.git (sheadovas/poradniki/goto/git/)

Paź 13, 2015 / goto (sheadovas/category/poradniki/goto/)

Wstęp do systemu kontroli wersji wraz z przykładową konfiguracją przenośnego repozytorium.

W tym wpisie zajmiemy się czymś co prędzej, czy później przyda się każdemu programiście, czyli systemem kontroli wersji, a konkretniej o git'cie.

To proste narzędzie pozwala na monitorowanie zmian w projekcie i w razie potrzeby na ich łatwe cofnięcie (np. poprzednia wersja była lepsza), jest także przydatne w momencie gdy projekt piszemy w grupie i chcielibyśmy zadbać o to aby każdy członek zespołu mógł pracować na tej wersji kodu, która jest aktualna.

Wysiłek potrzebny do korzystania z .git'a jest nikły, a korzyści są olbrzymie, jeżeli ktoś jest ciekawy czym dokładnie jest system kontroli wersji (https://pl.wikipedia.org/wiki/System_kontroli_wersji), czy też git (https://pl.wikipedia.org/wiki/Git_%28oprogramowanie%29) to odsyłam do polecam poczytać o tym sobie samemu.

Podstawy

Instalacja

Aby móc używać git’a to musimy sobie go na początku pobrać, w przypadku systemów opartych na Debianie wystarczy wpisać komendę:

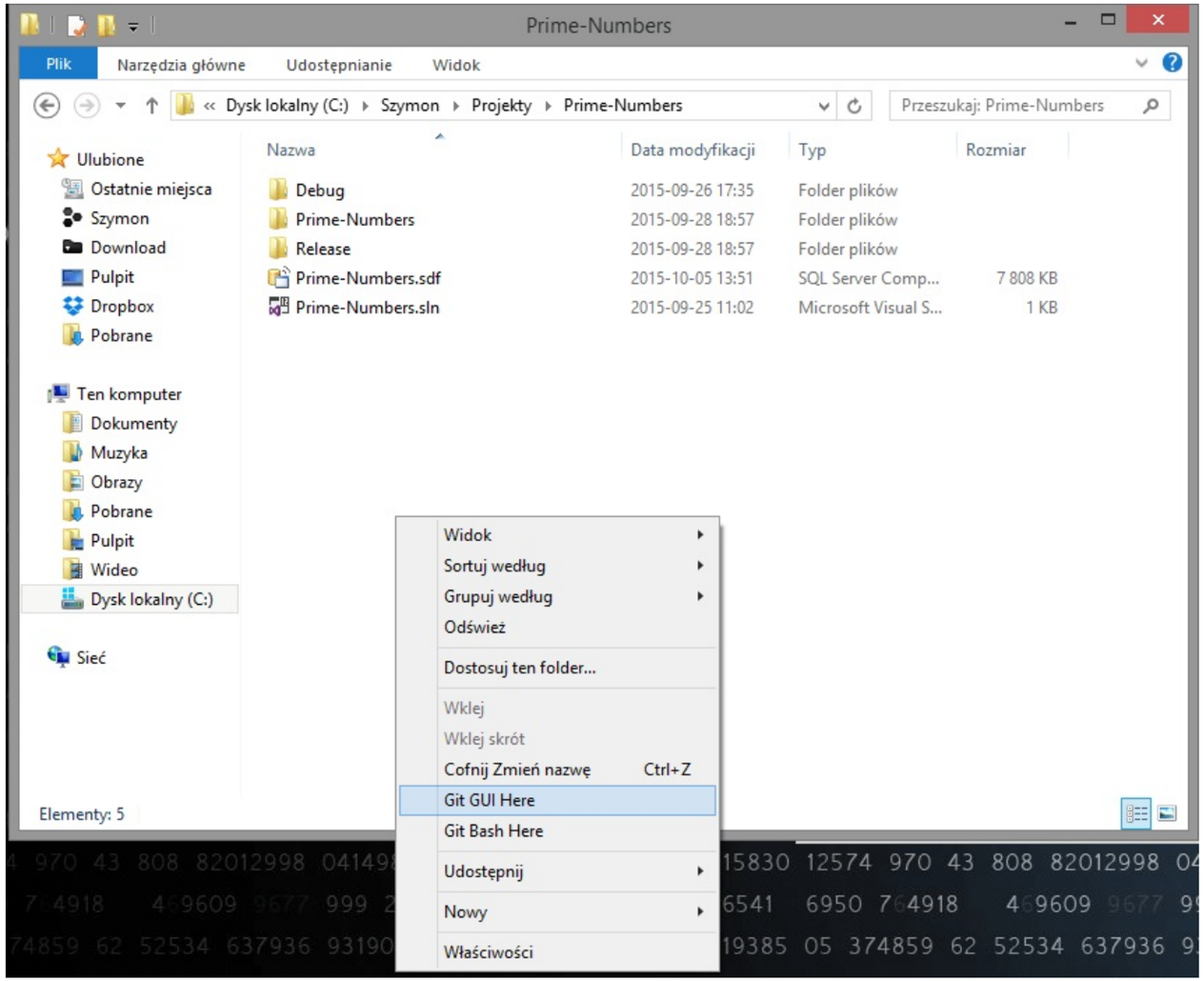
```
Instalacja git'a na systemach opartych na Debianie
$ apt-get install git
```

W przypadku innych Linuksów zmienia się jedynie ta część komendy służąca do pobierania i instalowania paczek.

Jeżeli jesteście użytkownikowi Windowsa, to polecam pobrać Wam tego (https://git-for-windows.github.io/) „klienta”, przy procesie instalacji zalecam zostawienie opcji domyślnych.

Tworzenie repozytorium

By utworzyć repozytorium udajemy się do folderu projektu następnie w nim klikamy PPM i wybieramy *Git Bash Here*, teraz powinna nam się pojawić pusta konsola Git’a.



(https://i0.wp.com/www.shead.ayz.pl/wp-content/uploads/2015/10/git-scrn1.png)

Tworzenie repozytorium odbywa się przy użyciu komendy:

```
$ git init
```

W przypadku pomyślnego utworzenia repozytorium w folderze powinien się pojawić ukryty folder „.git”, a w konsoli powinniśmy zobaczyć komunikat: „Initialized empty Git repository in <ścieżka pliku projektu>/.git/”.

Dodawanie plików do repozytorium

Obecnie co prawda posiadamy repozytorium, jednak jest puste. Aby dodać nowy plik do repozytorium należy wpisać:

```
C++
```

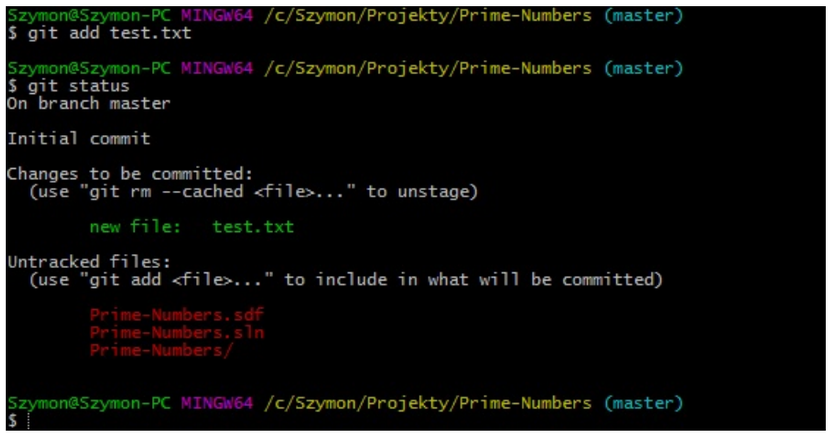
```
$ git add <nazwa pliku>
```

Jeżeli chcemy dodać wszystkie pliki znajdujące się w tym folderze (a także w podfolderach), to zamiast nazwy pliku piszemy ‚.' (kropkę). Możemy także przy użyciu ‚*' dodać pliki składające się z pewnej frazy, np.:

```
$ git add example.txt # doda do repozytorium jedynie plik "example.txt"
$ git add *.png # doda wszystkie pliki z rozszerzeniem .png
$ git add a*z # doda wszystkie pliki, których pierwsza litera to 'a' i ostania to 'z'
$ git add . # doda wszystkie pliki
```

Sprawdzenie stanu plików

Dość przydatną komendą jest `$ git status` która pokazuje, które pliki zostały dodane do repozytorium (na zielono pokazuje te, które zostaną zaktualizowane przy najbliższym commit'e, na czerwono te które nie są ignorowane, a ich zmiany nie zostały uwzględnione).



(https://i1.wp.com/www.shead.ayz.pl/wp-content/uploads/2015/10/git-scrn2.png)

Lista ignorowanych plików

Nie da się ukryć, że po prostu wielu plików nie będziemy chcieli dodawać do repozytorium, a dodawanie ich pojedynczo jest po prostu niewygodne. Jeżeli mamy takie pliki, to możemy ich nazwy, czy wzorce umieścić w pliku o nazwie „.gitignore", a sam plik należy umieścić w głównym folderze projektu.

Najprościej jest go utworzyć przy użyciu komendy `$ touch .gitignore`.

Jeżeli nie chcemy w swoim repozytorium plików „.png" to w tym pliku piszemy:

Wnętrze pliku .getignore C++
```
*.png
```

W sieci jest mnóstwo gotowych plików do ignorowania plików tworzonych przez IDE, np. w przypadku projektu tworzonego dla Visual Studio ignoruje się następujące pliki:

.gitignore dla Visual Studio i C++ C++
```
```

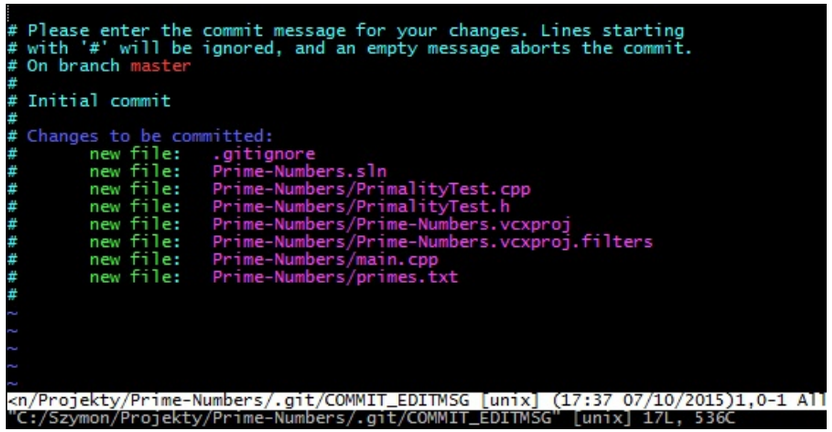
```
## Ignore Visual Studio temporary files, build results, and
## files generated by popular Visual Studio add-ons.

# User-specific files
*.suo
*.user
*.userosscache
*.sln.docstates

# User-specific files (MonoDevelop/Xamarin Studio)
*.userprefs

# Build results
[Dd]ebug/
[Dd]ebugPublic/
[Rr]elease/
[Rr]eleases/
x64/
x86/
build/
bld/
[Bb]in/
[Oo]bj/

# Visual Studio 2015 cache/options directory
.vs/
# Uncomment if you have tasks that create the project's static files in wwwroot
#wwwroot/

# MSTest test Results
[Tt]est[Rr]esult*/
[Bb]uild[Ll]og.*

```

Od teraz te pliki nie będą uwzględniane przez git'a.

Zatwierdzanie zmian

Do tej pory nie zapisaliśmy żadnych zmian, a jedynie dodawaliśmy te rzeczy, które chcielibyśmy uwzględnić przy najbliższej aktualizacji, aby zatwierdzić zmianę piszemy: `git commit -m 'nazwa commita'` lub `git commit` , jeżeli chcemy zobaczyć log plików, które zostaną zmienione w repozytorium.

Po uruchomieniu krótszej wersji komendy ukaże nam się „okno”



(https://i0.wp.com/www.shead.ayz.pl/wp-content/uploads/2015/10/git-scrn3.png)

Widzimy tutaj listę plików, które zostaną dodane/zmienione w repozytorium. Z racji, że każdy commit musi mieć swoją nazwę to musimy wejść w tryb wstawiania (klikamy I na klawiaturze), dzięki czemu możemy mu nadać nazwę. Następnie musimy wyjść z trybu wstawiania przy użyciu [Esc] oraz wpisujemy „:wq” w konsoli, aby zapisać zmiany i wyjść z tego widoku, co zatwierdzamy klawiszem [Enter].

Jeżeli teraz wywołasz `$ git status`, to powinieneś ujrzeć komunikat mówiący o tym, że wszystkie zmiany zostały naniesione do lokalnego repozytorium:

```
$ git status
On branch master
nothing to commit, working directory clean
```

Pracowanie na kopii

Każde repozytorium posiada tzw „branch’e”, które pozwalają pracować na kopii kodu, tak że zmiany są wciąż widoczne w repozytorium, ale nie oddziałują na główną (master) i najważniejszą „gałąź” projektu bezpośrednio. Przydaje się to np. gdy pracujesz nad nowym featerem, czy naprawą bug’u, dzięki czemu reszta zespołu nie musi się martwić tym, że to co robisz w jakiś sposób będzie oddziaływało na ich pracę. Gdy skończysz to możesz nanieść zmiany z twojego branch’a z powrotem do głównego, przy użyciu jednej komendy (pomijając sytuacje gdy występują konflikty).

Tworzenie nowego branch’a odbywa się przy użyciu linii: `$ git branch <nazwa>`

Aby zmienić aktywny branch, należy wpisać `git checkout <nazwa>`. Praca na nowym branchu działa dokładnie tak samo, jak poprzednio. Najlepiej zilustruje do przykład:

Utwórzmy nowy branch o nazwie *Test*. Następnie przełączmy się do niego i utwórzmy teraz nowy plik o nazwie „test.txt”, a także zatwierdźmy nasze zmiany przy użyciu commit’a.

Przykład

```
$ git branch Test
$ git checkout Test
$ git touch test.txt # możemy teraz go jakoś zedytować jeżeli chcemy
$ git add test.txt
$ git commit -m 'Dodanie pliku tekstowego'
```

Zauważ co się stanie w momencie gdy przełączysz się teraz z powrotem na master’a (`$ git checkout master`), plik test.txt zniknął! Jeżeli wykonaliśmy jakieś zmiany na reszcie repozytorium to zastalibyśmy je w dokładnie tym stanie, przy jakim je kopiowaliśmy.

Załóżmy, że dodaliśmy nowy feature i chcielibyśmy scalić nasz brach, z jakimś innym. Wtedy przechodzimy do tego branch’a, do którego chcemy nadpisać zmiany (w naszym przypadku do master’a) oraz używamy komendy: `$ git merge <nazwa>` (w przykładzie zamiast *<nazwa>* piszemy *Test*).

Czasami może się zdarzyć, że ktoś zmieni branch’a, do którego chcielibyśmy przenieść naszego branch’a (np. chcemy przenieść zmiany z *Test* do *master*, lecz ktoś w między czasie naniósł zmiany w tych samych plikach co my i naniósł je do *master*) i otrzymamy informację o konflikcie. Wtedy należy skorzystać z narzędzia do rozwiązywania konfliktów.

Manipulacja commit’ami

Skoro już wiemy jak w najprostszy sposób możemy używać repozytorium to zajmijmy się czymś ciekawszym.

Załóżmy, że przez przypadek przy robieniu commit’a dodaliśmy zbyt wiele plików. Aby usunąć dowolny plik z poczekalni należy użyć

`$git reset HEAD <nazwa_pliku>`

.

(...)

Zdalne repozytorium

To wszystko co robiliśmy odbywało się lokalnie, jedną z fajniejszych rzeczy jest możliwość korzystania z repozytorium przez sieć.

Aby sprawdzić jakie repozytoria zdalne są podłączone do naszego piszemy: `$ git remote`. Na początku nie powinniśmy mieć żadnych, dodawanie nowego odbywa się przy użyciu `$ git remote add <nazwa> <adres>`

Zazwyczaj zdalne repozytoria podają nam gotową komendę jaką powinniśmy się posłużyć aby zsynchronizować repozytorium online z lokalnymi.

Jeżeli zrobiliśmy commit’a i chcielibyśmy „pchnąć” zmiany na serwer to używamy komendy `$ git push -<nazwa_remote> <nazwa_branch>`.

Aby pobrać wszelkie zmiany do naszego lokalnego repozytorium to robimy to przez użycie `$ git fetch` a następnie `$ git pull`.

Wskazówki odnośnie korzystania z git’a

Korzystać z systemu kontroli wersji można na wiele sposobów, jednak chciałbym się podzielić z Wami kilkoma (dość oczywistymi) wskazówkami.

- do repozytorium dodajemy jedynie działające wersje projektu, tzn takie które się kompilują (pomaga to zachować pewny porządek, dzięki czemu nie trzeba się zastanawiać czy dana wersja jest działająca, czy nie);
- załóżmy, że mamy już zdalne repozytorium wtedy prace nad projektem wyglądają następująco:
    1. rozpoczynamy od pobrania aktualnej wersji repo (*fetch* i *pull*);
    2. jeżeli kończymy prace i mamy kompilujący się kod to aktualizujemy repo (*add*, *commit*, *push*);
    3. prace nad poszczególnymi segmentami robimy na oddzielnych branch’ach;
- nie wrzucamy do repozytorium zbędnych plików (np. osobistych ustawień pod kompilator, itp.).

Code ON!