

Systemy kontroli wersji

Wstęp

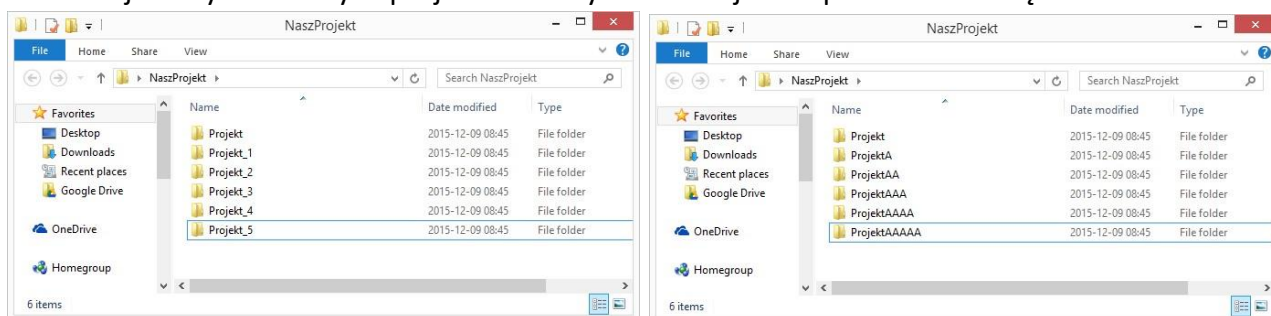
Niniejsze ćwiczenia przedstawiają podstawowe informacje na temat systemów kontroli wersji. Ponadto prezentowane są praktyczne zadania z wykorzystaniem systemu kontroli wersji Git.

Co dokładnie oznacza sformułowanie „system kontroli wersji”?

System kontroli wersji pozwala na zapisanie historii zmian w danych plikach, a także daje możliwość przywrócenia wersji pliku z konkretnego punktu w czasie. Taki system jest przydatny nie tylko dla programisty chcącego zapisywać historię zmian w kodzie źródłowym, ale także np. grafika zapisującego historię zmian w plikach graficznych lub osoby chcącej zapisać historię zmian w plikach tekstowych. Taki system jest również bardzo przydatny przy współpracy wielu osób nad tymi samymi plikami. System kontroli wersji lub inaczej VCS (ang. „Version Control System”) jest uniwersalnym narzędziem bez którego współczesna praca nad rozwojem oprogramowania nie byłaby możliwa. Inne możliwości jakie może nam dać to:

- Możliwość wycofania zmian z konkretnego pliku i cofnięcie jego stanu do wersji wcześniejszej.
- Cofnięcie całego repozytorium do konkretnego punktu w czasie. Pojęcie „repozytorium” oznacza zazwyczaj folder z danymi, które są wersjonowane.
- Możliwość porównywania zmian zachodzących w czasie w konkretnych plikach.
- Sprawdzenie kto ostatnio modyfikował kod lub kto odpowiada za zmiany w konkretnych plikach.
- Możliwość łatwej współpracy z zewnętrznymi serwisami umożliwiającymi przechowywanie kodu źródłowego swoich aplikacji.
- I wiele więcej...

Poniżej przykład „domowego” sposobu kontrolowania wersji, który być może część z Was zna z realizacji różnych szkolnych projektów. Oczywiście nie jest to polecane rozwiązanie:



Git

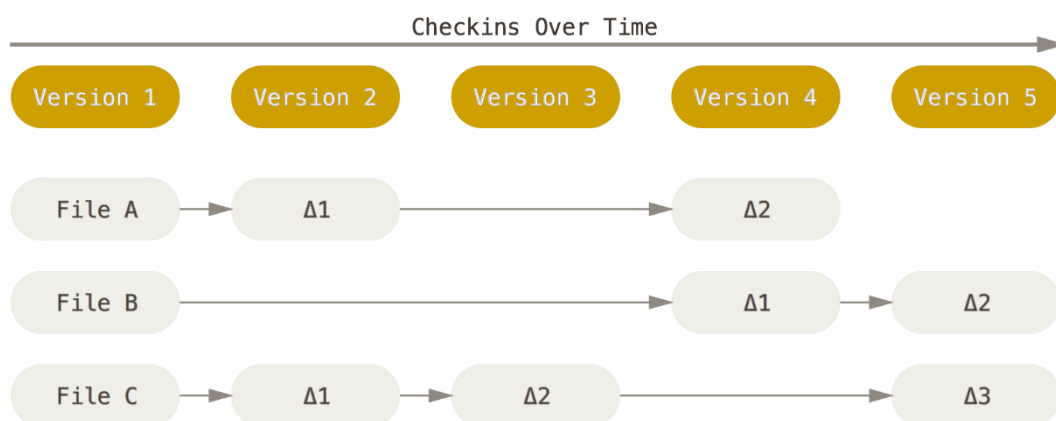
Git jest jednym z dostępnych systemów DVCS. Historia jego powstania związana jest z Linuksem. We wczesnych latach rozwoju systemów Linux (1991-2000) zmiany w oprogramowaniu były przechowywane i przekazywane między współpracującymi programistami jako łatki (ang. „patches”) i zarchiwizowane pliki. W roku 2002 kod jądra Linuksa zaczął być wersjonowany z pomocą systemu DVCS pod nazwą „BitKeeper”. W roku 2005 relacje między społecznością rozwijającą Linuksa i firmą stojącą za BitKeeper’em się pogorszyły. BitKeeper przestał być aplikacją z której twórcy Linuksa mogli korzystać bez opłat. W tym momencie Linus Torvalds (twórca Linuksa) zdecydował się na rozwinięcie własnego narzędzia kontroli wersji wykorzystując doświadczenia nabyte podczas pracy z BitKeeper’em. Główne cechy nowego systemu to:

- Szybkość
- Prostota obsługi
- Silne wsparcie dla nieliniowego trybu pracy (wiele równoległe rozwijanych repozytoriów, możliwość tworzenia gałęzi)
- W pełni rozproszony
- W wydajny sposób radzący sobie z dużymi projektami

W ten sposób w 2005 narodziła się pierwsza wersja Git’a.

Podstawy działania Git

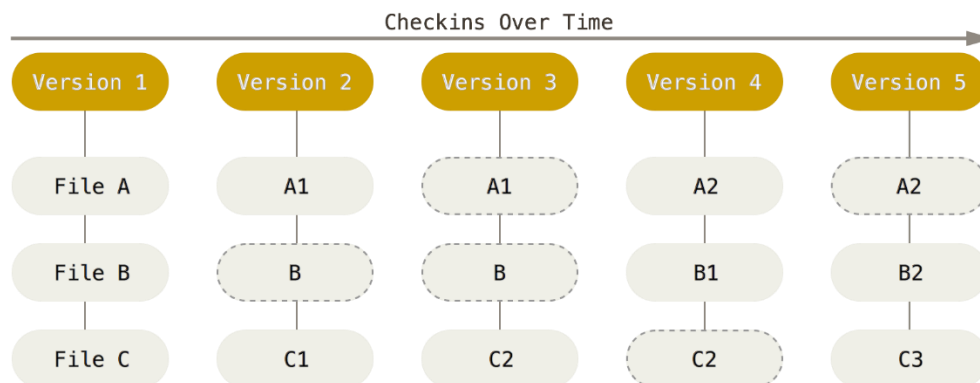
Zasada działania Git różni się w znaczny sposób od innych systemów VCS takich jak SVN. Główna różnica polega na tym w jaki sposób Git przechowuje historię zmian w projekcie. Większość innych systemów VCS takich jak SVN przechowuje historię zmian w oparciu o pliki i różnice między nimi. Takie systemy początkowo przechowują listę plików, a następnie zmiany (delta) zachodzące w plikach. Np. w postaci wspomnianych wcześniej tzw. „łatek”.



Rysunek 3 - przechowywanie historii w oparciu o zmiany, które zaszły w stosunku do pliku początkowego (źródło: <https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>)

Git funkcjonuje w inny sposób i działa w obrębie tzw. „migawek”. Innymi słowy każda nowa zmiana wprowadzona przez użytkownika do repozytorium tworzy nową migawkę całego projektu. Jeśli dany plik nie został zmodyfikowany Git posłuży się referencją do poprzedniej

wersji (skoro zmiany nie zaszły to nie musi tworzyć nowej wersji pliku). W ten sposób historia zmian w Git'cie przypomina ciąg migawek.



Rysunek 4 - kolejne wersje/migawki repozytorium przechowywane przez Git'a (źródło: <https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>)

W przypadku pracy z Git każdy klient posiada pełne repozytorium wraz z jego historią. Oznacza to, że może wprowadzać zmiany, cofać zmiany itd. bez potrzeby kontaktowania się z zewnętrznym serwerem jak w przypadku rozwiązań CVCS. Bez potrzeby połączenia internetowego mamy możliwość przeglądania pełnej historii zmian w projekcie. Jeśli chcemy otrzymać wersję pliku sprzed miesiąca Git jest w stanie nam ją zwrócić na podstawie lokalnych danych. To daje nam możliwość pracy nad projektem w niemal każdych warunkach (m.in. brak Internetu).

Wszystkie zmiany zapisywane przez Git'a są identyfikowane poprzez sumę kontrolną. Dzięki niej Git wie, czy w danych pikach zaszły jakieś zmiany. Ta funkcjonalność jest fundamentalna dla działania Git'a. Nigdy nie zmodyfikujemy pliku bez wiedzy Git'a. W celu obliczania sumy kontrolnej wykorzystywany jest algorytm SHA-1. Suma składa się z 40-znakowej wartości tekstowej - znaków heksadecymalnych (0-9, a-f) i obliczana jest na podstawie zawartości repozytorium.

Np.

24b9da6552252987aa493b52f8696cd6d3b00373

Niemal wszystkie operacje rejestrowane przez Git'a wiążą się z **dodawaniem** danych do bazy danych. Trudno jest wykonać operację, które nie byłaby odwracalna. W momencie kiedy stworzyliśmy kolejną migawkę danych trudno jest coś „popsuć” – tzn. stracić historię. Dzięki temu możemy łatwo eksperymentować z naszym kodem bez ryzyka utraty informacji.

Trzy stany występujące w Git'cie

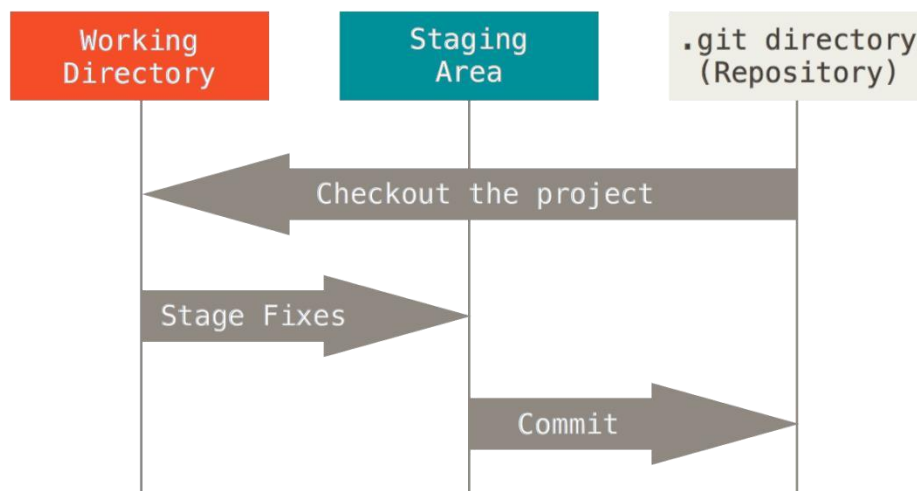
Niniejszy fragment jest bardzo ważny w celu zrozumienia istoty działania Git'a. Plik w repozytorium Git'a może znajdować się w jednym z 3 stanów: **committed**, **modified**, **staged**. **Committed** oznacza, że plik jest bezpiecznie przechowywany w lokalnej bazie danych.

Modified oznacza, że zaszły jakieś zmiany w pliku, ale nie zostały jeszcze zapisane w bazie lokalnej.

Staged oznacza, że oznaczyliśmy plik w stanie **modified** w jego obecnej formie, aby znalazł się w kolejnej migawce.

Projekt wersjonowany przez Git'a składa się z trzech sekcji:

- Folderu Git (.git)
- Folderu roboczego zawierającego kod, nad którym obecnie pracujemy □
Obszaru staging'owego



Rysunek 5 - grafika pokazuje trzy podstawowe obszary działania wykorzystywane przez Git'a

W folderze „.git” przechowywane są metadane wykorzystywane przez Git'a. Jest to najważniejszy dla Git'a folder. Zazwyczaj kopiujemy go razem z całym repozytorium i dzięki niemu mamy dostęp do pełnej historii zmian.

Folder roboczy zawiera migawkę projektu z konkretnego punktu w czasie, nad którą obecnie pracujemy. Zazwyczaj pracujemy na ostatniej (najnowszej) wersji projektu.

Obszar **staging** – jest to obszar znajdujący się wewnątrz folderu „.git”, który przechowuje informacje, które zostaną zapisane w kolejnej migawce. Obszar ten nazywany jest również jako **Index**.

Standardowy cykl działania wygląda następująco:

- Modyfikujemy pliki w naszym obszarze roboczym/folderze roboczym
- Oznaczamy zmodyfikowane pliki, aby znalazły się w kolejnej migawce
- Wykonujemy **commit**, czyli na podstawie informacji zapisanych w indeksie tworzymy migawkę, która umieszczana jest w historii zmian naszego projektu.

Podstawowa instalacja i konfiguracja Git'a

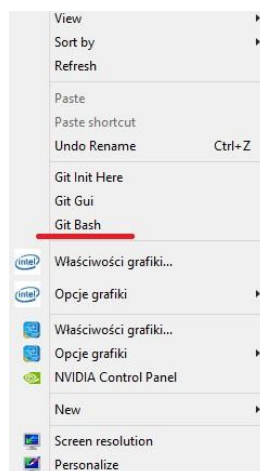
W celu rozpoczęcia pracy z Git'em musimy ściągnąć odpowiednie narzędzia. Na ćwiczeniach będziemy korzystać głównie z konsoli. Istnieje wiele narzędzi graficznych do współpracy z Git'em, jednak zazwyczaj nie obejmują one wszystkich dostępnych komend Git. Nauczenie się podstaw obsługi Git'a za pomocą konsoli pozwoli nam na łatwe zrozumienie działania dowolnego narzędzia graficznego do obsługi Git'a.

Plik instalacyjny znajdziemy na stronie:

<http://git-scm.com/download/win>

Ćwiczenie 1 – ustawiamy podstawowe informacje i tworzymy nowe repozytorium

Po zainstalowaniu Git'a pod prawym przyciskiem myszy powinniśmy mieć dostępne dodatkową opcję „Git Bash”.



Jej uruchomienie skutkuje dostępem do konsoli w której będziemy wpisywać wszystkie komendy Git'a.

Każda zmiana zapisywana przez Git'a musi być powiązana z osobą, która ją wykonała. Na początku musimy skonfigurować swojego użytkownika. To jego informacje będą związane z każdą wykonaną zmianą, czyli tzw. „commitem” (migawką). Wpisujemy komendę (oczywiście proszę wpisać swoje dane):

```
$ git config --global user.name "Imie Nazwisko"
$ git config --global user.email student@pjwstk.edu.pl
```

W ten sposób skonfigurowaliśmy ustawienia na poziomie globalnym. Teraz każdy wykonywany przez nas commit będzie zawierał powyższe informacje. Będziemy zamiennie używać terminu commit i migawka. Częściej jednak funkcjonuje nazwa commit. Podobne ustawienia możemy skonfigurować np. na poziomie poszczególnych projektów. Dzięki temu możemy funkcjonować jako różni użytkownicy w różnych projektach. Teraz jednak taka funkcjonalność nie jest nam potrzebna.

W celu sprawdzenia jakie są obecne ustawienia użytkownika możemy uruchomić:

```
$ git config --list
```

W celu sprawdzenia wartości konkretnego ustawienia możemy wpisać:

```
$ git config user.name
```

Jeśli chcemy uzyskać listę dostępnych komend możemy użyć instrukcji:

```
$ git help
```

Jeśli chcemy uzyskać pomoc na temat konkretnej instrukcji możemy użyć komendy:

```
$ git help config
```

Zakładanie nowego repozytorium

W tym ćwiczeniu założymy nowe repozytorium i przyjrzymy się temu, w jaki sposób rejestrować zachodzące w nim zmiany. Na początku poznamy komendy pracując z repozytorium lokalnym. Na kolejnych ćwiczeniach poznamy sposoby publikacji naszego kodu w jednym z darmowych serwisów takich jak Github lub Bitbucket. Dopiero takie rozwiązanie pozwoli nam w wygodny sposób pracować nad kodem zespołowo.

Najpierw za pomocą instrukcji „cd” przejdźmy do lokalizacji, w której chcielibyśmy przechowywać wersjonowane dane. Np:

A screenshot of a terminal window titled 'MINGW64:/c/Users/Knopers/Desktop/moje-repo'. The terminal shows the following commands and output: 'Knopers@Inugami MINGW64 ~/Desktop' followed by '\$ mkdir moje-repo', then 'Knopers@Inugami MINGW64 ~/Desktop' followed by '\$ cd moje-repo', and finally 'Knopers@Inugami MINGW64 ~/Desktop/moje-repo' followed by '\$ |'.

```
MINGW64:/c/Users/Knopers/Desktop/moje-repo
Knopers@Inugami MINGW64 ~/Desktop
$ mkdir moje-repo
Knopers@Inugami MINGW64 ~/Desktop
$ cd moje-repo
Knopers@Inugami MINGW64 ~/Desktop/moje-repo
$ |
```

Następnie w wybranej lokalizacji proszę uruchomić komendę:

```
$ git init
```

Komenda inicjalizuje w wybranej lokalizacji nowe, puste repozytorium. Po wejściu do naszego repozytorium powinniśmy odnaleźć folder „.git” (jest to folder ukryty, więc być może będziecie musieli zmienić stosowne uprawnienia w Windows’ie). W folderze znajdują się wszystkie niezbędne dla Git’a pliki. Tutaj przechowywana będzie historia naszego repozytorium. Usunięcie tego folderu oznacza usunięcie historii zmian. Każda osoba, której prześlemy całe repozytorium wraz z folderem „.git” będzie miała dostęp do pełnej historii zmian. Na tym polega główna zaleta Git’a. Nie występuje tutaj centralny węzeł (serwer) przechowujący pełną historię zmian. Dzięki temu mamy możliwość odtworzenia historii projektu z dowolnego, lokalnego repozytorium.

Skorzystajmy teraz z komendy:

```
$ git status
```

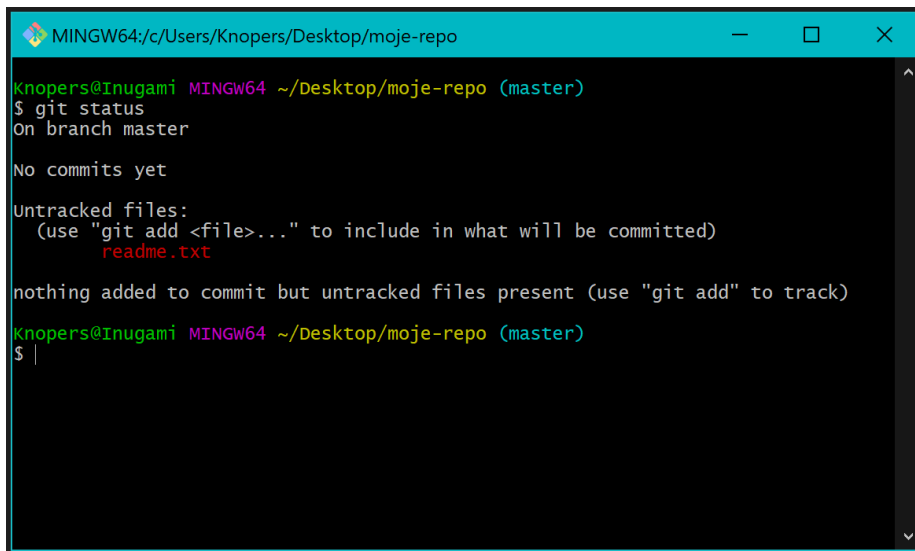
Będziemy często korzystać z tej komendy. Pozwala ona na podejrzenie kilku istotnych informacji. Po pierwsze pokazuje nam na jakiej gałęzi („branch’u”) jesteśmy.

Domyślną/główną gałęzią w większości repozytoriów jest gałąź „master”. Pojęcie gałęzi wyjaśnimy nieco później. To tutaj wyświetlane będą informacje o plikach znajdujących się w stanie „stage”, a także o tych, w których zaszły jakieś zmiany. Jednak na razie nie mamy żadnych plików w repozytorium.

Dodajmy do naszego repozytorium nowy, pusty plik „readme.txt”.

Z zawartością: „Zawartość 1”

Następnie jeszcze raz wywołajmy instrukcję „git status”.

A screenshot of a terminal window titled "MINGW64: c:/Users/Knopers/Desktop/moje-repo". The prompt is "Knopers@Inugami MINGW64 ~/Desktop/moje-repo (master)". The command "\$ git status" has been executed, resulting in the following output: "On branch master", "No commits yet", "Untracked files:", "(use \"git add <file>...\" to include in what will be committed)", "readme.txt", "nothing added to commit but untracked files present (use \"git add\" to track)". The prompt is now "\$ |".

```
MINGW64: c:/Users/Knopers/Desktop/moje-repo
Knopers@Inugami MINGW64 ~/Desktop/moje-repo (master)
$ git status
On branch master

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
Knopers@Inugami MINGW64 ~/Desktop/moje-repo (master)
$ |
```

Tym razem rezultat informuje nas, że w repozytorium znajduje się jeden plik, który nie jest „śledzony”. Aktualnie usunięcie tego pliku skutkowałoby jego stratą. W celu dodania nowego pliku do obszaru *staging’owego* należy skorzystać z instrukcji „git add”.

Instrukcja ta może posłużyć zarówno do dodawania pojedynczych plików, jak i całych folderów lub wszystkich plików w repozytorium. Poniżej przedstawiam kilka wariantów tej komendy. Wykorzystajmy dowolny z nich w celu dodania pliku do obszaru *stage*.

Instrukcja dodaje wszystkie pliki w stanie *untracked* lub *modified* do obszaru *stage*.

```
$ git add .
```

Instrukcja dodaje wszystkie pliki z rozszerzeniem „html” w stanie *untracked* lub *modified* do obszaru *stage*.

```
$ git add *.html
```

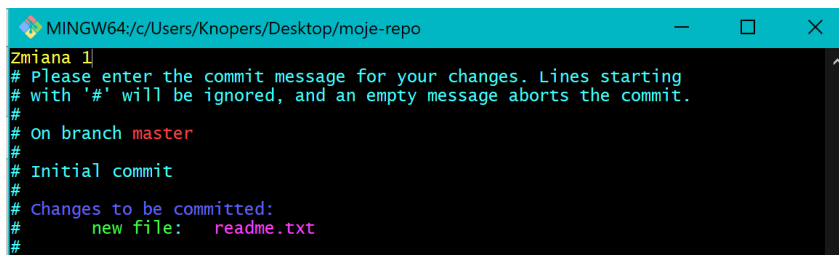
Instrukcja dodaje konkretny plik w stanie *untracked* lub *modified* do obszaru *stage*.

```
$ git add readme.txt
```

Następnie ponownie uruchamiamy instrukcję „git status”.


```
<ers/Desktop/moje-repo/.git/COMMIT_EDITMSG
-- WPROWADZANIE --
```

Następnie wpiszmy naszą wiadomość „Zmiana 1”.

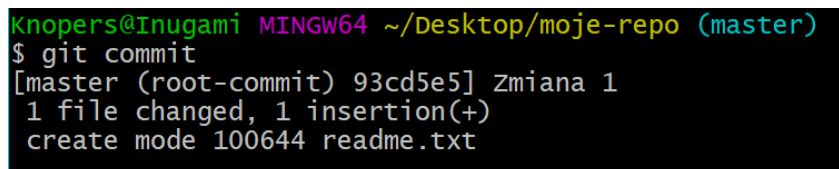


```
Zmiana 1
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   new file:   readme.txt
#
```

Na końcu klikamy „Esc” i wpisujemy:

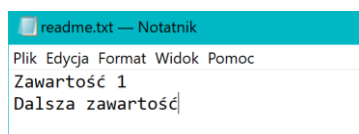
```
:wq
```

W ten sposób wyjdziemy z trybu edycji wiadomości commit’a i dodamy/zapiszemy go. Po zapisaniu każdy commit jest identyfikowany przez unikalny hash.



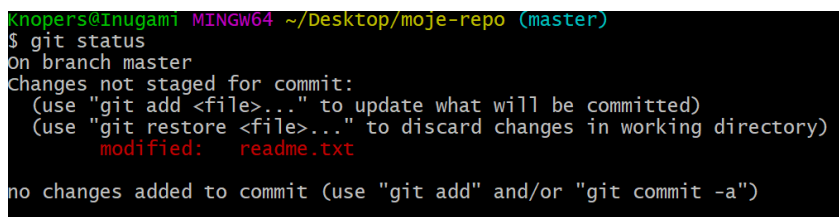
```
Knopers@Inugami MINGW64 ~/Desktop/moje-repo (master)
$ git commit
[master (root-commit) 93cd5e5] Zmiana 1
1 file changed, 1 insertion(+)
create mode 100644 readme.txt
```

Dodajmy kolejne zmiany w pliku „readme.txt”. Dodajmy następującą linie:



```
readme.txt - Notatnik
Plik Edycja Format Widok Pomoc
Zawartość 1
Dalsza zawartość
```

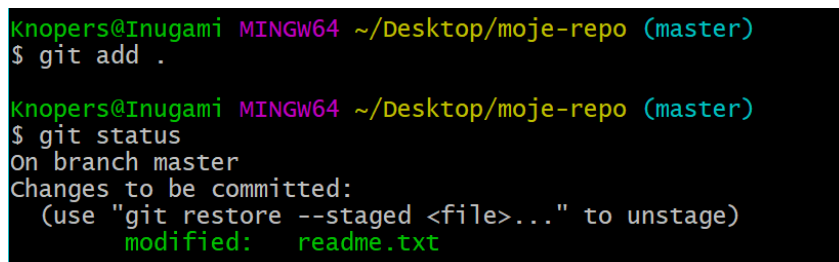
Wywołajmy ponownie komendę „git status”.



```
Knopers@Inugami MINGW64 ~/Desktop/moje-repo (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   readme.txt

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

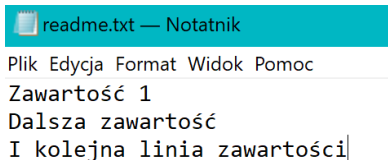
Widzimy, że git wykrył modyfikację wspomnianego pliku. Dodajmy zmiany w nim wprowadzone do obszaru stage. Wywołujemy ponownie komendę „git add .”. Następnie wpiszmy komendę „git status”:



```
Knopers@Inugami MINGW64 ~/Desktop/moje-repo (master)
$ git add .

Knopers@Inugami MINGW64 ~/Desktop/moje-repo (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   readme.txt
```

Jak widać w obszarze *stage* mamy modyfikację pliku „readme.txt”. Teraz ponownie wprowadźmy kolejną modyfikację w tym samym pliku.



Ponownie wywołajmy komendę „git status”.

```
Knopers@Inugami MINGW64 ~/Desktop/moje-repo (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   README.txt

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

Jak widać ten sam plik w stanie **modified** znajduje się teraz w dwóch miejscach. Zarówno w strefie **stage**, jak również w strefie **committed**. Co to oznacza? Otóż w strefie **stage** znajduje się plik i zawarte w nim zmiany z dokładnego punktu w czasie kiedy wywołaliśmy metodę **add**. Jeśli wprowadziliśmy po tej komendzie kolejne zmiany to musimy ponownie wywołać metodę „git add”. Wywołajmy ją zatem ponownie.

```
$ git add .
```

Następnie metoda git status powinno znowu pokazać jedynie zmiany w obszarze **stage**.

Dodajmy teraz kolejną migawkę, czyli kolejnego commit’a. Tym razem skorzystajmy ze skróconego zapisu komendy:

```
$ git commit -m "Druga zmiana"
```

```
Knopers@Inugami MINGW64 ~/Desktop/moje-repo (master)
$ git commit -m "Druga zmiana"
[master 5a80359] Druga zmiana
1 file changed, 3 insertions(+), 1 deletion(-)
```

W ten sposób dodaliśmy kolejny **commit**. Aktualnie mamy w bazie dwa **commity**. Aby podejrzeć historię naszego repozytorium wystarczy wpisać „git log”. Komenda zwróci nam listę wszystkich commit’ów. Każdy commit zawiera hash, autora, datę i wiadomość.

```
Knopers@Inugami MINGW64 ~/Desktop/moje-repo (master)
$ git log
commit 5a803599cf4ad2284724b937c625fc053c7c344e (HEAD -> master)
Author: Mateusz Knopers Knop <knopers.12@o2.pl>
Date:   Fri Jan 31 10:22:41 2020 +0100

    Druga zmiana

commit 93cd5e501b9aa64dfe141207574b14a60f5b746a
Author: Mateusz Knopers Knop <knopers.12@o2.pl>
Date:   Fri Jan 31 10:10:35 2020 +0100

    Zmiana 1
```

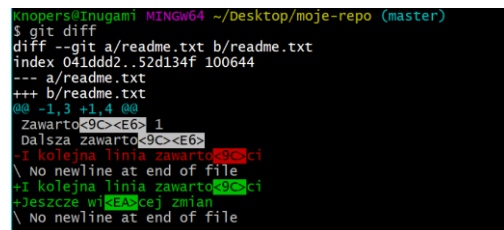
Wiemy już w jaki sposób zapisywać zmiany do naszego repozytorium. Dodajmy teraz ponownie zmiany. Dodajmy nowy plik i wprowadźmy zmiany w pliku „readme.txt”.

Wywołajmy metodę „git status”.

Mamy dwie zmiany. Jeden to nowy plik w stanie **untracked** – w ogóle nie śledzony. Kolejny plik jest w stanie **modified**.

Sprawdźmy teraz w jaki sposób kod w obszarze roboczym różni się od ostatniej migawki/commita w bazie. Wpiszmy komendę „git diff”.

```
$ git diff
```



```
Knopers@Inugami MINGW64 ~/Desktop/moje-repo (master)
$ git diff
diff --git a/readme.txt b/readme.txt
index 041ddd2..52d134f 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,3 +1,4 @@
 Zawartość
 Dalsza zawartość
-I kolejna linia zawartości
\ No newline at end of file
+I kolejna linia zawartości
+Jeszcze więcej zmian
\ No newline at end of file
```

Komenda wyświetli nam dokładne zmiany jakie zaszły w poszczególnych plikach od ostatniego commit’a. Innymi słowy porówna aktualny stan naszego repozytorium z ostatnim stanem zapisanym w bazie.

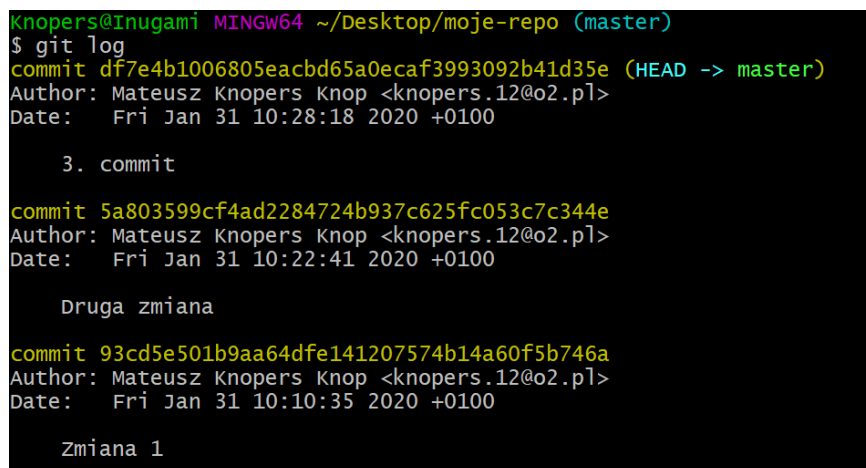
Widok w konsoli może wydać się mało czytelny, jednak jest wiele darmowych narzędzi, które potrafią w niezwykle intuicyjny sposób zaprezentować zmiany jakie zaszły w kodzie.

Dokonajmy zatem kolejnego zatwierdzenia zmian:

```
$ git add .
```

```
$ git commit -m "3. commit"
```

Teraz spróbujemy cofnąć się do poprzedniej wersji. W tym celu potrzebna nam będzie unikalna informacja determinująca jednoznacznie punkt do którego chcemy wrócić. Jest to oczywiście suma kontrolna commita. W celu jej pozyskania możemy ponownie wykonać komendę „git log”.



```
Knopers@Inugami MINGW64 ~/Desktop/moje-repo (master)
$ git log
commit df7e4b1006805eacbd65a0ecaf3993092b41d35e (HEAD -> master)
Author: Mateusz Knopers Knop <knopers.12@o2.pl>
Date: Fri Jan 31 10:28:18 2020 +0100

    3. commit

commit 5a803599cf4ad2284724b937c625fc053c7c344e
Author: Mateusz Knopers Knop <knopers.12@o2.pl>
Date: Fri Jan 31 10:22:41 2020 +0100

    Druga zmiana

commit 93cd5e501b9aa64dfe141207574b14a60f5b746a
Author: Mateusz Knopers Knop <knopers.12@o2.pl>
Date: Fri Jan 31 10:10:35 2020 +0100

    Zmiana 1
```

Chcąc podejrzeć repozytorium z commitu „5a803599cf4ad2284724b937c625fc053c7c344e” z opisem „Druga zmiana” należy wykonać komendę:

```
$ git checkout 5a803599cf4ad2284724b937c625fc053c7c344e
```

Zostanie nam utworzona nowa gałąź o takiej nazwie i zostaniemy do niej przeniesieni, a co za tym idzie, będziemy widzieć repozytorium z czasu commitu drugiego.

Aby powrócić do aktualnego stanu należy wykonać zmianę gałęzi na której byliśmy, czyli master:

```
$ git switch master
```

Więcej o komendach git i pracy zdalnej:

<https://git-scm.com/book/pl/v2/Podstawy-Gita-Praca-ze-zdalnym-repozytorium>