

11 Git Tricków do codziennej pracy



Mocny zespół ma wiele asów w rękawie!

11 Git Tricków do codziennej pracy

Cześć!

Jesteśmy Tomasz Skraskowski i Krzysztof Morcinek z [GitWarsztaty.pl](https://gitwarsztaty.pl), a Ty... nie lubisz marnować czasu!

Są toole, z których korzystasz raz na rok – optymalizowanie pracy z nimi to kiepsko dobrany priorytet.

Gita używamy w pracy cały czas!

Rozumiejąc „co się dzieje” i „co teraz najlepiej zrobić” pracujemy znacznie efektywniej! 💪

Na początek przygotowaliśmy dla Ciebie 11 tricków, które możesz zacząć używać od zaraz.

Enjoy!



Reverse search (**CTRL+R**)

Żeby wpisać komendę, możemy (i powinniśmy) wspomagać się **autouzupełnianiam - za pomocą klawisza TAB**. Jeśli u Ciebie autouzupełniania nie działa z Gitem, to w pierwszej kolejności wygoogluj jak je ustawić w konsoli, z której korzystasz, albo zacznij używać innej.

Zazwyczaj jest tak, że polecenie, które nas interesuje, niedawno wpisywaliśmy w dokładnie takiej samej formie, jaką obecnie potrzebujemy. Wówczas zaczynamy wciskać z zawrotną prędkością strzałkę w górę i po chwili się go doszukujemy. Zadowoleni, że nie musieliśmy ponownie wpisywać, odpychamy od siebie myśli, że wcale nie wyszło szybciej.

Jest 100 razy prostszy i szybszy sposób, o którym z niewiadomych dla nas powodów (stan na rok 2022), nikt nie mówi i z którego prawie nikt nie korzysta! Nazywa się **reverse search**.

Wystarczy, że **wciśniesz CTRL+R i wpiszesz dowolny fragment polecenia**, które chcesz znaleźć w swojej historii.

Np. **CTRL+R+dd+ENTER** w tym momencie w mojej konsoli wykonało polecenie *git add*.

W ten sposób konsola pokazuje nam ostatnio pasujący wynik, a **jeśli chcemy wybrać wcześniejszy wynik, wystarczy jeszcze raz kliknąć CTRL+R**.

Narzędzie to jest częścią terminala (nie pochodzi z Gita) i posiada go każda współczesna konsola (nawet Powershell).

Aliaszy

Chociaż dla wielu osób istnienie aliasów to temat oczywisty, to wciąż mało kto z nich korzysta – zresztą Tomek sam kiedyś wychodził z założenia, że mając reverse search, ich nie potrzebuje, co dzisiaj uważa za błąd i rzadko kiedy używa komend Gita w ich oryginalnym kształcie.

Aliaszy to świetny sposób, żeby **uproszczyć wpisywanie komend**, a czasem na **przemycenie do jednej „komendy” różnych skomplikowanych operacji**. Nigdy więcej nie będziesz musiał trzymać poleceń i argumentów w osobnych notatkach – **zbiór aliasów będzie jednocześnie Twoją „notatką”**.

Aliaszy można dodawać na poziomie całej konsoli, ale łatwiej jest wykorzystać mechanizm przygotowany przez Gita, więc na nim się skupiamy. Żeby dodać alias, **wystarczy w pliku `.gitconfig` dodać sekcję `[alias]` i w nowych liniach wpisywać aliasy w składni `alias = komenda [opcjonalny-parametr]`** - pomijamy słowo „git”, Git wie, że Git :D

Przykład: `cnf = config --global -e`

Użycie: `git cnf`

Aliaszy mogą łączyć kilka poleceń, a nawet przyjmować argumenty, ale na początek skup się na uproszczeniu sobie podstawowych komend do krótkiej formy, którą łatwo zapamiętasz np. `git st`, `git co`, `git reb`.

Tip: Nie szukaj pliku `.gitconfig` na dysku, tylko wpisz w konsoli `git config --global -e`

Tip2: Możesz podejrzeć aliasy Tomka w [publicznym giście na GitHubie](#) (user: `tometchy`)

git snapshot

Przed chwilą zachęcaliśmy do dodania krótkich aliasów do podstawowych komend, a teraz pokażemy Ci Tomka autorski, ulubiony alias, z którego korzysta dosłownie kilkadziesiąt razy dziennie!

git snapshot [opcjonalny-opis] – który najpierw doda wszystkie nasze obecne zmiany do *stage*, następnie utworzy tymczasowego commita, wpisując jako tytuł obecną datę i czas (+ opcjonalnie opis), po czym wycofa tego commita tak, jakbyśmy po prostu wykonali *git add --all*. **W efekcie nie tylko mamy zastagowane wszystkie nasze zmiany, ale jednocześnie zabezpieczone, zawsze możemy do nich wrócić.**

Kiedy snapshota używać? Cały czas w trakcie pracy, zanim nasze zmiany będą gotowe do zacommitowania. Nawet pdf, który właśnie czytasz, dostawał snapshot co kilka minut :)

Pomimo że zawsze zachęcamy do tworzenia małych commitów, to i tak, zanim one powstaną pracę mamy rozgrzebaną. Współczesne IDE np. od Jet Brainsów tworzą lokalnie historię zmiany pliku, ale trzymanie jej w Gicie jest bezpieczniejsze i prostsze – widzimy całość, a nie tylko pojedynczy plik.

```
snapshot = "!f() { git add --all; git commit -m \"SNAPSHOT_$(date +%Y-%m-%d_%H-%M-%S) ${1}\" --allow-empty; git reset --soft HEAD^; }; f"
```

show-snapshot = show HEAD@{1} # Drobną podpowiedź jak podejrzeć snapshot, w tej postaci, tylko jeśli po snapshotcie nie commitowaliśmy

Przykłady użycia: *git snapshot*, *git snapshot works*, *git snapshot 'maybe we finally got it'*, *git show-snapshot*
Później za pomocą *git reflog* możesz dany snapshot odszukać i podejrzeć lub przywrócić.

Zagadka: Tomek ustawił sobie dodatkowy alias *san* = *snapshot*. Zgadnij dlaczego, napisz na kontakt@GitWarsztaty.pl :)

git add [path] -p; git reset [path] -p ~Jedz słońca po kawałku~

Najlepszy sposób na kontrolowanie tego, co wprowadzamy, to stagowanie zmian pojedynczo.

Jest to jednocześnie dobry sposób, na ominięcie zmian, które nie pasują do commita, który właśnie tworzymy.

Wystarczy **do polecenia add dodać przełącznik -p**.
Dokładnie analogicznie możemy wycofywać pojedyncze zmiany, które już są w stage (**git reset -p**).

Jeśli chcemy wybierać tylko z części plików, **możemy wpisać konkretną ścieżkę (opcjonalnie z wildcardami)**.

Opcji do wyboru w trybie patchowania jest wiele (patrz [dokumentacja](#)), Git pytając o wybór je wypisuje.

Najważniejsze opcje to: **biorę (y)**, **odrzuć (n)**, **podziel na mniejsze (s)**, **przerwij (q)**. Gdy zmiany są blisko siebie, musimy wejść w **tryb edycji (e)** i podążać za wypisaną na dole instrukcją – oswojenie się z trybem edycji zajmuje trochę czasu, ale na początku się nim nie przejmuj, z czasem załapiesz jak się w nim odnaleźć.

Tip: Tomka alias na **add** to **a** (w configu: **a = add**), użycie: **git a -p**

```
1/1 + [?] [?] Tilix: Default
1: Terminal
set_main="DNS.1 = ${DOMAIN}"
- echo "$set_main"
echo "$set_main" >> /var/www/example.com/cert/custom-openssl.cnf

domains=$(echo $ALT_DOMAINS | tr ";" "\n")
Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? y
@@ -51,7 +50,7 @@ else
req_additional=""
fi


-openssl genrsa -out /out/${DOMAIN}.key 2048
+openssl genrsa -out /out/${DOMAIN}.key 4096
openssl req -x509 -new -key /out/${DOMAIN}.key -out /out/${DOMAIN}.crt${privateKeyEncryption} -days ${DAYS} -subj "/C=${C}/ST=${ST}/L=${L}/O=${O}/OU=${OU}/CN=${DOMAIN}/emailAddress=${EMAIL}"${req_additional}
openssl pkcs12 -export -in /out/${DOMAIN}.crt -inkey /out/${DOMAIN}.key -out /out/${DOMAIN}.pfx${privateKeyEncryption}
Stage this hunk [y,n,q,a,d,K,g,/,e,?]? y
```


Git commit z ręką na pulsie

Często spotykamy się z praktyką commitowania za pomocą `git commit -m 'commit title'`, którą na ogół odradzamy. Gdy korzystasz z lekkiego i szybkiego edytora tekstowego, to narzut na jego otwarcie jest nieodczuwalny, a gdy jest to edytor konsolowy (np. Vim), to jest dosłownie zerowy. **Jak commitować? Tak: `git commit -v`**

Git pokazuje Ci, które pliki są dodane, a które pozostawione i zawsze warto przed zapisaniem rzucić na to okiem, żeby mieć pewność, że robimy to, co zamierzaliśmy. Dodatkowo **z opcją `-v` Git pokazuje diffy, warto przynajmniej pokrótce „przescrollować” co konkretnie commitujemy** – gdy robimy małe commity (a warto!), zajmuje to dosłownie kilka sekund.

Co więcej, gdy korzystamy z edytora, który „rozumie Gita”, to **mamy pomoc przy tworzeniu czytelnych commitów** - kolorowanie, automatyczne łamanie linii, etc.



```
This title i a bit too long, see text colors in this editor
This text is in wrong line
Here is fine. And bellow you can see which changes are to be committed and which are not.
Moreover with -v flag you can see introduced changes (bellow).
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch git_jest_git_i_co_dalej_evening
# Your branch is up to date with 'tomrepo/git_jest_git_i_co_dalej_evening'.
#
# Changes to be committed:
#   modified:   wycisnij-soki-z-gita/index.html
#
# Changes not staged for commit:
#   modified:   index.html
#
# ----- >8 -----
# Do not modify or remove the line above.
# Everything below it will be ignored.
diff --git a/wycisnij-soki-z-gita/index.html b/wycisnij-soki-z-gita/index.html
index 9939fda..786c0d3 100644
--- a/wycisnij-soki-z-gita/index.html
+++ b/wycisnij-soki-z-gita/index.html
@@ -268,6 +268,10 @@ cover: img/prezentacja.jpg
     </aside>
   </section>
+
+   <section>
+     <h3><code>git commit</code> > <code>git commit -m</code></h3>
+   </section>
```

Too long text indication

Text in wrong line indication

See which files got staged, which didn't

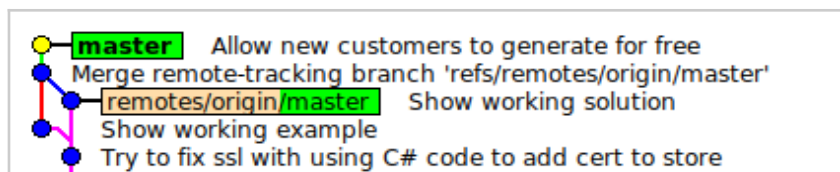
See changes (-v flag required)

Those colors are configured in VIM by default

Tip: Tomka alias do commitowanie to `cov` (w configu: `cov = commit -v`), użycie: `git cov`

git pull --rebase

Powszechnie wiadomo, że *pull* to „pod spodem” dwie operacje: *fetch* następnie *merge*. Natomiast **mało kto wie, że *pull*, zamiast *merge* może wykonać *rebase***, dzięki czemu unikniemy pajęczynkowego rozgałęzienia:



Służy do tego przełącznik *--rebase*, który dodajemy do polecenia *pull*. **Możemy skonfigurować Gita tak, żeby *rebase* było domyślnym zachowaniem dla *pulla***, wystarczy w pliku *.gitconfig* utworzyć sekcję *[pull]* i wpisać w niej *rebase = true*. Ten sam efekt da wywołanie polecenia *git config --global pull.rebase true*

Istnieje również możliwość, ustawienia tego zachowania tylko dla wybranych branchy, a nawet poinstruowania Gita, żeby ustawiał tę opcję sam, gdy zaczynamy śledzić zdalne branche wybranego typu. Są to jednak rzadkie przypadki, więc po szczegóły odsyłamy do dokumentacji. Wspomnimy tylko, że tego typu opcje konfiguracyjne lepiej ustawiać na poziomie konkretnego repozytorium, czyli w pliku *config* w katalogu repo (*.git*).

Tip: Tomka alias na *pull* z *rebase* to *pur* (w configu: *pur = pull --rebase*), użycie: *git pur*

Tip2: Tomek alias ma, ale i tak nie korzysta z *pull* wcale :) Woli robić ręcznie *fetch*, obserwować co nowego dochodzi i wówczas decydować co z nowymi commitami zrobić. Trochę jak kierowca rajdowy „strzelający ze sprzęgła” - w automacie się nie da :)

git rebase -i COMMIT_ID

Nie używaj rebase, to jest trudne, a jak popełnisz błąd, to wszyscy w firmie będą na Ciebie źli...

Powyższe zdanie to była prawda, jakieś 10 lat temu. Niestety zakorzeniła się w świadomości programistów i do dziś jest często powtarzana. To odstrasza wielu informatyków, żeby korzystali z usprawnień pracy, jakie daje *rebase*.

Dlaczego rzeczywiście kiedyś tak było? Po pierwsze dlatego, że *UI* polecenia *rebase* kiedyś był mniej dopracowany niż dzisiaj, ale przede wszystkim dlatego, że repozytoria w firmie to była „wolna amerykanka”, w której każdy świadomie lub nie, mógł mącić. **Dzisiaj standardem jest stosowanie *pull requestów* i blokowanie wprowadzania niezatwierdzonych zmian do głównej gałęzi. A na mojej roboczej gałęzi mogę sobie mącić do woli, pomimo że wypchnąłem ją na serwer od razu po utworzeniu!**

Nie tylko mogę modyfikować historię na swoim branchu, ale nawet powinienem – posprzątać przed opublikowaniem pull requesta. Mój wkład do wspólnego repozytorium powinien być schludny i czytelny, powinienem więc pousuwać niechciane commity lub dołączyć je do innych, zmienić kolejność na bardziej zrozumiałą, poprawić tytuły itp.

Zapomnij o „oh I forgot commit” czy tymczasowych commitach w stylu „wip”, „Changes...”, „Maybe it will work...”. Oczywiście roboczo mogą istnieć, ale nie są mile widziane przy commitach oddanych do review.

11 Git Tricków do codziennej pracy

Najprostszy sposób to polecenie *git rebase -i*

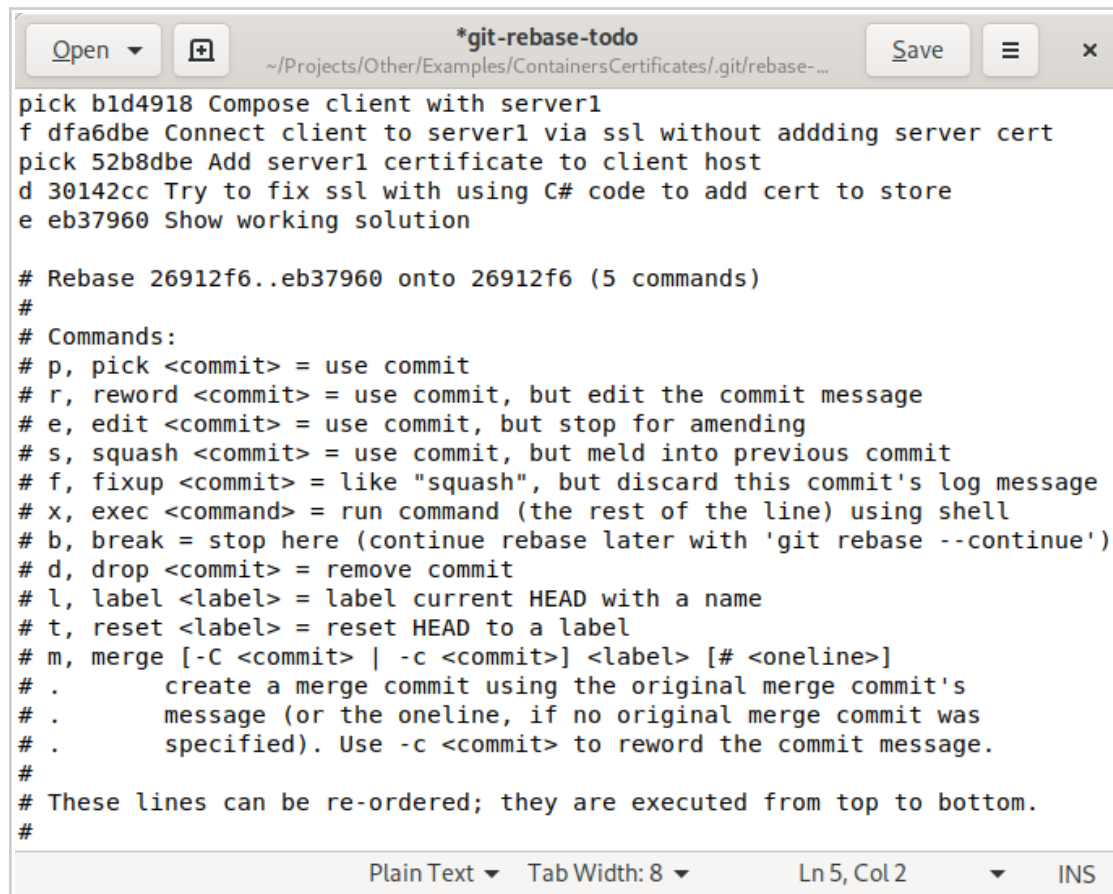
wraz z *ID commita*, do którego commity będziemy zmieniać. Np. *git rebase -i HEAD~5*

Git otworzy nam edytor tekstowy z wybranymi przez nas commitami – z tego poziomu **możemy zmieniać kolejność commitów (wystarczy zmienić kolejność linijek w pliku) lub wykonywać operacje na commitach.**

Dostępne opcje są wypisane w komentarzu, najpopularniejsze to: usuwanie commitów (**drop**), scalanie commitów (**squash** i **fixup**), zmiana opisu commita (**reword**), zmiana wprowadzonych zmian (**edit**).

Tip: Przed modyfikowaniem historii zrób pusha, wówczas, żeby wrócić do starej wersji nie będziesz potrzebować *reflog*, wystarczy *git reset --hard origin/branch-name*

Tip2: Gdy nadpisujesz historię zawsze używaj *git push --force-with-lease* – przed wypchnięciem Git sprawdzi, czy na serwerze w międzyczasie nie doszło coś nowego do brancha, który nadpisujesz



```
*git-rebase-todo
~/Projects/Other/Examples/ContainersCertificates/.git/rebase-...
Save

pick b1d4918 Compose client with server1
f dfa6dbe Connect client to server1 via ssl without adding server cert
pick 52b8dbe Add server1 certificate to client host
d 30142cc Try to fix ssl with using C# code to add cert to store
e eb37960 Show working solution

# Rebase 26912f6..eb37960 onto 26912f6 (5 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .      create a merge commit using the original merge commit's
# .      message (or the oneline, if no original merge commit was
# .      specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
```

git commit --amend [--no-edit]

Git *rebase interactive* wykorzystujemy zazwyczaj do „obrabiania” zakresu commitów i mamy w nim ogromne możliwości. Często jednak jedyne co chcemy zrobić, to dorzucić coś do commita, który przed chwilą utworzyliśmy, albo zmienić coś w jego opisie. Już zrobiliśmy commita (być może nawet już go wypushowaliśmy na swojej gałęzi), po czym dostrzegamy „a, jeszcze to”, albo „Aj, literówka w opisie commita”. **Oczywiście możemy w tej sytuacji skorzystać z *rebase*, ale w tym przypadku lepiej pasuje prostsza alternatywa – *commit --amend*.**

Wystarczy dodać do stage zmiany, które mają zostać scalone do ostatniego commita i wpisać *git commit -amend*. Wówczas zobaczymy edytor tekstowy, w którym możemy edytować opis ostatniego commita, a po zapisaniu **to, co dodaliśmy do stage, zostanie dołączone do commita**. Oczywiście nie musimy nic do stage dodawać, możemy jedynie poprawić opis commita.

Gdy zmieniliśmy coś w plikach i **chcemy zostawić opis commita bez zmian, możemy dodatkowo dopisać *--no-edit***. W efekcie zmiany ze stage zostaną dorzucone do ostatniego commita, opis pozostanie bez zmian, bez otwierania edytora tekstowego.

Tip: Tomka aliasy na *commit --amend* to *coa* i *coane*, użycie *git coa* lub *git coane*, w configu:

coa = commit --amend

coane = commit --amend --no-edit

Gdy zmieniliśmy wiele plików...

Standardowo w polecanym przez nas flow, commity robimy małe i spójne, mają więc zazwyczaj niewiele zmienionych plików. Przy tym podejściu właściwie jedyna sytuacja, gdy mamy zmienionych wiele plików, to wykonanie jakiejś akcji w IDE, np. zmiana nazwy klasy/metody/namespace/etc. **Nigdy nie odpalamy dużych renamów „przy okazji” zmieniania czegoś innego. Zawsze czekamy z taką akcją aż skończymy obecną czynność, zacommitujemy skończony etap i dopiero z czystym working directory odpalamy IDE.** Wówczas zmiany nie pomieszą się z innymi, co poprawi czytelność. Dodatkowo, jeśli coś pójdzie nie tak, a IDE nie będzie umiało zmian „odkręcić”, wystarczy **`git restore :/`** i wracamy do punktu wyjścia.

Czasem jednak pomimo szczyrych chęci, zmiany wymykają się spod kontroli i `git status` zaczyna wypisywać długą litanię. Warto wówczas pamiętać o kilku mechanizmach:

- Polecenia **`git status`**, **`git add`**, **`git checkout`**, **`git diff`**, **`git reset`** mogą przyjmować ścieżkę z opcjonalnymi **wildcardami** – możemy więc zawężyć sobie obszar działania, np. `git add src/server/*.csproj -p`
- Polecenie **`git status`** ma wariant **`--short / -s`**, który pokazuje wyłącznie zmienione pliki – dzięki czemu więcej informacji mieści się naraz na ekranie
- Polecenie **`git diff`** ma wariant **`--color-words`**, który jest czytelniejszy w przypadku zmian w pojedynczych słowach.

Tip: Możesz skorzystać z aliasu, który do statusu dopisuje listę commitów do pobrania/wypchnięcia (`sta` i `stsa`), skopiuj sobie z [publicznego GitHub Gista](#) (user: *tometchy*)

git bisect

- Co robisz?
- Szukam commita, przez którego system się źle zachowuje.
- Czemu ręcznie, zamiast z *Git Bisect*?
- A co to?

Powyższa rozmowa wydarzyła się naprawdę i nie jest odosobnionym przypadkiem, dlatego przy wielu okazjach wracamy do tematu *Git Bisect*. Zazwyczaj, gdy naprawiamy błąd, to analizujemy zachowanie i szukamy miejsca w kodzie, które je powoduje i przeważnie to podejście jest wystarczające. Niestety **czasem utykamy w tak zwanym martwym punkcie i pomysły co może być przyczyną zaczynają się wyczerpywać. Przydaje się wtedy alternatywne podejście, czyli przeskakiwanie do starszych wersji i sprawdzanie, czy błąd już był, czy nie.**

Jest to **szczególnie przydatne w przypadku odnotowania spadku wydajności aplikacji** - oczywiście w arsenale programisty są różne profilery pomagające znaleźć przyczynę, ale analizowanie wyników bywa czasochłonne i trudne.

Możemy przetestować którąś starszą wersję, np. z poprzedniego releasea albo w ciemno np. 100 commitów wstecz i zwyczajnie sprawdzić – czy błąd albo spowolnienie wydajności już wtedy był, czy jeszcze było ok.

Jeśli już był, wiemy, że został wprowadzony wcześniej, jeśli jeszcze go nie było, to wiemy, że został wprowadzony później, wiemy więc w którą „stronę” kontynuować poszukiwania. Dokładnie do tego służy ***Git Bisect* - przeprowadza nas przez ten proces w sposób uporządkowany**, korzystając z algorytmu binary search.

Example of real Git Bisect case (from old Akka.NET issue)

Haven't found anything obvious in the profiler that can explain the disparity in these numbers - so we're going to need to go line by line in the commit log and see which changes might be responsible for this:

1.3.18...1.4.6

I've modified your reproduction so I can copy local Akka.NET builds into the v1.4 branch and run performance comparisons that way: <https://github.com/Aaronontheweb/Akka.Net-versions-issue>

11 Git Tricków do codziennej pracy

Proces Git Bisect wygląda następująco:

0. Jesteśmy na wersji, na której jest błąd, zaczynamy poszukiwanie: **`git bisect start`**
1. Następnie mówimy Gitowi, że ta wersja źle działa - **`git bisect bad`**
2. Następnie mówimy Gitowi, która wersja działała dobrze np. **`git bisect good HEAD~50`**
3. W tym momencie Git wybierze nam commit ze środka zakresu (pomiędzy *dobrym* a *złym*), a my musimy sprawdzić i powiedzieć Gitowi, czy na tej wersji błąd już jest (**`git bisect bad`**), czy błędu nie ma (**`git bisect good`**).
4. Wracamy do punktu 3. dopóki nie znajdziemy commita winowajcy :) Dodatkowo możemy w każdym momencie skorzystać z **`git bisect visualize`**, żeby podejrzeć zakres commitów, który obecnie Git przyjął.
5. Gdy już commit jest znaleziony, za pomocą polecenia **`git bisect reset`** wychodzimy z trybu bisectowania.

A teraz pora na **gwóźdź programu – Git może sam, automatycznie znaleźć commit wprowadzający błąd!**

Czekamy na wersję, w której Git sam będzie błąd naprawiał :D Ale jak to? Ano zamiast mówić Gitowi, czy wersja jest dobra czy zła, możemy wykorzystać polecenie **`git bisect run AppOrScript [arguments]`** do którego prześlemy program lub skrypt, który sprawdzi wersję (np. wykona kompilację i uruchomi test jednostkowy albo UI) i jeśli wersja jest *good*, to zwróci kod wyjścia 0, a jeśli *bad* to kod z zakresu 1-127. **Jeśli więc mamy testy automatyczne (a powinniśmy!), to Git może dla nas zrobić z nich użytek. To jest kolejny powód, do tworzenia małych commitów, gdy commit ma tylko jedną, spójną zmianę, to w efekcie widzimy dokładnie co wprowadza błąd.** Oczywiście znalezienie wielkiego commita również jest bardzo pomocne. Ciekawostka: słowa *good* i *bad* można zmienić, gdyby inne lepiej odzwierciedlały co poszukujemy np. *fast/slow*, ale to rzadki przypadek, do tego więc [odsyłam do dokumentacji](#).

Ps. Tomek opisał szczegółowo [Git Bisect na blogu](#) oraz nagrał [video demo Git Bisect z CodeceptJS](#).

Przenoszenie commitów pomiędzy repozytoriami niezwiązanymi ze sobą

Git jest bardzo elastyczny, między innymi umożliwia synchronizowanie repozytoriów różnymi protokołami: *http(s)*, *ssh*, a **nawet można robić push i fetch wprost do innego katalogu na dysku**.

Mało kto wie, że **kilkanaście lat temu, do synchronizowania repozytoriów Gita, używano również... maila!**

Nie mówimy tutaj o „ludzkim” opisywaniu co zmienić w kodzie, tylko o wysyłaniu pojedynczych commitów.

Obecnie nie ma sensu w ten sposób pracować, ale **mechanizm, który był wówczas używany, można wykorzystać do bardzo sprytnego tricku – przenoszenia commitów pomiędzy repozytoriami**.

Eksport commitów do plików odbywa się za pomocą polecenia *git format-patch -liczba_commitów LAST_COMMIT_ID*, a nakładanie commitów z plików za pomocą *git am*.

Jest to przydatna sztuczka, gdy zrobiliśmy coś niezwiązanego bezpośrednio z projektem, przykłady:

- zmiany w plikach *.gitconfig* i *.gitattributes*
- zmiany związane z uniwersalnymi skryptami np. *start-Jekyll.sh*
- zmiany związane z IDE, stylem kodu, słownikami itp.

11 Git Tricków do codziennej pracy

Prześledź jak zostały przeniesione dwa commity z projektu *DevOpsBrokerServer* do projektu *ConnectingServices*

```
2/2 ▾ + [?] [?] Tilix: Default 🔍 ☰ ✕
1: Terminal ▾ □ ✕
tom@server~/Projects/DevOpsBrokerServer (master) $ git log -4 --oneline
dd5b37a (HEAD -> master) Show working solution
8362815 Add server certificate to client host
3d34252 Connect client to server
0a49c47 Compose client with server
tom@server~/Projects/DevOpsBrokerServer (master) $ git format-patch -2 HEAD~2
0001-Compose-client-with-server.patch
0002-Connect-client-to-server.patch
tom@server~/Projects/DevOpsBrokerServer (master) $ cp *.patch ../ConnectingServices/
tom@server~/Projects/DevOpsBrokerServer (master) $ cd ../ConnectingServices/
tom@server~/Projects/ConnectingServices (master) $ git am *.patch
Applying: Compose client with server
Applying: Connect client to server
tom@server~/Projects/ConnectingServices (master) $ git log -2 --oneline
b44225a (HEAD -> master) Connect client to server
00aa37c Compose client with server
tom@server~/Projects/ConnectingServices (master) $
```

Tip: To jest trick wykorzystywany stosunkowo rzadko – zdążysz zapomnieć, jakie dokładnie to były polecenia.

Zapisz sobie tego pdfa gdzieś, gdzie go zawsze odnajdziesz.

Jeśli i tak go zgubisz, nie wahaj się do nas napisać – przypomnimy :)

Bonus! How to reach *Aha! Moment* hint

Git tricki można stosować od zaraz, ale **jest coś, czego nie sposób przekazać w postaci pdfa czy artykułu.**

W dzisiejszych czasach praca z Gitem jest na tyle ustandaryzowana, a dostępne toole rozbudowane, że każdy może zacząć commitować i tworzyć pull requesty nawet po kilkuminutowym wprowadzeniu – i to jest super.

Gdy zaczynamy pracę w IT najważniejsze to zacząć „dowozić” - opanować główne narzędzie lub język programowania, którym wykonujemy pracę. Na tym etapie tworzenie pull requestów po najmniejszej linii oporu jest akceptowalne, byleby spełniać minimalne standardy schludności i procedury wypracowane przez zespół.

Kiedy podstawowe aspekty naszego rzemiosła mamy już opanowane i dowozimy zadania, czyli wartość dla klienta, warto zrozumieć co właściwie się dzieje w trakcie podstawowych czynności naszej pracy z repo.

- Czym w zasadzie są te abstrakcyjne *branche*?
- Co to jest *HEAD*, czy w ogóle jest do czegoś potrzebny?
- Jaki wpływ na repo ma *merge* i na czym polega ten mistyczny *rebase*? Jakie daje korzyści, a jakie są jego minusy? Kiedy i dlaczego czasem utrudnia życie? Jak go używać, żeby nigdy nie utrudniał, a wyłącznie ułatwiał?
- Do czego służy *cherry-pick*?
- Co zrobić, gdy zcommitowaliśmy lub wypushowaliśmy coś *nie tak*? *Reset*, *Revert*, *Rebase*, a może *Commit --Amend*?
- Jak przestrzegać zasady, tworząc *dobry commit message* i dlaczego akurat takie?
- Jak przygotować sobie *optymalne środowisko*? Jakie są *zasady konfiguracji Gita*? Jak działa *konfiguracja na poziomie repo*? Na czym polegają *pliki .gitignore i .gitattributes*?

11 Git Tricków do codziennej pracy

Odpowiedzi na te i inne pytania, da się opisać np. w postaci artykułów, ale, żeby je *poczuć* **potrzeba czegoś więcej**.

Na przestrzeni lat udało się nam owe *coś więcej* opracować. Przeprowadzamy warsztaty, w których każde zagadnienie zostaje najpierw w zrozumiały sposób przedstawione, a następnie uczestnicy przechodzą przez ćwiczenia praktyczne.

Rozwiązanie zadania samemu to przestrzeń, żeby teorię odkryć w praktyce, a z pomocą trenera każda wątpliwość zostaje od razu rozwiana.

Nasz cel na warsztatach to Twój *Aha! Moment*

Po co?

Ponieważ dzięki temu obsługa Gita, zarówno z konsoli, jak i z dowolnego *GUI toola* stanie się naturalna i zrozumiała - wliczając w to platformy takie jak GitHub, GitLab, Azure DevOps itp. **Wszystkie te narzędzia są zbudowane w oparciu o Gita i rządzą się tymi samymi prawami**, różnią się tylko sposobem interakcji i prezentowania rezultatów.

W jaki sposób przebiegają warsztaty?

Przyjedziemy do siedziby Twojej firmy (w każdym mieście Polski) lub spotkamy się zdalnie na Zoomie.

Trener każde zagadnienie najpierw objaśnia w teorii, następnie pomaga przećwiczyć w praktyce.

Zakres materiału i ćwiczenia są dopracowywane przez lata i zawsze spotykamy się z bardzo pozytywnym feedbackiem.

11 Git Tricków do codziennej pracy

Co, jeżeli przełożony ma wątpliwości?

Prześlij nam do niego kontakt, jeśli się zgodzi, to chętnie zadzwonimy i odpowiemy na wszystkie pytania.

Spróbuj uświadomić mu, że **silny zespół, to zespół, który cały czas się rozwija, a w efekcie pracuje wydajniej i ma większą motywację do pracy.** 💪

Jeżeli za decyzją przemawia budżet, to warto pamiętać, że Git to narzędzie, z którego każdy programista i tester korzysta niemalże bez przerwy. **Zrozumienie Gita i poznanie strategii na różne codzienne sytuacje to nie tylko przyjemniejsza praca, ale również realnie oszczędzony czas** – patrząc na pensje w IT taniej wychodzi pracownika przeszkolić. 💰

Dodatkowo **szkolenie podnosi morale zespołu, a czasem również oszczędza wiele stresu (całej firmie), gdy umiejętności pomogą uniknąć fackupów.** 🐛

Co więcej, **gdy od razu wiesz co zrobić w danej sytuacji, nie wybijasz się z flow pracy, co również przekłada się na jakość i tempo pracy.** 📈

Masz uwagi lub sugestie?

Zawsze możesz do nas napisać na adres kontakt@GitWarsztaty.pl lub zadzwonić – tel. [792-228-321](tel:792-228-321).

Szkolenie *Szlifowanie Gita*

Aha! Moment to większa wydajność i satysfakcja z pracy



Tomasz Skraskowski

[792-228-321](tel:792-228-321)

tometchy@gmail.com

Krzysztof Morcinek

[737-692-782](tel:737-692-782)

krzysztof.morcinek@gmail.com

www.GitWarsztaty.pl

kontakt@GitWarsztaty.pl

GITWARSZTATY.PL
