



Git & GitHub

Podręcznik kursowy

Mobilo © 2025



W tym miejscu zwykle pojawia się informacja o tym kto i jak może posługiwać się tym podręcznikiem. Bez regułek prawnych odwołam się po prostu do kilku prostych zasad, które oddają sens kto, kiedy i jak może wg zamysłu autora korzystać z tego materiału:

- Ten podręcznik jest integralnym elementem kursu online.
- Możesz z niego korzystać będąc uczestnikiem tego kursu. Podręcznik jest dla Ciebie i korzystaj z niego do woli – drukuj, wypełniaj, uzupełniaj, przeglądaj, póki Twoim celem jest samodzielne opanowanie tematu.
- Proszę nie umieszczaj go w publicznie dostępnych miejscach, jak blogi, repozytoria git hub, chomik itp.
- Nie wykorzystuj go w innych celach, np. organizacji własnych szkoleń, gdzie występujesz np. jako instruktor.
- Jeśli nie posłuchasz moich prośb, to jako autor zapiszę się do ZAIKS-u i następnym razem kupując smartfona zapłacisz za niego kilkaset złotych więcej 😊, więc może lepiej po prostu przestrzegaj praw autorskich

Z góry dziękuję!

Rafał Mobilo © 2025

Zapraszam do odwiedzenia strony:

<http://www.kursyonline24.eu/>

Review 2025-01-02

Spis treści

Jak się uczyć?	4
O kursie	5
Wprowadzenie do Git	6
Git init & config	6
Instalacja Git-a	7
70% git-a: add, commit, status	7
Znajdowanie różnic między commitami	8
Dodatkowe opcje podczas przeglądania historii	9
Checkout – podróże w czasie	9
Log – wyszukiwanie commitów	10
Revert - czyli commit odwrotny	11
Restore - odtwarzanie pliku z wcześniejszego commit-a	12
Reset - powrót do wcześniejszego commita.....	13
Grzebanie w historii: checkout, commit amend, rebase	14
Instalacja Visual Studio Code	14
Markdown.....	15
Przesiadka z linii komend do interfejsu graficznego	15
Branch - równoległa wersja kodu	16
Git log i Git Graph - rozkład jazdy branchy	16
Merge - scalanie branchy (bez konfliktów).....	17
Merge - scalanie branchy (z konfliktami).....	17
Merge - scalanie zmian (Merge Editor VSC)	18
Zarządzanie branchami	18
Tagowanie.....	19
Cherry-Pick - smakowity commit	19
Rebase i jego zalety.....	20
Rebase - interactive	21
Stash - ach, zrobię to później	22
GitHub - domek dla programisty.....	22
Pierwsze repozytorium na GitHub	23
Clone, push i uwierzytelnienie przez PAT	23
Konfiguracja VSC do pracy z GitHub	24
Wysłanie lokalnego repo do GitHub	24
Tworzenie i usuwanie tagów i branchy na GitHub	25
Zabezpieczanie repozytorium na GitHub.....	26
Pull Request	27
Pull request - przesyłanie zmian między branchami.....	27
Praca grupowa i security w GitHub.....	28
Issues & Projects	28
Integracja pracy z issues i projects.....	29
Fork i Pull Request	29
Spróbuj też!.....	30

Jak się uczyć?

Pracujesz z kodem, chcesz lub musisz przechowywać ten kod na GitHub, ale... nie zawsze wszystko do końca idzie po Twojej myśli? Ten kurs powinien pomóc w rozwiązyaniu tego problemu.

Naukę oczywiście zorganizujesz sobie po swojemu, ale pozwól, że zaproponuję kilka sposobów nauki, a Ty sam/a wybierzesz, co z tego Ci się podoba, a co wolisz zrobić po swojemu

1. **Co za dużo to niezdrowo** – nie rób na raz za dużo materiału. Nie od razu Kraków zbudowano. Jedna lub dwie lekcje na dzień powinny wystarczyć.
2. **Licz się regularność** – niekoniecznie uczyć trzeba się codziennie, ale jeśli postanowisz przerabiać lekcje we wtorki, czwartki i soboty to już coś!
3. **Ten kurs nie ma spisanych „zadań praktycznych”, ale...** śmiałe podążaj za tym, co prezentuję w materiałach video i twórz własne repozytoria, wysyłaj lub wycofuj commity tak samo, jak robię to w lekcjach video.
4. Uczę się uczę, a do głowy nie wchodzi – wszyscy tak mamy i to pewnie dlatego nauka w szkole trwa aż tyle lat! **Od czasu do czasu zrób sobie powtórkę**. Przecież nikt Cię nie goni i nie rozlicza z postępów.
5. Notatki w podręczniku są dla Twojej wygody. Niestety wygoda leży blisko lenistwa. Nie bądź leniem. Przygotuj sobie zeszyt lub kilka luźnych kartek i **zapisuj to czego się uczysz**. To co wejdzie oczami lub uszami, będzie wychodzić rękami i... nie ma wyjścia – po drodze zahaczy o mózg 😊
6. Jeśli możesz – wydrukuj sobie podręcznik, dopisz do niego własne notatki, uwagi itp.
7. Kiedy osiągniesz jakiś „kamień milowy”, ukończysz sekcję kursu, a może nawet cały kurs – **daj sobie nagrodę** – to niesamowicie zwiększa motywację!
8. **Nie bój się korzystać z innych materiałów**: książek, blogów, forów itp. Część z nich na początku może być nieco za trudna, ale nic nie stoi na przeszkodzie, żeby na początku nic nie mówić, tylko słuchać.
9. Kiedy już ukończysz kurs – **zaktualizuj CV na LinkedIn**, pochwal się swoim certyfikatem, daj się odnaleźć rekruterom, zaproś nas do znajomych (link w profilu). Chętnie potwierdzę Twoją nową umiejętność!

Powodzenia!

Twój trener Rafał

O kursie

Kto pracuje z kodem? Programiści? Analitycy? Skrypterzy? Tak. Ale tak się nam świat poukładał, że aktualnie kodu używa każdy, kto pracuje w IT, bo mamy Infrastructure as Code, Security as Code, Configuration as Code - w skrócie Everything as Code! A skoro tak, to praktycznie każdy w IT musi ten kod utrzymywać w repozytorium, aktualizować go, rozumieć proces produkcji oprogramowania, współpracować z innymi developerami - i tu rodzi się potrzeba znajomości git i GitHub.

Nieważne, czy jesteś programistą i być może pracujesz ciągle jeszcze nad małym projektem, gdzie wydaje się, że git nie jest potrzebny, czy jesteś administratorem, a może architektem, czy project managerem - przedzej czy później okaże się, że „na już” musisz skorzystać z repozytorium na GitHub. Znajomość git to w dzisiejszych czasach po prostu konieczność. Dlatego właśnie oddaje w Twoje ręce ten kurs, który opowiada o git i GitHub w zakresie pracy z kodem.

Tak właściwie, to nie ma specjalnie wygórowanych wymogów, żeby uczestniczyć w tym kursie. Nie musisz znać żadnego języka programowania. Wszystkie przykłady są oparte o pliki tekstowe lub Markdown, którego też przy okazji się nauczysz. Wystarczą więc chęci, komputer i dostęp do Internetu.

W pierwszej części kursu nauczysz się od podstaw, jak lokalnie na Twoim komputerze utworzyć repozytorium, dodawać zmiany, zatwierdzać je commitem, jak sprawdzać, co i kto ostatnio zmienił w kodzie, tworzyć równolegle wersje kodu w postaci branchy, jak je scalać polecienniem merge, jak odtwarzać wcześniejsze wersje plików, jak wyświetlać historie zmian. Dowiesz się, co to jest rebase, co to cherry-pick, checkout, czym różni się restore od reset czy revert. Każde z poleceń poznasz od podszezwki, uruchamiając je z linii komend, ale zobacysz też jak z nimi pracować w trybie graficznym w Visual Studio Code.

W drugiej części kursu przyjrzymy się GitHub jako centralnemu repozytorium, gdzie swoje zmiany zapisuje wielu programistów pracujących nad projektem. Dowiesz się, jak konfigurować dostęp do GitHub, jak korzystać z pull request, jak wykorzystać go w implementacji strategii branchingu. Zobaczysz też, jak unikać konfliktów bądź, jak je rozwiązywać, jeśli się już pojawią, jak dokumentować i śledzić pracę wykonywaną przez cały zespół oraz jak unikać zagrożeń związanych ze współdzieleniem kodu.

Kurs składa się z krótkich lekcji video, każda poświęcona pewnej funkcjonalności git lub GitHub. Masz też do dyspozycji podręcznik z krótką notatką z każdej lekcji.

Kończąc ten kurs, bez problemu poradzisz sobie z utrzymaniem kodu swoich programów w repozytoriach git i na GitHub. Nieważne, czy jesteś programistą, analitykiem skrypterem, czy adminem - drzwi do utrzymania kodu w repozytoriach będą po tym kursie dla ciebie otwarte.

Wszystko jest kodem. Zapraszam na kurs „Git & GitHub”!

Twój trener, Rafał

Wprowadzenie do Git

- Najważniejsze cechy dobrego systemu zarządzania wersją:
 - Utrzymywanie historii zmian
 - Wspieranie współpracy wielu programistów
- Git w odróżnieniu od centralizowanych systemów kontroli wersji, pozwala na przechowywanie kompletnego repozytorium kodu na wielu komputerach (Distributed Version Control)
- Podczas zatwierdzania kodu (commit) Git zapamiętuje migawkę (snapshot) wszystkich plików projektu. Jeśli plik się nie zmienił, to w snapshocie znajdzie się tylko odnośnik do jego wcześniejszej wersji.

Git init & config

- Katalog zamienia się w repozytorium poprzez uruchomienie polecenia

```
git init
```

- Powstaje wtedy ukryty katalog .git, który przechowuje poprzednie wersje plików z kodem oraz plik konfiguracyjny
- Zazwyczaj pracę z repozytorium zaczynamy od konfiguracji repozytorium, dwa najważniejsze parametry to user.name i user.email

```
git config user.name boss  
git config user.email gitcourse@mobil024.eu
```

- Mamy konfigurację na poziomie:
 - Komputera – plik konfiguracyjny w katalogu, w którym jest zainstalowany git
 - Użytkownika – plik konfiguracyjny w katalogu domowym użytkownika
 - Repozytorium – plik konfiguracyjny w katalogu .git
- Jeśli jakiś parametr zostanie skonfigurowany na niższym poziomie to jest ważniejszy niż ten sam parametr zdefiniowany na wyższym poziomie
- Jeśli chcesz zmienić konfiguracje na poziomie użytkownika, to do polecień git config dodaj opcję -global

```
git config --global user.name boss  
git config --global user.email gitcourse@mobil024.eu
```

Instalacja Git-a

- Git jest dostępny na MacOS, Linux i Windows
- Żeby sprawdzić wersję Git-a użyj:

```
git --version
```

70% git-a: add, commit, status

- Wyświetla informacje o stanie repozytorium, w tym o stanie zmian w plikach

```
git status
```

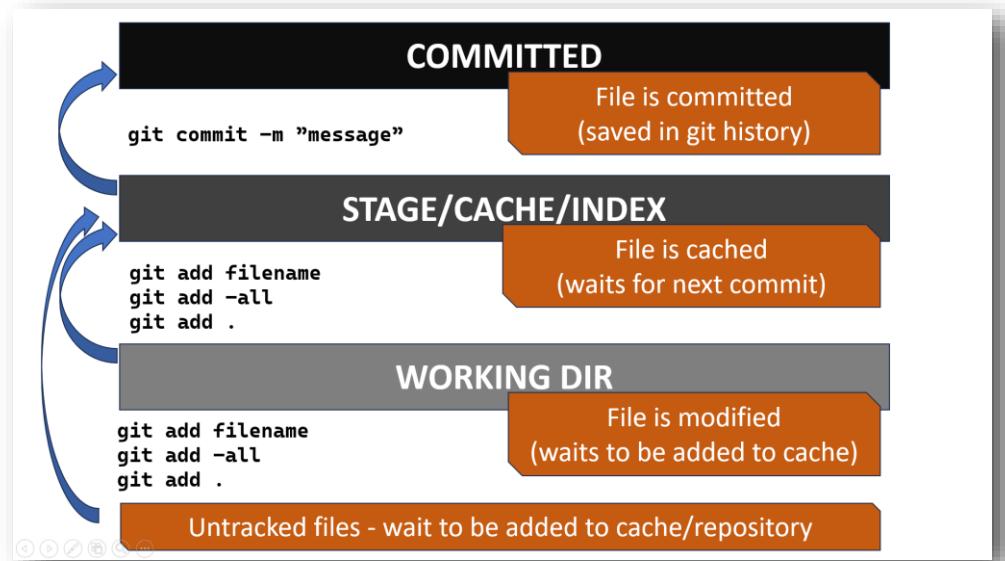
- Dodaje plik do cache/stage/index (te 3 nazwy są równoważne). Pliki znajdujące się w cache są przygotowanie do wysłania w kolejnym commit.

```
git add file_name
```

- Zatwierdza (committuje) pliki/zmiany w plikach, zapisuje je do historii git-a

```
Git commit -m „message”
```

- Pliki i/lub zmiany w plikach mogą mieć różne statusy:
 - **Untracked** – plik znajduje się poza repozytorium git i nie jest śledzony
 - **New file/Modified** – plik został zmodyfikowany w katalogu roboczym (working dir)
 - **Cached/Staged/Index** – plik znajduje się w cache i zostanie zatwierdzony przy okazji najbliższego commit
 - **Committed** – plik jest zatwierdzony. Plik może znajdować się w kilku stanach jednocześnie, np.:
 - Istnieje wersja pliku, która została ostatnio commitowana (zatwierdzona)
 - Plik po modyfikacji został dodany komendą `git add` do cache
 - Plik został jeszcze raz zmodyfikowany w working dir



Znajdowanie różnic między commitami

- Wyświetlenie listy commitów

```
git log
```

- Wyświetlenie listy commitów w taki sposób, że każdy commit mieści się w jednej linijce

```
git log --oneline
```

- Wyświetlenie różnic między dwoma wybranymi commitami

```
git diff commit_id_1 commit_id_2
```

- Wyświetlenie różnic między working dir, a cache/stage

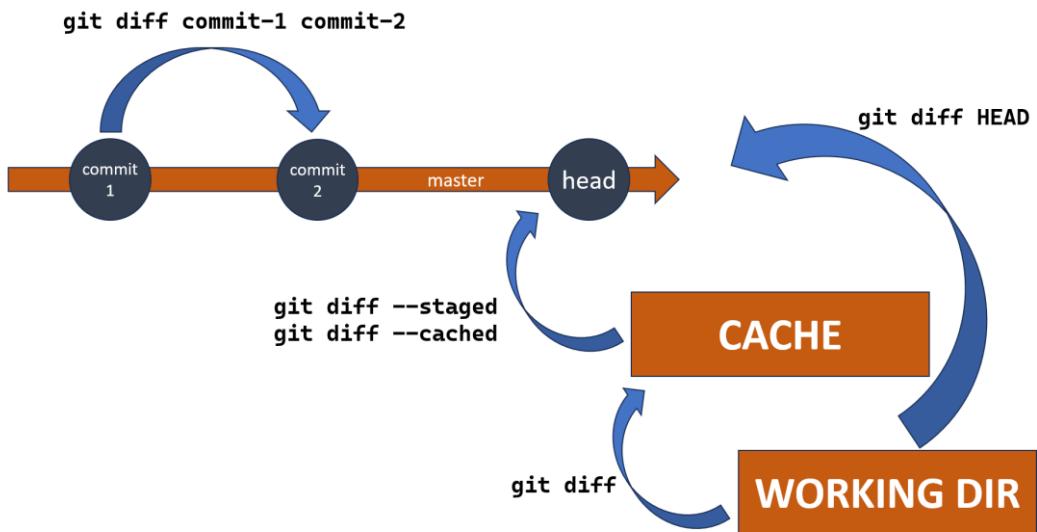
```
git diff
```

- Wyświetlenie różnic między cache/stage, a ostatnim commitem

```
git diff --cached
```

- Wyświetlenie różnic między working dir, a ostatnim commitem

```
git diff HEAD
```



Dodatkowe opcje podczas przeglądania historii

- Wyświetlenie statystyki zmian w poszczególnych plikach

```
git diff commit_id_1 commit_id_2 --stat
```

- Wyświetlenie tylko jednej linijki ze statystykami zmian

```
git diff commit_id_1 commit_id_2 --shortstat
```

- Wyświetlenie szczegółów na temat zmienianych linii

```
git diff commit_id_1 commit_id_2 --patch
```

- Wyświetlanie bez kolorów

```
git diff commit_id_1 commit_id_2 --no-color
```

- Wyświetlenie zmian tylko z jednego pliku, nawet jeśli dany commit modyfikował kilka plików

```
git diff commit_id_1 commit_id_2 -- file1
```

- Wyświetlenie zmian między „dziadkiem”, a ostatnim commitem

```
git diff HEAD~2 HEAD
```

Checkout – podróże w czasie

- Gdy chcesz usunąć plik z repozytorium to możesz to zrobić poleceniem systemu operacyjnego, po czym taką zmianę należy standardowo dodać do cache i commitować:

```
rm file.txt  
git add file.txt  
git commit -m "Removing file.txt"
```

- Alternatywnie można w tym celu posłużyć się poleceniem **git rm**. To polecenie od razu odkłada zmianę do cache, nie trzeba więc osobno uruchamiać **git add**

```
git rm file.txt  
git commit -m "Removing file.txt"
```

- Aby cofnąć się w czasie do wcześniejszego commit uruchom

```
git checkout commit_id
```

- W takim przypadku repozytorium znajduje się w stanie „detached head”, który polega na tym, że stan plików repozytorium NIE odpowiada ostatniemu commitowi w danym branchu. Jeśli uruchomisz

```
git log --oneline --all
```

to zobaczysz, że znacznik HEAD jest w innym miejscu niż znacznik master

- Aby wrócić do ostatniego commit w branchu użyj jednego z polecień:

```
git switch -  
git switch master
```

Log – wyszukiwanie commitów

- Do wyświetlania informacji o wykonanych commitach można użyć polecenia git log. To polecenie akceptuje wiele dodatkowych opcji:
 - --oneline - informacje umieszcza w jednym wierszu
 - --all - dodaje „przyszłe commity” (z punktu widzenia HEAD) oraz commity z innych branchy
 - -3 - pokaz tylko 3 ostatnie commity
 - --patch - dodaje informacje o zmianach w plikach (jakby połączenie polecenia log i diff)
 - --name-only - pokaż tylko nazwy plików, jakie były zmodyfikowane w danych commitach
- Do formatowania wyników można użyć parametru --pretty i dalej określić napis formatujący:

```
git log --pretty=format:"%h - %an, %ar: %s"
```

Lista dostępnych placeholderów znajduje się w dokumentacji: <https://git-scm.com/docs/pretty-formats>

- Do filtrowania można użyć parametrów:
 - --since =1.hour - commity z ostatniej godziny
 - --after="yesterday" - commity od wczoraj
 - --before="20271231" - commity przed 31 grudnia 2027
 - -- author="boss\|boss1" - commity, których autorem jest użytkownik boss lub boss1
 - -- filename - nazwa pliku dodana po podwójnym znaku - pozwala ograniczyć wyniki tylko do tych, które modyfikowały określony plik
 - --grep "diesel" - odfiltrowanie po nazwie commita zawierającego słowo diesel. Dodanie opcji -i wskazuje, że dopasowanie ma być wykonane bez patrzenia na wielkie czy małe litery
 - -S text - znajdź commity, które zawierają modyfikacje, które w modyfikowanych plikach zawierają wskazany text
- Opcje można ze sobą łączyć. Możliwość przeszukiwania commitów jest również zwykle dobrze implementowana w narzędziach graficznych.

Revert - czyli commit odwrotny

- Gdy w jednym kroku chcesz dodać wszystkie zmiany do cache i wykonać commit, to możesz użyć opcji -a dla polecenia commit:

```
git commit -a -m "message"
```

- Git revert pozawala na stworzenie commita przeciwnego do danego commita. Celem tego nowego commit jest wykonanie odwrotnych zmian do wskazanego commit. W efekcie żaden commit nie jest usuwany, tylko powstaje nowy commit:

```
git revert commit_id
```

- Domyślnie dla każdego wycofywanego commita powstaje jeden commit przeciwny. Jeśli chcesz wycofać więcej zmian tworząc tylko jeden commit wycofujący, to dodaj opcję --no-commit. W takim przypadku zmiany zostają wycofane, ale nie wykona się commit. Zmiany będą odłożone w cache i to programista jest odpowiedzialny za zatwierdzenie tych zmian:

```
git revert commit_id --no-commit
```

- Wycofując commity można korzystać z aliasów HEAD~1, HEAD~2 itp.

Restore - odtwarzanie pliku z wcześniejszego commit-a

- Git restore odtwarza plik. Odtworzony plik znajdzie się w working directory. Programista dalej podejmuje decyzję, czy ten plik należy commitować czy nie

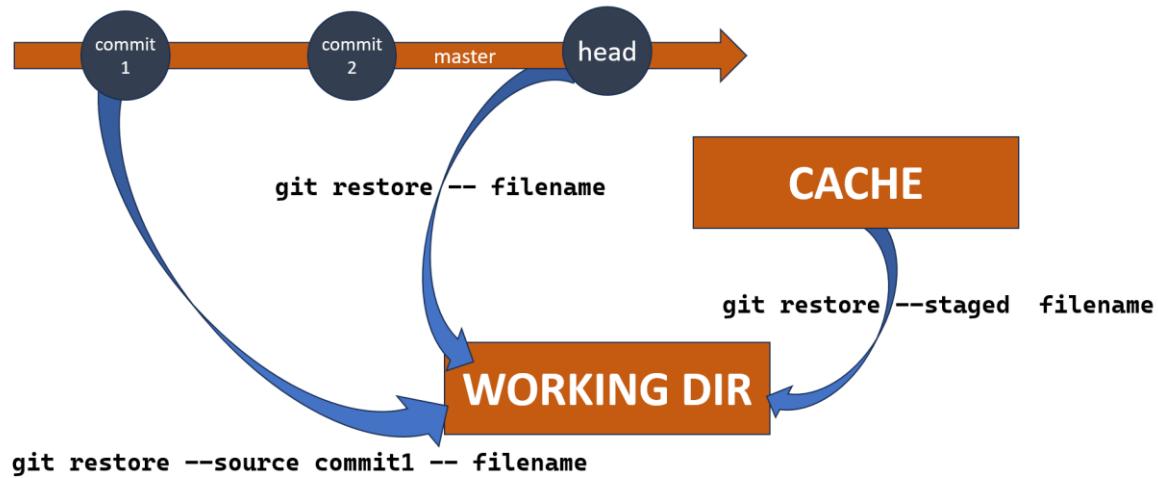
```
git restore --source commit_id -- filename
```

- Szybkie odtworzenie pliku do ostatnio commitowanej wersji (HEAD):

```
git restore -- filename
```

- Przesunięcie pliku ze stage do working dir (wycofanie operacji add):

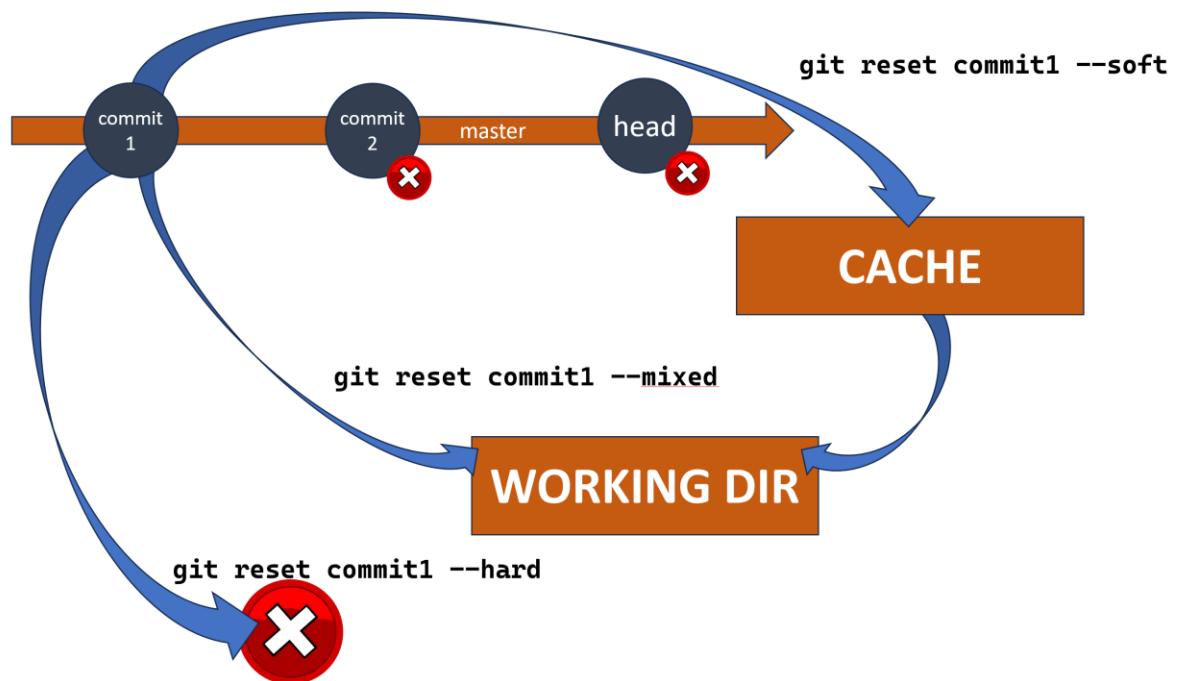
```
git restore --staged filename
```



Reset - powrót do wcześniejszego commita

- Polecam reset:
 - Przywraca cały wcześniejszy commit
 - Zmienia historię usuwając późniejsze commity
- W zależności od wykorzystanej opcji:
 - --soft - zmiany z wycofywanych commitów są łączone do jednej większej zmiany i umieszczane w stage i w working directory. W następnym kroku prawdopodobnie podejmiesz decyzję commitowania tych zmian.
 - --mixed - zmiany z wycofywanych commitów są łączone do jednej większej zmiany i umieszczenie w working directory. W następnym kroku prawdopodobnie dokończysz edycje plików, a potem dodasz je do stage i wykonasz commit. Mixed jest opcją domyślną.
 - --hard - repozytorium jest przywracane do stanu ze wskazanego commita, a zmiany są bezpowrotnie tracone
- Składnia polecenia:

```
git reset commit_id --mixed
```



Grzebanie w historii: checkout, commit amend, rebase

- Początkowo, git checkout obsługiwał wiele różnych sytuacji, które określało się za pomocą opcji. Z czasem w git zostały dodane dedykowane polecenia, które biorą na siebie część funkcjonalności git checkout. Stara składnia nadal jest obsługiwana, ale należy raczej stosować nową. I tak np.

```
git restore --source commit_id -- filename
```

Jest równoważne starszemu poleceniu

```
git checkout commit_id -- filename
```

- Jeśli po wykonaniu commita, zauważasz, że należałoby coś jeszcze w tym commitie dodać/zmienić, np. zmienić message dla commit, to możesz do polecenia git commit dodać opcję --amend. Powoduje ona aktualizację ostatnio wykonanego commita. Uwaga! Zmieni się przy tym commit id, tzn. ta komenda fałszuje historię:

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

- Gdy chcesz zmienić więcej commitów, to możesz użyć polecenia git rebase. Jeśli rebase zostanie uruchomiony w trybie interaktywnym (z opcją -i), to w notatniku będzie można zdefiniować, co ma się stać z każdym commitem, a do wyboru są między innymi:

- Pick - wybierz dany commit
- Drop - usuń commit
- Edit - wprowadź zmiany do commita
- Squash - spłaszcz commit wraz z innymi commitami
- Reword - zmień message dla commit

W zależności jakie polecenia zapiszesz w pliku tekstowym wyświetlonym po uruchomieniu git rebase, proces będzie się zatrzymywał pozwalając na wprowadzenie oczekiwanych zmian, np. na podanie nowego message dla commita:

```
git rebase -i HEAD~3
```

Instalacja Visual Studio Code

- Visual Studio Code to lekkie i darmowe narzędzie programistyczne dostarczane przez Microsoft, za darmo, na różne systemy operacyjne. Może być pobrane ze strony:
<https://code.visualstudio.com/download>
- Moc VSC bierze się z prostoty i licznych dodatków umożliwiających tworzenie kodu na różnych systemach operacyjnych.
- Kilka przydatnych skrótów klawiaturowych wspomnianych na lekcji:
 - Ctrl + Shift + P - otwarcie palety poleceń
 - F12 - przejdźcie do definicji funkcji
 - Ctrl + ` (odwrócony apostrof) - otwarcie/zamknięcie okna terminala

Markdown

- Markdown to język opisu prostych dokumentów tekstowych i jest często wykorzystywany przez programistów. Z jednej strony, jest prosty i czytelny nawet jeśli przeglądamy go w postaci pliku tekstu, z drugiej strony, po przekonwertowaniu do HTML daje przejrzyste dokumenty webowe.
- W Markdown można:
 - Opisywać strukturę dokumentu nagłówkami #, ##, ###, ...
 - Budować listy numerowane i punktowane
 - Formatować tekst (bold, italic)
 - Umieszczać linki [Opis](<https://www.kursyonline24.eu/> Kursy IT)
 - Umieszczać obrazki (link poprzedź znakiem wykrzyknik)
 - Tworzyć tabele
- Więcej szczegółów na stronie projektu: <https://www.Markdownguide.org/>
- VSC posiada rozszerzenia pozwalające na wygodną pracę z dokumentami Markdown.

Przesiadka z linii komend do interfejsu graficznego

- Visual Studio Code (VSC) pozwala na wygodne tworzenie programów i skryptów, a na dodatek dobrze integruje się z git-em.
- Najważniejsze miejsca w interfejsie to:
 - Panel „Source Control” w menu po lewej
 - Nazwa aktualnego branch - na pasku statusu u dołu po lewej
 - Command Panel - rozwijana lista u góry okna
- Wybierając polecenia, szukaj ikonek w pobliżu tych elementów interfejsu, próbuj również klikania prawym przyciskiem myszki
- Jeśli nie możesz znaleźć odpowiedniego polecenia w menu, lub po prostu chcesz wykonać jakieś polecenie z linii komend, to otwórz okno terminala i pracuj tam z gitem w trybie CLI

Branch - równoległa wersja kodu

- Branch to równoległa wersja kodu. Dzięki temu użytkownicy końcowi mogą uruchamiać aplikację bazującą na produkcyjnej, stabilnej wersji kodu, podczas gdy w branchu „Quality Assurance” testerzy mogą testować nową wersję aplikacji, a deweloperzy w branchu „Dev” mogą scalać swoje modyfikacje do przyszłej wersji aplikacji
- Tworzenie brancha:

```
git branch branch_name  
git switch --create branch_name  
git checkout -b branch_name
```

- Przełączanie między branchami

```
git switch branch_name  
git checkout branch_name
```

- Listowanie branchy

```
git branch
```

- Uwaga - polecenia tworzące branch nie zawsze przełączają cię do nowego brancha - jest tak np. w poleceniu git branch branch_name. Po wykonaniu tego polecenia należy się przełączyć do nowo utworzonego brancha poleceniem git switch branch_name
- Tworząc nowy branch zawsze zwracaj uwagę, w jakim branchu w tej chwili się znajdujesz. Nowy branch na początku znajduje się w takim stanie jak commit, na którym uruchomiono polecenie tworzące brancha.

Git log i Git Graph - rozkład jazdy branchy

- Historię zmian z podziałem na branche można uzyskać za pomocą polecenia
- `git log --oneline --all --graph --decorate`
- Do tego polecenia można utworzyć alias. Robi się to definiując parametr globalny, np. ll. Później ilekroć zostanie uruchomione polecenie git ll, to zostanie wykonana komenda git log ze skonfigurowanymi parametrami
- `git config alias.ll 'log --oneline --all --graph --decorate' --global`
`git ll`
- Przydatny dodatek do VSC pozwalający zilustrować branche to Git-Graph. Nie tylko że generuje ogólny graficzny opis zmian w repozytorium, ale dodatkowo pozwala na łatwe wykonywanie operacji na commitach i branchach z wykorzystaniem kliknięć myszy.

Merge - scalanie branchy (bez konfliktów)

- Nowe branchy można tworzyć w oparciu o inne istniejące branchy. Istotne jest, aby tworząc branch mieć aktywny właściwy branch „rodzicielski”
- Do mergowania używa się polecenia git merge. Polecenie to powinno również być uruchamiane w branchu do którego mają zostać dołączone zmiany ze wskazanego brancha. Tutaj do bieżącego brancha zostaną dołączone zmiany z brancha prague:

```
git merge prague
```

- Podczas mergowania:
 - Może istnieć potrzeba utworzenia nowego commita. Będzie tak wtedy, jeśli git zauważyc, że w czasie modyfikacji jednego brancha były również wykonywane zmiany na drugim branchu
 - Może zostać wykonany tzw. Fast Forward. Dzieje się tak wtedy, gdy wszystkie nowe zmiany miały miejsce tylko w jednym branchu, podczas gdy w drugim nie działało się nic. Ta operacja polega po prostu na przesunięciu do przodu znacznika z nazwą brancha - stąd nazwa „fast forward”. Przy tej metodzie domyślnie nie powstaje commit odpowiedzialny za merge i nie powstają żadne konflikty
- Polecenie git merge może zostać uruchomione z przełącznikiem -m pozwalającym na podanie „commit message”. Jeśli git merge zostanie uruchomione bez przełącznika -m, a będzie wymagany commit, to zostanie uruchomiony edytor pozwalający na wprowadzenie opisu commita
- Merge może być również uruchamiany w narzędziach z interfejsem graficznym użytkownika.

Merge - scalanie branchy (z konfliktami)

- Do konfliktów dochodzi, jeśli w dwóch scalanych branchach zostały zmodyfikowane te same linie tekstu. Git nie potrafi zdecydować, która zmiana jest poprawna, raportuje konflikt i zatrzymuje się.
- Plik, którym doszło do konfliktu zawiera dodatkowe znaczniki dodane przez git-a:

```
<<<<<< HEAD  
Wersja linii z pliku z aktualnego brancha  
=====  
Wersja linii z przychodzącego brancha  
>>>>> branch name
```

- Zadaniem programisty jest samodzielna ocena, jak ma finalnie wyglądać kod, który jest umieszony między znacznikami. Programista może zaakceptować jedną z widocznych wariantów, ale może też dokonać dowolnych zmian zarówno między znacznikami, jak i poza nimi. Po modyfikacjach wszystkie znaczniki i ewentualny niepotrzebny kod należy usunąć i zapisać plik.
- Po zakończeniu edycji plików z konfliktami należy dokończyć operację merge przez dodanie plików do stage i commit zmian:

```
git add .  
git commit -m "merging and resolving conflicts"
```

Merge - scalanie zmian (Merge Editor VSC)

- Visual Studio Code zawiera graficzne narzędzie Merge Editor, które:
 - Pokazuje zmiany między wersjami plików
 - Pozwala wygodnie akceptować zmiany z bieżącego brancha jak i z brancha przychodzącego
 - Może dokonać inteligentnego połączenia zmian z obu wersji plików
 - Pozwala podjąć ostateczne decyzje programiście
- Po rozwiązaniu konfliktów i zapisaniu zmian, pliki automatycznie są umieszczone w stage, a użytkownik musi tylko zatwierdzić zmiany uruchamiając commit.

Zarządzanie branchami

- Do wyświetlenia listy branchy użyj `git branch`
- Do wyświetlenia branchy, których zawartość została już scalona użyj `git branch --merged`, a do wyświetlenia branchy, które nie zostały jeszcze scalone użyj `git branch --no-merged`
- Zmiana nazwy brancha to `git branch move old_name new_name`
- Branch można usunąć używając `git branch branch_name -d`
- Uwaga! Jeśli branch nie był jeszcze scalony, to to polecenie zwróci błąd. W takim przypadku można uruchomić polecenie z opcją -D, ale wtedy usunięty zostanie wskazany branch wraz z jego commitami.
- Aby utworzyć branch w oparciu o istniejący commit użyj `git branch branch_name commit_id`

Tagowanie

- Tagowanie pozwala oznaczyć w projekcie ważniejsze commity, np. powiązane z określonym release lub ukończeniem ważnego etapu
- Taga utworzymy poleceniem

```
git tag tag_name commit_id
```

- Jeśli otagowany ma zostać bieżący commit, to w powyższej komendzie można opuścić commit_id
- Tagi można wyświetlić uruchamiając

```
git tag
```

- Jeśli tag jest niepotrzebny, to usuniesz go poleceniem

```
git tag tag_name -d
```

- Z tagów można korzystać podczas przełączania repozytorium do przeszłych commitów (podróże w czasie)

```
git checkout tag_name
```

- Podczas wysyłania repozytorium do GitHub tagi domyślnie się nie wysyłają. Trzeba użyć polecenia:

```
git push --tags
```

lub

```
git push origin v1.0
```

- Podobnie, aby usunąć zdalny tag uruchom:

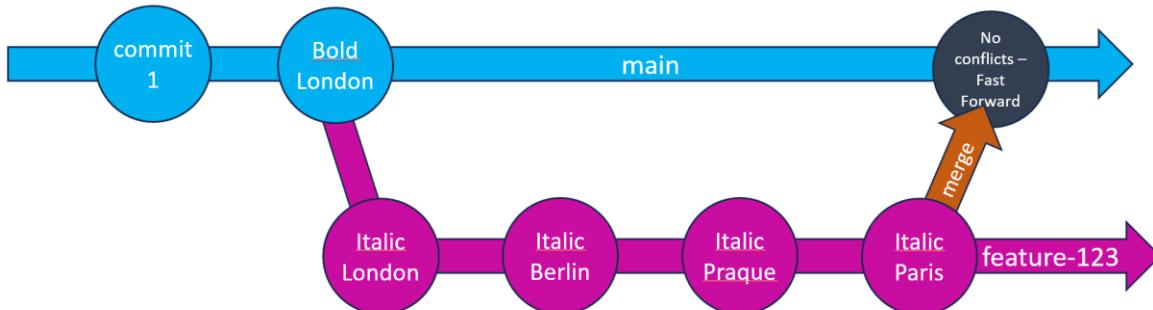
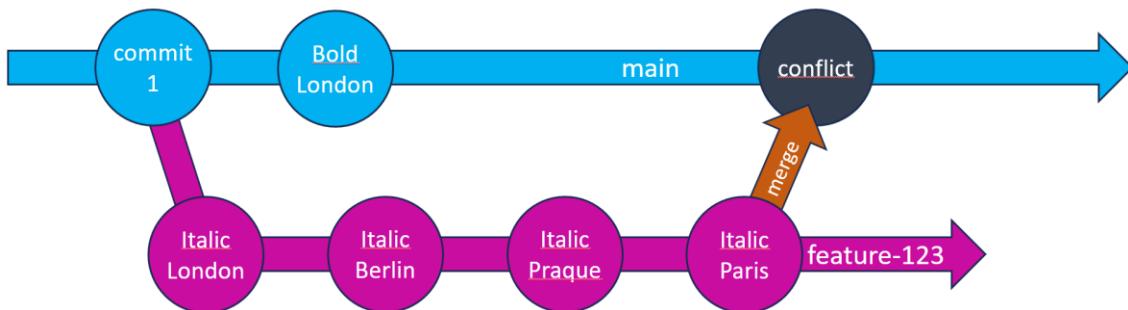
```
git push --delete origin v1.0
```

Cherry-Pick - smakowity commit

- Cherry pick przydaje się do pobrania zmian wprowadzonych przez jeden wybrany commit w innym branchu do naszego brancha
- git cherry-pick commit_id
- Zmiany zostaną „zimportowane” do bieżącego brancha, gdzie powstanie zupełnie nowy commit.
- Często, kiedyś w przyszłości oba branche będą podlegały scaleniu. W takim przypadku po uruchomieniu git merge może dojść do konfliktów, które należy rozwiązywać w standardowy sposób.

Rebase i jego zalety

- Jeśli potraktować feature branch, jako „potomka” głównego brancha, to git rebase powoduje, że ten potomek pochodzi od późniejszego commita u rodzica. Tutaj:
 - Commit_1 jest „wspólnym przodkiem” commitów występujących na branchu main i na branchu feature-123
 - Zmiany wprowadzane w commit „Bold London” oraz „Italic London” są konfliktowe. Ewentualny merge spowoduje konflikt, do którego dojdzie na branchu main
 - Po wykonaniu git rebase „wspólnym przodkiem” jest commit „Bold London”. Owszem, podczas wykonywania rebase mogło dojść do konfliktu, ale pojawił się on jeszcze przed prawdziwym merge i należy go rozwiązać w branchu feature-123
 - Później można wykonać merge brancha feature-123 do brancha main i tam zostanie wykonany fast forward, przy którym konflikty nie występują



- Zalety rebase to:
 - Ewentualny konflikt pojawia się w feature branch
 - Nie ma konfliktów podczas merge do brancha main
 - Historia jest liniowa
 - Feature branch „widzi” zmiany wprowadzane równolegle do brancha main
- Składnia to:

```
git switch feature-123
git rebase main
```

- Merge do brancha main będzie polegał na przesunięciu znacznika main - fast forward:

```
git switch main
git merge feature-123
```

Rebase - interactive

- Rebase pozwala na edytowanie commitów w trybie interaktywnym. Wystarczy w poleceniu dodać opcję `-i` oraz wskazać, ile commitów wstecz należy zmienić
- Tego trybu pracy można używać w wielu nietypowych sytuacjach, ale w lekcji przedstawiliśmy dwa scenariusze:
 - Commity znalazły się w branchu `main` zamiast w branchu `madrid`. Wtedy:

```
# przejdź do brancha madrid
git switch madrid
# przepisz do niego commity niepoprawnie zapisane do main
git rebase main
# wróć do brancha main
git switch main
# uruchom tryb interaktywny polecenia rebase
git rebase -i HEAD~3
# w pliku otwartym w edytorze zaznacz commity do usunięcia (polecenie drop)
# zapisz i zamknij plik
```

- Chcemy spłaszczyć 3 commity do jednego. Wtedy

```
# przejdź do brancha, w którym mają być spłaszczone commity
git switch madrid
# uruchom interaktywny rebase cofając się o 3 commity
git rebase -i HEAD~3
# w otwartym pliku tekstowym, wskaz, że:
# - pierwszy commit należy powtórzyć - komenda pick
# - kolejne commity należy spłaszczyć - komenda squash
# zapisz i zamknij plik
```

- Gdyby wprowadzone komendy miały błędy logiczne, to git rebase wyświetla stosowne komunikaty o tych błędach i proponuje sposoby naprawy sytuacji. Jednym z dobrych rozwiązań jest powrót do edycji komend zapisanych w pliku tekstowym definiującym pracę gita, a potem kontynuacja operacji rebase, działającej już wg nowej definicji:

```
git rebase --edit-todo
git rebase --continue
```

Stash - ach, zrobię to później

- Czasami, wykonanie pewnych komend nie jest możliwe, bo spowodowałoby one nadpisanie zmian znajdujących się w working directory, które jeszcze nie zostały zatwierdzone commitem
- W takim przypadku polecenie `git stash` może odłożyć te niezapisane zmiany „na bok”
- Można sprawdzić, co znajduje się w schowku uruchamiając `git stash list`
- Schowek na stash jest osobny dla każdego brancha
- Zmiany ze schowka można zaaplikować do plików w working directory uruchamiając `git stash pop`. To polecenie dodatkowo usuwa informacje znajdujące się w aplikowanym stash-u
- Ponieważ w schowku może być przechowywanych więcej zmian, to odkładając je, można je nazwać:

```
git stash save stash_name
```

- Od tej pory każde wywołanie `git stash list` wyświetli przy odłożonych zmianach również ich nazwy
- Zmiany ze schowka można przywrócić poleceniem `git stash apply` podając ewentualnie numer schowka. Polecenie `apply` nie usuwa wpisów ze stash.
- Aby usunąć wpis ze schowka użyj `git stash drop nr_schowka`
- Żeby usunąć wszystkie odłożone zmiany użyj `git stash clear`

GitHub - domek dla programisty

- GitHub.com to popularny serwis pozwalający na przechowywanie kodu
- Można tu założyć konto darmowe, które otrzymuje pełną funkcjonalność GitHub dla repozytoriów publicznych. Jeśli na darmowym koncie korzystasz z prywatnego repozytorium, to funkcjonalność będzie lekko okrojona
- Istnieją również plany płatne, które pozwalają na korzystanie z prywatnych repozytoriów z pełną funkcjonalnością, np. Teams lub Enterprise

Pierwsze repozytorium na GitHub

- Tworząc repozytorium zdecyduj o:
 - Nazwie repozytorium. Musi być unikalna w ramach konta/organizacji. Może być zmieniona później
 - Widoczności: publiczne lub prywatne
 - Dodaniu domyślnych plików:
 - README.md
 - LICENSE.md
 - .gitignore
- Interfejs webowy pozwala na wykonywanie większości prac z repo, np.:
 - Dodawanie/edykcja/usuwanie plików
 - Tworzenie i przełączanie się między branchami
- Zapisując zmiany, technicznie nie wykonuje się operacji „save” tylko „commit”.

Clone, push i uwierzytelnienie przez PAT

- Aby pobrać repozytorium z GitHub na komputer lokalny uruchom

```
git clone repo_url
```

- Polecenie git branch nie wykazuje domyślnie branchy zdalnych. Jeśli chcesz zobaczyć branche zdalne użyj opcji -r (remote) lub -a (all)
- Aby odesłać lokalne zmiany do GitHub używamy polecenia:

```
git push
```

- Do uwierzytelnienia sesji z GitHub można wykorzystać Personal Access Token. Wygenerujesz go przechodząc do Settings → Developer Settings → Personal Access Token
 - Token klasyczny daje dostęp do całego konta i wszystkich repozytoriów
 - Fine-Grained Token może dawać dostęp do wybranych repozytoriów i dlatego jest uznawany za bezpieczniejszy

Konfiguracja VSC do pracy z GitHub

- VSC może wykonywać większość komend git w trybie graficznym. W tym również klonowanie istniejącego repozytorium GitHub
- Zasady pracy gita, które znamy z linii komend nadal obowiązują w pracy z VSC. Np. jeśli przełączysz się na branch origin/dev, to w tle powstanie lokalny branch dev, który będzie się synchronizował z branchem origin/dev
- Za pull/push/fetch odpowiada polecenie „synchronize”
- PAT jest w Windows przechowywany w Credential Manager. To właśnie tam można w razie potrzeby zaktualizować PAT lub w przypadku problemów usunąć PAT, wygenerować nowy i wprowadzić go w okienku uwierzytelnienia wyświetlany podczas synchronizacji kodu
- Odpowiednikiem Credential Managera na MacOS jest OS Key Chain i można go skonfigurować z git komendą:

```
git config --global credential.helper osxkeychain
```

- Na Linux można skorzystać z lib secret:

```
sudo apt-get install libsecret-1-0 libsecret-1-dev
sudo make --directory=/usr/share/doc/git/contrib/credential/libsecret
git config --global credential.helper /usr/share/doc/git/contrib/credential/libsecret/git-credential-libsecret
```

Wysłanie lokalnego repo do GitHub

- Zdarza się, że na GitHub należy wysłać już istniejące lokalne repozytorium. W takim przypadku na GitHub tworzy się puste repozytorium (bez żadnych plików). W takim przypadku na stronie GitHub zobaczysz instrukcję mówiącą jakie czynności wykonać lokalnie, aby utworzyć lub zmodyfikować repozytorium zsynchronizowane z GitHub.
- Jeśli chcesz utworzyć nowe repozytorium i synchronizować je z GitHub użyj:

```
echo "# delme" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/user_name/repo_name.git
git push -u origin main
```

- Jeśli masz już repozytorium, które chcesz synchronizować użyj:

```
git remote add origin https://github.com/user_name/repo_name.git
git branch -M main
git push -u origin main
```

- W ten sposób synchronizacja zostanie ustawiona tylko dla brancha main. Jeśli masz inny branch, np. dev, który chcesz synchronizować z GitHub, to użyj polecenia

```
git switch dev
git push --set-upstream origin/dev
```

Tworzenie i usuwanie tagów i branchy na GitHub

- Domyślnie lokalnie utworzone tagi nie są wysyłane do GitHub przez polecenie push
 - Tagi można wysłać uruchamiając

```
git push --tags
```

- Podobnie, jeśli chcemy usunąć tag na GitHub korzystamy z poniższego polecenia, które jednak nie usuwa taga lokalnie, a wyłącznie zdalnie:

```
git push origin --delete tag_name
```

- Zgodnie z best practice, zmiany wprowadzane przez programistów nie powinny trafiać od razu do współdzielonych branchy. Zamiast tego programista powinien utworzyć swój własny branch tzw. „feature branch”, w którym wprowadza swoje zmiany, a potem scalą je z kolejnymi branchami.
 - Podczas wysyłania nowego brancha do GitHub, również nie wystarczy polecenie push. Zamiast tego należy uruchomić komendę, która tworzy nowego brancha na GitHub oraz powiąże lokalny branch z tym nowym, zdalnym branchem:

```
git push -u origin branch_name
```

- Po scaleniu zmian lokalny branch można usunąć. Wcześniej warto sprawdzić, czy branch został pomyślnie scalony. Poniższa komenda wyświetla informacje o scalonych branchach, które ewentualnie można usunąć:

```
git branch --merged
```

- Usuwanie lokalnego brancha wykonasz poleceniem

```
git branch -d branch_name
```

- Również i w tym przypadku nie wystarczy polecenie pull push, żeby wysłać do GitHub informację o usuniętym branchu. Zamiast tego należy uruchomić:

```
git push origin -delete branch_name
```

- Również pobranie informacji o branchu, który został usunięty zdalnie na GitHub do lokalnego repozytorium, nie dzieje się automatycznie. W takim przypadku należy:
 - Oczyścić repozytorium pobierając informacje o usuniętych branchach:

```
git remote prune origin
```

- To polecenie usuwa lokalne informacje o usuniętym zdalnym branch, ale nie usuwa brancha lokalnego, którego należy usunąć samodzielnie

```
git branch -d branch_name
```

- Stan branchy można wyświetlać poleceniem, które wyświetla wszystkie branche (opcja -a) z dodatkowymi szczegółami (opcja -vv)

```
git branch -a -vv
```

Zabezpieczanie repozytorium na GitHub

- Najważniejsze ustawienia repozytorium znajdziesz w sekcji **Settings → General**
 - Nazwa repozytorium
 - Domyślny branch
 - Konfiguracja wiki, issues, sponsorship, „preserve repository”, projects
 - Konfiguracja Pull Request
 - Allow merge commit
 - Allow squash merging
 - Allow rebase merging
 - Danger zone
 - Zmiana widoczności repozytorium na prywatne lub publiczne
 - Zawieszenie ochrony branchy
 - Przekazanie właścicielstwa repozytorium
 - Archiwizacja repozytorium
 - Usunięcie repozytorium
- **Pull request** to „oficjalna” prośba o pobranie zmian kodu, które zostały przygotowane przez developera. Developer „prosi” właściciela repozytorium o pobranie jego zmian, stąd ta nazwa **pull request**.
- **Settings → Collaboration** pozwala na zdefiniowanie zespołu programistów i wysłanie zaproszeń do współpracowników
- **Settings → Rules** pozwala na włączenie ochrony branchy i tagów. Reguły określają, które branche mają podlegać ochronie i na czym ta ochrona polega. Restrykcją mogą być objęte:
 - Tworzenie, modyfikacja, usuwanie branchy
 - Liniowa historia branchy
 - Przyjęcie zmian tylko po pomyślnym przejściu testów
 - Żądanie zatwierdzenia pull request przed scalaniem zmian, w tym wymóg prowadzenia dyskusji i uzyskania zdefiniowanej liczby pozytywnych recenzji
 - Force push

Pull Request

- Jedną z best practice w korzystaniu z Git-a jest niezapisywanie zmian bezpośrednio do głównych branchy projektu. Zamiast tego tworzymy najczęściej tzw. feature branch, w którym pracujemy nad nową funkcjonalnością. Kiedy zmiany są gotowe „oddajemy” je do brancha głównego.
 - W przypadku branchy, które nie są chronione, można wykonać merge
 - W przypadku branchy chronionych wymagane jest utworzenie formalnego pull request
- Pull request jest tworzony przez programistę, który chce by jego zmiany zostały pobrane do głównego brancha repozytorium. Główne branche repozytorium są często chronione przez policy, więc bezpośredni merge nie może być wykonany
- Po utworzeniu pull request:
 - Recenzenci mogą przejrzeć zmiany i je zaakceptować, odrzucić lub poprosić o dodatkowe zmiany
 - Podczas przeglądania pull request, łatwo można przejść do historii commitów i zobaczyć jakie zmiany i w jakich plikach były wprowadzane
- Po uzyskaniu wymaganych recenzji, wykonuje się pull request, który może być:
 - commit merge (scalenie dwóch branchy z rozwiązaniem konfliktów i zachowaniem pełnej kontroli)
 - squash (scalenie dwóch branchy z rozwiązaniem konfliktów, ale ze „spłaszczeniem” wszystkich przychodzących commitów do jednego commita)
 - rebase (przebazowanie brancha, a następnie jego scalenie przez fast forward)

Pull request - przesyłanie zmian między branchami

- Każda organizacja może określić własne zasady opisujące jak pracować z branchami. Istnieje wiele ogólnie przyjętych strategii, np.:
 - Git Flow Branching Strategy
 - GitHub Flow Branching Strategy
 - [What is the best Git branch strategy? | Git Best Practices](#)
- Żeby przesyłać zmiany np. z brancha dev do tst mogą być wymagane:
 - Utworzenie pull request
 - Uruchomienie automatycznych skanerów kodu
 - Zatwierdzenie zmian przez recenzentów
 - Scalenie zmian
- Podobnie może wyglądać przesyłanie zmian np. z brancha tst do prd
- Aby uniknąć innych sposobów scalania branchy, definiuje się policy, które chronią branche i uniemożliwiają scalenie zmian bez pull request.

Praca grupowa i security w GitHub

- GitHub wspiera pracę grupową, między innymi przez Wiki - do repozytorium można wbudować dokumentację w postaci Wiki, co pozwala lepiej organizować pracę całego zespołu
- W menu GitHub znajduje się pozycja Security. Jest to hub pozwalający przejrzeć różne ustawienia dotyczące bezpieczeństwa:
 - Security policy - sposoby zgłaszania luk w oprogramowaniu (do repozytorium zostanie dodany plik SECURITY.md z odpowiednią instrukcją)
 - Security advisories - miejsce przeznaczone na wewnętrzne dyskusje poświęcone podatnościom tworzonego kodu
 - Dependabot Reports - raporty dotyczące luk w pakietach, od których zależy kod repozytorium
 - Code Scanning - konfiguracja ustawień skanowania kodu w celu wykrycia „antywzorców” i złych praktyk
 - Secret Scanning - wykrywanie sekretów zaszytych w kodzie. Jeśli zostaną wykryte znane klucze/sekrety, to GitHub może je zaraportować do dostawcy danej usługi i automatycznie go unieważnić
- Istnieją też dodatkowe aplikacje, które można instalować na swoich repozytoriach, np. GitGuardian <https://www.gitguardian.com/>.

Issues & Projects

- GitHub pomaga w zarządzaniu pracą w projekcie informatycznym między innymi poprzez:
 - Issues – zadania przypisywane programistom, testerom itp. Issue posiada
 - Swój opis
 - Metadane, np. label, przypisana osoba, projekt, milestoneW oparciu o issue zapisany w Markdown jako checklist, można tworzyć kolejne issue klikając na pozycjach listy
 - Projects – zbiór issues. Projekty można śledzić na wiele sposobów, dobierając do nich odpowiedni szablon

Integracja pracy z issues i projects

- W commitach wykonywanych do domyślnych branch oraz w opisach pull request można stosować kilka słów kluczowych, które np. automatycznie zamkną issues, odnotowując postęp prac w milestones lub w projekcie:
 - Wpisz np. „closes”, potem naciśnij # i wybierz identyfikator issue, które powinno być automatycznie zamknięte przez dany pull request
 - Pełna lista słów uruchamiających automatyzację znajdziesz tu:
<https://docs.github.com/en/get-started/writing-on-github/working-with-advanced-formatting/using-keywords-in-issues-and-pull-requests>

Issue można również zamykać ręcznie.

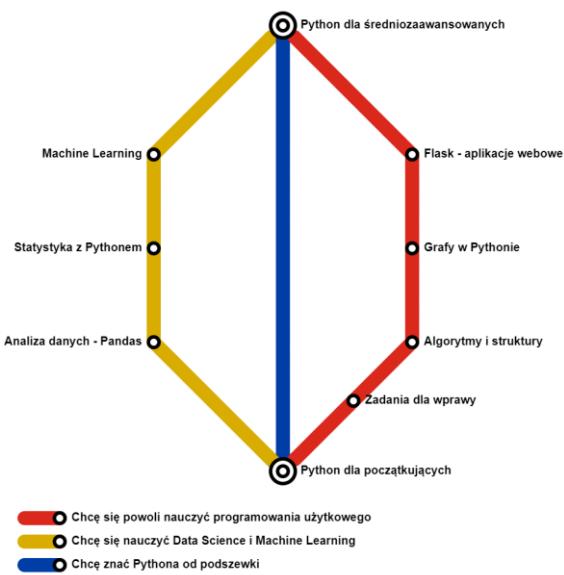
- Milestone (kamień milowy) to termin znany z zarządzania projektem. Oznacza on jakiś ważny etap prac nad produktem. Można śledzić, jak dobrze/źle idzie praca nad projektem, obserwując, ile procent pracy zostało wykonanej w ramach danych milestones. Projekt zazwyczaj zawiera wiele milestones, które mogą być powiązane np. z określonymi wersjami lub release.
- Projekt można analizować za pomocą wielu widoków projektu. Można też tworzyć własne widoki, wybierając kolumny i dane, jakie będą na nim prezentowane. Forma prezentacji widoku może zostać zmieniona nawet na formę wykresu.
- Zakładka Insights pozwala przeglądać informacje natury statystycznej na temat repozytorium: kto, jak często i ile zmian wysyła w danym repozytorium

Fork i Pull Request

- Fork to utworzenie kopii cudzego repozytorium na swoim własnym koncie. Operację fork można wyklikać na oknie oryginalnego repozytorium
- Użytkownik bez praw modyfikacji oryginalnego repozytorium ma pełne prawa na swoim własnym repozytorium
- Po wykonaniu zmian we własnym repozytorium użytkownik może odesłać zmiany, proponując właścicielowi oryginalnego repozytorium pobranie tych zmian. Robi się to przez Pull Request.
- Pull Request może zostać zatwierdzony, odrzucony, ale równie dobrze może wymagać ulepszeń. W takim przypadku należy rozpocząć dyskusję korzystając z komentarzy.

Spróbuj też!

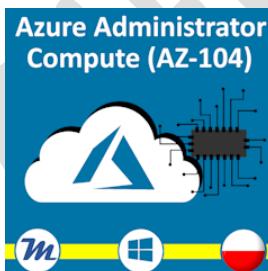
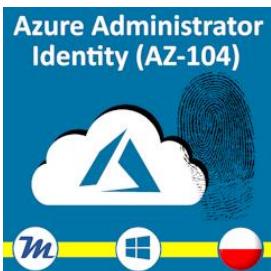
ŚCIEŻKA PYTHON



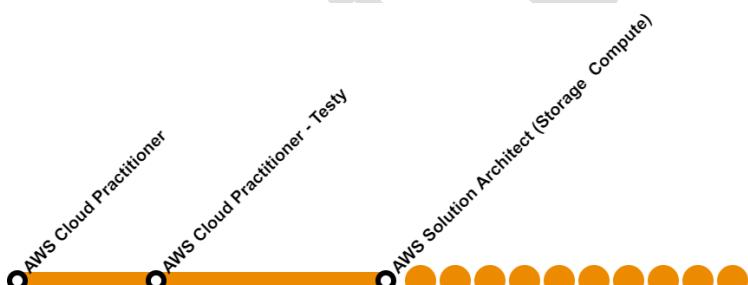
ŚCIEŻKA LINUX



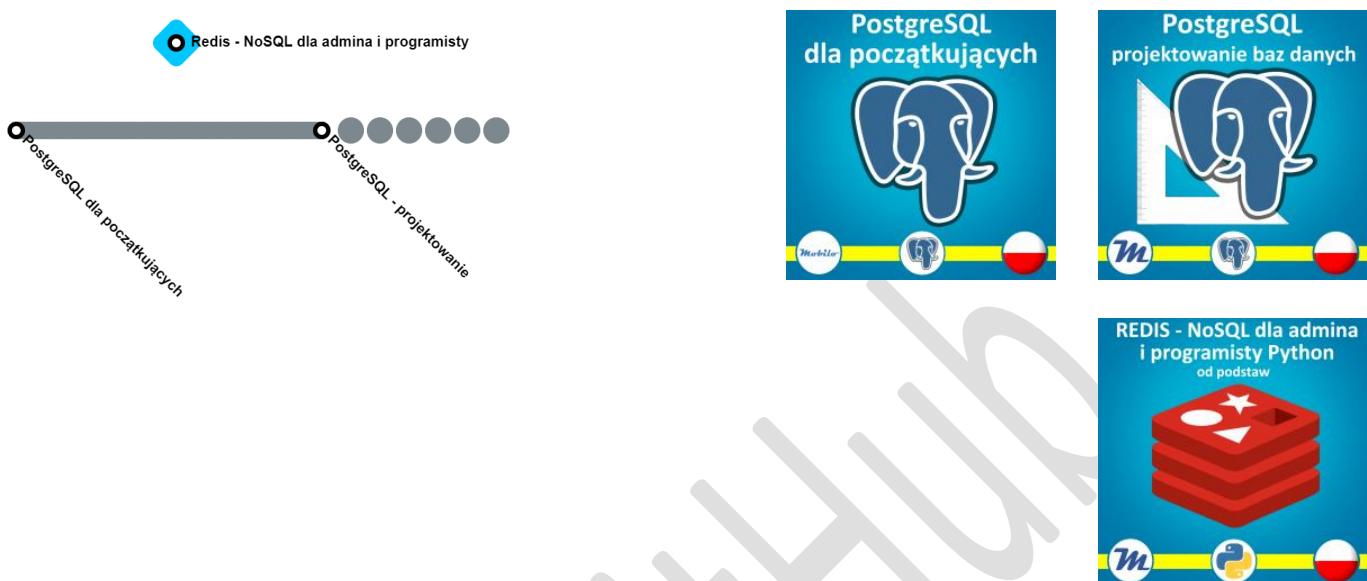
ŚCIEŻKA AZURE



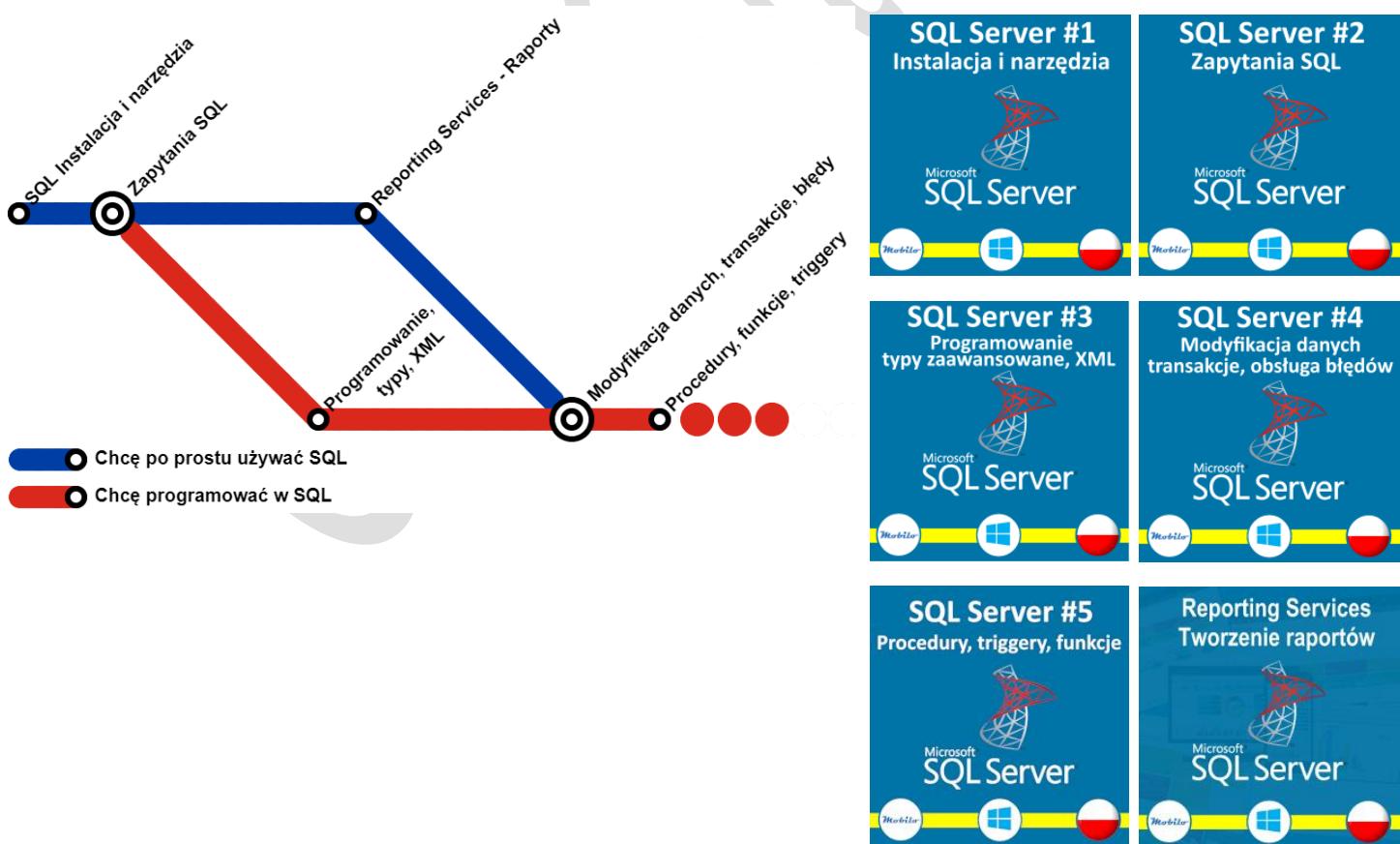
ŚCIEŻKA AWS



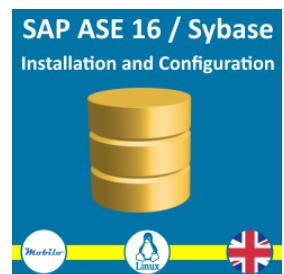
ŚCIEŻKA POSTGRESQL & REDIS



ŚCIEŻKA SQL SERVER



INNE KURSY



Git & GitHub