Technologie informacyjne
przed 4 wykładem

Andrzej Giniewicz

20.03.2024

W dzisiejszej porcji materiału dowiemy się, co to jest Git i jak z niego korzystać.

1 O systemach kontroli wersji

Git rozwiązuje dwa ważne problemy — synchronizacji z powodu replikacji oraz synchronizacji z powodu rozproszenia. W obu przypadkach musimy zmierzyć się z synchronizacją, czyli różnymi wersjami plików w różnych miejscach, jednakże powody tego, że pliki występują w różnych miejscach, mogą być różne.

Replikacja jest często spotykanym powodem konieczności synchronizacji. Dane nigdzie nie są bezpieczne — dysk w komputerze może ulec awarii, dysk przenośny również może ulec awarii lub zostać skradziony. Płyty kompaktowe, czy to CD, DVD, czy BD, mają bardzo różną żywotność, zależnie od standardu nagrywania — niechlubną chwałą cieszą się dwuwarstwowe płyty DVD+R, których żywotność przy właściwym przechowywaniu w suchym i chłodnym miejscu bez kurzu, wynosi zaledwie 5 – 10 lat — nie nazwalibyśmy tego dobrą metodą archiwizacji. Pamięci USB zwykle wytrzymują 10 – 100 tysięcy cykli odczytów i zapisów, co może się wydawać dużą liczbą zapisów, ale często przekłada się to na średnio 10 lat użytkowania. Przechowywanie danych w chmurze może wydawać się bezpieczniejsze, ale pojawia się problem zabezpieczeń dostawcy usługi chmurowej — czy regularnie aktualizuje zabezpieczenia i tworzy kopie zapasowe danych, które umieszcza w innej fizycznej lokalizacji? Było kilka firm, które przechowywały dane w jednym miejscu i niedawny pożar we francuskiej serwerowni jednego z większych dostawców, kilka z nich unieruchomił. Ogromne firmy często dbają o zabezpieczenia i duplikacje, ale z kolei często rezygnują z jakichś usług i dają ograniczony czas na przeniesienie danych w inne miejsce. Jeśli przegapimy wiadomości o przenosinach, możemy stracić dane. W skrócie oznacza to, że nie ma pewnego, w stu procentach bezpiecznego miejsca na dane. Nawet nagrane na płycie, zamkniętej w sejfie, mogą „wyparować". Wobec tego, aby zabezpieczyć dane, tworzymy repliki. Posiadanie wielu kopii danych w różnych miejscach tworzy nowe wyzwanie — synchronizacji. Czy nasze dane na dysku komputera, w chmurze i na dysku zewnętrznym są zgodne? Sprawdzanie kilku plików jest proste, ale jeśli sprawdzamy ich bardzo dużo, przyda się jakieś narzędzie do zarządzania wersjami, które nam pomoże w upewnieniu się, że wszędzie jest najnowsza wersja wszystkich plików, na których nam zależy.

Innym naturalnym miejscem, w którym pojawiają się repliki, jest współpraca kilku osób nad jednym projektem. Naturalnie, każdy z członków zespołu, ma na swoim komputerze kopię wszystkich danych, na których pracuje. Regularnie należy sprawdzać, czy nie trzeba połączyć zmian naszych ze zmianami innych osób. Jeśli każda osoba pracuje nad niezależną częścią projektu, to jest to stosunkowo proste — choć taki rodzaj pracy nie wykorzystuje w pełni potencjału, jaki daje praca w grupie — różnych spojrzeń na tę samą część projektu i różnych pomysłów, z których na drodze dyskusji i syntezy czasem wybiera się najlepszy a czasem, powstaje połączenie będące lepsze, niż każdy z pomysłów z osobna. Jeśli kilka osób pracuje nad tą samą częścią dokumentu w różnych miejscach w tym samym czasie, nie obejdzie się bez konfliktów, czyli dwóch niezależnych zmian tego samego fragmentu na różne sposoby. Bardzo pomocny jest system, który takie konflikty zauważy i nie pozwoli nam na „nadpisanie" cudzych zmian, tylko powiadomi nas o konflikcie i pozwoli go rozwiązać — być może czytając tekst od razu będzie jasne, co się wydarzyło? A być może będziemy musieli przedyskutować zmianę i wymyślić trzecią? Wyobraźmy sobie, że dwie osoby pracują nad napisaniem wspólnego dokumentu. Na początku obie miały w dokumencie

```
Ala ma kota.
```

Po czym jedna osoba zmieniła plik na

```
Ala ma psa.
```

a druga na

```
Ala ma chomika.
```

Żaden automat nie jest w stanie rozstrzygnąć, jaka wersja tej historii jest prawidłowa. Są to konfliktujące zmiany. Jeśli system nam o nich powie, możemy zwołać zebranie zespołu, który ustali między innymi:

- Skąd się wzięła rozbieżność?
- Co się stało z kotem?
- Czy Ala ma teraz psa, czy chomika, a może ma oba zwierzątka?
- Czy przez pewien czas Ala miała, a może ma nadal, trzy zwierzątka?
- Czy kot zjadł chomika i dlatego dostał nakaz eksmisji, a w jego miejsce wprowadził się pies?

Gdyby system nie powiadomił nas o konflikcie, jedna z osób mogłaby nadpisać zmiany drugiej osoby i mogłaby przejść autorom pliku koło nosa całkiem ciekawa i rozbudowana historia. Nie mówiąc o tym, że ostateczna wersja pliku może mieć informacje niezgodne ze stanem faktycznym.

Systemy kontroli wersji, do których należy właśnie Git, pomagają nam rozwiązać oba te wyzwania. Systemy kontroli wersji pozwalają na śledzenie zmian w plikach od początku ich powstania lub pierwszego dodania do systemu kontroli wersji, do obecnego momentu. Pozwalają porównywać między sobą wersje u różnych osób i pomagają rozwiązywać konflikty. Pozwalają też porównywać wersje w różnych punktach historii i cofnąć się do dowolnego miejsca w przeszłości.

2 Historia systemów kontroli wersji

Systemy kontroli wersji nie są nowym pomysłem. Pierwsze systemy kontroli wersji były **lokalnymi systemami kontroli wersji**. Oznacza to, że pomagały śledzić zmiany w plikach znajdujących się na jednym komputerze. Należały do nich między innymi SCCS z 1972 roku oraz RCS z 1982 roku. Ten pierwszy po prostu numerował wersje plików, na przykład „pamiętnik.txt.0", „pamiętnik.txt.1" i tak dalej, przy czym dostarczał narzędzi do porównywania i przywracania wersji. Zasadniczo automatyzował to, co wielu z nas robi samodzielnie, tworząc wersje dokumentów, np.: „plan v1.txt", „plan v2.txt", i tak dalej. System RCS wprowadził innowację, która spowodowała znaczne zmniejszenie ilości zajmowanego miejsca na dysku — zamiast całych plików, nawet jeśli zmieniły się niewiele, pamiętał tylko różnice pomiędzy dwoma plikami. Wymagał bardziej zaawansowanych mechanizmów, aby podróżować po historii plików, ale oszczędzał sporo miejsca, przez co stał się o wiele bardziej praktyczny. Niemniej jednak oba te systemy nie pomagały współpracować większej grupie osób.

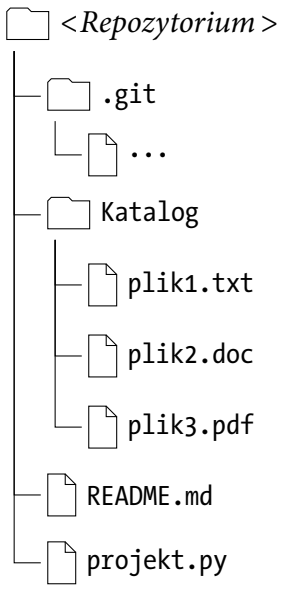
Kolejną innowacją było wprowadzenie **scentralizowanych systemów kontroli wersji**. Systemy scentralizowane pracują z serwerem, który przechowuje zmiany od początku istnienia projektu, do obecnego momentu. Kilka osób może pobrać z serwera kopię roboczą plików, wprowadzić zmiany i następnie wysłać je na serwer, aby kolejne osoby mogły ściągnąć zmiany. Większość systemów scentralizowanych może pracować lokalnie, jeśli uruchomimy własny serwer na lokalnym komputerze. Do popularnych systemów kontroli wersji należał CVS z 1990 roku, który bazował na koncepcji systemu RCS, ale przenosił te pomysły do aplikacji w architekturze klient-serwer. Pojawiło się też wiele komercyjnych systemów. Jednym z nich był Clear Case z 1992 roku, który kładł mocny nacisk na zarządzanie różnymi wariantami projektu, czyli tak zwane odgałęzienia. Komercyjny system Perforce z 1995 roku, który obecnie jest jednym z popularniejszych scentralizowanych rozwiązań, dostarcza interfejs graficzny, który uchodzi za zrozumiały również dla osób, które nie potrafią programować. W grupie systemów scentralizowanych znajduje się również darmowy SVN powstały w 2000 roku, który choć nie wprowadza wiele nowości względem CVS, to jest systemem zdecydowanie lepiej zaprojektowanym i korzystającym z doświadczeń swoich poprzedników. Prawdopodobnie dziś spośród scentralizowanych systemów najczęściej spotkamy właśnie darmowy SVN oraz komercyjny Perforce.

Najnowszą innowacją w dziedzinie systemów kontroli wersji było wprowadzenie **rozproszonych systemów kontroli wersji**. System rozproszony odpowiada na wady systemów scentralizowanych. W systemach scentralizowanych, jeśli nie mamy dostępu do serwera, niewiele możemy zrobić. Możemy edytować pliki, ale nie sprawdzimy historii i nie zapiszemy zmian do śledzenia. Co więcej, gdy serwer ulegnie awarii, jesteśmy w dużej mierze unieruchomieni, aż administrator przywróci kopię zapasową — o ile ma ją gdzieś w bezpiecznym miejscu. Systemy rozproszone idą dalej — nie ma jednego serwera, który jest krytycznym punktem całego systemu. Każda osoba ma w pełni funkcjonalny serwer z całą historią wstecz. Dzięki temu można sprawdzać historię, wykonywać zmiany i normalnie pracować nawet w samolocie czy innym miejscu, w którym nie możemy skorzystać z Internetu. Zaletą tego rozwiązania jest też to, że wszystkie operacje na historii są o wiele

szybsze. Potem gdy już uzyskamy dostęp do sieci, wystarczy wymienić się danymi z inną osobą. Systemy rozproszone pozwalają to zrobić na różne sposoby — bezpośrednio przez kontakt dwóch osób i przesłanie zmian lub za pomocą centralnego repozytorium, które najczęściej pełni wtedy rolę wyłącznie punktu wymiany. Powstały w 1999 roku BitKeeper był jednym z pierwszych systemów rozproszonych. Udostępniał on darmową wersję dla twórców darmowego oprogramowania i był wykorzystywany przez wiele projektów, w tym jądro Linuxa. Gdy w 2005 roku ogłoszono, że darmowa licencja nie będzie już dostępna, zaczęły się pojawiać alternatywne rozwiązania, takie jak Mercurial i Git. Choć Mercurial wciąż jest popularny, Git zdominował rynek systemów kontroli wersji zarazem w świecie Open Source, jak i komercyjnym. Co ciekawe Git został napisany przez Linusa Torvaldsa, twórcę Linuxa, jako narzędzie specjalnie do zarządzania jądrem Linuxa. Został napisany mniej więcej w tydzień a po miesiącu był na tyle gotowy, aby zarządzanie jądrem przenieść na niego. Po 8 miesiącach od pierwszej wersji, Git 1.0 był stabilny. W roku 2022 Git był używany regularnie przez 96,65% developerów na świecie[1]. Co ciekawe, BitKeeper, który „zmotywował" twórców do stworzenia innych systemów kontroli wersji, praktycznie nie jest już używany. Jego ostatnią wersję udostępniono na licencji Open Source w 2014 roku i teraz również jest darmowym narzędziem.

3 Podstawy Gita

Git przechowuje zmiany w tak zwanym **repozytorium**. Repozytorium to katalog na dysku, który zawiera całą historię zmian wstecz, wszystkich śledzonych plików. Obok repozytorium w gicie jest również **kopia robocza**, czyli obecna wersja wszystkich plików. Historia przechowywana jest w podkatalogu o nazwie `.git`. Przykład standardowego repozytorium może wyglądać następująco.



Wszystkie informacje, takie jak konfigurację i historię, Git przechowuje w katalogu `.git`. Pozostałe pliki są kopią roboczą.

[1] Źródło: `https://survey.stackoverflow.co/2022/#version-control-version-control-system-prof`.

Z katalogu `.git` zwykle nie musimy i nie powinniśmy korzystać samodzielnie, do zarządzania jego zawartością służy komenda `git` linii komend. Poniżej opiszę ogólny zarys działania Gita wraz z najważniejszymi nazwami komend. Git posiada bardzo dobrą dokumentację i każdą komendę możemy sprawdzić, na przykład komendą `git config --help` zobaczymy jak używać komendy `git config`. Wszystkie te komendy będą omówione dokładniej na laboratoriach.

Aby repozytorium pojawiło się na naszym komputerze, możemy stworzyć nowe, puste repozytorium komendą `git init` lub sklonować je z istniejącego repozytorium komendą `git clone`, co spowoduje skopiowanie całej historii. Każde repozytorium może mieć ustawione jedno lub więcej repozytorium zdalne. Jeśli klonujemy repozytorium, automatycznie to, z którego klonowaliśmy, stanie się dla nas repozytorium zdalnym o nazwie `origin`. Jeśli będziemy chcieli wysyłać zmiany, to domyślnie będą trafiać właśnie tam — na repozytorium, z którego klonowaliśmy naszą kopię. Można ustawić więcej repozytoriów zdalnych komendą `git remote` i w ten sposób przesyłać zmiany w inne miejsca niż to, z którego pobraliśmy pliki.

W repozytorium może być wiele wariantów plików. Warianty takie nazywamy odgałęzieniami (ang. branch). Domyślne odgałęzienie nazywa się najczęściej `master` lub `main`, zależnie od konfiguracji programu Git u osoby, która to repozytorium tworzyła. Do zarządzania odgałęzieniami służy komenda `git branch`. Do przełączania repozytorium na konkretne odgałęzienie projektu i konkretną wersję w historii, którą chcemy zobaczyć, służy komenda `git checkout`.

Kiedy już ściągniemy zmiany, wybierzemy odpowiednie odgałęzienie i wersję, pora na wprowadzenie zmian. Wykonujemy je, edytując pliki w kopii roboczej w ten sam sposób, co zwykle — w naszym ulubionym edytorze tekstu, grafiki, muzyki lub dowolnych innych narzędziach. Gdy wprowadzimy jakieś zmiany, możemy się upewnić, że Git je widzi komendą `git status`. Komenda ta może nam powiedzieć, że widzi pliki nieśledzone lub zmienione, które należy dodać komendą `add`. Dodawanie plików powoduje zapamiętanie ich obecnej wersji na tak zwanej scenie (ang. stage). Pliki ze sceny możemy wycofać do obszaru roboczego, ponieważ jeszcze nie zostały zapamiętane.

Wszystkie zmiany w gicie są pamiętane w tak zwanych commitach. Commit musi być opatrzony komentarzem opisującym zmianę oraz naszymi danymi: nazwą (zwykle imieniem i nazwiskiem) oraz e-mailem. Dane ustawiamy raz komendą `git config`. Gdy ustawimy dane osobowe, możemy zapamiętać pliki na scenie i stworzyć z nich jedną zmianę komendą `git commit`. Zmiana ta będzie zapisana na naszym komputerze w naszym repozytorium. Możemy spróbować wysłać ją na inne repozytorium komendą `git push`. Jeśli mamy uprawnienia do zapisu oraz zmiana nie wprowadza konfliktu, pojawi się na serwerze, aby inni mogli ją pobrać. Jeśli pojawi się konflikt, będziemy musieli pobrać zmiany i rozwiązać konflikt. Pobieranie zmian, przy konflikcie lub bez niego, odbywa się komendą `git pull`. Przy konflikcie pojawi się narzędzie do ich usuwania. Gdy zakończymy rozwiązywanie konfliktów, możemy stworzyć nową zmianę i przesłać ją ponownie na serwer.

Git nie przechowuje różnic, tylko całe pliki, które kompresuje co jakiś czas, aby zaoszczędzić miejsce. Dzięki temu cała historia zwykle ma nie więcej niż dwa razy tyle, co kopia robocza. Aby łatwo porównać historie u dwóch osób, nie są używane numery wersji

takie, jakie znamy, czyli liczby naturalne, ale sumy kontrolne SHA-1 wersji plików oraz historii. Dzięki temu, jeśli mamy dwa repozytoria, wystarczy porównać sumy kontrolne SHA-1 i w ten sposób można ustalić ostatnią wspólną wersję, dzięki czemu możemy ustalić, co się zmieniło. Załóżmy, że rozwijamy nasz projekt w formie liniowej. Zapiszmy zmiany wraz z początkami sum kontrolnych w formie listy, idąc od dołu do góry.

- `ca07ec7`: Pierwsza wersja sterownika
- `12187fa`: Dokumentacja prototypu wehikułu czasu
- `18adc16`: Zaczynam super projekt, pierwsze pliki!

Załóżmy, że projekt w tej wersji udostępniliśmy znajomemu fizykowi, aby przyjrzał się pomysłom i wprowadził poprawki. Oczywiście nie mogąc się doczekać, kontynuujemy pracę.

- `c2c9da0`: Metody do łączenia z bazą danych
- `ca07ec7`: Pierwsza wersja sterownika
- `12187fa`: Dokumentacja prototypu wehikułu czasu
- `18adc16`: Zaczynam super projekt, pierwsze pliki!

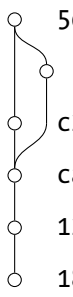
W tym samym czasie znajomy fizyk wprowadza poprawki.

- `0858a98`: Z rozpędu podzieliłeś przez zero!
- `ca07ec7`: Pierwsza wersja sterownika
- `12187fa`: Dokumentacja prototypu wehikułu czasu
- `18adc16`: Zaczynam super projekt, pierwsze pliki!

Ups — jak widać, pojawiła się pomyłka. Na szczęście ktoś sprawdził obliczenia. Niestety, projekty są teraz w różnych, konfliktujących wersjach. Gdy ściągniemy dane, repozytorium będzie w następującym stanie

- `0858a98`: Z rozpędu podzieliłeś przez zero!
- `c2c9da0`: Metody do łączenia z bazą danych
- `ca07ec7`: Pierwsza wersja sterownika
- `12187fa`: Dokumentacja prototypu wehikułu czasu
- `18adc16`: Zaczynam super projekt, pierwsze pliki!

Jak widać, powstało odgałęzienie projektu. Możemy albo ściągnięte zmiany nazwać nowym branchem i kontynuować pracę na głównym odgałęzieniu, albo częściej, będziemy chcieli zintegrować zmiany z obu historii w jedną. Taka operacja nazywa się scaleniem (ang. merge). Polega na przejrzeniu obu zmian i wprowadzeniu jednej, która je uwspólnia.



`5dbfeda`: Połączenie zmian, dzięki za poprawki!

`0858a98`: Z rozpędu podzieliłeś przez zero!

`c2c9da0`: Metody do łączenia z bazą danych

`ca07ec7`: Pierwsza wersja sterownika

`12187fa`: Dokumentacja prototypu wehikułu czasu

`18adc16`: Zaczynam super projekt, pierwsze pliki!

Z takiego repozytorium znajomy fizyk może pobrać zmiany i będzie widział zarazem swoją zmianę, jak i naszą zmianę.

Odgałęzienia nie zawsze powstają przez przypadek. Czasem powstają wtedy, gdy cofamy się w historii, by sprawdzić, czy alternatywne rozwiązanie jakiegoś problemu da lepsze efekty. Wtedy możemy cofnąć się do jakiejś wersji, wprowadzić zmiany, wrócić do wersji obecnej i tak skakać pomiędzy dwiema alternatywnymi wariantami plików. Odgałęzienia są też często używane przez twórców oprogramowania, którzy w głównym odgałęzieniu rozwijają nową funkcjonalność, a gdy wydają jakąś wersję, na przykład 3.1, tworzą osobny branch, na który wysyłają tylko zmiany z poprawkami błędów. Dzięki temu mogą jednocześnie przygotowywać wersję 3.1.1 z minimalnymi poprawkami dla obecnych użytkowników oraz eksperymentować na głównym odgałęzieniu.

Git dostarcza też wielu innych pomocnych narzędzi, które pomagają w pracy z dużymi repozytoriami. Do ciekawych należy komenda `git bisect`, której podajemy ostatni znany numer wersji, kiedy coś działało, pierwszy znany numer wersji, w którym coś nie działało, po czym Git wybiera wersję w połowie historii i prosi o sprawdzenie, czy dana funkcjonalność działa. W zależności od tego, czy odpowiemy tak, czy nie, przesunie się w przeszłość lub przyszłość, zgodnie z metodą bisekcji. Dzięki takiemu poruszaniu się po historii bardzo szybko jesteśmy w stanie namierzyć tę jedną konkretną zmianę, która spowodowała, że coś przestało działać. Jeśli widzimy, jakie linie kodu spowodowały, że pojawiła się awaria, będzie dużo łatwiej ją usunąć — od razu wiemy, gdzie szukać. Co więcej, jeśli mamy już zmianę z komentarzem i autorem, wiemy też do kogo napisać maila z pytaniem, aby razem z autorem tej konkretnej zmiany poszukać rozwiązania. Innym sposobem szukania osoby, przez którą coś nie działa, jest komenda `git blame`, której podajemy jako argument plik. Wyświetla ona zawartość pliku, a na lewo od każdej linii podaje, kto i kiedy jako ostatni ją zmieniał. Jest to inny, bardzo praktyczny sposób zlokalizowania autora zmian, co przy projektach na większą skalę jest bezcenne.

4 Serwery Gita

Do zajęć udostępniony mamy serwer dostosowany specjalnie do naszych potrzeb, ale po zajęciach, do innych projektów na innych kursach, warto rozejrzeć się za serwerem Gita, który pozwala na skorzystanie z prywatnych i publicznych repozytoriów. Na szczególną uwagę zasługują GitLab oraz GitHub, które są dwoma najpopularniejszymi serwerami

Gita. Do bezpiecznego połączenia się z serwerem, potrzebujemy klucza SSH. W przypadku GitLab i GitHub podajemy go w profilu użytkownika po rejestracji na stronie. W przypadku serwera do zajęć przesyłamy go do osoby prowadzącej laboratoria.

Serwer do zajęć udostępnia jedynie podstawową funkcjonalność, natomiast serwery GitLab i GitHub pozwalają na wiele więcej, choć mogą początkowo przytłaczać mnogością możliwości. Do najważniejszych możliwości należy zalecany na nich sposób pracy. Udostępniamy tam repozytorium, do którego tylko my mamy uprawnienia. Każda osoba może stworzyć tak zwany fork, czyli odgałęzienie projektu. W rzeczywistości jest to klon repozytorium, który jest na tym samym serwerze, ma tę samą nazwę, ale jest na koncie innego użytkownika. Forki tworzymy odpowiednim przyciskiem na stronie internetowej serwera. Gdy wprowadzimy jakieś zmiany, możemy nacisnąć przycisk „Make pull request", co powoduje w repozytorium osoby, której repozytorium sklonowaliśmy, pojawienie się informacji, że jest jakaś zmiana, którą chcemy przesłać. Wypełniamy odpowiedni formularz opisujący zmianę i potwierdzamy. Właściciel oryginalnego repozytorium ma przycisk do zaakceptowania lub odrzucenia zmiany, w zależności od tego, czy mu się podoba czy nie — przykładowo czy przeszła wszystkie testy, kontrolę jakości, kontrolę stylu i inne filtry, które narzucił sobie w projekcie jego właściciel. Ten sposób pracy jest bardzo popularny w projektach Open Source, ponieważ nie ma potrzeby dawania dostępu do głównego repozytorium wszystkim ludziom, a wciąż pozwalamy im w stosunkowo prosty sposób podzielić się swoim kodem i zmianami.