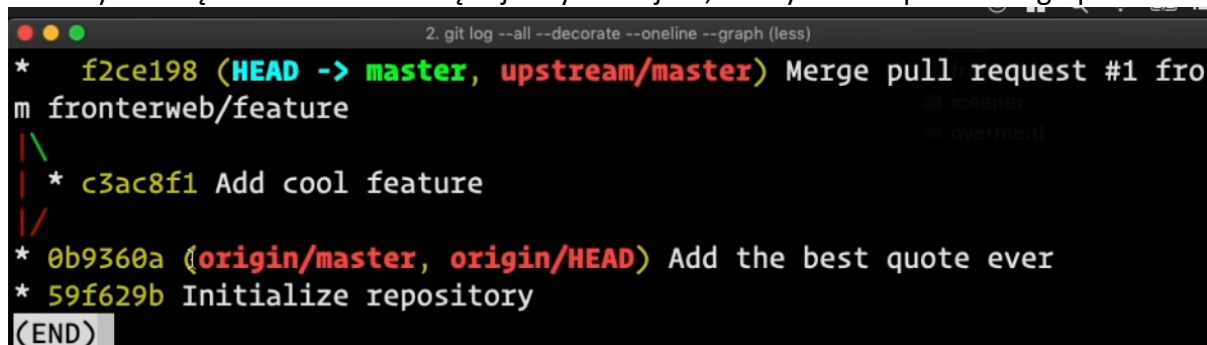
```
git fetch alias_repo - pobieranie zmian z danego repozytorium
```

Aby sprawdzić jakie wersje projektu są na zdalnych i lokalnym repozytorium trzeba wejść w historię i zobaczyć, w jakim miejscu są zaznaczone dane lokalizacje:



```
2. git log --all --decorate --oneline --graph (less)
* f2ce198 (HEAD -> master, upstream/master, origin/master, origin/HEAD) Merge pull request #1 from fronterweb/feature
|
| |
| * c3ac8f1 Add cool feature
|/
* 0b9360a Add the best quote ever
* 59f629b Initialize repository
(END)
```

Tu wszystkie są zaktualizowane i są w jednym miejscu, bo wykonano polecenie git push.



```
2. git log --all --decorate --oneline --graph (less)
* f2ce198 (HEAD -> master, upstream/master) Merge pull request #1 from
m fronterweb/feature
|
| |
| * c3ac8f1 Add cool feature
|/
* 0b9360a (origin/master, origin/HEAD) Add the best quote ever
* 59f629b Initialize repository
(END)
```

Na tym zdjęciu są jeszcze rozbieżności i nie ma pełnej synchronizacji. Aby wszyscy byli na bieżąco ze zmianami, właściciel głównego repozytorium musi je do siebie pobrać lokalnie poleceniem `git pull`.

Rozwiązywanie konfliktów

Występują one czasem w trakcie łączenia zmian z różnych branchy (merge). Jak zrobimy zmiany na jednym branchu i chcemy to połączyć z drugim to git porównuje zawartość modyfikowanego pliku algorytmem i na tej podstawie tworzy się ostateczna zawartość.

Przykładowy konflikt. Gdy chcemy wprowadzić zmiany z branchu2 na branch1 w danym pliku, ale przed zrobieniem tego modyfikujemy jeszcze raz ten plik na branchu1. Dostaniemy taką odpowiedź:

```
<!-- branch#1 -->
<html>
  <body>
<<<<<<< HEAD
    <h1>overment.com</h1>
=====
    <h2>overment</h2>
>>>>>>> branch#2
  </body>
</html>
```

Aby rozwiązać ten konflikt musimy ręcznie wprowadzić zmiany. Jak chcemy zostawić pierwszą wersję to usuwamy "`<<<<<<< HEAD`" oraz wszystko od "`===== ... >>>>>>> branch#2`". Analogicznie możemy zostawić drugie zmiany. Inną opcją jest też usunięcie wszystkiego i samemu wprowadzenie czegoś nowego. Potem zmiany zatwierdzamy `git add` i tworzymy commita.

Drugi rodzaj konfliktu to jeśli na jednym branchu usuwamy dany plik, a na drugim go modyfikujemy. Ponownie dostaniemy zapytanie o to, czy usuwać ten plik, czy zostawić modyfikację. Jak zrobimy `git status` to zobaczymy dostępne dla nas komendy. Jeśli chcemy zachować plik to trzeba użyć polecenia `git add plik`, a jeśli usunąć to `git rm plik`. Po wykonaniu `git add`, trzeba będzie jeszcze zrobić commit i merge, a jeśli usuniemy plik, to od razu zrobi się merge.

Jeśli będziemy chcieli zaakceptować pull request, który zawiera konflikty (np. właściciel głównego repo zmodyfikował dany plik i drugi użytkownik też u siebie zmodyfikował ten sam plik i robi pull request), to trzeba będzie albo rozwiązać je w edytorze na GitHub, albo zrobić to przez command line. W drugim przypadku trzeba będzie pobrać do siebie branch, rozwiązać konflikty i zrobić merge. Mamy do tego podane polecenia:

`git checkout -b nowy_branch master` (flaga `b` to tworzenie nowego brancha i przełączenie się na niego)

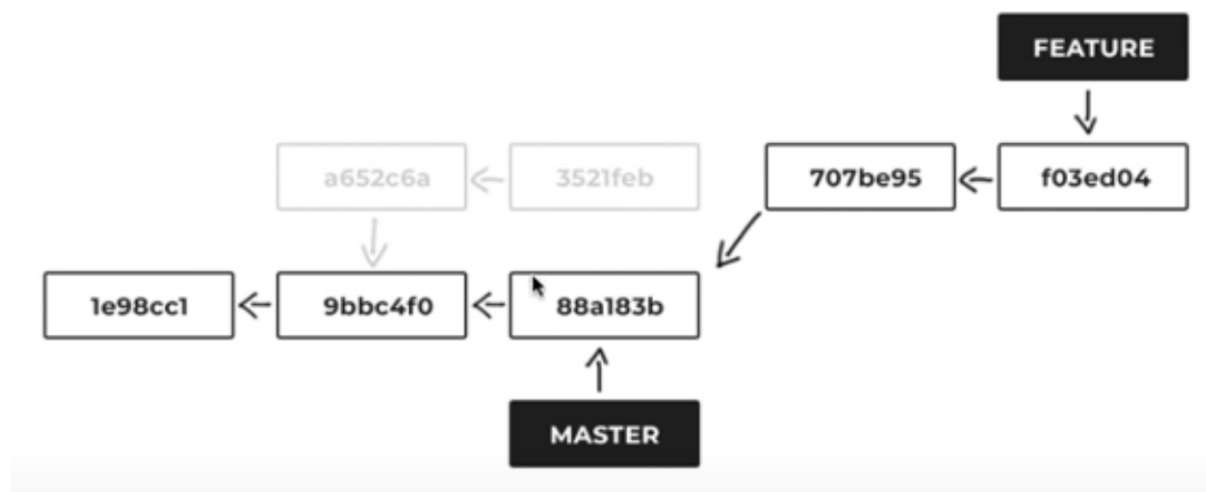
`git pull link_repo`

Teraz dostaniemy informację o konflikcie i musimy się nim zająć i ręcznie go rozwiązać. Po zapisaniu zmian robimy `git add` i `git commit`. Następnie z GitHuba bierzemy kolejne komendy:
`git checkout master` (przełączamy się na mastera)
`git merge --no-ff nowy_branch` (połączenie nowej gałęzi z masterem. Opcja `no-ff` - w momencie łączenia branchy tworzony jest nowy commit)
`git push origin master`
Teraz pull request już będzie zaakceptowany.

Rebase

Normalnie, gdy chcemy połączyć zmiany z danego brancha do głównego brancha to używamy `git merge`. Sytuacja się może utrudnić, gdy mamy duży projekt i wiele osób nad nim pracuje w tym samym czasie. W takiej sytuacji może nam się przydać `rebase`.

`Rebase` jest to proces polegający na zmianie commitu, na podstawie którego powstał dany branch.



Na rysunku wcześniej branch feature powstał w commitcie 9bbc4f0, ale już teraz master branch jest zmodyfikowany, więc poleceniem `git rebase master` przesuniemy branch feature na commit 88a183b. Poprzednie commity feature zostaną usunięte, ale ich zawartość zostanie skopiowana do nowych commitów. Dzięki temu będą w nich zapisane późniejsze zmiany na masterze. Teraz trzeba tylko przesunąć wskaźnik HEAD na najnowszy commit.

Należy pamiętać, że polecenie `git rebase` modyfikuje historię gita. Nigdy nie należy też modyfikować historii znajdującej się w zdalnym repozytorium. Przez to można niechcący nadpisywać lub modyfikować zmiany innych.

Aby wykonać rebase trzeba się przełączyć na feature branch i użyć polecenia:

```
git rebase master
```

Teraz już będzie zmodyfikowana historia gita.

Gdy chcemy zrobić rebase, ale w plikach nastąpi konflikt, to zostaniemy o tym poinformowani i dostaniemy polecenia, którymi możemy to naprawić. Po zapisaniu pliku i rozwiązaniu konfliktu robimy `git add` i następnie wykonujemy polecenie:

```
git rebase --continue
```

W tej sytuacji możemy też ominąć commit, który tworzy konflikt i dokończyć rebase: `git rebase --skip`, lub anulować całe polecenie rebase: `git rebase --abort`

Po rozwiązaniu konfliktu rebase się już dokończy.

Polecenie `git rebase` możemy też uruchomić w trybie interaktywnym z flagą `-i`:

`git rebase master -i`

Tutaj sami decydujemy o tym co ma się stać z danym commitem. Możemy użyć np polecenia `git squash`, które szybko dokończy cały rebase i zrobi ostateczny merge.

Tagi

Tagi to zakładki, które pozwalają nam odnaleźć ważny commit np możemy ich użyć, gdy chcemy oznaczyć, że na danym commicie została ukończona dana wersja naszej aplikacji. Dzięki temu łatwiej będzie wrócić do konkretnych wersji.

`git tag nazwa_tagu` - nadanie taga obecnej wersji projektu

`git tag` - wyświetlenie tagów

`git show nazwa_tagu` - wyświetlenie informacji o commicie, do którego jest dodany konkretny tag.

`git tag -d nazwa_tagu` - usuwanie taga

`git tag nazwa_tagu -a -m "tekst"` - tworzenie taga z informacją o autorze (flaga `a`) i z dodanym komentarzem (flaga `m`).

`git tag nazwa_tagu hash_commitu` - przypisanie taga do konkretnego commitu

Tagi podobnie jak commity trzeba wysłać do zdalnego repozytorium poleceniem:

`git push --tags`

`git push nazwa_repo nazwa_tagu` - wysłanie do zdalnego repozytorium tylko jednego taga

`git push nazwa_repo -d nazwa_tagu` - usuwanie danego taga ze zdalnego repozytorium

W GitHubie mamy podział na Release i Tag. Release możemy tworzyć na podstawie tagu. W trakcie jego tworzenia wybieramy tag, z którego korzystamy i możemy też dodać do niego pliki.

Jak cofać ostatnie zmiany i commity

`git commit --amend -m "komentarz"` - zmiana komentarza danego commitu

`git reset hash_poprzedniego_commitu` - najnowszy commit zostanie usunięty, ale jego zmiany zostaną przesłane do katalogu roboczego. Można teraz zrobić dodatkowe zmiany i znowu wprowadzić ten plik do repozytorium.

`git reset --hard hash_danego_commitu` - cofanie się do danego commitu, ale wszystkie zmiany z commitów, które usuwamy też będą usunięte (flaga `hard`).

`git reflog` - polecenie pokazujące jak zmieniała się pozycja wskaźnika HEAD. Dzięki temu jak trwale usunęłam dany commit ze zmianami, to mogę znaleźć jeszcze tutaj jego hash. Jeśli będę chciał jednak przywrócić usunięte dane, to mogę znowu użyć polecenia:

`git reset hash_usuniętego_commitu` - to pozwoli mi na powrót do usuniętych commitów i odzyskanie ich zmian.

Jeśli chcemy zmieniać historię w zdalnym repozytorium to nie należy robić tego przez `git reset`, ale przez `git revert`. Polecenie to tworzy nowy commit z odwróconymi zmianami, które były zrobione we wskazanym commicie.

`git revert hash_commitu` - tworzenie nowego commitu z odwrotnymi zmianami ze wskazanego commitu (czyli jak np w danym commicie dodaliśmy plik, to tutaj to polecenie go usunie).