Czym jest Git?

Git jest darmowym i open-source'owym systemem kontroli wersji, pomocny przy pracy zarówno nad małymi, jak i ogromnymi projektami. System kontroli wersji, inaczej VCS (Version Control System) to narzędzie, które pozwala zachowywać historię zmian w pliku, lub zestawie plików, w taki sposób, aby można było bezproblemowo cofnąć się do wcześniejszej wersji tych plików. Posługiwać się dla przykładu będziemy dzisiaj kodem, ale VCS można używać właściwie do trzymania historii wersji każdego pliku na komputerze.

Czemu Git jest taki super?

Zasadniczo wszystkie VCS poza Gitem przechowuje informacje jako historię zmian przeprowadzonych w plikach. Podejście Gita jest zupełnie inne – tu przechowywane są migawki *(snapshots).* Przy każdej wprowadzanej przez użytkowników zmianie, git tworzy obraz tego, jak wyglądają w danym momencie wszystkie pliki, ewentualnie linkuje do wcześniejszych wersji tych niezmienionych.

Gita traktować możemy jako swoistą bazę danych *(repozytorium)* migawek, które wraz z postępem prac nad projektem są zatwierdzane i dodawane *(commitowane)* do bazy, a nie w żaden sposób podmieniane czy zastępowane.

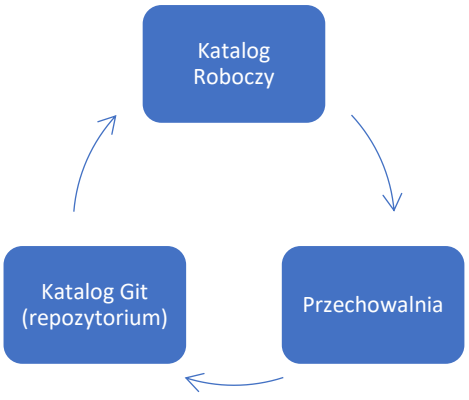
Ponadto, korzystając z Gita ciężko jest coś poważnie zepsuć w swoim kodzie. Pliki na komputerze mają swoją jedną wersję – tą, którą jako ostatnią zapisaliśmy. Dzięki korzystaniu z Gita, nawet jeśli nieźle w kodzie nabroiliśmy, zawsze możemy wrócić do jednaj z wcześniejszych migawek z poprawnie działającym programem. Git do swojej bazy danych pliki jedynie dodaje, żeby coś usunąć trzeba się już bardziej nagimnastykować.

Stany plików

Podstawą do zrozumienia zasady działania gita jest zrozumienie, czym są stany, w jakich mogą znajdować się w danym momencie pliki:

- Modified – pliki, w których wprowadzono zmiany, ale nie zostały jeszcze dodane do bazy danych.
- Staged – zmodyfikowane lub nowe pliki, które zostały oznaczone jako przeznaczone w bieżącej wersji do *zacommitowania*
- Commited– pliki, które są bezpiecznie przechowywane w lokalnej i, ewentualnie, zewnętrznej bazie danych

Zrozumienie stanów w jakich mogą znajdować się pliki prowadzi nas do kolejnej ważnej części: trzech sekcji projektu Git:



- Katalog roboczy *(working directory)* – obraz jednej z wersji projektu, pobrany z bazy danych w katalogu Git
- Przechowalnia *(staging area)* - prosty plik, zawierający informacje o tym, co wydarzy się w następnym *commicie*
- Katalog Git – tutaj Git przechowuje bazę danych migawek naszego projektu

Podczas pracy nad projektem, niezależnie czy własnym, czy zespołowym, proces pracy z Gitem wygląda następująco:

1. Dokonanie zmian w plikach w katalogu Roboczym
2. Oznaczenie plików, które chcemy dodać do repozytorium jako śledzone i dodanie ich migawki do poczekalni
3. Zatwierdzenie *(commit),* czyli zapisanie zawartości przechowalni jako snapshota w katalogu Git.

Instalacja Gita

Instalacja gita na linuxa jest możliwa bezpośrednio ze źródeł, co jest zapewne zalecane i najlepsze, ale również nie jest to najszybszy proces, dlatego zainteresowanych odsyłam do dokumentacji[1].

Dużo prostszą metodą instalacji Gita jest wykorzystanie narzędzia zarządzania pakietami wbudowanego w wiele z dystrybucji Linuxa. Korzystający z systemów opartych na Debianie, jak np. Ubuntu, powinni użyć polecenia:

```
sudo apt-get install git
```

Dla użytkowników Fedory polecenie będzie wyglądało następująco:

```
sudo yum install git-core
```

Dla systemu macOS, Gita najłatwiej jest pobrać i zainstalować stąd. Alternatywnie można skorzystać z Homebrew (http://brew.sh) działającego analogicznie do wymienionych wcześniej linuxowych instalatorów pakietów.

```
brew install git
```

Dla użytkowników Windowsa polecam git for windows, zawierający również graficzny interfejs Gita, do pobrania tutaj.

Konfiguracja gita

Pierwszą rzeczą, którą należy zrobić po zainstalowaniu gita jest ustawienie swojej nazwy użytkownika oraz adresu e-mail. To ważne, ponieważ każdy wykonany przez nas commit zawiera te informacje. Akcje te wykonujemy z poziomu konsoli używając poleceń:

```
git config --global user.name "Imię Nazwisko"

git config --global user.email "adres@email.com"
```

[1] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

używając opcji `--global` ustawiamy nasze dane dla wszystkich projektów. Jeśli chcemy to zmienić jedynie dla jednego konkretnego projektu, pomijamy tę opcję.

Polecenie

```
git config user.name
```

powie nam, jakie są nasze bieżące dane.

Praca z Gitem

Pracować z gitem możemy zarówno jedynie lokalnie, jak również używać repozytoriów zdalnych. Ze swojej strony polecam na razie jak najczęstsze korzystanie w swojej pracy z repozytoriów zdalnych, tym zajmiemy się za chwilę.

Na razie w dowolnym środowisku stworzymy nowy projekt w dowolnym języku. Na przykładach posługiwać się będę pustym projektem w C++, ale nasze przykłady zadziałają również w przypadku innych języków, może to być nawet HTML. Po utworzeniu projektu uruchamiamy wiersz poleceń i przechodzimy do katalogu z naszym projektem. Poleceniem

```
git init
```

utworzymy w tym katalogu nowe lokalne repozytorium. Następnie poleceniem

```
git add [nazwa pliku]
```

dodajemy do przechowalni pierwszy plik projektu, natomiast

```
git commit -m "initial commit"
```

Zacommituje nam zmiany w projekcie z wiadomością „initial commit". Opcję `-m` można pominąć, wówczas zostaniemy poproszeni o wpisanie komentarza. Komentarz jest wymagany przy każdym commicie. Przy małych projektach, na jakich będziecie na razie pracować, niezwykle przydatna może się okazać opcja -a. Umożliwia ona całkowite pominięcie przechowalni i zacommituje nam wszystkie już śledzone i zmodyfikowane od ostatniego commita pliki.

Wprowadźmy teraz jakieś zmiany do projektu nad którym teraz pracujemy. Teraz poleceniem

```
git status
```

podejrzeć można jaki status mają nasze pliki. Wyglądać to powinno na przykład tak:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   main.cpp

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

        .idea/
        CMakeLists.txt
        cmake-build-debug/
```

Pierwszy segment mówi nam, jakie śledzone przez nas pliki, czyli te, które znajdują się w repozytorium zostały zmodyfikowane, ale nie dodane do poczekalni.

Druga sekcja listuje nam pliki i katalogi, których nie śledzimy, ale które znajdują się w naszym katalogu roboczym. Są to różne pliki konfiguracyjne środowiska, katalogi kompilatora i inne. Tych plików nie chcemy w swoim repozytorium, dlatego powiemy gitowi, żeby je ignorował. Utwórz plik .gitignore, otwórz go i wpisz te pliki i katalogi w kolejnych liniach, następnie dodaj plik .gitignore do repozytorium i uruchom jeszcze raz poprzednie polecenie.

```
C:\Users\micha\CLionProjects\gitExample>git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .gitignore

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   main.cpp
```

Dodaj teraz zmodyfikowany plik do przechowalni i zacommituj zmiany.

Pokazać się powinien komunikat o ilości zmienonych plików, dodanych i usuniętych linii linii.

Szybkie zadanie na 5 minut

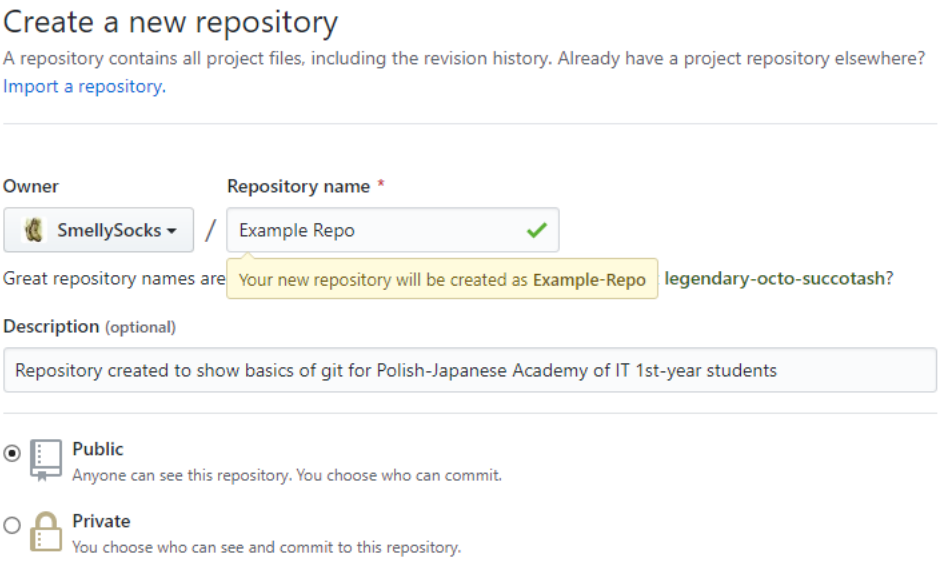
- Dodać do swojego kodu kilka nowych ficzerów, nie muszą być mądre ani specjalnie działać. Po dodaniu każdego zacommitować zmiany
- Stworzyć w projekcie i dodać do gita kolejny plik. Też nie musi być mądry, nie o to dzisiaj chodzi

Użyj teraz plecenia

```
git log
```

Wyświetli ono historię wszystkich commitów w projekcie od najnowszego do najstarszego.

Założymy teraz nowe repozytorium zdalne . Przechodzimy na stronę github.com, po zalogowaniu klikamy New i naszym oczom ukazuje się prosty kreator nowego, pustego repozytorium



Po utworzeniu, na razie pustego, repozytorium zdalnego zobaczymy kilka zaproponowanych przez githuba komend. Wyjaśnimy je sobie. Polecenie:

```
git remote add origin https://github.com/SmellySocks/Example-Repo.git
```

Połączy nas ze zdalnym repozytorium. Składnia polecenia `git remote` jest następująca:

```
git remote add [nazwa] [url]
```

Oznacza to, że w przyszłości będziemy mogli używać nazwy `origin` zamiast url naszego zdalnego repozytorium, jeśli będziemy chcieli dokonać jakichś zmian na serwerze.

```
git push -u origin master
```

wypycha do repozytorium zdalnego lokalne commity z gałęzi `master`. A skoro jesteśmy w temacie gałęzi…

Branche, czyli gałęzie

Gałęzie używane są do rozwijania funkcjonalności odizolowanych od siebie. Domyślną gałęzią jest `master`. Choć do szybkich, prostych i podstawowych projektów, można pracować tylko na tej jednej gałęzi, to przy większych projektach zespołowych należy każdą nowotworzoną funkcjonalność rozdzielać i rozwijać na oddzielnym branchu, a potem scalać, czyli *merge'ować* je z gałęzią główną.

```
git checkout -b [nazwa gałęzi]
```

Utworzy oraz automatycznie przełączy nas na nową gałąź. Polecenie

```
git branch
```

Wyświetli wszystkie gałęzie oraz zaznaczy, na której gałęzi obecnie pracujemy.

Szybkie zadanie na 5-10 minut

- Dodać do swojego kodu nową funkcjonalność. Nadal nie musi być mądra. Wymagane są przynajmniej dwa commity i push do remote. Nie wypychamy mastera, tylko naszego nowego brancha!
- Przełącz się na brancha master poleceniem

```
git checkout master
```

i dokonaj kilku zmian w kodzie. Następnie wypchnij je do remote.

Teraz dokonamy scalenia naszych gałęzi. Służy do tego polecenie

```
git merge [nazwa gałęzi]
```

scali ono nam gałęź aktywną z podaną w poleceniu. Upewnij się, że znajdujesz się na gałęzi `master` a następnie wykonaj polecenie scalenia gałęzi. Jeśli zmiany, które wprowadzaliśmy równolegle na branchach nie wchodzą sobie w żaden sposób w drogę, to mamy szczęście, nasze gałęzie zostały scalone. Jeśli jednak pojawiły się konflikty, zobaczymy komunikat:

```
C:\Users\micha\CLionProjects\gitExample>git merge feature_x
Auto-merging main.cpp
CONFLICT (content): Merge conflict in main.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

`git status`

podpowie nam, w których plikach pojawiły się konflikty.

```
C:\Users\micha\CLionProjects\gitExample>git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

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 modified:   main.cpp

no changes added to commit (use "git add" and/or "git commit -a")
```

Przechodzimy więc do pliku main.cpp i szukamy znaczników konfliktu, które wstawił nam git, gdy napotkał problem. Dla przykładu:

```
<<<<<<< HEAD
    cout<<"twoja liczba podniesiona do kwadratu"<<users_input*users_input;
=======
    cout<<"twoja liczba pomnozona przez 2"<<users_input*2;
    cout<<"twoja liczba pomnozona przez 6"<<users_input*6;



>>>>>>> feature_x
```

`<<<<<<< HEAD` Oznacza początek zmian z brancha głównego. Następnie `=======` oddziela nam zmiany wprowadzone w tym samym miejscu pomiędzy scalanymi branchami. `>>>>>>> feature_x` zaznacza koniec konfliktowych zmian. W tym momencie musimy zastanowić się, jakie zmiany chcemy wprowadzić, naprawić kod, jeśli zajdzie tak potrzeba i usunąć wszystkie znaczniki. W tym wypadku doszliśmy do wniosku, że wprowadzone zmiany z obu gałęzi są równie ważne dla sukcesu naszego projektu, dlatego po prostu usuwamy znaczniki, i commitujemy zmiany. Wróćmy na chwilę do terminala, aby zobaczyć log. Tym razem użyjemy podglądu graficznego, aby lepiej zrozumieć, co właśnie się wydarzyło.

`git log --graph`

Aby wyświetlić bardziej skompresowaną wersję tego loga należy dodać opcję `--oneline`. Teraz jest dobry moment aby wypchnąć zmiany do remote.

Kilka przydatnych poleceń

```
git checkout -- [nazwa pliku]
```

Jeśli nie dodaliśmy zmian do przechowalni, to polecenie cofnie wprowadzone w pliku zmiany do ostatniej zacommitowanej

```
gitk
```

względnie przyjazne GUI dla gita

```
git add -i
```

inteligentne zarządzanie przechowalnią