Git – podstawy

MB

25 marca 2021

Spis treści

1 Wstęp

1.1 Git

Zgodnie z definicją Git jest *systemem kontroli wersji*. Z kolei system kontroli wersji oznacza oprogramowanie służące do śledzenia zmian głównie w kodzie źródłowym oraz do pomocy programistom w łączeniu zmian dokonanych w plikach przez wiele osób w różnym czasie.

Tyle suchej definicji.

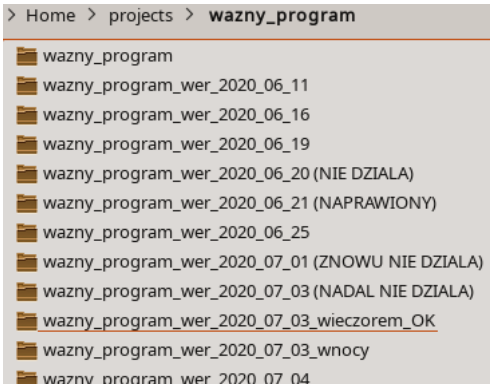
W prostych słowach systemem kontroli wersji to program, który ułatwia życie programistom – zarówno grupom współpracującym nad tym samym kodem (nieodzowne narzędzie w każdej firmie tworzącej oprogramowanie), ale i pojedynczym osobom, pragnącym zachować porządek w pisanym przez siebie kodzie.

Pomimo, że jego nauka może się wydawać na początku nieco zbyt abstrakcyjna i zupełnie niepotrzebna do opanowania sztuki programowania, o zaletach jego stosowania można się przekonać bardzo szybko.

Jeśli jesteś osobą początkującą i potrzebna Ci motywacja do rozpoczęcia poznawania programu Git być może poniższy przykład zachęci Cię do nauki.

Wyobraź sobie, że piszesz kod źródłowy ważnego projektu (np. z tego przedmiotu, lub też program będący częścią twojej przyszłej pracy dyplomowej). Będzie to czynność, która w zależności od stopnia złożoności projektu zajmie kilka dni, może tygodni, a może nawet miesięcy... Często będzie dochodzić do takiej sytuacji, w której kawałek kodu (np. jakaś ważna funkcja, klasa itp.) będzie już działał jak należy. Przed przystąpieniem do dalszego pisania, warto w takich chwilach zachować aktualną wersję. Po co? Chociażby po to aby w przyszłości, uniknąć sytuacji, w której przy okazji kolejnych zmian w kodzie program przestaje się kompilować/działać jak należy, a my, wyrywając sobie włosy z głowy, spędzamy kilka godzin (albo dni) na próbach przywrócenia kodu do stanu, w którym "jeszcze wszystko działało" i zastanawianiu się, która zmiana kodu spodowała wielką katastrofę. Tego typu sytuacje zdarzają się bardzo często, dlatego warto się przed nimi zabezpieczyć.

Istnieje proste rozwiązanie tego problemu. Co jakiś czas, katalog, w którym znajduje się kod źródłowy programu kopiujemy z odpowiednią adnotacją (albo zmieniając stosownie jego nazwę). Niestety to rozwiązanie ma też pewne wady. Jego stosowanie szybko doprowadzi do sytuacji, w której katalog przechowujący nasz program wygląda np. tak jak przedstawiono to na Rysunku 1.



Rysunek 1: Wygląd katalogu z kodem archiwizowanym metodami amatorskimi

W praktyce, do większości z tak zachowanych wersji kodu programista nigdy nie będzie wracał, a archiwum będzie się rozrastało w zawrotnym tempie.

System kontroli wersji pozwala zapobiec takim sytuacjom, przechowując kod programu oraz całą jego historię w sposób oszczędny, a ponadto pozwala wrócić do poprzednich wersji w prosty i wygodny sposób.

Git jest obecnie najpopularniejszym spośród istniejących systemów kontroli wersji, dlatego warto się go nauczyć na samym początku swojej przygody z programowaniem.

Istnieje wiele materiałów i kursów wprowadzających w podstawy gita, dlatego w celu dokładniejszego poznania można i warto dodatkowo zaznajomić się np. z tym, tym lub tym.

Dla osób nieco bardziej zaawansowanych - dokumentacja gita:
https://git-scm.com/book/pl/v2/

To źródło zawiera więcej szczegółów technicznych, jest ono jednak również napisane przystępnym językiem.

W dalszej części niniejszego materiału, w rozdziale 2, przedstawione zostaną podstawowe polecenia Gita, potrzebne i wystarczające do codziennej pracy.

1.2 Git a Github

Dodatkowym atutem używania gita jest istnienie darmowych platform, które pozwalają na łatwy backup swojego kodu oraz jego współdzielenie z innymi użytkownikami bez potrzeby

utrzymywania własnego serwera Git. Do najpopularniejszych należą obecnie GitHub, GitLab, Bitbucket czy SourceForge.

Aby móc z nich korzystać, należy założyć konto w ramach danej usługi, co pozwoli na dalszym etapie na komunikację między naszym lokalnym repozytorium, a jego zdalnym odpowiednikiem. Proces rejestracji jest intuicyjny i opisany w wielu źródłach w internecie, np. tu, gdzie opisana jest rejestracja w serwisie GitHub.

W dalszej części instrukcji przyjmuje się, że tego typu konto jest założone.

Uwaga!!!: W trakcie rejestracji nie ma obowiązku podawać naszego prawdziwego imienia czy nazwiska, jeśli tego nie chcemy.

1.3 Git a interfejsy graficzne

Git jest programem, z którym pracujemy z linii poleceń systemu. Istnieje wiele narzędzi - graficznych interfejsów do Gita - które pozwalają na wizualizane przedstawienie struktury historii oraz zarządzanie projektem przy użyciu wyłącznie myszy. Są one pomocne przy zarządzaniu wielkimi, złożonymi projektami, jednak na początkowym etapie poznawania Gita, programy tego typu bardziej przeszkadzają, niż pomagają w zrozumieniu działania systemu kontroli wersji, dlatego warto rozpocząć naukę od pracy w linii poleceń. Takie podejście jest po pierwsze szybsze, a po drugie uniezależnia od oprogramowania dostępnego dla programisty na danej maszynie (linia poleceń będzie dostępna na każdym komputerze i na każdym systemie na jakim przyjdzie Wam pracować).

2 Git w praktyce

Rozdział ten ma na celu pokazanie podstawowych poleceń Gita, przydatnych w codziennej pracy z kodem. Aby uczynić niniejszą instrukcję uniwersalną nie wybrano żadnego konkretnego języka programowania, ale zdecydowano, że archiwizowanym projektem będzie kilka plików tekstowych (Można sobie wyobrazić, że będą to rozdziały książki). Zakłada się również, że czytelnik posiłkując się źródłami w w.w. linkach zna znaczenie pojęć takich jak:

- repozytorium
- commit

2.1 Instalacja oprogramowania

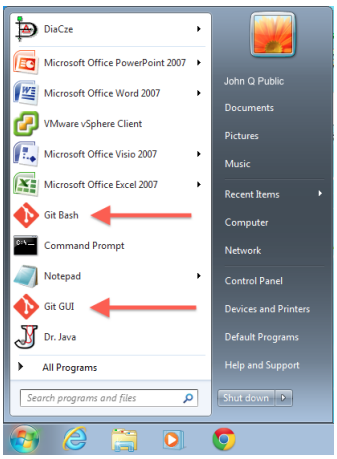
Proces instalacji Gita jest opisany np. tutaj: https://rogerdudler.github.io/git-guide/index.pl.html, gdzie w części "Instalacja" wybieramy używany przez siebie system operacyjny.

Uwaga: Jeśli instalujemy Gita w systemie Windows można posłużyć się tym przewodnikiem.

Uwaga: (Dla Windows) W trakcie instalacji warto zaznaczyć opcję instalacji "Git Bash Here" oraz "Git GUI Here".

Uwaga: (Dla Windows) W trakcie instalacji Gita jest możliwość wybrania domyślnego edytora tekstu, którym będziemy się posługiwać w celu edytowania komunikatów opisujących kolejne zmiany. Warto na tym etapie wybrać swój ulubiony edytor, a jeśli takiego nie mamy to wybrać nawet Windows'owy notatnik. W przeciwnym wypadku domyślnym edytorem będzie program Vim, z którego przeciętny użytkownik komputera z pewnością nie będzie chciał korzystać. Jeśli ten punkt opuścimy, lub (z jakichś względów, pomimo wybrania innego edytora jako domyślnego) Git będzie nadal otwierał Vim w celu edycji komunikatu zmian -> patrz tutaj -> 2.2.

Po zainstalowaniu programu, w systemie Linux uruchamiamy terminal, natomiast w systemie Windows, jeśli zaznaczyliśmy odpowiednią opcję w trakcie instalacji (patrz: paragraf wyżej)- uruchamiamy Git Bash (Rysunek 2).



Rysunek 2: Git Bash w menu Start systemu Windows

Poniższe polecenia zakładają, że użytkownik założył konto na platformie GitHub na poniższe dane:

Login: mojlogin

Name: Nowy Uzytkownik

Email: adres@costam.pl
a katalogiem przeznaczonym na projekty jest np.
~/projekty (w Linuksie), albo
C:/Users/mojlogin/projekty/ (w Windows).
Wpisujemy w linii poleceń:

```
cd C:/Users/mojlogin/projekty
```

aby uczynić ten katalog naszym katalogiem roboczym.

2.2 Konfiguracja Git

Na początku musimy wstępnie skonfigurować naszą instalację Gita. Najważniejsze to ustawienie nazwy użytkownika i adresu e-mail. Dokonuje się tego wpisując poniższe polecenia:

```
git config --global user.name mojlogin
git config --global user.email adres@costam.pl
```

W przypadku gdybyśmy chcieli zmienić domyślny edytor tekstowy programu Git (np. na notatnik Windowsa) dokonujemy tego poleceniem:

```
git config --global core.editor "notepad.exe"
```

2.3 Tworzenie (inicjalizacja) projektu

Tworzymy katalog przeznaczony na nasz projekt i przechodzimy do niego

```
mkdir projekt1
cd projekt1
```

Inicjalizujemy repozytorium git

```
git init
```

W tym momencie w katalogu projekt1 pojawić się powinien katalog .git. **To w nim będzie przechowywana cała konfiguracja oraz historia zmian repozytorium**.

2.4 Sprawdzanie stanu plików

Teraz możemy podejrzeć stan naszego repozytorium wpisując komendę:

```
git status
```

W odpowiedzi systemie zwróci nam:

```
On branch master


No commits yet


nothing to commit (create/copy files and use "git add" to track)
```

Kolejne linijki mówią użytkownikowi, że

- znajduje się aktualnie na gałęzi `master` (Więcej o gałęziach w gicie można znaleźć tutaj. W początkowej fazie nauki w zupełności wystarczy praca na jednej, domyślnej gałęzi, którą Git tworzy za nas,czyli właśnie gałęzi `master`. Z tego powodu temat gałęzi w tej instrukcji nie będzie kontynuowany)

- nie zatwierdzono jeszcze żadnych zmian (`commit`-ów) w kodzie
- nie ma żadnych zmian, które mogłyby wymagać zatwierdzenia. Ostatnia linjka zawiera również informację jakie następne kroki powinien podjąć użytkownik: stworzyć/skopiować plik (do aktualnego katalogu) i użyć polecenia `git add`, aby dodać plik do śledzonych (track) przez system.

Uwaga: *Zawsze* warto przeanalizować odpowiedź, którą zwraca Git. W większości wypadków program sugeruje w niej użytkownikowi, jakie następne kroki powinien on podjąć.

W katalogu projekt1 utworzymy teraz nowy plik o nazwie Rozdzial1.txt:

```
echo To jest tresc nowego pliku. > Rozdzial1.txt
```

Po ponownym wpisaniu

```
git status
```

tym razem odpowiedź programu będzie brzmiała:

```
On branch master

No commits yet

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

Rozdzial1.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Dwie pierwsze linie nie zmieniły się w porównaniu do poprzedniego wywołania polecenia. W trzeciej linii pojawia się nowa informacja, która mówi, że w katalogu pojawił się nowy plik nieśledzony (untracked) przez Gita. Nieśledzony oznacza, że Git widzi plik, którego nie miałeś w poprzedniej migawce (zatwierdzonej kopii); Git nie zacznie umieszczać go w przyszłych migawkach, dopóki sam mu tego nie polecisz.

2.5 Dodawanie plików do poczekalni

Aby sprawić, że Git będzie śledził zmiany w jakimś pliku, należy wydać odpowiednie polecenie (które zostało zasugerowane w ostatniej odpowiedzi programu).

```
git add Rozdzial1.txt
```

Jeśli uruchomisz teraz ponownie polecenie status, zobaczysz, że twój plik jest już śledzony i znalazł się w tzw. poczekalni, tzn. jest opatrzony nagłówkiem "Zmiany do zatwierdzenia" (Changes to be committed):

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

new file:   Rozdzial1.txt
```

2.6 Zatwierdzanie zmian

W tej chwili możemy dokonać pierwszego zatwierdzenia zmian. Wszystkie pliki, które znajdują się w poczekalni (Changes to be committed) zostaną zapamiętane w takim stanie, w jakim aktualnie znajdują się na dysku. Najprostszy sposób zatwierdzenia zmian to wpisanie:

```
git commit
```

Zostanie uruchomiony domyślny edytor tekstu (można go zmienić) z następującym tekstem:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
# new file:   Rozdzial1.txt
#
```

Jak widzisz, domyślny opis zmian zawiera aktualny wynik polecenia git status w postaci komentarza oraz jedną pustą linię na samej górze. Możesz usunąć komentarze i wpisać własny opis, lub pozostawić je, co pomoże zapamiętać zakres zatwierdzonych zmian.

Wyedytujmy treść naszego komunikatu tak, aby wyglądała jak poniżej:

```
Pierwszy commit. Inicjalizacja projektu.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
# new file:   Rozdzial1.txt
#
```

Po zapisaniu zmian w pliku (Save) i zamknięciu okna edytora Git stworzy nową migawkę opatrzoną twoim opisem zmian (uprzednio usuwając z niego komentarze i podsumowanie zmian). Stan projektu zostanie zapamiętany i będzie można do niego wrócić w przyszłości.

Można teraz ponownie sprawdzić stan projektu poleceniem `git status` i przeanalizować odpowiedź. Ze względów dydaktycznych, w czasie przechodzenia dalszych kroków tej instrukcji polecenie Polecenie `git status` można wywoływać po każdym kolejnym kroku dalszej instrukcji.

Ze względów dydaktycznych, zalecane jest aby od tej pory po każdym kolejnym poleceniu przedstawionym w instrukcji, wywoływać polecenie `git status` i analizować odpowiedź programu, nawet jeśli nie jest to wyraźnie napisane.

2.7 Dalsza praca

Zmieńmy plik `Rozdzial1.txt`, tak aby jego treścią był napis "To bedzie rozdzial 1.":

```
echo To bedzie rozdzial 1. > Rozdzial1.txt
```

Stwórzmy także dwa kolejne pliki:

```
echo Rozdzial2 > Rozdzial2.txt
echo Rozdzial3 > Rozdzial3.txt
```

i polećmy Gitowi, aby je śledził. Można to zrobić jednym poleceniem:

```
git add Rozdzial2.txt Rozdzial3.txt
```

lub sprytniej:

```
git add Rozdzial*.txt
```

Niech plik Rozdzial2.txt bedzie plikiem, w którym będzie przechowywana historia poleceń wykonanych w celu wykonania tej instrukcji. Otwórzmy go więc w edytorze i wyedytujmy tak, aby jego treść wyglądała jak poniżej:

```
Rozdzial2

git config --global user.name mojlogin
git config --global user.email adres@costam.pl

mkdir projekt1
cd projekt1

git init
echo To jest tresc nowego pliku. > Rozdzial1.txt
git status

git add Rozdzial1.txt
git status
git commit

echo To bedzie rozdzial 1. > Rozdzial1.txt
echo Rozdzial2 > Rozdzial2.txt
echo Rozdzial3 > Rozdzial3.txt

git add Rozdzial2.txt Rozdzial3.txt
```

Polecenie `git status` informuje, że dwa nowe pliki `Rozdzial2.txt` i `Rozdzial3.txt` są w poczekalni i czekają na zatwierdzenie (new file). Dodatkowo w pliku `Rozdzial2.txt` są zmiany (modified), które nastąpiły po jego dodaniu do poczekalni, i które, nie zostaną uwzględnione w kolejnym zatwierdzeniu dopóki ich również nie dodamy do poczekalni. Podobnie zmiany nastąpiły w pliku `Rozdzial1.txt`. Aby uwzględnić te niezatwierdzone zmiany i dodać je do poczekalni wygodnie będzie skorzystać z:

```
git add -u
```

które automatycznie dodaje do poczekalni wszelkie zmiany we wszystkich plikach, które Git obserwuje (modyfikator -u pochodzi od słowa 'update').

Następnie po wywołaniu

```
git commit
```

możemy wyedytować komentarz następnego commit'a, np. tak aby jego treść była jak poniżej:

```
wer. 0.1:
- dodano dwa kolejne pliki do projektu:
  * Rozdzial2.txt bedzie przechowywal historie polecen
  * nie wiadomo jeszcze co bedzie w pliku Rozdzial3.txt
- zmiana tresci w pliku Rozdzial1.txt

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
# new file:   Rozdzial2.txt
# new file:   Rozdzial3.txt
#
```

zapisujemy plik i zamykamy edytor.

2.8 Podgląd historii zmian

Po kilku zatwierdzeniach zmian poleceniem `commit` można sprawdzić historię dokonanych zmian. Służy do tego polecenie

```
git log
```

które w przypadku aktualnej wersji projektu zwróci mniej więcej coś takiego:

```
commit 2031dc1373cf4e9179e60ed0bbc95bd51f6a4c8e (HEAD -> master)
Author: przemarbor <marbor@prz.edu.pl>
Date:   Fri Oct 16 23:19:59 2020 +0200

    wer. 0.1:
    - dodano dwa kolejne pliki do projektu:
    * Rozdzial2.txt bedzie przechowywal historie polecen
    * nie wiadomo jeszcze co bedzie w pliku Rozdzial3.txt
    - zmiana tresci w pliku Rozdzial1.txt

commit 667a23c6172a5365513831439eb76c56f47e16b3
Author: przemarbor <marbor@prz.edu.pl>
Date:   Fri Oct 16 23:15:59 2020 +0200

    Pierwszy commit. Inicjalizacja projektu.
```

Domyślnie, polecenie `git log` uruchomione bez argumentów, listuje zmiany zatwierdzone w tym repozytorium w odwrotnej kolejności chronologicznej, czyli pokazując najnowsze zmiany w pierwszej kolejności. Jak widać polecenie wyświetliło zmiany wraz z ich sumą kontrolną SHA-1 (to te długie ciągi losowych znaków), nazwiskiem oraz e-mailem autora, datą zapisu oraz notką zmiany.

Polecenie `git log` posiada całą masę opcji i modyfikatorów pozwalających na na dokładne wybranie interesujących nas informacji. Tu nie będziemy się nimi zajmować i zainteresowanych odeślemy tylko do stosownego źródła.

2.9 Poruszanie się po historii zmian

W dowolnej chwili możemy zobaczyć jak wyglądał projekt w chwili każdego poprzedniego zatwierdzenia. Służy do tego polecenie `git checkout`. Przypomnijmy sobie sumę kontrolną naszego pierwszego commita. Wystarczy podać jej kilka pierwszych znaków (zaleca się 7), aby w katalogu roboczym odtworzyć stan projektu z danego momemntu historii.

Uwaga: Wartości sumy kontrolnej są unikatowe, tak więc w każdym indywidualnym przypadku, należy w poleceniu poniżej wartość SHA-1 występującą w Waszym projekcie!!!

```
git checkout 667a23c
```

Jeśli teraz spojrzymy do katalogu z projektem okaże się, że treść pliku `Rozdzial1.txt` powróciła do swojej oryginalnej postaci, a pliki `Rozdzial2.txt` oraz `Rozdzial3.txt` zniknęły.

Aby powrócić do aktualnej wersji projektu możemy wpisać:

```
git checkout 2031dc1
```

ale lepszym (bardziej uniwersalnym) rozwiązaniem będzie:

```
git checkout master
```

które zawsze odtwarza najnowszą wersję kodu.

Uwaga: Jeśli w trakcie przeglądania którejś z historycznych zmian dokonamy edycji plików, Git nie pozwoli na powrót do aktualnej wersji kodu, aby uniemożliwić (być może niechcianą) utratę właśnie dokonanych zmian. Jeśli nie potrzebujemy tych zmian zapamiętywać możemy wydać polecenie:

```
git checkout *
```

które każe Gitowi zapomnieć o wszystkich zmianach dokonanych w historycznej migawce i pozwala nam na bezproblemowy powrót do aktualnej wersji kodu przy pomocy `git checkout master`.

2.10 Synchronizacja zmian z serwerem zdalnym

Co jakiś czas warto robić backup aktualnej wersji naszego repozytorium. Posiadając konto np. na GitHubie w bardzo łatwy sposób możemy dokonywać tego typu synchronizacji.

Najpierw musimy założyć zdalne repozytorium i najprościej jest to zrobić poprzez przeglądarkę. Proces zakładania jest opisany np. tutaj (Rozdział 3. Utworzenie pierwszego repozytorium) i sprowadzą się właściwie do wybrania nazwy repozytorium i naciśnięcia przycisku "Create repository". Przyjmijmy, że nazwa zdalnego repozytorium to `projekt-testowy` (zauważmy, że nazwa zdalnego repozytorium nie musi być taka sama jak folderu, w którym znajduje się nasz kod lokalnie).

Przy pierwszym wysłaniu kodu na serwer posłużymy się

```
git remote add origin https://github.com/mojlogin/projekt-testowy.git
git branch -M main
git push -u origin main
```

Zostaniemy zapytani o nazwę użytkownika i hasło, po których podaniu git poinformuje nas umieszczeniu kodu w zdalnym repozytorium.

Uwzględnijmy jeszcze w naszym kodzie nowo poznane polecenia. Zrobimy to uzupełniając plik `Rozdzial2.txt` o poniższą treść:

```
git add -u

git commit

git log

git checkout 667a23c

git checkout master

git remote add origin https://github.com/mojlogin/projekt-testowy.git
git branch -M main
git push -u origin main

git add -u
git commit

git push

git clone
```

i zatwierdźmy tę zmianę w repozytorium:

```
git add -u
git commit
```

Wpisując w komentarzu

```
wer. 0.2:
- aktualizacja pliku Rozdzial2.txt o kilka ostatnich polecen
```

Następną synchronizację (i wszystkie kolejne) będziemy dokonywać przy pomocy samego:

```
git push
```

po którym będziemy musieli podać dane uwierzytelniające.

Teraz na dowolnym komputerze z zainstalowanym Gitem możemy pobrać aktualną wersję naszego kodu przy pomocy:

```
git clone <adres repozytorium>
```

Dla testu przejdź do katalogu projekty:

```
cd ..
```

Stwórz katalog

```
mkdir test-klonowania
```

Wejdź do niego

```
cd test-klonowania
```

i pobierz aktualną wersję swojego przed chwilą stworzonego repozytorium:

```
git clone https://github.com/mojlogin/projekt-testowy.git
```

Jeśli repozytorium zostało utworzone jako prywatne znowu będzie trzeba podać dane uwierzytelniające. W przypadku repozytorium publicznego kod zostanie pobrany bez dodatkowych pytań.

Repozytorium, które zawiera kod przedstawiony w niniejszej instrukcji można pobrać za pomocą polecenia:

```
git clone https://github.com/przemarbor/projekt-testowy.git
```

3 Podsumowanie - ściągawka

Typowy schemat pracy z kodem przy wykorzystaniu Gita:

1. Rozpoczęcie pracy na lokalnym repozytorium:
   - Możliwość 1: Rozpoczęcie nowego projektu lokalnie:
     (a) inicjalizacja lokalnego repozytorium (`git init`)
     (b) utworzenie repozytorium na zdalnym serwerze (przez przeglądarkę) i jego początkowa synchronizacja z lokalnym:
     ```
     git remote add origin <adres zdalnego repozytorium>
     git branch -M main
     git push -u origin main
     ```
   - Możliwość 2: Pobranie aktualnej wersji istniejącego już projektu (`git clone`)
2. Praca na lokalnym repozytorium:
   - dodanie plików do śledzenia przez Gita (`git add`)
   - edycja plików
   - dodanie wyedytowanych plików do poczekalni w celu zatwierdzenia zmiany (`git add -u`)
   - zatwierdzenie zmian (migawki) (`git commit`) wraz z odpowiednim opisem
3. Co jakiś czas (np. po zatwiedzeniu jakiejś ważnej zmiany w kodzie): synchronizacja stanu lokalnego repozytorium z repozytorium zdalnym (`git push`)

3.1 Polecenia

- najważniejsze:

```
git status
git add
git commit
git push
```

- rzadziej stosowane:

```
git pull
git checkout
git log
```

4 Tematy nieomówione w instrukcji, a warte samodzielnego przestudiowania

- ignorowanie plików (.gitignore)
- pomijanie poczekalni
- usuwanie plików z projektu
- zmiana nazw plików w projekcie
- usuwanie plików z poczekalni
- cofanie zmian
- prywatne i publiczne repozytoria na GitHubie/GitLabie/...
- uwierzytelnianie SSH