Git - Remote

Spis treści

Repozytorium zdalne

Chciałbym, żebyśmy przeszli do pracy z repozytorium zdalnym.

Wspomniałem, że zaznaczę to wystarczająco mocno moment, w którym dołożymy do naszej pracy **repozytorium zdalne** - teraz jest ten moment.

Przypomnę, że dotychczas pracowaliśmy w większości z repozytorium lokalnym. Jedyna nasza styczność z repozytorium zdalnym była wtedy, gdy tworzyliśmy konto na **GitHub** i gdy wykonaliśmy `git clone`, żeby odtworzyć projekt z repozytorium zdalnego na naszym komputerze.

git clone

Przypomnijmy tylko, że `git clone` służyło do tego, żeby sklonować repozytorium zdalne na naszą maszynę - nasz komputer. Inaczej mówiąc, pobieramy w ten sposób stan repozytorium zdalnego w danym momencie. Jeszcze inaczej mówiąc, robiąc `git clone` odtwarzamy repozytorium zdalne na naszym komputerze - czyli wykonujemy jego kopię.

Dlaczego repozytorium zdalne jest istotne?

Wspomniana kopia żyje swoim własnym życiem. Możemy w tej lokalnej kopii dodawać commity, tworzyć branche, usuwać branche, scalać gałęzie i dopóki nie zatroszczymy się o to, żeby nasze działania zostały **wypchane** do repozytorium zdalnego, to repozytorium zdalne nie ma o tym zielonego pojęcia. Jeżeli pracujemy w zespole w kilka osób - to samo dotyczy się każdej osoby, która ma lokalną kopię repozytorium zdalnego. Każdy z członków takiego zespołu musi się synchronizować z takim repozytorium zdalnym po to, żeby:

- dodawać do repozytorium zdalnego zmiany wprowadzane do projektu u siebie lokalnie,
- mieć na swoim komputerze najnowsze zmiany, które ktoś inny dodał do repozytorium zdalnego, czyli żeby być na bieżąco ze zmianami w projekcie.

W praktyce wygląda to w ten sposób, że gdy pracujemy w zespole, to takie repozytorium zdalne pełni funkcję **jednego źródła prawdy**, jednego centralnego miejsca, z którym każdy deweloper synchronizuje swoje zmiany i pobiera z niego zmiany wprowadzane przez innych członków zespołu. O tym, jak to zrobić w **Git** będziemy rozmawiać za moment.

Zatem w ogromnym uproszczeniu, praca z repozytorium zdalnym będzie obejmowała między innymi:

- pobieranie najnowszych zmian z repozytorium zdalnego. Zmiany takie, w projekcie, nad którym pracujemy, są wprowadzane przez innych członków zespołu.
- wysyłanie naszych zmian do repozytorium zdalnego po to, żeby podzielić się nimi, z innymi członkami zespołu i żebyśmy mieli jedno miejsce, które jest źródłem naszej aplikacji.

Przypomnę, że **Git** jest skonstruowany w ten sposób, że każdy członek zespołu ma u siebie wykonaną dokładną kopię projektu z całą jego historią zmian. Natomiast repozytorium zdalne jest często traktowane jako źródło, z którego tworzy się później kolejne wydania aplikacji, które są udostępniane klientom końcowym.

Z wyżej wymienionych powodów, oprócz pracy z repozytorium lokalnym, musimy również umieć synchronizować naszą pracę z repozytorium zdalnym, czyli pobierać najnowsze zmiany i wysyłać nasze najnowsze zmiany.

Pobranie zmian do siebie na maszynę

Gdy wykonamy `git clone`, to pobierzemy najnowszy stan repozytorium. Komenda ta może być rozumiana jako komenda początkowa - czyli wykonujemy ją raz, gdy chcemy pobrać repozytorium po raz pierwszy do siebie na komputer. Później stosuje się komendy, o których porozmawiamy poniżej.

git fetch

Pobieranie zmian w **Git** jest o tyle ciekawe, że **Git** dzieli to koncepcyjnie na dwie czynności:

- pobranie zmian ze zdalnego repozytorium - to jest jedna czynność,
- połączenie pobranych zmian z repozytorium lokalnym - to jest druga czynność.

Czynności te są dwiema niezależnymi czynnościami. Gdy wykonamy polecenie `git fetch` - **Git** pobierze wszystkie zmiany z repozytorium zdalnego, których nie mamy jeszcze dodanych lokalnie, ale nie scali ich z naszym lokalnym repozytorium.

Ważne jest tutaj, żeby pamiętać, że **Git** nie synchronizuje niczego "na żywo". Patrząc na stan naszego repozytorium, tak naprawdę widzimy stan commitów z ostatniego momentu, gdy ręcznie synchronizowaliśmy nasze repozytorium lokalne z repozytorium zdalnym. Czyli informacje w naszym repozytorium są tak nowe, jak nowe były, gdy pobraliśmy ostatnio te informacje ręcznie. Oczywiście w naszym repozytorium lokalnym widzimy też commity, które dodaliśmy sami i które nie zostały jeszcze zsynchronizowane z repozytorium zdalnym. Żeby zobaczyć nowe informacje o zmianach, które zostały wykonane w repozytorium zdalnym, musisz zrobić to ręcznie przy wykorzystaniu komendy `git fetch`.

Można to zrobić na przynajmniej trzy sposoby:

```
git fetch ①
git fetch origin ②
git fetch origin <nazwa_brancha> ③
```

① Ta komenda pobiera informacje o wszystkich gałęziach i commitach ze zdalnego repozytorium.

② Ta komenda pobiera informacje o wszystkich gałęziach i commitach ze zdalnego repozytorium.

③ Ta komenda pobiera informacje o commitach z jednej konkretnej zdalnej gałęzi. Do zdalnych branchy jeszcze przejdziemy.

O co chodzi z tym **origin**? Jest to alias (czyli taki skrót rozwiązujący jakąś dłuższą nazwę, do aliasów jeszcze wrócimy), który pozwala nam określić zdalne repozytorium, do którego w danym momencie chcemy się odwołać. Jeżeli w naszym projekcie jest tylko jedno zdalne repozytorium - domyślnie możemy się do niego odwoływać przez słówko **origin**.

Gdy są dostępne nowe zmiany w repozytorium zdalnym i wykonamy komendę `git fetch`, to zostanie wydrukowane coś podobnego do:

```
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 1), reused 6 (delta 1), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/zajavka/git-workshop
   cb785f5..fe2ac20  master -> origin/master ①
```

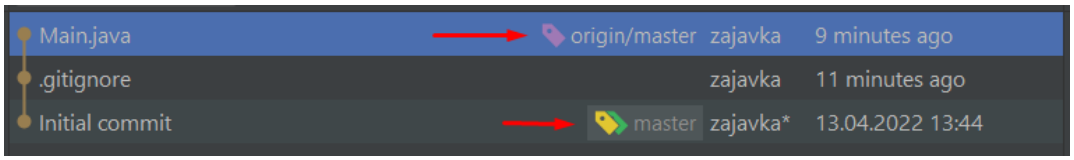
① **Git** w ten sposób informuje nas o tym, co zostało aktualizowane. Głównie interesuje nas ta linijka, bo ona nam mówi, który commit był najnowszy w naszym lokalnym repozytorium *cb785f5*, a który jest

najnowszy w zdalnym repozytorium *fe2ac20* na branchu **master**. Jednocześnie możesz zauważyć, że pojawia się tutaj **master** i **origin/master**. Pierwszy master od lewej oznacza nasz lokalny branch master, natomiast po prawej jego odpowiednik w repozytorium zdalnym. Do branchy zdalnych jeszcze przejdziemy.

Cały czas podkreślam, że `git fetch` tylko odświeżyło lokalne informacje o stanie repozytorium zdalnego, ale zmiany z repozytorium zdalnego nie zostały zaaplikowane do naszego lokalnego repozytorium. Jeżeli wykonasz teraz `git log`, to zwrócisz uwagę, że najnowsze zmiany ze zdalnego repozytorium nie są jeszcze widoczne:

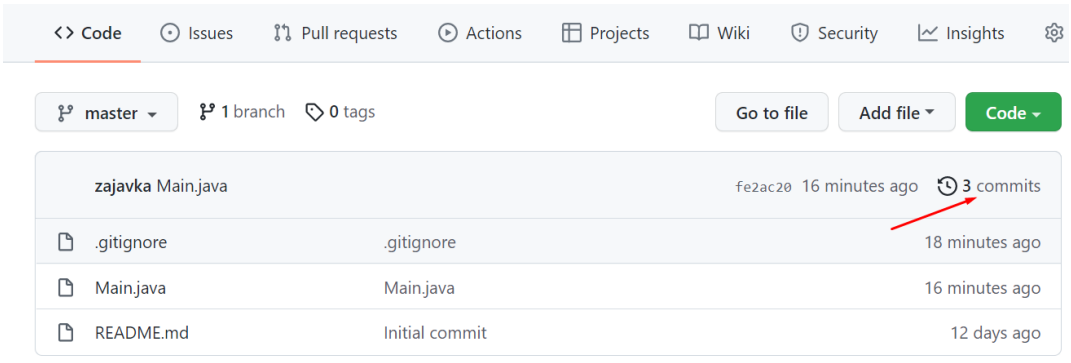
```
commit cb785f56fdbb4355b7a308c320eef5b8b2f0b4c4 (HEAD -> master)
Author: zajavka <103562129+zajavka@users.noreply.github.com>
Date:   Wed Apr 13 13:44:42 2022 +0200

    Initial commit
```

Zakładka **Git** > **Log** w Intellij pokazuje to w ten sposób:



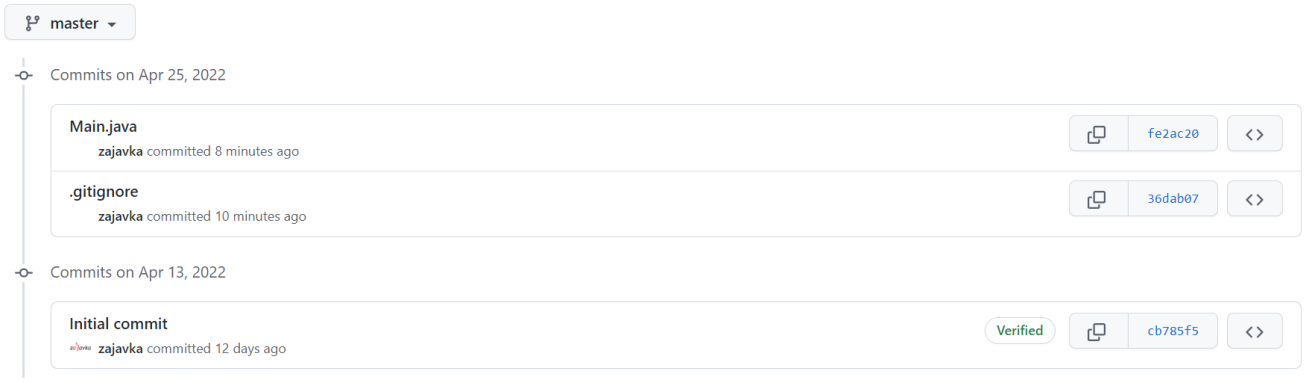
Obraz 1. Intellij Git Log

Strzałki pokazują dwa oznaczenia: **master** i **origin/master**. Commit, który jest oznaczony przez samo **master**, jest ostatnim commitem, który mamy dostępny lokalnie. Natomiast **commit**, który jest oznaczony przez **origin/master** jest ostatnim commitem, który jest dostępny na branchu zdalnym **master**. Widzimy już informację o nim, ale nie mamy go jeszcze zaaplikowanego w naszym lokalnym repozytorium. Widzimy, że jest, ale nie widzimy zmian, jakie ten commit wprowadził. Załóżmy, że ostatni commit (*Main.java*) wprowadza do repozytorium plik `Main.java`. Na tym etapie jeszcze tego pliku w repozytorium nie widać. Jeżeli teraz dla porównania spojrzymy na commity na branchu **master** w GitHub:



Obraz 2. GitHub master branch

To będzie to wyglądało tak:



Obraz 3. GitHub master branch

I chcę to zaznaczyć ponownie, po samym `git fetch`, zaktualizowaliśmy tylko informacje o repozytorium zdalnym, ale nie pobraliśmy jeszcze tych zmian do swojego repozytorium lokalnego. Czyli jeżeli ostatni commit w repozytorium zdalnym dodaje do projektu plik `Main.java`, to plik ten nie jest jeszcze u nas widoczny. Musimy wykonać następny krok.

git merge

Żeby scalić zmiany, które są dostępne w repozytorium zdalnym z naszym repozytorium lokalnym, musimy teraz wywołać **merge**. Możliwe też jest zrobienie **rebase**, ale o tym za moment. Wywołaj teraz taką komendę:

```
git merge origin/master
```

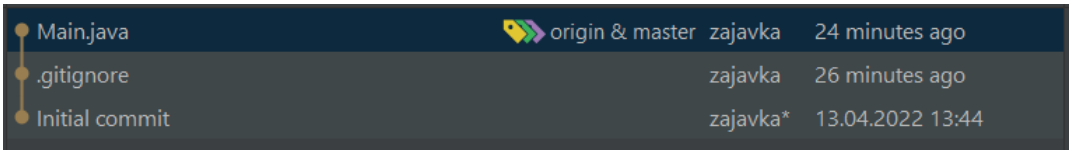
W przypadku idealnym, czyli takim, gdzie nie mamy żadnych **merge conflictów**, wszystko pójdzie gładko. Tzn., po wykonaniu powyższej komendy, na ekranie zostanie wydrukowane:

```
Updating cb785f5..fe2ac20
Fast-forward ①
 .gitignore | 3 +++
 Main.java  | 6 ++++++
 2 files changed, 9 insertions(+)
 create mode 100644 Main.java ②
```

① Czyli poszło gładko

② Dopiero teraz został dodany do repozytorium lokalnego wspominany wcześniej plik `Main.java`.

A w Intellij będzie to wyglądało teraz tak:



Obraz 4. Intellij Git Log

W powyższym przykładzie wszystko poszło gładko. Jeżeli natomiast będziemy mieli dodane jakieś commity na branchu **master**, zostanie wtedy dodany **merge commit**. Jeżeli wystąpią **merge conflicty** - będziemy musieli je rozwiązać. Historia zmian nie będzie wyglądała już wtedy tak przyjemnie jak wyżej,

ale do tego jeszcze przejdziemy.

git pull

Wymienione wyżej dwie czynności: `git fetch` i `git merge` można złączyć w jeden krok:

```
git pull
```

Domyślnie, komenda `git pull` wykonuje pod spodem `git fetch` i `git merge`. Możemy jednak wykonać tę komendę w taki sposób, żeby zamiast **merge** był wykonany **rebase**, do tego wrócimy później.

Należy pamiętać, że gdy wykonamy komendę `git pull`, to zostaną zaktualizowane zmiany na branchu, na którym się obecnie znajdujemy.

Jak często aktualizować stan repozytorium?

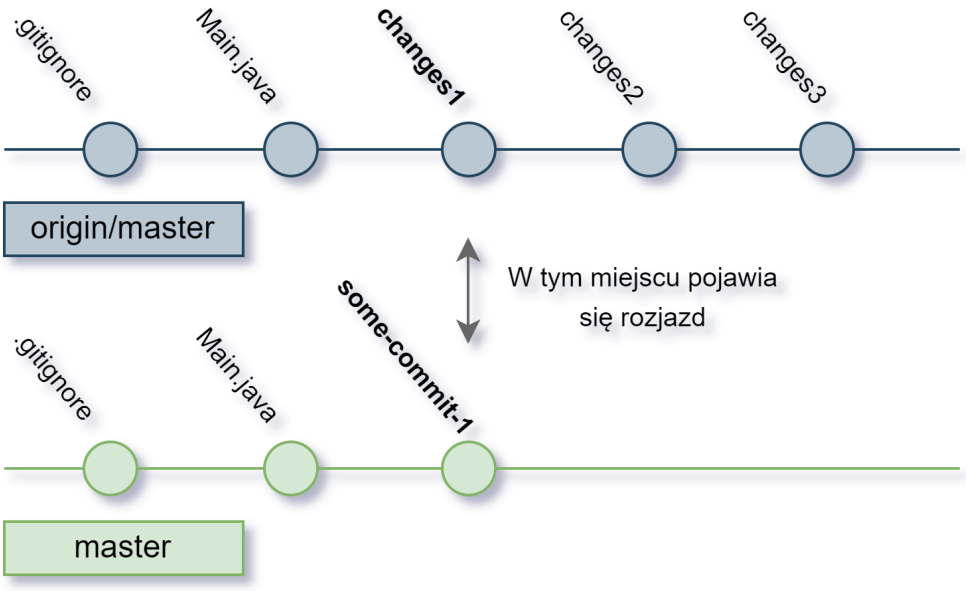
W praktyce, gdy pracujemy nad projektem w kilka osób, częstotliwość dokonywania zmian przez członków zespołu jest duża. Normalną sytuacją jest, że w trakcie kilku lub kilkunastu dni projekt może się zmienić diametralnie. Pamiętając, że lokalne repozytorium nie odświeża się na bieżąco, musimy pamiętać o tym, żeby aktualizować te zmiany manualnie. Najlepiej jest to robić zawsze przed rozpoczęciem pracy, jak również, gdy chcemy wypchać nasze zmiany do repozytorium zdalnego. Nie zaszkodzi nam również, jeżeli będziemy starali się na bieżąco utrzymywać stan zgodny z repozytorium zdalnym co kilka godzin. Celem takiego zabiegu jest zminimalizowanie prawdopodobieństwa wystąpienia merge conflictów.

Istnieją również pluginy do Intellij, które pozwolą na automatyczne wykonywanie `git fetch` np. co 15 minut, żebyśmy dostali powiadomienie, że są dostępne nowe zmiany w repozytorium zdalnym. Przykładem takiego pluginu może być: **GitToolBox**.

merge vs rebase

Czym będzie różniło się zastosowanie **merge** od **rebase** podczas wykonania komendy `git pull`? Przejdziemy przez przykład, który jeszcze na tym etapie będzie nieco teoretyczny, bo nie pokazaliśmy sobie jak wypchać nasze zmiany do repozytorium. Gdy już nauczysz się już jak publikować swoje zmiany, będziesz też w stanie odtworzyć ten przykład samodzielnie.

Załóżmy, że dodajesz kolejne commity do brancha **master** (na razie pomijamy to, czy to jest dobra, czy zła praktyka). W międzyczasie do tego samego brancha w repozytorium zdalnym ktoś dodał inne commity, czyli ktoś dodał commity do **origin/master**. Jeżeli na pewnym etapie wykonasz `git pull`, to możesz zdecydować, czy zmiany, które są dodane na **origin/master**, a których jeszcze nie ma u Ciebie, mają być wcielone do Twojego repozytorium przy wykorzystaniu **merge** (domyślne ustawienie), czy **rebase**. Sytuacja, o której rozmawiamy, mogłaby wyglądać w ten sposób:

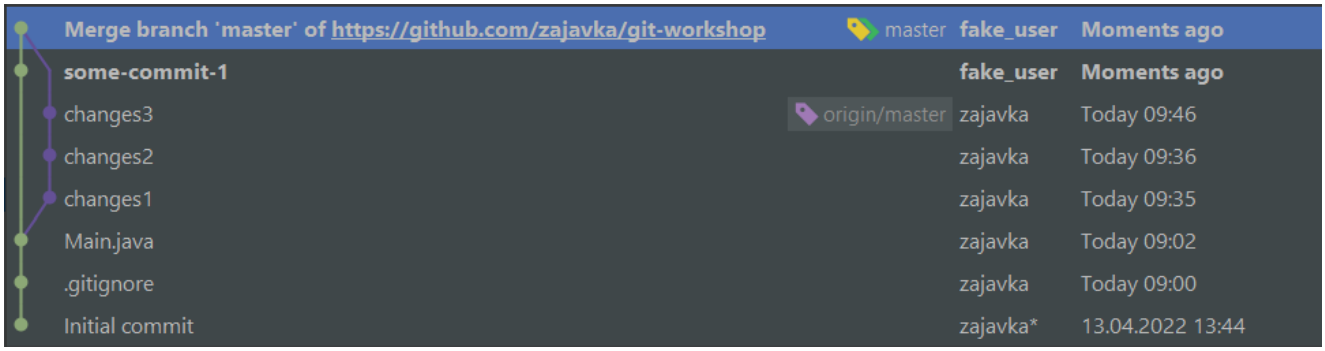


Obraz 5. Git pull merge vs rebase

Czyli w repozytorium zdalnym na branchu **origin/master** pojawiły się commity, natomiast w tym samym czasie dodaliśmy jeden commit w repozytorium lokalnym, na branchu **master**. Przypomnę, że temat branchy w repozytorium zdalnym omówimy bardziej dogłębnie.

Merge:

Jeżeli wykonamy teraz po prostu `git pull`, to historia naszego projektu będzie wyglądała jak poniżej. Czyli zbliżamy się do Guitar Hero. 😀

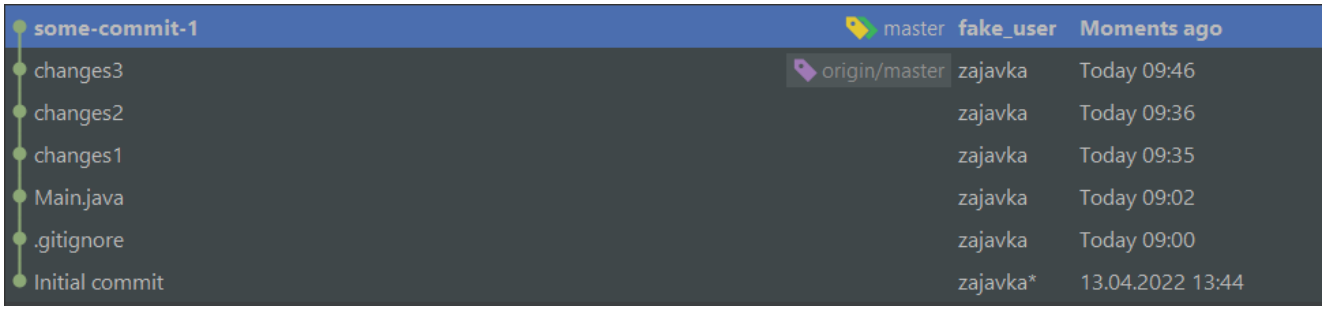


Obraz 6. Git pull merge vs rebase

Oczywiście ten przykład jest prosty, ale w ten sposób możemy zobrazować, co możemy zrobić ze swoją historią zmian, jeżeli będziemy pracowali na głównym branchu i "pullowali" zmiany robiąc merge.

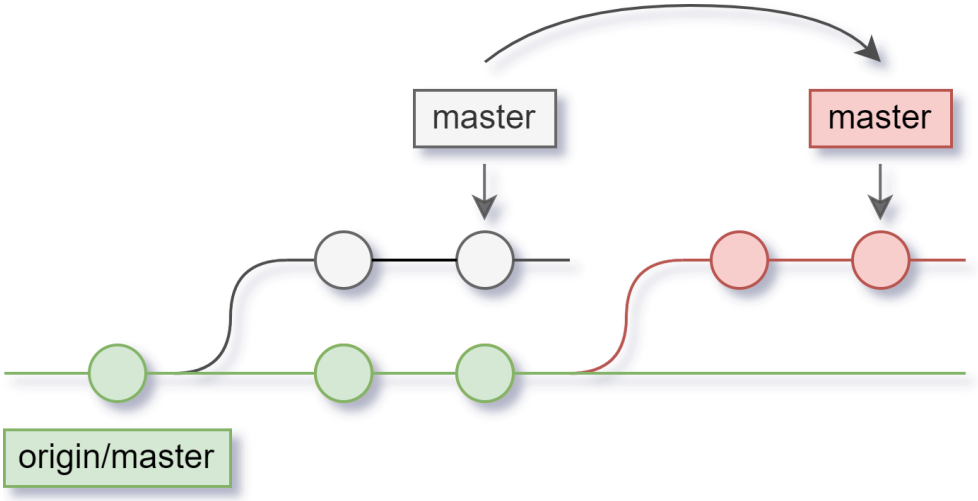
Rebase:

A jak zachowa się ten sam przypadek, gdy zrobimy **rebase**?



Obraz 7. Git pull merge vs rebase

Wynika to z tego, że stosując **rebase**, nasze lokalne "dodatkowe" zmiany na branchu **master** zostają przeniesione i zaczynają się teraz od końca brancha **origin/master**. Na obrazku wyglądałoby to tak:



Obraz 8. Git pull merge vs rebase

Intellij

W jaki sposób możemy pobrać zmiany z poziomu Intellij? Bardzo prosto.

W górnym pasku zakładek możemy wybrać **Git** i rozwinie nam się menu z opcjami. Mamy tutaj dostępne **Fetch** i **Pull**. Gdy wybierzemy opcję **Pull**, pojawi nam się okno, które pozwoli zdecydować, z jakiego zdalnego brancha chcemy pobrać zmiany i na jaki branch lokalny chcemy pobrać zmiany (o tym, jak branch lokalny mapuje się na zdalny będziemy jeszcze rozmawiać). Możemy również wybrać "Modify options" i pokaże nam się wtedy więcej możliwości.

Równie dobrze możemy wykorzystać skrót `Ctrl + T`, który wykonuje **Update Project**. Opcja ta pozwala nam wybrać, czy chcemy zrobić update przy wykorzystaniu **merge**, czy **rebase**. Gdy zerkniesz do konsoli **Git** > **Console**, zobaczysz, że Intellij wykonuje wtedy `git fetch` i `git merge` oddzielnie.

Wiele repozytoriów zdalnych

Chcę tutaj tylko nadmienić, że **Git** pozwala na zdefiniowanie wielu repozytoriów zdalnych.

Jeżeli chcemy sprawdzić, jakie mamy ustawienia repozytorium zdalnego, to możemy wykonać komendę:

```
git remote -v
```

Zostanie wtedy wydrukowana na ekranie informacja mówiąca, jaki jest adres repozytorium zdalnego, z którego pobieramy dane i jaki jest adres repozytorium zdalnego, do którego wysyłamy dane.

```
origin  https://github.com/zajavka/git-workshop.git (fetch)
origin  https://github.com/zajavka/git-workshop.git (push)
```

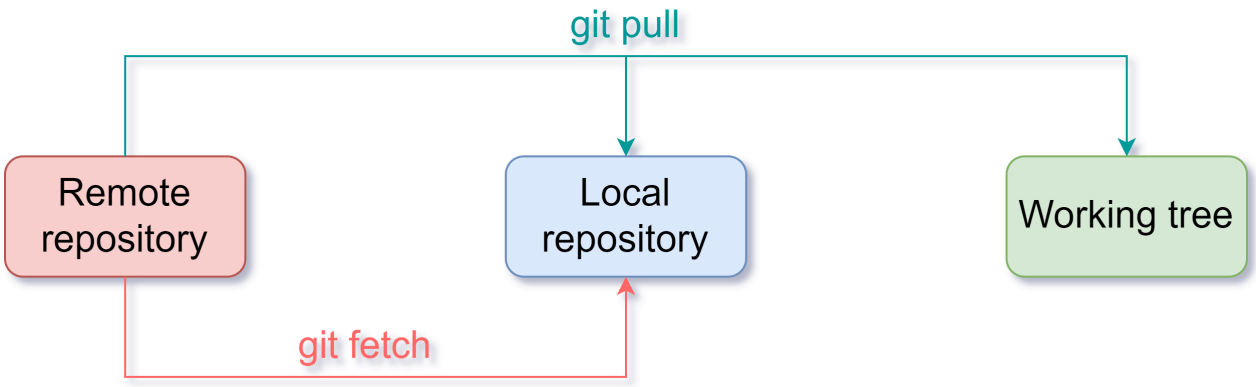
Zwróć uwagę, że pojawia się tutaj słówko **origin**, które było wspomniane wcześniej.

Te same ustawienia możemy sprawdzić w pliku *.git/config* w katalogu naszego projektu.

Nie chcę natomiast bardziej zagłębiać się w tę tematykę, gdyż wolę poświęcić ten czas na inne zagadnienia. Zapamiętaj natomiast gdzieś w głowie, że jest to możliwe, bo jak będzie Ci to potrzebne, to będziesz to zapewne Googlować. 😀

Podsumowanie

Podsumujmy `git fetch` kontra `git pull` poniższą grafiką:



Obraz 9. Fetch kontra Pull

Wysłanie zmian do repozytorium zdalnego

Przejdziemy teraz do omówienia, w jaki sposób możemy wysłać nasze zmiany do repozytorium zdalnego. Całe szczęście wiemy już, że za nasze repozytorium zdalne służy repozytorium, które stworzyliśmy na GitHub, także zostaje nam tylko wypchać zmiany i gotowe. Niestety nie będzie tak łatwo. Do wypchania zmian służy komenda:

```
git push
# lub
git push origin
# lub
git push origin master ①
```

① Na razie pracujemy cały czas na branchu zdalnym master, dlatego w tym przypadku wszystkie 3 komendy odniosą ten sam efekt. Niedługo przejdziemy do omówienia pracy na zdalnych branchach.

Wykonanie tej komendy spowoduje, że commity, które są dodane u nas lokalnie, zostaną "wypchane" do repozytorium zdalnego. Czyli w ten sposób synchronizujemy zmiany na naszym lokalnym repozytorium

z repozytorium zdalnym. Jeżeli natomiast spróbujesz wykonać tę komendę, to bardzo prawdopodobne jest, że zostanie w pierwszej kolejności, pojawi Ci się okienko do wpisania loginu i hasła do GitHub. Nawet jeżeli wpiszesz to poprawnie, to prawdopodobnie **Git** będzie kazał Ci ponownie wpisać to samo z poziomu konsoli. Jeżeli z poziomu konsoli wpiszesz dane poprawnie, to zostanie Ci wyświetlony taki komunikat:

```
Logon failed, use ctrl+c to cancel basic credential prompt.
...
remote: Support for password authentication was removed on August 13, 2021.
        Please use a personal access token instead.
remote: Please see https://github.blog/2020-12-15-token-authentication-requirements-for-git-operations/
        for more information.
fatal: Authentication failed for 'https://github.com/zajavka/git-workshop.git/'
```

Rozwiązaniem jest wygenerowanie **tokenu**, czyli takiego klucza, który umożliwi nam wysyłanie zmian do repozytorium. Opis tego, w jaki sposób wygenerować taki klucz i co z nim dalej zrobić znajdziesz tutaj.

Gdy się już z tym uporaliśmy, możemy spróbować wykonać ponownie komendę `git push`, na ekranie zostanie wtedy wydrukowany rezultat podobny do:

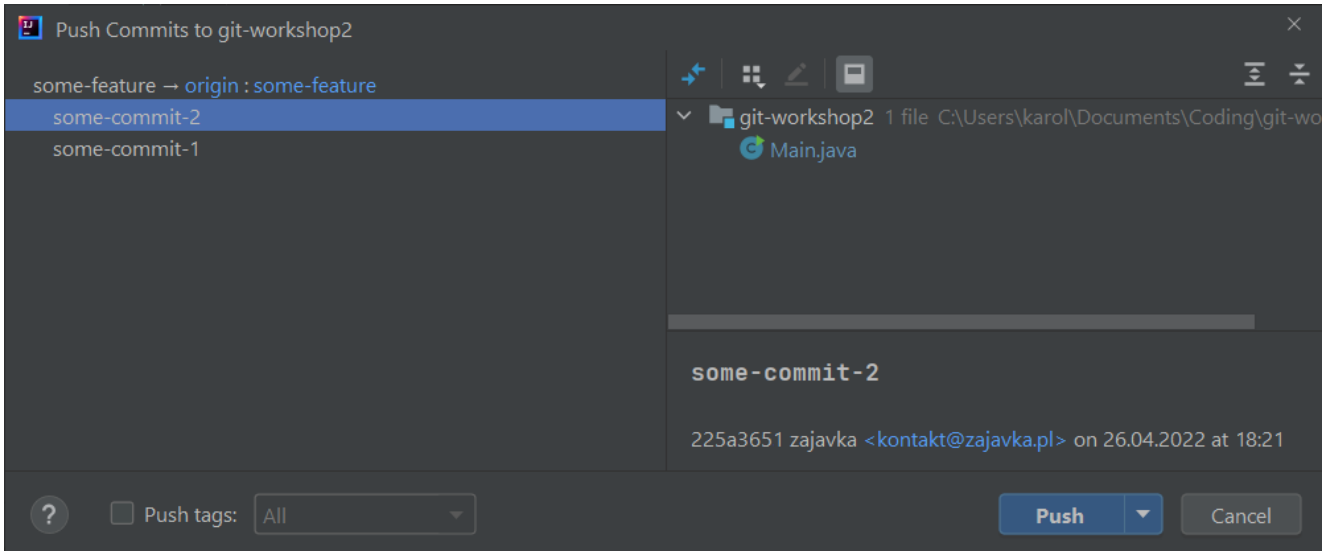
```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 313 bytes | 156.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/zajavka/git-workshop.git
   cb785f5..36dab07  master -> master ①
```

① Można tutaj zaobserwować hash commita (po lewej), który był ostatnim commitem w repozytorium zdalnym, oraz hash commita, który aktualnie publikujemy (po prawej).

Pokazany przypadek jest jednym z prostszych. Na razie poruszamy się w obrębie brancha **master** i publikujemy zmiany również do brancha **origin/master**. W praktyce taka sytuacja będzie miała najprawdopodobniej miejsce, gdy będziesz pracować nad jakimś projektem samodzielnie. Wtedy najczęściej nikomu się nie chce bawić w branche i wszystko jest wysyłane bezpośrednio do **origin/master**. Gdy już natomiast zaczniesz pracować z zespołem, to wypychanie zmian bezpośrednio do zdalnej gałęzi głównej nie jest dobrą praktyką. O tym porozmawiamy później.

Intellij

Zmiany możemy wypychać również z poziomu Intellij. Najprostszym sposobem na zrobienie `git push` z poziomu Intellij jest użycie skrótu `Ctrl + Shift + K`. Otworzy nam się wtedy okienko, w którym zostanie pokazane, że wypychamy np. do **origin** i np. do brancha **master**, który jest remote. Niedługo dowiesz się jak łączyć branche lokalne i branche zdalne. Widok ten jest o tyle przyjazny, że możemy jeszcze raz zobaczyć, jakie commity wypychamy i jakie zmiany zostały wprowadzone w każdym z nich.



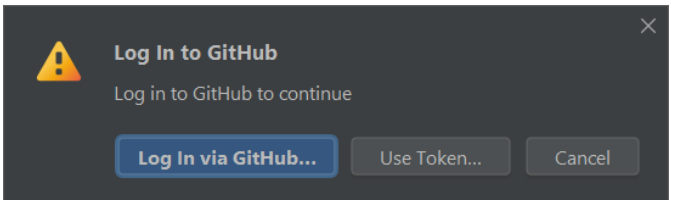
Obraz 10. Git Intellij Push

To samo okienko możemy otworzyć z górnego paska i opcji **Git**. Przy napisie **Push** będzie widoczna ikonka z zieloną strzałką, jest ona również widoczna w prawym górnym rogu ekranu.

Żeby zrobić push, możemy również użyć skrótu `Alt + `` i potem wybrać opcję **8**.

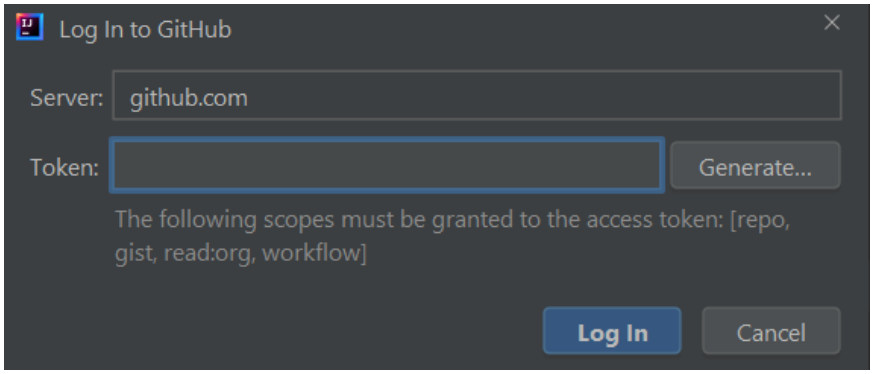
Kolejną możliwą opcją jest wybranie brancha w prawym dolnym rogu ekranu i tam też jest dostępna opcja **Push**. Możemy również przejść do zakładki **Git** > **Log**, kliknąć prawym przyciskiem na nazwę brancha i tam również będzie możliwość zrobienia **Push**. Do wyboru do koloru. 😀

Gdy spróbujemy zrobić **push** po raz pierwszy, pojawi nam się ten sam problem dotyczący uwierzytelnienia, co wcześniej. Musimy powiedzieć, że my to my. Wcześniej z poziomu terminala ten problem został rozwiązany inaczej - w notatkach był podany link do Stackoverflow. W tym przypadku możemy wybrać, żeby w ogarnięciu wszystkiego pomógł nam Intellij. Jeżeli Intellij nie jest w stanie sam wywnioskować, czy jesteśmy uwierzytelnieni, pokaże nam takie okienko:



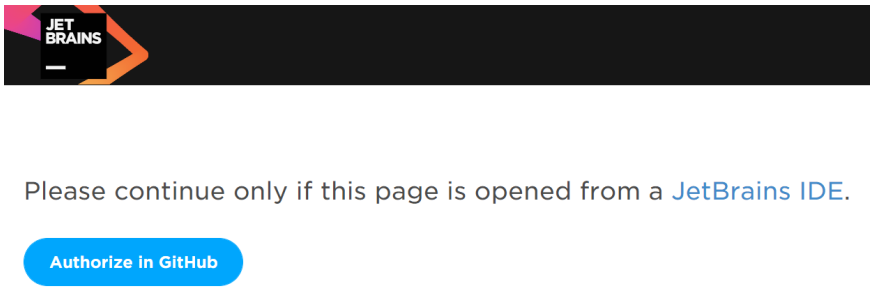
Obraz 11. Git Intellij Push

Możemy teraz wybrać opcję **Use Token** i wykorzystać token, który wygenerowaliśmy wcześniej, wklejając go tutaj:



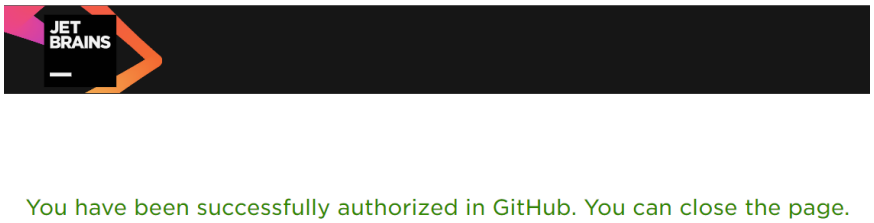
Obraz 12. Git Intellij Push

Możemy również wybrać opcję **Log In via GitHub**, otworzy nam się wtedy taka strona internetowa:



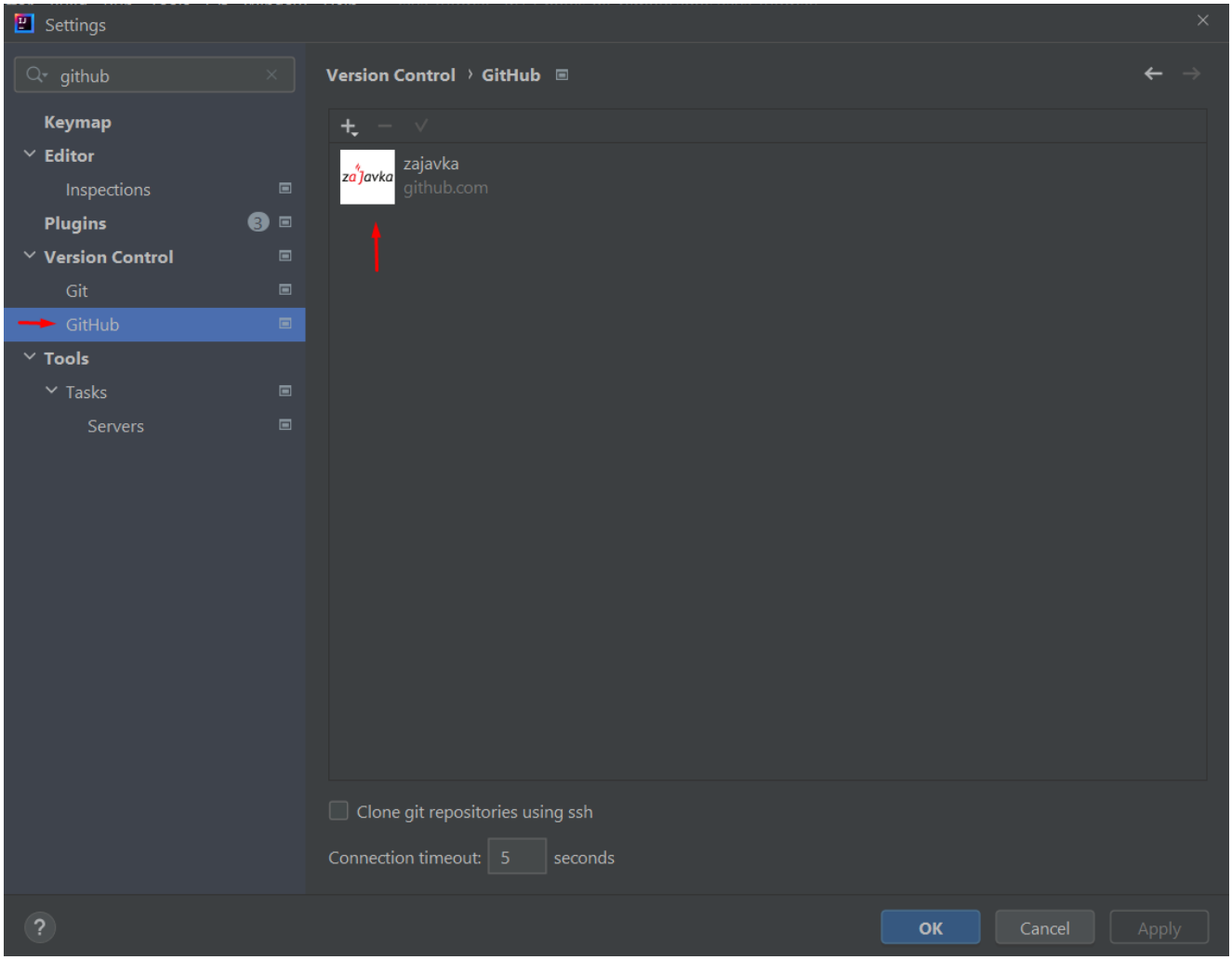
Obraz 13. Git Intellij Push

W moim przypadku, gdzie jestem zalogowany w tej przeglądarce do **GitHub**, jak wybrałem przycisk **Authorize in GitHub** - wszystko dalej zadziało się automatycznie i zobaczyłem taki ekran:



Obraz 14. Git Intellij Push

W Intellij, w ustawieniach `Ctrl + Alt + S` zostaje wtedy dodany taki wpis:



Obraz 15. Git Intellij Push

Jeżeli usuniemy ten wpis, możemy bawić się w ten sam proces ponownie. 😀

Praca ze zdalnymi branchami

No to teraz możemy skomplikować sprawę jeszcze bardziej. 😀

Dotychczas rozmawialiśmy o tym, że mamy ten poziom abstrakcji w postaci branchy. Pojawiały się stwierdzenia o branchach lokalnych, o branchach zdalnych i przyszedł moment, gdzie skupimy się na wyjaśnieniu, o co w tym chodzi.

Na tym etapie powinniśmy już dosyć dobrze rozumieć koncepcję brancha w rozumieniu takim, że jak zmieniamy branch, to zmieniamy stan plików, adekwatnie do stanu na danym branchu. Czyli jak przełączasz branche, to możesz to rozumieć, jakbyś wymieniał/-a cały swój workspace i skojarzone z nim pliki. Branche pozwalają nam pracować nad wieloma funkcjonalnościami jednocześnie i możemy się pomiędzy tymi funkcjonalnościami przełączać. Tyle wiemy mniej więcej na tym etapie, jeżeli chodzi o pracę z branchami lokalnymi.

Do tego dochodzi kolejny poziom skomplikowania w postaci branchy zdalnych. Jeżeli pomyślimy o repozytorium zdalnym jak o kopii repozytorium takiej jak nasza lokalna kopia, to na tym poziomie powinno wydawać się naturalne, że tam też mogą być branche tak samo, jak możemy tworzyć branche lokalnie. Komplikacja polega na tym, że **Git** musi wiedzieć, jak złączone są nasze branche lokalne, z tymi zdalnymi. Ubierając to w inne słowa, który branch zdalny jest sparowany z naszym lokalnym.

Gdy już wyobrazimy to sobie w ten sposób, to łatwiej jest zrozumieć, że nasz branch **master** może pracować w parze z branchem **origin/master**, a nasz branch lokalny **feature-new-1** może pracować w parze z branchem zdalnym **origin/feature-new-1**. Musimy jedynie powiedzieć o tym **Git**owi.

Wyświetlenie listy branchy

Wcześniej, gdy rozmawialiśmy o branchach, pokazana została komenda, która pokazywała, jakie mamy stworzone branche:

```
git branch ①
git branch -r ②
git branch -a ③
```

① Pokaż branche lokalne

② Pokaż tylko branche zdalne

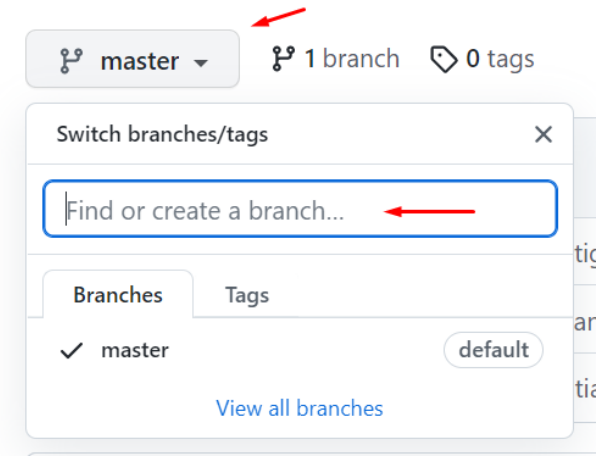
③ Pokaż wszystkie branche, lokalne i zdalne

Utworzenie brancha zdalnego

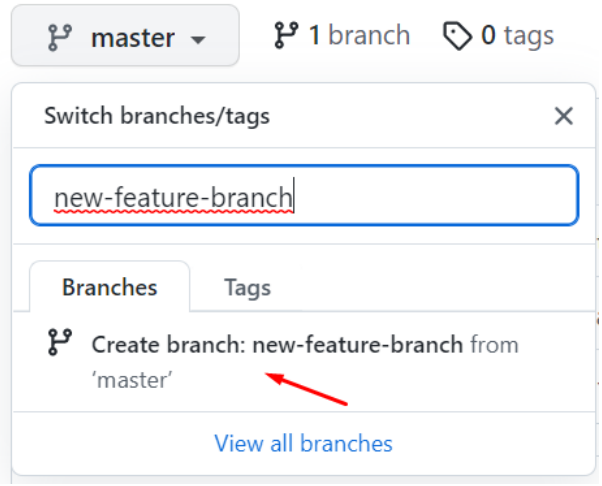
Jak już pewnie zdążyłeś/-aś zauważyć, w **Git** dostępnych jest wiele możliwości, żeby osiągnąć ten sam cel. Tak samo jest, gdy chcemy stworzyć branch zdalny. Pokazanych zostanie kilka sposobów.

Utworzenie brancha na GitHub

Jedną z możliwości jest utworzenie brancha bezpośrednio na GitHub:



Obraz 16. GitHub create branch 1



Obraz 17. GitHub create branch 2

Należy pamiętać, że taki branch zostanie utworzony na podstawie stanu głównego brancha zdalnego. Następnie możemy pobrać informacje o istnieniu takiego brancha, przy wykorzystaniu np. `git fetch`:

```
From https://github.com/zajavka/git-workshop
 * [new branch]      new-feature-branch -> origin/new-feature-branch
```

Jeżeli wykonamy teraz:

```
git checkout new-feature-branch
```

To **Git** wydrukuje:

```
Switched to a new branch 'new-feature-branch'
Branch 'new-feature-branch' set up to track remote branch 'new-feature-branch' from 'origin'.
```

Informacja ta mówi nam, że przełączyliśmy się na nowy branch i nasz lokalny branch **new-feature-branch** został "sparowany" ze zdalnym branchem **origin/new-feature-branch**.

Możemy teraz dodać parę commitów do tego brancha i wypchać zmiany na branch **origin/new-feature-branch**:

```
# Dodaj kilka zmian w plikach
git commit -am "commit-1"
# Dodaj kilka zmian w plikach
git commit -am "commit-2"
# Dodaj kilka zmian w plikach
git commit -am "commit-3"
git push
```

Rezultat:

```
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
```

```
Delta compression using up to 8 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 783 bytes | 783.00 KiB/s, done.
Total 7 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/zajavka/git-workshop.git
   bd1859e..918bc57  new-feature-branch -> new-feature-branch
```

Dzięki temu, że wcześniej nasz branch lokalny **new-feature-branch** został skojarzony z branchem zdalnym **new-feature-branch**, to wystarczy, że napiszemy `git push`. Jeżeli przejdziesz teraz na GitHub i klikniesz przycisk **commits**, to pokaże Ci się widok pokazujący wszystkie commity na nowym branchu. Możesz również zmienić w tym widoku branch zdalny, z którego pokazywane są commity. Co dalej zrobić z takim branchem, powiemy sobie później. 😀

Możemy również taki branch usunąć z poziomu GitHub. Gdy przejdziemy do widoku branchy, będzie tam dostępna ikonka kosza. Z tego samego poziomu możemy też zmienić nazwę brancha.

Najpierw tworzymy branch lokalnie

Stwórzmy branch lokalnie i przełączmy się na niego:

```
git branch feature-branch-from-local
git checkout feature-branch-from-local
```

Możemy teraz spróbować dodać parę commitów do tego brancha i wypchać zmiany na **origin**:

```
# Dodaj kilka zmian w plikach
git commit -am "commit-1"
# Dodaj kilka zmian w plikach
git commit -am "commit-2"
# Dodaj kilka zmian w plikach
git commit -am "commit-3"
git push
```

Na ekranie zostanie wydrukowane:

```
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
```

Informacja ta mówi nam, że mamy stworzony branch lokalny, ale nie powiązaliśmy go z żadnym branchem zdalnym. Możemy teraz to rozwiązać tak, jak **Git** nam podpowiada:

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

Gdy wykonamy tę komendę, na ekranie zostanie wydrukowane:

```
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 8 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 942 bytes | 942.00 KiB/s, done.
Total 9 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 1 local object.
remote:
remote: Create a pull request for 'feature-branch-from-local' on GitHub by visiting: ①
remote:      https://github.com/zajavka/git-workshop/pull/new/feature-branch-from-local
remote:
To https://github.com/zajavka/git-workshop.git
 * [new branch]      feature-branch-from-local -> feature-branch-from-local
Branch 'feature-branch-from-local' set up to track remote branch
        'feature-branch-from-local' from 'origin'. ②
```

① Do tego jeszcze przejdziemy

② Informacja ta mówi nam, że branch lokalny **feature-branch-from-local** śledzi gałąź zdalną **feature-branch-from-local**. Od tego momentu, gdy będziemy wypychać zmiany z gałęzi **feature-branch-from-local**, to będą one automatycznie wypychane do gałęzi zdalnej **feature-branch-from-local**.

Checkout brancha zdalnego

Gdy mamy już stworzony zdalny branch, możemy podejść do pracy z nim jeszcze inaczej. Możemy zastosować komendy:

```
git fetch
git switch -c some-feature origin/some-feature ①
# Dodajemy zmiany
git comit -am "some-new-feature"
git push
```

① Tworzymy i jednocześnie przełączamy się na branch **some-feature**, ustawiając jednocześnie jaki branch zdalny ma być śledzony.

Zdalny branch a git status

Umiesz już na tym etapie dodawać zmiany do branchy zdalnych. Sklonuj sobie zatem to samo repozytorium w innym miejscu na dysku, po to, żeby mieć dwie kopie tego samego repozytorium na swoim komputerze. Możesz w ten sposób imitować pracę kilku użytkowników nad jednym projektem. W tym drugim repozytorium dodaj kilka commitów do brancha **origin/master**, wróć do pierwszego repozytorium i wykonaj:

```
git fetch
git status
```

Na ekranie zostanie wtedy wydrukowane:

```
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
```

```
  (use "git pull" to update your local branch)
```

Git informuje nas w ten sposób, że jesteśmy jeden commit do tyłu w stosunku do **origin/master**. Jeżeli natomiast przejdziesz teraz do tego drugiego repozytorium i zaczniesz dodawać tam commity, ale nie wypchasz ich do **origin/master** i wykonasz `git status`, to rezultat będzie dawał taką wiadomość:

```
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)
```

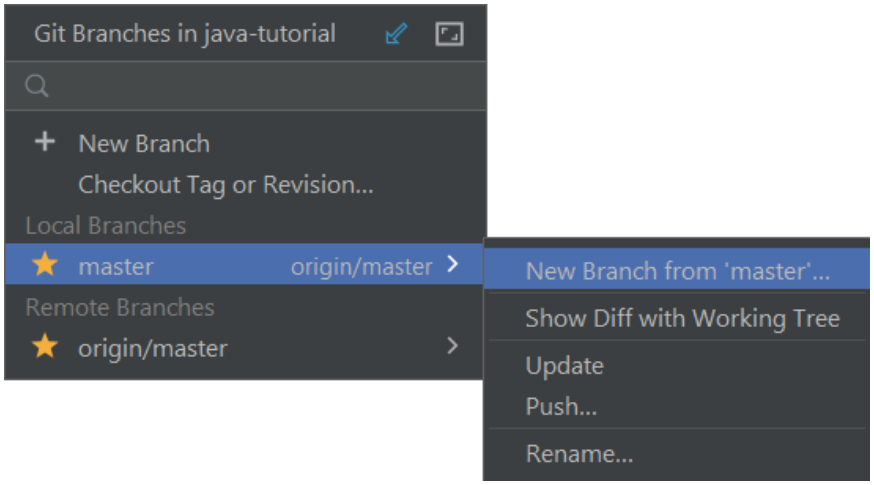
Czyli jesteśmy z naszym lokalnym branchem **master** o dwa commity dalej niż **origin/master**.

Zdalny branch a Intellij

Standardowo, praca z Intellij będzie wyglądała lekko inaczej w przypadku gdy pracujemy ze zdalnymi branchami, niż gdybyśmy robili to tylko z konsoli. Przede wszystkim musimy pamiętać o tym, że listę branchy można zobaczyć w wielu miejscach w Intellij:

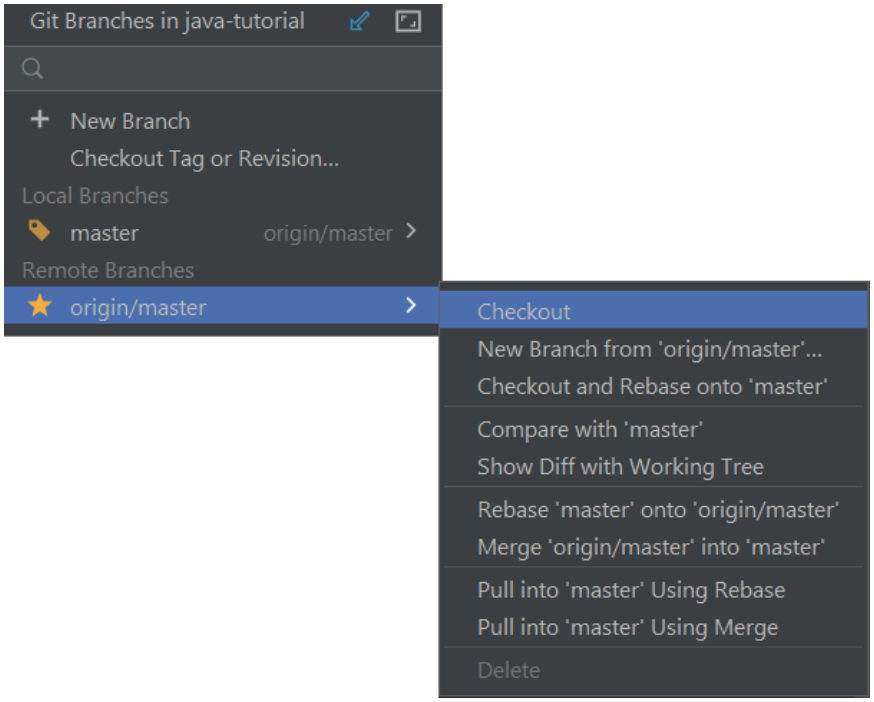
- w prawym dolnym rogu ekranu można kilknąć w nazwę brancha i pokaże nam się okienko z listą branchy,
- możemy wybrać skrót `Alt + `` i wybrać opcję 7 - wtedy również pokaże nam się okienko z listą branchy,
- możemy również przejść do zakładki **Git** > **Log** i tam też widzimy listę branchy

Zauważ, że lista branchy jest zawsze podzielona na branche lokalne i branche zdalne. W jednym i w drugim przypadku mamy dostępne różne opcje. Poniżej opcje dla lokalnego brancha:



Obraz 18. Intellij Git Local Branches

Oraz opcje dla zdalnego brancha:



Obraz 19. Intellij Git remote Branches

Wiele z tych opcji zostało już omówionych wcześniej. Te same opcje są dostępne, gdy klikniemy prawym przyciskiem myszki na branch w zakładce **Git** > **Log**. Najlepszym ćwiczeniem jednak będzie, gdy pobawisz się tymi możliwościami samodzielnie i zobaczysz jaki efekt to daje w zakładce **Git** > **Console**.

Zdalny branch a git revert

Pamiętasz `git revert`? To był ten sposób wycofania zmian, który skutkował dodaniem commita, a ten miał za zadanie kompensować/wycofywać zmiany wprowadzone wcześniej.

Ten sposób wycofania zmian nie modyfikował istniejącej historii, tylko dodawał nowe wpisy. W pracy ze zdalnym repozytorium to podejście nie będzie prowadziło do powstawania dziwnych błędów, bo **revert commit**, jest zmianą jak każda inna, różni się jednak tym, że wycofuje zmiany wprowadzone wcześniej. Taki commit dodajemy do zdalnego repozytorium, następnie inni członkowie zespołu pobierają ten wpis w historii i wszystko jest dalej w porządku. Można się kłócić, że ten sposób wycofania zmian "zaśmieca" nam historię commitów, ale przynajmniej to podejście jest bezpieczne, bo nie modyfikujemy historii wstecz.

Ciekawostką jest to, że można zrobić *revert reverta*, czyli cofnąć cofnięcie. Wtedy przywrócimy zmiany, które wycofaliśmy. 😀

Zdalny branch a git reset

Skoro wcześniej zostało wspomniane, że `git revert` jest bezpieczny, to jak w takim razie obchodzić się z `git reset` przy pracy z repozytorium zdalnym? W tym przypadku mamy kilka możliwości wykorzystania tej komendy w zależności od naszych potrzeb i wtedy trzeba zwracać uwagę na kilka istotnych kwestii.

Wycofanie zmian do ostatniego commita z repozytorium zdalnego

Gdy pracujemy już z repozytorium zdalnym, możemy je traktować jak taki twardy punkt kontrolny. Mówiąc to, mam na myśli, że jeżeli jakieś zmiany znajdą się już na gałęzi głównej w zdalnym repozytorium - możemy je traktować jak zapisane na stałe w historii projektu. Historia takich zmian nie powinna być już modyfikowana wstecznie. Dlatego właśnie piszę, że to, co już jest na gałęzi głównej w zdalnym repozytorium, może być traktowane jak odnośnik, na którym opieramy swoją pracę.

Należy pamiętać, że stosując `git reset`, pozbywamy się commitów. Oznacza to, że modyfikujemy historię zmian na danym branchu wstecznie. Wiedząc o tym oraz dokładając wiedzę dotyczącą traktowania repozytorium zdalnego jak odnośnik - możemy spróbować to wykorzystać.

Jak to wykorzystać? Jeżeli pobierzemy najnowsze zmiany z repozytorium zdalnego i zaczniemy dodawać swoje commity, a w pewnym momencie dojdziemy do wniosku, że jednak wszystkie commity, które dodaliśmy na tym branchu, nie są nam potrzebne i chcemy się tego kompletnie pozbyć - możemy wtedy zresetować branch do jego stanu w repozytorium zdalnym.

Trzeba jednak z tym uważać, bo operacja ta kompletnie wyrzuci zmiany, nad którymi pracowaliśmy, bo zresetujemy stan brancha do stanu brancha w repozytorium zdalnym.

Komenda, która to zrobi to:

```
git reset --hard origin/<nazwa_brancha>
```

Ponownie to podkreślę - wykonanie tego polecenia zresetuje stan naszego brancha do stanu, który jest ostatnio zapamiętany z repozytorium zdalnego, czyli wyrzucimy wszystkie nasze commity, które są na tym branchu lokalnym nowe.

Modyfikacja historii i synchronizacja z repozytorium zdalnym

Jeżeli chodzi o pracę z branchami, to możemy podzielić sobie branche (oczywiście w głowie) na takie, których używa wiele osób i na takie, z których będziemy korzystać tylko my. Jest to o tyle istotne, że z takiego brancha **master** korzysta cały nasz zespół i nie możemy takiego brancha popsuć. Jeżeli natomiast dodamy nasz własny branch **my-feature-branch**, czy to lokalny, czy zdalny i będziemy mieli świadomość, że tylko my na nim pracujemy, to dopóki nie skończymy na nim pracy - możemy z nim robić co mamy ochotę. Pod warunkiem, że używamy go tylko my!

Można powiedzieć jeszcze bardziej ogólnie, że jeżeli wypychamy jakiekolwiek zmiany do zdalnego repozytorium, to jest wtedy szansa, że każdy członek zespołu może potencjalnie pobrać te zmiany do siebie zacząć z tymi zmianami pracować. Dlatego właśnie powinniśmy zawsze mieć włączoną lampkę z tyłu głowy, że jeżeli wypychamy jakieś zmiany do zdalnego repozytorium, każdy może je pobrać i zacząć z tym pracować. Oczywiście można tak się umówić w zespole, że jak ktoś chce zacząć pracować na innym branchu niż **master**, to pyta autora brancha, czy może - kwestia dogadania.

Dlaczego jest to takie istotne? Branch **master** jest branchem głównym i na nim historii wstecz modyfikować nie możemy. Jest to w interesie wszystkich członków zespołu, żeby był jeden rejestr zmian, którego nie da się zmodyfikować wstecz. Jeżeli taki rejestr zostałby zmodyfikowany - każdy z członków zespołu musiałby rozwiązywać problemy z historią indywidualnie na swoim komputerze. Możesz nawet

spróbować wykonać teraz takie ćwiczenie, przejdź do swojego lokalnego brancha **master**, wykonaj na nim parę razy polecenie:

```
git reset --soft HEAD^
```

A następnie spróbuj wykonać `git push`. Na ekranie zostanie wydrukowana taka informacja:

```
To https://github.com/zajavka/git-workshop.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'https://github.com/zajavka/git-workshop.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Nie możemy wypchać takiego brancha do repozytorium zdalnego, bo wycofaliśmy commity, czyli zmodyfikowaliśmy historię wstecz. **Git** informuje nas, że nie możemy tego zrobić, bo końcówka naszego brancha lokalnego, jest **za** końcówką brancha zdalnego. Możemy teraz wyprostować swoje lokalne repozytorium, wykorzystując komendę:

```
git reset --hard origin/<nazwa_brancha>
```

Dlatego właśnie przy pracy z repozytorium zdalnym, o wiele bezpieczniejszą komendą na wycofanie zmian jest `git revert` - nie modyfikujemy wtedy historii wstecz.

Jest jednak sytuacja, gdzie wykonanie `git reset` może nam się przydać, wtedy jednak trzeba podejść do tego lekko inaczej. Należy przy tym pamiętać, żeby nie robić tego na branchu **master**.

Wyobraź sobie, że pracujesz nad jakimiś zmianami **na swoim własnym branchu** - nie na **master**. Udało Ci się już wypuścić kilka commitów na odpowiednik Twojego brancha w repozytorium zdalnym. Dochodzisz jednak do wniosku, że ostatnie 2 z 5 wykonanych commitów chcesz cofnąć i chcesz, żeby to samo stało się w repozytorium zdalnym. Czyli chcesz zmodyfikować historię brancha (ale Twojego własnego - nie rób tak na **master**). Jak to zrobić?

```
git reset --soft HEAD~2
```

No dobrze, ale przecież cofnęliśmy właśnie historię naszego własnego brancha. Jeżeli teraz robimy `git push`, to repozytorium zdalne odrzuci nasze zmiany:

```
To https://github.com/zajavka/git-workshop.git
 ! [rejected]        my-feature-branch -> my-feature-branch (non-fast-forward)
error: failed to push some refs to 'https://github.com/zajavka/git-workshop.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Co w takim razie zrobić? Wymusić wypchanie naszych zmian do repozytorium zdalnego. Pamiętaj tylko, że takie wymuszanie powinno się robić z Twoimi własnymi branchami, a nie z **master**. Z resztą przejdziemy jeszcze do tego, że bezpośrednie wypychanie zmian do **master** podczas pracy w zespole to zła praktyka. Żeby wymusić teraz wypchanie tych zmian do **origin**, możesz wykonać taką komendę:

```
git push --force
```

Git wydrukuje wtedy na ekranie:

```
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/zajavka/git-workshop.git
 + 33c1b95...1a506f7 my-feature-branch -> my-feature-branch (forced update)
```

Należy jednak pamiętać, że `--force` wymusza nadpisanie stanu brancha zdalnego, stanem naszego brancha lokalnego. Czyli w naszym przykładzie, gdzie cofnęliśmy 2 commity z 5 - zostały one bezpowrotnie usunięte ze zdalnego brancha, bo nadpisaliśmy ten branch naszym branchem lokalnym, na którym zostały tylko 3 commity. W powyższym przykładzie zrobiliśmy `git reset --soft`, więc wycofane zmiany zostały przeniesione do **staged**, natomiast gdybyśmy zrobili `--hard` to stracilibyśmy je bezpowrotnie.

Jeżeli pojawia się w Twojej głowie pytanie: *Co potem zrobić z tymi moimi zmianami na moim zdalnym branchu?* - spokojnie, jeszcze do tego przejdziemy.

Force push

Dowiedzieliśmy się już, że można wymusić nadpisanie brancha zdalnego naszym branchem lokalnym. Ponownie, pamiętaj, żeby nie robić tego z głównymi gałęziami, pozwalamy na takie operacje tylko na Twoich własnych branchach, które zostały stworzone do Twojej pracy nad jakąś funkcjonalnością. 😁

Kiedy w takim razie przyda nam się wymuszenie nadpisania brancha zdalnego naszą lokalną wersją? Za każdym razem, gdy zmodyfikujemy historię brancha wstecznie.

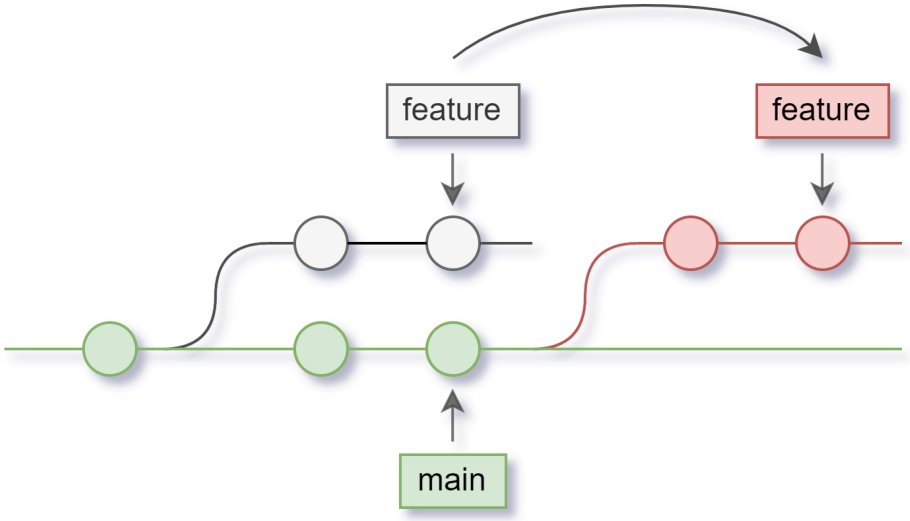
git reset

Dowiedzieliśmy się już wcześniej, że jeżeli chcemy zrobić `git reset`, czyli cofnąć jakieś commity, a potem wypchać te zmiany do repozytorium zdalnego - będziemy potrzebowali zastosować **force push**.

git rebase

Pamiętasz, że **rebase** jest operacją zmieniającą historię? W jakim przypadku w takim razie będziemy potrzebowali zastosować **force push**, jeżeli będziemy wcześniej stosowali **rebase**. Wyobraź sobie, że pracujesz na swoim własnym branchu, powstały już na nim 3 commity i zostały one już wypchane do repozytorium zdalnego. W międzyczasie do głównej gałęzi (**master**), inni członkowie zespołu dodali 2 commity. Dobrze jest aktualizować naszą własną gałąź do najnowszego stanu gałęzi **master** i wiesz już, że możesz to zrobić stosując **merge** lub **rebase** - wybierasz zatem **rebase**. Ale chwila! Przecież **rebase** zmienia historię brancha (tutaj rebasujemy *my-feature_branch* na *master*), skoro zmienia historię brancha, a ten branch ma już wypchaną jakąś historię do repozytorium zdalnego, to przecież będzie problem. Właśnie w takiej sytuacji trzeba użyć **force push**.

A skąd tutaj ta modyfikacja historii brancha? Pomoże w tym poniższa grafika:



Obraz 20. Git rebase visualisation

Zwróć uwagę, że używam branchy **main** i **master** wymiennie. Ciągle natomiast reprezentują one gałąź główną.

Nasz branch **feature** był stworzony od pewnego commita na branchu **main**. Gdy do brancha **main** zostały dodane nowe commity - nasz branch **feature** stał się przestarzały. Przenosimy go zatem na nowy początek - na nowe commity w branchu **main**. Pamiętasz, że jak wykonujemy **rebase** to tak naprawdę **Git** stworzy nowe commity na wzór tych poprzednich? To jest właśnie ta modyfikacja historii. Skoro zmodyfikowaliśmy istniejącą historię, będziemy musieli nadpisać historię na branchu zdalnym tą historią z naszego brancha lokalnego. W tym celu zastosujemy:

```
git push --force
```

git commit --amend

Kolejną operacją, którą już poznaliśmy, a która zmienia istniejącą historię jest `git commit --amend`. Flaga `--amend` służyła do modyfikacji istniejącego commita. Skoro modyfikujemy commit - oznacza to, że modyfikujemy historię wstecz. W takiej sytuacji musimy wymusić nadpisanie historii brancha zdalnego historią naszego lokalnego brancha.