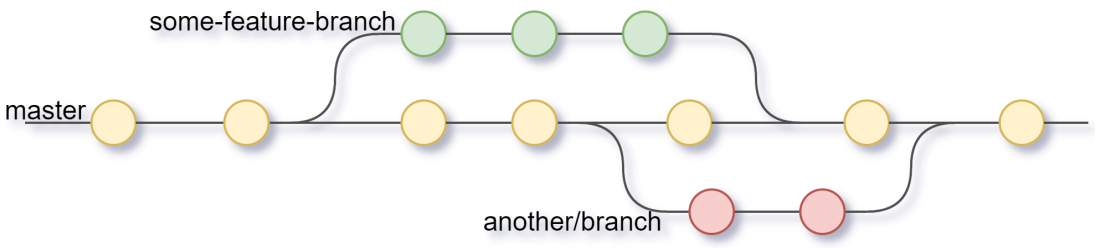
Git - Merge

Spis treści

Merge

Merge - czyli połączenie/scalenie. Jak sama nazwa mówi, **merge** służy do tego, żeby coś złączyć. W tym przypadku złączenie takie będzie się odnosiło do branchy, czyli gałęzi.



Obraz 1. Git merging visualisation

Na powyższej grafice możesz zauważyć, że mamy dwa rozgałęzienia, które finalnie zostają scalone z gałęzią główną. Właśnie to scalenie jest nazywane **merge**m. Czyli inaczej mówiąc, będziemy używać słowa **merge** w sytuacji, gdy będziemy chcieli złączyć ze sobą branche, w tym celu, żeby commity, które zostały dodane na dedykowanym branchu, znalazły się w gałęzi głównej. Należy tutaj znaznaczyć, że mergować ze sobą można różne branche, nie musimy mergować od razu do **master**. Możemy mergować dowolne branche z dowolnymi branchami.

Przypomnę, że najczęstsze nazwy gałęzi głównej to **main** lub **master**. Wcześniej używaliśmy gałęzi **main**, ale żeby nie utrwaliła się ona za mocno, teraz będę mówił o nazwie **master**. Pamiętaj, że jest to tylko nazwa gałęzi i przyjęło się, że gałąź główna nazywa się **main** lub **master**.

Jak najlepiej podejść do merge?

Krok 1: Aktualizacja feature brancha

Upewnij się, że masz u siebie na komputerze najnowszy stan Twojego feature brancha.

Mówiąc **feature branch** najczęściej mamy na myśli branch, który nie jest **master** i na którym dodawaliśmy jakieś funkcjonalności (*ficzery*).

Jeszcze o tym nie rozmawialiśmy i poruszymy ten temat, gdy przejdziemy do omawiania pracy z repozytorium zdalnym, ale branche mają swoją reprezentację nie tylko u nas lokalnie. Tworząc branch na naszym komputerze, możemy też stworzyć branch w repozytorium zdalnym. Dlatego o tym wspominam, bo zanim wykonamy merge, dobrze jest upewnić się, że nasz branch lokalny jest aktualny w stosunku do brancha zdalnego, który mu odpowiada. Należy o tym pamiętać, by uniknąć potencjalnych **konfliktów**, o których powiemy zaraz.

Krok 2: Przełączenie

Przełącz się na branch **DO KTÓREGO** chcesz wcielić swoje zmiany. Czyli jeżeli pracowałeś/-aś na branchu **my-feature-branch** i dodałeś/-aś tam kilka commitów i chcesz je teraz wcielić do brancha **master**, to musisz się przełączyć na gałąź **master**. Możesz do tego wykorzystać polecenie `git checkout` lub `git switch`.

Krok 3: Aktualizacja brancha master

Ponownie, jeszcze nie rozmawialiśmy o aktualizacji gałęzi w stosunku do jej stanu w repozytorium zdalnym, ale pamiętaj, że teraz musielibyśmy wykonać ten krok. Jak to zrobić dowiesz się już niedługo. Po przełączeniu się na gałąź **master** (albo jakąś inną, do której chcesz wcielić swoje zmiany), upewnij się, że jest ona aktualna.

Krok 4: Faktyczny merge

Możemy teraz wykonać faktyczne scalenie - czyli **merge**. Będąc na gałęzi **master**, możesz wykonać polecenie:

```
git merge my-feature-branch
```

Polecenie to oznacza, że zmiany, które zostały dodane na branchu **my-feature-branch** mają zostać wcielone do brancha, na którym się znajdujesz - czyli w tym przypadku **master**.

Przykład bez konfliktów

Będąc na branchu **master** dodaj do niego kilka commitów, np. w ten sposób, żeby `git log` wyglądał analogicznie do tego:

Branch **my-feature-branch**

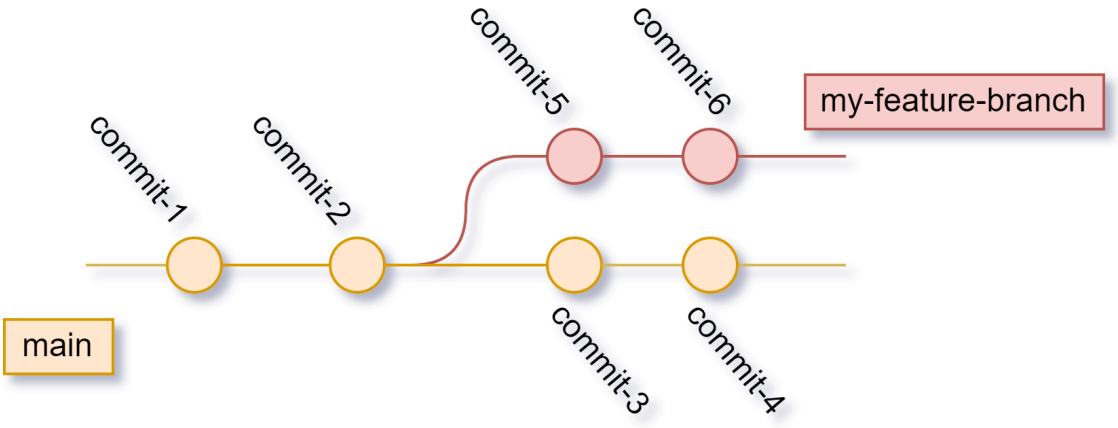
```
commit fbfe1813d71df55fda87b32673fd5d7efef11d9f (HEAD -> my-feature-branch)
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:43:12 2022 +0200

    commit-6

commit 1d52c9e3f6b9d0b0bd60de8658b85dcaf971ca4a
Author: fake_user <fakeemail@gmail.com>
```

```
Date:   Thu Apr 21 15:43:05 2022 +0200

    commit-5

commit c65cd057f028758a5a207af05c1e97c2e911eec5
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:40:35 2022 +0200

    commit-2

commit 9f318b31cbb7f7ffaa2fb1629ee1424e28c7c978
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:40:21 2022 +0200

    commit-1
```

Branch **master**

```
commit 85b7e60e157f051c185c0b8540d3e46e5ddccedb (HEAD -> master)
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:41:46 2022 +0200

    commit-4

commit 588770e3a2ced8d94fed8b3132f241df52c5ed7e
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:41:34 2022 +0200

    commit-3

commit c65cd057f028758a5a207af05c1e97c2e911eec5
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:40:35 2022 +0200

    commit-2

commit 9f318b31cbb7f7ffaa2fb1629ee1424e28c7c978
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:40:21 2022 +0200

    commit-1
```

Zwróć uwagę, że commity **commit-1** i **commit-2** są dostępne na obu branchach. Commity **commit-3** i **commit-4** są dodane tylko na branchu **master**, natomiast commity **commit-5** i **commit-6** są dostępne tylko na branchu **my-feature-branch**. Jeżeli teraz zaczniesz się przełączać między tymi branchami, to zobaczysz, że raz widzisz zmiany wprowadzone tylko na branchu **master**, a raz tylko na branchu **my-feature-branch**. Wizualizacja tej sytuacji mogłaby wyglądać tak:



Obraz 2. Git branches before merge

Jeżeli wykonamy teraz kroki rozpisane wcześniej, to doprowadzimy do takiej sytuacji:

Rezultat wywołania `git log`

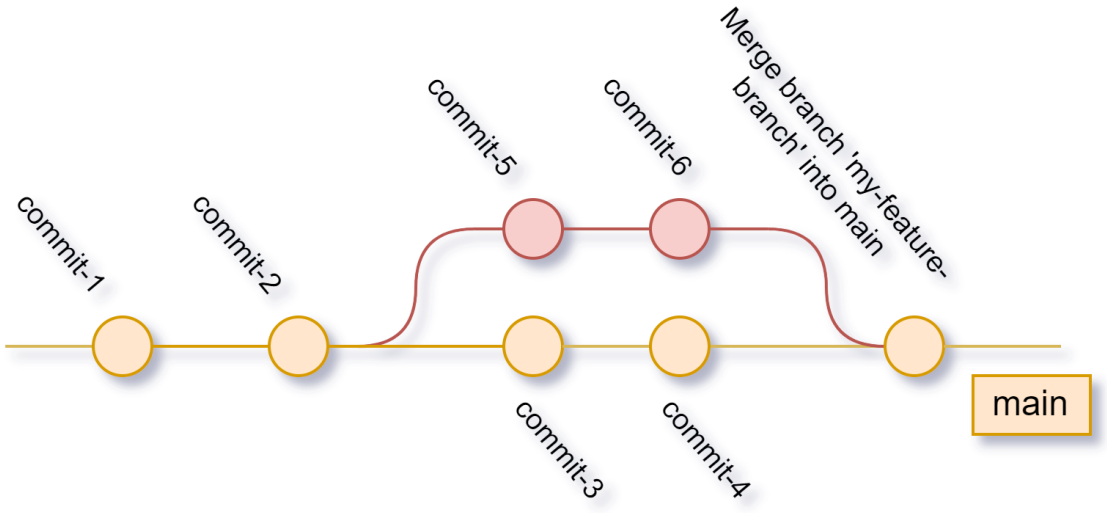
```
commit a455c272eb37536f2445e614dac72421ae9c9c92 (HEAD -> master)
Merge: 85b7e60 fbfe181
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:53:59 2022 +0200

    Merge branch 'my-feature-branch' into master

commit fbfe1813d71df55fda87b32673fd5d7efef11d9f (my-feature-branch)
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:43:12 2022 +0200

    commit-6

commit 1d52c9e3f6b9d0b0bd60de8658b85dcaf971ca4a
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:43:05 2022 +0200

    commit-5

commit 85b7e60e157f051c185c0b8540d3e46e5ddccedb
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:41:46 2022 +0200

    commit-4

commit 588770e3a2ced8d94fed8b3132f241df52c5ed7e
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:41:34 2022 +0200

    commit-3
```

Rezultat wywołania `git log` *cd.*

```
commit c65cd057f028758a5a207af05c1e97c2e911eec5
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:40:35 2022 +0200

    commit-2
```

```
commit 9f318b31cbb7f7ffaa2fb1629ee1424e28c7c978
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 15:40:21 2022 +0200

    commit-1
```

Zauważ, że do brancha **master** został dodany tzw. **merge commit**. Jest to commit, który został wygenerowany automatycznie przez **Git**. Commit ten reprezentuje moment, w którym nastąpił **merge**. Wizualizacja wyglądałaby tak:



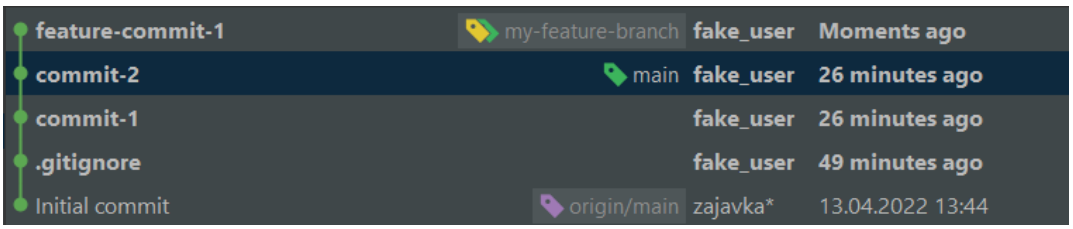
Obraz 3. Git branches after merge

Ostatni commit po prawej stronie to automatycznie wygenerowany **merge commit**.

Przykład bez merge commit

W poprzednim przykładzie został dodany **merge commit**, ze względu na to, że podczas naszej pracy na branchu **my-feature-branch** zostały dodane commity do brancha **master**. Jeżeli natomiast nasz branch **my-feature-branch** dodałby zmiany i zmergował je na tyle szybko, że na branchu **master** nie powstałyby żadne nowe zmiany, to sytuacja wyglądałaby inaczej.

Doprowadź projekt do takiego stanu:



Obraz 4. Git merge example

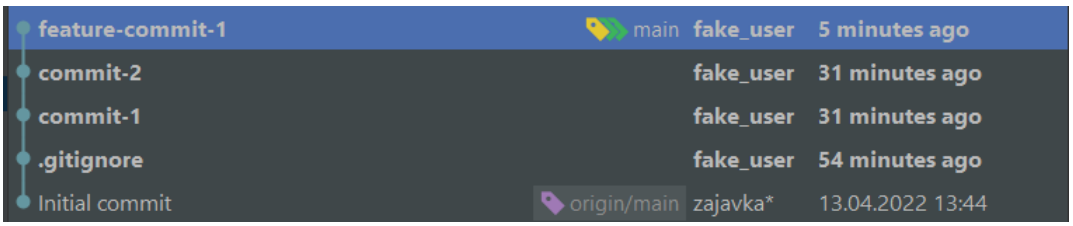
Czyli na branchu **my-feature-branch** masz dodany jeden commit **feature-commit-1**, natomiast od momentu powstania brancha **my-feature-branch**, na branchu **master** nie doszły żadne nowe zmiany. Jeżeli przełączysz się teraz na branch **master** i wykonasz poniższe polecenie:

```
git merge my-feature-branch
```

To na ekranie zostanie wydrukowana wiadomość:

```
Updating aa24fec..d8b9810
Fast-forward
 Dog.java | 2 ++
 1 file changed, 2 insertions(+)
```

Najważniejsze w tej wiadomości jest stwierdzenie **fast-forward**. Oznacza ono, że nie został dodany **merge commit**. Jeżeli spojrzysz teraz do historii, to będzie ona wyglądała tak:



Obraz 5. Git merge example

Fast-forward

Fast-forward merge może wystąpić, gdy mamy perfekcyjnie linearną historię na branchu, który mergujemy. Czyli inaczej mówiąc, gdy branch, który mergujemy jest **up2date** - na bieżąco z informacjami z brancha, do którego mergujemy. W takim przypadku **Git** nie musi pod spodem faktycznie mergować tych zmian i dodawać **merge commita**, tylko może "przesunąć" (*fast-forward*) końcówkę obecnego brancha do końcówki gałęzi docelowej (*target branch*). Taki zabieg jest w stanie skutecznie złączyć historię branchy, bo cała historia z brancha docelowego jest dostępna w feature branchu.

W powyższym przypadku nie został dodany **merge commit**, bo branch **my-feature-branch** był na bieżąco z branchem **master**

Jeżeli natomiast ktoś chce wiedzieć, w którym momencie w historii zmian był robiony merge, można wymusić w takiej sytuacji dodanie **merge commita**, dodając odpowiednią flagę do polecenia `git merge`:

```
git merge --no-ff my-feature-branch
```

Przykład z konfliktami

Powyższy przykład był prosty - obyło się bez **konfliktów**. W rzeczywistości nie jest tak kolorowo. Na czym polega konflikt?

Zacznijmy od tego, że na branchu **master** mamy klasę `Dog.java`, która jest pusta, czyli wygląda tak:

Klasa `Dog.java` *na branchu* **master**

```
1:  package pl.zajavka;
```

```
2:
3:  class Dog {
4:  }
```

Wyobraź sobie, że utworzyłeś/-aś branch **my-feature-branch** i modyfikujesz na nim klasę `Dog.java`. Przed modyfikacją klasa jest pusta, nie ma żadnych pól. W linijkach od 4 do 6 dodajesz pola jak poniżej:

Klasa `Dog.java` *na branchu* **my-feature-branch**

```
1:  package pl.zajavka;
2:
3:  class Dog {
4:      private String name;
5:      private Integer age;
6:      private String color;
7:  }
```

Dodajesz commit na branchu **my-feature-branch** i przełączasz się na branch **master**. Na branchu **master** zaczynasz pracować w tej samej klasie. Przypomnę, że na ten moment na branchu **master** klasa `Dog.java` jest pusta. Dodajemy do niej zatem takie zmiany:

Klasa `Dog.java` *na branchu* **master**

```
1:  package pl.zajavka;
2:
3:  class Dog {
4:      private Long id;
5:      private String nick;
6:
7:      void bark() {
8:          System.out.println("Bark!");
9:      }
10: }
```

Następnie dodajemy commit do brancha **master** z powyższymi zmianami. Spróbujmy wykonać teraz polecenie:

```
git merge my-feature-branch
```

Co się teraz stanie? **Git** wykrywa, że na dwóch branchach został zmodyfikowany ten sam plik w tym samym miejscu (te samej linijki). **Git** nie jest w stanie i pewnie nawet nie chce decydować za nas, jaki ma być finalny wygląd tego pliku. Skoro **Git** nie chce i nie umie nawet podjąć takiej decyzji jak finalnie ma wyglądać ten plik - zostanie nam zgłoszony **konflikt**. Jeżeli wykonamy powyższe polecenie `git merge`, to na ekranie zobaczymy coś takiego:

```
Auto-merging Dog.java
CONFLICT (content): Merge conflict in Dog.java
Automatic merge failed; fix conflicts and then commit the result.
```

Pamiętasz, że jak robiliśmy `git revert` to również doprowadziliśmy do sytuacji

> konfliktowej, tyle, że tam finalnie zrobiliśmy `--abort`? Przypominam o tej sytuacji, bo wspomniany konflikt moglibyśmy rozwiązać analogicznie do tego jak zostanie to pokazane poniżej.

Co możemy teraz zrobić?

Porzucamy

Możemy porzucić taki **merge** albo inaczej mówiąc, poddać się i wrócić do sytuacji sprzed wykonania **merge**. W takiej sytuacji branche wrócą do poprzedniego stanu, a my będziemy mogli spróbować rozwiązać tę sytuację inaczej, albo podejść do wykonania **merge** ponownie. Aby anulować proces scalania, możemy wykonać komendę:

```
git merge --abort
```

Rozwiązujemy

Możemy również rozwiązać takie **konflikty** i doprowadzić **merge** do końca. Gdy wystąpi **konflikt**, spróbuj wykonać polecenie `git status`, na ekranie pojawi się taki fragment:

```
You have unmerged paths.
  (fix conflicts and run "git commit") ①
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   Dog.java
```

① Tutaj **Git** nas informuje co powinniśmy zrobić - rozwiązać konflikty. 😀 Dziękujemy kapitanie oczywisty.

Jeżeli otworzysz teraz plik `Dog.java`, to zobaczysz znaczniki, które dodał **Git**, żeby poinformować nas, z którymi linijkami nie potrafi sobie poradzić. W ten sposób dowiadujemy się, gdzie jest konflikt. Klasa `Dog.java` będzie teraz wyglądała w ten sposób:

```
package pl.zajavka;

public class Dog {
<<<<<<< HEAD ①
    private Long id;
    private String nick;

    void bark() {
        System.out.println("Bark!");
    }
======= ②
    private String name;
    private Integer age;
    private String color;
>>>>>>> my-feature-branch ③
}
```

① W tej linijce **Git** nam mówi, że stąd rozpoczynają się zmiany z pliku `Dog.java` na branchu, gdzie obecnie się znajdujemy, stąd **HEAD** (przypomnę, że jesteśmy na branchu **master**).

② Ten znacznik oznacza, że doszliśmy do końca konfliktujących zmian w pliku na obecnym branchu i przechodzimy do zmian w pliku z brancha, który chcemy zmergować.

③ Ten znacznik mówi nam o końcu zmian, które są konfliktujące. Dostajemy również informacje, o tym, który branch mergujemy.

Zauważ, że reszta pliku `Dog.java` pozostała nieruszona. **Git** ingeruje tylko w tych fragmentach pliku, w których występują konflikty.

Gdy już wiemy, gdzie podczas mergowania wystąpiły konflikty, możemy tam wejść i ręcznie taki plik poprawić. Czyli musimy teraz zdecydować, która wersja pliku jest poprawna i ręcznie taki plik poprawić. Gdy już to zrobimy, musimy na skonfliktowanych plikach wykonać komendę `git add` - dajemy w ten sposób znać **Git**owi, że rozwiązaliśmy konflikty. Następnie musimy wykonać `git commit` - dzięki temu zostanie wygenerowany **merge commit** analogicznie jak poprzednio. Wykonajmy teraz zatem polecenia:

```
git add Dog.java
git commit
```

Otworzy nam się domyślny edytor tekstowy, z domyślną wiadomością **merge commita**. Jeżeli zamkniemy teraz ten edytor, to zostanie dodany **merge commit**, a finalnie zostaną uwzględnione zmiany, które sami wyprostowaliśmy ręcznie.

Wykonajmy jeszcze jedną komendę:

```
git show
```

Na ekranie zostanie wtedy wydrukowany rezultat:

```
Merge: a1cc808 a0be914 ①
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 18:11:47 2022 +0200

    Merge branch 'my-feature-branch' into master ②

diff --cc Dog.java ③
index 332f661,3322e20..45e8d8d
--- a/Dog.java
+++ b/Dog.java
@@@ -1,8 -1,5 +1,7 @@@
  public class Dog {
 +    private Long id;
 +    private String nick;
-
-     void bark() {
-         System.out.println("Bark!");
-     }
+     private String name;
+     private Integer age;
+     private String color;
```

```
}
```

① Tutaj mamy informacje o hashach commitów, które zostały zmergowane, po lewej jest hash commita z brancha **master**, po prawej z brancha **my-feature-branch**.

② Tutaj mamy domyślną wiadomość, która została nadana **merge commitowi**.

③ Tutaj widzimy rezultat **diff**a, gdzie **Git** pokazuje nam jakich zmian dokonaliśmy ręcznie podczas **merge**.

Cofamy merge

Po wykonaniu **merge** może się okazać, że jednak gdzieś popełniliśmy błąd i chcemy cofnąć wykonany **merge**. Możemy wtedy taki **merge** wycofać.

Przypomnij sobie, że podczas **merge** powstaje **merge commit**. **Merge commit** jest specyficznym rodzajem commita. Różni się on tym, że można go rozumieć jak commit spinający ze sobą dwa branche. Jak spinacz, który trzyma dwa sznurki. Jeżeli mergujemy gałąź **my-feature-branch** do gałęzi **master** to **merge commit** jest tworzony na gałęzi **master** i spina on wtedy dwie głowy, **HEAD** brancha **master** i **HEAD** brancha **my-feature-branch**.

Dzięki takiemu podejściu możemy to spięcie cofnąć. Podczas wycofywania takiego spięcia musimy wskazać które zmiany mają pozostać na obecnym branchu. Mamy tutaj dwie możliwości:

```
git revert -m 1 4d57bad ①
git revert -m 2 4d57bad ②
```

① Ten zapis spowoduje, że pozostaną zmiany, które były wprowadzone na branchu **master**.

② Ten zapis spowoduje, że pozostaną zmiany, które były wprowadzone na branchu **my-feature-branch**.

Wykonanie komendy `git revert` spowoduje dodanie **revert commita** z finalnym kształtem zmian zależnym od tego, czy w poleceniu wpisaliśmy `1`, czy `2`. Taki **revert commit** będzie kolejną migawką w repozytorium, która będzie miała zapisane zmiany, jakie wybraliśmy, decydując się na wykonanie `git revert`. Jeżeli wykonamy teraz `git log`, to zobaczymy na ekranie zapis podobny do:

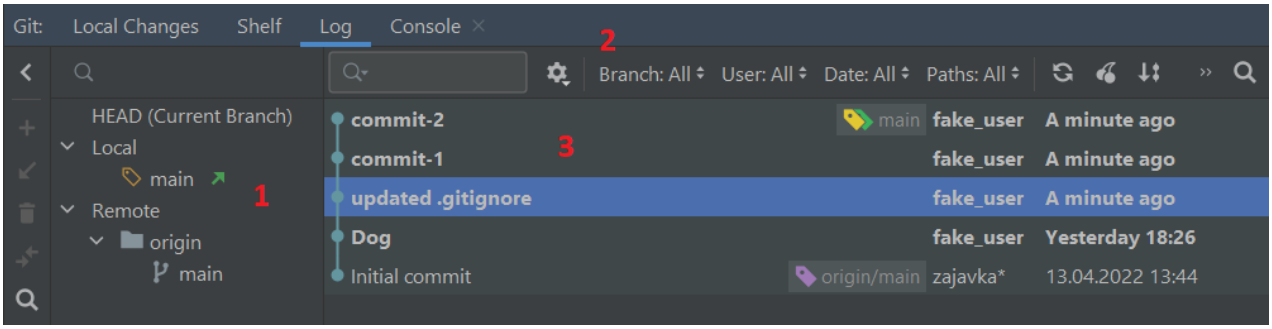
```
commit 9ff8f223ba0a1e482efdb41fb4f0530883baf5a2 (HEAD -> master)
Author: fake_user <fakeemail@gmail.com>
Date:   Thu Apr 21 18:21:26 2022 +0200

    Revert "Merge branch 'my-feature-branch' into master"

    This reverts commit 4304bef8de93d96ec79808fba20d66536bbf58da, reversing
    changes made to a0be914d0db8e5b1a45bce930234caae282cf54c.
```

Zapis ten świadczy o tym, że został wykonany **revert commit**, czyli taki commit, który jest kolejnym wpisem w repozytorium, ale zmienia on stan plików adekwatnie do tego, czy zdecydowaliśmy się na stan plików z brancha **master**, czy z brancha **my-feature-branch**.

Intellij

To co widzieliśmy wcześniej, dotyczyło złączania (mergowania) branchy przy wykorzystaniu samego **Git**a. Jak możesz się domyślić, Intellij pozwala nam zrobić to samo, przy czym tutaj będzie to wyglądało lekko inaczej.

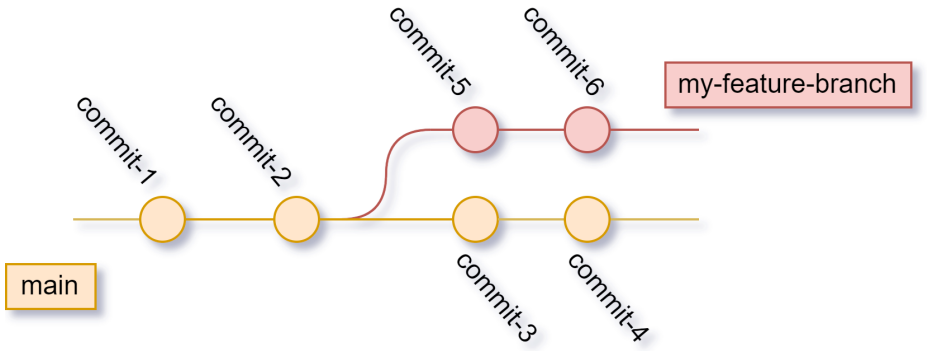
Zanim przejdziemy do wykonania merge, przejdź do zakładki **Git** > **Log** i zobacz, jak Intellij przedstawia Ci branche i commity, które zostały na nich dodane. W tym momencie mamy dostępny tylko branch **master** (na grafice jest **main**), na którym znajduje się kilka commitów.



Obraz 6. Intellij Git Branching

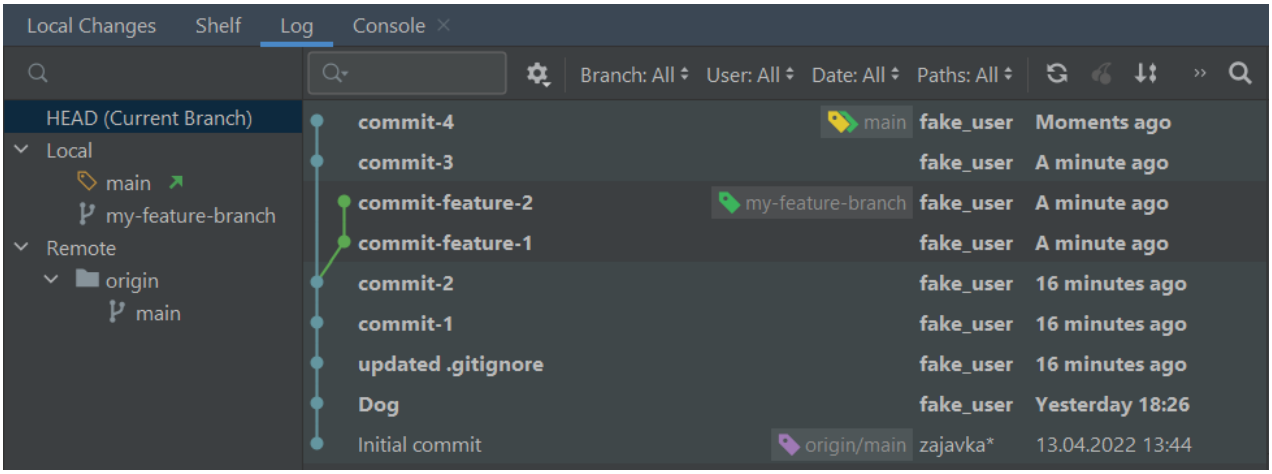
Po lewej stronie ekranu (oznaczone cyfrą 1) możesz zobaczyć dostępne branche local i remote. Jak na razie rozmawiamy tylko o **local**, natomiast niedługo przejdziemy do omówienia branchy zdalnych (***remote***). W tym przypadku mamy dostępny tylko branch lokalny **master** (na grafice **main**), ale w każdej chwili możemy dodać więcej branchy. Pod numerkiem 2 widzisz filtry, które pozwolą nam ograniczyć ilość pokazywanych commitów. Możemy zawęzić wybór do commitów na konkretnym branchu, commitów konkretnego użytkownika, konkretnej daty lub commitów dotyczących plików znajdujących się pod konkretną ścieżką w projekcie. Mamy dostępnych tutaj więcej możliwości, ale nie będziemy się w nie zagłębiać. Pod numerkiem 3 widzisz konkretne commity w naszym repozytorium - analogia do `git log`. Commit jest reprezentowany przez kropkę, wiadomość, autora i datę dodania. Gdy wybierzesz konkretny commit, to po prawej stronie pokaże się jego zawartość - konkretne pliki, jakich commit dotyczył, jego hash i więcej szczegółowych informacji.

W przypadku Intellij poruszymy od razu przypadek merge z konfliktem. Przygotuj sobie zatem dwa branche, które później zmergujemy. Dodaj jakieś zmiany na jednym i drugim w taki sposób, żeby otrzymać konflikt. Przypomnijmy to w formie graficznej:



Obraz 7. Git branches before merge

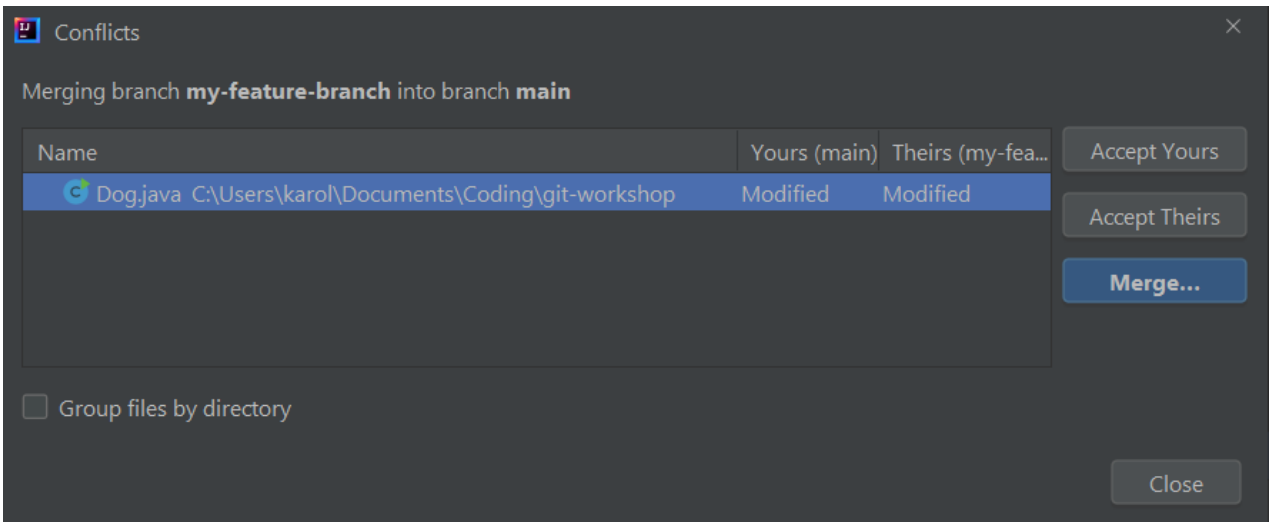
Gdy już to zrobisz, spójrz ponownie w zakładkę **Git** > **Log**, powinna ona wyglądać w ten sposób:



Obraz 8. Intellij Git Branching

Intellij pokazuje w tym momencie wszystkie dostępne branche. Widzisz też, że Intellij wizualizuje branche w sposób analogiczny do wizualizacji branchy we wcześniejszych grafikach. Po lewej stronie ekranu jest pokazany branch, który właśnie dodaliśmy, a **commit-feature-2** ma przyklejoną etykietkę mówiącą, że jest to ostatni commit na branchu **my-feature-branch**. Wytłuszczone commity oznaczają commity zrobione przez Ciebie.

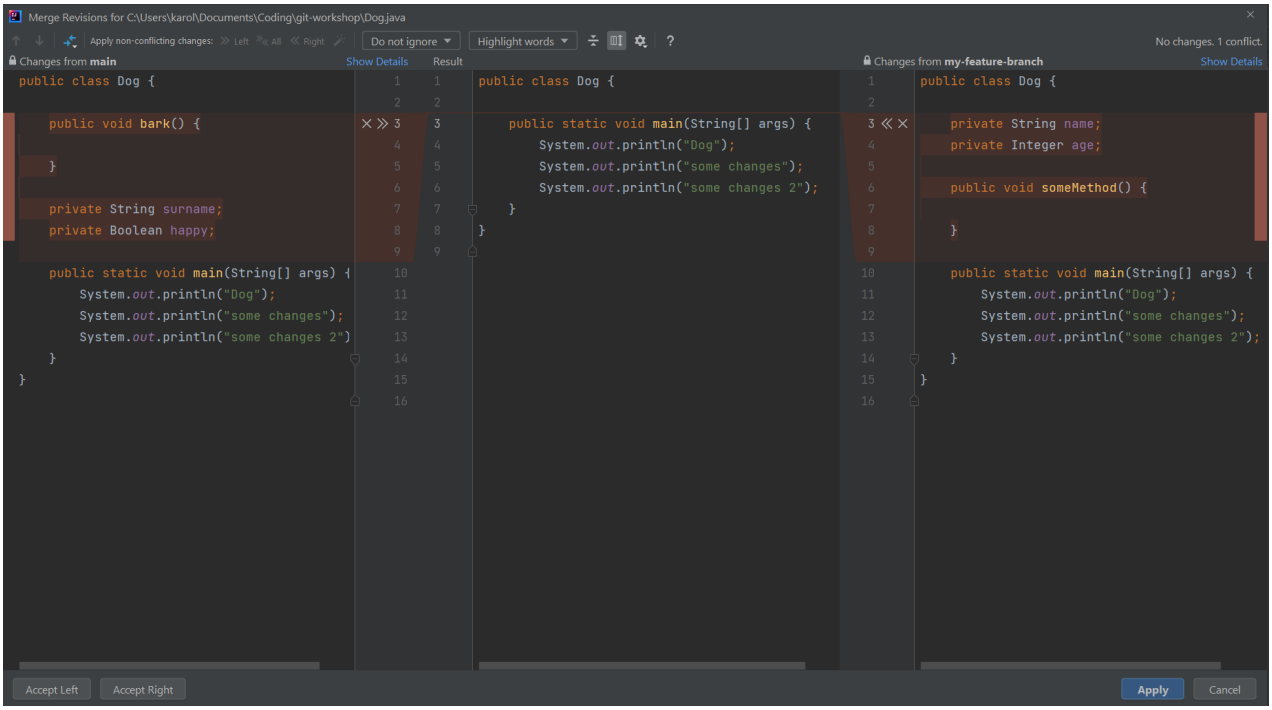
Przejdźmy teraz do wykonania merge. Merge z poziomu Intellij możemy wykonać przy wykorzystaniu okna `Alt + `` i 7. Otworzy nam się wtedy widziane już wcześniej okienko z branchami. Możemy wtedy kliknąć na branch, który chcemy zmergować i wybrać opcję: *Merge 'my-feature-branch' into 'master'*. Jeżeli merge przebiegłby bez konfliktów, to w zasadzie na tym byłby koniec. W tym przypadku celowo wprowadziliśmy konflikt. Pokaże Ci się zatem takie okienko:



Obraz 9. Intellij Git Merge

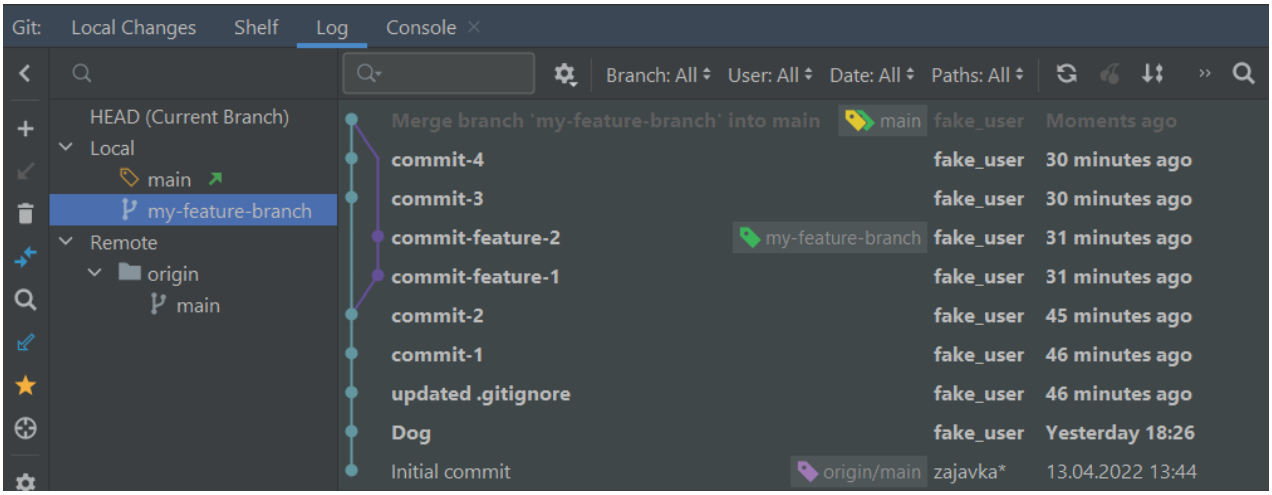
Intellij informuje Cię, co stało się z każdym wymienionym w tym okienku plikiem na każdym z branchy. Akurat mamy tylko jeden plik, więc pokazany jest tylko jeden plik, ale gdyby konflikt dotyczył większej ilości plików, musielibyśmy rozwiązać konflikty dla każdego pliku oddzielnie. Napis *Modified* oznacza, że na obu branchach ten plik został zmodyfikowany. Może też wystąpić taka sytuacja, że na jednym branchu plik został zmodyfikowany, a na drugim usunięty. Możesz też zauważyć dwie kolumny w tabelce: *Yours* i *Theirs*. W nawiasach są nazwy branchy, więc Intellij sprawnie informuje nas o tym, jak należy to rozumieć. Nazwy tych kolumn są nam potrzebne do przycisków po prawej stronie. Możemy bez zastanowienia przyjąć wszystkie zmiany *Yours*, kompletnie odrzucając *Theirs*. Możemy również bez zastanowienia przyjąć wszystkie zmiany *Theirs*, kompletnie odrzucając *Yours*. Najczęściej jednak

wybierzemy przycisk *Merge*. Będziemy musieli to zrobić dla każdego pliku, który będzie konfliktował oddzielnie. Po wybraniu przycisku *Merge* pokaże nam się takie okno:



Obraz 10. Intellij Git Branching

Ogólnie rzecz biorąc, będziesz się z tym widokiem spotykać często, bo jest to okno, w którym Intellij pomaga nam rozwiązywać konflikty. Po lewej stronie widzisz wersję pliku z brancha **master**. Po prawej stronie widzisz wersję pliku z brancha **my-feature-branch**. W środkowej części widzimy efekt, czyli na środku wypracowujemy, jak ma wyglądać ten plik po rozwiązaniu konfliktów. Na czerwono Intellij zaznacza nam wszystkie konfliktujące zmiany. W lewym dolnym rogu możemy ponownie podjąć decyzję czy bezrefleksyjnie akceptujemy zmiany po lewej lub po prawej. Po rozwiązaniu wszystkich konfliktów możemy wybrać **Apply**. Jeżeli wszystko zakończy się sukcesem, to w zakładce **Git** > **Log** zobaczymy taki widok:



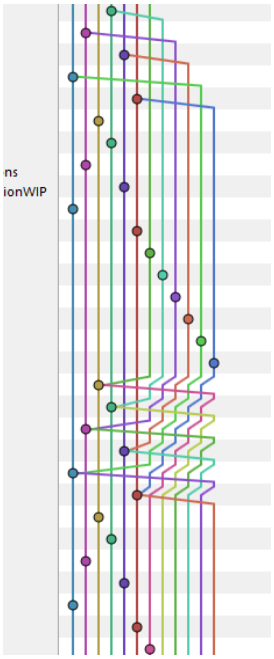
Obraz 11. Intellij Git Branching

Możesz tutaj zaobserwować **merge commit**, o którym rozmawialiśmy wcześniej. Odczyt historii w przypadku gdy zrobiliśmy merge, może być nieco utrudniony. Utrudnienie to polega na tym, że jak spojrzymy na zapis commitów, to pomimo że commity **commit-feature-1** oraz **commit-feature-2** są widoczne, gdy znajdujemy się na branchu **master**, to jednak na wizualizacji są one obok tego brancha.

Jeżeli natomiast wykonasz teraz komendę `git log`, to zobaczysz taki fragment:

```
commit 2171d83e38f02f5ea2a84cfecaf67d9843486123 (my-feature-branch)
Author: fake_user <fakeemail@gmail.com>
Date:   Sat Apr 23 12:32:40 2022 +0200

    commit-feature-2
```

Czyli przy tym konkretnym commicie nadal widnieje informacja, że dotyczy on brancha **my-feature-branch**. Ten przypadek jest prosty w zrozumieniu, natomiast w praktyce może to być bardziej pokręcone i cięższe w zrozumieniu. Przykładem skrajnym jest ta grafika pod spodem, gdzie autor śmieje się, że branche wyglądają jak z gry Guitar Hero:



Obraz 12. Źródło: https://twitter.com/HenryHoffman

Merge możemy również wykonać z poziomu zakładki **Git** > **Log**. Wystarczy, że w tym widoku klikniesz prawym przyciskiem myszy na branchu **my-feature-branch** i pojawi Ci się ta sama opcja: *Merge 'my-feature-branch' into 'master'*.

Więcej na ten temat możesz przeczytać tutaj.