Git lab 1

Piotr Kicki

November 2021

1 Wprowadzenie

Spodziewam sie, że każdy Was miał już do czynienia z systemem kontroli wersji. Potencjalnie wiekszość z Was zetkneła sie z systemem Git, który bedzie tematem tych zajeć. Jednakże, istnieje duża szansa, że wasza styczność z git'em była do tej pory dość ograniczona. W toku tych zajeć mam nadzieje, że uda mi sie przypomnieć Wam podstawy oraz pokazać pewne bardziej zaawansowane możliwości jakie daje Git.

2 Podstawowe komendy

Pierwszym krokiem do korzystania z git'a jest utworzenie nowego repozytorium. Repozytorium, nazywane też projektem, zazwyczaj zawiera w sobie cała aplikacje, lub jej niezależna cześć. Np. korzystajac z ROSa czesto za jedno repozytorium uznamy jedna paczke, lub zbiór paczek połaczonych tematycznie. Jako że podczas tych zajeć bedziemy korzystać z zewnetrznego hostingu systemu kontroli wersji git, koniecznym jest posiadanie konta na takim hostingu np. github, gitlab.

1. Stwórz katalog `A` i wejdź do niego. Zainicjalizuj repozytorium git'a używajac polecenia `git init`. To samo można uzyskać wykonujac komende `git init A`.

2. Sprawdź aktualny stan tego repozytorium wykonujac polecenie `git status`. Powinna pokazać sie informacja na temat aktualnego brancha (do czego przejdziemy później), oraz o braku commitów (czym zaraz sie zajmiemy) oraz braku rzeczy do commitowania.

3. Stwórz plik i ponownie wykonaj polecenie `git status`. Powinna pojawić sie lista "Untracked files". Sa to pliki, które znajduja sie w katalogu, ale nie zostały dodane do repozytorium i git nie bedzie śledził zmian w nich zachodzacych.

4. Aby dodać taki plik do repozytorium, należy wykonać polecenie `git add <NAZWA_PLIKU>`. Jeżeli jeszcze raz sprawdzimy status repozytorium zobaczymy, że plik ten został dodany do listy "Changes to be commited".

5. Aby zarejestrować wszystkie wprowadzone zmiany, musimy utworzyć commit. Commit powinien być w miare możliwości niepodzielna kontrubucja do repozytorium np. utworzeniem pojedynczej funkcji. W tym celu wywołujemy komende `git commit` a nastepnie wprowadzamy opis wprowadzonych zmian. To samo można uzyskać wykonujac komende `git commit -m "<OPIS_WPROWADZANYCH_ZMIAN>"`. Opisy zmian sa bardzo istotne, gdyż ułatwiaja nam oraz członkom zespołu orientowanie sie w tym jakie zmiany w repozytorium wprowadza ten konkretny commit.

6. Zapewne pokaże sie informacja z prośba o podanie tożsamości, uczyń to korzystajac z poleceń `git config user.name <NAZWA_UZYTKOWNIKA>` oraz `git config user.email <EMAIL_UZYTKOWNIKA>`. Te polecenia zmienia użytkownika w danym repozytorium, jednakże czesto w przypadku działania na jednej maszynie, można pokusić sie o dodanie flagi `--global` co spowoduje że dane te zaaplikuja sie do wszystkich repozytoriów.

7. Kolejne sprawdzenie statusu pokaże nam, że drzewo pracy jest czyste, co jest pozytywna informacja o tym, że wszelkie do tej pory wprowadzone zmiany sa śledzone.

8. Spróbuj teraz sam stworzyć nowy plik oraz zmodyfikować stary i zacommitować wprowadzone zmiany.

9. Ważnym elementem dla którego kochamy git'a jest możliwość przechowywania zmian na zewnetrznym serwerze, co umożliwia ewentualny zewnetrzny backup naszych poczynań oraz jest niezbedne do efektywnej współpracy w zespole. Aby poinformować świat o swoim nowym dziele (tak, mamy na myśli powyższe 2 commity) zaloguj sie na swoje konto github i stwórz nowe repozytorium/projekt.

10. `git remote add origin https://github.com/<NAZWA_UZYTKOWNIKA>/<NAZWA_REPOZYTORIUM>.git`

11. `git push --set-upstream origin master` może sie okazać że konieczny bedzie PAC

12. Ponadto zapewne zostaniecie zapytani o swój email i nazwe użytkownika. aby nie wpisywać ich ciagle, można skorzystać z polecenia `git config credential.helper store`.

13. Spróbuj stworzyć nowe zmiany, zacommitować je i wykonać polecenie `git push`, tym razem już bez żadnych argumentów.

14. Historie wprowadzanych commitów można podejrzeć poleceniem `git log`, zobaczycie tam dane autorów, treści commitów oraz daty wprowadzanych zmian.

15. Ok, czas na coś ciekawego! Powspółpracujmy sami ze soba. Aby sklonować repozytorium do nowego miejsca w systemie, aby emulować "drugiego dewelopera" należy użyć polecenia `git clone https://github.com/<NAZWA_UZYTKOWNIKA>/<NAZWA_REPOZYTORIUM>.git`. To polecenie utworzy w nowym folderze o nazwie `<NAZWA_REPOZYTORIUM>` kolna repozytorium które jest hostowane na w/w serwerze github. Jeżeli chcecie aby sklonowało sie ono do folderu o innej nazwie wystarczy na końcu dopisać te nazwe
np. `git clone https://github.com/<NAZWA_UZYTKOWNIKA>/<NAZWA_REPOZYTORIUM>.git B`.
Dalej bedziemy sie odnosić do tego repozytorium, jako repozytorium B.

16. Spróbuj stworzyć nowe zmiany w folderze A, zacommitować je i wykonać polecenie `git push`.

17. Aby być na bieżaco ze zmianami wprowadzanymi przez innych deweloperów, przed zaczeciem pracy w repozytorium B, warto wykonać komende `git pull`, dzieki niej pobierzemy wszystkie zmiany które inni deweloperzy wprowadzali.

18. Komendy tej jednak nie można wykonać jeżeli nie zacommitowaliśmy swoich zmian, a ktoś inny zmodyfikował te same miejsca w plikach co my. Spróbuj wywołać taka sytuacje, aby niemożliwym było wykonanie komendy `git pull`.

19. W takiej sytuacji mamy kilka dróg wyjścia:

- możemy usunać zmiany które wprowadziliśmy korzystajac z polecenia `git restore [--staged ]<file>` (przećwicz ten wariant, a nastepnie ponownie sprokuruj sytuacje w której nie można pociagnać zmian z serwera)
- możemy dokończyć swoja prace, zacommitować ja, a nastepnie pobrać zmiany od kolegi, co doprowadzi do merga, który omówimy za chwile (przećwicz ten wariant, a nastepnie ponownie sprokuruj sytuacje w której nie można pociagnać zmian z serwera)

20. Zapewne w tym momencie wyświetla Ci sie informacja, że nie ma merge'a bo jest konflikt. Tak sie dzieje gdy 2 osoby edytuja te same miejsca w pliku i git nie jest w stanie sam połaczyć wprowadzanych modyfikacji. Prosi on wtedy o pomoc programiste. Wejdź do edytowanego pliku w którym wystepuje konflikt. Konfilkty oznaczane sa duża liczba znaków mniejszości, równości i wiekszości wraz z oznaczeniami commitów. rozwiaż konflikt poprzez ustalenie jak ma wygladać docelowy plik i usuń te dodatkowe oznaczenia. Nastepnie dodaj połaczony (zmergeowany) plik i zacommituj go. Domyślnie treść commita, bedzie zawierała informacje o merge'u. Nastepnie wypchnij wprowadzone zmiany. Zaobserwuj, że merge to tka naprawde osobny commit. Jeżeli chcesz poznać prawde o merge'u użyj polecenia `git log --graph --decorate --oneline`, które wyświetli drzewo commitów. Zobacz, jak repozytorium rozdzieliło sie na 2 ścieżki, a nastepnie połaczyło w merge'u.

3 Wiecej niż jeden branch to?

Skoro udało nam sie już rozgałezić drzewo commitów, to chyba czas nauczyć sie działać na gałeziach profesionalnie.

- Aby utworzyć nowa gałaź projektu, należy skorzystać z polecenia `git branch <NAZWA>`.
- Aby przejść na nowo utworzona gałaź warto skorzystać z polecenia `git switch <NAZWA>` albo `git checkout <NAZWA>`.
- Aby wykonać 2 poprzednie punkty na raz można skorzystać z polecenia `git checkout -b <NAZWA>`.
- W nowo stworzonym branchu zaimplementuj jakaś drobna funkcjonalność, np. stwórz nowy plik, a nastepnie zacommituj ja i wypchnij. Podejrzyj jak wyglada log.
- Kiedy rozwój nowej funkcjonalności sie zakończy możemy go właczyć do brancha master. W tym celu przejdź na branch master, a nastgepnie wykonaj polecenie `git merge <NAZWA_BRANCHA_KTORY_CHECMY_WLACZYC_DO_MASTERA>`.
- Usuń stworzony branch.
- Stwórz nowy branch. Wykonaj kilka commitów w branchu master oraz nowo utworzonym branchu. Zobacz, że zawartość branchy jest różna.
- Zmergeuj nowy branch w mastera. Zobacz jak wyglada historia commitów. Zobacz, że nowy branch nie ma zmian które były wprowadzane na masterze. Aby temu zaradzić zmerguj mastera w nowy branch.
- Jeżeli chcesz, aby Twoje drzewo lokalnego branch'a wygladało ładniej i było troche mniej bujnie, warto rozważyć skorzystanie z komendy `git rebase`. Stwórz nowy branch. Wykonaj kilka commitów w branchu master oraz nowo utworzonym branchu. Wykonaj polecenie `git`

`rebase master <BRANCH>`, które sprawi, że zmiany wykonane na masterze zostana wcielone w nowym branchu, a historia bedzie prosta.

- Czasami czego nam trzeba to tylko jeden commit z innego brancha, w tym celu stworzone zostało polecenie `git cherry-pick`. Stwórz 2 commity na nowycm branchu i wypchnij je, a nastepnie na branchu master uzyj polecenia `git cherry-pick <ID_COMMITA_DO_WCIELENIA_DO_MASTERA>` podajac ID drugiego commita.

4 A co jeżeli popełnie błąd?

- Przydatna komenda do odkrywania zmian (w tym potencjalnie tych złych) jest `git diff`, dzieki niej możesz poznać np. jakie zmiany zostały wprowadzone do plików które nie zostały jeszcze właczone do commit'a. Ponadto, dzieki niej możesz sprawdzić różnice pomiedzy poszczególnymi commitami, w tym celu wywołaj polecenie
`git diff <IDENTYFIKATOR_COMMITA_1> <IDENTYFIKATOR_COMMITA_2>`. Identyfikatory to te takie długie ciagi cyfr i liter. Można też odnieść sie do aktualnego stanu `HEAD`, lub jego poprzedników np. 2 commita od aktualnego stanu `HEAD^2`.

- Jeżeli zmiana została już zacommitowana, to można odwrócić go korzystajac z polecenia `git revert <ID_COMMITA>`. Polecenie to stworzy nowy commit ktory odwraca działanie starego.

- W podobnym stylu, ale troche inaczej zadziała polecenie `git reset`, które usunie commity i sprawi, że wprowadzone przez nie zmiany znikna z folderu, a jedynie z repozytorium. Uwaga, rzecz ta sie dzieje lokalnie, wiec przy próbie push'a git bedzie twierdził że konieczny jest pull, który przywróci stan zewnetrznego repozytorium, aby temu zapobiec można dodać flage `--force`.

- Czesto zdarza sie, że zapomnieliśmy dodać coś do ostatniego commita. W takim przypadku można naprawić swój bład tworzac zawartość dowego commita, a jedynie przy samym poleceniu `git commit` dodać flage `--amend`, dzieki temu nie utworzymy nowego commita a jedynie zedytujemy stary.

- Jeżeli szukamy autora danej linijki pomocnym jest polecenie `git blame <NAZWA_PLIKU>`, które umożliwia podglad w którym commicie i kto zmieniał zawartość pliku.

5 Inne przydatne bajery

1. Ciekawa pomoca w zarzadzaniu repozytorium jest plik `.gitignore`, w którym zapisujemy jakie pliki maja być niewidoczne dla git'a. Dzieki niemu można np. wykluczyć pliki o rozszerzeniu `*.png` albo `*.pyc`, które nie powinny być przechowywane w repozytorium kodu.

2. Inna, fajna funkcjonalnościa sa submoduły https://git-scm.com/book/en/v2/Git-Tools-Submodules. Dzieki nim jestesmy w stanie w kontrolowany i zgrabny sposób przechowywać w repozytorium inne repozytoria np. z jakaś biblioteka z której korzystamy.