Git - powtórka

Źródła

https://www.jetbrains.com/help/idea/using-git-integration.html

https://learn.microsoft.com/en-us/collections/o1njfe825p602p?source=docs

https://www.youtube.com/watch?v=Jdc0i7RcBv8&t=19065s

Cel:

Podstawowa umiejętność posługiwania się GIT z poziomu konsoli oraz z poziomu inteliJ IDE

Konfiguracja GIT'a

1. Sprawdzenie aktualnej konfiguracji:
2. Dostępne parametry do konfiguracji:

```
git config –list
git config –help
git config --system user.name "Michał Białek"
git config --system user.email myEmail@email.com
```

Poziomy GIT'a

Dostępne mamy 3 poziomy konfiguracji git'a:

1. System - ustawienia dotyczą całego systemu operacyjnego
2. Global - ustawienia dotyczą konkretnego użytkownika systemu
3. Local - ustawienia dotyczą pojedyńczego repozytorium

Konfiguracja działa hierarchicznie

Sposoby zmiany ustawień GIT'a

1. Z poziomu CMD komendą git config
2. Edytując plik konfiguracyjny

Skrót

alt+ back tick ` → inteliJ - otwarcie panelu Git

Komendy

Commit —amend

1. Pozwala edytować ostatni commit w historii (zarówno jego commit message, jak i zawartość commitu)
2. Powoduje ona zmiane commit hasha
3. Jest to dobre rozwiązanie, ale tylko w przypadku, gdy jeszcze nie wypchaliśmy zmian do repozytorium zdalengo

Git restore

W przypadku, gdy chcemy przywrócić stan pliku, do takiego jaki był podczas ostatniego commitu

```
Git restore nazwaPliku
```

W przypadku, gdy chcemy usunąć plik z staging area (cofnięcie git add)

```
git restore --staged <plik>
```

Jeżeli chcemy przywrócić plik o kilka commitów w tył (do określonego commita)

```
git restore --source=<commit_hash> <plik>
```

Możemy również cofnąć stan wszystkich plików, przy pomocy komendy

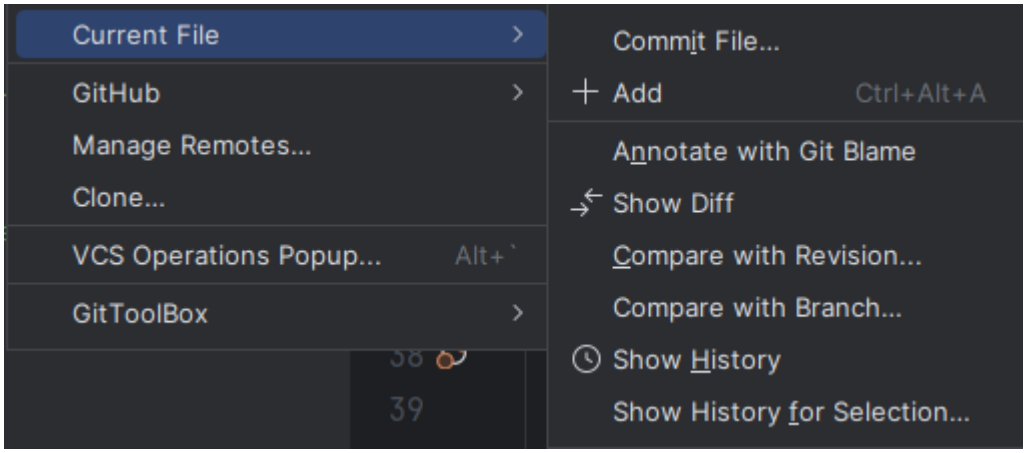
```
git restore .
```

Git Diff

Komenda służy do sprawdzenia, jakie są różnice pomiędzy bierzącymi zmainami, a ostatnim zatwierdzeniem HEAD.

Można również porónywać pomiędzy 2 commitami, branchami ect.

Najczęściej Git'a stosuje się w inteliJ: Git → Current File → Show Diff



Można rónież porównać zmiany z innymi branchami np. z zdalnym itp.

Git log

Służy do wyświetlania historii commitów (wyśeietla całą historię wraz z informacjami)

Można użyć również filtrów takich jak np.

```
1. Commity związany z określonym plikiem
git log <file>
2. Commity związane z konkretnym autorem
git log --author="Michal Bialek"
3. Forma skrócona
git log --oneline
```

Co to jest HEAD

Head pokazuje, gdzie aktualnie znajdujesz się na repozytorium

W przypadku checkout na commit z innego branchu, head się przenosi do tego commitu na wybrnym branchu

Można rónież przenosić się na inne commity z wybranrego branchu (git checount #hashBrancha) i następnie nasz head również przechodzi do wybranego brancha.

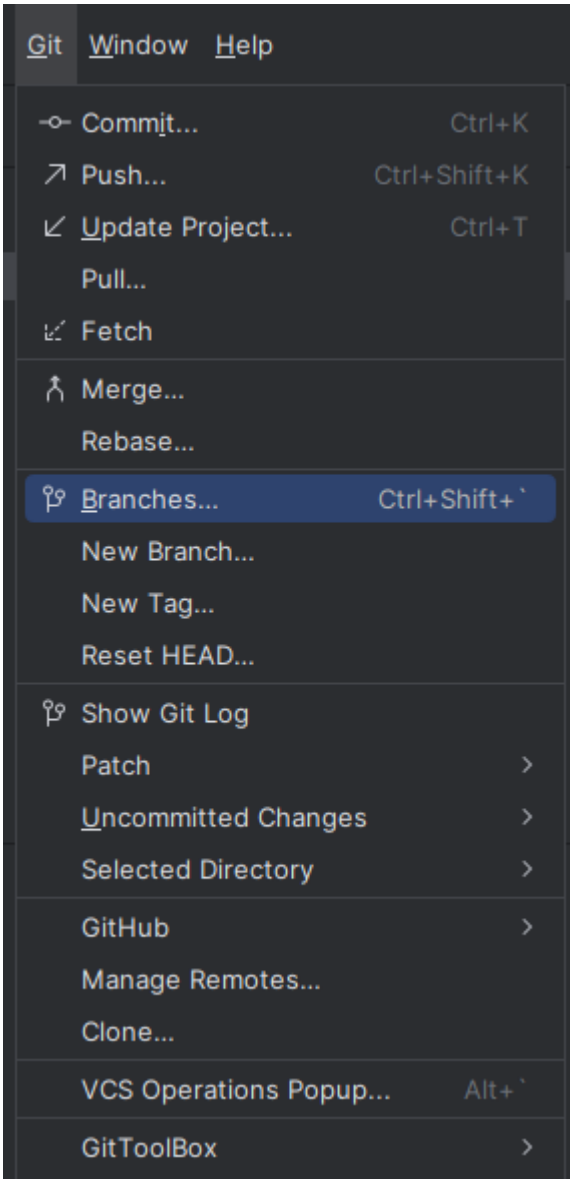
Jest to przydatne, gdy chcemy utworzyć nowego brancha od jednego z poprzedniego commita.
1. checout na poprzedni commit (head też się przenosi i jest w stanie detached) i przenosimy się na poprzedni commit

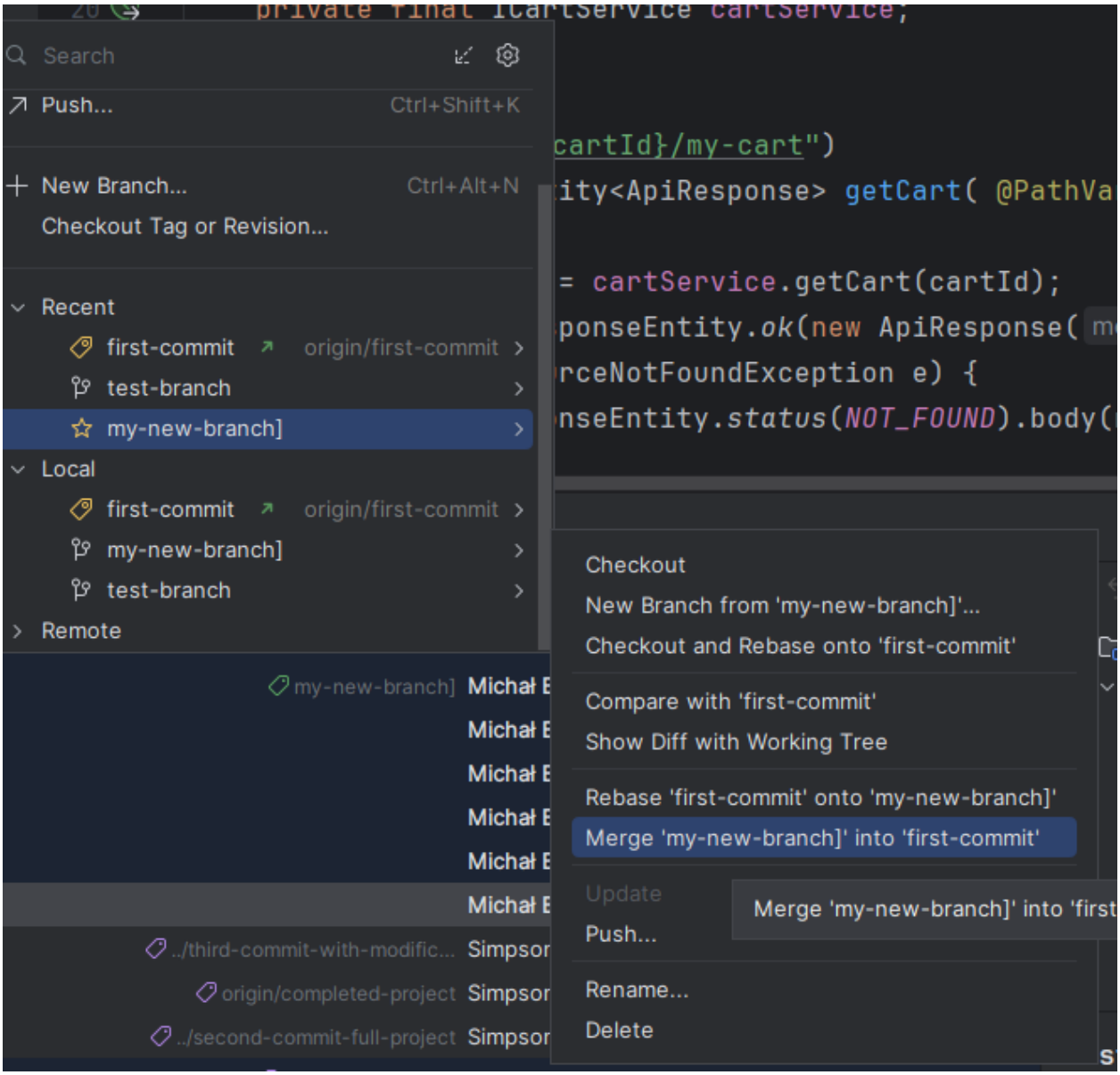
2. Następnie należy zrobić nowego brancha i dodawać zminay.
3. W przypadku, gdy nie utworzymy nowego brancha, commity nie będą widoczne

Opcje łączenia Branchy w InteliJ

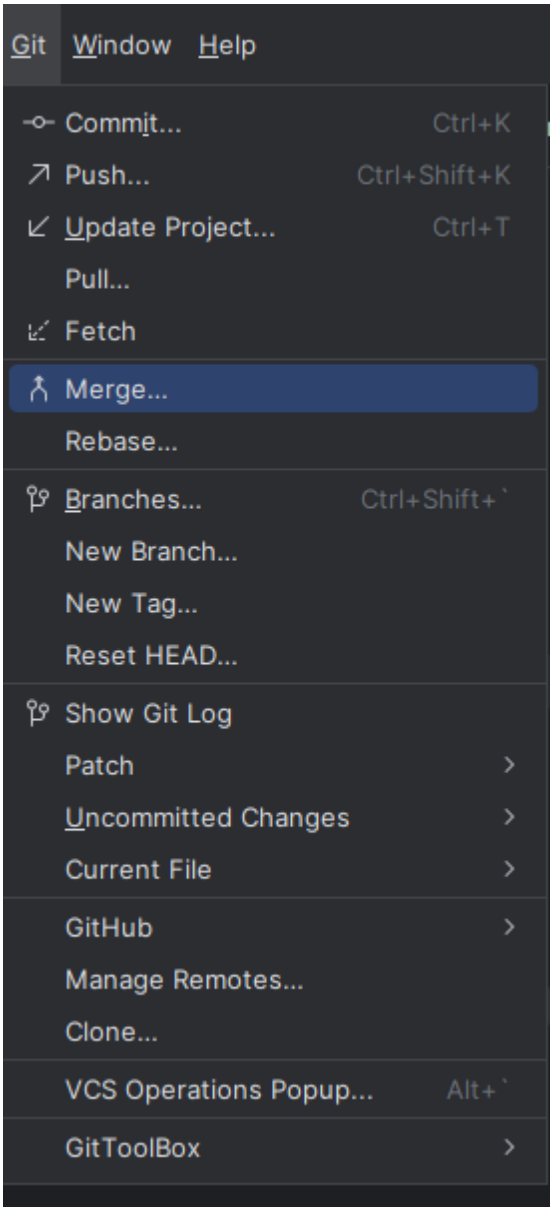
https://www.jetbrains.com/help/idea/apply-changes-from-one-branch-to-another.html#merge

1. Merge


Git
Window
Help
Commit... Ctrl+K
Push... Ctrl+Shift+K
Update Project... Ctrl+T
Pull...
Fetch
Merge...
Rebase...
Branches... Ctrl+Shift+`
New Branch...
New Tag...
Reset HEAD...
Show Git Log
Patch
Uncommitted Changes
Selected Directory
GitHub
Manage Remotes...
Clone...
VCS Operations Popup... Alt+`
GitToolBox


Search
Push... Ctrl+Shift+K
New Branch... Ctrl+Alt+N
Checkout Tag or Revision...
Recent
first-commit origin/first-commit
test-branch
my-new-branch]
Local
first-commit origin/first-commit
my-new-branch]
test-branch
Remote
Checkout
New Branch from 'my-new-branch]'...
Checkout and Rebase onto 'first-commit'
Compare with 'first-commit'
Show Diff with Working Tree
Rebase 'first-commit' onto 'my-new-branch]'
Merge 'my-new-branch]' into 'first-commit'
Update
Push...
Rename...
Delete
Merge 'my-new-branch]' into 'first

lub



Cofanie zmian - Git reset oraz git revert

1. Git revert - cofanie zmian poprzez dodanie revert commita cofającego zmianę. Możemy go stosować, nawet gdy zmiany są już wypchane do zdalnego repozytorium (nie zmienia historii, ale jego głównym problemem jest ‘zaśmiecanie histori’),
2. Git reset usuwa historię wstecz i mamy dostępne 5 opcji:
    a. Soft

i. Przenosi head do wybranego commita, jednocześnie wycofując pliki dodane podczas commitów, ale zostawia je w staging area, a commity usuwane są (nie ma ich)

ii. Zrzuca pliki, które były commited na staging area (do zacommitowania)

iii. Kiedy używać ?

1. Kiedy jak debil zamiast zrobić 10 commitów (i dotyczny ona 1 tylko 1 funkcjonalności, bo powinno być jak najwięcej commitów) to można wykonać reset soft i następnie scalić tego commita w 1
2. Przypadkowo zacommitowaliśmy plik, którego nie chcemy, wtedy możemy zrobić reset soft i następnie

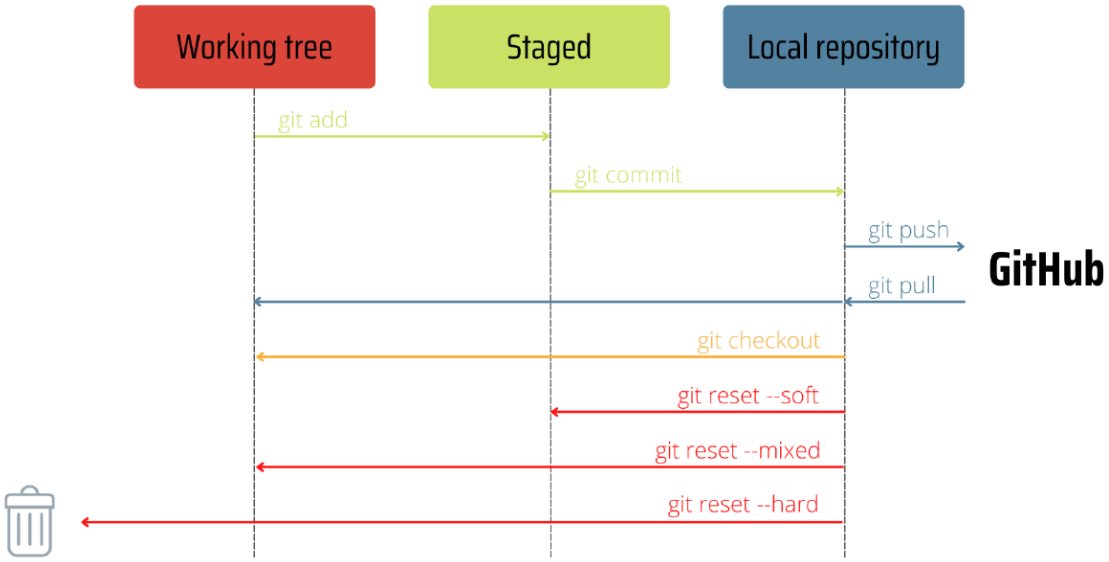
b. Mixed

i. Jedyną różnicą pomiędzy soft jest to, że pliki po cofnięciu nie będą w stanie staged - będą w working directory (musimy dodać poprzez add)

c. Hard !!!! Uwaga - usuwa historię z zmianami

i. Cofa commita, to stanu danego commita (nie zachwując zmian, jakie nastąpiły po nim) !

ii. Przypadek, gdy chcemy usunąć wszystkie zmiany od ostatniego commita:



Obraz 5. Git reset

Github branches

1. Git branch - zwraca nam dostępne aktualne branche
2. Git branch nazwaBrancha - tworzy nowy branch
3. Git checkout / Git switch - zmiana brancha
4. Git branch new-branch main - tworzymy bazując na gałęzi main
5. Git branch new-branch #hash - tworzy branch bazując od commita
6. Git checout -b - tworzy i jednocześnie przełącza się na podany branch

Stan detached HEAD

Stan Detached HEAD - oznacza, że wskaźnik HEAD wskazuje na konkretny commit, a nie na ostatni
commit w bieżącej gałęzi.

Konsekwencje pracy w stanie Detached HEAD są takie, że zmiany, które wykonujemy, nie zostaną

zapisane w żadnym branchu, a zamiast tego zostaną zapisane tylko w lokalnym repozytorium. Wszelkie
nowe commity, które wykonujemy w tym stanie, nie będą dodawane do żadnego brancha, co może
prowadzić do utraty wprowadzanych zmian.

Aby wyjść ze stanu Detached HEAD, należy utworzyć nowego brancha na aktualnym commicie i
następnie przełączyć się na ten branch. W ten sposób commity, które zostaną wykonane w trakcie pracy
w tym nowo utworzonym branchu, zostaną zapisane w historii zmian. Jest to polecany przez nas sposób
na wyjście ze stanu Detached HEAD, jeżeli planujesz w tym stanie wprowadzać jakieś zmiany.
Najbezpieczniej jest jednak wyjść z tego stanu, przełączając się z powrotem na branch, a stanu tego
używać, tylko do podglądania stanu repozytorium w danym commicie

Merge

1. Najważniejsze - przełącz się na branch DO KÓREGO chcesz mergować
2. git merge my-feature-brnach

Fast forword

Jest to sytuacja w której np. z future-branch mergujemy do main, na którym nie ma ŻADNYCH zmina i wtedy nie tworzy się merge scalający. Jeżeli chcemy wymusić istnienie merge’a scalającego to dodajmy flagę —no-ff

Case confliktów

Opcje w przypadku wystąpienia konfliktów:

1. git merge —abort

Cofanie merge’a (utworzenie nowego commita)

git revert -m 1 4d57bad → zostaną zmiany z master

git revert -m 2 4d57bad → zostaną zmiany z future-branch

Git rebase

Rebase jest procesem przenoszenia commitow w nowe miejsce (nową baze), gdzie ostatni commit otrzymuje nowego rodzica (parenta), którym jest ostatni commit z branchu main. Ważne jest, że przeniesione commity będą takie same, lecz będą miały nowych hash. czyli najlepiej zrobić rebase od main, rozwiązać konflikty i następnie checoutujemy się na main, i wykonujemy merge, który będzie fast-forward.

Podejście rozsądne:

Co jakiś czas wykonywać rebase z future-branch na main, i jak skończymy, przełączamy się main, i następnie wykonujemy merg fast forward.

Czego nie robić

1. Nie stosuje rebase na głównm branchu - broń boże

Git stash

Jest to komenda, która odkłada zmiany na potem (jak sama nazwa stash wskazuje).
Jest to przydatna komenta w przypadku, gdy zmiany nie są gotowe aby stworzyć commita, a trzeba np. przełączyć się na inny branch.

Zmiany są zapisywane w stosie (najstarszy ma najwyższy numer i jest na samym dole, najmłodsze na samej górze zaczynając od indeksu 0)
Komenda: git stash

Aby sprawdzić stash stack, należy wpisać komendę: git stash list

W dowolnym momencie i w dowolnej gałęzi można dodać: git stash apply stash@{stashNumber}

Aby usunąć git stash poprzez: git stash pop lub git stash drop ‘stash@{1}’

W inteliJ funkcja stash istnieje pod nazwą: shelve changes

https://www.youtube.com/watch?v=BSLzA8oCT7g

Pobiera wszystkie niezacommitowane (domyślnie tylko staged, ale idzie zmienić, żeby też zachowywał pliku untracked) w wkłada je do stosu

Wyświetlenie wszystkich stash →

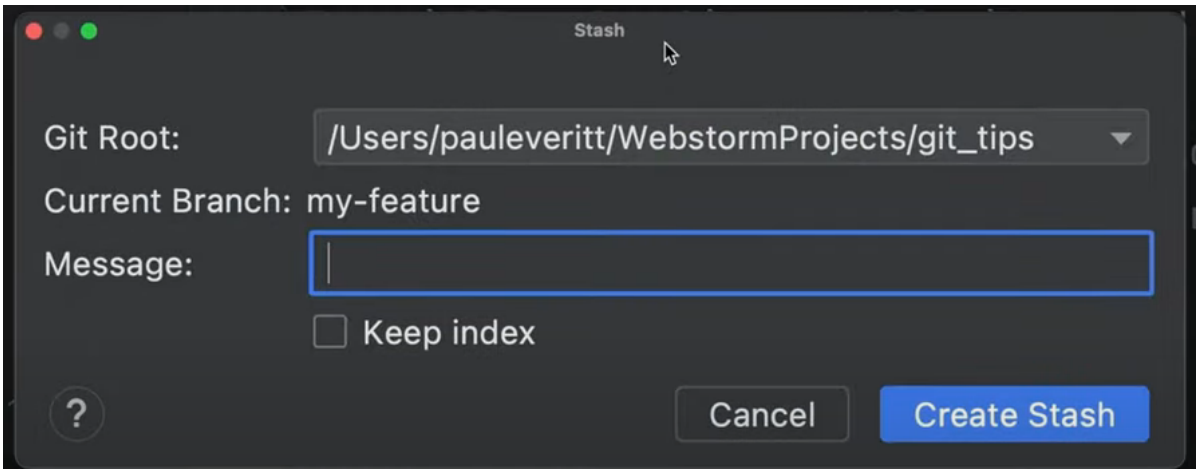
```
git stash list
```

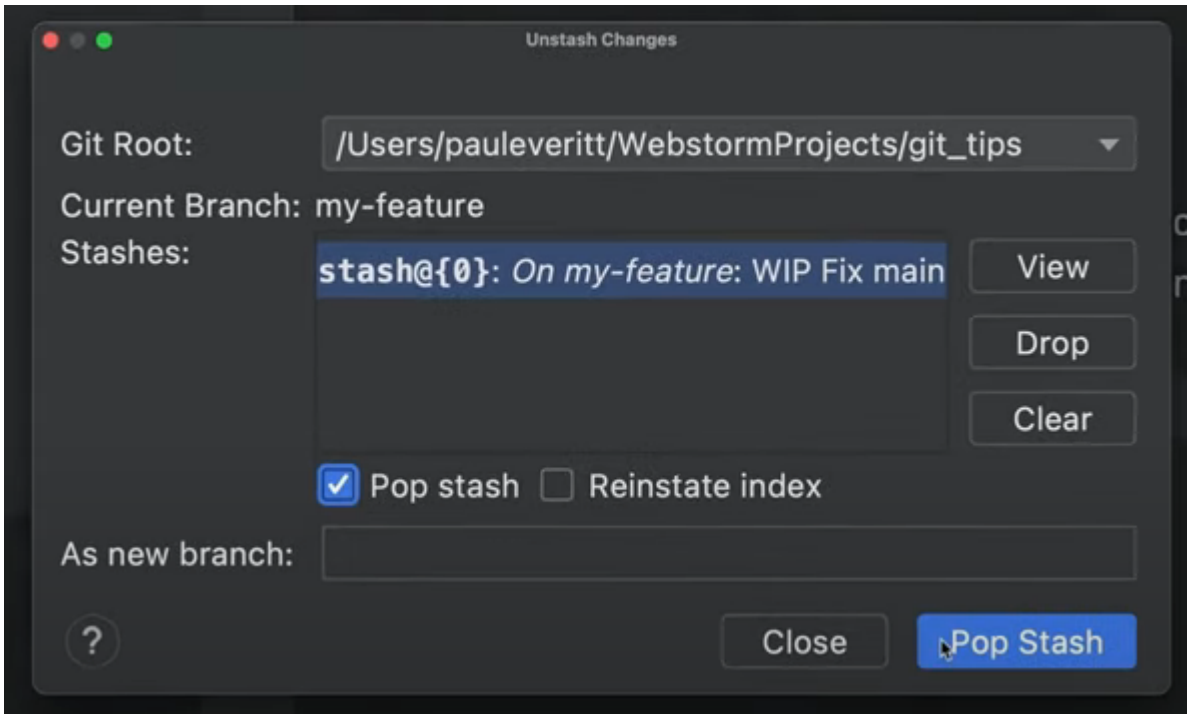
https://www.youtube.com/watch?v=rQXUSI50uhs

Historyjka:

1. Pracujesz na własnym branchu (masz 2 pliki, które nie są zacommitowane, ale dodane do trackowania (staged)) i musisz się przełączyć na inny branch (git checkout/switch)
2. InteliJ pokazuje, że są na branchu są rzeczy, które nie są zacommitowane, i można podjąć jedną z 3 akcji:
    a. Force Checkout - tracimy niezapisaną na branchu pracę, przechodzimy na wybrany branch
    b. Don’t check out - nie przechodzimy na podany branch, tylko wracamy do brancha i plików, które nie zostały zacommitowane
    c. Smart checkout - przekierwuje do akcji, w której chcemy wykonać merge do wybranego brancha - używamy, kiedy wprowadzone zmiany, chcemy przenieść na inny commit

    Najlepsze rozwiązanie → Don’t checkout → stash → przejście do innego brancha, gdzie chcemy coś wykonać → unstash





Rezpozytorium zdalne

1. Git clone - pobiera zdalne repozytorium
2. Repozytorium zdalne pełni funkcję **Single source of truth**
3. Praca z repozytorium zdalnym polega na

    a. Pobieraniu najnowszych zmian z repozytorium zdalnego, żeby być na bieżąco w projekcie
    b. Wysyłanie naszych zmian
4. Rozróżnienie na master (lokalny) oraz orgin master - zdalny

Git fetch

Komenda wyłącznie polega na pobraniu zmian, które są na repozytorium zdalnym (nie zostaną scalone zmiany do naszego kodu!)

git fetch lub git fetch origin - obie komendy służa do pobierania informacji o wszystkich gałęziach i commitach w repozytorium zdalnym

gir fetch orgin <nazwa_brancha> - pobiera informacje o konkretnej gałęzi

Jeżeli chcemy wciągnąć zmiany do projektu z repo zdalnego, robimy git merge (nie jest to nalepszy sposób)

Git pull = git fetch + git merge

jak często aktualizować repozytoria ? Zależy. Najlepiej stosować plugn GitToolBox.

W inteliJ mamy opcję Fetch, Pull lub kliknięcie Update Project w którym wybieramy, czy chcemy zrobić fetch+merge czy fetch+rebase

Wiele repozytoriów zdalnych

Git pozwala na zdefinowanie wiele repozytoriów zdalnych, jeżeli chcemy sprawdzić jakie mamy ustawione, wpisujemy: gi remote -v

Wysłanie zmian do repozytorium zdalnego

Mamy komendy

1. git push
2. git push orgin

3. git push orgin maser

Wszystkie robią to samo, bo mamy zdefinowany branch master

W przypadku chęci wypchnięcia, należy wygenerować **personal access tokens (PAT), którego następnie dodajemy do Credentials Menager (dla windows) i/lub inteliJ**

Opisnay problem: https://stackoverflow.com/questions/68775869/message-support-for-password-authentication-was-removed

Branche zdalne i lokalne

Branche lokalne muszą wiedzieć / być połączone z branchami zdalnymi.

Do stworzenia brancha zdalengo możemy stworzyć z poziomu GitHub, i następnie informację o jego istnieniu pobrać przy pomocy git fetch. Po przełączeniu się na niego, następuje skojarzenie brancha lokalnego z zdalnym.

Możemy w drugą stronę, stworzyć brancha lokalnie, przeączyć sie na niego, pracować na nim i wypushować, z tą różnicą, że wyskoczy komunikat:

```
fatal: The current branch feature-branch-from-local has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feature-branch-from-local
```

Dlatego musimy utworzyć brancha zdalnego poprzez komendę:

```
git push --set-upstream origin feature-branch-from-local
# lub
git push -u origin feature-branch-from-local
```

I otrzymamy informację, że należy złożyć pull request

Revert na zdalnym

W zdalnym nie cofamy się w histori, tylko robimy commita cofającego (reverta). Co prawda zaśmieca historię, ale jest skuteczny.

Pull vs Fetch +merge

Pullt = Fetch + merge

Fetch - pobiera dane z remote

Merge - łączy lokalne z remote (czyli przechodzi pointer main do mojego commita)

Fetch + merge daje większą kontrolę, bo sami musimy zrobić merge. Zamiast tego, możemy również zrobić pull z odpowiednimi flagami:

1. -ff-only - w tym przypadku tworzony jest obok (nowy branch z main) który może zostać obsłużony przez dewelopera w sposób, jaki on chce - nie występuje konflik
2. -ff - od razu łączy z obecnym poprzez merge (to jest domyślne zachowanie git pull, bez -ff)
3. -rebase (najlepsza opcja, bo od razu łączy w jeden)

Best Practices

General:

https://medium.com/datreeio/top-10-github-best-practices-for-developers-d6309a613227

1. Protekcja brancha master

   Należy zabezpieczyć bnracha master przed bezpośrednimi commitami.

   Najlepszą praktyką jest stosowanie wielu branchy pośrednich → dev→test→ prod

   Opcję protekcji, porzez wymaganie pull request ustawia się w github → repo → setings → branches → (wybierz master) → add rule → require pull request

....

2. Pamiętaj o konfiguracji użytkownika git

   Na początku poradnika jest napisane jak to skonfigurować

3. Używaj Code Owners

   https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners

   Po co to jest ?

   Code Owners jest stosowana, aby automatycznie przypisywać określone osoby do zespołów, i następnie przeglądania i zatwierdzania zmian.

   Jak to się robi ?

   Tworzy się plik w repozytorium o nazwie `CODEOWNERS` , umieszczamy w katalogu `.github/` lub `.docs/` Format pliku:

   ```
   [ścieżka/pliku] [nazwa_użytkownika_github] lub [@zespół]

   ex.

   # Wszyscy w zespole backend są odpowiedzialni za pliki w fold
   src/backend/ @backend-team

   # Użytkownik `janek` jest właścicielem pliku `README.md`
   README.md @janek
   ```

4. Nie podawaj secrets/credentials w source code

   Wiadomo, takich rzeczy nie podaje się w src. Najeży takie dane albo wstrzykiwać jako zmienne środowiskowe, a najlepiej używać narzędzi, które umożliwiają na bezpieczne przechowywanie i wsytrzykiwania credentiali np. AWS Secrets Manager

   Dobrą praktyką jest stosowanie narzędzi takich jak:

    1. Git secrets - https://github.com/awslabs/git-secrets - jest to narzędzie stworzone przez AWS, które działa jako git hook, i sprawdza przy operacjach git, czy podane zmiany nie zawieraja patternów, które mogłby sugerować, że podawane są credentiale.
    2. Git hooks - też można bezpośrednio wyszukiwać zdefinowanych patternów
5. Nie wrzucaj dependencji do src

   Takie rzeczy uwzględnia się w git ignore, żeby tylko scr był zawarty
6. Nie powinno się również umieszczać w git plików konfiguracyjnych
7. Twórz dobry plik gitignore

   Templatki można uzyskać tutaj: https://www.gitignore.io
8. Jawne określenie wersji zależności w plikach manifest (dla mvn → pom.xml, gradle → build.gradle)

Commiting best practices

https://gist.github.com/luismts/495d982e8c5b1a0ced4a57cf3d93cf60

1. Single responsibility principle w kontekście commitów
    a. commity powinien być mały, i dotyczyć tylko 1 funkcjonalności. Nie powinno się zrobić commitów, które odejmują więcej niż 1 funkcjonlność.
2. Commit often
    a. Commity powinny być możliwe małe
3. Nie commituj zmian, które nie są w pełni ukończone
    a. Tak. Później wraca się do projektu, i wiecej czasu zajmuje ogarnianie, co trzeba dokończyć
4. Testowanie kodu, zanim wykona się commit
    a. W idelanym świecie testuj wprowadzony kod przed wdrożeniem commita, powinno się testować

5. Pisz dobre commit message:
    a. do 50 znaków
    b. Odpowiadające na pytania:
        i. What was the motivation for the change?
        ii. How does it differ from the previous implementation?
    c. Używaj imperative, present tense → tryb rozkazujący teraźniejszy
        i. «change», not «changed» or «changes»
6. Używaj branchy
7. Uzgodnijcie workflow

Cheat Sheet:

Operation	Command
Clone an existing repository	`$ git clone ssh://user@domain.com/repo.git`
Create a new local repository	`$ git init`

Local Changes

Operation	Command
Changed files in your working directory	`$ git status`
Changes to tracked files	`$ git diff`
Add all current changes to the next commit	`$ git add .`
Add some changes in to the next commit	`$ git add -p <file>`
Commit all local changes in tracked files	`$ git commit -a`
Commit previously staged changes	`$ git commit`
Change the last commit (**Don't amend published commits!**)	`$ git commit --amend`

Commit History

Operation	Command
Show all commits, starting with newest	`$ git log`
Show changes over time for a specific file	`$ git log -p <file>`
Who changed what and when in	`$ git blame <file>`

Branches and Tags

Operation	Command
List all existing branches	`$ git branch`
Switch HEAD branch	`$ git checkout <branch>`
Create a new branch based on your current HEAD	`$ git branch <new-branch>`
Create a new tracking branch based on a remote branch	`$ git checkout --track <remote/branch>`
Delete a local branch	`$ git branch -d <branch>`
Mark the current commit with a tag	`$ git tag <tag-name>`

Update and Publish

Operation	Command
List all currently configured remotes	`$ git remote -v`
Show information about a remote	`$ git remote show <remote>`
Add new remote repository, named	`$ git remote add <remote> <url>`
Download all changes from , but don't integrate into HEAD	`$ git fetch <remote>`
Download changes and directly merge/ integrate into HEAD	`$ git pull <remote> <branch>`
Publish local changes on a remote	`$ git push <remote> <branch>`
Delete a branch on the remote	`$ git branch -dr <remote/branch>`
Publish your tags	`$ git push --tags`

Merge and Rebase

Operation	Command
Merge into your current HEAD	`$ git merge <branch>`
Rebase your current HEAD onto (**Don't rebase published commits!**)	`$ git rebase <branch>`
Abort a rebase	`$ git rebase --abort`
Continue a rebase after resolving conflicts	`$ git rebase --continue`
Use your configured merge tool to solve conflicts	`$ git mergetool`
Use your editor to manually solve con- flicts and (after resolving) mark file as resolved	`$ git add <resolved-file> $ git rm <resolved-file>`

Undo

Operation	Command
Discard all local changes in your working directory	`$ git reset --hard HEAD`
Discard local changes in a specific file	`$ git checkout HEAD <file>`
Revert a commit (by producing a new commit with contrary changes)	`$ git revert <commit>`
Reset your HEAD pointer to a previous commit and discard all changes since then	`$ git reset --hard <commit>`
Reset your HEAD pointer to a previous commit and preserve all changes as unstaged changes	`$ git reset <commit>`
Reset your HEAD pointer to a previous commit and preserve uncommitted local changes	`$ git reset --keep <commit>`

Best practices for pull requests

Good to know:

→ Creating a pull request - przypadek, gdy jesteśmy częścią zespołu i mamy możliwość tworzenia branchy

→ Creating a pull request from a fork - przypadek, gdy nie jesteśmy częścią zespołu i nie mamy możliwości tworzenia branchy

Co to jest Pull Request ?
Jest to mechanizm wdrażania zmian - swojego brancha do głównego brancha projektowego. Powinien on posiadać informacje na temat wprowadzonych zmia. Inni członkowie mogą przeglądać zmiany, zgłaszać poprawki, komentować ect.. W przypadku pozytywnego code review przez innych programistów, zmiany są wdrażane

Dobre praktyki

1. Powinny być małe i obejmować 1 funkcjonalność
2. Sprawdz samemu, zbuduj i przetestuj kod zanim wystawisz pull request
3. Opis
    a. Czemu zmieniamy kod

b. Co zostało zmienione

c. Opcjonalnie: linki ect.

Pulling merge requests

Ta sekcja dotyczy właścicieli repozytoriów

Tagowanie

Pozwalają na oznaczanie konkretnych punktów w historii zmian - np. wersję stabilną, wersję produkcyjną ect.

Na podstawie tagu, możemy przełączyć się na jego

1. Tworzenie tagu

```
git tag nazwaTagu hashCommitu
ex. git tag v1.1.1 49034
```

2. Jeżeli chcemy otagować najnowszy commit

```
git tag v1.1.1
```

3. Zobaczenie listy tagów

```
git tag
//lepiej
git tag --pretty=oneline
```

1 commit może mieć wiele tagów, natomiast 1 tag może mieć tylko 1 commita.

Czyli:

Nie możemy, żeby kilka commitów miało tą samą nazwę taga - np.

```
//Tak nie można
git tag v1.1 434224234
```

```
git tag v1.1 897898778
```

Ale za to, można użyć flagi -f (force), która powoduje, że jak już jakiś commit ma ten tag, to zostanie on przeniesiony, do aktualne nadawanego commita (i zostanie usunięty z starego)

```
//Przeniesienie tagu
git tag v1.1 434224234
...
git tag -f v1.1 897898778
```

Rodzaje tagów

Tagi różnią się od siebie liczbą przechowywanych metadanych.

1. Lightweight tags - przechowuje tylko nazwę i wskaźnik na konkretny commit (zwykły git tag)
2. Annotated tag - może zawierać więcej metadanych między innymi:
    a. Datę utworzenia taga
    b. Wiadomość taga
    c. Osoba+mail do osoby, która tego taga stworzyła

Stworzenie annotated tag:

```
git tag -a v1.1 -m "Tag message"
```

Wyszukiwanie tagów

Podana komenda pokazuje informacje na temat annotated tag

```
Git show v1.1
```

Wypychanie taga do repozytorium zdalnego

Zwykłe wypchanie nie spowoduje wypchania tagów. Do wypchania trzeba podać taga:

```
git push orgin tagName
```

Jeżeli chcemy wypchać wszystkie tagi, możemy użyć komendy:

```
git push --tags orgin
```

Na portalu github, wszystkie tagi znajdziemy obok informacji o branch'ach:

To umożliwia wybranie konkretnego tago, oraz pobranie zip'a z stanem pod danym tagiem

Usunięcie taga:

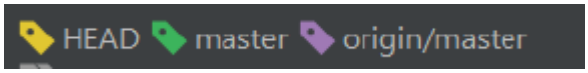
Są 2 opcje do usunięcia taga → z poziomu GitHub → z poziomu CMD

1. GitHub → wybranie taga → opcja Delete
2. CLI (po prostu dodanie flagi/opcji, delete)

```
git tag -d  v1.1
//lub
git --delete v1.1
```

InteliJ

HEAD, Master oraz Orgin/Master przypisane tagi:



InteliJ dodatkowo z poziomu widoku commitów, poprzez kliknięcie prawego przycisku na commit, umożliwia nam stworzenie commita, jak i również jest możliwa opcja utworzenia go z górnego panelu.

Kasować tak samo możemy z poziomu inteliJ

Aliasy

Umożliwa nam na stosowanie skróconych zapisów, dla długich komend np.

Chcemy używać podanej komendy:

```
git log --pretty=format:'%h %ad | %s %d [%an]' --date=short
```

ale jest długa, to możemy sobie stworzyć alias np. o nazwie nice-log:

```
git config --global alias.nice-log "log --pretty=format:'%h %ad
%s %d [%an]' --date=short"
```

Teraz, po zdefinowaniu aliasu, nie musimy ciągle wpisywać długiej komendy, tylko:

```
git nice-log
```

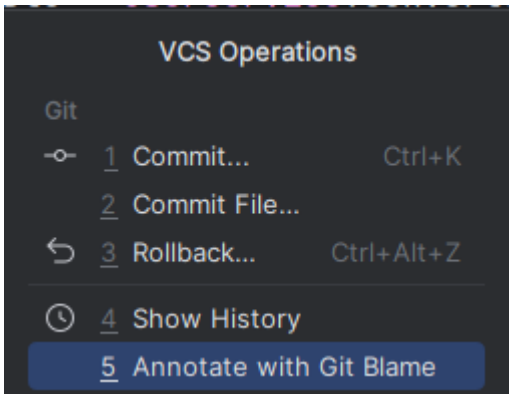
Możemy również w aliasach operować na zmiennych jak np.

%h - Skrócona wersja commit hash,
- %H - Długa wersja SHA-1 commit hash,
- %ad - Data powstania,
- %s - Wiadomość dodawana do commita,
- %d - Branch lub Tag,
- %an - Autor, %Cgreen - Drukuje na zielono

Git blame

Pokazuje, historię pliku - kto co kiedy zrobił

Z poziomu inteliJ:



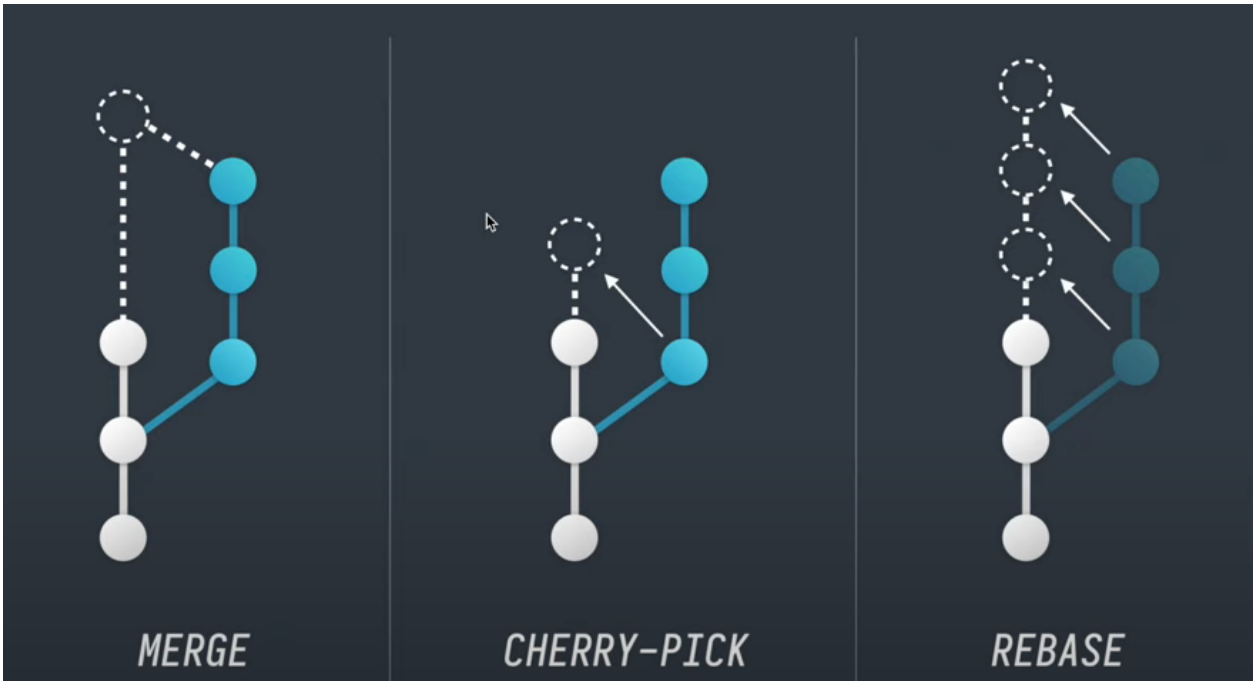
Więcej informacji: https://www.jetbrains.com/help/mps/investigate-changes.html#nonroot-node-history

Cherry Pick

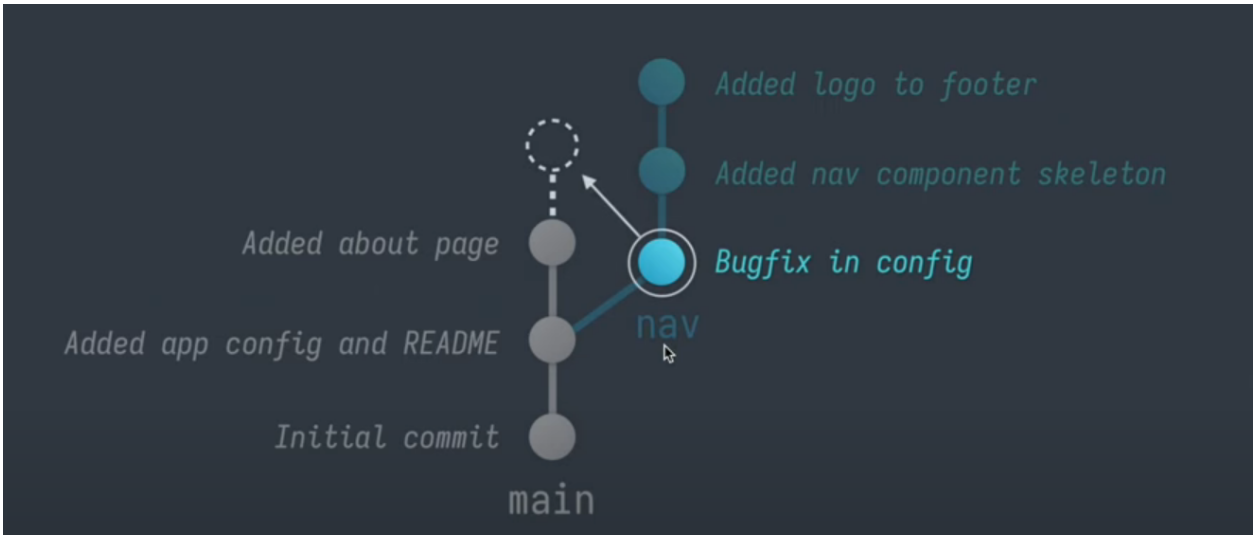
Tzw. wyciąganie wisieniek, a raczej commitów z innych branchy

https://www.youtube.com/watch?v=i657Bg_HAWI

Tak więc, Cherry Pick pozwala nam na wybranie jednego commita, z naszego brancha i wdrożenie tylko jego, do innego brancha.



Przykład:
Ktoś w swoim branchu, w pierwszym commicie naprawił jakiś bug, który znajduje się na main.
Nie wdraża się całego brancha, tylko commit naprawiający bug, poprzez **cherry pick.**



Przebieg:

1. Będąc na branchu main, możemy sprawdzić wypchane zmiany brancha kolegi, który rozwiązał bug'a

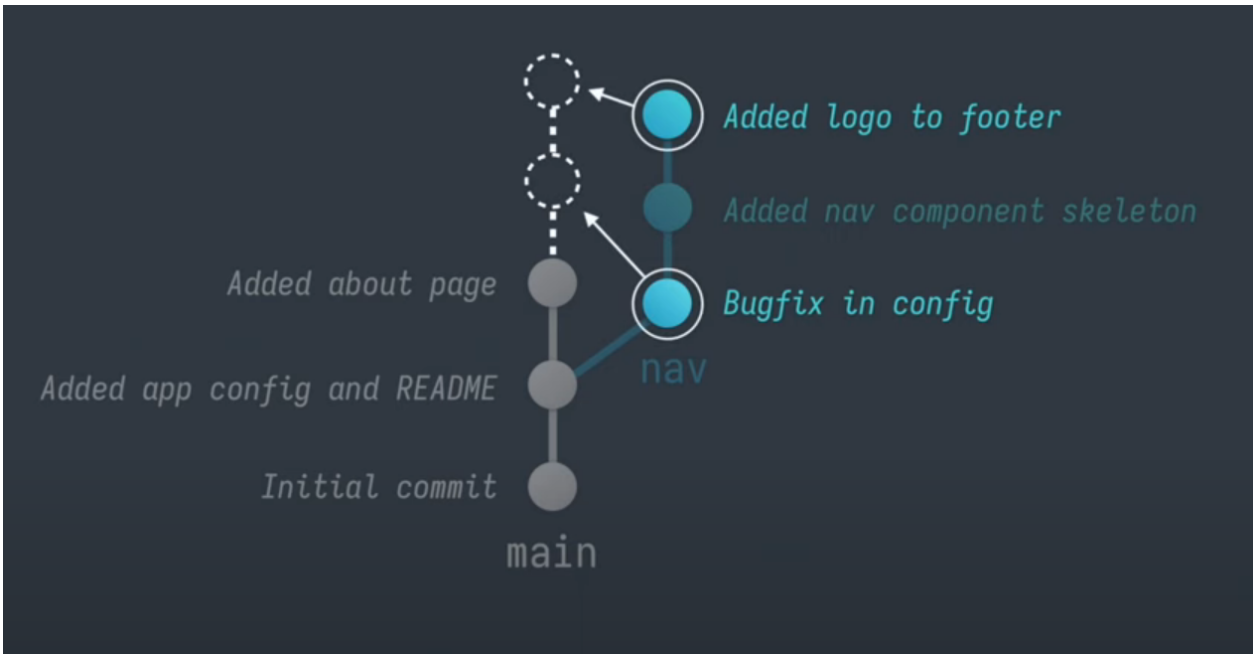
```
git log collegeBranchWithFixBug --oneline

//EXAMPLE RESULT:
fs323f2 Initial commit
23424r2 Add new feature to login
g5g4345 Bugfix LoginService method resolveRequest
... -> kopujemy commit który chcemy do cherry pick (g5g4345)
//Upewnienie się, że jest się na odpowienim branchu
git status
//cherry pick
git cherry-pick g5g4345
```

Commit dodany do brancha, uzyskuje nowego hash'a (obvious)

Wyciąganie wielu wisieniek

Jest możliwość również wyciąganiu wielu commitów, z pominięciem tych, które nam nie odpowiadają poprzez cherry-pick - jedynie co potrzebujemy to hash'e od tych commitów



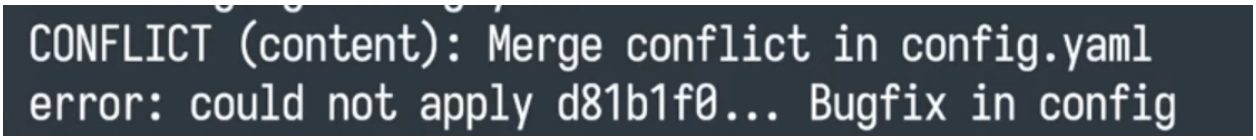
Komenda niczym się nie różni (dodaje się po koleji hash commitów) i są one dodawane w kolejności, w kolejności, jakiej się podało w komendzie

```
git cherry-pick g5g4345 (pierwszy) f2r323 (drugi)
```

Co jak są konflity ?

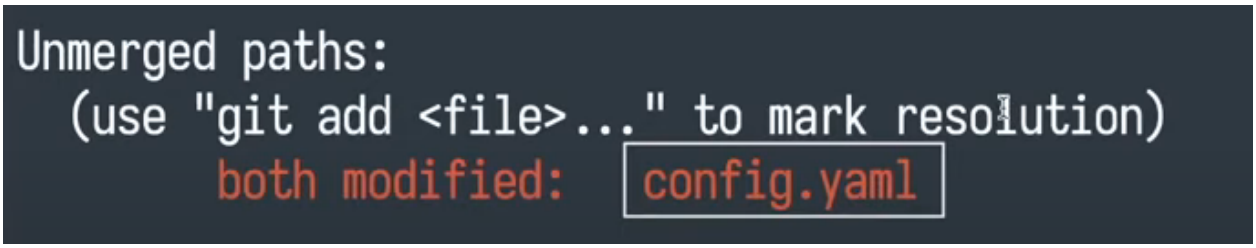
Tak samo jak dla merge confliktów:

1. Po wykonaniu cherry-pick pojawi się komunikat o konflikcie np.

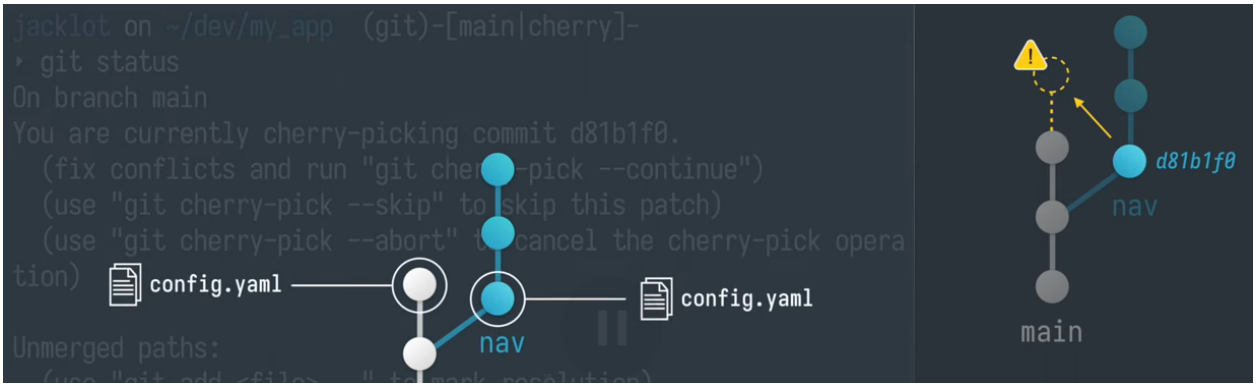


Wpisując git status dostajemy podpowiedź o:

1. Możliwych opcjach takich jak:
    a. cherry-pick —skipp (poija commity, które są konfliktowe, a resztę wdraża)
    b. cherry-pick —abort (przeywa cały proies, żadne commity nie są wdrażane)
2. Informację o plikach, gdzie występują konflikty:



Opis sytuacji:



Fix:

Wchodzimy do pliku, który jest konfliktowy

Git wskazuje nam:

```
4 <<<<<<< HEAD
5 twitter_username: my_app
6 youtube_username: MyApp
7 show_drafts: true
8 =======
9 twitter_username:
10 instagram_username:
11 youtube_username:
12 show_drafts: false
13 >>>>>>> d81b1f0 (Bugfix in config)
```

Część pierwsza do kreski - sytuacja w aktualnym pliku

Część po kresce - sytuacja jaka jest w pliku, który chcemy wdrożyć

I robimy, żeby było dobrze i zapisujemy + git add

Kontynuujemy → git cherry-pick —continue

https://www.youtube.com/watch?v=aUeNbpSkY8k

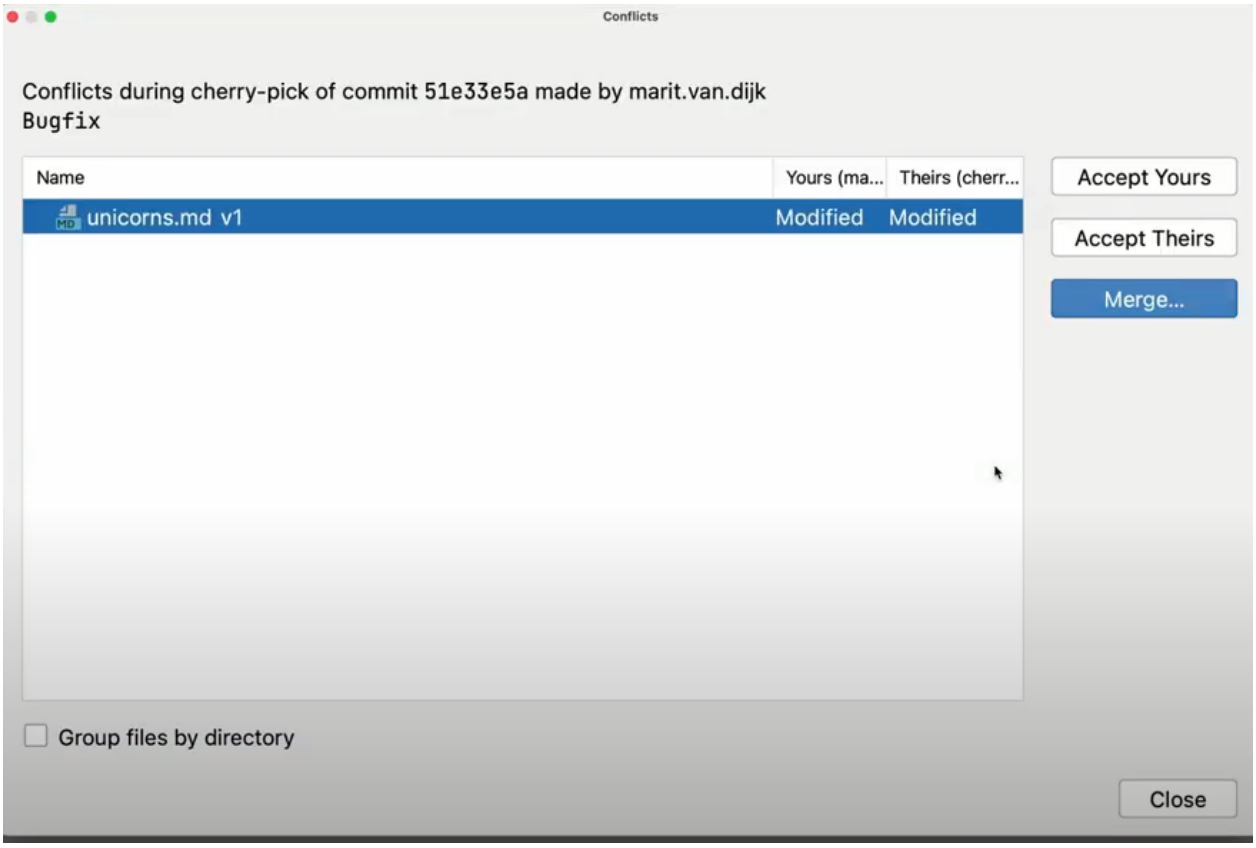
Cherry pick w inteliJ
Przykład:

https://www.youtube.com/watch?v=SkcvWURJkWQ

W naszym feature-branch zrobiliśmy naprawę fixbug, który nie powinien tutaj być.

Rozwiązanie: Stworzenie brancha FixBug i wykonanie cherry-pick z fixbug commitem z feature-branch.

W inteliJ w przypadku konfliktów wystąpi podany pop-out dialog:



Colaborations

https://www.youtube.com/watch?v=asVGkEAvK9A&t=255s

https://www.youtube.com/watch?v=jhtbhSpV5YA