Systemy kontroli wersji: GIT

Jakub Wierzbowski

AGENDA

Komendy terminala

Git - Wprowadzenie

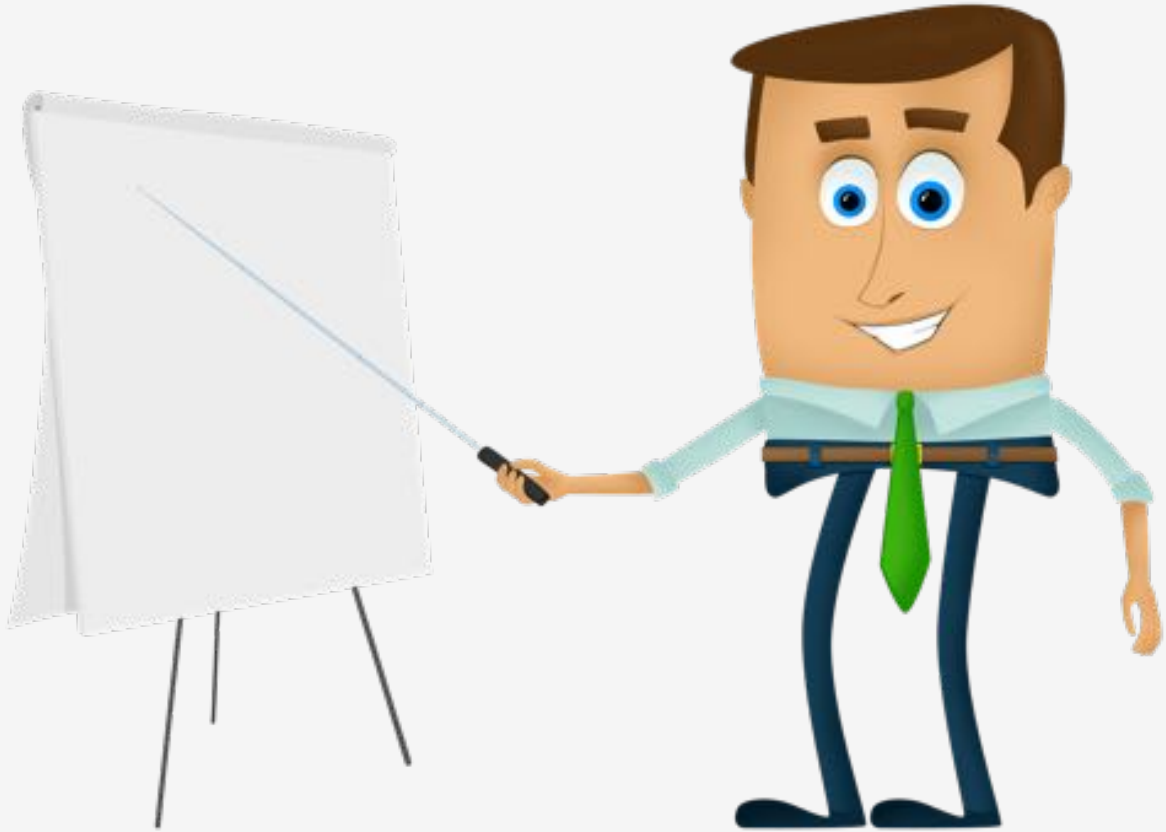
Repozytoria

Przechowalnia - Stage

Branche

Scalanie branchy - merge

Schowek - stash



Zadanie: Przygotowanie środowiska pracy

1. Zainstaluj Git-a: https://git-scm.com/download
2. Uruchom Terminal:
   - Windows: **Git Bash**
   - Linux & Mac: **Terminal**

Budowa polecenia

Przykład prostego polecenia wyświetlającego pliki w aktualnym katalogu:

- ls

Polecenie może składać się z nieobowiązkowych opcji oraz argumentów:

- ls [opcja/e] [argument/y]
- Przykład prostego polecenia wyświetlającego wszystkie pliki (-a) w katalogu użytkownika (~):
  ls -a ~

Opcje mogą być wprowadzane w postaci:

- Flag – jeden myślnik (-) + jedna litera
  ls -a
- Pełnych nazw – dwa myślniki (--) + nazwa polecenia
  ls --all

Wprowadzanie wielu opcji:

- ls -a -h
- ls -ah
- ls --all --human-readable

Zadanie: Opcje polecenia

1. Wyświetl pliki w katalogu domowym (nie pokazuj ukrytych)
2. Wyświetl **wszystkie** pliki w katalogu domowym (łącznie z ukrytymi)
3. Wyświetl pliki w katalogu domowym w postaci **listy (-l)**
4. Wyświetl **wszystkie** pliki w katalogu domowym w postaci **listy**

Polecenia terminala – dokumentacja i historia

Dokumentację polecenia można uzyskać wpisując („q” - wyjście)

- **man [polecenie]**
  np. **man mkdir**

Historię Wprowadzonych poleceń uzyskujemy komendą:

- **history**

Poprzednio wprowadzone polecenia przeglądamy **klawiszami strzałek** (góra-dół)

Wyszukiwanie wprowadzonych poleceń: **ctrl+R**

Zadanie: Używanie dokumentacji

1. Znajdź w **dokumentacji** polecenia **ls** opcje odpowiadającą za rekursywne wyświetlania zawartości w podkatalogach
2. Wyświetl wszystkie pliki wraz z podkatalogami w katalogu użytkownika

Podstawowe polecenia terminala

- ls - wyświetlenie zawartości katalogu
- pwd - wypisanie ścieżki bieżącego katalogu
- cd <sciezka> - przejście do katalogu podanego w <sciezka> np. *cd /home/jwierzbo*
  - cd ~ - przejście do katalogu domowego
  - cd .. - przejście do katalogu wyżej (rodzica)
- touch <nazwa_pliku> - zmienia czas dostępu i modyfikacji pliku, lub jeśli plik nie istnieje- tworzy go
- echo – powtarza na standardowym wyjściu słowa podane w argumencie np. *echo Hello World*
  - echo > test.txt - utworzenie nowego pliku *test.txt* (znak **>** przekierowuje wyjście na plik)
  - echo hello world > test.txt - utworzenie nowego pliku z zawartością „*hello world*”
  - echo dopisek >> test.txt – dopisanie tekstu „*dopisek*” do istniejącego pliku (znak **>>** dokleja wyjście z programu na koniec pliku)
- mkdir <nazwa> - utworzenie nowego katalogu np. *mkdir nowy_katalog*
- cp <zrodlu> <cel> - kopia plików lub katalogów np. *cp /usr/x/test.txt ~/nowy.txt*
- mv <zrodlo> <cel> - przenieś plik np. *mv /usr/x/test.txt ~/nowy.txt*
- rm <plik> - usuniecie plików (lub katalogów z opcją `-r`)
  - rm -Rf stary_katalog – usunięcie katalogu wraz z zawartością, bez potwierdzania (f – force)
- vim plik.txt - edytor tekstowy
  edycja: `i`
  zapis: `Esc` + `:w` + `Enter`
  koniec pracy: `Esc` + `:q` + `Enter`

Zadanie: Podstawowe polecenia

1. Przejdź do katalogu domowego i wyświetl wszystkie pliki. Czy znajdują się tam pliki ukryte?
2. W katalogu domowym utwórz katalog **zadanie2**
3. Utwórz katalog **zadanie3**
4. Wejdź do katalogu **zadanie3** i przenieś tu katalog **zadanie2**
   Jak wyglądałaby komenda przeniesienia, jeżeli wykonalibyśmy ją z katalogu domowego?
5. Wróć do katalogu roboczego i usuń katalog **zadanie3**

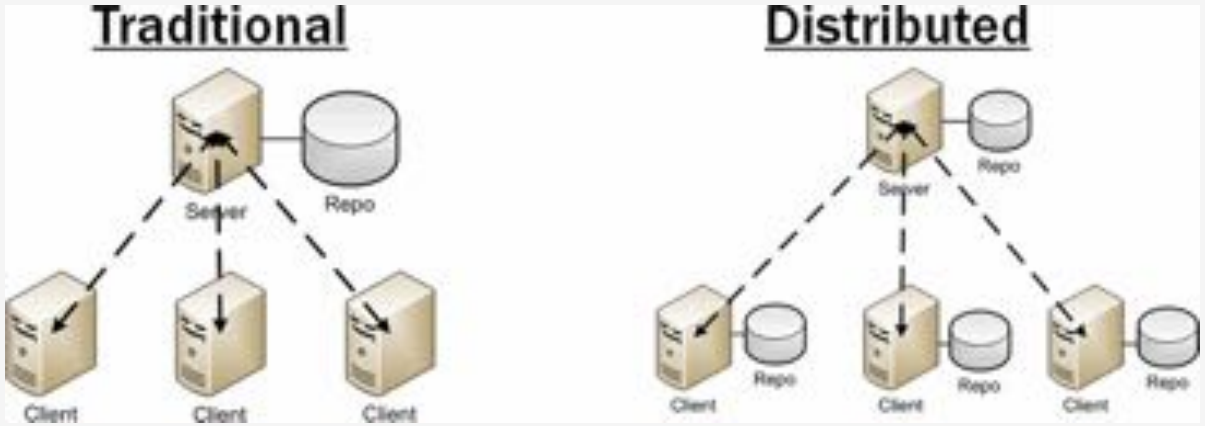
Zadanie: Podstawowe polecenia cz.2

1. Utwórz strukturę katalogów:
   tmpX/
   java/
   resources/
2. Utwórz plik ***tmpX/java/Main.java*** z zawartością:
   public class Main
3. Skopiuj plik ***tmpX/java/Main.java*** do ***tmpX/resources/Main.java.bak***
4. Zmień nazwę ***tmpX/resources/Main.java.bak*** na ***Main.bak***
5. Edytuj plik **Main.bak** tak, aby w środku znalazł się wpis:
   public class Main.bak
6. Usuń katalog katalog ***src*** razem z zawartością

Systemy kontroli wersji – *VCS (Version control systems)*

- służy do śledzenia zmian w kodzie źródłowym
- pomaga programistom pracować na tym samym kodzie źródłowym
- pomaga łączyć rożne zmiany dokonane przez wiele osób w rożnym czasie

VCS: centralizacja vs rozproszenie



Systemy kontroli wersji – przegląd

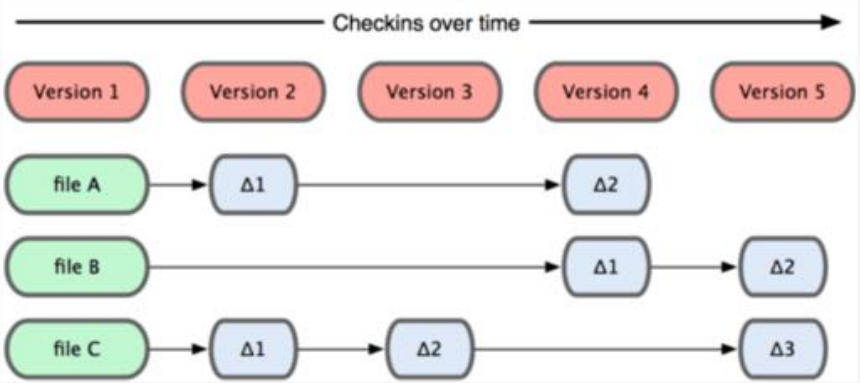
- Lokalne:
  - foldery, pliki, kopie
- Scentralizowane:
  - Subversion (SVN), CVS, Perforce, Clear Case
- Rozproszone:
  - GIT, BitKeeper, Bazaar, Mercurial

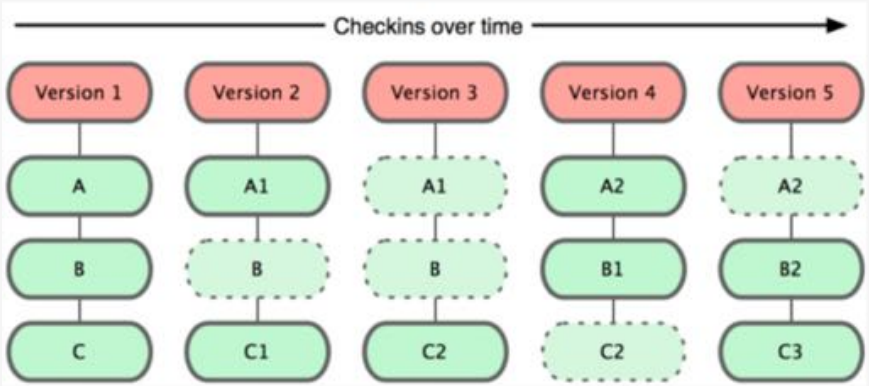
Git – dlaczego?

- Szybkość
- Prosta konstrukcja
- Praca Offline (w przeciwieństwie do innych VCS) – niemal każda operacja jest lokalna!
- Wsparcie dla istniejących protokołów sieciowych (HTTP/s, SSH, git)
- Silne wsparcie dla nieliniowego rozwoju (tysięcy równoległych gałęzi)
- Pełne rozproszenie
- Wydajna obsługa dużych projektów, takich jak jądro Linuksa (szybkość i rozmiar danych)

Git - Migawki, nie różnice

- Różnicowe podejście (np. Subversion): przechowywanie informacje jako zbiór plików i zmian dokonanych na każdym z nich w okresie czasu
- Migawki (ang. Snapshots) – Git tworzy obraz przedstawiający to jak wyglądają wszystkie pliki w danym momencie i przechowuje referencję do tej migawki. W celu uzyskania dobrej wydajności, jeśli dany plik nie został zmieniony, Git nie zapisuje ponownie tego pliku, a tylko referencję do jego poprzedniej, identycznej wersji, która jest już zapisana





Git – konfiguracja

- Pierwsze uruchomienie wymaga ”przedstawienia się” aplikacji.

```
git config --global user.name "TWOJE_IMIE"
git config --global user.email twój@email.com
```

- Wyświetlanie aktualnej konfiguracji:

```
git config --list
```

Zadanie: Konfiguracja Git-a

Dokonaj konfiguracji użytkownika oraz maila

(patrz poprzedni slajd)

Repozytoria w Git

Lokalne

- To taki lokalny folder z naszymi plikami
- Można pracować offline
- Tworzenie za pomocą narzędzia **git**

Zdalne

- Można porównać np. do Dropbox'a z którym synchronizujemy nasz lokalny folder
- Synchronizacja wymaga połączenia z siecią
- Przykłady serwisów w których można utworzyć zdalne repozytorium:
  - GitHub
  - BitBucket

Sumy kontrolne – mechanizm spójności danych

- Każdy obiekt Git'a (np. commit, branch, pojedynczy plik) posiada przypisaną **unikalną sumę kontrolną (taka sygnatura)**
- Metoda wyznaczania sumy kontrolnej:
  Funkcja skrótu / haszująca / mieszająca SHA-1
- **Funkcja skrótu w szybki sposób wylicza sumę kontrolną, w zależności od zawartości pliku (nigdy nie jest taka sama dla dwóch różnych plików)**
  np. po zmianie jednego z pikseli w dowolnej klatce dwugodzinnego filmu, zmieni się również wartość sumy kontrolnej!
- Git korzysta z wewnętrznej bazy danych, gdzie kluczami są skróty SHA-1, a wartościami pliki, lub struktury katalogów

- Przykład SHA-1:
  24b9da6552252987aa493b52f8696cd6d3b00373
- W przypadku git-a wystarczy użyć pierwsze 8 znaków: *24b9da65*

Repozytorium - tworzenie

Tworzenie nowego lokalnego repozytorium:

Utwórz folder, otwórz go i wykonaj komendę:

git init

Tworzenie kopii lokalnego repozytorium:

git clone /path/to/repository

Pobieranie kopii zdalnego repozytorium:

git clone https://github.com/XYZ.git

git clone https://github.com/XYZ.git **katalog**

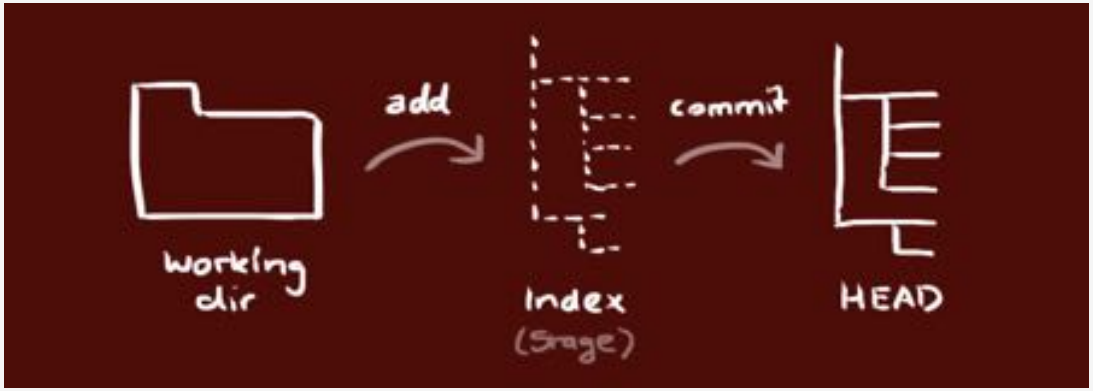
Zadanie: Lokalne repozytorium

1. Stwórz katalog o nazwie **repo1** w swoim katalogu domowym i przejdź do niego (podpowiedź: **~**, **mkdir** oraz **cd**)
2. Będąc w katalogu **repo1** zainicjalizuj repozytorium git-a

Git - obszary pracy w repozytorium lokalnym

Lokalne repozytorium ma postać "drzewa", które jest zarządzane przez Git-a:

- **Working Directory** (Katalog Roboczy) - tu trzymana jest nasza aktualna praca: wszystkie modyfikacje na plikach oraz nowe stworzone pliki
- **Index / Stage** (Przechowalnia / Kolejka) - miejsce do którego przenosimy pliki, które zamierzamy zapisać. Do przenoszenia plików używamy komendy ***git add***
- **HEAD** - znacznik JESTEŚ TUTAJ. Oznacza ostatni punkt zapisu danych (taki "save" z gry) **w lokalnym repozytorium.**
  Tworzenie punktów zapisu wykonujemy komendą ***git commit***



Git - obszary pracy: Przykład

Obszary działań git-a można przyrównać do procesu zbierania zdjęć:

- **Working Directory**- kamera, którą wykonujemy zdjęcia (tworzymy pliki)
- **Stage**- lista ujęć/zdjęć które nam się podobają i zamierzamy je zachować. Odkładanie wybranych zdjęć (plików) na stos wykonujemy poleceniem ***git add <plik>***
- **Repository**- finalnie zdjęcia, które wybraliśmy, umieszczane są w albumie (lokalne repozytorium). Umieszczanie zdjęć (zapisywanie plików) wykonujemy poleceniem ***git commit -m "Opis"***

add & commit

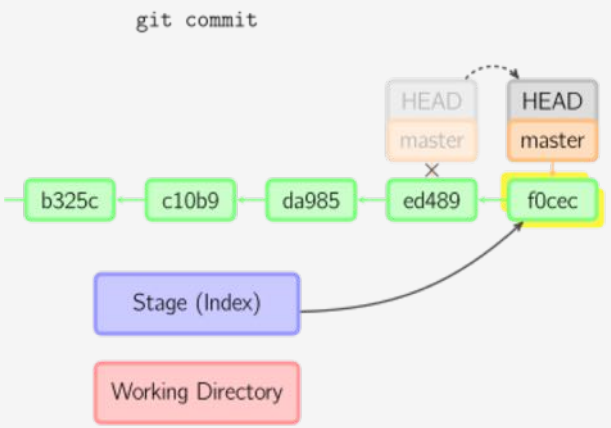
Tworzenie listy zmian do zapisu, czyli umieszczanie plików w przechowalni (Stage), wykonujemy poleceniem:

- git add <filename>
- git add . # dodaje do przechowalni wszystkie nowe i zmodyfikowane pliki, pomija usunięte
- git add -A # dodaje do kolejki wszystkie pliki: dodane, zmodyfikowane oraz usunięte

Stworzenie punktu zapisu ("save" w grze):

- git commit -m "Opis commitu"

Po wykonaniu commit'a nasz HEAD aktualizuje się - teraz wskazuję na najnowszy punkt zapisu.
Zmiana dotyczy jednak tylko lokalnego repozytorium.



Zadanie: add & commit

1. W katalogu **repo1** utwórz plik **test.txt** z zawartością „**hello**”. Do tego celu możesz wykorzystać polecenie:
   echo hello > test.txt
2. Dodaj stworzony plik do śledzenia
3. Zapisz zmiany używając komentarza: ”**Pierwszy commit**”
4. Zweryfikuj zadanie wywołując polecenie: **gitk**

GitHub – repozytorium zdalne

- Pozwala zapisać kopię naszego kodu w sieci – kopia zapasowa
- Dzięki prywatnym repozytoriom tylko zaproszeni użytkownicy mogą zobaczyć co się dzieje w naszym projekcie

Zadanie: Konto GitHub

1. Załóż konto w serwisie https://github.com
2. Stwórz pierwsze repozytorium **publiczne** o nazwie „**repo1**”:
   - Znaczek „+” (prawy górny róg) -> „New repository”

Łączenie lokalnego repozytorium ze zdalnym

Lokalne repozytorium (np. folder **repo1**) nie jest obecnie synchronizowane (połączone) z żadnym repozytorium zdalnym (np. repozytorium **repo1 na github.com**)

Dodawanie zdalnego repozytorium do lokalnego, wykonujemy poleceniem:
git remote add nazwa_zdalnego_repo https://github.com/XYZ.git
Gdzie:

- **nazwa_zdalnego_repo -** nazwa pod którą rejestrujemy zdalne repozytorium. Domyślnie używa się nazwy **origin**

Sprawdzenie listy zarejestrowanych repozytoriów zdalnych:
git remote -v

Synchronizacja repozytoriów - polecenia

- Aby połączyć lokalne repozytorium z GitHubem (co umożliwi nam wysyłanie danych na serwer):
  `git remote add origin https://github.com/{user}/repo1.git`
- Wyślij zmiany na serwer:
  `git push origin master`
- Pobierz zmiany z serwera i aplikuj je:
  `git pull origin master`

ZAPAMIĘTAJ:

origin – nazwa pod którą zarejestrowaliśmy zdalne repozytorium

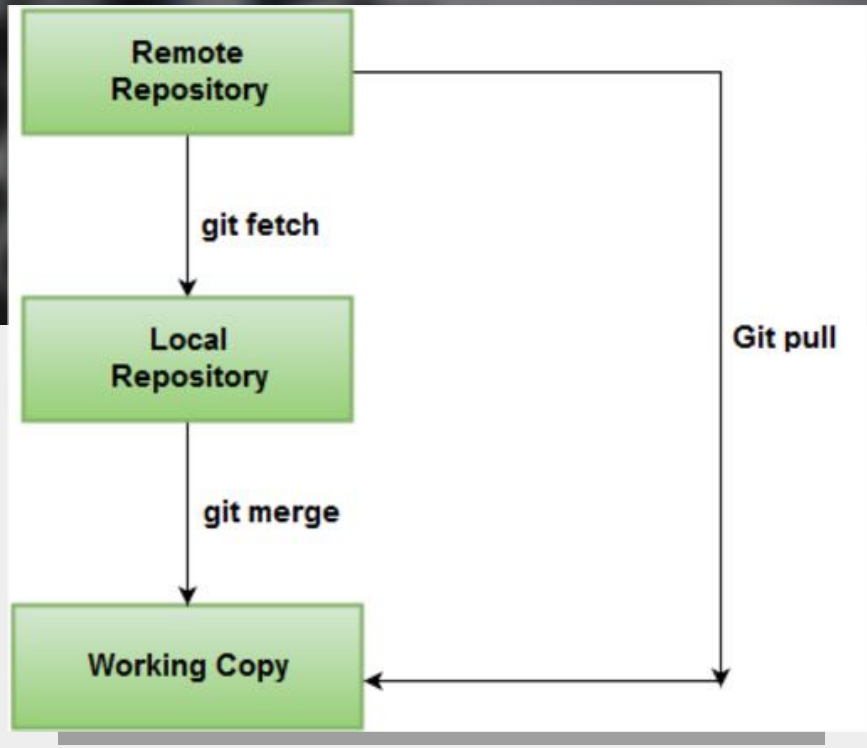
master – nazwa aktualnej gałęzi (brancha) na której pracujemy – będzie o tym później ☺

Zadanie

1. Skonfiguruj swoje repozytorium lokalne (**repo1**) tak, aby było podłączone do repozytorium na github.com, które stworzyliśmy wcześniej (**repo1**)
2. Sprawdź listę repozytoriów zdalnych – zweryfikuj poprawność adresu serwera zdalnego
3. Wyślij zmiany na serwer zdalny
4. Zaobserwuj na stronie www (github.com) Twojego repozytorium zdalnego jakie nastąpiły zmiany
5. Zweryfikuj zadanie wywołując polecenie: **gitk**

git pull vs. git fetch

- W Git można pobrać zmiany, bez nakładania ich na pliku w katalogu roboczym, za pomocą polecenia:
  git fetch
- W każdej chwili można nałożyć pobrane zmiany używając:
  git merge
- Zazwyczaj jednak chcemy pobrać zmiany i od razu je zaaplikować, dlatego używamy:
  git pull



Git – klonowanie zdalnego repozytorium

- Poznaliśmy sposób na stworzenie własnego repozytorium od podstaw.
- Dołączając do pracy (projektu) możemy spotkać już rozwijany kod i istniejące repozytoria
- Pobieranie (klonowanie) repozytoriów to drugi sposób na stworzenie lokalnego repozytorium
- W celu pobrania zdalnego repozytorium do lokalnego folderu (nie musi istnieć - zostanie utworzony automatycznie) użyj polecania:

  git clone https://github.com/XYZ.git

  git clone https://github.com/XYZ.git katalog

Zadanie: Pobieranie zdalnego repozytorium

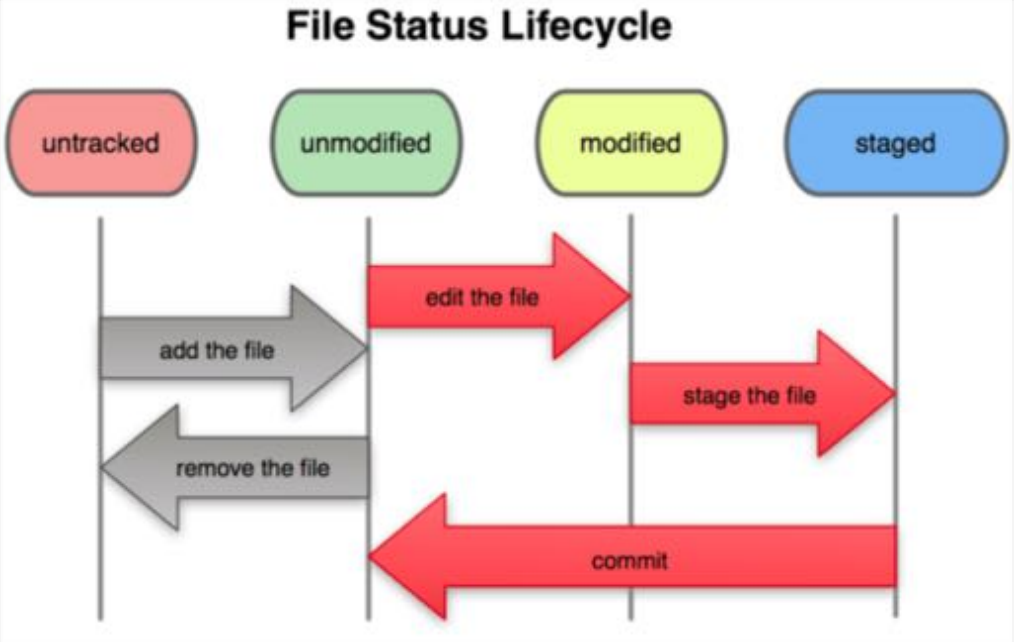
1. Wejdź na stronę https://github.com/jakubwierzbowski/REST-movie-service i skopiuj link do klonowania repozytorium za pomocą **HTTPS**
2. Używając polecenia **git clone** pobierz repozytorium
3. Wejdź do utworzonego katalogu i porównaj listę plików z tymi na stronie www (polecenie do listowania: **ls**)

Zadanie: Śledzenie plików

1. W katalogu **repo1** utwórz plik **test2.txt** oraz **test3.txt**
2. Dodaj plik **test2.txt** do śledzenia
3. Zapisz zmiany używając komentarza: ”**Sledzenie plikow**”
4. Wyślij zmiany na serwer
5. Porównaj listę plików na **github.com** oraz lokalnie w katalogu **repo1**

Cykl życia stanu plików

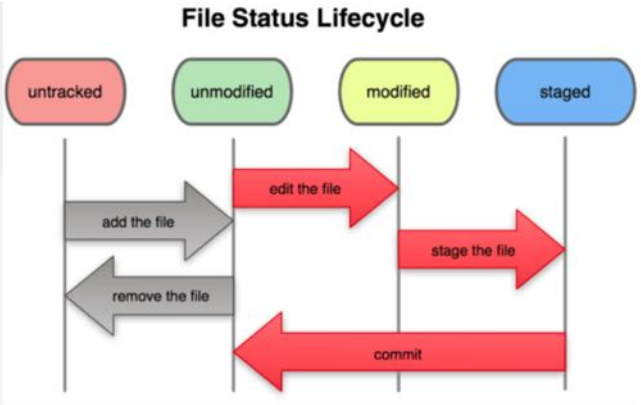
- nieśledzone (**untracked**)- nowo dodane pliki, które nie znajdują się w historii ani poczekalni. Operacja **add** pozwala dodać je do śledzenia
- Śledzone (tracked):
  - aktualne (**unmodified**)- pliki zgodne ze stanem zapisanym w repozytorium (po edycji stają się zmodyfikowane)
  - zmodyfikowane (**modified**)- pliki zmienione w stosunku do wersji zatwierdzonej w repozytorium. Aktualny obszar roboczy. Operacja **add** pozwala je zakolejkować
  - zakolejkowane (**staged**)- pliki w przechowalni (Stage) gotowe do zatwierdzenia (operacja **commit**)- po zatwierdzeniu stają się aktualne



Sprawdzenie stanu plików:

```
git status
```

Poczekalnia (stage) – kolejkowanie plików



Dodawanie pliku do przechowalni (plik w stanie **staged**):
- **git add <ścieżka do pliku lub katalogu>**
- **git add –A** LUB **git add .**

Usuwanie pliku z repozytorium, zaprzestanie śledzenia pliku
(plik wraca do stanu **untracked,** dalej jednak istnieje tak jak nowo utworzony plik).
W praktyce raczej rzadko używane- zazwyczaj po prostu usuwamy plik ☺:
- **git rm --cached <ścieżka do pliku>**

Przywracanie stanu przechowalni - pliki wracają do obszaru roboczego w stanie **modified**, zachowując wprowadzone zmiany:
- **git reset <ścieżka do pliku>**
- **git reset**

Przywracanie stanu repozytorium - pliki wracają do obszaru roboczego w stanie **unmodified**, tracąc wprowadzone zmiany:
- **git checkout <ścieżka do pliku>**
- **git reset --hard**

Zapisanie stanu plików znajdujących się w przechowalni do repozytorium (pliki otrzymują status **unmodified**):
- **git commit –m „Krótki opis zmian”**

Zadanie: status pliku

1. W repo1 powinien istnieć nieśledzony plik **test3.txt** (jeśli go nie ma, to go stwórz) – zweryfikuj jego stan poleceniem:
   git status
2. Dodaj do śledzenia plik **test3.txt** a następnie sprawdź jego stan
3. Do pliku **test3.txt** dopisz: ”**blablabla**”, np. poleceniem:
   echo blablabla >> test3.txt
   Sprawdź stan pliku
4. Do pliku **test2.txt** dopisz „**Ala ma kota**”. Sprawdź stan
5. Przywróć plik **test2.txt** do stanu sprzed zmian. Sprawdź stan
6. Zapisz zmiany używając komentarza „**Test git status**”. Sprawdź stan

Sprawdzanie różnic i zatwierdzanie zmian

Sprawdzenie różnic w formie łatki, pomiędzy obszarem roboczym, a przechowalnią:

- **git diff**

```
diff --git a/test.txt b/test.txt
index 8432188..b3ae473 100644
--- a/test.txt
+++ b/test.txt
@@ -1 +1 @@
-Ala ma kota
+Ala ma psa
```

Sprawdzenie różnic gotowych do zatwierdzenia (staged):

- **git diff --cached**

Zatwierdzenie zmian:

- **git commit -m <komentarz do wersji>**

Zatwierdzenie zmian z dodaniem pliku do poczekalni:

- **git commit --all -m <komentarz do wersji>**

Poprawianie ostatnich zatwierdzonych zmian (**Ctrl+x** zapisuje zmiany i zamyka tryb interaktywny)

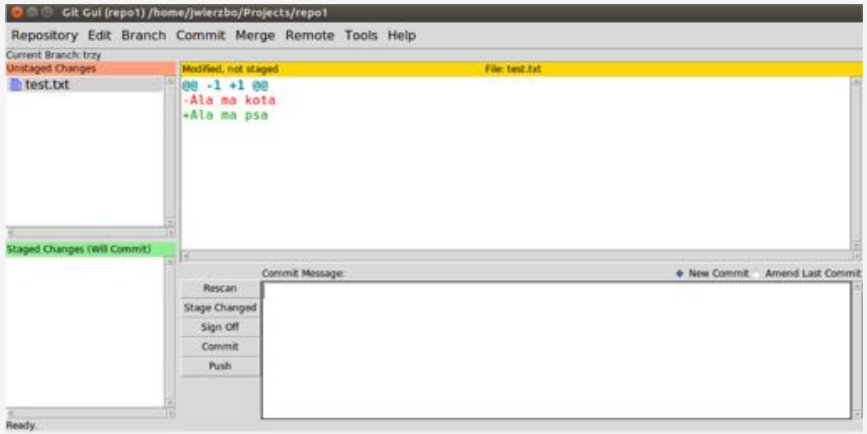
- **git commit --amend**

Zadanie: git diff

1. Do zawartości pliku **test3.txt** dodaj: „**terefere**”
2. Zweryfikuj wprowadzone zmiany w obszarze roboczym: **git diff**
3. Dodaj zmiany w pliku **test3.txt** do śledzenia
4. Porównaj wynik poleceń **git diff** (obszar roboczy) oraz **git diff --cached** (kolejka)
5. Zapisz wprowadzone zmiany dołączając je do ostatniego wykonanego commitu („**Test git status**”)

GIT GUI

- Narzędzie z okienkowym interfejsem użytkownika
- Pozwala na zarządzanie przechowalnią i commit-owanie zmian
- Podstawowe funkcjonalności:
  - Dodanie plików do przechowalni: **Commit -> Stage To Commit**
  - Commit-owanie zmian z przechowalni: **Commit -> Commit**
    - *Uwaga: opis commit-u wprowadzamy w oknie: **Commit message***
  - Wycofanie pliku z przechowalni (reset): **Commit -> Unstage From Commit**
  - Wycofanie wprowadzonych zmian na pliku (reset --hard): **Commit -> Revert Changes**
    - *Uwaga: plik nie może być w przechowalni - musi zostać najpierw z niej wycofany!*



Zadanie: status pliku – GIT GUI

1. Znajdując się w **repo1** otwórz **git gui**. Kolejne z poleceń uruchom w osobnym terminalu, a stan plików obserwuj poprzez odświeżanie widoku w **git gui** przyciskiem „**Rescan**”
2. W **repo1** utwórz plik **test4.txt** – zweryfikuj jego stan w git gui
3. Dodaj do śledzenia plik **test3.txt** a następnie sprawdź jego stan
4. Do pliku **test4.txt** dopisz: ”**Zaraz przerwa**”. Sprawdź stan pliku
5. Do pliku **test2.txt** dopisz „**Ala ma kota**”. Sprawdź stan
6. Przywróć plik **test2.txt** do stanu sprzed zmian. Sprawdź stan
7. Zapisz zmiany używając komentarza „**Test git gui**”. Sprawdź stan

Praca z plikiem .gitignore

- Plik zawierający wzorce, pozwalające ignorować konkretne pliki i katalogi
- Ignorowane pliki nie są widziane przez git-a
- *.gitignore* może znajdować się w każdym folderze repozytorium, co określa zasięg jego działania:
  - plik z katalogu głównego odnosi się do całej zawartości repozytorium
  - instrukcje z katalogów wewnętrznych odnoszą się tylko do ograniczonego przezeń obszaru

```
1  .idea
2  target
3  *.iml
```

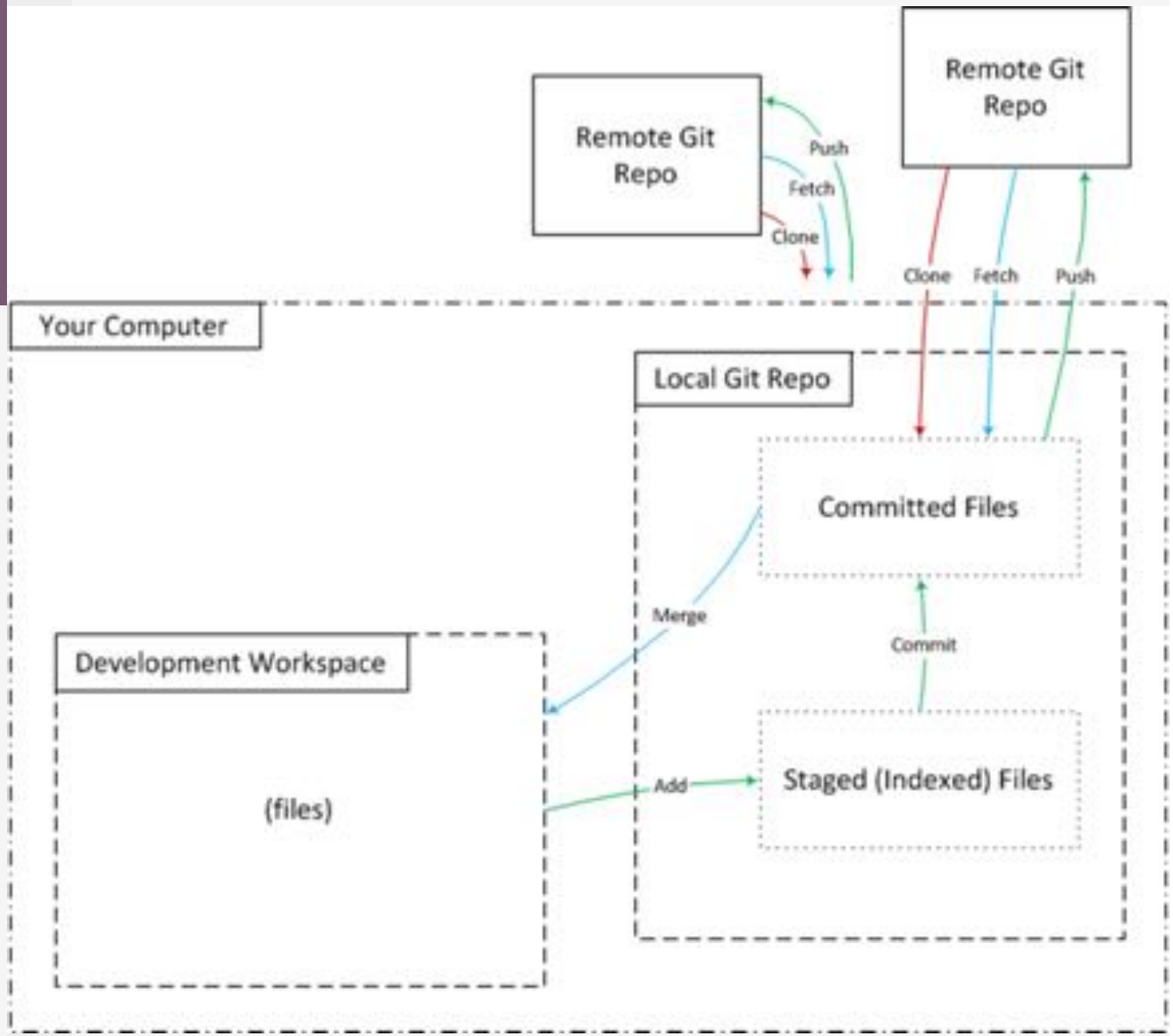
Przykładowa zawartość pliku .gitignore

Zadanie: .gitingore

1. W katalogu głównym repozytorium **repo1** stwórz plik **.gitignore** z zawartością:
   .idea
   **.bak*
2. Utwórz plik **test.bak**
3. Sprawdź stan repozytorium
4. Zapisz wprowadzone zmiany dołączając komentarz „**gitignore**”

Podsumowanie

Rysunek obok obrazuje **połączenia** pomiędzy, dotychczas omówionymi **elementami**, składającymi się na system **Git**



Zadanie: Podsumowanie bloku I

1. Stwórz katalog o nazwie **repo2** w swoim katalogu domowym i utwórz w nim repozytorium gita
2. Stwórz plik **readme.txt** z zawartością „**koniec bloku 1**”, następnie utwórz commit o nazwie **init**
3. Na swoim koncie w serwisie https://github.com utwórz nowe publiczne repozytorium o nazwie **repo2**
4. Skonfiguruj swoje repozytorium lokalne (**repo2**) tak, aby było podłączone do repozytorium zdalnego, utworzonego w poprzednim kroku
5. Wyślij zmiany na serwer
6. W katalogu roboczym utwórz folder **biblioteka**, a w nim plik **filmy.txt** oraz **ksiazki.txt**, umieszczając w nich po dwa dowolne tytuły. Zmiany dodaj do commitu o nazwie **biblioteka**
7. Wyślij zmiany na serwer

Zadanie: git add / commit / push

1. Przejdź do swojego katalogu domowego
2. Stwórz katalog o nazwie **repo3** i przejdź do niego
3. Zainicjalizuj nowe repozytorium gita
4. Stwórz plik **start.txt** z zawartością: **zaczynamy** oraz plik **.gitignore** z zawartością: **.idea**
5. Utwórz commit o nazwie **init**
6. Na swoim koncie w serwisie https://github.com utwórz nowe publiczne repozytorium o nazwie **repo3**
7. Skonfiguruj swoje repozytorium lokalne (**repo3**) tak, aby było podłączone do repozytorium zdalnego, utworzonego w poprzednim kroku (patrz komenda **git remote**)
8. Wyślij zmiany na serwer

Zadanie: git pull

1. Na https://github.com wejdź do **repo3** i kliknij na plik **start.txt**
2. Przejdź do edycji pliku: w prawym górnym rogu kliknij na ikonę ołówka
3. Dopisz w linii nr 2: „**zdalne zmiany**”, a następnie utwórz commit o nazwie: „**Zdalny commit**”
4. Wróć do swojego lokalnego repozytorium **repo3** i pobierz zmiany z serwera zdalnego
5. Sprawdź zawartość pliku **start.txt**
6. Narzędziem **gitk** przenalizuj zmiany

Wycofywanie zmian

Jeżeli chcemy wycofać zmiany z commitu:

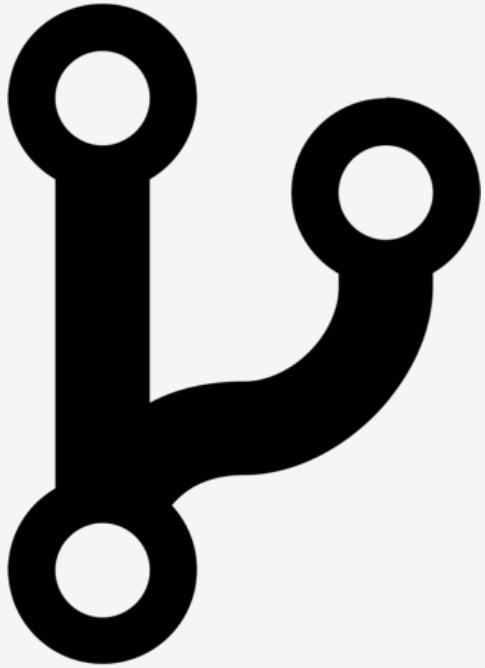
- **git revert <commit>**
  Tworzy nowy commit, który wycofa nasze zmiany, które jednak ciągle zostaną w historii
- **git reset <commit>**
  Przesuwa czubek gałęzi na podany commit
  Zmiany zostaną w folderze roboczym
- **git reset --hard <commit>**
  To samo co reset, ale czyści również folder roboczy.
  Usuwa zarówno commit jak i nasze zmiany!

Zadanie: cofanie zmian

1. Poleceniem **git log** lub narzędziem **gitk** odczytaj sumę kontrolną przypisanę do pierwszego commita („init”). Skopiuj jej wartość.
2. Poleceniem **git reset--hard suma_kontrolna_commita_init** cofinij się do poprzedniej wersji kodu
3. Zweryfikuj zawartość pliku **start.txt** oraz historię commitów poleceniem **gitk**
4. Powróć do najświeższej wersji poleceniem **git merge** lub **git pull**
5. Czym różnią się oba polecenia? Dlaczego ich wynik jest taki sam?
6. Spróbuj wykonać polecenie: **git push origin master**

Git – branche (gałęzie)

- Branch (gałąź) służy do bezpiecznego wdrażania nowej funkcjonalności
- Tworzona jest specjalna lokalna migawka kodu, więc programista jest oddzielony od działającej aplikacji
- Po wdrożeniu i przetestowaniu funkcjonalności branch jest włączany do stabilnej wersji aplikacji (tzw. trunk'a)
- Do wizualnego przeglądania zawartości branchu można skorzystać z narzędzia gitk: **gitk --branches=***

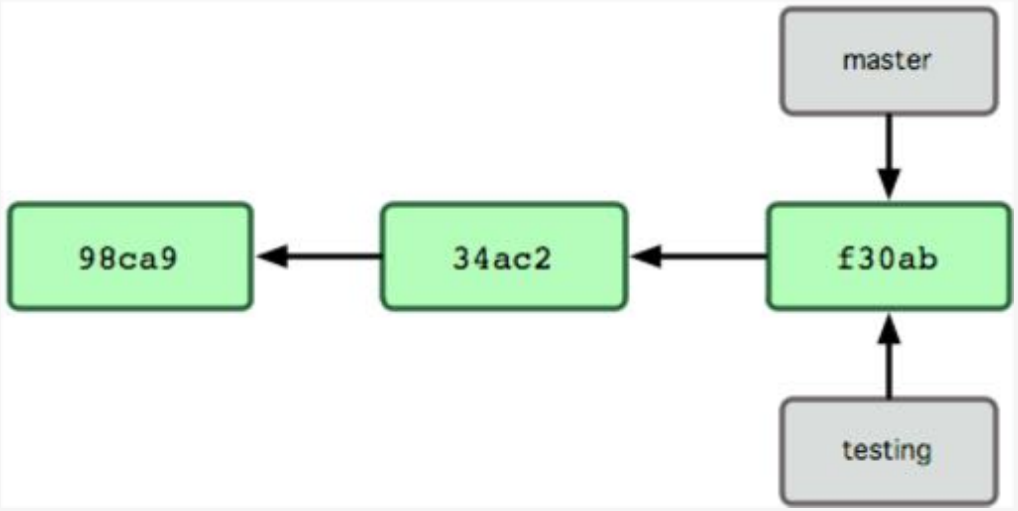


Czym jest branch?

Tworzenie **brancha** w Git można przyrównać do „**Zapisz jako…**” dla folderów – w tym wypadku naszego **Katalogu Roboczego**

Tworzenie branchy

Tworzenie nowego brancha o nazwie testing:
`git branch testing`
Tworzenie brancha wraz z automatycznym przełączaniem na niego:
`git checkout -b testing`

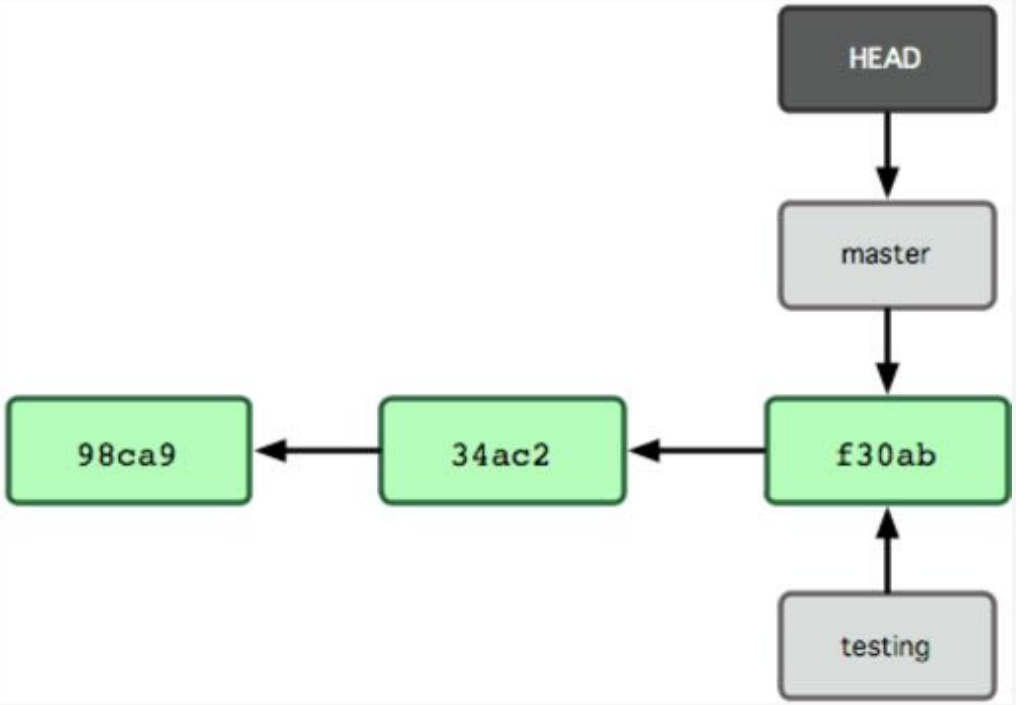


Przełączanie branchy

Git wie na której gałęzi aktualnie się znajdujemy przy pomocy specjalnego wskaźnika **HEAD**.

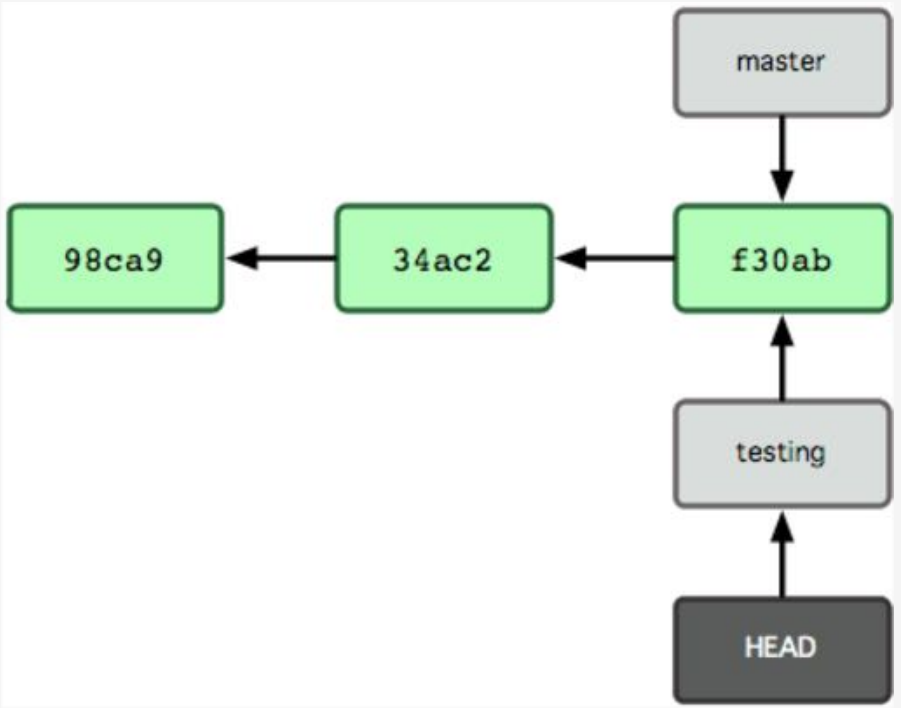
Jak widać na rysunku Git pomimo utworzenia brancha nie przełączył nas na niego. Aby to zrobić musimy skorzystać z komendy

```
git checkout testing
```



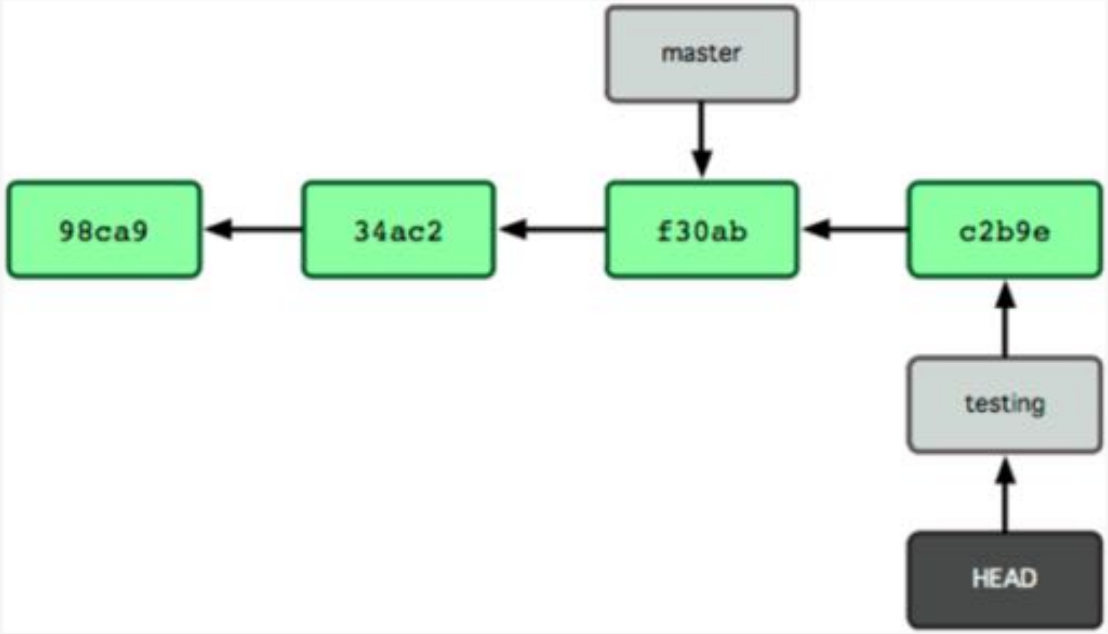
HEAD

HEAD zostaje zmieniony tak, by wskazywać na gałąź **testing**.



Dodawanie commitów do branchy

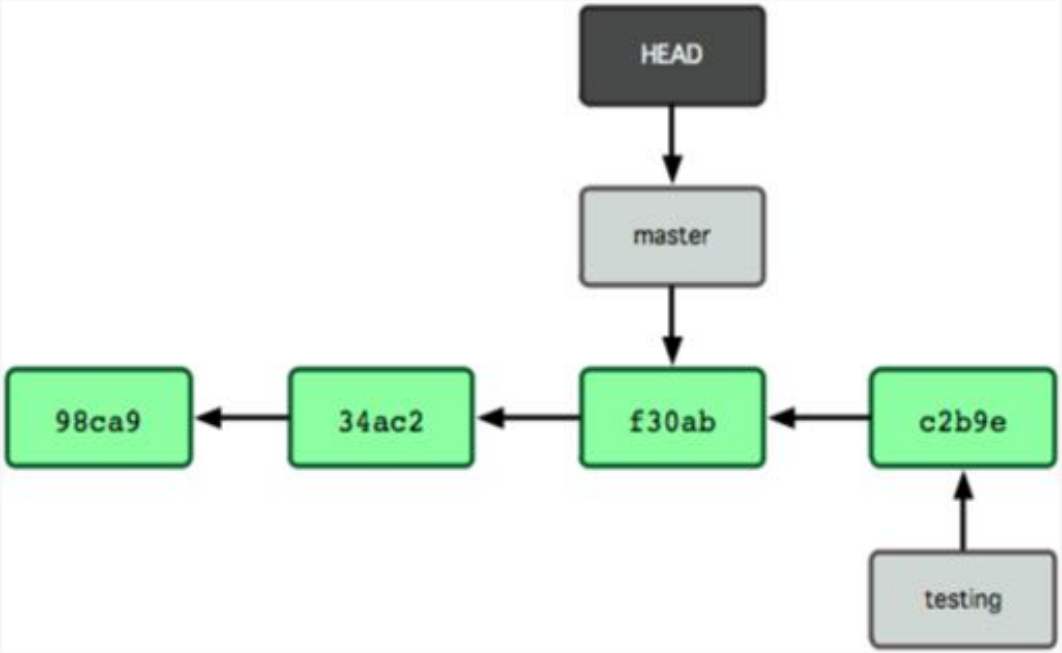
Utwórzmy nowy plik i wypełnijmy go tekstem, po czym dodajmy go do repozytorium i wcommitujmy zmiany.



Branche w praktyce

Teraz gałąź **testing** przesunęła się do przodu, jednak gałąź master ciągle wskazuje ten sam zestaw zmian, co w momencie użycia **git checkout** do zmiany aktywnej gałęzi. Przełączmy się zatem z powrotem na gałąź master:

git checkout master



Branche w praktyce 2

Polecenie dokonało dwóch rzeczy.

1. Przesunęło wskaźnik HEAD z powrotem na gałąź master
2. Przywróciło pliki w katalogu roboczym do stanu z migawki, na którą wskazuje master.

Oznacza to również, że zmiany, które od tej pory będą wprowadzane, spowodują rozwidlenie od starszej wersji projektu. W gruncie rzeczy cofa to tymczasowo pracę, jaką wykonałeś na gałęzi testing, byś mógł z dalszymi zmianami pójść w innym kierunku.

Zadanie: git branch

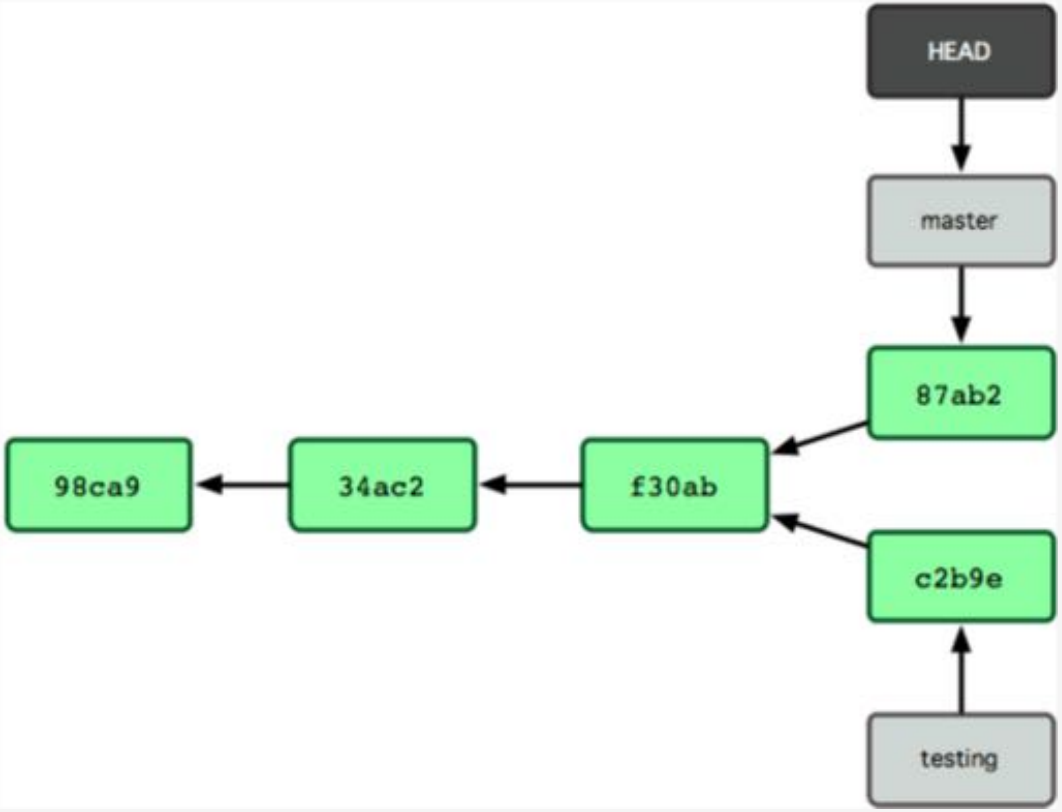
1. Na swoim koncie https://github.com utwórz nowe publiczne repozytorium o nazwie **repo4** i skopiuj do niego link
2. Przejdź do swojego katalogu domowego
3. Sklonuj utworzone **repo4** za pomocą skopiowanego linku i wejdź do niego (katalog **repo4**)
4. Stwórz plik **branch.txt** z zawartością: „**Na branchu MASTER”** oraz plik **.gitignore** z zawartością: **.idea**
5. Utwórz commit o nazwie **init**
6. Wyślij zmiany na serwer
7. Utwórz branch **test** i przejdź na niego.
8. Stwórz plik **nowy_branch.txt** z zawartością: „**Na branchu TEST”**. **Scommituj** zmiany z opisem „**nowy branch**”
9. Wyślij zmiany na serwer -> UWAGA wysyłamy branch **test**

Scalanie branchy

Stworzyliśmy i przełączyliśmy się na gałąź, wdrożyliśmy nową funkcjonalność, a następnie wróciliśmy na gałąź główną i wykonaliśmy inną pracę. Oba zestawy zmian są od siebie odizolowane w odrębnych gałęziach: możemy przełączać się pomiędzy nimi oraz scalić je razem komendą

`git merge testing`

kiedy uznamy pracę za zakończoną.



Zadanie: git merge

1. W repo1 przejdź do brancha master
2. Porównaj polecenie **gitk** oraz **gitk --branches=***
3. Zmerguj branch test
4. Wyślij zmiany na serwer
5. Sprawdź wynik poleceniem **gitk**

Listowanie branchy

Wyświetlanie wszystkich lokalnych branchy:

- **`git branch`**

Wyświetlanie wszystkich lokalnych branchy, które są zmergowane:

- **`git branch --merged`**

Wyświetlanie wszystkich lokalnych branchy, które nie są zmergowane:

- **`git branch --no-merged`**

Usuwanie branchy

Usuwanie brancha, który został zmergowany:

- **`git branch -d <nazwa_brancha>`**

Wymuszone usuwanie, bez względu na to czy branch został zmergowany:

- **`git branch -D <nazwa_brancha>`**

Zadanie: git branch

1. W **repo3** Tworzysz **branch „obwod”**, dla nowej funkcji, nad którą pracujesz. Przechodzisz na niego
2. Tworzysz tam **plik wzor.txt**

Na tym etapie otrzymujesz telefon, że inny problem jest obecnie priorytetem i potrzeba błyskawicznej poprawki. Oto, co robisz:

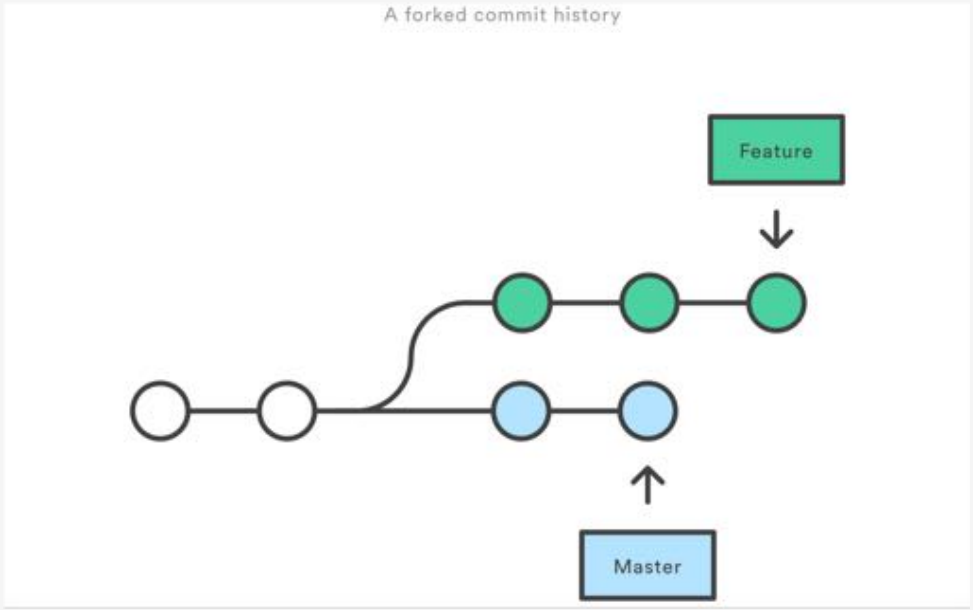
1. Commitujesz aktualna pracę pod nazwą „**tajny wzor**” i wracasz na branch produkcyjny - **master.**
2. Tworzysz nowy branch **hot-fix**, przechodzisz na niego i tworzysz tam plik **poprawka.txt**
3. Wracasz na branch **master** i scalasz **(merge)** branch z poprawką **(hot-fix)**
4. Wysyłasz zmiany na serwer zdalny
5. Wracasz z powrotem do brancha **obwod** (i kontynuujesz prace...)
6. Usuwasz branch **hot-fix**

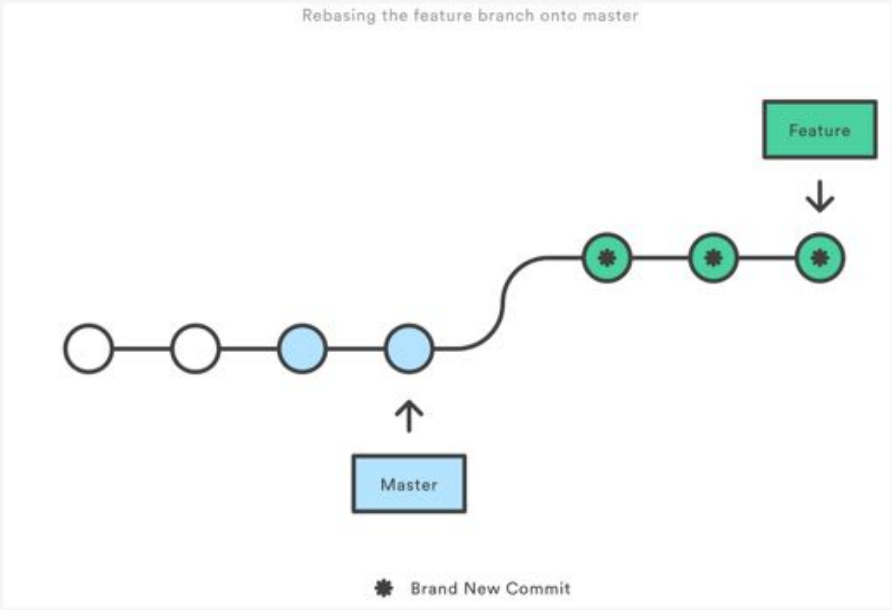
Sprawdź układ branchy: **gitk --branches=***

Rebase

Zmiana przodka naszego brancha:

- git checkout <branch_docelowy>
- git rebase <nowy_przodek>





Zadanie: praktyka

Wejdź na stronę https://learngitbranching.js.org/ i wykonaj zadania 1-4 z sekcji „ **Introduction Sequence”**

Konflikty

Zmiany wprowadzone na tym samym pliku w dwóch różnych branch-ach, które zamierzamy merg-ować doprowadzą do konfliktu,
np. zmiana „*Ala ma kota*” w *test.txt*:

```
<<<<<<< HEAD
Ala ma psa
=======
Ala ma rybki
>>>>>>> rybki_branch
```

- Ręczne rozwiazywanie:
  1. Otwórz plik w edytorze
  2. Sprawdź znaczniki konfliktu
  3. Wybierz popraw lub usuń zmiany
  4. Dodaj plik do śledzenia
  5. Commit
- Półautomatyczne rozwiazywanie konfliktów

  Idea IntelliJ – VCS -> Git -> Resolve Confilcts...

Zadanie: konflikty

1. W **repo3** utwórz nowy branch „**rybki**” i przejdź do niego
2. Zmodyfikuj plik **test.txt** zamieniając wyraz ”**kota**” na ”**rybki**”. Zapisz zmiany z opisem ”**Nowe zwierze – rybki**”
3. Wróć na branch **master**.
4. Zmodyfikuj plik **test.txt** zamieniając wyraz ”**kota**” na ”**psa**”. Zapisz zmiany z opisem ”**Nowe zwierze – pies**”
5. Wykonaj polecenie **git merge rybki**
6. Rozwiąż konflikty zachowując wersję z branch-a **rybki**. Zapisz zmiany z opisem „**merge z rybki**” [ten punkt możesz wykonać za pomocą Idea Intellij]

Zadanie: konflikty – Idea IntelliJ

1. Wykonaj poprzednie zadanie jeszcze raz, tym razem za pomocą **Idea Intellij**
2. Zamień zwierzątka:
   - **rybki -> lew**
   - **pies -> mis**
3. Rozwiąż konflikty zachowując wersję z branch-a **lew**. Zapisz zmiany z opisem „**merge z lew**”

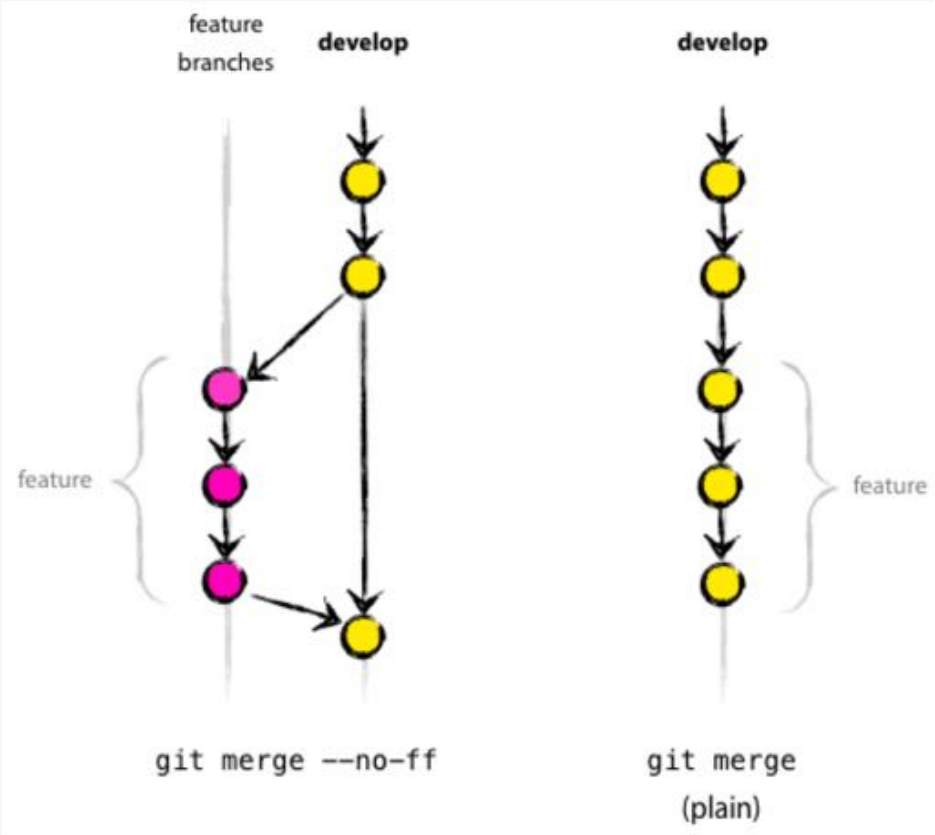
Fast-Forward Merging Strategy

Strategia Fast-Forward polega na merg-owaniu zmian po uprzednim wykonaniu rebase na branch-u do którego zamierzamy się merg-ować. Dzięki temu zachowujemy czystą i przejrzystą historię commit-ów.

Procedura mergowania w trybie Fast-Forward:

- git checkout feature_branch
- <wykonaj zmiany, zapisz je (commit)>
- git rebase master

  W przypadku konfliktu, rozwiąż go, a następnie:
  - git rebase --continue
- git checkout master
- git merge feature_branch



Pull Request & Fork

Praca na wspólnym repozytorium:

- Tworzenie pull requesta: https://help.github.com/articles/creating-a-pull-request/

Wprowadzanie zmian do “obcych” repozytoriów (np. Zgłaszanie poprawek):

- Tworzenie forka: https://help.github.com/articles/fork-a-repo/
- Tworzenie pull requesta z forka: https://help.github.com/articles/creating-a-pull-request-from-a-fork/

Pull Request

1. Utwórz nowy branch o nazwie **pr-test** w **repo3** i przejdź do niego
2. Stwórz plik **test.txt** z dowolną zawartością, następnie zamień ostatnią linie w **start.txt** na wartość „**edycja z PR**”
3. Stwórz commita o nazwie **PR** i wyślij zmiany na serwer
4. Na swoim koncie w serwisie https://github.com utwórz **pull request’a** z brancha **PR** do brancha **master**

Fork & PR

1. Stwórz **fork'a** z repozytorium https://github.com/jakubwierzbowski/REST-movie-service
2. **Sklonuj** utworzonego forka- https://github.com/TWOJE_KONTO/REST-movie-service
3. W pliku **README.md** dopisz swoje **imię i nazwisko** na samym końcu
4. Stwórz **commit** ze zmianami o nazwie **HOT_FIX** i **wyślij** do zdalnego repozytorium
5. Utwórz **pull request'a z forka do repozytorium bazowego** https://github.com/jakubwierzbowski/REST-movie-service

Stash – czyli schowek

Komenda stash przenosi aktualny stan folderu roboczego i umieszcza go na stosie zmian, które w dowolnym czasie możesz nałożyć ponownie.

- Gdy chcemy zmienić branch, a mamy zmiany
- Kiedy chcesz odłożyć zmiany i spróbować czegoś innego

git stash – wrzuć wszystkie niezapisane zmiany do schowka
git stash list – aby wyświetlić listę zmian
git stash apply <stash> - aby nałożyć zmiany z powrotem

git stash pop <stash> - aby nałożyć ostatnie zmiany z listy z powrotem

git stash drop <stash> - aby usunąć dany schowek

Dodatkowe komendy

- git cherry-pick – pobiera wybiórczą zmianę i merguje z aktualną gałęzią
- git revert – cofa jeden, bądź wiele ostatnich commitów i tworzy nowy commit
- git squash – zmienia główną gałąź
- git tag -a <tag_name> – tagowanie wersji
- git stash - wrzucanie zmian do schowka

Zadanie: konfiguracja logowania po SSH

1. Skonfiguruj autoryzacje z kontem Git po protokole SSH wg poradnika:
   - Windows: https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/#platform-windows
   - Mac: https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/#platform-mac
   - Linux: https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/#platform-linux
2. Zweryfikuj poprawność pobierając klonując repo przy użyciu protokołu git:
   git clone git@github.com:jakubwierzbowski/REST-movie-service.git

Kurs online

Pod adresem: http://gitreal.codeschool.com znajdziemy rozbudowany kurs online w formie video/prezentacji wraz z interaktywnymi zadaniami

Praca z repozytorium

git --help – wyświetla pomoc narzędzia git
git init - przygotowanie katalogu, aby stał się repozytorium
git remote add origin ... - dodanie repozytorium zdalnego
git status - sprawdzenie stanu repozytorium np. czy posiadamy różnice lokalnie
git commit -m "komentarz" - włączenie zmian do repozytorium (lokalnie)
git push origin master - przesłanie zmian na serwer
git pull origin master - pobranie najnowszych zmian na serwer
git clone – sklonowanie repozytorium z serwera
git checkout <branch> - przełączenie się do brancha <nazwa_brancha>
git branch <branch> - utworzenie brancha < nazwa_brancha >
git merge <branch> - scalenie brancha < nazwa_brancha > do aktualnegeo brancha

Do poczytania

https://git-scm.com/book/pl/v1

Na koniec:

Ankiety ☺