

SURVEY METHODOLOGY

SURVEY METHODOLOGY

This is the Subtitle

Robert M. Groves

Universitat de les Illes Balears

Floyd J. Fowler, Jr.

University of New Mexico



Kreatif Industri Nusantara

Survey Methodology / Robert M. Groves . . . [et al.].

p. cm.—(Wiley series in survey methodology)

“Wiley-Interscience.”

Includes bibliographical references and index.

ISBN 0-471-48348-6 (pbk.)

1. Surveys—Methodology. 2. Social sciences—Research—Statistical methods. I. Groves, Robert M. II. Series.

HA31.2.S873 2007

001.4'33—dc22

2004044064

Penulis:

Rolly Maulana Awangga

ISBN : 978-602-53897-0-2

Editor:

M. Yusril Helmi Setyawan

Penyunting:

Syafril Fachrie Pane

Khaera Tunnisa

Diana Asri Wijayanti

Desain sampul dan Tata letak:

Deza Martha Akbar

Penerbit:

Kreatif Industri Nusantara

Redaksi:

Jl. Ligar Nyawang No. 2

Bandung 40191

Tel. 022 2045-8529

Email : awangga@kreatif.co.id

Distributor:

Informatics Research Center

Jl. Sariasih No. 54

Bandung 40151

Email : irc@poltekpos.ac.id

Cetakan Pertama, 2007

Hak cipta dilindungi undang-undang

Dilarang memperbanyak karya tulis ini dalam bentuk dan dengan cara
apapun tanpa ijin tertulis dari penerbit

To my parents

CONTRIBUTORS

MASAYKI ABE, Fujitsu Laboratories Ltd., Fujitsu Limited, Atsugi, Japan

L. A. AKERS, Center for Solid State Electronics Research, Arizona State University,
Tempe, Arizona

G. H. BERNSTEIN, Department of Electrical and Computer Engineering, University
of Notre Dame, Notre Dame, South Bend, Indiana; formerly of Center for Solid
State Electronics Research, Arizona State University, Tempe, Arizona

CONTENTS IN BRIEF

1	Home	1
2	Perintah Dasar Bash	7
3	Environment Setup	9
4	Life Cycle	23
5	Create Operation	29
6	Clone Operation	39
7	Perform Changes	63
8	Review Changes	77
9	Commit Changes	91
10	Push Operation	111
11	Update Operation	129
12	Stash Operation	145
13	Move Operation	159
14	Rename Operation	173

DAFTAR ISI

Daftar Gambar	xiii
Daftar Tabel	xv
Foreword	xvii
Kata Pengantar	xix
Acknowledgments	xxi
Acronyms	xxiii
Glossary	xxv
List of Symbols	xxvii
Introduction	xxix
<i>Catherine Clark, PhD.</i>	
Daftar Pustaka	xxix
1 Home	1
1.1 Latihan	1
1.1.1 Konfigurasi Key	1
1.1.2 Fork Repositori	4
	ix

1.1.3	Navigasi direktori dengan Git Bash	4
1.1.4	Sinkronisasi dengan Repo Utama	4
1.1.5	Bekerja dengan Git Bash	4
2	Perintah Dasar Bash	7
2.1	Perintah Navigasi	7
3	Environment Setup	9
3.1	Environment Setup	9
3.1.1	Port Mac	10
4	Life Cycle	23
4.1	Life Cycle	23
4.1.1	SQLite Database	24
5	Create Operation	29
5.1	Create Oprator	29
5.1.1	Dasar-Dasar Git	29
5.1.2	Penjelasan git untuk menambah data	32
5.1.3	Setup Git Untuk Pertama Kalinya	35
5.1.4	Cek Status dari Berkas	36
5.1.5	Apakah Branch Itu	37
6	Clone Operation	39
6.1	Clone Opration	40
6.1.1	Memantau berkas baru	42
6.1.2	Mengabaikan berkas	45
6.1.3	Comit perubahan atau yang telah di edit	53
6.1.4	Menghapus berkas	57
6.1.5	Memindahkan berkas	60
7	Perform Changes	63
7.1	Dasar Git	63
7.1.1	Apa itu GIT?	71
7.2	Repository	71
8	Review Changes	77
8.1	Dasar Git	77

8.1.1	Apa itu GIT?	83
8.2	Repository	84
9	Commit Cahnges	91
9.1	Pengertian	92
9.1.1	Commit Changes	92
9.2	Berikut adalah langkah-langkah untuk menjalankan perintah	94
9.3	Tambahkan secara otomatis	99
10	Push Operation	111
10.1	Pengertian	112
10.1.1	Push Operation	112
10.2	Alur kerja Git yang paling umum adalah sebagai berikut	114
10.2.1	Pengertian Lanjutan	115
10.3	Welcome Contributors	116
10.4	Material Tabs	116
11	Update Operation	129
11.1	Dasar Kontrol Versi	131
11.1.1	Git	132
12	Stash Operation	145
12.1	Penjelasan	145
12.1.1	Membatalkan Apapun	150
12.2	Sistem Kontrol Versi	154
12.2.1	Distributed Sistem Kontrol Versi	156
12.2.2	Keuntungan dari Git	156
13	Move Operation	159
13.1	Move Operation	159
13.1.1	Semua Operasi Dilakukan Secara Lokal	161
13.1.2	Git Memiliki Integritas	162
13.2	Mendapatkan File untuk Pindah	172
14	Rename Operation	173
14.1	Rename Operation	173
14.1.1	Intalasi Git	175
14.1.2	Inisiasi	176

14.1.3	Penambahan File ke Repository	177
14.1.4	Mengubah Isi File	178
14.1.5	Membaca File Lama, dan Menjalankan Mesin Waktu	184
14.2	Penyelesaian Masalah	186
Daftar Pustaka		187

DAFTAR GAMBAR

1.1	Hasil Luaran 2	2
1.2	Hasil Luaran 2	2
1.3	Setting	3
1.4	New SSH key	3
1.5	Direktori	5
3.1	git setup envirotnment	9
4.1	Life Cycle	23
5.1	oprasi	33
6.1	Logo	39
7.1	File Status Lifecycle	67
8.1	Stages Of Git	81
9.1	Commit Changes	91
10.1	Push Operation	111

11.1	learn git	130
12.1	Version Control System	155
13.1	PerpindahanFile	172
14.1	Penamaan	186

DAFTAR TABEL

11.1	Alur kerja dasar git	140
12.1	Fungsi VCS	155

FOREWORD

This is the foreword to the book.

KATA PENGANTAR

This is an example preface. This is an example preface. This is an example preface.
This is an example preface.

R. K. WATTS

Durham, North Carolina
September, 2007

ACKNOWLEDGMENTS

From Dr. Jay Young, consultant from Silver Spring, Maryland, I received the initial push to even consider writing this book. Jay was a constant “peer reader” and very welcome advisor during this year-long process.

To all these wonderful people I owe a deep sense of gratitude especially now that this project has been completed.

G. T. S.

ACRONYMS

ACGIH	American Conference of Governmental Industrial Hygienists
AEC	Atomic Energy Commission
OSHA	Occupational Health and Safety Commission
SAMA	Scientific Apparatus Makers Association

GLOSSARY

NormGibbs	Draw a sample from a posterior distribution of data with an unknown mean and variance using Gibbs sampling.
pNull	Test a one sided hypothesis from a numerically specified posterior CDF or from a sample from the posterior
sintegral	A numerical integration using Simpson's rule

SYMBOLS

- A Amplitude
- $\&$ Propositional logic symbol
- a Filter Coefficient

- \mathcal{B} Number of Beats

INTRODUCTION

CATHERINE CLARK, PHD.

Harvard School of Public Health
Boston, MA, USA

The era of modern began in 1958 with the invention of the integrated circuit by J. S. Kilby of Texas Instruments. His first chip is shown in Fig. I. For comparison, Fig. I.2 shows a modern microprocessor chip,.

This is the introduction. This is the introduction. This is the introduction. This is the introduction. This is the introduction. This is the introduction.

$$ABCDE\mathcal{F}\alpha\beta\Gamma\Delta\sum_{def}^{abc} \tag{I.1}$$

Daftar Pustaka

1. J. S. Kilby, “Invention of the Integrated Circuit,” *IEEE Trans. Electron Devices*, **ED-23**, 648 (1976).
2. R. W. Hamming, *Numerical Methods for Scientists and Engineers*, Chapter N-1, McGraw-Hill, New York, 1962.
3. J. Lee, K. Mayaram, and C. Hu, “A Theoretical Study of Gate/Drain Offset in LDD MOSFETs” *IEEE Electron Device Lett.*, **EDL-7**(3). 152 (1986).

BAB 1

HOME

1.1 Latihan

Bisa karena biasa, rumit karena tak dicicil. Maka pemakaian git adalah masalah kebiasaan. Oleh karena itu, dalam pembukaan awal buku ini justru kita tidak dulu masuk kepada teori git tapi langsung praktek. setelah berkali-kali praktek maka akan timbul beberapa pertanyaan mengenai fungsi dari perintah-perintah git yang bisa dibaca pada bagian selanjutnya.

Skenarionya adalah kita akan melakukan kontribusi terhadap sebuah repo *Open Source* di GitHub. Jadi latihan ini adalah latihan bagaimana ikut berkontribusi kepada repo *Open Source* yang ada di Github yang selama ini digunakan oleh para relawan kode untuk bersama membangun kode program berbasis *Open Source*.

1.1.1 Konfigurasi Key

Pertama kita masuk kepada tahapan *setting* atau konfigurasi *key* untuk mengakses semua repo dari *profile* kita dari *git bash*:

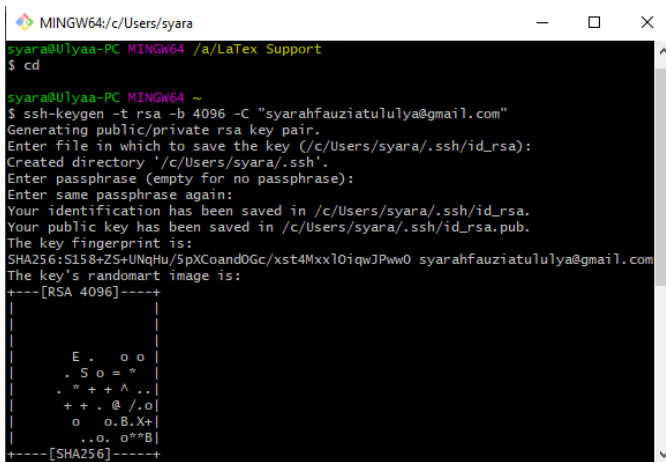
1. Buat akun di www.github.com terlebih dahulu

2. Install **Git Bash** dari alamat git-scm.com/downloads di komputer Anda, kemudian buka *git bash*.
3. Pastikan sudah ada di *home directory* dengan mengetikkan perintah `cd` kemudian *enter*. Untuk mengetahui posisi Anda ada di direktori mana ketikkan perintah `pwd` dan *enter*.
4. **Generate Key** dengan perintah

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

Listing 1.1 Perintah Membuat Key

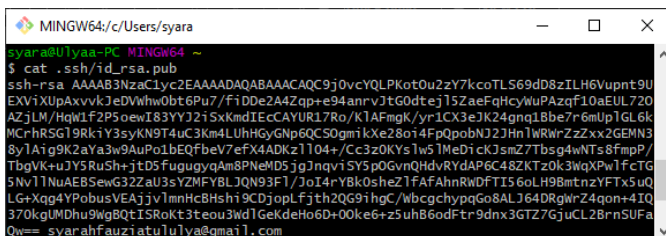
Hasilnya seperti yang terlihat pada gambar 1.1.



```
MINGW64:/c/Users/syara
syara@Ulyaa-PC MINGW64 /a/LaTeX Support
$ cd
syara@Ulyaa-PC MINGW64 ~
$ ssh-keygen -t rsa -b 4096 -C "syarahfauziatululya@gmail.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/syara/.ssh/id_rsa):
Created directory '/c/Users/syara/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/syara/.ssh/id_rsa.
Your public key has been saved in /c/Users/syara/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:51S8+ZS+UNqHu/SpXCoandOGc/xst4Mxx10iqwJPww0 syarahfauziatululya@gmail.com
The key's random image is:
+-----[RSA 4096]-----+
|
|  .   . o o
|  . S o = *
|  . * + + ^ . .
|  + . . @ / . o
|  o . o . B . X +
|  . . o . o * * B
+-----[SHA256]-----+
```

Gambar 1.1 Hasil Luaran 2

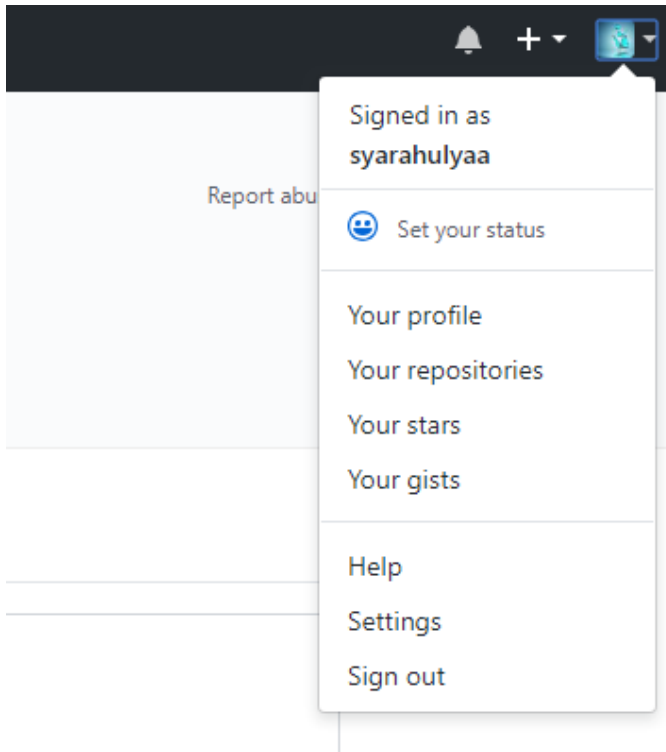
5. Baca *key* yang sudah di *generate* dengan menggunakan perintah `cat .ssh/id_rsa.pub` Hasilnya seperti pada gambar 1.2.



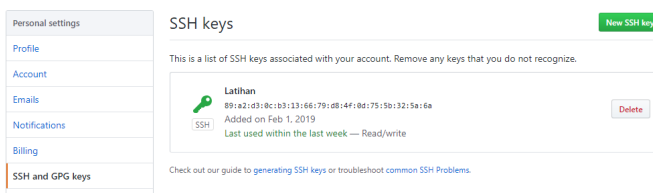
```
MINGW64:/c/Users/syara
syara@Ulyaa-PC MINGW64 ~
$ cat .ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQACQ9j0vcYQLPKot0u2zy7kcoTL569dD8zILH6Vupnt9U
EXViXUpAxxvkJedVmh0bt6Pu7/fiDDe2A4Zqp+e94anrvJtG0dtej15ZaeFqHcyWuPAZqf10aEUL720
AZjLM/HqW1f2P5oewI83YYj2iSxKmdIEcAYUR17Ro/K1AFmgK/yr1CX3eJK24gnq18be7r6mUp1GL6k
McRnRSG19RkiY3syKN9T4uC3Km4LuhHgyGnp6QCS0gmikXe28oi4FpQpobNj2JHn1WRWrZzxx2GEMN3
8y1Aig9K2aYa3w9AuPo1bEqfBeV7efX4ADKz1104+/Cc3z0KYs1w51MeDi cKJsmZ7TbSg4wNTS8fmpP/
TbgVk+uJY5RuSh+jtD5FugugyqAm8PNeM5jgJnqviSY5p0GvndHdvRYdAP6C48ZKTz0k3WqXPw1fCTG
5Nv11NuAEBSeWg3Z2aU3sYZMFYBLJQN93F1/JoI4rYBk0sheZ1FAFAhnrWDFtI56oLH98mntzYFTx5uQ
LG+Xqg4YPobusVEAjv1mnHcBHshi9CDjopLfjth2QG9ihgC/WbcgchypqGo8ALJ64DRgwrZ4gon+4IQ
370kgUMDhu9WgBQtISRoKt3teou3wd1GekdeHo6D+00ke6+zsuhB6odFtr9dnx3GTZ7GjuCL2BrnSUFa
Qw== syarahfauziatululya@gmail.com
```

Gambar 1.2 Hasil Luaran 2

6. Hasil luaran yang dibaca sebelumnya merupakan *key* kita. Kemudian masukkan *key* dengan masuk ke menu **Setting** yang ada dipojok kanan atas, seperti pada gambar 1.3. Lalu pada menu **SSH and GPG Keys** tambahkan **New SSH key** seperti pada gambar 1.4.



Gambar 1.3 Setting



Gambar 1.4 New SSH key

1.1.2 Fork Repositori

Pertama kita cari repositori utama yang akan kita ikut berkontribusi kepada repositori tersebut. Jika sudah ketemu, kemudian klik Fork(Tombol kanan atas) yang dilanjutkan dengan memilih akun kita sebagai tujuan clone fork tersebut. Setelah selesai maka kita akan memiliki repo yang sama dengan repo yang anda fork. Contoh apabila anda melakukan Fork dari <https://github.com/RepoAsal/Testing> maka anda akan memiliki repo <https://github.com/usernameAnda/Testing>, dan kita akan bekerja pada repo hasil fork ini.

1.1.3 Navigasi direktori dengan Git Bash

Pertama kita tentukan terlebih dahulu folder tempat kita mengerjakan repo hasil fork kita. Misal di Drive D: folder Ganteng. Maka buka git bash kita dan arahkan menuju folder tersebut dengan perintah `cd /D/Ganteng` lalu buka web repo fork kita di github klik tombol hijau **CLone or Download** pilih Clone with SSH lalu salin kode yang tampak seperti `git@github.com:usernameAnda/Testing.git`. Pada Git Bash ketik `git clone git@github.com:usernameAnda/Testing.git`. Setelah selesai, ketik perintah `ls` maka akan muncul direktori baru yaitu folder Testing atau sesuai dengan nama repo Anda. masuk ke direktori tersebut dengan perintah `cd Testing`. Kemudian ketik `git status` akan muncul status dari repo git kita.

1.1.4 Sinkronisasi dengan Repo Utama

Karena ini repo hasil Fork maka kita harus selalu di sinkronisasi dari repo aslinya(tidak otomatis tersinkron). Sehingga perlu melakukan setting sumber repo tujuan yang merupakan repo asal. Pertama pastikan git bash sudah pada direktori repositori. Untuk melakukan sinkronisasi dengan repo asal kita harus mengeset satu kali dengan perintah :

```
1 git remote add upstream https://github.com/RepoAsal/Testing.git
```

Listing 1.2 Set Repo Asal Sebagai Upstream

Setelah melakukan *setting* sekali di awal, kemudian selanjutnya kita tinggal melakukan sinkronisasi terus menerus sebelum melakukan perubahan dengan dua perintah berikut :

```
1 git fetch upstream
2 git pull upstream master
```

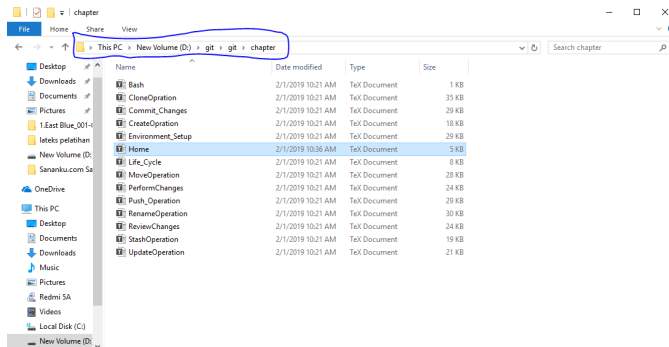
Listing 1.3 Perintah Sinkronisasi dengan repo asal

1.1.5 Bekerja dengan Git Bash

Sekarang kita mulai bekerja pada repo kita. Melakukan penambahan atau perubahan pada *file*. Kemudian perubahan tersebut diminta untuk dimasukkan di repo utama

tempat kita *fork* repo kita. Urutan pekerjaan yang kita ulang terus menerus adalah sebagai berikut :

1. `git fetch upstream`
2. `git pull upstream master`
3. `git push origin master`
4. Silahkan edit satu file yang akan di ubah atau ditambah lalu di simpan dan di tutup yang berada di direktori yang sudah disetting di navigasi direktori. Seperti terlihat pada gambar 1.5.



Gambar 1.5 Direktori

5. `git status`
6. untuk memasukkan file yang sudah diubah/diedit yaitu dengan perintah `git add namafilenya.tex`
7. `git status`
8. `git commit -m 'perubahan apa yang telah kita lakukan di ceritakan di sini secara lengkap'`
9. `git status`
10. `git push origin master`
11. Buka web repo kita di [github.com](https://github.com/usernameAnda/Testing) Contoh disini <https://github.com/usernameAnda/Testing> kemudian klik New pull request. Pastikan yang sebelah kiri atau base kita set kepada repo utama tempat fork kita yaitu RepoAsal/Testing dan compare pada sebelah kanan adalah repo kita, disini dicontohkan usernameAnda/Testing. Klik tombol hijau 'Create Pull Request'. Share. Dan klik kembali tombol hijau lagi.
12. Beritahukan admin repo utama untuk accept Pull Request Anda. Jika sudah di accept lakukan lagi langkah dari awal.

BAB 2

PERINTAH DASAR BASH

2.1 Perintah Navigasi

Perintah navigasi direktori

BAB 3

ENVIRONMENT SETUP

3.1 Environment Setup



Gambar 3.1 git setup environment

Tutorial ini menuntun Anda melalui cara menyiapkan Git di berbagai platform sehingga Anda bisa membaca dan menulis kode YUI dalam waktu singkat. Setelah lingkungan Anda diatur, pastikan untuk membaca tutorial untuk belajar bagaimana mengkloning kode sumber YUI ke lingkungan pengembangan lokal Anda, dan kirimkan CLA untuk berkontribusi pada YUI.

Instal Git di Mac

Singa

Download dan Install Xcode dari Mac App Store (gratis)

Setelah instalasi buka terminal dan ketik: `git --version`

Anda harus melihat sesuatu seperti Versi ini dapat bervariasi : `git version 1.7.4.4`

Git sekarang terpasang dan bekerja

[Macan Tutul Salju

Download dan Instal Git Client untuk OS X

Setelah instalasi buka terminal dan ketik: `git --version`

Anda harus melihat sesuatu seperti Versi ini dapat bervariasi : `git version 1.7.6.1`

Git sekarang terpasang dan bekerja

Harimau

Download dan Install paket Git untuk OS X

Anda perlu menambahkan `/usr/local/bin` ke `$PATH`

`vim ~/.bashrc`

Di bagian bawah file tambahkan baris ini:

`ekspor PATH= $PATH:/usr/local/bin`

Setelah instalasi buka terminal dan ketik: `git --version`

Anda harus melihat sesuatu seperti ini: `git version 1.7.6.1`

Git sekarang terpasang dan bekerja

3.1.1 Port Mac

Jika Anda menginstal Mac Ports, lakukan hal berikut:

Instal paket

`$git-core`

`sudo port install git-core \par`

Setelah instalasi buka terminal dan ketik: `$ $git --version \p`

Anda harus melihat sesuatu seperti ini: `$ $git version 1.6.0 \`

Git sekarang terpasang dan bekerja `\par`

Pengadu

Git ada di pohon yang tidak stabil di Fink, dan aku tidak bisa memasangnya. Anda harus memilih metode lain

Instal Git di Linux

Topi merah

Untuk Fedora 7 dan yang lebih baru.

Instal paket git-core
`sudo yum install git-core`

Setelah instalasi buka terminal dan ketik: `git --version`
Anda harus melihat sesuatu seperti ini: `git version 1.7.6.1`

Git sekarang terpasang dan bekerja

Untuk Fedora 6 dan yang lebih tua, git-core adalah bagian dari Fedora Extras

Bagi pengguna Red Hat Enterprise Linux dan klonnya (seperti CentOS dan Scientific Linux), pengguna harus membangun git dengan tangan. Sumber tersedia di sini: <http://git.or.cz/>

Ubuntu

Instal paket git-core
`sudo apt-get install git-core`

Setelah instalasi buka terminal dan ketik: `git --version`
Anda harus melihat sesuatu seperti ini: `git version 1.7.0.4`
Git sekarang terpasang dan bekerja

Instal Git di Windows

Download dan Instal Git Client untuk Windows
Selama penginstalan, Anda akan ditanya SSH Executable yang ingin Anda gunakan. Kecuali Anda sudah menggunakan plink / putty / kontes dan sudah terbiasa mengatur kunci di dalamnya, pilih "Use OpenSSH". Ini menyederhanakan pembuatan dan penggunaan kunci ssh.

Setelah instalasi buka Git Bash dan ketik: `git --version`

Anda harus melihat sesuatu seperti ini: `git version 1.7.6.1`

Git sekarang terpasang dan bekerja

Opsi Config Penting Anda perlu menambahkan opsi konfigurasi ini untuk membantu Windows menangani akhiran baris: `git config --global core.autocrlf true`

Konfigurasi Git

Menetapkan `user.name` dan `user.email` adalah opsi konfigurasi minimum yang perlu diatur sehingga nama dan email Anda akan muncul dalam pesan komit Anda.

```
git config --global user.name "Dav Glass" git config --global user.email
dav.glass@yahoo.com
```

Gunakan nama dan email anda tentu saja; Ada banyak pilihan konfigurasi lain yang bisa Anda atur. Konfigurasi ini harus ditulis ke file ini (sama untuk Windows, saat bekerja di Git Bash): `~/gitconfig`. Saya saat ini `~/gitconfig` terlihat seperti ini:

```
[merge] tool = opendiff [core] excludesfile = /Users/davglass/.gitignore [alias] st =
status ci = commit co = checkout br = branch [color] ui = auto status = auto branch
= auto diff = never [user] name = Dav Glass email = dav.glass@yahoo.com
```

Dan file `~/gitignore`

```
.DS_Store ._* .svn .hg .*.swp
```

Set up SSH

GitHub memiliki beberapa tutorial bermanfaat untuk menyiapkan SSH untuk digunakan dengan git. Klik link di bawah ini untuk informasi lebih lanjut:

Membangkitkan kunci SSH (OS X)

Membangkitkan kunci SSH (Win / msysgit)

Mengatasi masalah SSH

Kode Komitmen: Tips Git

Git sedikit berbeda dari CVS, setiap orang memiliki salinan repositori pada mesin lokal mereka. Jadi Anda bisa berkomitmen untuk itu sebanyak yang Anda suka. Tidak ada bangunan yang akan dipicu.

Sebagian besar alias built in untuk git akan membantu karena mirip dengan CVS.

`git pull` Ini akan menarik salinan repositori baru.

`git commit` Ini akan melakukan perubahan pada repositori lokal Anda (bukan server).

`git add` Ini sedikit berbeda, Anda menggunakan `git add` untuk menambahkan file ke indeks komit. Ini bukan hanya untuk file baru.

`git push` Ini yang memindahkan repositori lokal Anda ke server yang Anda periksa.

`git status` Menunjukkan status repositori Anda.

`git diff` Memberikan Anda diff file.

`git rm` Ini memungkinkan Anda menghapus file dan direktori.

`git` tambah

Contoh ini menunjukkan penggunaan `git add` untuk menambahkan setiap file ke indeks komit sebelum melakukan.

```
$ cd /my/working/directory $ vim README # Edit the file $ git add README $
git commit -m "Updated the README file" # Then you will see something like
this: Created commit 794a86f: Updated the README file 1 files changed, 1 inser-
tions(+), 0 deletions(-)
```

`git` komit semua

Contoh ini menunjukkan penggunaan `git -a` untuk melakukan semua file yang diubah

```
$ cd util/dd/src/js $ vim drag.js # Edit the file $ git commit -a # At this point
git will open your default editor and ask for a commit message # Then you will
see something like this: Created commit ed16907: fixed #2345: patched file to fix
something 1 files changed, 2 insertions(+), 0 deletions(-)
```

Putuskan dorongan gagal

Jika Anda mengalami kesalahan (non-fast forward) saat push:

```
To git@github.com:username/yui3.git ! [rejected] master -> master (non-fast forward)
error: failed to push some refs to 'git@github.com:username/yui3.git'
```

Umumnya itu berarti pohon Anda sekarang tidak sinkron dengan repositori utama. Anda perlu `cd` projectroot dan melakukan `git pull`. Ini harus membuat pohon Anda selaras dengan repositori dan kemudian Anda dapat mendorongnya.

Kembalikan Perubahan di Pohon Kerja Anda

Untuk mengembalikan file yang dimodifikasi di pohon kerja Anda (setara dengan `cvs co -C`):

```
$ git checkout -- path/to/file
```

git status akan memberi Anda daftar file di pohon kerja Anda yang telah dimodifikasi.

Buat Cabang Jauh

CATATAN: Metode ini tidak mengharuskan Anda untuk beralih ke cabang lokal untuk mendorongnya ke server hapus.

Buat cabang lokal (dari cabang yang saat ini Anda masuk (umumnya "master")):

```
$ git branch [branch name]
```

Dorong ke server asal:

```
$ git push origin [branch name]
```

Pengguna kemudian dapat melakukan checkout cabang remote dengan menggunakan:

```
$ git checkout -b [local branch name] origin/[branch name]
```

Yang akan membuat cabang lokal baru, yang melacak cabang remote, dan juga mengubah pohon kerja ke cabang lokal (jadi setiap perubahan yang Anda buat, Anda lakukan di cabang).

Pengguna kemudian dapat beralih antar cabang sesuai kebutuhan dengan menggunakan:

```
$ git checkout [branch name]
```

Beralih kembali ke "master", yang merupakan cabang pelacakan itu sendiri, bekerja dengan cara yang sama ...

```
git checkout "master"
```

Setel ulang ke Tag

Penyiapan Lingkungan Lokal

Jika Anda ingin mengatur lingkungan Anda untuk bahasa pemrograman C, Anda memerlukan dua perangkat lunak berikut yang tersedia di komputer Anda, (a) Text Editor dan (b) Kompiler C.

Text Editor

Ini akan digunakan untuk mengetikkan program anda. Contoh beberapa editor termasuk Windows Notepad, perintah OS Edit, Brief, Epsilon, EMACS, dan vim atau vi.

Nama dan versi editor teks dapat bervariasi pada sistem operasi yang berbeda. Misalnya, Notepad akan digunakan pada Windows, dan vim atau vi bisa digunakan di windows maupun di Linux atau UNIX.

File yang Anda buat dengan editor Anda disebut file sumber dan berisi kode sumber program. File sumber untuk program C biasanya dinamai dengan ekstensi ".c".

Sebelum memulai pemrograman, pastikan Anda memiliki satu editor teks dan Anda memiliki cukup pengalaman untuk menulis program komputer, simpan di file, kompilasi dan akhirnya jalankan.

Kompiler C

Kode sumber yang ditulis dalam file sumber adalah sumber yang dapat dibaca manusia untuk program Anda. Ini perlu "dikompilasi", ke dalam bahasa mesin sehingga CPU Anda benar-benar dapat menjalankan program sesuai instruksi yang diberikan.

Kompilator mengkompilasi kode sumber ke dalam program eksekusi akhir. Kompiler yang paling sering digunakan dan tersedia gratis adalah kompiler GNU C / C ++, jika tidak, anda dapat memiliki kompiler dari HP atau Solaris jika Anda memiliki sistem operasi masing-masing.

Bagian berikut menjelaskan cara menginstal kompiler GNU C / C ++ di berbagai OS. Kami terus menyebutkan C / C ++ bersama-sama karena kompiler GNU gcc bekerja untuk bahasa pemrograman C dan C ++.

Setup Git Pertama Kali

Sekarang setelah Anda memiliki Git di sistem Anda, Anda pasti ingin melakukan beberapa hal untuk menyesuaikan lingkungan Git Anda. Anda harus melakukan hal-hal ini hanya sekali pada komputer tertentu; mereka akan bertahan di antara upgrade. Anda juga bisa mengubahnya kapan saja dengan menjalankan perintah lagi.

Git dilengkapi dengan tool yang disebut git config yang memungkinkan Anda mendapatkan dan mengatur variabel konfigurasi yang mengendalikan semua aspek bagaimana Git terlihat dan beroperasi. Variabel ini dapat disimpan di tiga tempat berbeda:

/etc/gitconfig file: Berisi nilai untuk setiap pengguna pada sistem dan semua repositori mereka. Jika Anda melewatkan opsi `--system` ke git config, maka bacalah dan tulis dari file ini secara khusus.

`~/gitconfig` atau `~/config/git/config` : Spesifik untuk pengguna Anda. Anda bisa membuat Git membaca dan menulis ke file ini secara khusus dengan melewati opsi `-global` .

file config di direktori Git (yaitu, `.git/config`) dari repositori apa pun yang saat ini Anda gunakan: Khusus untuk repositori tunggal itu.

Setiap level menimpa nilai pada level sebelumnya, jadi nilai pada `.git/config` truf di `/etc/gitconfig` .

Pada sistem Windows, Git mencari file `.gitconfig` di direktori `$HOME` (`C: \Users \ $USER` untuk kebanyakan orang). Ini juga masih mencari `/etc/gitconfig` , meskipun relatif terhadap akar MSys, dimanapun Anda memutuskan untuk menginstal Git pada sistem Windows Anda saat Anda menjalankan installer. Jika Anda menggunakan versi 2.x atau yang lebih baru dari Git untuk Windows, ada juga file konfigurasi tingkat-sistem di `C: \Documents and Settings \All Users \Application Data \Git \config` pada Windows XP, dan di `C: \ProgramData \Git \config` pada Windows Vista dan yang lebih baru. File konfigurasi ini hanya bisa diubah oleh git config -f ;file, sebagai admin.

Identitas anda

Hal pertama yang harus Anda lakukan saat memasang Git adalah menyetel nama pengguna dan alamat email Anda. Hal ini penting karena setiap Git berkomitmen menggunakan informasi ini, dan secara konstan dipanggang dalam komit yang Anda mulai ciptakan:

```
$ git config --global user.name "John Doe" $ git config --global user.email john-doe@example.com
```

Sekali lagi, Anda perlu melakukan ini hanya sekali jika Anda melewati opsi `-global` , karena dengan begitu Git akan selalu menggunakan informasi itu untuk semua hal yang Anda lakukan pada sistem itu. Jika Anda ingin menimpa ini dengan nama atau alamat email yang berbeda untuk proyek tertentu, Anda dapat menjalankan perintah tanpa opsi `-global` saat berada dalam proyek itu.

Banyak alat GUI akan membantu Anda melakukan hal ini saat pertama kali menjalankannya.

Editor Anda

Setelah identitas Anda disiapkan, Anda dapat mengonfigurasi editor teks default yang akan digunakan saat Git membutuhkan Anda untuk mengetikkan pesan. Jika tidak dikonfigurasi, Git menggunakan editor default sistem Anda.

Jika Anda ingin menggunakan editor teks yang berbeda, seperti Emacs, Anda dapat melakukan hal berikut:

```
$ git config --global core.editor emacs
```

Pada sistem Windows, jika Anda ingin menggunakan editor teks yang berbeda, Anda harus menentukan path lengkap ke file executable-nya. Ini bisa berbeda tergantung bagaimana editor anda dikemas.

Dalam kasus Notepad ++, editor pemrograman populer, Anda cenderung ingin menggunakan versi 32-bit, karena pada saat penulisan versi 64-bit tidak mendukung semua plug-in. Jika Anda menggunakan sistem Windows 32-bit, atau Anda memiliki editor 64-bit pada sistem 64-bit, Anda akan mengetikkan sesuatu seperti ini:

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

Jika Anda memiliki editor 32 bit pada sistem 64-bit, program akan diinstal di C:\Program Files (x86) :

```
$ git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiInst -nosession"
```

Catatan

Vim, Emacs dan Notepad ++ adalah editor teks populer yang sering digunakan oleh pengembang pada sistem berbasis Unix seperti Linux dan macos atau sistem Windows. Jika Anda tidak terbiasa dengan editor ini, Anda mungkin perlu mencari petunjuk khusus tentang bagaimana mengatur editor favorit Anda dengan Git.

PERINGATAN

Anda mungkin menemukan, jika Anda tidak menset editor Anda seperti ini, Anda masuk ke keadaan yang benar-benar membingungkan saat Git mencoba meluncurkannya. Contoh pada sistem Windows mungkin mencakup operasi Git yang dihentikan secara prematur selama Git memulai pengeditan

Memeriksa Setelan Anda

Jika Anda ingin memeriksa pengaturan Anda, Anda dapat menggunakan perintah `git config --list` untuk mencantumkan semua pengaturan yang dapat ditemukan oleh Git pada saat itu:

```
$ git config --list user.name=John Doe user.email=johndoe@example.com color.status=auto color.branch=auto color.interactive=auto color.diff=auto ...
```


Anda mungkin melihat tombol lebih dari satu kali, karena Git membaca kunci yang sama dari file yang berbeda (`/etc/gitconfig` dan `~/.gitconfig` , misalnya). Dalam kasus ini, Git menggunakan nilai terakhir untuk setiap kunci unik yang dilihatnya.

Anda juga dapat memeriksa apakah Git menganggap nilai kunci tertentu adalah dengan mengetikkan `git config |key|` :

```
$ git config user.name John Doe
```

Mendapatkan bantuan

Jika Anda memerlukan bantuan saat menggunakan Git, ada tiga cara untuk mendapatkan manual page manual (`manpage`) untuk perintah Git manapun:

```
$ git help |verb| $ git |verb| --help $ man git-|verb|
```

Misalnya, Anda bisa mendapatkan manual bantuan untuk perintah konfigurasi dengan menjalankan

```
$ git help config
```

Perintah ini bagus karena Anda bisa mengaksesnya dimana saja, bahkan offline. Jika halaman manual dan buku ini tidak cukup dan Anda memerlukan bantuan langsung, Anda dapat mencoba saluran `#git` atau `#github` di server Freenode IRC (`irc.freenode.net`). Saluran ini secara teratur dipenuhi oleh ratusan orang yang sangat berpengetahuan luas tentang Git dan sering bersedia membantu.

Set Up Git

Inti GitHub adalah sistem kontrol versi open source (VCS) yang disebut Git . Git bertanggung jawab atas segala sesuatu yang berhubungan dengan GitHub yang terjadi secara lokal di komputer Anda.

Untuk menggunakan Git pada baris perintah, Anda harus mendownload, menginstal, dan mengkonfigurasi Git di komputer Anda.

Jika Anda ingin bekerja dengan Git secara lokal, namun tidak ingin menggunakan command line, Anda dapat mendownload dan menginstal klien Desktop GitHub . Untuk informasi lebih lanjut, lihat ” Memulai Desktop GitHub ”.

Jika Anda tidak perlu bekerja dengan file secara lokal, GitHub memungkinkan Anda menyelesaikan banyak tindakan terkait Git secara langsung di browser, termasuk:

1. Membuat repositori
2. Forking sebuah repositori

3. Mengelola file
4. Menjadi sosial
5. Menyiapkan Git
6. Download dan instal versi terbaru Git
7. Tetapkan nama pengguna Anda di Git
8. Tetapkan alamat email komit di Git
9. Langkah selanjutnya: Mengautentikasi dengan GitHub dari Git
10. Saat Anda terhubung ke repositori GitHub dari Git, Anda harus melakukan otentikasi dengan GitHub menggunakan HTTPS atau SSH.
11. Menghubungkan HTTPS (disarankan)
12. Jika Anda mengkloning dengan HTTPS , Anda dapat menyimpan sandi GitHub Anda di Git menggunakan penolong kredensial.
13. Menghubungkan melalui SSH
14. Jika Anda mengkloning SSH , Anda harus membuat kunci SSH di setiap komputer yang Anda gunakan untuk mendorong atau menarik dari GitHub.

Selamat, sekarang kalian sudah menyiapkan Git dan GitHub! Apa yang ingin kamu lakukan selanjutnya?

Siapkan Git

” Buat repositori ”

” Garpu repositori ”

” Jadilah sosial”

Setup Git Pertama Kali

Sekarang setelah Anda memiliki Git di sistem Anda, Anda pasti ingin melakukan beberapa hal untuk menyesuaikan lingkungan Git Anda. Anda harus melakukan hal-hal ini hanya sekali pada komputer tertentu; mereka akan bertahan di antara upgrade. Anda juga bisa mengubahnya kapan saja dengan menjalankan perintah lagi.

Git dilengkapi dengan tool yang disebut `git config` yang memungkinkan Anda mendapatkan dan mengatur variabel konfigurasi yang mengendalikan semua aspek bagaimana Git terlihat dan beroperasi. Variabel ini dapat disimpan di tiga tempat berbeda:

/etc/gitconfig file: Berisi nilai untuk setiap pengguna pada sistem dan semua repositori mereka. Jika Anda melewatkan opsi `-system` ke `git config`, maka bacalah dan tulis dari file ini secara khusus.

`~/.gitconfig` atau `~/.config/git/config`: Spesifik untuk pengguna Anda. Anda bisa membuat Git membaca dan menulis ke file ini secara khusus dengan melewatkan opsi `-global`.

file config di direktori Git (yaitu, `.git/config`) dari repositori apa pun yang saat ini Anda gunakan: Khusus untuk repositori tunggal itu.

Setiap level menimpa nilai pada level sebelumnya, jadi nilai pada `.git/config` truf di `/etc/gitconfig`.

Pada sistem Windows, Git mencari file `.gitconfig` di direktori `$HOME` (`C:\Users\ $USER` untuk kebanyakan orang). Ini juga masih mencari `/etc/gitconfig`, meskipun relatif terhadap akar MSys, dimanapun Anda memutuskan untuk menginstal Git pada sistem Windows Anda saat Anda menjalankan installer. Jika Anda menggunakan versi 2.x atau yang lebih baru dari Git untuk Windows, ada juga file konfigurasi tingkat-sistem di `C:\Documents and Settings\All Users\Application Data\Git\config` pada Windows XP, dan di `C:\ProgramData\Git\config` pada Windows Vista dan yang lebih baru. File konfigurasi ini hanya bisa diubah oleh `git config -f` sebagai admin.

Identitas anda

Hal pertama yang harus Anda lakukan saat memasang Git adalah menyetel nama pengguna dan alamat email Anda. Hal ini penting karena setiap Git berkomitmen menggunakan informasi ini, dan secara konstan dipanggang dalam komit yang Anda mulai ciptakan:

```
$ git config --global user.name "John Doe" $ git config --global user.email john-doe@example.com
```

Sekali lagi, Anda perlu melakukan ini hanya sekali jika Anda melewatkan opsi `-global`, karena dengan begitu Git akan selalu menggunakan informasi itu untuk semua hal yang Anda lakukan pada sistem itu. Jika Anda ingin menimpa ini dengan nama atau alamat email yang berbeda untuk proyek tertentu, Anda dapat menjalankan perintah tanpa opsi `-global` saat berada dalam proyek itu.

Banyak alat GUI akan membantu Anda melakukan hal ini saat pertama kali menjalankannya.

Editor Anda

Setelah identitas Anda disiapkan, Anda dapat mengonfigurasi editor teks default yang akan digunakan saat Git membutuhkan Anda untuk mengetikkan pesan. Jika tidak dikonfigurasi, Git menggunakan editor default sistem Anda.

Jika Anda ingin menggunakan editor teks yang berbeda, seperti Emacs, Anda dapat melakukan hal berikut:

```
$ git config --global core.editor emacs
```

Pada sistem Windows, jika Anda ingin menggunakan editor teks yang berbeda, Anda harus menentukan path lengkap ke file executable-nya. Ini bisa berbeda tergantung bagaimana editor anda dikemas.

Dalam kasus Notepad ++, editor pemrograman populer, Anda cenderung ingin menggunakan versi 32-bit, karena pada saat penulisan versi 64-bit tidak mendukung semua plug-in. Jika Anda menggunakan sistem Windows 32-bit, atau Anda memiliki editor 64-bit pada sistem 64-bit, Anda akan mengetikkan sesuatu seperti ini:

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

Jika Anda memiliki editor 32 bit pada sistem 64-bit, program akan diinstal di C:\Program Files (x86) :

```
$ git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiInst -nosession"
```

Memeriksa Setelan Anda

Jika Anda ingin memeriksa pengaturan Anda, Anda dapat menggunakan perintah git config --list untuk mencantumkan semua pengaturan yang dapat ditemukan oleh Git pada saat itu:

```
$ git config --list user.name=John Doe user.email=johndoe@example.com color.status=auto color.branch=auto color.interactive=auto color.diff=auto ...
```

Anda mungkin melihat tombol lebih dari satu kali, karena Git membaca kunci yang sama dari file yang berbeda (/etc/gitconfig dan ~/.gitconfig , misalnya). Dalam kasus ini, Git menggunakan nilai terakhir untuk setiap kunci unik yang dilihatnya.

Anda juga dapat memeriksa apakah Git menganggap nilai kunci tertentu adalah dengan mengetikkan git config ;key; :

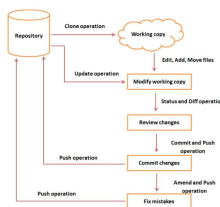
```
gitconfiguser.nameJohnDoe
```

(3.1)

BAB 4

LIFE CYCLE

4.1 Life Cycle



Gambar 4.1 Life Cycle

Git adalah sebuah perangkat lunak untuk mengontrol versi sebuah perangkat lunak "VCS/Version Control System". Git diciptakan oleh Linux Torvalds, yang pada awalnya ditujukan untuk pengembangan kernel Linux. Saat ini banyak perangkat lunak yang terkenal menggunakan Git sebagai pengontrol revisinya.

Pada tutorial ini Anda akan mempelajari bagaimana cara menggunakan Git seperti proses life cycle Git, operasi-operasi dasar dan bagaimana cara menangani masalah saat menggunakan Git.

Activity life cycle adalah sebuah siklus kehidupan activity. Sangat penting untuk memahami activity lifecycle ini untuk menghindari kebocoran memory. Dengan mengimplementasikan activity lifecycle ini kita bisa mengoptimalkan performa dari aplikasi yang kita buat. Setidaknya ada 6 siklus activity yang perlu kita pelajari, yaitu onCreate, onStart, onResume onPause, onStop, dan onDestroy.

1. onCreate onCreate adalah state pada saat sebuah halaman atau activity pertama kali di buat atau di attach kedakam memory. onCreate ini hanya di jalankan pada saat pertama kali di buat, jika sebuah activity telah tersedia dalam backstack onCreate ini tidak di panggil lagi.
2. onStart onStart adalah state setelah onCreate di jalankan. onStart juga tetap di jalankan walaupun onCreate tidak di panggil dengan syarat sebuah activity telah tersimpan di dalam backstack memory. onStart ini juga di jalankan jika masih berada dalam state onStop.
3. onResume onResume di eksekusi setelah on start. onResume ini di jalankan jika kita memanggil onPause.
4. onPause onPause ini setelah onResume. Ketika onPause ini di jalankan activity sudah dalam keadaan hilang atau sudah tidak tampil di user interface.
5. onStop onStop ini di akses setelah on pause dan sebelum activity ini benar - benar di hapus dari memory.
6. onDestroy onDestroy adalah bagian terakhir ketika activity benar - benar di hapus dari memory. Ketika sebuah activity telah di destroy maka tidak bisa melakukan onResume atau onStart.

4.1.1 SQLite Database

SQLite adalah salah satu portable database yang bisa kita gunakan untuk menyimpan data sesuai dengan kebutuhan aplikasi yang kita buat. Ada beberapa lite database yang bisa di gunakan dalam pembuatan aplikasi android. Namun SQLite database ini adalah yang ada secara default yang tersedia di android. Untuk SQLite ini juga menggunakan query yang cukup mirip - mirip dengan MYSQL. Jadi jika sudah terbiasa dengan MYSQL maka tidak akan kesulitan untuk memulai menggunakan SQLite ini.

Content Provider adalah sebuah fasilitas di android untuk kita gunakan untuk mengatur akses ke satu set terstruktur data. Dengan Content Provider memungkinkan kita untuk bisa share data dengan aplikasi lain.

1. Determine URI's
2. Update Contract
3. Fill Out our URI matcher
4. Implemen Function

Sebagian dari anda mungkin sering bercengkerama dengan yang namanya komputer. Terutama para programmer, apalagi yang masih jomblo. Kasihan ya, yang dibuat cengkrama keseharian cuma komputer, server, dan kroni-kroninya.. hehe :p. Saya doakan deh para jomblo-jomblo yang belum menikah, semoga diberi jalan dan kemudahan Oleh Allah Subhanahu Wa Taala untuk segera menemukan dambaan hatinya..aamiin

Kembali ke laptop, di tutorial kali ini saya akan membagikan materi tentang pengenalan GIT. Ini adalah salah satu bagian dari materi pelatihan git dan gitlab di kantor Telkomsel di Jakarta. Pelatihan ini adalah ini atas permintaan dari pihak Telkomsel untuk membagikan pengalaman penggunaan GIT dan GITLAB, terutama ketika di implementasikan dalam pengembangan project-project sistem yang sering di pegang oleh Profio Teknova Indonesia. Memang semenjak awal kami berdiri, hampir 2 tahun lalu, kami langsung menerapkan sistem kerja menggunakan Repository(GIT). Terbukti ini adalah formula yang cukup ampuh ketika kita melakukan development project secara bersama sama.

Git adalah tools yang berfungsi sebagai Version Control System (VCS) pelacak perubahan pada file. Git sendiri dibuat oleh orang yang menciptakan Kernel Linux., yah mungkin temen-temen sudah kenal nama linus torvalds ya.. dialah Pencetus git, yang kita gunakan. Git diciptakan tahun 2005 saat bitkeeper mulai retak dan pencipta linux harus mencari alternatif lain untuk mendukung pengembangan kernel linux. Dengan menggunakan Git, setiap orang dalam sebuah tim dapat melakukan perubahan pada source-code tanpa harus takut terjadi bentrok ataupun kesulitan dalam menggabungkan hasil perubahan yang mereka lakukan.

Dengan menggunakan Git, setiap perubahan pada source-code akan terlacak pesan perubahannya, apa saja yang diubah, siapa yang mengubah dan kapan waktunya.

Langsung saja, kita coba berkenalan dengan perintah perintah dasar yang sering digunakan ketika bekerja menggunakan sistem repository:

```
git init
```


Untuk membuat repo lokal baru pada perintah ini akan dibuat sebuah folder baru yang bernama ".git"

```
$ git init
```

```
$ git status
```

untuk melihat status dari repo lokal

Contoh : masuk ke direktori repo lokal

```
$ git status
```

git add

Untuk menambahkan file ke dalam repo yang sebelumnya sudah dibuat

```
$ git add myfile.txt
```

git commit

untuk menyimpan seluruh perubahan yang terjadi

```
$ git commit -m "first commit"
```

git remote

untuk menambahkan remote repo baru

Contoh :

```
gitremoteaddorigingit@bitbucket.org : xxxx/xxxx.git (4.1)
```

git pull / push

untuk menyimpan dan mengambil data dari remote repo

Contoh : \$ git pull origin master

\$ git push origin master

git diff

untuk membandingkan perubahan file

Contoh : \$ git diff

git reset

ntuk membatalkan perubahan pada repo local

Contoh : \$ Git reset --soft HEAD ^ (Lokal)

\$ Git reset --hard /

git Clone

uuntuk melakukan cloning pekerjaan yang telah exist ke local

Contoh : \$ git clone git@bitbucket.org:xxxxx/xxxxx.git

git merge

untuk melakukan penggabungan antar branch

git checkout

untuk pindah branch

BAB 5

CREATE OPERATION

5.1 Create Oprator

- Dasar Git

5.1.1 Dasar-Dasar Git

Jadi, sebenarnya apa yang dimaksud dengan Git? Ini adalah bagian penting untuk dipahami, karena jika anda memahami apa itu Git dan cara kerjanya, maka dapat dipastikan anda dapat menggunakan Git secara efektif dengan mudah. Selama mempelajari Git, cobalah untuk melupakan VCS lain yang mungkin telah anda kenal sebelumnya, misalnya Subversion dan Perforce. Git sangat berbeda dengan sistem-sistem tersebut dalam hal menyimpan dan memperlakukan

informasi yang digunakan, walaupun antar-muka penggunaannya hampir mirip. Dengan memahami perbedaan tersebut diharapkan dapat membantu anda menghindari kebingungan saat menggunakan Git.

Salah satu perbedaan yang mencolok antar Git dengan VCS lainnya (Subversion dan kawan-kawan) adalah dalam cara Git memperlakukan datanya. Secara konseptual, kebanyakan sistem lain menyimpan informasi sebagai sebuah daftar perubahan berkas. Sistem seperti ini (CVS, Subversion, Bazaar, dan yang lainnya) memperlakukan informasi yang disimpannya sebagai sekumpulan berkas dan perubahan yang terjadi pada berkas-berkas tersebut,

Git tidak bekerja seperti ini. Melainkan, Git memperlakukan datanya sebagai sebuah kumpulan snapshot dari sebuah miniatur sistem berkas. Setiap kali anda melakukan commit, atau melakukan perubahan pada proyek Git anda, pada dasarnya Git merekam gambaran keadaan berkas-berkas anda pada saat itu dan menyimpan referensi untuk gambaran tersebut. Agar efisien, jika berkas tidak mengalami perubahan, Git tidak akan menyimpan berkas tersebut melainkan hanya pada file yang sama yang sebelumnya telah disimpan.

perbedaan penting antara Git dengan hampir semua VCS lain. Hal ini membuat Git mempertimbangkan kembali hampir setiap aspek dari version control yang oleh kebanyakan sistem lainnya disalin dari generasi sebelumnya. Ini membuat Git lebih seperti sebuah miniatur sistem berkas dengan beberapa tool yang luar biasa ampuh yang dibangun di atasnya, ketimbang sekadar sebuah VCS.

Kebanyakan operasi pada Git hanya membutuhkan berkas-berkas dan resource lokal tidak ada informasi yang dibu-

tuhkan dari komputer lain pada jaringan anda. Jika Anda terbiasa dengan VCS terpusat dimana kebanyakan operasi memiliki overhead latensi jaringan, aspek Git satu ini akan membuat anda berpikir bahwa para dewa kecepatan telah memberkati Git dengan kekuatan. Karena anda memiliki seluruh sejarah dari proyek di lokal disk anda, dengan kebanyakan operasi yang tampak hampir seketika.

Sebagai contoh, untuk melihat history dari proyek, Git tidak membutuhkan data histori dari server untuk kemudian menampilkannya untuk anda, namun secara sadar Git membaca historinya langsung dari basisdata lokal proyek tersebut. Ini berarti anda melihat history proyek hampir secara instant. Jika anda ingin membandingkan perubahan pada sebuah berkas antara versi saat ini dengan versi sebulan yang lalu, Git dapat mencari berkas yang sama pada sebulan yang lalu dan melakukan perbandingan perubahan secara lokal, bukan dengan cara meminta remote server melakukannya atau meminta server mengirimkan berkas versi yang lebih lama kemudian membandingkannya secara lokal.

Hal ini berarti bahwa sangat sedikit yang tidak bisa anda kerjakan jika anda sedang offline atau berada diluar VPN. Jika anda sedang berada dalam pesawat terbang atau sebuah kereta dan ingin melakukan pekerjaan kecil, anda dapat melakukan commit sampai anda memperoleh koneksi internet hingga anda dapat menguploadnya. Jika anda pulang ke rumah dan VPN client anda tidak bekerja dengan benar, anda tetap dapat bekerja. Pada kebanyakan sistem lainnya, melakukan hal ini cukup sulit atau bahkan tidak mungkin sama sekali. Pada Perforce misalnya, anda tidak dapat berbuat banyak ketika anda tidak terhubung dengan server; pada Subversion dan CVS, anda dapat mengubah berkas, tapi anda tidak dapat melakukan commit pada basisdata anda (karena anda tidak terhubung dengan basisdata). Hal ini mungkin saja bukanlah masalah yang besar, namun anda

akan terkejut dengan perbedaan besar yang disebabkan nya.

Segala sesuatu pada Git akan melalui proses checksum terlebih dahulu sebelum disimpan yang kemudian direferensikan oleh hasil checksum tersebut. Hal ini berarti tidak mungkin melakukan perubahan terhadap berkas manapun tanpa diketahui oleh Git. Fungsionalitas ini dimiliki oleh Git pada level terendahnya dan ini merupakan bagian tak terpisahkan dari filosofi Git. Anda tidak akan kehilangan informasi atau mendapatkan file yang cacat tanpa diketahui oleh Git.

- Secara umum git hanya menambah data

5.1.2 Penjelasan git untuk menambah data

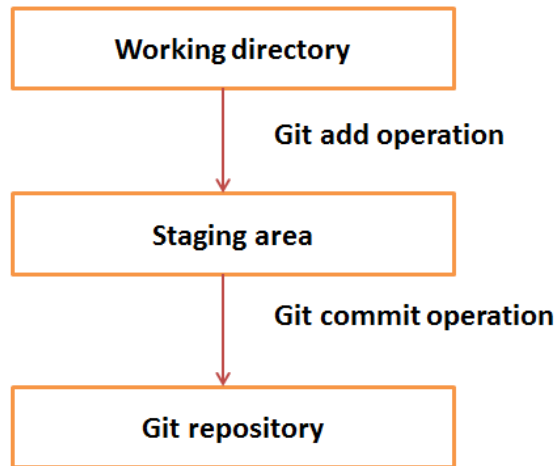
Ketika anda melakukan operasi pada Git, kebanyakan dari operasi tersebut hanya menambahkan data pada basisdata Git. It is very difficult to get the system to do anything that is not undoable or to make it erase data in any way. Seperti pada berbagai VCS, anda dapat kehilangan atau mengacaukan perubahan yang belum di-commit; namun jika anda melakukan commit pada Git, akan sangat sulit kehilangannya, terutama jika anda secara teratur melakukan push basisdata anda pada repositori lain.

Hal ini menjadikan Git menyenangkan karena kita dapat berexperimen tanpa keawajiban untuk mengacaukan proyek. Untuk lebih jelas dan dalam lagi tentang bagaimana Git menyimpan datanya dan bagaimana anda dapat mengembalikan yang hilang, lihat "Under the Covers" pada Bab 9.

Sekarang perhatikan. Ini adalah hal utama yang harus diingat tentang Git jika anda ingin proses belajar anda berjalan

lancar. Git memiliki 3 keadaan utama dimana berkas anda dapat berada: committed, modified dan staged. Committed berarti data telah tersimpan secara aman pada basisdata lokal. Modified berarti anda telah melakukan perubahan pada berkas namun anda belum melakukan commit pada basisdata. Staged berarti anda telah menandai berkas yang telah diubah pada versi yang sedang berlangsung untuk kemudian dilakukan commit.

- mulai operasi



Gambar 5.1 operasi

Direktori Git adalah dimana Git menyimpan metadata dan database objek untuk projek anda. Ini adalah bahagian terpenting dari Git, dan inilah yang disalin ketika anda melakukan kloning sebuah repository dari komputer lain.

Direktori kerja adalah sebuah checkout tunggal dari satu versi dari projek. Berkas-berkas ini kemudian ditarik keluar dari basisdata yang terkompresi dalam direktori Git dan disimpan pada disk untuk anda gunakan atau modifikasi.

Staging area adalah sebuah berkas sederhana, umumnya berada dalam direktori Git anda, yang menyimpan informasi mengenai apa yang menjadi commit selanjutnya. Ini terkadang disebut sebagai index, tetapi semakin menjadi standard untuk menyebutnya sebagai staging area.

Alur kerja dasar Git adalah seperti ini:

Anda mengubah berkas dalam direktori kerja anda.

Anda membawa berkas ke stage, menambahkan snapshot-nya ke staging area.

Anda melakukan commit, yang mengambil berkas seperti yang ada di staging area dan menyimpan snapshotnya secara permanen ke direktori Git anda.

Jika sebuah versi tertentu dari sebuah berkas telah ada di direktori git, ia dianggap 'committed'. Jika berkas diubah (modified) tetapi sudah ditambahkan ke staging area, maka itu adalah 'staged'. Dan jika berkas telah diubah sejak terakhir dilakukan checked out tetapi belum ditambahkan ke staging area maka itu adalah 'modified'

Mari memulai menggunakan Git. Pertama, tentu saja anda harus menginstallnya terlebih dahulu. Anda dapat melakukan melalui berbagai cara; dua cara paling populer adalah menginstallnya dari kode sumbernya atau menginstallkan paket yang telah disediakan untuk platform anda.

Jika anda dapat melakukannya, akan sangat berguna untuk dapat menginstallnya dari kode sumber, karena anda akan mendapatkan versi terbaru dari Git. Setiap versi dari Git cenderung akan menampilkan kemajuan pada sisi antar-

muka pengguna, jadi menggunakan versi terbaru seringkali menjadi jalan terbaik jika anda terbiasa melakukan kompilasi perangkat lunak dari kode sumbernya. Dan juga menjadi masalah bahwa banyak distribusi Linux yang menyertakan versi Git yang sangat lama; kecuali anda mempergunakan distribusi Linux paling up-to-date atau menggunakan backport, menginstall dari kode sumbernya mungkin menjadi solusi terbaik.

5.1.3 Setup Git Untuk Pertama Kalinya

Sekarang anda telah memiliki Git pada sistem anda, berikutnya anda akan harus melakukan beberapa penyesuaian pada lingkungan Git anda. Anda hanya perlu melakukan hal ini sekali saja; pada saat memperbaharui versi Git anda, penyesuaian tidak perlu dilakukan lagi. Anda pun dapat mengubah penyesuaian tersebut setiap saat.

Pada Git terdapat sebuah perkakas yang disebut dengan `git config` yang memungkinkan anda untuk memperoleh informasi dan menetapkan variable konfigurasi yang mengontrol segala aspek bagaimana Git beroperasi dan berperilaku. Variable-variable ini dapat disimpan pada tiga tempat berbeda: `gitconfig` file: Menyimpan berbagai nilai-nilai variable untuk setiap pengguna pada sistem dan semua repositori milik para pengguna tersebut. Jika anda memberikan opsi `-system` pada `git config`, maka Git akan membaca dan menulis file konfigurasi ini secara spesifik. `gitconfig` file: Spesifik hanya untuk pengguna yang bersangkutan. Anda dapat membuat Git membaca dan menulis pada berkas ini secara spesifik dengan memberikan opsi `-global`.

`config` file pada direktori git (yaitu, `.git config`) atau repositori manapun yang sedang anda gunakan: Spesifik hanya

pada repositori itu saja. Setiap nilai pada setiap tingkat akan selalu menimpa nilai yang telah ditetapkan pada level sebelumnya, jadi nilai yang telah di-set pada `.git` config akan menimpa nilai yang telah di-set pada `gitconfig`.

5.1.4 Cek Status dari Berkas

Anda Alat utama yang Anda gunakan untuk menentukan berkas-berkas mana yang berada dalam keadaan tertentu adalah melalui perintah `git status`. Jika Anda menggunakan alat ini langsung setelah sebuah clone, Anda akan melihat serupa seperti di bawah ini: `git status` On branch master
nothing to commit, working directory clean

Ini berarti Anda memiliki direktori kerja yang bersih-dengan kata lain, tidak ada berkas terpantau yang berubah. Git juga tidak melihat berkas-berkas yang tak terpantau, karena pasti akan dilaporkan oleh alat ini. Juga, perintah ini memberitahu Anda tentang cabang tempat Anda berada. Pada saat ini, cabang akan selalu berada di master, karena sudah menjadi default-nya; Anda tidak perlu khawatir tentang cabang dulu. Bab berikutnya akan membahas tentang percabangan dan referensi secara lebih detail.

Memantau Berkas Baru Untuk mulai memantau berkas baru, Anda menggunakan perintah `git add`. Untuk mulai memantau berkas `README` tadi, Anda menjalankannya seperti berikut:

`git add README` Jika Anda menjalankan perintah `status` lagi, Anda akan melihat bahwa berkas `README` Anda sekarang sudah terpantau dan sudah masuk ke dalam area `stage`:

```
git status
On branch master
```

```
Changes to be committed:  
use git reset HEAD <file>... to unstage  
new file:   README
```

Anda dapat mengatakan bahwa berkas tersebut berada di dalam area stage karena tertulis di bawah judul `Changes to be committed`. Jika Anda melakukan `commit` pada saat ini, versi berkas pada saat Anda menjalankan `git add` inilah yang akan dimasukkan ke dalam sejarah snapshot. Anda mungkin ingat bahwa ketika Anda menjalankan `git init` sebelumnya, Anda melanjutkannya dengan `git add` (nama berkas) yang akan mulai dipantau di direktori Anda. Perintah `git add` ini mengambil alamat dari berkas ataupun direktori; jika sebuah direktori, perintah tersebut akan menambahkan seluruh berkas yang berada di dalam direktori secara rekursif.

Bekerja Berjarak

Untuk dapat berkolaborasi untuk proyek Git apapun, Anda perlu mengetahui bagaimana Anda dapat mengatur repositori berjarak dari jarak jauh. Repositori berjarak adalah sekumpulan versi dari proyek Anda yang disiarkan di Internet atau di jaringan. Anda dapat memiliki beberapa repositori berjarak, masing-masing bisanya dengan akses terbatas untuk membaca saja ataupun baca/tulis. Berkolaborasi dengan pihak lain menuntut kemampuan untuk mengatur repositori berjarak ini dan menarik dan mendorong data ke dan dari repositori berjarak tersebut ketika Anda butuh untuk membagi hasil kerja Anda.

5.1.5 Apakah Branch Itu

Untuk benar-benar mengerti cara Git melakukan branching, kita perlu kembali ke belakang dan membahas bagaimana Git menyimpan datanya. Seperti yang mungkin anda ingat dari Bab 1, Git tidak menyimpan data sebagai serangka-

ian kumpulan perubahan atau delta, melainkan sebagai serangkaian snapshot. Contoh :

CreateOperation (5.1)

Ketika anda melakukan commit dalam Git, Git menyimpan sebuah object commit yang berisi pointer ke snapshot dari konten yang anda staged, metadata pembuat (author) dan pesan (message), dan nol atau lebih pointer ke commit yang merupakan parent (induk) langsung dari commit ini: nol jika ini commit yang pertama, satu jika ini commit yang normal, dan beberapa jika ini commit yang dihasilkan dari gabungan antara dua atau lebih branch.

Hampir setiap VCS memiliki sejumlah dukungan atas branching (percabangan). Branching adalah membuat cabang dari repositori utama dan melanjutkan melakukan pekerjaan pada cabang yang baru tersebut tanpa perlu khawatir mengacaukan yang utama. Dalam banyak VCS, branching adalah proses yang agak mahal, karena seringkali mengharuskan anda untuk membuat salinan baru dari direktori kode sumber, dimana dapat memakan waktu lama untuk proyek-proyek yang besar.

Beberapa orang menyebut model branching dalam Git sebagai "killer feature," hal inilah yang membuat Git berbeda di komunitas VCS. Mengapa begitu istimewa? Cara Git membuat cabang sangatlah ringan, membuat operasi branching hampir seketika dan berpindah bolak-balik antara cabang umumnya sama cepatnya. Tidak seperti VCS lainnya, Git mendorong alur kerja dimana kita sering membuat cabang dan kemudian menggabungkannya, bahkan dapat beberapa kali dalam sehari. Memahami dan menguasai fitur ini memberi anda perangkat yang ampuh, unik, dan benar-benar dapat mengubah cara anda melakukan pengembangan (develop).

BAB 6

CLONE OPERATION

Clone Opration

- Logo



Gambar 6.1 Logo

6.1 Clone Opration

cara install

Lakukan inisialisasi dengan mengetikkan perintah berikut pada Git Bash tadi

- `git init`

Perintah tersebut akan membuat sebuah repository lokal untuk proyek kita

Langkah berikutnya adalah memasukkan file-file source code serta folder pada proyek kedalam staging area, yaitu suatu kondisi dimana file serta folder source code dimasukkan ke dalam repository namun dalam keadaan temporary, belum disimpan. Untuk melakukannya gunakan perintah berikut.

- `git add *`

Perintah tersebut akan memasukkan seluruh file dan folder yang ada pada folder ProyekPHP. Jika ingin memasukkan satu persatu cukup tuliskan nama file lengkap dengan ekstensinya atau nama folder jika hanya ingin menambahkan satu folder

- `git add index.php`
- `git add nama folder`

Setelah itu kita siap untuk menyimpan source code kita kedalam repository. Ketikkan perintah berikut

`git commit -m` (dan teks penambahan data atau quotes apa saja)

Perintah diatas akan menyimpan source code kita sekaligus memberikan catatan supaya mudah kita ingat

Sekarang login ke Github.com dan buatlah sebuah repository baru dengan mengeklik tombol yang terletak pada kanan atas. Perhatikan gambar berikut

Buat repository dengan nama "PHPKeren " misalnya

Sekarang kita bisa mengakses remote repository dengan url

Kembali ke Git Bash. Tambahkan remote repository yang barusan kita buat supaya proyek kita bisa diupload. Berikut perintahnya

- `git remote add origin` (link github yang di copy ssh nya)

Selanjutnya kita download terlebih dahulu file readme yang ada secara default ketika kita membuat repository di github dengan mengetikkan perintah

- `git pull origin master`

Maka file readme.md akan berada pada folder proyek kita

Terakhir adalah mengupload ke Github dengan perintah

- `git push origin master`

masukkan username serta password jika diminta

Cek pada github maka file ktia sudah berada disana

6.1.1 Memantau berkas baru

Untuk mulai memantau berkas baru, Anda menggunakan perintah `git add`. Untuk mulai memantau berkas `README` tadi, Anda menjalankannya seperti berikut:

```
$ git add README
```

Jika Anda menjalankan perintah `status` lagi, Anda akan melihat bahwa berkas `README` Anda sekarang sudah terpantau dan sudah masuk ke dalam area stage:

```
git status \\}
On branch master \\}
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
new~file:~ README \\}
```

Anda dapat mengatakan bahwa berkas tersebut berada di dalam area stage karena tertulis di bawah judul "Changes to be committed". Jika Anda melakukan commit pada saat ini, versi berkas pada saat Anda menjalankan `git add` inilah yang akan dimasukkan ke dalam sejarah snapshot. Anda mungkin ingat bahwa ketika Anda menjalankan `git`

init sebelumnya, Anda melanjutkannya dengan `git add` (nama berkas) - yang akan mulai dipantau di direktori Anda. Perintah `git add` ini mengambil alamat dari berkas ataupun direktori; jika sebuah direktori, perintah tersebut akan menambahkan seluruh berkas yang berada di dalam direktori secara rekursif.

Memasukan berkas baru di edit ke dalam stage yang dibuat

Mari kita ubah sebuah berkas yang sudah terpantau. Jika Anda mengubah berkas yang sebelumnya terpantau bernama `benchmarks.rb` dan kemudian menjalankan perintah `status` lagi, Anda akan mendapatkan keluaran kurang lebih seperti ini:

```
git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   README \\}
Changes not staged for commit:
  (use "git add <file>..." to update what will
   ~~~modified: benchmarks.rb
```

Berkas `benchmarks.rb` terlihat di bawah bagian yang bernama "Changes not staged for commit" - yang berarti bahwa sebuah berkas terpantau telah berubah di dalam direktori kerja namun belum masuk ke area stage. Untuk memasukkannya ke area stage, Anda menjalankan perintah `git add` (perintah ini adalah perintah multiguna - Anda menggunakannya untuk mulai memantau berkas baru, untuk memasukkannya ke area stage, dan untuk melakukan hal lain seperti menandai berkas terkonflik menjadi terpecahkan). Mari kita sekarang jalankan `git add` untuk

memasukkan berkas rb ke dalam area stage, dan jalankan git status lagi:

```
git add benchmarks.rb
git status
On branch master \\}
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
new~file:~    README
modified:~    benchmarks.rb
```

Kedua file sekarang berada di area stage dan akan masuk ke dalam commit Anda berikutnya. Pada saat ini, semisal Anda teringat satu perubahan yang Anda ingin buat di benchmarks.rb sebelum Anda lakukan commit. Anda buka berkas tersebut kembali dan melakukan perubahan tersebut, dan Anda siap untuk melakukan commit. Namun, mari kita coba jalankan git status kembali:

```
vim benchmarks.rb
git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

new~file:~    README
modified:~    benchmarks.rb
Changes not staged for commit:
  (use "git add <file>..." to update what will
modified:~    benchmarks.rb
```

Git memasukkan berkas ke area stage tepat seperti ketika Anda menjalankan perintah git add. Jika Anda commit

sekarang, versi `benchmarks.rb` pada saat Anda terakhir lakukan perintah `git add`-lah yang akan masuk ke dalam `commit`, bukan versi berkas yang saat ini terlihat di direktori kerja Anda ketika Anda menjalankan `git commit`. Jika Anda mengubah sebuah berkas setelah Anda menjalankan `git add`, Anda harus menjalankan `git add` kembali untuk memasukkan versi berkas terakhir ke dalam area `stage`:

```
$ git add benchmarks.rb
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD ;file;..." to unstage)
```

```
#
```

```
# new file: README
```

```
# modified: benchmarks.rb
```

```
#
```

6.1.2 Mengabaikan berkas

Terkadang, Anda memiliki sekumpulan berkas yang Anda tidak ingin Git tambahkan secara otomatis atau bahkan terlihat sebagai tak-terpantau. Biasanya berkas hasil keluaran seperti berkas `log` atau berkas yang dihasilkan oleh sistem `build` Anda. Dalam kasus ini, Anda dapat membuat sebuah berkas bernama `.gitignore` yang berisi pola dari berkas terabaikan. Berikut adalah sebuah contoh isi dari berkas

`.gitignore`:

`$ cat .gitignore`

`*.[oa]`

`* ~`

Baris pertama memberitahu Git untuk mengabaikan semua file yang berakhiran `.o` atau `.a` - berkas object dan arsip yang mungkin dihasilkan dari kompilasi kode Anda. Baris kedua memberitahu Git untuk mengabaikan semua file yang berakhiran dengan sebuah tilde (`~`), yang biasanya digunakan oleh banyak aplikasi olah-kata seperti Emacs untuk menandai berkas sementara. Anda juga dapat memasukkan direktori `log`, `tmp` ataupun `pid`; dokumentasi otomatis; dan lainnya. Menata berkas `.gitignore` sebelum Anda mulai bekerja secara umum merupakan ide yang baik sehingga Anda tidak secara tak-sengaja melakukan commit terhadap berkas yang sangat tidak Anda inginkan berada di dalam repositori Git.

Aturan untuk pola yang dapat Anda gunakan di dalam berkas `.gitignore` adalah sebagai berikut:

Baris kosong atau baris dimulai dengan `#` akan diabaikan.

Pola glob standar dapat digunakan.

Anda dapat mengakhiri pola dengan sebuah slash (`/`) untuk menandai sebuah direktori.

Anda dapat menegaskan sebuah pola dengan memulainya menggunakan karakter tanda seru (`!`).

Pola Glob adalah seperti regular expression yang disederhanakan yang biasanya digunakan di shell. Sebuah asterisk (*) berarti 0 atau lebih karakter; [abc] terpasangkan dengan karakter apapun yang ditulis dalam kurung siku (dalam hal ini a, b, atau c); sebuah tanda tanya (?) terpasangkan dengan sebuah karakter; dan kurung siku yang melingkupi karakter yang terpisahkan dengan sebuah tanda hubung([0-9]) terpasangkan dengan karakter apapun yang berada diantaranya (dalam hal ini 0 hingga 9).

Berikut adalah contoh lain dari isi berkas .gitignore:

```
# sebuah komentar akan diabaikan
```

```
# abaikan berkas .a
```

```
*.a
```

```
# tapi pantau lib.a, walaupun Anda abaikan berkas .a di atas
```

```
!lib.a
```

```
# hanya abaikan berkas TODO yang berada di root, bukan di subdir/TODO
```

```
/TODO
```

```
# abaikan semua berkas di dalam direktori build/
```

```
build/
```

```
# abaikan doc/notes.txt, tapi bukan doc/server/arch.txt
```

```
doc/*.txt
```

melihat perubahan di dalam stage dan di luar stage

Jika perintah git status terlalu kabur untuk Anda - Anda ingin mengetahui secara pasti apa yang telah berubah, bukan hanya berkas mana yang berubah - Anda dapat menggunakan perintah git diff. Kita akan bahas git diff secara lebih detil nanti; namun Anda mungkin menggunakannya paling sering untuk menjawab 2 pertanyaan berikut: Apa yang Anda ubah tapi belum dimasukkan ke area stage? Dan apa yang telah Anda ubah yang akan segera Anda commit? Walaupun git status menjawab pertanyaan tersebut secara umum, git diff menunjukkan kepada Anda dengan tepat baris yang ditambahkan dan dibuang - dalam bentuk patch-nya.

Mari kita anggap Anda mengubah dan memasukkan berkas README ke area stage lagi dan kemudian mengubah berkas benchmarks.rb tanpa memasukkannya ke area stage. Jika Anda jalankan perintah status Anda, Anda akan sekali lagi melihat keluaran seperti berikut:

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD ;file;..." to unstage)
```

```
#
```

```
# new file: README
```

```
#
```

```
# Changes not staged for commit:
```

```
# (use "git add ;file;" to update what will be committed)
```

```
#
```

```
# modified: benchmarks.rb
```

```
#
```

Untuk melihat apa yang Anda telah ubah namun belum masuk ke area stage, ketikkan git diff tanpa argumen lainnya.

```
$ git diff
```

```
diff -git a/benchmarks.rb b/benchmarks.rb
```

```
index 3cb747f..da65585 100644
```

```
— a/benchmarks.rb
```

```
+++ b/benchmarks.rb
```

```
@@ -36,6 +36,10 @@ def main
```

```
@commit.parents[0].parents[0].parents[0]
```

```
end
```

```
+   run _code(x, 'commits 1') do
```

```
+     git.commits.size
```

```
+   end
```

```
+
```



```
run _code(x, 'commits 2') do  
  
log = git.commits('master', 15)  
  
log.size
```

Perintah di atas membandingkan apa yang ada di direktori kerja Anda dengan apa yang ada di area stage. Hasilnya memberitahu Anda bahwa perubahan yang Anda ubah namun belum masuk ke area stage.

Jika Anda ingin melihat apa yang telah Anda masukkan ke area stage yang nantinya akan masuk ke commit Anda berikutnya, Anda dapat menggunakan `git diff --cached`. (Di Git versi 1.6.1 atau yang lebih tinggi, Anda dapat juga menggunakan `git diff --staged`, yang mungkin lebih mudah untuk diingat). Perintah ini membandingkan area stage Anda dengan commit Anda terakhir:

```
$ git diff --cached  
diff --git a/README b/README  
new file mode 100644  
index 00000000..03902a1  
--- /dev/null  
+++ b/README2  
@@ -0,0 +1,5 @@  
+grit
```

+ by Tom Preston-Werner, Chris Wanstrath

+ <http://github.com/mojombo/grit>

+

+Grit is a Ruby library for extracting information from a Git repository

Satu hal penting yang harus dicatat adalah bahwa git diff saja tidak memperlihatkan semua perubahan yang telah Anda lakukan sejak terakhir Anda commit - hanya perubahan yang belum masuk ke area stage saja. Mungkin agak sedikit membingungkan, karena jika Anda telah memasukkan semua perubahan ke area stage, git diff akan memberikan keluaran kosong.

Sebagai contoh lain, jika Anda memasukkan berkas benchmarks.rb ke area stage dan kemudian meng-editnya, Anda dapat menggunakan git diff untuk melihat perubahan di berkas tersebut yang telah masuk ke area stage dan perubahan yang masih di luar area stage:

```
$ git add benchmarks.rb
```

```
$ echo ' # test line' && benchmarks.rb
```

```
$ git status
```

```
# On branch master
```

```
#
```

```
# Changes to be committed:
```

```
#
```

```
#   modified:   benchmarks.rb
```

```
#
```

```
# Changes not staged for commit:
```

#

modified: benchmarks.rb

#

Sekarang Anda dapat menggunakan git diff untuk melihat apa saja yang masih belum dimasukkan ke area stage:

```
$ git diff
```

```
diff -git a/benchmarks.rb b/benchmarks.rb
```

```
index e445e28..86b2f7c 100644
```

```
— a/benchmarks.rb
```

```
+++ b/benchmarks.rb
```

```
@@ -127,3 +127,4 @@ end
```

```
main()
```

```
# #pp Grit::GitRuby.cache _client.stats
```

```
+ # test line
```

dan git diff -cached untuk melihat apa yang telah Anda masukkan ke area stage sejauh ini:

```
$ git diff -cached
```

```
diff -git a/benchmarks.rb b/benchmarks.rb
```

```
index 3cb747f..e445e28 100644
```

```
— a/benchmarks.rb
```

```
+++ b/benchmarks.rb
```

```
@@ -36,6 +36,10 @@ def main
```

```
@commit.parents[0].parents[0].parents[0]
```

```
end
```

```
+   run _code(x, 'commits 1') do
```

```
+     git.commits.size
```

```
+   end
```

```
+ 
```

```
run _code(x, 'commits 2') do
```

```
log = git.commits('master', 15)
```

```
log.size
```

6.1.3 Comit perubahan atau yang telah di edit

Sekarang setelah area stage Anda tertata sebagaimana yang Anda inginkan, Anda dapat melakukan commit terhadap perubahan Anda. Ingat bahwa apapun yang masih di luar area stage - berkas apapun yang Anda telah buat atau ubah yang belum Anda jalankan git add terhadapnya sejak terakhir Anda edit - tidak akan masuk ke dalam commit ini. Perubahan tersebut akan tetap sebagai berkas berubah di cakram Anda. Dalam hal ini, saat terakhir Anda jalankan git status, Anda telah melihat bahwa semuanya telah masuk

ke stage, sehingga Anda siap untuk melakukan commit dari perubahan Anda. Cara termudah untuk melakukan commit adalah dengan mengetikkan git commit:

```
$ git commit
```

Dengan melakukan ini, aplikasi olahkata pilihan Anda akan dijalankan (Ini ditata oleh variabel lingkungan \$EDITOR di shell Anda - biasanya vim atau emacs, walaupun Anda dapat mengkonfigurasinya dengan apapun yang Anda inginkan

```
# Please enter the commit message for your changes. Lines
starting
```

```
# with ' #' will be ignored, and an empty message aborts
the commit.
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD ;file;..." to unstage)
```

```
#
```

```
#    new file:   README
```

```
#    modified:  benchmarks.rb
```

```
~
```

```
~
```

```
~
```

```
".git/COMMIT_EDITMSG" 10L, 283C
```

Anda dapat melihat bahwa pesan commit standar berisi keluaran terakhir dari perintah git status yang terkomentari dan sebuah baris kosong di bagian atas. Anda dapat membuang komentar-komentar ini dan mengetikkan pesan commit Anda, atau Anda dapat membiarkannya untuk membantu Anda mengingat apa yang akan Anda commit. (Untuk pengingat yang lebih eksplisit dari apa yang Anda ubah, Anda dapat menggunakan opsi -v di perintah git commit. Melakukan hal ini akan membuat diff dari perubahan Anda di dalam olahkata sehingga Anda dapat melihat secara tepat apa yang telah Anda lakukan). Ketika Anda keluar dari olahkata, Git akan membuat commit Anda dengan pesan yang Anda buat (dengan bagian terkomentari dibuang).

Cara lainnya, Anda dapat mengetikkan pesan commit Anda sebaris dengan perintah commit dengan mencantumkannya setelah tanda -m seperti berikut:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
```

```
[master]: created 463dc4f: "Fix benchmarks for speed"
```

```
2 files changed, 3 insertions(+), 0 deletions(-)
```

```
create mode 100644 README
```

Sekarang Anda telah membuat commit pertama Anda ~ Anda dapat lihat bahwa commit tersebut telah memberi Anda beberapa keluaran tentang dirinya sendiri: cabang apa yang Anda jadikan target commit (master) berapa banyak berkas yang diubah, dan statistik tentang jumlah baris yang ditambah dan dibuang dalam commit tersebut.

Ingat bahwa commit merekam snapshot yang Anda telah tata di area stage. Apapun yang tidak Anda masukkan ke area stage akan tetap berada di tempatnya, tetap dalam keadaan terubah; Anda dapat melakukan commit lagi untuk memasukkannya ke dalam sejarah Anda. Setiap saat Anda melakukan sebuah commit, Anda merekamkan sebuah snapshot dari proyek Anda yang bisa Anda kembalikan atau Anda bandingkan nantinya.

Melewatkan area stage

Walaupun dapat menjadi sangat berguna untuk menata commit tepat sebagaimana Anda inginkan, area stage terkadang sedikit lebih kompleks dibandingkan apa yang Anda butuhkan di dalam alurkerja Anda. Jika Anda ingin melewati area stage, Git menyediakan sebuah jalan pintas sederhana. Dengan memberikan opsi -a ke perintah git commit akan membuat Git secara otomatis menempatkan setiap berkas yang telah terpantau ke area stage sebelum melakukan commit, membuat Anda dapat melewati bagian git add:

```
$ git status
```

```
# On branch master
```

```
#
```

```
# Changes not staged for commit:
```

```
#
```

```
#   modified:   benchmarks.rb
```

```
#
```

```
$ git commit -a -m 'added new benchmarks'
```

```
[master 83e38c7] added new benchmarks
```

```
1 files changed, 5 insertions(+), 0 deletions(-)
```

Perhatikan bagaimana Anda tidak perlu menjalankan `git add` terhadap berkas `benchmarks.rb` dalam hal ini sebelum Anda commit.

6.1.4 Menghapus berkas

Untuk menghapus sebuah berkas dari Git, Anda harus menghapusnya dari berkas terpantau (lebih tepatnya, menghapus dari area stage) dan kemudian commit. Perintah `git rm` melakukan hal tadi dan juga menghapus berkas tersebut dari direktori kerja Anda sehingga Anda tidak melihatnya sebagai berkas yang tak terpantau nantinya.

Jika Anda hanya menghapus berkas dari direktori kerja Anda, berkas tersebut akan muncul di bagian "Changes not staged for commit" (yaitu, di luar area stage) dari keluaran `git status` Anda:

```
$ rm grit.gemspec
```

```
$ git status
```

```
# On branch master
```

```
#
```

```
# Changes not staged for commit:
```



```
# (use "git add/rm ;file;..." to update what will be  
committed)
```

```
#
```

```
# deleted: grit.gemspec
```

```
#
```

Kemudian, jika Anda jalankan `git rm`, Git akan memasukkan penghapusan berkas tersebut ke area stage:

```
$ git rm grit.gemspec
```

```
rm 'grit.gemspec'
```

```
$ git status
```

```
# On branch master
```

```
#
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD ;file;..." to unstage)
```

```
#
```

```
# deleted: grit.gemspec
```

```
#
```

Pada saat Anda commit nantinya, berkas tersebut akan hilang dan tidak lagi terpantau. Jika Anda mengubah berkas tersebut dan menambahkannya lagi ke index, Anda harus memaksa penghapusannya dengan menggunakan opsi `-f`. Ini adalah fitur keamanan (safety) untuk mencegah ketidaksengajaan penghapusan terhadap data yang belum

terekam di dalam snapshot dan tak dapat dikembalikan oleh Git.

Hal berguna lain yang Anda dapat lakukan adalah untuk tetap menyimpan berkas di direktori kerja tetapi menghapusnya dari area kerja. Dengan kata lain, Anda mungkin ingin tetap menyimpan berkas tersebut di dalam cakram keras tetapi tidak ingin Git untuk memantaunya lagi. Hal ini khususnya berguna jika Anda lupa untuk menambahkan sesuatu ke berkas `.gitignore` Anda dan secara tak-sengaja menambahkannya, seperti sebuah berkas `log` yang besar, atau sekumpulan berkas hasil kompilasi `.a`. Untuk melakukan ini, gunakan opsi `-cached`:

```
$ git rm -cached readme.txt
```

Anda dapat menambahkan nama berkas, direktori, dan pola glob ke perintah `git rm`. Ini berarti Anda dapat melakukan hal seperti

```
$ git rm log/ \*.log
```

Perhatikan karakter backslash (`\`) di depan tanda `*`. Ini dibutuhkan agar Git juga meng-ekspansi nama berkas sebagai tambahan dari ekspansi nama berkas oleh shell Anda. Perintah ini menghapus semua berkas yang memiliki ekstensi `.log` di dalam direktori `log/`. Atau, Anda dapat melakukannya seperti ini:

```
$ git rm \* ~
```

Perintah ini akan membuang semua berkas yang berakhiran dengan `~`.

6.1.5 Memindahkan berkas

Tidak seperti kebanyakan sistem VCS lainnya, Git tidak secara eksplisit memantau perpindahan berkas. Jika Anda mengubah nama berkas di Git, tidak ada metada yang tersimpan di Git yang menyatakan bahwa Anda mengubah nama berkas tersebut. Namun demikian, Git cukup cerdas untuk menemukannya berdasarkan fakta yang ada - kita akan membicarakan tentang mendeteksi perpindahan berkas sebentar lagi.

Untuk itu agak membingungkan bahwa Git memiliki perintah `mv`. Jika Anda hendak mengubah nama berkas di Git, Anda dapat menjalankan seperti berikut

```
$ git mv file _from file _to
```

dan itu berjalan baik. Bahkan, jika Anda menjalankannya seperti ini kemudian melihat ke status, Anda akan melihat bahwa Git menganggapnya sebagai perintah pengubahan nama berkas.

```
$ git mv README.txt README
```

```
$ git status
```

```
# On branch master
```

```
# Your branch is ahead of 'origin/master' by 1 commit.
```

```
#
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD ;file;..." to unstage)
```

```
#
```

```
# renamed: README.txt -> README
```

```
#
```

Namun sebetulnya hal ini serupa dengan menjalankan perintah-perintah berikut:

```
$ mv README.txt README
```

```
$ git rm README.txt
```

```
$ git add README
```

Git mengetahui secara implisit bahwa perubahan yang terjadi merupakan proses pengubahan nama, sehingga sebetulnya tidaklah terlalu bermasalah jika Anda mengubah nama sebuah berkas dengan cara ini atau dengan menggunakan perintah `mv`. Satu-satunya perbedaan utama adalah `mv` berjumlah satu perintah dan bukannya tiga - yang membuat fungsi ini lebih nyaman digunakan. Lebih penting lagi, Anda sebetulnya dapat menggunakan alat apapun yang Anda suka untuk mengubah nama berkas, tinggal tambahkan perintah `add/rm` di bagian akhir, sesaat sebelum Anda melakukan `commit`.

PERFORM CHANGES

7.1 Dasar Git

Git adalah *version control system* yang digunakan para developer untuk mengembangkan software secara bersama-sama. Fungsi utama git yaitu mengatur versi dari source code program anda dengan mengasih tanda baris dan code mana yang ditambah atau diganti. Git ini sebenarnya memudahkan programmer untuk mengetahui perubahan source codenya daripada harus membuat file baru seperti *Program.java*, *ProgramRevisi.java*, *ProgramRevisi2.java*, *ProgramFix.java*. Selain itu, dengan git kita tak perlu khawatir code yang kita kerjakan bentrok, karena setiap developer bias membuat branch sebagai *workspacenya*. Fitur yang tak kalah keren lagi, pada git kita bisa memberi komentar pada source code yang telah ditambah/diubah, hal ini

mempermudah developer lain untuk tahu kendala apa yang dialami developer lain.

Untuk mengetahui bagaimana menggunakan git, berikut perintah-perintah dasar git:

1. Git init : untuk membuat *repository* pada file lokal yang nantinya ada folder *.git*
2. Git status : untuk mengetahui status dari *repository* lokal
3. Git add : menambahkan file baru pada *repository* yang dipilih
4. Git commit : untuk menyimpan perubahan yang dilakukan, tetapi tidak ada perubahan pada *remote repository*.
5. Git push : untuk mengirimkan perubahan file setelah di commit ke *remote repository*.
6. Git branch : melihat seluruh *branch* yang ada pada *repository*
7. Git checkout : menukar *branch* yang aktif dengan *branch* yang dipilih
8. Git merge : untuk menggabungkan *branch* yang aktif dan *branch* yang dipilih
9. Git clone : membuat Salinan *repository* lokal

Contoh dari *software version control system* adalah github, bitbucket, snowy evening, dan masih banyak lagi. Jika anda sebagai developer belum mengetahui fitur git ini, maka anda wajib mencoba dan memakainya. Karena banyak manfaat yang akan didapat dengan git ini.

Git itu bukanlah sebuah bahasa seperti halnya HTML, CSS atau Js bukan pula sebuah konsep atau aturan baku dalam pemrograman, melainkan sebuah software yang berfungsi untuk mengatur source code dari aplikasi yang sedang anda buat.

Fungsi utamanya adalah untuk mengatur versi dari source code anda, menambahkan tanda/checkpoint ketika terjadi perubahan pada kode Anda dan tentunya akan mempermudah Anda untuk tetap mengetahui apa saja yang berubah dari source code Anda.

Sebagai contoh, misalkan Anda sedang membangun sebuah website, dan anda akan menambahkan beberapa fitur dalam website anda. Agar tidak membingungkan nantinya anda membuat sebuah catatan terhadap apa yang telah anda lakukan seperti :

01-04-2014 Memulai Project Website, File HTML & CSS Dasar

02-04-2014 Penambahan Menu Utama

03-04-2014 Penambahan Layout Standar

04-04-2014 Penambahan Fitur Pengubah Layout

Pada contoh diatas kita menggunakan tanggal sebagai tanda akan apa yang telah kita lakukan, dengan demikian kita bisa tahu kapan perubahan terjadi dan apa perubahan yang dilakukan. Dan dalam Git semua itu bisa dilakukan dengan mudah dan asyiknya jika Anda merusak kode sehingga membuat aplikasi error, maka anda dapat mengembalikan kode tersebut berdasarkan pada tanda/tanggal dimana kode masih normal, lebih mirip seperti restore point.

Git juga tidak hanya digunakan untuk perorangan, beberapa orang pun dapat bekerja secara bersamaan mengerjakan kode Anda dan Anda masih memiliki kontrol penuh terhadap kode Anda, Anda bisa menambahkan kode yang ditambahkan oleh orang lain atau mengabaikannya sama sekali. oleh karena itu Git sering digunakan sebagai pengatur dalam projek kolaborasi dimana tidak hanya satu orang yang mengerjakan sebuah kode tapi beberapa orang sekaligus yang mengerjakan kode tersebut.

Git Punya Integritas

Segala sesuatu pada Git akan melalui proses checksum terlebih dahulu sebelum disimpan yang kemudian direferensikan oleh hasil checksum tersebut. Hal ini berarti tidak mungkin melakukan perubahan terhadap berkas manapun tanpa diketahui oleh Git. Fungsionalitas ini dimiliki oleh Git pada level terendahnya dan ini merupakan bagian tak terpisahkan dari filosofi Git. Anda tidak akan kehilangan informasi atau file yang tidak dimiliki oleh Git.

Mekanisme checksum yang digunakan oleh Git adalah SHA-1 hash. Ini merupakan sebuah susunan string yang terdiri dari 40 karakter heksadesimal (0 sampai 9 dan a sampai f) dan dihitung berdasarkan bentuk dari suatu berkas atau struktur pada Git. sebuah hash SHA-1 seperti berikut:

Anda akan melihat seperti ini pada berbagai tempat di Git. Faktanya, Git tidak memiliki nama file pada basisdatanya, nilai hash dari isi berkas.

Direktori Git adalah dimana Git menyimpan metadata dan database objek untuk proyek anda. Ini adalah bahagian penting dari Git, dan inilah yang disalin saat anda melakukan kloning sebuah repositori dari komputer lain.

Direktori kerja adalah sebuah checkout tunggal dari satu versi dari proyek. File-berkas ini kemudian ditarik keluar dari basisdata yang terkompresi dalam direktori Git dan disimpan pada disk untuk anda pakai atau modifikasi.

Alur kerja dasar Git adalah seperti ini:

Anda mengubah berkas dalam direktori kerja anda.

Anda membawa ke tahap, menambahkan snapshot-nya ke area stage.

Anda melakukan komit

Aplikasi git dapat diakses melalui terminal/Command Prompt jadi silahkan buka Terminal/Command

Prompt dan ketik `git --version` untuk melihat versi dari git yang terinstall dan untuk mengkonfirmasi jika proses instalasi berjalan mulus.

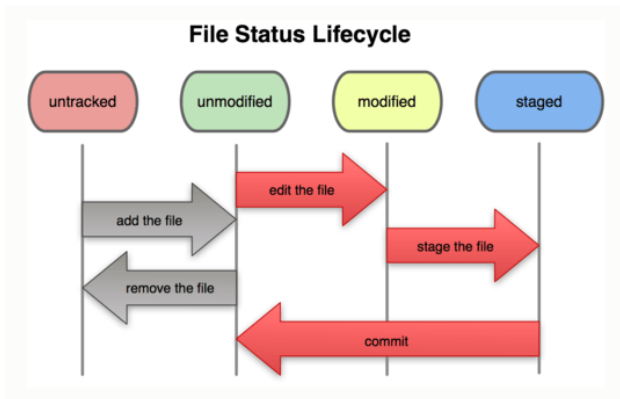
Untuk mempelajari git ada baiknya kita buat sebuah studi kasus, pertama-tama kita buat folder baru untuk proyek kita dan kita beri nama latihan-git. Anda bisa melakukannya melalui aplikasi Windows Explorer/File Manager/Finder namun kali ini untuk mengasah kemampuan Terminal/Command Prompt Anda silahkan ketikkan perintah berikut:

```
mkdir latihan-git
```

setelah folder latihan-git dibuat, kita navigasikan terminal ke dalam folder tersebut dengan mengetikkan perintah:

```
cd latihan-git
```

▪ File Status Lifecycle



Gambar 7.1 File Status Lifecycle

Git Init

Agar proyek kita dapat diatur oleh git, maka kita perlu melakukan inisiasi git terlebih dahulu, caranya dengan mengetikkan perintah :

git init

Perintah tersebut akan membuat folder .git dan didalamnya berisi file-file yang akan digunakan oleh Git untuk mengatur dan mengontrol project kita.

Git Status

Untuk mengetahui status dari git, ketikkan perintah :

git status

Anda akan mendapatkan keterangan seperti berikut:

On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)

Dari sana kita bisa mengetahui bahwa kita berada dalam branch master, dan kita telah melakukan initial commit, mengenai branch dan commit semuanya akan saya bahas nanti.

Sekarang mari kita buat file baru, misalkan buat file index.html lalu tambahkan kode berikut ke dalamnya

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Belajar Git</title>
</head>
<body>
<p>Hello Git</p>
</body>
</html>
```

Git Add

Jika anda mengetikkan kembali perintah git status maka yang anda dapatkan kurang lebih seperti berikut :

On branch master

Initial commit

Untracked files:

(use "git add .." to include in what will be committed)

index.html

nothing added to commit but untracked files present (use "git add" to track)

Dari informasi diatas kita mendapatkan informasi bahwa Anda file baru yang belum terlacak. Kita perlu menambahkan file tersebut ke dalam git agar dapat dilacak perubahan-perubahan yang terjadi. Untuk itu anda dapat mengetikkan perintah:

```
git add index.html
```

Dengan demikian anda telah menambahkan file index.html kedalam git agar bisa dimonitor/diawasi nantinya. Dan jika anda kembali mengetikkan git status yang akan anda dapatkan adalah:

Changes to be committed:

(use "git rm --cached .." to unstage)

new file: index.html

Git Commit

Anda telah menambahkan file baru, namun anda belum melakukan commit. Oke, kembali ke contoh kasus dalam pembuka artikel ini, commit merupakan istilah untuk menandai terhadap perubahan yang telah anda lakukan, dalam contoh sebelumnya kita menandainya dengan tanggal dan keterangan singkat.

Nah untuk menandai setiap perubahan yang telah anda lakukan dan anda ingin agar git mengingatnya Anda harus melakukan commit terlebih dahulu. Untuk melakukan commit ketikkan perintah berikut:

```
git commit -m "Added index.html"
```

Dalam perintah diatas harus disertakan juga pesan commit, tanda -m digunakan untuk menambahkan pesan commit dan

teks selanjutnya adalah pesan commit kita yakni " Added index.html ". Jika semau anda lakukan dengan benar maka anda akan mendapatkan keterangan seperti berikut:

```
[master (root-commit) 0181f7b] Added index.html  
1 file changed, 10 insertions(+)  
create mode 100644 index.html  
Branch
```

Misalkan anda ingin menambahkan suatu fitur, namun anda tidak mau kode yang ada sekarang rusak karena fitur yang akan anda tambahkan masih belum stabil, Dalam Git anda dapat membuat branch terlebih dahulu. Branch ini bisa diartikan sebagai cabang dari branch master. segala perubahan yang anda lakukan pada branch yang anda buat tidak akan berpengaruh pada branch lainnya.

Sebagai contoh, kita buat branch dengan nama branch " fix-css " dengan mengetikkan perintah:

```
git branch fix-css
```

Jika perintah dijalankan dengan benar maka ketika anda mengetikkan perintah git branch akan muncul branch-branch yang telah dibuat.

```
fix-css
```

```
* master
```

tanda Bintang menandakan bahwa anda sedang bekerja pada branch master, untuk berpindah ke branch yang baru saja dibuat (fix-css) ketikkan perintah berikut:

```
git checkout fix-css
```

Jika perintah di atas benar, maka akan ada pemberitahuan seperti berikut:

```
Switched to branch 'fix-css'
```

Github merupakan situs sharing code dan menggunakan git sebagai SCM-nya. Mungkin Anda pernah mendownload beberapa library/source code dari situs ini. Kini anda tahu mengapa kebanyakan source code dapat anda temui di github.

7.1.1 Apa itu GIT?

Git adalah version control, dengan menggunakan Git kita dapat menyimpan tiap perubahan yang kita lakukan pada file. Seperti menggunakan "undo" tetapi dapat digunakan pada seluruh file kita dan kita mempunyai log pada tiap perubahan yang kita simpan,

Hal ini sangat berguna pada production environment, karena dengan adanya versioning, kita dapat dengan mudah melakukan rollback. Selain itu Git juga sangat membantu dalam proses pengembangan aplikasi. Salah satu platform Git yang terkenal adalah GitHub. Kita akan menggunakan GitHub untuk artikekl berikut.

7.2 Repository

Di Git Repository biasanya digunakan untuk mengorganisir sebuah proyek. Proyek dapat berisi files, folder, gambar, video dan lainnya yang dibutuhkan untuk sebuah project. Kami menyarankan untuk menambahkan file README yang bertujuan untuk memberi informasi terkait isi dari project tersebut. Kita dapat mengasumsikan Repository sebagai folder untuk menyimpan project kita. Satu berada di Local, dan yang satu berada di Remote disini kita menggunakan GitHub.

Seperti Ini Perform Changes

Jerry mengkloning repositori dan memutuskan untuk menerapkan operasi string dasar. Jadi dia menciptakan file

string.c. Setelah menambahkan isinya, string.c akan terlihat seperti berikut:

```
#include <stdio.h>
int my_strlen (char * s)
{
    char * p = s;
    sementara (* p)
        ++ p;
    return (p - s);
}
int main (void)
{
    int i;
    char * s [] =
    {
        "Git tutorial",
        "Tutorial Point"
    };
    untuk (i = 0; i <2; ++ i)
        printf ("panjang string %s = %d \n", s [i], my
        _strlen (s [i]));
    kembali 0;
}
```

Dia menyusun dan menguji kodenya dan semuanya berjalan baik. Sekarang, dia bisa menambahkan perubahan ini ke repositori dengan aman.

Git menambahkan operasi menambahkan file ke area stage.

```
[jerry @ CentOS project] $ git status -s
?? tali
?? string.c
[jerry @ CentOS project] $ git add string.c
```

Git menunjukkan tanda tanya sebelum nama file. Jelas, file-file ini bukan bagian dari Git, dan karena itulah Git tidak tahu apa yang harus dilakukan dengan file-file ini. Itu sebabnya, Git menunjukkan tanda tanya sebelum nama file.

Jerry telah menambahkan file ke area penyimpanan, perintah status git akan menampilkan file yang ada di area stage.

```
[jerry @ CentOS project] $ git status -s  
String.c  
?? tali
```

Untuk melakukan perubahan, dia menggunakan perintah komit git diikuti dengan opsi -m. Jika kita menghilangkan opsi -m. Git akan membuka text editor dimana kita bisa menulis multiline commit message.

```
[jerry @ CentOS project] $ git commit -m 'Implementasikan fungsi my _strlen'
```

Perintah di atas akan menghasilkan hasil sebagai berikut:

```
[master cbe1249] Melaksanakan fungsi my _strlen  
1 file berubah, 24 sisipan (+), 0 penghapusan (-)  
buat mode 100644 string.c
```

Setelah berkomitmen untuk melihat rincian log, dia menjalankan perintah git log. Ini akan menampilkan informasi dari semua commit dengan commit komit mereka, commit author, commit date dan SHA-1 hash of commit.

```
[jerry @ CentOS project] $ git log
```

Perintah di atas akan menghasilkan hasil sebagai berikut:

```
melakukan cbe1249b140dad24b2c35b15cc7e26a6f02d2277  
Penulis: Jerry Mouse <jerry@tutorialspoint.com>  
Tanggal: Rabu Sep 11 08:05:26 2013 +0530  
Diimplementasikan fungsi my _strlen
```

```
komit 19ae20683fc460db7d127cf201a1429523b0e31  
Penulis: Tom Cat <tom@tutorialspoint.com>  
Tanggal: Rabu Sep 11 07:32:56 2013 +0530
```

Komit awal

Jerry memeriksa versi terbaru dari repositori dan mulai mengerjakan sebuah proyek. Dia membuat file `array.c` di dalam direktori `trunk`.

```
[jerry @ CentOS ~] $ cd project _repo / trunk /
[jerry @ CentOS trunk] $ cat array.c
Perintah di atas akan menghasilkan hasil berikut.
#include <stdio.h>
#define MAX 16
int main (void) {
    int i, n, arr [MAX];
    printf ("Masukkan jumlah elemen:");
    scanf ("% d", & n);
    printf ("Enter the elements \n");
    untuk (i = 0; i < n; ++ i) scanf ("% d", & arr [i]);
    printf ("Array memiliki elemen berikut \n");
    untuk (i = 0; i < n; ++ i) printf (" | % d |", arr [i]);
    printf ("\n");
    kembali 0;
}
```

Dia ingin menguji kodenya sebelum melakukan.

```
[jerry @ CentOS trunk] $ buat array
cc array.c -o array
[jerry @ CentOS trunk] $ ./array
Masukkan jumlah elemen: 5
Masukkan elemen
```

```
1
2
3
4
5
```

Array memiliki elemen berikut

```
| 1 | | 2 | | 3 | | 4 | | 5 |
```

Dia menyusun dan menguji kodenya dan semuanya berjalan seperti yang diharapkan, sekarang saatnya melakukan perubahan.

```
[jerry @ CentOS trunk] status $ svn  
? array.c  
? array
```

Subversion menunjukkan '?' di depan nama file karena tidak tahu apa yang harus dilakukan dengan file-file ini. Sebelum komit, Jerry perlu menambahkan file ini ke daftar perubahan yang tertunda.

```
[jerry @ CentOS trunk] $ svn tambahkan array.c  
  
Sebuah array.c
```

Mari kita periksa dengan operasi 'status'. Subversion menunjukkan A sebelum array.c, artinya, file tersebut berhasil ditambahkan ke daftar perubahan yang tertunda.

```
[jerry @ CentOS trunk] status $ svn  
A array
```

Sebuah array.c

Untuk menyimpan file array.c ke repositori, gunakan perintah komit dengan opsi -m diikuti oleh pesan komit. Jika Anda menghilangkan opsi -m, Subversion akan menampilkan editor teks tempat Anda bisa mengetikkan pesan multi-baris.

```
[jerry @ CentOS trunk] $ svn commit -m "Ini  
tial commit"  
Menambahkan trunk / array.c  
Mengirimkan data file  
Komitmen revisi 2.
```

Sekarang file `array.c` berhasil ditambahkan ke repositori, dan nomor revisi bertambah satu.

BAB 8

REVIEW CHANGES

8.1 Dasar Git

Jadi, sebenarnya apa yang dimaksud dengan Git? Ini adalah bagian penting untuk dipahami, karena jika anda memahami apa itu Git dan cara kerja, maka dapat dipastikan anda dapat menggunakan Git secara efektif dengan mudah. Selama menerapkan Git, cobalah untuk menggantikan VCS lain yang mungkin sudah anda kenal sebelumnya, misalnya Subversion dan Perforce. Git sangat berbeda dengan sistem-sistem ini dalam hal simpan atau informasi yang digunakan, meski antar muka sangat mirip. Dengan memahami perbedaan ini diharapkan dapat membantu anda menghindari penggunaan saat menggunakan Git.

Snapshot, Bukan Perbedaan

Salah satu perbedaan yang mencolok antar Git dengan VCS lainnya (Subversion dan kawan-kawan) adalah dalam cara Git para datanya. Konsep konseptual,. Sistem seperti ini (CVS, Subversion, Bazaar, dan yang lainnya) informasi yang tersimpannya sebagai sekumpulan berkas dan perubahan yang terjadi pada berkas-berkas tersebut,

Git dianggap datanya sebagai sebuah kumpulan snapshot dari sebuah miniatur sistem. Setiap kali anda melakukan komit, atau melakukan perubahan pada proyek Git anda, pada butir Git anda secara otomatis. Agar efisien, jika berkas tidak mengalami perubahan, Git tidak akan menyimpan file tersebut pada hanya pada file yang sama yang sebelumnya telah disimpan.

Git Punya Integritas

Segala sesuatu pada Git akan melalui proses checksum terlebih dahulu sebelum disimpan yang kemudian direferensikan oleh hasil checksum tersebut. Hal ini berarti tidak mungkin melakukan perubahan terhadap berkas manapun tanpa diketahui oleh Git.

Fungsionalitas ini dimiliki oleh Git pada level terendahnya dan ini merupakan bagian tak terpisahkan dari filosofi Git. Anda tidak akan kehilangan informasi atau file yang tidak dimiliki oleh Git.

Mekanisme checksum yang digunakan oleh Git adalah SHA-1 hash. Ini merupakan sebuah susunan string yang terdiri dari 40 karakter heksadesimal (0 sampai 9 dan a sampai f) dan dihitung berdasarkan bentuk dari suatu berkas atau struktur pada pada Git. sebuah hash SHA-1 seperti berikut:

Anda akan melihat seperti ini pada berbagai tempat di Git. Faktanya, Git tidak memiliki nama file pada basisdatanya, pela nilai hash dari isi berkas.

Secara Umum Git Hanya Selesai Data

Bila anda melakukan operasi pada Git, hanya dari penambahan data pada basisdata Git. Sangat sulit membuat sistem melakukan sesuatu yang tidak bisa diurungkan atau membuatnya menghapus data dengan cara apa pun.

Seperti pada berbagai VCS, anda bisa kehilangan atau mengacaukan perubahan yang belum di-commit; namun jika anda melakukan komit pada Git, akan sangat sulit kehilangannya, apalagi jika anda secara teratur melakukan push basisdata anda pada repositori lain. Hal ini membuat Git menyenangkan karena kita dapat berexperimen tanpa kekhawatiran untuk mengacaukan proyek. Untuk lebih jelas dan dalam lagi tentang bagaimana Git menyimpan datanya dan bagaimana anda bisa mengembalikan yang hilang

Direktori Git adalah dimana Git menyimpan metadata dan database objek untuk proyek anda. Ini adalah bahagian penting dari Git, dan inilah yang disalin saat anda melakukan kloning sebuah repositori dari komputer lain.

Direktori kerja adalah sebuah checkout tunggal dari satu versi dari proyek. File-berkas ini kemudian ditarik keluar dari basisdata yang terkompresi dalam direktori Git dan disimpan pada disk untuk anda pakai atau modifikasi.

Alur kerja dasar Git adalah seperti ini:

Anda mengubah berkas dalam direktori kerja anda.

Anda membawa ke tahap, menambahkan snapshot-nya ke area stage.

Anda melakukan komit

Jika sebuah versi tertentu dari sebuah berkas telah ada di direktori git, ia dianggap 'berkomitmen'. Jika berkas diubah (sudah diubah) maka sudah ditambahkan ke area stage, maka itu adalah 'staged'. Dan jika berkas telah diubah sejak terakhir dilakukan check out belum ditambahkan ke area stage maka itu adalah 'modified'.

Pada, anda akan lebih banyak membahas mengenai keadaan-keadaan ini dan bagaimana anda dapat memanfaatkan

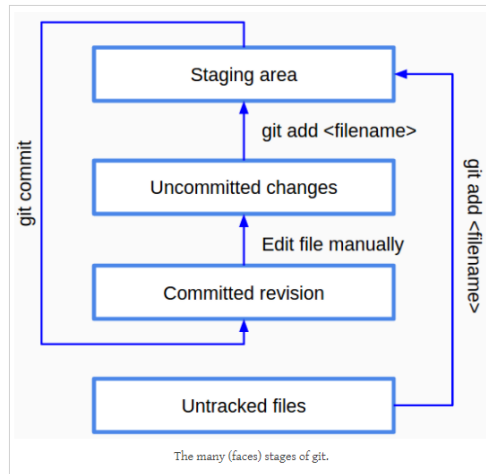
keadaan-keadaan yang bersangkutan dengan bagian 'bertahap'.

Git adalah *version control system* yang digunakan para developer untuk mengembangkan software secara bersama-sama. Fungsi utama git yaitu mengatur versi dari source code program anda dengan mengasih tanda baris dan code mana yang ditambah atau diganti. Git ini sebenarnya memudahkan programmer untuk mengetahui perubahan source codenya daripada harus membuat file baru seperti *Program.java*, *ProgramRevisi.java*, *ProgramRevisi2.java*, *ProgramFix.java*. Selain itu, dengan git kita tak perlu khawatir code yang kita kerjakan bentrok, karena setiap developer bias membuat branch sebagai *workspacenya*. Fitur yang tak kalah keren lagi, pada git kita bisa memberi komentar pada source code yang telah ditambah/diubah, hal ini mempermudah developer lain untuk tahu kendala apa yang dialami developer lain.

Untuk mengetahui bagaimana menggunakan git, berikut perintah-perintah dasar git:

1. Git init : untuk membuat *repository* pada file lokal yang nantinya ada folder *.git*
2. Git status : untuk mengetahui status dari *repository* lokal
3. Git add : menambahkan file baru pada *repository* yang dipilih
4. Git commit : untuk menyimpan perubahan yang dilakukan, tetapi tidak ada perubahan pada *remote repository*.
5. Git push : untuk mengirimkan perubahan file setelah di commit ke *remote repository*.
6. Git branch : melihat seluruh *branch* yang ada pada *repository*
7. Git checkout : menukar *branch* yang aktif dengan *branch* yang dipilih

8. Git merge : untuk menggabungkan *branch* yang aktif dan *branch* yang dipilih
9. Git clone : membuat Salinan *repository* lokal
 - The Many (Faces) Stages Of Git



Gambar 8.1 Stages Of Git

Contoh dari *software version control system* adalah github, bitbucket, snowy evening, dan masih banyak lagi. Jika anda sebagai developer belum mengetahui fitur git ini, maka anda wajib mencoba dan memakainya. Karena banyak manfaat yang akan didapat dengan git ini.

Git itu bukanlah sebuah bahasa seperti halnya HTML, CSS atau Js bukan pula sebuah konsep atau aturan baku dalam pemrograman, melainkan sebuah software yang berfungsi untuk mengatur source code dari aplikasi yang sedang anda buat.

Fungsi utamanya adalah untuk mengatur versi dari source code anda, menambahkan tanda/checkpoint ketika terjadi perubahan pada kode Anda dan tentunya akan mempermudah Anda untuk tetap mengetahui apa saja yang berubah dari source code Anda.

Sebagai contoh, misalkan Anda sedang membangun sebuah website, dan anda akan menambahkan beberapa fitur dalam website anda. Agar tidak membingungkan nantinya anda membuat sebuah catatan terhadap apa yang telah anda lakukan seperti :

- 01-04-2014 Memulai Project Website, File HTML & CSS Dasar
- 02-04-2014 Penambahan Menu Utama
- 03-04-2014 Penambahan Layout Standar
- 04-04-2014 Penambahan Fitur Pengubah Layout

Pada contoh diatas kita menggunakan tanggal sebagai tanda akan apa yang telah kita lakukan, dengan demikian kita bisa tahu kapan perubahan terjadi dan apa perubahan yang dilakukan.

Dan dalam Git semua itu bisa dilakukan dengan mudah dan asyiknya jika Anda merusak kode sehingga membuat aplikasi error, maka anda dapat mengembalikan kode tersebut berdasarkan pada tanda/tanggal dimana kode masih normal, lebih mirip seperti restore point.

Git juga tidak hanya digunakan untuk perorangan, beberapa orang pun dapat bekerja secara bersamaan mengerjakan kode Anda dan Anda masih memiliki kontrol penuh terhadap kode Anda,

Anda bisa menambahkan kode yang ditambahkan oleh orang lain atau mengabaikannya sama sekali. oleh karena itu Git sering digunakan sebagai pengatur dalam proyek kolaborasi dimana tidak hanya satu orang yang mengerjakan sebuah kode tapi beberapa orang sekaligus yang mengerjakan kode tersebut.

Misalkan anda ingin menambahkan suatu fitur, namun anda tidak mau kode yang ada sekarang rusak karena fitur yang akan anda tambahkan masih belum stabil,

Dalam Git anda dapat membuat branch terlebih dahulu. Branch ini bisa diartikan sebagai cabang dari branch master. segala perubahan yang anda lakukan pada branch yang anda buat tidak akan berpengaruh pada branch lainnya.

Sebagai contoh, kita buat branch dengan nama branch "fix-css" dengan mengetikkan perintah:

```
git branch fix-css
```

Jika perintah dijalankan dengan benar maka ketika anda mengetikkan perintah `git branch` akan muncul branch-branch yang telah dibuat.

```
fix-css
```

```
* master
```

tanda Bintang menandakan bahwa anda sedang bekerja pada branch master, untuk berpindah ke branch yang baru saja dibuat (fix-css) ketikkan perintah berikut:

```
git checkout fix-css
```

Jika perintah di atas benar, maka akan ada pemberitahuan seperti berikut:

```
Switched to branch 'fix-css'
```

Github

Github merupakan situs sharing code dan menggunakan git sebagai SCM-nya. Mungkin Anda pernah mendownload beberapa library/source code dari situs ini. Kini anda tahu mengapa kebanyakan source code dapat anda temui di github.

8.1.1 Apa itu GIT?

Git adalah version control, dengan menggunakan Git kita dapat menyimpan tiap perubahan yang kita lakukan pada file. Seperti menggunakan "undo" tetapi dapat digunakan pada seluruh file kita dan kita mempunyai log pada tiap perubahan yang kita simpan, Hal ini sangat berguna pada production environment, karena dengan adanya versioning, kita dapat dengan mudah melakukan rollback. Selain itu Git juga sangat membantu dalam proses pengembangan aplikasi. Salah

satu platform Git yang terkenal adalah GitHub. Kita akan menggunakan GitHub untuk artikel berikut.

8.2 Repository

Di Git Repository biasanya digunakan untuk mengorganisir sebuah proyek. Proyek dapat berisi files, folder, gambar, video dan lainnya yang dibutuhkan untuk sebuah project. Kami menyarankan untuk menambahkan file README yang bertujuan untuk memberi informasi terkait isi dari project tersebut. Kita dapat mengasumsikan Repository sebagai folder untuk menyimpan project kita. Satu berada di Local, dan yang satu berada di Remote disini kita menggunakan GitHub.

Seperti Ini Review Changes

Setelah melihat rincian komit, Jerry menyadari bahwa panjang string tidak boleh negatif, karena itulah dia memutuskan untuk mengubah jenis fungsi `my_strlen` yang kembali.

Jerry menggunakan perintah `git log` untuk melihat detail log.

```
$ git log
```

```
Perintah di atas akan menghasilkan hasil berikut:  
melakukan cbe1249b140dad24b2c35b15cc7e26a6f0  
Penulis: Jerry Mouse <jerry@tutorialspoint.com>  
Tanggal: Rabu Sep 11 08:05:26 2013 +0530  
Diimplementasikan fungsi my_strlen
```

Jerry menggunakan perintah `git show` untuk melihat rincian komit. Perintah `git show` mengambil SHA-1 commit ID sebagai parameter.

```
$ git show cbe1249b140dad24b2c35b15cc7e26a6f0
Perintah di atas akan menghasilkan hasil sebagai berikut:
melakukan cbe1249b140dad24b2c35b15cc7e26a6f02
Penulis: Jerry Mouse <jerry@tutorialspoint.com>
Tanggal: Rabu Sep 11 08:05:26 2013 +0530
Diimplementasikan fungsi my_strlen
```

```
diff - git a / string.c b / string.c
mode file baru 100644
indeks 0000000..187afb9
— / dev / null
+++ b / string.c
@@ -0,0 +1,24 @@
+ #include <stdio.h>
+
+ int my_strlen (char * s)
+ {
+     char * p = s;
+
+     while (* p)
+         ++ p;
+     return (p - s);
+ }
+
```

Dia mengubah jenis fungsi kembali dari `int` menjadi `size_t`. Setelah menguji kode tersebut, dia mengulas perubahannya dengan menjalankan perintah `diff git`.

```
$ git diff
```

Perintah di atas akan menghasilkan hasil sebagai berikut:

```
diff - git a / string.c b / string.c
indeks 187afb9..7da2992 100644
— a / string.c
+++ b / string.c
@@ -1,6 +1,6 @@
#include <stdio.h>
-int my_strlen (char * s)
+size_t my_strlen (char * s)
{
    char * p = s;
    @@ -18,7 +18,7 @@ int main (void)
};
untuk (i = 0; i < 2; ++ i)
{
    - printf ("panjang string %s = %d \n", s[i], my
_strlen (s[i]));
    + printf ("panjang string %s = %lu \n", s[i], my
_strlen (s[i]));
    kembali 0;
}
```

Git diff menunjukkan tanda '+' sebelum baris, yang baru ditambahkan dan '-' untuk baris yang dihapus.

Melihat Sejarah Komit

Setelah Anda membuat beberapa commit, atau jika Anda telah mengkloning sebuah repositori dengan riwayat komit yang ada, Anda mungkin ingin melihat kembali untuk melihat apa yang telah terjadi. Alat yang paling dasar dan ampuh untuk melakukan ini adalah perintah git log.

Contoh-contoh ini menggunakan proyek sederhana yang disebut "simplegit". Untuk mendapatkan proyek, jalankan git klon <https://github.com/schacon/simplegit-progit>. Saat Anda menjalankan git log dalam proyek ini, Anda harus mendapatkan keluaran yang terlihat seperti ini:

```
$ git log
melakukan ca82a6dff817ec66f44342007202690a93763949
Penulis: Scott Chacon <schacon@gee-mail.com>
Tanggal: Sen 17 Mar 21:52:11 2008 -0700
    mengubah nomor versinya
komit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Penulis: Scott Chacon <schacon@gee-mail.com>
Tanggal: Sat 15 Mar 16:40:33 2008 -0700
lepaskan tes yang tidak perlu
lakukan a11bef06a3f659402fe7563abf99ad00de2209e6
Penulis: Scott Chacon <schacon@gee-mail.com>
Tanggal: Sat Mar 15 10:31:28 2008 -0700
komit pertama
```

Secara default, tanpa argumen, git log mencantumkan komit yang dibuat di repositori tersebut dalam urutan kronologis terbalik - yaitu, komit terbaru muncul lebih dulu. Seperti yang dapat Anda lihat, perintah ini mencantumkan masing-masing komit dengan checksum SHA-1, nama penulis dan e-mail, tanggal penulisan, dan pesan komit.

Sejumlah besar dan berbagai pilihan untuk perintah git log tersedia untuk menunjukkan dengan tepat apa yang Anda cari. Di sini, kami akan menunjukkan beberapa yang paling populer.

Salah satu pilihan yang lebih bermanfaat adalah `-p`, yang menunjukkan perbedaan yang diperkenalkan pada masing-masing komit. Anda juga bisa menggunakan `-2`, yang membatasi output hanya pada dua entri terakhir:

```
$ git log -p -2
melakukan ca82a6dff817ec66f44342007202690a93763949
Penulis: Scott Chacon <schacon@gee-mail.com>
Tanggal: Sen 17 Mar 21:52:11 2008 -0700
    mengubah nomor versinya
diff - git a / Rakefile b / Rakefile
indeks a874b73..8f94139 100644
— a / Rakefile
+++ b / Rakefile
```

```

@@ -5,7 +5,7 @@ memerlukan 'rake / gempackage-
task'
spec = Gem :: Specification.new do | s |
  s.platform = Gem :: Platform :: RUBY
  s.name = "simplegit"
- s.version = "0.1.0"
+ s.version = "0.1.1"
  s.author = "Scott Chacon"
  s.email = "schacon@gee-mail.com"
  s.summary = "Sebuah permata sederhana untuk
menggunakan Git dalam kode Ruby."
  komit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
  Penulis: Scott Chacon <schacon@gee-mail.com>
  Tanggal: Sat 15 Mar 16:40:33 2008 -0700
  lepaskan tes yang tidak perlu
    diff - git a / lib / simplegit.rb b / lib / simple-
git.rb
    indeks a0a60ae..47c6340 100644
    — a / lib / simplegit.rb
    +++ b / lib / simplegit.rb
    @@ -18,8 +18,3 @@ kelas SimpleGit
    akhir
akhir
-
- if $ 0 == _ FILE _
- git = SimpleGit.new
- menempatkan git.show
- akhir
\ Tidak ada baris baru di akhir file

```

Pilihan ini menampilkan informasi yang sama namun dengan diff langsung mengikuti setiap entri. Ini sangat membantu untuk tinjauan kode atau untuk menelusuri dengan cepat apa yang terjadi selama serangkaian komit yang telah ditambahkan oleh kolaborator. Anda juga bisa menggunakan serangkaian opsi merangkum dengan log git. Misalnya, jika Anda ingin melihat beberapa statistik singkat untuk setiap komit, Anda dapat menggunakan opsi `-stat`:

\$ git log -stat

melakukan ca82a6dff817ec66f44342007202690a93763949

Penulis: Scott Chacon ;schacon@gee-mail.com;

Tanggal: Sen 17 Mar 21:52:11 2008 -0700

mengubah nomor versinya

Rakefile | 2 + -

1 file berubah, 1 insertion (+), 1 deletion (-)

komit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Penulis: Scott Chacon ;schacon@gee-mail.com;

Tanggal: Sat 15 Mar 16:40:33 2008 -0700

lepaskan tes yang tidak perlu

lib / simplegit.rb | 5 —

1 file berubah, 5 deletions (-)

lakukan a11bef06a3f659402fe7563abf99ad00de2209e6

Penulis: Scott Chacon ;schacon@gee-mail.com;

Tanggal: Sat Mar 15 10:31:28 2008 -0700

komit pertama

README | 6 ++++++

Rakefile | 23 ++++++

lib / simplegit.rb | 25

+++++

3 file berubah, 54 sisipan (+)

COMMIT CAHNGES



Gambar 9.1 Commit Changes

9.1 Pengertian

9.1.1 Commit Changes

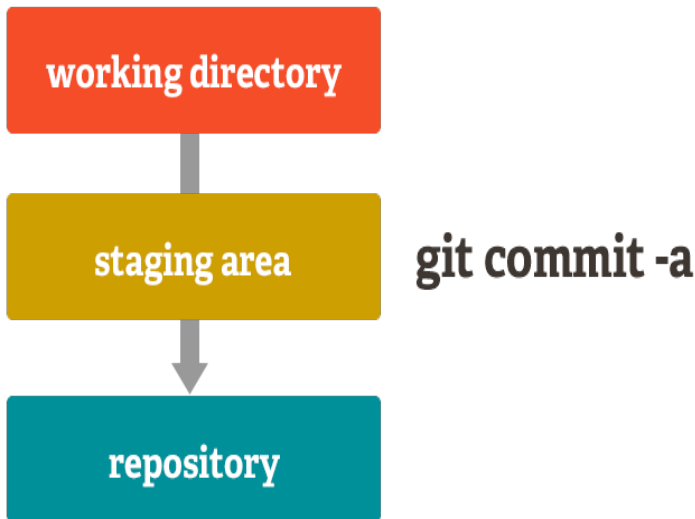
Melakukan perubahan perubahan tentunya anda sudah memiliki repositori Git yang bonafide dan sebuah salinan kerja dari semua berkas untuk proyek tersebut. Anda harus membuat beberapa perubahan dan commit perubahan tersebut ke dalam repositori setiap saat proyek mencapai sebuah keadaan yang ingin Anda rekam. Ingat bahwa setiap berkas di dalam direktori kerja Anda dapat berada di 2 keadaan: ter-pantau atau tak-terpantau. Berkas ter-pantau adalah berkas yang sebelumnya berada di snapshot terakhir; mereka dapat berada dalam kondisi belum berubah, berubah, ataupun staged (berada di area stage). Berkas tak-terpantau adalah kebalikannya - merupakan berkas-berkas di dalam direktori kerja yang tidak berada di dalam snapshot terakhir dan juga tidak berada di area staging. Ketika Anda pertama kali menduplikasi sebuah repositori, semua berkas Anda akan ter-pantau dan belum berubah karena Anda baru saja melakukan checkout dan belum mengubah apapun. Cek Status Anda Alat utama yang Anda gunakan untuk menentukan berkas-berkas mana yang berada dalam keadaan tertentu adalah melalui perintah git status. Jika Anda menggunakan alat ini langsung setelah sebuah clone, Anda akan melihat serupa seperti di bawah ini:

```
$ git status
```

```
# On branch master
```

```
nothing to commit, working directory clean
```

Ini berarti Anda memiliki direktori kerja yang bersih-dengan kata lain, tidak ada berkas ter-pantau yang berubah. Git juga tidak melihat berkas-berkas yang tak ter-pantau, karena pasti akan dilaporkan oleh alat ini. Juga, perintah ini memberitahu Anda tentang cabang tempat Anda berada. Pada saat ini, cabang akan selalu berada di master, karena sudah menjadi default-nya; Anda tidak perlu khawatir tentang cabang dulu. Bab berikutnya akan membahas tentang percabangan dan referensi secara lebih detail.



pythonregularexpression (9.1)

Mari kita umpamakan Anda menambahkan sebuah berkas baru ke dalam proyek Anda, misalnya sederhana berkas README. Jika file tersebut belum ada sebelumnya, dan Anda melakukan git status, Anda akan melihatnya sebagai berkas tak-terpantau seperti berikut ini:

```

$ \ $ $ vim README \par
\noindent
$ \ $ $ git status \par
\noindent
$ \# $ On branch master \par
\noindent
$ \# $ Untracked files: \par
\noindent
$ \# $~~ (use "git add <file>..."
to include in what will be committed) \par
\noindent
$ \# $ \par
\noindent
$ \# $~~ README \par
\noindent
nothing added to commit but

```

untracked files present (use "git add" to tr

Anda akan melihat bahwa berkas baru Anda README belum terpantau, karena berada di bawah judul "Untracked files" di keluaran status Anda. Untracked pada dasarnya berarti bahwa Git melihat sebuah berkas yang sebelumnya tidak Anda miliki di snapshot (commit) sebelumnya; Git tidak akan mulai memasukkannya ke dalam snapshot commit hingga Anda secara eksplisit memerintahkan Git. Git berlaku seperti ini agar Anda tidak secara tak-sengaja mulai menyertakan berkas biner hasil kompilasi atau berkas lain yang tidak Anda inginkan untuk disertakan. Anda hanya ingin mulai menyertakan README, mari kita mulai memantau berkas tersebut.

9.2 Berikut adalah langkah-langkah untuk menjalankan perintah

1. Untuk mulai memantau berkas baru, Anda menggunakan perintah git add. Untuk mulai memantau berkas README tadi, Anda menjalankannya seperti berikut:

```
$ \ $ git add README
```

2. Jika Anda menjalankan perintah status lagi, Anda akan melihat bahwa berkas README Anda sekarang sudah terpantau dan sudah masuk ke dalam area stage:

```
$ \ $ git status \par
\noindent
$ \# $ On branch master \par
\noindent
$ \# $ Changes to be committed: \par
```

```

\noindent
$ \# $~~ (use "git reset HEAD <file>...")
\noindent
$ \# $ \par
\noindent
$ \# $~~~new~file:      README \par
\noindent
$ \# $ \par
\noindent

```

3. Anda dapat mengatakan bahwa berkas tersebut berada di dalam area stage karena tertulis di bawah judul "Changes to be committed". Jika Anda melakukan commit pada saat ini, versi berkas pada saat Anda menjalankan git add inilah yang akan dimasukkan ke dalam sejarah snapshot. Anda mungkin ingat bahwa ketika Anda menjalankan git init sebelumnya, Anda melanjutkannya dengan git add (nama berkas) - yang akan mulai dipantau di direktori Anda. Perintah git add ini mengambil alamat dari berkas ataupun direktori; jika sebuah direktori, perintah tersebut akan menambahkan seluruh berkas yang berada di dalam direktori secara rekursif.
4. Mari kita ubah sebuah berkas yang sudah terpantau. Jika Anda mengubah berkas yang sebelumnya terpantau bernama benchmarks.rb dan kemudian menjalankan perintah status lagi, Anda akan mendapatkan keluaran kurang lebih seperti ini:

```

$ \$ $ git status \par
\noindent
$ \# $ On branch master \par
\noindent
$ \# $ Changes to be committed: \par
\noindent
$ \# $~~ (use "git

```

```

    reset HEAD <file>..."
  to unstage) \par
\noindent
$ \# $ \par
\noindent
$ \# $~~~new~file:    README \par
\noindent
$ \# $ \par
\noindent
$ \# $ Changes not staged for commit: \p
\noindent
$ \# $~~ (use "git add <file>..."
    to update what will be committed) \par
\noindent
$ \# $ \par
\noindent
$ \# $~~~modified:~  benchmarks.rb \par
\noindent
$ \# $ \par
\vspace{12pt}
\vspace{12pt}
\noindent

```

5. Berkas `benchmarks.rb` terlihat di bawah bagian yang bernama "Changes not staged for commit" - yang berarti bahwa sebuah berkas terpantau telah berubah di dalam direktori kerja namun belum masuk ke area stage. Untuk memasukkannya ke area stage, Anda menjalankan perintah `git add` (perintah ini adalah perintah multiguna - Anda menggunakannya untuk mulai memantau berkas baru, untuk memasukkannya ke area stage, dan untuk melakukan hal lain seperti menandai berkas terkonflik menjadi terpecahkan). Mari kita sekarang jalankan `git add` untuk memasukkan berkas `benchmarks.rb` ke dalam area stage, dan jalankan `git status` lagi:

```

$ \$ $ git add benchmarks.rb \par
\noindent
$ \$ $ git status \par
\noindent
$ \# $ On branch master \par
\noindent
$ \# $ Changes to be committed: \par
\noindent
$ \# $~~ (use "git reset HEAD
<file>..." to unstage) \par
\noindent
$ \# $ \par
\noindent
$ \# $~~~new~file:      README \par
\noindent
$ \# $~~~modified:~    benchmarks.rb \par
\noindent
$ \# $ \par
\vspace{12pt}
\vspace{12pt}
\vspace{12pt}
\noindent

```

6. Kedua file sekarang berada di area stage dan akan masuk ke dalam commit Anda berikutnya. Pada saat ini, semisal Anda teringat satu perubahan yang Anda ingin buat di benchmarks.rb sebelum Anda lakukan commit. Anda buka berkas tersebut kembali dan melakukan perubahan tersebut, dan Anda siap untuk melakukan commit. Namun, mari kita coba jalankan git status kembali:


```

$ \$ $ vim benchmarks.rb \par
\noindent
$ \$ $ git status \par
\noindent
$ \# $ On branch master \par
\noindent
$ \# $ Changes to be committed: \par
\noindent
$ \# $~~ (use "git reset HEAD <file>..."
to unstage) \par
\noindent
$ \# $ \par
\noindent
$ \# $~~~new~file:      README \par
\noindent
$ \# $~~~modified:~    benchmarks.rb \par
\noindent
$ \# $ \par
\noindent
$ \# $ Changes not staged for commit: \par
\noindent
$ \# $~~ (use "git add <file>..."
to update what will be committed) \par
\noindent
$ \# $ \par
\noindent
$ \# $~~~modified:~    benchmarks.rb \par
\noindent
$ \# $ \par
\vspace{12pt}
\noindent

```

7. Apa? Sekarang benchmarks.rb terdaftar di dua tempat: area stage dan area berubah. Bagaimana hal ini bisa terjadi? Ternyata Git memasukkan berkas ke area stage tepat seperti ketika Anda menjalankan perintah git add. Jika Anda commit sekarang, versi benchmarks.rb pada saat Anda terakhir lakukan perintah git add-lah yang akan masuk ke dalam commit, bukan versi berkas yang

saat ini terlihat di direktori kerja Anda ketika Anda menjalankan git commit. Jika Anda mengubah sebuah berkas setelah Anda menjalankan git add, Anda harus menjalankan git add kembali untuk memasukkan versi berkas terakhir ke dalam area stage:

```
$ \ $ git add benchmarks.rb \par
\noindent
$ \ $ git status \par
\noindent
$ \# $ On branch master \par
\noindent
$ \# $ Changes to be committed: \par
\noindent
$ \# $~~ (use "git reset HEAD <file>..."
to unstage) \par
\noindent
$ \# $ \par
\noindent
$ \# $~~~new~file:      README \par
\noindent
$ \# $~~~modified:~    benchmarks.rb \par
\noindent
$ \# $ \par
```

9.3 Tambahkan secara otomatis

Terkadang, Anda memiliki sekumpulan berkas yang Anda tidak ingin Git tambahkan secara otomatis atau bahkan terlihat sebagai tak-terantau. Biasanya berkas hasil keluaran

seperti berkas log atau berkas yang dihasilkan oleh sistem build Anda. Dalam kasus ini, Anda dapat membuat sebuah berkas bernama `.gitignore` yang berisi pola dari berkas terabaikan. Berikut adalah sebuah contoh isi dari berkas `.gitignore`:

```
$ cat .gitignore
*.[oa]
* ~
```

Baris pertama memberitahu Git untuk mengabaikan semua file yang berakhiran `.o` atau `.a` - berkas object dan arsip yang mungkin dihasilkan dari kompilasi kode Anda. Baris kedua memberitahu Git untuk mengabaikan semua file yang berakhiran dengan sebuah tilde (`~`), yang biasanya digunakan oleh banyak aplikasi olah-kata seperti Emacs untuk menandai berkas sementara. Anda juga dapat memasukkan direktori `log`, `tmp` ataupun `pid`; dokumentasi otomatis; dan lainnya. Menata berkas `.gitignore` sebelum Anda mulai bekerja secara umum merupakan ide yang baik sehingga Anda tidak secara tak-sengaja melakukan commit terhadap berkas yang sangat tidak Anda inginkan berada di dalam repositori Git.

Aturan untuk pola yang dapat Anda gunakan di dalam berkas `.gitignore` adalah sebagai berikut:

- Baris kosong atau baris dimulai dengan `#` akan diabaikan.
- Pola glob standar dapat digunakan.
- Anda dapat mengakhiri pola dengan sebuah slash (`/`) untuk menandai sebuah direktori.
- Anda dapat menegaskan sebuah pola dengan memulainya menggunakan karakter tanda seru (`!`).

Pola Glob adalah seperti regular expression yang disederhanakan yang biasanya digunakan di shell. Sebuah asterisk (`*`) berarti 0 atau lebih karakter; `[abc]` terpasangkan dengan karakter apapun yang ditulis dalam kurung siku (dalam hal ini `a`, `b`, atau `c`); sebuah tanda tanya (`?`) terpasangkan dengan

sebuah karakter; dan kurung siku yang melingkupi karakter yang terpisahkan dengan sebuah tanda hubung([0-9]) terpasangkan dengan karakter apapun yang berada diantaranya (dalam hal ini 0 hingga 9).

Berikut adalah contoh lain dari isi berkas .gitignore:

```
# sebuah komentar akan diabaikan
```

```
# abaikan berkas .a
```

```
*.a
```

```
# tapi pantau lib.a, walaupun Anda abaikan berkas .a di atas  
!lib.a
```

```
# hanya abaikan berkas TODO yang berada di root, bukan  
di subdir/TODO
```

```
/TODO
```

```
# abaikan semua berkas di dalam direktori build/  
build/
```

```
# abaikan doc/notes.txt, tapi bukan doc/server/arch.txt  
doc/*.txt
```

Jika perintah git status terlalu kabur untuk Anda - Anda ingin mengetahui secara pasti apa yang telah berubah, bukan hanya berkas mana yang berubah - Anda dapat menggunakan perintah git diff. Kita akan bahas git diff secara lebih detil nanti; namun Anda mungkin menggunakannya paling sering untuk menjawab 2 pertanyaan berikut: Apa yang Anda ubah tapi belum dimasukkan ke area stage? Dan apa yang telah Anda ubah yang akan segera Anda commit? Walaupun git status menjawab pertanyaan tersebut secara umum, git diff menunjukkan kepada Anda dengan tepat baris yang ditambahkan dan dibuang - dalam bentuk patch-nya.

Mari kita anggap Anda mengubah dan memasukkan berkas README ke area stage lagi dan kemudian mengubah berkas benchmarks.rb tanpa memasukkannya ke area stage. Jika Anda jalankan perintah status Anda, Anda akan sekali lagi melihat keluaran seperti berikut:

```
git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD [file;...]" to unstage)
```

```
#
# new file: README
#
# Changes not staged for commit:
# (use "git add <file>..." to update what will be committed)
#
# modified:  benchmarks.rb
#
```

Untuk melihat apa yang Anda telah ubah namun belum masuk ke area stage, ketikkan git diff tanpa argumen lainnya.

```
git diff
diff -git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
  @commit.parents[0].parents[0].parents[0]
  end

+  run _code(x, 'commits 1') do
+    git.commits.size
+  end
+
  run _code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size
```

Perintah di atas membandingkan apa yang ada di direktori kerja Anda dengan apa yang ada di area stage. Hasilnya memberitahu Anda bahwa perubahan yang Anda ubah namun belum masuk ke area stage.

Jika Anda ingin melihat apa yang telah Anda masukkan ke area stage yang nantinya akan masuk ke commit Anda berikutnya, Anda dapat menggunakan git diff --cached. (Di Git versi 1.6.1 atau yang lebih tinggi, Anda dapat juga menggunakan git diff --staged, yang mungkin lebih mudah untuk diingat). Perintah ini membandingkan area stage Anda dengan commit Anda terakhir:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git
repository
```

Satu hal penting yang harus dicatat adalah bahwa git diff saja tidak memperlihatkan semua perubahan yang telah Anda lakukan sejak terakhir Anda commit - hanya perubahan yang belum masuk ke area stage saja. Mungkin agak sedikit membingungkan, karena jika Anda telah memasukkan semua perubahan ke area stage, git diff akan memberikan keluaran kosong.

Sebagai contoh lain, jika Anda memasukkan berkas benchmarks.rb ke area stage dan kemudian meng-editnya, Anda dapat menggunakan git diff untuk melihat perubahan di berkas tersebut yang telah masuk ke area stage dan perubahan yang masih di luar area stage:

```
$ git add benchmarks.rb
$ echo '# test line' && benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#
#   modified:   benchmarks.rb
#
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
#
```

Sekarang Anda dapat menggunakan git diff untuk melihat apa saja yang masih belum dimasukkan ke area stage:

```
$ git diff
diff -git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
— a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
main()
```

```
# #pp Grit::GitRuby.cache _client.stats
+ # test line
```

dan git diff -cached untuk melihat apa yang telah Anda masukkan ke area stage sejauh ini:

```
$ git diff -cached
diff -git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
— a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
  @commit.parents[0].parents[0].parents[0]
end

+   run _code(x, 'commits 1') do
+     git.commits.size
+   end
+
  run _code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size
```

Sekarang setelah area stage Anda tertata sebagaimana yang Anda inginkan, Anda dapat melakukan commit terhadap perubahan Anda. Ingat bahwa apapun yang masih di luar area stage - berkas apapun yang Anda telah buat atau ubah yang

```
$ git commit
```

Aplikasi olahkata akan menampilkan teks berikut (contoh berikut adalah dari layar Vim):

with ' #' will be ignored, and an empty message aborts the commit.

”`.git/COMMIT_EDITMSG`” 10L, 283C

Anda dapat melihat bahwa pesan commit standar berisi keluaran terakhir dari perintah git status yang terkomentari dan sebuah baris kosong di bagian atas. Anda dapat mem-

buang komentar-komentar ini dan mengetikkan pesan commit Anda, atau Anda dapat membiarkannya untuk membantu Anda mengingat apa yang akan Anda commit. (Untuk pengingat yang lebih eksplisit dari apa yang Anda ubah, Anda dapat menggunakan opsi -v di perintah git commit. Melakukan hal ini akan membuat diff dari perubahan Anda di dalam olahkata sehingga Anda dapat melihat secara tepat apa yang telah Anda lakukan). Ketika Anda keluar dari olahkata, Git akan membuat commit Anda dengan pesan yang Anda buat (dengan bagian terkomentari dibuang). Cara lainnya, Anda dapat mengetikkan pesan commit Anda sebaris dengan perintah commit dengan mencantumkannya setelah tanda -m seperti berikut:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
2 files changed, 3 insertions(+), 0 deletions(-)
create mode 100644 README
```

Sekarang Anda telah membuat commit pertama Anda ~ Anda dapat lihat bahwa commit tersebut telah memberi Anda beberapa keluaran tentang dirinya sendiri: cabang apa yang Anda jadikan target commit (master), ceksum SHA-1 apa yang commit tersebut miliki (463dc4f), berapa banyak berkas yang diubah, dan statistik tentang jumlah baris yang ditambah dan dibuang dalam commit tersebut.

Ingat bahwa commit merekam snapshot yang Anda telah tata di area stage. Apapun yang tidak Anda masukkan ke area stage akan tetap berada di tempatnya, tetap dalam keadaan terubah; Anda dapat melakukan commit lagi untuk memasukkannya ke dalam sejarah Anda. Setiap saat Anda melakukan sebuah commit, Anda merekamkan sebuah snapshot dari proyek Anda yang bisa Anda kembalikan atau Anda bandingkan nantinya.

Walaupun dapat menjadi sangat berguna untuk menata commit tepat sebagaimana Anda inginkan, area stage terkadang sedikit lebih kompleks dibandingkan apa yang Anda bu-

tuhkan di dalam alurkerja Anda. Jika Anda ingin melewati area stage, Git menyediakan sebuah jalan pintas sederhana. Dengan memberikan opsi -a ke perintah git commit akan membuat Git secara otomatis menempatkan setiap berkas yang telah terpantau ke area stage sebelum melakukan commit, membuat Anda dapat melewati bagian git add:

```
$ git status
# On branch master
#
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
1 files changed, 5 insertions(+), 0 deletions(-)
```

Perhatikan bagaimana Anda tidak perlu menjalankan git add terhadap berkas benchmarks.rb dalam hal ini sebelum Anda commit.

Menghapus sebuah berkas dari Git

Untuk menghapus sebuah berkas dari Git, Anda harus menghapusnya dari berkas terpantau (lebih tepatnya, mengpus dari area stage) dan kemudian commit. Perintah git rm melakukan hal tadi dan juga menghapus berkas tersebut dari direktori kerja Anda sehingga Anda tidak melihatnya sebagai berkas yang tak terpantau nantinya.

Jika Anda hanya menghapus berkas dari direktori kerja Anda, berkas tersebut akan muncul di bagian "Changes not staged for commit" (yaitu, di luar area stage) dari keluaran git status Anda:

```
$ rm grit.gemspec
$ git status
# On branch master
#
# Changes not staged for commit:
```

```
# (use "git add/rm ;file..." to update what will be committed)
#
#    deleted:   grit.gemspec
#
```

Kemudian, jika Anda jalankan `git rm`, Git akan memasukkan penghapusan berkas tersebut ke area stage:

```
$ git rm grit.gemspec
rm 'grit.gemspec'
$ git status
# On branch master
#
# Changes to be committed:
#   (use "git reset HEAD ;file..." to unstage)
#
#    deleted:   grit.gemspec
#
```

Pada saat Anda commit nantinya, berkas tersebut akan hilang dan tidak lagi terpantau. Jika Anda mengubah berkas tersebut dan menambahkannya lagi ke index, Anda harus memaksa penghapusannya dengan menggunakan opsi `-f`. Ini adalah fitur keamanan (safety) untuk mencegah ketidaksengajaan penghapusan terhadap data yang belum terekam di dalam snapshot dan tak dapat dikembalikan oleh Git.

Hal berguna lain yang Anda dapat lakukan adalah untuk tetap menyimpan berkas di direktori kerja tetapi menghapusnya dari area kerja. Dengan kata lain, Anda mungkin ingin tetap menyimpan berkas tersebut di dalam cakram keras tetapi tidak ingin Git untuk memantaunya lagi. Hal ini khususnya berguna jika Anda lupa untuk menambahkan sesuatu ke berkas `.gitignore` Anda dan secara tak-sengaja menambahkannya, seperti sebuah berkas log yang besar, atau sekumpulan berkas hasil kompilasi `.a`. Untuk melakukan ini, gunakan opsi `-cached`:

```
$ git rm --cached readme.txt
```

Anda dapat menambahkan nama berkas, direktori, dan pola glob ke perintah `git rm`. Ini berarti Anda dapat melakukan hal seperti

```
$ git rm log/ \*.log
```

Perhatikan karakter backslash (`\`) di depan tanda `*`. Ini dibutuhkan agar Git juga meng-ekspansi nama berkas sebagai tambahan dari ekspansi nama berkas oleh shell Anda. Perintah ini menghapus semua berkas yang memiliki ekstensi `.log` di dalam direktori `log/`. Atau, Anda dapat melakukannya seperti ini:

```
$ git rm \* ~
```

Tidak seperti kebanyakan sistem VCS lainnya, Git tidak secara eksplisit memantau perpindahan berkas. Jika Anda mengubah nama berkas di Git, tidak ada metada yang tersimpan di Git yang menyatakan bahwa Anda mengubah nama berkas tersebut. Namun demikian, Git cukup cerdas untuk menemukannya berdasarkan fakta yang ada - kita akan membicarakan tentang mendeteksi perpindahan berkas sebentar lagi.

Untuk itu agak membingungkan bahwa Git memiliki perintah `mv`. Jika Anda hendak mengubah nama berkas di Git, Anda dapat menjalankan seperti berikut

```
$ git mv file _from file _to
```

dan itu berjalan baik. Bahkan, jika Anda menjalankannya seperti ini kemudian melihat ke status, Anda akan melihat bahwa Git menganggapnya sebagai perintah pengubahan nama berkas.

```
$ git mv README.txt README
```

```
$ git status
```

```
# On branch master
```

```
# Your branch is ahead of 'origin/master' by 1 commit.
```

```
#
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD ;file;" to unstage)
```

```
#
```

```
# renamed: README.txt -> README
```

#

Namun sebetulnya hal ini serupa dengan menjalankan perintah-perintah berikut:

```
$ mv README.txt README
```

```
$ git rm README.txt
```

```
$ git add README
```

Git mengetahui secara implisit bahwa perubahan yang terjadi merupakan proses pengubahan nama, sehingga sebetulnya tidaklah terlalu bermasalah jika Anda mengubah nama sebuah berkas dengan cara ini atau dengan menggunakan perintah `mv`. Satu-satunya perbedaan utama adalah `mv` berjumlah satu perintah dan bukannya tiga - yang membuat fungsi ini lebih nyaman digunakan. Lebih penting lagi, Anda sebetulnya dapat menggunakan alat apapun yang Anda suka untuk mengubah nama berkas, tinggal tambahkan perintah `add/rm` di bagian akhir, sesaat sebelum Anda melakukan commit.

BAB 10

PUSH OPERATION



Gambar 10.1 Push Operation

10.1 Pengertian

10.1.1 Push Operation

Bekerja dalam satu proyek secara bersama-sama sudah menjadi hal yang lumrah sekarang ini. Termasuk juga proyek-proyek dalam dunia komputer yang sebagian besar menuliskan kode untuk menciptakan software, sistem informasi, aplikasi, atau apa pun namanya. Salah satu sistem yang banyak dipakai untuk membantu pengerjaan proyek bersama tersebut adalah Source Control Management (SCM) atau disebut juga dengan Version Control System (VCS). Ada banyak contoh produk-produk yang tergolong ke dalam jenis VCS, salah satunya adalah Git. Git adalah VCS yang dibangun oleh Linus Torvalds, dikenal juga sebagai pencipta sistem operasi linux, untuk membantu pengembangan kernel linux. Sebelum menggunakan git, pengembangan kernel linux menggunakan produk VCS bernama BitKeeper. Ketika BitKeeper telah menjadi software yang berbayar dan tidak menjadi produk opensource lagi, maka Linus menciptakan Git sebagai pengganti dalam pengembangan kernel linux.

Git adalah VCS yang banyak digunakan saat ini. Banyak pengerjaan proyek bersama menggunakan bantuan Git untuk mengelola artefak proyek. Github, salah satu layanan VCS berbasis Git dengan web-based interface, telah menjadi tempat dikembangkannya proyek-proyek open source diseluruh dunia. Tidak hanya untuk proyek opensource, Github juga dapat digunakan untuk mengerjakan proyek non-opensource.

Jadi kenapa Git begitu spesial?

Bersifat tersebar, sifat tersebar menyebabkan Git terhindar dari permasalahan kegagalan disatu titik. Jika terjadi kerusakan repo pada salah satu mesin maka dapat dipulihkan dengan menyalin repo yang sama dari mesin yang berbeda. Sifat tersebar juga memudahkan developer untuk menjalankan aplikasi dalam rangka *debugging* karena seluruh artifak proyek ada pada mesin developer tersebut.

Opensource, Git dikembangkan dengan tujuan untuk membantu pengembangan kernel linux yang bersifat opensource. Konsep Git dapat dipastikan memenuhi aspek-aspek pendukung pengerjaan proyek-proyek opensource. Satu hal yang pasti bahwa git bebas digunakan tanpa pusing memikirkan lisensi.

Snapshots, Bukan Perbedaan, Berbeda dengan VCS lain yang umumnya melakukan pencatatan artefak proyek dengan konsep daftar perubahan file (*list of file-based changes*), Git melakukan pencatatan terhadap artefak proyek dengan konsep seperti pengambilan foto (*snapshot*). Konsep daftar perubahan file akan menyalin seluruh file pada versi sebelumnya untuk dimasukkan pada versi sekarang walaupun tidak terjadi perubahan pada file tersebut. Konsep *snapshot* akan mengambil gambaran artefak proyek secara keseluruhan pada versi pertama. Versi selanjutnya dibuat dengan cara menyimpan perbedaan antara versi sekarang dengan versi selanjutnya ditambah dengan penghubung (*pointer; link*) ke snapshot pada versi sekarang. Jadi konsep snapshot tidak mencatat file yang tidak berubah secara berulang-ulang dan juga tidak kehilangan informasi mengenai versi sebelumnya.

Hampir Seluruh Operasi Tidak Melibatkan Jaringan, Konsep terdistribusi pada Git menyebabkan git dapat beroperasi tanpa melibatkan mesin lain secara umum. Hal ini berbeda dengan Centralized VCS(CVCS) yang hampir seluruh operasinya memerlukan koneksi jaringan. Sebagai contoh ketika kita berada di pesawat terbang, kereta api, ataupun area lain yang tidak memiliki jaringan ke server proyek yang sedang dikerjakan, maka kita tidak bisa bekerja secara optimal jika menggunakan CVCS.

Git Memiliki Integritas, Segala sesuatu pada Git akan diberi tanda khusus (*checksum*) sebelum disimpan ke dalam database. Hal ini menghindari terjadi *file corruption* tanpa diketahui oleh Git. Git menggunakan algoritma SHA-1 untuk melakukan *checksum* tersebut.

Konsep Tiga Status, Git memiliki 3 status utama dalam melakukan pencatatan terhadap artefak proyek. Konsep 3

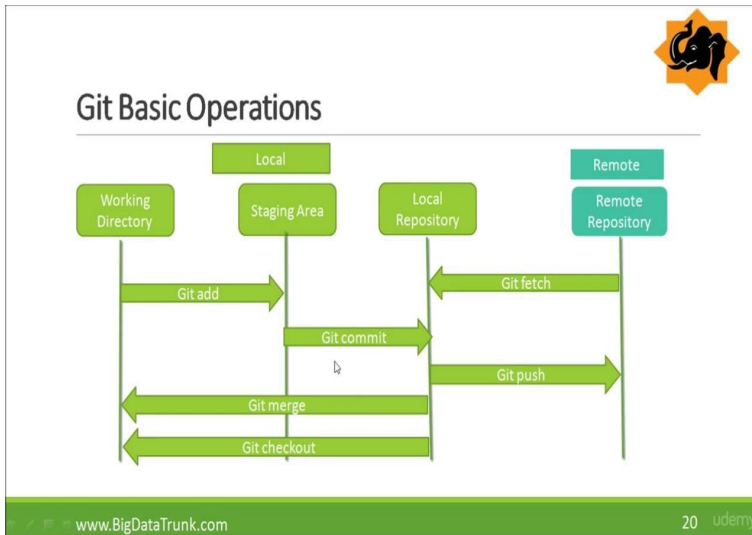
status ini adalah kunci utama dalam memahami penggunaan Git secara baik dan benar. Jadi pastikan developer paham dengan konsep 3 status pada Git sebelum menggunakannya. Jika dirasa perlu maka dapat dikumpulkan informasi yang lebih banyak tentang konsep ini untuk mempermudah penggunaan Git kedepannya. File pada Git memiliki salah satu dari 3 status berikut: *committed*, *modified*, dan *staged*. *Committed* artinya perubahan terhadap file telah disimpan ke dalam database Git. *Modified* artinya telah terjadi perubahan pada file dan perubahan sama sekali belum disimpan oleh Git. *Staged* artinya perubahan pada file telah tersimpan ke dalam penyimpanan sementara dan siap untuk disimpan ke dalam database. Konsep 3 status menyebabkan proyek pada Git terbagi atas 3 bagian utama yaitu: database Git (*Git directory or Git database or Git Repository*), area kerja (*working directory*), dan area perantara (*staging area*)

Database git adalah tempat dimana Git menyimpan seluruh informasi tambahan dan perubahan-perubahan untuk setiap versi artefak proyek. Ini adalah bagian yang sangat penting dari Git dan bagian ini adalah bagian yang akan disalin dalam proses *cloning repository* dari satu mesin ke mesin yang lain. Area kerja adalah folder induk yang mengandung seluruh artefak proyek (file-file) yang sedang dikerjakan. Area perantara adalah tempat menyimpan informasi perubahan terhadap artefak proyek yang siap untuk disimpan ke dalam database Git. Bentuk nyatanya adalah sebuah file yang terletak di dalam direktori Git. Area perantara terkadang disebut juga sebagai index.

10.2 Alur kerja Git yang paling umum adalah sebagai berikut

1. Developer melakukan perubahan terhadap file yang berada pada area kerja
2. Developer menyimpan perubahan ke dalam area perantara untuk proses pengambil *snapshot*

3. Developer menyimpan perubahan secara permanen ke dalam database Git



AlurGit

(10.1)

10.2.1 Pengertian Lanjutan

Mungkin sudah terlalu sering kita bekerja mandiri pada project pribadi, disini saya akan menunjukkan bagaimana caranya berkontribusi di project lain. Hal termudah untuk mencapai itu kita cari proyek open sources, apa itu proyek opensources, Proyek perangkat lunak open source merupakan proyek yang memberikan kode program kepada pengguna secara bebas, dan tak jarang pengembangannya dilakukan secara terbuka: siapapun boleh berkontribusi dalam menulis kode tersebut. Menggunakan perangkat lunak open source tentunya sangat baik, karena selain tidak melakukan pembajakan, kita juga mendukung para pengembang dari perangkat lunak yang kita gunakan. Tetapi, akan lebih baik lagi jika kita juga ikut berkontribusi, mulai dari kontribusi penggunaan, pelaporan bug, sampai dengan kontribusi kode. Kontribusi pada proyek open source akan membantu kita un-

tuk meningkatkan kemampuan pengembangan perangkat lunak.”.

10.3 Welcome Contributors

Fork it!

Create your feature branch: `git checkout -b my-new-feature`

Commit your changes: `git commit -am 'Add some feature'`

Push to the branch: `git push origin my-new-feature`

Submit a pull request ;)

Tahapan berkontribusi yang baik,

Cari proyek opensources. *(disini saya sebagai pengembang android saya akan memberikan contoh cara berkontribusi di salah satu proyek opensources pada library android)*

10.4 Material Tabs

Coba cari cari info tentang aturan kontribusi atau bisa dengan cara melakukan pendekatan pada developer terkait baik via email atau sosial akun.

Jika memang telah melakukan pendekatan atau tidak ada jalur lain untuk menempuh jalan tersebut, anda bisa langsung saja untuk melakukan fork project yang akan berkontribusi di dalamnya dimana nantinya kode akan di review oleh pemilik repository untuk di gabungkan atau tidaknya.

Setelah selesai fork, maka repository akan masuk ke list repo milik anda, untuk selanjutnya mungkin langsung saja, kali ini masuk pada git command apa aja yang bisa kita gunakan untuk berkontribusi di proyek opensources beserta simulasi yang akan saya contoh kan untuk berkontribusi pada proyek library android - material tabs .

`git -help` to see another commands cloning project yang suda anda fork ke akun anda

```

it clone git@github.com:CreatorB/MaterialTabs.git
untuk memudahkan development, hendaknya kita menam-
bahkan repository pusat dengan lokal milik kita agar tidak
terjadi konflik dengan kontributor lainnya. git remote add
;nama-repo; ;alamat-repo;
git remote add upstream
git://github.com/neokree/MaterialTabs.git

```

Setelah remote repositori selesai dan seperti yang saya katakan tadi agar kita tidak hanya asal berkontribusi tapi memang benar benar clear dalam membantu development maka kita hendaknya juga membuat branch baru terlebih dahulu agar tidak merusak history dan nantinya juga akan memudahkan untuk racking code. `git checkout -b ;nama-cabang;`
`git checkout -b sample-project`

Okkay sekarang setelah remote dan cabang / branch dibuat maka di cabang baru ini lah kita akan untuk melakukan perubahan kode yang nantinya bisa kita push ke repo pusat. Untuk berpindah branch bisa kita gunakan `. git check-out ;nama-cabang;` dimana di repo lokal saya sekarang ada dua cabang cabang pertama adalah master dan cabang kedua adalah sample-project dan di sample-project saya bisa melakukan banyak perubahan kode. Jika memang ada penambahan file bisa menggunakan `git add ;nama-file-baru;` atau kalo memang banyak file yang ditambahkan dan males add satu per satu, anda bisa langsung menuju root dari repository lalu `git add .` ya dot / titik berarti menambahkan semua perubahan yang ada di direktori tersebut secara rekursif. Setelah selesai perubahan silahkan lakukan commit, berisi pesan apa yang telah anda kontribusikan / lakukan perubahan. `git commit -m "fixing sample project and compile gradle added"`

Setelah selesai melakukan commit, kita lakukan persiapan untuk pull ke repo pusat, sebelum itu kita pindah branch dulu ke master.

```
git checkout master
```

Setelah itu, kita akan mengambil kode lagi dari pusat, untuk memastikan tidak terdapat konflik pada kontribusi kode kita. Konflik dapat terjadi jika dua atau lebih kontribu-

tor melakukan perubahan pada satu berkas, terutama jika perubahan dilakukan pada baris yang sama, terlepas dari apakah tujuan perubahan sama atau tidak. Perintah yang digunakan adalah git fetch dan git merge pada cabang utama.

git fetch upstream
Setelah fetching selesai, sekarang kita merge dengan master.

git merge upstream/master
Dengan beberapa proses diatas, setidaknya kita telah bisa memastikan bahwanya tidak ada konflik dengan repo pusat. Sekarang kita kembali ke branch lokal development saya .

sample-project

git checkout sample-project
dan menggabungkan cabang tersebut dengan cabang utama, sehingga kontribusi dapat dikirimkan kembali ke repositori pusat milik neokree, Material Tabs android library, dengan perintah git rebase ;nama-branch;

git rebase master

Sebelum push ke repositori pusat milik neokree, maka terlebih dahulu akan saya push ke repository milik saya hasil fork di awal pembahasan tadi, dimana banyak perubahan yang dilakukan di branch tersebut.

git push origin sample-project

Setelah di push maka sekarang kita pergi menuju browser untuk melakukan pull request, cek / compare perubahan apa yang telah anda lakukan di branch anda pada branch master milik repo pusat, dan anda juga bisa menyisipkan pesan untuk memberitahukan developer pemilik repo pusat tentang apa yang anda lakukan, setelah yakin terhadap perubahan yang telah anda lakukan silahkan pilih create pull request dan selamat menunggu pemilik repo pusat untuk menanggapi di terima tidaknya kontribusi anda.

Cara Push

Pada dasarnya Git merupakan source control yang berbasis command line, akan tetapi berbagai macam aplikasi GUI Tools sudah banyak di jumpai agar dapat memudahkan kita dalam proses pengerjaan sebuah project atau mempermudah kita yang tidak terlalu mahir dalam penggunaan syntax-

syntax tersebut, salah satunya itu SourceTree ini. Lang-

sung saja saya mulai untuk proses yang ingin saya contohkan yaitu cara push, pull, commit project ke github dengan menggunakan aplikasi SourceTree. Lalu apa saja yang perlu dipersiapkan ? Berikut adalah beberapa yang perlu anda persiapkan untuk memulai peroses.

1 install aplikasi gitbash

Bagi anda yang belum memiliki aplikasi SourceTree, anda dapat men-download nya pada web resmi yang sudah disediakan oleh pengembang aplikasi SourceTree. Aplikasi ini gratis, benar-banar gratis untuk berbagai macam keperluan komersial atau non-komersial. Anda dapat men-download nya pada web dari githu

2 Akun Github

Pastikan anda sudah memiliki akun Github, jika belum memilikinya anda dapat membuatnya terlebih dahulu dengan melakukan register terlebih dahulu, <https://github.com/> atau lihat tutorial cara mendaftar akun Github Lalu kemudian anda login pada akun Github nya.

Jika sumua keperluan diatas sudah anda siapkan, sekarang kita mulai masuk pada tutorial, ikuti contoh dan prosesnya seperti dibawah ini :

Pertama yang harus anda lakukan adalah untuk menyimpan data-data project pada Github, terlebih dahulu anda membuat repository. Caranya anda klik tanda + pada bagian kanan-atas lalu pilih "**New Repository**" Lalu anda buat nama untuk repositorinya, lalu klik "Create Repository"

Setelah itu buka aplikasi SourceTree, jika sudah terbuka aplikasinya maka tampilannya akan seperti gambar dibawah ini.
*Pastikan anda sudah login aplikasi SourceTree nya dengan menggunakan akun Github yang anda sudah buat.

repository yang kita buat caranya adalah klik browse pada gambar "**bumi**", maka akan tampil semua repository yang

kita sudah pernah buat, lalu pilih salah satu repository. Kemudian pada bagian **Destination Path** : itu adalah lokasi penyimpanan lokal yang kita ingin tempatkan. Harus anda

ingat lokasi penyimpanan lokal anda, karena ketika kita akan melakukan pull, push, commit dilakukan pada penyimpanan lokal tersebut. Jika sudah klik **Clone**.

Melakukan Proses Commit & Push

Kemudian anda buka folder lokal kalian, lalu pindahkan dan letakkan semua file project kalian folder lokal kalian. Saya berikan contoh meletakkan satu file kedalam folder lokal, lihat seperti contoh gambar dibawah ini.

Kemudian anda buka folder lokal kalian, lalu pindahkan dan letakkan semua file project kalian folder lokal kalian.

Lalu anda simpan file yang sudah dibuat, sekarang kembali ke aplikasi SourceTree ketika ada file baru yang di letakkan pada folder lokal maka akan terdeteksi pada aplikasi SourceTree

Kemudian anda pindahkan file yang tadi sudah dibuat menjadi keatas dengan cara klik button **"Stage All"** kemudian anda isikan komentar yang anda inginkan. Jika sudah anda lakukan Commit, tunggu beberapa saat hingga proses selesai.

Jika sudah maka pada bagian menu **Push** diatas terdapat notifikasi angka 1 itu artinya ada satu file yang sudah siap di kirim pada server Github. Lalu anda klik menu **Push** tersebut sehingga muncul *dialog box* lalu anda klik button **Push** tunggu beberapa saat hingga proses upload selesai. Jika sudah selesai anda bisa cek pada Github, apakah sudah masuk apa belum.

Sampai disini proses commit dan push sudah selesai dilakukan, Kemudian kita akan mencoba melakukan pull,

yakni menarik/mengambil file yang ada pada repository. Terlebih dahulu anda membuat file baru pada Github untuk mencoba menarik/mengambil file yang telah dibuat pada repository ke dalam folder lokal. Caranya adalah dengan menekan tombol menu **Pull**. Kemudian akan muncul *dialog box* Kemudian klik **OK**.

Tunggu beberapa saat hingga proses selesai dilakukan, jika proses sudah selesai sekarang cek pada folder lokal anda apakah file yang di tarik tadi sudah masuk atau belum.

Atau jika cara diatas tidak berhasil anda gunakan maka bisa menggunakan cara berikut ini menggunakan gitbash

Setelah kita mempunya akun GitHub dan menginstall software git maka kita bisa langsung meng upload project kita. Buatlah repository di GitHub dengan mengklik icon repo "Create new repository".

Kemudian beri nama repository nya lalu beri deskripsi untuk repository yang kita buat jika perlu, kemudian setting public/private, kalau public berarti bisa di akses oleh semua orang, kemudian centang "initialize this repository with a README" dan tambahkan kategori repository jika perlu. kemudian klik "create repository".

Jika repository berhasil dibuat, anda akan di berikan kunci akses berupa HTTP/SSH, ini yang akan kita gunakan untuk remote repository dari software GIT. Misal saya punya kunci HTTP <http://github.com/acchoblues/APeK.git>

1. Setelah anda berhasil membuat repository sekarang klik kanan pada folder project yang akan di upload.
2. Klik kanan pada project klik "Git Bash".
3. Kemudian akan muncul command prompt / CMD
4. Jika anda baru pertama kali menggunakan software GIT, sebaiknya konfigurasi username dan email dulu.

Ketik


```

Git config --global user.name "username anda"
\noindent
Git config --global user.email isi $
\n_ $dengan $ \_ $email $
\n_ $anda@ymail.com \par
\noindent

```

Setelah melakukan konfigurasi username dan email, sekarang kita lakukan inisiasi, ketikan

Git init

Kemudian kita tambahkan semua file yang ada dalam folder project kita, ketikan

Git add *

Kemudian kita buat commit project nya, misal disini saya kasih commit "versi 1.0.0", ketikan

Git commit m "versi 1.0.0"

Setelah kita buat commit untuk project nya, sekarang kita remote repository yang kita buat tadi, tentunya kita menggunakan kunci HTTP yang ada pada repository tadi, kalo ane kan tadi contoh nya <http://github.com/acchoblues/APeK.git>, ketikan

Git remote add origin <http://github.com/acchoblues/APeK.git>

Setelah me-remote repository kita tadi, sekarang kita pull project nya, ketikan

Git pull origin master

Terakhir kita kirim project kita ke repository kita, ketikan

Git push origin master

Biasanya ketika kita ketikan perintah push ini, kita akan diminta username dan password kita dan perlu di perhatikan untuk password nya biasa nya ketika kita mengetikan password maka pada command prompt nya tidak ditampilkan karakter apapun.

So jangan khawatir tunggu sampai selesai di upload project-nya.

kalau sudah selesai proses uploadnya silahkan refresh repository anda, insya allah jika sesuai dengan instruksi di atas maka project anda sudah terupload disana.

```
$ \ $ \textbf{git init --bare proyekKhadapi}
Initialized empty Git repository in F:/proyek
```

Perintah di atas akan membuat sebuah *bare repository* yang hanya dipakai untuk melakukan sinkronisasi antara pekerjaan Edi dan Bram nantinya.

Di komputer Edi, berbekal dengan flash disk yang mengandung *bare repository* tersebut, ia akan melakukan menkloning *repository* tersebut dengan memberikan perintah berikut ini:

```
$ cd ~/Desktop
```

Sekarang, di folder *~/Desktop* akan ada sebuah folder

proyekKhadapi yang merupakan kloningan dari yang ada di flash disk. Edi hanya perlu meletakkan kode programnya di folder komputer tersebut, dan bukan di *bare repository* di flash disk.

Bram kemudian meminjam flash disk berisi *bare repository* tersebut, dan memberikan perintah yang sama untuk membuat kloning dari *bare repository* yang ada di flash disk ke desktop-nya.

Kedua developer tersebut kemudian bekerja di komputer masing-masing. Sebagai contoh, ini adalah simulasi pekerjaan yang dilakukan oleh Edi di repository miliknya:

```
{echo 'Menu 1' >> menu.txt} \par
{echo 'Isi modul 1' >> modul1.txt} \par
{git add .} \par
```

```
{git commit -m 'Membuat modul 1'} \par
```

Edi membuat dua file dengan nama *menu.txt* dan *modul1.txt*. Anggap saja ini adalah kode program. Pada saat yang bersamaan, Bram juga menambahkan file pada repository miliknya:

```
$ echo 'Menu 2' && menu.txt
$ echo 'Isi modul 2' && modul2.txt
$ git add .
$ git commit -m 'Membuat modul 2'
```

Setelah semuanya selesai, Bram men-*push* perubahan yang dilakukannya ke *upstream* di flash disk. Ia memberikan perintah berikut ini:

```
$ git push origin master
```

Sekarang, commit dengan pesan Membuat modul 2 milik

Bram sudah dipindahkan ke *upstream* di flash disk. Untuk memastikannya, ia berpindah ke folder di flash disk, dan memberikan perintah berikut ini:

```
$ cd F:/proyekKhadapi
```

```
$ git show-branch
```

```
[master] Membuat modul 2
```

Tugas Bram sudah selesai, ia kemudian menyerahkan flash disk yang berisi *bare repository* ke Edi. Kebetulan Edi juga telah selesai mengerjakan modul 1, sehingga ia berniat men-*push* perubahannya ke *upstream*. Akan tetapi Edi akan memperoleh pesan kesalahan seperti berikut ini:

```
$ git push origin master
```

```
To file:///F:/proyekKhadapi
```

```
! [reject] master -& master (fetch first)
```

```
error: failed to push some refs to 'file:///F:/proyekKhadapi'
```

Pesan kesalahan ini muncul karena pada saat Edi menger-

jakan modul 1, Bram telah duluan men-*push* kode program-nya ke *upstream* di flash disk. Agar bisa men-*push* ke *upstream*, Edi perlu terlebih dahulu mengambil perubahan yang dibuat oleh Bram di *upstream* dan men-*merge* perubahan tersebut dengan perubahan miliknya. Edi memberikan perintah berikut ini:

```
$ git pull
```

```
...
```

```
Auto-merging menu.txt
```

```
CONFLICT (add/add): Merge conflict in menu.txt
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

Git biasanya cukup pintar dalam melakukan merging file

yang konflik. Tapi pada kasus ini, baik Edi dan Bram sama-sama menambah item baru pada baris pertama, sehingga Git tidak tahu mana yang harus dipakai. Edi perlu memperbaiki konflik ini dengan mengubah file *menu.txt* secara manual menjadi seperti berikut ini:

```
Menu 1
```

```
Menu 2
```

Setelah itu, Edi akan men-*commit* perubahan ini dengan

memberikan perintah berikut ini:

```
$ git add menu.txt
```

```
$ git commit -m 'Menyesuaikan dengan perubahan dari Bram'
```

Sekarang, Edi akan memiliki 3 file, yaitu *menu.txt*, *modul1.txt* dan *modul2.txt*. Ini mewakili hasil akhir yang diharapkan. Untuk melihat riwayat perubahan, Edi dapat memberikan perintah berikut ini:

```
$ git log
```

```
commit 8ce2a6bdc83387c2f81c56d60e06fa0ce62a3df8
```

Merge: 1bcc42b 841b46d

Author: Edi <edi@dev.com>

Date: Fri Apr 12 06:19:25 2013 +0700

Menyesuaikan dengan perubahan dari Bram

commit 841b46d0dea229f55fd215e9cfadcd58b8aa64c2

Author: Bram <bram@dev.com>

Date: Fri Apr 11 05:57:37 2013 +0700

Membuat modul 2

commit 1bcc42b2346b95830bd6b3363425a9c8fa3a92f4

Author: Edi <edi@dev.com>

Date: Fri Apr 10 05:56:45 2013 +0700

Membuat modul 1

Saat ini perubahan masih ada di repository lokal milik Edi.

Untuk memindahkan perubahan ke *upstream* yang ada di flash disk, Edi memberikan perintah berikut ini:

\$ git push

Oke, Edi kemudian

meninggalkan komputer dan menikmati hidupnya untuk sejenak.

Kita kembali lagi ke Bram. Saat ini Bram hanya memiliki kode program modul 2 dan tidak memiliki kode program modul 1 yang dibuat Edi. Bram juga ingin memperoleh kode program terbaru! Ia meminjam flash disk yang berisi *bare repository*, mencolokkannya di komputer rumahnya, lalu Bram memberikan perintah berikut ini:

\$ git pull

...

Updating 841b46d..8ce2a6b

Fast-forward

menu.txt | 1 +

modul1.txt | 1 +

```
2 files changed, 2 insertions(+)  
create mode 100644 modul1.txt
```

Kali ini tidak ada konflik saat melakukan *pull* dari *up-*

stream. Hal ini karena Edi sudah menyelesaikan (atau mengatasi) konflik tersebut sebelumnya. Sampai pada tahap ini, isi repository Bram akan sama persis dengan milik Edi (terdiri atas 3 file yaitu *menu.txt*, *modul1.txt* dan *modul2.txt*).

Pada contoh ini, Bram dan Edi memakai *bare repository* yang disimpan di sebuah flash disk sebagai perantara. Hal ini bukanlah sesuatu yang efisien terutama bila mereka dipisahkan oleh jarak yang jauh. Bila komputer mereka sama-sama terkoneksi ke internet, mereka dapat men-*setup* sebuah server Git sebagai *upstream*. Selain mendukung protokol git (*native*), Git juga mendukung protokol lainnya seperti HTTP, HTTPS, SSH, dan SCP. Sebuah server Git paling mudah dikonfigurasi bila memakai sistem operasi Linux. Bila tidak ingin repot-repot men-setup server sendiri, Bram dan Edi dapat layanan dari pihak penyedia server seperti GitHub.

UPDATE OPERATION

11.1:

Git adalah version control system yang digunakan para developer untuk mengembangkan software secara bersama-sama. Fungsi utama git yaitu mengatur versi dari source code program anda dengan mengasih tanda baris dan code mana yang ditambah atau diganti.

Git ini sebenarnya memudahkan programmer untuk mengetahui perubahan source codenya daripada harus membuat file baru seperti Program.java, ProgramRevisi.java, ProgramRevisi2.java, ProgramFix.java. Selain itu, dengan git kita tak perlu khawatir code yang kita kerjakan ben-trok, karena setiap developer bias membuat branch sebagai workspacenya. Fitur yang tak kalah keren lagi, pada git kita bisa memberi komentar pada source code yang telah dita-



Gambar 11.1 learn git

mbah/diubah, hal ini mempermudah developer lain untuk tahu kendala apa yang dialami developer lain. Untuk mengetahui bagaimana menggunakan git, berikut perintah-perintah dasar git:

1. Git init : untuk membuat repository pada file lokal yang nantinya ada folder .git
2. Git status : untuk mengetahui status dari repository lokal
3. Git add : menambahkan file baru pada repository yang dipilih
4. Git commit : untuk menyimpan perubahan yang dilakukan, tetapi tidak ada perubahan pada remote repository.
5. Git push : untuk mengirimkan perubahan file setelah di commit ke remote repository.
6. Git branch : melihat seluruh branch yang ada pada repository

7. Git checkout : menukar branch yang aktif dengan branch yang dipilih
8. Git merge : untuk menggabungkan branch yang aktif dan branch yang dipilih
9. Git clone : membuat Salinan repository lokal

Contoh dari software version control system adalah github, bitbucket, snowy evening, dan masih banyak lagi. Jika anda sebagai developer belum mengetahui fitur git ini, maka anda wajib mencoba dan memakainya. Karena banyak manfaat yang akan didapat dengan git ini.

Dalam melakukan pemrograman, perubahan spesifikasi atau kebutuhan adalah hal yang tidak dapat dihindari. Tidak ada program yang dapat dituliskan dengan sempurna pada percobaan pertama. Hal ini menyebabkan pengembang perangkat lunak sangat dekat dengan sistem kontrol versi, baik secara manual maupun menggunakan perangkat lunak khusus. Seri tulisan ini akan membahas tentang sistem kontrol versi, kegunaannya, serta contoh kasus menggunakan git, salah satu perangkat lunak populer untuk kontrol versi.

11.1 Dasar Kontrol Versi

Kegunaan utama dari sistem kontrol versi ialah sebagai alat untuk manajemen kode program. Terdapat dua kegunaan utama dari sistem ini, yaitu:

1. Menyimpan versi lama dari kode, maupun
2. Menggabungkan perubahan-perubahan kode dari versi lama (misal: untuk mengembalikan fitur yang telah dihapus) ataupun menggabungkan perubahan dari orang lain (misal: menggabungkan fitur yang dikembangkan oleh anggota tim lain).

Tanpa menggunakan sistem kontrol versi, yang sering saya temukan (dan dulunya saya gunakan, sebelum mengetahui tentang kontrol versi) ialah penggunaan direktori untuk

memisahkan beberapa versi program. Sistem kontrol versi, seperti git, hg, atau bzt, dikembangkan untuk menyelesaikan masalah-masalah di atas. Karena tidak ingin membahas terlalu banyak, artikel ini hanya akan menjelaskan penggunaan git, karena kelihatannya git merupakan perangkat lunak kontrol versi yang paling populer untuk sekarang (mengingat popularitas Github dan penggunaan git pada kernel Linux).

11.1.1 Git

Git adalah sebuah perangkat lunak untuk mengontrol versi sebuah perangkat lunak "VCS/Version Control System". Git diciptakan oleh Linux Torvalds, yang pada awalnya ditujukan untuk pengembangan kernel Linux. Saat ini banyak perangkat lunak yang terkenal menggunakan Git sebagai pengontrol revisinya. Pada bab ini akan mempelajari

bagaimana cara menggunakan Git seperti proses life cycle Git, operasi-operasi dasar dan bagaimana cara menangani masalah saat menggunakan Git. Git menyimpan semen-

tara perubahan yang telah di buat pada copy-an pekerjaan Anda sehingga Anda dapat mengerjakan sesuatu yang lain, lalu kembali dan terapkan kembali nanti. Stashing berguna jika Anda perlu mengubah konteks dan mengerjakan hal lain dengan lebih cepat, tapi Anda sedang melewati perubahan kode dan tidak cukup siap untuk melakukannya. Perintah git

stash mengambil perubahan yang tidak terikat (baik yang dipasang maupun yang tidak terpasang), menyimpannya untuk penggunaan selanjutnya, lalu mengembalikannya dari salinan pekerjaan Anda. Sebagai contoh:

```
\$ git status
```

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

\\$ git stash

Saved working directory and index state WIP o
homepage

HEAD is now at 5002d47 our new homepage

\\$ git status

On branch master

nothing to commit, working tree clean

Pada poin ini Anda bebas melakukan perubahan, membuat commit baru, mengganti cabang, dan melakukan operasi Git lainnya; Kemudian kembali dan pasang kembali simpanan Anda saat Anda siap.

1. Perhatikan bahwa simpanannya adalah lokal ke tempat penyimpanan Git Anda.
2. Mengajukan kembali perubahan tersimpan Anda
3. Anda dapat mengajukan permohonan kembali sebelumnya menyimpan perubahan dengan git stash pop:

\\$ git status

On branch master

```
nothing to commit, working tree clean
```

```
\$ git stash pop
```

```
On branch master
```

```
Changes to be committed:
```

```
new file:   style.css
```

```
Changes not staged for commit:
```

```
modified:   index.html
```

```
Dropped refs/stash@{0} (32b3aa1d185dfe6d57b
```

Memindahkan simpanan Anda akan menghilangkan perubahan dari simpanan Anda dan memasangnya kembali ke salinan pekerjaan Anda.

Sebagai alternatif, Anda dapat mengajukan permohonan kembali perubahan pada copy pekerjaan Anda dan menyimpannya di tempat penyimpanan dengan git stash berlaku:

```
\$ git stash apply
```

```
On branch master
```

```
Changes to be committed:
```

```
new file:   style.css
```

```
Changes not staged for commit:
```

```
modified:   index.html
```

Ini berguna jika Anda ingin menerapkan perubahan tersimpan yang sama ke beberapa cabang.

Sekarang setelah Anda mengetahui dasar-dasar stashing, ada satu peringatan dengan penyimpanan git yang perlu

Anda sadari: Secara default Git tidak akan menyimpan perubahan yang dibuat pada file yang tidak terlacak atau diabaikan.

Menyembunyikan file yang tidak terlacak atau diabaikan

Secara default, menjalankan git stash akan menyimpan:

1. perubahan yang telah ditambahkan ke indeks Anda (perubahan bertahap)
2. Perubahan yang dilakukan pada file yang saat ini dilacak oleh Git (perubahan yang tidak terhapus)

Tapi itu tidak akan disimpan:

1. file baru dalam copy pekerjaan Anda yang belum dipentaskan
2. file yang telah diabaikan

Jadi jika kita menambahkan file ketiga ke contoh kita di atas, tapi jangan tingkatkan (misal kita tidak menjalankan git add), git stash tidak akan menyimpannya.

```
\$ script.js
```

```
\$ git status
```

```
On branch master
```

```
Changes to be committed:
```

```
new file: style.css
```

```
Changes not staged for commit:
```

```
modified: index.html
```

```
Untracked files:
```

```
script.js
```

```
\$ git stash
```

```
Saved working directory and index state WIP o  
homepage
```

```
HEAD is now at 5002d47 our new homepage
```

```
\$ git status
```

```
On branch master
```

```
Untracked files:
```

```
script.js
```

Menambahkan opsi -u (atau `--include-untracked`) memberitahu git stash untuk juga menyimpan file yang tidak terlacak.

Jadi, sebenarnya apa yang dimaksud dengan Git? Ini adalah bagian penting untuk dipahami, karena jika anda memahami apa itu Git dan cara kerjanya, maka dapat dipastikan anda dapat menggunakan Git secara efektif dengan mudah. Selama mempelajari Git, cobalah untuk melupakan VCS lain yang mungkin telah anda kenal sebelumnya, misalnya Subversion dan Perforce. Git sangat berbeda dengan sistem-sistem tersebut dalam hal menyimpan dan memperlakukan informasi yang digunakan, walaupun antar-muka penggunaannya hampir mirip. Dengan memahami perbedaan tersebut diharapkan dapat membantu anda menghindari kebingungan saat menggunakan Git.

Salah satu perbedaan yang mencolok antar Git dengan VCS lainnya (Subversion dan kawan-kawan) adalah dalam cara Git memperlakukan datanya. Secara konseptual, kebanyakan sistem lain menyimpan informasi sebagai sebuah daftar perubahan berkas. Sistem seperti ini (CVS, Subver-

sion, Bazaar, dan yang lainnya) memperlakukan informasi yang disimpannya sebagai sekumpulan berkas dan perubahan yang terjadi pada berkas-berkas tersebut.

Git memperlakukan datanya sebagai sebuah kumpulan snapshot dari sebuah miniatur sistem berkas. Setiap kali anda melakukan commit, atau melakukan perubahan pada proyek Git anda, pada dasarnya Git merekam gambaran keadaan berkas-berkas anda pada saat itu dan menyimpan referensi untuk gambaran tersebut. Agar efisien, jika berkas tidak mengalami perubahan, Git tidak akan menyimpan berkas tersebut melainkan hanya pada file yang sama yang sebelumnya telah disimpan.

Ini adalah sebuah perbedaan penting antara Git dengan hampir semua VCS lain. Hal ini membuat Git mempertimbangkan kembali hampir setiap aspek dari version control yang oleh kebanyakan sistem lainnya disalin dari generasi sebelumnya. Ini membuat Git lebih seperti sebuah miniatur sistem berkas dengan beberapa tool yang luar biasa ampuh yang dibangun di atasnya, ketimbang sekadar sebuah VCS. Kita akan mempelajari beberapa manfaat yang anda dapatkan dengan memikirkan data anda dengan cara ini ketika kita membahas "Git branching" pada Bab 3.

Hampir Semua Operasi Dilakukan Secara Lokal

Kebanyakan operasi pada Git hanya membutuhkan berkas-berkas dan resource lokal - tidak ada informasi yang dibutuhkan dari komputer lain pada jaringan anda. Jika Anda terbiasa dengan VCS terpusat dimana kebanyakan operasi memiliki overhead latensi jaringan, aspek Git satu ini akan membuat anda berpikir bahwa para dewa kecepatan telah memberkati Git dengan kekuatan. Karena anda memiliki seluruh sejarah dari proyek di lokal disk anda, dengan kebanyakan operasi yang tampak hampir seketika.

Sebagai contoh, untuk melihat history dari proyek, Git tidak membutuhkan data histori dari server untuk kemudian

menampilkannya untuk anda, namun secara sederhana Git membaca historinya langsung dari basisdata lokal proyek tersebut. Ini berarti anda melihat histori proyek hampir secara instant. Jika anda ingin membandingkan perubahan pada sebuah berkas antara versi saat ini dengan versi sebulan yang lalu, Git dapat mencari berkas yang sama pada sebulan yang lalu dan melakukan perbandingan perubahan secara lokal, bukan dengan cara meminta remote server melakukannya atau meminta server mengirimkan berkas versi yang lebih lama kemudian membandingkannya secara lokal.

Hal ini berarti bahwa sangat sedikit yang tidak bisa anda kerjakan jika anda sedang offline atau berada diluar VPN. Jika anda sedang berada dalam pesawat terbang atau sebuah kereta dan ingin melakukan pekerjaan kecil, anda dapat melakukan commit sampai anda memperoleh koneksi internet hingga anda dapat menguploadnya. Jika anda pulang ke rumah dan VPN client anda tidak bekerja dengan benar, anda tetap dapat bekerja. Pada kebanyakan sistem lainnya, melakukan hal ini cukup sulit atau bahkan tidak mungkin sama sekali. Pada Perforce misalnya, anda tidak dapat berbuat banyak ketika anda tidak terhubung dengan server; pada Subversion dan CVS, anda dapat mengubah berkas, tapi anda tidak dapat melakukan commit pada basisdata anda (karena anda tidak terhubung dengan basisdata). Hal ini mungkin saja bukanlah masalah yang besar, namun anda akan terkejut dengan perbedaan besar yang disebabkan.

Git Memiliki Integritas

Segala sesuatu pada Git akan melalui proses checksum terlebih dahulu sebelum disimpan yang kemudian direferensikan oleh hasil checksum tersebut. Hal ini berarti tidak mungkin melakukan perubahan terhadap berkas manapun tanpa diketahui oleh Git. Fungsionalitas ini dimiliki oleh Git pada level terendahnya dan ini merupakan bagian tak terpisahkan dari filosofi Git. Anda tidak akan kehilangan

informasi atau mendapatkan file yang cacat tanpa diketahui oleh Git.

Mekanisme checksum yang digunakan oleh Git adalah SHA-1 hash. Ini merupakan sebuah susunan string yang terdiri dari 40 karakter heksadesimal (0 hingga 9 dan a hingga f) dan dihitung berdasarkan isi dari sebuah berkas atau struktur direktori pada Git.

11.1.1.1 Secara Umum Git Hanya Menambahkan Data Ketika anda melakukan operasi pada Git, kebanyakan dari operasi tersebut hanya menambahkan data pada basisdata Git. It is very difficult to get the system to do anything that is not undoable or to make it erase data in any way. Seperti pada berbagai VCS, anda dapat kehilangan atau mengacaukan perubahan yang belum di-commit; namun jika anda melakukan commit pada Git, akan sangat sulit kehilanngannya, terutama jika anda secara teratur melakukan push basisdata anda pada repositori lain.

Hal ini menjadikan Git menyenangkan karena kita dapat berexperimen tanpa kekhawatiran untuk mengacaukan proyek. Untuk lebih jelas dan dalam lagi tentang bagaimana Git menyimpan datanya dan bagaimana anda dapat mengembalikan yang hilang, lihat "Under the Covers" pada Bab 9.

11.1.1.2 Tiga Keadaan Sekarang perhatikan. Ini adalah hal utama yang harus diingat tentang Git jika anda ingin proses belajar anda berjalan lancar. Git memiliki 3 keadaan utama dimana berkas anda dapat berada: committed, modified dan staged. Committed berarti data telah tersimpan secara aman pada basisdata lokal. Modified berarti anda telah melakukan perubahan pada berkas namun anda belum melakukan commit pada basisdata. Staged berarti anda telah menandai

berkas yang telah diubah pada versi yang sedang berlangsung untuk kemudian dilakukan commit.

Direktori Git adalah dimana Git menyimpan metadata dan database objek untuk proyek anda. Ini adalah bagian terpenting dari Git, dan inilah yang disalin ketika anda melakukan kloning sebuah repository dari komputer lain.

Direktori kerja adalah sebuah checkout tunggal dari satu versi dari proyek. Berkas-berkas ini kemudian ditarik keluar dari basisdata yang terkompresi dalam direktori Git dan disimpan pada disk untuk anda gunakan atau modifikasi.

Staging area adalah sebuah berkas sederhana, umumnya berada dalam direktori Git anda, yang menyimpan informasi mengenai apa yang menjadi commit selanjutnya. Ini terkadang disebut sebagai index, tetapi semakin menjadi standard untuk menyebutnya sebagai staging area.

Tabel 11.1 Alur kerja dasar git

No	Keterangan
.1	Mengubah berkas dalam direktori kerja
.2	Menambahkan snapshotnya ke staging area
.3	Commit, Ambil berkas di staging area dan menyimpan snapshotnya ke direktori Git

Jika sebuah versi tertentu dari sebuah berkas telah ada di direktori git, ia dianggap 'committed'. Jika berkas diubah (modified) tetapi sudah ditambahkan ke staging area, maka itu adalah 'staged'. Dan jika berkas telah diubah sejak terakhir dilakukan checked out tetapi belum ditambahkan ke staging area maka itu adalah 'modified'. Pada Bab 2, anda akan mempelajari lebih lanjut mengenai keadaan-keadaan ini dan bagaimana anda dapat memanfaatkan keadaan-keadaan tersebut ataupun melewati bagian 'staged' seluruhnya.

Modifikasi Fungsi yang Ada

Tom melakukan operasi kloning dan menemukan file baru `string.c`. Dia ingin tahu siapa yang menambahkan file ini ke repositori dan untuk tujuan apa, maka, dia menjalankan perintah `git log`.

```
[tom@CentOS] git clone gi-
tuser@git.server.com:project.git
```

Perintah di atas akan menghasilkan hasil sebagai berikut:

```
Initialized empty Git repository in /home/tom/project/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
Receiving objects: 100% (6/6), 726 bytes, done.
remote: Total 6 (delta 0), reused 0 (delta 0)
```

Operasi Clone akan membuat direktori baru di dalam direktori kerja saat ini. Dia mengubah direktori ke direktori yang baru dibuat dan menjalankan perintah `git log`.

```
[tom@CentOS ] cd project/
[tom@CentOSproject]git log
```

Perintah di atas akan menghasilkan hasil sebagai berikut:

```
commit d1e19d316224cddc437e3ed34ec3c931ad803958
Author: Jerry Mouse <jerry@tutorialspoint.com>
Date: Wed Sep 11 08:05:26 2013 +0530
Changed return type of my_strlen to size_t
commit 19ae20683fc460db7d127cf201a1429523b0e319
Author: Tom Cat <tom@tutorialspoint.com>
Date: Wed Sep 11 07:32:56 2013 +0530
Initial commit
```

Setelah mengamati log, dia menyadari bahwa file `string.c` ditambahkan oleh Jerry untuk mengimplementasikan operasi string dasar. Dia penasaran dengan kode Jerry. Jadi dia membuka `string.c` di editor teks dan langsung menemukan bug. Dalam fungsi `my_strlen`, Jerry tidak menggunakan pointer konstan. Jadi, dia memutuskan untuk memo-

difikasi kode Jerry. Setelah modifikasi, kode tersebut terlihat seperti berikut:

```
[tom@CentOS project]$ git diff
```

Perintah di atas akan menghasilkan hasil sebagai berikut:

```
diff --git a/string.c b/string.c
index 7da2992..32489eb 100644
      - - -a/string.c                                (11.1)
```

```
+++ b/string.c
@@ -1,8 +1,8 @@
#include <stdio.h>
-size_t my_strlen(char *s)
+size_t my_strlen(const char *s)
{
- char *p = s;
+ const char *p = s;
while (*p)
++p;
}
```

Setelah melakukan pengujian, dia menyimpan perubahannya. `[tom@CentOSproject] git status -s M string.c ??`

string

```
[tom@CentOSproject ] git add string.c
[tom@CentOSproject ] git commit -m 'Changed char
pointer to const char pointer' [master cea2c00] Changed
char pointer to const char pointer 1 files changed, 2 inser-
tions(+), 2 deletions(-)
[tom@CentOSproject ] git log
```

perintah diatas akan menghasilkan hasil sebagai berikut:

```
commit cea2c000f53ba99508c5959e3e12fff493b Author:
Tom Cat <tom@tutorialspoint.com> Date: Wed Sep 11
08:32:07 2013 +0530
```

```
Changed char pointer to const char pointer
```

```
commit d1e19d316224cddc437e3ed34ec3c931ad803958
Author: Jerry Mouse <jerry@tutorialspoint.com>
Date: Wed Sep 11 08:05:26 2013 +0530
Changed return type of my_strlen to size_t
commit 19ae20683fc460db7d127cf201a1429523b0e319
Author: Tom Cat <tom@tutorialspoint.com>
Date: Wed Sep 11 07:32:56 2013 +0530
Initial commit
```

Tom menggunakan git push untuk melakukan push atas perubahan yang dilakukannya.

```
[tom@CentOSproject] git push origin master
```

perintah diatas akan menghasilkan seperti berikut:

```
Counting objects: 5, done.
```

```
Compressing objects: 100 Writing objects: 100 Total 3
```

```
(delta 1), reused 0 (delta 0)
```

```
To gituser@git.server.com:project.git
```

```
d1e19d3..cea2c00 master ↵ master
```


STASH OPERATION

12.1 Penjelasan

Salah satu fitur menarik di Git adalah *stashing*. Dengan melakukan *stashing*, developer dapat dengan mudah ' *menyapkan*' perubahan kode program, melakukan perubahan lain, lalu kembali lagi ke kode program yang sebelumnya dikerjakan. Langkah-langkah tersebut sebenarnya dapat diwakili oleh beberapa perintah Git lainnya, tapi *stashing* membuatnya menjadi mudah karena hanya perlu memanggil satu perintah.

Sebagai contoh, saya akan menggunakan Git yang diakses melalui IntelliJ IDEA. Saya membuat sebuah proyek Groovy baru dengan nama *latihan*. Lalu saya membuat se-

buah script Groovy sederhana bernama *Latihan.groovy* untuk menghitung sisa inventory secara FIFO, seperti berikut ini:

```
List pembelian = [10, 20, 30, 40]
List penjualan = [5, 6, 3, 2, 1, 3]
println stokFifo(pembelian, hitungTotal(pembelian, pen-
jualan))
int hitungTotal(List pembelian, List penjualan) {
    pembelian.sum() - penjualan.sum()}
List stokFifo(List pembelian, int sisa) {
    List hasil = []
    pembelian.each { int jumlah ->
    if (sisa > 0) {
        int delta = (jumlah >= sisa)? jumlah: sisa // BUG YANG
        DISENGAJA!
        sisa -= delta
        hasil << delta
    }
    }
    hasil
}
```

Untuk menambahkan proyek ke dalam repository Git, pilih menu VCS, Import into Version Control, Create Git Repository.

Pada kotak dialog yang muncul, pilih untuk meletakkan repository bersamaan dengan lokasi proyek dan kemudian klik tombol OK.

Kemudian, klik kanan pada file *Latihan.groovy*, memilih menu *Git, Add*

Sekarang file tersebut telah berada di lokasi *index* atau *staging* dari Git. Berikutnya, commit perubahan. Caranya adalah dengan memilih menu *VCS, Commit Changes*.

Anggap saja ini adalah sebuah aplikasi yang dikembangkan bersama dengan developer lain, yaitu xXx. xXx tersebut telah men-*fetch* repository saya dari remote/upstream, kemudian ia akan men-develop kode program yang berhubungan dengan kode program saya di atas.

$$Mappembelian = [10 : 10000, 20 : 10100, 30 : 10200, 40 : 11000] \quad (12.1)$$

```

    int hitungTotal(List pembelian, List penjualan)
    pembelian.sum() - penjualan.sum()
    List stokFifo(Map pembelian, int sisa)
    List hasil = []
    pembelian.each { jumlah, hargaBeli ->
    if (sisa < 0)
    int delta = (jumlah <= sisa)? jumlah: sisa // BUG YANG
    DISENGAJA!

```

Saya belum selesai mengetik, bahkan belum sempat menjalankan kode program ketika tiba-tiba telepon berdering.

Suara xXx terdengar sangat panik. Dia mengatakan bahwa 30 menit lagi dirinya harus memberikan presentasi ke pengguna, tapi ada yang aneh dengan perhitungan inventory buatan saya! xXx menemukan sebuah kesalahan. Lebih dari itu, xXx meminta saya untuk segera memperbaikinya secepat mungkin sehingga dia bisa men-pull perbaikan dari saya!

Masalahnya: saya sudah mengubah banyak kode program tersebut sehingga tidak sama lagi seperti yang dipakai oleh Lena. Saya tidak ingin menghapus perubahan yang sudah saya buat sejauh ini karena nantinya perubahan ini pasti akan dipakai.

Salah satu solusi yang dapat saya pakai adalah dengan memakai fasilitas `stash` di Git. Saya memilih menu `VCS, Git, Stash Changes`

Setelah men-klik tombol `Create Stash`, isi kode program saya secara ajaib kembali lagi seperti semula! Sama persis seperti `commit` terakhir yang dirujuk oleh `HEAD`. Saya segera memanfaatkan kesempatan ini untuk mencari dan memperbaiki kesalahan yang ditemukan xXx:

```
List pembelian = [10, 20, 30, 40]
List penjualan = [5, 6, 3, 2, 1, 3]
println stokFifo(pembelian, hitungTotal(pembelian,
    penjualan))
int hitungTotal(List pembelian, List penjualan) {
    pembelian.sum() - penjualan.sum()
}
List stokFifo(List pembelian, int sisa) {
    List hasil = []
    pembelian.each { int jumlah -> if (sisa > 0) {
        int delta = (jumlah >= sisa) ? sisa : jumlah
        sisa -= delta
        hasil << delta
    }
}
hasil }
```

Saya segera men-commit perubahan dengan memilih menu `VCS, Commit Changes...` Pada komentar, saya mengisi dengan 'Perbaikan perhitungan sisa inventory yang salah.' dan men-klik tombol **Commit**. Saya kemudian men-*push* perubahan ke *upstream*, sehingga Lena bisa men-*pull* perubahan dari saya. Tidak lama kemudian suara Lena

terdengar lega seolah-olah baru saja luput dari malapetaka. Ia memberitahukan bahwa kini semuanya baik-baik saja.

Lalu bagaimana dengan kode program yang sedang saya 'ketik' sebelum menerima telepon dari Lena? Kemana perginya? Saya bisa mengembalikannya dengan memilih menu **VCS, Git ,UnStash Changes....** Pada kotak dialog yang muncul, saya men-klik tombol **Pop Stash**

Kode program saya akan kembali seperti terakhir kali:

```
Map pembelian = [10: 10000, 20: 10100, 30: 10200,
40: 11000]
Map penjualan = [5: 12000, 6: 13000, 3: 11000,
2: 11000, 1: 15000, 3: 12000]
println stokFifo(pembelian, hitungTotal(pembelian.keySet().toList(),
penjualan.keySet().toList()))
int hitungTotal(List pembelian, List penjualan) {
    pembelian.sum() - penjualan.sum()
}
List stokFifo(Map pembelian, int sisa) {
    List hasil = []
    pembelian.each { jumlah, hargaBeli ->
        if (sisa > 0) {
            int delta = (jumlah >= sisa)? sisa: jumlah
            sisa -= delta
            hasil << [delta, hargaBeli]
        }
    }
    int hitungProfit(List stokFifo, Map penjualan) {
        def
```

Tunggu dulu! Tidak persis sama seperti terakhir kali!! Fasilitas *stashing* bukan saja hanya mengembalikan kode program sebelumnya, tetapi juga melakukan merging dengan perubahan saat ini. Perbaikan yang diminta oleh Lena tidak hilang setelah saya men-pop stash.

Pada contoh ini, hanya terdapat satu file yang berubah. Fasilitas *stashing* akan semakin berguna bila perubahan telah dilakukan pada banyak file yang berbeda. Tanpa Git dan *stashing*, pada kasus ini, saya harus memakai cara manual yang lebih repot seperti memberikan komentar atau mengedit file untuk sementara.

12.1.1 Membatalkan Apapun

Pada setiap tahapan, Anda mungkin ingin membatalkan sesuatu. Di sini, kita akan membahas beberapa alat dasar untuk membatalkan perubahan yang baru saja Anda lakukan. Harus tetap diingat bahwa kita tidak selalu dapat membatalkan apa yang telah kita batalkan. Ini adalah salah satu area dalam Git yang dapat membuat Anda kehilangan apa yang telah Anda kerjakan jika Anda melakukannya dengan tidak tepat.

Merubah Commit Terakhir Anda

Salah satu pembatalan yang biasa dilakukan adalah ketika kita melakukan commit terlalu cepat dan mungkin terjadi lupa untuk menambah beberapa berkas, atau Anda salah memberikan pesan commit Anda. Jika Anda ingin untuk mengulang commit tersebut, Anda dapat menjalankan commit dengan opsi–amend:

```
$ git commit –amend
```

Perintah ini mengambil area stage Anda dan menggunakannya untuk commit. Jika Anda tidak melakukan perubahan apapun sejak commit terakhir Anda (seumpama, Anda menjalankan perintah ini langsung setelah commit Anda sebelumnya), maka snapshot Anda akan sama persis dengan sebelumnya dan yang Anda dapat ubah hanyalah pesan commit Anda.

Pengolah kata akan dijalankan untuk mengedit pesan commit yang telah Anda buat pada commit sebelumnya. Anda dapat ubah pesan commit ini seperti biasa, tetapi pesan commit sebelumnya akan tertimpa.

Sebagai contoh, jika Anda melakukan commit dan menyadari bahwa Anda lupa untuk memasukkan beberapa perubahan dalam sebuah berkas ke area stage dan Anda in-

gin untuk menambahkan perubahan ini ke dalam commit terakhir, Anda dapat melakukannya sebagai berikut:

```
\$ git commit -m 'initial commit'\vspace{14pt}
```

```
\$ git add forgotten\_file\vspace{14pt}
```

```
\$ git commit --amend\vspace{14pt}
```

Ketiga perintah ini tetap akan bekerja di satu commit - commit kedua akan menggantikan hasil dari commit pertama.

Mengeluarkan Berkas dari Area Stage

Dua seksi berikutnya akan menunjukkan bagaimana menangani area stage Anda dan perubahan terhadap direktori kerja Anda. Sisi baiknya adalah perintah yang Anda gunakan untuk menentukan keadaan dari kedua area tersebut juga mengingatkan Anda bagaimana membatalkan perubahannya. Sebagai contoh, mari kita anggap Anda telah merubah dua berkas dan ingin melakukan commit kepada keduanya sebagai dua perubahan terpisah, tetapi Anda secara tidak sengaja mengetikkangit add *dan memasukkan keduanya ke dalam area stage. Bagaimana Anda dapat mengeluarkan salah satu dari keduanya? Perintahgit status-mengingatkan Anda:

```
$ git add
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
# modified: README.txt
```

```
# modified: benchmarks.rb
```

```
#
```

Tepat di bawah tulisan "Changes to be committed", tercantum anjuran untuk menggunakan `git reset HEAD <file>` untuk mengeluarkan dari area stage. Mari kita gunakan anjuran tersebut untuk mengeluarkan berkas `benchmarks.rb` dari area stage:

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
# modified: README.txt
```

```
#
```

```
# Changes not staged for commit:
```

```
# (use "git add <file>..." to update what will be committed)
```

```
# (use "git checkout -- <file>..." to discard changes in working directory)
```

```
#
```

```
# modified: benchmarks.rb
```

#

Perintahnya terlihat agak aneh, tetapi menyelesaikan masalah. Berkas `benchmarks.rb` sekarang menjadi terubah dan sudah berada di luar area stage.

Mengembalikan Berkas Terubah

Apa yang terjadi jika Anda menyadari bahwa Anda tidak ingin menyimpan perubahan terhadap berkas `benchmarks.rb`? Bagaimana kita dapat dengan mudah mengembalikan berkas tersebut ke keadaan yang sama dengan saat Anda melakukan commit terakhir (atau saat awal menduplikasi, atau bagaimanapun Anda mendapatkannya ketika masuk ke direktori kerja Anda)? Untungnya, `git status` memberitahu Anda lagi bagaimana untuk melakukan hal itu. Pada contoh keluaran sebelumnya, area direktori kerja terlihat seperti berikut:

```
# Changes not staged for commit:
```

```
# (use "git add <file>..." to update what will be committed)
```

```
# (use "git checkout - <file>..." to discard changes in working directory)
```

#

```
# modified: benchmarks.rb
```

#

Terlihat secara eksplisit cara Anda dapat membuang perubahan yang telah Anda lakukan (paling tidak, hanya versi Git 1.6.1 atau yang lebih baru yang memperlihatkan cara ini - jika Anda memiliki versi yang lebih tua, kami sangat merekomendasikan untuk memperbaharui Git untuk mendapatkan fitur yang lebih nyaman digunakan). Mari kita lakukan apa yang tertulis di atas:


```
$ git checkout – benchmarks.rb
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
# modified: README.txt
```

```
#
```

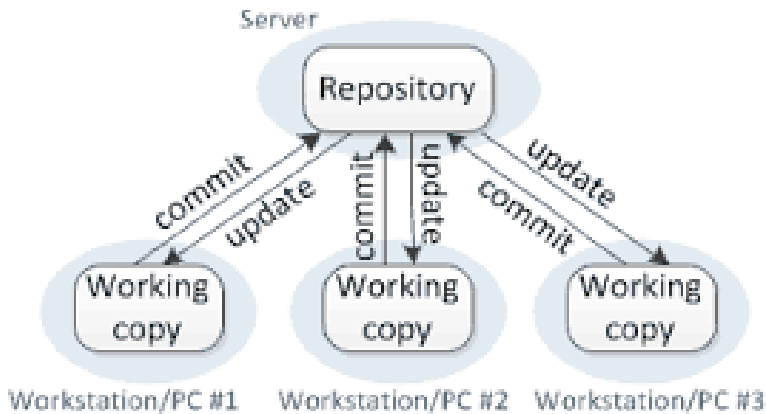
Anda dapat lihat bahwa perubahan telah dikembalikan. Anda juga seharusnya menyadari bahwa perintah ini juga berbahaya: perubahan apapun yang Anda buat di berkas tersebut akan hilang - Anda baru saja menyalin berkas lain ke perubahan Anda. Jangan pernah gunakan perintah ini kecuali Anda sangat yakin bahwa Anda tidak menginginkan berkas tersebut. Jika Anda hanya butuh untuk menyingkirkan perubahan untuk sementara, kita dapat bahas tentang penyimpanan (*to stash*) dan pencabangan (*to branch*) di bab berikutnya; kedua cara tersebut secara umum adalah cara yang lebih baik untuk dilakukan.

Ingat bahwa apapun yang dicommit di dalam Git dapat hampir selalu dikembalikan. Bahkan commit yang berada di cabang yang sudah terhapus ataupun commit yang sudah ditimpa dengan commit –amend masih dapat dikembalikan. Namun, apapun hilang yang belum pernah dicommit besar kemungkinan tidak dapat dilihat kembali.

12.2 Sistem Kontrol Versi

12.1:

Centralized version control



Gambar 12.1 Version Control System

Version Control System (VCS) adalah perangkat lunak yang membantu pengembang perangkat lunak untuk bekerja sama dan menjaga sejarah lengkap dari pekerjaan mereka.

Tabel 12.1 Fungsi VCS

No	Keterangan
.1	Memungkinkan pengembang untuk bekerja secara bersamaan
.2	Tidak memungkinkan Timpa perubahan masing-masing
.3	Mempertahankan sejarah setiap versi

Berikut ini adalah jenis VCS:

- sistem kontrol versi terpusat (CVCS).
- Didistribusikan / Desentralisasi sistem kontrol versi (DVCS).

Dalam bab ini, kita akan berkonsentrasi hanya pada sistem kontrol versi didistribusikan dan terutama pada Git. Git berada di bawah sistem kontrol versi terdistribusi.

12.2.1 Distributed Sistem Kontrol Versi

sistem terpusat kontrol versi (CVCS) menggunakan server pusat untuk menyimpan semua file dan memungkinkan kolaborasi tim. Tapi kelemahan utama dari CVCS adalah titik tunggal kegagalan, yaitu, kegagalan server pusat. Sayangnya, jika server pusat turun selama satu jam, kemudian pada jam itu, tidak ada yang bisa berkolaborasi sama sekali. Dan bahkan dalam kasus terburuk, jika disk server pusat akan rusak dan cadangan yang tepat belum diambil, maka Anda akan kehilangan seluruh sejarah proyek. Di sini, sistem terdistribusi kontrol versi (DVCS) datang ke dalam gambar.

DVCS klien tidak hanya memeriksa snapshot terbaru dari direktori tetapi mereka juga penuh dari repositori tersebut. Jika memutuskan turun, maka repositori dari klien dapat disalin kembali ke server untuk mengembalikannya. Setiap checkout adalah salinan lengkap dari repositori. Git tidak bergantung pada server pusat dan itulah sebabnya Anda dapat melakukan banyak operasi ketika Anda sedang offline. Anda dapat melakukan perubahan, membuat cabang, lihat log, dan melakukan operasi lain ketika Anda sedang offline. Anda memerlukan koneksi jaringan hanya untuk mempublikasikan perubahan dan mengambil perubahan terbaru.

12.2.2 Keuntungan dari Git

free dan open source

Git dirilis di bawah lisensi open source GPL ini. Ini tersedia secara bebas melalui internet. Anda dapat menggunakan Git untuk mengelola proyek kepatutan tanpa membayar satu sen dolar. Karena merupakan open source, Anda dapat mendownload kode sumbernya dan juga melakukan perubahan sesuai dengan kebutuhan Anda.

Cepat dan kecil

Karena sebagian besar operasi dilakukan secara lokal, memberikan manfaat yang sangat besar dalam hal kecepatan. Git tidak bergantung pada server pusat; itu sebabnya, tidak ada kebutuhan untuk berinteraksi dengan server remote untuk setiap operasi. Bagian inti dari Git ditulis dalam C, yang menghindari overhead runtime yang terkait dengan bahasa tingkat tinggi lainnya. Meskipun Git cermin seluruh repositori, ukuran data di sisi client kecil. Ini menggambarkan efisiensi Git di mengompresi dan menyimpan data di sisi client.

backup implisit

Kemungkinan kehilangan data sangat jarang ketika ada beberapa salinan dari itu. Data hadir di setiap sisi klien cermin repositori, karena itu dapat digunakan dalam hal terjadi kecelakaan atau korupsi disk.

Keamanan

Git menggunakan fungsi hash kriptografi umum yang disebut fungsi hash aman (SHA1), untuk nama dan mengidentifikasi objek dalam database. Setiap file dan komit check-dijumlahkan dan diambil oleh checksum-nya pada saat checkout. Ini menyiratkan bahwa, tidak mungkin untuk mengubah file, tanggal, dan pesan komit dan data lainnya dari database Git tanpa mengetahui Git.

Tidak perlu perangkat keras yang kuat

Dalam kasus CVCS, server pusat harus cukup kuat untuk melayani permintaan dari seluruh tim. Untuk tim yang lebih kecil, itu tidak masalah, tetapi sebagai ukuran tim tumbuh, keterbatasan hardware server dapat menjadi hambatan kinerja. Dalam kasus DVCS, pengembang tidak berinteraksi dengan server kecuali mereka butuhkan untuk mendorong atau menarik perubahan. Semua angkat berat terjadi pada

sisi klien, sehingga hardware server dapat memang sangat sederhana.

percabangan mudah

CVCS menggunakan mekanisme copy murah, Jika kita membuat cabang baru, itu akan menyalin semua kode ke cabang baru, sehingga memakan waktu dan tidak efisien. Juga, penghapusan dan penggabungan cabang di CVCS rumit dan memakan waktu. Tapi manajemen cabang dengan Git sangat sederhana. Dibutuhkan hanya beberapa detik untuk membuat, menghapus, dan menggabungkan cabang.

BAB 13

MOVE OPERATION

13.1 Move Operation

Seperti namanya, operasi memindahkan direktori atau file dari satu lokasi ke lokasi lain. direktori yang dimodifikasi akan muncul sebagai berikut:

```
[tom@CentOS project] $ \ $ $ pwd}
/home/tom/project}
[tom@CentOS project] $ \ $ $ ls}

{README string string.c}
[tom@CentOS project] $ \ $ $ mkdir src

[tom@CentOS project] $ \ $ $ git mv string.c
```

```
[tom@CentOS project] $ \ $ git status -s  
R string.c $ - $> src/string.c  
?? string}
```

Untuk membuat perubahan ini permanen, harus mendorong struktur direktori yang dimodifikasi ke repositori jauh sehingga pengembang lain dapat melihat ini.

```
[tom@CentOS project] $ git commit -m "Modified directory structure"
```

```
[master 7d9ea97] Modified directory structure  
1 files changed, 0 insertions(+), 0 deletions(-)  
rename string.c => src/string.c (100 %)
```

```
[tom@CentOS project] $ git push origin master  
Counting objects: 4, done.  
Compressing objects: 100 % (2/2), done.  
Writing objects: 100 % (3/3), 320 bytes, done.  
Total 3 (delta 0), reused 0 (delta 0)  
To gituser@git.server.com:project.git  
e86f062..7d9ea97 master -> master
```

Di gudang lokal Jerry, sebelum operasi penarikan, ia akan menunjukkan struktur direktori lama.

```
[jerry@CentOS project] $ pwd  
/home/jerry/jerry_repo/project
```

```
[jerry@CentOS project] $ ls  
README string string.c
```

Tapi setelah operasi tarik, struktur direktori akan diperbarui. Sekarang, Jerry bisa melihat direktori src dan file yang ada di dalam direktori itu.

```
[jerry@CentOS project] $ git pull  
remote: Counting objects: 4, done.
```

```

remote: Compressing objects: 100 % (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100 % (3/3), done.
From git.server.com:project
e86f062..7d9ea97 master -> origin/master
First, rewinding head to replay your work on top of it...
Fast-forwarded                master                to
7d9ea97683da90bcd87c28ec9b4f64160673c8a.

```

```

[jerry@CentOS project] $ ls
README src string

```

```

[jerry@CentOS project] $ ls src/
string.c

```

13.1.1 Semua Operasi Dilakukan Secara Lokal

Kebanyakan operasi pada Git hanya membutuhkan berkas-berkas dan resource lokal tidak ada informasi yang dibutuhkan dari komputer lain pada jaringan. Jika terbiasa dengan VCS terpusat dimana kebanyakan operasi memiliki overhead latensi jaringan, aspek Git satu ini akan membuat berpikir bahwa para dewa kecepatan telah memberkati Git dengan kekuatan. Karena memiliki seluruh sejarah dari proyek di lokal disk, dengan kebanyakan operasi yang tampak hampir seketika.

Sebagai contoh, untuk melihat history dari proyek, Git tidak membutuhkan data histori dari server untuk kemudian menampilkannya untuk, namun secara sadar Git membaca historinya langsung dari basisdata lokal proyek tersebut. Ini berarti melihat histori proyek hampir secara instant. Jika ingin membandingkan perubahan pada sebuah berkas antara versi saat ini dengan versi sebulan yang lalu, Git dapat mencari berkas yang sama pada sebulan yang lalu dan melakukan perbandingan perubahan secara lokal, bukan dengan cara meminta remote server melakukannya atau meminta server mengirimkan berkas versi yang lebih lama kemudian membandingkannya secara lokal.

Hal ini berarti bahwa sangat sedikit yang tidak bisa dikerjakan jika sedang offline atau berada diluar VPN. Jika sedang berada dalam pesawat terbang atau sebuah kereta dan ingin melakukan pekerjaan kecil, dapat melakukan commit sampai memperoleh koneksi internet hingga dapat menguploadnya. Jika pulang ke rumah dan VPN client tidak bekerja dengan benar, tetap dapat bekerja. Pada kebanyakan sistem lainnya, melakukan hal ini cukup sulit atau bahkan tidak mungkin sama sekali. Pada Perforce misalnya, tidak dapat berbuat banyak ketika tidak terhubung dengan server; pada Subversion dan CVS, dapat mengubah berkas, tapi tidak dapat melakukan commit pada basisdata (karena tidak terhubung dengan basisdata). Hal ini mungkin saja bukanlah masalah yang besar, namun akan terkejut dengan perbedaan besar yang disebabkan.

13.1.2 Git Memiliki Integritas

Segala sesuatu pada Git akan melalui proses checksum terlebih dahulu sebelum disimpan yang kemudian direferensikan oleh hasil checksum tersebut. Hal ini berarti tidak mungkin melakukan perubahan terhadap berkas manapun tanpa diketahui oleh Git. Fungsionalitas ini dimiliki oleh Git pada level terendahnya dan ini merupakan bagian tak terpisahkan dari filosofi Git. Tidak akan kehilangan informasi atau mendapatkan file yang cacat tanpa diketahui oleh Git.

Mekanisme checksum yang digunakan oleh Git adalah SHA-1 hash. Ini merupakan sebuah susunan string yang terdiri dari 40 karakter heksadesimal (0 hingga 9 dan a hingga f) dan dihitung berdasarkan isi dari sebuah berkas atau struktur direktori pada Git. sebuah hash SHA-1 berupa seperti berikut:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Nilai seperti ini pada berbagai tempat di Git. Faktanya, Git tidak menyimpan nama berkas pada basisdatanya, melainkan nilai hash dari isi berkas.

Ketika melakukan operasi pada Git, kebanyakan dari operasi tersebut hanya menambahkan data pada basis-

data Git. Seperti pada berbagai VCS, dapat kehilangan atau mengacaukan perubahan yang belum di-commit; namun jika melakukan commit pada Git akan sangat sulit kehilangannya, terutama jika secara teratur melakukan push basisdata pada repositori lain.

Hal ini menjadikan Git menyenangkan karena kita dapat berexperimen tanpa kekhawatiran untuk mengacaukan proyek. Git memiliki 3 keadaan utama dimana berkas dapat berada: committed, modified dan staged. Committed berarti data telah tersimpan secara aman pada basisdata lokal. Modified berarti telah melakukan perubahan pada berkas namun belum melakukan commit pada basisdata. Staged berarti telah menandai berkas yang telah diubah pada versi yang sedang berlangsung untuk kemudian dilakukan commit.

Direktori Git adalah dimana Git menyimpan metadata dan database objek untuk proyek. Ini adalah bahagian terpenting dari Git, dan inilah yang disalin ketika melakukan kloning sebuah repository dari komputer lain.

Direktori kerja adalah sebuah checkout tunggal dari satu versi dari proyek. Berkas-berkas ini kemudian ditarik keluar dari basisdata yang terkompresi dalam direktori Git dan disimpan pada disk untuk digunakan atau modifikasi.

Staging area adalah sebuah berkas sederhana, umumnya berada dalam direktori Git, yang menyimpan informasi mengenai apa yang menjadi commit selanjutnya. Ini terkadang disebut sebagai index, tetapi semakin menjadi standard untuk menyebutnya sebagai staging area. Alur kerja dasar Git adalah seperti ini:

Jika sebuah versi tertentu dari sebuah berkas telah ada di direktori git dianggap 'committed'. Jika berkas diubah (modified) tetapi sudah ditambahkan ke staging area maka itu adalah 'staged'. Dan jika berkas telah diubah sejak terakhir dilakukan checked out tetapi belum ditambahkan ke staging area maka itu adalah 'modified'.

13.3 Perintah Untuk Membuat Sebuah Proyek

Membuat direktori baru di repositori Git dengan git init. Melakukan direktori setiap saat, benar-benar lokal. Ex-

ecutive git init dalam direktori, Membuat Git repositori. Sebagai contoh, buat item w3big:

```
$ mkdir w3big
```

```
$ cd w3big/
```

```
$ git init
```

```
Initialized empty Git repository in /Users/tianqixin/www/w3big/.git/
```

```
# /www/w3big/.git/ Git
```

Sekarang dapat melihat subdirektori git yang dihasilkan dalam proyek. Ini adalah repositori Git, dan semua data yang terkait dengan snapshot dari proyek disimpan di sini.

```
ls -a
```

Gunakan git clone repositori Git untuk salinan lokal, sehingga dapat melihat item atau memodifikasinya. Jika membutuhkan sebuah proyek kerjasama dengan orang lain atau ingin menyalin sebuah proyek, melihat kode, dapat mengkloning proyek. Jalankan:

```
git clone [url]
```

Sebagai contoh, kloning proyek pada Github:

```
$ git clone git@github.com:schacon/simplegit.git
```

```
Cloning into 'simplegit'...
```

```
remote: Counting objects: 13, done.
```

```
remote: Total 13 (delta 0), reused 0 (delta 0), pack-reused 13
```

```
Receiving objects: 100 % (13/13), done.
```

```
Resolving deltas: 100 % (2/2), done.
```

```
Checking connectivity... done.
```

Setelah kloning selesai di direktori saat ini akan menghasilkan simplegit direktori:

```
$ Cd simplegit / $ ls README Rakefile lib
```

operasi akan menyalin semua catatan proyek.

```
$ ls -a
```

```
git README Rakefile lib
```

```
$ cd .git
```

```
$ ls
```

```
HEAD    description info    packed-refs
branches hooks    logs    refs
config  index    objects
```

Secara default, Git akan mengikuti nama URL yang tersedia item untuk membuat direktori proyek lokal ditunjukkan. URL biasanya nama item terakhir / setelah. Jika ingin nama yang berbeda dapat menambahkan nama yang diinginkan setelah perintah.

13.4 Snapshot Dasar

Pekerjaan Git adalah untuk membuat dan menyimpan snapshot dari proyek dan setelah snapshot dan membandingkan. Bab ini akan tentang menciptakan sebuah snapshot dari proyek dan mengirimkan pengenalan perintah. Git add perintah untuk menambahkan file ke cache, seperti yang menambahkan dua file berikut:

```
$ touch README
$ touch hello.php
$ ls
README hello.php
$ git status -s
?? README
?? hello.php
$
```

Perintah git status digunakan untuk melihat status proyek. Selanjutnya jalankan git add perintah untuk menambahkan file:

```
$ git add README hello.php
```

Sekarang jalankan git status, dapat melihat dua dokumen tersebut telah ditambahkan untuk pergi :

```
$ git status -s
A README
A hello.php
$
```

Proyek baru, menambahkan semua file yang sama, kita dapat menggunakan `git add`. Perintah untuk menambahkan semua file dalam proyek saat ini. Sekarang memodifikasi file `README`:

```
$ vim README
i pre
i p README L b # w3big Git i/b i/p
i p git status i/p
$ git status -s
AM README
A hello.php
```

”AM” status berarti bahwa file tersebut setelah kami menemukannya ke cache ada perubahan. Setelah perubahan menjalankan `git add` perintah untuk menemukannya ke cache:

```
$ git add .
$ git status -s
A README
A hello.php
```

Bila ingin perubahan yang terkandung dalam snapshot laporan yang akan datang dalam waktu, harus menjalankan `git add`.

`Git status` untuk melihat setelah komit terakhir jika ada perubahan. Menunjukkan perintah ini ketika ditambahkan `-s` parameter untuk mendapatkan hasil yang singkat. Jika tidak menambahkan parameter ini akan keluaran rinci:

```
$ git status
On branch master
Initial commit
```

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: README

new file: hello.php

Status git diff git eksekutif untuk melihat rincian hasil eksekusi. Git perintah diff dan menampilkan cache write telah dimodifikasi tapi belum ditulis ke cache perubahan perbedaan. git diff Ada dua skenario utama.

- Perubahan tidak cache:diff git
- Lihatperubahan cache: git diff --cached
- Lihat cache dan uncached semuaperubahan: git diff KEPALA
- Tampilkan ringkasan daripada seluruhdiff: git diff --stat

Masukkan berikut dalam file hello.php:

```
<?php
echo 'www.w3big.com' ;
?>
git status -s
A README
AM hello.php
git diff
diff --git a/hello.php b/hello.php
index e69de29..69b5711 100644
--- a/hello.php
+++ b/hello.php
@@ -0,0 +1,3 @@
+<?php
+echo 'www.w3big.com' ;
+?>
```

Menampilkan status git pada untuk berubah setelah update atau menulis garis perubahan cache dengan garis dan

git diff menunjukkan secara spesifik apa perubahan tersebut. Selanjutnya melihat git berikutnya diff pelaksanaan –cached hasil:

```
$ git add hello.php
$ git status -s
A README
A hello.php
$ git diff –cached
diff –git a/README b/README
new file mode 100644
index 0000000..8f87495
— /dev/null
+++ b/README
@@ -0,0 +1 @@
+ # w3big Git
diff –git a/hello.php b/hello.php
new file mode 100644
index 0000000..69b5711
— /dev/null
+++ b/hello.php
@@ -0,0 +1,3 @@
+?php
+echo 'www.w3big.com';
+?;
```

Gunakan git menambahkan perintah ingin menulis isi dari buffer snapshot, dan mengeksekusi git commit akan menambahkan konten ke gudang penyangga. Git mengirimkan masing-masing nama dan alamat e-mail yang tercatat, sehingga langkah pertama perlu mengkonfigurasi nama pengguna dan alamat e-mail.

```
$ git config –global user.name 'w3big'
$ git config –global user.email test@w3big.com
```

Berikutnya menulis caching, dan menyerahkan semua perubahan hello.php tersebut. Dalam contoh pertama menggunakan opsi -m untuk memberikan baris perintah untuk mengirimkan komentar :

```
$ git add hello.php
```

```
$ git status -s
A README
A hello.php
$ $ git commit -m ''
[master (root-commit) d32cf1f]
2 files changed, 4 insertions(+)
create mode 100644 README
create mode 100644 hello.php
```

Sekarang telah mencatat snapshot. Jika di jalankan git status:

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

Output di atas menunjukkan bahwa setelah pengajuan terakhir, tidak membuat perubahan apapu. Jika tidak menetapkan opsi -m, Git mencoba untuk membuka editor untuk mengisi informasi yang disampaikan. Git jika tidak dapat menemukan informasi yang relevan dalam konfigurasi, default akan membuka vim. Layar akan terlihat seperti ini:

```
# Please enter the commit message for your changes. Lines starting
# with ' #' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD ;file;..." to unstage)
#
# modified:   hello.php
#
~
~
".git/COMMIT_EDITMSG" 9L, 257C
```

Jika berpikir git add disampaikan proses cache yang terlalu rumit, Git juga memungkinkan untuk menggunakan opsi -a untuk melewati langkah ini. format perintah adalah sebagai berikut:


```
git commit -a
```

Memodifikasi file `hello.php` sebagai berikut:

```

i?php
echo 'www.w3big.com';
echo 'www.w3big.com';
?i

```

Kemudian jalankan perintah berikut:

```

git commit -am 'hello.php '
[master 71ee2cb] hello.php
1 file changed, 1 insertion(+)

```

Git reset perintah HEAD untuk menghapus konten cache. Mari mengubah file berkas README, sebagai berikut:

```

# w3big Git
#

```

File `hello.php` diubah sebagai berikut:

```

i?php
echo 'www.w3big.com';
echo 'www.w3big.com';
echo 'www.w3big.com';
?i

```

Sekarang setelah dua file diubah disampaikan ke zona penyangga, sekarang ingin membatalkan salah satu dari cache, sebagai berikut:

```

$ git status -s
M README
M hello.php
$ git add .
$ git status -s
M README
M hello.pp
$ git reset HEAD -- hello.php
Unstaged changes after reset:
M      hello.php
$ git status -s

```

```
M README
```

```
M hello.php
```

Sekarang menjalankan git commit, perubahan hanya akan diserahkan berkas README, tapi hello.php tidak.

```
$ git commit -m ''
```

```
[master f50cfda]
```

```
1 file changed, 1 insertion(+)
```

```
$ git status -s
```

```
M hello.php
```

Melihat file perubahan hello.php dan untuk pengajuan. Maka dapat menggunakan perintah berikut untuk memodifikasi hello.php menyerahkan:

```
$ git commit -am 'hello.php'
```

```
[master 760f74d] hello.php
```

```
1 file changed, 1 insertion(+)
```

```
$ git status
```

```
On branch master
```

```
nothing to commit, working directory clean
```

Singkatnya, melakukan git reset HEAD untuk membatalkan sebelum git add untuk menambahkan, tetapi tidak ingin untuk memasukkan dalam cache snapshot di commit selanjutnya.

Entri rm git akan dihapus dari cache. ulang kepala git ini membatalkan entri cache yang berbeda. "Batal Cache", yang berarti bahwa pemulihan akan membuat perubahan ke cache. Secara default, git file rm akan dihapus dari file cache dan hard drive (direktori kerja). Jika ingin menyimpan file dalam direktori kerja, dapat menggunakan git rm --cached: Seperti kita menghapus hello.php file:

```
$ git rm hello.php
```

```
rm 'hello.php'
```

```
$ ls
```

```
README
```

Tidak menghapus file dari ruang kerja:

```
$ git rm --cached README
rm 'README'
$ ls
README
```

Git perintah `mv` untuk melakukan semua hal yang `git rm` perintah operasi `--cached`, mengubah nama file pada disk, dan kemudian jalankan `git add` untuk menambahkan file baru ke cache. `README` pertama kita hapus hanya menambahkan kembali:

```
$ git add README
```

Kemudian nama yang sama yaitu:

```
$ git mv README README.md
$ ls
README.md
```

13.2 Mendapatkan File untuk Pindah

Buat salinan repositori A sehingga bisa memindahkannya tanpa terlalu mengkhawatirkan kesalahan. Sebaiknya hapus tautan ke repositori asli agar tidak sengaja membuat perubahan jarak jauh (baris 3). Ini berjalan melalui file, menghapus apapun yang tidak ada dalam direktori 1. Hasilnya adalah isi direktori 1 dipindahkan ke basis repositori A. Mengimpor file-file ini ke dalam repositori B di dalam direktori, jadi memindahkan semua menjadi satu sekarang (baris 5/6). Komit perubahan dan menggabungkan file-file ini ke dalam repositori yang baru 13.1 :

```
1 git clone <git repository A url>
2 cd <git repository A directory>
3 git remote rm origin
4 git filter-branch --subdirectory-filter <directory 1> -- --all
5 mkdir <directory 1>
6 mv * <directory 1>
7 git add .
8 git commit
```

Gambar 13.1 PerpindahanFile

BAB 14

RENAME OPERATION

14.1 Rename Operation

Sampai sekarang, baik Tom dan Jerry menggunakan perintah manual untuk menyusun proyek mereka. Sekarang, Jerry memutuskan untuk membuat Makefile untuk proyek mereka dan juga memberi nama yang tepat untuk file "string.c".

```
[jerry@CentOS project] $ pwd  
/home/jerry/jerry_repo/project  
[jerry@CentOS project] $ ls
```

```
README src
```

```
[jerry@CentOS project] $ cd src/
```

```
[jerry@CentOS src] $ git add Makefile
```

```
[jerry@CentOS src] $ git mv string.c string_operations.c
```

```
[jerry@CentOS src] $ git status -s
```

A Makefile

```
R string.c → string_operations.c
```

Git menunjukkan R sebelum nama file untuk menunjukkan bahwa file telah diganti namanya.

Untuk komit operasi, Jerry menggunakan -bendera, yang membuat git komit secara otomatis mendeteksi file yang dimodifikasi :

```
[jerry@CentOS src] $ \ $ git commit -a -m
Makefile and renamed strings.c to string $ \
$operations.c '
```

```
[master 94f7b26] Added Makefile and renamed s
string $ \_ $operations.c 1 files changed,
insertions(+), 0 deletions(-)
```

```
create mode 100644 src/Makefile
rename src/ $ \{ $string.c => string $ \_
.c $ \} $ (100 $ \% $)
```

```
[jerry@CentOS src] $ \ $ $
```

Perintah di atas akan menghasilkan hasil sebagai berikut:

Counting objects: 6, done.

Compressing objects: 100 % (3/3), done.

Writing objects: 100 % (4/4), 396 bytes, done.

Total 4 (delta 0), reused 0 (delta 0)

To gituser@git.server.com:project.git

7d9ea97..94f7b26 master → master

Sekarang, pengembang lain dapat melihat modifikasi ini dengan memperbarui repositori lokal mereka.

Kegunaan utama dari sistem kontrol versi ialah sebagai alat untuk manajemen kode program. Terdapat dua kegunaan utama dari sistem ini, yaitu:

- Menggabungkan perubahan-perubahan kode dari versi lama (misal: untuk mengembalikan fitur yang telah dihapus)
- Menggabungkan perubahan dari orang lain (misal: menggabungkan fitur yang dikembangkan oleh anggota tim lain)

14.1.1 Instalasi Git

Git berjalan pada semua sistem operasi populer (Mac, Windows, Linux). Jika menggunakan Windows atau Mac, masuk ke situs utama git pada lalu lakukan download dan instalasi software tersebut. Pengguna Linux dapat melakukan instalasi melalui repositori distribusi yang dilakukan, melalui perintah sejenis:

```
yum install git
```

pada repositori berbasis RPM, atau perintah :

```
apt-get install git
```

Untuk repositori berbasis deb. Kembali lagi, perintah hanya diberikan untuk distribusi paling populer (Debian / Ubuntu dan RedHat / Fedora), karena keterbatasan ruang. Jika menggunakan distrusi lain (seperti Gentoo atau Arch, maka diasumsikan telah mengetahui cara instalasi git atau perangkat lunak lain pada umumnya).

Khusus untuk sistem operasi Windows, pastikan instalasi diambil dari , karena pada paket yang tersedia di website tersebut telah diikutkan juga OpenSSH, yang akan sangat berguna jika ingin berkolaborasi dengan programmer lain. Perintah git juga harus memberikan respon yang benar:

```
bert@LYNNSLENIA ~
```

```
$ git
```

```
usage: git [-version] [-exec-path[=;pathi]] [-html-path] [-man-path] [-info-path]
        [-p | -paginate | -no-pager] [-no-replace-objects] [-bare]
        [-git-dir=;pathi] [-work-tree=;pathi] [-namespace=;namei]
        [-c name=value] [-help]
        ;commandi [;argsi]
```

14.1.2 Inisiasi

Untuk dapat menggunakan sistem kontrol versi, terlebih dahulu kita harus mempersiapkan repositori. Sebuah repositori menyimpan seluruh versi dari kode program kita. Tidak usah takut, karena repositori tidak akan memakan banyak ruang *hard disk*, karena penyimpanan tidak dilakukan terhadap keseluruhan file. Repositori hanya akan menyimpan *perubahan* yang terjadi pada kode kita dari satu versi ke versi lainnya. Bahasa kerennya, repositori hanya menyimpan delta dari kode pada setiap versinya.

Pada (di saat kontrol versi yang populer adalah cvs dan programmer pada umumnya berjanggut putih), membangun repositori kode baru adalah hal yang sangat sulit dilakukan. Harus memiliki sebuah *server* khusus yang dapat diakses oleh seluruh anggota tim. Jika server tidak dapat diakses karena jaringan rusak atau internet putus, maka tidak dapat melakukan kontrol versi (dan harus kembali ke metode direktori, atau tidak bekerja).

Git merupakan sistem kontrol versi terdistribusi, yang berarti git dapat dijalankan tanpa perlu adanya repositori terpusat. Yang diperlukan untuk membuat repositori ialah mengetikkan perintah tertentu di direktori utama. Mulai membuat repositori baru.:

```
bert@LYNNSLENIA ~  
$ cd Desktop/projects/git-tutor/  
bert@LYNNSLENIA ~/Desktop/projects/git-tutor  
$ ls  
bert@LYNNSLENIA ~/Desktop/projects/git-tutor  
$
```

Menambahkan kode baru ke dalam direktori ini. Buat sebuah file baru yang bernama *cerita.txt* di dalam direktori tersebut:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor  
$ echo "ini adalah sebuah cerita" & cerita.txt  
bert@LYNNSLENIA ~/Desktop/projects/git-tutor  
$ ls  
cerita.txt
```

Kemudian masukkan perintah `git init` untuk melakukan inisialisasi repositori:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor
```

```
$ git init
```

```
Initialized empty Git repository in c:/Users/bert/Desktop/projects/git-tutor/.git/
```

Setelah melakukan inisialisasi, git secara otomatis akan membuat direktori `.git` pada repositori (lihat potongan kode di bawah). Direktori tersebut merupakan direktori yang digunakan oleh git untuk menyimpan basis data delta kode, dan berbagai metadata lainnya. Mengubah direktori tersebut dapat menyebabkan hilangnya seluruh *history* dari kode.

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ ls -a
```

```
git cerita.txt
```

14.1.3 Penambahan File ke Repository

Penyimpanan sejarah dapat dimulai dari saat pertama: kapan file tersebut dibuat dan ditambahkan ke dalam repositori. Untuk menambahkan file ke dalam repositori, gunakan perintah `git add`:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git add .
```

```
warning: LF will be replaced by CRLF in cerita.txt.
```

```
The file will have its original line endings in your working directory.
```

Secara sederhana, sintaks dari perintah `git add` adalah sebagai berikut:

```
git add [nama file atau pola]
```

Memasukkan nama file dalam perintah `git add` pada dasarnya akan memerintahkan git untuk menambahkan **semua** file baru dalam repositori. Jika hanya ingin menambahkan satu file (misalkan ada file yang belum yakin akan ditambahkan ke repositori), nama file spesifik dapat dimasukkan:


```
git add cerita.txt
```

Setelah menambahkan file ke dalam repositori, harus melakukan *commit*. Perintah *commit* memberitahukan kepada git untuk menyimpan sejarah dari file yang telah ditambahkan. Pada git, penambahan, perubahan, ataupun penghapusan sebuah file baru akan tercatat jika perintah *commit* telah dijalankan. Mari lakukan *commit* dengan menjalankan perintah git commit:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
$ git commit
```

Jika langkah di atas diikuti dengan benar, maka kembali ke git bash, dengan pesan berikut:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
$ git commit
[master (root-commit) 1d4cdc9] Inisialisasi repo. Penambahan cerita.txt.
warning: LF will be replaced by CRLF in cerita.txt.
The file will have its original line endings in your working directory.
1 file changed, 1 insertion(+)
create mode 100644 cerita.txt
```

14.1.4 Mengubah Isi File

Kegunaan utama kontrol versi (yang tercermin dari namanya) ialah melakukan manajemen perubahan secara otomatis untuk kita. dan kemudian jalankan perintah git commit lagi:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
$ git commit
# On branch master
# Changes not staged for commit:
#   (use "git add ;file_..." to update what will be committed)
#   (use "git checkout - ;file_..." to discard changes in working directory)
#
#       modified:   cerita.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Perhatikan bahwa git secara otomatis mengetahui file mana saja yang berubah, tetapi tidak melakukan pencatatan perubahan tersebut. Untuk memerintahkan git mencatat perubahan tersebut, gunakan perintah git commit -a:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git commit -a
```

```
[master 61c4707] Kapitalisasi dan melengkapi kalimat.
```

```
1 file changed, 1 insertion(+), 1 deletion(-)
```

Selain melakukan perubahan, tentunya terkadang kita ingin mengetahui perubahan-perubahan apa saja yang terjadi selama pengembangan. Untuk melihat daftar perubahan yang telah dilakukan, kita dapat menggunakan perintah git log:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git log
```

```
commit 61c47074ee583dbdd16fa9568019e80d864fb403
```

```
Author: Alex Xandra Albert Sim <bertzzie@gmail.com>
```

```
Date: Sun Dec 23 16:36:46 2012 +0700
```

```
commit 1d4cdc9350570230d352ef19aededf06769b0698
```

```
Author: Alex Xandra Albert Sim <bertzzie@gmail.com>
```

```
Date: Sun Dec 23 16:10:33 2012 +0700
```

```
  Inisialisasi repo. Penambahan cerita.txt.
```

Jalankan perintah git log sekali lagi, untuk melihat hasil pekerjaan kita sejauh ini:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git log
```

```
commit 28dabb1c54a086cce567ecb890b10339416bcbfa
```

```
Author: Alex Xandra Albert Sim <bertzzie@gmail.com>
```

```
Date: Sun Dec 23 16:49:21 2012 +0700
```

```
commit 61c47074ee583dbdd16fa9568019e80d864fb403
```

```
Author: Alex Xandra Albert Sim <bertzzie@gmail.com>
```

```
Date: Sun Dec 23 16:36:46 2012 +0700
```

```
commit 1d4cdc9350570230d352ef19aededf06769b0698
```

```
Author: Alex Xandra Albert Sim <bertzzie@gmail.com>
```

```
Date: Sun Dec 23 16:10:33 2012 +0700
```

Git memungkinkan kita untuk mengembalikan kode ke dalam keadaan sebelumnya, yaitu *commit* terakhir. Melakukan pengembalian kode ini dengan menggunakan perintah git checkout seperti berikut:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git checkout HEAD – cerita.txt
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ ls
```

```
cerita.txt
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ cat cerita.txt
```

Ini adalah sebuah cerita tentang seekor kera yang terkurung dan terpenjara dalam goa.

Kera ini bernama Sun Go Kong. Dari manakah Sun Go Kong berasal?

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

Parameter HEAD pada perintah yang kita jalankan merupakan parameter untuk memberitahukan git checkout bahwa kita ingin mengembalikan kode pada revisi terakhir (HEAD dalam istilah git). Karena hanya ingin mengembalikan file cerita.txt, maka kita harus memberitahukan git checkout, melalui parameter – cerita.txt. Perintah git checkout juga memiliki banyak kegunaan lainnya selain mengembalikan kode ke revisi tertentu.

Untuk melihat bagaimana fitur ini bekerja, mari lakukan perubahan pada repositori terlebih dahulu. Tambahkan sebuah file baru ke dalam repositori:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ ls
```

```
cerita.txt
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ echo "Seekor kera, terpuruk, terpenjara dalam goa. Di gunung suci sunyi  
tempat hukuman para dewa." & lagu-intro.txt
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ ls
```

```
cerita.txt lagu-intro.txt
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git add .
```

```
warning: LF will be replaced by CRLF in lagu-intro.txt.
```

```
The file will have its original line endings in your working directory.
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git commit
```

```
[master 03d0628] Penambahan lagu intro.
```

```
warning: LF will be replaced by CRLF in lagu-intro.txt.
```

```
The file will have its original line endings in your working directory.
```

```
1 file changed, 1 insertion(+)
```

```
create mode 100644 lagu-intro.txt
```

Kemudian kita akan melakukan edit terhadap `cerita.txt` dan mengganti nama `lagu-intro.txt` menjadi `lagu-intro-awal.txt`:

```
ert@LYNNSLENIA $ \sim $/Desktop/projects/g
(master)
```

```
$ \$ $ ls \par
cerita.txt~ lagu-intro.txt \par
```

```
bert@LYNNSLENIA $ \sim $/Desktop/projects/
(master)
```

```
$ \$ $ notepad cerita.txt \par
```

```
bert@LYNNSLENIA $ \sim $/Desktop/projects/
(master)
```

```
$ \$ $ mv lagu-intro.txt lagu-intro-awal.t
```

```
bert@LYNNSLENIA $ \sim $/Desktop/projects/
(master)
```

```
$ \$ $ ls
```

```
cerita.txt lagu-intro-awal.txt
```

Setelah melakukan perubahan tersebut, kita mengalami amnesia sesaat karena kucing kantor jatuh ke kepala kita (kucing yang menyebalkan!). Karena telah lupa akan perubahan yang dilakukan, kita dapat melihat apa saja yang berubah dengan menggunakan perintah `git status`:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git status
```

```
# On branch master
# Changes not staged for commit:
#   (use "git add/rm |file_..." to update what will be committed)
#   (use "git checkout -- |file_..." to discard changes in working directory)
#
#       modified:   cerita.txt
#       deleted:    lagu-intro.txt
#
# Untracked files:
#   (use "git add |file_..." to include in what will be committed)
#
#       lagu-intro-awal.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

”Changes not staged for commit ” menampilkan daftar file yang berubah, tetapi belum di-*commit*. File yang tercatat ini termasuk file yang diubah dan dihapus.

”Untracked files ” menampilkan file yang belum ditambahkan ke dalam repositori.

Jika ingin melihat apa saja yang diubah pada file `cerita.txt`, kita dapat menggunakan perintah `git diff`:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git diff cerita.txt
```

```
diff --git a/cerita.txt b/cerita.txt
```

```
index 846114d..dbcb596 100644
```

```
--- a/cerita.txt
```

```
+++ b/cerita.txt
```

```
@@ -1,3 +1,3 @@
```

Ini adalah sebuah cerita tentang seekor kera yang terkurung dan terpenjara dala

-Kera ini bernama Sun Go Kong. Dari manakah Sun Go Kong berasal?

+Kera ini bernama Sun Go Kong. Dari manakah Sun Go Kong berasal???

(END)

Format yang ditampilkan mungkin agak membingungkan, tetapi tidak usah takut, karena bagian yang perlu diperhatikan hanyalah pada bagian yang bertanda - dan +. Pada `git bash`, bahkan bagian ini diberi warna (merah untuk - dan hijau untuk +). Tanda +, tentunya berarti bagian yang

ditambahkan, dan tanda - berarti bagian yang dihapus. Dengan melihat perubahan pada baris yang bersangkutan, kita dapat mengetahui bahwa ? diubah menjadi ???! pada akhir baris.

Setelah mengetahui perubahan yang dilakukan, dan menganggap perubahan tersebut aman untuk di-*commit*, kita lalu dapat melakukan *commit* seperti biasa:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
$ git add lagu-intro-awal.txt
warning: LF will be replaced by CRLF in lagu-intro-awal.txt.
The file will have its original line endings in your working directory.
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
$ git commit
[master 306f422] Dramatisasi cerita dan perubahan nama file lagu.
warning: LF will be replaced by CRLF in lagu-intro-awal.txt.
The file will have its original line endings in your working directory.
1 file changed, 1 insertion(+)
create mode 100644 lagu-intro-awal.txt
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
$ git log
commit 306f42258f4bfee95d10396777391ae013bc6edd
Author: Alex Xandra Albert Sim <bertzzie@gmail.com>
Date: Sun Dec 23 18:22:30 2012 +0700
```

Dramatisasi cerita dan perubahan nama file lagu.

```
commit 03d06284462f7fc43b610d522678f4f22cdd9a40
Author: Alex Xandra Albert Sim <bertzzie@gmail.com>
Date: Sun Dec 23 18:08:10 2012 +0700
```

Penambahan lagu intro.

```
commit 28dabb1c54a086cce567ecb890b10339416bcbfa
Author: Alex Xandra Albert Sim <bertzzie@gmail.com>
Date: Sun Dec 23 16:49:21 2012 +0700
```

Penambahan misteri terbesar di dunia.

```
commit 61c47074ee583dbdd16fa9568019e80d864fb403
```

Author: Alex Xandra Albert Sim ;bertzzie@gmail.com;

Date: Sun Dec 23 16:36:46 2012 +0700

Kapitalisasi dan melengkapi kalimat.

commit 1d4cdc9350570230d352ef19aededf06769b0698

Author: Alex Xandra Albert Sim ;bertzzie@gmail.com;

Date: Sun Dec 23 16:10:33 2012 +0700

Inisialisasi repo. Penambahan cerita.txt.

14.1.5 Membaca File Lama, dan Menjalankan Mesin Waktu

Nomor revisi, seperti yang telah dijelaskan sebelumnya, berguna sebagai tanda untuk memisahkan antara satu *commit* dengan *commit* lainnya. Misalnya jika ingin melihat isi file cerita.txt pada saat awal pertama kali dibuat, kita dapat menggunakan perintah `git show`, yang sintaksnya adalah: `git show [nomor revisi]:[nama file]`

contoh penggunaan:

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git show 1d4cdc:cerita.txt
```

ini adalah sebuah cerita

Perhatikan bahwa nomor commit yang dimasukkan hanyalah enam karakter saja. Jika keenam karakter tersebut sama untuk beberapa nomor *commit*, kita baru perlu memasukkan karakter selanjutnya, sampai tidak terdapat konflik nama lagi.

Sesuai dengan nomor revisi dengan menggunakan `git checkout` yang telah dijelaskan sebelumnya. Contohnya :

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ ls
```

```
cerita.txt lagu-intro-awal.txt
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ cat cerita.txt
```

Ini adalah sebuah cerita tentang seekor kera yang terkurung dan terpenjara dalam goa.

Kera ini bernama Sun Go Kong. Dari manakah Sun Go Kong berasal???!

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git checkout 61c470 cerita.txt
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ cat cerita.txt
```

Ini adalah sebuah cerita tentang seekor kera yang terkurung dan terpenjara dalam goa.

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git checkout 1d4cdc cerita.txt
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ cat cerita.txt
```

ini adalah sebuah cerita

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ git checkout 03d0628 cerita.txt
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
```

```
$ cat cerita.txt
```

Ini adalah sebuah cerita tentang seekor kera yang terkurung dan terpenjara dalam goa.

Kera ini bernama Sun Go Kong. Dari manakah Sun Go Kong berasal?

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
$ git checkout HEAD cerita.txt
```

```
bert@LYNNSLENIA ~/Desktop/projects/git-tutor (master)
$ cat cerita.txt
```

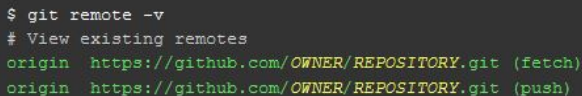
Ini adalah sebuah cerita tentang seekor kera yang terkurung dan terpenjara dalam goa.

Kera ini bernama Sun Go Kong. Dari manakah Sun Go Kong berasal????!

Perhatikan bahwa pada saat menggunakan perintah `git checkout`, menggunakan `cat` untuk melihat isi file. Hal ini dikarenakan `git checkout` benar-benar mengubah file yang ada pada repositori, berbeda dengan `git show` yang hanya menampilkan file tersebut pada revisi tertentu.

14.2 Penyelesaian Masalah

Mengalami kesalahan ini saat mencoba mengganti nama remote. Tidak bisa mengganti nama config section 'remote [nama lama]' ke 'remote [nama baru]' kesalahan ini berarti remote yang dicoba dengan nama remote lama yang diketik tidak ada. Memeriksa remote yang ada saat ini dengan perintah 14.1 Penamaan Remote `git remote -v`:



```
$ git remote -v
# View existing remotes
origin  https://github.com/OWNER/REPOSITORY.git (fetch)
origin  https://github.com/OWNER/REPOSITORY.git (push)
```

Gambar 14.1 Penamaan

Kesalahan ini berarti nama remote yang ingin digunakan sudah ada. Untuk mengatasi ini, gunakan nama jauh yang berbeda, atau ganti nama remote asli.

DAFTAR PUSTAKA

- [Kil76] J. S. Kilby, "Invention of the Integrated Circuit," *IEEE Trans. Electron Devices*, **ED-23**, 648 (1976).
- [Ham62] R. W. Hamming, *Numerical Methods for Scientists and Engineers*, Chapter N-1, McGraw-Hill, New York, 1962.
- [Hu86] J. Lee, K. Mayaram, and C. Hu, "A Theoretical Study of Gate/Drain Offset in LDD MOSFETs" *IEEE Electron Device Lett.*, **EDL-7**(3). 152 (1986).
- [Ber87] A. Berenbaum, B. W. Colbry, D.R. Ditzel, R. D Freeman, and K.J. O'Connor, "A Pipelined 32b Microprocessor with 13 kb of Cache Memory," in *Int. Solid State Circuit Conf.*, Dig. Tech. Pap., p. 34 (1987).

