Git - Branches

Spis treści

Branch . . . 1
- Git clone . . . 3
- Nowy branch . . . 3
  - Sami wybieramy źródłową gałąź . . . 4
  - Branch na podstawie commit hash . . . 4
  - HEAD . . . 4
  - Detached HEAD . . . 5
- Dodawanie zmian na branchach . . . 5
- Przełączanie między branchami . . . 6
  - Przełączenie z utworzeniem brancha . . . 7
  - Git switch . . . 7
  - Przełączenie na konkretny commit . . . 8
  - Detached HEAD . . . 9
- Zmiana nazwy brancha . . . 9
- Usunięcie brancha . . . 9
- Intellij . . . 10

Branch

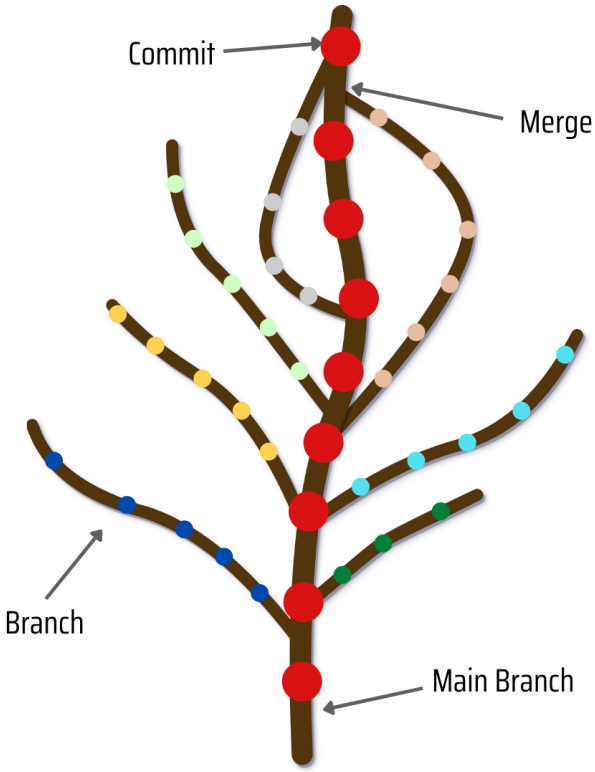
Większość systemów kontroli wersji ma możliwość rozgałęziania kodu. Jednakże w innych systemach kontroli wersji, żeby wydajnie pracować z **branch**ami, często trzeba było mieć ciągły dostęp do internetu. **Git** natomiast pozwala robić rozgałęzienia lokalnie.

Teraz włącz abstrakcyjne myślenie. 😀

Branch (*gałąź*) może być rozumiana analogicznie do gałęzi na drzewie. Centralną częścią drzewa jest **pień**. Z pnia wyrastają gałęzie. Z gałęzi mogą wyrastać kolejne gałęzie. W praktyce konkretna gałąź raczej rzadko wrasta z powrotem w pień, ale tutaj tak będzie. 😀 Dlatego wyobraź sobie drzewo, w którym wystające gałęzie mogą wrastać z powrotem w pień. Cała kwintesencja istnienia drzewa opiera się o to, by to miało pień. Pień można rozumieć jako taką specjalną główną gałąź. Bez pnia - czyli głównej gałęzi, nie ma drzewa.

Branch (*gałąź*) w **VCS**ach służy do tego, żeby móc równolegle pracować nad nowymi funkcjonalnościami w oprogramowaniu, jednocześnie nie wpływając na główne funkcjonalności projektu (czyli na **pień** drzewa). **Branch** tworzony jest wtedy gdy rozpoczynamy pracę nad nowymi funkcjonalnościami w projekcie i gdy zostaną one ukończone, **gałąź** na której pracowaliśmy wrasta z powrotem w **pień**. Przy takim wrośnięciu się, do pnia zostają dodane funkcjonalności wypracowane na tej gałęzi. Gałąź w drzewie może równie dobrze nigdy nie wrosnąć z powrotem w pień, uschnąć i odpaść.

Po co ta analogia z drzewami i gałęziami? Przypomnij sobie, że w systemach kontroli wersji zapisujemy punkty w czasie razem ze zmianami w projekcie w danym punkcie w czasie. Wszystko po to, żeby było wiadomo kto i kiedy wprowadził konkretne zmiany do projektu. Spróbuj sobie teraz wyobrazić, że takie punkty w czasie możemy wizualnie ulokować na gałęziach drzewa. Mam nadzieję, że pomoże Ci w tym poniższa grafika:



Obraz 1. Git Branching analogy

Kolorowe punkty na gałęzi reprezentują konkretną zmianę w projekcie, która ma cechy takie jak m.in.: swój unikalny identyfikator, czas dokonania tej zmiany, autora oraz zakres zmian.

Pniem w **Git**, czyli główną gałęzią jest branch, który nazywa się **master**. **Branch**e mają swoje nazwy. Czasami główny **branch** może być również nazwany **main** (taką nazwę zasugerował GitHub przy tworzeniu repozytorium). W repozytorium może się znajdować tylko jedna główna gałąź, tak samo, jak drzewo ma tylko jeden pień.

Gdy zaczynasz zapisywać swoje zmiany w repozytorium, musisz je zapisywać na jakiejś gałęzi. Jeżeli pracujesz samodzielnie, to możesz pracować na gałęzi głównej i my właśnie to robiliśmy dotychczas - pracowaliśmy na gałęzi głównej **main**. W zespołach kilkuosobowych podchodzi się do tego tematu inaczej i pracuje na wielu gałęziach jednocześnie. Żeby zmiany z takiej gałęzi trafiły finalnie do klienta końcowego, muszą one być dodane do gałęzi głównej - czyli gałęzie muszą wrosnąć z powrotem w pień. W wyniku dodawania kolejnych zmian na gałęziach ich długość zaczyna się zwiększać. Pojawiają się kolejne punkty w czasie z kolejnymi zmianami. Czyli można powiedzieć, że gałąź rośnie. 😀

Podczas pracy z kodem możemy stworzyć wiele branchy i wiele branchy może istnieć równolegle. Natomiast żeby dodawać nasze zmiany do repozytorium, musimy dodawać je znajdując się na jakimś **branch**u. Mówiąc "znajdując się", chodzi o to, żebyśmy mieli wybrany branch, do którego będziemy dodawali zmiany. Możliwe jest dodawanie zmian od razu do gałęzi głównej. Możemy również pracować na nowych gałęziach, a na koniec przerzucać nasze zmiany do gałęzi głównej (nazywa się to **merge** i wrócimy do tego później).

Git clone

Wcześniej pojawiło się stwierdzenie `git clone`. Klonowanie służyło do tego żebyśmy odtworzyli projekt ze zdalnego repozytorium w naszym repozytorium lokalnym. Inaczej mówiąc, żebyśmy ten projekt pobrali. Skoro rozmawiamy teraz o **branch**ach, to warto zaznaczyć, że gdy wykonaliśmy `git clone`, to został automatycznie ustawiony **branch** główny, gdzie w naszym przypadku jego nazwa to **main**. Przypomnij sobie, że wcześniej było to już widoczne na ekranie gdy wpisywaliśmy komendę `git status`. Jak jeszcze można sprawdzić, na jakim **branch**u aktualnie się znajdujemy?

```
git branch
```

Zostaną wtedy wypisane **branch**e jakie mamy dostępne lokalnie i gwiazdką zostanie zaznaczony ten, który mamy aktualnie wybrany.

Możemy również ograniczyć ilość wyświetlanych branchy, przy wykorzystaniu flagi `-l`, która pozwala nam na podanie wzorca dopasowania nazwy brancha:

```
git branch -l 'm*'
```

Otrzymamy wtedy branche, których nazwa zaczyna się na `m`. Po tej literce może być dowolny znak.

Nowy branch

Zacznijmy od tego jak można rozgałęzić pracę na naszym kodzie lokalnie. W tym celu stworzylibyśmy nowy **branch**:

```
git branch nazwa-nowego-brancha
```

Jeżeli teraz wpiszesz ponownie komendę `git branch` to zobaczysz, że został utworzony nowy branch o nazwie **nazwa-nowego-brancha**, ale my nadal korzystamy z **main**.

Wywołanie komendy tworzącej nowy **branch** spowodowało utworzenie nowej gałęzi na podstawie tej, na której się obecnie znajdowaliśmy. Jeżeli znajdowalibyśmy się na innej gałęzi niż **main**, to ta inna gałąź byłaby podstawą do stworzenia nowego **branch**a. Nowa gałąź jest dokładną kopią obecnego stanu **main**. Zmiany które będą dodawane teraz na tej nowej gałęzi nie mają żadnego wpływu na zmiany widoczne w gałęzi **main** i odwrotnie. Jeżeli chcemy by te zmiany były widoczne, musimy o tym zdecydować i wpisać odpowiednie komendy - nie stanie się to automatycznie.

W jaki sposób możemy się teraz przełączyć na nową gałąź? Musimy wpisać komendę:

```
git checkout nazwa-nowego-brancha
```

Jeżeli teraz wpiszesz komendę `git branch` to zobaczysz, że używamy już nowego **branch**a jako naszego aktualnego. Możesz teraz wrócić do korzystania z poprzedniego **branch**a wykorzystując komendę:

```
git checkout main
```

Sami wybieramy źródłową gałąź

W poprzednim przypadku nie podaliśmy jaka gałąź ma służyć za podstawę utworzenia naszego **branch**a, możliwe jest natomiast żeby to określić. Jeżeli jesteśmy teraz na **nazwa-nowego-brancha** i chcemy stworzyć **branch**a na podstawie **main**, możemy napisać taką komendę:

```
git branch nowy-branch main
```

Dzięki temu poleceniu utworzyliśmy gałąź **nowy-branch** bazującą na gałęzi **main**.

Branch na podstawie commit hash

W praktyce wystąpią takie sytuacje, gdzie będziesz potrzebować sprawdzić, czy kod w takiej postaci w jakiej istniał 3 dni temu działał jak trzeba. Wynikać to może z tego, że kod w obecnym stanie nie zachowuje się poprawnie i chcesz się upewnić, która zmiana miała na to wpływ. Może okazać się wtedy, że kod nie działał poprawnie jeszcze wcześniej i będziesz mieć potrzebę cofania się w historii commitów jeszcze dalej. Powodów może być więcej, ale to jest sytuacja, która w praktyce występuje i czeka Cię ona prędzej czy później.

Swoją drogą to jeżeli znajdziemy się w opisanej sytuacji to idealnie jeżeli używamy w projekcie jakiegoś **VCS**. Dzięki temu będziemy mogli cofnąć się w czasie do konkretnego momentu w projekcie żeby sprawdzić, czy już wtedy coś było nie tak, czy to my coś popsuliśmy.

Możesz w takiej sytuacji stworzyć **branch** na podstawie **commit hash**, gdzie najnowszą zmianą na takim **branch**u będzie właśnie podany przez Ciebie **commit**. Należy wykonać wtedy komendę:

```
git branch branch-from-commit cf941b7
```

HEAD

Spotkaliśmy się już wcześniej ze stwierdzeniem **HEAD**. Przypomnijmy, że stwierdzenie **HEAD** jest używane odpowiedzi na pytanie *Gdzie ja teraz jestem?*. Co ciekawe, **Git** nawet zapisuje takie informacje w pliku `.git/HEAD`. Jeżeli teraz wpiszesz w terminalu (będąc w katalogu projektu):

```
notepad .git/HEAD
```

To zawartość tego pliku pokaże Ci, jaki branch jest ustawiony jako Twój aktualnie używany branch.

Przypomnij sobie, że jak wykonywaliśmy polecenie `git log`, to przykładowo na ekranie mógł być wtedy wydrukowany taki rezultat:

```
commit 6665d10dd03b972a70a323744ed85901c41a1ded (HEAD -> main) ①
```

```
Author: <user_name> <email>
Date:   Mon Apr 18 13:58:46 2022 +0200

    My next commit message
```

① Tutaj pojawia się oznaczenie **HEAD**, które mówi nam, że obecnym **HEAD** jest branch **main**.

Detached HEAD

Może wystąpić taka sytuacja, gdzie **HEAD** nie będzie wskazywało na konkretny branch, tylko np. na konkretny **commit**. Taką sytuację określa się stwierdzeniem, że repozytorium jest w stanie **Detached HEAD**. Wrócimy jeszcze do tego.

Dodawanie zmian na branchach

Skoro wiemy już jak tworzyć branche, możemy zobaczyć, jak zachowa się repozytorium, gdy zaczniemy dodawać pewne modyfikacje na branchach i zaczniemy sie między tymi branchami przełączać.

Będąc na nowym branchu, np: **new-branch**, wykonaj kilka modyfikacji w projekcie i dodaj commit z tymi zmianami do tego brancha. Dodaj jeszcze parę takich commitów. Następnie przełącz się na branch **main** i poszukaj dokonanych przez Ciebie zmian. **Nie ma?** I ma nie być, bo zmiany zostały wprowadzone na branchu **new-branch**, zatem nie są widoczne na branchu **main**.

Jeżeli wywołamy teraz `git log` na branchu **new-branch**, to będzie to wyglądało np. w ten sposób:

```
commit 05e6486ea09a0ed47eef8ef24be1157c8273c118 (HEAD -> new-branch)
Author: <user_name> <user_email>
Date:   Wed Apr 20 07:41:41 2022 +0200

    commit-message-on-branch-2

commit 408f73479147539bc948f30a92cae42b8165ccc4
Author: <user_name> <user_email>
Date:   Wed Apr 20 07:39:11 2022 +0200

    commit-message-on-branch-1

commit f82bfbec78761ecb7489354e518342a7f6e881e1 (main)
Author: <user_name> <user_email>
Date:   Wed Apr 20 07:37:42 2022 +0200

    commit-message-2

commit 4b0b9d104b5219c13c0dddb811586cf306cd4d37
Author: <user_name> <user_email>
Date:   Wed Apr 20 07:35:54 2022 +0200

    commit-message-1
```

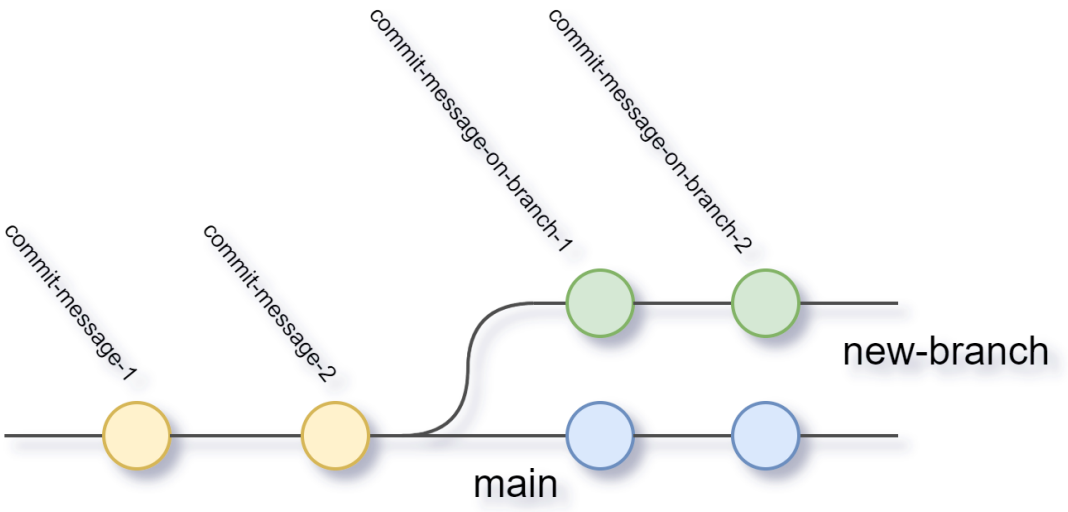
Jeżeli natomiast przełączysz się na branch **main** i wykonasz `git log`, to zobaczysz na ekranie zapis podobny do:

```
commit f82bfbec78761ecb7489354e518342a7f6e881e1 (HEAD -> main)
```

```
Author: <user_name> <user_email>
Date:   Wed Apr 20 07:37:42 2022 +0200

    commit-message-2

commit 4b0b9d104b5219c13c0dddb811586cf306cd4d37
Author: <user_name> <user_email>
Date:   Wed Apr 20 07:35:54 2022 +0200

    commit-message-1
```

Oznacza to, że na branchu **main** nie ma zmian, które zostały dodane na branchu **new-branch**. Nawet jeżeli spojrzysz na zawartość pliku, który był modyfikowany to będzie się ona zmieniała w zależności od tego, który branch jest wybrany jako aktualny. Zwróć też uwagę, że branch **new-branch** ma dostępne te same **commity**, które są widoczne na branchu **main**. Widać dzięki temu, że branch **new-branch** powstał na bazie brancha **main** i zostały do niego dodane kolejne wpisy historyczne. Jeżeli chcielibyśmy teraz, żeby zmiany dodane na branchu **new-branch**, zostały dodane do brancha **main**, należałoby wykonać **merge** - nie stanie się to automatycznie. O tym dowiemy się później.

A w jaki sposób można to sobie zwizualizować?



Obraz 2. Git Branching visualisation

Punkty żółte to commity, które były na branchu **main**, zanim powstał branch **new-branch**. Następnie tworzymy branch **new-branch**, który powstaje na podstawie brancha **main**. Oznacza to, że widać na nim taką samą historię, jaka była na branchu **main**. Na branchu **new-branch** zostały dodane nowe commity, które nie są widoczne na branchu **main** - oznaczamy je kolorem zielonym. Kolor niebieski reprezentuje commity, które w przykładzie wyżej nie zostały dodane, ale mogłyby. Zaznaczone zostały na niebiesko, żeby pokazać, że jeżeli w tym miejscu dodalibyśmy commity do brancha **main**, to nie byłyby one widoczne na branchu **new-branch**.

Przełączanie między branchami

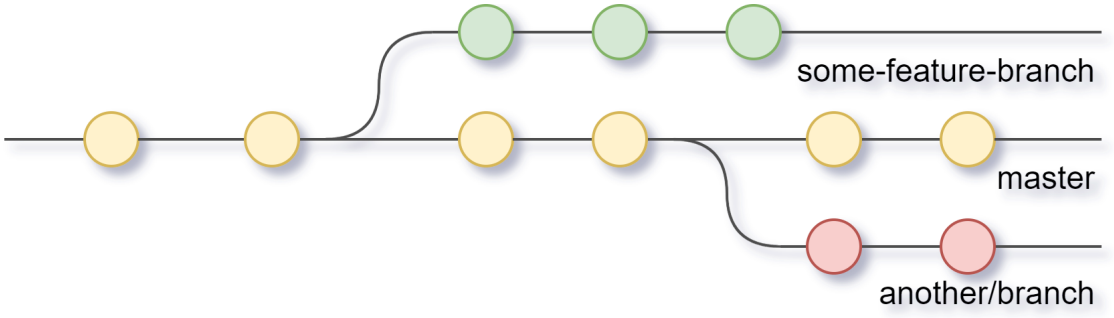
Praca na wielu branchach jest normalna i popularna w praktyce. Każdy branch może wtedy mieć nazwę dedykowaną pod konkretną funkcjonalność wprowadzaną do projektu. Do tego, na każdym branchu możemy pracować wtedy nad jedną konkretną funkcjonalnością niezależnie od innych. Praca ułożona w

ten sposób oznacza, że będziemy musieli umieć sprawnie się pomiędzy tymi branchami przełączać.

Poznaliśmy już podstawową komendę `git checkout`, która pozwala na przełączanie się między branchami, przykładowo:

```
git checkout some-feature-branch
git checkout master
git checkout another/branch
```

Gdybyśmy chcieli to zwizualizować, możemy przedstawić to tak jak na poniższym obrazku:



Obraz 3. Git Branching visualisation

Możemy teraz przełączać się między tymi branchami, żeby pracować na konkretnych zmianach, które są widoczne tylko na konkretnym branchu. Przypominając, jeżeli chcemy, żeby przykładowo zmiany z brancha **some-feature-branch** zostały wprowadzone na branch **master**, musimy ręcznie wykonać **merge** - o nim będzie później. Jeżeli dojdziemy do wniosku, że jednak zmiany z brancha **another/branch** mają finalnie nie trafić do projektu, możemy ten branch po prostu usunąć.

Gdy przełączymy branch, automatycznie widzimy najnowsze zmiany na danym branchu. Po przełączeniu się na inny branch będziemy widzieli najświeższy commit oraz poprzednie. Odnosząc się do obrazka wyżej, najświeższy, czyli ten najbardziej po prawej.

Przełączenie z utworzeniem brancha

Komenda `git checkout`, oprócz przełączenia na nowy branch, może też zostać wykorzystana do utworzenia nowego brancha w locie. Wystarczy wtedy wykorzystać flagę `-b`, przykład:

```
git checkout -b new-branch-to-be-created
```

Flaga `-b` może zostać użyta łącznie z wariantem, gdzie chcemy wskazać branch na podstawie którego chcemy nowy branch utworzyć. Możemy też ją wykorzystać z wariantem, w którym tworzymy branch na podstawie **commit hash**.

Git switch

Git w wersji 2.23 wprowadził nową komendę `git switch`. Komenda ta ma być czytelniejsza i bardziej zrozumiała niż `git checkout`. W pewnym sensie można powiedzieć, że `git switch` może być używane zamiast `git checkout`. Jakie są podobieństwa, a jakie różnice przy wykorzystaniu tych komend?

Czynność	git checkout	git switch
Przełączenie między branchami	git checkout new-branch	git switch new-branch
Utworzenie brancha i przełączenie między branchami	git checkout -b new-branch	git switch -c new-branch

Przełączenie na konkretny commit

Polecenie `git checkout` pozwala na przełączenie się na konkretny commit o konkretnym **hash**u. Jeżeli wykonam teraz taką komendę:

```
git checkout c3bec1cef9a929b8ff1188cd6a5d2957fa75cba1
```

To na ekranie zostanie wydrukowana taka informacja:

```
Note: switching to 'c3bec1cef9a929b8ff1188cd6a5d2957fa75cba1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at c3bec1c Some commit message
```

Jeżeli wejdziesz teraz do pliku `.git/HEAD` to zobaczysz, że będzie tam wpisany **commit hash**, a nie nazwa brancha, który jest **HEAD**em. Wiadomość, jaką wydrukował właśnie **Git**, wydaje mi się wystarczająca. Możemy teraz podejrzeć, jak wyglądał stan repozytorium w punkcie w czasie, jakim jest podany **commit**. Możemy dokonać zmian, które następnie możemy odrzucić, albo możemy na podstawie tych zmian stworzyć nowy branch. Zwracam tutaj też jednocześnie uwagę, że sam **Git** napisał, że repozytorium jest teraz w stanie **Detached HEAD**.

Stan, w jakim obecnie znajduje się repozytorium, możemy traktować jako taki stan tymczasowy. Musimy z tego stanu tymczasowego wyjść w jakiś sposób, albo odrzucając wprowadzone zmiany, albo tworząc branch, albo możemy równie dobrze wrócić na jakiś konkretny branch przy wykorzystaniu komendy `git checkout`, albo `git switch`.

Po co checkoutować się na konkretny commit? Wyobraź sobie, że pracujesz nad nowymi funkcjonalnościami w projekcie i chcesz sprawdzić, czy to, że kod Ci nie działa, wynika z Twoich obecnych zmian, czy może ten sam kod nie działał już prawidłowo 4 commity temu. W takiej sytuacji możesz wykonać checkout na konkretny commit i sprawdzić co w trawie piszczało. Drugą możliwością jest utworzenie brancha z konkretnego commita i tam można zrobić to samo.

Detached HEAD

Stan *Detached HEAD* oznacza, że aktualnie pracujemy na commitcie, który nie jest związany z żadnym branchem. Oznacza to, że Git nie będzie automatycznie zapisywał na żadnym branchu zmian, które wykonujemy.

Stan *Detached HEAD* oznacza, że wskaźnik HEAD wskazuje na konkretny commit, a nie na ostatni commit w bieżącej gałęzi (wcześniej określiliśmy to stwierdzeniem *wskazywanie na branch*). Innymi słowy, jeżeli nie wiesz, co robisz i nie chcesz zrobić sobie kuku i namieszać w swojej historii zmian, używaj stanu *Detached HEAD* tylko do oglądania stanu repozytorium w konkretnym commicie.

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.

Gdy już znajdziemy się w stanie *Detached HEAD*, należy uważać, aby nie zacząć dodawać nowych commitów, które mogą być trudne do odnalezienia w przyszłości, gdyż nie będą one dodawane do żadnego brancha. Więcej na temat problemów związanych z "byciem" w stanie *Detached HEAD* możesz przeczytać tutaj.

Zmiana nazwy brancha

Nazwę utworzonego brancha można zmienić wykorzystując flagę `-m`. Przykład:

```
git branch -m old-name new-name
```

Jeżeli chcemy zmienić nazwę brancha, na którym się obecnie znajdujemy, możemy wykorzystać komendę:

```
git branch -m new-name
```

Usunięcie brancha

W praktyce tworzy się często dużo branchy. Ważne jest to, żeby pilnować porządku, bo im więcej tych branchy stworzymy, tym łatwiej będzie się w tym zgubić. Dlatego najlepiej jest usuwać branche, gdy nie są nam już potrzebne. Branch przestaje nam być najczęściej potrzebny, gdy zmiany na nim wprowadzone zostaną złączone z gałęzią główną.

Nie możemy usunąć brancha, na którym obecnie się znajdujemy. To trochę tak jakbyśmy ucięli gałąź

drzewa, na której obecnie siedzimy. Musimy najpierw przełączyć się na inną gałąź - `git checkout`. Gdy znajdujemy się już na innej gałęzi, możemy wykonać:

```
git branch -d new-branch
# lub
git branch --delete new-branch
```

Git jest zmyślnym narzędziem, więc nie pozwoli nam usunąć gałęzi z commitami, które nie zostały włączone do gałęzi głównej. **Git** zwróci nam również uwagę, w sytuacji, gdy commity z brancha, którego chcemy usunąć, nie znalazły się w zdalnym repozytorium.

Cały czas poruszamy się w obrębie repozytorium lokalnego. Nie zostało jeszcze pokazane, w jaki sposób wypchać swoje zmiany do repozytorium zdalnego ani jak opracować z branchami z repozytorium zdalnym. Spokojnie, niedługo do tego przejdziemy.

Gdy spróbujemy usunąć branch, na którym znajdują się commity, które nie zostały złączone z główną gałęzią, **Git** odpowie nam taką wiadomością:

```
error: The branch 'new-branch' is not fully merged.
If you are sure you want to delete it, run 'git branch -D new-branch'.
```

Jeżeli jesteśmy natomiast całkowicie pewni, że chcemy się pozbyć takiego brancha, bo na pewno nie będą nam potrzebne commity, które są na nim dodane, możemy wykonać komendę, którą **Git** sam nam podpowiedział:

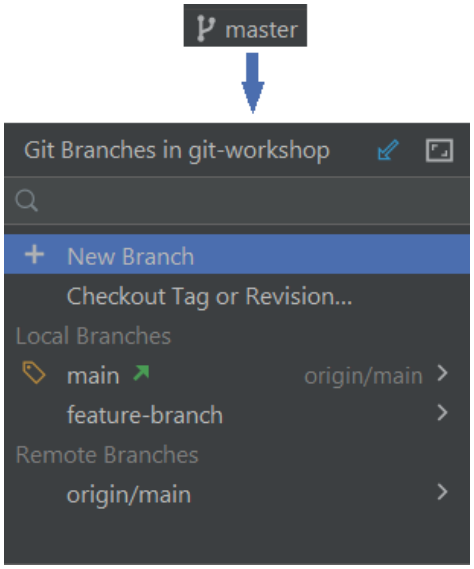
```
git branch -D new-branch
```

Możesz zwrócić uwagę, że już któryś raz **Git** podpowiada nam, co możemy zrobić w danej sytuacji. Jeżeli trakcie pisania popełnisz jakąś literówkę, to prawdopodobne jest, że **Git** również zauważy, gdzie został popełniony błąd i zasugeruje Ci, jak można go poprawić. Prawda, że zmyślne narzędzie? 😀

Intellij

Jak możesz się domyślić, tworzenie branchy i przełączanie między nimi jest też możliwe z poziomu Intellij. Poniżej zostaną pokazane cztery sposoby:

1. W prawym dolnym roku ekranu widać ikonkę rozgałęzienia z napisem **main** lub **master** lub inną nazwą brancha. Jeżeli na nią klikniesz, to pojawi Ci się poniższe okienko. Z tego poziomu możesz stworzyć branch albo przełączać się pomiędzy branchami wybierając ich nazwy. Widoczne tutaj są też branche **remote** - do tego przejdziemy później.



Obraz 4. Intellij Git Branches

2. Możesz przejść do górnego paska **Git** i tam masz dostępną opcję **Branches**. Gdy ją wybierzesz - otworzy nam się analogiczne okienko do poprzedniego. To samo okienko otworzy się, gdy wciśniemy skrót `Ctrl + Shift + `` .
3. Możesz przejść na drzewo projektu po lewej stronie, kliknąć prawym przyciskiem myszy i wybrać opcję **Git**. Uruchomi Ci się wtedy multum opcji, z których coraz więcej już pewnie kojarzysz.
4. Gdy przejdziesz do zakładki **Git** > **Log**, po lewej stronie również zobaczysz dostępne branche.

Korzystając z dowolnej z tych opcji, możesz wybrać branch na podstawie jego nazwy i pokaże Ci się więcej opcji, które powinny być zrozumiałe, jeżeli rozumiesz komendy, jakie są dostępne w **Git**. Możesz z tego poziomu zmieniać branche, możesz zmieniać nazwę brancha lub go usunąć. Pamiętaj tylko, że nie możesz usunąć brancha, na którym się znajdujesz.

Więcej na ten temat możesz przeczytać tutaj.