

System kontroli wersji GIT

Spis treści

1.	Wprowadzenie	2
1.1.	Linus Torvalds – twórca systemu GIT	2
1.2.	Najważniejsze cechy GITa	2
1.3.	Stany plików w GIT	2
1.4.	Terminologia	3
1.5.	Obiekty GITa	3
1.6.	Git Bash	4
2.	Instalacja GITa	4
3.	Konsola Git Bash	11
3.1.	Konfiguracja GITa	11
3.2.	Rejestrowanie zmian w repozytorium	12
3.2.1.	Sprawdzenie stanu plików w repozytorium	12
3.2.2.	Śledzenie nowych plików	13
3.2.3.	Dodawanie zmian do poczekalni	14
3.2.4.	Podgląd dokonanych zmian	15
3.2.5.	Zatwierdzanie zmian	16
3.2.6.	Historia zmian	17
3.3.	Rozszerzone mechanizmy	20
3.3.1.	Cofanie zmian	20
3.3.2.	Tagowanie źródeł	21
3.3.3.	Ignorowanie plików	21
3.4.	Gałęzie w Gicie	22
3.4.1.	Rozgałęzienie historii projektu	23
3.4.2.	Scalanie gałęzi (merge)	24
3.4.3.	Konflikty scalania	25
3.5.	Praca ze zdalnym repozytorium [remote]	28

1. Wprowadzenie

System kontroli wersji (ang. version/revision control system) – oprogramowanie służące do śledzenia zmian głównie w kodzie źródłowym oraz pomocy programistom w łączeniu zmian dokonanych w plikach przez wiele osób w różnym czasie.

1.1. Linus Torvalds – twórca systemu GIT

GIT to system kontroli wersji, będący wolnym oprogramowaniem o otwartym kodzie. Projektowany był z myślą o jednoczesnej pracy wielu ludzi nad jednym kodem.

System GIT stworzył Linus Torvalds jako narzędzie wspomagające rozwój jądra Linux ponieważ wszystkie istniejące (darmowe) systemy kontroli wersji były niewystarczające. System ten miał być rozproszony, szybki, miał chronić przed błędami w repozytorium i nie posiadać wad CVS-a.

Prace nad Gitem rozpoczęły się 3 kwietnia 2005 roku, projekt został ogłoszony 6 kwietnia, 7 kwietnia Git obsługiwał kontrolę wersji swojego własnego kodu, 18 kwietnia pierwszy raz wykonano łączenie kilku gałęzi kodu, 27 kwietnia Git został przetestowany pod względem szybkości z wynikiem 6,7 lat na sekundę, a 16 czerwca Linux 2.6.12 był hostowany przez Gita.

Jak widać w pełni funkcjonalny system kontroli wersji powstał w niecały miesiąc. A ojcem tego sukcesu jest oczywiście sam Linus Torvalds, który większość projektu wykonał sam!

1.2. Najważniejsze cechy GITa

Do najważniejszych cech systemu GIT należą:

- Dobre wsparcie dla rozgałęzionego procesu tworzenia oprogramowania: jest dostępnych kilka algorytmów łączenia zmian z dwóch gałęzi, a także możliwość dodawania własnych algorytmów.
- Praca off-line: każdy programista posiada własną kopię repozytorium, do której może zapisywać zmiany bez połączenia z siecią; następnie zmiany mogą być wymieniane między lokalnymi repozytoriami.
- Wsparcie dla istniejących protokołów sieciowych: dane można wymieniać przez HTTP(S), FTP, rsync, SSH.
- Efektywna praca z dużymi projektami: system Git według zapewnień Torvaldsa, a także według testów fundacji Mozilla, jest o rzędy wielkości szybszy niż niektóre konkurencyjne rozwiązania.
- Wsparcie dla nieliniowego programowania (branche)
- Adresowanie przez zawartość (SHA-1)

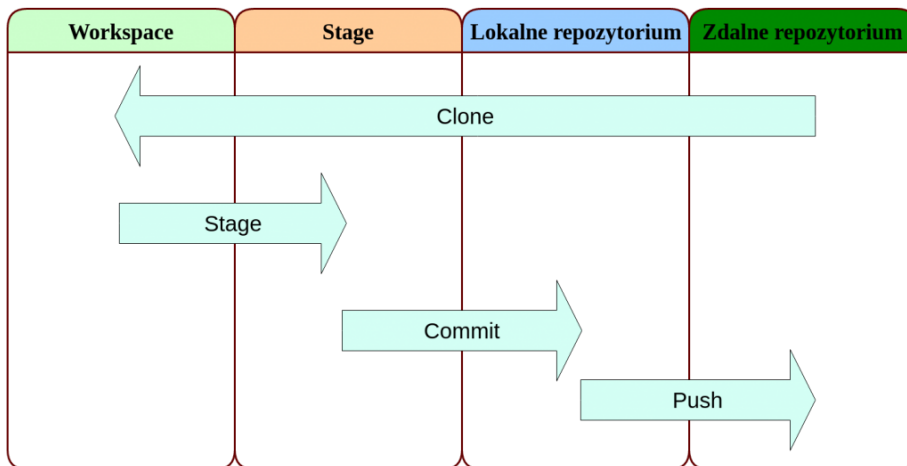
1.3. Stany plików w GIT

Aby móc pracować z GITEM trzeba zrozumieć, w jakich stanach mogą znajdować się zarządzane przez system pliki. Git wprowadza trzy główne stany dla zmian: zmodyfikowany, śledzony oraz zatwierdzony.

- zmodyfikowany – plik był edytowany, ale zmiana o tym nie została jeszcze nigdzie zapisana;
- śledzony – zmodyfikowany plik został oznaczony do zatwierdzenia przy najbliższej operacji commit;
- zatwierdzony – dokonana zmiana została zapisana i utrwalona w lokalnej bazie danych;

Przesłanie zmian do zdalnego repozytorium jest już operacją opcjonalną.

Stany GIT



- katalog Git – to trzon lokalnego repozytorium. W nim Git przechowuje metadane o plikach oraz obiektową bazę danych. Ten katalog jest kopiowany podczas klonowania repozytorium.
- katalog roboczy – jest to odtworzony obraz wersji projektu. To właśnie zawartość tego katalogu jest modyfikowana przez użytkownika.
- przechowalnia (stage) – to miejsce pośrednie, między katalogiem roboczym, a lokalną bazą danych. Dzięki niej można utrwalić tylko wybrane zmiany.

1.4. Terminologia

Podczas pracy z GITem możemy spotkać się z następującym nazewnictwem:

- **Branch** - równoległa gałąź projektu rozwijana oddzielnie od głównej.
- **Tag** – marker konkretnej wersji (rewizja w SVN'ie) projektu.
- **Working Dir** – katalog roboczy na którym pracujemy
- **Index** – rodzaj „cache”, czyli miejsca gdzie trzymane są zmiany do commita
- **Master Branch** – główny branch z którym łączymy (merge) nasze zmiany przed wysłaniem do zdalnego repozytorium.
- **Development Branch** – gałąź na której łączone są gałęzie feature. Przy wyjściu kolejnej wersji mergowany jest z Master Branchem.
- **Feature Branch** – gałąź na której rozwijane jest konkretne narzędzie bądź dodatek do głównego projektu.
- **HotFix** – branch tworzony na potrzeby szybkich poprawek, naprawienia niezgodności lub bugu.

1.5. Obiekty GITa

- **Commit** – wskazuje na tree oraz ojca, zawiera przykładowo takie informacje jak autor, data i treść wiadomości.
- **Tree** – reprezentuje stan pojedynczego katalogu (lista obiektów blob oraz zagnieżdżonych obiektów tree)
- **Blob** – zawiera zawartość pliku bez żadnej dodatkowej struktury
- **Tag** – wskazuje na konkretny commit oraz zawiera opis taga.

1.6. Git Bash

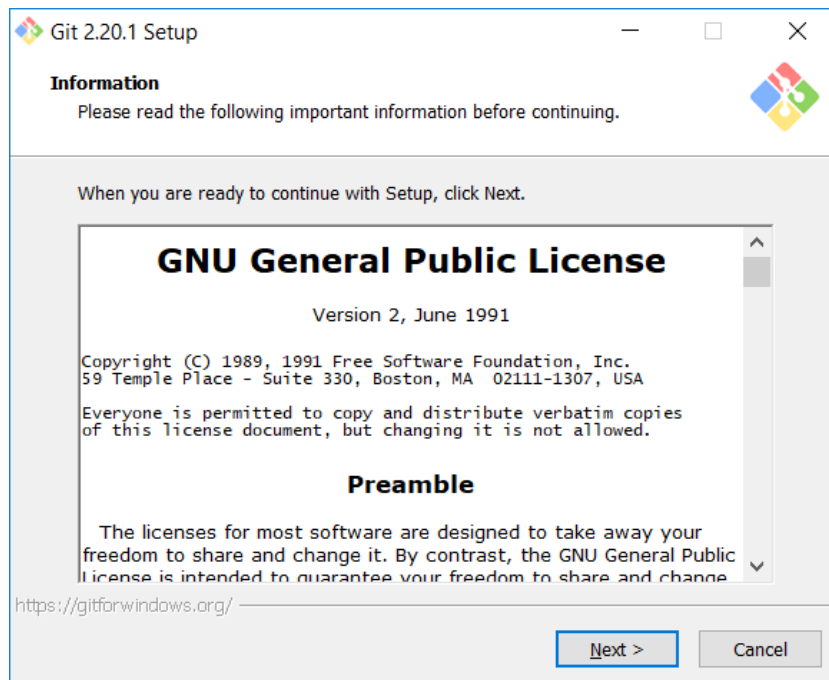
Git Bash to konsola systemu GIT, która umożliwia w pełni funkcjonalne zarządzanie repozytorium. Za pomocą odpowiednich komend konsolowych można m.in. tworzyć repozytorium, dodawać pliki, śledzić zmiany itd.

Do najważniejszych poleceń Git Basha należą:

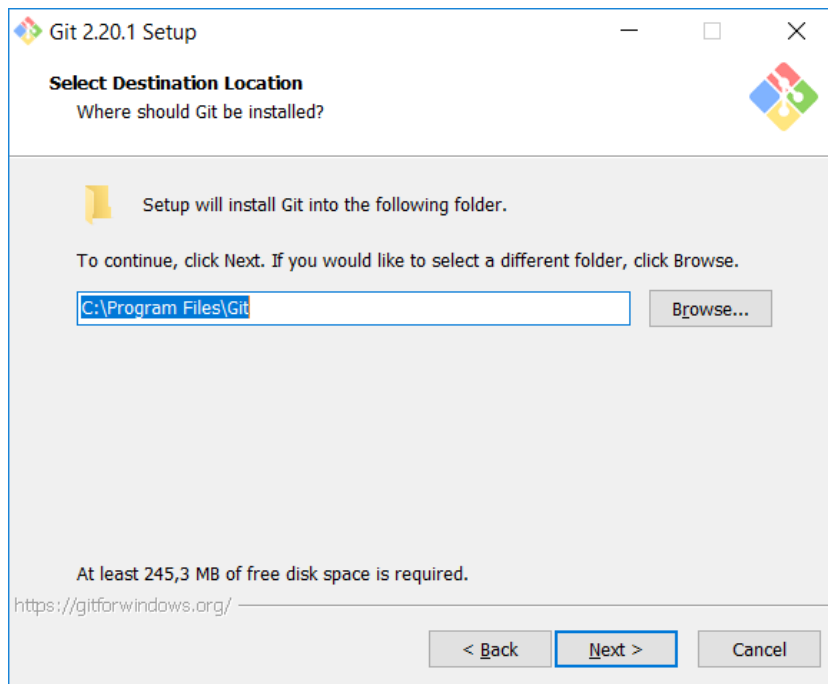
- **git init [nazwa]** – tworzy nowe repozytorium lokalne.
- **git clone origin [link]** – klonuje repozytorium z serwera zdalnego na komputer.
- **git remote add origin [link]** – dodaje repozytorium zdalne do repozytorium lokalnego.
- **git add [plik]** – dodaje wszystkie zmienione pliki do staged area
- **git checkout -- .** – usuwa wszystkie pliki z staged area.
- **git branch [nazwa]** – tworzy nowy branch
- **git checkout -b [nazwa]** – tworzy nowy branch i ustawia go jako aktualny branch.
- **git checkout [nazwa]** – przełącza na wybrany branch
- **git commit -m 'tytuł' -m 'opis'** – tworzy commit z plików w staged area o wybranym tytule oraz opisie.
- **git push** – wypycha zmiany lokalne na serwer zdalny
- **git pull** – ściąga oraz przyłącza zmiany z serwera zdalnego na serwer lokalny
- **git fetch** – pobiera zmiany z repozytorium zdalnego, ale nie przyłącza ich do working directory.
- **git merge** – łączy zmiany z dwóch różnych branchy, ścieżek, lub zmiany pobrane z repozytorium zdalnego za pomocą git fetch
- **git stash save [plik]** – dodaje zmieniony plik do schowka
- **git stash apply** – dodaje zmiany ze schowka do working directory.

2. Instalacja GITa

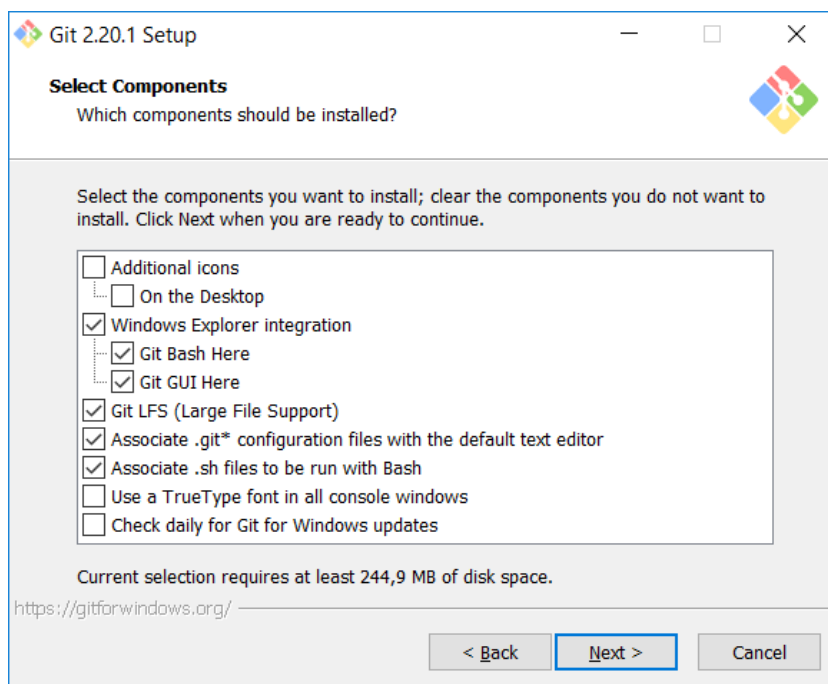
Klikamy **Next**:



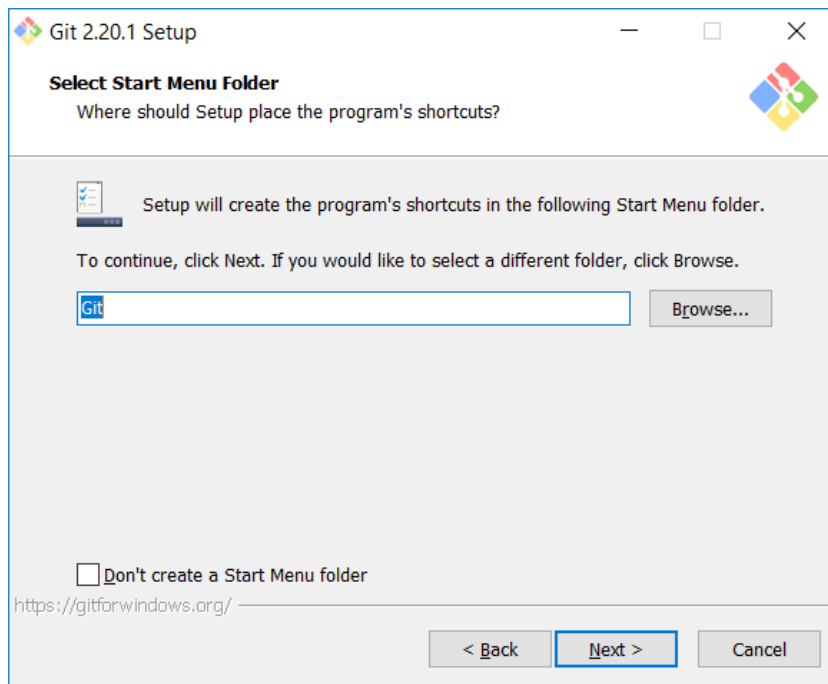
Wybieramy folder instalacji i klikamy **Next**:



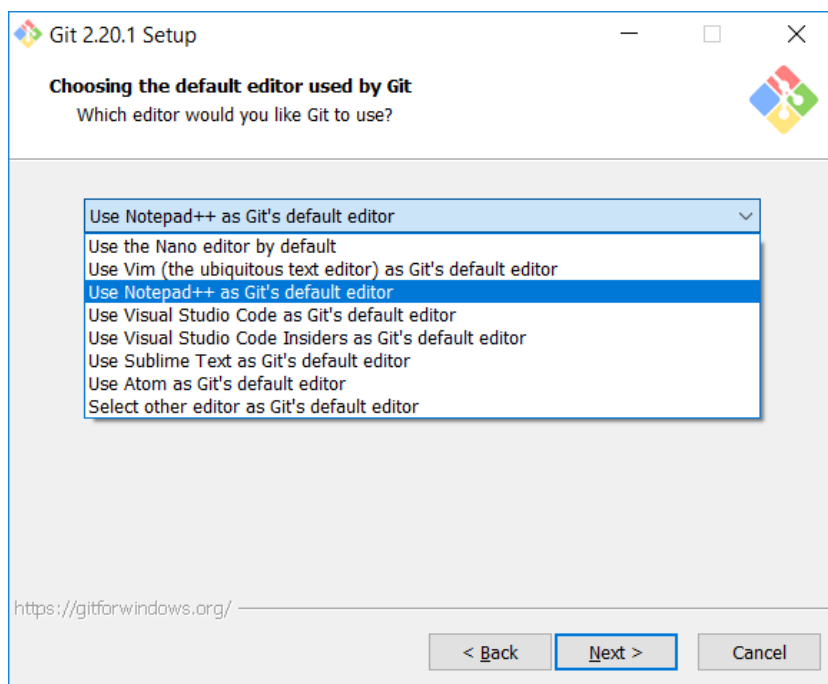
Wybieramy komponenty do zainstalowania (pozostawiamy domyślne) i klikamy **Next**:



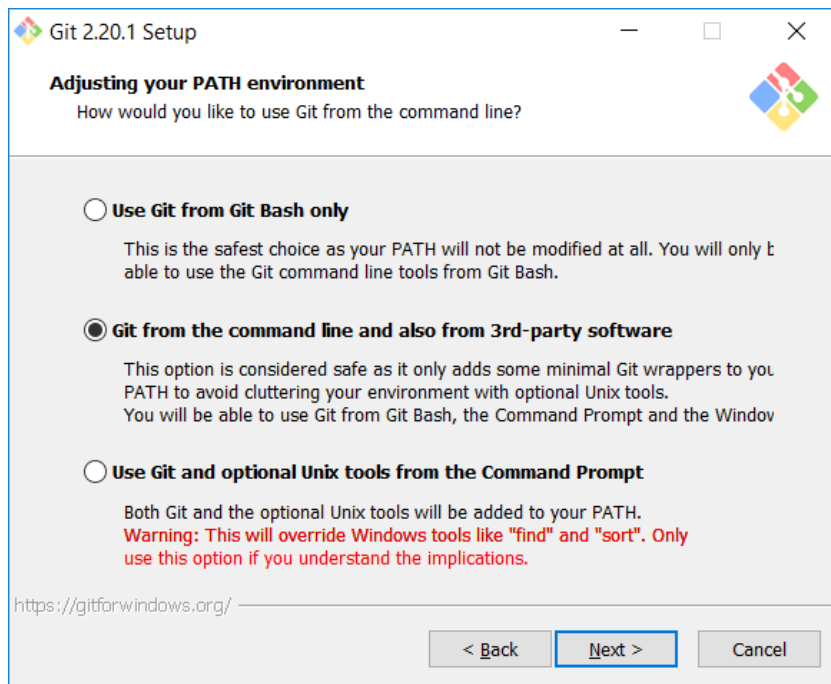
Tworzenie folderu dla **Gita** w menu **Start**:



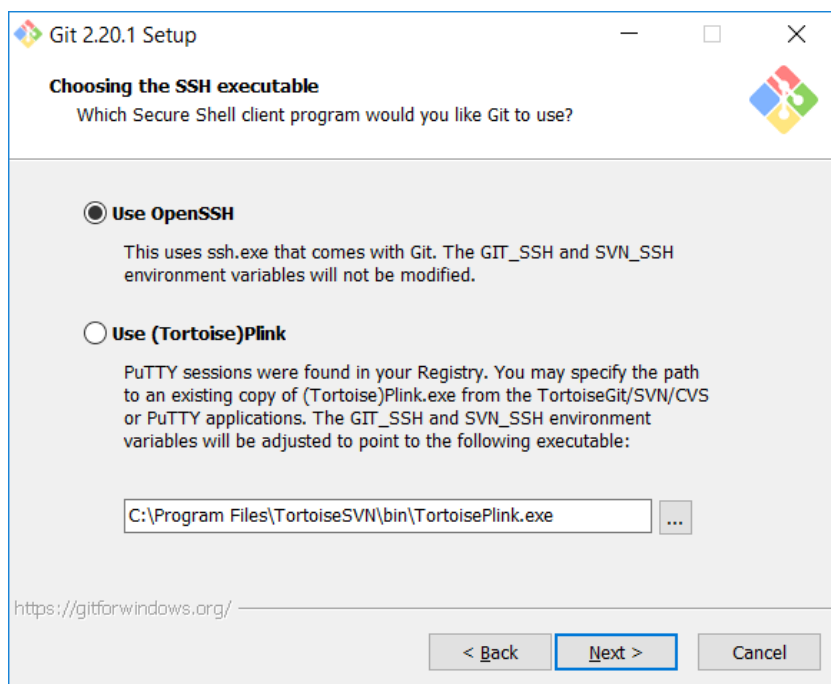
Wybieramy edytor dla Gita np. **Notepad++**:

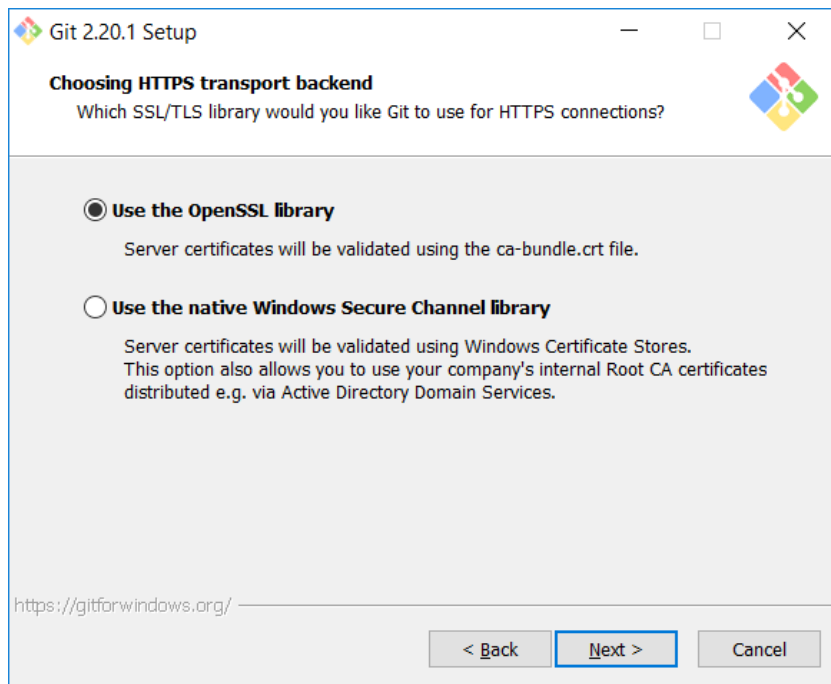


Wybieramy ustawienia zmiennej środowiskowej **PATH**:

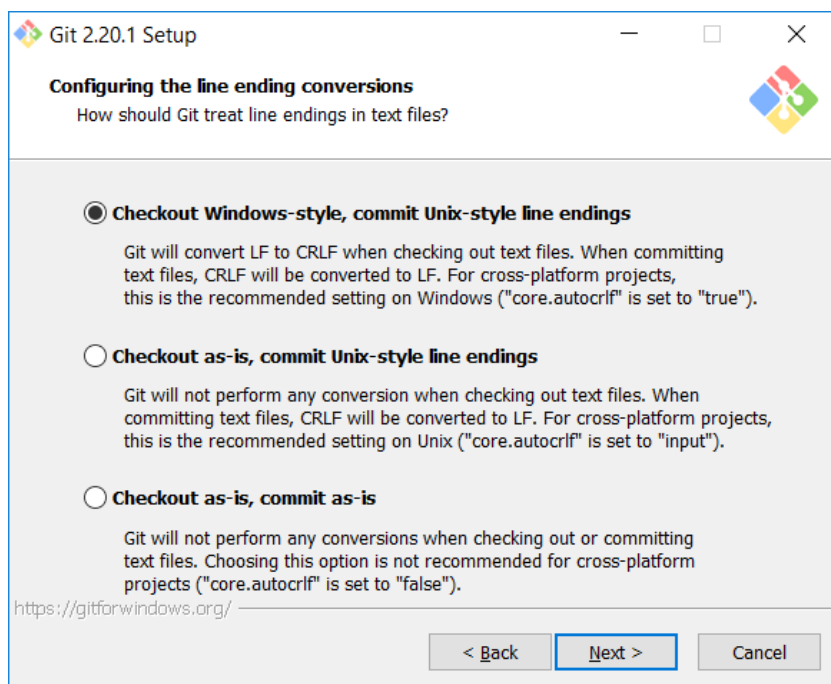


Wybieramy ustawienia sieciowe dla Gita:

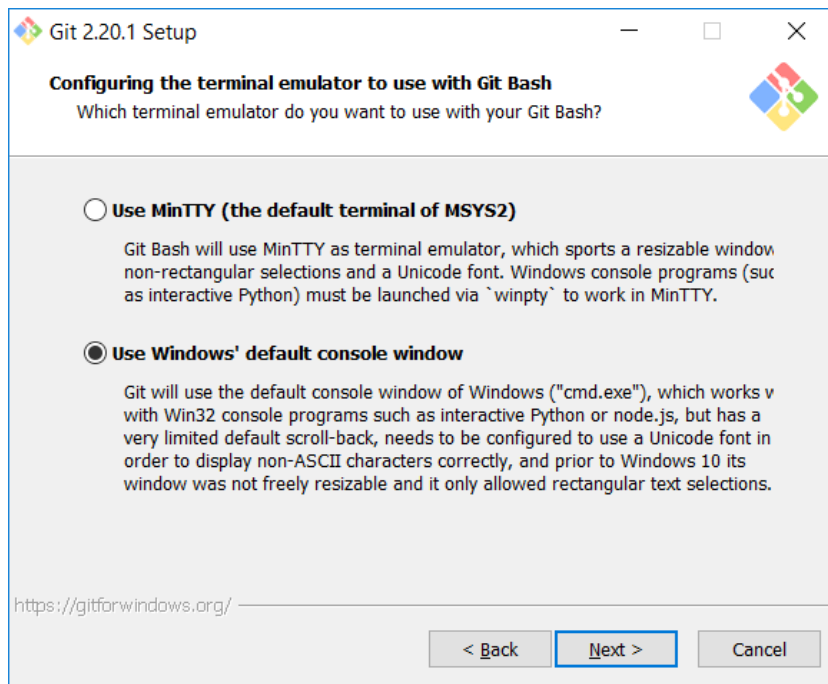




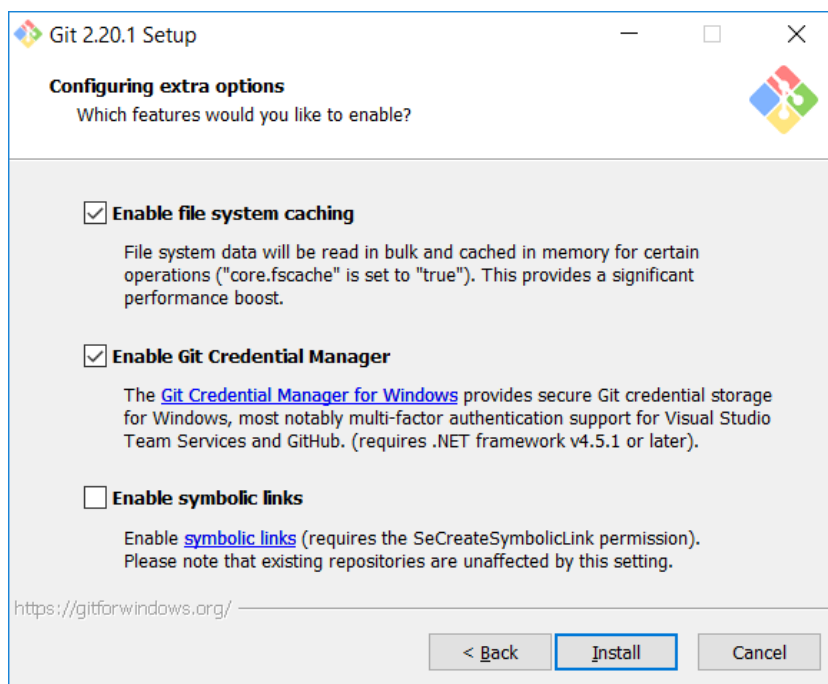
Wybieramy ustawienia znaku końca wiersza dla plików tekstowych:



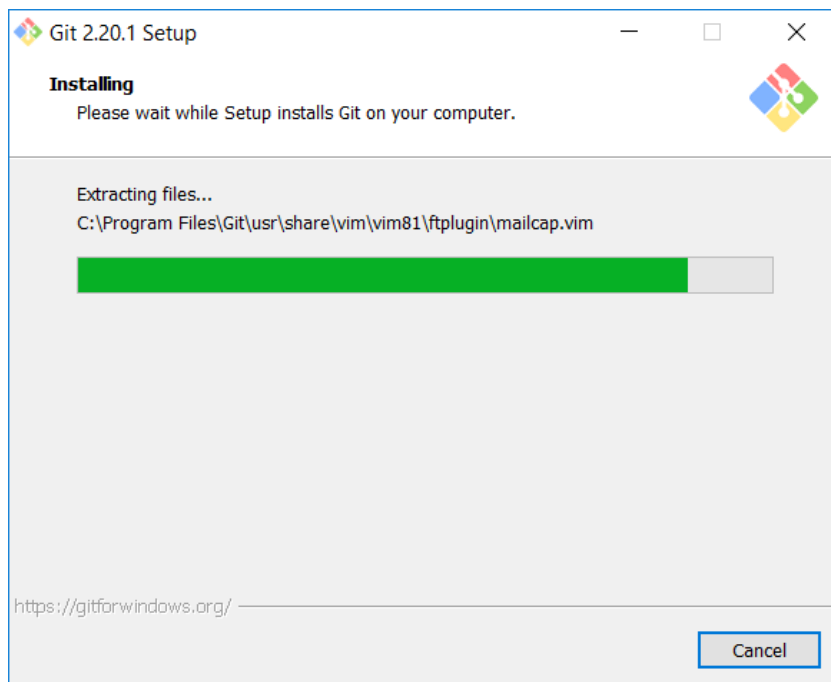
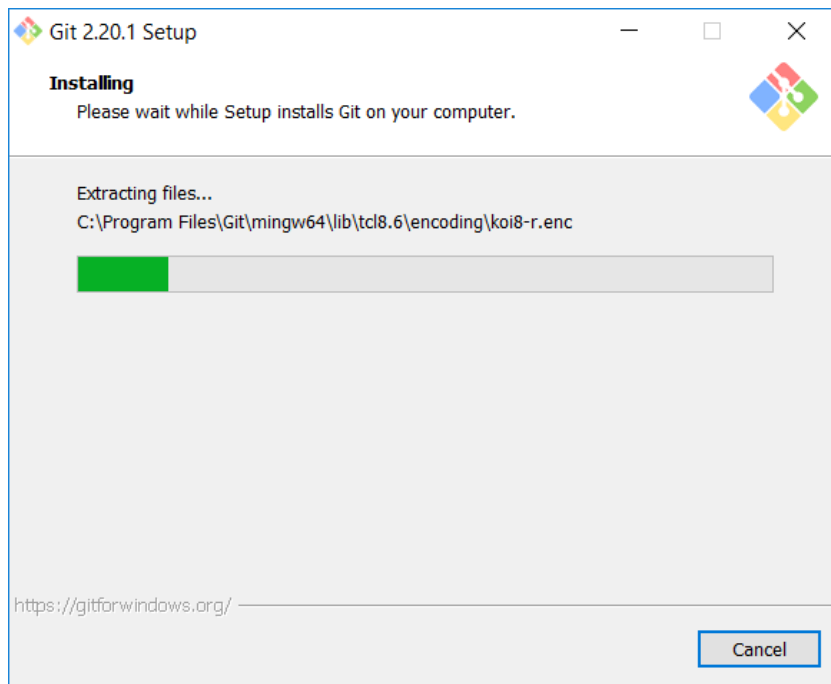
Wybieramy terminal:



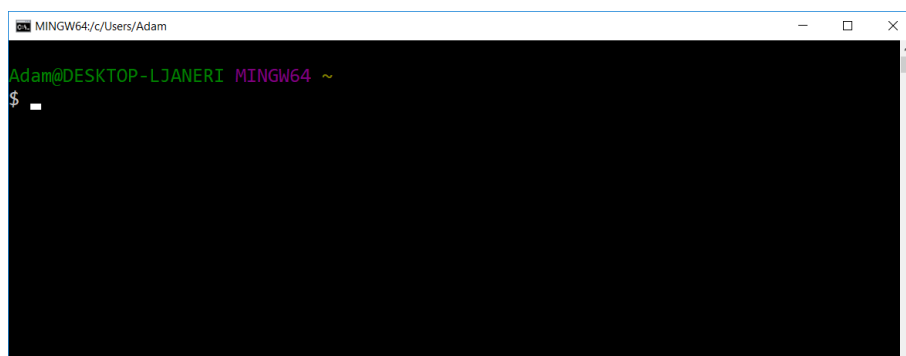
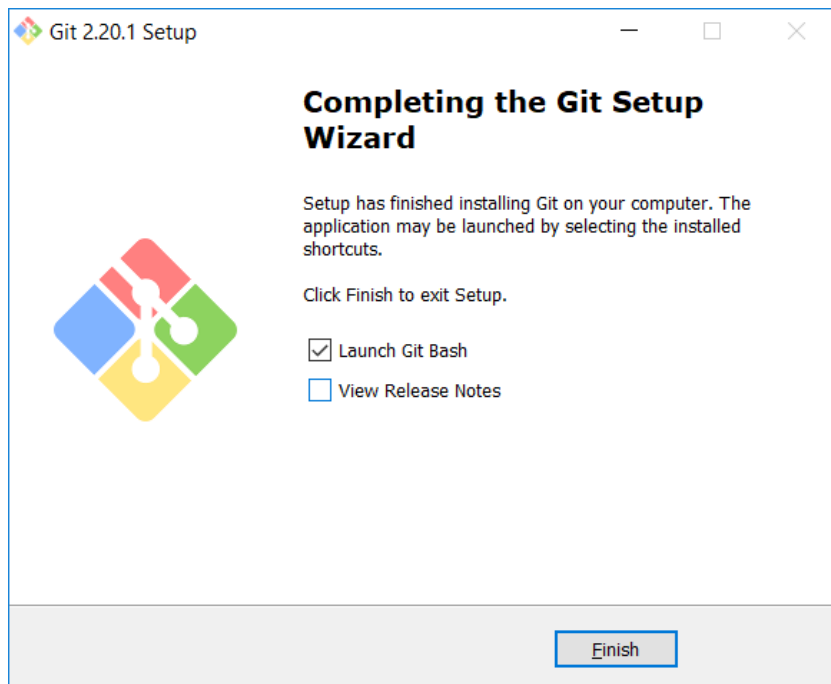
Opcje dodatkowe (domyślne):



Instalacja:



Kończymy instalację i uruchamiamy **Git Bash**:



3. Konsola Git Bash

3.1. Konfiguracja GITa

Konfiguracja GITa sprowadza się do ustawienia nazwy użytkownika, adresu e-mail oraz inicjalizacji repozytorium.

Do tego celu służą kolejno polecenia:

```
git config --global user.name "Twoje imię i nazwsko"
```

```
git config --global user.email twój@email.com
```

```
git init
```

Polecenia te należy wywołać będąc w katalogu, w którym chcemy utworzyć repozytorium.

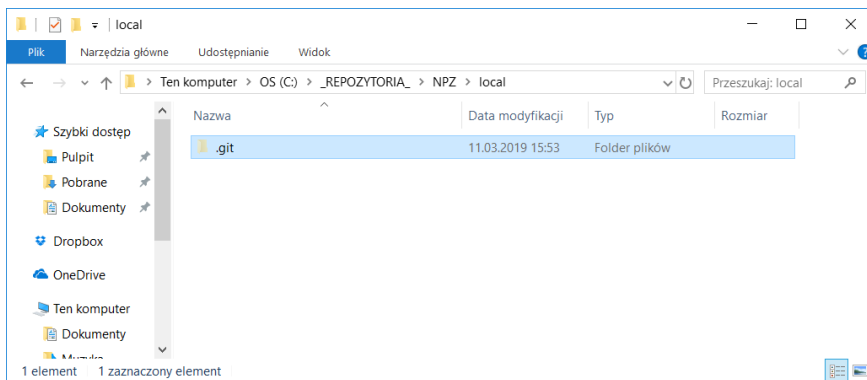
```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local
$ git config --global user.name Adam

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local
$ git config --global user.email [redacted]@gmail.com

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local
$ git init
Initialized empty Git repository in C:/_REPOZYTORIA_/NPZ/local/.git/

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

W wyniku w wybranym folderze utworzony został ukryty katalog **.git** zawierający pliki konfiguracyjne GITA.



ZADANIE

Utwórz na pulpicie katalog o nazwie **moje_repozytorium** i zainicjalizuj go jako repozytorium GITA.

3.2. Rejestrowanie zmian w repozytorium

Rejestrowanie zmian w repozytorium podzielone jest na kilka etapów bezpośrednio związanych z cyklem życia zmian.

3.2.1. Sprawdzenie stanu plików w repozytorium

Do sprawdzenia stanu plików w repozytorium (statusu) służy polecenie:

git status

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git status
On branch master

No commits yet

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

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

Jak można zauważyć, w naszym repozytorium nie ma żadnych plików, które są śledzone.

ZADANIE

Utwórz w katalogu repozytorium nowy plik tekstowy **text.txt** i zapisz w nim tekst *Ala ma kota*. Następnie sprawdź status plików w repozytorium.

W wyniku wykonanych czynności otrzymujemy komunikat:

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git status
On branch master

No commits yet

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

    text.txt

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

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

GIT wykrył, że dodaliśmy nowy plik i ostrzega nas, że nie jest on jeszcze śledzony przez repozytorium.

3.2.2. Śledzenie nowych plików

Aby rozpocząć śledzenie naszego pliku posłużymy się poleceniem

`git add`

Składnia polecenia:

- `git add text.txt` – dodaje jeden wybrany plik
- `git add -A` – dodaje wszystkie nieśledzone pliki
- `git add .` – dodaje do śledzenia bieżący katalog ze wszystkimi plikami i katalogami, które się w nim znajdują

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git add text.txt

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git status
On branch master

No commits yet

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

        new file:   text.txt

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

Jak można zauważyć, status pliku **text.txt** zmienił się.

ZADANIE

Dodaj do śledzenia plik **text.txt** i następnie sprawdź status repozytorium.

3.2.3. Dodawanie zmian do poczekalni

Jak zostało wcześniej napisane, GIT operuje na pojedynczych zmianach, a nie na plikach. Bardzo dobrze to widać podczas dodawania plików do poczekalni.

ZADANIE

Zmodyfikuj plik **text.txt** dodając do niego kolejną linię tekstu: *Kot ma Alę*. Następnie sprawdź status plików w repozytorium.

Ponowne wywołanie `git status` pokazuje, że plik **text.txt** jest jednocześnie w dwóch sekcjach:

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git status
On branch master

No commits yet

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

        new file:   text.txt

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:   text.txt

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

Dzieje się tak, ponieważ GIT umieszcza plik w poczekalni w dokładnie takiej wersji, w jakiej znajdował się podczas odpalenia komendy `git add`. Jeżeli w tym momencie zostanie uruchomiona komenda `git commit` to zatwierdzona zostanie wersja z poczekalni, a nie ta widoczna w katalogu roboczym.

Żeby zaktualizować plik w poczekalni, trzeba go zwyczajnie jeszcze raz dodać przez `git add`.

```
MINGW64:/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git add text.txt

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git status
On branch master

No commits yet

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

        new file:   text.txt

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

Wykorzystanie poczekalni w procesie zatwierdzania zmian daje ogromne możliwości, dzięki temu można wybrać, które konkretnie modyfikacje chce się zatwierdzić i utrwalić w repozytorium.

ZADANIE

Zaktualizuj plik **text.txt** poprzez wywołanie komendy `git add`. Sprawdź status plików.

3.2.4. Podgląd dokonanych zmian

Jest to bardzo ważna funkcjonalność. Przed zatwierdzeniem zmian zawsze warto zweryfikować, czy wszystko poszło zgodnie z planem. Może się zdarzyć, że zmiany zostały zrobione nie w tym miejscu, co trzeba lub pojawiły się jakieś dodatkowe wygenerowane pliki, których nie chcemy utrzymywać w repozytorium. Dzięki weryfikacji zmian w podglądzie można uniknąć tego typu błędów.

Polecenie `git status` dostarczy tylko informacji, które pliki zostały zmodyfikowane, natomiast dzięki `git diff` można dokładnie zobaczyć te zmiany.

- różnica między katalogiem roboczym a poczekalnią – `git diff`
- różnica między poczekalnią a repozytorium – `git diff --cached`

```
MINGW64:/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git status
On branch master

No commits yet

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

        new file:   text.txt

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git diff

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git diff --cached
diff --git a/text.txt b/text.txt
new file mode 100644
index 0000000..a3d9b0f
--- /dev/null
+++ b/text.txt
@@ -0,0 +1,2 @@
+Ala ma kota.
+Kot ma AlkEAS.
\ No newline at end of file

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
```

ZADANIE

Sprawdzić działanie poleceń z punktu 3.2.4

3.2.5. Zatwierdzanie zmian

Jeżeli mamy już pewność, że dokonane zmiany są poprawne, można utrwalić je w lokalnym repozytorium. W tym celu posłużymy się komendą

```
git commit
```

Wywołanie komendy bez żadnych argumentów uruchomi najpierw domyślny edytor tekstowy w celu podania komentarza dla zatwierdzanych zmian.

Opcjonalnie można podać komentarz jako argument już przy samym wywołaniu komendy:

```
git commit -m „komentarz”
```

Po potwierdzeniu zmiany zostaną zapisane w repozytorium w postaci nowej migawki.

```
MINGW64:/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git commit --m "komentarz do commita"
[master (root-commit) 925edbd] komentarz do commita
1 file changed, 2 insertions(+)
create mode 100644 text.txt

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```


3.2.6. Historia zmian

Do przeglądania historii zmian służy polecenie:

`git log`

Domyślnie log bez podania żadnych argumentów wyświetla zmiany od najnowszego do najstarszego.

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git log
commit 5c5be10ce1db72c4f50aa3c51f73385da0223dc2 (HEAD -> master)
Author: Adam <[redacted]@gmail.com>
Date:   Mon Mar 11 21:07:22 2019 +0100

    dodanie tekstu w text2.txt

commit aa97abaaa01e86b50c1e95f5c3a4cc8936d68fba
Author: Adam <[redacted]@gmail.com>
Date:   Mon Mar 11 21:06:28 2019 +0100

    komentarz 2

commit 925edbd2f93f014e4011bda0554e66e803e716fc
Author: Adam <[redacted]@gmail.com>
Date:   Mon Mar 11 21:02:40 2019 +0100

    komentarz do commita

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

Polecenie log jest bardzo rozbudowane i zawiera wiele opcji konfiguracyjnych, ich pełną listę można znaleźć korzystając z pomocy:

`git help log`

Jedną z najprzydatniejszych opcji jest `-p`. Pokazuje ona różnice wprowadzone z każdą rewizją. Dodatkowo można użyć opcji `-2` aby ograniczyć zbiór do dwóch ostatnich wpisów:

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git log -p -2
commit 5c5be10ce1db72c4f50aa3c51f73385da0223dc2 (HEAD -> master)
Author: Adam <[redacted]@gmail.com>
Date:   Mon Mar 11 21:07:22 2019 +0100

    dodanie tekstu w text2.txt

diff --git a/text2.txt b/text2.txt
index e69de29..c9c1808 100644
--- a/text2.txt
+++ b/text2.txt
@@ -0,0 +1 @@
+Narzędzia Pracy Zespołowej
\ No newline at end of file

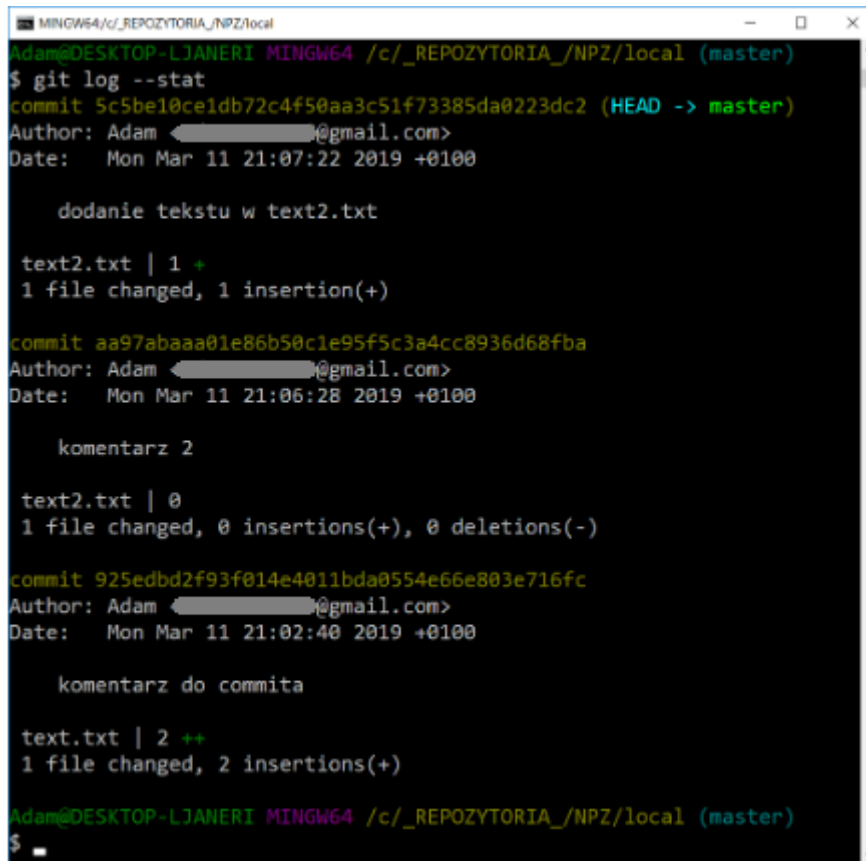
commit aa97abaaa01e86b50c1e95f5c3a4cc8936d68fba
Author: Adam <[redacted]@gmail.com>
Date:   Mon Mar 11 21:06:28 2019 +0100

    komentarz 2

diff --git a/text2.txt b/text2.txt
new file mode 100644
index 0000000..e69de29

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
```

Opcja spowodowała wyświetlanie tych samych informacji z tą różnicą, że bezpośrednio po każdym wpisie został pokazywany tzw. *diff*, czyli różnica. Jest to szczególnie przydatne podczas recenzowania kodu albo szybkiego przeglądania zmian dokonanych przez współpracowników. Dodatkowo można skorzystać z całej serii opcji podsumowujących wynik działania `git log`. Na przykład, aby zobaczyć skrócone statystyki każdej z zatwierdzonych zmian należy użyć opcji `--stat`:

A screenshot of a terminal window titled 'MINGW64/c/_REPOZYTORIA_/NPZ/local'. The prompt is 'Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)'. The command '\$ git log --stat' has been executed. The output shows three commits. The first commit (5c5be10ce1db72c4f50aa3c51f73385da0223dc2) is the HEAD and master branch, with the message 'dodanie tekstu w text2.txt'. It shows a diff for 'text2.txt' with 1 insertion. The second commit (aa97abaaa01e86b50c1e95f5c3a4cc8936d68fba) has the message 'komentarz 2' and shows no changes to 'text2.txt'. The third commit (925edbd2f93f014e4011bda0554e66e803e716fc) has the message 'komentarz do commita' and shows 2 insertions in 'text.txt'. The terminal ends with the prompt '\$ '.

Przydatnymi opcjami są również `--since` oraz `--until`. Wprowadzają one ograniczenia czasu dla wyświetlanych logów, np. wyświetlenie informacji o zmianach dokonanych w ciągu ostatnich kilkunastu minut:

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git log --since =17.minutes
commit 5c5be10ce1db72c4f50aa3c51f73385da0223dc2 (HEAD -> master)
Author: Adam <[REDACTED]@gmail.com>
Date: Mon Mar 11 21:07:22 2019 +0100

    dodanie tekstu w text2.txt

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git log --since =18.minutes
commit 5c5be10ce1db72c4f50aa3c51f73385da0223dc2 (HEAD -> master)
Author: Adam <[REDACTED]@gmail.com>
Date: Mon Mar 11 21:07:22 2019 +0100

    dodanie tekstu w text2.txt

commit aa97abaaa01e86b50c1e95f5c3a4cc8936d68fba
Author: Adam <[REDACTED]@gmail.com>
Date: Mon Mar 11 21:06:28 2019 +0100

    komentarz 2

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

Polecenie to obsługuje mnóstwo formatów - można uściślić konkretną datę (np. "2008-01-15") lub podać datę względną jak np. 2 lata 1 dzień 3 minuty temu.

Można także odfiltrować listę pozostawiając jedynie rewizje spełniające odpowiednie kryteria wyszukiwania. Opcja --author pozwala wybierać po konkretnym autorze, a opcja --grep na wyszukiwanie po słowach kluczowych zawartych w notkach zmian.

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git log --author=Adam
commit 5c5be10ce1db72c4f50aa3c51f73385da0223dc2 (HEAD -> master)
Author: Adam <[REDACTED]@gmail.com>
Date: Mon Mar 11 21:07:22 2019 +0100

    dodanie tekstu w text2.txt

commit aa97abaaa01e86b50c1e95f5c3a4cc8936d68fba
Author: Adam <[REDACTED]@gmail.com>
Date: Mon Mar 11 21:06:28 2019 +0100

    komentarz 2

commit 925edbd2f93f014e4011bda0554e66e803e716fc
Author: Adam <[REDACTED]@gmail.com>
Date: Mon Mar 11 21:02:40 2019 +0100

    komentarz do commita

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git log --author=Ala

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

```
WybierzMINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git log --grep="komentarz"
commit aa97abaaa01e86b50c1e95f5c3a4cc8936d68fba
Author: Adam <[redacted]@gmail.com>
Date: Mon Mar 11 21:06:28 2019 +0100

    komentarz 2

commit 925edbd2f93f014e4011bda0554e66e803e716fc
Author: Adam <[redacted]@gmail.com>
Date: Mon Mar 11 21:02:40 2019 +0100

    komentarz do commita

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

Jeżeli potrzebujemy określić zarówno autora jak i słowa kluczowe, musimy dodać opcję `--all-match` - w przeciwnym razie polecenie dopasuje jedynie wg jednego z kryteriów.

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git log --author=Adam --grep="dodanie" --all-match
commit 5c5be10ce1db72c4f50aa3c51f73385da0223dc2 (HEAD -> master)
Author: Adam <[redacted]@gmail.com>
Date: Mon Mar 11 21:07:22 2019 +0100

    dodanie tekstu w text2.txt

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

ZADANIE

Utworzyć kolejny plik **text2.txt** i dodać go do repozytorium oraz zacomitować zmiany (komentarz dowolny). Następnie zmodyfikować plik **text2.txt** umieszczając w nim tekst: *Stoi na stacji lokomotywa. Dodać zmiany (add). Dodać kolejną linie tekstu w pliku **tekst2.txt**: Ciężka, ogromna i pot z niej spływa – tłusta oliwa.* Dodać zmiany i wykonać commit. Sprawdzić działanie polecenia `git log` z różnymi opcjami, o których mowa w tym podrozdziale.

3.3. Rozszerzone mechanizmy

3.3.1. Cofanie zmian

W celu wycofania zmian, które zostały już wysłane do zdalnego repozytorium, można skorzystać z commitów wycofujących. Ten mechanizm nie modyfikuje historii, a generuje commit, który jest przeciwieństwem zmiany, którą chcemy wycofać. Służy do tego polecenie:

`git revert [opcje]`

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git revert HEAD~0
[master 125c02f] Revert "ciuch"
 1 file changed, 1 insertion(+), 2 deletions(-)

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
```

Powyższe wywołanie cofa ostatni commit w rewizji HEAD.

ZADANIE

Dodać do pliku **text2.txt** kolejną linię tekstu: *ciuch, ciuch*. Dodać zmiany i wykonać commit. Następnie dodać jeszcze jedną linię: *lokomotywa odjeżdża* i również dodać zmiany i wykonać commit. Dalej wycofać ostatnie 2 commity i sprawdzić zawartość pliku **text2.txt**.

Usunąć plik **text.txt** i zatwierdzić zmiany wykonując polecenia `add` i `commit`. Następnie wycofać ostatni commit i sprawdzić zawartość repozytorium. Sprawdzić historię zmian w repozytorium.

3.3.2. Tagowanie źródeł

Tagowanie (etykietowanie) źródeł to mechanizm pozwalający na oznaczenie ważniejszych miejsc w historii zmian projektu. Najczęściej jest wykorzystywany do oznaczania wersji aplikacji (np. wersja 3.1.5, itp). Git posiada dwa rodzaje etykiet: lekkie oraz opisane. Dla nas ważne będą lekkie.

W celu oznaczenia aktualnych źródeł nowym tagiem wpisujemy komendę:

```
git tag [nazwa-tagu]
```

Natomiast sama komenda `git tag`, bez podania żadnych argumentów, wyświetli listę wszystkich znanych tagów.

ZADANIE

Utworzyć etykietę dla bieżącej wersji repozytorium. Następnie dokonać kilku zmian w repozytorium wraz z commitami i otagować każdy commit. Wyświetlić listę tagów.

3.3.3. Ignorowanie plików

W większości projektów mamy do czynienia z plikami, których nie chcemy wersjonować. Są to np. pliki generowane automatycznie. Dodanie ich do repozytorium powoduje tylko zaciemnienie obrazu wprowadzanych zmian.

Można, co prawda, pomijać tego typu pliki przy zatwierdzaniu zmian, jednak nie jest to zbyt pragmatyczne podejście. Dużo lepszym wyjściem jest oznaczenie takiej klasy plików jako ignorowane. Od tego momentu nie będą nawet widoczne jako pliki zmodyfikowane.

Mechanizm ignorowania plików oparty jest o plik tekstowy **.gitignore**. Poniżej przykładowa zawartość:

```
*.tmp #komentarz
```

```
Tmp #komentarz2
```

Kolejne klasy ignorowanych plików wpisujemy w osobnych liniach. Pierwsza linijka odpowiedzialna jest za ignorowanie wszystkich plików o rozszerzeniu *.tmp*, natomiast druga za cały katalog *tmp* oraz jego zawartość.

Warto już na starcie zdefiniować, które pliki mają być ignorowane. Pozwoli to w przyszłości na uniknięcie zabawy z niepotrzebnymi plikami.

Ponieważ plik **.gitignore** jest zwykłym plikiem tekstowym przechowywanym w głównym katalogu repozytorium, on również może podlegać wersjonowaniu. Po jego dodaniu lub modyfikacji warto zacommitować naniesione zmiany lub jego też oznaczyć do ignorowania.

ZADANIE

Utworzyć plik **.gitignore** i umieścić w nim kilka rozszerzeń plików (np. **.tmp*, **.inf*) oraz katalog np. *temp*. Wykonać commit. Następnie dodać do repozytorium po 1 pliku z każdym z ignorowanych

rozszerzeń oraz katalog podany do ignorowania. W katalogu tym powinno znajdować się kilka plików. Sprawdzić status repozytorium, tzn. czy są jakieś zmiany widziane przez GITa.

3.4. Gałęzie w Gicie

Rozgałęzianie projektu (np. kodu programu) to jedna z ważniejszych funkcjonalności GITa. Praca z gałęziami (*branch*) jest bardzo szybka, w odróżnieniu od innych podobnych rozwiązań. Gałąź w Gicie została zaimplementowana jako lekki, przesuwalny wskaźnik na miejsce w historii.

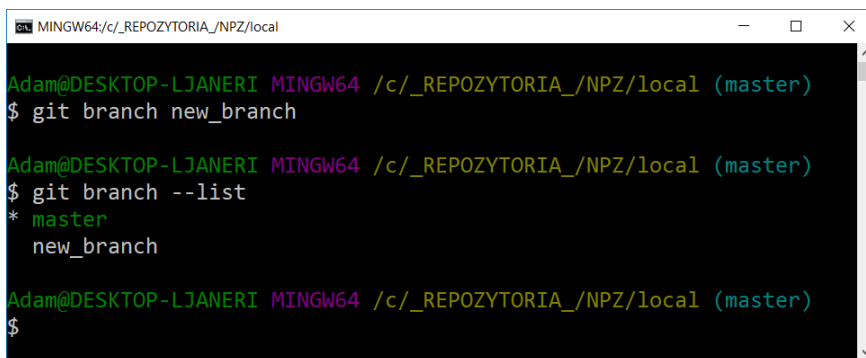
Domyślna nazwa gałęzi to *master*. Do tej pory pracowaliśmy tylko na niej. Pracując na kilku branchach system musi wiedzieć, na którym aktualnie się znajdujemy, w tym celu wprowadzono wskaźnik *HEAD*.

Do tworzenia nowych gałęzi służy polecenie:

```
git branch [nazwa-brancha]
```

Aby zobaczyć listę gałęzi należy użyć polecenia:

```
git branch --list
```



```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git branch new_branch

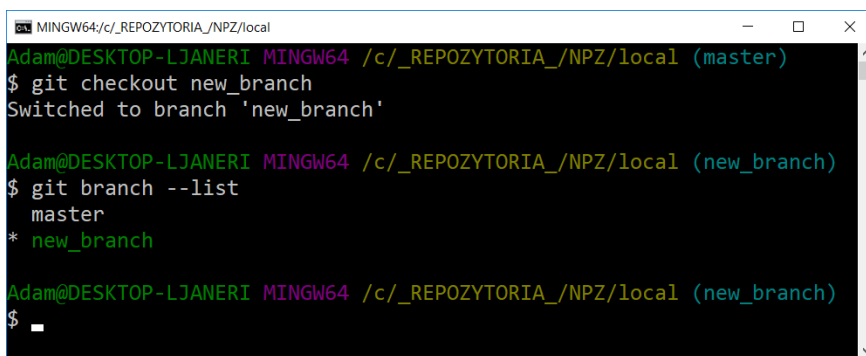
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git branch --list
* master
  new_branch

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

Branch *master* został oznaczony gwiazdką oraz kolorem zielonym. Jest to wskaźnik *HEAD* oznaczający, że aktualnie znajdujemy się na tej gałęzi.

Polecenie `git branch` nie zmienia aktualnego brancha, tylko tworzy nowy. W celu zmiany aktualnego brancha należy wywołać komendę:

```
git checkout [nazwa-brancha]
```



```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git checkout new_branch
Switched to branch 'new_branch'

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (new_branch)
$ git branch --list
  master
* new_branch

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (new_branch)
$
```

Jak można zauważyć, nazwa aktywnego brancha wyświetlana jest również w nawiasach na końcu ścieżki przed znakiem zachęty.

ZADANIE

Utworzyć nowy branch o nazwie *new_branch*. Przejść do nowo utworzonej gałęzi. Następnie utworzyć plik tekstowy *n_branch.txt* i zapisać w nim tekst: *Nowy branch*. Zastosować wprowadzone zmiany w repozytorium (dodać plik i zacommitować).

3.4.1. Rozgałęzienie historii projektu

Czasami zachodzi potrzeba rozwidlenia historii projektu, żeby móc np. rozwijać na osobnej gałęzi nową funkcjonalność, jednocześnie nie przeszkadzając innym w pracy. Wprowadzane w ten sposób zmiany są od siebie niezależne, łączy je tylko wspólny punkt w historii.

Żeby to lepiej zademonstrować, utworzony zostanie nowy branch *feature*, do którego wprowadzona zostanie nowa funkcjonalność, natomiast do brancha *master* szybka poprawka **hotfix**.

Ogólnie przyjętym zwyczajem jest rozwijanie nowych, niestabilnych funkcjonalności na osobnych branchach, żeby zachować jak największą stabilność głównej gałęzi *master*.

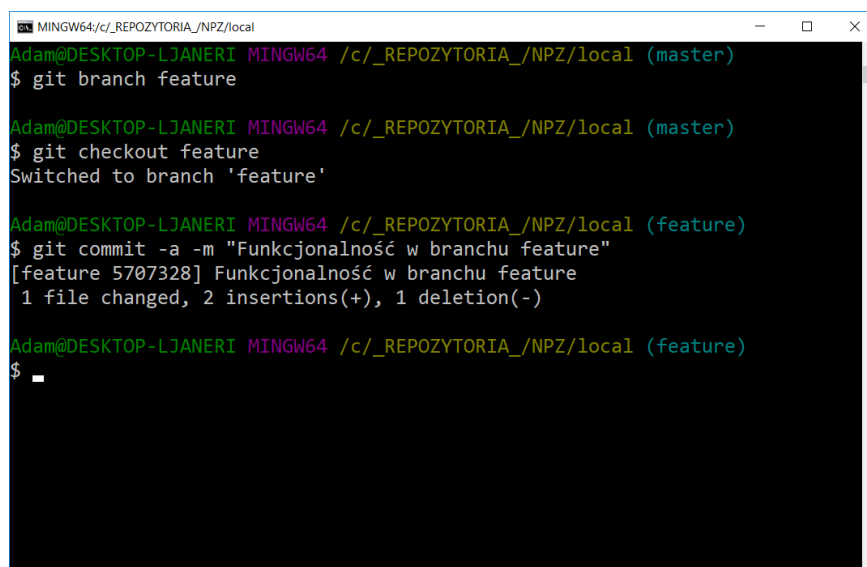
3.4.1.1. Nowa funkcjonalność

W naszym przykładzie nowa funkcjonalność będzie polegała na dodaniu nowej linii tekstu do pliku *feature.txt*. Ponieważ plik ten nie istnieje, dlatego musimy go utworzyć w branchu *master* i utworzyć dla niego historię zmian poprzez modyfikację jego zawartości.

ZADANIE

W branchu *master* utworzyć nowy plik o nazwie *feature.txt* i umieścić w nim tekst: *Linia 1*. Dodać plik do repozytorium i zacommitować zmiany. Następnie w nowych liniach tego pliku dodać tekst: *Linia 2* i *Linia 3*. Ponownie zatwierdzić wszystkie zmiany.

Utworzyć nową gałąź o nazwie *feature* i przejść do niej. Do pliku *feature.txt* dodać linię tekstu: *Funkcjonalność w branchu feature*. Zacommitować zmiany. (Zob. rys. poniżej).



```
MINGW64:/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git branch feature

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git checkout feature
Switched to branch 'feature'

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (feature)
$ git commit -a -m "Funkcjonalność w branchu feature"
[feature 5707328] Funkcjonalność w branchu feature
1 file changed, 2 insertions(+), 1 deletion(-)

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (feature)
$
```

Zamiast 2 komend:

```
git branch feature
```

```
git checkout feature
```

można użyć jednej, która jednocześnie tworzy nowy branch i przechodzi do niego:

```
git checkout -b feature
```


3.4.1.2. Szybka poprawka Hot Fix

W momencie pracy nad nową funkcjonalnością może się zdarzyć, że np. zostanie wykryty jakiś błąd w wersji podstawowej projektu (która już działa i jest wykorzystywana przez klienta) i konieczne będzie naniesienie poprawek. Funkcjonalność w branchu *feature* jest jednak w fazie, w której jej zakończenie jest niemożliwe. Nie możemy zatem nanieść żądanych poprawek i zakończyć brancha *feature* (scalać z *master*). W takim przypadku przełączamy się na branch *master*, gdzie nie ma jeszcze nowej funkcjonalności i tam wprowadzamy konieczne poprawki.

Dzięki temu zabiegowi mamy działającą wersję w gałęzi *master* wraz z naniesionymi poprawkami oraz spokojnie możemy kontynuować pracę nad nową funkcjonalnością w branchu *feature*.

ZADANIE

Przełączyć się na branch *master*, dodać w pliku *feature.txt* linię tekstu: *Poprawka HotFix*. Linia ta powinna znajdować się między liniami 1 i 2. Zacommitować zmiany.

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (feature)
$ git checkout master
Switched to branch 'master'

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git commit -a -m "hotfix w feature.txt"
[master 60164b0] hotfix w feature.txt
1 file changed, 3 insertions(+), 1 deletion(-)

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

3.4.2. Scalanie gałęzi (merge)

Do naszej głównej gałęzi projektu została wprowadzona poprawka i można spokojnie wrócić do pracy nad rozwojem projektu. Chcielibyśmy jednak pracować na możliwie najnowszych źródłach, dlatego potrzebujemy również zmian, które zostały wprowadzone w gałęzi *master*.

Jednym z możliwych sposobów, by to osiągnąć, jest **zmerge'owanie** gałęzi *master* do gałęzi *feature*. W tym celu przechodzimy do gałęzi *feature* i wywołujemy komendę

```
git merge master
```

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git checkout feature
Switched to branch 'feature'

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (feature)
$ git merge master
Auto-merging feature.txt
Merge made by the 'recursive' strategy.
 feature.txt | 1 +
1 file changed, 1 insertion(+)

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (feature)
$
```

Udało się automatycznie przenieść wszystkie zmiany z *master* do *feature*. Czasami jednak mogą wystąpić konflikty, ale do tego wrócimy.

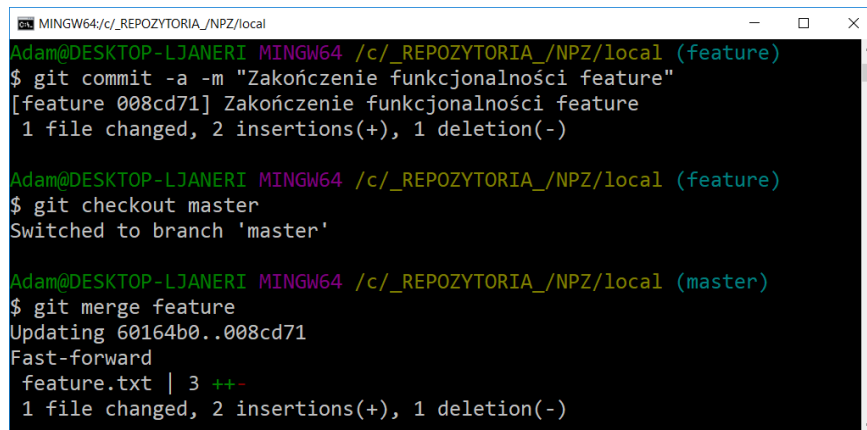
Kończymy rozwijanie nowej funkcjonalności. Teraz można gotową funkcjonalność zacommitować i zmerge'ować do gałęzi *master*.

ZADANIE

Będąc w gałęzi *feature* dodać na końcu pliku *feature.txt* następującą linię tekstu: *Zakończenie funkcjonalności feature*. Zacommitować zmiany. Scalić gałąź *feature* z gałęzią *master* wykorzystując polecenie:

```
git merge feature
```

UWAGA: Polecenie to należy wykonać po przełączeniu się na gałąź *master*.



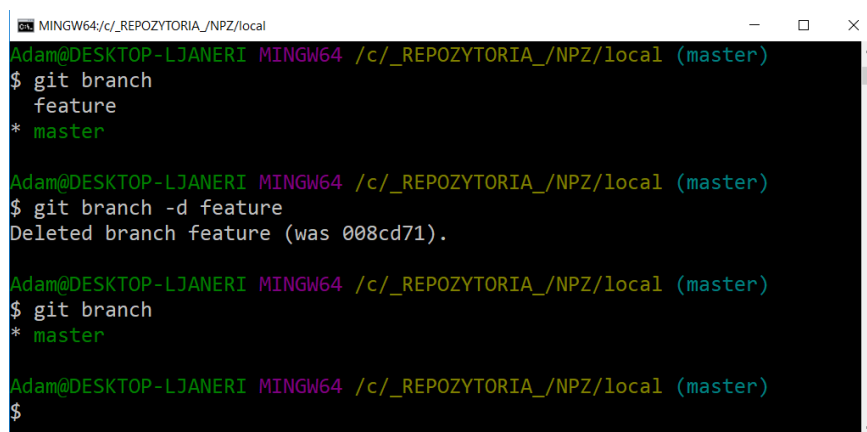
```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (feature)
$ git commit -a -m "Zakończenie funkcjonalności feature"
[feature 008cd71] Zakończenie funkcjonalności feature
1 file changed, 2 insertions(+), 1 deletion(-)

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (feature)
$ git checkout master
Switched to branch 'master'

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git merge feature
Updating 60164b0..008cd71
Fast-forward
 feature.txt | 3 ++-
1 file changed, 2 insertions(+), 1 deletion(-)
```

Wszystkie zmiany zostały naniesione na gałąź *master*. Tym razem jednak nie powstał commit merge'owy, ponieważ git mógł skorzystać z mechanizmu Fast-forward. Stało się tak, ponieważ podczas łączenia zmian wystarczyło przesunąć wskaźnik gałęzi i nie trzeba było wykonywać pełnego merge'owania zmian.

Po zmerge'owaniu brancha z nową funkcjonalnością można już go usunąć:



```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git branch
  feature
* master

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git branch -d feature
Deleted branch feature (was 008cd71).

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git branch
* master

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

Jak widać na powyższym obrazku, pozostał tylko branch *master*.

3.4.3. Konflikty scalania

Nie zawsze jednak scalanie gałęzi przebiega bezproblemowo. Mamy wtedy do czynienia z konfliktami w kodzie, które należy rozwiązać. Dzieje się tak najczęściej, jeżeli dany fragment kodu będzie edytowany na obu gałęziach.

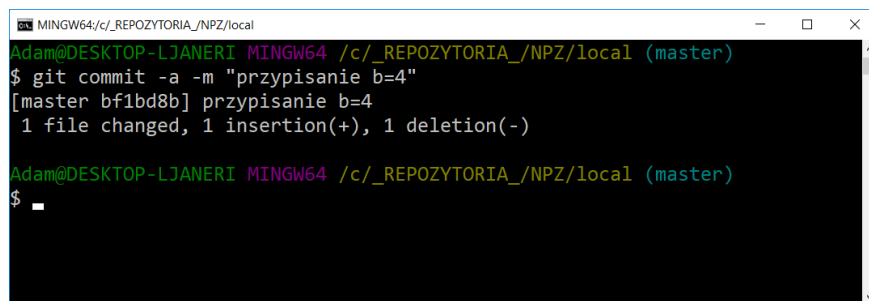
ZADANIE

W gałęzi *master* utworzyć plik *main.cpp* i umieścić w nim kod:

```
int a = 2;
int b = 3;
int main(void)
{
    if(a+b>5)
        return 1;
    else
        return 0;
}
```

Dodać zmiany do repozytorium i zacommitować je. Utworzyć nową gałąź o nazwie *my_branch* (nie przechodzić do niej, pozostać w *master*!)

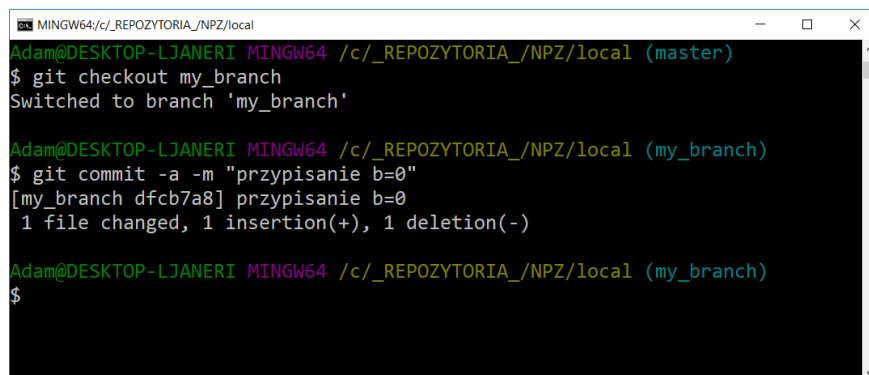
Następnie zmodyfikować kod przypisując do zmiennej *b* wartość 4 i wykonać commit zgodnie z poniższym rysunkiem:



```
MINGW64~/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git commit -a -m "przypisanie b=4"
[master bf1bd8b] przypisanie b=4
1 file changed, 1 insertion(+), 1 deletion(-)

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$
```

Przejsć do gałęzi *my_branch* i zmodyfikować kod programu przypisując do zmiennej *b* wartość 0. Wykonać commit:

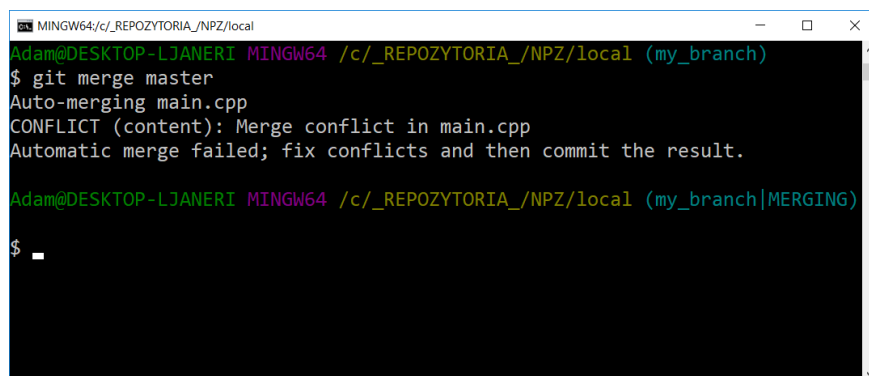


```
MINGW64~/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (master)
$ git checkout my_branch
Switched to branch 'my_branch'

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (my_branch)
$ git commit -a -m "przypisanie b=0"
[my_branch dfcb7a8] przypisanie b=0
1 file changed, 1 insertion(+), 1 deletion(-)

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (my_branch)
$
```

Pozostając w gałęzi *my_branch* scalić ją z gałęzią *master*:



```
MINGW64~/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (my_branch)
$ git merge master
Auto-merging main.cpp
CONFLICT (content): Merge conflict in main.cpp
Automatic merge failed; fix conflicts and then commit the result.

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (my_branch|MERGING)
$
```

Podczas próby automatycznego merge'owania wystąpił konflikt i będziemy musieli dokonać ręcznego połączenia zmian. Sprawdźmy status repozytorium:

```
MINGW64/c/_REPOZYTORIA_/NPZ/local
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (my_branch|MERGING)
$ git status
On branch my_branch
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")
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (my_branch|MERGING)
$
```

Otrzymujemy szersze informacje o konflikcie, m.in. że dotyczy on pliku *main.cpp*. Ponadto zawartość pliku konfliktowego zmieniła się:

```
main.cpp
1  int a = 2;
2  <<<<<< HEAD
3  int b = 0;
4  =====
5  int b = 4;
6  >>>>>> master
7
8  int main(void)
9  {
10     if(a+b>5)
11         return 1;
12     else
13         return 0;
14 }
```

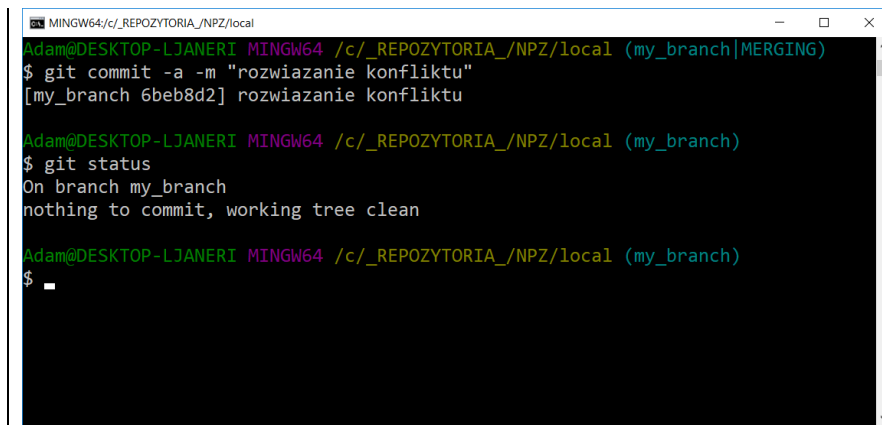
GIT umieścił kod z obu gałęzi między specjalnymi znacznikami.

Teraz należy rozwiązać konflikt poprzez poprawienie kodu oraz zacommitowanie go do repozytorium. Trzeba też usunąć znaczniki dodane podczas merge'owania.

Przeniesienie pliku do poczekalni oznacza w Gicie rozwiązanie konfliktu.

ZADANIE

Rozwiązać konflikt, który wystąpił podczas merge'owania poprzez pozostawienie linii kodu z wartością *b=4* oraz usunięcie zbędnych linii w pliku *main.cpp*. Zacommitować zmiany i sprawdzić status repozytorium:

A screenshot of a terminal window with a black background and green text. The window title is 'MINGW64/c/_REPOZYTORIA_/NPZ/local'. The user is 'Adam@DESKTOP-LJANERI'. The terminal shows the following commands and output:

```
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (my_branch|MERGING)
$ git commit -a -m "rozwiązanie konfliktu"
[my_branch 6beb8d2] rozwiązanie konfliktu

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (my_branch)
$ git status
On branch my_branch
nothing to commit, working tree clean

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/local (my_branch)
$
```

3.4.3.1. Narzędzia graficzne do rozwiązywania konfliktów (mergetool)

Istnieje również możliwość rozwiązania konfliktów, wspomagając się narzędziami graficznymi. Zamiast edytować pliki ręcznie, wystarczy uruchomić komendy:

```
git config --global merge.tool kdiff3
```

```
git mergetool
```

Pierwsza komenda ustawia domyślny edytor na kdiff3, a druga go wywołuje. Można też posłużyć się innym narzędziem, np.: opendiff, kdiff3, tkdiff, xxdiff, meld, tortoisemerge, gvimdiff, diffuse, diffmerge, ecmerge, p4merge, codecompare, emerge, vimdiff.

Po rozwiązaniu konfliktu i zapisaniu plików z poziomu wybranego narzędzia można już zacommitować zmiany.

3.5. Praca ze zdalnym repozytorium [remote]

Dzięki wykorzystaniu możliwości zdalnego repozytorium można współpracować z innymi osobami, nie ograniczając się już tylko do pracy na jednym komputerze. Git umożliwia współpracę jednocześnie z kilkoma różnymi zdalnymi repozytoriami. Można z nich pobierać kod, wysyłać swoje zmiany oraz zarządzać gałęziami kodu itp.

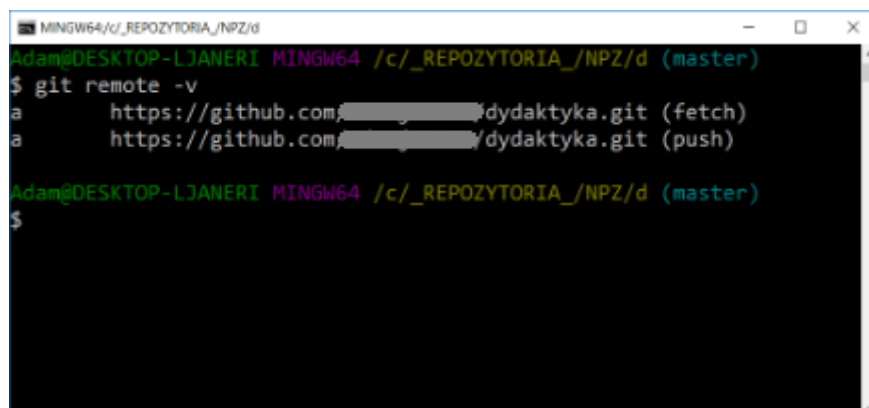
3.5.1. Wyświetlanie zdalnych repozytoriów

Żeby wyświetlić listę wszystkich skonfigurowanych zdalnych repozytoriów, należy użyć komendy:

```
git remote
```

lub aby wyświetlić szersze informacje o zdalnych repozytoriach:

```
git remote -v
```

A screenshot of a terminal window with a black background and green text. The window title is 'MINGW64/c/_REPOZYTORIA_/NPZ/d'. The user is 'Adam@DESKTOP-LJANERI'. The terminal shows the following commands and output:

```
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/d (master)
$ git remote -v
a      https://github.com/[redacted]/dydaktyka.git (fetch)
a      https://github.com/[redacted]/dydaktyka.git (push)

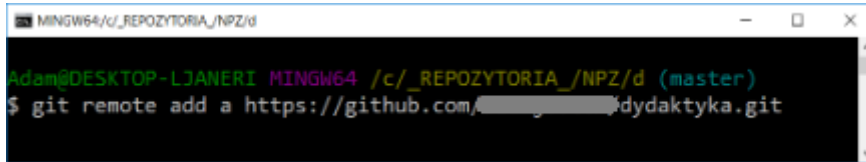
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/d (master)
$
```

3.5.2. Dodawanie zdalnego repozytorium

Konfiguracja nowego zdalnego repozytorium polega na wywołaniu komendy w ogólnym formacie:

```
git remote add [skrót] [url]
```

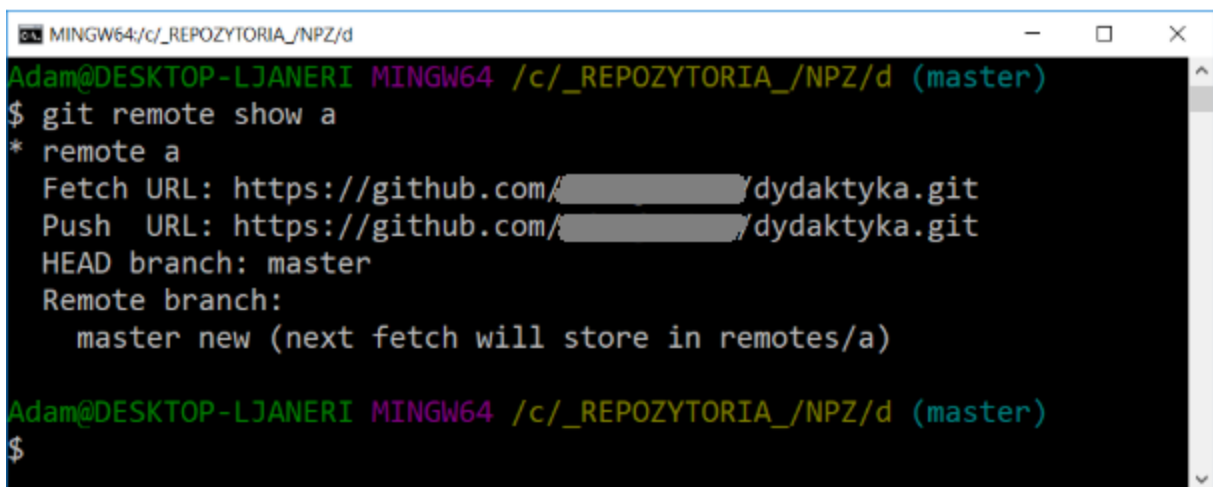
Od tego momentu nowe zdalne repozytorium jest już dowiązane do naszego lokalnego.



```
MINGW64:/c/_REPOZYTORIA_/NPZ/d
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/d (master)
$ git remote add a https://github.com/[redacted]/dydaktyka.git
```

Wyświetlenie informacji dotyczących zdalnego repozytorium:

```
git remote show [skrót]
```



```
MINGW64:/c/_REPOZYTORIA_/NPZ/d
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/d (master)
$ git remote show a
* remote a
  Fetch URL: https://github.com/[redacted]/dydaktyka.git
  Push URL: https://github.com/[redacted]/dydaktyka.git
  HEAD branch: master
  Remote branch:
    master new (next fetch will store in remotes/a)

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/d (master)
$
```

3.5.3. Pobranie zmian z globalnego repozytorium

Należy zauważyć, że pobranie zmian ze zdalnego repozytorium oraz połączenie ich z lokalnym repozytorium to dwie całkowicie niezależne czynności.

Polecenie

```
git fetch [skrót]
```

pobiera wszystkie zmiany, których jeszcze nie było lokalnie.

Po wykonaniu tego polecenia zawartość lokalnego repozytorium nie uległa zmianie - GIT nie zmodyfikował jeszcze lokalnej kopii repozytorium. Konieczne jest jeszcze naniesienie pobranych zmian na lokalne repozytorium. W tym celu wykorzystać można merge'owanie:

```
git merge [skrót]/[branch]
```

```
MINGW64/C:/_REPOZYTORIA/_NPZ/d
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA/_NPZ/d (master)
$ git fetch a
remote: Enumerating objects: 94, done.
remote: Counting objects: 100% (94/94), done.
remote: Compressing objects: 100% (70/70), done.
remote: Total 94 (delta 21), reused 87 (delta 18), pack-reused 0
Unpacking objects: 100% (94/94), done.
From https://github.com/[redacted]/dydaktyka
* [new branch]      master      -> a/master

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA/_NPZ/d (master)
$ git merge a/master

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA/_NPZ/d (master)
$
```

W tym momencie może oczywiście dojść do konfliktów, jednak ich rozwiązywanie przebiega analogicznie jak przy łączeniu lokalnych gałęzi.

Powyższe dwa kroki zazwyczaj są wykonywane bezpośrednio po sobie, dlatego zostały połączone w jedną komendę:

`git pull [skrót] [branch]`

To polecenie jest równoznaczne z pobraniem najnowszych zmian z globalnego repozytorium, którego skróconą nazwę należy podać w miejscu [skrót] - oraz zmerge'owaniem ich do lokalnego brancha (należy wskazać nazwę gałęzi).

```
MINGW64/C:/_REPOZYTORIA/_NPZ/d
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA/_NPZ/d (master)
$ git pull a master
remote: Enumerating objects: 94, done.
remote: Counting objects: 100% (94/94), done.
remote: Compressing objects: 100% (70/70), done.
remote: Total 94 (delta 21), reused 87 (delta 18), pack-reused 0
Unpacking objects: 100% (94/94), done.
From https://github.com/[redacted]/dydaktyka
* branch            master      -> FETCH_HEAD
* [new branch]      master      -> a/master

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA/_NPZ/d (master)
$
```

3.5.4. Wysyłanie lokalnych zmian do repozytorium globalnego

Jeżeli dokonane zostały zmiany w lokalnej wersji repozytorium i chcemy je wysłać na zewnątrz (do globalnego repozytorium), użyjemy komendy:

`git push [skrót] [branch]`

```
MINGW64/c/_REPOZYTORIA_/NPZ/d
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/d (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        abc.txt

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

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/d (master)
$ git add .

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/d (master)
$ git commit -m "abc"

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/d (master)
$ git commit -m "abc"
[master 49241ff] abc
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 abc.txt

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/d (master)
$ git push a master
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 687 bytes | 343.00 KiB/s, done.
Total 6 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/.../dydaktyka.git
c1d2a40..49241ff master -> master
```

Nadpisanie zmian w zdalnym repozytorium powiedzie się tylko, jeżeli od czasu ostatniego pobierania zmian z zewnątrz, nikt nie dodał swoich zmian. Jeżeli już do tego doszło, musimy najpierw takie zmiany pobrać i połączyć je ze swoimi.

3.5.5. Klonowanie zdalnego repozytorium

Jeżeli chcemy uzyskać kopię istniejącego już repozytorium GITA - na przykład projektu, w którym chcemy zacząć się udzielać i wprowadzać własne zmiany – niezbędnym poleceniem będzie

```
git clone [url]
```

Po wykonaniu polecenia `git clone` zostanie pobrana każda rewizja, każdego pliku w historii projektu. W praktyce nawet jeśli dysk serwera zostanie uszkodzony, możemy użyć któregoś z dostępnych klonów aby przywrócić serwer do stanu w jakim był w momencie klonowania.

```
MINGW64/c/_REPOZYTORIA_/NPZ/dydaktyka
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ
$ git clone https://github.com/.../dydaktyka.git
Cloning into 'dydaktyka'...
remote: Enumerating objects: 100, done.
remote: Counting objects: 100% (100/100), done.
remote: Compressing objects: 100% (74/74), done.
remote: Total 100 (delta 23), reused 92 (delta 19), pack-reused 0
Receiving objects: 100% (100/100), 12.12 MiB | 394.00 KiB/s, done.
Resolving deltas: 100% (23/23), done.

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ
$ cd dydaktyka

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/dydaktyka (master)
$ ls
abc.txt 'Aplikacje Internetowe I/' 'Narzędzia Pracy Zespołowej/'

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/dydaktyka (master)
$
```

3.5.6. Inicjalizacja zdalnego repozytorium

Do inicjalizacji zdalnego repozytorium służy polecenie

`git init --bare [ścieżka]`

```
MINGW64/c/_REPOZYTORIA_/NPZ/globalne
Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ
$ git init --bare globalne
Initialized empty Git repository in C:/_REPOZYTORIA_/NPZ/globalne/

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ
$ cd globalne/

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/globalne (BARE:master)
$ ls
config  description  HEAD  hooks/  info/  objects/  refs/

Adam@DESKTOP-LJANERI MINGW64 /c/_REPOZYTORIA_/NPZ/globalne (BARE:master)
$
```

Powyższe polecenie tworzy katalog repozytorium zdalnego wraz w odpowiednią jego zawartością.

Jeżeli znajdujemy się w katalogu, który ma być repozytorium zdalnym – wówczas nie podajemy ścieżki.

Aby lepiej zrozumieć czym jest zdalne repozytorium i jak działa, utworzymy 3 katalogi – pierwszy będzie pełnił funkcję naszego zdalnego repozytorium, zaś dwa pozostałe będą repozytoriami lokalnymi. Jak łatwo zauważyć wszystkie katalogi będą znajdować się na tym samym komputerze, który będzie jednocześnie pełnił rolę serwera oraz lokalnego komputera.

ZADANIE

1. Utworzyć na pulpicie folder ze swoim imieniem, a w nim umieścić 3 katalogi: *global*, *local* i *local2*.
2. Przejść do folderu *local* i utworzyć w nim nowe repozytorium.
3. Przejść do folderu *global* i utworzyć w nim nowe repozytorium globalne/zdalne korzystając z komendy `git init --bare`.
4. Wrócić do folderu *local* i utworzyć w nim plik tekstowy *lokomotywa.txt*, w którym umieścić tekst:

Stoi na stacji lokomotywa,

Ciężka, ogromna i pot z niej spływa:

Tłusta oliwa.

Stoi i sapie, dyszy i dmucha,
Żar z rozgrzanego jej brzucha bucha:
Buch - jak gorąco!
Uch - jak gorąco!
Puff - jak gorąco!
Uff - jak gorąco!
Już ledwo sapie, już ledwo zipie,
A jeszcze palacz węgiel w nią sypie.

5. Dodać nowy plik do repozytorium i wykonać commit z opisem „*ciuchcia*”.
6. Ustawić jako zdalne repozytorium folder *global*.
7. „Wypchnąć” aktualną wersję repozytorium znajdującą się w katalogu *local* do repozytorium globalnego.
8. Zadeklarować GITowi, aby nie zawierał plików *.docx* w repozytorium i zapisać tą zmianę z opisem „*ignorowanie .docx*”.
9. Utworzyć plik *smieci.docx* o treści: „*nie ma tu nic ciekawego*”.
10. Zapisać log zmian w repozytorium w pliku *log.txt* po czym dodać go do repozytorium i zapisać z opisem „*logi*”.
11. Uaktualnić zdalne repozytorium.
12. W pliku *lokomotywa.txt* zamienić wszystkie słowa *gorąco* na *zimno*.
13. Wygenerować plik o nazwie *roznica.txt* zawierający różnice w stosunku do oryginału pliku *lokomotywa.txt*.
14. Przejść do katalogu *local2* i sklonować do niego zawartość zdalnego repozytorium *global*.
15. Do pliku *lokomotywa.txt* (w katalogu *local2/global*) dodać kolejną zwrotkę:

Wagony do niej podoczepiali
Wielkie i ciężkie, z żelaza, stali,
I pełno ludzi w każdym wagonie,
A w jednym krowy, a w drugim konie,
A w trzecim siedzą same grubasy,
Siedzą i jedzą tłuste kiełbasy,
A czwarty wagon pełen bananów,
A w piątym stoi sześć fortepianów,
W szóstym armata - o! jaka wielka!
Pod każdym kołem żelazna belka!
W siódmym dębowe stoły i szafy,
W ósmym słoń, niedźwiedź i dwie żyrafy,
W dziewiątym - same tuczone świnie,
W dziesiątym - kufry, paki i skrzynie.
A tych wagonów jest ze czterdzieści,
Sam nie wiem, co się w nich jeszcze mieści.
Lecz choćby przyszło tysiąc atletów
I każdy zjadłby tysiąc kotletów,
I każdy nie wiem jak się wytężał,
To nie udźwigną, taki to ciężar.

16. Zapisać zmiany lokalnie z opisem „*druga strofa*” po czym wysłać je do *global*.
17. Wrócić do folderu *local* i zapisać do repozytorium lokalnego tylko plik *lokomotywa.txt* z opisem „zimno”.
18. Założyć nową gałąź „zimno”.
19. Będąc nadal w *master* proszę wysłać zmiany do *global* - wymaga to rozwiązania konfliktu. W *lokomotywa.txt* ma pozostać pierwsza strofa ze słowem „*zimno*” oraz druga strofa oryginalna. Opis commita proszę zostawić domyślny.