Jacek Drąg

Git

Od koncepcji do praktyki

Helion

Redaktor prowadzący: Tomasz Gojowy

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/gitsre
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-289-1249-6

Printed in Poland.

Spis treści

ROZDZIAŁ 2.

Repozytorium lokalne

2.1. git — the stupid content tracker

Dokumentacja: `git` (*https://git-scm.com/docs/git*).

Na samym początku zobaczmy, co Git mówi sam o sobie:

Linux

```
man git
```

Windows

```
git help git
```

Powinniśmy zobaczyć coś w stylu:

```
NAME
       git - the stupid content tracker

(...)

DESCRIPTION
      Git is a fast, scalable, distributed revision control system with an unusually
      rich command set that provides both high-level operations and full access to
      in ternals.

(...)

      After you mastered the basic concepts, you can come back to this page to learn
      what commands Git offers.
```

Jak już wspomniano, w tej książce poznamy:

- *basic concepts,*
- *commands*:
  - sporo *high-level* (zwanych czasem *porcelain*),
  - trochę niskopoziomowych (zwanych czasem *plumbing*).

Polecenia przyjmują zazwyczaj wiele parametrów. Niektóre z parametrów często się powtarzają.

Parametry powtarzające się w wielu poleceniach

`(-n | --dry-run)`

Tylko pokazanie, co zostanie wykonane.

`(-v | --verbose)`

Wersja bardziej gadatliwa.

`(-q | --quiet)`

Wersja cicha.

`[--] [<spec>...]`

Niektóre polecenia operują na plikach/ścieżkach. Plików/ścieżek można wówczas specyfikować wiele. Często warto jawnie oddzielić inne parametry od specyfikacji plików. Parametry występują przed podwójnym myślnikiem (--), natomiast pliki/ścieżki po. Pozwala to czasem uniknąć niejednoznaczności (np. czy *foo* ma być potraktowane jako nazwa pliku, czy też gałęzi).

2.1.1. Śledzenie historii zmian, czyli historia commitów

Gita używa się, jeśli chce się śledzić historię zmian jakiegoś katalogu na dysku. Najczęściej będzie to katalog zawierający jakiś projekt informatyczny, ale równie dobrze może on zawierać pracę magisterską, rozprawę doktorską czy cokolwiek innego. Po prostu, gdy w jakimś katalogu mamy ważne pliki, których historia zmian jest dla nas istotna, można użyć Gita. Prawdopodobnie będzie to świetny wybór!

Taki wybrany katalog będziemy nazywać **projektem**, bo najczęściej zawiera on pewien „projekt".

Git śledzi zawartość projektu i pozwala:

- Przeglądać historię jego zmian.
- Przywracać zawartość ze wskazanego stanu z historii.

Przy czym nie śledzi tej zawartości samoczynnie, lecz zapamiętuje jedynie te stany projektu — migawki (ang. *snapshots*) — które użytkownik uznał za ważne, więc *jawnie* wskazał Gitowi, że należy je zapamiętać.

Przywracać zmian też nie można z *dowolnego* momentu w czasie; można przywracać jedynie *zapamiętane* stany (migawki).

Dokładniej praca z Gitem wygląda następująco:

- Użytkownik dodaje, edytuje, usuwa pliki w projekcie (tj. w śledzonym katalogu).
- Gdy uzna, że dobrze byłoby zapamiętać aktualny stan projektu (wszystkie lub tylko niektóre pliki):
  - Wskazuje Gitowi, które pliki mają zostać zapamiętane w aktualnej postaci.
  - Zleca Gitowi zapamiętanie stanu.

- Podaje przy tym komunikat opisujący nowy stan (ang. *commit message*) — najczęściej istotne zmiany wprowadzone od poprzedniego zapamiętanego stanu. Komunikat ten jest potem przydatny podczas przeglądania historii.

Można tu dostrzec pewną analogię do wprowadzania i zatwierdzania zmian w relacyjnej bazie danych. Najpierw dokonujemy zmian w bazie (być może w kilku krokach), następnie zmieniony stan bazy zatwierdzamy. Przed zatwierdzeniem można się ze zmian wycofać.

Taki zapamiętany stan projektu nazywamy **commitem** (ang. *Git commit*). Co ważne, Git jako swoje commity zapamiętuje całe migawki projektu, a nie np. zmiany dokonane w projekcie od czasu zapamiętania poprzedniej migawki czy cokolwiek innego.

Tak więc commit to zapamiętana migawka *całej zawartości* projektu. Może nie wydawać się to istotne, czy Git przechowuje migawki, czy tylko różnice pomiędzy poprzednią a bieżącą zawartością. A jednak jest to istotne, co okaże się w swoim czasie.

Git przechowuje historię commitów, tj. migawek *całej* zawartości *projektu* (śledzonego katalogu). Migawki do zapamiętania są jawnie wskazywane przez użytkownika.

2.1.2. Zawartość projektu

Powtórzmy: Git śledzi zawartość swojego projektu, a projekt jest po prostu pewnym katalogiem na dysku.

Przez zawartość projektu Git rozumie:

- Strukturę jego katalogów.
- Pliki w tych katalogach.

Przy czym „uważa", że *prawdziwa (za)wartość* siedzi w plikach, w związku z czym kompletnie nie interesują go puste katalogi.

Tak więc, jeśli utworzymy pusty katalog, np. *foo*, lub nawet katalog z podkatalogiem, np. *foo/bar*, ale nie umieścimy tam *żadnego pliku*, to Git taki katalog zignoruje — nie zapamięta go w żadnym commicie. Ba! Nawet nie ma takiej możliwości! Żeby zobrazować to zachowanie, przeprowadźmy mały eksperyment.

Mały eksperyment

```
mkdir experiment1
cd experiment1
git init ❶
mkdir -p foo/bar ❷
git status ❸
git add foo ❹
git status ❺
git commit -m"Do commit anyway" ❻
touch foo/bar/x.txt ❼
```

```
git status ❽
git add foo ❾
git status ❿
git commit -m"Do commit, second try" ⓫
git slog ⓬
git show ⓭
```

❶ Zainicjowanie katalogu *experiment1* jako projektu Gita (inaczej: utworzenie repozytorium).

❷ Utworzenie katalogu z podkatalogiem.

❸ Sprawdzenie, że Git zdaje się nie zauważać utworzonych katalogów.

❹ Próba dodania katalogu do przyszłego commita.

❺ Sprawdzenie, że nic się nie zmieniło.

❻ Próba utworzenia pierwszego commita.

❼ Dodanie pliku.

❽ Sprawdzenie, że coś się zmieniło.

❾ Dodanie katalogu *foo* (wraz z jego podkatalogami) do przygotowywanego commita.

❿ Sprawdzenie, co się teraz zmieniło.

⓫ Utworzenie commita.

⓬ Wyświetlenie (jednoelementowej) historii commitów.

⓭ Wyświetlenie commita.

Git **nie** zapamiętuje pustych katalogów. Aby katalog został zapamiętany, on sam lub któryś z jego podkatalogów **musi** zawierać jakiś plik. Sam plik może już być pusty.

2.1.3. Pliki tekstowe, pliki binarne

Git najlepiej nadaje się do śledzenia plików tekstowych, np. kodu źródłowego programów, dokumentacji w formatach LATEX, ADOC, MD. Jeśli jednak w naszym projekcie korzystamy z plików binarnych (np. pliki Office, JPG itp.), to jak najbardziej możemy używać Gita, aby zapamiętywać/przeglądać historię zmian projektu z takimi plikami.

Często podnoszony problem to objętość repozytorium zawierającego pliki binarne. Są one duże i kiepsko się kompresują, bo zazwyczaj już są skompresowane. Załóżmy, że mamy plik 100 MB i zapamiętanych 10 jego wersji, czyli potrzebujemy 1 GB miejsca na dysku. Może to się wydawać dużo, ale skoro naprawdę chcemy zapamiętać te 10 wersji, to przecież trzeba znaleźć na nie miejsce. Oczywiście, można sobie wyobrazić zapamiętywanie tylko zmian pomiędzy wersjami, a nie całych plików, ale

Git tak nie działa. Co więcej, to właśnie implementacja niezapamiętująca żadnych różnic, lecz całe pliki powoduje, że Git jest tak szybki.

Można używać Gita z plikami binarnymi, ale z rozsądkiem.

2.2. Katalog .git, czyli repozytorium Gita

Można zadać pytanie: skoro Git przechowuje historię projektu, to musi ją przechowywać *gdzieś*. Gdzie to jest?

Mały eksperyment

```
mkdir experiment2
cd experiment2
git init ❶
ls -al
tree .git ❷
```

❶ Git prawdopodobnie odpowiedział:

Initialized empty Git repository in <somepath>/experiment2/.git/.

❷ W katalogu *.git* zostało utworzonych kilka plików i podkatalogów.

Niektóre z nich będą omawiane w dalszych rozdziałach.

Uwaga: użytkownicy Windowsa być może muszą użyć `tree.com .git`.

Podejrzewamy więc, że Git przechowuje historię projektu w podkatalogu *.git*. Tak rzeczywiście jest. Odnotujmy kilka konsekwencji tego faktu:

- Usunięcie tego katalogu spowoduje usunięcie **całej** historii projektu.
- Skopiowanie tego katalogu to zrobienie kopii zapasowej **całej** historii projektu. Może prymitywne, ale działa!
- Aby dowiedzieć się, czy bieżący katalog jest śledzony przez Gita (czyli w naszej nomenklaturze: czy jest *projektem* Gita), wystarczy sprawdzić, czy zawiera podkatalog *.git*.
- Nie jest to do końca prawdą, bo katalog *.git* musi zawierać pewne składniki, aby być gitowym repozytorium, ale jeśli katalog został utworzony przez Gita, to będzie zawierał wszystkie niezbędne składniki (patrz powyższe polecenie `tree .git`), a kto normalny tworzyłby katalog *.git* ręcznie?

No właśnie, spróbujmy utworzyć katalog *.git* ręcznie.

Mały eksperyment

```
mkdir experiment3
cd experiment3
mkdir .git
mkdir .git/objects
mkdir .git/refs
```

```
echo "ref: refs/heads/master" > .git/HEAD
tree .git
git status
```

Hm, niewiele trzeba, żeby Git uznał katalog *.git* za swoje repozytorium.

Mały eksperyment — cd.

```
touch a.txt
git add a.txt
git commit -m"init repo"
tree .git
cat .git/HEAD
cat .git/refs/heads/master
```

`HEAD` oraz pliki z *refs/heads* i *refs/objects* to byty, o których traktują następne rozdziały.

Sam katalog *.git* to **repozytorium Gita** (ang. *Git repository*), czyli wszystko, co Git wie o projekcie, w szczególności historia projektu (cała). Jest to tzw. repozytorium *lokalne*. Lokalne, bo operujemy na nim jak na zwykłym katalogu lokalnym.

Jeśli pracujemy sami i chcemy tylko przechowywać historię projektu lokalnie, wówczas takie lokalne repozytorium to wszystko, czego nam potrzeba. Zazwyczaj jednak współpracujemy z kimś, współdzieląc wykonaną pracę. Takie współdzielenie to, technicznie rzecz biorąc, przesyłanie zapamiętanych wersji projektu pomiędzy kilkoma repozytoriami. Tj. mogę do swojego repozytorium lokalnego pobrać (`git fetch`) wersje zapamiętane przez współpracowników w ich repozytoriach (lokalnych z ich punktu widzenia, zdalnych z mojego) oraz przesłać im (`git push`) wersje zapamiętane lokalnie u mnie (zdalne z ich punktu widzenia).

Repozytorium Gita to miejsce, gdzie Git przechowuje wszystko, czego potrzebuje do zarządzania projektem. W szczególności w repozytorium przechowywana jest cała historia projektu. Normalnie tym repozytorium jest podkatalog *.git*, będący podkatalogiem śledzonego katalogu (*projektu*).

2.3. Obiekty w repozytorium, graf commitów, mapa adresowana zawartością

2.3.1. Obiekt commit

Wiemy już, że każdy commit przechowuje migawkę projektu. Dodatkowo commit przechowuje też kilka innych ważnych informacji. Ściśle rzecz ujmując, każdy commit przechowuje następujące dane:

- pełną zawartość projektu (migawkę);
- *commit message*, czyli komunikat podany podczas tworzenia commita;
- *listę swoich rodziców* (ang. *parent commits*);

- autora (ang. *author*) — nazwę i e-mail;
- czas utworzenia commita;
- zatwierdzającego (ang. *committer*) — nazwę i e-mail;
- czas zatwierdzenia commita.

Commit przechowujący opisane wyżej treści uważany jest przez Gita za pewien rodzaj obiektu. Dla każdego swojego obiektu Git wyznacza jego identyfikator. Spróbujmy zaobserwować identyfikator i składniki commita.

Mały eksperyment

```
mkdir experiment4
cd experiment4
git init
touch a.txt
git add a.txt
git commit -m"init repo"
touch b.txt
git add b.txt
git commit -m"second commit"
```

Wyświetlenie informacji o aktualnym commicie

```
git show --name-only
```

Wyświetlony tekst

```
commit a2156780bfe1b5e0803f1b02dc8a3862dc3c1f72 (HEAD -> master)
Author: Jacek Drag <jacadrag@gmail.com>
Date:   Thu Aug 25 16:44:20 2022 +0200

    second commit

b.txt
```

Widzimy m.in. identyfikator commita (ciąg liter i cyfr).

Lub

Wyświetlenie informacji o commicie jako o obiekcie Gita

```
git cat-file -p HEAD
```

Wyświetlony tekst

```
tree 2bdf04adb23d2b40b6085efb230856e5e2a775b7
parent e9796c1ddf8508bf95da41c6118b9537ebef1b82
author Jacek Drag <jacadrag@gmail.com> 1661438660 +0200
committer Jacek Drag <jacadrag@gmail.com> 1661438660 +0200

second commit
```

Rzeczywiście możemy zobaczyć poszczególne składniki commita:

`tree`

Pod tą tajemniczą nazwą drzewo (ang. *tree/tree object*) przechowywana jest zawartość projektu (migawka).

`parent`

Identyfikator wskazujący commit rodzica (lista rodziców jest akurat jednoelementowa).

`author`

Autor, wraz ze znacznikiem czasu.

`committer`

Zatwierdzający, wraz ze znacznikiem czasu.

To samo co w `author`.

`commit message`

Treść komunikatu.

Pójdźmy krok dalej i wyświetlmy zawartość projektu zapamiętaną w commicie, czyli zawartość drzewa z commita.

Wyświetlenie zawartości drzewa

```
git cat-file -p HEAD^{tree}
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    a.txt
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    b.txt
```

Hm, mamy tu:

- jakieś nowe identyfikatory, a właściwie dwukrotnie taki sam nowy identyfikator;
- coś o jakimś *blob*;
- nazwy plików.

OK, nazwy plików odpowiadają tym utworzonym na dysku. Zdaje się mieć to sens.

A *blob*? Blob to (w pewnym uproszczeniu, później będzie to omówione dokładniej) treść pliku. W tym przypadku oba pliki zawierają taką samą treść, a konkretnie są puste. `e69de29bb2d1d6434b8b29ae775ad8c2e48c5391` to identyfikator pustej treści pliku.

Commit to obiekt utworzony poleceniem `git commit`. Każdy taki obiekt składa się z pięciu elementów:

- migawki zawartości projektu,
- komunikatu podanego podczas zatwierdzania,
- listy rodziców commita (każdy rodzic to też pewien commit),
- informacji o autorze wraz ze znacznikiem czasu powstania,
- informacji o zatwierdzającym wraz ze znacznikiem czasu zatwierdzenia.

2.3.2. Graf commitów

Przyjrzyjmy się temu wszystkiemu nieco dokładniej.

Na początek można pominąć zatwierdzającego i jego znacznik czasu (polecenie `git show` też uznało to za stosowne), gdyż zazwyczaj będą one identyczne z danymi autora. To, że dane te mogą być istotnie różne od danych autora, okaże się we właściwym czasie.

Niezwykle istotna jest natomiast *lista rodziców*. Ustala ona relację *bycia rodzicem* pomiędzy parami commitów. Dalej z tej relacji wynikają relacje bycia **przodkiem** (ang. *ancestor*) i bycia **potomkiem** (ang. *descendant*).

Mówimy, że commit `A` jest **rodzicem** (ang. *parent*) commita `B`, jeśli commit `A` znajduje się na liście rodziców commita `B`.

Intencja jest następująca i chyba jasna. Powiedzmy, że ostatnio zapamiętany commit to `A`. Dokonujemy pewnych zmian w projekcie i zatwierdzamy, w wyniku czego powstaje commit `B`. W pewnym — dość chyba jasnym — sensie commit `A` poprzedza (bezpośrednio) commit `B` w historii rozwoju naszego projektu. Tworząc (na nasze polecenie) commit `B`, Git automatycznie doda commit `A` do listy rodziców commita `B`. Co więcej, lista ta będzie jednoelementowa. Innymi słowy, można powiedzieć, że bieżący commit automatycznie staje się (jedynym) rodzicem nowo tworzonego commita. Lista ta:

- Najczęściej jest jednoelementowa.
- Jest pusta tylko dla jednego commita w repozytorium. Mianowicie tego, który został utworzony jako pierwszy, czyli Git nie miał skąd wziąć rodzica dla niego.
- No dobrze — da się wytworzyć więcej takich commitów, ale normalnie jest tylko jeden.
- Jeśli jest dwuelementowa lub zawiera ich więcej, to commit taki nazywamy merge commitem.
- Przy czym listę więcej niż dwuelementową spotyka się bardzo rzadko.

Commity wraz z relacją bycia rodzicem tworzą graf[1], dokładniej acykliczny graf skierowany (ang. *directed acyclic graph*, w skrócie DAG).

[1] Graf składa się z węzłów i krawędzi, czyli połączeń pomiędzy dwoma wybranymi węzłami. W grafie skierowanym krawędź ma określony kierunek. Możemy na przykład wyobrazić sobie graf jako miasta na mapie wraz z drogami (nieprzecinającymi się, nie może być skrzyżowań) łączącymi niektóre pary miast. Jeśli drogi będą jednokierunkowe, to mamy do czynienia z grafem skierowanym. Graf jest acykliczny, gdy nie ma cykli. To znaczy, że startując z dowolnego węzła i jadąc po krawędziach, nigdy nie da się wrócić do początkowego węzła. Czyli nie da się zrobić zamkniętej pętli.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Kontroluj rozwój Twojego oprogramowania

Git to aktualnie najpopularniejszy na świecie system kontroli wersji. Dzięki niemu możliwe jest śledzenie modyfikacji w kodzie źródłowym i łączenie zmian w plikach dokonanych przez różne osoby w różnym czasie. Git oferuje wiele możliwości, jest bardzo elastyczny i nie narzuca sposobu, w jaki należy w nim pracować. Można to robić na przykład przy użyciu programów graficznych, które znacznie ułatwiają realizację niektórych celów, szczególnie w zakresie przeglądania historii pracy nad kodem i rozwiązywania konfliktów.

Ta książka koncentruje się głównie na rozwijaniu repozytorium kodu, czyli tworzeniu grafu commitów zawierających poszczególne wersje. Do realizacji tego zadania idealnym, bo najpotężniejszym narzędziem jest wiersz poleceń — i właśnie z niego korzystamy w poradniku. Druga kwestia, którą się zajmujemy, to próba zrozumienia, co Git mówi do nas w trakcie pracy. Tak, mówi, ponieważ gdy zlecimy mu wykonanie jakiejś komendy, Git najprawdopodobniej nie tylko ją wykona, ale także skomentuje stan obecny, co nieco podpowie, zwróci uwagę, jeśli coś się nie uda, i wskaże sposób, jak to poprawić.

Uruchom Gita, otwórz książkę i poznaj:

- Koncepcje (byty, abstrakcje), którymi posługuje się Git
- Polecenia wysokiego poziomu wraz z najczęstszymi przypadkami ich użycia
- Sposoby pracy w środowisku lokalnym i rozproszonym

Helion

helion.pl

HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-1249-6

9 788328 912496

Cena: 39,90 zł