



Teoria i podstawowe polecenia

czyli zrozumieć fundamenty Gita



Definicje



Repozytorium - baza danych projektu

Repozytorium (często określane też repo [czyt. ripo]) to historia stanów projektu, obejmująca wszystkie commity i stworzone w ich wyniku snapshot'y oraz relację między nimi (kolejność).

Repozytorium jest przechowywane w ukrytym katalogu `.git` projektu.

```
commit 39242ee6cdb1803f555c703990caf5bc5ddcac69 (HEAD -> master)
Author: Bartek Borowczyk <bartek@websamuraj.pl>
Date:   Tue Mar 5 09:16:27 2019 +0100

    create contact.html

commit 6aa335611cee750cbaacc29c2fa01bc652c75e4d
Author: Bartek Borowczyk <bartek@websamuraj.pl>
Date:   Tue Mar 5 09:14:42 2019 +0100

    initial commit
```

- hooks
- info
- logs
- objects
- refs
- COMMIT_EDITMSG
- config
- description
- HEAD
- index



Commit

Zapisanie (wysłanie, zgłoszenie) w repozytorium informacji zawartych w commicie (pliki + metadane) i utworzenie nowego snapshota (kompletnego zapisu stanu projektu).

Commit można potraktować potocznie jako zapis (save) nowej wersji pliku (plików). Każdy commit można zidentyfikować. Każdy commit jest obiektem w bazie danych repozytorium Git.

Oczywiście możemy określić jakie zmiany w plikach ma uwzględnić commit określając zawartość staging area (poczekalni, indeksu)



Snapshot

(obiekt projektu, migawka, stopklatka, zamrożenie, synonim commitu)

Zapisany w repozytorium pełny stan projektu (w istocie zapisuje stan indeksu w chwili commitu). Snapshot zawiera kompletny projekt (oczywiście Git to optymalizuje) wraz z metadanymi, przy czym znajdują się w nim też pliki (czy też referencje do plików), który się nie zmieniły.

Snapshot zawiera też informację o strukturze projektu (nazwy, katalagi, zagnieżdżenia) oraz odniesienie do commitu poprzedzającego.

W każdej chwili użytkownicy repozytorium mają dostęp do wszystkich takich snapshotów (obiektów).

Snapshot można też traktować (i taka bardzo często się robi) jako synonim commitu (commit można traktować jako snapshot). Wiele użytkowników Gita nie używa nawet określenia snapshot.



Trzy obszary repozytorium

- Working directory
- Staging Area (index)
- Git folder (Git repository)



Trzy obszary repozytorium

Working directory - katalog roboczy - nasze pliki i foldery (zetkniesz się też z określeniem working tree)

Staging Area (index)

Git folder (Git repository)



Trzy obszary repozytorium

Working directory

Staging Area (index) - poczekalnia, przechowalnia, indeks. "Pośredni" obszar zawierająca pliki (z zawartością aktualną na chwilę umieszczenia w przechowalni) dla których chcemy wykonać commit. Patrząc na indeks szerzej, to zawiera on także te pliki których nie modyfikowaliśmy od ostatniego commita. Jeśli więc po commicie nic nie zmienialiśmy w katalogu roboczym, to w indeksie są wszystkie pliki w takiej samej wersji.

Git folder (Git repository)



Trzy obszary repozytorium

Working directory

Staging Area (index)

Git folder (Git repository) - stricte repozytorium, katalog przechowujący repozytorium.



Trzy byty

Przechowywane w ukrytym katalogu `/.git`

katalog roboczy

Główny katalog z naszym projektem (i podkatalogami) objętym repozytorium. Dla Git jest to folder który zawiera katalog `.git`, czyli w którym zainicjalizowaliśmy repozytorium.

poczekalnia
(indeks)

Elementy, który mają być objęte kolejnym commitem. Po wykonaniu commita, staging area jest czyszczona (w istocie tuż po commicie indeks zawiera pliki w tej samej wersji co w commit).

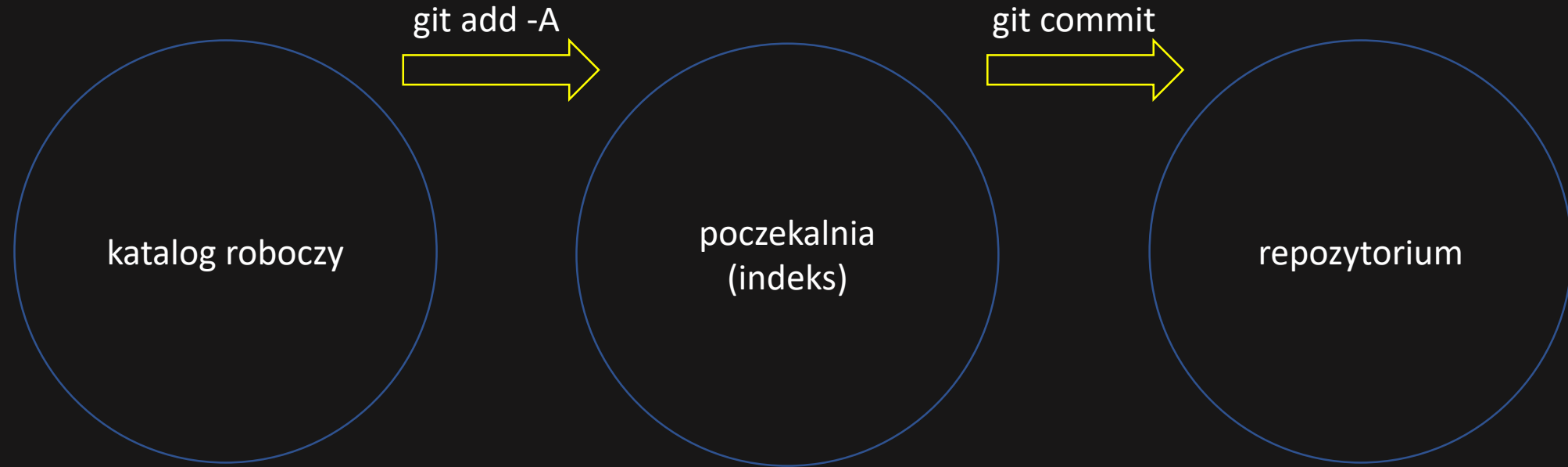
repozytorium

Przechowywanie historii projektów - ich stanów, struktury i metadanych



Przenoszenie plików w wersji z katalogu roboczego do poczekalni (zastępują one pliki, które już tam są)

Zapisanie aktualnego stanu projektu (zawartość indeksu)



Pliki nad którymi pracuje developer

Pliki, które będą objęte kolejnym commitem

Zapisane wersje projektów (opisane w indeksie) w momencie commita i ścieżki do plików projektu. Umieszczenie kolejnego commita na liście commitów.



Warto zapamiętać

katalog roboczy
working directory

Aktualne zmiany, to na czym
pracujemy

poczekalnia
(indeks)
staging area


Zmiany, które chcemy zatwierdzić, inne
niż w repozytorium, ale też niekoniecznie
aktualne (czyli niekoniecznie te same co
w katalogu roboczym)
W skrócie: przechowuje zmiany do
zatwierdzenia w commicie

repozytorium
git repository

Zmiany zatwierdzone i
przechowywane w bazie danych
Gita



Obiekty w repozytorium



repozytorium
.git/objects

```
echo "Bartek" >> name.html  
git add . //na tym etapie powstanie obiekt  
git commit -m "add file with name Bartek"
```

```
git cat-file -p 3ec46f01213700b65cc0bb2e3097907ca9ef075f  
B a r t e k  
//taki wynik w PowerShell
```

Git przechowuje dane jako obiekty posiadające swój hash (identyfikator obiektu). Identyfikator dotyczy commitów czy plików. Taka sama treść (i taki sam zapis) da taki sam identyfikator, nawet jeśli nazwa pliku będzie inna.



4 stany plików w repozytorium (dla czytelności)

- nieśledzony (untracked) - gdy dodamy nowy plik (lub usuniemy ze śledzonych)
- śledzony niezmodyfikowany (tracked unmodified) - pliki które zostały już dodane do repozytorium (gdy wykonamy commit) i od tego czasu w katalogu roboczym nie był zmieniany
- śledzony zmodyfikowany (tracked modified) - plik dodany wcześniej do repozytorium, a aktualnie zmieniony (edytowany) w folderze roboczym
- śledzony w poczekalni (tracked staged) - plik, który został dodany do indeksu (stage area) i oczekuje na commit (wcześniej mógł to być nieśledzony czy śledzony zmodyfikowany)



5 typ - ignorowany?

- **nieśledzony (untracked)** - gdy dodamy nowy plik
- **śledzony niezmodyfikowany (tracked unmodified)** - pliki które zostały już dodane do repozytorium (gdy wykonamy commit) i od tego czasu w katalogu roboczym nie był zmieniany
- **śledzony zmodyfikowany (tracked modified)** - plik dodany wcześniej do repozytorium, a aktualnie zmieniony (edytowany) w folderze roboczym
- **śledzony w poczekalni (tracked staged)** - plik, który został dodany do indeksu (stage area) i oczekuje na commit (wcześniej mógł to być nieśledzony czy śledzony zmodyfikowany)

Możemy jeszcze dodać "piąty typ" (czy trzeci, jeśli poprzednie dzielimy na śledzone i nieśledzone), a mianowicie pliki ignorowane, które wskażemy w specjalnym pliku `.gitignore`. Plik `.gitignore` określa jakie pliki są ignorowane przez repozytorium (bo są tymczasowe, bo generują się automatycznie itd.). Pliki ignorowane przez git i określone w `.gitignore` muszą mieć status "nieśledzone".

```
1 #lista użytkowników
2 list.txt
3
4 #dane użytkowników
5 /users
```



pliki nieśledzone - untracked

```
$ git status
On branch master

No commits yet

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

        index.html
        style.css

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

W przykładzie stworzyliśmy **dwa nowe pliki**. Takie nowe pliki nie są z automatu dodane do staging area (indeksu), ich zmiany nie są śledzone, **Git wie jedynie, że istnieją** i informuje nas o tym.

Jest to stan początkowy każdego nowego pliku.

Stan pliku **index.html**: **nieśledzony**, śledzony niezmodyfikowany, śledzony zmodyfikowany, śledzony w poczekalni



pliki śledzone w poczekalni - tracked staged

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: index.html

Untracked files:

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

style.css

Po dodaniu za pomocą polecenia `git add index.html` plik index.html zmienił stan na śledzony i jednocześnie znalazł się w staging area (indeks, poczekalnia). Warto wiedzieć, że możemy usunąć też plik z poczekalni (i przywrócić jego poprzednią zawartość) za pomocą polecenia `git rm --cached index.html`

Stan pliku `index.html`: nieśledzony, śledzony niezmodyfikowany, śledzony zmodyfikowany, **śledzony w poczekalni**



pliki śledzony niezmodyfikowany - tracked unmodified

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

    style.css

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

Po zrobieniu commita (np. `git commit --message="initial commit"`) nasz plik `index.html` został usunięty z indeksu (nie ma już status "śledzony w poczekalni") i ma status śledzony niezmodyfikowany. Plików z tym statusem nie widzimy po wpisaniu polecenia `git status`. Każda zmiana jednak w tym pliku będzie śledzona (w przeciwieństwie do plików nieśledzonych, w naszym przypadku jest to ciąge `style.css`, z którym nic nie robiliśmy)

Stan pliku `index.html`: nieśledzony, **śledzony niezmodyfikowany**, śledzony zmodyfikowany, śledzony w poczekalni



plik śledzony zmodyfikowany - tracked modified

```
on branch master
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:   index.html

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

        style.css

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

Jakakolwiek zmiana w pliku śledzonym (np. coś dodamy w nim czy usuniemy) spowoduje, że staje się on plikiem śledzonym zmodyfikowanym. Taki plik możemy dodać do indeksu (staging area) i przygotować do commitu (git add index.html). Możemy też cofnąć zmiany (do stanu z poprzedniego commitu) za pomocą polecenia git checkout -- index.html

Stan pliku [index.html](#): nieśledzony, śledzony niezmodyfikowany, **śledzony zmodyfikowany**, śledzony w poczekalni



plik w indeksie i plik zmodyfikowany jednocześnie?

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

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:   index.html

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

    style.css
```

1. tworzę plik index.html
2. dodaje go do indeksu (git add index.html)
-- na tym etapie jest gotowy do commitu
-- na zielono w konsoli
3. ponownie edytuję index.html
-- pojawia się w jako śledzony zmodyfikowany. To aktualna wersja w katalogu roboczym.

Wersja w staging area jest taka jak w chwili wykonania polecenia git add. Tak więc mamy dwie wersje! (Zwykły) Commit obejmie wersję znajdującą się w staging area

Stan pliku `index.html`: nieśledzony, śledzony niezmodyfikowany, **śledzony zmodyfikowany**, **śledzony w poczekalni**

P.s. zwróć uwagę, że możemy usunąć z indeksu (poczekalni, staging area) ten plik za pomocą polecenia `git reset HEAD index.html` (lub `git reset -- index.html`)



Komendy Git

podstawowe, które musisz znać i wyjaśnienia



Konfiguracja Gita

```
git config --global user.name "Jan Nowak"
```

```
git config --global user.email jan.nowak@jakasdomena.com.pl
```

//oczywiście możemy wielokrotnie

//w pliku w katalogu domowym (~/.gitconfig - w Windows users/name/)

```
git config --unset --global user.email //usunięcie
```

```
git config user.name "JanekN" //do pojedynczego repozytorium
```

// w pliku .git/config

```
git config user.name //sprawdzamy jaka nazwa będzie użyta w danym repozytorium
```

```
git config --global core.editor //jaki edytor jest ustawiony globalnie dla Git
```



Konfiguracja Gita

c:/Users/work/.gitconfig //plik z globalną konfiguracją

cat ~/.gitconfig //zawartość pliku

git config --global --list //lista ustawień globalnych

git config --list //lista ustawień dla danego repozytorium (obejmująca więc też ustawienia globalne jeśli nie ma lokalnych)



git init

Tworzy nowe repozytorium w katalogu, w którym polecenie jest wywołane.

Komunikat: Initialized empty Git repository in /path

Oczywiście repozytorium możemy stworzyć w folderze w którym są już pliki (wtedy wszystkie pliki w katalogu roboczym będą miały status początkowy "nieśledzone")

Repozytorium znajduje się w katalogu /.git. Katalog w którym znajduje się katalog git pełni rolę katalogu roboczego.



git init path

możemy też podać ścieżkę, wtedy stworzymy też katalog (o ile nie będzie istniał)

```
git init counter_app
```

```
git init projekty/serwis-o-pogodzie
```

```
git init D:/apps/gop
```

Warto zapamiętać:

```
git init -h //jakich parametrów możemy użyć przy różnych poleceniach w tym przypadku init
```

```
git init --help //dokumentacja
```



git clone

Możemy stworzyć nowe repozytorium, albo sklonować (skopiować) już istniejące. Najczęściej kopiujemy zdalne repozytorium za pomocą protokołów HTTP czy SSH.

Klonowanie to stworzenie nowego repozytorium, które jest identyczne.



git clone

```
D:\kurs-git>
```

//folder z repozytorium

```
D:\kurs-git-clone> git clone ..\kurs-git .
```

//wskazaliśmy ścieżkę do repo i katalog

Tym sposobem mamy kompletną kopię repozytorium

```
D:\ > git clone kurs-git kopia-kurs-git
```



git add

Dodawanie aktualnej wersji plików z katalogu roboczego do staging area (indeksu, poczekalni)

Swoim zakresem może obejmować pliki zmodyfikowane (zmodyfikowane śledzone) i nowe pliki (nieśledzone).

Wymaga podania parametru



git add

Wszystkie pliki (zmodyfikowane i nieśledzone), także z podfolderów.

```
git add -A
```

```
git add --all
```

```
git add . //katalog bieżący z podkatalogami
```

Konkretne pliki

```
git add file1 file2
```



git status

Przegląd aktualnej zawartości zmian w plikach nadzorowanych przez repozytorium (także w katalogu roboczym).

Pokazuje nowe pliki i pliki w jakiś sposób zmodyfikowane.

```
PS C:\Users\iMikser\git_course\git1> git status
```

```
On branch master
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
new file:   a/podkatalog.html
```

```
renamed:    index.html -> indexx.html
```

```
modified:   indexx.html
```



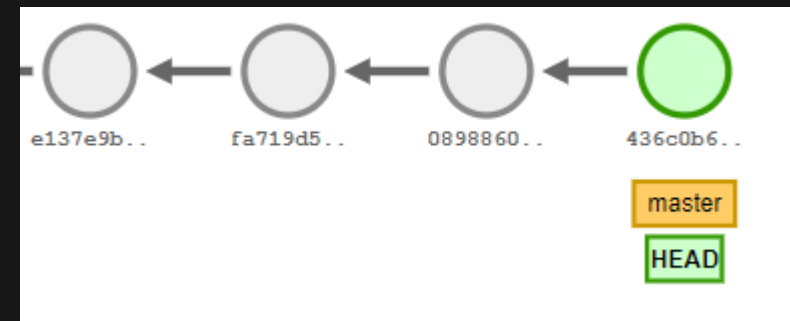
git commit

Zatwierdzenie wprowadzonych do indeksu zmian i zapisanie ich w repozytorium Git.
Git commit pozwala aktualizować repozytorium.



co robi commit

- tworzy obiekt commita w bazie
- tworzy informacje o nowym commicie w gałęzi, w której jesteśmy
- ustawia HEAD na najnowszą zmianę.



źródło: <https://onlywei.github.io/explain-git-with-d3/>



git commit -m

Git wymaga by commit zawierał komentarz do zmiany. Jeśli tego nie zrobimy w poleceniu (parametr m), to otworzy się domyślny edytor, w którym możemy podać opis.



opisy commitów

```
git commit -m "create new title in chapter 1
```

```
lorem ipsum"
```

Opis może być z więcej niż z jednego wiersza (nie powinien być dłuższy niż 40 do 60 znaków).

Wiersze nie powinny być za długie. Dodatkowe wiersze po pustej linii.

Po angielsku. W czasie present simple (pierwszy wiersz)

Sam commit często nie jest zbyt duży (nie powinien być) więc ilość znaków też pełni rolę "cenzora".



git commit -a (git commit --all)

Dwa kroki w jednym. Przenosi do indeksu wszystkie śledzone zmienione pliki z katalogu roboczego (git add -A) i robi commit (git commit)

podobnie jeśli do git commit dołączymy nazwę pliku, np.:

`git commit plik.js`

Commit będzie się składał z pliku z katalogu roboczego (może to być ta sama wersja co w indeksie, ale niekoniecznie). Plik taki musi być śledzony.



git commit --amend

Poprawki w ostatnim (najnowszym) commicie. Zmiany w zawartości i message, nowe metadane też.

Edytujemy ostatni commit (w istocie edycja oznacza usunięcie z gałęzi tego commita i na jego miejsce dodanie nowego).

Jeśli chcemy zmienić tylko komentarz, to możemy napisać:

```
git commit --amend -m "our new message"
```



git log

historia commitów (od najnowszego) znajdujących się w repozytorium.

```
commit 21a38830476237dc3e8ddfa58508efd40134bfc1 (HEAD -> master)
Author: Barti B <bartek@websamuraj.pl>
Date:   Sun Mar 10 13:07:30 2019 +0100

    new title in chapter1
```

Pamiętaj o "q" (od quit - porzucić), które pozwoli Ci powrócić do aktywnego wiersza poleceń.



git log

git log --oneline //w jednej linii podstawowe informacje

git log --oneline 10 //ile linii (od najnowszych)

git log --since "2019" //od 2019

git log --since="5.4.2019" //od konkretnej daty; innym zapisem

```
21a3883 (HEAD -> master) new title in chapter1
5b8d8d0 Create chapter1 in book
9f92dcb new file
5cb8c4d delete plik.txt
```



git log --grep //wyszukiwanie commitów

git log --grep "plik.txt"

//wyszukiwanie w message, wielkość liter ma znaczenie

git log --grep="delete"

//jw, ale innym zapisie

```
PS C:\Users\iMikser\Desktop\gitgit> git log --grep "create"
commit 1c6a7fd1022a7ac22695ec6bee2c27c252a809b1 (HEAD -> master)
Author: Barti B <bartek@websamuraj.pl>
Date:   Sun Mar 10 13:40:31 2019 +0100

    create index.html
```



git log --stat

które pliki zmienił commit i w jaki sposób
wyświetla listę commitów jak git log

```
PS C:\Users\iMikser\Desktop\gitgit> git log --stat
commit 21a38830476237dc3e8ddfa58508efd40134bfc1 (HEAD -> master)
Author: Barti B <bartek@websamuraj.pl>
Date:   Sun Mar 10 13:07:30 2019 +0100

    new title in chapter1

test.txt | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```




git show

Co się zmieniło (szczegółowo) w danym commicie (w najnowszym). Możemy też wskazać o jaki commit nam chodzi, np.

`git show 5b8d8d0febd48406dc3dcc93599e23177d60b01e`

`git show 5b8d8d0`

```
PS C:\Users\iMikser\Desktop\gitgit> git show
commit 21a38830476237dc3e8ddfa58508efd40134bfc1 (HEAD -> master)
Author: Barti B <bartek@websamuraj.pl>
Date:   Sun Mar 10 13:07:30 2019 +0100

    new title in chapter1

diff --git a/test.txt b/test.txt
index ebf7998..bdec3d9 100644
--- a/test.txt
+++ b/test.txt
@@ -1,3 +1,3 @@
-<EF><BB><BF>Rodział 1
+<EF><BB><BF>Rodział 1 - Czy wszechświat będzie miał<C5><82> swój koniec?

Lorem ipsum...
\ No newline at end of file
```



git diff (porównanie)

git diff //pokazuje domyślnie różnicę między plikami - wersją zmodyfikowaną w katalogu roboczym a wersją w indeksie

Świetnie pokazuje co zostało zmodyfikowane względem tego co aktualnie w staging area.

```
PS C:\Users\iMikser\Desktop\gitgit> git diff
diff --git a/test.txt b/test.txt
index bdec3d9..827852b 100644
--- a/test.txt
+++ b/test.txt
@@ -1,3 +1,3 @@
-␣␣␣Rodział 1 - Czy wszechświat będzie miał ␣␣␣ swój koniec?
+␣␣␣Rodział 1 - Czy wszechświat będzie miał swój koniec?

-Lorem ipsum...
\ No newline at end of file
+Co␣␣␣o ma początek, powinno mieć też koniec, ale...
\ No newline at end of file
PS C:\Users\iMikser\Desktop\gitgit>
```



git diff (porównanie)

```
git diff nazwa-pliku //konkretny plik
```

`git diff --cached` //pliki w stage (pliki sledzone w poczekalni) z plikami z repozytorium (domyślnie z tymi w HEAD, czyli ostatniego commitu w gałęzi). Zamiast `cached` można użyć też parametru `--staged`, który robi dokładnie to samo.

Możliwość porównania plików w różnych commitach

```
git diff 852ff1d 962a5ab nazwa-pliku //porównanie wersji z commitów
```



git diff

git diff --staged

git diff *nazwa-pliku*



Komendy Git cz.2

polecenia, które warto znać i jeszcze więcej teorii



git rm plik

Usuwa pliki z katalogu roboczego i wersję z indeksu. Informacją o tym znajduje się w staging area (najbliższy commit to uwzględni). Informację o tym zobaczymy po wyświetleniu statusu repozytorium (git status). Oczywiście nie usuwa takiego pliku z historii w repozytorium.

```
PS C:\Users\iMikser\Desktop\gitgit> git rm index.html  
rm 'index.html'
```

```
PS C:\Users\iMikser\Desktop\gitgit> git status  
On branch master  
Changes to be committed:  
  (use "git reset HEAD <file>..." to unstage)  
  
    deleted:    index.html
```

Samo polecenie **rm index.html**
usunie tylko plik z katalogu
roboczego



git rm --cached plik

Usuwa pliki z indeksu, ale nie z katalogu roboczego (po tym poleceniu plik ma stan nieśledzony).

Można więc powiedzieć, że robi 1 z 2 etapów, które wykonuje git rm plik.

rm file // usuwa z katalogu roboczego.

git rm -- cached file // usuwamy z indeksu.

git rm file //usuwa z katalogu roboczego i z indeksu.



git mv index.txt index.html

Zamiana nazwy pliku. Przy czym w praktyce widzimy, że mamy zadanie renamed w następnym commicie, które oznacza polecenie usunięcia jednego pliku i dodania nowego.

```
PS D:\kurs-git> git mv page.html index.html
PS D:\kurs-git> git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    page.html -> index.html
```




git checkout plik

Zmiany z wersji roboczej są usuwane i przywracana jest (w katalogu roboczym) wersja, które znajduje się w indeksie. W praktyce jeśli nie mamy żadnej zmiany w staging area to wersja jaka była po ostatnim commicie.

STYLE.CSS

katalog roboczy

indeks

```
p {  
  font-size: 20px;  
  text-align: center;  
}
```

```
p {  
  font-size: 20px;  
}
```



git checkout style.css

```
p {  
  font-size: 20px;  
}
```



git checkout -- plik

lepiej tak `git checkout -- plik`, niż tak: `git checkout plik`

Bezpieczniej z dwoma kreskami po poleceniu checkout. Bez dwóch kresek Git nie zawsze uzna, że kolejnym parametrem będzie plik.

```
git checkout -- *.txt
```

```
git checkout HEAD -- plik
```

//do katalogu roboczego przywracana jest wersja pliku z ostatniego commita



git checkout 5a33dd2

Przywrócenie indeksu z tego commitu. W praktyce nasz katalog roboczy po takiej operacji będzie się składał z indeksu w chwili wykonania commita o danym identyfikatorze.

```
git checkout id-commita // ustawienie HEAD na tym commicie  
git checkout nazwa-brancha // przełączenie na inną gałąź
```



git reset / git reset HEAD

Usuwa pliki ze staging area (pliki są nadal śledzone i są oczywiście w katalogu roboczym).

Przywracamy stan indeksu do stanu po ostatnim commicie, więc wszystko co dodaliśmy do indeksu (staging area, poczekalnia) już się po tej operacji w nim nie znajduje.

Użycie wskaźnika HEAD jest czytelniejsze, wiemy do czego wracamy (stanu po ostatnim commicie). Ponadto możemy spokojnie użyć po nim ścieżki do pliku (przy użyciu po reset mogą być problemy i wtedy lepiej `git reset -- nazwa pliku`)



git reset / git reset HEAD

git reset

git reset HEAD

Alternatywnie wskazujemy plik/pliki/katalog, którego zmiany chcemy usunąć ze staging area (przywrócić pierwotną wersję).

git reset -- plik

git reset HEAD plik



Warto zapamiętać

`git rm plik`

usuwa plik z indeksu (staging area) i z katalogu roboczego

`git rm --cached plik`

usuwa plik z indeksu, plik staje się nieśledzony.

`git checkout -- plik`

do katalogu roboczego przywracana jest wersja, która znajduje się aktualnie w indeksie

`git reset HEAD plik`

usuwa z indeksu zmiany (przywracamy do indeksu wersję jaka była po ostatnim commicie)



I na koniec jeszcze coś

ale ciągle podstawy



.gitignore

Plik, najlepiej w katalogu głównym

Określamy listę plików i folderów, które nie trafią do repozytorium.

.gitignore obejmie tylko pliki nieśledzone, więc każdy nowo dodany plik (katalog z plikami), by był ignorowany, musimy dodać do .gitignore. Jeśli pliki są już śledzone, musimy zmienić ich stan na nieśledzone).



gałąź master

git branch

```
$ git branch  
feature1  
* master
```

Na jakim branchu (gałęzi) się znajdujemy.

Początkowo mamy jedną gałąź z nazwą master. Po wpisaniu git branch dostaniemy informację o istniejących gałęziach (by to zobaczyć wcześniej musimy zrobić choć jeden commit)



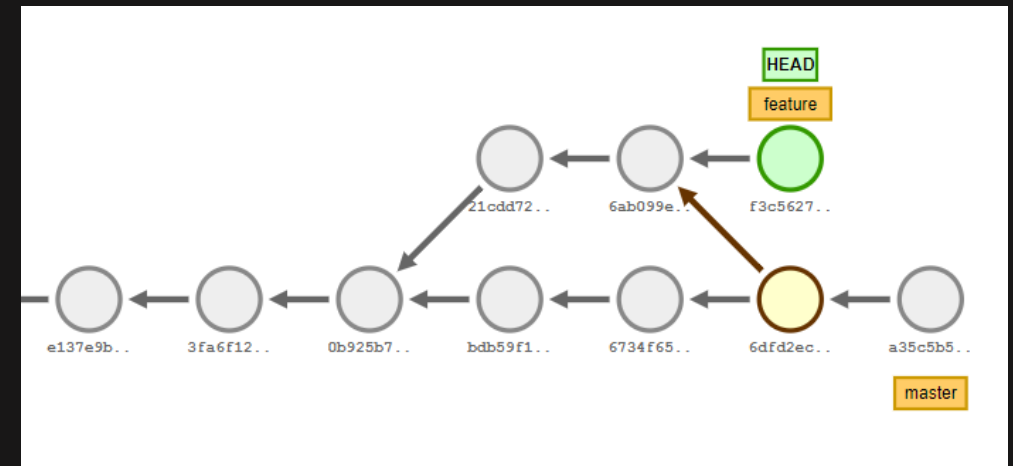
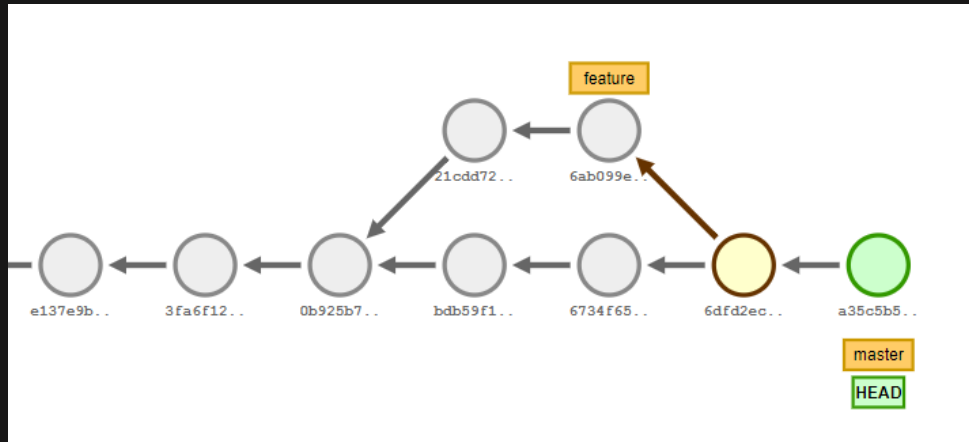
branches - gałęzie

Gałąź główna - master (pierwsza gałąź w naszym repozytorium)

Tworzenie osobnych gałęzi służy rozwijaniu niezależnych/testowych/pobocznych elementów projektu.

Gałęzie mogą być scalane - merge

Gałęzie mogą mieć zupełnie różną zawartość.





Git branch - przykładowe polecenie

git branch //lista wszystkich branchów

git branch nazwa-nowego-brancha //tworzymy nową gałąź

git checkout nazwa-istniejącego-brancha //przełączamy się na inną gałąź

git merge nazw-brancha //łączenie gałęzi na której jesteśmy ze wskazaną gałęzią.



HEAD

określa do której gałęzi się odnosimy

ref: refs/heads/master

//zawartość pliku .git/HEAD wskazująca na aktualną gałąź (oczywiście może na inną np. refs/heads/news-page gdy gałąź aktywna to news-page)

Head wskazuje też na bieżący (ostatni) commit (wierzchołek gałęzi). Kolejny commit odniesie się do HEAD (i sam stanie się HEAD)



Git push

przekazywanie zmian z jednego repozytorium (naszego lokalnego) do innego repozytorium (zdalnego).

Aktualizujemy repozytorium które sklonowaliśmy (zdalne)



Git pull

Pobieranie zmian z jednego repozytorium (zdalnego) na inne (nasze lokalne).

Nasze sklonowane repozytorium jest aktualizowane.



Przejdźmy do VSC