# Pengantar Git

## Tentang *Version Control*

*Version control* adalah sebuah sistem yang mencatat beberapa perubahan sebuah *file* atau satu set *file*, sehingga kita bisa melihat versi-versi yang lebih spesifik pada waktu mendatang.

*Version control* atau sering disebut *Version Control System* (*VCS*) memungkinkan kita untuk melakukan *revert* sebuah *file* atau *project* yang sudah dipilih untuk dikembalikan ke tahap sebelumnya, membandingkan perbedaan dari setiap perubahan versinya, melihat siapa (*user*) yang mengubah isi *file* atau *project* tersebut, dan lain-lain.

## Sejarah Singkat Git

Git dibangun pada 2005 oleh Linus Torvalds—pencipta Linux—dengan beberapa tujuan:

1. *Speed*
2. *Simple design*
3. *Strong support for non-linear development (thousands of parallel branches)*
4. *Fully distributed*
5. *Able to handle large projects like Linux kernel efficiently (speed and data size)*

## Dasar-Dasar Git



Git *three states*

Proses kerja Git berjalan melalui *three states*; *committed*; *modified*; dan *staged*.

1. Saat *modified state*, *file* apa pun yang kita edit akan terdeteksi oleh Git, namun belum terbaca (*untracked file*).
2. Saat *staged state*, *untracked file* sudah terbaca oleh Git—lalu menjadi *tracked file*—dan siap untuk “disimpan” (di-*commit*) ke *database* (*local repository*) komputer kita.
3. Saat *committed state*, *file* atau data yang diproses Git telah berhasil disimpan di *local repository* komputer kita.

## Penggunaan *Command Line* untuk Git

Salah satu *tools* untuk mengoperasikan Git melalui komputer kita adalah *Command Line* (atau *Command Line Interface* [*CLI*]).

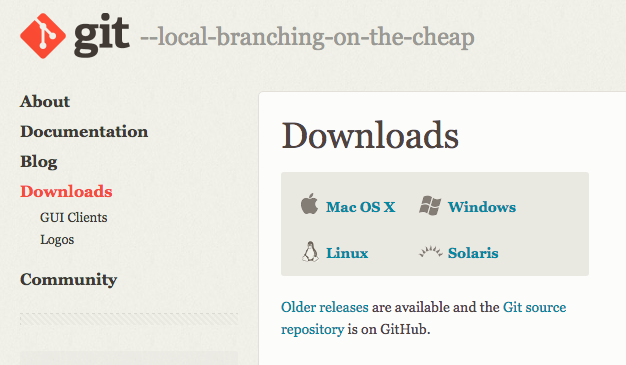
*Command Line* adalah program *text interface* yang terdapat di komputer kita untuk menjalankan perintah (*command*) yang diberikan *user* kepada *operating system* (*OS*).

Contoh program *CLI* yang dimiliki Mac OS adalah *Terminal* dan pada Windows adalah *Command Prompt* (*cmd*).

## Macintosh HD:Users:irfandyaditya:Documents:PhinCon:Guidebook:ss_terminal.pngContoh *Terminal* di Mac OS X.

## Instalasi Git

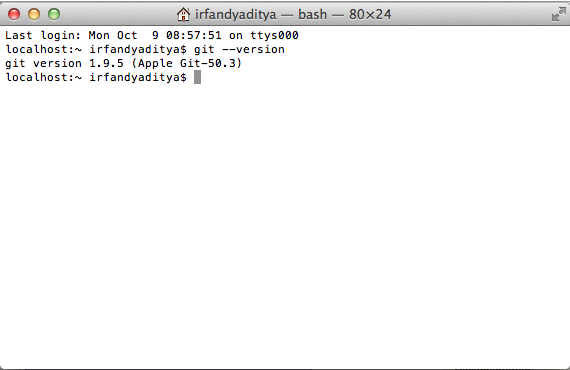
Sebelum menggunakan Git melalui *CLI*, kita perlu melakukan instalasi *package*-nya terlebih dahulu, yang bisa di-*download* secara gratis di situs resmi Git dengan tautan *https://git-scm.com/downloads* (tersedia untuk *OS* Windows, Mac OS X, Linux, dan Solaris).



Git *package* untuk beragam *OS*.

## Setelan Awal Git

Setelah instalasi *package* Git, kita bisa mengecek versinya dengan mengetik git --version di *CLI*. Jika *CLI* menampilkan versi, berarti instalasi *package* Git di komputer kita telah berhasil dan Git sudah siap untuk digunakan.



Pengecekan versi Git melalui *Terminal* di Mac OS X.

Langkah berikutnya adalah mengeset *user name* dan *email address* yang dibutuhkan Git setiap kali kita menyimpan perubahan *file* (*commit*). Berikut contoh input untuk mengesetnya:

git config --global user.name “Irfandy Aditya”

git config --global user.email “irfandyaditya17@gmail.com”

Lalu, kita bisa mengecek hasil input *user name* dan *email address* dengan mengetik git config --list.

Macintosh HD:Users:irfandyaditya:Documents:PhinCon:Guidebook:ss_check_identity.png

Nah, kini kita siap untuk menggunakan Git melalui *CLI*.

# Dasar-Dasar Git

## *Repository*

[Menurut Charles Duan](https://www.sbf5.com/~cduan/technical/git/git-1.shtml), direktur dari *Patent Reform Project* di [Public Knowledge](https://www.publicknowledge.org/), Git menyimpan *file* atau *project* kita ke dalam sebuah struktur data bernama *repository*.

Sebuah *repository* Git memiliki satu set *commit objects* dan satu set *references* untuk *commit objects* bernama *heads*.

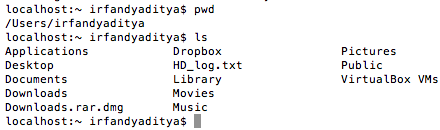
Perlu diperhatikan, *repository* Git disimpan di dalam *directory* yang sama dengan *file* atau *project* kita, di sebuah *subdirectory* bernama .git.

### *Initialization*

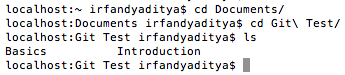
Jika kita memiliki *file* atau *project* yang belum dikontrol oleh Git, kita bisa mengeceknya di *directory* melalui *CLI*—contoh di bawah menggunakan *Terminal* di Mac OS X.

Untuk melihat posisi kita di *directory* saat ini, tik pwd.

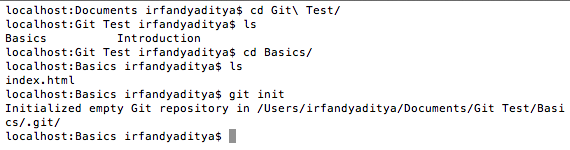
Lalu, tik ls untuk melihat *folder* yang terdapat di *directory*.



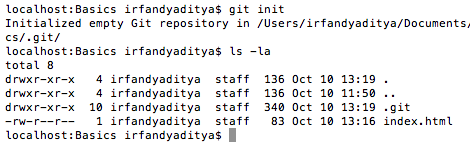
Pilih folder yang akan digunakan untuk menginisialisasi repository Git. Untuk berpindah *directory*—atau berpindah ke *folder* lain—tik cd <folder name>.



Selanjutnya, kita inisialisasi *repository* Git di *folder* yang akan dikontrol oleh Git dengan mengetik git init.



Pada tahap ini, Git sudah mengenali dan akan mengontrol setiap *commit* dalam *directory* tersebut. Untuk mengecek keberadaan *subdirectory* Git, tik ls -la.

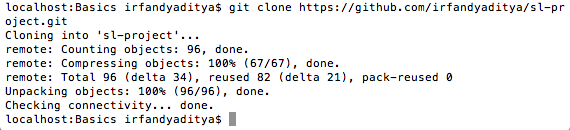


### *Cloning*

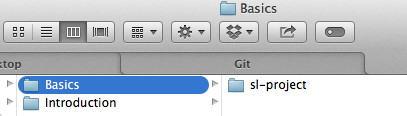
Git memungkinkan penggunanya untuk berkolaborasi dalam suatu *project*.

Jika kita baru akan bergabung ke *project* tersebut, maka perlu melakukan *clone repository* yang sudah tersedia, misalnya *repository* dari GitHub\*.

Caranya, dengan mengetik git clone <url GitHub repository> di *CLI*.



Pada tahap ini, *repository* dari GitHub berhasil di-clone ke komputer kita.



Setelah memiliki *local repository* Git dan *cloned repository* GitHub, kita bisa mengerjakan *project* dengan memanfaatkan sistem pengontrol versi Git.

\*)Penggunaan GitHub akan dijelaskan di dokumen terpisah.

## *Recording Changes* di *Repository*

Sebelum mempelajari tahap ini, kita perlu mengingat kembali *Dasar-Dasar Git*, bahwa setiap *file* yang terdapat pada *working directory* berada dalam keadaan *tracked* atau *untracked*.

Ketika pertama kali *cloning repository*, semua *file* kita terbaca *tracked* dalam kondisi *unmodified* karena Git baru saja mengeceknya dan kita belum mengedit apa pun. Jadi belum terdeteksi pembaruan versi *file*.



Git *status lifecycle*

Jika suatu *file* masih dinyatakan *untracked*, maka kita *add file* tersebut supaya menjadi *tracked* dan kemudian berlanjut ke *staged state*.

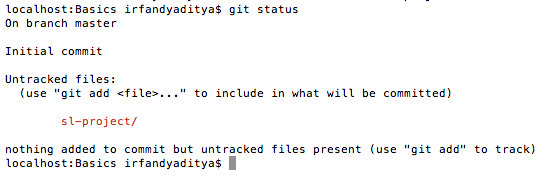
Pada *staged state*, *file* sudah bisa di-*commit* dan status setelah *committed* kembali menjadi *unmodified*.

Apabila kita mengeditnya lagi, status *file* akan berubah menjadi *modified*, yang berarti telah terjadi pembaruan versi *file*.

Jika ingin menyimpan pembaruan versinya, kita harus kembali *add file* supaya bisa diproses ke *staged state* dan kemudian di-*commit* ke *local repository* Git.

### Mengecek Status *File*

Setelah *cloning repository* sukses, kita bisa mengecek status *file* yang di-*clone* tersebut dengan mengetik git status di *CLI*.

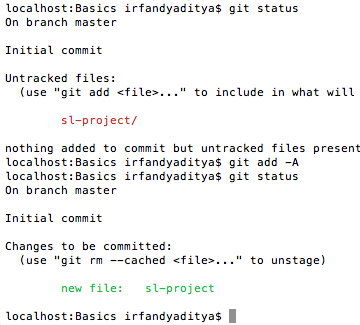


Tertera pada gambar, bahwa Git menemukan sebuah *untracked files* dan menganjurkan kita untuk *add file* bila ingin di-*commit*.

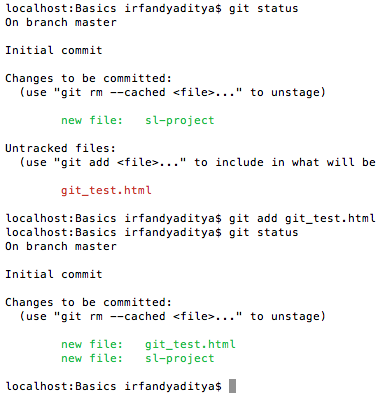
### *Tracking New File*

Jika sudah mengetahui status *file*, selanjutnya kita *add file* tersebut dengan mengetik

* git add -A untuk men-*track* seluruh *file* dalam *folder*, atau
* git add <files> jika hanya ingin men-*track* *file* tertentu.

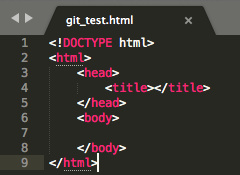
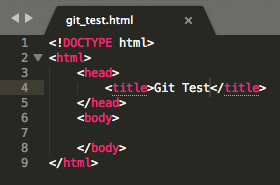


Simulasi berikut memperlihatkan terjadinya penambahan *new file* berupa *HTML* ke *folder* terkait yang akan terbaca oleh Git. Ingat selalu untuk mengecek status *file* sebelum dan sesudah penambahan *new file*.



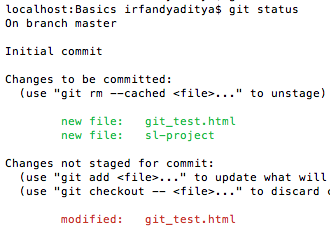
### *Staging Modified File*

Pada bagian ini, mari kita coba untuk mengedit sebuah *tracked file*, contohnya pada *file HTML* yang diedit menggunakan *Sublime Text*.

**

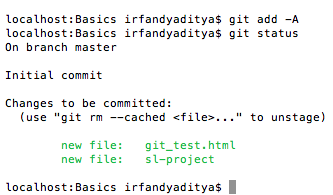
Sebelum diedit Sesudah diedit

Lalu, kita buka *CLI* dan cek status *file* yang baru saja kita edit.



Git mendeteksi *file HTML* telah diedit (*modified*) di *working directory*, tetapi belum diproses ke *staged state*.

Nah, untuk memprosesnya, tik git add -A dan cek status terbarunya dengan git status.



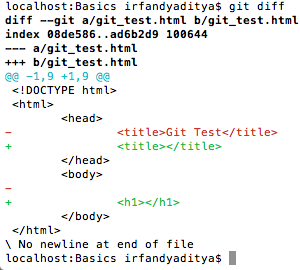
Setelah tahap ini, *file* dapat di-*commit* ke *local repository* kita.

### *Staged* *&* *Unstaged Changes*

Dalam Git, cara mengecek status *file* tidak hanya menggunakan *command* git status.

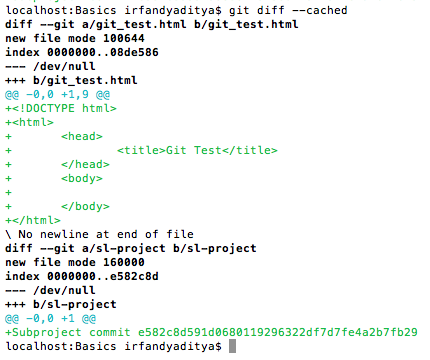
Git juga memiliki *command* git diff, yang akan menunjukkan perubahan mendetail di *file* yang diedit.

Sebagai contoh, *file HTML* yang diedit dengan mengurangi dan menambahkan beberapa teks.



Dengan *command* git diff, Git akan mendeteksi per baris di perubahan *file* berformat teks; warna merah bagian yang masih *unstaged*; warna hijau bagian yang sudah *staged*.

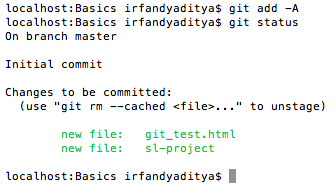
Sementara, untuk melihat bagian apa saja yang telah kita proses ke *staged state*, tik git diff --cached (atau juga bisa dengan *command* git diff --staged).



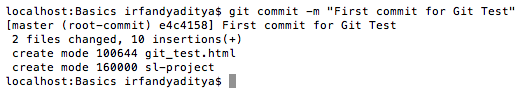
### *Committing Changes*

Setelah melewati *modified state* dan *staged state*, berikutnya kita proses *file* di *committed state* untuk disimpan ke *local repository* Git.

Pastikan file yang akan di-*commit* telah di-*track* dengan *command* git add - A dan dicek kembali dengan *command* git status.



Jika kondisi *file* sudah dinyatakan staged dan siap untuk di-*commit* seperti gambar di atas, selanjutnya kita akan melakukan *commit* untuk pertama kalinya dengan menyertakan *message*; tik di *CLI* git commit -m “insert message here”.



Jika *commit* sukses, *output CLI* akan menampilkan

* *branch* master tempat kita melakukan *commit*,
* fungsi *cryptographic hash* *checksum* (e4c4158),
* jumlah *file* yang diedit, dan
* jumlah baris teks yang ditambahkan atau dihapus.

Sampai tahap ini, kita telah berhasil melakukan *commit* ke *local repository* Git.

Siklus *three states* akan terus berulang selama menggunakan Git.

Perlu diingat bahwa setiap kali melakukan *commit* berarti kita menyimpan sebuah *snapshot* dari versi *file*, yang nantinya bisa di-*revert* atau di-*compare* dengan versi *file* setelahnya.

### *Skipping the Staging Area*

Git menyediakan sebuah “jalan pintas” untuk kita yang ingin meloncati *staging area*.

Cukup tambahkan *flag* -a ke *command* git commit supaya Git menjalankan *staged state* setiap *tracked file* secara otomatis tanpa harus *add file*.

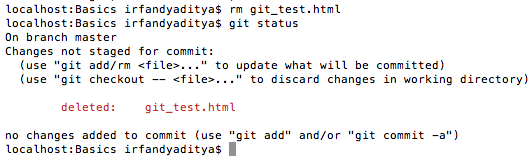
Macintosh HD:Users:irfandyaditya:Documents:PhinCon:Guidebook:Git:ss_git_commit_a_m_second-commit.png

Ketika menggunakan *command* ini, kita tidak perlu lagi mengetik git add karena *flag* -a sudah meliputi semua *file* yang diubah.

Namun, kita juga harus waspada ketika melakukan *commit* dengan *flag* -a karena bisa mencakupi perubahan yang tidak diinginkan saat *commit*.

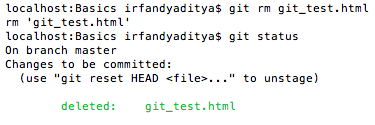
### Menghapus *File*

Untuk menghapus *file* dari Git, kita harus menghapusnya dari *staging area* terlebih dahulu dengan mengetik rm <file>—contoh berikut adalah *tracked file HTML*.



Berdasarkan gambar di atas, *file* sudah terhapus dari *working directory*, tetapi belum terhapus dari Git karena belum di-*commit*.

Oleh karena itu, tik git rm <file> untuk menghapusnya dari Git. Lalu cek statusnya dengan git status.



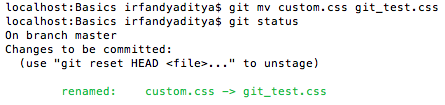
Pada *commit* berikutnya, *file HTML* tersebut akan hilang dan tidak akan terbaca (*tracked*) oleh Git.

Bagaimana jika kita terlanjur mengedit lalu memprosesnya ke *staging area*? Maka, kita harus *force removal file HTML* tersebut dengan menambahkan *flag* -f di *command* git rm.

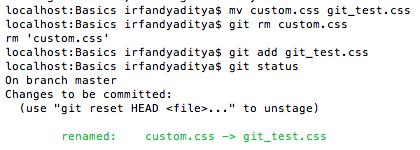
Ini adalah fitur keamanan yang bertujuan untuk mencegah ketidaksengajaan penghapusan data yang belum di-*commit*—belum tersimpan di *snapshot* dari versi *file*—dan data yang tidak bisa di-*recover* dari Git.

### Mengganti Nama *File*

Git memiliki *command* mv untuk mengganti nama *file*. Salah satu caranya dengan mengetik git mv file\_from file\_to.



Selain *command* mv di Git, kita juga bisa menggunakan cara berikut untuk mengganti nama *file*.



Kedua cara tersebut menampilkan *output* penggantian nama yang sama.

Namun, *command* mv di Git memiliki keunggulan karena menyederhanakan penggantian nama *file*—cukup sekali input.

## Melihat *Commit* *History* di Git

Setelah melakukan *commit* beberapa kali, kita bisa mengecek apa saja yang telah diproses dengan mengetik git log di *CLI*.



Secara *default* dan tanpa *arguments*, *command* git log mengurutkan daftar *commit* dimulai dari yang terakhir hingga awal (*reverse chronological order*). Berdasarkan gambar di atas, *command* ini mengurutkan setiap commit dengan

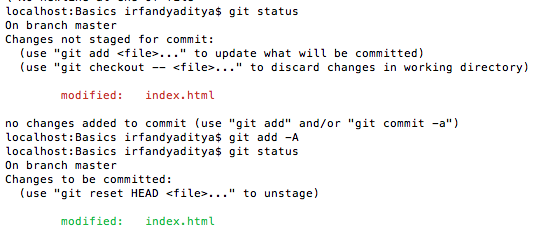
* *cryptographic hash*, yakni *Secure Hash Algorithm 1* (*SHA-1*) *checksum*,
* *author name* & *email* yang melakukan *commit*,
* tanggal beserta waktu *commit*, dan
* *commit message*.

Untuk melihat detail perubahan setiap commit dengan singkat, kita bisa menambahkan opsi --stat setelah *command* git log. 

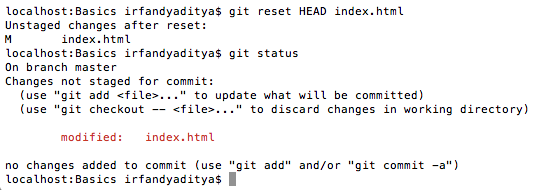
## *Undoing Things*

### *Unstaging* dari *Staged File*

Pada suatu keadaan, kita bisa saja tak sengaja memproses file yang (sebetulnya) belum siap untuk diproses ke *staged state*, seperti contoh di bawah.



Tidak perlu khawatir, kita bisa melakukan *unstaging* sebuah *file* yang terlanjur terproses ke *staged state* dengan mengetik *command* git reset HEAD <file>.



## Bekerja Secara *Remote*

### Menunjukkan *Remote*

Pada bagian 2.1.2., kita telah membahas cara *cloning* sebuah *remote repository* dari GitHub ke *local repository*. Nah, sekarang kita akan memanfaatkan *remote repository*.

Sebelumnya, kita perlu mengetahui konfigurasi yang digunakan *remote repository* itu, caranya dengan mengetik git remote –v.

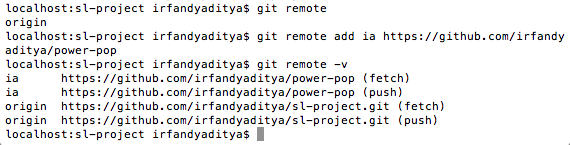
Macintosh HD:Users:irfandyaditya:Documents:PhinCon:Guidebook:Git:ss_git_remote_v.png

*CLI* akan menampilkan hasil command berupa kata origin—adalah nama *default* yang diberikan Git untuk *server* tempat kita melakukan *cloning*—dan *URL* dari *remote repository* yang kita *clone*.

### Menambahkan *Remote Repositories*

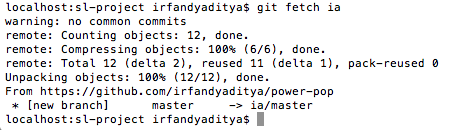
Setelah mengetahui konfigurasi *remote repository* yang akan kita gunakan, kita bisa menambah *remote repository* baru dengan nama pendek sesuai keinginan—bukan nama *default* origin.

Berikut cara menambah *remote repository* baru, tik *command* git remote add <shortname> <url GitHub repository>.



Sekarang, kita bisa menggunakan string “ia”—atau *shortname* apa pun yang kita tentukan—di *CLI* sebagai pengganti *URL repository* GitHub.

Sebagai contoh, jika kita ingin mengambil (*fetch*) semua informasi yang dimiliki *user* irfandyaditya$, tetapi belum kita miliki di *repository* kita, tik git fetch <shortname>.



Setelahnya, *master branch* *user* irfandyaditya$ bisa diakses secara *local* sebagai ia/master—nantinya dapat kita *merge*\* menjadi salah satu *branch* yang kita miliki, atau kita dapat mengecek *local branch* jika kita ingin menginspeksinya.

\*) Pembahasan mendetail mengenai *merge* dan *branching* akan dijelaskan pada subbab berikutnya.

### *Fetching* dan *Pulling* dari *Remote*

Seperti pada bagian sebelumnya, *command* git fetch berfungsi untuk mengambil atau menarik data dari *remote repository* ke *local repository*.

Jika kita melakukan clone sebuah *file* atau *project*, *command* akan membuat *remote repository* dengan nama *default* origin secara otomatis.

Jadi, melalui *command* git fetch origin, kita bisa mengambil semua pekerjaan baru yang telah di-*push* ke *server* yang *repository*-nya telah kita *clone*.

Namun perlu dicatat, bahwa *command* git fetch hanya mengunduh (*download*) data ke *local repository* kita—dan secara otomatis tidak melakukan *merge* dengan *file* atau *project* yang sedang kita kerjakan.

Sedikit beralih ke *branching*. Jika *branch* kita saat ini sudah diset untuk men-*track* sebuah *remote branch*, kita bisa menggunakan *command* git pull untuk *fetch* data secara otomatis dan kemudian melakukan *merge* dari *remote branch* ke *local branch* yang kita miliki.

Mungkin cara ini lebih mudah dan nyaman untuk kita gunakan; dan secara default, *command* git clone mengeset *local branch* untuk men-*track* *remote branch* dari *server* yang *repository*-nya telah kita *clone*.

Umumnya, *command* git pull melakukan *fetch* data dari *server* yang *repository*-nya telah kita *clone* dan secara otomatis mencoba *merge* data tersebut ke dalam *file* berisi *code* yang sedang kita kerjakan.

### *Pushing* ke *Remote*

Jika kita memiliki *file* atau *project* yang ingin di-*share* ke tim *project* atau ke khalayak umum, kita harus melakukan *push* dengan *command* git push <remote> <branch>.

Misalnya, ketika kita ingin melakukan *push* *master branch* ke *server* origin, bisa tik *command* git push origin master untuk *push* *commit* apa pun—yang kita lakukan secara lokal—ke *server*.

Perlu dicatat lagi, *command* push hanya berfungsi ketika kita melakukan *clone* dari *server* yang telah kita akses dan saat tidak ada tim *project* lain yang sedang melakukan *push* pada waktu yang sama—karena jika kita dan rekan tim lainnya berbarengan melakukan *clone* dan *push*, tetapi mereka sudah lebih dahulu melakukannya, maka *command* push kita akan ditolak.

Jadi, kita harus melakukan *fetch* *file* atau *project* tim kita dahulu dan menggabungkannya dengan dokumen kita sebelum dipersilakan melakukan *push*.

### Menginspeksi *Remote*

Jika kita ingin melihat informasi seputar *remote repository* tertentu, tik *command* git remote show <remote shortname>.



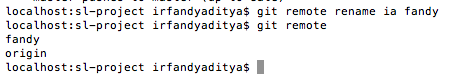
*CLI* menampilkan daftar *URL* *remote repository* dan juga informasi mengenai *tracking branch*.

Tertera pula pada *CLI*, bahwa ketika kita berada di *local master branch* dan menjalankan *command* git pull, secara otomatis akan melakukan *merge* di *remote master branch* setelah melakukan *fetch* semua referensi *remote*—dan juga menampilkan daftar semua referensi *remote* yang telah di-*pull*.

### Mengganti Nama dan Menghapus *Remote*

Untuk mengganti nama *remote*,

tik git remote rename <name before> <name after>.



Jika nama *remote* diubah, berarti nama *remote-tracking branch* juga berubah.

Seperti contoh sebelumnya, apa pun yang direferensikan di *master branch* ia/master, sekarang berada di fandy/master.

Untuk menghapus *remote*,

tik git remote remove <remote name>.

Macintosh HD:Users:irfandyaditya:Desktop:ss_git_remote_remove_remote-name.png

Jika kita menghapus *remote repository* karena suatu alasan, maka semua *remote-tracking branches* dan pengaturan konfigurasi yang berhubungan dengan *remote repository* tersebut juga akan dihapus.

# Fitur *Branching* di Git

## Penjelasan Singkat *Branches*

### Membuat *New Branch*

Dengan *new branch*, kita bisa mengeksplorasi hal-hal yang berkaitan dengan *file* atau *project* yang sedang dikerjakan.

Misalnya, kita akan membuat *new branch* bernama copy, tik *command* git branch <branch name>.

Ini akan membuat *branch* copy menunjuk ke *commit* yang sama dengan *branch* master.

Untuk mengecek keduanya menunjuk ke *commit* yang sama, tik *command* git branch -v.

Macintosh HD:Users:irfandyaditya:Documents:PhinCon:Guidebook:Git:ss_git_branch_v.png

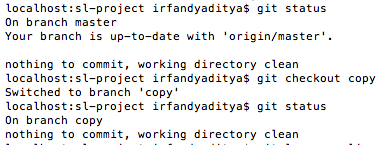
Git mampu mengetahui posisi *local branch* kita saat ini berkat sebuah *special pointer* bernama HEAD.

Sebagai tambahan, cara mengetahui *branch* yang sedang ditunjuk oleh *special pointer* HEAD bisa melalui *command* git log dengan opsi --decorate.

Namun, *command* git branch hanya sekadar membuat *new branch* dan tidak menukarnya (*switch*) ke *new branch* tersebut.

### *Switching Branches*

Masih melanjutkan topik sebelumnya, penukaran (*switch*) *branch* master ke *branch* copy bisa dilakukan dengan mengetik *command* git checkout <branch name>.



Setelah menjalankan command tersebut, *special pointer* HEAD akan berpindah ke *branch* copy.

Lalu, kita *switch* kembali ke *branch* master untuk melanjutkan tutorial.

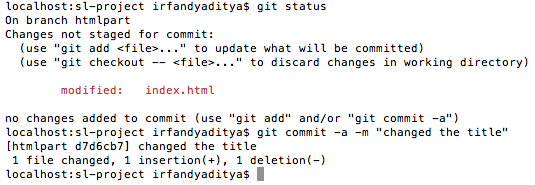
## Dasar *Branching* dan *Merging*

### Dasar *Branching*

Misalkan, kita sedang mengerjakan *project* dan telah melakukan beberapa *commit* di *branch* master. Akhirnya, kita memutuskan untuk memperbaiki bagian *file* *HTML* di *project* kita. Untuk membuat *new branch* sekaligus melakukan *switch* ke *new branch* itu, tik *command* git checkout -b <branch name>.

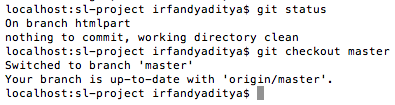
Macintosh HD:Users:irfandyaditya:Documents:PhinCon:Guidebook:Git:ss_git_checkout_b_branch-name.png

Lalu, jika bagian *file* *HTML* telah selesai diperbaiki, seperti biasa kita *stage* dan *commit* *file* tersebut (sekaligus) dengan *command* git commit -a -m “insert message here”.



Git memudahkan penggunanya supaya tidak perlu lagi men-*deploy* perbaikan yang kita kerjakan bersamaan dengan perubahan yang kita lakukan di *branch* htmlpart. Jadi, kita hanya perlu switch kembali ke *branch* master untuk mengeksekusinya (tik *command* git checkout master).

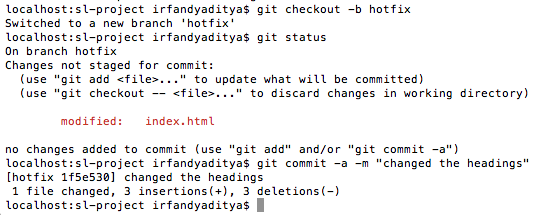
Namun sebelumnya, pastikan sudah tidak ada perubahan yang belum di-*stage* dan di-*commit* di *staging area* atau *working directory* kita—karena Git tidak akan melakukan *switch branch* jika *working state* kita belum dinyatakan *clean*.



Sampai tahap ini, *working directory* *project* kita sudah kembali seperti semula—sebelum melakukan perbaikan *file* *HTML*—dan kita bisa berkonsentrasi ke *hotfix*.

Ini penting untuk kita ingat: ketika kita *switch branch*, Git akan me-*reset* *working directory* kita menjadi seperti saat terakhir kali melakukan *commit* di *branch* tersebut.

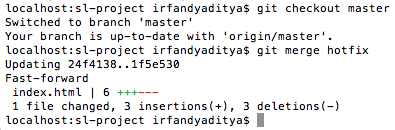
Selanjutnya, kita harus melakukan *hotfix*, mari kita buat *branch* bernama hotfix sebagai tempat perbaikan hingga tuntas.



Berdasarkan gambar di atas, *hotfix* telah selesai diperbaiki, di-*stage*, dan berhasil di-*commit*, maka berikutnya kita akan melakukan *merge* *branch* hotfix dengan *branch* master.

### Dasar *Merging*

Untuk mengujicobanya, pastikan *hotfix* telah sesuai kebutuhan—jangan lupa untuk *checkout branch*—dan kita bisa melakukan *merge* *branch* hotfix dengan *branch* master untuk men-*deploy* *project* kita. Gunakan *command* git merge.



Karena kita tidak lagi membutuhkan *branch* hotfix, maka kita bisa menghapusnya dengan menambahkan opsi -ddi *command* git branch.

Macintosh HD:Users:irfandyaditya:Documents:PhinCon:Guidebook:Git:ss_git_branch_d_branch-name.png

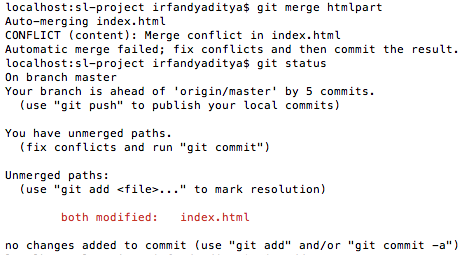
Setelah itu, kita akan kembali bekerja di *branch* htmlpart.

### Dasar *Merge Conflicts*

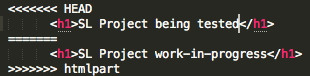
Suatu saat, kita bisa saja lupa sedang mengedit *file* yang sama pada dua atau lebih *branch* yang berbeda.

Hal ini bisa terdeteksi oleh Git ketika kita menjalankan *command* git merge.

Misalnya seperti contoh di bawah, *branch* htmlpart terlibat konflik dengan *branch* master.

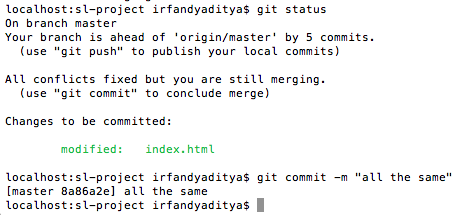


Sekarang, kita cek secara manual di *text editor*—pada tutorial ini menggunakan *Sublime Text*—pada bagian manakah *file HTML* tersebut terlibat konflik satu sama lain.

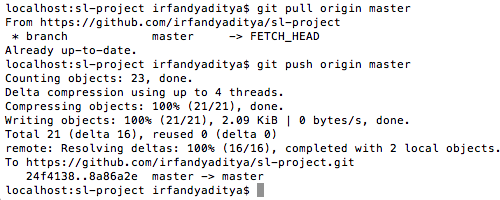


Tertera pada gambar, simbol <<<<<<< dan teks HEAD—yang berarti *branch* master. Lalu, simbol ======= memisahkan dua *heading*. Sementara, di bagian bawah terdapat simbol >>>>>>> dan teks htmlpart—yang berarti *branch* htmlpart terletak di bagian bawah.

Nah, kedua bagian tersebut harus kita sesuaikan dengan kebutuhan yang sebenarnya. Jika sudah yakin, *save file* tersebut dan kembali ke *CLI* untuk melakukan *stage* dan *commit*.



Jika ingin memperbarui semua *file* dan *commit* dari *local repository* ke *remote repository*—dalam tutorial ini menggunakan GitHub, kita lakukan *pull* terlebih dahulu sebelum *push* semua datanya.



Sejauh ini, kita telah mempelajari fitur dasar Git. Selanjutnya, kita akan mempelajari dasar-dasar penggunaan GitHub. Terima kasih.

# Referensi

Book. (n.d.). Diakses 26 September 2017, dari https://git-scm.com/book/en/v2