**GIT - zaawansowane koncepty**

**Wstęp**

GIT nie jest prostym narzędziem. Wymaga od młodszych programistów godzin poświęcenia i prac, aby być w stanie łatwo i efektywnie korzystać z dobrodziejstw omawianego systemu kontroli wersji. Zauważyłem, że GIT sprawia najwięcej kłopotów osobie niedoświadczonej w pracy zespołowej. Mogą się wówczas pojawić problemy związane z zsynchronizacją pracy, pracowaniem na wielu gałęziach kodu, rozwiązywaniem **merge konfliktów**, **tworzeniem i zatwierdzeniem Merge Requestów**, aby przeprowadzać skuteczne Code Review zgodnie z duchem **Git** oraz **GitLab Flow**, czy też **cofaniem błędnych commitów**. Jak sam widzisz, konceptów jest wiele, dlatego rozbijemy to wszystko na czynniki pierwsze i skutecznie poznamy każde z zagadnień.

**Praca z wieloma gałęziami kodu**

Do tej pory pokazywałem Ci, jak pracować z GIT-em, gdy wyróżniamy jedną główną gałąź nazywaną **main** (lub **master**). Ciekawym i bardzo użytecznym konceptem, które zapewnia omawiany VCS jest praca z wieloma gałęziami (branchami) projektu.

Przypomnę Ci, że typowym workflowem dla pracy w teamie jest “koło życia”. Wszystkie zmiany, które są ważne dla nas i całego projektu trzymamy na zdalnym branchu “master”/”main” on też się znajduje na zdalnym repozytorium. To kod, w przypadku którego mamy pewność, że jest poprawny i nie uszkodzi nam aplikacji.

Niejednokrotnie w programie zachodzi jednak potrzeba tworzenia, tzw. **Pull Requestów** (nazywanych też **Merge Requestami**). Polega to na tym, że developerzy dostarczają pewną funkcjonalność na branchu A, a później wystawiają prośbę do innych developerów (np. na GitHubie) w celu połączenia zmian (np. między dwoma różnymi branchami lub też z gałęzią główna projektu).

I tutaj znajdujemy odpowiedź, do czego sprawdzają się gałęzie w projekcie. Umożliwiają one przeprowadzanie prac niezależnie, przez różnych programistów, by następnie połączyć rezultat ich pracy w jedną całość.

Tworzenie nowego brancha przy użyciu GIT-a jest bardzo proste. Wchodzimy do głównego katalogu projektu (tam gdzie znajduje się katalog .git) i uruchamiamy komendę:

| git branch <nazwa brancha> |
| --- |

Przykładowo jeśli naszym zadaniem jest stworzenie modułu do zarządzania użytkownikami w aplikacji, stworzymy brancha o nazwie zgodnej z jego przeznaczeniem:

| git branch feature/Add-exercise-checker-functionality |
| --- |

| **Uwaga!**  Tworzony branch przechowuje niezależną historię commitów. W momencie gdy z brancha X tworzysz branch Y, to dotychczasowa historia commitów z X jest przenoszona na Y, jednak branche te będą miały spójną historię do momentu, kiedy nie wykonasz pierwszego oddzielnego commita na gałęzi Y. Wówczas każda z obu gałęzi będzie przechowywała niezależne i różne historie zacommitowanych zmian. |
| --- |

Oprócz wykreowanej w naszej głowie nazwy dodaliśmy jeszcze na początku prefix **feature/** co spowoduje, że Git zapisze tego brancha w "katalogu" o nazwie feature. Dzięki temu przeglądając w przyszłości spis branchy będziemy od razu wiedzieć, że dany branch dotyczy nowej funkcjonalności, a nie jest na przykład poprawą błędu (**bugfix**).

Bugfixy powinny mieć tworzony osobny katalog o nazwie bugfix. Taka kategoryzacja branchy jest zgodna z dobrymi praktykami i często jest indywidualnie definiowana przez zespoły w ramach **DoD** (Definition of Done). W przypadku firmy, w której obecnie pracuję, zakładamy, że branch, w którym wprowadzamy funkcjonalność oznaczamy prefixem **FEAT**, a każdy, który naprawia daną funkcjonalność w kodzie - **FIX**.

Teraz zwróćmy uwagę na jeszcze jeden fakt. Samo stworzenie brancha nie oznacza, że będziemy mogli od razu rozpocząć pracę nad kodem.

Owszem, stworzyliśmy niezależną kopię (osobną gałąź), ale ona jak narazie jest częścią naszego lokalnego repozytorium. Jeśli chcemy przełączyć na tego brancha kod projektu znajdujący się w working directory, wówczas musimy wykonać komendę:

| git switch <nazwa brancha> |
| --- |

Możliwe jest również wykonanie poniższej komendy w celu przełączenia się między branchami:

| git checkout <nazwa brancha> |
| --- |

Jednak z powodu uniwersalności checkout-a (i możliwości wykorzystywania go do wielu różnych operacji), w celu przełączania gałęzi - zaleca się wykorzystywanie polecenia git switch, a nie git checkout.

W ten sposób kod naszego projektu będzie zawierał dokładnie tą wersję kodu, która znajduje się na branchu w naszym lokalnym repozytorium. W tym miejscu dobrze jest wspomnieć, że istnieje pewne uproszczenie, które pozwala równocześnie **utworzyć nowy branch** jak i **przełączyć się** na niego w working directory za pomocą jednego polecenia. Wystarczy że użyjemy komendy:

| git switch -c <nazwa brancha> |
| --- |

lub

| git checkout -b <nazwa brancha> |
| --- |

**Wprowadzenie zmian na nowym branchu**

Po utworzeniu nowego brancha poleceniem git branch feature/Add-exercise-checker-functionality, wprowadźmy na nim pierwsze zmiany.

Przejdźmy zatem na wskazaną wyżej gałąź (git switch feature/Add-exercise-checker-functionality). Utworzymy na niej przykładowy plik ExerciseChecker.java w naszym repozytorium (który na razie ma tylko jeden główny branch).

Dodamy do tego pliku dowolny interfejs (bez wnikania, co konkretnie będzie zawierał kod).

Następnie wykorzystując dotąd poznaną wiedzę, dodamy plik do **staging area** i ostatecznie go zacommitujemy.

| git add ExerciseChecker.java  git commit -m “[Feature] Add first file” |
| --- |

Pamiętamy jednak, że wszystkie operacje (add, commit) wykonujemy na branchu feature/Add-exercise-checker-functionality. Zatem jeśli po wykonaniu powyższych operacji przełączymy się na branch main (za pomocą komendy git switch), wówczas w naszym kodzie projektu (working area) nie będzie takiego pliku jak ExerciseChecker.java.

**Wystawianie Merge (Pull) Requestów**

No dobrze, to teraz, gdy potrafimy operować już na kilku gałęziach projektu i wiesz, że jest to powszechna praktyka w pracy zespołowej, przyszła pora na zapoznanie się z tym, jak scalać (mergować) ze sobą zmiany oraz, jak wystawiać merge requesty (nazywane też pull requestami) do wprowadzanych zmian.

**Czym są pull requesty?**

Zacznijmy od teorii i zrozumienia, dlaczego w ogóle chcielibyśmy wystawiać merge (pull) requesty. Otóż dostarczanie ciągłych funkcjonalności w Twoim kodzie jest nieustannie koordynowane przez Code Review (proces, w którym inni developerzy sprawdzają Twój kod i dostarczają Ci ewentualne wskazówki co do jego poprawy). Dzieje się to w momencie, gdy chcemy dołączyć nasze zmiany do gałęzi głównej zdalnego repozytorium.

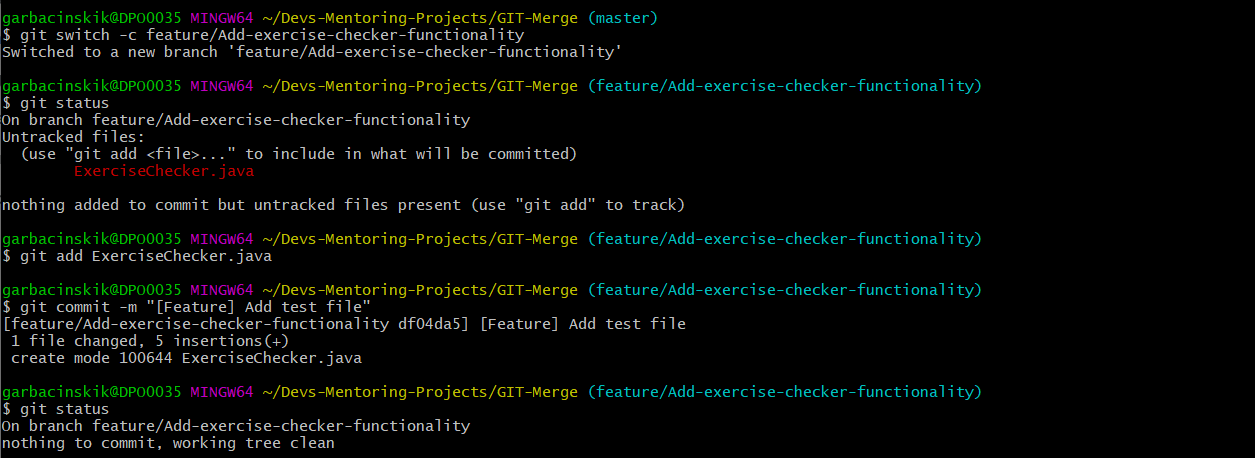
W praktyce wygląda to tak, że po zacommitowaniu swoich zmian na wybranej gałęzi, pushujesz modyfikacje na inny niż main branch - do swojego repozytorium. Wówczas wystawiasz innym programistom merge request i jeżeli zostanie on zaakceptowany przez innych programistów, możesz dołączyć wprowadzoną funkcjonalność na gałąź główną.

**Jak wystawić pull request?**

Z racji, że w pracy ze swoim mentorem często będzie wystawiał pull requesty, aby Twój kod został dogłębnie przeanalizowany i w przypadku braku zastrzeżeń - zmergowany, przedstawiam Ci flow pracy, jak wystawiać omawiane requesty z wykorzystaniem GIT-a i GitHub-a.

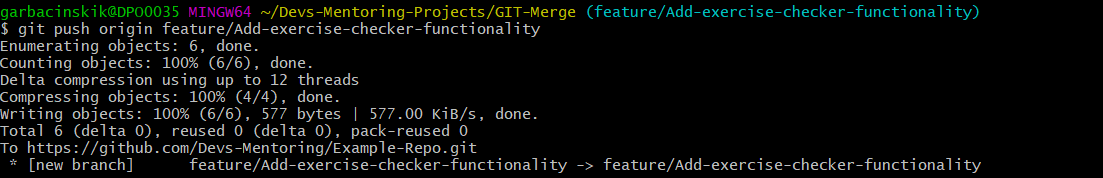
Pierwszym krokiem jest już znany Ci proces - utworzenie odpowiedniego brancha, na którym będziesz pracował i zacommitowania zmian.

Spójrz na przeprowadzone poniżej polecenia (kolejno przeniesienie zmian na nowy branch, dodanie ich do staging area oraz zacommitowanie).



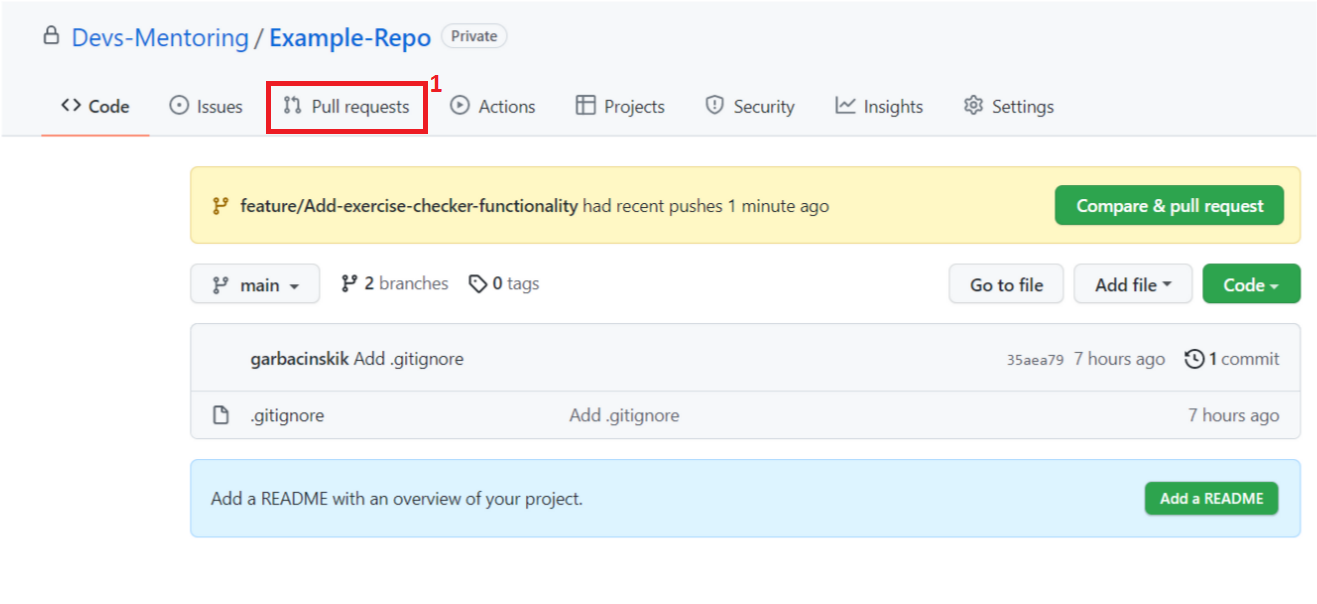
Dodanie zmian na gałęzi

Następnie, będąc na tym samym branchu, wypushujemy zmiany do swojego repozytorium, na oddzielną gałąź. Z poziomu zdalnego repozytorium zajmiemy się ręcznym wystawianiem merge requesta.



Wypushowanie zmian

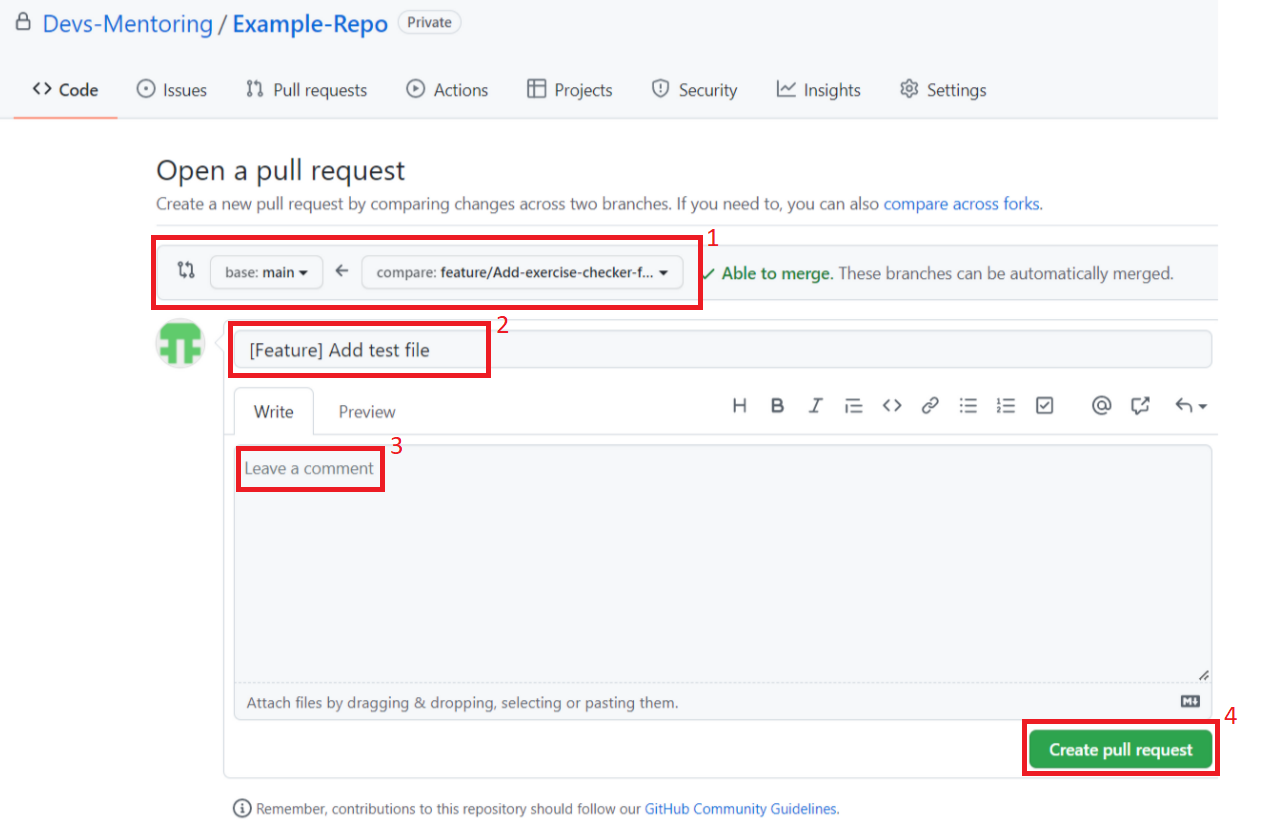
Po tym, jak zaupluoadowaliśmy zmiany, ujrzymy następujący interfejs:



W tym momencie możemy tak naprawdę jednym przyciskiem - **Compare & pull request** - wystawić pull request.

Opcja ta zazwyczaj pojawia się zaraz po wypushowaniu zmian, dlatego jeśli chciałbyś wystawić PR po jakimś czasie od wysłania zmian, możesz przejść do zakładki **Pull requests** (1) i stamtąd ręcznie wystawić prośbę o scalenie gałęzi.

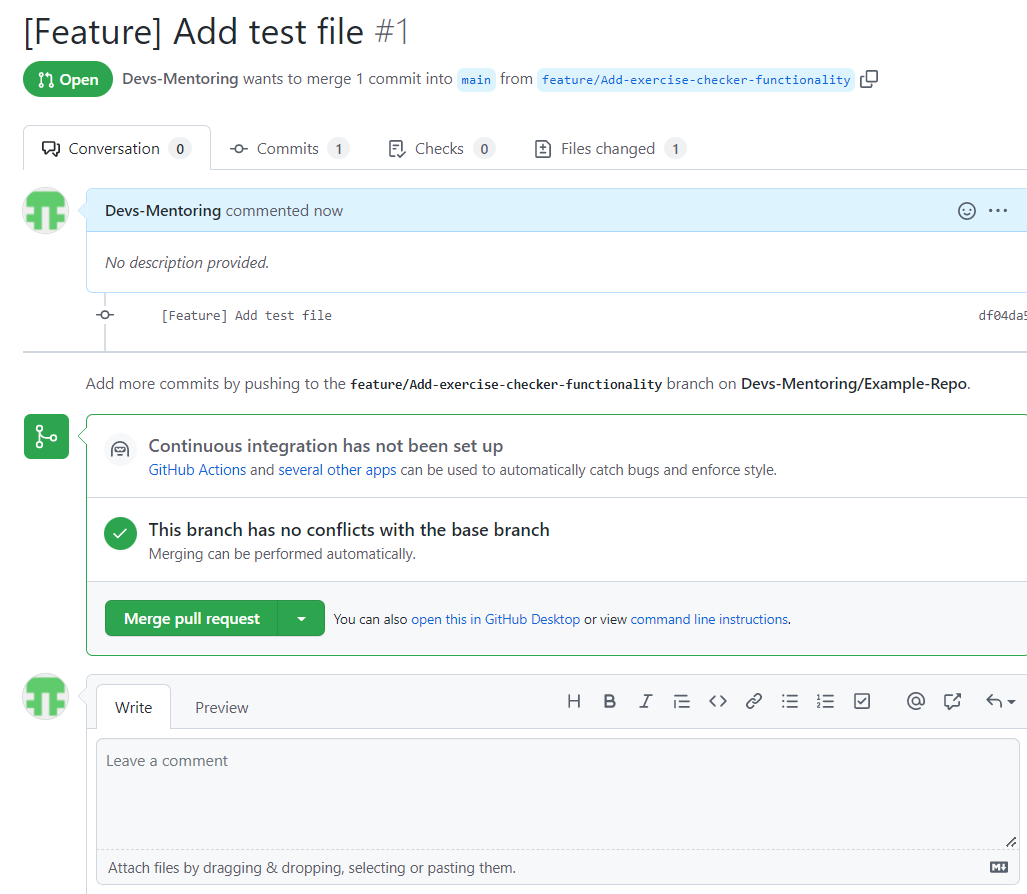
Po wybraniu opcji scalenia ze sobą dwóch gałęzi, ujrzymy następujący interfejs:



W trakcie wystawiania pull requesta możesz określić pod jakim commit messagem ma być dostępny taki request (2). My pozostawiamy bez zmian w formie *[Feature] Add test file*. Pod opcją (1) określamy, z jakiej gałęzi ma nastąpić merge oraz do którego brancha mają być dołączone zmiany. W naszym przypadku merge-a dokonujemy z brancha feature-owego do maina.

Zauważ, że pod cyfrą (3) możemy zostawić również dodatkowe komentarze do naszego requesta, aby w końcu sfinalizować request przyciskiem (4).

Po naciśnięciu tego przycisku ukaże nam się gotowy do review i zmergowania request.



I w tym momencie “do akcji” powinni wkroczyć inni developerzy - w celu zreviewowania Twojego kodu. To, kiedy zmianę będzie można uznać za sprawdzoną i zatwierdzoną przez innych w dużej mierze zależy od wewnętrznej polityki Twojego zespołu. Z mojego doświadczenia mogę Ci powiedzieć, że zazwyczaj wymaga się **co najmniej dwóch zatwierdzeń** od innych programistów. Dopiero wtedy można dołączać naszą zmianę do gałęzi głównej projektu.

Oprócz samego Code Review, we właściwie tworzonym i zarządzanym projekcie powinien pojawić się również system **Continuous Integration**, który w najprostszym wytłumaczeniu - automatycznie weryfikuje, czy wprowadzone przez nas zmiany są poprawne (uruchamiając szereg testów). Wówczas, jeśli zfailowałby któryś z tzw. jobów w obrębie twojego CI, musiałbyś w pierwszej kolejności zająć się naprawieniem odpowiednich błędów.

Po wciśnięciu przycisku **Merge pull request** nasze zmiany zostaną dołączone do maina.

Nic trudnego, prawda? Z merge requestami może pojawić się niestety kilka problemów, które trzeba rozwiązać w pierwszej kolejności, aby móc przeprowadzić proces scalania gałęzi.

Jednym z tych przeciwności jest sytuacja, kiedy to musimy naprawić błędy pojawiające się w obrębie CI. Drugim przypadkiem może być natomiast sytuacja, gdy próbujemy wypushować na mastera takie fragmenty kodu, które zostały wcześniej zmodyfikowane (przed naszym pushel) i dostarczone przez innego programistę. Następuje wówczas, tzw. **merge conflict**.

**Merge Conflicty i ich rozwiązywanie**

Aby dobrze zrozumieć merge conflicty, przeanalizujmy scenariusz, w którym mamy dwóch programistów pracujących nad tym samym kodem. Załóżmy, że programista numer 1 to nasz kolega pracujący w zespole, a programistą numer 2 jesteśmy my.

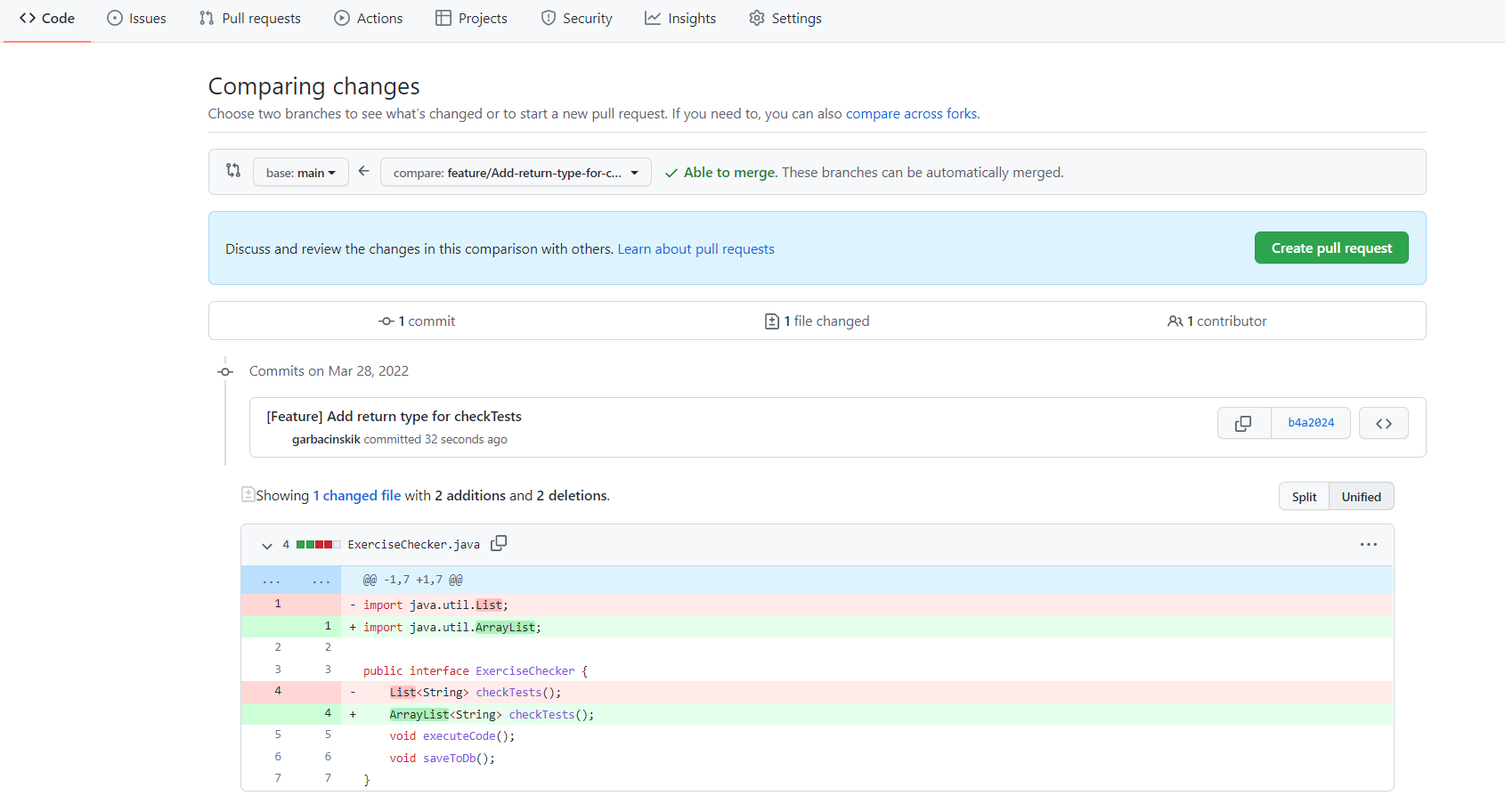
1. Zakładamy, że wypushowaliśmy i zmergowaliśmy już z mainem wcześniejsze zmiany - dodanie interfejsu *ExeciseChecker*. Nastąpiła teraz potrzeba zmodyfikowania typu zwracanego metody *checkTests* na listę Stringów (która reprezentować będzie nazwy uruchomionych testów).
2. Programista nr 1 tworzy zatem nowy branch o nazwie *functionality/Modify-ExerciseChecker*, a programista nr 2 gałąź o nazwie *feature/Add-return-type-for-checkTests* (nazewnictwo gałęzi dowolne).
3. Z racji, że programista nr 1 miał więcej czasu - szybciej dostarczył swoje zmiany i zostały one zmergowane z główną gałęzią projektu. Zmiana, jaką wprowadził w pliku została oznaczona kolorem. Zauważ, że pomylił się on jednak w założeniach i metoda checkTests nie zwraca listy Stringów, tylko jest typem void.

| import java.util.List;  public interface ExerciseChecker {  void checkTests();  void executeCode();  void saveToDb(); } |
| --- |

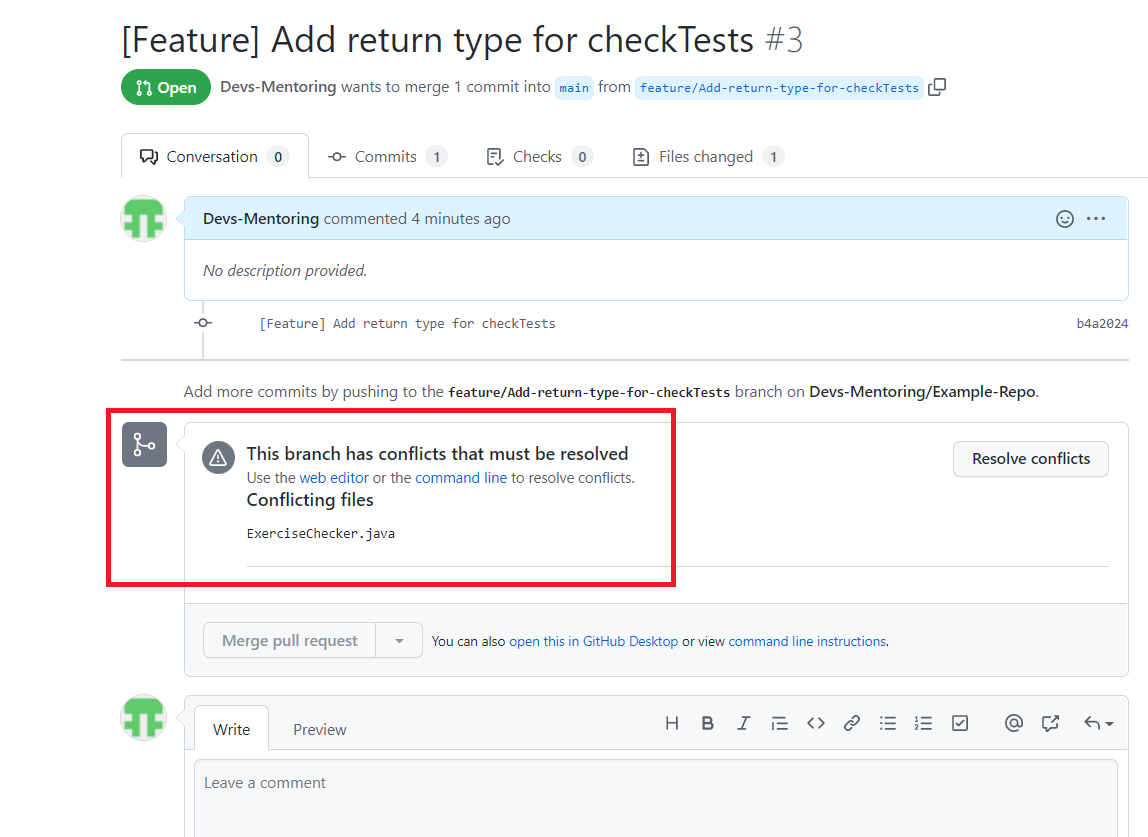
1. Drugi programista również wprowadził zmiany u siebie w kodzie - z tymże napisał nieco inny kod niż programista nr 1. Zamiast typu zwracanego *void* wpisał *ArrayList<String>* oraz zmienił nazwę importowanego zasobu. Tak jak poniżej.

| import java.util.ArrayList;  public interface ExerciseChecker {  ArrayList<String> checkTests();  void executeCode();  void saveToDb(); } |
| --- |

1. Dalsze kroki, jakie podjął programista nr 2 związane są z wypuszowaniem brancha i wystawienie dla niego pull requesta. Po wrzuceniu nowego brancha do repozytorium, otrzymamy następujący widok:



Po kliknięciu przycisku Create pull request, zostaniemy przekierowani do widoku:



Kluczową dla nas informacją jest wiadomość umieszczona w czerwonym prostokącie.

**This branch has conflicts that must be resolved.**

Informacja ta mówi nam o tym, że pojawiły się konflikty w naszych zmianach i musimy je ręcznie rozwiązać. Wynika to właśnie z tego, że programista nr 1 dokonał zmian na gałęzi głównej w tym samym pliku i w tych samych miejscach, co programista nr 2. W tym momencie mamy dwuznaczność zmian i musimy sami zadecydować, czy:

1. Usuwamy zmiany kolegi i zastępujemy je naszymi
2. Czy usuwamy nasze zmiany i zastępujemy je modyfikacjami kolegi
3. Czy idziemy na kompromis i łączymy obie zmiany, tworząc funkcjonalność o wspólnych mianownikach

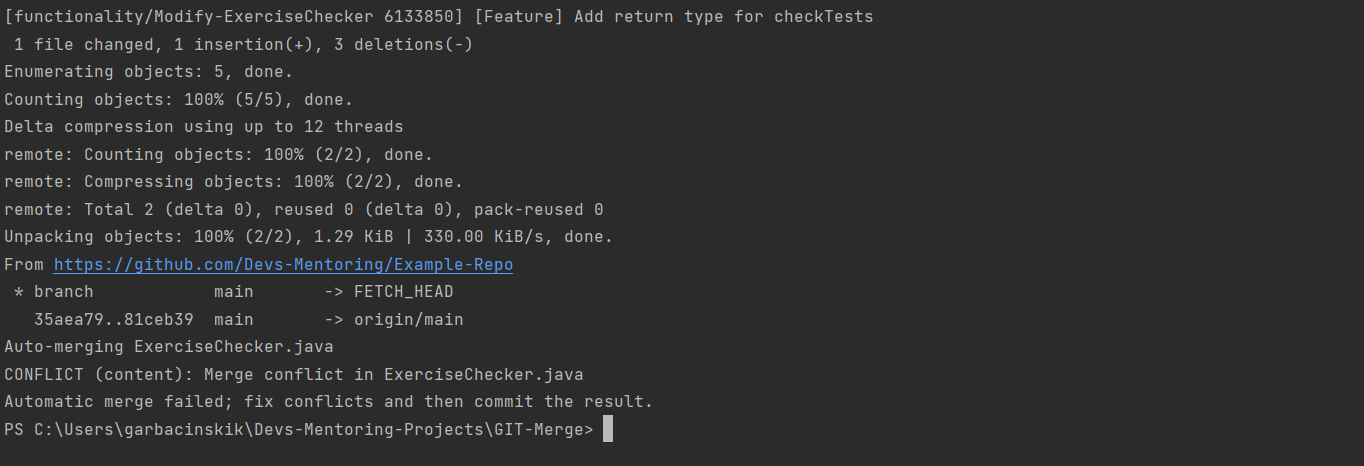
Problemy takie często rozwiązuje się po wzajemnej wymianie opinii, spostrzeżeń i dojściu do konsensusu między autorami danych commitów. Jednym słowem musimy zmienić kod tak, aby nie było w nim żadnej dwuznaczności i prezentować konkretne rozwiązanie.

W powyższym przypadku rozwiązanie merge conflictu jest oczywiste. To programista nr 1 popełnił błąd, dlatego programista nr 2 - domergowujący swoje zmiany, u którego nastąpił konflikt musi wdrożyć swoje zmiany (typ zwracany ma być bowiem reprezentowany przez ArrayList, nie void).

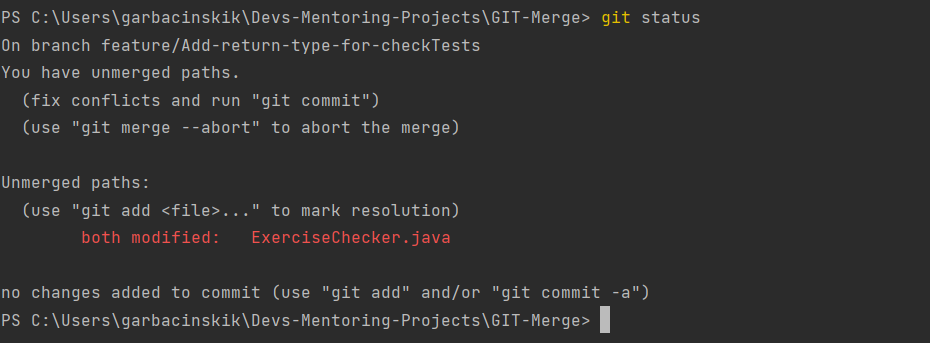
Tego typu konflikty możemy rozwiązywać z poziomu GitHuba lub wygodniej - w swoim lokalnym edytorze kodu. Aby to zrobić musimy pobrać zmiany z maina do siebie lokalnie (poznanym przez Ciebie już poleceniem git pull), a następnie ręcznie rozwiązać napotkane konflikty.

Możemy to rozwiązać zatem w następujący sposób:

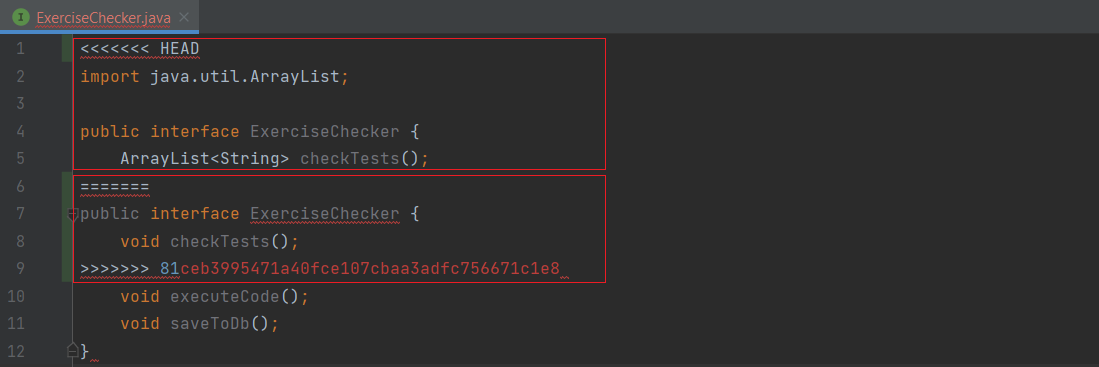
1. Będąc na swoim branchu - *feature/Add-return-type-for-checkTests*, wykonujemy polecenie git pull origin main. Spowoduje to pobranie aktualnej wersji głównego brancha i dołączenie go do gałęzi, na której się aktualnie znajdujemy.
2. Z oczywistych powodów scalenie obu gałęzi nie będzie mogło nastąpić, pojawi się komunikat o merge conflict-cie - tak jak poniżej:



1. Od tego momentu, aby móc sfinalizować scalanie, musimy ręcznie wprowadzić zmiany w plikach, w których nastąpił konflikt. Aby podejrzeć, które zasoby są problematyczne, wpiszemy git status.



1. Widzimy, że plik ExerciseChecke jest “both modified” i to do niego musimy przejść w celu zaradzenia problemowi. Taki plik najłatwiej jest edytować z poziomu Twojego edytora. Gdy otworzymy plik ExerciseChecker.java, ujrzymy następującą postać kodu:



1. Prostokątami oznaczyłem dwa kluczowe miejsca w naszym kodzie. Zauważ, że każde z nich znajduje się pomiędzy **=====** oraz **<<<<<<.** Sekcja znajdująca się w pierwszym prostokącie (nad którą umieszczone jest również słowo HEAD), jest naszą lokalną zmianą, a modyfikacje z drugiego prostokąta - te, które zostały wypuszowane przez programistę nr 1 - na branch main.

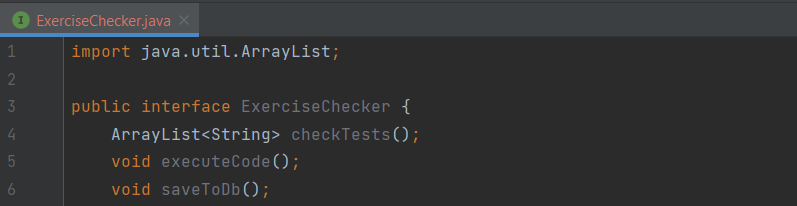
W tym momencie, aby zażegnać merge conflict, musimy ręcznie usunąć ten skrawek kodu z 1-ego lub 2-iego prostokąta, który nie odpowiada naszym oczekiwaniom. W naszym przypadku będzie to:

| ======= public interface ExerciseChecker {  void checkTests(); >>>>>>> 81ceb3995471a40fce107cbaa3adfc756671c1e8 |
| --- |

… oraz pierwsza linia pliku:

| <<<<<<< HEAD |
| --- |

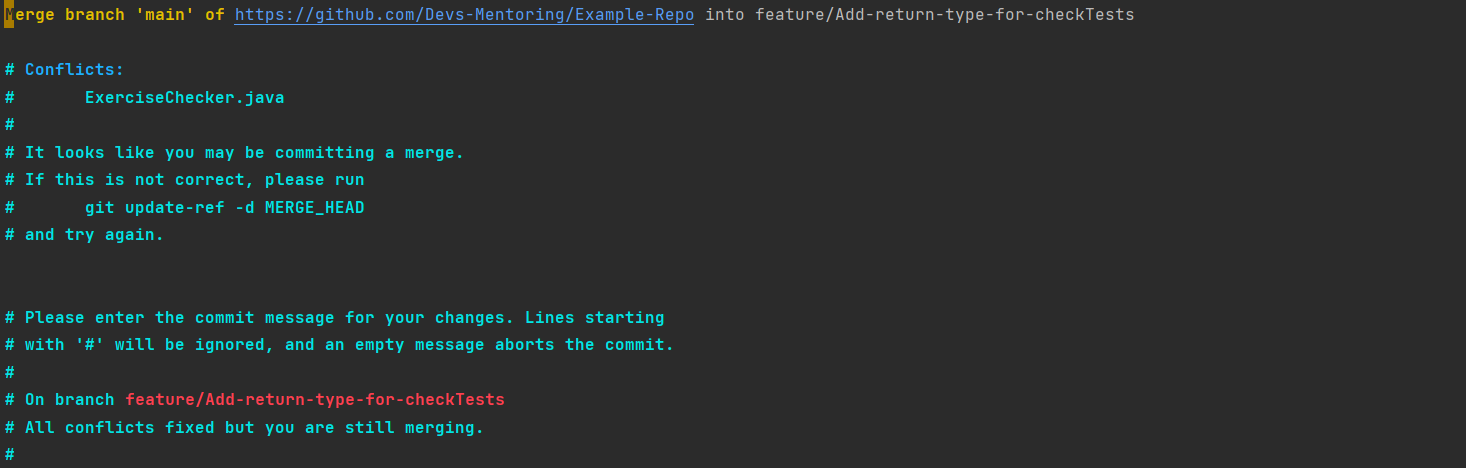
W rezultacie plik po zmianach będzie wyglądał następująco:



7) Następnym krokiem będzie dodanie naszych plików do statusu staged i zatwierdzenie naszej zmiany i kontynuowanie procesu mergowania. Użyjemy zatem następujących poleceń:

| git add ExerciseChecker  git merge --continue |
| --- |

Po rozwiązaniu merge conflictu i scalenia gałęzi, zostanie utworzony nowy commit i ujrzymy okno, w którym będziemy mogli wprowadzić odpowiedni commit message. Ja akurat pozostawiam go bez zmian.

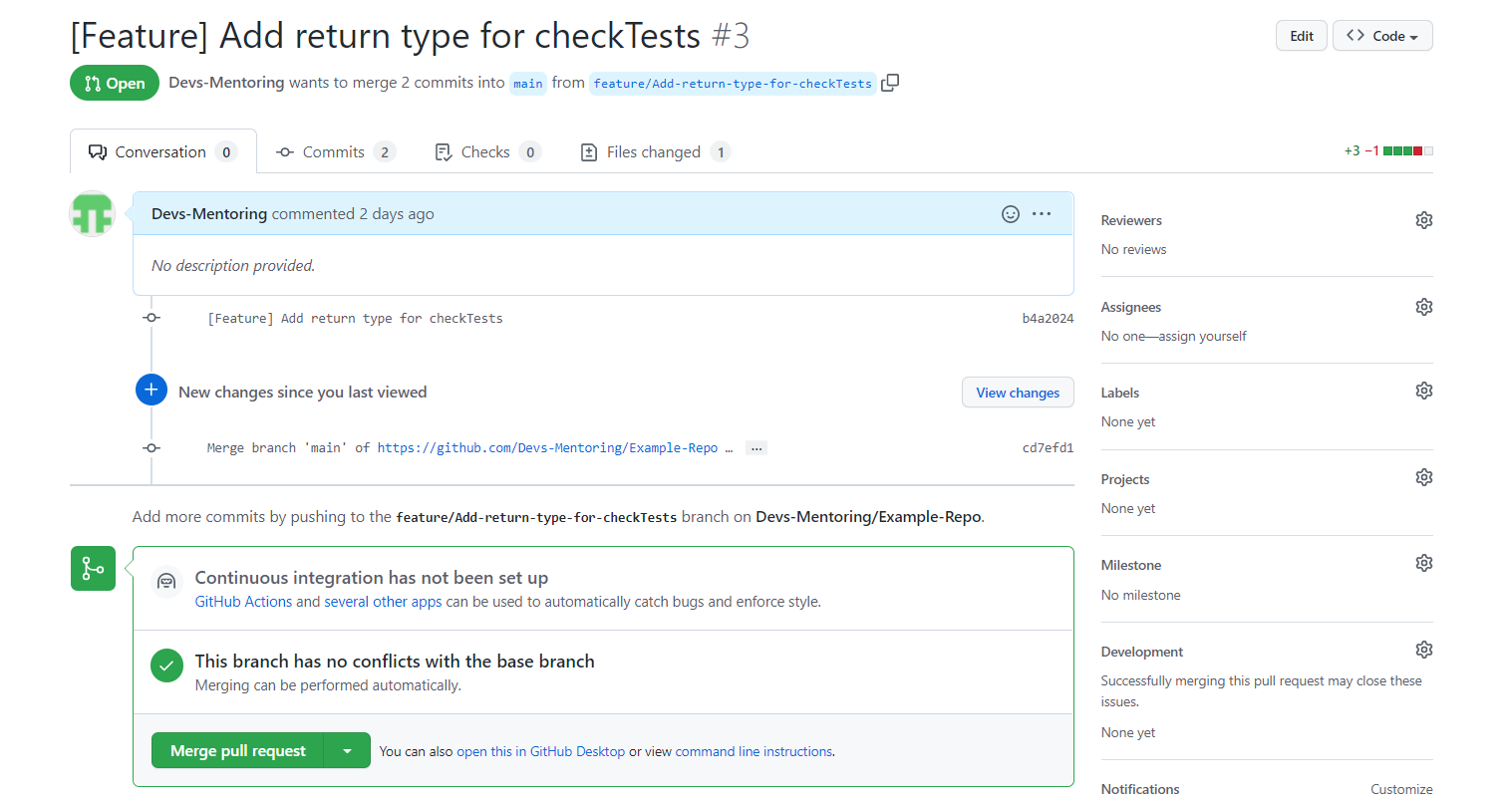


| **Uwaga!**  Jeżeli Twoim domyślnym edytorem commit messagów jest vim, to aby zapisać wprowadzoną wiadomość i wyjść z edytora, użyjemy kombinacji: *Escape + :wq* |
| --- |

Po takich zmianach, możemy jeszcze raz wypushować nasze zmiany na branch feature/Add-return-type-for-checkTests.

git push origin feature/Add-return-type-for-checkTests

Wówczas, gdy przejdziemy z powrotem do widoku na naszą gałąź - ujrzymy:



Widzimy, że merge request dla brancha *feature/Add-return-type-for-checkTests* został zaktualizowany i pozybliśmy się dwuznacznych zmian. Możemy wcisnąć przycisk *Merge pull request* w celu scalenia ze sobą nowej gałęzi i mastera.

Rozwiązaliśmy właśnie nasz pierwszy merge conflict!

**GIT vs GitLab Flow**

Jak widzisz praca z wieloma branchami projektu wymaga nieco doświadczenia i praktyki. Aby uschematyzować cały flow kodowania i dostarczania rozwiązań, w pracy z profesjonalnymi projektami, wprowadza się koncept **Git** lub **GitLab Flow**.

Założenia te opisują, **w jaki sposób** powinniśmy pracować nad swoim projektem oraz **jakie gałęzie projektu** wyróżnić, aby praca przebiegała na najwyższej jakości oraz zgodnie z założeniami profesjonalnej pracy.

We wcześniejszych szkoleniach wspomniałem Ci, że powinniśmy zwracać uwagę na nazewnictwo branchy - tak, aby choćby dodawać do nich odpowiednie prefixy: [REFACTOR], [FEATURE] itd.

Git oraz GitLab Flow dodatkowo rozszerzają koncept pracy z wieloma gałęziami.

Poniżej szczegóły:

**Git Flow**

Model Git Flow polega na odpowiednim uporządkowaniu gałęzi w swoim repozytorium. W tym przypadku będziemy operować na pięciu rodzajach branchy:

* **main (master)** – produkcyjna wersja aplikacji. Do tego brancha będziemy mergować tylko te zmiany, które już zostały wydane na produkcję oraz krytyczne hotfixy.
* **hotfix** – jedyna gałąź bazująca ma masterze. To właśnie ona służy do szybkiego naprawiania krytycznych błędów występujących na produkcji.
* **release** – na tym branchu przygotowywany jest release kolejnej wersji aplikacji. To właśnie wersja aplikacji z tego brancha trafia na produkcję.
* **develop** – gałąź ta jest „nieoficjalnym” masterem podczas pracy nad releasem. Z tego brancha programiści tworzą swoje gałęzie robocze i do niego mergują (rebase-ują) swoją pracę. Gdy praca nad wszystkimi funkcjonalnościami w danym releasie jest gotowa, branch ten jest mergowany do gałęzi „release”.
* **gałęzie robocze (features)** – na tych gałęziach pracujemy na co dzień i tworzymy nowe funkcjonalności.

**GitLab Flow (nazywany również GitHub Flow)**

**Model GitHub Flow przewiduje tylko dwa rodzaje gałęzi:**

* **master (main)** – na tym branchu znajduje się produkcyjna i aktualna wersja aplikacji. Gałąź ta jest zawsze gotowa do tego, aby w razie potrzeby mogła być deployowana na serwery produkcyjne. To z tej gałęzi programiści będą tworzyć swoje branche i do tej gałęzi mergują swoje zmiany.
* **gałęzie robocze (features)** – na tych gałęziach tworzymy nowe funkcjonalności oraz naprawiamy wszelkiego rodzaju błędy w aplikacji.

Jaki sposób na pracę przyjmiemy zależeć będzie głównie od tego, na jaką strategię zdecyduje się Twój cały zespół lub firma. Z doświadczenia mogę powiedzieć Ci, że oba podejścia są skuteczne przy pracy z omawianym systemem kontroli wersji. GIT Flow jest jednak nieco bardziej złożonym podejściem wymagającym od programisty większego usystematyzowania pracy.

Podczas mentoringu i pracy nad projektami, wykorzystywać będziemy GitLab Flow.

**Cofanie zmian**

Ostatnim tematem, który powinienem poruszyć w kontekście efektywnej pracy z GIT-em jest cofanie zmian w repozytorium i przełączanie się między commitami.

W poprzednich szkoleniach zapoznałeś się z takim poleceniem jak git reflog. Przypomnę Ci tylko, że służyło ono do podejrzenia zmian w obrębie lokalnego repozytorium. W pracy z GIT-em niejednokrotnie będzie się pojawiała konieczność wracania do konkretnych commitów i zmian. Nierzadko będziemy chcieli po prostu cofnąć wszystkie dotąd wprowadzone commity i zmiany.

**Git reset**

Omawiane polecenie można wykonać na cztery sposoby, używając argumentów --soft, --hard, --mixed lub bez żadnego argumentu.

W przypadku --soft cofniemy się do wybranego commita, jednakże wszelkie zmiany dokonane po commit nie zostaną usunięte. Plik ze zmianami będą widoczne jako "zmiany do potwierdzenia".

Gdy użyjemy opcji --mixed lub nie podamy argumentu, to pliki zostaną zachowane, ale nie zostaną one zgłoszone do zatwierdzenia.

Mamy także opcję --hard, która robi najlepiej, to co chcemy, jeśli chodzi o cofanie zmian, ale jest niebezpieczna. Gdy użyjemy tej opcji, to cofamy się do wybranego commita i kasujemy wszelkie zmiany, jakie były dokonane po tym commit-cie.

Pora na przykład w konsoli:

| git init  git commit -m "First commit"  touch config1.xml  git add config1.xml  git commit -m "Add config1.xml"  touch config2.xml  git add config2.xml  git commit -m "Add config2.xml" |
| --- |

Przykład jest dość bazowy i trywialny, jednak dobrze ukaże możliwości, jakie niesie za sobą polecenie git reset.

Po wykonaniu powyższych poleceń reflog prezentuje się następująco:



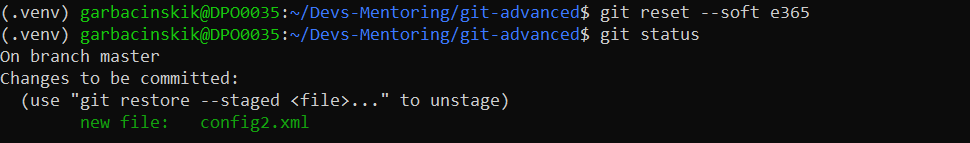
Z takiego refloga płyną ważne informacje.

1. Możemy odczytać unikalne ID (hashe) commitów z listy
2. Obok hashy commitów widzimy również, gdzie umieszczony jest HEAD. Atrybut ten wskazuje, na którym commitcie obecnie się znajdujemy.
3. Z refloga możemy odczytać również commit messages, które mówią nam, z jaką wiadomością została zacommitowana dana zmiana.

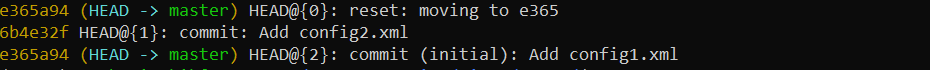
**Git reset --soft**

Przetestujmy w pierwszej kolejności użycie git reset --soft na naszych commitach. Zresetujmy zatem bieżący status lokalnego repozytorium do commita e365a94.

Polecenia:

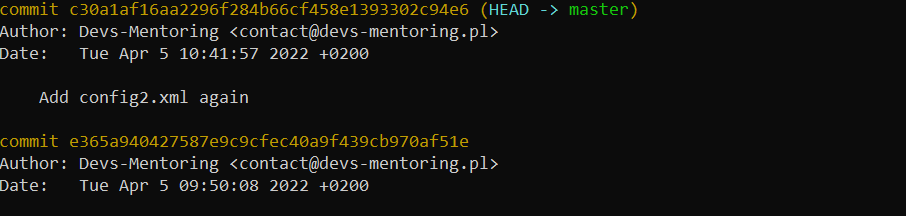


Reflog:

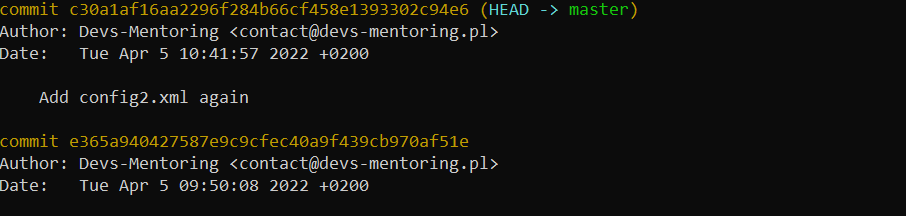


Kluczowe jest to, że po wykonaniu resetu, faktycznie cofnęliśmy się do stanu projektu sprzed kilku commitów, a zmiany z aktualnego commita (config2.xml), zostały przeniesione do statusu staged w repozytorium.

Gdybyśmy chcieli teraz ponownie zacommitować nasze zmiany, ale z nieco zmodyfikowaną commit wiadomością…



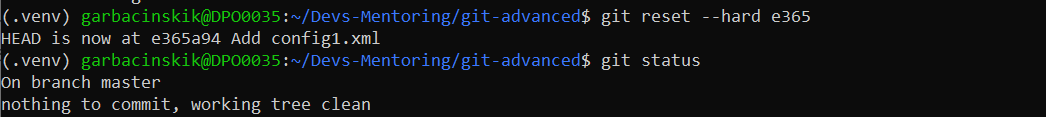
… to nasz ostatni commit zostanie zastąpiony nowym, a historia commitów (git log) będzie wyglądała następująco.



Jak widzisz, nie ma w nim już wcześniej dodanego commitu (“Add config2.xml”), na którym wykonaliśmy polecenie git reset.

**Git reset --hard**

Przetestujmy teraz działanie polecenia git reset --hard. Jak już wspomniałem we wstępie, jest ono o wiele groźniejsze niż zwykłe cofanie zmian z flagą --soft. A to za sprawą tego, że twardy reset nie zachowuje zmian wprowadzonych z najświeższym commitem i całkowicie kasuje on wszystkie zmiany z commita.



Mogę teraz zacząć pracę - tak jakby nie dodał pliku ‘config1.xml'.

Dopóki nie zrobię nowego commita, ta zmiana jest jeszcze odwracalna. Używając "git reflog" mogę zobaczyć historię swoich resetów i wybrać ten commit, na którym pracowaliśmy przed wykonaniem resetu. Jest to przydatne w momencie, gdy wykonamy reset przez przypadek i nieumyślnie skasujemy zacommitowane zasoby.

**Zamiast commit hasha…**

Bądź świadomy, że chcąc cofnąć się w określone miejsce w historii commitów, nie zawsze musimy podawać commit hash przy poleceniu git reset. Opisywany system kontroli wersji umożliwia również relatywne określanie, w które miejsce chcemy się cofnąć w projekcie.

Umożliwia to znak **karety (^)** lub **tylda (~)**. I tak przykładowo, gdybyś chciał cofnąć się do poprzedniego commita, zapiszesz:

* git reset --hard HEAD~1 lub też nierzadko git reset --hard HEAD^1

To, jaki znak w danym momencie wykorzystasz zależeć będzie od powiązań między commitami na danym branchu. W większości przypadków używać będziesz znak tyldy, jednak w momencie, gdy operujesz na zmergowanych commitach, lepiej sprawdzić się może ^.

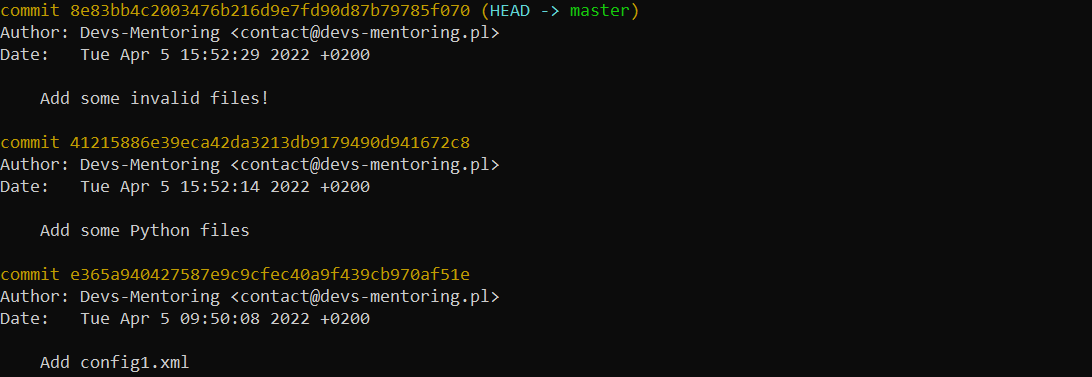
Na ten moment nie jest dla mnie najważniejsze, abyś dokładnie rozumiał, jak ma wyglądać pod spodem drzewo commitów, gdy wykorzystujesz tyldę, a jak, gdy opieraasz się na karecie, jednak nie byłbym sobą, gdybym nie rozwinął tak złożonego tematu. Spójrz poniżej na schemat commitów i różnicę między ^ a ~. Prezentuje on, jakiego zapisu należy użyć, aby odwołać się do konkretnego commita (zakładając, że HEAD ustawiony jest na commit A).

| G H I J  \ / \ /  D E F  \ | / \  \ | / |  \|/ |  B C  \ /  \ /  A <- HEAD | A = A^0  B = A^ = A^1 = A~1  C = A^2  D = A^^ = A^1^1 = A~2  E = B^2 = A^^2  F = B^3 = A^^3  G = A^^^ = A^1^1^1 = A~3  H = D^2 = B^^2 = A^^^2 = A~2^2  I = F^ = B^3^ = A^^3^  J = F^2 = B^3^2 = A^^3^2 |
| --- | --- |

**Git revert**

Polecenie to, podobnie jak reset, również umożliwia nam cofnięcie zmian w repozytorium. Różnica w stosunku do wcześniej omówionego polecenia jest jednak taka, że revert pozostawia historię commitów w nienaruszonym stanie i nie nadpisuje cofniętych commitów. Revert zatem zmienia aktualne drzewo projektu, jednak podczas cofania zmian powstaje nowy commit - który przechowuje wprowadzone modyfikacje.

Rozwiązanie to sprawdzi się zatem, gdy chcemy cofnąć jakąś zmianę, ale zachować w historii, że wykonano taką operację. Spójrzmy, jak możemy to zrobić:

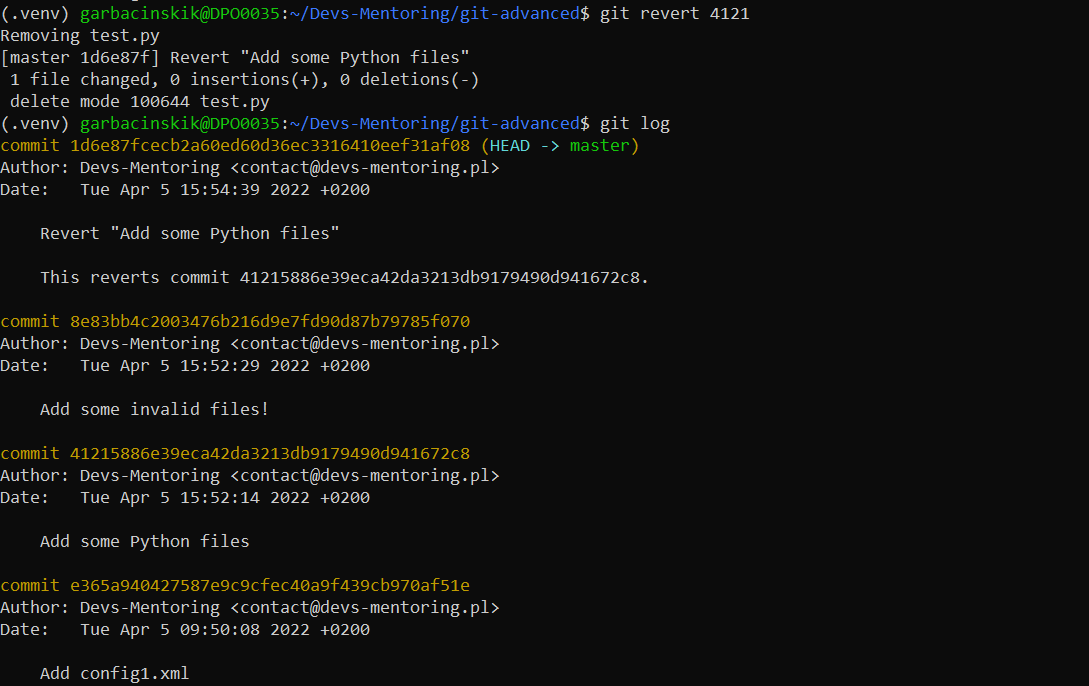


Załóżmy, że chcielibyśmy cofnąć ostatni commit o dość jednoznacznej nazwie: “Add some invalid files!”. Naszym celem będzie zrevertowanie commitu o pierwszych 4 znakach hashu: 4121.

Zapiszemy zatem…

git revert 4121

Z racji, że po wpisaniu polecenia git revert <commit\_id>, powstaje nowy commit, powinien automatycznie uruchomić się edytor kodu, w którym będziesz musiał umieścić swój commit message. Jeżeli nie wprowadzisz żadnych modyfikacji i pozostawisz go w domyślnej formie (tak jak ja) po wpisaniu git log, otrzymasz następujący rezultat:



Dzięki takiemu cofnięciu zmian, nie mamy już w projekcie błędnych zmian wynikających z wprowadzenia commitu o hashu 8e83… Widzimy, że powstał nowy commit, który reprezentuje aktualny status plików w projekcie. Z doświadczenia mogę Ci powiedzieć, że w pracy projektowej, na kodzie produkcyjnym, revert będzie głównym poleceniem, który będziesz wykorzystywał do cofania zmian w repozytorium.

A to dlatego, że omawiana komenda jest bezpieczniejsza niż reset i w każdej chwili możesz przywrócić swoją zmianę cofania. Co też w karierze programisty mi się zdarzyło nie raz.