

Mengenai Git

Anggie Bratadinata | mas_ab@masputih.com

2012

| | |
|--|----|
| <i>Mengenal Git</i> | 2 |
| Bab 1. Introduksi | 6 |
| 1.1 Version Control System..... | 6 |
| 1.1.1 Mengapa Perlu Version Control? | 6 |
| Bring Order to Chaos..... | 6 |
| Short-term & Long-term Undo..... | 7 |
| Tracking Changes..... | 8 |
| Sandboxing..... | 8 |
| Kolaborasi..... | 8 |
| 1.1.2 Centralized VCS | 8 |
| 1.1.3 Distributed VCS | 9 |
| 1.1.4 Memilih VCS..... | 9 |
| 1.2 Git | 10 |
| 1.2.1 Data Model | 11 |
| 1.2.2 Istilah Penting..... | 13 |
| Bab 2. Instalasi Git | 15 |
| 2.1 Windows | 15 |
| 2.1.1 Git Bash | 15 |
| 2.2 Mac OSX | 16 |
| 2.3 Ubuntu/Mint..... | 17 |
| 2.4 Built-in Git GUI | 18 |
| 2.5 Konfigurasi..... | 18 |
| 2.5.1 User..... | 18 |
| Global | 18 |
| Lokal (Per Repositori) | 19 |
| 2.5.2 Default Message Editor | 19 |
| 2.5.3 Diff & Merge Utilities | 20 |
| Bab 3. Perintah-perintah Dasar | 22 |
| 3.1 Membuat Repositori | 22 |
| 3.2 Commit | 23 |

| | |
|--|----|
| 3.2.1 Staging Index | 23 |
| Membatalkan Staging (Soft Reset) | 25 |
| 3.2.2 Membatalkan Commit..... | 27 |
| 3.2.3 log | 27 |
| 3.3 Menangani File | 29 |
| 3.3.1 Membandingkan Isi File (diff)..... | 29 |
| Unstaged files..... | 30 |
| Staged files..... | 30 |
| Committed files..... | 31 |
| Membandingkan Isi Semua File..... | 31 |
| 3.3.1 Membatalkan Modifikasi (Hard Reset)..... | 33 |
| Hard Reset File..... | 33 |
| Hard Reset Working copy..... | 34 |
| 3.3.2 Menghapus File | 35 |
| 3.4 Berpindah Versi..... | 36 |
| 3.5 Branching & Merging..... | 37 |
| 3.5.1 Membuat Branch | 37 |
| 3.5.2 Berpindah Branch..... | 39 |
| 3.5.3 Upstream Branch | 41 |
| 3.5.4 Merging | 41 |
| Merge Conflict | 42 |
| Menggunakan Diff Utility untuk Menyelesaikan Konflik | 44 |
| 3.6 Menangani Interupsi dengan Stash..... | 45 |
| 3.7 Tagging | 47 |
| 3.8 Contoh Kasus : Website PT Maju Jaya | 48 |
| 3.8.1 Persiapan | 48 |
| 3.8.2 Home Page | 49 |
| 3.8.3 Profile Page..... | 50 |
| 3.8.4 Portfolio | 52 |

| | |
|--|----|
| 3.8.5 Interupsi : Profile Tidak Memiliki Heading | 52 |
| 3.8.6 Kembali ke Portfolio | 54 |
| 3.8.7 Release v1 | 55 |
| 3.8.8 Kesimpulan..... | 55 |
| Bab 4. Shared Repository..... | 57 |
| 4.1 Bare Repository | 57 |
| 4.2 Repositori Lokal | 58 |
| 4.2.1 Origin..... | 59 |
| 4.3 Remote Branch | 60 |
| 4.3.1 Tracking dan Non-tracking Branch | 61 |
| 4.4 Sinkronisasi..... | 62 |
| 4.4.1 Push dan Pull | 62 |
| 4.4.2 Non Fast-forward dan Merging Conflict | 63 |
| 4.5 Kesimpulan | 65 |
| Bab 5. Rebase..... | 67 |
| Bab 6. Git Hosting | 70 |
| 6.1 SSH Key | 70 |
| 6.1.1 Windows..... | 70 |
| 6.1.2 Linux & OSX..... | 71 |
| 6.2 Bitbucket | 72 |
| 6.2.1 Registrasi SSH Key | 72 |
| 6.2.2 Membuat Repositori | 74 |
| 6.2.3 Berbagi-pakai Repositori | 74 |
| 6.3 Github | 75 |
| 6.3.1 Registrasi SSH Key | 76 |
| 6.3.2 Membuat Repositori | 77 |
| 6.3.3 Berbagi-pakai Repositori | 78 |
| Bab 7. Penutup..... | 80 |
| 7.1 Referensi | 80 |

BAB 1. INTRODUKSI

Git adalah nama sebuah *version control system* (VCS) atau *source control management* (SCM) yang dibuat oleh Linus Torvalds (orang yang juga membuat Linux) dan pertama kali digunakan sekitar tahun 2005 untuk pengembangan *linux kernel*.

Dalam bab ini, kita akan mengenal VCS secara umum serta Git dan bagaimana cara kerjanya.

1.1 Version Control System

Version Control atau *Revision Control* adalah sebuah sistem yang merekam perubahan (revisi) yang terjadi dari waktu ke waktu pada sebuah file atau sekumpulan file sehingga kita bisa kembali ke revisi tertentu dengan mudah -- mirip *backup system* misalnya *Restore Point* di Windows Vista/7 dan *Time Machine* di Mac OSX. Sebuah database di mana revisi-revisi tersebut disimpan disebut repositori.

VCS tidak hanya berguna bagi para programmer tetapi juga bagi semua orang yang bekerja dengan file. Jika Anda setiap hari berurusan dengan dokumen-dokumen penting, Anda bisa memanfaatkan VCS untuk menyimpan dokumen dalam beberapa versi atau untuk mengerjakan satu dokumen bersama orang lain. Begitu pula jika Anda seorang *graphic artist*, Anda bisa memiliki beberapa versi dari desain yang sama.

1.1.1 Mengapa Perlu Version Control?

Bring Order to Chaos

Menggunakan VCS untuk menyimpan revisi (versioning) lebih baik dan lebih mudah daripada cara "kreatif" yang mungkin sering Anda lakukan dengan memanfaatkan fitur "**Save as**" dari editor favorit Anda, misalnya menamai file dengan nama `logo_v1.psd`, `logo_v2.psd`, dan sebagainya. Mungkin Anda biasa menambahkan tanggal revisi seperti `logo_v1_02Jan-2012.psd`, `logo_v2_01Maret2012.psd`, dan seterusnya. Atau mungkin Anda perlu memberi keterangan tambahan, misalnya `logo_v3_approved.psd`, `logo_v2_bg_coklat.psd`, `logo-v2_bg_biru.psd`, dan sebagainya.

Jika Anda hanya mengerjakan sedikit dokumen yang tidak pernah atau jarang diubah, tentu ini bukan masalah. Bayangkan jika kita harus mengerjakan banyak dokumen yang saling terkait satu sama lain, misalnya *source code* sebuah website atau mungkin selusin file Microsoft Word yang akan kita susun menjadi sebuah buku, tentunya menyimpan beberapa versi secara manual seperti ini sangat tidak efisien dan membuat direktori kerja kita dikotori oleh file yang sebagian besar mungkin sudah tidak kita perlukan lagi.



Gambar 1-1 Direktori Kerja tanpa VCS

VCS dapat membantu kita mengatasi masalah tersebut dengan membuat "backup" secara otomatis setiap kali kita melakukan *check-in* dan menyembunyikan backup tersebut sampai kita membutuhkannya. Hasilnya adalah sebuah direktori kerja yang bersih dan tidak membingungkan.



Gambar 1-2 Direktori Kerja dengan VCS

Short-term & Long-term Undo

Kita tahu bahwa *undo-level* setiap editor seperti Photoshop, Microsoft Word dan lain-lain, ada batasnya dan proses *undo* hanya bisa dilakukan terhadap satu file, bukan banyak file sekaligus. VCS memungkinkan kita melakukan undo per file dan juga per direktori. Bahkan kalau kita mau, kita bisa melakukan undo terhadap semua file di dalam direktori kerja hanya dengan satu perintah.

Ingin membatalkan semua update sejak *check-in* terakhir atau kembali ke revisi yang kita

lakukan tiga bulan yang lalu? Semua bisa kita lakukan dengan mudah. Di sini VCS berfungsi seperti mesin waktu yang memungkinkan kita maju atau mundur ke titik manapun di dalam sejarah file atau direktori kerja, kapanpun kita mau.

Tracking Changes

Setiap kali kita melakukan *check-in*, kita bisa menambahkan keterangan misalnya perubahan apa saja yang kita lakukan dan mengapa. Keterangan ini nantinya akan ditampilkan dalam bentuk log sehingga kita bisa melacak apa saja yang telah kita lakukan dan file apa yang kita ubah.

Ingin tahu apa perbedaan antara `index.html` terbaru dengan `index.html` setahun yang lalu? Mudah. Kita bahkan tidak perlu berpindah ke revisi setahun yang lalu. Cukup kita lakukan satu atau dua perintah.

Sandboxing

Dalam membuat *software* atau website, revisi terhadap *source code* adalah rutinitas. Tentu kita ingin memastikan bahwa perubahan yang kita lakukan sudah benar dan tidak menimbulkan masalah baru. VCS memungkinkan kita melakukan eksperimen dan pengujian tanpa harus melakukan perubahan permanen terhadap *source code* di dalam sebuah lingkungan terisolasi (*sandbox*). Jika tidak ada masalah, kita bisa lakukan *check-in* untuk menyimpan revisi. Sebaliknya, kalau ternyata ada kesalahan, kita bisa membatalkan semua update dan mengembalikan semua file yang kita modifikasi ke kondisi saat *check-in* terakhir.

Kolaborasi

Kita akan merasakan manfaat VCS terbesar pada saat kita berkolaborasi dengan orang lain. Kita dan rekan kita bisa melakukan *update* terhadap *source code* secara bersamaan tanpa khawatir saling mengganggu. Pada saatnya nanti, kita lakukan sinkronisasi dan penggabungan semua *update* yang kita dan rekan kita lakukan.

1.1.2 Centralized VCS

Centralized VCS (CVCS) seperti Subversion, ClearCase, CVS dan lain-lain, menggunakan satu repositori untuk menyimpan semua revisi file. Semua orang yang memiliki hak akses bekerja dengan cara *check-out* & *check-in*. Pada waktu kita melakukan *check-out* sebuah file di repositori, kita mengambil perubahan terakhir dari file tersebut dan mengaplikasikannya pada file yang sama di direktori kerja kita. Sebaliknya, proses *check-in* mengirim perbedaan (*diff*) antara versi terbaru dari file tersebut di direktori kerja kita dengan versi sebelumnya ke repositori.

CVCS juga memiliki fitur lock yang mencegah sebuah file dimodifikasi oleh lebih dari satu orang pada saat bersamaan. Fitur ini berguna untuk meminimalkan konflik tetapi juga memiliki

resiko yaitu kalau kita lupa membuka lock setelah kita selesai bekerja, orang lain tidak bisa check-in file tersebut sampai kita *unlock*.

Kelemahan utama dari sistem ini adalah sifatnya yang terpusat menjadikan repositori yang umumnya diletakkan di sebuah server sebagai *single point of failure*. Jika server *down* atau kita mengalami gangguan koneksi, kita tidak bisa meneruskan pekerjaan. Lebih parah lagi kalau *hard disk* server rusak, semua histori hilang.

1.1.3 Distributed VCS

DVCS dibuat sebagai alternatif yang lebih bisa diandalkan daripada CVCS. Masing-masing orang yang bekerja memiliki *full copy* dari repositori sehingga jika salah satu repositori rusak, recovery bisa dilakukan dengan mudah, cukup dengan mengkopi repositori yang lain. DVCS memungkinkan kita mengubah file, menambah atau menghapus file dan merekam perubahan-perubahan tersebut ke dalam repositori lokal tanpa harus terkoneksi dengan server atau repositori lain.

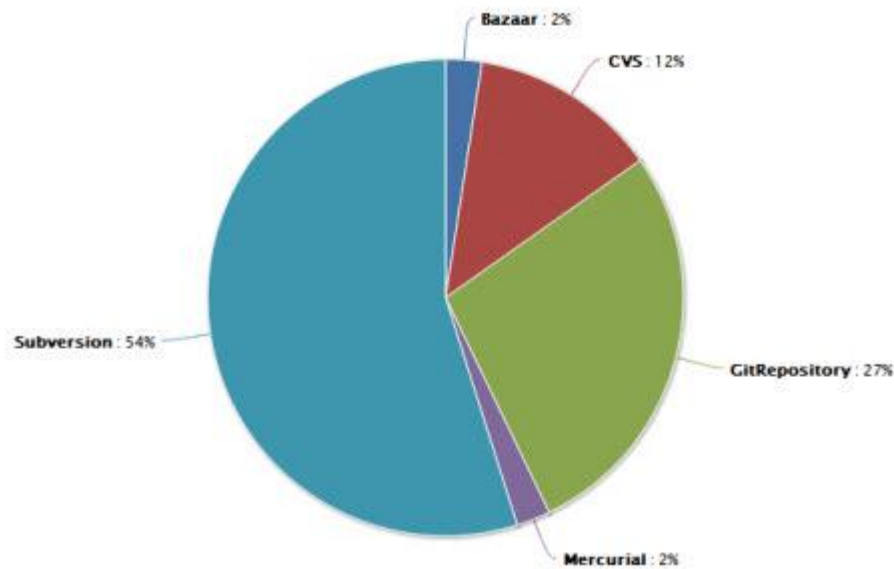
Kelemahan DVCS adalah kemungkinan terjadinya konflik antar repositori lebih besar daripada CVCS karena DVCS tidak memiliki fitur *lock*. Konflik terjadi pada saat *merging* (penggabungan) *source code* di mana dua versi file memiliki perbedaan isi. Banyak orang menyebutnya dengan istilah "**Merging Hell**". Hal ini bisa diminimalisir dengan komunikasi dan arsitektur software yang baik.

1.1.4 Memilih VCS

Dengan sedemikian banyak VCS, bagaimana kita memilih VCS yang tepat untuk proyek yang akan kita kerjakan? Kita bisa menggunakan VCS manapun disesuaikan dengan pengalaman kita atau rekan dalam tim. Semua VCS memiliki kelebihan dan kekurangan masing-masing namun pada intinya, semuanya bertujuan membantu kita bekerja lebih efisien. Berikut ini beberapa faktor yang bisa kita pertimbangkan dalam memilih VCS.

- Popularitas. Semakin populer sebuah VCS, semakin banyak tutorial dan artikel yang bisa kita temukan di Internet sehingga mudah untuk mencari solusi apabila kita menemukan masalah.
- Kemudahan atau *ease of use*.
- *Development activity*. Tentu tidak masuk akal jika kita menggunakan VCS yang sudah ditinggalkan oleh pengembangnya.

Berikut ini data popularitas VCS yang dikumpulkan oleh ohloh.com, sebuah *crawler* yang mengumpulkan data proyek-proyek *opensource*.



Gambar 1-3 Popularitas VCS (sumber : ohloh.com)

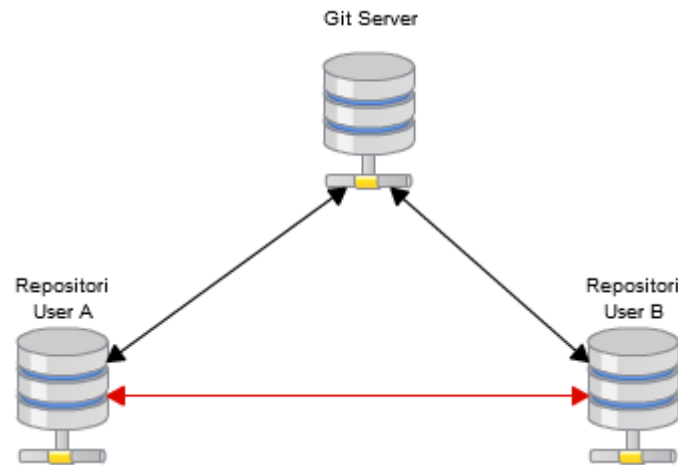
Dari gambar di atas, kita lihat Subversion masih menjadi VCS yang paling banyak digunakan. Salah satu alasannya adalah karena usianya yang lebih tua daripada Git sehingga banyak *legacy project* yang masih menggunakannya. Walaupun begitu, banyak developer yang beralih dari Subversion ke salah satu DVCS yaitu Git atau Mercurial sehingga dapat kita perkirakan bahwa dalam beberapa waktu mendatang, grafik di atas akan berubah.

1.2 Git

Menurut Linus Torvalds, Git sebenarnya lebih cocok dikategorikan sebagai *file management system*, karena pada dasarnya Git bekerja seperti sebuah *content tracker* yang menyimpan histori sebuah *file system* (direktori dan file) dalam bentuk *full backup*. Hanya saja, kemudian terbukti bahwa Git juga bisa digunakan sebagai SCM (*Source Code Management*) sehingga Git menjadi sangat populer seperti saat ini.

"In many ways you can just see git as a filesystem — it's content-addressable, and it has a notion of versioning, but I really really designed it coming at the problem from the viewpoint of a file system person (hey, kernels is what I do), and I actually have absolutely zero interest in creating a traditional SCM system." – Linus Torvalds

Git adalah sistem yang terdistribusi mirip jaringan peer-to-peer. Central repository hanya berfungsi sebagai perantara dalam proses sinkronisasi. Secara teknis kita bisa melakukan sinkronisasi langsung antar repositori tapi hal ini sangat tidak disarankan.



Gambar 1-4 Repositori Git

Semua repositori mempunyai derajat yang sama, artinya user yang memiliki repositori tersebut bisa membuat *branch*, melakukan *commit* dan *reset* atau *revert*, memanipulasi histori secara independen tanpa harus berkomunikasi dengan server dan repositori yang lain.

Pada waktu sinkronisasi diperlukan, user melakukan *push* ke Git server dan user yang lain melakukan *fetch* (ambil) kemudian *merge* (menggabungkan) *update* terbaru di *Git server* dengan dengan repositori. Dengan kata lain, proses sinkronisasi dilakukan untuk menyamakan repositori lokal yang dimiliki oleh seorang user dengan user yang lain dengan perantara *Git server*.

Kalau kita ingin menggunakan Git bersama orang lain tanpa perlu repot membuat server sendiri, kita bisa menggunakan jasa beberapa provider antara lain:

- Bitbucket (www.bitbucket.org)
- Github (www.github.com)
- Googlecode (code.google.com)
- Beanstalk (www.beanstalkapp.com)

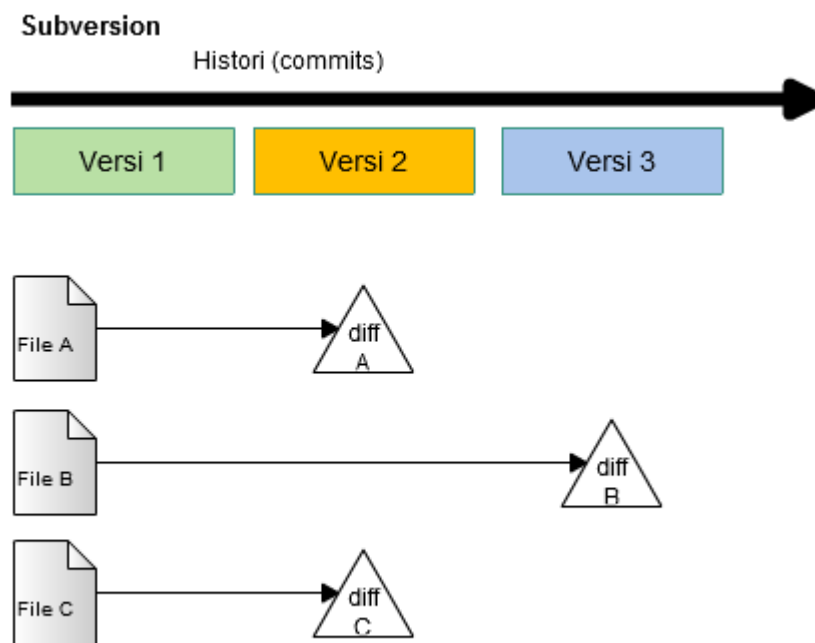
1.2.1 Data Model

Berbeda dengan CVCS yang hanya menyimpan bagian-bagian yang berubah (disebut juga *diff* atau *delta*) dalam setiap revisi, Git menyimpan sebuah *tree* yang berisi full-backup atau snapshot dari kondisi terakhir repositori dalam bentuk compressed file yang dapat diekspansi menjadi sebuah struktur direktori lengkap dengan file dan subdirektornya. Ini alasannya mengapa repositori Git selalu berukuran lebih besar daripada repositori CVCS seperti Subversion. Git menyimpan *working directory* secara utuh, bukan hanya bagian yang berubah.

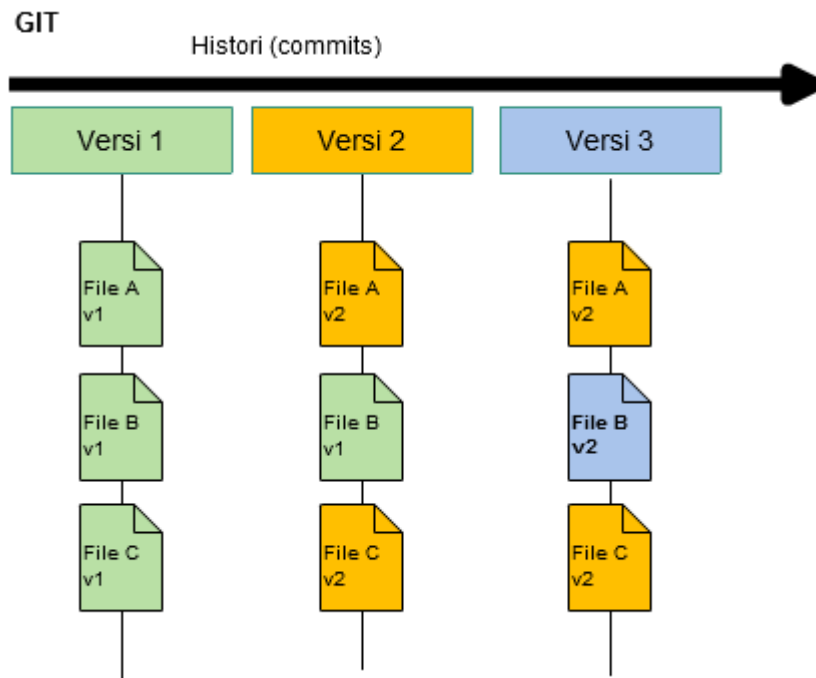
Gambar-gambar berikut menunjukkan perbedaan Subversion dan Git dalam menyimpan revisi. Dalam **Gambar 1-5**, **File B** dalam Versi 2, sama dengan **File B** dalam Versi 1 karena

yang berubah hanya File A dan File C. Dalam Versi 3, File B mengalami perubahan (revisi) sementara File A dan File C masih sama dengan yang ada dalam Versi 2. Setiap kali kita melakukan *check-in*, Subversion menyimpan perbedaan (*diff*) antara file yang baru dengan versi sebelumnya.

Gambar 1-6 memperlihatkan Git menyimpan struktur direktori secara utuh dalam setiap revisi. Dalam Versi 2, File A dan File C sudah dimodifikasi tetapi pada waktu kita melakukan *commit*, File B juga ikut disimpan dalam database. Begitu juga pada Versi 3, File A dan File C yang tidak dimodifikasi juga ikut disimpan bersama File B yang baru.



Gambar 1-5 Cara Subversion Menyimpan Revisi



Gambar 1-6 Cara Git Menyimpan Revisi

1.2.2 Istilah Penting

Sebelum kita belajar lebih lanjut, kita perlu mengenal beberapa istilah berikut ini yang detilnya akan kita pelajari dalam bab-bab selanjutnya.

Working directory

Working directory disebut juga *working tree* atau *working copy* adalah direktori di komputer kita (lokal) yang memiliki repositori. Indikasinya adalah adanya satu subdirektori bernama `.git`.

Repositori

Repositori adalah *database* yang menyimpan histori dari *working copy*. Penambahan, penghapusan, dan perubahan isi file semua direkam dalam database ini.

Commit

Commit adalah kondisi *working directory* pada satu waktu (*snapshot*) yang direkam dalam histori. *Snapshot* terbaru yang sudah direkam dalam histori disebut *HEAD commit*. *Commit* juga berarti proses penyimpanan *snapshot*.

Staging Index

Berbeda dengan CVCS, perubahan yang terjadi dalam direktori kerja tidak langsung disimpan dalam repositori, tetapi harus disimpan dulu di dalam *staging*

index. Repositori akan diperbarui (*update*) setelah kita melakukan *commit* atas file yang ada di dalam *staging index*.

Staged File

Staged file adalah file yang ada di dalam *staging index*.

Unstaged File

Unstaged File adalah file yang sudah dimodifikasi tetapi belum masuk ke dalam *staging index*.

Branch

Setiap repositori memiliki minimal satu cabang (*branch*) yang disebut *master branch*. Kita bisa membuat lebih dari satu cabang dalam satu repositori di mana masing-masing cabang memiliki histori sendiri-sendiri. Cabang-cabang ini nantinya bisa digabungkan satu sama lain melalui proses *merging*.

Merge

Merge adalah proses penggabungan sebuah *branch* lain ke dalam *branch* di mana kita berada.

Untracked File

Untracked file adalah file yang tidak pernah dimasukkan ke dalam *staging index* sehingga file tersebut berada di luar kontrol Git.

Tracked File

Tracked file adalah file yang dikontrol atau dimonitor oleh Git.

HEAD

HEAD adalah pointer/referensi yang merujuk pada *commit* terakhir.

Push

Push adalah proses sinkronisasi antara repositori lokal dengan *remote repository*. Di dalam proses ini, semua *commit* terbaru dikirim (*upload*) ke repositori tujuan.

Fetch

Fetch adalah kebalikan dari *push*. Di dalam proses ini semua *commit* terbaru di *remote repository* diunduh (*download*) ke repositori lokal.

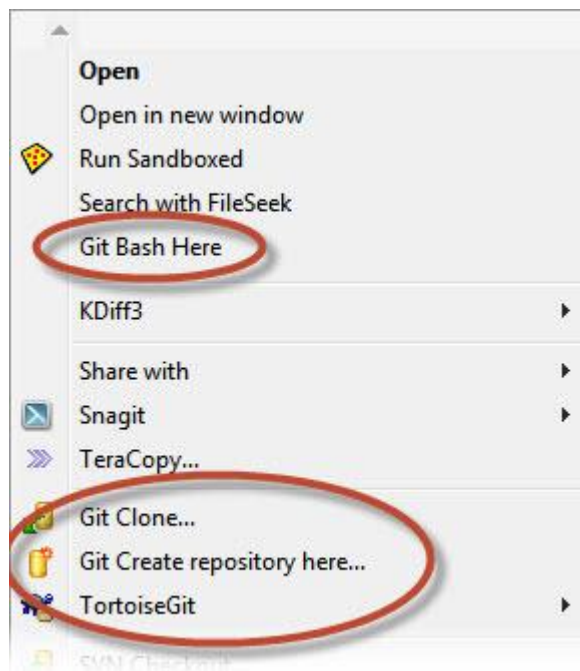
BAB 2. INSTALASI GIT

2.1 Windows

Pengguna Windows bisa menggunakan paket msysgit dari www.git-scm.com atau TortoiseGit yang menyediakan integrasi dengan Windows Explorer. Paket instalasinya berisi msysgit, Git GUI, dan git console (git bash). TortoiseGit bisa kita unduh di alamat berikut:

<http://code.google.com/p/tortoisegit/>

Setelah Anda unduh dan install, Anda akan melihat menu baru di context-menu seperti gambar berikut:



Gambar 2-1 Git di *context-menu*

2.1.1 Git Bash

Git bash adalah aplikasi konsol (terminal) yang disertakan dalam paket instalasi msysgit. Konsol ini adalah emulator linux terminal untuk windows jadi beberapa perintah linux seperti `touch`, `ls`, `rm`, `echo`, `cat`, dan `grep` bisa kita jalankan di sini.

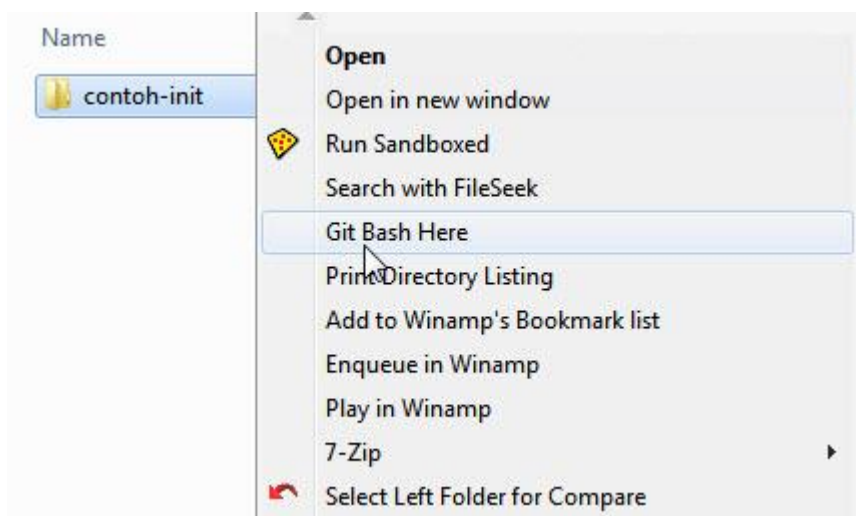


Gambar 2-2 Window Git Bash

Kita membuka Git Bash dengan cara :

Klik kanan pada sebuah direktori

Klik "Git Bash Here" di context-menu.



Gambar 2-3 Membuka Git Bash melalui context-menu

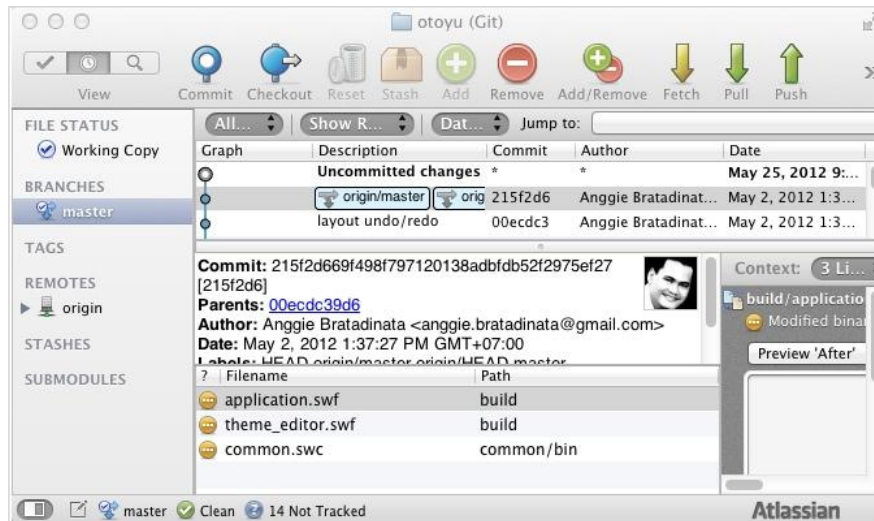
Jika Anda pengguna Windows yang belum pernah menggunakan Linux, silakan pelajari perintah-perintah dasarnya terutama yang berkaitan dengan file management karena dalam buku ini, kita akan lebih sering menggunakan terminal dengan tujuan mengenal Git dengan lebih baik.

2.2 Mac OSX

Paket instalasi Git untuk Mac bisa kita dapatkan di alamat berikut dalam bentuk .dmg.

<http://git-scm.com/download/mac>

Ada beberapa Git GUI gratis untuk Mac, salah satunya adalah Source Tree dari Atlassian yang bisa kita dapatkan di Apple AppStore.



Gambar 2-4 Atlassian Source Tree

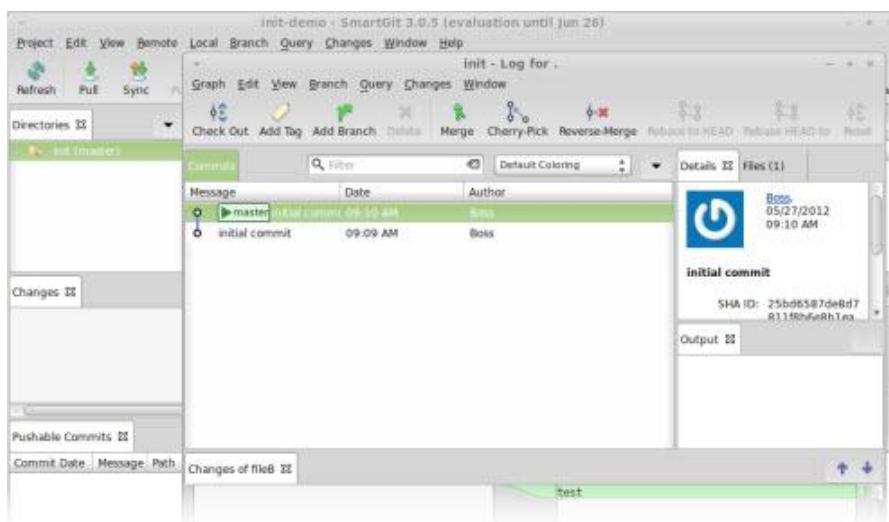
2.3 Ubuntu/Mint

Pengguna Ubuntu atau Linux Mint bisa menginstal Git melalui apt:

```
$ apt-get install git
```

Salah satu Git GUI yang tersedia untuk Linux adalah Smartgit dari Syntevo yang bisa Anda unduh di alamat :

<http://www.syntevo.com/smartgit/>

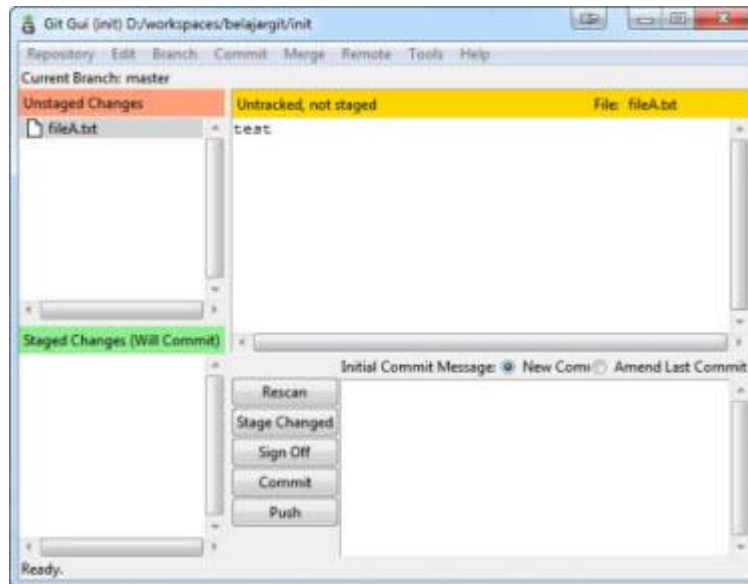


Gambar 2-5 SmartGit

2.4 Built-in Git GUI

Kalau Anda tidak ingin menggunakan aplikasi dari pihak ketiga, Anda bisa menggunakan GUI yang ter-install bersama paket git dengan perintah:

```
$ git gui
```



Gambar 2-6 Git gui

2.5 Konfigurasi

2.5.1 User

Sebelum kita menggunakan repositori, Git perlu tahu identitas kita (nama dan email) melalui konfigurasi user. Ada dua jenis konfigurasi yang bisa kita gunakan yaitu konfigurasi global dan lokal.

Global

Konfigurasi ini akan berfungsi sebagai identitas *default* semua repositori yang kita buat. Caranya adalah dengan menjalankan perintah `git config` seperti contoh berikut:

```
$ git config --global user.name "Dr. Jekyll"  
$ git config --global user.email "jekyll@home.com"
```

Selain dengan perintah di atas, kita juga bisa melakukan konfigurasi dengan mengedit langsung file `.gitconfig` dan menambahkan baris berikut:

```
[user]  
    name = Dr. Jekyll
```

```
email = jekyll@home.com
```

Jika Anda baru pertama kali menginstal Git, file `.gitconfig` belum ada di user directory. Anda bisa membuatnya sendiri melalui terminal dengan perintah `touch .gitconfig` atau menjalankan perintah `git config` seperti di atas. Berikut ini lokasi file `.gitconfig` untuk masing-masing sistem operasi :

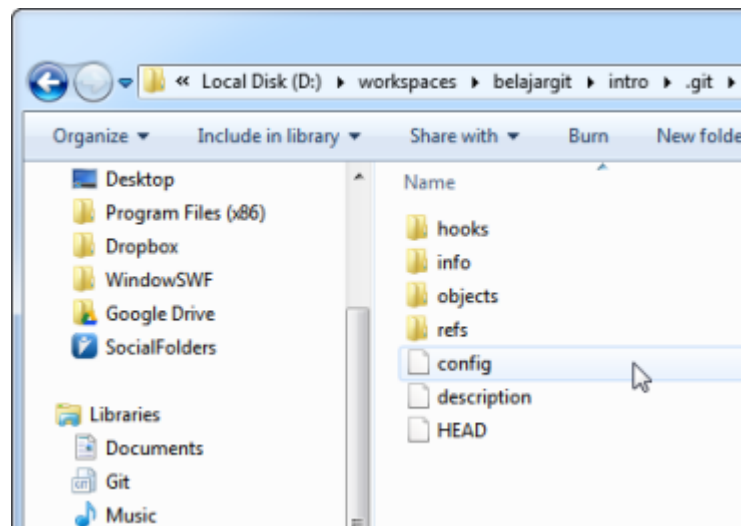
- `/Users/<namauser>/.``gitconfig` (Mac)
- `/home/<namauser>/.``gitconfig` (Linux)
- `C:\Users\<namauser>\.``gitconfig` (Windows Vista/7)

Lokal (Per Repositori)

Dalam beberapa situasi mungkin kita perlu menggunakan identitas yang berbeda untuk setiap repositori. Kita bisa membuat identitas lokal dengan perintah `git config` seperti di atas tapi tanpa argumen `--global` seperti berikut:

```
$ git config user.name "Mr. Hyde"
$ git config user.email "hyde@evil.com"
```

Konfigurasi lokal disimpan dalam file bernama `config` di dalam repositori seperti gambar berikut:



Gambar 2-7 Local Config

2.5.2 Default Message Editor

Setiap kali kita melakukan commit, secara default kita diharuskan menulis komentar (*message*). Kita bisa mengatur Git untuk membuka teks editor yang kita sukai untuk menulis *commit message*. Konfigurasi ini bukan keharusan tetapi akan membantu jika kita ingin menulis

message yang panjang, lebih dari satu baris. Secara default, git di linux/Mac dan juga Git Bash menggunakan *vi* sebagai message editor. Berikut ini contoh konfigurasi agar git menggunakan editor teks selain *vi*:

Notepad (Windows)

```
$ git config --global core.editor "notepad"
```

Notepad++ (Windows)

```
$ git config --global core.editor "'C:/Program  
Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession  
-noPlugin"
```

Textmate (Mac)

```
$ git config --global core.editor "mate -w"
```

Nano (Linux)

```
$ git config --global core.editor "nano"
```

2.5.3 Diff & Merge Utilities

Selain Git client, kita membutuhkan software diff/merge utility untuk membandingkan isi file dan menggabungkan dua file atau lebih antara lain:

- **KDiff3** -- Gratis & tersedia untuk Windows, Mac, & Linux.
- **Beyond Compare** -- Bayar. Untuk Windows & Linux.
- **Perforce P4Merge** -- Gratis. Untuk Windows, Mac, dan Linux.

Kalau Anda menggunakan TortoiseGit, Anda bisa menggunakan program Tortoise Merge yang sudah termasuk dalam instalasi atau software di atas. Berikut ini contoh konfigurasi agar git menggunakan diff/merge utility di atas sebagai default *mergetool*.

Beyond Compare 3 (www.scootersoftware.com)

```
$ git config --global diff.tool bc3  
$ git config --global difftool.bc3.path "C:/Program Files  
(x86)/Beyond Compare 3/BComp.exe"  
  
$ git config --global merge.tool bc3  
$ git config --global mergetool.bc3.path "C:/Program Files  
(x86)/Beyond Compare 3/BComp.exe"
```

KDiff3 (kdiff3.sourceforge.net)

```
$ git config --global diff.tool kdiff3
```

```
$ git config --global difftool.kdiff3.path "C:/Program Files
(x86)/KDiff3/KDiff3.exe"
$ git config --global difftool.kdiff3.keepBackup "false"

$ git config --global merge.tool kdiff3
$ git config --global mergetool.kdiff3.path "C:/Program Files
(x86)/KDiff3/KDiff3.exe"
$ git config --global mergetool.kdiff3.keepBackup "false"
```

Perforce P4Merge (www.perforce.com)

```
$ git config --global diff.tool p4merge
$ git config --global difftool.p4merge.path "C:/Program
Files/Perforce/p4merge.exe"
$ git config --global difftool.p4merge.keepBackup "false"

$ git config --global merge.tool p4merge
$ git config --global mergetool.p4merge.path "C:/Program
Files/Perforce/p4merge.exe"
$ git config --global difftool.p4merge.keepBackup "false"
```

Sama seperti konfigurasi user, Anda juga bisa mengedit langsung file .gitconfig, berikut ini contoh konfigurasi p4merge di atas sebagai referensi.

```
[diff]
    tool = p4merge

[difftool "p4merge"]
    path = C:/Program Files/Perforce/p4merge.exe
    keepBackup = false

[merge]
    tool = p4merge

[mergetool "p4merge"]
    path = C:/Program Files/Perforce/p4merge.exe
    keepBackup = false
```

BAB 3. PERINTAH-PERINTAH DASAR

3.1 Membuat Repositori

Kita bisa mulai menggunakan repositori Git setelah proses inialisasi yang terdiri dari dua langkah yaitu :

1. Inialisasi repositori
2. *Initial commit*

Sebagai contoh, mari kita buat direktori bernama `contoh-init` sebagai *working copy*. Untuk membuat repositori, kita jalankan perintah `git init` di direktori tersebut. Perintah ini membuat subdirektori baru bernama `.git`.

```
$ git init
Initialized empty Git repository in
/home/boss/Desktop/gitbook/contoh-init/.git/
```

Jika Anda belum pernah menggunakan linux terminal, perlu Anda ketahui bahwa dalam contoh di atas dan contoh-contoh berikutnya, karakter **dollar sign (\$)** adalah *terminal prompt* atau lebih tepatnya *default bash prompt*. Anda tidak perlu mengetik karakter ini di terminal, cukup perintah yang ada di sebelah kanannya saja.

Repositori yang baru kita buat belum bisa digunakan karena Git tidak tahu apa yang harus dikontrol sampai kita melakukan *initial commit*. Karena *working copy* masih kosong, kita harus membuat minimal satu buah file sebab Git akan mengabaikan direktori kosong. Karena itu, kita buat file bernama `File_A` dan melakukan *initial commit* seperti berikut:

```
// buat file_A
$ touch File_A

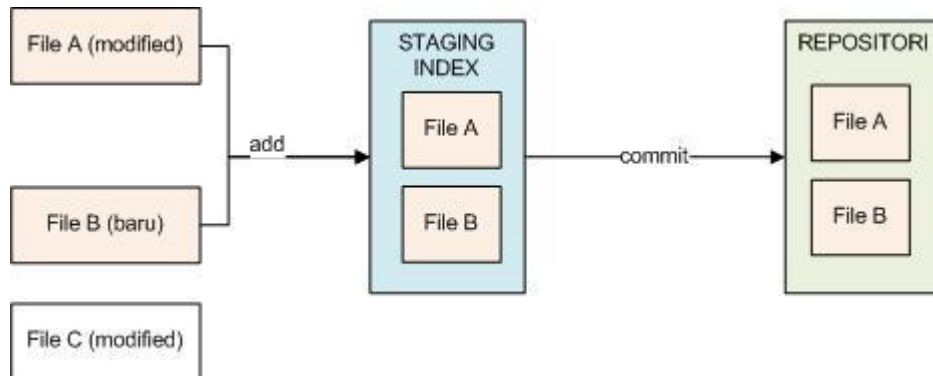
// tambahkan file ke dalam staging index
$ git add File_A

// initial commit
$ git commit -m "Initial commit"
```

"Initial commit" adalah message yang umum dipakai untuk initial commit. Anda bisa menulis pesan lain yang Anda mau.

3.2 Commit

Commit adalah perintah atau proses update repositori. Pada saat perintah ini dijalankan, Git menduplikasi versi terakhir *working copy* kemudian menggunakan file yang ada di *staging index* untuk memperbarui duplikat tersebut dan menyimpannya sebagai versi baru.



Gambar 3-1 Git Staging & Commit

Commit kita lakukan dengan perintah `git commit` dengan argumen opsional `-m` yang berisi keterangan (*commit message*). Argumen ini kita gunakan setiap kali kita ingin menulis satu baris keterangan singkat.

```
$ git commit -m <keterangan>
```

Kalau kita tidak menggunakan argumen `-m`, Git akan membuka editor teks bagi kita untuk menulis *commit message*. Ini kita lakukan jika kita ingin menulis keterangan yang panjang, lebih dari satu baris. Proses *commit* akan ditunda sampai kita menutup editor.

3.2.1 Staging Index

Staging index adalah tempat di mana file yang sudah dimodifikasi dan file baru disimpan sebelum di-commit ke repositori. Perintah `commit` hanya akan memproses file yang ada di *staging index*. Untuk menambahkan file yang dimodifikasi atau file baru ke dalam *staging index*, kita gunakan perintah `git add`.

Sebagai contoh, kita buat *working copy* baru bernama `staging-demo` dan kita buat repositori di direktori tersebut. Kemudian kita lakukan langkah-langkah berikut:

1. Membuat `File_A` dan `File_B`

2. Menambahkan kedua file tersebut ke dalam *staging index*
3. Initial commit

```
// buat direktori
$ mkdir staging-demo
$ cd staging-demo

// inisialisasi repositori
$ git init

// buat file
$ touch File_A File_B

// tambahkan file ke staging index
$ git add .

// initial commit
$ git commit -m "Initial commit"
```

Berikutnya, kita ubah isi `File_A` dengan memasukkan teks "hello".

```
$ echo 'hello' > File_A
```

Jika kita coba lakukan *commit* saat ini, Git akan memberi tahu kita bahwa `File_A` belum ada di *staging index* dan tidak ada yang bisa kita *commit*.

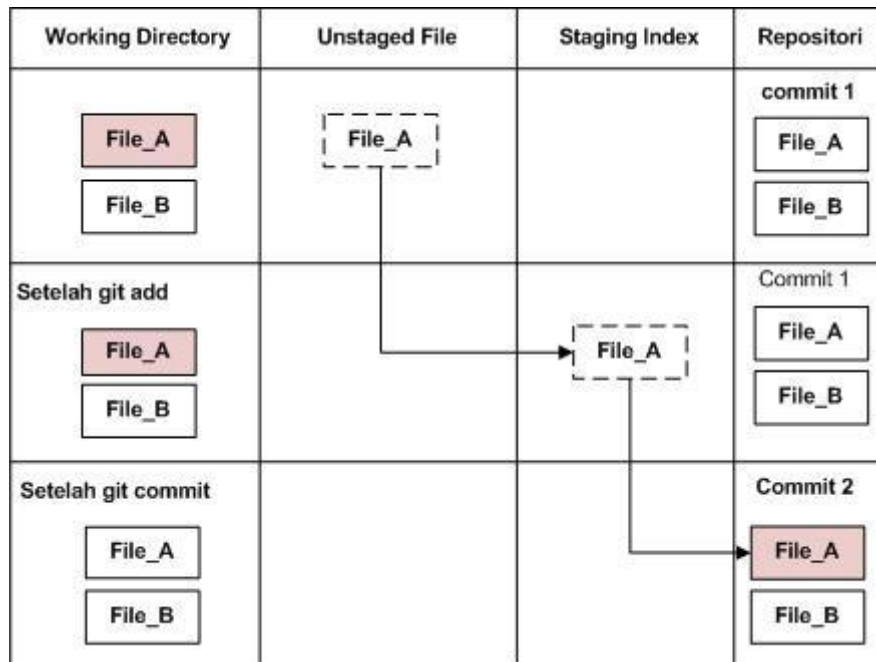
```
$ git commit -m "update file A"
# On branch master
# Changes not staged for commit:
#   modified:   File_A
#
no changes added to commit
```

Sekarang kita tambahkan `File_A` ke staging dan cek status. Kita akan melihat `File_A` siap untuk di-*commit*.

```
// tambahkan file ke staging index
$ git add File_A

$ git status
# On branch master
# Changes to be committed:
#   modified:   File_A
```

Gambar 3-2 adalah ilustrasi sederhana kondisi setelah `File_A` diubah isinya sampai *commit* dilakukan. **Commit 1** dalam kolom paling kanan, baris pertama dan kedua, menunjukkan *commit* terakhir sebelum `File_A` dimodifikasi. **Commit 2** adalah *commit* yang dilakukan untuk menyimpan versi terbaru dari `File_A` setelah modifikasi.



Gambar 3-2 Proses Staging sampai Commit

Baris pertama menggambarkan kondisi setelah isi `File_A` diubah dimana Git mengenalinya sebagai unstaged file. Kalau saat ini kita melakukan *commit*, `File_A` tidak termasuk file yang diproses karena belum ditambahkan ke dalam *staging index*. Dalam **baris kedua**, `File_A` sudah ditambahkan ke *staging index* dan siap di-*commit*. **Baris ketiga** menggambarkan kondisi repositori setelah *commit*, di mana `HEAD` berisi versi terbaru dari `File_A` dan juga `File_B` yang masih sama dengan `File_B` versi sebelumnya karena file ini tidak mengalami modifikasi.

File yang tidak dimodifikasi tetap disimpan dalam database pada saat *commit* karena Git menyimpan full-backup (snapshot) dari *working copy*.

Kita menggunakan perintah `git add` untuk menambahkan satu atau lebih file ke dalam *staging index* dengan menggunakan nama file-file tersebut sebagai argumen.

```
$ git add File_A File_B File_C
```

Jika kita ingin memproses semua file, baik file lama (*tracked file*) yang sudah dimodifikasi maupun file baru, kita bisa gunakan argumen dot (`.`).

```
$ git add .
```

Membatalkan Staging (Soft Reset)

Untuk membatalkan (*undo*) perintah `git add` dan mengeluarkan file dari *staging index*,

kita jalankan perintah `git reset HEAD`. Proses ini disebut juga *unstaging* karena status file yang sebelumnya ada di *staging index* (*staged*) berubah menjadi *unstaged*.

HEAD adalah pointer atau referensi yang merujuk pada *commit* terakhir.

Soft reset mengeluarkan file dari *staging index* tanpa mereset isi file. Perubahan yang kita lakukan terhadap isi file yang bersangkutan tetap ada hanya saja file tersebut tidak bisa diikuti dalam proses *commit*. Dalam contoh berikut, kita lakukan *soft-reset* setelah `File_A` kita modifikasi. Kita lihat bahwa setelah reset, status `File_A` tetap "*modified*" yang berarti perubahan yang kita lakukan tidak hilang.

```
// isi File_A dengan teks hello world
$ echo 'hello world' > File_A

// lihat status
$ git status
# On branch master
# Changes not staged for commit:
#   modified:   File_A

// tambahkan file ke staging index
$ git add File_A

// lihat status working directory
$ git status
# On branch master
# Changes to be committed:
#   modified:   File_A

// soft-reset
$ git reset HEAD
Unstaged changes after reset:
    M       File_A

// File_A dikeluarkan dari staging index,
// tapi tetap dalam kondisi modified
$ git status
# On branch master
# Changes not staged for commit:
#   modified:   File_A
```

Soft-reset juga bisa dilakukan terhadap file secara individu sehingga kita bisa memilih file mana saja yang ingin kita keluarkan dari *staging index*. Untuk melakukannya, kita jalankan perintah yang sama dengan nama file di belakangnya. Sebagai contoh, untuk melakukan *soft-reset* terhadap `File_A` saja, kita jalankan perintah:

```
$ git reset HEAD File_A
```

3.2.2 Membatalkan Commit

Kita tidak bisa membatalkan commit yang telah kita lakukan tetapi kita bisa memperbaikinya dengan melakukan *commit* pengganti dengan perintah `git commit --amend`.

TODO : contoh

3.2.3 log

Untuk melihat semua *commit* yang sudah kita lakukan, kita jalankan perintah `git log` seperti contoh berikut:

```
$ git log

commit eb47f853eb4ecff2933d5682229109ceba2513fe
Author: Angie Bratadinata <anggie.bratadinata@gmail.com>
Date:   Wed May 16 19:54:45 2012 +0700

    update File_A lagi

commit c17d0ad486426cce17c88c66d82e447b9dcbd72a
Author: Angie Bratadinata <anggie.bratadinata@gmail.com>
Date:   Wed May 16 18:50:45 2012 +0700

    update File_A

commit 1c673212d02c20e9610dcf98cbc6877ccf0c5fbc
Author: Angie Bratadinata <anggie.bratadinata@gmail.com>
Date:   Wed May 16 17:50:42 2012 +0700

    Initial commit
```

Deretan karakter yang tampaknya acak di belakang kata *commit* adalah hash-value yang merupakan nomor unik sebagai identitas *commit* yang bersangkutan. Jadi sebenarnya Git tidak mengenal nomor revisi, yang ada adalah sekumpulan *commit* ID namun untuk mempermudah, kita anggap saja *commit* ID sebagai nomor revisi.

Kalau kita bekerja dengan orang lain menggunakan shared repository, log tidak hanya menampilkan semua *commit* yang kita lakukan tetapi juga yang dilakukan oleh rekan kita.

Kita juga bisa menampilkan log dalam format yang lebih ringkas dengan menambahkan argumen `--oneline` seperti berikut:

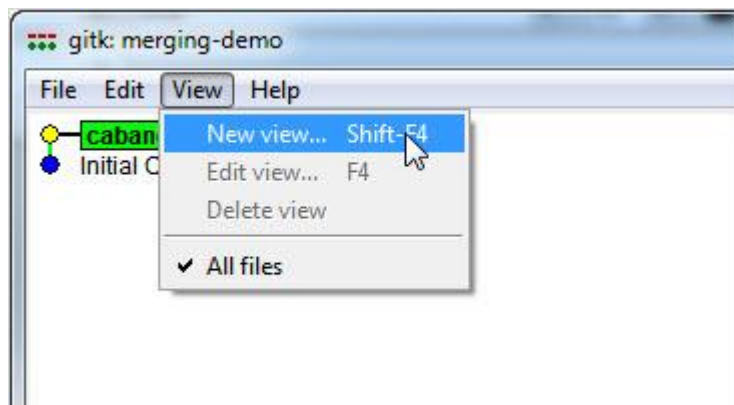
```
$ git log --oneline
eb47f85 update file A lagi
c17d0ad update File_A
1c67321 Initial commit
```

Untuk melihat *commit* log secara visual, kita bisa menggunakan perintah `gitk`.

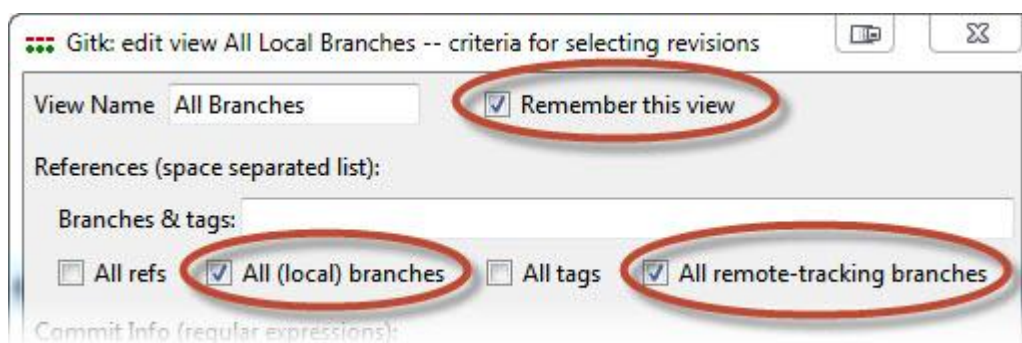
```
$ gitk
```

Secara default, `gitk` hanya menunjukkan *branch* di mana kita berada pada saat perintah tersebut dijalankan. Untuk melihat semua *branch*, lakukan langkah-langkah berikut:

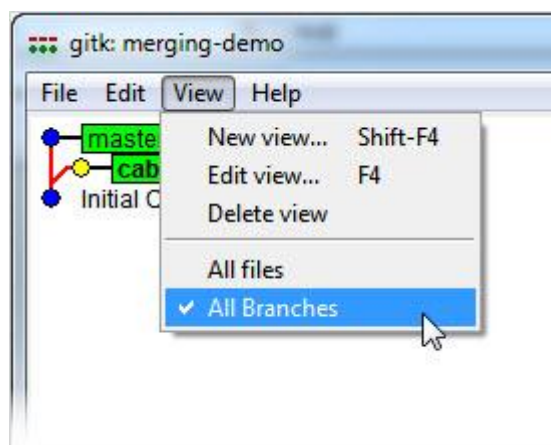
1. Klik View->New view (**Gambar 3-3**)
2. Dalam dialog window, isi nama view, cek "*Remember this view*", "All (local) branches", dan "All remote-tracking branches" (**Gambar 3-4**)
3. Klik OK
4. Untuk menampilkan view yang sama, kita pilih nama view dari menu seperti **Gambar 3-5** setiap kali kita menjalankan perintah `gitk`.



Gambar 3-3 Membuat View di gitk



Gambar 3-4 View Dialog



Gambar 3-5 Membuka View di gitk

Satu hal yang sering dilupakan orang adalah menulis log *commit message* yang informatif. Log memegang peranan penting dalam semua *version control* karena log yang baik dan informatif mempermudah maintenance dan pengembangan produk. Log yang baik memberi informasi tentang sejarah proyek dan mempermudah perbaikan bug. Sebaliknya log yang ditulis asal-asalan malah mengganggu dan mempersulit.

3.3 Menangani File

3.3.1 Membandingkan Isi File (diff)

Seringkali kita perlu melihat modifikasi apa saja yang kita lakukan terhadap file *di working directory* dengan cara membandingkan isi file terbaru dengan versi sebelumnya. Ada tiga situasi yang membutuhkan perintah yang berbeda untuk melihat perbedaan file yaitu:

1. File belum kita tambahkan ke *staging index* (*unstaged file*)
2. File sudah ada di *staging index* (*staged file*)
3. File sudah kita *commit*

Sebagai contoh, kita buat direktori baru bernama `diff-demo` lalu kita buat dua buah file, `fileA` dan `fileB`, kemudian kita lakukan *initial commit*.

```
$ mkdir diff-demo
$ cd diff-demo

// inisialisasi repositori
$ git init

// buat file
$ touch fileA

// initial commit
$ git commit -m "initial commit"
```

Unstaged files

Untuk melihat perbedaan file yang belum kita masukkan ke dalam *staging index*, kita gunakan perintah `git diff` dengan argumen nama file yang ingin kita cek. Kita coba ubah isi `fileA` lalu kita jalankan perintah tersebut.

```
// ubah isi fileA
$ echo 'hello' > fileA

// lihat modifikasi yang kita lakukan
// terhadap fileA
$ git diff fileA
diff --git a/fileA b/fileA
index e69de29..ce01362 100644
--- a/fileA
+++ b/fileA
@@ -0,0 +1 @@
+hello
```

Dalam contoh di atas, kita lihat kata `+hello`. Ini adalah baris yang kita tambahkan ke dalam `fileA`. Tanda plus (+) berarti baris baru (*added line*). Berikutnya, kita coba *commit* `fileA` lalu lakukan modifikasi lagi.

```
//tambahkan semua file ke dalam staging index
$ git add .

// initial commit
$ git commit -m 'update fileA'

// ubah isi fileA
$ echo 'hello world' > fileA

//lihat perbedaan isi fileA
$ git diff fileA
diff --git a/fileA b/fileA
index ce01362..14be0d4 100644
--- a/fileA
+++ b/fileA
@@ -1,1 +1,1 @@
-hello // baris yang dihapus
+hello world // baris baru
```

Kita lihat baris `-hello` dan `+hello world`. Ini berarti baris `hello` dihapus dan baris `hello world` ditambahkan. Tanda minus (-) berarti baris dihapus (*removed line*).

Staged files

Untuk file yang sudah masuk dalam *staging index*, perintah `git diff` tidak menampilkan apa-apa kecuali kita tambahkan argumen `--staged` seperti contoh berikut di mana kita tambahkan `fileA` ke dalam *staging index* terlebih dahulu.

```
$ git add fileA

// lihat modifikasi isi fileA yang
// sudah ada di dalam staging index
$ git diff --staged fileA

diff --git a/fileA b/fileA
index ce01362..14be0d4 100644
--- a/fileA
+++ b/fileA
@@ -1,1 @@
-hello
+hello world
```

Committed files

Kita bisa membandingkan fileA antara dua *commit* manapun dengan menggunakan *commit ID* sebagai argumen.

Sekarang kita coba *commit* fileA.

```
$ git commit -m 'update fileA lagi'
```

Untuk melihat perbedaan antara fileA dalam *commit* terakhir dengan fileA dalam *commit* sebelumnya kita perlu tahu ID kedua *commit*. Jadi, kita perlu melihat *log* terlebih dahulu.

```
$ git log --oneline
053ea41 update fileA lagi
5acdc27 update fileA
8363fcf initial commit
```

Dalam contoh di atas, id *commit* terakhir adalah 053ea41 dan *commit* sebelumnya adalah 5acdc27. Kita gunakan kedua id tersebut sebagai argumen perintah `git diff`.

```
$ git diff 5acdc27..053ea41 fileA
diff --git a/fileA b/fileA
index ce01362..14be0d4 100644
--- a/fileA
+++ b/fileA
@@ -1,1 @@
-hello
+hello world
```

Membandingkan Isi Semua File

Bagaimana jika kita ingin melihat perbedaan isi semua file dalam *working directory* tanpa mengetikkan perintah `git diff` berulang kali? Mudah. Jangan gunakan nama file sebagai argumen. Sebagai contoh, mari kita buat fileB lalu kita lakukan *commit*.

```
$ touch fileB
```

```
$ git add fileB
$ git commit -m 'add fileB'
```

Kemudian kita modifikasi `fileA` dan `fileB` seperti berikut.

```
$ echo 'ini isi fileA' > fileA
$ echo 'ini isi fileB' > fileB
```

Untuk melihat perbedaan isi `fileA` dan `fileB` dibandingkan dengan isi kedua file dalam *commit* sebelumnya, kita jalankan perintah `git diff` tanpa argumen seperti berikut.

```
$ git diff

diff --git a/fileA b/fileA
index 3b18e51..3432579 100644
--- a/fileA
+++ b/fileA
@@ -1,1 @@
-hello world
+ini isi fileA

diff --git a/fileB b/fileB
index e69de29..98d3e2f 100644
--- a/fileB
+++ b/fileB
@@ -0,0 +1 @@
+ini isi fileB
```

Sekarang kita coba *commit* lalu kita lihat perbedaan semua file antara *commit* terakhir dengan *commit* sebelumnya.

```
$ git add .
$ git commit -m 'update all'
[master 3a8eecd] update all
 2 files changed, 2 insertions(+), 1 deletions(-)

// lihat log
$ git log --oneline
3a8eecd update all
d68b9ef add fileB
cbea6a2 update fileA lagi
8820be7 update fileA
0dcf712 initial commit

// lihat semua modifikasi terhadap semua file
// dalam working directory
$ git diff d68b9ef..3a8eecd

diff --git a/fileA b/fileA
index 3b18e51..3432579 100644
--- a/fileA
+++ b/fileA
@@ -1,1 @@
-hello world
```



```
+ini isi fileA

diff --git a/fileB b/fileB
index e69de29..98d3e2f 100644
--- a/fileB
+++ b/fileB
@@ -0,0 +1 @@
+ini isi fileB
```

Sebagai alternatif, kita bisa memanfaatkan diff/merge utility untuk melihat perbedaan file dengan menjalankan perintah `git difftool` dengan argumen yang sama dengan contoh argumen `git diff` di atas, seperti contoh berikut:

```
$ git difftool fileA
$ git difftool --staged fileA
$ git difftool d68b9ef..3a8eecd
```

3.3.1 Membatalkan Modifikasi (Hard Reset)

Hard reset adalah proses mereset isi file dan mengembalikan kondisinya seperti kondisi pada saat *commit* terakhir. *Hard reset* bisa dilakukan terhadap file secara individual atau terhadap keseluruhan *working copy*.

Soft reset tidak mengubah isi file. *Hard reset* mengembalikan isi file seperti kondisi pada saat *commit* terakhir.

Hard Reset File

Perintah `git checkout` kita gunakan jika kita ingin mereset isi sebuah file dan mengembalikannya seperti kondisi pada saat *commit* terakhir. Perintah ini pada dasarnya akan mengambil sebuah file dari database Git untuk menggantikan file yang ada di *working copy*. Contoh:

```
$ git checkout -- File_A
```

Perintah di atas hanya efektif sebelum file belum kita masukkan ke dalam *staging index*. Jika file tersebut sudah terlanjur kita masukkan ke dalam *staging index*, kita harus melakukan *soft reset* terlebih dahulu.

```
$ git reset HEAD File_A
$ git checkout -- File_A
```

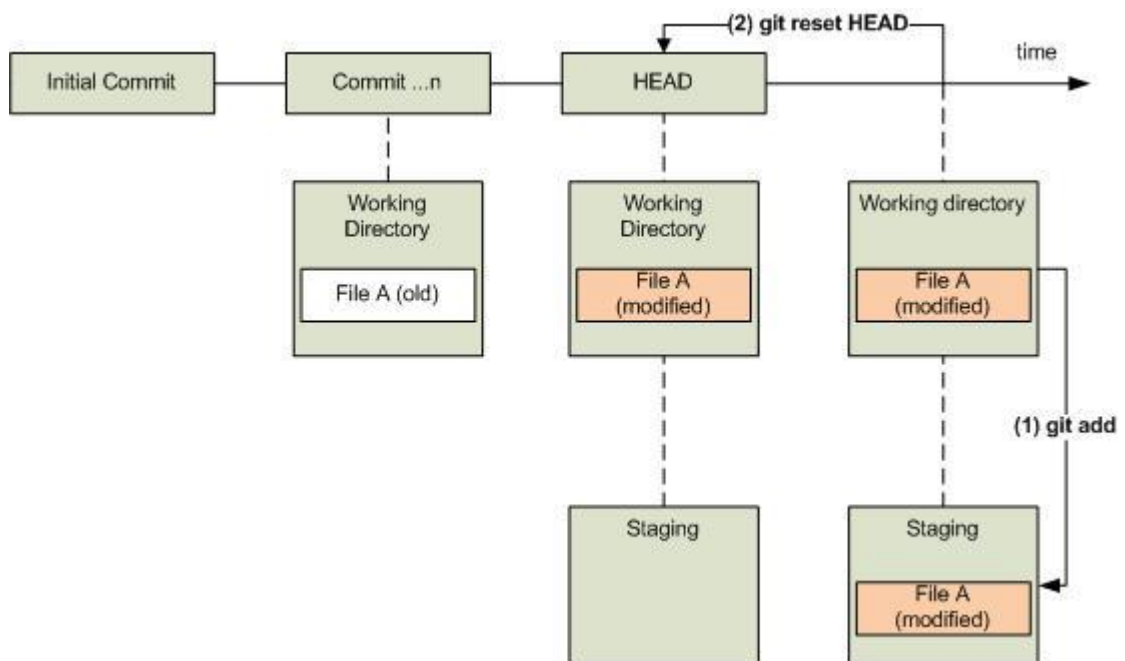
Hard Reset Working copy

Untuk mengembalikan kondisi *working copy* kepada kondisi setelah *commit* terakhir dan membatalkan semua perubahan yang kita lakukan terhadap *tracked file*, kita jalankan perintah `git reset --hard`.

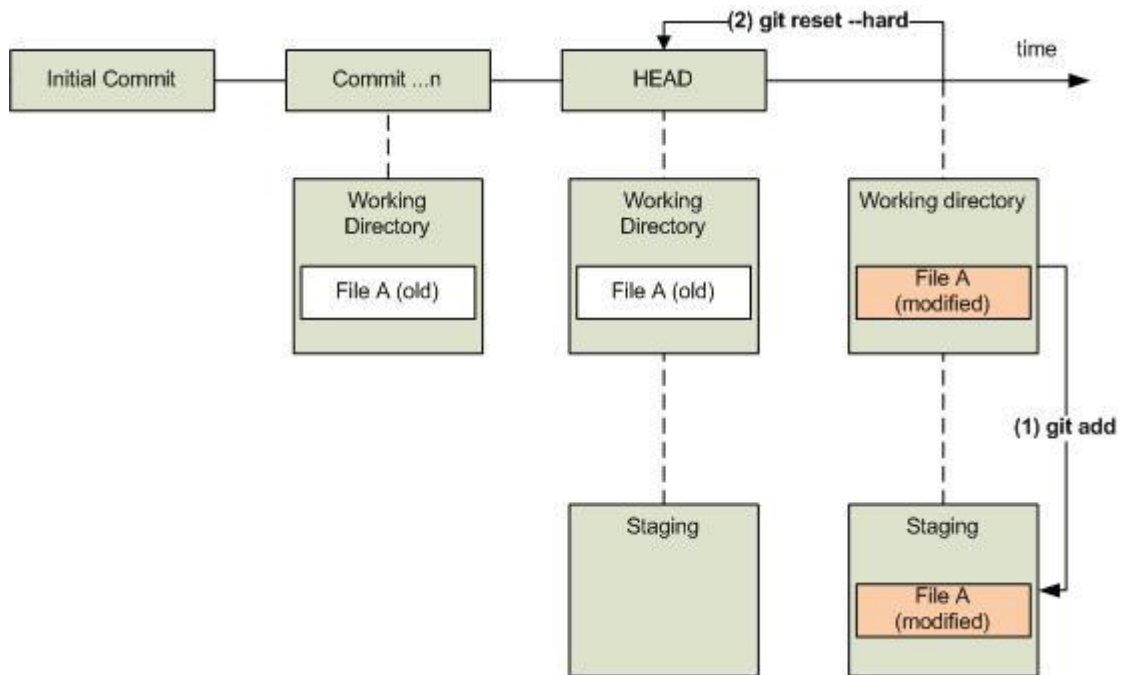
```
$ git reset --hard
HEAD is now at d975655 Initial commit

$ git status
# On branch master
nothing to commit (working copy clean)
```

Perbedaan soft dan hard reset bisa diilustrasikan dengan gambar-gambar berikut.



Gambar 3-6 Soft Reset



Gambar 3-7 Hard Reset

3.3.2 Menghapus File

Kita bisa menghapus file dengan atau tanpa perintah git. Perintah untuk menghapus file adalah `git rm`. Perintah ini akan menghapus file dari repositori dan juga dari *working directory*.

```
$ touch fileA
$ git add fileA
$ git commit -m 'add fileA'

// hapus file dari repositori
$ git rm fileA
$ git status
# On branch master
# Changes to be committed:
#   deleted:   fileA

$ git commit -m 'hapus fileA'
```

Kita juga bisa menghapus file tanpa perintah `git rm`, misalnya lewat *Windows explorer*. Penghapusan file tanpa perintah git, tidak otomatis menghapus file dari database git. Jadi kita perlu melanjutkannya dengan perintah `git rm` atau menjalankan perintah `git commit` dengan argumen `-a`.

```
$ touch fileB
$ git add fileB
$ git commit -m 'add fileB'

// hapus file tanpa perintah git
$ rm fileB
```

```
$ git status
# On branch master
# Changes not staged for commit:
#   deleted:   fileB
no changes added to commit

// git rm dilanjutkan dengan commit
$ git rm fileB
$ git commit -m 'hapus fileB'

//alternatif : commit dengan argumen -a
$ git commit -a -m 'hapus fileB'
```

3.4 Berpindah Versi

Kita bisa maju atau mundur ke versi manapun kapan saja kita mau dengan menjalankan perintah `git reset` dengan argumen *commit* ID yang kita tuju. Kita tidak perlu mengetikkan keseluruhan *commit* ID, tetapi cukup menggunakan beberapa karakter awal seperti yang kita lihat dalam log ringkas yang telah kita pelajari dalam bagian sebelumnya.

Untuk menghindari masalah, kalau ada file yang telah dimodifikasi, file tersebut sebaiknya Anda *commit* terlebih dahulu sebelum menjalankan perintah reset untuk kembali ke versi lama. Kedua, jangan mengubah file atau struktur *working copy* selama berada dalam versi lama. Ketiga, jika Anda tidak menggunakan Git GUI seperti TortoiseGit, sebelum melakukan reset, catat terlebih dahulu *commit* ID yang terakhir supaya mudah untuk kembali ke versi terbaru.

Misalnya kita ingin kembali ke *commit* dengan keterangan "*Initial commit*" yang memiliki ID dengan awalan `1c67321` maka kita jalankan perintah:

```
$ git reset 1c67321
```

Untuk melihat di versi mana kita berada saat ini, kita jalankan perintah `git log`. Baris paling atas menunjukkan posisi kita saat ini.

```
$ git log --oneline
1c67321 Initial commit
```

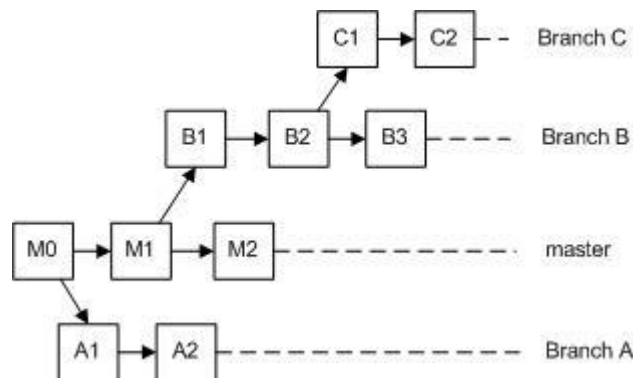
Kita lihat di log, *commit* terakhir adalah "*Initial commit*". Itu berarti kondisi *working copy* saat ini sama dengan kondisi setelah *commit* tersebut. Untuk kembali ke versi terbaru, kita jalankan perintah yang sama tetapi dengan *commit* ID yang terbaru:

```
$ git reset eb47f85
$ git log --oneline
eb47f85 update file A lagi
```

```
c17d0ad update File_A
1c67321 Initial commit
```

3.5 Branching & Merging

Branching adalah proses pembuatan "cabang". Fitur ini merupakan salah satu alasan utama banyak orang yang lebih suka menggunakan VCS daripada backup konvensional. Dengan membuat cabang, kita bisa memiliki beberapa versi alternatif yang masing-masing memiliki histori sendiri. Git, sebagai VCS terdistribusi, memungkinkan kita membuat *branch* lokal tanpa harus terkoneksi dengan server.



Gambar 3-8 Git Branch

Sebuah *branch* pada awalnya adalah duplikat dari *branch* lain di mana perintah `git branch` dijalankan namun nantinya *branch* ini memiliki histori yang independen seperti **Gambar 3-8**. *Branch* di mana perintah `git branch` dijalankan disebut sebagai *ancestor* atau *parent branch* sedangkan *branch* yang baru dibuat disebut *child branch*. Setiap repositori Git memiliki minimal satu *branch* yaitu `master` yang dibuat pada waktu kita menjalankan perintah `git init`.

Untuk menjalankan contoh-contoh *branching*, kita buat direktori baru bernama `branch-demo` dan kita lakukan inisialisasi.

```
$ mkdir branch-demo
$ cd branch-demo
$ git init
$ touch File_A.txt
$ git add .
$ git commit -m "Initial commit"
```

3.5.1 Membuat Branch

Untuk membuat cabang baru sebagai *child* dari *branch* di mana kita berada saat init, kita jalankan perintah `git branch <nama branch>` seperti berikut:

```
$ git branch cabang_A
```

Karena `cabang_A` dibuat pada waktu kita berada di `master`, maka pada awalnya, `cabang_A` adalah duplikat dari `master`. Di sini `master` disebut sebagai *ancestor* atau *parent* dari `cabang_A` dan `cabang_A` adalah *child* dari `master`.

Untuk melihat semua local *branch* kita jalankan perintah:

```
$ git branch
  cabang_A
* master
```

Di sini kita lihat ada dua *branch* yaitu `cabang_A` dan `master`.

Branch dengan tanda asterisk (*), adalah *branch* di mana kita berada saat ini.

Dalam Git Bash, nama *branch* di mana kita berada disebut di dalam tanda kurung, di belakang directory path. Contoh berikut menunjukkan bahwa kita sekarang berada di `master` *branch*:

```
/d/workspaces/belajargit/branch-demo (master)
```

Untuk berpindah *branch*, kita jalankan perintah `git checkout <nama branch>` seperti berikut:

```
$ git checkout cabang_A
Switched to branch 'cabang_A'
$ git status
# On branch cabang_A
```

Sebagai cara alternatif, kita juga bisa membuat *branch* baru dan langsung berpindah ke *branch* tersebut dengan satu baris perintah berikut:

```
$ git checkout -b cabang_A
```

Perintah di atas sama dengan:

```
$ git branch cabang_A
$ git checkout cabang_A
```

Selain perintah di atas, kita juga bisa membuat *child branch* dari *branch* manapun dengan cara berpindah ke *branch* yang akan dijadikan *parent* dan kemudian membuat *child branch* dengan perintah berikut:

```
$ git checkout parent-branch -b branch-baru
```

Perintah di atas sama dengan :

```
$ git checkout parent-branch
$ git checkout -b branch-baru
```

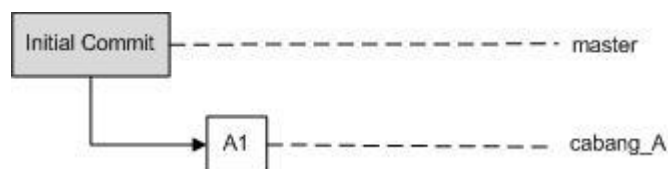
3.5.2 Berpindah Branch

Selama kita bekerja di dalam `cabang_A`, `master` tidak akan berubah karena perubahan yang kita lakukan sifatnya lokal dan hanya terjadi di `cabang_A`.

Kita coba buat `fileB.txt` di dalam `cabang_A`. Kemudian kita *commit*.

```
$ git checkout cabang_A
$ touch fileB.txt
$ git add .
$ git commit -m 'A1'
```

Saat ini kondisi `cabang_A` sudah berbeda dengan `master` dan data repositori dapat digambarkan sebagai berikut.

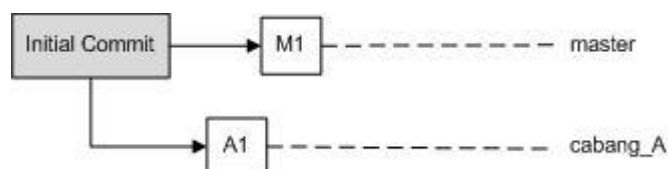


Gambar 3-9 Cabang Baru

Kembali ke *branch* `master` dan kita buat file baru bernama `fileC.txt`.

```
$ git checkout master
$ touch fileC.txt
$ git add fileC.txt
$ git commit -m "M1"
```

Setelah kita melakukan *commit*, data repositori menjadi seperti gambar berikut. Saat ini `cabang_A` dan `master` sudah terpisah jauh dan berada dalam kondisi yang disebut **diverging**.

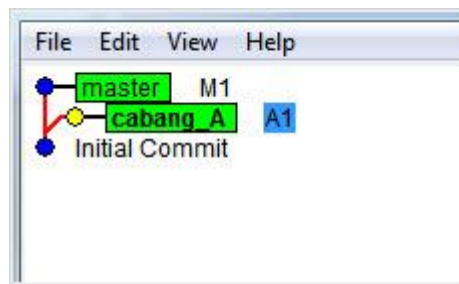


Gambar 3-10 Diverging Branch

Kalau kita berpindah lagi ke `cabang_A`, kita tidak akan menemukan `fileC.txt` karena file tersebut hanya ada di `master`. Sebaliknya, karena `fileB.txt` dibuat di dalam `cabang_A`, maka ketika kita kembali ke `master`, kita tidak akan menemukan file tersebut.

Berikut ini tampilan log repositori setelah kita melakukan *commit* di *branch* `master` seperti

contoh.



Gambar 3-11 Git log setelah commit master

Sebelum berpindah *branch*, semua modifikasi terhadap *tracked file* harus kita *commit* terlebih dahulu. Jika tidak, perintah `git checkout` akan gagal dan kita mendapat pesan *error* seperti berikut:

```
error: Your local changes to the following files would be
overwritten by checkout:
  fileA.txt
Please, commit your changes or stash them before you can
switch branches.
Aborting
```

Kesalahan (*error*) di atas hanya terjadi jika kita berpindah ke *branch* yang sudah dalam kondisi *diverging* dan ada kemungkinan terjadi konflik karena *branch* yang kita tuju memiliki modifikasi yang lebih baru. Selama *branch* yang kita tuju dan *branch* asal belum berada dalam kondisi *diverging* dan tidak ada potensi konflik, misalnya antara `master` dan sebuah *child branch* baru, modifikasi terhadap *tracked file* akan terbawa ke *branch* baru tersebut. Seperti contoh berikut, kita memodifikasi `fileA.txt` dan kemudian membuat *branch* baru, `cabang_B`, tanpa melakukan *commit* terlebih dahulu.

```
// ubah isi fileA.txt
$ echo 'update file A di master' > fileA.txt

// cek status
$ git status
# On branch master
# Changes not staged for commit:
#   modified:   fileA.txt
#
no changes added to commit

// buat cabang baru dan checkout
$ git checkout -b cabang_B

// cek status
$ git status
# On branch cabang_B
# Changes not staged for commit:
#   modified:   fileA.txt
#
```



```
no changes added to commit
```

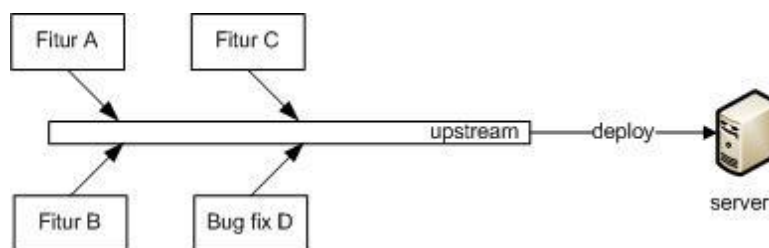
Dalam situasi ini, kita bisa berpindah dari `cabang_B` ke `master` atau sebaliknya tanpa mengalami *error* selama kita belum melakukan *commit* baru di `cabang_B` ataupun `master` yang otomatis akan membuat kedua cabang menjadi *diverging*. Walaupun dalam situasi seperti ini tidak terjadi *error*, sebaiknya kita biasakan untuk melakukan *commit* sebelum membuat *branch* baru.

3.5.3 Upstream Branch

Jika kita mengerjakan proyek di mana file di direktori kerja harus dikirim (*deployed*) ke komputer lain, misalnya sebuah *web server*, kita perlu menentukan *upstream branch* yaitu *branch* dari mana file-file tersebut berasal. Sebagai *best-practice*, kita sebaiknya selalu gunakan *branch master* sebagai *upstream branch*.

Upstream branch disebut juga *production branch*.

Bagaimana dengan *branch* lain? *Branch* selain `master` adalah *branch* sementara yang disebut juga *feature branch*. *Feature branch* kita gunakan untuk menambah fitur atau memperbaiki *bug*. Tujuannya adalah untuk menjaga supaya *upstream branch* berisi *source code* yang sudah diuji, bersih dari kode-kode eksperimen, dan kalau bisa, bebas dari *bug*. Setelah kita selesai bekerja di sebuah *feature branch*, kita lakukan *merging* antara *branch* tersebut dengan `master`, baru kemudian kita lakukan *deployment*.



Gambar 3-12 Upstream Branch & Deployment

Sebagai *best-practice*, sebelum kita melakukan modifikasi terhadap file di *branch* selain `master`, selalu jalankan perintah `git merge master` untuk melakukan sinkronisasi *branch* sama dengan `master`. Tujuannya adalah jika ada konflik, bisa kita selesaikan secepatnya.

3.5.4 Merging

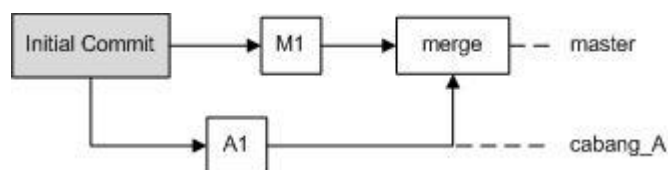
Merging adalah proses penggabungan sebuah *branch* dengan *branch* yang lain. Proses ini dijalankan dengan perintah `git merge` dengan argumen nama *branch* yang akan digabungkan ke dalam *branch* di mana perintah tersebut dijalankan.

Kita coba berpindah ke `master` lalu menggabungkan `cabang_A` ke dalamnya.

```
// pindah ke master
$ git checkout master

// gabungkan cabang_A ke dalam master
$ git merge cabang_A
Merge made by the 'recursive' strategy.
0 files changed
create mode 100644 fileB.txt
```

Dari status dari perintah `merge` atas, kita lihat `fileB.txt` dari `cabang_A` ditambahkan ke dalam `master` sehingga `fileB.txt` sekarang ada di `cabang_A` dan juga di `master`. Data repositori dapat digambarkan sebagai berikut:



Gambar 3-13 Merge cabang A dengan master

Karena proses ini dijalankan di `master`, maka `cabang_A` tidak berubah. `fileB.txt` yang ada di `cabang_A` ditambahkan ke `master` tetapi sebaliknya, `fileC.txt` yang ada di `master` *tidak* ditambahkan ke dalam `cabang_A`.

Proses *merging* adalah proses satu arah dan hanya memperbarui cabang di mana perintah `git merge` dijalankan.

Merge Conflict

Tidak selamanya proses *merging* berjalan mulus tanpa masalah. Kadang kita mengalami *conflict* karena dalam kedua *branch* yang akan kita gabungkan ada satu atau beberapa file yang isinya berbeda di *branch* yang satu dengan yang lain. Hal ini akan lebih sering kita alami ketika kita berkolaborasi dengan orang lain menggunakan *shared repository*.

Sekarang kita coba untuk mengubah isi `fileB.txt` di kedua cabang dengan memasukkan teks "hello cabang A" di `cabang_A` dan "hello master" di `master`. Kemudian kita lakukan *merge* lagi.

```
// pindah ke cabang A
$ git checkout cabang_A
// ubah isi fileB.txt
$ echo "hello cabang A" > fileB.txt
$ git add .
$ git commit -m "A2"
```

```
// pindah ke master
$ git checkout master
// ubah isi fileB.txt
$ echo "hello master" > fileB.txt
$ git add .
$ git commit -m "M1"

// merge cabang_A ke dalam master
$ git merge cabang_A
Auto-merging fileB.txt
CONFLICT (content): Merge conflict in fileB.txt
Automatic merge failed; fix conflicts and then commit the
result.
```

Di sini kita berada dalam situasi yang disebut **merge conflict** karena `fileB.txt` di `cabang_A` dan `fileB.txt` di `master`, memiliki perbedaan isi tepat pada baris yang sama.

Conflict terjadi jika sebuah file dalam dua buah *branch* yang akan digabungkan memiliki perbedaan tepat pada baris yang sama.

Untuk melihat baris dan file yang menyebabkan konflik, kita gunakan perintah `git diff`.

```
$ git diff
diff --cc fileB.txt
index 22a5dfd,093991f..0000000
--- a/fileB.txt
+++ b/fileB.txt
@@@ -1,1 -1,1 +1,5 @@@
++<<<<<< HEAD
+hello master
++=====
+ hello cabang A
++>>>>>> cabang_A
```

Baris yang mengakibatkan konflik ditandai dengan deretan karakter `<<<<<` dan `>>>>>` di dalam file yang mengalami konflik, dalam contoh di atas, `fileB.txt`.

```
<<<<<< HEAD
hello master
=====
hello cabang A
>>>>>> cabang_A
```

Dalam contoh di atas, baris yang bermasalah adalah baris pertama yang berisi teks "hello master" (`master`) dan "hello cabang A" (`cabang_A`).

Untuk menyelesaikan konflik, kita buka `fileB.txt`. Ada dua opsi yang kita miliki yaitu menghapus salah satu baris atau menggabungkan keduanya. Opsi manapun yang kita pilih, kita harus menghapus baris yang berisi tanda `<<<<` dan `>>>>` serta baris yang berisi `=====` lalu

melakukan *commit*.

Kita gabungkan kedua teks tersebut dengan mengubah isi `fileB.txt` menjadi seperti berikut:

```
hello master
hello cabang_A
```

Kemudian kita lakukan *commit*:

```
$ git add fileB.txt
$ git commit -m "fileB.txt merge conflict resolved"
```

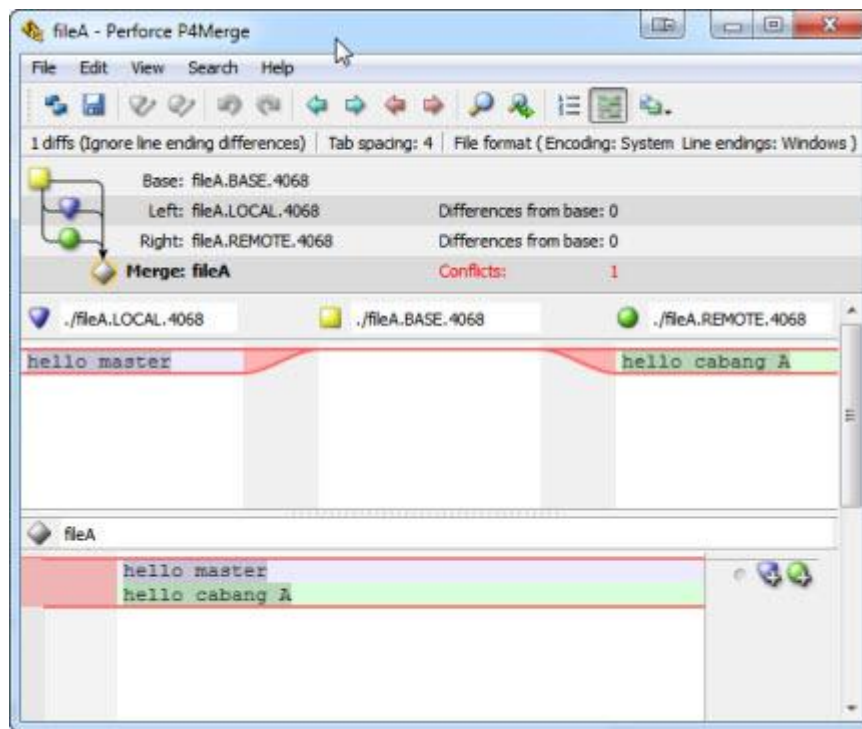
Menggunakan Diff Utility untuk Menyelesaikan Konflik

Kita juga bisa menyelesaikan konflik menggunakan *diff utility* yang telah kita set sebagai *default merge tool*, dengan perintah `git mergetool` seperti berikut.

```
$ git mergetool fileA
Merging:
fileA

Normal merge conflict for 'fileA':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (p4merge):
```

Bagaimana cara menggunakan *diff utility* tidak bisa dibahas di dalam buku ini karena masing-masing memiliki cara dan menu yang sangat berbeda. Silakan Anda baca manual *utility* yang bersangkutan.



Gambar 3-14 P4Merge untuk menyelesaikan konflik

Untuk kembali ke terminal dan melanjutkan proses *merging*, kita simpan hasil *merging* lalu kita tutup program *diff utility* yang kita gunakan.

3.6 Menangani Interupsi dengan Stash

Bagaimana jika kita terpaksa harus berpindah branch tetapi kondisi branch di mana kita berada saat ini belum bisa kita *commit*?

Misalkan kita sedang bekerja membuat sebuah *website* di mana branch `master` adalah *branch* utama yang berisi file untuk kita *upload* ke *web server*. Suatu saat sebuah *bug* baru ditemukan dan kita membuat *branch* bernama `bug_fix` untuk memperbaikinya. Sementara kita bekerja memperbaiki *bug* tersebut, sebuah *bug* lain yang sifatnya *critical* ditemukan di *branch* `master` dan harus secepatnya diperbaiki. Masalahnya, kita tidak bisa berpindah ke `master` karena kedua *branch* dalam keadaan *diverging* dan modifikasi pada *branch* `bug_fix` belum kita *commit*. Kita tidak ingin melakukan *commit* karena pekerjaan kita di *branch* `bug_fix` belum selesai.

Git menyediakan perintah yang sangat berguna untuk situasi semacam ini yaitu `git stash` yang berfungsi untuk:

1. Menyimpan modifikasi yang belum kita *commit* ke dalam sebuah tempat penyimpanan sementara (*stash*)
2. Melakukan *hard-reset* terhadap *working copy* sehingga kita bisa berpindah *branch*.

Setelah kita selesai memperbaiki bug di `master`, kita bisa kembali ke `bug_fix` lalu mengaplikasikan kembali modifikasi yang disimpan di dalam `stash` dengan perintah `git stash apply`.

Sebagai contoh, mari kita buat direktori baru bernama `stash-demo`, buat `fileA`, dan lakukan inisialisasi repositori. Setelah itu kita buat cabang baru bernama `dev`.

```
$ mkdir stash-demo
$ cd stash-demo
// inisialisasi repositori
$ git init
$ touch fileA
$ git add .
$ git commit -m 'initial commit'

// buat branch baru
$ git branch dev
```

Berikutnya, kita modifikasi `fileA` di `master` lalu kita `commit` sehingga kedua `branch` dalam keadaan *diverging*.

```
// ubah isi fileA
$ echo 'teks ini dari master' > fileA
// commit master
$ git commit -a -m 'M1'
```

Kemudian kita berpindah lagi ke branch `dev` dan lakukan modifikasi terhadap `fileA` juga.

```
// pindah ke cabang dev
$ git checkout dev
// ubah isi fileA
$ echo 'Teks ini dari dev' > fileA
```

Tanpa melakukan `commit`, kita coba kembali ke `master`.

```
$ git checkout master
error: Your local changes to the following files would be
overwritten by checkout:
    fileA
Please, commit your changes or stash them before you can
switch branches.
Aborting
```

Kita gunakan perintah `git stash` supaya kita bisa berpindah ke `master`. Setelah kita berpindah ke `master`, kita modifikasi `fileA` sekali lagi.

```
$ git stash
Saved working directory and index state WIP on dev: f46e549
initial commit
HEAD is now at f46e549 initial commit
```

```
$ git checkout master
$ echo 'teks ini dari master v2' > fileA
$ git commit -a -m 'M2'
```

Sekarang kita kembali ke dev lagi dan mengaplikasikan kembali modifikasi yang kita simpan di dalam *stash* sehingga isi `fileA` kembali seperti sebelum kita berpindah ke `master`.

```
$ git checkout dev
$ git stash apply
```

3.7 Tagging

Tag adalah keterangan tambahan yang bisa kita gunakan untuk memberi tanda/label pada sebuah *commit*. Umumnya, *tag* digunakan sebagai penanda revisi-revisi penting seperti *upload/deployment* ke server, *final release*, *beta release*, dan sebagainya.

Sebagai contoh, kita buat direktori baru bernama `tag-demo` lalu inisialisasi repositori git di dalamnya.

```
$ mkdir tag-demo
$ cd tag-demo
$ git init
$ touch fileA
$ git add .
$ git commit -m 'initial commit'
```

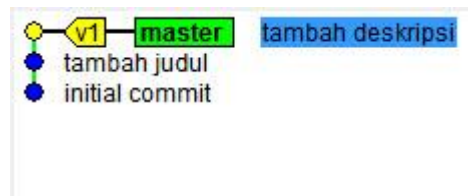
Berikutnya, kita tambahkan teks ke dalam `fileA`.

```
$ echo 'Judul : Tag demo' > fileA
$ git commit -a -m 'tambah judul'
$ echo 'Deskripsi : Ini file untuk demo tagging' >> fileA
$ git commit -a -m 'tambah deskripsi'
```

Kita ingin memberi label 'v1' pada commit terakhir. Jadi kita jalankan perintah `git tag` seperti berikut.

```
$ git tag 'v1'
```

Kalau kita membuka *log* dengan menggunakan `gitk`, kita akan melihat *tag* di samping *commit* terakhir.



Gambar 3-15 Tag

Kita bisa melihat semua tag yang ada di repositori dengan menambahkan argumen `-l` atau `--list` pada perintah `git tag`.

```
$ git tag --list
```

Tag bisa kita gunakan juga untuk membatasi *log* seperti contoh berikut di mana kita akan melihat *log* sampai dengan *commit* yang diberi label (*tag*) `v1` saja.

```
$ git log --tags 'v1'
```

3.8 Contoh Kasus : Website PT Maju Jaya

Bagian ini berisi contoh alur kerja (*workflow*) menggunakan Git dalam proyek sederhana. Kita akan melihat kapan dan bagaimana perintah-perintah Git yang kita pelajari sebelumnya digunakan untuk membantu pekerjaan dan menangani revisi.

Contoh berikut ini sangat sederhana dengan tujuan agar kita bisa fokus pada alur kerja, bukan pada contoh websitenya. Di dunia nyata, untuk proyek sederhana mungkin kita hanya memerlukan perintah `git add` dan `git commit`.

Kita mendapat proyek pembuatan website PT Maju Jaya, sebuah perusahaan konstruksi besar yang berdiri sejak tahun 1900. Perusahaan ini dijalankan oleh orang-orang yang termasuk kelompok *old-school* dan tidak begitu paham tentang Internet. Karena mereka sendiri belum tahu informasi apa yang mereka ingin masukkan ke dalam *website*, kita tidak memiliki spesifikasi yang jelas tentang proyek ini. Dapat kita simpulkan bahwa selama proyek ini berjalan, akan ada banyak revisi yang diminta oleh klien sehingga kita perlu melakukan antisipasi. Salah satu cara untuk mempermudah penanganan banyak revisi adalah dengan menggunakan Git dan selalu bekerja dalam *feature branch*.

3.8.1 Persiapan

Langkah pertama yang kita lakukan adalah membuat dua buah server untuk *demo/testing* dan *production server*. *Demo server* kita gunakan untuk menunjukkan kemajuan proyek agar klien bisa memberi masukan atau mengajukan revisi sebelum file kita upload ke *production server*.

Berikutnya kita membuat sebuah direktori kerja dan menginisialisasi repositori di dalamnya.

```
$ mkdir majujaya  
$ cd majujaya  
$ git init
```

Kemudian kita buat file `index.html` dan `style.css`. Kita buka file `index.html` dan kita isi dengan kerangka dasar halaman web seperti berikut.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<head>  
<title>  
PT Maju Jaya Official Site  
</title>  
  
<link rel="stylesheet" href="style.css" type="text/css" />  
  
</head>  
<body>  
  
  
  
  
  
  
  
  
  
</body>  
</html>
```

Selanjutnya, kita lakukan initial commit.

```
$ git add .  
$ git commit -m 'initial commit'
```

3.8.2 Home Page

Kita ingin `master` menjadi *production branch*, jadi kita buat sebuah *feature branch* bernama `home` untuk membuat *home page* dan berpindah ke cabang tersebut.

```
$ git checkout -b home
```

Kita buka lagi file `index.html` kemudian kita tambahkan konten seperti berikut.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<head>  
<title>  
PT Maju Jaya Official Site  
</title>  
  
<link rel="stylesheet" href="style.css" type="text/css" />  
  
</head>  
<body>
```

```
<h1>Home</h1>

<p>PT Maju Jaya delivers program and construction management
services to clients around the world - from project concept to
completion and commissioning. Led by our Tishman Construction
group - one of the world's leading builders, ranked No. 1 Con-
struction Manager by Indonesian Construction Group - PT Maju
Jaya's construction services team offers clients a single and
comprehensive source of solutions backed by PT Maju Jaya's
combined local knowledge and global expertise.</p>

<p>Our construction services experts provide clients with an
extensive range of pre-construction and construction-related
services and solutions for projects of varying scope, budget,
schedule and complexity.</p>

</body>
</html>
```

Kita ingin menunjukkan isi *home page* kepada klien agar mereka bisa memberi masukan atau permintaan revisi jika ada. Kita lakukan *commit* lalu pindah ke `master` dan lakukan *merging*.

```
$ git commit -a -m 'home content'

// pindah ke master
$ git checkout master

// merge
$ git merge home

// commit master
$ git commit -a -m 'home content'
```

Lalu kita *upload* semua file dari branch `master` ke *testing server* dan kita kirim email ke klien untuk meminta masukan dari mereka.

3.8.3 Profile Page

Sementara kita menunggu respon dari klien yang kadang memakan waktu cukup lama karena kesibukan mereka. Kita lanjutkan dengan membuat fitur lain yaitu *Profile page*.

Kita kembali ke `master`, lalu kita buat *branch* baru bernama `profile`.

```
$ git checkout master
$ git checkout -b profile
```

Di sini kita gunakan file `index.html` sebagai *template* untuk file `profile.html`.

```
$ cp index.html profile.html
```

Kita buka `profile.html` lalu masukkan teks yang menjelaskan sejarah perusahaan.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<head>
<title>
PT Maju Jaya Official Site | Profile
</title>

<link rel="stylesheet" href="style.css" type="text/css" />

</head>
<body>

<p>More than 20 years ago, our founder, Abi Bakir, and a hand-
ful of CV Lumpur Sehat employees shared a dream of creating an
industry-leading firm dedicated to making the world a better
place. On April 6, 2004, PT Maju Jaya evolved to become an in-
dependent firm. While our official founding was in 1990, some
of our predecessor firms had distinguished histories dating
back to the early 1900s.</p>

<p>Since then, more than 40 companies have joined PT Maju Ja-
ya, and in 2007, we became a publicly traded company on the
Jakarta Stock Exchange.</p>

<p>Today, listed on the Forbes as one of Indonesia's largest
companies, PT Maju Jaya's talented employees – including ar-
chitects, engineers, designers, planners, scientists and man-
agement professionals – now serve clients in more than 130
countries around the world. </p>

<p>PT Maju Jaya's technical expertise and creative excellence
come from the rich history of some of the world's finest engi-
neering, environmental, construction management as well as
planning and design companies. From transportation, energy and
water systems to enhancing environments and creating new
buildings and communities, our vision remains constant – to
make the world a better place.
</p>

<p>What sets us apart is our collaborative way of working
globally and delivering locally. A trusted partner to our cli-
ents, we draw together teams of engineers, planners, archi-
tects, landscape architects, environmental specialists, econo-
mists, scientists, consultants, as well as cost construction,
project and program managers dedicated to finding the most in-
novative and appropriate solutions to create, enhance and sus-
tain the world's built, natural and social environments.</p>

</body>
</html>

```

Kita *commit* lalu lakukan *merge* di `master` sebelum kita *upload* file ke *testing server* dan memberitahu klien agar mereka juga memeriksa halaman *Profile*.

```

$ git commit -a -m 'profile content'

// pindah ke master

```

```
$ git checkout master

// merge profile dengan master
$ git merge profile

// commit master
$ git commit -a -m 'profile content'
```

3.8.4 Portfolio

Sampai saat ini kita belum juga menerima masukan dari klien. Namun itu bukan masalah karena kita bisa melanjutkan pekerjaan kita dengan membuat halaman *Portfolio*. Seperti halaman yang lain, kita buat *branch* tersendiri untuk halaman *Portfolio* lalu kita gunakan file `index.html` sebagai *template* untuk `portfolio.html`.

```
// buat branch portfolio
$ git checkout -b portfolio

// buat file portfolio.html dari index.html
$ cp index.html portfolio.html

// tambahkan ke staging index
$ git add portfolio.html
```

Karena isi `portfolio.html` sama dengan `index.html`, kita hapus teks dan kita ganti teks heading (`h1`).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<head>
<title>
PT Maju Jaya Official Site | Portfolio
</title>

<link rel="stylesheet" href="style.css" type="text/css" />

</head>
<body>

<b>Portfolio</b>

</body>
</html>
```

3.8.5 Interupsi : Profile Tidak Memiliki Heading

Sebelum kita selesai mengisi `portfolio.html`, kita menerima email dari klien yang memberitahu bahwa halaman *Profile* tidak memiliki *heading*. Mereka ingin bug ini diperbaiki secepatnya karena website akan diperlihatkan kepada investor.

Kita bisa langsung memperbaikinya di *branch* `portfolio`, tetapi lebih baik kita lakukan di *branch* `profile`. Karena `portfolio.html` belum selesai, kita tidak ingin melakukan *commit*, jadi sebagai gantinya, kita lakukan *stashing* supaya kita bisa berpindah ke `profile` tanpa membawa `portfolio.html` yang setengah jadi ke *branch* tersebut.

```
$ git stash
// pindah ke profile
$ git checkout profile
```

Sebelum kita mengubah `profile.html`, kita pastikan bahwa kode di *branch* `profile` sama dengan yang ada di `master`.

```
$ git merge master
```

Selanjutnya, kita buka `profile.html` dan kita tambahkan *heading* (`h1`).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<head>
<title>
PT Maju Jaya Official Site | Profile
</title>

<link rel="stylesheet" href="style.css" type="text/css" />

</head>
<body>

<b><h1>Profile</h1>

<p>More than 20 years ago, our founder, Abi Bakir, and a hand-
ful of CV Lumpur Sehat employees shared a dream of creating an
industry-leading firm dedicated to making the world a better
place. On April 6, 2004, PT Maju Jaya evolved to become an in-
dependent firm. While our official founding was in 1990, some
of our predecessor firms had distinguished histories dating
back to the early 1900s.</p>

<p>Since then, more than 40 companies have joined PT Maju Ja-
ya, and in 2007, we became a publicly traded company on the
Jakarta Stock Exchange.</p>

<p>Today, listed on the Forbes as one of Indonesia's largest
companies, PT Maju Jaya's talented employees – including ar-
chitects, engineers, designers, planners, scientists and man-
agement professionals – now serve clients in more than 130
countries around the world. </p>
```

```
<p>PT Maju Jaya's technical expertise and creative excellence  
come from the rich history of some of the world's finest engi-  
neering, environmental, construction management as well as  
planning and design companies. From transportation, energy and  
water systems to enhancing environments and creating new  
buildings and communities, our vision remains constant – to  
make the world a better place.  
</p>  
  
<p>What sets us apart is our collaborative way of working  
globally and delivering locally. A trusted partner to our cli-  
ents, we draw together teams of engineers, planners, archi-  
tects, landscape architects, environmental specialists, econo-  
mists, scientists, consultants, as well as cost construction,  
project and program managers dedicated to finding the most in-  
novative and appropriate solutions to create, enhance and sus-  
tain the world's built, natural and social environments.</p>  
  
</body>  
</html>
```

Kita lakukan *commit* lalu *merge* di `master` dan *upload* kembali. Tak lupa kita beritahu klien bahwa website di *testing server* sudah kita perbarui.

```
$ git commit -a -m 'fix : missing heading'  
  
// pindah ke master  
$ git checkout master  
  
// merge  
$ git merge profile  
  
// commit master  
$ git commit -a -m 'fix : missing heading'
```

3.8.6 Kembali ke Portfolio

Kita kembali ke *branch* `portfolio` untuk menyelesaikan `portfolio.html`. Sebelum melanjutkan pekerjaan, kita pastikan kode di `portfolio` sinkron dengan `master`.

```
// pindah ke portfolio  
$ git checkout portfolio  
// merge master ke dalam portfolio  
$ git merge master
```

Selanjutnya kita aktifkan kembali modifikasi yang tadi kita lakukan sebelum kita berpindah branch.

```
$ git stash apply
```

Kita lanjutkan dengan mengisi teks untuk `portfolio.html` misalnya kita tambahkan in-

formasi tentang *copyright*.

```
...  
<p>Copyright - 2012 Some dude </p>  
</body>  
</html>
```

Kemudian kita *commit*.

```
$ git commit -a -m 'portfolio content'
```

Kembali ke `master`, kita merge cabang `portfolio` lalu kita *upload* file ke *demo server*.

```
$ git checkout master  
$ git merge portfolio  
$ git commit -a -m 'portfolio content'
```

3.8.7 Release v1

Klien memberitahu kita bahwa mereka sudah melihat demo website dan ingin publik bisa melihat website baru mereka. Sekarang saatnya mengirim file ke *production server* namun sebelumnya, kita tandai dulu revisi terakhir agar nantinya kita mudah untuk melihat revisi mana yang sudah kita *deploy* ke *production server*.

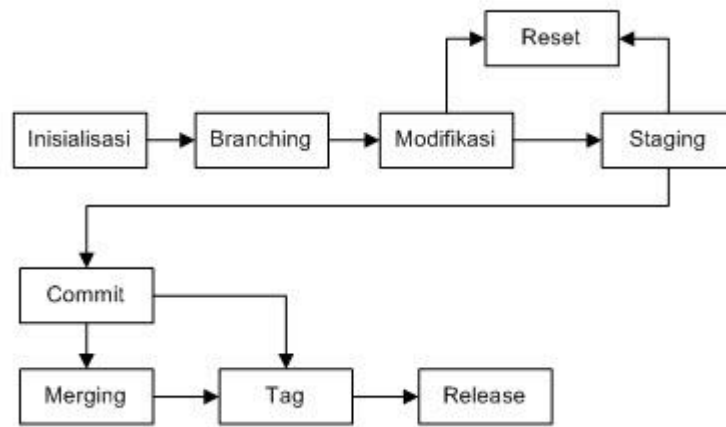
Kita pastikan bahwa kita berada di *upstream branch* (`master`) kemudian kita jalankan perintah `git tag`. Setelah itu baru kita *upload* file dari `master` ke *production server*.

```
$ git checkout master  
$ git tag 'v1, deployed 22/03/2012'
```

Tagging kita lakukan setiap kali kita akan mengirim file ke *production server*.

3.8.8 Kesimpulan

Workflow dengan menggunakan Git sebenarnya tidak terlalu rumit. Kita hanya perlu sedikit beradaptasi dan membiasakan diri. Secara garis besar, tahapan dalam menggunakan Git adalah seperti berikut:



Gambar 3-16 Workflow Dasar

Pada awalnya, *workflow* seperti ini memang terlihat memperumit pekerjaan terutama jika kita mengerjakan proyek-proyek sederhana tetapi dalam jangka panjang, Git dan juga *version control* lainnya akan sangat membantu dalam proses *maintenance* dan pengembangan software atau website yang kita buat. Bayangkan jika website sederhana seperti contoh di atas berkembang menjadi website yang jauh lebih besar dan kompleks dengan puluhan halaman ditambah banyak fitur interaktif seperti flash dan Ajax tentu *maintenance* akan jauh lebih sulit. Apalagi jika ditambah versi khusus untuk *mobile device*.

Workflow yang baik akan sangat membantu dalam membuat produk yang berkualitas. *Maintenance* dan pengembangan menjadi lebih mudah dan pada akhirnya dapat menghemat waktu dan biaya. Jadi mempelajari Git atau *version control* lainnya bisa kita anggap sebagai investasi jangka panjang yang manfaatnya tidak hanya untuk kita tetapi juga untuk klien dan *end user*.

BAB 4. SHARED REPOSITORY

Seperti yang telah kita pelajari dalam bagian introduksi, manfaat terbesar Git dan juga semua VCS adalah dalam memfasilitasi *team work*. Untuk berkolaborasi dengan orang lain, kita harus membuat sebuah *shared repository* yang bisa diakses bersama. Kita bisa saja memberikan akses langsung ke repositori lokal yang kita buat, tetapi hal ini sangat tidak disarankan. Sebaiknya kita membuat sebuah direktori tersendiri, bisa di dalam sebuah *shared folder* di komputer kita atau di komputer lain.

Untuk mempermudah demonstrasi dan pemahaman materi, dalam bab ini kita akan melakukan *simulasi team work* dengan membuat tiga konfigurasi *user* yang berbeda dalam tiga direktori. Jadi kita akan membuat empat direktori yaitu satu direktori sebagai *shared repository* dan tiga direktori untuk repositori lokal masing-masing *user*. Keempat direktori ini kita buat di dalam *parent directory* yang sama.

4.1 Bare Repository

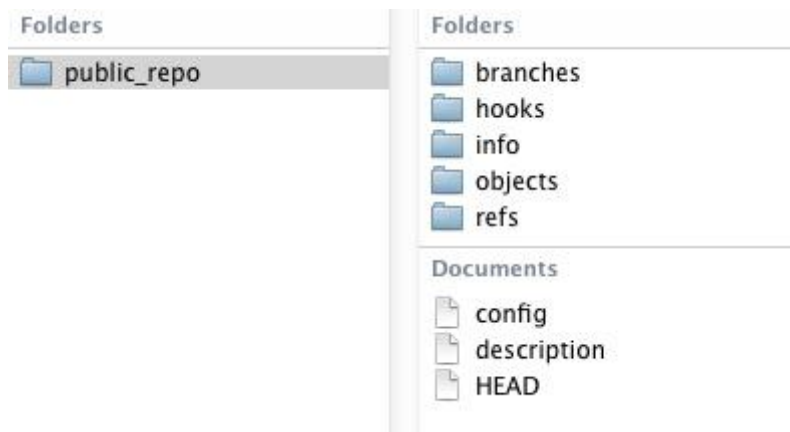
Sebagai *best-practice*, untuk berkolaborasi dengan orang lain, kita berbagi-pakai sebuah *bare repository* yaitu sebuah repositori yang *tidak* memiliki *working directory*. *Bare-repository* bisa kita buat di sebuah server atau sebuah *shared folder* dengan perintah `git init --bare`.

Kita buat direktori `public_repo` dan menjadikannya sebagai sebuah *bare-repository*.

```
$ mkdir public_repo
$ cd public_repo/

//inisialisasi bare repo
$ git init --bare
Initialized empty Git repository in
/Users/boss/Desktop/belajargit/shared-demo/public_repo/
```

Gambar 4-1 menunjukkan isi *bare-repository* yang kita buat. Kita lihat tidak ada subdirektori `.git` di dalamnya dan isi repositori serupa dengan isi subdirektori `.git` dalam contoh-contoh bab yang lalu.



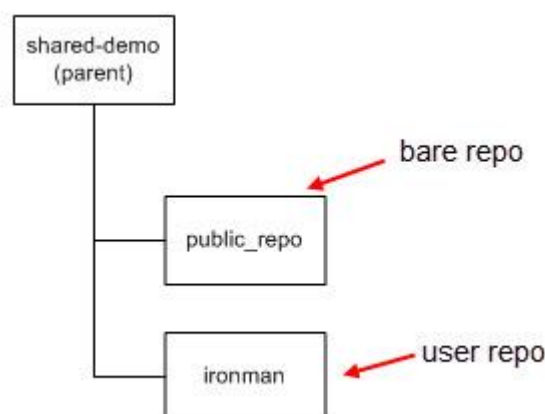
Gambar 4-1 Bare Repository

4.2 Repositori Lokal

Sebuah *shared repository* disebut sebagai *remote repository* walaupun repositori tersebut berada di komputer kita (lokal). Repositori lokal adalah *clone* dari *remote repository* yang dibuat dengan perintah `git clone` di komputer user. Semua *user* bekerja dengan repositori lokal masing-masing. Untuk melakukan sinkronisasi, user menjalankan perintah `git push` untuk *update remote repository* dan `git pull` untuk melakukan *update* terhadap repositori lokal.

Untuk user pertama, kita beri nama *Ironman*, kita buat sebuah direktori di lokasi yang sama dengan direktori `public_repo`.

```
// buat user directory
$ mkdir ironman
```



Gambar 4-2 Struktur direktori shared-demo

```
$ cd ironman

// clone shared repository,
// jangan lupa titik (dot) di belakang
```

```
$ git clone ../public_repo/ .
Cloning into ....
done.
warning: You appear to have cloned an empty repository.
```

Karena kita ingin menggunakan identitas yang berbeda di setiap repositori lokal untuk simulasi, kita set identitas user terlebih dahulu sebelum kita melakukan *commit*.

```
$ git config user.name "Ironman"
$ git config user.email "tonystark@avengers.com"
```

Remote repository yang kita *clone* masih dalam keadaan kosong, jadi sebagai user pertama, *Ironman* harus melakukan *initial commit* dan *push* setelah terlebih dahulu membuat minimal satu buah file di *working directory*.

```
$ touch readme
$ echo 'contoh shared repository' > readme
$ git add .
$ git commit -m 'initial commit'

// push = update remote repository
// origin = nama alias untuk remote repository
// master = default branch di remote repository
$ git push origin master
```

Berikutnya kita buat dua user lain yaitu Hulk dan Thor.

```
// kembali ke parent directory
$ cd ..
// buat directory untuk user Hulk
$ mkdir hulk
$ cd hulk
// clone public repository
$ git clone ../public_repo/ .
// konfigurasi identitas user
$ git config user.name "Hulk"
$ git config user.email "dr.banner@avengers.com"

// kembali ke parent directory
$ cd ..
// buat directory untuk user Thor
$ mkdir thor
$ cd thor
// clone public repository
$ git clone ../public_repo/ .
// konfigurasi identitas user
$ git config user.name "Thor"
$ git config user.email "thor@avengers.com"
```

4.2.1 Origin

Repositori lokal memiliki referensi *remote repository* dalam bentuk nama alias. Secara

default, nama alias sebuah *remote repository* adalah `origin`. Kita bisa menggantinya dengan nama lain yang kita mau dengan cara mengedit langsung file config di dalam direktori `.git`.

Misalnya kita ingin mengganti `origin` dengan `shared`, kita buka file `.git/config` lalu cari baris-baris berikut:

```
[remote "origin"]
    // baris lain tidak ditunjukkan di sini

[remote "master"]
    remote = origin
```

Kita ganti dengan baris berikut:

```
[remote "shared"]
    // baris lain tidak ditunjukkan di sini

[remote "master"]
    remote = shared
```

Selanjutnya `shared` bisa kita gunakan sebagai argumen dalam perintah yang berhubungan dengan *remote repository*. Contoh:

```
$ git push shared master
```

Agar tidak membingungkan, contoh-contoh dalam bab ini tetap menggunakan nama alias *origin*.

4.3 Remote Branch

Seperti halnya repositori lokal, *remote repository* memiliki minimal satu *branch* yaitu `master`. *Branch* ini baru terbentuk setelah salah satu *user* melakukan *push*. Untuk melihat semua *branch* baik lokal maupun *remote*, kita jalankan perintah `git branch` dengan argumen `-a` seperti contoh berikut:

```
$ git branch -a
* master           //local branch
remotes/origin/master //remote branch
```

Untuk membuat *remote branch* baru, pertama kita harus membuat *local branch* terlebih dahulu lalu melakukan *push* ke *remote repository*. Sebagai contoh, Thor membuat *branch* baru dan ingin membaginya dengan Hulk & Ironman sehingga mereka bisa bekerja sama menggunakan *branch* tersebut. Thor membuat *branch* baru lalu melakukan *push* ke `origin`.

```
// pindah ke working directory thor
```

```
$ cd thor
// buat branch
$ git checkout -b thor_branch
// push branch ke remote repository sebagai thor_branch
$ git push -u origin thor_branch
```

Hulk & Ironman kemudian melakukan *pull* untuk sinkronisasi repositori lokal mereka dengan *remote repository*.

```
// pindah ke working directory Hulk
$ cd ../hulk/
// update repository lokal
$ git pull
From /Users/boss/Desktop/belajargit/shared-
demo/hulk/../../public_repo
* [new branch]      thor_branch -> origin/thor_branch
Already up-to-date.

// pindah ke working directory Ironman
$ cd ../ironman

// update repositori lokal
$ git pull
From /Users/boss/Desktop/belajargit/shared-
demo/ironman/../../public_repo
* [new branch]      thor_branch -> origin/thor_branch
Already up-to-date.
```

Sebelum menggunakan `thor_branch`, Hulk dan Ironman harus membuat *local tracking branch* terlebih dahulu lalu berpindah ke *branch* tersebut.

```
$ git branch --track thor_branch origin/thor_branch
Branch thor_branch set up to track remote branch thor_branch
from origin.

$ git checkout thor_branch

$ git branch -a
master
* thor_branch
remotes/origin/master
remotes/origin/thor_branch
```

Sampai di sini, ketiga *user* dapat berkolaborasi dengan bergantian melakukan *push* dan *pull* untuk sinkronisasi *branch*.

4.3.1 Tracking dan Non-tracking Branch

Berdasarkan koneksi dengan *remote repository*, *local branch* dapat dibedakan menjadi dua yaitu *tracking branch* dan *non-tracking branch*. *Tracking branch* adalah *local branch* yang memiliki

referensi ke *remote branch*. Sebaliknya, non-tracking branch tidak memiliki hubungan dengan *remote branch*.

Perintah `git pull` dan `git push` hanya bisa dilakukan di *tracking branch* karena kedua perintah ini membutuhkan referensi ke *remote branch*. Dengan kata lain, kalau kita ingin berbagi-pakai sebuah *branch*, *branch* tersebut harus berupa *tracking branch*.

Untuk membuat sebuah *tracking branch* dari sebuah *remote branch*, kita menjalankan perintah `git branch --track`. Sebagai contoh, kita lanjutkan kolaborasi Thor, Ironman, dan Hulk di atas. Setelah menjalankan perintah `git pull`, Hulk dan Ironman memiliki referensi ke *remote branch* `thor_branch` tetapi mereka tidak bisa menggunakan *branch* tersebut sampai mereka membuat sebuah *tracking branch*. Jadi, langkah selanjutnya adalah membuat *tracking branch* seperti berikut:

```
// buat tracking branch dengan nama thor_branch
// yang berhubungan dengan remote branch : origin/thor_branch
$ git branch --track thor_branch origin/thor_branch
Branch thor_branchA set up to track remote branch thor_branch
from origin.
```

Sebagai *best-practice*, nama *tracking branch* sebaiknya sama dengan nama *remote branch* tapi kita juga bisa menggunakan nama lain misalnya `local_thor_branch` seperti contoh berikut.

```
$ git branch --track local_thor_branch origin/thor_branch
```

4.4 Sinkronisasi

4.4.1 Push dan Pull

Push adalah proses untuk melakukan *update* database *remote repository*. Dalam proses ini, *commit* yang kita lakukan di repositori lokal dikirimkan ke *remote repository* untuk kemudian disimpan dalam database. Setelah proses ini selesai, kondisi *remote repository* sama dengan repositori lokal milik *user* yang melakukan *push*. Proses ini kita jalankan dengan perintah `git push`.

Kebalikan dari proses *push* adalah *pull*, yaitu proses yang dijalankan untuk melakukan *update* atas repositori lokal. Perintah *pull* akan mengunduh data dari *remote repository* dan menggabungkannya (*merge*) dengan repositori lokal.

Misalkan Ironman mengubah isi file `readme` di *branch* `thor_branch` seperti berikut:

```
//pindah ke direktori ironman
$ cd ironman
```

```
// pindah ke branch thor_branch
$ git checkout thor_branch
//update isi readme
$ echo 'teks ini dari ironman' >> readme
```

Agar Hulk dan Thor bisa melakukan sinkronisasi, Ironman harus melakukan *push* kemudian memberitahu Thor dan Hulk agar mereka melakukan *pull* untuk memperbarui *branch* `thor_branch` di repositori masing-masing.

```
//commit
$ git add readme
$ git commit -m 'update teks ironman'

//push ke origin/thor_branch
$ git push
```

Thor dan Hulk kemudian melakukan *pull* untuk memperbarui *branch* `thor_branch` di repositori mereka.

```
// pindah ke branch thor_branch di repositori thor
$ git checkout thor_branch

// pull thor_branch dari shared repository
$ git pull
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /Users/boss/Desktop/belajargit/shared-
demo/thor/./public_repo
   8e2b1bd..64e75b4  thor_branch -> origin/thor_branch
Updating 8e2b1bd..64e75b4
Fast-forward
   readme |      1 +
   1 files changed, 1 insertions(+), 0 deletions(-)

// lihat isi file readme
$ cat readme
contoh shared repository
teks ini dari ironman
```

4.4.2 Non Fast-forward dan Merging Conflict

Misalkan Hulk mengubah isi file `readme` lalu melakukan *push*.

```
// HULK
$ echo 'teks ini dari hulk' >> readme
$ git commit -a -m 'update teks hulk'
$ git push
```

Sementara itu, Thor juga mengubah file `readme` dan melakukan *push*.

```
// THOR
$ echo 'teks ini dari thor' >> readme
$ git commit -a -m 'update teks thor'

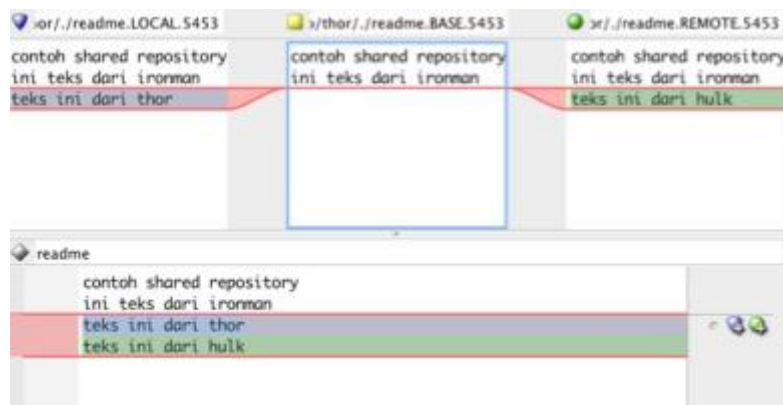
$ git push
To /Users/boss/Desktop/belajargit/shared-
demo/thor/../../public_repo/
! [rejected] thor_branch -> thor_branch (non-fast-forward)
```

Thor gagal melakukan push karena `thor_branch` di *remote repository* memiliki revisi yang lebih baru sebagai hasil *push* yang dilakukan oleh Hulk. Ia harus melakukan *pull* terlebih dahulu untuk mensinkronkan `thor_branch` miliknya dengan *remote branch*.

```
$ git pull
Auto-merging readme
CONFLICT (content): Merge conflict in readme
Automatic merge failed; fix conflicts and then commit the re-
sult.
```

Di sini timbul masalah baru yaitu *merging conflict*¹ jadi Thor harus menyelesaikan konflik ini terlebih dahulu. Ia lebih suka menggunakan GUI untuk menyelesaikan konflik, jadi ia menggunakan *merge utility*².

```
$ git mergetool
```



Gambar 4-3 Merging Conflict

Setelah itu Thor melakukan *commit* lalu *push* dan tak lupa memberitahu Ironman dan Hulk untuk melakukan *pull*.

```
$ git commit -a -m 'merge readme'
```

¹ Lihat topik Merge Conflict di Bab 3

² Lihat topik Diff Utility di Bab 2


```
$ git push
```

Karena `thor_branch` telah diperbarui lagi, Ironman dan Hulk melakukan *pull* untuk sinkronisasi.

```
$ git checkout thor_branch
$ git pull
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
From /Users/boss/Desktop/belajargit/shared-
demo/ironman/../../public_repo
    e70f329..e58a466  thor_branch -> origin/thor_branch
Updating e70f329..e58a466
Fast-forward
  readme |      2 ++
  1 files changed, 2 insertions(+), 0 deletions(-)

// lihat isi readme
$ cat readme
contoh shared repository
ini teks dari ironman
teks ini dari thor
teks ini dari hulk
```

Berikut ini log repositori sampai update terakhir yang dilakukan oleh Thor.

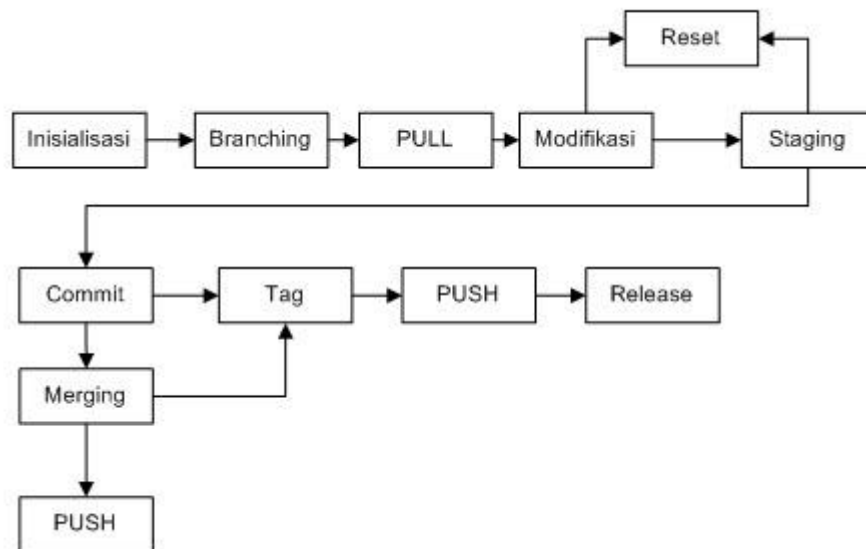


Gambar 4-4 Log Repositori

4.5 Kesimpulan

Pada dasarnya, berkolaborasi dengan orang lain menggunakan *shared repository* tidak berbeda jauh dengan bekerja sendiri menggunakan *private repository*. Hanya ada dua perintah baru yang harus kita pahami yaitu `git pull` dan `git push`. Sementara perintah-perintah yang lain tidak ada bedanya dalam konteks kolaborasi maupun bekerja sendiri.

Push kita lakukan untuk memperbarui *remote repository* supaya rekan kita bisa melakukan sinkronisasi. Sementara *pull* kita lakukan pada saat kita ingin melakukan sinkronisasi dengan *remote repository*.



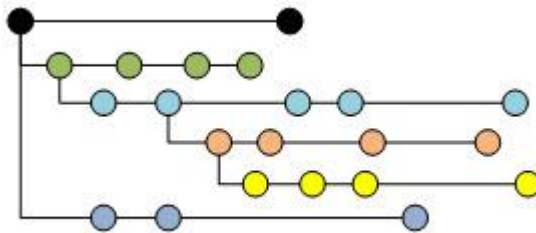
Gambar 4-5 Workflow untuk Kolaborasi

Gambar di atas menunjukkan alur kerja (*workflow*) yang umum digunakan dalam berkolaborasi.

BAB 5. REBASE

Rebase adalah fitur tingkat lanjut (advanced) yang bisa kita gunakan untuk menyederhanakan *branch* khususnya jika repositori kita memiliki banyak *branch* dan *child-branch*. *Rebase* dapat digunakan sebagai pengganti atau persiapan *merging* di mana semua *commit* dari sebuah *branch*, misalnya `branch_A`, diubah menjadi sederetan *commit* di *parent branch*, misalnya, *branch dev* sehingga seakan-akan semua *commit* tersebut dilakukan di *branch dev*.

Umumnya kita cukup melakukan *merging* namun dalam beberapa kasus di mana *branch* memiliki sebuah *child-branch* yang juga memiliki *child-branch* seperti **Gambar 5-1**, proses *merging* menjadi lebih sulit. Dalam situasi seperti ini, perlu kita pertimbangkan untuk *rebase* sebelum melakukan *merge*.



Gambar 5-1 Complex Branches

Secara teknis kita bisa melakukan *rebase* terhadap dua *branch* walaupun tidak ada hubungan *parent-child* antara keduanya, namun hal ini sebaiknya dihindari.

Rebase berpotensi menghilangkan (*overwrite*) modifikasi yang telah kita lakukan di dalam sebuah *branch* jadi kita harus sangat berhati-hati dalam menggunakannya.

Jangan lakukan *rebase* dengan melibatkan *branch* yang kita *share* dengan orang lain.

Untuk mencoba *rebase*, kita buat sebuah direktori baru bernama `rebase-demo` lalu kita inisialisasi repositori di dalamnya.

```
$ mkdir rebase-demo
$ cd rebase-demo
$ git init
$ git touch fileA
```

```
$ git add .
$ git commit -m 'initial commit'
```

Berikutnya kita buat *branch* `branch_A` lalu membuat `master` dan `branch_A` *diverging* seperti berikut.

```
$ git checkout -b branch_A
$ touch fileB
$ git add fileB
$ git commit -m 'buat fileB @branchA'
$ git add fileC
$ git add fileC
$ git commit -m 'buat fileC @branchA'

$ git checkout master
$ touch fileD
$ git add fileD
$ git commit -m 'buat fileD @master'

// rebase
$ git rebase branch_A
```



Gambar 5-2 Diverging Branch sebelum Rebase

Gambar 5-2 menunjukkan kondisi repositori saat ini di mana `master` dan `branch_A` dalam kondisi *diverging*. Di sini kita memiliki dua opsi yaitu *merge* atau *rebase*. Kalau kita memilih *merge*, repositori menjadi seperti **Gambar 5-3** sementara hasil proses *rebase* ditunjukkan dalam **Gambar 5-4**.



Gambar 5-3 Diverging Branch sesudah Merge

Untuk melakukan *rebase*, kita jalankan perintah `git rebase` dengan argumen nama *branch* yang akan kita gabungkan ke dalam *branch* di mana kita berada saat ini.

```
$ git rebase branch_A
```



Gambar 5-4 Diverging Branch sesudah Rebase

Kita lihat dalam gambar di atas, setelah kita lakukan rebase, kondisi histori dari branch master tampak seakan-akan branching tidak pernah kita lakukan.

BAB 6. GIT HOSTING

Dalam bab ini kita akan mempelajari tentang Git hosting dengan menggunakan jasa pihak ketiga (*provider*) dan *hosting* sendiri (*self-hosting*). *Provider* yang kita bahas di sini adalah Bitbucket (www.bitbucket.org) dan Github (www.github.com) karena keduanya menyediakan hosting gratis dan relatif lebih populer dibanding *provider* yang lain.

Dalam bagian pertama bab ini, kita akan membahas SSH karena semua *git provider* mengharuskan kita mendaftarkan *SSH key* sebelum kita bisa mengakses repositori. Selain itu, kalau kita ingin membuat *shared repository* di server kita sendiri tanpa menggunakan jasa *provider*, SSH adalah cara yang paling mudah.

6.1 SSH Key

SSH (*Secure Shell*) adalah protokol jaringan yang menyediakan jalur komunikasi yang aman dan terenkripsi untuk eksekusi perintah di *remote computer*. Akses melalui SSH disediakan oleh server dengan sistem operasi Linux. Informasi lebih lanjut tentang SSH bisa kita baca di alamat berikut:

http://en.wikipedia.org/wiki/Secure_Shell

SSH Key digunakan untuk otentikasi identitas *user*. Dengan menggunakan SSH key, kita tidak perlu memasukkan *username* dan *password* setiap kali kita melakukan koneksi dengan server. Satu set SSH key terdiri atas sepasang key yaitu *private* dan *public key*. Kedua *key* ini dibuat secara otomatis oleh program yang disebut *key generator*. Kita hanya bisa mengakses server yang mengenali *public key* kita. *Private key* akan digunakan oleh SSH client di komputer kita. Jika *private* dan *public key* tidak cocok, koneksi akan ditolak.

6.1.1 Windows

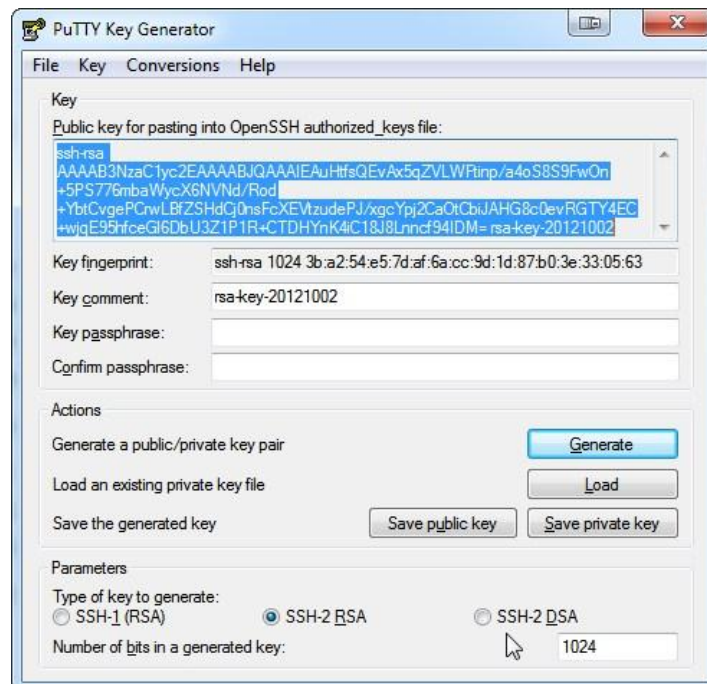
Untuk membuat SSH key di Windows, kita bisa menggunakan utiliti gratis bernama Putty Gen yang bisa kita dapatkan di alamat berikut:

<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>

Di website tersebut pilih *putty.zip* atau *Windows Installer* karena kita juga membutuhkan *putty.exe*, sebuah *SSH client* yang kita gunakan untuk mengakses server linux melalui *SSH*, dan *pagent.exe*, sebuah program yang mempermudah kita mengakses server tanpa berulang kali memasukkan *username* dan *password*.

Berikut ini langkah-langkah untuk membuat *ssh key* menggunakan *puttygen*.

1. Jalankan program *puttygen.exe*
2. Klik **Generate**
3. Simpan *public key* sebagai sebuah file dengan ekstensi *.pub* dan *private key* sebagai file dengan ekstensi *.ppk*



Gambar 6-1 Generate ssh key

6.1.2 Linux & OSX

Pengguna linux dan Mac OSX tidak membutuhkan program lain karena *SSH key* bisa dibuat melalui terminal dengan perintah berikut :

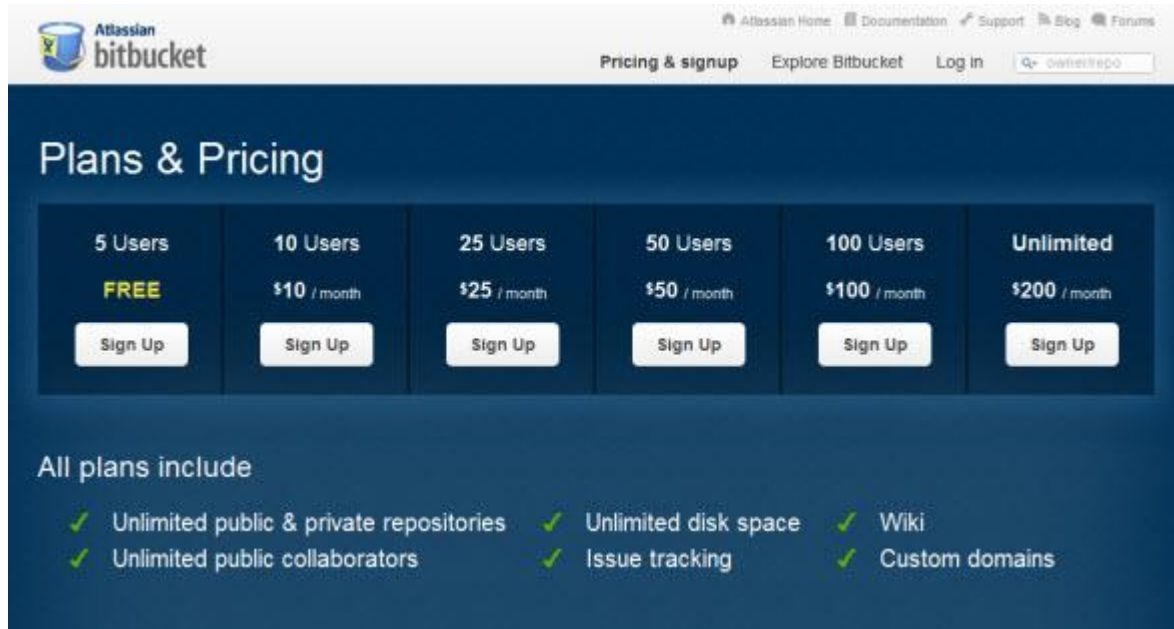
```
$ ssh-keygen -t rsa
```

SSH key yang dihasilkan, secara *default* disimpan di dalam direktori berikut:

- */home/<nama user>/.ssh/* (Linux)
- */Users/<nama user>/.ssh/* (Mac)

6.2 Bitbucket

Bitbucket menyediakan jasa hosting version control gratis dan berbayar dengan harga yang relatif murah. Kelebihan bitbucket dibanding hosting lain adalah kita membuat *private repository* dalam jumlah tak terbatas bahkan jika kita memilih paket gratis.



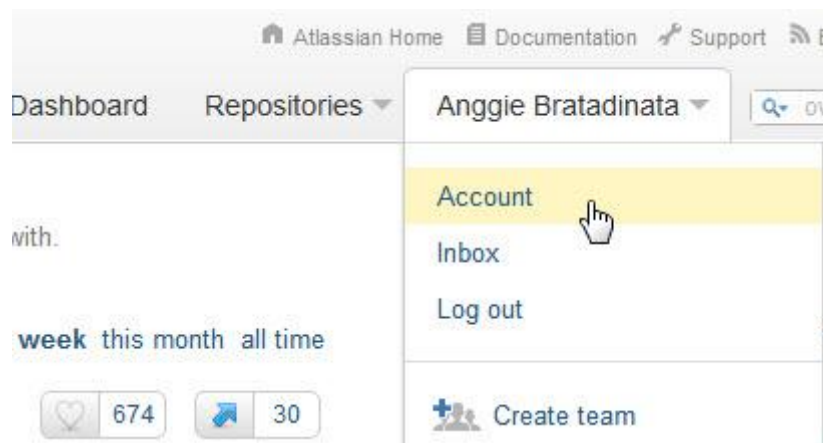
Gambar 6-2 Bitbucket Pricing

Membuka akun di Bitbucket sangat mudah. Kita tinggal pilih *plan* yang kita inginkan lalu klik tombol *Sign Up* dan ikuti prosedurnya.

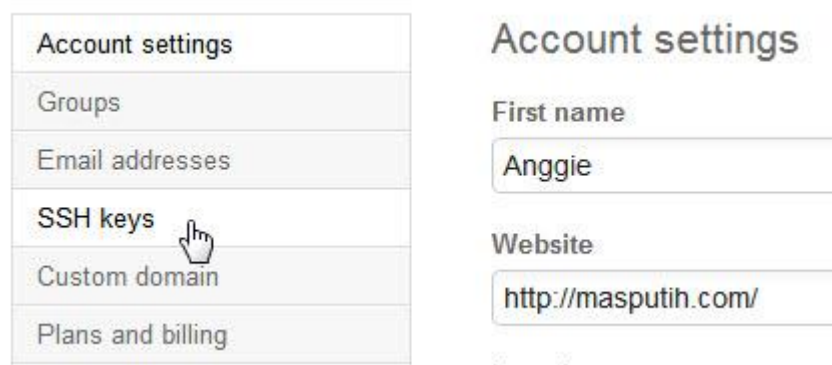
6.2.1 Registrasi SSH Key

Setelah kita memiliki akun di Bitbucket, langkah selanjutnya adalah mendaftarkan SSH key agar kita dapat mengakses repositori. Berikut ini prosedur untuk mendaftarkan SSH key.

1. Buka halaman Account (**Gambar 6-3**)
2. Pilih menu SSH Keys (**Gambar 6-4**)
3. Buka *public key* (file dengan ekstensi .pub) dengan editor teks, lalu kopi isinya ke dalam input field yang disediakan.



Gambar 6-3 Bitbucket Account



Gambar 6-4 SSH menu di Bitbucket

SSH keys

Use SSH and no longer have to authenticate when committing to Bitbucket.
[Learn more about generating a SSH key.](#)

| | |
|---------|------|
| macbook | edit |
| pc | edit |

Add a new key

Label

SSH Key

Paste your key here...

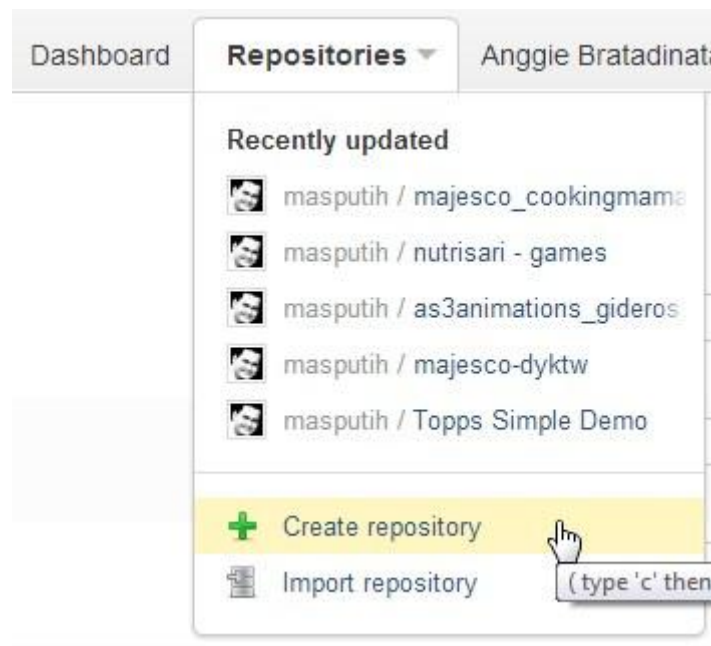
Copy your key to your clipboard

Add key

Gambar 6-5 Form untuk mengatur SSH key di Bitbucket

6.2.2 Membuat Repositori

Untuk membuat repositori, kita klik tab [Repositories](#) lalu pilih menu [Create Repository](#) (**Gambar 6-6**). Kita akan dibawa ke halaman repositori yang bersangkutan. URL yang bisa kita gunakan untuk melakukan clone ditampilkan di bagian atas (**Gambar 6-7**).



Gambar 6-6 Menu Repositori



Gambar 6-7 Clone URL

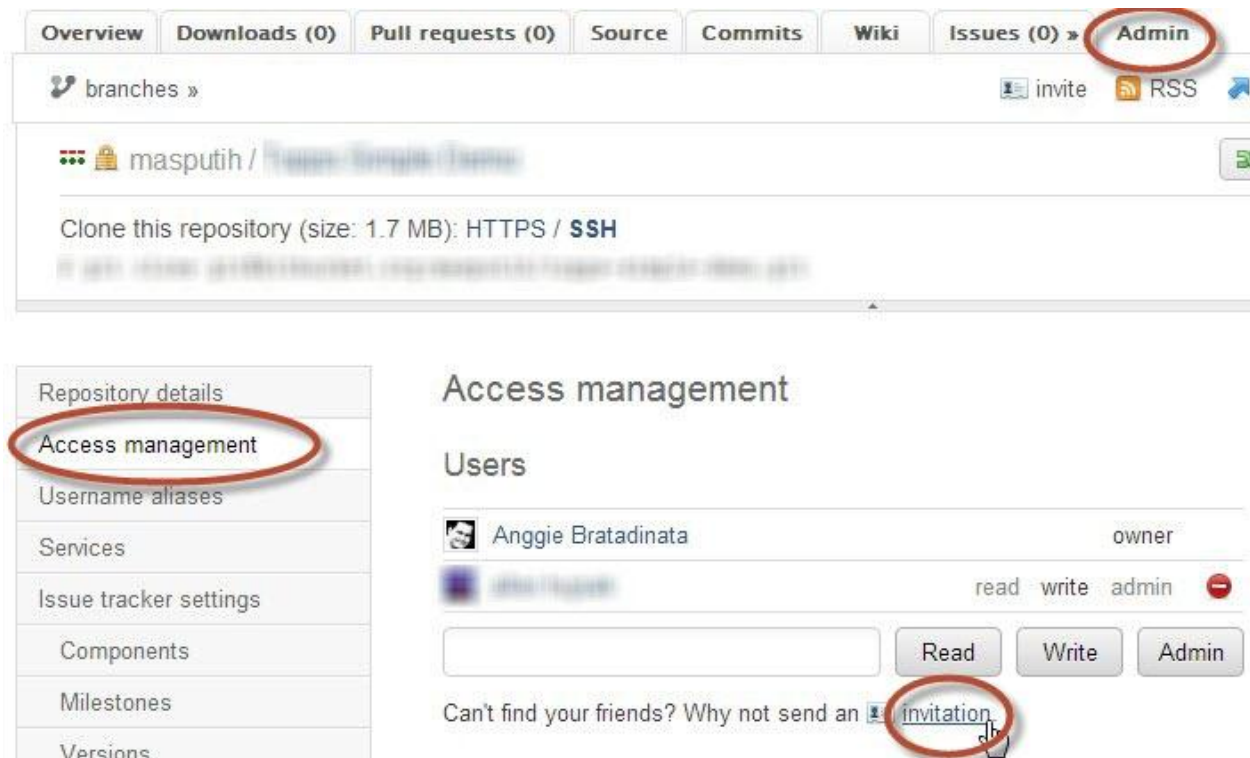
6.2.3 Berbagi-pakai Repositori

Untuk berbagi-pakai repositori dengan orang lain, kita buka halaman [Admin](#) lalu pilih [Access Management](#) dari menu di sebelah kiri (**Gambar 6-8**). Jika orang tersebut sudah memiliki akun di Bitbucket, kita bisa mengetikkan namanya di dalam input field. Jika tidak, kita perlu mengirimkan "undangan" dengan mengklik link [invitation](#).

Ada tiga macam hak akses yang bisa kita berikan kepada orang lain yaitu *Read*, *Write*, atau *Admin*.

- **Read** : hanya bisa melakukan pull.

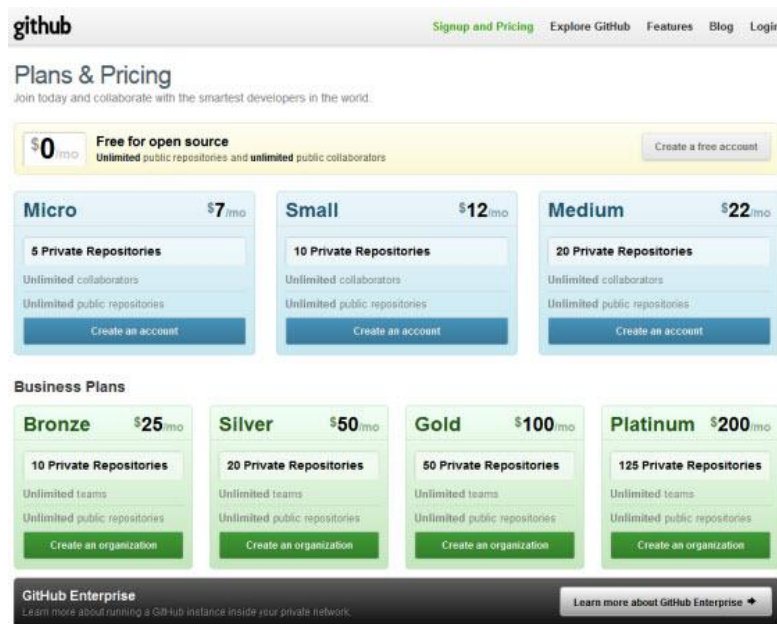
- **Write** : bisa melakukan pull dan push
- **Admin** : sama dengan Write dan juga bisa melakukan tindakan administrasi (menambah user, mengubah hak akses, dll)



Gambar 6-8 Admin Page

6.3 Github

Github adalah provider yang paling populer khususnya untuk proyek-proyek opensource. Provider ini memiliki andil besar dalam mempopulerkan Git. Sama seperti Bitbucket, kita bisa memilih paket yang kita inginkan namun tidak seperti Bitbucket, paket gratis dari Github hanya memperbolehkan kita membuat satu private repository.



Gambar 6-9 Github Pricing

Registrasi juga sangat mudah. Kita tinggal mengikuti prosedur yang mirip dengan registrasi akun di Bitbucket. Kita tinggal memilih *plan* yang kita inginkan lalu klik link yang tersedia kemudian kita tinggal ikuti prosedurnya (isi form dan lain-lain).

Plans & Pricing

Join today and collaborate with the smartest developers in the world.



Gambar 6-10 Github Plans

6.3.1 Registrasi SSH Key

Untuk melakukan registrasi SSH key, kita buka [Account Settings](#) dengan mengklik ikon di sebelah nama kita.



Gambar 6-11 Account Settings

Di halaman *Account Settings*, kita pilih menu **SSH Keys** dan klik tombol **Add SSH Key**. Kemudian kita isi form yang disediakan (langkah-langkahnya sama dengan menambah key di Bitbucket)



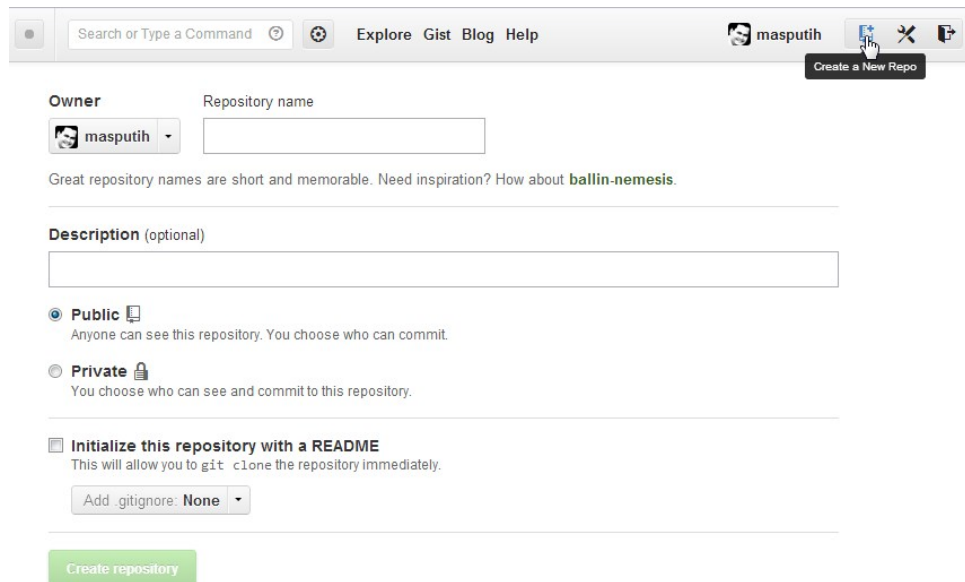
Gambar 6-12 SSH Keys

 The image shows a form titled 'Add an SSH Key'. It contains two input fields: 'Title' (a single-line text box) and 'Key' (a large multi-line text area). At the bottom left of the form is a green button labeled 'Add key'.

Gambar 6-13 Form untuk menambah SSH Key

6.3.2 Membuat Repositori

Untuk membuat repositori, kita klik ikon di sebelah *username* kita lalu isi form dan klik tombol **Create Repository**. Setelah itu kita ikuti petunjuk selanjutnya untuk melakukan clone di komputer kita.



The screenshot shows the GitHub interface for creating a new repository. At the top, there's a navigation bar with a search bar, icons for repository, settings, and links to Explore, Gist, Blog, and Help. The user's profile 'masputih' is on the right. Below the navigation bar, the 'Create a New Repo' button is highlighted. The main form has two sections: 'Owner' with a dropdown menu showing 'masputih', and 'Repository name' with an empty text box. A tip suggests repository names should be short and memorable, with an example 'ballin-nemesis'. The 'Description (optional)' section has a large text area. Under 'Visibility', the 'Public' option is selected, with a sub-note 'Anyone can see this repository. You choose who can commit.' The 'Private' option is also visible with a sub-note 'You choose who can see and commit to this repository.' There's a checkbox for 'Initialize this repository with a README' and a note 'This will allow you to git clone the repository immediately.' Below this is a dropdown for 'Add .gitignore: None'. At the bottom is a green 'Create repository' button.

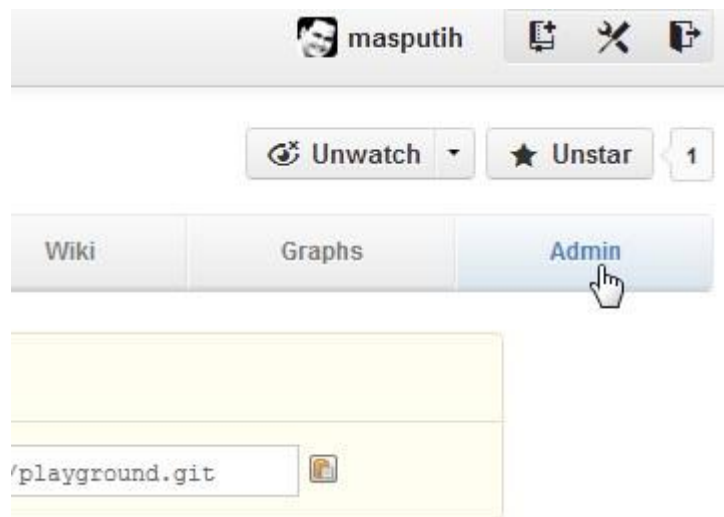
Gambar 6-14 Form untuk repositori baru

6.3.3 Berbagi-pakai Repositori

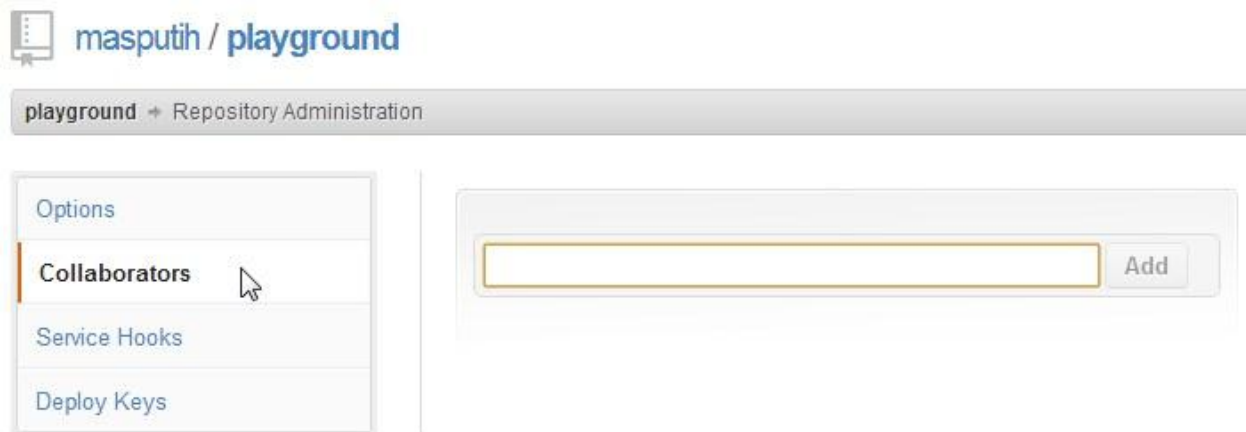
Untuk berbagi-pakai repositori, pertama kita buka *homepage* akun kita dengan mengklik *username* kita lalu pilih repositori yang ingin kita bagi (**Gambar 6-15**). Berikutnya kita buka halaman admin dari repositori yang bersangkutan (**Gambar 6-16**). Dari menu di sebelah kiri (**Gambar 6-17**), kita pilih Collaborator lalu ketikkan nama orang yang kita maksud (orang tersebut harus sudah memiliki akun di Github)



Gambar 6-15 Membuka daftar repositori



Gambar 6-16 Membuka halaman Admin



Gambar 6-17 Menambah User (kolaborator)

BAB 7. PENUTUP

Saat ini saya harap Anda sudah paham tentang dasar-dasar Git. Tidak terlalu sulit, bukan? Langkah selanjutnya adalah menerapkan apa yang Anda pelajari dari buku ini dalam pekerjaan Anda. Saya yakin banyak manfaat yang akan Anda peroleh dengan menggunakan Git, khususnya jika Anda bekerja dalam tim bersama orang lain.

Anda bisa mengirim koreksi dan pertanyaan melalui email ke mas_ab@masputih.com.

7.1 Referensi

- *Getting Good with Git*, Andrew Burgess, Rockablepress
- *Git Internals*, Scott Chacon, PeepCode Press