

GIT Cheat Sheet

mgr inż. Maciej Małecki

maciej.malecki@pwr.edu.pl

27 lutego 2018

Instalacja narzędzi

Należy pobrać i zainstalować wersję klienta GIT odpowiednią dla stosowanego systemu operacyjnego: <https://git-scm.com/downloads>. Klient ten zawiera narzędzia dostępne z linii poleceń i jest wystarczający do pracy na zajęciach.

Zalecane narzędzie graficzne nazywa się GitExtensions i można je pobrać z następującej strony: <https://github.com/gitextensions/gitextensions>.

Konfiguracja dostępu SSH

Aby mieć dostęp do prywatnych repozytoriów GitHub, konieczna jest właściwa konfiguracja autentykacji. Zalecanym narzędziem jest OpenSSH (także w systemach Windows) oraz użycie pary kluczy RSA. OpenSSH dla systemu Windows jest dostarczane razem z klientem GIT. Instrukcja generowania kluczy dostępna jest na GitHub¹.

Wygenerowany klucz publiczny należy wgrać do ustawień swojego konta GitHub². W przypadku, gdy posiadamy tylko jedno konto GitHub, wystarczające będzie skopiowanie klucza prywatnego do katalogu `users/<user name>/.ssh` pod nazwą `id_rsa`.

Dostęp SSH dla GitExtensions

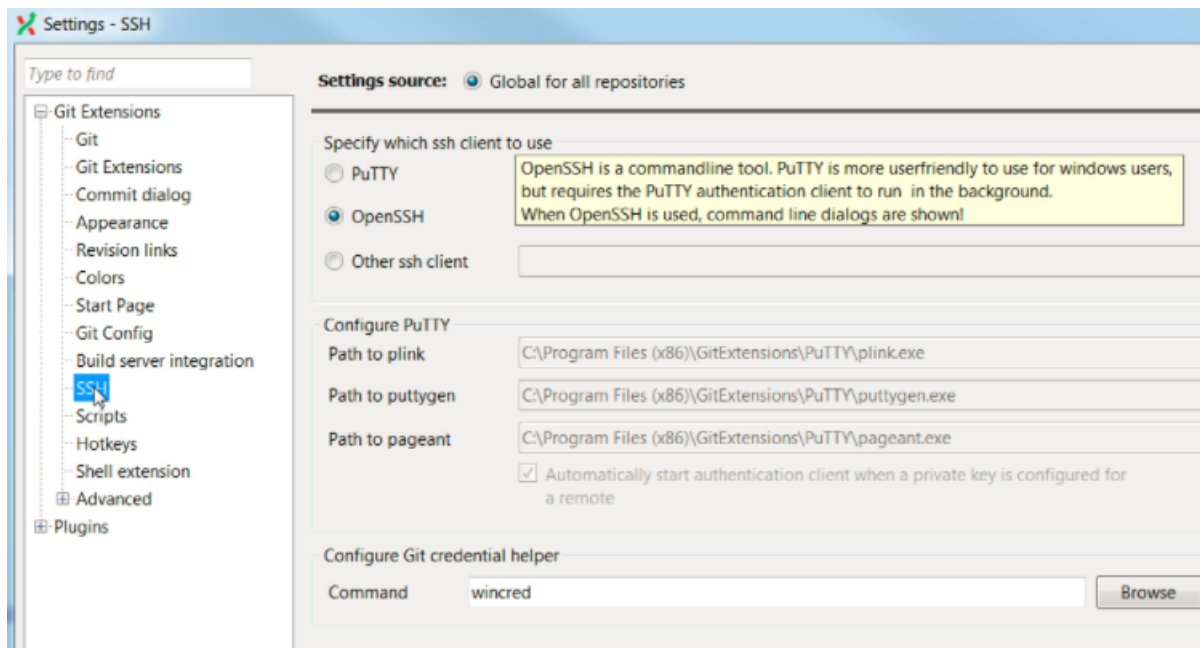
W przypadku klienta GitExtensions konieczne jest ustawienie OpenSSH jako metody autentykacji. Zmiany dokonujemy przy użyciu opcji menu `Settings/Git Extensions/SSH` (zobacz rys. 1). Wybór metody autentykacji możliwy jest także podczas instalacji narzędzia.

Konfiguracja SSH dla wielu kont GitHub

W przypadku, gdy posiadamy wiele kont w portalu GitHub, możliwy jest wybór klucza prywatnego (a więc i użytkownika) dokonany dla każdego repozytorium oddzielnie. Przy założeniu, że dysponujemy oddzielnymi parami kluczy RSA dla każdego z kont, konieczne jest utworzenie pliku `config` w katalogu `users/<user name>/.ssh` o następującej zawartości:

¹<https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/#generating-a-new-ssh-key>

²<https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/>



Rysunek 1: Konfiguracja SSH w kliencie GitExtensions.

```
Host priv
  HostName github.com
  User git
  IdentityFile ~/.ssh/id_rsa_priv
  IdentitiesOnly yes
Host pwr
  HostName github.com
  User git
  IdentityFile ~/.ssh/id_rsa_pwr
  IdentitiesOnly yes
```

UWAGA: dla każdego wpisu pole `User` zawsze musi mieć wartość `git`.

Powyższy plik należy rozbudować o kolejne wpisy w przypadku, gdy posiadamy więcej kont. Aby GIT (a także GitExtensions) użył dedykowanego klucza prywatnego do dostępu do repozytorium, należy repozytorium sklonować używając nazwy `Host` z pliku `config`, np.:

```
git clone pwr:pwr-piisw/oasp-seed
```

sklonuje repozytorium `oasp-seed` wykorzystując użytkownika autentykowanego kluczem `id_rsa_pwr`. Natomiast:

```
git clone priv:pwr-piisw/oasp-seed
```

dokona tego wykorzystując klucz `id_rsa_priv`.

Praca z GITem

Podstawowe komendy GITa przydatne w pracy nad projektem.

Sklonowanie repozytorium

```
git clone https://github.com/pwr-piisw/oaspseed
```

lub

```
git clone pwr:pwr-piisw/oaspseed
```

w przypadku, gdy stosujemy wiele kluczy RSA; **pwr** jest wówczas wyróżnikiem klucza SSH.

Konfiguracja repozytorium

Po sklonowaniu dobrze jest ustawić właściwie imię i nazwisko użytkownika. Po wejściu do sklonowanego repozytorium możemy sprawdzić aktualną konfigurację przy pomocy komendy:

```
git config -l
```

istotne są parametry `user.name` oraz `user.email`. Ich zmianę lokalną przeprowadzamy w następujący sposób:

```
git config --add user.name "Maciej Małecki"  
git config --add user.email maciej.malecki@pwr.edu.pl
```

Możliwa jest także zmiana globalna, działająca domyślnie dla wszystkich repozytoriów, najwygodniej w tym celu użyć narzędzia w trybie edycji:

```
git config --global -e
```

Pobieranie zdalnych zmian

Git jest narzędziem rozproszonym, lokalnie zawsze pracujemy na lokalnej kopii repozytorium. Aby zaktualizować jego zawartość, musimy użyć komendy `pull`, przy czym zawsze zalecane jest użycie trybu `rebase`:

```
git pull --rebase
```

Tryb `rebase` wgra wszystkie zmiany zdalne „pod” nasze zmiany, dzięki czemu zachowamy liniowość historii zmian (zwiększa to czytelność drzewa historii GITa). Operacja `pull` nie powiedzie się, jeśli mamy zmiany lokalne, które nie zostały dodane i zatwierdzone do historii.

Wprowadzanie zmian

Każde repozytorium GITa zawiera trzy obszary robocze:

1. Lokalny system plików
2. *Staging*
3. Repozytorium

Więcej szczegółów na <https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>.

Modyfikując pliki projektowe zawsze pracujemy na obszarze 1 (system plików). W każdej chwili możemy sprawdzić status zmian przy użyciu:

```
git status
```

Wybrane zmiany możemy dodać do obszaru 2 (*staging*). Szczególnie wygodne jest tutaj narzędzie graficzne GitExtensions. W przypadku linii poleceń stosujemy komendę add:

```
git add .
```

aby dodać wszystkie zmiany z systemu plików do *staging* lub

```
git add doc/*.txt
```

aby dodać wszystkie pliki o rozszerzeniu `txt` z katalogu `doc`. Zatwierdzanie zmian (czyli dodanie ich do obszaru 3 - repozytorium) możliwe jest z wykorzystaniem komendy `commit`:

```
git commit -m "Komentarz"
```

Bardzo istotne jest stosowanie opisowych komentarzy do każdej zmiany.

Przeglądanie zmian

Do przeglądania zmian w repozytorium szczególnie przydatne jest narzędzie graficzne GitExtensions. Możliwe jest także użycie linii komend (więcej szczegółów: <https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History>). W szczególności:

```
git log
```

wyświetla listę ostatnich zmian w repozytorium.

```
git log -p
```

wyświetla listę zmian wraz z wyszczególnieniem różnic w plikach.

```
git log --graph --pretty=format:"%h - %an, %ar: %s"
```

wyświetla listę zmian w formie kompaktowej oraz z wyszczególnieniem struktury drzewiastej.

Alias

Wpisywanie długich poleceń (jak np. `git log --graph --pretty=format:@"%h - %an, %ar: %s"`) jest dość żmudne. Konfiguracja GIT'a umożliwia definiowanie skrótów (ang. *aliases*). Na przykład:

```
git config --global alias.lp "log --graph
--pretty=format:@"%h - %an, %ar: %s\""
```

zdefiniuje skrót, który można następnie wywołać z linii poleceń w następujący sposób: `git lp`. W przypadku systemów Unixowych, treść aliasa należy ująć w apostrofy zamiast cudzysłowów.

Anulowanie i modyfikowanie zmian

Zmiany dokonane na lokalnym repozytorium mogą być w bezpieczny sposób anulowane lub zmodyfikowane. Jeśli zatwierdziliśmy zmianę, ale chcemy dodać coś jeszcze do tego commita, można to łatwo zrobić (dla ostatniego commita) przy użyciu:

```
git commit --amend
```

W podobny sposób można także zmienić komentarz do ostatniego commita:

```
git commit --amend -m "Nowy komentarz"
```

Aby usunąć wszystkie niezatwierdzone zmiany (z obszaru staging oraz z lokalnego systemu plików) można użyć:

```
git reset --hard
```

UWAGA: komenda ta trwale usuwa wszelkie lokalne i niezatwierdzone zmiany bez prośby o potwierdzenie! Więcej szczegółów: <https://git-scm.com/docs/git-reset>.

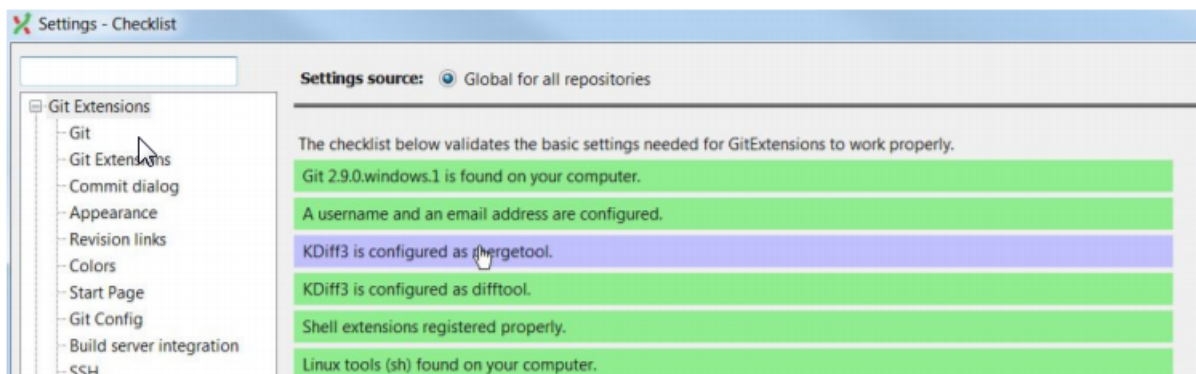
Przenoszenie zmian na zdalne repozytorium

Wszelkie zatwierdzone zmiany muszą być scalone z repozytorium zdalnym na GitHub tak aby inni członkowie zespołu mieli do nich dostęp. Pierwszym etapem zawsze jest rebase, gdyż w każdej chwili mogą na zdalnym repozytorium pojawić się nowe zmiany. Rebase wykonujemy znana nam już komendą pull:

```
git pull --rebase
```

Podczas tej operacji możliwy jest konflikt (czyli sytuacja, w której dwie osoby dokonały w tym samym czasie modyfikacji tego samego fragmentu kodu). Do rozwiązywania konfliktów najwygodniej użyć narzędzia GitExtensions z zainstalowanym pluginem KDiff3 (zobacz rysunek 2). Rozwiązywanie konfliktów w większości przypadków następuje automatycznie. Ingerencja potrzebna jest tylko tam, gdy narzędzie nie potrafi samo zdecydować jak powinien wyglądać kod po scaleniu (np modyfikacja dotyczyła tej samej linii kodu)³. Po rozwiązaniu wszelkich konfliktów zmiany należy jak najszybciej przenieść na zdalne repozytorium:

³Polecam lekturę: <https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>.



Rysunek 2: Konfiguracja narzędzia do scalania w kliencie GitExtensions.

```
git push
```

Oczywiście, w sytuacji, gdy w czasie rozwiązywania konfliktów na repozytorium zdalnym pojawiły się nowe zmiany, `git push` się nie powiedzie i proces rebase oraz rozwiązywania konfliktów musi być powtórzony.

Praca z gałęziami

Gałęzie (ang. *branches*) tworzą rozwidlenie drzewa wersji i pozwalają na równoległą pracę nad kodem przez wiele osób. Gałęzie używane są także w procesie wersjonowania i wydawania kodu aplikacji. Utworzenie gałęzi umożliwia komenda:

```
git branch <branch name>
```

gdzie **branch name** określa nazwę nowotworzonej gałęzi. Rozwidlenie zostanie utworzone w miejscu wskazywanym przez wskaźnik **HEAD**. Po utworzeniu możliwe jest przejście na utworzoną gałąź przy pomocy komendy: `git checkout <branch name>`. Odtąd zmiany wprowadzane będą bezpośrednio na wybranej gałęzi.

Istnieją zasadniczo dwie metody integracji dwóch gałęzi. W przypadku gdy chcemy przenieść zmiany z jednej gałęzi do drugiej (z reguły druga gałąź jest gałęzią, z której nastąpiło rozwidlenie) wykorzystujemy operację *merge*: `git merge <branch>` (zobacz rysunek 3).

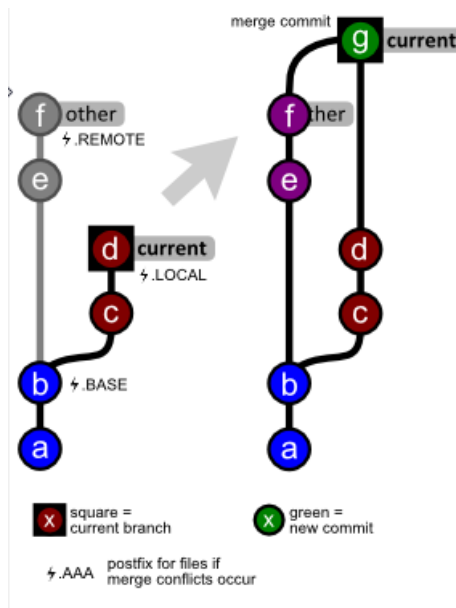
W sytuacji w której chcemy wprowadzić do gałęzi potomnej zmiany z gałęzi bazowej dokonane po momencie rozwidlenia, używamy operacji *rebase*: `git rebase <branch>` (zobacz rysunek 4). Operacja *rebase* powoduje nadpisanie historii i należy używać jej z rozwagą podczas pracy na gałęziach współdzielonych (wypushowywanie takiej gałęzi wymaga użycia opcji `--force`).

Tagowanie

W trakcie prac laboratoryjnych oraz oddawania poszczególnych etapów prosimy o tagowanie zmian, które podlegają ocenie. Tagi wymagają oddzielnego pushowania do zdalnego repozytorium. Tag jest etykietą posiadającą nazwę i jest przypisany do konkretnego commita w historii.

Poniższa komenda:

```
git tag lab03
```



Rysunek 3: Integracja zmian poprzez *merge*.

utworzy tag o nazwie `lab03` na najświeższym commicie. Alternatywnie możliwe jest tagowanie starszych commitów z przy pomocy następującej komendy:

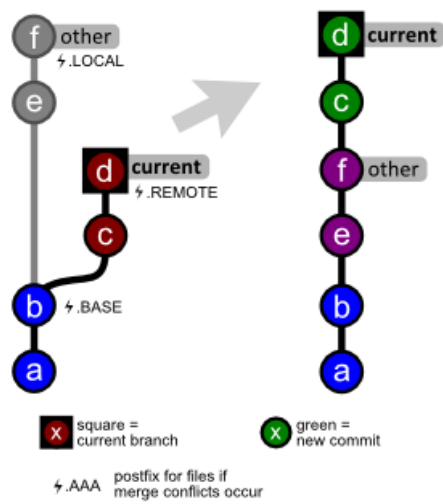
```
git tag lab03 <commit sha>
```

gdzie "commit sha" jest skrótem, który możemy odczytać np. przy użyciu komendy (jest to zawartość pierwszej kolumny - 7-mio cyfrowa liczba heksadecymalna):

```
git log --pretty=format:"%h - %an, %ar: %s"
```

Utworzony tag należy następnie wypushować do zdalnego repozytorium (nie dzieje się to automatycznie, nie robi tego też komenda `git push` bez dodatkowych parametrów):

```
git push origin lab03
```



Rysunek 4: Integracja zmian poprzez *rebase*.