Git przekleństwo i zbawienie w jednym.

Git jest jak nóż, można go używać dobrze albo źle. W poniższym artykule przedstawię jak ułatwić sobie i innym życie z pomocą Git'a, oraz czego się wystrzegać. Nie jest to artykuł z podstaw Git'a, nie jest to artykuł teoretyczny i na pewno nie wyczerpuje tematu. Jest on praktyczną pomocą dla osób pracujących w zespołach i nie mających dużego doświadczenia z Git'em.

1. Nie bój się gita!

Najważniejsze, nie bój się. Jeśli nie jesteś pewien czy dobrze robisz, przetestuj to.

a. reset –hard

Jeśli chcesz coś zrobić na branchu, ale boisz się że coś popsujesz, to wystarczy że upewnisz się że masz jego identyczną wersję w źródłowej wersji (origin) `git diff origin/<*nazwa brancha*>` (różnica miedzy pull a fetch). Wtedy cokolwiek zrobisz na branchu, zawsze będziesz mógł go zresetować do wersji z której zaczynałeś `git reset –hard origin/<*nazwa brancha*>`.

b. branche testowe

Git pozwala tworzyć nowe branche w dowolnych ilościach i to tak żeby nikt inny ich nie widział. Więc nie ma żadnego powodu dla którego nie mielibyśmy tworzyć sobie branche do testów. Np. jeśli z jakiegoś powodu twój branch różni się od jego wersji `origin`, to możesz utworzyć jego wersję testową poprzez utworzenie nowego brancha na jego podstawie `git branch -b <nazwa brancha>-test`. Jeśli coś pójdzie nie po naszej myśli (np. rebase), możemy przejść z powrotem na oryginalny branch `git checkout <nazwa brancha>`, usunąć testowy branch `git branch -D <nazwa brancha>-test` i utworzyć go ponownie.

c. push –force

`git push –force` pozwala nadpisać historię brancha w repozytorium. Gdy usunęliśmy przypadkowo mastera, lub domergowaliśmy stage do mastera, możemy poprosić kogoś z teamu, aby (nie robiąc uprzednio pulla) wszedł na popsuty branch i jeśli miał go aktualnego (przed twoimi zmianami), to może po prostu nadpisać swoją wersją, wersję popsutą/usuniętą. Jest to metoda “atomowa” więc trzeba upewnić się trzy razy zanim się to zrobi. Dodatkowo jest to wymagane po rebasowaniu brancha.

d. checkout

Jeśli nie mamy nikogo w zespole, albo nikogo kto miałby ostatnią stabilną wersję brancha, możemy jeszcze sami odzyskać prawidłową wersję.

Wystarczy że znamy hash ostatniego commita ze stabilnej wersji. Można go znaleźć np. w pipelines gitlaba `stage 944551f3`, dokumentacji wyjść produkcyjnych, logach gita, czy nazwie katalogu z wersją aplikacji na serwerze (zależnie od strategii i zasad wewnętrznych firmy). Jeśli mamy hash to wystarczy że przejdziemy na niego `git checkout <hash commita>`, usuniemy branch który chcemy naprawić/odzyskać (jeśli już tego nie zrobiliśmy) `git branch -D <nazwa brancha>` i utworzyć go na nowo `git checkout -b <nazwa brancha>.

e. reflog

Git jest na tyle zmyślnym systemem że niemal wszystko da się naprawić. W ostateczności możemy skorzystać z reflog’a. W poniższym przykładzie zrebasowałem branch `b` do mastera i chciałbym to cofnąć.

```
b18aff1 (HEAD -> b) HEAD@{0}: rebase finished: returning to refs/heads/b
b18aff1 (HEAD -> b) HEAD@{1}: rebase: test3
6377c8f (master) HEAD@{2}: rebase: checkout master
a85e451 HEAD@{3}: checkout: moving from master to b
```

Wyszukujemy ostatni wpis sprzed rozpoczęcia rebase’a i kopiujemy jest hash (a85e451), lub numer wpisu (HEAD@{3}) i przechodzimy na niego `git checkout <hash lub numer wpisu>`, po czym usuwamy uszkodzonego brancha `git branch -D <nazwa brancha>` i tworzymy go na nowo `git checkou -b <nazwa brancha>`.

2. Konsola nie gryzie

Druga najważniejsza sprawa. Nie korzystaj z programów do pracy z gitem jeśli nie wiesz jak działają i jak działa git. Często osoby które ledwo co wiedzą o git'ie korzystają z różnych programów, do jego obsługi, aby ułatwić sobie życie. W większości przypadków prowadzi to do złych przyzwyczajeń, bałaganu w repozytorium i dziwnych sytuacji w których nie wiadomo co się stało. Pamiętaj że są różne `flow` pracy z gitem i narzędzie z którego korzystasz ma domyślne ustawienia, które mogą być niekompatybilne z `flow` projektu w którym pracujesz.
Ja po krótkiej pracy z gitkrakenem i funkcjonalnością GIT w phpstormie, wróciłem do konsoli, bo w niej bardziej panuję nad tym co się dzieje. Chociaż konflikty rozwiązuję z pomocą PHPStorma (https://www.jetbrains.com/help/phpstorm/resolve-conflicts.html).
Jeśli chcesz zrozumieć jak działa git, to praca w konsoli zdecydowanie Ci to ułatwi.

3. Workflow

Aby przejść dalej, musimy najpierw wiedzieć jakie są typy pracy na Git'ie. Poniżej wypiszę kilka z nich, wraz z plusami i minusami

a. Trunk-based Development

dokumentacja: https://trunkbaseddevelopment.com

To podejście, w którym wszyscy deweloperzy commitują swoje zmiany bezpośrednio do jednej głównej gałęzi (trunk). Zmiany są często mergowane do trunku, minimalizując liczbę długoterminowych branczy.

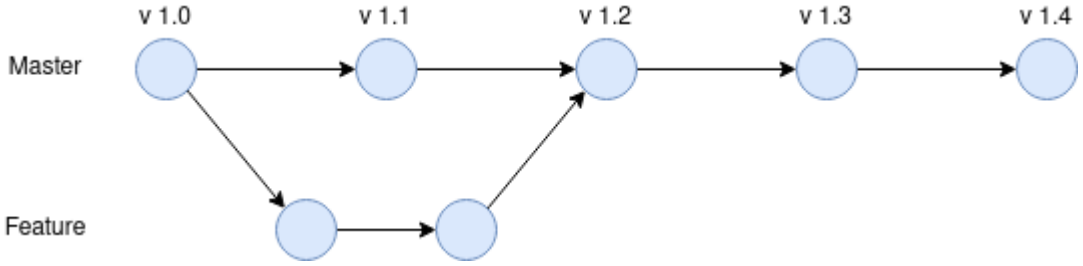
Plusy:
- szybkość wykrywania błędów
- bardzo duża szybkość wdrażania funkcjonalności
- minimalna ilość konfliktów
- bardzo prosty workflow

Minusy:

- bardzo duży brak stabilności (można go zmniejszyć np. testami automatycznymi)
- wymaga doświadczonych programistów którzy są odpowiedzialni i zdyscyplinowani
- brak możliwości wydawania konkretnych funkcjonalności

b. Feature Branch Workflow

dokumentacja: https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow



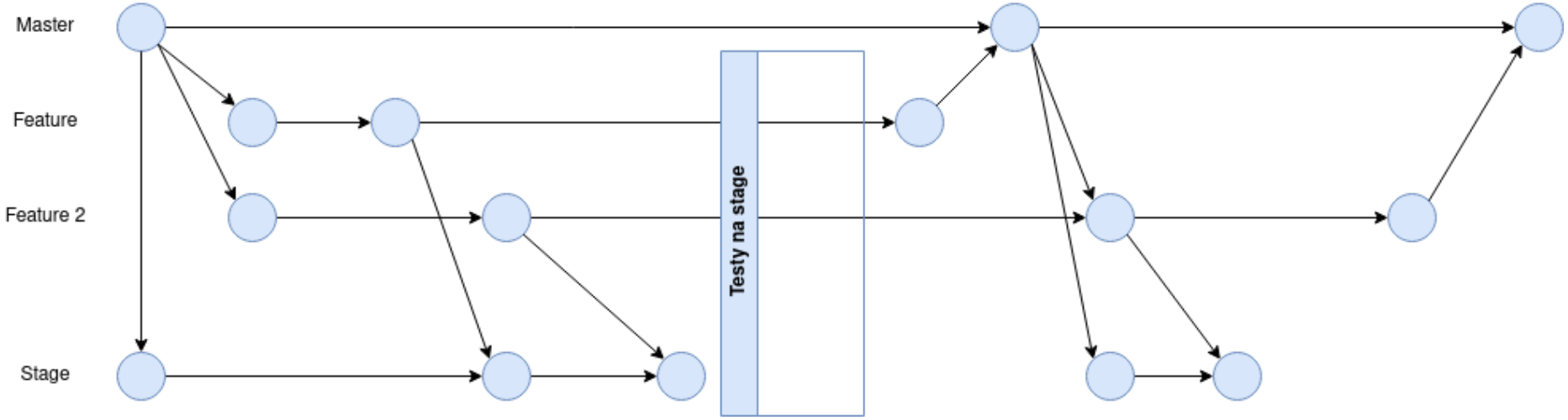
W tym systemie mamy jeden główny branch z którego są tworzone feature branche. Każdy dla osobnej funkcjonalności. Po skończeniu prac jest on bezpośrednio mergowany do mastera.

Plusy:
- częstość występowania konfliktów jest niska
- duża szybkość wprowadzania nowych funkcjonalności

Minusy:
- brak stabilnego brancha który w każdej chwili może być wydany na produkcję
- brak z automatu możliwości wydawania konkretnych funkcjonalności
- duża częstotliwość błędów uniemożliwiających pracę (które można zminimalizować np. dobrą organizacją)

Stosowane głównie gdy tworzony jest nowy projekt, który nie jest wydawany produkcyjnie aż do ostatecznej wersji. Wtedy stabilność projektu nie jest aż tak istotna.

c. Stage flow



W tym systemie jest jeden branch służący do wyjść produkcyjnych (master), jeden branch do wyjść na środowisko testowe (stage) i feature branche z funkcjonalnościami. Każdy feature branch jest tworzony z mastera i po wykonaniu prac jest mergowany do brancha testowego (stage). Po przejściu testów, feature branch jest mergowany do mastera. Po każdym wyjściu produkcyjnym branch stage jest tworzony od nowa aby zapobiec konfliktom i dublowaniu się commitów.

Plusy:

- stabilny branch główny
- możliwość wydawania pojedynczych funkcjonalności
- mała ilość krytycznych błędów w repozytorium

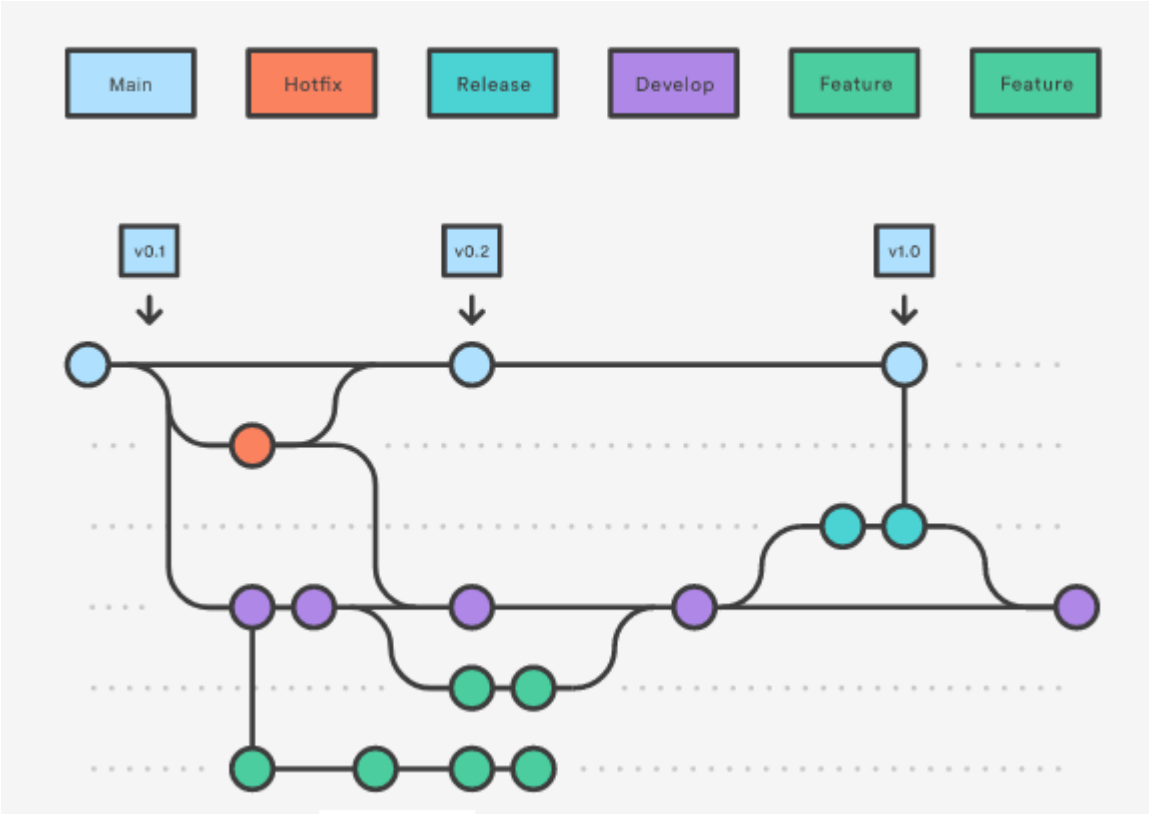
Minusy:

- częste występowanie konfliktów na różnych etapach pracy
- potrzeba usuwania brancha testowego i tworzenia go najlepiej co deploy z mastera
- potrzeba rebasowania feature branchy do mastera co merge innego feature brancha do mastera
- duże skomplikowanie

Stosowany głównie w projektach rozwojowych w których funkcjonalności muszą być wydawane w różnym czasie i wymagana jest możliwość wprowadzania szybkich zmian.

d. Git Flow

dokumentacja: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow



Z głównego brancha jest tworzony branch rozwojowy (develop) i z niego tworzone są feature branche. One są mergowane do brancha rozwojowego. Jak uzbiera się ich odpowiednia/ustalona ilość, tworzony jest branch wydaniowy (release) z rozwojowego. Jest on doprowadzany do pełnej stabilności, a potem mergowany do głównego brancha i ten ostatni jest tagowany.

Plusy:

- częstość występowania konfliktów jest stosunkowo mała
- szybkość wprowadzania zmian ogólnodostępnych (poza feature branch)
- stabilność branch głównego

Minusy:

- brak możliwości wydawania pojedynczej funkcjonalności
- brak możliwości korzystania z pojedynczego środowiska testowego

4. Dlaczego warto dbać o porządek w repozytorium?

- Aby ułatwić sobie życie
- Aby ułatwić innym życie
- Abyś nie marnował czasu innych
- Abyś nie wkurzał innych
- Abyś nie dodawał pracy innym
- Abyś nie marnował swojego czasu
- Abyś nie wkurzał siebie
- Abyś nie dodawał sobie pracy
- Aby żyło się wszystkim lepiej i zapanował na świecie pokój i harmonia

5. Dobre praktyki

Git nie jest tylko po to żeby przechowywać zmiany w projekcie i je upubliczniać. Git przede wszystkim przechowuje historię zmian dającą olbrzymie możliwości ułatwienia Ci pracy. Jednak do tego potrzebny jest porządek w repozytorium.

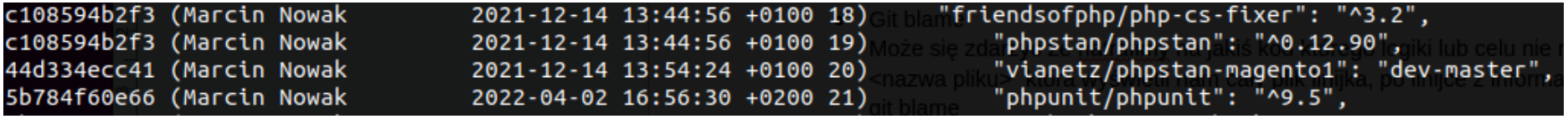
a. Commits

Poniżej jest opisanych kilka dobrych praktyk, które ułatwiają życie. Jak je stosować i jednocześnie nie utrudniać sobie życia, jest opisane w następnym podpunkcie.

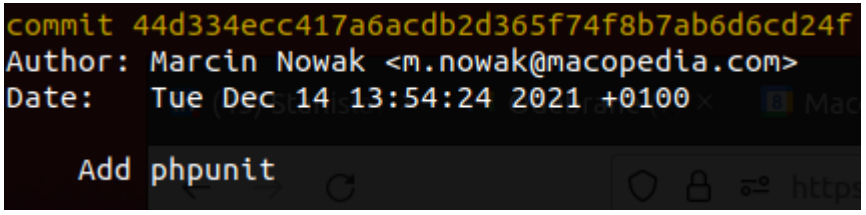
- Tytuł w commitach powinien:

  - opisywać ogólny zarys zmian
  - być zwięzły, ale bez przesady
  - rozpoczynać się od numeru zadania

Może się zdarzyć że natrafimy na jakiś kod którego logiki lub celu nie rozumiemy. Wtedy możemy skorzystać np. z komendy `git blame <nazwa pliku>` która wyświetli nam cały plik linijka, po linijce z informacją kto, kiedy i w jakim commicie zrobił ostatnio zmianę.



Następnie z pomocą `git show <hash commita>` możemy zobaczyć wszystkie zmiany, wraz z tytułem i komentarzem do commita.



Od tego jaki jest tytuł i komentarz oraz czytelność zmian, zależy czy sami dojdziemy co autor miał na myśli, czy będziemy go ścigać aby sobie przypomniał i nam wytłumaczył. Jeśli to świeży commit, to możliwe że odpowie na nasze pytania. Jeśli jednak jest to zmiana sprzed miesiąca, lub więcej, to szanse na to są znikome. Dlatego warto robić komentarze tak żeby ktoś inny nie zawracał nam głowy pytaniami i abyśmy sami nie musieli tracić czasu na przypominanie sobie dlaczego coś zrobiliśmy tak a nie inaczej.

Tytuł nie powinien być rozwlekły, ale powinien opisywać ogólnie czego dotyczyły zmiany. Tytuł typu “WIP”, czy “zmiany zgodne z zadaniem …” nie niosą żadnej informacji.

Przykładowe sytuacje:

Tytuł “Refactoring” nie za wiele nam mówi, poza tym że nie był tworzony nowy mechanizm, ale przerabiany już istniejący. Dokładniejszy opis “Refactoring Entity\User, Entity\Address, Entity\Order, Service\Factory\User, Service\Factory\Order, Service\Factory\Address”, nie dość że jest niezbyt czytelny, będzie ucinany przez gitlaba, to ma limit 72 znaków. Zamiast pisać wszystko w jednej linii, możemy szczegółowy opis dodać w komentarzu (następna linia po wpisaniu tytułu commita, bez limitu znaków).
“Refactoring

- Entity\User
- Entity\Address
- Entity\Order
- Service\Factory\User
- Service\Factory\Order
- Service\Factory\Address”

Dzięki temu przeglądając historię zmian pliku (git blame), lub historię commitów (git log), możemy skupić się najpierw na przeglądaniu tytułów, a potem weryfikacji komentarza. Przykładowo jeśli jakaś funkcjonalność przestała działać, to najpierw powinniśmy szukać commitów z refactoryzacją, a potem sprawdzić czy na liście zmienianych plików jest jakiś powiązany z tą funkcjonalnością.
W komentarzu do commita nie należy opisywać szczegółowo logiki wprowadzonych zmian. Od tego jest opis zadania np. w jira. Tytuły i komentarze są po to aby ułatwić sobie życie w przeglądaniu historii.

- 1 commit = 1 funkcjonalność

Czasami jakaś funkcjonalność przestała działać i chcesz znaleść commity które ostatnio zostały dodane i jej dotyczyły. Dobrze żeby komentarz w pierwszej linii informował jakiej funkcjonalności dotyczy.
Co jeśli commit dotyczy wielu funkcjonalności? I tu jest kolejny ważny temat. 1 commit = 1 funkcjonalność. Dzięki temu komentarz będzie prostszy i czytelniejszy, a zmiany w kodzie będą łatwiejsze do zrozumienia. Nie chodzi aby każdą zmianę w pojedynczym pliku commitować osobno. Przykładowo masz zadanie w którym masz utworzyć nową encję, przygotować kontroler z listą rekordów w tabeli, formularz do dodawania i edycji, migrację, fixtury i testy jednostkowe. Zamiast robić jeden wielki commit, albo kilkanaście drobnych, możemy je pogrupować.

- “[TICKET-121] Add entity User” - Utworzenie encji, podpięcie relacji i dodanie migracji
- “[TICKET-121] Add User grid” - Utworzenie kontrolera z tabelą.
- “[TICKET-121] Add add/edit form for User” - Utworzenie formularza do tworzenia/edycji encji, oraz podpięcie do przycisków edycji i dodawania nowych
- “[TICKET-121] Add translation for User controls” - Dodanie tłumaczeń do pliku z tłumaczeniami
- “[TICKET-121] Add User fixtures” - Dodanie fixture
- “{TICKET-121] Add unit tests” - Dodanie testów jednostkowych

Należy tylko pamiętać, że każdy z tych commitów musi być działający, czyli jak przejdę na niego to aplikacja nie będzie zwracać błędów. W powyższym przykładzie w commicie “[TICKET-121] Add User grid” ma być sama tabelka, bez przycisków do dodawania i edycji.

Może się zdarzyć że z jakiegoś powodu będziesz zmuszony fizycznie nadpisać jaką klasę, lub metodę i wprowadzić zmiany. Z pewnością czytelniej będzie jak najpierw zrobisz commit dodający nowy plik w wersji oryginalnej, a dopiero w drugim dodasz potrzebne zmiany.
Jeśli podczas realizacji zadania, zauważysz że metoda z której korzystasz wymaga refactoringu, to dobrze jest realizowane zmiany wrzucić do osobnego commitu. Dzięki temu przy Code Review, łatwiej będzie oddzielić nową logikę od samego refactoringu.

Dzięki tej zasadzie:
- łatwiej odnaleźć kommit powodujący błędy
- można wycofać część funkcjonalności która powoduje błędy
- łatwiej robi się Code Review

b. Branche

- Nie rób nadmiarowych branchy **na wspólnym repozytorium**

Zdarza się że podczas pracy na feature branchu coś poszło nie tak. Np. potrzebujesz funkcjonalność z czyjegoś brancha (np. frontent wymaga backendu) który jeszcze nie jest domergowany do głównego brancha i próbujesz domergować go do swojego. Albo po zrobieniu rebasa wobec głównego brancha, aplikacja przestała działać. Zamiast naprawić problem, albo “zresetować” brancha, niektórzy tworzą nowego brancha z jakimś dopiskiem. I potem mamy kilka branchy z tym samym numerem zadania, ale z różnymi dopiskami.
Zamiast tego “zresetuj” branch `git reset –hard origin/<nazwa brancha>’.
Jeśli już wypchnąłeś “popsutego” brancha do wspólnego repozytorium, to nie twórz nowego brancha. Usuń go `git remove -D <nazwa brancha>` i utwórz go ponownie. Jak już wszystko będzie działać, to wypnij go nadpisując popsutą wersję `git push –force --set-upstream origin <nazwa brancha>`. Dzięki temu nikt nie będzie się zastanawiał który branch jest dobry.

6. Przydatne metody

a. Tabulator

Tabulator, czyli autouzupełnianie. Dzięki niemu nie będziesz robić literówek, nie musisz pamiętać całych komend i jej parametrów, nie musisz kopiować nazw branchy. Wystarczy trochę praktyki a zobaczysz jak bardzo ułatwia życie.

b. add .

Dodaje wszystkie zmodyfikowane pliki do przechowalni (staging area). Nie musisz dodawać każdego pliku osobno.

c. add *

Działa jak powyższa komenda, ale bez dodawania plików i katalogów zaczynających się od kropki.

d. commit -am

`a` w tym przypadku oznacza że git przed dokonaniem commita, ma dodać wszystkie zmiany do przechowalni. Należy pamiętać że działa to tylko na plikach już wersjonowanych (uprzednio dodanych do repozytorium). Czyli jak utworzysz nowy plik, to nie zostanie on dodany.

e. stash

Metoda ta pozwala na wycofanie wszystkich zmian bez ich utraty. Należy pamiętać że dotyczy to tylko plików “wersjonowanych” (git add).
Stash ma dużo opcji ale tutaj skupimy się na podstawowych.
Aby przywrócić zmiany wystarczy dodać parametr `pop` (`git stash pop`), który przywraca zmiany i usuwa je z listy.
Kiedy używać stash’a piszę tutaj.

f. checkout -

Parametr `-` oznacza poprzedni branch na którym byliśmy.

g. wyświetlanie aktualnego brancha

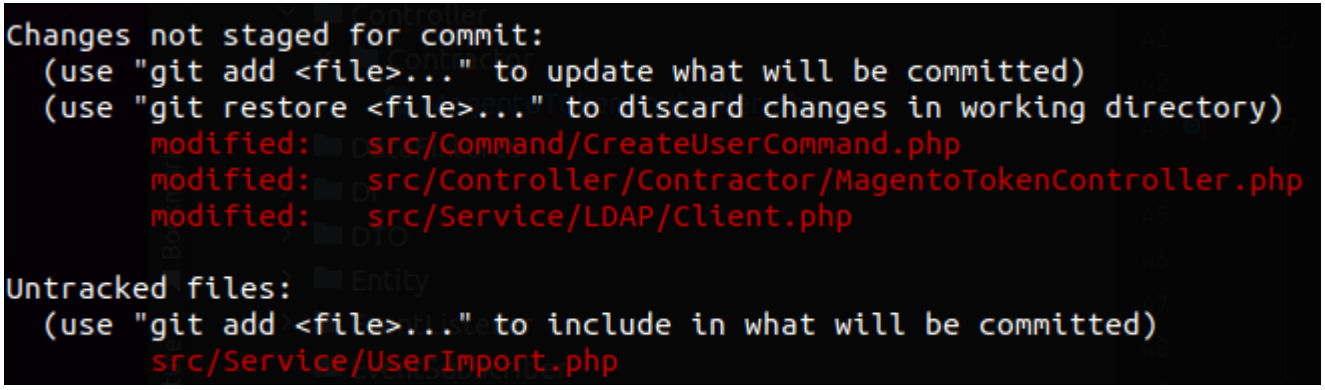
Jeśli pracujesz z gitem w konsoli, warto ustawić wyświetlanie aktualnego brancha .

Jeśli pracujesz na linuxie i nie masz żadnych nakładek to wystarczy że na końcu pliku `~/.bashrc` wrzucisz poniższy kod:

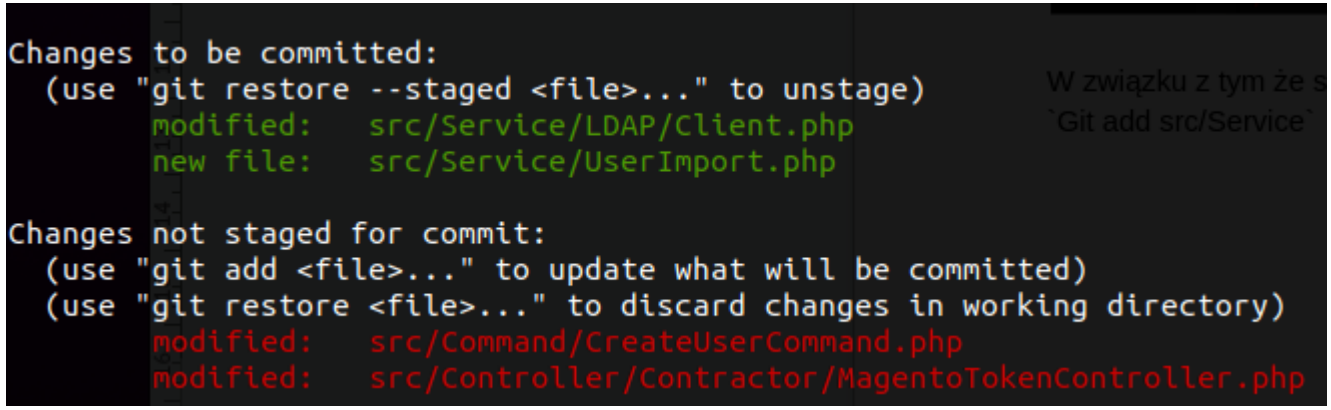
```
parse_git_branch() {
        git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
export PS1="\u@\h \[\033[32m\]\w\[\033[33m\]\$(parse_git_branch)\[\033[00m\] $ "
```

h. add <katalog>

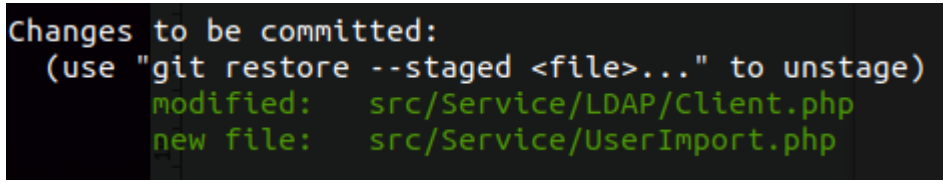
Czasami zdarzy się że z jakiegoś powodu chcemy cofnąć zmianę w dużej liczbie plików. Zamiast ręcznie kopiować je do metody `git checkout`, można spróbować wykorzystać przechowalnię.
Poniżej przykład gdzie utworzono i zmodyfikowano serwis, a dla testów zmodyfikowano komendę i kontroler.



W związku z tym że serwisy mają wspólną ścieżkę `src/Service` w której nie ma komendy i kontrolera, możemy je dodać na jej podstawie.
`Git add src/Service`

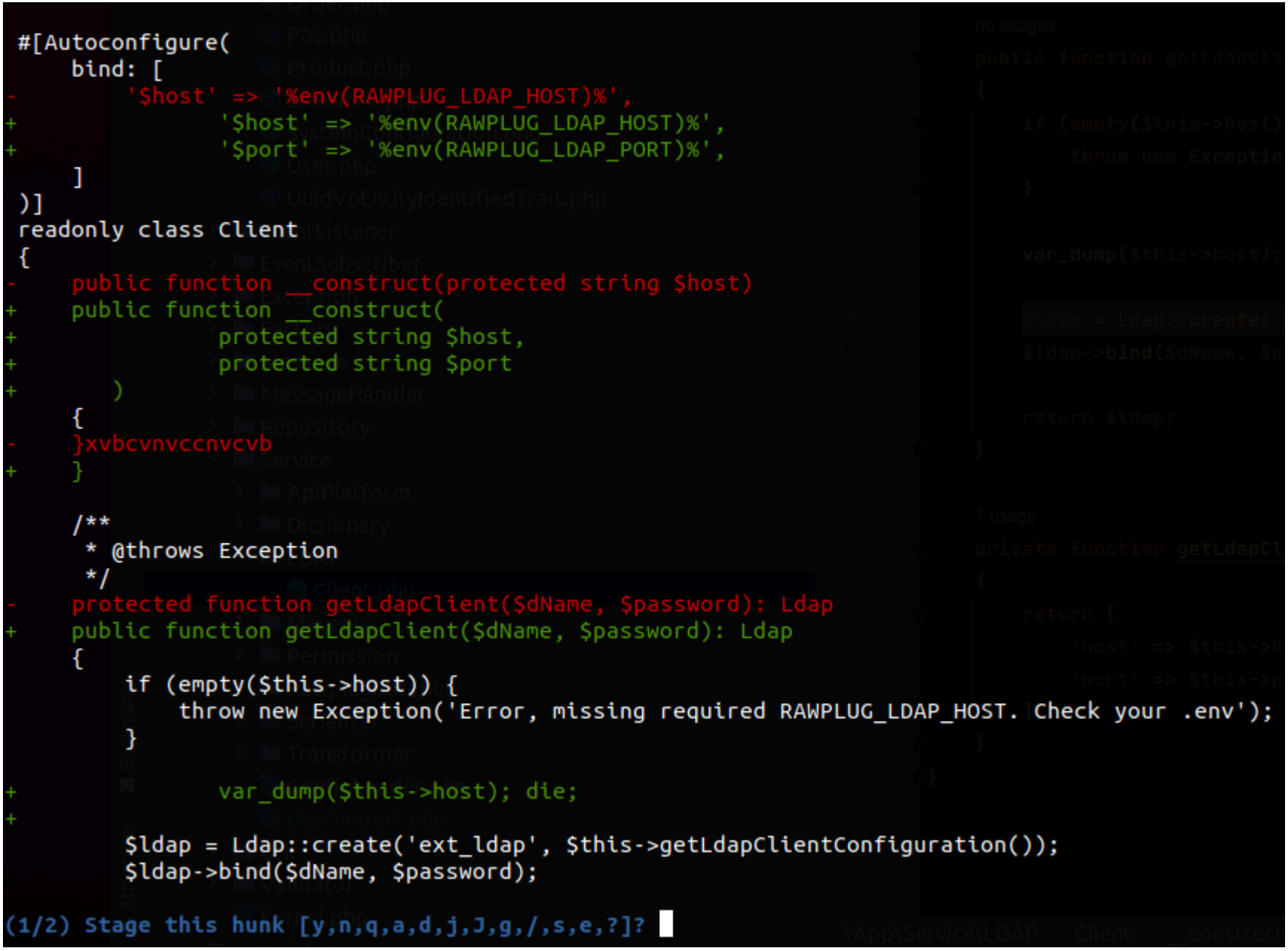


A następnie po prostu cofnąć wszystkie zmiany spoza przechowalni `git checkout .`



i. add -p

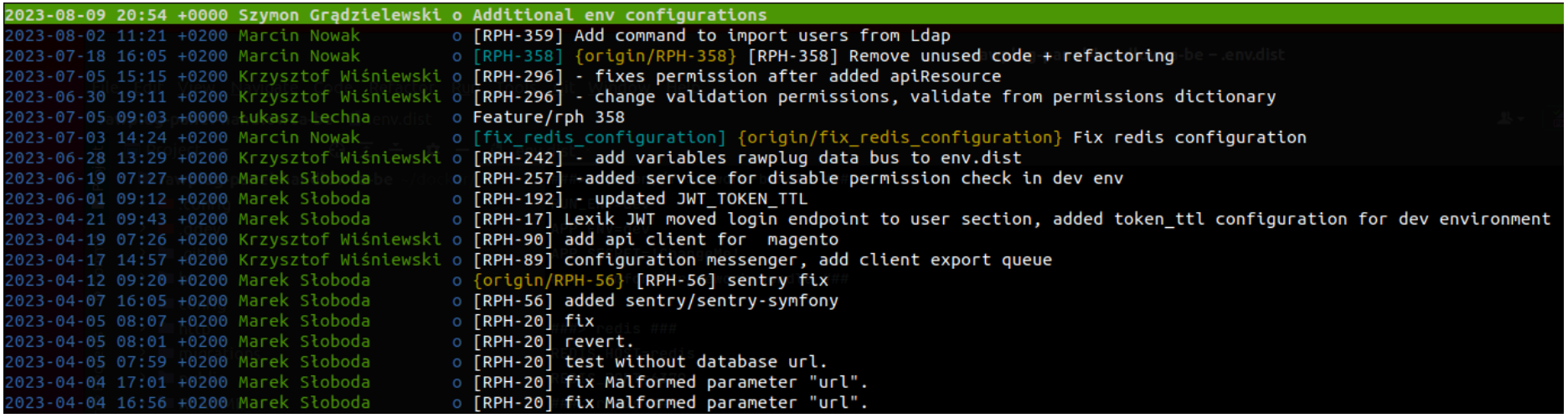
Może się zdarzyć że nie chcemy od razu wszystkich zmian w pliku dodawać do przechowalni. Np. aby rozdzielić zmiany do różnych commitów. Dodawanie plików do przechowalni z parametrem `p` powoduje że git rozdziela zmiany na części i przy każdej pyta się co z nią zrobić.
y - powoduje dodanie całej paczki zmian do przechowalni
n - pomija zmiany
q - anuluj dodawanie
s - rozdziela na mniejsze części
e - pozwala ręcznie zedytować co zostanie dodane z przedstawianej paczki
? - wyświetla listę wszystkich opcji wraz z krótkim opisem



```
 #[Autoconfigure(
     bind: [
-        '$host' => '%env(RAWPLUG_LDAP_HOST)%',
+                '$host' => '%env(RAWPLUG_LDAP_HOST)%',
+                '$port' => '%env(RAWPLUG_LDAP_PORT)%',
     ]
 )]
 readonly class Client
 {
-    public function __construct(protected string $host)
+    public function __construct(
+                protected string $host,
+                protected string $port
+        )
     {
-    }xvbcvnvccnvcvb
+    }

     /**
      * @throws Exception
      */
-    protected function getLdapClient($dName, $password): Ldap
+    public function getLdapClient($dName, $password): Ldap
     {
         if (empty($this->host)) {
             throw new Exception('Error, missing required RAWPLUG_LDAP_HOST. Check your .env');
         }

+                var_dump($this->host); die;
+
         $ldap = Ldap::create('ext_ldap', $this->getLdapClientConfiguration());
         $ldap->bind($dName, $password);

(1/2) Stage this hunk [y,n,q,a,d,j,J,g,/,s,e,?]?
```

j. tig

Tig to narzędzie do interaktywnego przeglądania historii gita z konsoli. Po odpaleniu w konsoli ‘tig’ wyświetli się lista commitów z ich datą, autorem, komentarzem i uproszczonym drzewkiem branchy. Strzałkami ↑ i ↓ zmiania się zaznaczony commit. Enterem otwiera się okno podglądu commita. Przycisk ‘Q’ powoduje wyłączenie podglądu. W otworzonym podglądzie można przesuwać się w dół za pomocą entera o jedną linię, lub ‘Pg dn’ i ‘Pd up’ o całą stronę w dół, lub w górę.
Moim zdaniem główną zaletą jest możliwość przejrzenia historii konkretnego pliku. W tym celu przekazujemy jako parametr adres pliku
`tig <adres pliku>`



Dzięki temu możemy nie tyle zobaczyć kto i kiedy zmienił daną linijkę, ale możemy prześledzić całą historię zmian konkretnego pliku. Przydaje się to w szczególności gdy podczas rozwiązywania konfliktów, lub przy automatycznym mergowaniu “zniknie” jakiś fragment kodu.

7. Jak utrzymać porządek w repozytorium

a. Poprawianie commita

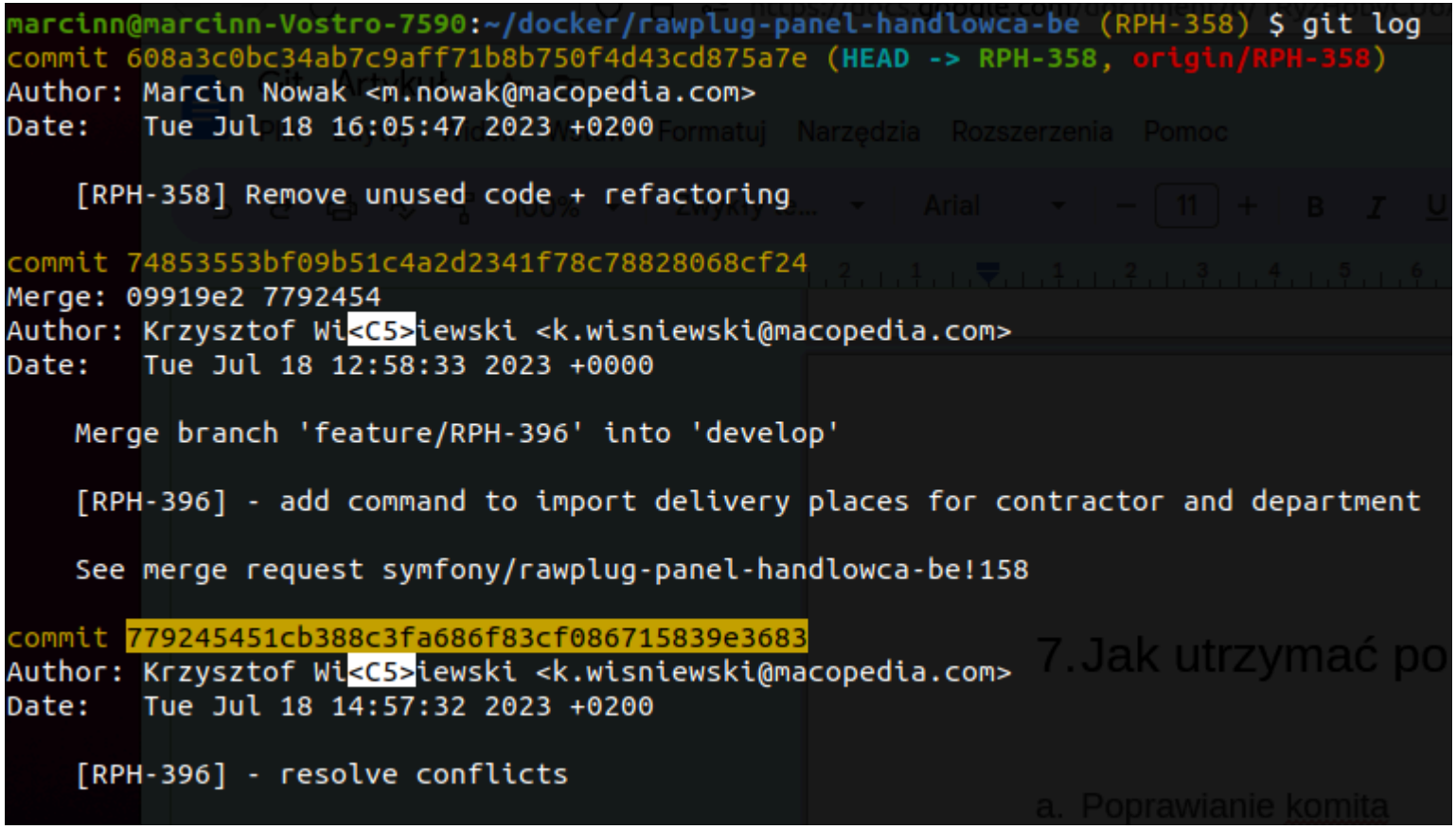
Okazało się że w commicie znajduje się plik którego nie powinno być, albo dopiero po zrobieniu wszystkich commitów odpaliłeś CS Fixera i masz zmiany do dodania, to masz dwie opcje.

○ Ostatni commit

Jeśli zmiany chcesz dodać do ostatniego commita, to wystarczy że dodasz je do przechowalni, a następnie zrobisz commita z parametrem `ammend`. Wtedy git nie utworzy nowego commita, ale doda zmiany do ostatniego wraz z możliwością zmiany tytułu i komentarza.
Parametr ten pozwala zmieniać tytuł i komentarz nawet jeśli nie ma żadnych zmian w przechowalni.

○ Któryś z poprzednich commitów

W przypadku zmian dla starszych commitów, sytuacja nieco się komplikuje. Tak jak zwykle dodaj zmiany do przechowalni. Wyszukaj w historii commita którego chcesz zaktualizować i skopiuj jego hash.



Następnie zcommituj zmiany z parametrem `fixup`

`git commit --fixup=779245451cb388c3fa686f83cf086715839e3683`

i zrebasuj `git rebase --interactive --autosquash 779245451cb388c3fa686f83cf086715839e3683^`, lub `git rebase --i --autosquash 779245451cb388c3fa686f83cf086715839e3683^`. Jeśli się zdarzą konflikty, to je rozwiąż. Masz już poprawionego commita.

b. Usuwanie zbędnych commitów

○ Ostatnie commity

Jeśli chcemy usunąć z historii brancha kilka ostatnich commitów, wystarczy wykorzystać metodę `git reset –hard ~<ilość commitów do usunięcia>`. Należy pamiętać że commity mergujące są inaczej interpretowane przez gita, więc wynik może być inny niż oczekiwany.

○ commity z historii

Jeśli chcemy usunąć jakiś commit który nie jest ostatnim w historii brancha, możemy użyć do tego metody rebase.

`git rebase --interactive <hash commita>^`

Parametr `interactive` spowoduje że wyświetli nam się edytowalna lista commitów, zaczynająca się od podanego przez nas commita, w kolejności od najstarszego. Przed każdym commitem znajduje się napis `pick` co oznacza że commit normalnie ma być brany pod uwagę w historii. Aby dany commit został usunięty z historii zamiast `pick` musimy wpisać `d` (od `drop`). Opis wszystkich dostępnych komend znajduje się pod listą commitów. Należy jednak pamiętać, że jeśli w późniejszych commitach znajdują się zmiany powiązane z tym z usuwanego commita, to będziemy musieli rozwiązać konflikt.

○ Gdy dużo konfliktów

Jeśli podczas rebasowania występuje dużo konfliktów, lub gdy poza usunięciem, chcemy jeszcze dodać po drodze dodatkowy commit, to możemy skorzystać z metody cherry-pick. Spisujemy sobie hashe commitów które chcemy zachować, tworzymy nowy branch z bazowego brancha i dodajemy w dowolnej kolejności branche `git cherry-pick <hash brancha>`. Po każdym cherry-picku możemy dodać jakiś nowy commit.

c. Wyłączenie części zmian

Może się zdarzyć że chcemy usunąć jakiś commit, ale nie możemy modyfikować historii brancha, lub jest on na tyle głęboko w historii, że jego usunięcie powoduje mnóstwo konfliktów do rozwiązania. Wtedy z pomocą przychodzi metoda `revert`.
`git revert <hash commita>` spowoduje utworzenie nowego commita ze zmianami odwrotnymi do tych w podanym komicie. Jeśli jakaś funkcjonalność którą chcemy wyłączyć jest w kilku commitach, to należy pamiętać aby revertować w kolejności od najnowszego do najstarszego commita, aby uniknąć konfliktów.
Jeśli w przyszłości będziemy chcieli przywrócić daną funkcjonalność to możemy po prostu usunąć commit revertujący.

d. Jak posprzątać branch z nadmiaru commitów

○ Miękkie wycofanie commitów

Podczas pracy może być nam szkoda czasu na robienie commitów w sposób uporządkowany. Dlatego jak już skończyliśmy pracę nad zadaniem, możemy wycofać wszystkie utworzone na szybko commity, nie tracąc zmian i utworzyć commity zgodnie z dobrymi praktykami. Do tego służy metoda reset z parametrem `soft`. Dzięki niej usuwamy podaną ilość ostatnich commitów, przy jednoczesnym przerzuceniu ich zmian do przechowalni.

`git reset –soft HEAD~<ilość commitów>`

Następnie z pomocą parametru `p` metody `add` możemy tworzyć commity według uznania.

- Mergowanie/squash commitów

Jeśli z jakiegoś powodu chcemy połączyć kilka commitów, będących jeden po drugim w historii, w jeden, to możemy skorzystać z metody rebase.
`git rebase –interactive <hash commita do którego chcemy zmergować pozostałe>`
Przy commitach które chcemy zmergować musimy zmienić przypis `pick` na `squash` (jeśli chcemy zachować komentarza commita), lub `fixup` (jeśli nie chcemy mergować komentarza commita).

e. Wypychanie poprawionych commitów do repozytorium

Jeśli commit który chcemy poprawić został wcześniej wypchnięty do wspólnego repozytorium, to zanim zaczniemy poprawiać należy poinformować wszystkich w projekcie że chcesz zmodyfikować historię brancha i poprosić żeby nikt nic do niego nie wypychał. Na wszelki wypadek utwórz nowy branch na którym utworzysz commita ze wszystkimi zmianami. Następnie pobierz najnowszą wersję brancha. Po wykonaniu zmian, zmieni się historia w stosunku do tej istniejącej w głównym repozytorium. W związku z tym, gdy sprawdzisz status (git status), to zobaczysz komunikat podobny do poniższego

```
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)
```

Aby ujednolicić historię musisz wypchnąć swoje zmiany z parameterme `force` (git push –force), co nadpisze dotychczasową historię twoją wersją. Teraz poinformuj pozostałych że muszą odświeżyć swoje historie brancha poprzez `git fetch`, `git rebase origin/<nazwa brancha>` i rozwiązać ewentualne konflikty.
Jest to dosyć kłopotliwa i niebezpieczna czynność, więc zastanów się czy na pewno możesz to zrobić.
Dobrą praktyką jest ustawienie blokad na branchach bazowych, tak aby nie można było do nich nic pushować, a jedynie mergować. Wtedy nawet przez przypadek co najwyżej popsujemy jeden branch rozwojowy.

f. Stash VS Commit

Czy zawsze, kiedy potrzebujemy na chwilę “odłożyć zmiany” aby później do nich wrócić, należy je stashować? Teoretycznie nie ma w tym nic złego. Jednak stash ma tendencję do puchnięcia, bo często zapominamy o zmianach które w nim trzymamy, albo podczas ich przywracania następuje konflikt i po jego rozwiązaniu nie usuwamy stasha. Dlatego proponuję, gdy zmieniamy wątek (przechodzimy na innego brancha) stosować commita. Zawsze po powrocie możemy go usunąć nie tracąc zmian (pkt 7 d). Stasha zaś stosować w jednym wątku, gdy np. chcemy zrobić pula, lub rebasa; przetestować mechanizm bez dotychczasowych zmian.

g. Rebase VS Merge

Czasem zdarza się że podczas rebasowania kilkukrotnie musimy rozwiązywać te same konflikty. Prowadzi to niektórych do wniosku, że lepiej jest mergować, bo wtedy rozwiązujemy wszystkie konflikty jednorazowo.
Powoduje to jednak spore komplikacje:

- Przy próbie mergowania takiego brancha do brancha źródłowego, otrzymamy te same konflikty, ponieważ w obydwu branchach będą te same zmiany ale w różnych commitach
- W MR gitlab wyświetli wszystkie domergowane zmiany jako nowe
- Może to doprowadzić do sytuacji w której kod który został świadomie usunięty z repozytorium, po domergowaniu takiego brancha, ponownie się pojawi.