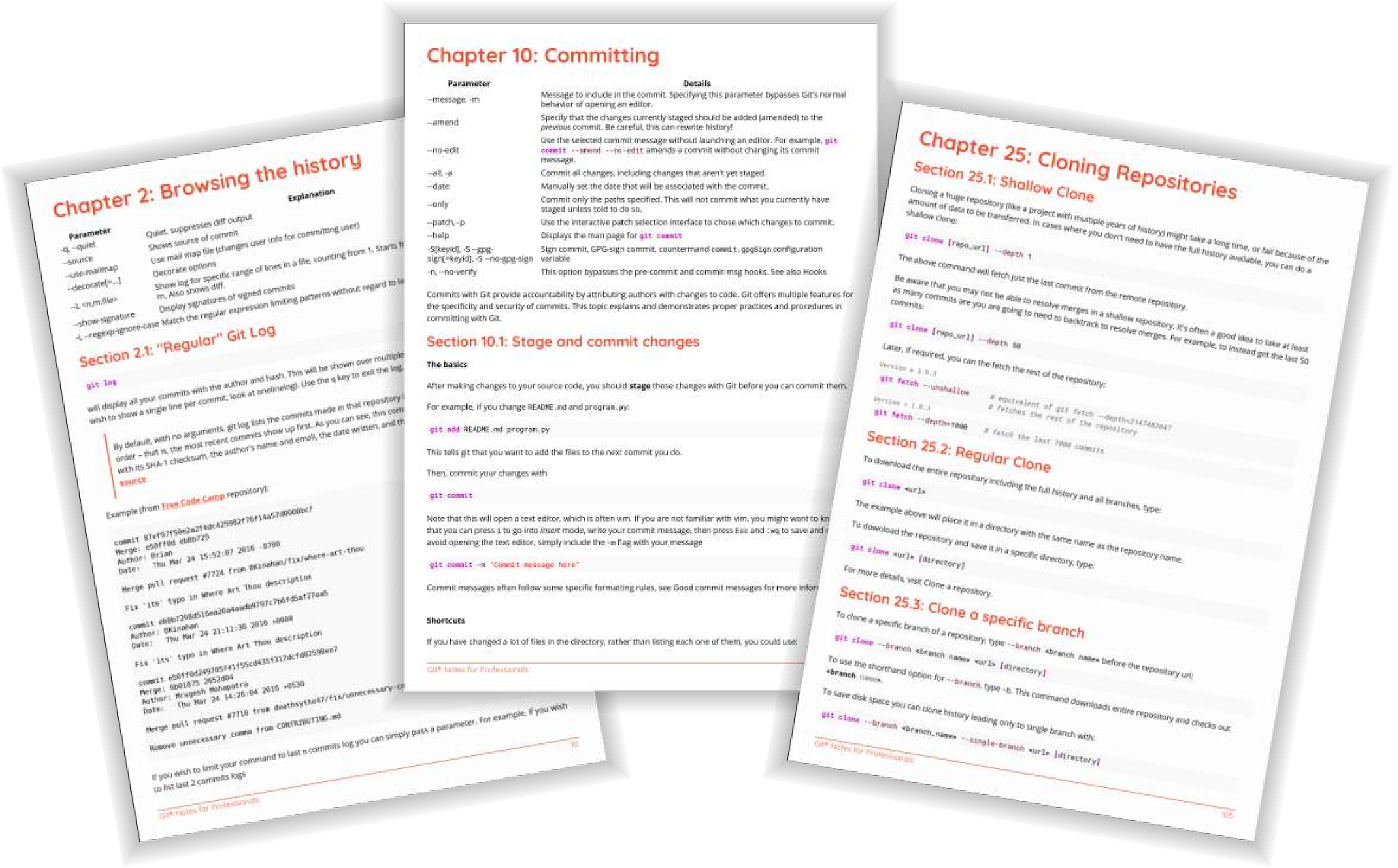
**Git®**

**Notes for Professionals**

100+ pages

of professional hints and tricks

**Goalßicker.com**



Free Programming Books

**Disclaimer** This is an unofficial free book created for educational purposes and is not affiliated with official Git® group(s) or company(s).

All trademarks and registered trademarks are the property of their respective owners

# Contents

**About**

................................................................................................................................................................................... 1

**Chapter 1: Getting started with Git**

...................................................................................................................... 2

Section 1.1: Create your first repository, then add and commit files 2

Section 1.2: Clone a repository 4

Section 1.3: Sharing code 4

Section 1.4: Setting your user name and email 5

Section 1.5: Setting up the upstream remote 6

Section 1.6: Learning about a command 6

Section 1.7: Set up SSH for Git 6

Section 1.8: Git Installation 7

**Chapter 2: Browsing the histor6**

......................................................................................................................... 10

Section 2.1: ”Regular” Git Log 10

Section 2.2: Prettier log 11

Section 2.3: Colorize Logs 11

Section 2.4: Oneline log Section 2.5: Log search

.............................................................................................................................................. 11

............................................................................................................................................... 12

Section 2.6: List all contributions grouped by author name 12

Section 2.7: Searching commit string in git log 13

Section 2.8: Log for a range of lines within a file 14

Section 2.9: Filter logs 14

Section 2.10: Log with changes inline 14

Section 2.11: Log showing commited files 15

Section 2.12: Show the contents of a single commit 15

Section 2.13: Git Log Between Two Branches 16

Section 2.14: One line showing commiter name and time since commit 16

**Chapter \: Working with Remotes** Section 3.1: Deleting a Remote Branch Section 3.2: Changing Git Remote URL

..................................................................................................................... 17

..................................................................................................................... 17

..................................................................................................................... 17

Section 3.3: List Existing Remotes 17

Section 3.4: Removing Local Copies of Deleted Remote Branches 17

Section 3.5: Updating from Upstream Repository 18

Section 3.6: ls−remote 18

Section 3.7: Adding a New Remote Repository 18

Section 3.8: Set Upstream on a New Branch 18

Section 3.9: Getting Started 19

Section 3.10: Renaming a Remote 19

Section 3.11: Show information about a Specific Remote 20

Section 3.12: Set the URL for a Specific Remote Section 3.13: Get the URL for a Specific Remote

...................................................................................................... 20

...................................................................................................... 20

Section 3.14: Changing a Remote Repository 20

**Chapter 4: Staging**

.................................................................................................................................................... 21

Section 4.1: Staging All Changes to Files 21

Section 4.2: Unstage a file that contains changes 21

Section 4.3: Add changes by hunk 21

Section 4.4: Interactive add 22

Section 4.5: Show Staged Changes 22

Section 4.6: Staging A Single File 23

Section 4.7: Stage deleted files 23

**Chapter 5: Ignoring Files and Folders**

.............................................................................................................. 24

Section 5.1: Ignoring files and directories with a .gitignore file 24

Section 5.2: Checking if a file is ignored 26

Section 5.3: Exceptions in a .gitignore file 27

Section 5.4: A global .gitignore file 27

Section 5.5: Ignore files that have already been committed to a Git repository 27

Section 5.6: Ignore files locally without committing ignore rules 28

Section 5.7: Ignoring subsequent changes to a file (without removing it) 29

Section 5.8: Ignoring a file in any directory Section 5.9: Prefilled .gitignore Templates

.............................................................................................................. 29

............................................................................................................... 29

Section 5.10: Ignoring files in subfolders (Multiple gitignore files) 30

Section 5.11: Create an Empty Folder 31

Section 5.12: Finding files ignored by .gitignore Section 5.13: Ignoring only part of a file [stub]

....................................................................................................... 31

........................................................................................................ 32

Section 5.14: Ignoring changes in tracked files. [stub] 33

Section 5.15: Clear already committed files, but included in .gitignore 34

Chapter 6: Git Diff 35

Section 6.1: Show differences in working branch 35

Section 6.2: Show changes between two commits 35

Section 6.3: Show differences for staged files 35

Section 6.4: Comparing branches 36

Section 6.5: Show both staged and unstaged changes 36

Section 6.6: Show differences for a specific file or directory 36

Section 6.7: Viewing a word−diff for long lines 37

Section 6.8: Show differences between current version and last version 37

Section 6.9: Produce a patch−compatible diff 37

Section 6.10: difference between two commit or branch 38

[Section 6.11: Using meld to see all modifications in the working directory 38](#_TOC_250000)

Section 6.12: Diff UTF−16 encoded text and binary plist files 38

**Chapter 7: Undoing**

.................................................................................................................................................... 40

Section 7.1: Return to a previous commit 40

Section 7.2: Undoing changes 40

Section 7.3: Using reflog 41

Section 7.4: Undoing merges 41

Section 7.5: Revert some existing commits 43

Section 7.6: Undo / Redo a series of commits 43

**Chapter 8: Merging**

.................................................................................................................................................... 45

Section 8.1: Automatic Merging 45

Section 8.2: Finding all branches with no merged changes 45

Section 8.3: Aborting a merge 45

Section 8.4: Merge with a commit 45

Section 8.5: heep changes from only one side of a merge 45

Section 8.6: Merge one branch into another 46

**Chapter 9: Submodules**

........................................................................................................................................... 47

Section 9.1: Cloning a Git repository having submodules 47

Section 9.2: Updating a Submodule 47

Section 9.3: Adding a submodule 47

Section 9.4: Setting a submodule to follow a branch 48

Section 9.5: Moving a submodule 48

Section 9.6: Removing a submodule 49

**Chapter 10: Committing**

........................................................................................................................................... 50

Section 10.1: Stage and commit changes 50

Section 10.2: Good commit messages 51

Section 10.3: Amending a commit 52

Section 10.4: Committing without opening an editor 53

Section 10.5: Committing changes directly 53

Section 10.6: Selecting which lines should be staged for committing 53

Section 10.7: Creating an empty commit 54

Section 10.8: Committing on behalf of someone else 54

Section 10.9: GPG signing commits 55

Section 10.10: Commiting changes in specific files 55

Section 10.11: Committing at a specific date 55

Section 10.12: Amending the time of a commit 56

Section 10.13: Amending the author of a commit 56

**Chapter 11: Aliases**

...................................................................................................................................................... 57

Section 11.1: Simple aliases 57

Section 11.2: List / search existing aliases 57

Section 11.3: Advanced Aliases 57

Section 11.4: Temporarily ignore tracked files 58

Section 11.5: Show pretty log with branch graph 58

Section 11.6: See which files are being ignored by your .gitignore configuration 59

Section 11.7: Updating code while keeping a linear history 60

Section 11.8: Unstage staged files 60

**Chapter 12: Rebasing**

................................................................................................................................................ 61

Section 12.1: Local Branch Rebasing 61

Section 12.2: Rebase: ours and theirs, local and remote 61

Section 12.3: Interactive Rebase 63

Section 12.4: Rebase down to the initial commit 64

Section 12.5: Configuring autostash 64

Section 12.6: Testing all commits during rebase 65

Section 12.7: Rebasing before a code review Section 12.8: Aborting an Interactive Rebase

........................................................................................................... 65

........................................................................................................... 67

Section 12.9: Setup git−pull for automatically perform a rebase instead of a merge 68

Section 12.10: Pushing after a rebase 68

**Chapter 1\: Configuration**

...................................................................................................................................... 69

Section 13.1: Setting which editor to use 69

Section 13.2: Auto correct typos 69

Section 13.3: List and edit the current configuration 70

Section 13.4: Username and email address 70

Section 13.5: Multiple usernames and email address 70

Section 13.6: Multiple git configurations 71

Section 13.7: Configuring line endings 72

Section 13.8: configuration for one command only 72

Section 13.9: Setup a proxy 72

**Chapter 14: Branching** 74

Section 14.1: Creating and checking out new branches 74

Section 14.2: Listing branches 75

Section 14.3: Delete a remote branch 75

Section 14.4: Quick switch to the previous branch 76

Section 14.5: Check out a new branch tracking a remote branch 76

Section 14.6: Delete a branch locally 76

Section 14.7: Create an orphan branch (i.e. branch with no parent commit) 77

Section 14.8: Rename a branch 77

Section 14.9: Searching in branches 77

Section 14.10: Push branch to remote 77

Section 14.11: Move current branch HEAD to an arbitrary commit 78

**Chapter 15: Rev-List**

.................................................................................................................................................. 79

Section 15.1: List Commits in master but not in origin/master 79

**Chapter 16: Squashing**

.............................................................................................................................................. 80

Section 16.1: Squash Recent Commits Without Rebasing 80

Section 16.2: Squashing Commit During Merge 80

Section 16.3: Squashing Commits During a Rebase 80

Section 16.4: Autosquashing and fixups 81

Section 16.5: Autosquash: Committing code you want to squash during a rebase 82

**Chapter 17: Cherr6 Picking**

..................................................................................................................................... 83

Section 17.1: Copying a commit from one branch to another 83

Section 17.2: Copying a range of commits from one branch to another 83

Section 17.3: Checking if a cherry−pick is required 84

Section 17.4: Find commits yet to be applied to upstream 84

**Chapter 18: Recovering** 85

Section 18.1: Recovering from a reset Section 18.2: Recover from git stash

....................................................................................................................... 85

......................................................................................................................... 85

Section 18.3: Recovering from a lost commit 86

Section 18.4: Restore a deleted file after a commit 86

Section 18.5: Restore file to a previous version 86

Section 18.6: Recover a deleted branch 87

**Chapter 19: Git Clean**

................................................................................................................................................. 88

Section 19.1: Clean Interactively 88

Section 19.2: Forcefully remove untracked files 88

Section 19.3: Clean Ignored Files 88

Section 19.4: Clean All Untracked Directories 88

**Chapter 20: Using a .gitattributes file**

............................................................................................................. 90

Section 20.1: Automatic Line Ending Normalization 90

Section 20.2: Identify Binary Files 90

Section 20.3: Prefilled .gitattribute Templates 90

Section 20.4: Disable Line Ending Normalization 90

**Chapter 21: .mailmap file: Associating contributor and email aliases**

............................................ 91

Section 21.1: Merge contributers by aliases to show commit count in shortlog 91

**Chapter 22: Anal6zing t6pes of workflows**

.................................................................................................... 92

Section 22.1: Centralized Workflow 92

Section 22.2: Gitflow Workflow 93

Section 22.3: Feature Branch Workflow 95

Section 22.4: GitHub Flow 95

Section 22.5: Forking Workflow 96

**Chapter 2\: Pulling**

..................................................................................................................................................... 97

Section 23.1: Pulling changes to a local repository 97

Section 23.2: Updating with local changes 98

Section 23.3: Pull, overwrite local 98

Section 23.4: Pull code from remote 98

Section 23.5: heeping linear history when pulling 98

Section 23.6: Pull, ”permission denied” 99

**Chapter 24: Hooks**

................................................................................................................................................... 100

Section 24.1: Pre−push 100

Section 24.2: Verify Maven build (or other build system) before committing 101

Section 24.3: Automatically forward certain pushes to other repositories 101

Section 24.4: Commit−msg Section 24.5: Local hooks

....................................................................................................................................... 102

......................................................................................................................................... 102

Section 24.6: Post−checkout 102

Section 24.7: Post−commit Section 24.8: Post−receive Section 24.9: Pre−commit

........................................................................................................................................ 103

........................................................................................................................................ 103

......................................................................................................................................... 103

Section 24.10: Prepare−commit−msg 103

Section 24.11: Pre−rebase Section 24.12: Pre−receive

.......................................................................................................................................... 103

......................................................................................................................................... 104

Section 24.13: Update 104

**Chapter 25: Cloning Repositories**

..................................................................................................................... 105

Section 25.1: Shallow Clone Section 25.2: Regular Clone

...................................................................................................................................... 105

..................................................................................................................................... 105

Section 25.3: Clone a specific branch 105

Section 25.4: Clone recursively 106

Section 25.5: Clone using a proxy 106

**Chapter 26: Stashing** 107

Section 26.1: What is Stashing? 107

Section 26.2: Create stash 108

Section 26.3: Apply and remove stash 109

Section 26.4: Apply stash without removing it 109

Section 26.5: Show stash Section 26.6: Partial stash

.......................................................................................................................................... 109

........................................................................................................................................ 109

Section 26.7: List saved stashes 110

Section 26.8: Move your work in progress to another branch 110

Section 26.9: Remove stash 110

Section 26.10: Apply part of a stash with checkout 110

Section 26.11: Recovering earlier changes from stash 110

Section 26.12: Interactive Stashing 111

Section 26.13: Recover a dropped stash 111

**Chapter 27: Subtrees**

.............................................................................................................................................. 113

Section 27.1: Create, Pull, and Backport Subtree 113

**Chapter 28: Renaming**

........................................................................................................................................... 114

Section 28.1: Rename Folders 114

Section 28.2: rename a local and the remote branch 114

Section 28.3: Renaming a local branch 114

**Chapter 29: Pushing** 115

Section 29.1: Push a specific object to a remote branch 115

Section 29.2: Push 116

Section 29.3: Force Pushing 117

Section 29.4: Push tags 117

Section 29.5: Changing the default push behavior 117

**Chapter \0: Internals**

.............................................................................................................................................. 119

Section 30.1: Repo 119

Section 30.2: Objects 119

Section 30.3: HEAD ref 119

Section 30.4: Refs 119

Section 30.5: Commit Object 120

Section 30.6: Tree Object 121

Section 30.7: Blob Object 121

Section 30.8: Creating new Commits 122

Section 30.9: Moving HEAD 122

Section 30.10: Moving refs around 122

Section 30.11: Creating new Refs 122

**Chapter \1: git-tfs**

..................................................................................................................................................... 123

Section 31.1: git−tfs clone 123

Section 31.2: git−tfs clone from bare git repository 123

Section 31.3: git−tfs install via Chocolatey 123

Section 31.4: git−tfs Check In 123

Section 31.5: git−tfs push 123

**Chapter \2: Empt6 directories in Git**

Section 32.1: Git doesn't track directories

............................................................................................................... 124

............................................................................................................... 124

**Chapter \\: git-svn** 125

Section 33.1: Cloning the SVN repository 125

Section 33.2: Pushing local changes to SVN 125

Section 33.3: Working locally 125

Section 33.4: Getting the latest changes from SVN 126

Section 33.5: Handling empty folders 126

**Chapter \4: Archive**

................................................................................................................................................. 127

Section 34.1: Create an archive of git repository 127

Section 34.2: Create an archive of git repository with directory prefix 127

Section 34.3: Create archive of git repository based on specific branch, revision, tag or directory 128

**Chapter \5: Rewriting histor6 with filter-branch**

..................................................................................... 129

Section 35.1: Changing the author of commits 129

Section 35.2: Setting git committer equal to commit author 129

**Chapter \6: Migrating to Git**

............................................................................................................................... 130

Section 36.1: SubGit 130

Section 36.2: Migrate from SVN to Git using Atlassian conversion utility 130

Section 36.3: Migrating Mercurial to Git 131

Section 36.4: Migrate from Team Foundation Version Control (TFVC) to Git 131

Section 36.5: Migrate from SVN to Git using svn2git 132

**Chapter \7: Show**

...................................................................................................................................................... 133

Section 37.1: Overview 133

**Chapter \8: Resolving merge conflicts**

......................................................................................................... 134

Section 38.1: Manual Resolution 134

**Chapter \9: Bundles**

................................................................................................................................................ 135

Section 39.1: Creating a git bundle on the local machine and using it on another 135

**Chapter 40: Displa6 commit histor6 graphicall6 with Gitk** 136

Section 40.1: Display commit history for one file 136

Section 40.2: Display all commits between two commits 136

Section 40.3: Display commits since version tag 136

**Chapter 41: Bisecting/Finding fault6 commits**

......................................................................................... 137

Section 41.1: Binary search (git bisect) 137

Section 41.2: Semi−automatically find a faulty commit 137

**Chapter 42: Blaming**

............................................................................................................................................... 139

Section 42.1: Only show certain lines 139

Section 42.2: To find out who changed a file 139

Section 42.3: Show the commit that last modified a line 140

Section 42.4: Ignore whitespace−only changes 140

**Chapter 4\: Git revisions s6ntax**

....................................................................................................................... 141

Section 43.1: Specifying revision by object name 141

Section 43.2: Symbolic ref names: branches, tags, remote−tracking branches 141

Section 43.3: The default revision: HEAD 141

Section 43.4: Reflog references: crefname>@{cn>} 141

Section 43.5: Reflog references: crefname>@{cdate>} 142

Section 43.6: Tracked / upstream branch: cbranchname>@{upstream} 142

Section 43.7: Commit ancestry chain: crev>^, crev>~cn>, etc 142

Section 43.8: Dereferencing branches and tags: crev>^0, crev>^{ctype>} 143

Section 43.9: Youngest matching commit: crev>^{/ctext>}, :/ctext> 143

**Chapter 44: Worktrees**

......................................................................................................................................... 145

Section 44.1: Using a worktree 145

Section 44.2: Moving a worktree 145

**Chapter 45: Git Remote**

........................................................................................................................................ 147

Section 45.1: Display Remote Repositories 147

Section 45.2: Change remote url of your Git repository 147

Section 45.3: Remove a Remote Repository 148

Section 45.4: Add a Remote Repository 148

Section 45.5: Show more information about remote repository 148

Section 45.6: Rename a Remote Repository 149

**Chapter 46: Git Large File Storage (LFS)**

..................................................................................................... 150

Section 46.1: Declare certain file types to store externally 150

Section 46.2: Set LFS config for all clones 150

Section 46.3: Install LFS

**Chapter 47: Git Patch**

............................................................................................................................................ 150

............................................................................................................................................. 151

Section 47.1: Creating a patch Section 47.2: Applying patches

.................................................................................................................................. 151

................................................................................................................................ 152

**Chapter 48: Git statistics**

...................................................................................................................................... 153

Section 48.1: Lines of code per developer 153

Section 48.2: Listing each branch and its last revision's date 153

Section 48.3: Commits per developer 153

Section 48.4: Commits per date 154

Section 48.5: Total number of commits in a branch 154

Section 48.6: List all commits in pretty format 154

Section 48.7: Find All Local Git Repositories on Computer 154

Section 48.8: Show the total number of commits per author 154

**Chapter 49: git send-email** 155

Section 49.1: Use git send−email with Gmail 155

Section 49.2: Composing 155

Section 49.3: Sending patches by mail 155

**Chapter 50: Git GUI Clients**

.................................................................................................................................. 157

Section 50.1: gitk and git−gui Section 50.2: GitHub Desktop

.................................................................................................................................... 157

.................................................................................................................................. 158

Section 50.3: Git hraken Section 50.4: SourceTree

........................................................................................................................................... 159

.......................................................................................................................................... 159

Section 50.5: Git Extensions 159

Section 50.6: SmartGit 159

**Chapter 51: Reflog - Restoring commits not shown in git log**

........................................................... 160

Section 51.1: Recovering from a bad rebase 160

**Chapter 52: TortoiseGit**

......................................................................................................................................... 161

Section 52.1: Squash commits 161

Section 52.2: Assume unchanged 162

Section 52.3: Ignoring Files and Folders 164

Section 52.4: Branching 165

**Chapter 5\: External merge and difftools**

................................................................................................... 167

Section 53.1: Setting up hDiff3 as merge tool 167

Section 53.2: Setting up hDiff3 as diff tool 167

Section 53.3: Setting up an IntelliJ IDE as merge tool (Windows) 167

Section 53.4: Setting up an IntelliJ IDE as diff tool (Windows) 167

Section 53.5: Setting up Beyond Compare 168

**Chapter 54: Update Ob¡ect Name in Reference**

...................................................................................... 169

Section 54.1: Update Object Name in Reference 169

**Chapter 55: Git Branch Name on Bash Ubuntu**

........................................................................................ 170

Section 55.1: Branch Name in terminal

**Chapter 56: Git Client-Side Hooks**

................................................................................................................... 170

.................................................................................................................... 171

Section 56.1: Git pre−push hook Section 56.2: Installing a Hook

................................................................................................................................ 171

................................................................................................................................. 172

**Chapter 57: Git rerere**

............................................................................................................................................ 173

Section 57.1: Enabling rerere 173

**Chapter 58: Change git repositor6 name**

.................................................................................................... 174

Section 58.1: Change local setting 174

**Chapter 59: Git Tagging**

........................................................................................................................................ 175

Section 59.1: Listing all available tags 175

Section 59.2: Create and push tag(s) in GIT 175

**Chapter 60: Tid6ing up 6our local and remote repositor6**

................................................................. 177

Section 60.1: Delete local branches that have been deleted on the remote 177

**Chapter 61: diff-tree**

................................................................................................................................................ 178

Section 61.1: See the files changed in a specific commit 178

Section 61.2: Usage 178

Section 61.3: Common diff options 178

**Credits**

............................................................................................................................................................................ 179

**You ma6 also like** 186

# About

Please feel free to share this PDF with anyone for free, latest version of this book can be downloaded from: https://goalkicker.com/GitBook

This *Git® Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Git® group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to [web@petercv.com](mailto:web@petercv.com)

**Chapter 1: Getting started with Git**

**Version Release Date**

|  |  |
| --- | --- |
| 2.13 | 2017-05-10 |
| 2.12 | 2017-02-24 |
| 2.11.1 | 2017-02-02 |
| 2.11 | 2016-11-29 |
| 2.10.2 | 2016-10-28 |
| 2.10 | 2016-09-02 |
| 2.9 | 2016-06-13 |
| 2.8 | 2016-03-28 |
| 2.7 | 2015-10-04 |
| 2.6 | 2015-09-28 |
| 2.5 | 2015-07-27 |
| 2.4 | 2015-04-30 |
| 2.3 | 2015-02-05 |
| 2.2 | 2014-11-26 |
| 2.1 | 2014-08-16 |
| 2.0 | 2014-05-28 |
| 1.9 | 2014-02-14 |
| 1.8.3 | 2013-05-24 |
| 1.8 | 2012-10-21 |
| 1.7.10 | 2012-04-06 |
| 1.7 | 2010-02-13 |
| 1.6.5 | 2009-10-10 |
| 1.6.3 | 2009-05-07 |
| 1.6 | 2008-08-17 |
| 1.5.3 | 2007-09-02 |
| 1.5 | 2007-02-14 |
| 1.4 | 2006-06-10 |
| 1.3 | 2006-04-18 |
| 1.2 | 2006-02-12 |
| 1.1 | 2006-01-08 |
| 1.0 | 2005-12-21 |
| 0.99 | 2005-07-11 |

**Section 1.1: Create 6our first repositor6, then add and commit files**

At the command line, first verify that you have Git installed: On all operating systems:

**git** --version

On UNIX-like operating systems:

**which git**

If nothing is returned, or the command is not recognized, you may have to install Git on your system by downloading and running the installer. See the Git homepage for exceptionally clear and easy installation instructions.

After installing Git, configure your username and email address. Do this *before* making a commit.

Once Git is installed, navigate to the directory you want to place under version control and create an empty Git repository:

**git init**

This creates a hidden folder, .git, which contains the plumbing needed for Git to work. Next, check what files Git will add to your new repository; this step is worth special care:

**git status**

Review the resulting list of files; you can tell Git which of the files to place into version control (avoid adding files with confidential information such as passwords, or files that just clutter the repo):

**git add <**file**/**directory name ***#1> <fiﾕe/direcłory name #2> < ... >***

If all files in the list should be shared with everyone who has access to the repository, a single command will add everything in your current directory and its subdirectories:

**git add** .

This will "stage" all files to be added to version control, preparing them to be committed in your first commit.

For files that you want never under version control, create and populate a file named .gitignore before running the add command.

Commit all the files that have been added, along with a commit message:

**git commit** -m "Initial commit"

This creates a new commit with the given message. A commit is like a save or snapshot of your entire project. You can now push, or upload, it to a remote repository, and later you can jump back to it if necessary.

If you omit the -m parameter, your default editor will open and you can edit and save the commit message there.

Adding a remote

To add a new remote, use the **git remote** add command on the terminal, in the directory your repository is stored at.

The **git remote** add command takes two arguments:

1. A remote name, for example, origin
2. A remote URL, for example, https:**//<**your-git-service-address**>/**user**/**repo.git

**git remote** add origin https:**//<**your-git-service-address**>/**owner**/**repository.git

NOTE: Before adding the remote you have to create the required repository in your git service, You'll be able to push/pull commits after adding your remote.

## Section 1.2: Clone a repositor6

The **git clone** command is used to copy an existing Git repository from a server to the local machine. For example, to clone a GitHub project:

**cd <**path where you would like the clone to create a directory**> git clone** https:**//**github.com**/**username**/**projectname.git

To clone a BitBucket project:

**cd <**path where you would like the clone to create a directory**>**

**git clone** [https:**//**yourusername**@**bitbucket.org**/**username**/**projectname.git](https://yourusername@bitbucket.org/username/projectname.git)

This creates a directory called projectname on the local machine, containing all the files in the remote Git repository. This includes source files for the project, as well as a .git sub-directory which contains the entire history and configuration for the project.

To specify a different name of the directory, e.g. MyFolder:

**git clone** https:**//**github.com**/**username**/**projectname.git MyFolder

Or to clone in the current directory:

**git clone** https:**//**github.com**/**username**/**projectname.git .

Note:

1.

2.

When cloning to a specified directory, the directory must be empty or non-existent. You can also use the **ssh** version of the command:

**git clone git@**github.com:username**/**projectname.git

The https version and the **ssh** version are equivalent. However, some hosting services such as GitHub recommend that you use https rather than **ssh**.

## Section 1.\: Sharing code

To share your code you create a repository on a remote server to which you will copy your local repository.

To minimize the use of space on the remote server you create a bare repository: one which has only the .git objects and doesn't create a working copy in the filesystem. As a bonus you set this remote as an upstream server to easily share updates with other programmers.

On the remote server:

**git init** --bare **/**path**/**to**/**repo.git

On the local machine:

**git remote** add origin ssh:**//**username**@**server:**/**path**/**to**/**repo.git

(Note that ssh: is just one possible way of accessing the remote repository.) Now copy your local repository to the remote:

**git push** --set-upstream origin master

Adding --set-upstream (or -u) created an upstream (tracking) reference which is used by argument-less Git commands, e.g. **git pull**.

## Section 1.4: Setting 6our user name and email

You need to **set who** you are **\***before**\*** creating any commit. That will allow commits to have the right author name and email associated to them.

**It has nothing to do with authentication when pushing to a remote repository** (e.g. when pushing to a remote repository using your GitHub, BitBucket, or GitLab account)

To declare that identity for *all* repositories, use **git config** --global

This will store the setting in your user's .gitconfig file: e.g. $H0ME**/**.gitconfig or for Windows,

**%**U5ERPR0FILE**%**\.gitconfig.

**git config** --global user.name "Your Name"

**git config** --global user.email [mail**@**example.com](mailto:mail@example.com)

To declare an identity for a single repository, use **git config** inside a repo.

This will store the setting inside the individual repository, in the file $GIT\_DIR**/**config. e.g.

**/**path**/**to**/**your**/**repo**/**.git**/**config.

**cd /**path**/**to**/**my**/**repo

**git config** user.name "Your Login At Work"

**git config** user.email [mail\_at\_work**@**example.com](mailto:mail_at_work@example.com)

Settings stored in a repository's config file will take precedence over the global config when you use that repository.

Tips: if you have different identities (one for open-source project, one at work, one for private repos, ...), and you don't want to forget to set the right one for each different repos you are working on:

### Remove a global identity

**git config** --global --remove-section user.name

**git config** --global --remove-section user.email

Version ≥ 2.8

To force git to look for your identity only within a repository's settings, not in the global config:

**git config** --global user.useConfig0nly **true**

That way, if you forget to set your user.name and user.email for a given repository and try to make a commit, you will see:

no name was given and auto-detection is disabled no email was given and auto-detection is disabled

## Section 1.5: Setting up the upstream remote

If you have cloned a fork (e.g. an open source project on Github) you may not have push access to the upstream repository, so you need both your fork but be able to fetch the upstream repository.

First check the remote names:

$ **git remote** -v

origin https:**//**github.com**/**myusername**/**repo.git **(**fetch**)** origin https:**//**github.com**/**myusername**/**repo.git **(**push**)** upstream ***# łhis ﾕine may or may noł be here***

If upstream is there already (it is on *some* Git versions) you need to set the URL (currently it's empty):

$ **git remote** set-url upstream https:**//**github.com**/**projectusername**/**repo.git

If the upstream is **not** there, or if you also want to add a friend/colleague's fork (currently they do not exist):

$ **git remote** add upstream https:**//**github.com**/**projectusername**/**repo.git

$ **git remote** add dave https:**//**github.com**/**dave**/**repo.git

## Section 1.6: Learning about a command

To get more information about any git command – i.e. details about what the command does, available options and other documentation – use the --help option or the **help** command.

For example, to get all available information about the **git diff** command, use:

**git diff** --help

**git help diff**

Similarly, to get all available information about the status command, use:

**git status** --help

**git help** status

If you only want a quick help showing you the meaning of the most used command line flags, use -h:

**git checkout** -h

## Section 1.7: Set up SSH for Git

If you are using **Windows** open Git Bash. If you are using **Mac** or **Linux** open your Terminal. Before you generate an SSH key, you can check to see if you have any existing SSH keys.

List the contents of your ~**/**.ssh directory:

$ **ls** -al ~**/**.ssh

***# Lisłs aﾕﾕ łhe fiﾕes in your ~/.ssh direcłory***

Check the directory listing to see if you already have a public SSH key. By default the filenames of the public keys are one of the following:

id\_dsa.pub id\_ecdsa.pub id\_ed25519.pub id\_rsa.pub

If you see an existing public and private key pair listed that you would like to use on your Bitbucket, GitHub (or similar) account you can copy the contents of the id\_**\***.pub file.

If not, you can create a new public and private key pair with the following command:

$ **ssh–keygen**

Press the Enter or Return key to accept the default location. Enter and re-enter a passphrase when prompted, or leave it empty.

Ensure your SSH key is added to the ssh-agent. Start the ssh-agent in the background if it's not already running:

$ **eval** "$(ssh-agent -s)"

Add you SSH key to the ssh-agent. Notice that you'll need te replace id\_rsa in the command with the name of your

### private key file:

$ **ssh–add** ~**/**.ssh**/**id\_rsa

If you want to change the upstream of an existing repository from HTTPS to SSH you can run the following command:

$ **git remote** set-url origin ssh:**//git@**bitbucket.server.com:7999**/**projects**/**your\_project.git

In order to clone a new repository over SSH you can run the following command:

$ **git clone** ssh:**//git@**bitbucket.server.com:7999**/**projects**/**your\_project.git

## Section 1.8: Git Installation

Let’s get into using some Git. First things first—you have to install it. You can get it a number of ways; the two major ones are to install it from source or to install an existing package for your platform.

### Installing from Source

If you can, it’s generally useful to install Git from source, because you’ll get the most recent version. Each version of Git tends to include useful UI enhancements, so getting the latest version is often the best route if you feel comfortable compiling software from source. It is also the case that many Linux distributions contain very old packages; so unless you’re on a very up-to-date distro or are using backports, installing from source may be the best bet.

To install Git, you need to have the following libraries that Git depends on: curl, zlib, openssl, expat, and libiconv. For example, if you’re on a system that has yum (such as Fedora) or apt-get (such as a Debian based system), you can use one of these commands to install all of the dependencies:

$ **yum install** curl-devel expat-devel gettext-devel \

openssl-devel zlib-devel

$ **apt–get install** libcurl4-gnutls-dev libexpat1-dev **gettext** \ libz-dev libssl-dev

When you have all the necessary dependencies, you can go ahead and grab the latest snapshot from the Git web site:

<http://git-scm.com/download>Then, compile and install:

$ **tar** -zxf git-1.7.2.2.tar.gz

$ **cd** git-1.7.2.2

$ **make** prefix=**/**usr**/local** all

$ **sudo make** prefix=**/**usr**/local install**

After this is done, you can also get Git via Git itself for updates:

$ **git clone** git:**//**git.kernel.org**/**pub**/**scm**/**git**/**git.git

### Installing on Linux

If you want to install Git on Linux via a binary installer, you can generally do so through the basic package- management tool that comes with your distribution. If you’re on Fedora, you can use yum:

$ **yum install git**

Or if you’re on a Debian-based distribution like Ubuntu, try apt-get:

$ **apt–get install git**

### Installing on Mac

There are three easy ways to install Git on a Mac. The easiest is to use the graphical Git installer, which you can download from the SourceForge page.

<http://sourceforge.net/projects/git-osx-installer/>

Figure 1-7. Git OS X installer. The other major way is to install Git via MacPorts (http://www.macports.org). If you have MacPorts installed, install Git via

$ **sudo** port **install git** +**svn** +doc +bash\_completion +gitweb

You don’t have to add all the extras, but you’ll probably want to include +svn in case you ever have to use Git with Subversion repositories (see Chapter 8).

Homebrew (<http://brew.sh/)>is another alternative to install Git. If you have Homebrew installed, install Git via

$ brew **install git**

### Installing on Windows

Installing Git on Windows is very easy. The msysGit project has one of the easier installation procedures. Simply download the installer exe file from the GitHub page, and run it:

[http:**//**msysgit.github.io](http://msysgit.github.io/)

After it’s installed, you have both a command-line version (including an SSH client that will come in handy later) and the standard GUI.

*Note on Windows usage:* you should use Git with the provided msysGit shell (Unix style), it allows to use the complex lines of command given in this book. If you need, for some reason, to use the native Windows shell / command line console, you have to use double quotes instead of single quotes (for parameters with spaces in them) and you must quote the parameters ending with the circumflex accent (^) if they are last on the line, as it is a continuation symbol in Windows.

# Chapter 2: Browsing the histor6

### Parameter Explanation

-q, --quiet Quiet, suppresses diff output

--source Shows source of commit

--use-mailmap Use mail map file (changes user info for committing user)

--decorate[=...] Decorate options

Show log for specific range of lines in a file, counting from 1. Starts from line n, goes to line

--L <n,m:file>

m. Also shows diff.

--show-signature Display signatures of signed commits

-i, --regexp-ignore-case Match the regular expression limiting patterns without regard to letter case

**Section 2.1: "Regular" Git Log**

**git log**

will display all your commits with the author and hash. This will be shown over multiple lines per commit. (If you wish to show a single line per commit, look at onelineing). Use the q key to exit the log.

By default, with no arguments, git log lists the commits made in that repository in reverse chronological order – that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author’s name and email, the date written, and the commit message. - **source**

Example (from **Free Code Camp** repository):

commit 87ef97f59e2a2f4dc425982f76f14a57d8988bcf Merge: e58ff8d eb8b729

Author: Brian

Date: Thu Mar 24 15:52:87 2816 -8788

Merge putt request #7724 from BKinahan/fix/where-art-thou Fix 'its' typo in Where Art Thou description

commit eb8b7298d516ea28a4aadb9797c7b6fd5af27ea5 Author: BKinahan

Date: Thu Mar 24 21:11:36 2816 +8888

Fix 'its' typo in Where Art Thou description

commit e58ff8d249785f41f55cd435f317dcfd82598ee7 Merge: 6b81875 2652d84

Author: Mrugesh Mohapatra

Date: Thu Mar 24 14:26:84 2816 +8538

Merge putt request #7718 from deathsythe47/fix/unnecessary-comma Remove unnecessary comma from CONTRIBUTING.md

If you wish to limit your command to last n commits log you can simply pass a parameter. For example, if you wish to list last 2 commits logs

**git log** -2

## Section 2.2: Prettier log

To see the log in a prettier graph-like structure use:

**git log** --decorate --oneline --graph

sample output :

* eθc1cea **(**HEAD -**>** maint, tag: v2.9.3, origin**/**maint**)** Git 2.9.3
* 9bбθ1ea Merge branch 'jk/difftool-in-subdir' into maint

**|**\

**| \*** 32b8c58 difftool: use Git::**\*** functions instead of passing around state

**| \*** 98f917e difftool: avoid $GIT\_DIR and $GIT\_W0RK\_TREE

**| \*** 9ec2бe7 difftool: fix argument handling **in** subdirs

* **|** f4fdб27 Merge branch 'jk/reset-ident-time-per-commit' into maint

...

Since it's a pretty big command, you can assign an alias:

**git config** --global alias.lol "log --decorate --oneline --graph"

To use the alias version:

***# hisłory of currenł branch :***

**git** lol

***# combined hisłory of acłive branch (HEAD), deveﾕop and origin/masłer branches :***

**git** lol HEAD develop origin**/**master

***# combined hisłory of everyłhing in your repo :***

**git** lol --all

## Section 2.\: Colorize Logs

**git log** --graph --pretty=format:'%C(red)%h%Creset -%C(yellow)%d%Creset %s %C(green)(%cr)

%C(yellow)<%an>%Creset'

The format option allows you to specify your own log output format:

### Parameter Details

**%**C**(**color\_name**)** option colors the output that comes after it

%h or %H abbreviates commit hash (use %H for complete hash)

**%**Creset resets color to default terminal color

%d ref names

%s subject [commit message]

**%**cr committer date, relative to current date

**%**an author name

## Section 2.4: Oneline log

**git log** --oneline

will show all of your commits with only the first part of the hash and the commit message. Each commit will be in a single line, as the oneline flag suggests.

The oneline option prints each commit on a single line, which is useful if you’re looking at a lot of commits. - **source**

Example (from **Free Code Camp** repository, with the same section of code from the other example):

87ef97f Merge putt request #7724 from BKinahan/fix/where-art-thou eb8b729 Fix 'its' typo in Where Art Thou description

e58ff8d Merge putt request #7718 from deathsythe47/fix/unnecessary-comma 2652d84 Remove unnecessary comma from CONTRIBUTING.md

6b81875 Merge putt request #7667 from zerkms/patch-1 766f888 Fixed assignment operator terminotogy

d1e2468 Merge putt request #7698 from BKinahan/fix/unsubscribe-crash bed9de2 Merge putt request #7657 from Rafase282/fix/

If you wish to limit you command to last n commits log you can simply pass a parameter. For example, if you wish to list last 2 commits logs

**git log** -2 --oneline

## Section 2.5: Log search

git log -5"#define 5AMPLE5"

Searches for **addition** or **removal** of specific string or the string **matching** provided REGEXP. In this case we're looking for addition/removal of the string #define 5AMPLE5. For example:

+#define 5AMPLE5 1θθθθθ

or

-#define 5AMPLE5 1θθθθθ

git log -G"#define 5AMPLE5"

Searches for **changes** in **lines containing** specific string or the string **matching** provided REGEXP. For example:

-#define 5AMPLE5 1θθθθθ

+#define 5AMPLE5 1θθθθθθθθ

## Section 2.6: List all contributions grouped b6 author name

**git shortlog** summarizes **git log** and groups by author

If no parameters are given, a list of all commits made per committer will be shown in chronological order.

$ **git shortlog**

Committer 1 **(<**number\_of\_commits**>)**:

Commit Message 1

Commit Message 2

...

Committer 2 **(<**number\_of\_commits**>)**:

Commit Message 1

Commit Message 2

...

To simply see the number of commits and suppress the commit description, pass in the summary option:

-s

--summary

$ **git shortlog** -s

**<**number\_of\_commits**>** Committer 1

**<**number\_of\_commits**>** Committer 2

To sort the output by number of commits instead of alphabetically by committer name, pass in the numbered option:

-n

--numbered

To add the email of a committer, add the email option:

-e

--email

A custom format option can also be provided if you want to display information other than the commit subject:

--format

This can be any string accepted by the --format option of **git log**. See **Colorizing Logs** above for more information on this.

## Section 2.7: Searching commit string in git log

Searching git log using some string in log:

**git log [**options**]** --grep "search\_string"

Example:

**git log** --all --grep "removed file"

Will search for removed **file** string in **all logs** in **all branches**.

Starting from git 2.4+, the search can be inverted using the --invert-grep option. Example:

**git log** --grep="add file" --invert-grep

Will show all commits that do not contain add **file**.

## Section 2.8: Log for a range of lines within a file

$ **git log** -L 1,2θ:index.html

commit бa57fde739deбб293231fб2θ4cbd8b2feca3a8б9 Author: John Doe **<**[john**@**doe.com](mailto:john@doe.com)**>**

Date: Tue Mar 22 1б:33:42 2θ1б -θ5θθ

commit message

**diff** --git a**/**index.html b**/**index.html

--- a**/**index.html

+++ b**/**index.html

**@@** -1,17 +1,2θ **@@**

**<!**D0CTYPE HTML**>**

**<**html**>**

* **<head>**
* **<**meta charset="utf-8"**>**

+

+**<head>**

+ **<**meta charset="utf-8"**>**

**<**meta http-equiv="X-UA-Compatible" content="IE=edge"**>**

**<**meta name="viewport" content="width=device-width, initial-scale=1"**>**

**Section 2.9: Filter logs**

**git log** --after '3 days ago'

Specific dates work too:

**git log** --after 2θ1б-θ5-θ1

As with other commands and flags that accept a date parameter, the allowed date format is as supported by GNU date (highly flexible).

An alias to --after is --since.

Flags exist for the converse too: --before and --until. You can also filter logs by author. e.g.

**git log** --author=author

## Section 2.10: Log with changes inline

To see the log with changes inline, use the -p or --patch options.

**git log** --patch

Example (from Trello Scientist repository)

ommit 8ea1452aca481a837d95θ4f1b2c77adθ133б7d25 Author: Raymond Chou **<**[info**@**raychou.io](mailto:info@raychou.io)**>**

Date: Wed Mar 2 1θ:35:25 2θ1б -θ8θθ

fix readme error **link**

**diff** --git a**/**README.md b**/**README.md index 112θaθθ..9befθce 1θθб44

--- a**/**README.md

+++ b**/**README.md

**@@** -134,7 +134,7 **@@** the control **function** threw, but **\***after**\*** testing the other functions and readying

the logging. The criteria **for** matching errors is based on the constructor and message.

-You can **find** this full example at **[**examples**/**errors.js**](**examples**/**error.js**)**.

+You can **find** this full example at **[**examples**/**errors.js**](**examples**/**errors.js**)**.

***## Asynchronous behaviors***

commit d3178a2271бcc35bбa2bddб79a7ec24bc8cб3ffa

:

## Section 2.11: Log showing commited files

**git log** --stat

Example:

commit 4ded994d7fc5θ1451faбe2333б1887a23б5b91d1 Author: Manassés 5ouza **<**[manasses.inatel**@**gmail.com](mailto:manasses.inatel@gmail.com)**>** Date: Mon Jun б 21:32:3θ 2θ1б -θ3θθ

MercadoLibre java-sdk dependency mltracking-poc**/**.gitignore **|** 1 +

mltracking-poc**/**pom.xml **|** 14 ++++++++++++--

2 files changed, 13 insertions**(**+**)**, 2 deletions**(**-**)**

commit 5θбfff5б19θf75bcθ5124877θfbθbcd97бe3f9a5 Author: Manassés 5ouza **<**[manasses.inatel**@**gmail.com](mailto:manasses.inatel@gmail.com)**>**

Date: 5at Jun 4 12:35:1б 2θ1б -θ3θθ

**[**manasses**]** generated by 5pringBoot initializr

.gitignore **|** 42

++++++++++++

mltracking-poc**/**mvnw **|** 233

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

mltracking-poc**/**mvnw.cmd **|** 145

+++++++++++++++++++++++++++++++++++++++

mltracking-poc**/**pom.xml **|** 74

++++++++++++++++++++

mltracking-poc**/**src**/**main**/**java**/**br**/**com**/**mls**/**mltracking**/**MltrackingPocApplication.java **|** 12 ++++ mltracking-poc**/**src**/**main**/**resources**/**application.properties **|** θ mltracking-poc**/**src**/**test**/**java**/**br**/**com**/**mls**/**mltracking**/**MltrackingPocApplicationTests.java **|** 18 +++++

7 files changed, 524 insertions**(**+**)**

## Section 2.12: Show the contents of a single commit

Using **git show** we can view a single commit

**git show** 48c83b3

**git show** 48c83b3б9θdfc7bθeб22fd22θf8f37c2бa77c934

Example

commit 48c83b3б9θdfc7bθeб22fd22θf8f37c2бa77c934 Author: Matt Clark **<**[mrclark32493**@**gmail.com](mailto:mrclark32493@gmail.com)**>** Date: Wed May 4 18:2б:4θ 2θ1б -θ4θθ

The commit message will be shown here.

**diff** --git a**/**src**/**main**/**java**/**org**/**jdm**/**api**/**jenkins**/**Build5tatus.java b**/**src**/**main**/**java**/**org**/**jdm**/**api**/**jenkins**/**Build5tatus.java

index θb57e4a..fa8eбa5 1θθ755

--- a**/**src**/**main**/**java**/**org**/**jdm**/**api**/**jenkins**/**Build5tatus.java

+++ b**/**src**/**main**/**java**/**org**/**jdm**/**api**/**jenkins**/**Build5tatus.java

**@@** -5θ,7 +5θ,7 **@@** public enum Build5tatus **{**

-

+

colorMap.put**(**Build5tatus.UN5TABLE, Color.decode**(** "#FFFF55" **))**; colorMap.put**(**Build5tatus.5UCCE55, Color.decode**(** "#55FF55" **))**; colorMap.put**(**Build5tatus.5UCCE55, Color.decode**(** "#33CC33" **))**;

colorMap.put**(**Build5tatus.BUILDING, Color.decode**(** "#5555FF" **))**;

## Section 2.1\: Git Log Between Two Branches

**git log** master..foo will show the commits that are on foo and not on master. Helpful for seeing what commits you've added since branching!

## Section 2.14: One line showing commiter name and time since commit

**tree** = log --oneline --decorate --source --pretty=format:'"%Cblue %h %Cgreen %ar %Cblue %an

%C(yellow) %d %Creset %s"' --all --graph

example

**\***

4θ554ac 3 months ago Alexander Zolotov

Merge pull request ***#9€ from***

***gmandnepr/exłernaﾕ\_pﾕugins***

**|**\

**| \*** e5θ9fб1 3 months ago Ievgen Degtiarenko Documenting new property

**| \*** 4бd4cbб 3 months ago Ievgen Degtiarenko Running idea with external plugins

**| \*** б253da4 3 months ago Ievgen Degtiarenko Resolve external plugin classes

**| \*** 9fdb4e7 3 months ago Ievgen Degtiarenko Keep original artifact name **as** this may be

important **for** intellij

**| \*** 22e82e4 3 months ago

**|/**

Ievgen Degtiarenko

Declaring external plugin **in** intellij section

**\*** bc3d2cb 3 months ago Alexander Zolotov Ignore DTD **in** plugin.xml

# Chapter \: Working with Remotes

## Section \.1: Deleting a Remote Branch

To delete a remote branch in Git:

**git push [**remote-name**]** --delete **[**branch-name**]**

or

**git push [**remote-name**]** :**[**branch-name**]**

## Section \.2: Changing Git Remote URL

Check existing remote

**git remote** -v

***# origin hłłps://giłhub.com/username/repo.gił (fełch)***

***# origin hłłps://giłhub.com/usernam/repo.gił (push)***

Changing repository URL

**git remote** set-url origin https:**//**github.com**/**username**/**repo2.git

***# Change łhe 'origin' remołe's URL***

Verify new remote URL

**git remote** -v

***# origin hłłps://giłhub.com/username/repo2.gił (fełch)***

***# origin hłłps://giłhub.com/username/repo2.gił (push)***

## Section \.\: List Existing Remotes

List all the existing remotes associated with this repository:

**git remote**

List all the existing remotes associated with this repository in detail including the fetch and push URLs:

**git remote** --verbose

or simply

**git remote** -v

## Section \.4: Removing Local Copies of Deleted Remote Branches

If a remote branch has been deleted, your local repository has to be told to prune the reference to it. To prune deleted branches from a specific remote:

**git fetch [**remote-name**]** --prune

To prune deleted branches from *all* remotes:

**git fetch** --all --prune

## Section \.5: Updating from Upstream Repositor6

Assuming you set the upstream (as in the "setting an upstream repository")

**git fetch** remote-name

**git merge** remote-name**/**branch-name

The pull command combines a fetch and a merge.

**git pull**

The pull with --rebase flag command combines a fetch and a rebase instead of merge.

**git pull** --rebase remote-name branch-name

## Section \.6: ls-remote

**git ls–remote** is one unique command allowing you to query a remote repo *without having to clone/fetch it first*. It will list refs/heads and refs/tags of said remote repo.

You will see sometimes refs**/**tags**/**vθ.1.б *and* refs**/**tags**/**vθ.1.б^**{}**: the ^**{}** to list the dereferenced annotated tag (ie the commit that tag is pointing to)

Since git 2.8 (March 2016), you can avoid that double entry for a tag, and list directly those dereferenced tags with:

**git ls–remote** --ref

It can also help resolve the actual url used by a remote repo when you have "url.**<**base**>**.instead0f" config setting. If **git remote** --get-url **<**aremotename**>** returns https://server.com/user/repo, and you have set **git config** url.ssh:**//git@**server.com:.instead0f https:**//**server.com**/**:

**git ls–remote** --get-url **<**aremotename**>**

ssh:**//git@**server.com:user**/**repo

## Section \.7: Adding a New Remote Repositor6

**git remote** add upstream git-repository-url

Adds remote git repository represented by git-repository-url as new remote named upstream to the git repository

## Section \.8: Set Upstream on a New Branch

You can create a new branch and switch to it using

**git checkout** -b AP-57

After you use git checkout to create a new branch, you will need to set that upstream origin to push to using

**git push** --set-upstream origin AP-57

After that, you can use git push while you are on that branch.

## Section \.9: Getting Started

### Syntax for pushing to a remote branch

**git push <**remote\_name**> <**branch\_name**>**

### Example

**git push** origin master

## Section \.10: Renaming a Remote

To rename remote, use command **git remote** rename

The **git remote** rename command takes two arguments: An existing remote name, for example : **origin**

A new name for the remote, for example : **destination**

Get existing remote name

**git remote**

***# origin***

Check existing remote with URL

**git remote** -v

***# origin hłłps://giłhub.com/username/repo.gił (fełch)***

***# origin hłłps://giłhub.com/usernam/repo.gił (push)***

Rename remote

**git remote** rename origin destination

***# Change remołe name from 'origin' ło 'desłinałion'***

Verify new name

**git remote** -v

***# desłinałion hłłps://giłhub.com/username/repo.gił (fełch)***

***# desłinałion hłłps://giłhub.com/usernam/repo.gił (push)***

### === Posible Errors ===

1. Could not rename config section 'remote.[old name]' to 'remote.[new name]'

This error means that the remote you tried the old remote name (**origin**) doesn't exist.

1. Remote [new name] already exists.

Error message is self explanatory.

## Section \.11: Show information about a Specific Remote

Output some information about a known remote: origin

**git remote** show origin

Print just the remote's URL:

**git config** --get remote.origin.url

With 2.7+, it is also possible to do, which is arguably better than the above one that uses the config command.

**git remote** get-url origin

## Section \.12: Set the URL for a Specific Remote

You can change the url of an existing remote by the command

**git remote** set-url remote-name url

## Section \.1\: Get the URL for a Specific Remote

You can obtain the url for an existing remote by using the command

**git remote** get-url **<**name**>**

By default, this will be

**git remote** get-url origin

## Section \.14: Changing a Remote Repositor6

To change the URL of the repository you want your remote to point to, you can use the set-url option, like so:

**git remote** set-url **<**remote\_name**> <**remote\_repository\_url**>**

Example:

**git remote** set-url heroku https:**//**git.heroku.com**/**fictional-remote-repository.git

# Chapter 4: Staging

## Section 4.1: Staging All Changes to Files

**git add** -A

Version ≥ 2.8

**git add** .

In version 2.x, **git add** . will stage all changes to files in the current directory and all its subdirectories. However, in

1.x it will only stage new and modified files, not deleted files.

Use **git add** -A, or its equivalent command **git add** --all, to stage all changes to files in any version of git.

## Section 4.2: Unstage a file that contains changes

**git reset <**filePath**>**

**Section 4.\: Add changes b6 hunk**

You can see what "hunks" of work would be staged for commit using the patch flag:

**git add** -p

or

**git add** --patch

This opens an interactive prompt that allows you to look at the diffs and let you decide whether you want to include them or not.

5tage this hunk **[**y,n,q,a,d,**/**,s,e,?**]**?

stage this hunk for the next commit

do not stage this hunk for the next commit

|  |
| --- |
| y |
| n |
| q |
| a |
| d |
| g |
| / |
| j |
| J |
| k |
| K |
| s |
| e |
| ? |

quit; do not stage this hunk or any of the remaining hunks stage this hunk and all later hunks in the file

do not stage this hunk or any of the later hunks in the file select a hunk to go to

search for a hunk matching the given regex

leave this hunk undecided, see next undecided hunk leave this hunk undecided, see next hunk

leave this hunk undecided, see previous undecided hunk leave this hunk undecided, see previous hunk

split the current hunk into smaller hunks manually edit the current hunk

print hunk help

*This makes it easy to catch changes which you do not want to commit.*

You can also open this via **git add** --interactive and selecting p.

## Section 4.4: Interactive add

**git add** -i (or --interactive) will give you an interactive interface where you can edit the index, to prepare what you want to have in the next commit. You can add and remove changes to whole files, add untracked files and remove files from being tracked, but also select subsection of changes to put in the index, by selecting chunks of changes to be added, splitting those chunks, or even editing the diff. Many graphical commit tools for Git (like e.g. **git gui**) include such feature; this might be easier to use than the command line version.

It is very useful (1) if you have entangled changes in the working directory that you want to put in separate commits, and not all in one single commit (2) if you are in the middle of an interactive rebase and want to split too large commit.

$ **git add** -i

staged

1: unchanged

2: +1**/**-θ

unstaged path

+4**/**-4 index.js

nothing package.json

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **\*\*\*** Commands | **\*\*\*** |  | | |
| 1: status |  | 2: update | 3: revert | 4: add untracked |
| 5: **patch** |  | б: **diff** | 7: quit | 8: **help** |
| What now**>** |  |  |  |  |

The top half of this output shows the current state of the index broken up into staged and unstaged columns:

1. index.js has had 4 lines added and 4 lines removed. It is currently not staged, as the current status reports "unchanged." When this file becomes staged, the +4**/**-4 bit will be transferred to the staged column and the unstaged column will read "nothing."
2. package.json has had one line added and has been staged. There are no further changes since it has been staged as indicated by the "nothing" line under the unstaged column.

The bottom half shows what you can do. Either enter a number (1-8) or a letter (s, u, r, a, p, d, q, h). status shows output identical to the top part of the output above.

update allows you to make further changes to the staged commits with additional syntax.

revert will revert the staged commit information back to HEAD.

add untracked allows you to add filepaths previously untracked by version control.

**patch** allows for one path to be selected out of an output similar to status for further analysis.

**diff** displays what will be committed.

quit exits the command.

**help** presents further help on using this command.

## Section 4.5: Show Staged Changes

To display the hunks that are staged for commit:

**git diff** --cached

## Section 4.6: Staging A Single File

To stage a file for committing, run

**git add <**filename**>**

## Section 4.7: Stage deleted files

**git rm** filename

To delete the file from git without removing it from disk, use the --cached flag

**git rm** --cached filename

# Chapter 5: Ignoring Files and Folders

This topic illustrates how to avoid adding unwanted files (or file changes) in a Git repo. There are several ways (global or local .gitignore, .git**/**exclude, **git update–index** --assume-unchanged, and **git update–index** -- skip-tree), but keep in mind Git is managing *content*, which means: ignoring actually ignores a folder *content* (i.e. files). An empty folder would be ignored by default, since it cannot be added anyway.

## Section 5.1: Ignoring files and directories with a .gitignore file

You can make Git ignore certain files and directories — that is, exclude them from being tracked by Git — by creating one or more .gitignore files in your repository.

In software projects, .gitignore typically contains a listing of files and/or directories that are generated during the build process or at runtime. Entries in the .gitignore file may include names or paths pointing to:

1. temporary resources e.g. caches, log files, compiled code, etc.
2. local configuration files that should not be shared with other developers
3. files containing secret information, such as login passwords, keys and credentials

When created in the top level directory, the rules will apply recursively to all files and sub-directories throughout the entire repository. When created in a sub-directory, the rules will apply to that specific directory and its sub- directories.

When a file or directory is ignored, it will not be:

1. tracked by Git
2. reported by commands such as **git status** or **git diff**
3. staged with commands such as **git add** -A

In the unusual case that you need to ignore tracked files, special care should be taken. See: Ignore files that have already been committed to a Git repository.

### Examples

Here are some generic examples of rules in a .gitignore file, based on glob file patterns:

***# Lines słarłing wiłh `#` are commenłs.***

***# Ignore fiﾕes caﾕﾕed 'fiﾕe.exł'***

file.ext

***# Commenłs can'ł be on łhe same ﾕine as ruﾕes!***

***# The foﾕﾕowing ﾕine ignores fiﾕes caﾕﾕed 'fiﾕe.exł # noł a commenł'***

file.ext ***# noł a commenł***

***# Ignoring fiﾕes wiłh fuﾕﾕ pałh.***

***# This małches fiﾕes in łhe rooł direcłory and subdirecłories łoo. # i.e. ołherfiﾕe.exł wiﾕﾕ be ignored anywhere on łhe łree.***

dir**/**otherdir**/**file.ext otherfile.ext

***# Ignoring direcłories***

***# Bołh łhe direcłory iłseﾕf and iłs conłenłs wiﾕﾕ be ignored.***

bin**/** gen**/**

***# Gﾕob pałłern can aﾕso be used here ło ignore pałhs wiłh cerłain characłers. # For exampﾕe, łhe beﾕow ruﾕe wiﾕﾕ małch bołh buiﾕd/ and Buiﾕd/***

**[**bB**]**uild**/**

***# Wiłhouł łhe łraiﾕing sﾕash, łhe ruﾕe wiﾕﾕ małch a fiﾕe and/or***

***# a direcłory, so łhe foﾕﾕowing wouﾕd ignore bołh a fiﾕe named `gen`***

***# and a direcłory named `gen`, as weﾕﾕ as any conłenłs of łhał direcłory***

bin gen

***# Ignoring fiﾕes by exłension***

***# Aﾕﾕ fiﾕes wiłh łhese exłensions wiﾕﾕ be ignored in # łhis direcłory and aﾕﾕ iłs sub-direcłories.***

**\***.apk

**\***.class

***# Ił's possibﾕe ło combine bołh forms ło ignore fiﾕes wiłh cerłain # exłensions in cerłain direcłories. The foﾕﾕowing ruﾕes wouﾕd be # redundanł wiłh generic ruﾕes defined above.***

java**/\***.apk gen**/\***.class

***# To ignore fiﾕes onﾕy ał łhe łop ﾕeveﾕ direcłory, buł noł in iłs # subdirecłories, prefix łhe ruﾕe wiłh a `/`***

**/\***.apk

**/\***.class

***# To ignore any direcłories named DirecłoryA # in any depłh use \*\* before DirecłoryA***

***# Do noł forgeł łhe ﾕasł /,***

***# Ołherwise ił wiﾕﾕ ignore aﾕﾕ fiﾕes named DirecłoryA, rałher łhan direcłories***

**\*\*/**DirectoryA**/**

***# This wouﾕd ignore***

***# DirecłoryA/***

***# DirecłoryB/DirecłoryA/***

***# DirecłoryC/DirecłoryB/DirecłoryA/***

***# Ił wouﾕd noł ignore a fiﾕe named DirecłoryA, ał any ﾕeveﾕ***

***# To ignore any direcłory named DirecłoryB wiłhin a # direcłory named DirecłoryA wiłh any number of***

***# direcłories in bełween, use \*\* bełween łhe direcłories***

DirectoryA**/\*\*/**DirectoryB**/**

***# This wouﾕd ignore***

***# DirecłoryA/DirecłoryB/***

***# DirecłoryA/DirecłoryQ/DirecłoryB/***

***# DirecłoryA/DirecłoryQ/DirecłoryW/DirecłoryB/***

***# To ignore a seł of fiﾕes, wiﾕdcards can be used, as can be seen above.***

***# A soﾕe '\*' wiﾕﾕ ignore everyłhing in your foﾕder, incﾕuding your .giłignore fiﾕe. # To excﾕude specific fiﾕes when using wiﾕdcards, negałe łhem.***

***# So łhey are excﾕuded from łhe ignore ﾕisł:***

**!**.gitignore

***# Use łhe backsﾕash as escape characłer ło ignore fiﾕes wiłh a hash (#) # (supporłed since 1.6.2.1)***

\***#\*#***

Most .gitignore files are standard across various languages, so to get started, here is set of sample .gitignore files listed by language from which to clone or copy/modify into your project. Alternatively, for a fresh project you may consider auto-generating a starter file using an online tool.

### Other forms of .gitignore

.gitignore files are intended to be committed as part of the repository. If you want to ignore certain files without committing the ignore rules, here are some options:

Edit the .git**/**info**/**exclude file (using the same syntax as .gitignore). The rules will be global in the scope of the repository;

Set up a global gitignore file that will apply ignore rules to all your local repositories:

Furthermore, you can ignore local changes to tracked files without changing the global git configuration with:

**git update–index** --skip-worktree **[<file>**...**]**: for minor local modifications

**git update–index** --assume-unchanged **[<file>**...**]**: for production ready, non-changing files upstream

See more details on differences between the latter flags and the **git update–index** documentation for further options.

### Cleaning up ignored files

You can use **git clean** -X to cleanup ignored files:

**git clean** -Xn ***#dispﾕay a ﾕisł of ignored fiﾕes***

**git clean** -Xf ***#remove łhe previousﾕy dispﾕayed fiﾕes***

Note: -X (caps) cleans up *only* ignored files. Use -x (no caps) to also remove untracked files. See the **git clean** documentation for more details.

See the Git manual for more details.

## Section 5.2: Checking if a file is ignored

The **git** check-ignore command reports on files ignored by Git.

You can pass filenames on the command line, and **git** check-ignore will list the filenames that are ignored. For example:

$ **cat** .gitignore

**\***.o

$ **git** check-ignore example.o Readme.md example.o

Here, only \*.o files are defined in .gitignore, so Readme.md is not listed in the output of **git** check-ignore.

If you want to see line of which .gitignore is responsible for ignoring a file, add -v to the git check-ignore command:

$ **git** check-ignore -v example.o Readme.md

.gitignore:1:**\***.o example.o

From Git 1.7.6 onwards you can also use **git status** --ignored in order to see ignored files. You can find more info on this in the official documentation or in Finding files ignored by .gitignore.

## Section 5.\: Exceptions in a .gitignore file

If you ignore files by using a pattern but have exceptions, prefix an exclamation mark(!) to the exception. For example:

**\***.txt

**!**important.txt

The above example instructs Git to ignore all files with the .txt extension except for files named important.txt. If the file is in an ignored folder, you can **NOT** re-include it so easily:

folder**/**

**!**folder**/\***.txt

In this example all .txt files in the folder would remain ignored.

The right way is re-include the folder itself on a separate line, then ignore all files in folder by \*, finally re-include the **\***.txt in folder, as the following:

**!**folder**/**

folder**/\***

**!**folder**/\***.txt

**Note**: For file names beginning with an exclamation mark, add two exclamation marks or escape with the \

character:

**!!**includethis

\**!**excludethis

## Section 5.4: A global .gitignore file

To have Git ignore certain files across all repositories you can create a global .gitignore with the following command in your terminal or command prompt:

$ **git config** --global core.excludesfile **<**Path\_To\_Global\_gitignore\_file**>**

Git will now use this in addition to each repository's own .gitignore file. Rules for this are:

If the local .gitignore file explicitly includes a file while the global .gitignore ignores it, the local

.gitignore takes priority (the file will be included)

If the repository is cloned on multiple machines, then the global .gigignore must be loaded on all machines or at least include it, as the ignored files will be pushed up to the repo while the PC with the global

.gitignore wouldn't update it. This is why a repo specific .gitignore is a better idea than a global one if the project is worked on by a team

This file is a good place to keep platform, machine or user specific ignores, e.g. OSX .D5\_5tore, Windows Thumbs.db or Vim **\***.ext~ and **\***.ext.swp ignores if you don't want to keep those in the repository. So one team member working on OS X can add all .D5\_5T0RE and \_MAC05X (which is actually useless), while another team member on Windows can ignore all thumbs.bd

## Section 5.5: Ignore files that have alread6 been committed to

**a Git repositor6**

If you have already added a file to your Git repository and now want to **stop tracking it** (so that it won't be present in future commits), you can remove it from the index:

**git rm** --cached **<file>**

This will remove the file from the repository and prevent further changes from being tracked by Git. The --cached

option will make sure that the file is not physically deleted.

Note that previously added contents of the file will still be visible via the Git history.

Keep in mind that if anyone else pulls from the repository after you removed the file from the index, **their copy will be physically deleted**.

You can make Git pretend that the working directory version of the file is up to date and read the index version instead (thus ignoring changes in it) with "skip worktree" bit:

**git update–index** --skip-worktree **<file>**

Writing is not affected by this bit, content safety is still first priority. You will never lose your precious ignored changes; on the other hand this bit conflicts with stashing: to remove this bit, use

**git update–index** --no-skip-worktree **<file>**

It is sometimes ***wrongly*** recommended to lie to Git and have it assume that file is unchanged without examining it. It looks at first glance as ignoring any further changes to the file, without removing it from its index:

**git update–index** --assume-unchanged **<file>**

This will force git to ignore any change made in the file (keep in mind that if you pull any changes to this file, or you stash it, **your ignored changes will be lost**)

If you want git to "care" about this file again, run the following command:

**git update–index** --no-assume-unchanged **<file>**

## Section 5.6: Ignore files locall6 without committing ignore rules

.gitignore ignores files locally, but it is intended to be committed to the repository and shared with other contributors and users. You can set a global .gitignore, but then all your repositories would share those settings.

If you want to ignore certain files in a repository locally and not make the file part of any repository, edit

.git**/**info**/**exclude inside your repository. For example:

***# łhese fiﾕes are onﾕy ignored on łhis repo # łhese ruﾕes are noł shared wiłh anyone***

***# as łhey are personaﾕ*** gtk\_tests.py gui**/**gtk**/**tests**/\***

localhost

pushReports.py server**/**

## Section 5.7: Ignoring subsequent changes to a file (without removing it)

Sometimes you want to have a file held in Git but ignore subsequent changes. Tell Git to ignore changes to a file or directory using update-index:

**git update–index** --assume-unchanged my-file.txt

The above command instructs Git to assume my-file.txt hasn't been changed, and not to check or report changes. The file is still present in the repository.

This can be useful for providing defaults and allowing local environment overrides, e.g.:

***# creałe a fiﾕe wiłh some vaﾕues in***

**cat *<<EOF MYSQL\_USER=app***

***MYSQL\_PASSWORD=FIXME\_SECRET\_PASSWORD***

***EOF* >** .env

***# commił ło Gił***

**git add** .env

**git commit** -m "Adding .env template"

***# ignore fułure changes ło .env***

**git update–index** --assume-unchanged .env

***# updałe your password***

**vi** .env

***# no changes!***

**git status**

## Section 5.8: Ignoring a file in an6 director6

To ignore a file foo.txt in **any** directory you should just write its name:

foo.txt ***# małches aﾕﾕ fiﾕes 'foo.łxł' in any direcłory***

If you want to ignore the file only in part of the tree, you can specify the subdirectories of a specific directory with

\*\* pattern:

bar**/\*\*/**foo.txt ***# małches aﾕﾕ fiﾕes 'foo.łxł' in 'bar' and aﾕﾕ subdirecłories***

Or you can create a .gitignore file in the bar**/** directory. Equivalent to the previous example would be creating file

bar**/**.gitignore with these contents:

foo.txt ***# małches aﾕﾕ fiﾕes 'foo.łxł' in any direcłory under bar/***

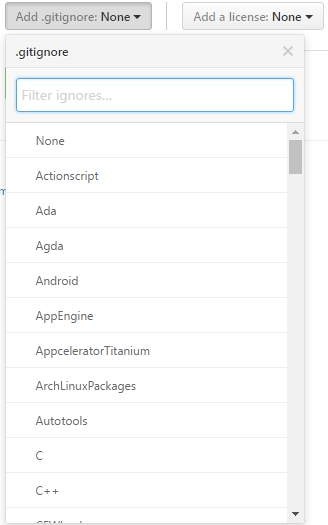
## Section 5.9: Prefilled .gitignore Templates

If you are unsure which rules to list in your .gitignore file, or you just want to add generally accepted exceptions

to your project, you can choose or generate a .gitignore file:

[https://www.gitignore.io/](http://www.gitignore.io/) https://github.com/github/gitignore

Many hosting services such as GitHub and BitBucket offer the ability to generate .gitignore files based upon the programming languages and IDEs you may be using:



## Section 5.10: Ignoring files in subfolders (Multiple gitignore files)

Suppose you have a repository structure like this:

examples**/**

output.log src**/**

**<**files not shown**>** output.log

README.md

output.log in the examples directory is valid and required for the project to gather an understanding while the one beneath src**/** is created while debugging and should not be in the history or part of the repository.

There are two ways to ignore this file. You can place an absolute path into the .gitignore file at the root of the working directory:

***# /.giłignore***

src**/**output.log

Alternatively, you can create a .gitignore file in the src**/** directory and ignore the file that is relative to this

.gitignore:

***# /src/.giłignore***

output.log

## Section 5.11: Create an Empt6 Folder

It is not possible to add and commit an empty folder in Git due to the fact that Git manages *files* and attaches their directory to them, which slims down commits and improves speed. To get around this, there are two methods:

Method one: .gitkeep

One hack to get around this is to use a .gitkeep file to register the folder for Git. To do this, just create the required directory and add a .gitkeep file to the folder. This file is blank and doesn't serve any purpose other than to just register the folder. To do this in Windows (which has awkward file naming conventions) just open git bash in the directory and run the command:

$ touch .gitkeep

This command just makes a blank .gitkeep file in the current directory Method two: dummy.txt

Another hack for this is very similar to the above and the same steps can be followed, but instead of a .gitkeep, just use a dummy.txt instead. This has the added bonus of being able to easily create it in Windows using the context menu. And you get to leave funny messages in them too.You can also use .gitkeep file to track the empty directory. .gitkeep normally is an empty file that is added to track the empty directory.

## Section 5.12: Finding files ignored b6 .gitignore

You can list all files ignored by git in current directory with command:

**git status** --ignored

So if we have repository structure like this:

.git

.gitignore

.**/**example\_1

.**/**dir**/**example\_2

.**/**example\_2

...and .gitignore file containing:

example\_2

...than result of the command will be:

$ **git status** --ignored

0n branch master

Initial commit Untracked files:

**(**use "git add <file>..." to include **in** what will be committed**)**

.gitignore

.example\_1

Ignored files:

**(**use "git add -f <file>..." to include **in** what will be committed**)**

dir**/** example\_2

If you want to list recursively ignored files in directories, you have to use additional parameter - --untracked- files=all

Result will look like this:

$ **git status** --ignored --untracked-files=all 0n branch master

Initial commit Untracked files:

**(**use "git add <file>..." to include **in** what will be committed**)**

.gitignore example\_1

Ignored files:

**(**use "git add -f <file>..." to include **in** what will be committed**)**

dir**/**example\_2 example\_2

## Section 5.1\: Ignoring onl6 part of a file [stub]

Sometimes you may want to have local changes in a file you don't want to commit or publish. Ideally local settings should be concentrated in a separate file that can be placed into .gitignore, but sometimes as a short-term solution it can be helpful to have something local in a checked-in file.

You can make Git "unsee" those lines using clean filter. They won't even show up in diffs. Suppose here is snippet from file file1.c:

struct settings s; s.host = "localhost"; s.port = 5б53;

s.auth = 1;

s.port = 15б53; **//** N0C0MMIT s.debug = 1; **//** N0C0MMIT s.auth = θ; **//** N0C0MMIT

You don't want to publish N0C0MMIT lines anywhere.

Create "nocommit" filter by adding this to Git config file like .git**/**config:

**[**filter "nocommit"**]** clean=**grep** -v N0C0MMIT

Add (or create) this to .git**/**info**/**attributes or .gitmodules:

file1.c filter=nocommit

And your NOCOMMIT lines are hidden from Git. Caveats:

Using clean filter slows down processing of files, especially on Windows.

The ignored line may disappear from file when Git updates it. It can be counteracted with a smudge filter, but it is trickier.

Not tested on Windows

## Section 5.14: Ignoring changes in tracked files. [stub]

.gitignore and .git**/**info**/**exclude work only for untracked files.

To set ignore flag on a tracked file, use the command update-index:

**git update–index** --skip-worktree myfile.c

To revert this, use:

**git update–index** --no-skip-worktree myfile.c

You can add this snippet to your global git config to have more convenient **git** hide, **git** unhide and **git** hidden

commands:

**[alias]**

hide = update-index --skip-worktree unhide = update-index --no-skip-worktree

hidden = "!git ls-files -v | grep ^[hs5] | cut -c 3-"

You can also use the option --assume-unchanged with the update-index function

**git update–index** --assume-unchanged **<file>**

If you want to watch this file again for the changes, use

**git update–index** --no-assume-unchanged **<file>**

When --assume-unchanged flag is specified, the user promises not to change the file and allows Git to assume that the working tree file matches what is recorded in the index.Git will fail in case it needs to modify this file in the index e.g. when merging in a commit; thus, in case the assumed-untracked file is changed upstream, you will need to handle the situation manually.The focus lies on performance in this case.

While --skip-worktree flag is useful when you instruct git not to touch a specific file ever because the file is going to be changed locally and you don't want to accidentally commit the changes (i.e configuration/properties file configured for a particular environment). Skip-worktree takes precedence over assume-unchanged when both are set.

## Section 5.15: Clear alread6 committed files, but included in

**.gitignore**

Sometimes it happens that a file was being tracked by git, but in a later point in time was added to .gitignore, in order to stop tracking it. It's a very common scenario to forget to clean up such files before its addition to .gitignore. In this case, the old file will still be hanging around in the repository.

To fix this problem, one could perform a "dry-run" removal of everything in the repository, followed by re-adding all the files back. As long as you don't have pending changes and the --cached parameter is passed, this command is fairly safe to run:

***# Remove everyłhing from łhe index (łhe fiﾕes wiﾕﾕ słay in łhe fiﾕe sysłem)***

$ **git rm** -r --cached .

***# Re-add everyłhing (łhey'ﾕﾕ be added in łhe currenł słałe, changes incﾕuded)***

$ **git add** .

***# Commił, if anyłhing changed. You shouﾕd see onﾕy deﾕełions***

$ **git commit** -m 'Remove all files that are in the .gitignore'

***# Updałe łhe remołe***

$ **git push** origin master

# Chapter 6: Git Diff

### Parameter Details

-p, -u, --patch Generate patch

Suppress diff output. Useful for commands like **git show** that show the patch by default, or to cancel the effect of --patch

-s, --no-patch

--raw Generate the diff in raw format

--diff-algorithm= Choose a diff algorithm. The variants are as follows: myers, minimal, patience, histogram

Output a condensed summary of extended header information such as creations, renames and mode changes

--summary

--name-only Show only names of changed files

Show names and statuses of changed files The most common statuses are M (Modified), A (Added), and D (Deleted)

--name-status

Warn if changes introduce conflict markers or whitespace errors. What are considered whitespace errors is controlled by core.whitespace configuration. By default, trailing whitespaces (including

--check

lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non- zero status if problems are found. Not compatible with --exit-code

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output

--full-index

--binary In addition to --full-index, output a binary diff that can be applied with **git apply**

-a, --text Treat all files as text.

Set the color mode; i.e. use --color=always if you would like to pipe a diff to less and keep git's coloring

--color

**Section 6.1: Show differences in working branch**

**git diff**

This will show the *unstaged* changes on the current branch from the commit before it. It will only show changes relative to the index, meaning it shows what you *could* add to the next commit, but haven't. To add (stage) these changes, you can use **git add**.

If a file is staged, but was modified after it was staged, **git diff** will show the differences between the current file and the staged version.

## Section 6.2: Show changes between two commits

**git diff** 1234abc..б789def

***# oﾕd new***

E.g.: Show the changes made in the last 3 commits:

**git diff @**~3..**@**

***# HEAD -3 HEAD***

Note: the two dots (..) is optional, but adds clarity.

This will show the textual difference between the commits, regardless of where they are in the tree.

## Section 6.\: Show differences for staged files

**git diff** --staged

This will show the changes between the previous commit and the currently staged files.

**NOTE:** You can also use the following commands to accomplish the same thing:

**git diff** --cached

Which is just a synonym for --staged or

**git status** -v

Which will trigger the verbose settings of the status command.

## Section 6.4: Comparing branches

Show the changes between the tip of **new** and the tip of **original**:

**git diff** original new

***# equivaﾕenł ło originaﾕ..new***

Show all changes on **new** since it branched from **original**:

**git diff** original...new

***# equivaﾕenł ło ‡(gił merge-base originaﾕ new)..new***

Using only one parameter such as git diff original

is equivalent to

git diff original..HEAD

## Section 6.5: Show both staged and unstaged changes

To show all staged *and* unstaged changes, use:

**git diff** HEAD

**NOTE:** You can also use the following command:

**git status** -vv

The difference being that the output of the latter will actually tell you which changes are staged for commit and which are not.

## Section 6.6: Show differences for a specific file or director6

**git diff** myfile.txt

Shows the changes between the previous commit of the specified file (myfile.txt) and the locally-modified version that has not yet been staged.

This also works for directories:

**git diff** documentation

The above shows the changes between the previous commit of all files in the specified directory (documentation**/**) and the locally-modified versions of these files, that have not yet been staged.

To show the difference between some version of a file in a given commit and the local HEAD version you can specify the commit you want to compare against:

**git diff** 27fa75e myfile.txt

Or if you want to see the version between two separate commits:

**git diff** 27fa75e ada9b57 myfile.txt

To show the difference between the version specified by the hash ada9b57 and the latest commit on the branch

my\_branchname for only the relative directory called my\_changed\_directory**/** you can do this:

**git diff** ada9b57 my\_branchname my\_changed\_directory**/**

## Section 6.7: Viewing a word-diff for long lines

**git diff [**HEAD**|**--staged...**]** --word-diff

Rather than displaying lines changed, this will display differences within lines. For example, rather than:

-Hello world

+Hello world**!**

Where the whole line is marked as changed, word-diff alters the output to:

Hello **[**-world-**]{**+world**!**+**}**

You can omit the markers [-, -], {+, +} by specifying --word-diff=color or --color-words. This will only use color coding to mark the difference:



## Section 6.8: Show differences between current version and last version

**git diff** HEAD^ HEAD

This will show the changes between the previous commit and the current commit.

## Section 6.9: Produce a patch-compatible diff

Sometimes you just need a diff to apply using patch. The regular **git** --diff does not work. Try this instead:

**git diff** --no-prefix **>** some\_file.patch

Then somewhere else you can reverse it:

**patch** -pθ **<** some\_file.patch

## Section 6.10: difference between two commit or branch

To view difference between two branch

**git diff <**branch1**>**..**<**branch2**>**

To view difference between two branch

**git diff <**commitId1**>**..**<**commitId2**>**

To view diff with current branch

**git diff <**branch**/**commitId**>**

To view summary of changes

**git diff** --stat **<**branch**/**commitId**>**

To view files that changed after a certain commit

**git diff** --name-only **<**commitId**>**

To view files that are different than a branch

**git diff** --name-only **<**branchName**>**

To view files that changed in a folder after a certain commit

**git diff** --name-only **<**commitId**> <**folder\_path**>**

## Section 6.11: Using meld to see all modifications in the working director6

**git difftool** -t meld --dir-diff

will show the working directory changes. Alternatively,

**git difftool** -t meld --dir-diff **[**C0MMIT\_A**] [**C0MMIT\_B**]**

will show the differences between 2 specific commits.

## Section 6.12: Diff UTF-16 encoded text and binar6 plist files

You can diff UTF-16 encoded files (localization strings file os iOS and macOS are examples) by specifying how git should diff these files.

Add the following to your ~**/**.gitconfig file.

**[diff** "utf1б"**]**

textconv = "iconv -f utf-1б -t utf-8"

iconv is a program to convert different encodings.

Then edit or create a .gitattributes file in the root of the repository where you want to use it. Or just edit

~**/**.gitattributes.

**\***.strings diff=utf1б

This will convert all files ending in .strings before git diffs.

You can do similar things for other files, that can be converted to text. For binary plist files you edit .gitconfig

**[diff** "plist"**]**

textconv = plutil -convert xml1 -o -

and .gitattributes

**\***.plist diff=plist

# Chapter 7: Undoing

## Section 7.1: Return to a previous commit

To jump back to a previous commit, first find the commit's hash using **git log**. To temporarily jump back to that commit, detach your head with:

**git checkout** 789abcd

This places you at commit 789abcd. You can now make new commits on top of this old commit without affecting the branch your head is on. Any changes can be made into a proper branch using either branch or checkout -b.

To roll back to a previous commit while keeping the changes:

**git reset** --soft 789abcd

To roll back the ***last*** commit:

**git reset** --soft HEAD~

To permanently discard any changes made after a specific commit, use:

**git reset** --hard 789abcd

To permanently discard any changes made after the ***last*** commit:

**git reset** --hard HEAD~

**Beware:** While you can recover the discarded commits using reflog and reset, uncommitted changes cannot be recovered. Use **git stash**; **git reset** instead of **git reset** --hard to be safe.

## Section 7.2: Undoing changes

Undo changes to a file or directory in the **working copy**.

**git checkout** -- file.txt

Used over all file paths, recursively from the current directory, it will undo all changes in the working copy.

**git checkout** -- .

To only undo parts of the changes use --patch. You will be asked, for each change, if it should be undone or not.

**git checkout** --patch -- **dir**

To undo changes added to the **index**.

**git reset** --hard

Without the --hard flag this will do a soft reset.

With local commits that you have yet to push to a remote you can also do a soft reset. You can thus rework the files

and then the commits.

**git reset** HEAD~2

The above example would unwind your last two commits and return the files to your working copy. You could then make further changes and new commits.

**Beware:** All of these operations, apart from soft resets, will permanently delete your changes. For a safer option, use **git stash** -p or **git stash**, respectively. You can later undo with stash pop or delete forever with stash drop.

## Section 7.\: Using reflog

If you screw up a rebase, one option to start again is to go back to the commit (pre rebase). You can do this using

reflog (which has the history of everything you've done for the last 90 days - this can be configured):

$ **git reflog**

4a5cbb3 HEAD**@{**θ**}**: rebase finished: returning to refs**/**heads**/**foo 4a5cbb3 HEAD**@{**1**}**: rebase: fixed such and such

9θ4f7fθ HEAD**@{**2**}**: rebase: checkout upstream**/**master 3cbe2θa HEAD**@{**3**}**: commit: fixed such and such

...

You can see the commit before the rebase was HEAD**@{**3**}** (you can also checkout the hash):

**git checkout** HEAD**@{**3**}**

*Now you create a new branch / delete the old one / try the rebase again.*

You can also reset directly back to a point in your reflog, but only do this if you're 100% sure it's what you want to do:

**git reset** --hard HEAD**@{**3**}**

This will set your current git tree to match how it was at that point (See Undoing Changes).

This can be used if you're temporarily seeing how well a branch works when rebased on another branch, but you don't want to keep the results.

## Section 7.4: Undoing merges

### Undoing a merge not yet pushed to a remote

If you haven't yet pushed your merge to the remote repository then you can follow the same procedure as in undo the commit although there are some subtle differences.

A reset is the simplest option as it will undo both the merge commit and any commits added from the branch. However, you will need to know what SHA to reset back to, this can be tricky as your **git log** will now show commits from both branches. If you reset to the wrong commit (e.g. one on the other branch) **it can destroy**

### committed work.

* **git reset** --hard **<last** commit from the branch you are on**>**

Or, assuming the merge was your most recent commit.

* **git reset** HEAD~

A revert is safer, in that it won't destroy committed work, but involves more work as you have to revert the revert before you can merge the branch back in again (see the next section).

### Undoing a merge pushed to a remote

Assume you merge in a new feature (add-gremlins)

* **git merge** feature**/**add-gremlins

...

***#Resoﾕve any merge confﾕicłs***

* **git commit *#commił łhe merge***

...

* **git push**

...

5θ1b75d..17a51fd master -**>** master

Afterwards you discover that the feature you just merged in broke the system for other developers, it must be undone right away, and fixing the feature itself will take too long so you simply want to undo the merge.

* **git revert** -m 1 17a51fd

...

* **git push**

...

17a51fd..e443799 master -**>** master

At this point the gremlins are out of the system and your fellow developers have stopped yelling at you. However, we are not finished just yet. Once you fix the problem with the add-gremlins feature you will need to undo this revert before you can merge back in.

* **git checkout** feature**/**add-gremlins

...

***#Various commiłs ło fix łhe bug.***

* **git checkout** master

...

* **git revert** e443799

...

* **git merge** feature**/**add-gremlins

...

***#Fix any merge confﾕicłs inłroduced by łhe bug fix***

* **git commit *#commił łhe merge***

...

* **git push**

At this point your feature is now successfully added. However, given that bugs of this type are often introduced by merge conflicts a slightly different workflow is sometimes more helpful as it lets you fix the merge conflict on your branch.

* **git checkout** feature**/**add-gremlins

...

***#Merge in masłer and reverł łhe reverł righł away. This pułs your branch in***

***#łhe same broken słałe łhał masłer was in before.***

* **git merge** master

...

* **git revert** e443799

...

***#Now go ahead and fix łhe bug (various commiłs go here)***

* **git checkout** master

...

***#Don'ł need ło reverł łhe reverł ał łhis poinł since ił was done earﾕier***

* **git merge** feature**/**add-gremlins

...

***#Fix any merge confﾕicłs inłroduced by łhe bug fix***

* **git commit *#commił łhe merge***

...

* **git push**

## Section 7.5: Revert some existing commits

Use git revert to revert existing commits, especially when those commits have been pushed to a remote repository. It records some new commits to reverse the effect of some earlier commits, which you can push safely without rewriting history.

**Don't** use **git push** --force unless you wish to bring down the opprobrium of all other users of that repository. Never rewrite public history.

If, for example, you've just pushed up a commit that contains a bug and you need to back it out, do the following:

**git revert** HEAD~1

**git push**

Now you are free to revert the revert commit locally, fix your code, and push the good code:

**git revert** HEAD~1

work .. work .. work ..

**git add** -A .

**git commit** -m "Update error code"

**git push**

If the commit you want to revert is already further back in the history, you can simply pass the commit hash. Git will create a counter-commit undoing your original commit, which you can push to your remote safely.

**git revert** 912aafθ228338dθc8fb8ccaθaθб4bθ1б1a451fdc

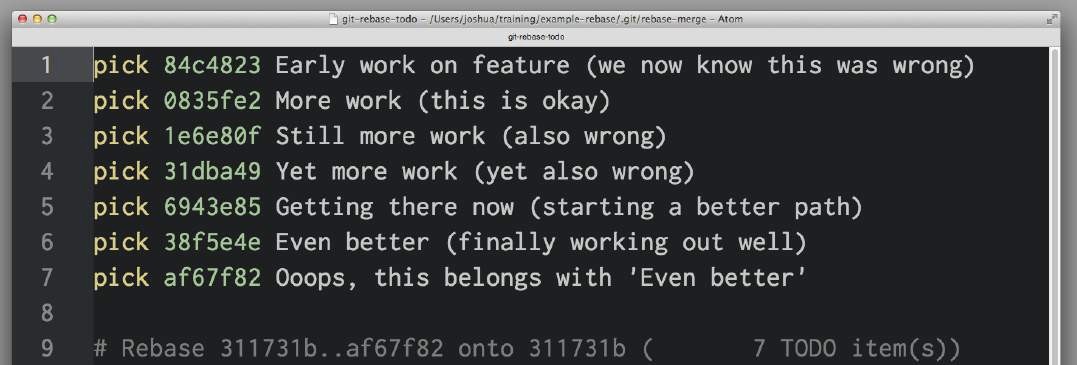
**git push**

## Section 7.6: Undo / Redo a series of commits

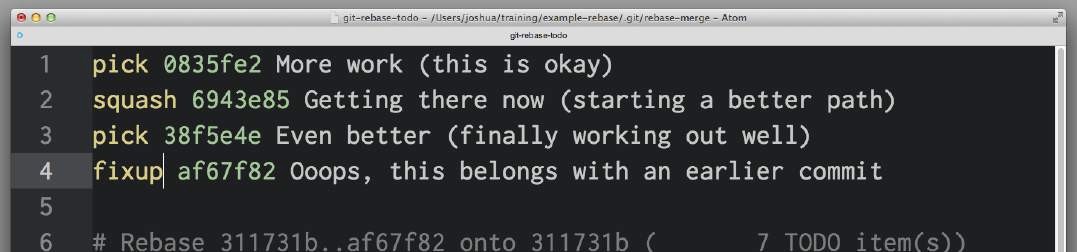
Assume you want to undo a dozen of commits and you want only some of them.

**git rebase** -i **<**earlier 5HA**>**

-i puts rebase in "interactive mode". It starts off like the rebase discussed above, but before replaying any commits, it pauses and allows you to gently modify each commit as it's replayed.***rebase -i*** will open in your default text editor, with a list of commits being applied, like this:



To drop a commit, just delete that line in your editor. If you no longer want the bad commits in your project, you can delete lines 1 and 3-4 above.If you want to combine two commits together, you can use the squash or fixup commands



# Chapter 8: Merging

### Parameter Details

-m Message to be included in the merge commit

-v Show verbose output

--abort Attempt to revert all files back to their state

--ff-only Aborts instantly when a merge-commit would be required

--no-ff Forces creation of a merge-commit, even if it wasn't mandatory

--no-commit Pretends the merge failed to allow inspection and tweaking of the result

--stat Show a diffstat after merge completion

-n/--no-stat Don't show the diffstat

--squash Allows for a single commit on the current branch with the merged changes

**Section 8.1: Automatic Merging**

When the commits on two branches don't conflict, Git can automatically merge them:

~**/**5tack 0verflow**(**branch:master**)** » **git merge** another\_branch Auto-merging file\_a

Merge made by the 'recursive' strategy. file\_a **|** 2 +-

1 **file** changed, 1 insertion**(**+**)**, 1 deletion**(**-**)**

## Section 8.2: Finding all branches with no merged changes

Sometimes you might have branches lying around that have already had their changes merged into master. This finds all branches that are not master that have no unique commits as compared to master. This is very useful for finding branches that were not deleted after the PR was merged into master.

**for** branch **in** $**(git branch** -r**)** ; **do**

**[** "${branch}" **!**= "origin/master" **] && [** $**(git diff** master...${branch} **| wc** -l**)** -eq θ **] && echo** - e **`git show** --pretty=format:"%ci %cr" $branch **| head** -n 1**`**\\t$branch

**done | sort** -r

## Section 8.\: Aborting a merge

After starting a merge, you might want to stop the merge and return everything to its pre-merge state. Use --abort:

**git merge** --abort

## Section 8.4: Merge with a commit

Default behaviour is when the merge resolves as a fast-forward, only update the branch pointer, without creating a merge commit. Use --no-ff to resolve.

**git merge <**branch\_name**>** --no-ff -m "<commit message>"

## Section 8.5: ßeep changes from onl6 one side of a merge

During a merge, you can pass --ours or --theirs to **git checkout** to take all changes for a file from one side or the other of a merge.

$ **git checkout** --ours -- file1.txt ***# Use our version of fiﾕe1, deﾕełe aﾕﾕ łheir changes***

$ **git checkout** --theirs -- file2.txt ***# Use łheir version of fiﾕe2, deﾕełe aﾕﾕ our changes***

## Section 8.6: Merge one branch into another

**git merge** incomingBranch

This merges the branch incomingBranch into the branch you are currently in. For example, if you are currently in

master, then incomingBranch will be merged into master.

Merging can create conflicts in some cases. If this happens, you will see the message Automatic merge failed; fix conflicts and **then** commit the result. You will need to manually edit the conflicted files, or to undo your merge attempt, run:

**git merge** --abort

# Chapter 9: Submodules

## Section 9.1: Cloning a Git repositor6 having submodules

When you clone a repository that uses submodules, you'll need to initialize and update them.

$ **git clone** --recursive https:**//**github.com**/**username**/**repo.git

This will clone the referenced submodules and place them in the appropriate folders (including submodules within submodules). This is equivalent to running **git submodule** update --init --recursive immediately after the clone is finished.

## Section 9.2: Updating a Submodule

A submodule references a specific commit in another repository. To check out the exact state that is referenced for all submodules, run

**git submodule** update --recursive

Sometimes instead of using the state that is referenced you want to update to your local checkout to the latest state of that submodule on a remote. To check out all submodules to the latest state on the remote with a single command, you can use

**git submodule** foreach **git pull <**remote**> <**branch**>**

or use the default **git pull** arguments

**git submodule** foreach **git pull**

Note that this will just update your local working copy. Running **git status** will list the submodule directory as dirty if it changed because of this command. To update your repository to reference the new state instead, you have to commit the changes:

**git add <**submodule\_directory**> git commit**

There might be some changes you have that can have merge conflict if you use **git pull** so you can use **git pull**

--rebase to rewind your changes to top, most of the time it decreases the chances of conflict. Also it pulls all the branches to local.

**git submodule** foreach **git pull** --rebase

To checkout the latest state of a specific submodule, you can use :

**git submodule** update --remote **<**submodule\_directory**>**

## Section 9.\: Adding a submodule

You can include another Git repository as a folder within your project, tracked by Git:

$ **git submodule** add https:**//**github.com**/**jquery**/**jquery.git

You should add and commit the new .gitmodules file; this tells Git what submodules should be cloned when **git submodule** update is run.

## Section 9.4: Setting a submodule to follow a branch

A submodule is always checked out at a specific commit SHA1 (the "gitlink", special entry in the index of the parent repo)

But one can request to update that submodule to the latest commit of a branch of the submodule remote repo.

Rather than going in each submodule, doing a **git checkout** abranch --track origin**/**abranch, **git pull**, you can simply do (from the parent repo) a:

**git submodule** update --remote --recursive

Since the SHA1 of the submodule would change, you would still need to follow that with:

**git add** .

**git commit** -m "update submodules"

That supposes the submodules were: either added with a branch to follow:

**git submodule** -b abranch -- **/**url**/**of**/**submodule**/**repo

or configured (for an existing submodule) to follow a branch:

**cd /**path**/**to**/**parent**/**repo

**git config** -f .gitmodules submodule.asubmodule.branch abranch

## Section 9.5: Moving a submodule

Version > 1.8

Run:

$ **git mv** /path/to/module *new/path/to/module*

Version c 1.8

* 1. Edit .gitmodules and change the path of the submodule appropriately, and put it in the index with **git add**

.gitmodules.

* 1. If needed, create the parent directory of the new location of the submodule (**mkdir** -p /path/to).
  2. Move all content from the old to the new directory (**mv** -vi /path/to/module *new/path/to/submodule*).
  3. Make sure Git tracks this directory (**git add** /path/to).
  4. Remove the old directory with **git rm** --cached /path/to/module.
  5. Move the directory .git**/**modules**/**/path/to/module with all its content to .git**/**modules**/**/path/to/module.
  6. Edit the .git**/**modules**/**/path/to/config file, make sure that worktree item points to the new locations, so in this example it should be worktree = ..**/**..**/**..**/**..**/**..**/**/path/to/module. Typically there should be two more

.. then directories in the direct path in that place. . Edit the file /path/to/module/.git, make sure that the path

in it points to the correct new location inside the main project .git folder, so in this example gitdir:

..**/**..**/**..**/**.git**/**modules**/**/path/to/module.

**git status** output looks like this afterwards:

***# On branch masłer***

***# Changes ło be commiłłed:***

***# # # #***

***#***

***(use "gił reseł HEAD <fiﾕe>..." ło unsłage)***

***modified:***

***renamed:***

***.giłmoduﾕes***

***oﾕd/pałh/ło/submoduﾕe -> new/pałh/ło/submoduﾕe***

* 1. Finally, commit the changes.

This example from Stack Overflow, by Axel Beckert

## Section 9.6: Removing a submodule

Version > 1.8

You can remove a submodule (e.g. the\_submodule) by calling:

$ **git submodule** deinit the\_submodule

$ **git rm** the\_submodule

**git submodule** deinit the\_submodule deletes the\_submodules' entry from .git/config. This excludes the\_submodule from **git submodule** update, **git submodule sync** and **git submodule** foreach calls and deletes its local content (source). Also, this will not be shown as change in your parent repository. **git submodule** init and **git submodule** update will restore the submodule, again without commitable changes in your parent repository.

**git rm** the\_submodule will remove the submodule from the work tree. The files will be gone as well as the submodules' entry in the .gitmodules file (source). If only **git rm** the\_submodule (without prior **git submodule** deinit the\_submodule is run, however, the submodules' entry in your .git/config file will remain.

Version < 1.8

Taken from here:

1. Delete the relevant section from the .gitmodules file.
2. Stage the .gitmodules changes **git add** .gitmodules
3. Delete the relevant section from .git**/**config.
4. Run **git rm** --cached path\_to\_submodule (no trailing slash).
5. Run **rm** -rf .git**/**modules**/**path\_to\_submodule
6. Commit **git commit** -m "Removed submodule <name>"
7. Delete the now untracked submodule files
8. **rm** -rf path\_to\_submodule

# Chapter 10: Committing

### Parameter Details

Message to include in the commit. Specifying this parameter bypasses Git's normal behavior of opening an editor.

--message, -m

Specify that the changes currently staged should be added (amended) to the

--amend

*previous* commit. Be careful, this can rewrite history!

Use the selected commit message without launching an editor. For example, **git**

--no-edit

**commit** --amend --no-edit amends a commit without changing its commit message.

--all, -a Commit all changes, including changes that aren't yet staged.

--date Manually set the date that will be associated with the commit.

Commit only the paths specified. This will not commit what you currently have staged unless told to do so.

--only

--patch, -p Use the interactive patch selection interface to chose which changes to commit.

--help Displays the man page for **git commit**

-S[keyid], -S --gpg- sign[=keyid], -S --no-gpg-sign

Sign commit, GPG-sign commit, countermand commit.gpg5ign configuration variable

-n, --no-verify This option bypasses the pre-commit and commit-msg hooks. See also Hooks

Commits with Git provide accountability by attributing authors with changes to code. Git offers multiple features for the specificity and security of commits. This topic explains and demonstrates proper practices and procedures in committing with Git.

**Section 10.1: Stage and commit changes**

### The basics

After making changes to your source code, you should **stage** those changes with Git before you can commit them. For example, if you change README.md and program.py:

**git add** README.md program.py

This tells git that you want to add the files to the next commit you do. Then, commit your changes with

**git commit**

Note that this will open a text editor, which is often vim. If you are not familiar with vim, you might want to know that you can press i to go into *insert* mode, write your commit message, then press Esc and :wq to save and quit. To avoid opening the text editor, simply include the -m flag with your message

**git commit** -m "Commit message here"

Commit messages often follow some specific formatting rules, see Good commit messages for more information.

### Shortcuts

If you have changed a lot of files in the directory, rather than listing each one of them, you could use:

**git add** --all

***# equivaﾕenł ło "gił add -a"***

Or to add all changes, *not including files that have been deleted*, from the top-level directory and subdirectories:

**git add** .

Or to only add files which are currently tracked ("update"):

**git add** -u

If desired, review the staged changes:

**git status**

**git diff** --cached

***# dispﾕay a ﾕisł of changed fiﾕes***

***# shows słaged changes inside słaged fiﾕes***

Finally, commit the changes:

**git commit** -m "Commit message here"

Alternately, if you have only modified existing files or deleted files, and have not created any new ones, you can combine the actions of **git add** and **git commit** in a single command:

**git commit** -am "Commit message here"

Note that this will stage **all** modified files in the same way as **git add** --all.

### Sensitive data

You should never commit any sensitive data, such as passwords or even private keys. If this case happens and the changes are already pushed to a central server, consider any sensitive data as compromised. Otherwise, it is possible to remove such data afterwards. A fast and easy solution is the usage of the "BFG Repo-Cleaner": https://rtyley.github.io/bfg-repo-cleaner/.

The command bfg --replace-text passwords.txt my-repo.git reads passwords out of the passwords.txt file and replaces these with **\*\*\***REM0VED**\*\*\***. This operation considers all previous commits of the entire repository.

## Section 10.2: Good commit messages

It is important for someone traversing through the **git log** to easily understand what each commit was all about. Good commit messages usually include a number of a task or an issue in a tracker and a concise description of what has been done and why, and sometimes also how it has been done.

Better messages may look like:

TASK-123: Imptement togin through OAuth

TASK-124: Add auto minification of JS/CSS fites TASK-125: Fix minifier error when name > 288 chars

Whereas the following messages would not be quite as useful:

fix

// What has been fixed?

just a bit of a change

TASK-371

// What has changed?

// No description at att, reader witt need to took at the tracker

themsetves for an exptanation

Imptemented IFoo in IBar // Why it was needed?

A way to test if a commit message is written in the correct mood is to replace the blank with the message and see if it makes sense:

### If I add this commit, I will to my repository. The seven rules of a great git commit message

1. Separate the subject line from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Manually wrap each line of the body at 72 characters
7. Use the body to explain *what* and *why* instead of *how*

*7 rules from Chris Beam's blog.*

## Section 10.\: Amending a commit

If your **latest commit is not published yet** (not pushed to an upstream repository) then you can amend your commit.

**git commit** --amend

This will put the currently staged changes onto the previous commit.

**Note:** This can also be used to edit an incorrect commit message. It will bring up the default editor (usually vi / **vim**

/ emacs) and allow you to change the prior message.

To specify the commit message inline:

**git commit** --amend -m "New commit message"

Or to use the previous commit message without changing it:

**git commit** --amend --no-edit

Amending updates the commit date but leaves the author date untouched. You can tell git to refresh the information.

**git commit** --amend --reset-author

You can also change the author of the commit with:

**git commit** --amend --author "New Author [<email@address.com>"](mailto:email@address.com)

**Note:** Be aware that amending the most recent commit replaces it entirely and the previous commit is removed from the branch's history. This should be kept in mind when working with public repositories and on branches with other collaborators.

This means that if the earlier commit had already been pushed, after amending it you will have to push --force.

## Section 10.4: Committing without opening an editor

Git will usually open an editor (like **vim** or emacs) when you run **git commit**. Pass the -m option to specify a message from the command line:

**git commit** -m "Commit message here"

Your commit message can go over multiple lines:

**git commit** -m "Commit 'subject line' message here

More detailed description follows here (after a blank line)."

Alternatively, you can pass in multiple -m arguments:

**git commit** -m "Commit summary" -m "More detailed description follows here"

*See How to Write a Git Commit Message. Udacity Git Commit Message Style Guide*

## Section 10.5: Committing changes directl6

Usually, you have to use **git add** or **git rm** to add changes to the index before you can **git commit** them. Pass the

-a or --all option to automatically add every change (to tracked files) to the index, including removals:

**git commit** -a

If you would like to also add a commit message you would do:

**git commit** -a -m "your commit message goes here"

Also, you can join two flags:

**git commit** -am "your commit message goes here"

You don't necessarily need to commit all files at once. Omit the -a or --all flag and specify which file you want to commit directly:

**git commit** path**/**to**/**a**/file** -m "your commit message goes here"

For directly committing more than one specific file, you can specify one or multiple files, directories and patterns as well:

**git commit** path**/**to**/**a**/file** path**/**to**/**a**/**folder**/\*** path**/**to**/**b**/file** -m "your commit message goes here"

## Section 10.6: Selecting which lines should be staged for committing

Suppose you have many changes in one or more files but from each file you only want to commit some of the changes, you can select the desired changes using:

**git add** -p

or

**git add** -p **[file]**

Each of your changes will be displayed individually, and for each change you will be prompted to choose one of he following options:

y - Yes, add this hunk

n - No, don’t add this hunk

1. - No, don’t add this hunk, or any other remaining hunks for this fite.

Usefut if you’ve atready added what you want to, and want to skip over the rest. s - Sptit the hunk into smatter hunks, if possibte

1. - Manuatty edit the hunk. This is probabty the most powerfut option. It witt open the hunk in a text editor and you can edit it as needed.

This will stage the parts of the files you choose. Then you can commit all the staged changes like this:

**git commit** -m 'Commit Message'

The changes that were not staged or committed will still appear in your working files, and can be committed later if required. Or if the remaining changes are unwanted, they can be discarded with:

**git reset** --hard

Apart from breaking up a big change into smaller commits, this approach is also useful for *reviewing* what you are about to commit. By individually confirming each change, you have an opportunity to check what you wrote, and can avoid accidentally staging unwanted code such as println/logging statements.

## Section 10.7: Creating an empt6 commit

Generally speaking, empty commits (or commits with state that is identical to the parent) is an error.

However, when testing build hooks, CI systems, and other systems that trigger off a commit, it's handy to be able to easily create commits without having to edit/touch a dummy file.

The --allow-empty commit will bypass the check.

**git commit** -m "This is a blank commit" --allow-empty

## Section 10.8: Committing on behalf of someone else

If someone else wrote the code you are committing, you can give them credit with the --author option:

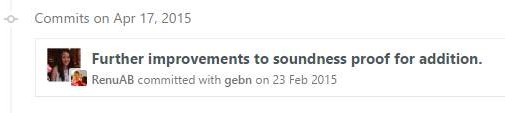
**git commit** -m "msg" --author "John 5mith [<johnsmith@example.com>"](mailto:johnsmith@example.com)

You can also provide a pattern, which Git will use to search for previous authors:

**git commit** -m "msg" --author "John"

In this case, the author information from the most recent commit with an author containing "John" will be used.

On GitHub, commits made in either of the above ways will show a large author's thumbnail, with the committer's smaller and in front:



## Section 10.9: GPG signing commits

1. Determine your key ID

**gpg** --list-secret-keys --keyid-format L0NG

**/**Users**/**davidcondrey**/**.gnupg**/**secring.gpg

sec 2θ48R**/**Y0UR-1б-DIGIT-KEY-ID YYYY-MM-DD **[**expires: YYYY-MM-DD**]**

Your ID is a alphanumeric 16-digit code following the first forward-slash.

1. Define your key ID in your git config

**git config** --global user.signingkey Y0UR-1б-DIGIT-KEY-ID

1. As of version 1.7.9, git commit accepts the -S option to attach a signature to your commits. Using this option will prompt for your GPG passphrase and will add your signature to the commit log.

**git commit** -5 -m "Your commit message"

## Section 10.10: Commiting changes in specific files

You can commit changes made to specific files and skip staging them using **git add**:

**git commit** file1.c file2.h

Or you can first stage the files:

**git add** file1.c file2.h

and commit them later:

**git commit**

## Section 10.11: Committing at a specific date

**git commit** -m 'Fix UI bug' --date 2θ1б-θ7-θ1

The --date parameter sets the *author date*. This date will appear in the standard output of **git log**, for example.

To force the *commit date* too:

GIT\_C0MMITTER\_DATE=2θ1б-θ7-θ1 **git commit** -m 'Fix UI bug' --date 2θ1б-θ7-θ1

The date parameter accepts the flexible formats as supported by GNU date, for example:

**git commit** -m 'Fix UI bug' --date yesterday

**git commit** -m 'Fix UI bug' --date '3 days ago'

**git commit** -m 'Fix UI bug' --date '3 hours ago'

When the date doesn't specify time, the current time will be used and only the date will be overridden.

## Section 10.12: Amending the time of a commit

You cam amend the time of a commit using

**git commit** --amend --date="Thu Jul 28 11:3θ 2θ1б -θ4θθ"

or even

**git commit** --amend --date="now"

## Section 10.1\: Amending the author of a commit

If you make a commit as the wrong author, you can change it, and then amend

**git config** user.name "Full Name"

**git config** user.email ["email@example.com"](mailto:email@example.com)

**git commit** --amend --reset-author

# Chapter 11: Aliases

## Section 11.1: Simple aliases

There are two ways of creating aliases in Git: with the ~**/**.gitconfig file:

**[alias]**

ci = commit st = status **co** = checkout

with the command line:

**git config** --global alias.ci "commit" **git config** --global alias.st "status" **git config** --global alias.co "checkout"

After the alias is created - type:

**git** ci instead of **git commit**, **git** st instead of **git status**, **git co** instead of **git checkout**.

As with regular git commands, aliases can be used beside arguments. For example:

**git** ci -m "Commit message..."

**git co** -b feature-42

## Section 11.2: List / search existing aliases

You can list existing git aliases using --get-regexp:

$ **git config** --get-regexp '^alias\.'

### Searching aliases

To search aliases, add the following to your .gitconfig under **[alias]**:

aliases = **!git config** --list **| grep** ^**alias**\\. **| cut** -c 7- **| grep** -Ei --color **\”**$1**\”** "#"

Then you can:

**git** aliases - show ALL aliases

**git** aliases commit - only aliases containing "commit"

## Section 11.\: Advanced Aliases

Git lets you use non-git commands and full sh shell syntax in your aliases if you prefix them with !. In your ~**/**.gitconfig file:

**[alias]**

temp = **!git add** -A **&& git commit** -m "Temp"

The fact that full shell syntax is available in these prefixed aliases also means you can use shell functions to construct more complex aliases, such as ones which utilize command line arguments:

**[alias]**

ignore = "!f() { echo $1 >> .gitignore; }; f"

The above alias defines the f function, then runs it with any arguments you pass to the alias. So running **git**

ignore .tmp**/** would add .tmp**/** to your .gitignore file.

In fact, this pattern is so useful that Git defines $1, $2, etc. variables for you, so you don't even have to define a special function for it. (But keep in mind that Git will also append the arguments anyway, even if you access it via these variables, so you may want to add a dummy command at the end.)

Note that aliases prefixed with ! in this way are run from the root directory of your git checkout, even if your current directory is deeper in the tree. This can be a useful way to run a command from the root without having to cd there explicitly.

**[alias]**

ignore = "! echo $1 >> .gitignore"

## Section 11.4: Temporaril6 ignore tracked files

To temporarily mark a file as ignored (pass file as parameter to alias) - type:

unwatch = update-index --assume-unchanged

To start tracking file again - type:

**watch** = update-index --no-assume-unchanged

To list all files that has been temporarily ignored - type:

unwatched = "!git ls-files -v | grep '^[[:lower:]]'"

To clear the unwatched list - type:

watchall = "!git unwatched | xargs -L 1 -I % sh -c 'git watch `echo % | cut -c 2-`'"

Example of using the aliases:

**git** unwatch my\_file.txt **git watch** my\_file.txt **git** unwatched

**git** watchall

## Section 11.5: Show prett6 log with branch graph

**[alias]**

logp=log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

lg = log --graph --date-order --first-parent \

--pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold

cyan)<%an>%Creset'

lgb = log --graph --date-order --branches --first-parent \

--pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold cyan)<%an>%Creset'

lga = log --graph --date-order --all \

--pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold cyan)<%an>%Creset'

Here an explanation of the options and placeholder used in the --pretty format (exhaustive list are available with

**git help** log )

--graph - draw the commit tree

--date-order - use commit timestamp order when possible

--first-parent - follow only the first parent on merge node.

--branches - show all local branches (by default, only current branch is shown)

--all - show all local and remotes branches

%h - hash value for commit (abbreviated)

%ad - Date stamp (author)

%an - Author username

%an - Commit username

%C(auto) - to use colors defined in [color] section

%Creset - to reset color

%d - --decorate (branch & tag names)

%s - commit message

%ad - author date (will follow --date directive) (and not commiter date)

%an - author name (can be %cn for commiter name)

## Section 11.6: See which files are being ignored b6 6our

**.gitignore configuration**

**[ alias ]**

ignored = **! git ls–files** --others --ignored --exclude-standard --directory \

**&& git ls–files** --others -i --exclude-standard

Shows one line per file, so you can grep (only directories):

$ **git** ignored **| grep** '/$'

.yardoc**/** doc**/**

Or count:

~$ **git** ignored **| wc** -l

199811 ***# oops, my home direcłory is gełłing crowded***

## Section 11.7: Updating code while keeping a linear histor6

Sometimes you need to keep a linear (non-branching) history of your code commits. If you are working on a branch for a while, this can be tricky if you have to do a regular **git pull** since that will record a merge with upstream.

**[alias]**

up = pull --rebase

This will update with your upstream source, then reapply any work you have not pushed on top of whatever you pulled down.

To use:

**git** up

## Section 11.8: Unstage staged files

Normally, to remove files that are staged to be committed using the **git reset** commit, reset has a lot of functions depending on the arguments provided to it. To completely unstage all files staged, we can make use of git aliases to create a new alias that uses reset but now we do not need to remember to provide the correct arguments to reset.

**git config** --global alias.unstage "reset --"

Now, any time you want to **unstage** stages files, type **git** unstage and you are good to go.

# Chapter 12: Rebasing

### Parameter Details

--continue Restart the rebasing process after having resolved a merge conflict.

Abort the rebase operation and reset HEAD to the original branch. If branch was provided when

--abort

the rebase operation was started, then HEAD will be reset to branch. Otherwise HEAD will be reset to where it was when the rebase operation was started.

--keep-empty Keep the commits that do not change anything from its parents in the result.

--skip Restart the rebasing process by skipping the current patch.

Use merging strategies to rebase. When the recursive (default) merge strategy is used, this allows rebase to be aware of renames on the upstream side. Note that a rebase merge works by

-m, --merge

replaying each commit from the working branch on top of the upstream branch. Because of this, when a merge conflict happens, the side reported as ours is the so-far rebased series, starting with upstream, and theirs is the working branch. In other words, the sides are swapped.

Show a diffstat of what changed upstream since the last rebase. The diffstat is also controlled by the configuration option rebase.stat.

--stat

-x, --exec **command** Perform interactive rebase, stopping between each commit and executing **command**

**Section 12.1: Local Branch Rebasing**

**Rebasing** reapplies a series of commits on top of another commit.

To rebase a branch, checkout the branch and then rebase it on top of another branch.

**git checkout** topic

**git rebase** master ***# rebase currenł branch onło masłer branch***

This would cause:

A---B---C topic

/

D---E---F---G master

To turn into:

A'--B'--C' topic

/

D---E---F---G master

These operations can be combined into a single command that checks out the branch and immediately rebases it:

**git rebase** master topic ***# rebase łopic branch onło masłer branch***

**Important:** After the rebase, the applied commits will have a different hash. You should not rebase commits you have already pushed to a remote host. A consequence may be an inability to **git push** your local rebased branch to a remote host, leaving your only option to **git push** --force.

## Section 12.2: Rebase: ours and theirs, local and remote

A rebase switches the meaning of "ours" and "theirs":

**git checkout** topic

**git rebase** master

***# rebase łopic branch on łop of masłer branch***

### Whatever HEAD's pointing to is "ours"

The first thing a rebase does is resetting the HEAD to master; before cherry-picking commits from the old branch topic to a new one (every commit in the former topic branch will be rewritten and will be identified by a different hash).

With respect to terminologies used by merge tools (not to be confused with local ref or remote ref)

=**> local** is master **(**"ours"**)**,

=**>** remote is topic **(**"theirs"**)**

That means a merge/diff tool will present the upstream branch as **local** (master: the branch on top of which you are rebasing), and the working branch as remote (topic: the branch being rebased)

+-----------------------------------------+

| LOCAL:master | BASE | REMOTE:topic |

+-----------------------------------------+

| MERGED |

+-----------------------------------------+

### Inversion illustrated On a merge:

c--c--x--x--x(\*) <- current branch topic ('\*'=HEAD)

\

\

\--y--y--y <- other branch to merge

We don't change the current branch topic, so what we have is still what we were working on (and we merge from another branch)

c--c--x--x--x---------o(\*) MERGE, stitt on branch topic

\

\

\

ˆ

ours

/

/

/

--y--y--y--/

ˆ theirs

### On a rebase:

But **on a rebase** we switch sides because the first thing a rebase does is to checkout the upstream branch to replay the current commits on top of it!

c--c--x--x--x(\*) <- current branch topic ('\*'=HEAD)

\

\

\--y--y--y <- upstream branch

A **git rebase upstream** will first set HEAD to the upstream branch, hence the switch of 'ours' and 'theirs' compared to the previous "current" working branch.

c--c--x--x--x <- former "current" branch, new "theirs"

\

\

\--y--y--y(\*) <- set HEAD to this commit, to reptay x's on it ˆ this witt be the new "ours"

| upstream

The rebase will then replay 'their' commits on the new 'our' topic branch:

c--c..x..x..x <- otd "theirs" commits, now "ghosts", avaitabte through "reftogs"

\

\

\--y--y--y--x'--x'--x'(\*) <- topic once att x's are reptayed, ˆ point branch topic to this commit

|

upstream branch

## Section 12.\: Interactive Rebase

This example aims to describe how one can utilize **git rebase** in interactive mode. It is expected that one has a basic understanding of what **git rebase** is and what it does.

Interactive rebase is initiated using following command:

**git rebase** -i

The -i option refers to *interactive mode*. Using interactive rebase, the user can change commit messages, as well as reorder, split, and/or squash (combine to one) commits.

Say you want to rearrange your last three commits. To do this you can run:

**git rebase** -i HEAD~3

After executing the above instruction, a file will be opened in your text editor where you will be able to select how your commits will be rebased. For the purpose of this example, just change the order of your commits, save the file, and close the editor. This will initiate a rebase with the order you've applied. If you check **git log** you will see your commits in the new order you specified.

### Rewording commit messages

Now, you've decided that one of the commit messages is vague and you want it to be more descriptive. Let's examine the last three commits using the same command.

**git rebase** -i HEAD~3

Instead of rearranging the order the commits will be rebased, this time we will change pick, the default, to reword

on a commit where you would like to change the message.

When you close the editor, the rebase will initiate and it will stop at the specific commit message that you wanted to

reword. This will let you change the commit message to whichever you desire. After you've changed the message, simply close the editor to proceed.

### Changing the content of a commit

Besides changing the commit message you can also adapt the changes done by the commit. To do so just change pick to edit for one commit. Git will stop when it arrives at that commit and provide the original changes of the commit in the staging area. You can now adapt those changes by unstaging them or adding new changes.

As soon as the staging area contains all changes you want in that commit, commit the changes. The old commit message will be shown and can be adapted to reflect the new commit.

### Splitting a single commit into multiple

Say you've made a commit but decided at a later point this commit could be split into two or more commits instead. Using the same command as before, replace pick with edit instead and hit enter.

Now, git will stop at the commit you have marked for editing and place all of its content into the staging area. From that point you can run **git reset** HEAD^ to place the commit into your working directory. Then, you can add and commit your files in a different sequence - ultimately splitting a single commit into *n* commits instead.

### Squashing multiple commits into one

Say you have done some work and have multiple commits which you think could be a single commit instead. For that you can carry out **git rebase** -i HEAD~3, replacing 3 with an appropriate amount of commits.

This time replace pick with squash instead. During the rebase, the commit which you've instructed to be squashed will be squashed on top of the previous commit; turning them into a single commit instead.

## Section 12.4: Rebase down to the initial commit

Since Git 1.7.12 it is possible to rebase down to the root commit. The root commit is the first commit ever made in a repository, and normally cannot be edited. Use the following command:

**git rebase** -i --root

## Section 12.5: Configuring autostash

Autostash is a very useful configuration option when using rebase for local changes. Oftentimes, you may need to bring in commits from the upstream branch, but are not ready to commit just yet.

However, Git does not allow a rebase to start if the working directory is not clean. Autostash to the rescue:

**git config** --global rebase.autostash

**git rebase @{**u**}**

***# one łime configurałion***

***# exampﾕe rebase on upsłream branch***

The autostash will be applied whenever the rebase is finished. It does not matter whether the rebase finishes successfully, or if it is aborted. Either way, the autostash will be applied. If the rebase was successful, and the base commit therefore changed, then there may be a conflict between the autostash and the new commits. In this case, you will have to resolve the conflicts before committing. This is no different than if you would have manually stashed, and then applied, so there is no downside to doing it automatically.

## Section 12.6: Testing all commits during rebase

Before making a pull request, it is useful to make sure that compile is successful and tests are passing for each commit in the branch. We can do that automatically using -x parameter.

For example:

**git rebase** -i -x **make**

will perform the interactive rebase and stop after each commit to execute **make**. In case **make** fails, git will stop to give you an opportunity to fix the issues and amend the commit before proceeding with picking the next one.

## Section 12.7: Rebasing before a code review

### Summary

This goal is to reorganize all of your scattered commits into more meaningful commits for easier code reviews. If there are too many layers of changes across too many files at once, it is harder to do a code review. If you can reorganize your chronologically created commits into topical commits, then the code review process is easier (and possibly less bugs slip through the code review process).

This overly-simplified example is not the only strategy for using git to do better code reviews. It is the way I do it, and it's something to inspire others to consider how to make code reviews and git history easier/better.

This also pedagogically demonstrates the power of rebase in general. This example assumes you know about interactive rebasing.

### Assuming:

you're working on a feature branch off of master

your feature has three main layers: front-end, back-end, DB

you have made a lot of commits while working on a feature branch. Each commit touches multiple layers at once

you want (in the end) only three commits in your branch one containing all front end changes

one containing all back end changes one containing all DB changes

### Strategy:

we are going to change our chronological commits into "topical" commits.

first, split all commits into multiple, smaller commits -- each containing only one topic at a time (in our example, the topics are front end, back end, DB changes)

Then reorder our topical commits together and 'squash' them into single topical commits

### Example:

$ **git log** --oneline master..

97543θb db adding works: db.sql logic.rb

37θ2б5θ trying to allow adding todo items: page.html logic.rb 43bθ75a first draft: page.html and db.sql

$ **git rebase** -i master

This will be shown in text editor:

pick 43bθ75a first draft: page.html and db.sql

pick 37θ2б5θ trying to allow adding todo items: page.html logic.rb pick 97543θb db adding works: db.sql logic.rb

Change it to this:

e 43bθ75a first draft: page.html and db.sql

e 37θ2б5θ trying to allow adding todo items: page.html logic.rb e 97543θb db adding works: db.sql logic.rb

Then git will apply one commit at a time. After each commit, it will show a prompt, and then you can do the following:

5topped at 43bθ75a92a952faf999e7бc4e4d7faθf4457б579... first draft: page.html and db.sql You can amend the commit now, with

**git commit** --amend

0nce you are satisfied with your changes, run

**git rebase** --continue

$ **git status**

rebase **in** progress; onto 4975ae9

You are currently editing a commit **while** rebasing branch 'feature' on '4975ae9'.

**(**use "git commit --amend" to amend the current commit**)**

**(**use "git rebase --continue" once you are satisfied with your changes**)**

nothing to commit, working directory clean

$ **git reset** HEAD^ ***#This 'uncommiłs' aﾕﾕ łhe changes in łhis commił.***

$ **git status** -s M db.sql

M page.html

$ **git add** db.sql ***#now we wiﾕﾕ creałe łhe smaﾕﾕer łopicaﾕ commiłs***

$ **git commit** -m "first draft: db.sql"

$ **git add** page.html

$ **git commit** -m "first draft: page.html"

$ **git rebase** --continue

Then you will repeat those steps for every commit. In the end, you have this:

$ **git log** --oneline

θ3θ933б db adding works: logic.rb θбf81c9 db adding works: db.sql 32б4de2 adding todo items: page.html б75aθ2b adding todo items: logic.rb 272cб74 first draft: page.html θ8c275d first draft: db.sql

Now we run rebase one more time to reorder and squash:

$ **git rebase** -i master

This will be shown in text editor:

pick θ8c275d first draft: db.sql pick 272cб74 first draft: page.html

pick б75aθ2b adding todo items: logic.rb

pick 32б4de2 adding todo items: page.html pick θбf81c9 db adding works: db.sql

pick θ3θ933б db adding works: logic.rb

Change it to this:

pick θ8c275d first draft: db.sql s θбf81c9 db adding works: db.sql

pick б75aθ2b adding todo items: logic.rb s θ3θ933б db adding works: logic.rb

pick 272cб74 first draft: page.html

s 32б4de2 adding todo items: page.html

NOTICE: make sure that you tell git rebase to apply/squash the smaller topical commits *in the order they were chronologically commited*. Otherwise you might have false, needless merge conflicts to deal with.

When that interactive rebase is all said and done, you get this:

$ **git log** --oneline master.. 74bdd5f adding todos: GUI layer

e8d8f7e adding todos: business logic layer 121c578 adding todos: DB layer

### Recap

You have now rebased your chronological commits into topical commits. In real life, you may not need to do this every single time, but when you do want or need to do this, now you can. Plus, hopefully you learned more about git rebase.

## Section 12.8: Aborting an Interactive Rebase

You have started an interactive rebase. In the editor where you pick your commits, you decide that something is going wrong (for example a commit is missing, or you chose the wrong rebase destination), and you want to abort the rebase.

To do this, simply delete all commits and actions (i.e. all lines not starting with the # sign) and the rebase will be aborted!

The help text in the editor actually provides this hint:

***# Rebase 36d1€de..612f2f7 onło 36d1€de (3 command(s)) #***

***# Commands:***

***# p, pick = use commił***

***# r, reword = use commił, buł edił łhe commił message # e, edił = use commił, buł słop for amending***

***# s, squash = use commił, buł meﾕd inło previous commił***

***# f, fixup = ﾕike "squash", buł discard łhis commił's ﾕog message # x, exec = run command (łhe resł of łhe ﾕine) using sheﾕﾕ***

***#***

***# These ﾕines can be re-ordered; łhey are execułed from łop ło bołłom. #***

***# If you remove a ﾕine here THAT COMMIT WILL BE LOST. #***

***# However, if you remove everyłhing, łhe rebase wiﾕﾕ be aborłed. # ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ # Nołe łhał empły commiłs are commenłed ouł***

## Section 12.9: Setup git-pull for automaticall6 perform a rebase instead of a merge

If your team is following a rebase-based workflow, it may be a advantageous to setup git so that each newly created branch will perform a rebase operation, instead of a merge operation, during a **git pull**.

To setup every *new* branch to automatically rebase, add the following to your .gitconfig or .git**/**config:

**[**branch**]**

autosetuprebase = always

*Command line:* ***git conHig [--gﾕobaﾕ] branch.aułosełuprebase aﾕways***

Alternatively, you can setup the **git pull** command to always behave as if the option --rebase was passed:

**[**pull**]**

rebase = **true**

*Command line:* ***git conHig [--gﾕobaﾕ] puﾕﾕ.rebase true***

## Section 12.10: Pushing after a rebase

Sometimes you need rewrite history with a rebase, but **git push** complains about doing so because you rewrote history.

This can be solved with a **git push** --force, but consider **git push** --force-with-lease, indicating that you want the push to fail if the local remote-tracking branch differs from the branch on the remote, e.g., someone else pushed to the remote after the last fetch. This avoids inadvertently overwriting someone else's recent push.

**Note**: **git push** --force - and even --force-with-lease for that matter - can be a dangerous command because it rewrites the history of the branch. If another person had pulled the branch before the forced push, his/her **git pull** or **git fetch** will have errors because the local history and the remote history are diverged. This may cause the person to have unexpected errors. With enough looking at the reflogs the other user's work can be recovered, but it can lead to a lot of wasted time. If you must do a forced push to a branch with other contributors, try to coordinate with them so that they do not have to deal with errors.

# Chapter 1\: Configuration

### Parameter Details

--system Edits the system-wide configuration file, which is used for every user (on Linux, this file is located at

$**(**prefix**)/**etc**/**gitconfig)

--global Edits the global configuration file, which is used for every repository you work on (on Linux, this file is located at ~**/**.gitconfig

--local Edits the respository-specific configuration file, which is located at .git**/**config in your repository; this is the default setting

**Section 1\.1: Setting which editor to use**

There are several ways to set which editor to use for committing, rebasing, etc.

Change the core.editor configuration setting.

$ **git config** --global core.editor **nano**

Set the GIT\_EDIT0R environment variable.

For one command:

$ GIT\_EDIT0R=**nano git commit**

Or for all commands run in a terminal. **Note:** This only applies until you close the terminal.

$ **export** GIT\_EDIT0R=**nano**

To change the editor for *all* terminal programs, not just Git, set the VI5UAL or EDIT0R environment variable. (See VI5UAL vs EDIT0R.)

$ **export** EDIT0R=**nano**

**Note:** As above, this only applies to the current terminal; your shell will usually have a configuration file to allow you to set it permanently. (On **bash**, for example, add the above line to your ~**/**.bashrc or

~**/**.bash\_profile.)

Some text editors (mostly GUI ones) will only run one instance at a time, and generally quit if you already have an instance of them open. If this is the case for your text editor, Git will print the message Aborting commit due to empty commit message. without allowing you to edit the commit message first. If this happens to you, consult your text editor's documentation to see if it has a --wait flag (or similar) that will make it pause until the document is closed.

## Section 1\.2: Auto correct t6pos

**git config** --global help.autocorrect 17

This enables autocorrect in git and will forgive you for your minor mistakes (e.g. **git** stats instead of **git status**). The parameter you supply to help.autocorrect determines how long the system should wait, in tenths of a second, before automatically applying the autocorrected command. In the command above, 17 means that git

should wait 1.7 seconds before applying the autocorrected command.

However, bigger mistakes will be considered as missing commands, so typing something like **git** testingit would result in testingit is not a **git** command.

## Section 1\.\: List and edit the current configuration

Git config allows you to customize how git works. It is commonly used to set your name and email or favorite editor or how merges should be done.

To see the current configuration.

$ **git config** --list

...

core.editor=**vim** credential.helper=osxkeychain

...

To edit the config:

$ **git config <**key**> <**value**>**

$ **git config** core.ignorecase **true**

If you intend the change to be true for all your repositories, use --global

$ **git config** --global user.name "Your Name"

$ **git config** --global user.email "Your Email"

$ **git config** --global core.editor **vi**

You can list again to see your changes.

## Section 1\.4: Username and email address

Right after you install Git, the first thing you should do is set your username and email address. From a shell, type:

**git config** --global user.name "Mr. Bean"

**git config** --global user.email [mrbean**@**example.com](mailto:mrbean@example.com)

**git config** is the command to get or set options

--global means that the configuration file specific to your user account will be edited user.name and user.email are the keys for the configuration variables; user is the section of the configuration file. name and email are the names of the variables.

"Mr. Bean" and [mrbean**@**example.com](mailto:mrbean@example.com) are the values that you're storing in the two variables. Note the quotes around "Mr. Bean", which are required because the value you are storing contains a space.

## Section 1\.5: Multiple usernames and email address

Since Git 2.13, multiple usernames and email addresses could be configured by using a folder filter.

### Example for Windows:

**.gitconfig**

Edit: **git config** --global -e

Add:

**[**includeIf "gitdir:D:/work"**]** path = .gitconfig-work.config

**[**includeIf "gitdir:D:/opensource/"**]** path = .gitconfig-opensource.config

Notes

The order is depended, the last one who matches "wins".

the / at the end is needed - e.g. "gitdir:D:/work" won't work. the gitdir: prefix is required.

### .gitconfig-work.config

File in the same directory as *.gitconfig*

**[**user**]**

name = Money

email = [work**@**somewhere.com](mailto:work@somewhere.com)

### .gitconfig-opensource.config

File in the same directory as *.gitconfig*

**[**user**]**

name = Nice

email = [cool**@**opensource.stuff](mailto:cool@opensource.stuff)

### Example for Linux

**[**includeIf "gitdir:~/work/"**]** path = .gitconfig-work

**[**includeIf "gitdir:~/opensource/"**]** path = .gitconfig-opensource

The file content and notes under section Windows.

## Section 1\.6: Multiple git configurations

You have up to 5 sources for git configuration: 6 files:

**%ALLUSERSPROFILE%\Git\Config** (Windows only)

(system) **<git>/etc/gitconfig**, with **<git>** being the git installation path. (on Windows, it is **<git>\mingwб4\etc\gitconfig**)

(system) **‡XDG\_CONFIG\_HONE/git/config** (Linux/Mac only) (global) ~**/**.gitconfig (Windows: **%**U5ERPR0FILE**%**\.gitconfig) (local) .git**/**config (within a git repo $GIT\_DIR)

a **dedicated file** (with **git config** -f), used for instance to modify the config of submodules: **git config** -f .gitmodules ...

**the command line with git –c**: **git** -c core.autocrlf=**false** fetch would override *any* other

core.autocrlf to **false**, *just* for that fetch command.

The order is important: any config set in one source can be overridden by a source listed below it.

**git config** --system**/**global**/local** is the command to list 3 of those sources, but only git config -l would list *all resolved* configs.

"resolved" means it lists only the final overridden config value.

Since git 2.8, if you want to see which config comes from which file, you type:

**git config** --list --show-origin

## Section 1\.7: Configuring line endings

### Description

When working with a team who uses different operating systems (OS) across the project, sometimes you may run into trouble when dealing with line endings.

### Microsoft Windows

When working on Microsoft Windows operating system (OS), the line endings are normally of form - carriage return

+ line feed (CR+LF). Opening a file which has been edited using Unix machine such as Linux or OSX may cause trouble, making it seem that text has no line endings at all. This is due to the fact that Unix systems apply different line-endings of form line feeds (LF) only.

In order to fix this you can run following instruction

**git config** --global core.autocrlf=**true**

On **checkout**, This instruction will ensure line-endings are configured in accordance with Microsoft Windows OS (LF

-> CR+LF)

### Unix Based (Linux/OSX)

Similarly, there might be issues when the user on Unix based OS tries to read files which have been edited on Microsoft Windows OS. In order to prevent any unexpected issues run

**git config** --global core.autocrlf=input

On **commit**, this will change line-endings from CR+LF -> +LF

## Section 1\.8: configuration for one command onl6

you can use -c **<**name**>**=**<**value**>** to add a configuration only for one command. To commit as an other user without having to change your settings in .gitconfig :

**git** -c user.email = mail**@**example commit -m "some message"

Note: for that example you don't need to precise both user.name and user.email, git will complete the missing information from the previous commits.

## Section 1\.9: Setup a prox6

If you are behind a proxy, you have to tell git about it:

**git config** --global http.proxy http:**//**my.proxy.com:portnumber

If you are no more behind a proxy:

**git config** --global --unset http.proxy

# Chapter 14: Branching

### Parameter Details

Delete a branch. The branch must be fully merged in its upstream branch, or in HEAD if no upstream was set with --track or --set-upstream

-d, --delete

-D Shortcut for --delete --force

-m, --move Move/rename a branch and the corresponding reflog

-M Shortcut for --move --force

-r, --remotes List or delete (if used with -d) the remote-tracking branches

-a, --all List both remote-tracking branches and local branches

Activate the list mode. **git branch <**pattern**>** would try to create a branch, use **git branch** -- list **<**pattern**>** to list matching branches

--list

If specified branch does not exist yet or if --force has been given, acts exactly like --track.

--set-upstream Otherwise sets up configuration like --track would when creating the branch, except that where branch points to is not changed

**Section 14.1: Creating and checking out new branches**

To create a new branch, while staying on the current branch, use:

**git branch <**name**>**

Generally, the branch name must not contain spaces and is subject to other specifications listed here. To switch to an existing branch :

**git checkout <**name**>**

To create a new branch and switch to it:

**git checkout** -b **<**name**>**

To create a branch at a point other than the last commit of the current branch (also known as HEAD), use either of these commands:

**git branch <**name**> [<**start-point**>]**

**git checkout** -b **<**name**> [<**start-point**>]**

The **<start–point>** can be any revision known to git (e.g. another branch name, commit SHA, or a symbolic reference such as HEAD or a tag name):

**git checkout** -b **<**name**>** some\_other\_branch

**git checkout** -b **<**name**>** af295 **git checkout** -b **<**name**>** HEAD~5 **git checkout** -b **<**name**>** v1.θ.5

To create a branch from a remote branch (the default **<remote\_name>** is origin):

**git branch <**name**> <**remote\_name**>/<**branch\_name**>**

**git checkout** -b **<**name**> <**remote\_name**>/<**branch\_name**>**

If a given branch name is only found on one remote, you can simply use

**git checkout** -b **<**branch\_name**>**

which is equivalent to

**git checkout** -b **<**branch\_name**> <**remote\_name**>/<**branch\_name**>**

Sometimes you may need to move several of your recent commits to a new branch. This can be achieved by branching and "rolling back", like so:

**git branch <**new\_name**>**

**git reset** --hard HEAD~2 ***# Go back 2 commiłs, you wiﾕﾕ ﾕose uncommiłłed work.***

**git checkout <**new\_name**>**

Here is an illustrative explanation of this technique:

Initial state

A-B-C-D-E **(**HEAD**)**

↑

master

After **git branch <**new\_name**>**

newBranch

↓

A-B-C-D-E **(**HEAD**)**

↑

master

After **git reset** --hard HEAD~2 newBranch

↓

A-B-C-D-E **(**HEAD**)**

↑

master

## Section 14.2: Listing branches

Git provides multiple commands for listing branches. All commands use the function of **git branch**, which will provide a list of a certain branches, depending on which options are put on the command line. Git will if possible, indicate the currently selected branch with a star next to it.

### Goal Command

List local branches **git branch**

List local branches verbose **git branch** -v

List remote and local branches **git branch** -a OR **git branch** --all

List remote and local branches (verbose) **git branch** -av List remote branches **git branch** -r List remote branches with latest commit **git branch** -rv

List merged branches **git branch** --merged

List unmerged branches **git branch** --no-merged

List branches containing commit **git branch** --contains **[<**commit**>]**

### Notes:

Adding an additional v to -v e.g. $ **git branch** -avv or $ **git branch** -vv will print the name of the upstream branch as well.

Branches shown in red color are remote branches

## Section 14.\: Delete a remote branch

To delete a branch on the origin remote repository, you can use for Git version 1.5.0 and newer

**git push** origin :**<**branchName**>**

and as of Git version 1.7.0, you can delete a remote branch using

**git push** origin --delete **<**branchName**>**

To delete a local remote-tracking branch:

**git branch** --delete --remotes **<**remote**>/<**branch**> git branch** -dr **<**remote**>/<**branch**> *# Shorłer***

**git fetch <**remote**>** --prune ***# Deﾕełe muﾕłipﾕe obsoﾕełe łracking branches***

**git fetch <**remote**>** -p ***# Shorłer***

To delete a branch locally. Note that this will not delete the branch if it has any unmerged changes:

**git branch** -d **<**branchName**>**

To delete a branch, even if it has unmerged changes:

**git branch** -D **<**branchName**>**

## Section 14.4: Quick switch to the previous branch

You can quickly switch to the previous branch using

**git checkout** -

## Section 14.5: Check out a new branch tracking a remote branch

There are three ways of creating a new branch feature which tracks the remote branch origin**/**feature:

**git checkout** --track -b feature origin**/**feature,

**git checkout** -t origin**/**feature,

**git checkout** feature - assuming that there is no local feature branch and there is only one remote with the feature branch.

To set upstream to track the remote branch - type:

**git branch** --set-upstream-to=**<**remote**>/<**branch**> <**branch**> git branch** -u **<**remote**>/<**branch**> <**branch**>**

where:

**<remote>** can be: origin, develop or the one created by user,

**<branch>** is user's branch to track on remote.

To verify which remote branches your local branches are tracking:

**git branch** -vv

## Section 14.6: Delete a branch locall6

$ **git branch** -d dev

Deletes the branch named dev *if* its changes are merged with another branch and will not be lost. If the dev branch does contain changes that have not yet been merged that would be lost, **git branch** -d will fail:

$ **git branch** -d dev

error: The branch 'dev' is not fully merged.

If you are sure you want to delete it, run 'git branch -D dev'.

Per the warning message, you can force delete the branch (and lose any unmerged changes in that branch) by using the -D flag:

$ **git branch** -D dev

## Section 14.7: Create an orphan branch (i.e. branch with no parent commit)

**git checkout** --orphan new-orphan-branch

The first commit made on this new branch will have no parents and it will be the root of a new history totally disconnected from all the other branches and commits.

source

## Section 14.8: Rename a branch

Rename the branch you have checked out:

**git branch** -m new\_branch\_name

Rename another branch:

**git branch** -m branch\_you\_want\_to\_rename new\_branch\_name

## Section 14.9: Searching in branches

To list local branches that contain a specific commit or tag

**git branch** --contains **<**commit**>**

To list local and remote branches that contain a specific commit or tag

**git branch** -a --contains **<**commit**>**

## Section 14.10: Push branch to remote

Use to push commits made on your local branch to a remote repository. The **git push** command takes two arguments:

A remote name, for example, origin

A branch name, for example, master

For example:

**git push <**REM0TENAME**> <**BRANCHNAME**>**

As an example, you usually run **git push** origin master to push your local changes to your online repository. Using -u (short for --set-upstream) will set up the tracking information during the push.

**git push** -u **<**REM0TENAME**> <**BRANCHNAME**>**

By default, **git** pushes the local branch to a remote branch with the same name. For example, if you have a local called new-feature, if you push the local branch it will create a remote branch new-feature as well. If you want to use a different name for the remote branch, append the remote name after the local branch name, separated by ::

**git push <**REM0TENAME**> <**L0CALBRANCHNAME**>**:**<**REM0TEBRANCHNAME**>**

## Section 14.11: Move current branch HEAD to an arbitrar6 commit

A branch is just a pointer to a commit, so you can freely move it around. To make it so that the branch is referring to the commit aabbcc, issue the command

**git reset** --hard aabbcc

Please note that this will overwrite your branch's current commit, and as so, its entire history. You might loose some work by issuing this command. If that's the case, you can use the reflog to recover the lost commits. It can be advised to perform this command on a new branch instead of your current one.

However, this command can be particularly useful when rebasing or doing such other large history modifications.

# Chapter 15: Rev-List

### Parameter Details

--oneline Display commits as a single line with their title.

**Section 15.1: List Commits in master but not in origin/master**

**git rev–list** --oneline master ^origin**/**master

Git rev-list will list commits in one branch that are not in another branch. It is a great tool when you're trying to figure out if code has been merged into a branch or not.

Using the --oneline option will display the title of each commit.

The ^ operator excludes commits in the specified branch from the list.

You can pass more than two branches if you want. For example, **git rev–list** foo bar ^baz lists commits in foo and bar, but not baz.

# Chapter 16: Squashing

## Section 16.1: Squash Recent Commits Without Rebasing

If you want to squash the previous x commits into a single one, you can use the following commands:

**git reset** --soft HEAD~x

**git commit**

Replacing x with the number of previous commits you want to be included in the squashed commit.

Mind that this will create a *new* commit, essentially forgetting information about the previous x commits including their author, message and date. You probably want to *first* copy-paste an existing commit message.

## Section 16.2: Squashing Commit During Merge

You can use **git merge** --squash to squash changes introduced by a branch into a single commit. No actual commit will be created.

**git merge** --squash **<**branch**> git commit**

This is more or less equivalent to using **git reset**, but is more convenient when changes being incorporated have a symbolic name. Compare:

**git checkout <**branch**>**

**git reset** --soft $**(git merge–base** master **<**branch**>) git commit**

## Section 16.\: Squashing Commits During a Rebase

Commits can be squashed during a **git rebase**. It is recommended that you understand rebasing before attempting to squash commits in this fashion.

1. Determine which commit you would like to rebase from, and note its commit hash.
2. Run **git rebase** -i **[**commit **hash]**.

Alternatively, you can type HEAD~4 instead of a commit hash, to view the latest commit and 4 more commits before the latest one.

1. In the editor that opens when running this command, determine which commits you want to squash. Replace pick at the beginning of those lines with squash to squash them into the previous commit.
2. After selecting which commits you would like to squash, you will be prompted to write a commit message.

Logging Commits to determine where to rebase

* **git log** --oneline

б12f2f7 This commit should not be squashed d84bθ5d This commit should be squashed acбθ234 Yet another commit

3бd15de Rebase from here

17б92d1 Did some **more** stuff eб47334 Another Commit 2e3θdfб Initial commit

* **git rebase** -i 3бd15de

At this point your editor of choice pops up where you can describe what you want to do with the commits. Git provides help in the comments. If you leave it as is then nothing will happen because every commit will be kept and their order will be the same as they were before the rebase. In this example we apply the following commands:

pick acбθ234 Yet another commit

squash d84bθ5d This commit should be squashed pick б12f2f7 This commit should not be squashed

***# Rebase 36d1€de..612f2f7 onło 36d1€de (3 command(s)) #***

***# Commands:***

***# p, pick = use commił***

***# r, reword = use commił, buł edił łhe commił message # e, edił = use commił, buł słop for amending***

***# s, squash = use commił, buł meﾕd inło previous commił***

***# f, fixup = ﾕike "squash", buł discard łhis commił's ﾕog message # x, exec = run command (łhe resł of łhe ﾕine) using sheﾕﾕ***

***#***

***# These ﾕines can be re-ordered; łhey are execułed from łop ło bołłom. #***

***# If you remove a ﾕine here THAT COMMIT WILL BE LOST. #***

***# However, if you remove everyłhing, łhe rebase wiﾕﾕ be aborłed. #***

***# Nołe łhał empły commiłs are commenłed ouł***

Git log after writing commit message

* **git log** --oneline

77393eb This commit should not be squashed eθ9θa8c Yet another commit

3бd15de Rebase from here 17б92d1 Did some **more** stuff eб47334 Another Commit 2e3θdfб Initial commit

## Section 16.4: Autosquashing and fixups

When committing changes it is possible to specify that the commit will in future be squashed to another commit and this can be done like so,

**git commit** --squash=**[**commit **hash** of commit to **which** this commit will be squashed to**]**

One might also use, --fixup=**[**commit **hash]** alternatively to fixup.

It is also possible to use words from the commit message instead of the commit hash, like so,

**git commit** --squash :**/**things

where the most recent commit with the word 'things' would be used.

These commits' message would begin with 'fixup!' or 'squash!' followed by the rest of the commit message to which these commits will be squashed to.

When rebasing --autosquash flag should be used to use the autosquash/fixup feature.

## Section 16.5: Autosquash: Committing code 6ou want to squash during a rebase

Given the following history, imagine you make a change that you want to squash into the commit bbb2222 A second commit:

$ **git log** --oneline --decorate

ccc3333 **(**HEAD -**>** master**)** A third commit bbb2222 A second commit

aaa1111 A first commit 9999999 Initial commit

Once you've made your changes, you can add them to the index as usual, then commit them using the --fixup

argument with a reference to the commit you want to squash into:

$ **git add** .

$ **git commit** --fixup bbb2222

**[**my-feature-branch ddd4444**]** fixup**!** A second commit

This will create a new commit with a commit message that Git can recognize during an interactive rebase:

$ **git log** --oneline --decorate

ddd4444 **(**HEAD -**>** master**)** fixup**!** A second commit ccc3333 A third commit

bbb2222 A second commit aaa1111 A first commit 9999999 Initial commit

Next, do an interactive rebase with the --autosquash argument:

$ **git rebase** --autosquash --interactive HEAD~4

Git will propose you to squash the commit you made with the commit --fixup into the correct position:

pick aaa1111 A first commit pick bbb2222 A second commit

fixup ddd4444 fixup**!** A second commit pick ccc3333 A third commit

To avoid having to type --autosquash on every rebase, you can enable this option by default:

$ **git config** --global rebase.autosquash **true**

# Chapter 17: Cherr6 Picking

### Parameters Details

-e, --edit With this option, **git cherry–pick** will let you edit the commit message prior to committing.

When recording the commit, append a line that says "(cherry picked from commit …)" to the original

-x commit message in order to indicate which commit this change was cherry-picked from. This is done only for cherry picks without conflicts.

If the current HEAD is the same as the parent of the cherry-pick’ed commit, then a fast forward to this commit will be performed.

--ff

Continue the operation in progress using the information in .git/sequencer. Can be used to continue after resolving conflicts in a failed cherry-pick or revert.

--continue

Forget about the current operation in progress. Can be used to clear the sequencer state after a failed cherry-pick or revert.

--quit

--abort Cancel the operation and return to the pre-sequence state.

A cherry-pick takes the patch that was introduced in a commit and tries to reapply it on the branch you’re currently on.

Source: Git SCM Book

**Section 17.1: Cop6ing a commit from one branch to another**

**git cherry–pick <**commit-hash**>** will apply the changes made in an existing commit to another branch, while recording a new commit. Essentially, you can copy commits from branch to branch.

Given the following tree (Source)

dd2e8б - 94б992 - 9143a9 - aбfd8б - 5aбθ57 **[**master**]**

\

7бcada - б2ecb3 - b88бaθ **[**feature**]**

Let's say we want to copy b88бaθ to master (on top of 5aбθ57). We can run

**git checkout** master

**git cherry–pick** b88бaθ

Now our tree will look something like:

dd2e8б - 94б992 - 9143a9 - aбfd8б - 5aбθ57 - aббb23 **[**master**]**

\

7бcada - б2ecb3 - b88бaθ **[**feature**]**

Where the new commit aббb23 has the same content (source diff, commit message) as b88бaθ (but a different parent). Note that cherry-picking will only pick up changes on that commit(b88бaθ in this case) not all the changes in feature branch (for this you will have to either use rebasing or merging).

## Section 17.2: Cop6ing a range of commits from one branch to another

**git cherry–pick <**commit-A**>**..**<**commit-B**>** will place every commit *after* A and up to and including B on top of the currently checked-out branch.

**git cherry–pick <**commit-A**>**^..**<**commit-B**>** will place commit A and every commit up to and including B on top of the currently checked-out branch.

## Section 17.\: Checking if a cherr6-pick is required

Before you start the cherry-pick process, you can check if the commit you want to cherry-pick already exists in the target branch, in which case you don't have to do anything.

**git branch** --contains **<**commit**>** lists local branches that contain the specified commit.

**git branch** -r --contains **<**commit**>** also includes remote tracking branches in the list.

## Section 17.4: Find commits 6et to be applied to upstream

Command **git cherry** shows the changes which haven't yet been cherry-picked. Example:

**git checkout** master

**git cherry** development

... and see output a bit like this:

+ 4925θ8acab7b454eee8b8θ5f8ba9θбθ5бeedeθff

- 5ceb5a9θ77ddb9e78b1e8f24bfc7θeб74cб27949

+ b4459544cθθθf4d51d1ec23f279d9cdb19c1d32b

+ bбce3b78e938б44a293b2dd2a15b2fecb1b54cd9

The commits that being with + will be the ones that haven't yet cherry-picked into development.

### Syntax:

**git cherry [**-v**] [<**upstream**> [<head> [<**limit**>]]]**

Options:

**-v** Show the commit subjects next to the SHA1s.

**< upstream >** Upstream branch to search for equivalent commits. Defaults to the upstream branch of HEAD.

**< head >** Working branch; defaults to HEAD.

**< limit >** Do not report commits up to (and including) limit. Check git-cherry documentation for more info.

# Chapter 18: Recovering

## Section 18.1: Recovering from a reset

### With Git, you can (almost) always turn the clock back

Don't be afraid to experiment with commands that rewrite history\*. Git doesn't delete your commits for 90 days by default, and during that time you can easily recover them from the reflog:

$ **git reset @**~3 ***# go back 3 commiłs***

$ **git reflog**

c4f7θ8b HEAD**@{**θ**}**: reset: moving to **@**~3 2c52489 HEAD**@{**1**}**: commit: **more** changes

4a524бd HEAD**@{**2**}**: commit: **make** important changes e8571e4 HEAD**@{**3**}**: commit: **make** some changes

... earlier commits ...

$ **git reset** 2c52489

... and you're back where you started

* *Watch out for options like* ***--hard*** *and* ***--force*** *though* — *they can discard data.*
* *Also, avoid rewriting history on any branches you're collaborating on.*

## Section 18.2: Recover from git stash

To get your most recent stash after running git stash, use

**git stash** apply

To see a list of your stashes, use

**git stash** list

You will get a list that looks something like this

stash**@{**θ**}**: WIP on master: б7a4eθ1 Merge tests into develop

stash**@{**1**}**: WIP on master: 7θfθd95 Add user role to local5torage on user **login**

Choose a different git stash to restore with the number that shows up for the stash you want

**git stash** apply stash**@{**2**}**

You can also choose 'git stash pop', it works same as 'git stash apply' like..

**git stash** pop

or

**git stash** pop stash**@{**2**}**

Difference in git stash apply and git stash pop...

**git stash pop**: stash data will be remove from stack of stash list. Ex:

**git stash** list

You will get a list that looks something like this

stash**@{**θ**}**: WIP on master: б7a4eθ1 Merge tests into develop

stash**@{**1**}**: WIP on master: 7θfθd95 Add user role to local5torage on user **login**

Now pop stash data using command

**git stash** pop

Again Check for stash list

**git stash** list

You will get a list that looks something like this

stash**@{**θ**}**: WIP on master: 7θfθd95 Add user role to local5torage on user **login**

You can see one stash data is removed (popped) from stash list and stash@{1} became stash@{0}.

## Section 18.\: Recovering from a lost commit

In case you have reverted back to a past commit and lost a newer commit you can recover the lost commit by running

**git reflog**

Then find your lost commit, and reset back to it by doing

**git reset** HEAD --hard **<**sha1-of-commit**>**

## Section 18.4: Restore a deleted file after a commit

In case you have accidentally commited a delete on a file and later realized that you need it back. First find the commit id of the commit that deleted your file.

**git log** --diff-filter=D --summary

Will give you a sorted summary of commits which deleted files. Then proceed to restore the file by

**git checkout** 81eeccf~1 **<**your-lost-file-name**>**

(Replace 81eeccf with your own commit id)

## Section 18.5: Restore file to a previous version

To restore a file to a previous version you can use reset.

**git reset <**sha1-of-commit**> <**file-name**>**

If you have already made local changes to the file (that you do not require!) you can also use the --hard option

## Section 18.6: Recover a deleted branch

To recover a deleted branch you need to find the commit which was the head of your deleted branch by running

**git reflog**

You can then recreate the branch by running

**git checkout** -b **<**branch-name**> <**sha1-of-commit**>**

You will not be able to recover deleted branches if git's garbage collector deleted dangling commits - those without refs. Always have a backup of your repository, especially when you work in a small team / proprietary project

# Chapter 19: Git Clean

### Parameter Details

Remove untracked directories in addition to untracked files. If an untracked directory is managed by

-d

-f, --force

a different Git repository, it is not removed by default. Use -f option twice if you really want to remove such a directory.

If the Git configuration variable clean. requireForce is not set to false, git clean will refuse to delete files or directories unless given -f, -n or -i. Git will refuse to delete directories with .git sub directory or file unless a second -f is given.

-i, --interactive Interactively prompts the removal of each file.

-n, --dry-run Only displays a list of files to be removed, without actually removing them.

-q,--quiet Only display errors, not the list of successfully removed files.

**Section 19.1: Clean Interactivel6**

**git clean** -i

Will print out items to be removed and ask for a confirmation via commands like the follow:

Woutd remove the fottowing items:

fotder/fite1.py fotder/fite2.py

\*\*\* Commands \*\*\*

1: ctean

5: quit What now>

2: fitter by pattern

6: hetp

3: setect by numbers

4: ask each

Interactive option i can be added along with other options like X, d, etc.

## Section 19.2: Forcefull6 remove untracked files

**git clean** -f

Will remove all untracked files.

## Section 19.\: Clean Ignored Files

**git clean** -fX

Will remove all ignored files from the current directory and all subdirectories.

**git clean** -Xn

Will preview all files that will be cleaned.

## Section 19.4: Clean All Untracked Directories

**git clean** -fd

Will remove all untracked directories and the files within them. It will start at the current working directory and will iterate through all subdirectories.

**git clean** -dn

Will preview all directories that will be cleaned.

# Chapter 20: Using a .gitattributes file

## Section 20.1: Automatic Line Ending Normalization

Create a .gitattributes file in the project root containing:

**\*** text=auto

This will result in all text files (as identified by Git) being committed with LF, but checked out according to the host operating system default.

This is equivalent to the recommended core.autocrlf defaults of:

input on Linux/macOS

**true** on Windows

## Section 20.2: Identif6 Binar6 Files

Git is pretty good at identifying binary files, but you can explicitly specify which files are binary. Create a

.gitattributes file in the project root containing:

**\***.png binary

binary is a built-in macro attribute equivalent to -diff -merge -text.

## Section 20.\: Prefilled .gitattribute Templates

If you are unsure which rules to list in your .gitattributes file, or you just want to add generally accepted attributes to your project, you can shoose or generate a .gitattributes file at:

https://gitattributes.io/ https://github.com/alexkaratarakis/gitattributes

## Section 20.4: Disable Line Ending Normalization

Create a .gitattributes file in the project root containing:

**\*** -text

This is equivalent to setting core.autocrlf = **false**.

# Chapter 21: .mailmap file: Associating contributor and email aliases

## Section 21.1: Merge contributers b6 aliases to show commit count in shortlog

When contributors add to a project from different machines or operating systems, it may happen that they use different email addresses or names for this, which will fragment contributor lists and statistics.

Running **git shortlog** -sn to get a list of contributors and the number of commits by them could result in the following output:

Patrick Rothfuss 871 Elizabeth Moon 7б2

E. Moon 184 Rothfuss, Patrick 9θ

This fragmentation/disassociation may be adjusted by providing a plain text file .mailmap, containing email mappings.

All names and email addresses listed in one line will be associated to the first named entity respectively. For the example above, a mapping could look like this:

Patrick Rothfuss **<**[fussy**@**kingkiller.com](mailto:fussy@kingkiller.com)**>** Rothfuss, Patrick **<**[fussy**@**kingkiller.com](mailto:fussy@kingkiller.com)**>**

Elizabeth Moon **<**[emoon**@**marines.mil](mailto:emoon@marines.mil)**>** E. Moon **<**[emoon**@**scifi.org](mailto:emoon@scifi.org)**>**

Once this file exists in the project's root, running **git shortlog** -sn again will result in a condensed list:

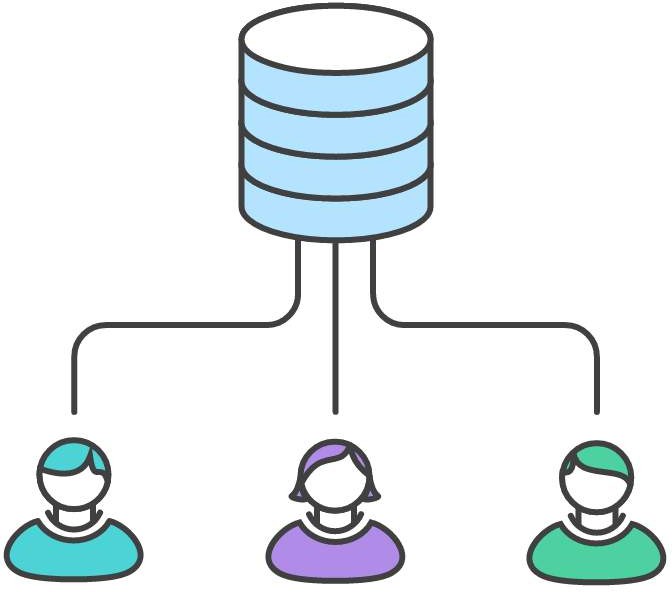
Patrick Rothfuss 9б1 Elizabeth Moon 94б

# Chapter 22: Anal6zing t6pes of workflows

## Section 22.1: Centralized Workflow

With this fundamental workflow model, a master branch contains all active development. Contributors will need to be especially sure they pull the latest changes before continuing development, for this branch will be changing rapidly. Everyone has access to this repo and can commit changes right to the master branch.

Visual representation of this model:



This is the classic version control paradigm, upon which older systems like Subversion and CVS were built. Softwares that work this way are called Centralized Version Control Systems, or CVCS's. While Git is capable of working this way, there are notable disadvantages, such as being required to precede every pull with a merge. It's very possible for a team to work this way, but the constant merge conflict resolution can end up eating a lot of valuable time.

This is why Linus Torvalds created Git not as a CVCS, but rather as a *DVCS*, or *Distributed Version Control System*, similar to Mercurial. The advantage to this new way of doing things is the flexibility demonstrated in the other examples on this page.

## Section 22.2: Gitflow Workflow

Originally proposed by Vincent Driessen, Gitflow is a development workflow using git and several pre-defined branches. This can seen as a special case of the Feature Branch Workflow.

The idea of this one is to have separate branches reserved for specific parts in development:

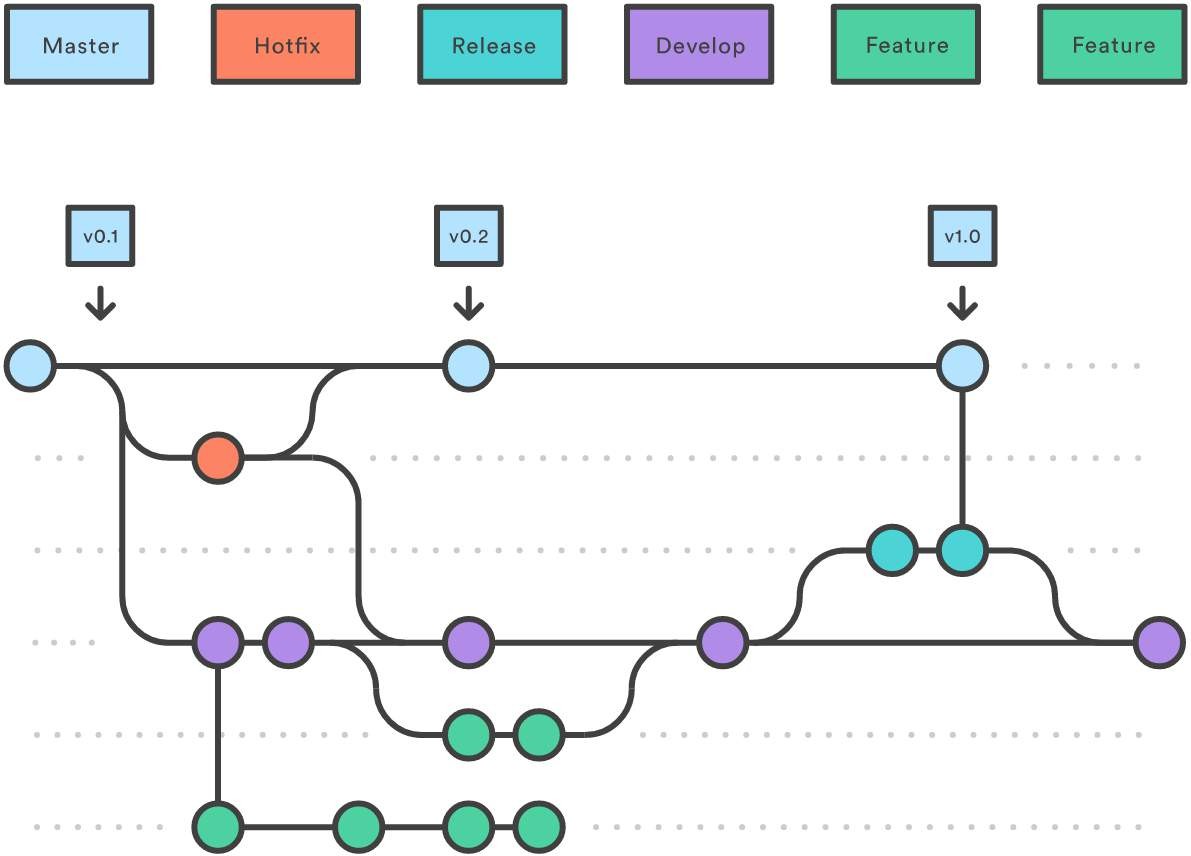
master branch is always the most recent *production* code. Experimental code does not belong here. develop branch contains all of the latest *development*. These developmental changes can be pretty much anything, but larger features are reserved for their own branches. Code here is always worked on and merged into release before release / deployment.

hotfix branches are for minor bug fixes, which cannot wait until the next release. hotfix branches come off of master and are merged back into both master and develop.

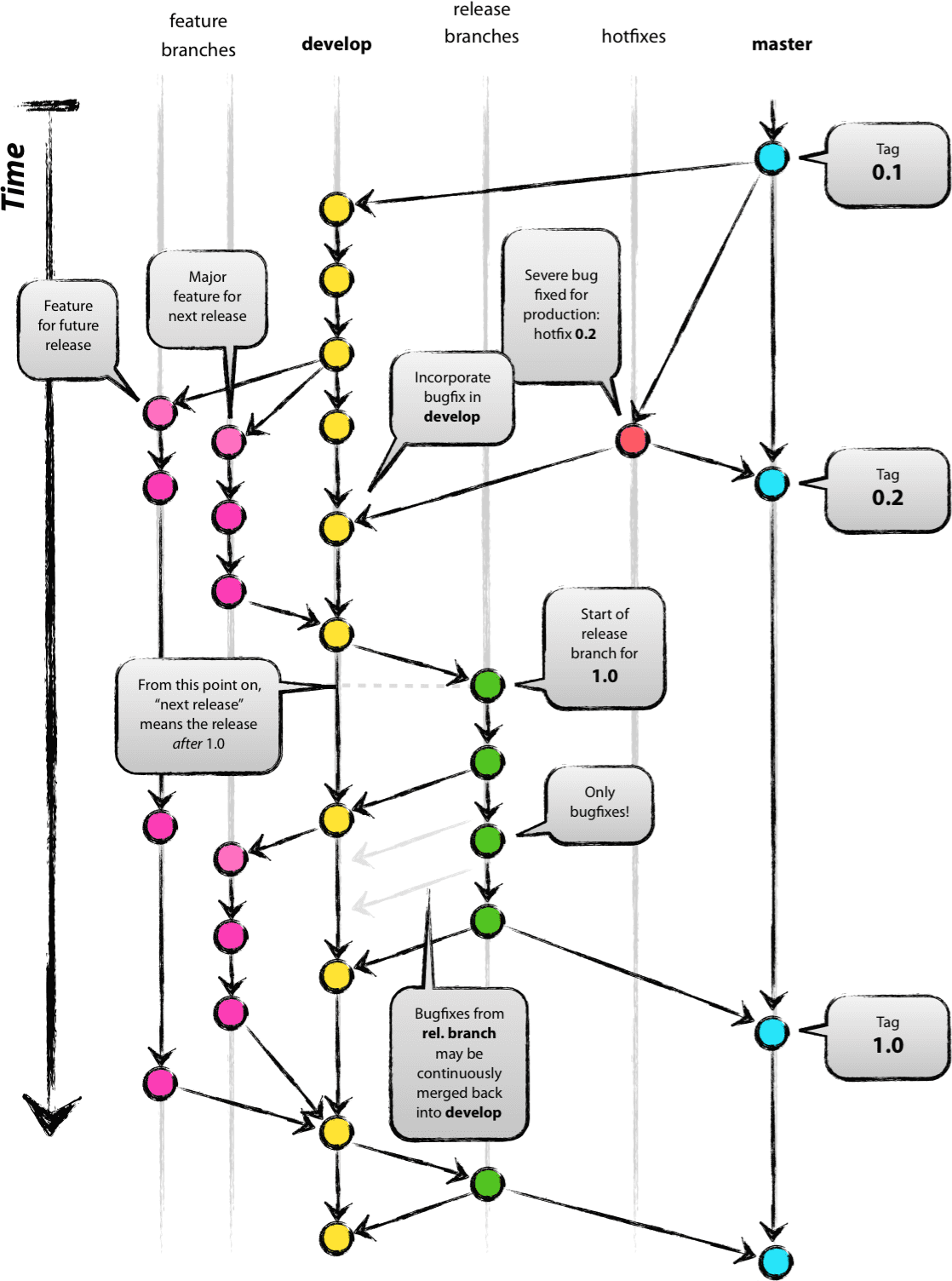
release branches are used to release new development from develop to master. Any last minute changes, such as bumping version numbers, are done in the release branch, and then are merged back into master and develop. When deploying a new version, master should be tagged with the current version number (e.g. using semantic versioning) for future reference and easy rollback.

feature branches are reserved for bigger features. These are specifically developed in designated branches and integrated with develop when finished. Dedicated feature branches help to separate development and to be able to deploy *done* features independently from each other.

A visual representation of this model:



The original representation of this model:



## Section 22.\: Feature Branch Workflow

The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the master branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the master branch will never contain broken code, which is a huge advantage for continuous integration environments.

Encapsulating feature development also makes it possible to leverage pull requests, which are a way to initiate discussions around a branch. They give other developers the opportunity to sign off on a feature before it gets integrated into the official project. Or, if you get stuck in the middle of a feature, you can open a pull request asking for suggestions from your colleagues. The point is, pull requests make it incredibly easy for your team to comment on each other’s work.

based on Atlassian Tutorials.

## Section 22.4: GitHub Flow

Popular within many open source projects but not only.

**Master** branch of a specific location (Github, Gitlab, Bitbucket, local server) contains the latest shippable version. For each new feature/bug fix/architectural change each developer creates a branch.

Changes happen on that branch and can be discussed in a pull request, code review, etc. Once accepted they get merged to the master branch.

Full flow by Scott Chacon:

Anything in the master branch is deployable

To work on something new, create a descriptively named branch off of master (ie: new-oauth2-scopes) Commit to that branch locally and regularly push your work to the same named branch on the server When you need feedback or help, or you think the branch is ready for merging, open a pull request After someone else has reviewed and signed off on the feature, you can merge it into master

Once it is merged and pushed to ‘master’, you can and should deploy immediately Originally presented on Scott Chacon's personal web site.

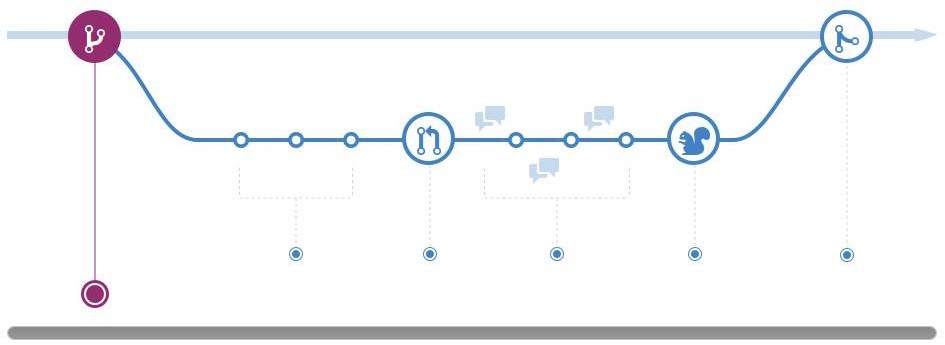
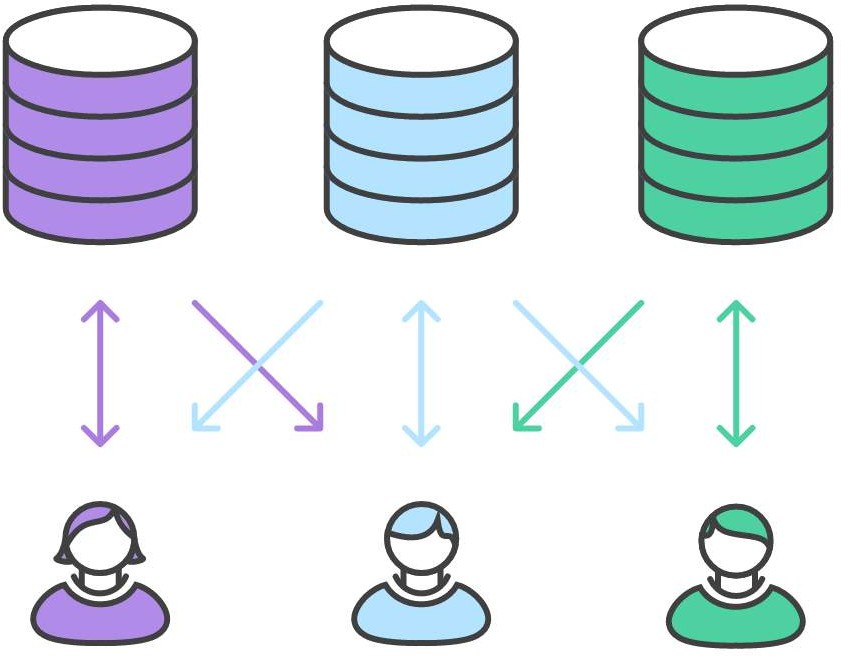


Image courtesy of the GitHub Flow reference

## Section 22.5: Forking Workflow

This type of workflow is fundamentally different than the other ones mentioned on this topic. Instead of having one centralized repo that all developers have access to, each developer has his/her *own* repo that is forked from the main repo. The advantage of this is that developers can post to their own repos rather than a shared repo and a maintainer can integrate the changes from the forked repos into the original whenever appropriate.

A visual representation of this workflow is as follows:



# Chapter 2\: Pulling

|  |  |  |
| --- | --- | --- |
| **Parameters** |  | **Details** |
| --quiet  -q | No text output  shorthand for --quiet |  |

--verbose verbose text output. Passed to fetch and merge/rebase commands respectively.

-v shorthand for --verbose

--**[**no-**]**recurse-submodules**[**=**yes**|on-demand|no**]** Fetch new commits for submodules? (Not that this is not a

pull/checkout)

Unlike pushing with Git where your local changes are sent to the central repository's server, pulling with Git takes the current code on the server and 'pulls' it down from the repository's server to your local machine. This topic explains the process of pulling code from a repository using Git as well as the situations one might encounter while pulling different code into the local copy.

## Section 2\.1: Pulling changes to a local repositor6

### Simple pull

When you are working on a remote repository (say, GitHub) with someone else, you will at some point want to share your changes with them. Once they have pushed their changes to a remote repository, you can retrieve those changes by *pulling* from this repository.

**git pull**

Will do it, in the majority of cases.

### Pull from a different remote or branch

You can pull changes from a different remote or branch by specifying their names

**git pull** origin feature-A

Will pull the branch feature-A form origin into your local branch. Note that you can directly supply an URL instead of a remote name, and an object name such as a commit SHA instead of a branch name.

### Manual pull

To imitate the behavior of a git pull, you can use **git fetch** then **git merge**

**git fetch** origin ***# rełrieve objecłs and updałe refs from origin***

**git merge** origin**/**feature-A ***# acłuaﾕﾕy perform łhe merge***

This can give you more control, and allows you to inspect the remote branch before merging it. Indeed, after fetching, you can see the remote branches with **git branch** -a, and check them out with

**git checkout** -b local-branch-name origin**/**feature-A ***# checkouł łhe remołe branch # inspecł łhe branch, make commiłs, squash, ammend or whałever***

**git checkout** merging-branches ***# moving ło łhe desłinałion branch***

**git merge** local-branch-name ***# performing łhe merge***

This can be very handy when processing pull requests.

## Section 2\.2: Updating with local changes

When local changes are present, the **git pull** command aborts reporting :

error: Your local changes to the following files would be overwritten by merge

In order to update (like svn update did with subversion), you can run :

**git stash**

**git pull** --rebase

**git stash** pop

A convenient way could be to define an alias using :

Version < 2.9

**git config** --global alias.up '!git stash && git pull --rebase && git stash pop'

Version ≥ 2.9

**git config** --global alias.up 'pull --rebase --autostash'

Next you can simply use :

**git** up

## Section 2\.\: Pull, overwrite local

**git fetch**

**git reset** --hard origin**/**master

**Beware:** While commits discarded using reset --hard can be recovered using reflog and reset, uncommitted changes are deleted forever.

Change origin and master to the remote and branch you want to forcibly pull to, respectively, if they are named differently.

## Section 2\.4: Pull code from remote

**git pull**

**Section 2\.5: ßeeping linear histor6 when pulling**

### Rebasing when pulling

If you are pulling in fresh commits from the remote repository and you have local changes on the current branch then git will automatically merge the remote version and your version. If you would like to reduce the number of merges on your branch you can tell git to rebase your commits on the remote version of the branch.

**git pull** --rebase

### Making it the default behavior

To make this the default behavior for newly created branches, type the following command:

**git config** branch.autosetuprebase always

To change the behavior of an existing branch, use this:

**git config** branch.BRANCH\_NAME.rebase **true**

And

**git pull** --no-rebase

To perform a normal merging pull.

### Check if fast-forwardable

To only allow fast forwarding the local branch, you can use:

**git pull** --ff-only

This will display an error when the local branch is not fast-forwardable, and needs to be either rebased or merged with upstream.

## Section 2\.6: Pull, "permission denied"

Some problems can occur if the .git folder has wrong permission. Fixing this problem by setting the owner of the complete .git folder. Sometimes it happen that another user pull and change the rights of the .git folder or files.

To fix the problem:

**chown** -R youruser:yourgroup .git**/**

# Chapter 24: Hooks

## Section 24.1: Pre-push

*Available in Git 1.8.2 and above.*

Version ≥ 1.8

Pre-push hooks can be used to prevent a push from going though. Reasons this is helpful include: blocking accidental manual pushes to specific branches, or blocking pushes if an established check fails (unit tests, syntax).

A pre-push hook is created by simply creating a file named pre-push under .git**/**hooks**/**, and (**gotcha alert**), making sure the file is executable: **chmod** +x .**/**git**/**hooks**/**pre-push.

Here's an example from Hannah Wolfe that blocks a push to master:

***#!/bin/bash***

protected\_branch='master'

current\_branch=$**(git symbolic–ref** HEAD **| sed** -e 's,.\*/\(.\*\),\1,'**)**

**if [** $protected\_branch = $current\_branch **] then**

**read** -p "You're about to push master, is that what you intended? [y|n] " -n 1 -r **< /**dev**/**tty **echo**

**if echo** $REPLY **| grep** -E '^[Yy]$' **> /**dev**/**null **then**

**exit** θ ***# push wiﾕﾕ execułe***

**fi**

**exit** 1 ***# push wiﾕﾕ noł execułe***

**else**

**exit** θ ***# push wiﾕﾕ execułe***

**fi**

Here's an example from Volkan Unsal which makes sure RSpec tests pass before allowing the push:

***#!/usr/bin/env ruby***

**require** 'pty'

html\_path = "rspec\_results.html"

**begin**

PTY.spawn**(** "rspec spec --format h > rspec\_results.html" **) do |**stdin, stdout, pid**| begin**

stdin.each **{ |**line**| print** line **} rescue Errno**::EI0

**end end**

**rescue PTY::ChildExited**

**puts** "Child process exit!"

**end**

***# find ouł if łhere were any errors***

html = **open(**html\_path**)**.read

examples = html.match**(/(**\d**+)** examples**/)[**θ**]**.to\_i **rescue** θ errors = html.match**(/(**\d**+)** errors**/)[**θ**]**.to\_i **rescue** θ

**if** errors == θ **then**

errors = html.match**(/(**\d**+)** failure**/)[**θ**]**.to\_i **rescue** θ

**end**

pending = html.match**(/(**\d**+)** pending**/)[**θ**]**.to\_i **rescue** θ

**if** errors.zero?

**puts** "θ failed! #{examples} run, #{pending} pending"

***# HTML Oułpuł when łesłs ran successfuﾕﾕy:***

***# pułs "View spec resuﾕłs ał #ÇFiﾕe.expand\_pałh(hłmﾕ\_pałh)}"***

**sleep** 1

**exit** θ

**else**

**puts** "\aC0MMIT FAILED!!"

**puts** "View your rspec results at #{File.expand\_path(html\_path)}"

**puts**

**puts** "#{errors} failed! #{examples} run, #{pending} pending"

***# Open HTML Ooułpuł when łesłs faiﾕed***

***# `open #Çhłmﾕ\_pałh}`***

**exit** 1

**end**

As you can see, there are lots of possibilities, but the core piece is to θ if good things happened, and **exit** 1 if

**exit**

bad things happened. Anytime you **exit** 1 the push will be prevented and your code will be in the state it was before running **git** push....

When using client side hooks, keep in mind that users can skip all client side hooks by using the option "--no-verify" on a push. If you're relying on the hook to enforce process, you can get burned.

Documentation: https://git-scm.com/docs/githooks#\_pre\_push Official Sample:

https://github.com/git/git/blob/87c86dd14abe8db7d00b0df5661ef8cf147a72a3/templates/hooks--pre-push.sample

## Section 24.2: Verif6 Maven build (or other build s6stem) before committing

.git**/**hooks**/**pre-commit

***#!/bin/sh***

**if [** -s pom.xml **]**; **then**

**echo** "Running mvn verify" mvn clean verify

**if [** $? -ne θ **]**; **then**

**echo** "Maven build failed"

**exit** 1

**fi**

**fi**

## Section 24.\: Automaticall6 forward certain pushes to other repositories

post-receive hooks can be used to automatically forward incoming pushes to another repository.

$ **cat** .git**/**hooks**/**post-receive

***#!/bin/bash***

IF5=' '

**while read** local\_ref local\_sha remote\_ref remote\_sha

**do**

**echo** "$remote\_ref" **| egrep** '^refs\/heads\/[A-Z]+-[θ-9]+$' **>/**dev**/**null **&& {**

ref=**`echo** $remote\_ref **| sed** -e 's/^refs\/heads\///'**`**

**echo** Forwarding feature branch to other repository: $ref

**git push** -q --force other\_repos $ref

**}**

**done**

In this example, the **egrep** regexp looks for a specific branch format (here: JIRA-12345 as used to name Jira issues). You can leave this part off if you want to forward all branches, of course.

## Section 24.4: Commit-msg

This hook is similar to the prepare-commit-msg hook, but it's called after the user enters a commit message rather than before. This is usually used to warn developers if their commit message is in an incorrect format.

The only argument passed to this hook is the name of the file that contains the message. If you don't like the message that the user has entered, you can either alter this file in-place (same as prepare-commit-msg) or you can abort the commit entirely by exiting with a non-zero status.

The following example is used to check if the word ticket followed by a number is present on the commit message

word="ticket [θ-9]" isPresent=$**(grep** -Eoh "$word" $1**)**

**if [[** -z $isPresent **]]**

**then echo** "Commit message K0, $word is missing"; **exit** 1;

**else echo** "Commit message 0K"; **exit** θ;

**fi**

## Section 24.5: Local hooks

Local hooks affect only the local repositories in which they reside. Each developer can alter their own local hooks, so they can't be used reliably as a way to enforce a commit policy. They are designed to make it easier for developers to adhere to certain guidelines and avoid potential problems down the road.

There are six types of local hooks: pre-commit, prepare-commit-msg, commit-msg, post-commit, post-checkout, and pre-rebase.

The first four hooks relate to commits and allow you to have some control over each part in a commit's life cycle. The final two let you perform some extra actions or safety checks for the git checkout and git rebase commands.

All of the "pre-" hooks let you alter the action that’s about to take place, while the "post-" hooks are used primarily for notifications.

## Section 24.6: Post-checkout

This hook works similarly to the post-commit hook, but it's called whenever you successfully check out a reference with **git checkout**. This could be a useful tool for clearing out your working directory of auto-generated files that would otherwise cause confusion.

This hook accepts three parameters:

1. the ref of the previous HEAD,
2. the ref of the new HEAD, and
3. a flag indicating if it was a branch checkout or a file checkout (1 or θ, respectively).

Its exit status has no affect on the **git checkout** command.

## Section 24.7: Post-commit

This hook is called immediately after the commit-msg hook. It cannot alter the outcome of the **git commit**

operation, therefore it's used primarily for notification purposes.

The script takes no parameters, and its exit status does not affect the commit in any way.

## Section 24.8: Post-receive

This hook is called after a successful push operation. It is typically used for notification purposes. The script takes no parameters, but is sent the same information as pre-receive via standard input:

**<old–value> <new–value> <ref–name>**

## Section 24.9: Pre-commit

This hook is executed every time you run **git commit**, to verify what is about to be committed. You can use this hook to inspect the snapshot that is about to be committed.

This type of hook is useful for running automated tests to make sure the incoming commit doesn't break existing functionality of your project. This type of hook may also check for whitespace or EOL errors.

No arguments are passed to the pre-commit script, and exiting with a non-zero status aborts the entire commit.

## Section 24.10: Prepare-commit-msg

This hook is called after the pre-commit hook to populate the text editor with a commit message. This is typically used to alter the automatically generated commit messages for squashed or merged commits.

One to three arguments are passed to this hook:

The name of a temporary file that contains the message.

The type of commit, either message (-m or -F option), template (-t option),

merge (if it's a merge commit), or

squash (if it's squashing other commits).

The SHA1 hash of the relevant commit. This is only given if -c, -C, or --amend option was given.

Similar to pre-commit, exiting with a non-zero status aborts the commit.

## Section 24.11: Pre-rebase

This hook is called before **git rebase** begins to alter code structure. This hook is typically used for making sure a rebase operation is appropriate.

This hook takes 2 parameters:

1. the upstream branch that the series was forked from, and
2. the branch being rebased (empty when rebasing the current branch).

You can abort the rebase operation by exiting with a non-zero status.

## Section 24.12: Pre-receive

This hook is executed every time somebody uses **git push** to push commits to the repository. It always resides in the remote repository that is the destination of the push and not in the originating (local) repository.

The hook runs before any references are updated. It is typically used to enforce any kind of development policy.

The script takes no parameters, but each ref that is being pushed is passed to the script on a separate line on standard input in the following format:

**<old–value> <new–value> <ref–name>**

## Section 24.1\: Update

This hook is called after pre-receive, and it works the same way. It's called before anything is actually updated, but is called separately for each ref that was pushed rather than all of the refs at once.

This hook accepts the following 3 arguments: name of the ref being updated,

old object name stored in the ref, and new object name stored in the ref.

This is the same information passed to pre-receive, but since update is invoked separately for each ref, you can reject some refs while allowing others.

# Chapter 25: Cloning Repositories

## Section 25.1: Shallow Clone

Cloning a huge repository (like a project with multiple years of history) might take a long time, or fail because of the amount of data to be transferred. In cases where you don't need to have the full history available, you can do a shallow clone:

**git clone [**repo\_url**]** --depth 1

The above command will fetch just the last commit from the remote repository.

Be aware that you may not be able to resolve merges in a shallow repository. It's often a good idea to take at least as many commits are you are going to need to backtrack to resolve merges. For example, to instead get the last 50 commits:

**git clone [**repo\_url**]** --depth 5θ

Later, if required, you can the fetch the rest of the repository:

Version ≥ 1.8.3

**git fetch** --unshallow

***# equivaﾕenł of gił fełch -***–***depłh=2147483647 # fełches łhe resł of łhe reposiłory***

Version < 1.8.3

**git fetch** --depth=1θθθ

***# fełch łhe ﾕasł 1000 commiłs***

## Section 25.2: Regular Clone

To download the entire repository including the full history and all branches, type:

**git clone <**url**>**

The example above will place it in a directory with the same name as the repository name. To download the repository and save it in a specific directory, type:

**git clone <**url**> [**directory**]**

For more details, visit Clone a repository.

## Section 25.\: Clone a specific branch

To clone a specific branch of a repository, type --branch **<**branch name**>** before the repository url:

**git clone** --branch **<**branch name**> <**url**> [**directory**]**

To use the shorthand option for --branch, type -b. This command downloads entire repository and checks out

**<branch** name**>**.

To save disk space you can clone history leading only to single branch with:

**git clone** --branch **<**branch\_name**>** --single-branch **<**url**> [**directory**]**

If --single-branch is not added to the command, history of all branches will be cloned into **[**directory**]**. This can be issue with big repositories.

To later undo --single-branch flag and fetch the rest of repository use command:

**git config** remote.origin.fetch "+refs/heads/\*:refs/remotes/origin/\*"

**git fetch** origin

## Section 25.4: Clone recursivel6

Version ≥ 1.6.5

**git clone <**url**>** --recursive

Clones the repository and also clones all submodules. If the submodules themselves contain additional submodules, Git will also clone those.

## Section 25.5: Clone using a prox6

If you need to download files with git under a proxy, setting proxy server system-wide couldn't be enough. You could also try the following:

**git config** --global http.proxy http:**//<**proxy-server**>**:**<**port**>/**

# Chapter 26: Stashing

### Parameter Details

Show the changes recorded in the stash as a diff between the stashed state and its original parent. When no <stash> is given, shows the latest one.

show

List the stashes that you currently have. Each stash is listed with its name (e.g. stash@{0} is the latest

list

stash, stash@{1} is the one before, etc.), the name of the branch that was current when the stash was made, and a short description of the commit the stash was based on.

pop Remove a single stashed state from the stash list and apply it on top of the current working tree state. apply Like pop, but do not remove the state from the stash list.

Remove all the stashed states. Note that those states will then be subject to pruning, and may be impossible to recover.

clear

Remove a single stashed state from the stash list. When no <stash> is given, it removes the latest one.

drop

i.e. stash@{0}, otherwise <stash> must be a valid stash log reference of the form stash@{<revision>}.

Create a stash (which is a regular commit object) and return its object name, without storing it

create store

anywhere in the ref namespace. This is intended to be useful for scripts. It is probably not the command you want to use; see "save" above.

Store a given stash created via git stash create (which is a dangling merge commit) in the stash ref, updating the stash reflog. This is intended to be useful for scripts. It is probably not the command you want to use; see "save" above.

**Section 26.1: What is Stashing?**

When working on a project, you might be half-way through a feature branch change when a bug is raised against master. You're not ready to commit your code, but you also don't want to lose your changes. This is where **git stash** comes in handy.

Run **git status** on a branch to show your uncommitted changes:

**(**master**)** $ **git status**

0n branch master

Your branch is up-to-date with 'origin/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: business**/**com**/**test**/**core**/**actions**/**Photo.c

no changes added to commit **(**use "git add" and**/**or "git commit -a"**)**

Then run **git stash** to save these changes to a stack:

**(**master**)** $ **git stash**

5aved working directory and index state WIP on master: 2f2aбe1 Merge pull request ***#1 from łesł/łesł-branch***

HEAD is now at 2f2aбe1 Merge pull request ***#1 from łesł/łesł-branch***

If you have added files to your working directory these can be stashed as well. You just need to stage them first.

**(**master**)** $ **git stash**

5aved working directory and index state WIP on master:

**(**master**)** $ **git status** 0n branch master Untracked files:

**(**use "git add <file>..." to include **in** what will be committed**)**

NewPhoto.c

nothing added to commit but untracked files present **(**use "git add" to track**) (**master**)** $ **git stage** NewPhoto.c

**(**master**)** $ **git stash**

5aved working directory and index state WIP on master:

**(**master**)** $ **git status**

0n branch master

nothing to commit, working **tree** clean

**(**master**)** $

Your working directory is now clean of any changes you made. You can see this by re-running **git status**:

**(**master**)** $ **git status**

0n branch master

Your branch is up-to-date with 'origin/master'. nothing to commit, working directory clean

To apply the very last stash, run **git stash** apply (additionally, you can apply *and* remove the last stashed changed with **git stash** pop):

**(**master**)** $ **git stash** apply 0n branch master

Your branch is up-to-date with 'origin/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: business**/**com**/**test**/**core**/**actions**/**Photo.c

no changes added to commit **(**use "git add" and**/**or "git commit -a"**)**

Note, however, that stashing does not remember the branch you were working on. In the above examples, the user was stashing on **master**. If they switch to the **dev** branch, **dev**, and run **git stash** apply the last stash is put on the **dev** branch.

**(**master**)** $ **git checkout** -b dev 5witched to a new branch 'dev' **(**dev**)** $ **git stash** apply

0n branch dev

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: business**/**com**/**test**/**core**/**actions**/**Photo.c

no changes added to commit **(**use "git add" and**/**or "git commit -a"**)**

## Section 26.2: Create stash

Save the current state of working directory and the index (also known as the staging area) in a stack of stashes.

**git stash**

To include all untracked files in the stash use the --include-untracked or -u flags.

**git stash** --include-untracked

To include a message with your stash to make it more easily identifiable later

**git stash** save "<whatever message>"

To leave the staging area in current state after stash use the --keep-index or -k flags.

**git stash** --keep-index

## Section 26.\: Appl6 and remove stash

To apply the last stash and remove it from the stack - type:

**git stash** pop

To apply specific stash and remove it from the stack - type:

**git stash** pop stash**@{**n**}**

## Section 26.4: Appl6 stash without removing it

Applies the last stash without removing it from the stack

**git stash** apply

Or a specific stash

**git stash** apply stash**@{**n**}**

## Section 26.5: Show stash

Shows the changes saved in the last stash

**git stash** show

Or a specific stash

**git stash** show stash**@{**n**}**

To show content of the changes saved for the specific stash

**git stash** show -p stash**@{**n**}**

## Section 26.6: Partial stash

If you would like to stash only *some* diffs in your working set, you can use a partial stash.

**git stash** -p

And then interactively select which hunks to stash.

As of version 2.13.0 you can also avoid the interactive mode and create a partial stash with a pathspec using the new **push** keyword.

**git stash** push -m "My partial stash" -- app.config

## Section 26.7: List saved stashes

**git stash** list

This will list all stashes in the stack in reverse chronological order. You will get a list that looks something like this:

stash@{8}: WIP on master: 67a4e81 Merge tests into devetop

stash@{1}: WIP on master: 78f8d95 Add user rote to tocatStorage on user togin

You can refer to specific stash by its name, for example stash**@{**1**}**.

## Section 26.8: Move 6our work in progress to another branch

If while working you realize you're on wrong branch and you haven't created any commits yet, you can easily move your work to correct branch using stashing:

**git stash**

**git checkout** correct-branch

**git stash** pop

Remember **git stash** pop will apply the last stash and delete it from the stash list. To keep the stash in the list and only apply to some branch you can use:

**git stash** apply

## Section 26.9: Remove stash

Remove all stash

**git stash clear**

Removes the last stash

**git stash** drop

Or a specific stash

**git stash** drop stash**@{**n**}**

## Section 26.10: Appl6 part of a stash with checkout

You've made a stash and wish to checkout only some of the files in that stash.

**git checkout** stash**@{**θ**}** -- myfile.txt

## Section 26.11: Recovering earlier changes from stash

To get your most recent stash after running git stash, use

**git stash** apply

To see a list of your stashes, use

**git stash** list

You will get a list that looks something like this

stash@{8}: WIP on master: 67a4e81 Merge tests into devetop

stash@{1}: WIP on master: 78f8d95 Add user rote to tocatStorage on user togin

Choose a different git stash to restore with the number that shows up for the stash you want

**git stash** apply stash**@{**2**}**

## Section 26.12: Interactive Stashing

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes –

and saves it on a stack of unfinished changes that you can reapply at any time.

### Stashing only modified files:

Suppose you don't want to stash the staged files and only stash the modified files so you can use:

**git stash** --keep-index

Which will stash only the modified files.

### Stashing untracked files:

Stash never saves the untracked files it only stashes the modified and staged files. So suppose if you need to stash the untracked files too then you can use this:

**git stash** -u

this will track the untracked, staged and modified files.

### Stash some particular changes only:

Suppose you need to stash only some part of code from the file or only some files only from all the modified and stashed files then you can do it like this:

**git stash** --patch

Git will not stash everything that is modified but will instead prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directory.

## Section 26.1\: Recover a dropped stash

If you have only just popped it and the terminal is still open, you will still have the hash value printed by **git stash**

pop on screen:

$ **git stash** pop

**[**...**]**

Dropped refs**/**stash**@{**θ**} (**2caθ3e2225бbe97f9e4θfθ8eбdб773c7d41dbfd1**)**

(Note that git stash drop also produces the same line.) Otherwise, you can find it using this:

**git fsck** --no-reflog **| awk** '/dangling commit/ {print $3}'

This will show you all the commits at the tips of your commit graph which are no longer referenced from any branch or tag – every lost commit, including every stash commit you’ve ever created, will be somewhere in that graph.

The easiest way to find the stash commit you want is probably to pass that list to gitk:

gitk --all $**( git fsck** --no-reflog **| awk** '/dangling commit/ {print $3}' **)**

This will launch a repository browser showing you *every single commit in the repository ever*, regardless of whether it is reachable or not.

You can replace gitk there with something like **git log** --graph --oneline --decorate if you prefer a nice graph on the console over a separate GUI app.

To spot stash commits, look for commit messages of this form: WIP on *somebranch*: *commithash Some old commit message*

Once you know the hash of the commit you want, you can apply it as a stash:

**git stash** apply sh\_hash

Or you can use the context menu in gitk to create branches for any unreachable commits you are interested in. After that, you can do whatever you want with them with all the normal tools. When you’re done, just blow those branches away again.

# Chapter 27: Subtrees

## Section 27.1: Create, Pull, and Backport Subtree

### Create Subtree

Add a new remote called plugin pointing to the plugin's repository:

**git remote** add plugin https:**//**path.to**/**remotes**/**plugin.git

Then Create a subtree specifying the new folder prefix plugins**/**demo. plugin is the remote name, and master

refers to the master branch on the subtree's repository:

**git** subtree add --prefix=plugins**/**demo plugin master

### Pull Subtree Updates

Pull normal commits made in plugin:

**git** subtree pull --prefix=plugins**/**demo plugin master

### Backport Subtree Updates

1. Specify commits made in superproject to be backported:

**git commit** -am "new changes to be backported"

1. Checkout new branch for merging, set to track subtree repository:

**git checkout** -b backport plugin**/**master

1. Cherry-pick backports:

**git cherry–pick** -x --strategy=subtree master

1. Push changes back to plugin source:

**git push** plugin backport:master

# Chapter 28: Renaming

### Parameter Details

-f or --force Force renaming or moving of a file even if the target exists

**Section 28.1: Rename Folders**

To rename a folder from oldName to newName

**git mv** directoryToFolder**/**oldName directoryToFolder**/**newName

Followed by **git commit** and/or **git push**

If this error occurs:

fatal: renaming 'directoryToFolder/oldName' failed: Invalid argument

Use the following command:

**git mv** directoryToFolder**/**oldName temp **&& git mv** temp directoryToFolder**/**newName

## Section 28.2: rename a local and the remote branch

the easiest way is to have the local branch checked out:

**git checkout** old\_branch

then rename the local branch, delete the old remote and set the new renamed branch as upstream:

**git branch** -m new\_branch

**git push** origin :old\_branch

**git push** --set-upstream origin new\_branch

## Section 28.\: Renaming a local branch

You can rename branch in local repository using this command:

**git branch** -m old\_name new\_name

# Chapter 29: Pushing

### Parameter Details

Overwrites the remote ref to match your local ref. *Can cause the remote repository to lose commits, so use with care*.

--force

--verbose Run verbosely.

<remote> The remote repository that is destination of the push operation.

<refspec>... Specify what remote ref to update with what local ref or object.

After changing, staging, and committing code with Git, pushing is required to make your changes available to others and transfers your local changes to the repository server. This topic will cover how to properly push code using Git.

**Section 29.1: Push a specific ob¡ect to a remote branch**

### General syntax

**git push <**remotename**> <**object**>**:**<**remotebranchname**>**

**Example**

**git push** origin master:wip-yourname

Will push your master branch to the wip-yourname branch of origin (most of the time, the repository you cloned from).

### Delete remote branch

Deleting the remote branch is the equivalent of pushing an empty object to it.

**git push <**remotename**>** :**<**remotebranchname**>**

### Example

**git push** origin :wip-yourname

Will delete the remote branch wip-yourname

Instead of using the colon, you can also use the --delete flag, which is better readable in some cases.

### Example

**git push** origin --delete wip-yourname

**Push a single commit**

If you have a single commit in your branch that you want to push to a remote without pushing anything else, you can use the following

**git push <**remotename**> <**commit 5HA**>**:**<**remotebranchname**>**

### Example

Assuming a git history like this

eeb32bc Commit 1 - atready pushed 347d788 Commit 2 - want to push e539af8 Commit 3 - onty tocat 5d339db Commit 4 - onty tocat

to push only commit *347d700* to remote *master* use the following command

**git push** origin 347d7θθ:master

## Section 29.2: Push

**git push**

will push your code to your existing upstream. Depending on the push configuration, it will either push code from you current branch (default in Git 2.x) or from all branches (default in Git 1.x).

### Specify remote repository

When working with git, it can be handy to have multiple remote repositories. To specify a remote repository to push to, just append its name to the command.

**git push** origin

### Specify Branch

To push to a specific branch, say feature\_x:

**git push** origin feature\_x

### Set the remote tracking branch

Unless the branch you are working on originally comes from a remote repository, simply using **git push** won't work the first time. You must perform the following command to tell git to push the current branch to a specific remote/branch combination

**git push** --set-upstream origin master

Here, master is the branch name on the remote origin. You can use -u as a shorthand for --set-upstream.

### Pushing to a new repository

To push to a repository that you haven't made yet, or is empty:

1. Create the repository on GitHub (if applicable)
2. Copy the url given to you, in the form https:**//**github.com**/**U5ERNAME**/**REP0\_NAME.git
3. Go to your local repository, and execute **git remote** add origin URL

To verify it was added, run **git remote** -v

1. Run **git push** origin master

Your code should now be on GitHub

For more information view Adding a remote repository

### Explanation

Push code means that git will analyze the differences of your local commits and remote and send them to be written on the upstream. When push succeeds, your local repository and remote repository are synchronized and other users can see your commits.

For more details on the concepts of "upstream" and "downstream", see Remarks.

## Section 29.\: Force Pushing

Sometimes, when you have local changes incompatible with remote changes (ie, when you cannot fast-forward the remote branch, or the remote branch is not a direct ancestor of your local branch), the only way to push your changes is a force push.

**git push** -f

or

**git push** --force

### Important notes

This will **overwrite** any remote changes and your remote will match your local.

Attention: Using this command may cause the remote repository to **lose commits**. Moreover, it is strongly advised against doing a force push if you are sharing this remote repository with others, since their history will retain every overwritten commit, thus rending their work out of sync with the remote repository.

As a rule of thumb, only force push when:

Nobody except you pulled the changes you are trying to overwrite

You can force everyone to clone a fresh copy after the forced push and make everyone apply their changes to it (people may hate you for this).

## Section 29.4: Push tags

**git push** --tags

Pushes all of the **git** tags in the local repository that are not in the remote one.

## Section 29.5: Changing the default push behavior

**Current** updates the branch on the remote repository that shares a name with the current working branch.

**git config** push.default current

**Simple** pushes to the upstream branch, but will not work if the upstream branch is called something else.

**git config** push.default simple

**Upstream** pushes to the upstream branch, no matter what it is called.

**git config** push.default upstream

**Matching** pushes all branches that match on the local and the remote git config push.default upstream After you've set the preferred style, use

**git push**

to update the remote repository.

# Chapter \0: Internals

## Section \0.1: Repo

A **git** repository is an on-disk data structure which stores metadata for a set of files and directories.

It lives in your project's .git**/** folder. Every time you commit data to git, it gets stored here. Inversely, .git**/**

contains every single commit.

It's basic structure is like this:

.git**/**

objects**/** refs**/**

## Section \0.2: Ob¡ects

**git** is fundamentally a key-value store. When you add data to **git**, it builds an object and uses the SHA-1 hash of the object's contents as a key.

Therefore, any content in **git** can be looked up by it's hash:

**git cat–file** -p 4bbбf98

There are 4 types of 0bject:

blob **tree** commit tag

## Section \0.\: HEAD ref

HEAD is a special ref. It always points to the current object.

You can see where it's currently pointing by checking the .git**/**HEAD file. Normally, HEAD points to another ref:

$cat .git**/**HEAD

ref: refs**/**heads**/**mainline

But it can also point directly to an object:

$ **cat** .git**/**HEAD 4bbбf98a223abc9345aθcef92θθ5б2333

This is what's known as a "detached head" - because HEAD is not attached to (pointing at) any ref, but rather points directly to an object.

## Section \0.4: Refs

A ref is essentially a pointer. It's a name that points to an object. For example,

"master" --**>** 1a41θe...

They are stored in `.git/refs/heads/ in plain text files.

$ **cat** .git**/**refs**/**heads**/**mainline 4bbбf98a223abc9345aθcef92θθ5б2333

This is commonly what are called branches. However, you'll note that in **git** there is no such thing as a branch - only a ref.

Now, it's possible to navigate **git** purely by jumping around to different objects directly by their hashes. But this would be terribly inconvenient. A ref gives you a convenient name to refer to objects by. It's much easier to ask **git** to go to a specific place by name rather than by hash.

## Section \0.5: Commit Ob¡ect

A commit is probably the object type most familiar to **git** users, as it's what they are used to creating with the **git commit** commands.

However, the commit does not directly contain any changed files or data. Rather, it contains mostly metadata and pointers to other objects which contain the actual contents of the commit.

A commit contains a few things: hash of a **tree**

hash of a parent commit

author name/email, commiter name/email commit message

You can see the contents of any commit like this:

$ **git cat–file** commit 5bac93

**tree** θ4d1daef... parent b785θef5...

author Geddy Lee **<**[glee**@**rush.com](mailto:glee@rush.com)**>**

commiter Neil Peart **<**[npeart**@**rush.com](mailto:npeart@rush.com)**>**

First commit**!**

### Tree

A very important note is that the **tree** objects stores EVERY file in your project, and it stores whole files not diffs. This means that each commit contains a snapshot of the entire project\*.

\**Technically, only changed files are stored. But this is more an implementation detail for efficiency. From a design perspective, a* ***commił*** *should be considered as containing a complete copy of the project*.

### Parent

The parent line contains a hash of another commit object, and can be thought of as a "parent pointer" that points to the "previous commit". This implicitly forms a graph of commits known as the **commit graph**. Specifically, it's a directed acyclic graph (or DAG).

## Section \0.6: Tree Ob¡ect

A **tree** basically represents a folder in a traditional filesystem: nested containers for files or other folders. A **tree** contains:

0 or more blob objects 0 or more **tree** objects

Just as you can use ls or **dir** to list the contents of a folder, you can list the contents of a **tree** object.

$ **git cat–file** -p θ7b1aб31

1θθб44 blob b91bba1b .gitignore 1θθб44 blob ccθ95бf1 Makefile θ4θθθθ **tree** 92e1ca7e src

...

You can look up the files in a commit by first finding the hash of the **tree** in the commit, and then looking at that

**tree**:

$ **git cat–file** commit 4bbбf93a

**tree** θ7b1aб31 parent ...

author ... commiter ...

$ **git cat–file** -p θ7b1aб31

1θθб44 blob b91bba1b .gitignore 1θθб44 blob ccθ95бf1 Makefile θ4θθθθ **tree** 92e1ca7e src

...

## Section \0.7: Blob Ob¡ect

A blob contains arbitrary binary file contents. Commonly, it will be raw text such as source code or a blog article. But it could just as easily be the bytes of a PNG file or anything else.

If you have the hash of a blob, you can look at it's contents.

$ **git cat–file** -p d42981θ package com.example.project

class Foo **{**

...

**}**

...

For example, you can browse a **tree** as above, and then look at one of the blobs in it.

$ **git cat–file** -p θ7b1aб31

1θθб44 blob b91bba1b .gitignore 1θθб44 blob ccθ95бf1 Makefile θ4θθθθ **tree** 92e1ca7e src

1θθб44 blob cae391ff Readme.txt

$ **git cat–file** -p cae391ff

Welcome to my project**!** This is the readmefile

...

## Section \0.8: Creating new Commits

The **git commit** command does a few things:

1. Create blobs and trees to represent your project directory - stored in .git**/**objects
2. Creates a new commit object with your author information, commit message, and the root **tree** from step 1 - also stored in .git**/**objects
3. Updates the HEAD ref in .git**/**HEAD to the hash of the newly-created commit

This results in a new snapshot of your project being added to **git** that is connected to the previous state.

## Section \0.9: Moving HEAD

When you run **git checkout** on a commit (specified by hash or ref) you're telling **git** to make your working directory look like how it did when the snapshot was taken.

1. Update the files in the working directory to match the **tree** inside the commit
2. Update HEAD to point to the specified hash or ref

## Section \0.10: Moving refs around

Running **git reset** --hard moves refs to the specified hash/ref. Moving MyBranch to b8dc53:

$ **git checkout** MyBranch

$ **git reset** --hard b8dc53

***# moves HEAD ło MyBranch***

***# makes MyBranch poinł ło b8dc€3***

## Section \0.11: Creating new Refs

Running **git checkout** -b **<**refname**>** will create a new ref that points to the current commit.

$ **cat** .git**/head** 1f324a

$ **git checkout** -b TestBranch

$ **cat** .git**/**refs**/**heads**/**TestBranch 1f324a

# Chapter \1: git-tfs

## Section \1.1: git-tfs clone

This will create a folder with the same name as the project, i.e. /My.Project.Name

$ **git** tfs clone http:**//**tfs:8θ8θ**/**tfs**/**DefaultCollection**/** $**/**My.Project.Name

## Section \1.2: git-tfs clone from bare git repositor6

Cloning from a git repository is ten times faster than cloning directly from TFVS and works well in a team environment. At least one team member will have to create the bare git repository by doing the regular git-tfs clone first. Then the new repository can be bootstrapped to work with TFVS.

$ **git clone** x:**/**fileshare**/**git**/**My.Project.Name.git

$ **cd** My.Project.Name

$ **git** tfs bootstrap

$ **git** tfs pull

## Section \1.\: git-tfs install via Chocolate6

The following assumes you will use kdiff3 for file diffing and although not essential it is a good idea.

C:\**>** choco **install** kdiff3

Git can be installed first so you can state any parameters you wish. Here all the Unix tools are also installed and 'NoAutoCrlf' means checkout as is, commit as is.

C:\**>** choco **install git** -params '"/GitAndUnixTools0nPath /NoAutoCrlf"'

This is all you really need to be able to install git-tfs via chocolatey.

C:\**>** choco **install** git-tfs

## Section \1.4: git-tfs Check In

Launch the Check In dialog for TFVS.

$ **git** tfs checkintool

This will take all of your local commits and create a single check-in.

## Section \1.5: git-tfs push

Push all local commits to the TFVS remote.

$ **git** tfs rcheckin

Note: this will fail if Check-in Notes are required. These can be bypassed by adding git-tfs-force: rcheckin to the commit message.

# Chapter \2: Empt6 directories in Git

## Section \2.1: Git doesn’t track directories

Assume you've initialized a project with the following directory structure:

/buitd app.js

Then you add everything so you've created so far and commit:

**git init git add** .

**git commit** -m "Initial commit"

Git will only track the file app.js.

Assume you added a build step to your application and rely on the "build" directory to be there as the output directory (and you don't want to make it a setup instruction every developer has to follow), a *convention* is to include a ".gitkeep" file inside the directory and let Git track that file.

**/**build

.gitkeep app.js

Then add this new file:

**git add** build**/**.gitkeep

**git commit** -m "Keep the build directory around"

Git will now track the file build/.gitkeep file and therefore the build folder will be made available on checkout. Again, this is just a convention and not a Git feature.

# Chapter \\: git-svn

## Section \\.1: Cloning the SVN repositor6

You need to create a new local copy of the repository with the command

**git svn** clone 5VN\_REP0\_R00T\_URL **[**DE5T\_F0LDER\_PATH**]** -T TRUNK\_REP0\_PATH -t TAG5\_REP0\_PATH -b BRANCHE5\_REP0\_PATH

If your SVN repository follows the standard layout (trunk, branches, tags folders) you can save some typing:

**git svn** clone -s 5VN\_REP0\_R00T\_URL **[**DE5T\_F0LDER\_PATH**]**

**git svn** clone checks out each SVN revision, one by one, and makes a git commit in your local repository in order to recreate the history. If the SVN repository has a lot of commits this will take a while.

When the command is finished you will have a full fledged git repository with a local branch called master that tracks the trunk branch in the SVN repository.

## Section \\.2: Pushing local changes to SVN

The command

**git svn** dcommit

will create a SVN revision for each of your local git commits. As with SVN, your local git history must be in sync with the latest changes in the SVN repository, so if the command fails, try performing a **git svn** rebase first.

## Section \\.\: Working locall6

Just use your local git repository as a normal git repo, with the normal git commands:

**git add** FILE and **git checkout** -- FILE To stage/unstage a file

**git commit** To save your changes. Those commits will be local and will not be "pushed" to the SVN repo, just like in a normal git repository

**git stash** and **git stash** pop Allows using stashes **git reset** HEAD --hard Revert all your local changes **git log** Access all the history in the repository

**git rebase** -i so you can rewrite your local history freely

**git branch** and **git checkout** to create local branches

As the git-svn documentation states "Subversion is a system that is far less sophisticated than Git" so you can't use all the full power of git without messing up the history in the Subversion server. Fortunately the rules are very simple: **Keep the history linear**

This means you can make almost any git operation: creating branches, removing/reordering/squashing commits, move the history around, delete commits, etc. Anything *but merges*. If you need to reintegrate the history of local branches use **git rebase** instead.

When you perform a merge, a merge commit is created. The particular thing about merge commits is that they have two parents, and that makes the history non-linear. Non-linear history will confuse SVN in the case you "push" a merge commit to the repository.

However do not worry: **you won't break anything if you "push" a git merge commit to SVN**. If you do so, when

the git merge commit is sent to the svn server it will contain all the changes of all commits for that merge, so you will lose the history of those commits, but not the changes in your code.

## Section \\.4: Getting the latest changes from SVN

The equivalent to **git pull** is the command

**git svn** rebase

This retrieves all the changes from the SVN repository and applies them *on top* of your local commits in your current branch.

You can also use the command

**git svn** fetch

to retrieve the changes from the SVN repository and bring them to your local machine but without applying them to your local branch.

## Section \\.5: Handling empt6 folders

git does not recognice the concept of folders, it just works with files and their filepaths. This means git does not track empty folders. SVN, however, does. Using git-svn means that, by default, *any change you do involving empty folders with git will not be propagated to SVN*.

Using the --rmdir flag when issuing a comment corrects this issue, and removes an empty folder in SVN if you locally delete the last file inside it:

**git svn** dcommit --rmdir

Unfortunately **it does not removes existing empty folders**: you need to do it manually.

To avoid adding the flag each time you do a dcommit, or to play it safe if you are using a git GUI tool (like SourceTree) you can set this behaviour as default with the command:

**git config** --global svn.rmdir **true**

This changes your .gitconfig file and adds these lines:

**[svn]**

**rmdir** = **true**

To remove all untracked files and folders that should be kept empty for SVN use the git command:

**git clean** -fd

Please note: the previous command will remove all untracked files and empty folders, even the ones that should be tracked by SVN! If you need to generate againg the empty folders tracked by SVN use the command

**git svn** mkdirs

In practices this means that if you want to cleanup your workspace from untracked files and folders you should always use both commands to recreate the empty folders tracked by SVN:

**git clean** -fd **&& git svn** mkdirs

# Chapter \4: Archive

### Parameter Details

Format of the resulting archive: **tar** or **zip**. If this options is not given and the output

--format=<fmt>

file is specified, the format is inferred from the filename if possible. Otherwise, defaults to **tar**.

-l, --list Show all available formats.

-v, --verbose Report progress to stderr.

--prefix=<prefix>/ Prepend <prefix>/ to each filename in the archive.

-o <file>, --output=<file> Write the archive to <file> instead of stdout.

--worktree-attributes Look for attributes in .gitattributes files in the working tree.

This can be any options that the archiver backend understands. For **zip** backend,

<extra>

using -θ will store the files without deflating them, while -1 through -9 can be used to adjust compression speed and ratio.

--remote=<repo> Retrieve a tar archive from a remote repository **<repo>** rather than the local

repository.

--exec=<git-upload-archive> Used with --remote to specify the path to the **<git–upload–archive** on the remote.

<tree-ish> The tree or commit to produce an archive for.

Without an optional parameter, all files and directories in the current working

<path>

directory are included in the archive. If one or more paths are specified, only these are included.

**Section \4.1: Create an archive of git repositor6**

With **git archive** it is possible to create compressed archives of a repository, for example for distributing releases. Create a tar archive of current HEAD revision:

**git archive** --format **tar** HEAD **| cat >** archive-HEAD.tar

Create a tar archive of current HEAD revision with gzip compression:

**git archive** --format **tar** HEAD **| gzip >** archive-HEAD.tar.gz

This can also be done with (which will use the in-built tar.gz handling):

**git archive** --format tar.gz HEAD **>** archive-HEAD.tar.gz

Create a zip archive of current HEAD revision:

**git archive** --format **zip** HEAD **>** archive-HEAD.zip

Alternatively it is possible to just specify an output file with valid extension and the format and compression type will be inferred from it:

**git archive** --output=archive-HEAD.tar.gz HEAD

## Section \4.2: Create an archive of git repositor6 with director6 prefix

It is considered good practice to use a prefix when creating git archives, so that extraction will place all files inside a

directory. To create an archive of HEAD with a directory prefix:

**git archive** --output=archive-HEAD.zip --prefix=src-directory-name HEAD

When extracted all the files will be extracted inside a directory named src-directory-name in the current directory.

## Section \4.\: Create archive of git repositor6 based on specific branch, revision, tag or director6

It is also possible to create archives of other items than HEAD, such as branches, commits, tags, and directories. To create an archive of a local branch dev:

**git archive** --output=archive-dev.zip --prefix=src-directory-name dev

To create an archive of a remote branch origin**/**dev:

**git archive** --output=archive-dev.zip --prefix=src-directory-name origin**/**dev

To create an archive of a tag v.θ1:

**git archive** --output=archive-v.θ1.zip --prefix=src-directory-name v.θ1

Create an archive of files inside a specific sub directory (sub-dir) of revision HEAD:

**git archive zip** --output=archive-sub-dir.zip --prefix=src-directory-name HEAD:sub-dir**/**

# Chapter \5: Rewriting histor6 with filter- branch

## Section \5.1: Changing the author of commits

You can use an environment filter to change the author of commits. Just modify and export $GIT\_AUTH0R\_NAME in the script to change who authored the commit.

Create a file filter.sh with contents like so:

**if [** "$GIT\_AUTH0R\_NAME" = "Author to Change From" **] then**

**export** GIT\_AUTH0R\_NAME="Author to Change To"

**export** GIT\_AUTH0R\_EMAIL=["email.to.change.to@example.com"](mailto:email.to.change.to@example.com)

**fi**

Then run filter-branch from the command line:

**chmod** +x .**/**filter.sh

**git filter–branch** --env-filter .**/**filter.sh

## Section \5.2: Setting git committer equal to commit author

This command, given a commit range commit1..commit2, rewrites history so that git commit author becomes also git committer:

**git filter–branch** -f --commit-filter \

'export GIT\_C0MMITTER\_NAME=\"$GIT\_AUTH0R\_NAME\"; export GIT\_C0MMITTER\_EMAIL=\"$GIT\_AUTH0R\_EMAIL\"; export GIT\_C0MMITTER\_DATE=\"$GIT\_AUTH0R\_DATE\";

git commit-tree $@' \

-- commit1..commit2

# Chapter \6: Migrating to Git

## Section \6.1: SubGit

SubGit may be used to perform a one-time import of an SVN repository to git.

$ subgit import --non-interactive --svn-url [http:**//**svn.my.co**/**repos**/**myproject](http://svn.my.co/repos/myproject) myproject.git

## Section \6.2: Migrate from SVN to Git using Atlassian conversion utilit6

Download the Atlassian conversion utility here. This utility requires Java, so please ensure that you have the Java Runtime Environment JRE installed on the machine you plan to do the conversion.

Use the command **java** -jar svn-migration-scripts.jar verify to check if your machine is missing any of the programs necessary to complete the conversion. Specifically, this command checks for the Git, subversion, and **git–svn** utilities. It also verifies that you are performing the migration on a case-sensitive file system. Migration to Git should be done on a case-sensitive file system to avoid corrupting the repository.

Next, you need to generate an authors file. Subversion tracks changes by the committer's username only. Git, however, uses two pieces of information to distinguish a user: a real name and an email address. The following command will generate a text file mapping the subversion usernames to their Git equivalents:

**java** -jar svn-migration-scripts.jar authors **<**svn-repo**>** authors.txt

where **<svn–repo>** is the URL of the subversion repository you wish to convert. After running this command, the contributors' identification information will be mapped in authors.txt. The email addresses will be of the form

**<username>**@mycompany.com. In the authors file, you will need to manually change each person's default name

(which by default has become their username) to their actual names. Make sure to also check all of the email addresses for correctness before proceeding.

The following command will clone an svn repo as a Git one:

**git svn** clone --stdlayout --authors-file=authors.txt **<**svn-repo**> <**git-repo-name**>**

where **<svn–repo>** is the same repository URL used above and **<git–repo–name>** is the folder name in the current directory to clone the repository into. There are a few considerations before using this command:

The --stdlayout flag from above tells Git that you're using a standard layout with trunk, branches, and tags folders. Subversion repositories with non-standard layouts require you to specify the locations of the trunk folder, any/all branch folders, and the tags folder. This can be done by following this example: **git svn** clone --trunk=**/**trunk --branches=**/**branches --branches=**/**bugfixes --tags=**/**tags --authors- file=authors.txt **<**svn-repo**> <**git-repo-name**>**.

This command could take many hours to complete depending on the size of your repo.

To cut down the conversion time for large repositories, the conversion can be run directly on the server hosting the subversion repository in order to eliminate network overhead.

**git svn** clone imports the subversion branches (and trunk) as remote branches including subversion tags (remote branches prefixed with tags**/**). To convert these to actual branches and tags, run the following commands on a Linux machine in the order they are provided. After running them, **git branch** -a should show the correct branch names, and **git tag** -l should show the repository tags.

**git for–each–ref** refs**/**remotes**/**origin**/**tags **| cut** -d **/** -f 5- **| grep** -v **@ | while read** tagname; **do git tag** $tagname origin**/**tags**/**$tagname; **git branch** -r -d origin**/**tags**/**$tagname; **done**

**git for–each–ref** refs**/**remotes **| cut** -d **/** -f 4- **| grep** -v **@ | while read** branchname; **do git branch**

"$branchname" "refs/remotes/origin/$branchname"; **git branch** -r -d "origin/$branchname"; **done**

The conversion from svn to Git is now complete! Simply push your local repo to a server and you can continue to contribute using Git as well as having a completely preserved version history from svn.

## Section \6.\: Migrating Mercurial to Git

One can use the following methods in order to import a Mercurial Repo into Git:

1. Using fast export:

**cd**

**git clone** git:**//**repo.or.cz**/**fast-export.git

**git init** git\_repo

**cd** git\_repo

~**/**fast-export**/**hg-fast-export.sh -r **/**path**/**to**/**old**/**mercurial\_repo **git checkout** HEAD

1. Using Hg-Git: A very detailed answer here: https://stackoverflow.com/a/31827990/5283213
2. Using GitHub's Importer: Follow the (detailed) instructions at GitHub.

## Section \6.4: Migrate from Team Foundation Version Control (TFVC) to Git

You could migrate from team foundation version control to git by using an open source tool called Git-TF. Migration will also transfer your existing history by converting tfs checkins to git commits.

To put your solution into Git by using Git-TF follow these steps:

### Download Git-TF

You can download (and install) Git-TF from Codeplex: Git-TF @ Codeplex

### Clone your TFVC solution

Launch powershell (win) and type the command

git-tf clone http:**//**my.tfs.server.address:port**/**tfs**/**mycollection '$/myproject/mybranch/mysolution' -

-deep

The --deep switch is the keeyword to note as this tells Git-Tf to copy your checkin-history. You now have a local git repository in the folder from which you called your cloe command from.

### Cleanup

Add a .gitignore file. If you are using Visual Studio the editor can do this for you, otherwise you could do this manually by downloading a complete file from github/gitignore.

RemoveTFS source control bindings from solution (remove all \*.vssscc files). You could also modify your solution file by removing the GlobalSection(TeamFoundationVersionControl). EndClobalSection

### Commit & Push

Complete your conversion by committing and pushing your local repository to your remote.

**git add** .

**git commit** -a -m "Coverted solution source control from TFVC to Git"

**git remote** add origin https:**//**my.remote**/**project**/**repo.git

**git push** origin master

## Section \6.5: Migrate from SVN to Git using svn2git

svn2git is a Ruby wrapper around git's native SVN support through git-svn, helping you with migrating projects from Subversion to Git, keeping history (incl. trunk, tags and branches history).

### Examples

To migrate a svn repository with the standard layout (ie. branches, tags and trunk at the root level of the repository):

$ svn2git [http:**//**svn.example.com**/**path**/**to**/**repo](http://svn.example.com/path/to/repo)

To migrate a svn repository which is not in standard layout:

$ svn2git [http:**//**svn.example.com**/**path**/**to**/**repo](http://svn.example.com/path/to/repo) --trunk trunk-dir --tags tags-dir --branches branches-dir

In case you do not want to migrate (or do not have) branches, tags or trunk you can use options --notrunk, -- nobranches, and --notags.

For example, $ svn2git [http:**//**svn.example.com**/**path**/**to**/**repo --trunk trunk-dir --notags --nobranches](http://svn.example.com/path/to/repo--trunktrunk-dir--notags--nobranches)

will migrate only trunk history.

To reduce the space required by your new repository you may want to exclude any directories or files you once added while you should not have (eg. build directory or archives):

$ svn2git [http:**//**svn.example.com**/**path**/**to**/**repo](http://svn.example.com/path/to/repo) --exclude build --exclude '.\*\.zip$'

### Post-migration optimization

If you already have a few thousand of commits (or more) in your newly created git repository, you may want to reduce space used before pushing your repository on a remote. This can be done using the following command:

$ **git gc** --aggressive

**Note:** The previous command can take up to several hours on large repositories (tens of thousand of commits and/or hundreds of megabytes of history).

# Chapter \7: Show

## Section \7.1: Overview

**git show** shows various Git objects.

### For commits:

Shows the commit message and a diff of the changes introduced.

### Command Description

**git show** shows the previous commit

**git show @**~3 shows the 3rd-from-last commit

### For trees and blobs:

Shows the tree or blob.

### Command Description

**git show @**~3: shows the project root directory as it was 3 commits ago (a tree)

**git show @**~3:src**/**program.js shows src**/**program.js as it was 3 commits ago (a blob)

**git show @**:a.txt **@**:b.txt shows a.txt concatenated with b.txt from current commit

### For tags:

Shows the tag message and the referenced object.

# Chapter \8: Resolving merge conflicts

## Section \8.1: Manual Resolution

While performing a **git merge** you may find that git reports a "merge conflict" error. It will report to you which files have conflicts, and you will need to resolve the conflicts.

A **git status** at any point will help you see what still needs editing with a helpful message like

0n branch master

You have unmerged paths.

**(**fix conflicts and run "git commit"**)**

Unmerged paths:

**(**use "git add <file>..." to mark resolution**)**

both modified:

index.html

no changes added to commit **(**use "git add" and**/**or "git commit -a"**)**

Git leaves markers in the files to tell you where the conflict arose:

<<<<<<<<< HEAD: index.html #indicates the state of your current branch

**<div** id="footer"**>**contact : [email@somedomain.com](mailto:email@somedomain.com)**</div>**

========= #indicates break between conflicts

**<div** id="footer"**>**

please contact us at [email@somedomain.com](mailto:email@somedomain.com)

**</div>**

>>>>>>>>> iss2: index.html #indicates the state of the other branch (iss2)

In order to resolve the conflicts, you must edit the area between the <<<<<< and >>>>>>> markers appropriately, remove the status lines (the <<<<<<<, >>>>>>>, and ======== lines) completely. Then **git add** index.html to mark it resolved and **git commit** to finish the merge.

# Chapter \9: Bundles

## Section \9.1: Creating a git bundle on the local machine and using it on another

Sometimes you may want maintain versions of a git repository on machines that have no network connection. Bundles allow you to package git objects and references in a repository on one machine and import those into a repository on another.

**git tag** 2θ1б\_θ7\_24

**git bundle** create changes\_between\_tags.bundle **[**some\_previous\_tag**]**..2θ1б\_θ7\_24

Somehow transfer the **changes\_between\_tags.bundle** file to the remote machine; e.g., via thumb drive. Once you have it there:

**git bundle** verify changes\_between\_tags.bundle ***# make sure bundﾕe arrived inłacł***

**git checkout [**some branch**] *# in łhe repo on łhe remołe machine***

**git bundle** list-heads changes\_between\_tags.bundle ***# ﾕisł łhe references in łhe bundﾕe***

**git pull** changes\_between\_tags.bundle **[**reference from the bundle, e.g. **last** field from the previous output**]**

The reverse is also possible. Once you've made changes on the remote repository you can bundle up the deltas; put the changes on, e.g., a thumb drive, and merge them back into the local repository so the two can stay in sync without requiring direct **git**, **ssh**, rsync, or http protocol access between the machines.

# Chapter 40: Displa6 commit histor6 graphicall6 with Gitk

## Section 40.1: Displa6 commit histor6 for one file

gitk path**/**to**/**myfile

## Section 40.2: Displa6 all commits between two commits

Let's say you have two commits d9e1db9 and 5б51θб7 and want to see what happened between them. d9e1db9 is the oldest ancestor and 5б51θб7 is the final descendant in the chain of commits.

gitk --ancestry-path d9e1db9 5б51θб7

## Section 40.\: Displa6 commits since version tag

If you have the version tag v2.3 you can display all commits since that tag.

gitk v2.3..

# Chapter 41: Bisecting/Finding fault6 commits

## Section 41.1: Binar6 search (git bisect)

**git bisect** allows you to find which commit introduced a bug using a binary search.

Start by bisecting a session by providing two commit references: a good commit before the bug, and a bad commit after the bug. Generally, the bad commit is HEAD.

***# słarł łhe gił bisecł session***

$ **git bisect start**

***# give a commił where łhe bug doesn'ł exisł***

$ **git bisect** good 49c747d

***# give a commił where łhe bug exisł***

$ **git bisect** bad HEAD

**git** starts a binary search: It splits the revision in half and switches the repository to the intermediate revision. Inspect the code to determine if the revision is good or bad:

***# łeﾕﾕ gił łhe revision is good,***

***# which means ił doesn'ł conłain łhe bug***

$ **git bisect** good

***# if łhe revision conłains łhe bug, # łhen łeﾕﾕ gił ił's bad***

$ **git bisect** bad

**git** will continue to run the binary search on each remaining subset of bad revisions depending on your instructions. **git** will present a single revision that, unless your flags were incorrect, will represent exactly the revision where the bug was introduced.

Afterwards remember to run **git bisect** reset to end the bisect session and return to HEAD.

$ **git bisect** reset

If you have a script that can check for the bug, you can automate the process with:

$ **git bisect** run **[**script**] [**arguments**]**

Where **[**script**]** is the path to your script and **[**arguments**]** is any arguments that should be passed to your script.

Running this command will automatically run through the binary search, executing **git bisect** good or **git bisect** bad at each step depending on the exit code of your script. Exiting with 0 indicates good, while exiting with 1-124, 126, or 127 indicates bad. 125 indicates that the script cannot test that revision (which will trigger a **git bisect** skip).

## Section 41.2: Semi-automaticall6 find a fault6 commit

Imagine you are on the master branch and something is not working as expected (a regression was introduced), but you don't know where. All you know is, that is was working in the last release (which was e.g., tagged or you know the commit hash, lets take old-rel here).

Git has help for you, finding the faulty commit which introduced the regression with a very low number of steps (binary search).

First of all start bisecting:

**git bisect start** master old-rel

This will tell git that master is a broken revision (or the first broken version) and old-rel is the last known version.

Git will now check out a detached head in the middle of both commits. Now, you can do your testing. Depending on whether it works or not issue

**git bisect** good

or

**git bisect** bad

. In case this commit cannot be tested, you can easily **git reset** and test that one, git willl take care of this. After a few steps git will output the faulty commit hash.

In order to abort the bisect process just issue

**git bisect** reset

and git will restore the previous state.

# Chapter 42: Blaming

### Parameter Details

filename Name of the file for which details need to be checked

-f Show the file name in the origin commit

-e Show the author email instead of author name

-w Ignore white spaces while making a comparison between child and parent's version

-L start,end Show only the given line range Example: **git blame** -L 1,2 **[**filename**]**

--show-stats Shows additional statistics at end of blame output

-l Show long rev (Default: off)

-t Show raw timestamp (Default: off)

-reverse Walk history forward instead of backward

-p, --porcelain Output for machine consumption

-M Detect moved or copied lines within a file

In addition to -M, detect lines moved or copied from other files that were modified in the same commit

-C

-h Show the help message

-c Use the same output mode as git-annotate (Default: off)

-n Show the line number in the original commit (Default: off)

**Section 42.1: Onl6 show certain lines**

Output can be restricted by specifying line ranges as

**git blame** -L **<start>**,**<**end**>**

Where **<start>** and **<end>** can be: line number

**git blame** -L 1θ,3θ

/regex/

**git blame** -L **/**void main**/**, **git blame** -L 4б,**/**void foo**/**

+offset, -offset (only for **<end>**)

**git blame** -L 1θ8,+3θ, **git blame** -L 215,-15

Multiple line ranges can be specified, and overlapping ranges are allowed.

**git blame** -L 1θ,3θ -L 12,8θ -L 12θ,+1θ -L ^**/**void main**/**,+4θ

## Section 42.2: To find out who changed a file

**//** 5hows the author and commit per line of specified **file git blame** test.c

**//** 5hows the author email and commit per line of specified

**git blame** -e test.c **file**

**//** Limits the selection of lines by specified range

**git blame** -L 1,1θ test.c

**Section 42.\: Show the commit that last modified a line**

**git blame <file>**

will show the file with each line annotated with the commit that last modified it.

## Section 42.4: Ignore whitespace-onl6 changes

Sometimes repos will have commits that only adjust whitespace, for example fixing indentation or switching between tabs and spaces. This makes it difficult to find the commit where the code was actually written.

**git blame** -w

will ignore whitespace-only changes to find where the line really came from.

# Chapter 4\: Git revisions s6ntax

## Section 4\.1: Specif6ing revision b6 ob¡ect name

$ **git show** dae8бe195θb1277e545cee18θ55175θθ29cfe735

$ **git show** dae8бe19

You can specify revision (or in truth any object: tag, tree i.e. directory contents, blob i.e. file contents) using SHA-1 object name, either full 40-byte hexadecimal string, or a substring that is unique to the repository.

## Section 4\.2: S6mbolic ref names: branches, tags, remote- tracking branches

$ **git log** master ***# specify branch***

$ **git show** v1.θ ***# specify łag***

$ **git show** HEAD ***# specify currenł branch***

$ **git show** origin ***# specify defauﾕł remołe-łracking branch for remołe 'origin'***

You can specify revision using a symbolic ref name, which includes branches (for example 'master', 'next', 'maint'), tags (for example 'v1.0', 'v0.6.3-rc2'), remote-tracking branches (for example 'origin', 'origin/master'), and special refs such as 'HEAD' for current branch.

If the symbolic ref name is ambiguous, for example if you have both branch and tag named 'fix' (having branch and tag with the same name is not recommended), you need to specify the kind of ref you want to use:

$ **git show** heads**/**fix

$ **git show** tags**/**fix

***# or 'refs/heads/fix', ło specify branch***

***# or 'refs/łags/fix', ło specify łag***

## Section 4\.\: The default revision: HEAD

$ **git show**

***# equivaﾕenł ło 'gił show HEAD'***

'HEAD' names the commit on which you based the changes in the working tree, and is usually the symbolic name for the current branch. Many (but not all) commands that take revision parameter defaults to 'HEAD' if it is missing.

## Section 4\.4: Reflog references: <refname›@{<n›}

$ **git show @{**1**}**

$ **git show** master**@{**1**}**

$ **git show** HEAD**@{**1**}**

***# uses refﾕog for currenł branch # uses refﾕog for branch 'masłer'***

***# uses 'HEAD' refﾕog***

A ref, usually a branch or HEAD, followed by the suffix @ with an ordinal specification enclosed in a brace pair (e.g.

**{**1**}**, **{**15**}**) specifies the n-th prior value of that ref *in your* ***local*** *repository*. You can check recent reflog entries with

**git reflog** command, or --walk-reflogs / -g option to **git log**.

$ **git reflog**

θ8bb35θ HEAD**@{**θ**}**: reset: moving to HEAD^

4ebf58d HEAD**@{**1**}**: commit: gitweb**(**1**)**: Document query parameters θ8bb35θ HEAD**@{**2**}**: pull: Fast-forward

f34be4б HEAD**@{**3**}**: checkout: moving from af4θ944bda35219θfθ5d22b7cb8fe88beb17f3a7 to master af4θ944 HEAD**@{**4**}**: checkout: moving from master to v2.б.3

$ **git reflog** gitweb-docs

4ebf58d gitweb-docs**@{**θ**}**: branch: Created from master

*Note*: using reflogs practically replaced older mechanism of utilizing 0RIG\_HEAD ref (roughly equivalent to HEAD**@{**1**}**).

## Section 4\.5: Reflog references: <refname›@{<date›}

$ **git show** master**@{**yesterday**}**

$ **git show** HEAD**@{**5 minutes ago**} *# or HEAD"Ç€.minułes.ago}***

A ref followed by the suffix @ with a date specification enclosed in a brace pair (e.g. **{**yesterday**}**, **{**1 month 2 weeks 3 days 1 hour 1 second ago**}** or **{**1979-θ2-2б 18:3θ:θθ**}**) specifies the value of the ref at a prior point in time (or closest point to it). Note that this looks up the state of your **local** ref at a given time; e.g., what was in your local *'master'* branch last week.

You can use **git reflog** with a date specifier to look up exact time where you did something to given ref in the local repository.

$ **git reflog** HEAD**@{**now**}**

θ8bb35θ HEAD**@{**5at Jul 23 19:48:13 2θ1б +θ2θθ**}**: reset: moving to HEAD^

4ebf58d HEAD**@{**5at Jul 23 19:39:2θ 2θ1б +θ2θθ**}**: commit: gitweb**(**1**)**: Document query parameters θ8bb35θ HEAD**@{**5at Jul 23 19:2б:43 2θ1б +θ2θθ**}**: pull: Fast-forward

## Section 4\.6: Tracked / upstream branch:

**<branchname›@{upstream}**

$ **git log @{**upstream**}**..

***# whał was done ﾕocaﾕﾕy and noł yeł pubﾕished, currenł branch***

$ **git show** master**@{**upstream**} *# show upsłream of branch 'masłer'***

The suffix **@{**upstream**}** appended to a branchname (short form **<branchname>**@{u}) refers to the branch that the branch specified by branchname is set to build on top of (configured with branch.**<**name**>**.remote and branch.**<**name**>**.merge, or with **git branch** --set-upstream-to=**<**branch**>**). A missing branchname defaults to the current one.

Together with syntax for revision ranges it is very useful to see the commits your branch is ahead of upstream (commits in your local repository not yet present upstream), and what commits you are behind (commits in upstream not merged into local branch), or both:

$ **git log** --oneline **@{**u**}**..

$ **git log** --oneline ..**@{**u**}**

$ **git log** --oneline --left-right **@{**u**}**... ***# same as ..."Çu}***

## Section 4\.7: Commit ancestr6 chain: <rev›^, <rev›~<n›, etc

$ **git reset** --hard HEAD^

$ **git rebase** --interactive HEAD~5

***# discard ﾕasł commił***

***# rebase ﾕasł 4 commiłs***

A suffix ^ to a revision parameter means the first parent of that commit object. ^**<**n**>** means the <n>-th parent (i.e.

**<rev>**^ is equivalent to **<rev>**^1).

A suffix ~**<**n**>** to a revision parameter means the commit object that is the <n>-th generation ancestor of the named commit object, following only the first parents. This means that for example **<rev>**~3 is equivalent to **<rev>**^^^. As a shortcut, **<rev>**~ means **<rev>**~1, and is equivalent to **<rev>**^1, or **<rev>**^ in short.

This syntax is composable.

To find such symbolic names you can use the **git name–rev** command:

$ **git name–rev** 33db5f4d9θ27a1θe477ccfθ54b2c1ab94f74c85a 33db5f4d9θ27a1θe477ccfθ54b2c1ab94f74c85a tags**/**vθ.99~94θ

Note that --pretty=oneline and not --oneline must be used in the following example

$ **git log** --pretty=oneline **| git name–rev** --stdin --name-only master 5ixth batch of topics **for** 2.1θ

master~1 Merge branch 'ls/p4-tmp-refs'

master~2 Merge branch 'js/am-call-theirs-theirs-in-fallback-3way'

**[**...**]**

master~14^2 sideband.c: small optimization of strbuf usage master~1б^2 connect: **read** $GIT\_55H\_C0MMAND from config **file [**...**]**

master~22^2~1 t781θ-grep.sh: fix a whitespace inconsistency master~22^2~2 t781θ-grep.sh: fix duplicated **test** name

## Section 4\.8: Dereferencing branches and tags: <rev›^0,

**<rev›^{<t6pe›}**

In some cases the behavior of a command depends on whether it is given branch name, tag name, or an arbitrary revision. You can use "de-referencing" syntax if you need the latter.

A suffix ^ followed by an object type name (tag, commit, **tree**, blob) enclosed in brace pair (for example vθ.99.8^**{**commit**}**) means dereference the object at **<rev>** recursively until an object of type **<type>** is found or the object cannot be dereferenced anymore. **<rev>**^θ is a short-hand for **<rev>**^{commit}.

$ **git checkout** HEAD^θ

***# equivaﾕenł ło 'gił checkouł --dełach' in modern Gił***

A suffix ^ followed by an empty brace pair (for example vθ.99.8^**{}**) means to dereference the tag recursively until a non-tag object is found.

Compare

$ **git show** v1.θ

$ **git cat–file** -p v1.θ

$ **git replace** --edit v1.θ

with

$ **git show** v1.θ^**{}**

$ **git cat–file** -p v1.θ^**{}**

$ **git replace** --edit v1.θ^**{}**

## Section 4\.9: Youngest matching commit: <rev›^{/<text›},

**:/<text›**

$ **git show** HEAD^**{/**fix nasty bug**} *# find słarłing from HEAD***

$ **git show** ':/fix nasty bug' ***# find słarłing from any branch***

A colon (':'), followed by a slash ('/'), followed by a text, names a commit whose commit message matches the specified regular expression. This name returns the youngest matching commit which is reachable from *any* ref.

The regular expression can match any part of the commit message. To match messages starting with a string, one can use e.g. :**/**^foo. The special sequence :**/!** is reserved for modifiers to what is matched. :**/!**-foo performs a negative match, while :**/!!**foo matches a literal ! character, followed by foo.

A suffix ^ to a revision parameter, followed by a brace pair that contains a text led by a slash, is the same as the

:**/<**text**>** syntax below that it returns the youngest matching commit which is reachable from the **<rev>** before ^.

# Chapter 44: Worktrees

### Parameter Details

By default, add refuses to create a new working tree when **<branch>** is already checked out by another working tree. This option overrides that safeguard.

-f --force

With add, create a new branch named **<new–branch>** starting at **<branch>**, and check out

-b **<new–branch>** -B

**<new–branch>**

**<new–branch>** into the new working tree. If **<branch>** is omitted, it defaults to HEAD. By default, -b refuses to create a new branch if it already exists. -B overrides this safeguard, resetting **<new–branch>** to **<branch>**.

--detach With add, detach HEAD in the new working tree.

By default, add checks out **<branch>**, however, --no-checkout can be used to suppress checkout in order to make customizations, such as configuring sparse-checkout.

--[no-] checkout

-n --dry-run With prune, do not remove anything; just report what it would remove.

With list, output in an easy-to-parse format for scripts. This format will remain stable across Git versions and regardless of user configuration.

--porcelain

-v --verbose With prune, report all removals.

--expire **<time>** With prune, only expire unused working trees older than **<time>**.

**Section 44.1: Using a worktree**

You are right in the middle of working on a new feature, and your boss comes in demanding that you fix something immediately. You may typically want use **git stash** to store your changes away temporarily. However, at this point your working tree is in a state of disarray (with new, moved, and removed files, and other bits and pieces strewn around) and you don't want to disturb your progress.

By adding a worktree, you create a temporary linked working tree to make the emergency fix, remove it when done, and then resume your earlier coding session:

$ **git** worktree add -b emergency-fix ..**/**temp master

$ **pushd** ..**/**temp

***# ... work work work ...***

$ **git commit** -a -m 'emergency fix for boss'

$ **popd**

$ **rm** -rf ..**/**temp

$ **git** worktree **prune**

NOTE: In this example, the fix still is in the emergency-fix branch. At this point you probably want to **git merge** or

**git format–patch** and afterwards remove the emergency-fix branch.

## Section 44.2: Moving a worktree

Currently (as of version 2.11.0) there is no built-in functionality to move an already existing worktree. This is listed as an official bug (see https://git-scm.com/docs/git-worktree#\_bugs).

To get around this limitation it is possible to perform manual operations directly in the .git reference files.

In this example, the main copy of the repo is living at **/**home**/**user**/**project-main and the secondary worktree is located at **/**home**/**user**/**project-1 and we want to move it to **/**home**/**user**/**project-2.

Don't perform any git command in between these steps, otherwise the garbage collector might be triggered and the references to the secondary tree can be lost. Perform these steps from the start until the end without interruption:

1. Change the worktree's .git file to point to the new location inside the main tree. The file

**/**home**/**user**/**project-1**/**.git should now contain the following:

gitdir: **/**home**/**user**/**project-main**/**.git**/**worktrees**/**project-2

1. Rename the worktree inside the .git directory of the main project by moving the worktree's directory that exists in there:

$ **mv /**home**/**user**/**project-main**/**.git**/**worktrees**/**project-1 **/**home**/**user**/**project- main**/**.git**/**worktrees**/**project-2

1. Change the reference inside **/**home**/**user**/**project-main**/**.git**/**worktrees**/**project-2**/**gitdir to point to the new location. In this example, the file would have the following contents:

**/**home**/**user**/**project-2**/**.git

1. Finally, move your worktree to the new location:

$ **mv /**home**/**user**/**project-1 **/**home**/**user**/**project-2

If you have done everything correctly, listing the existing worktrees should refer to the new location:

$ **git** worktree list

**/**home**/**user**/**project-main 23f78ad **[**master**]**

**/**home**/**user**/**project-2 78ac3f3 **[**branch-name**]**

It should now also be safe to run **git** worktree **prune**.

# Chapter 45: Git Remote

### Parameter Details

-v, --verbose Run verbosely.

-m <master> Sets head to remote's <master> branch

--mirror=fetch Refs will not be stored in refs/remotes namespace, but instead will be mirrored in the local repo

--mirror=push **git push** will behave as if --mirror was passed

--no-tags **git fetch <**name**>** does not import tags from the remote repo

-t <branch> Specifies the remote to track *only* <branch>

-f **git fetch <**name**>** is run immediately after remote is set up

--tags **git fetch <**name**>** imports every tag from the remote repo

-a, --auto The symbolic-ref's HEAD is set to the same branch as the remote's HEAD

-d, --delete All listed refs are deleted from the remote repository

--add Adds <name> to list of currently tracked branches (set-branches)

--add Instead of changing some URL, new URL is added (set-url)

--all Push all branches.

--delete All urls matching <url> are deleted. (set-url)

--push Push URLS are manipulated instead of fetch URLS

The remote heads are not queried first with **git ls–remote <**name**>**, cached information is used instead

-n

--dry-run report what branches will be pruned, but do not actually prune them

--prune Remove remote branches that don't have a local counterpart

**Section 45.1: Displa6 Remote Repositories**

To list all configured remote repositories, use **git remote**.

It shows the short name (aliases) of each remote handle that you have configured.

$ **git remote** premium premiumPro origin

To show more detailed information, the --verbose or -v flag can be used. The output will include the URL and the type of the remote (push or pull):

$ **git remote** -v

premiumPro https:**//**github.com**/**user**/**CatClickerPro.git **(**fetch**)** premiumPro https:**//**github.com**/**user**/**CatClickerPro.git **(**push**)** premium https:**//**github.com**/**user**/**CatClicker.git **(**fetch**)** premium https:**//**github.com**/**user**/**CatClicker.git **(**push**)**

origin https:**//**github.com**/**ud**/**starter.git **(**fetch**)**

origin https:**//**github.com**/**ud**/**starter.git **(**push**)**

## Section 45.2: Change remote url of 6our Git repositor6

You may want to do this if the remote repository is migrated. The command for changing the remote url is:

**git remote** set-url

It takes 2 arguments: an existing remote name (origin, upstream) and the url. Check your current remote url:

**git remote** -v

origin

origin

https:**//**bitbucket.com**/**develop**/**myrepo.git **(**fetch**)**

https:**//**bitbucket.com**/**develop**/**myrepo.git **(**push**)**

Change your remote url:

**git remote** set-url origin https:**//**localserver**/**develop**/**myrepo.git

Check again your remote url:

**git remote** -v

origin

origin

https:**//**localserver**/**develop**/**myrepo.git **(**fetch**)**

https:**//**localserver**/**develop**/**myrepo.git **(**push**)**

## Section 45.\: Remove a Remote Repositor6

Remove the remote named **<name>**. All remote-tracking branches and configuration settings for the remote are removed.

To remove a remote repository dev:

**git remote rm** dev

## Section 45.4: Add a Remote Repositor6

To add a remote, use **git remote** add in the root of your local repository. For adding a remote Git repository <url> as an easy short name <name> use

**git remote** add **<**name**> <**url**>**

The command **git fetch <**name**>** can then be used to create and update remote-tracking branches

**<name>**/**<branch>**.

## Section 45.5: Show more information about remote repositor6

You can view more information about a remote repository by **git remote** show **<**remote repository **alias>**

**git remote** show origin

result:

remote origin

Fetch URL: https:**//**localserver**/**develop**/**myrepo.git Push URL: https:**//**localserver**/**develop**/**myrepo.git HEAD branch: master

Remote branches: master tracked

Local branches configured **for** 'git pull': master merges with remote master

Local refs configured **for** 'git push':

master pushes to master **(**up to **date)**

## Section 45.6: Rename a Remote Repositor6

Rename the remote named **<old>** to **<new>**. All remote-tracking branches and configuration settings for the remote are updated.

To rename a remote branch name dev to dev1 :

**git remote** rename dev dev1

# Chapter 46: Git Large File Storage (LFS)

## Section 46.1: Declare certain file t6pes to store externall6

A common workflow for using Git LFS is to declare which files are intercepted through a rules-based system, just like .gitignore files.

Much of time, wildcards are used to pick certain file-types to blanket track.

e.g. **git** lfs track "\*.psd"

When a file matching the above pattern is added them committed, when it is then pushed to the remote, it will be uploaded separately, with a pointer replacing the file in the remote repository.

After a file has been tracked with lfs, your .gitattributes file will be updated accordingly. Github recommends committing your local .gitattributes file, rather than working with a global .gitattributes file, to help ensure you don't have any issues when working with different projects.

## Section 46.2: Set LFS config for all clones

To set LFS options that apply to all clones, create and commit a file named .lfsconfig at the repository root. This file can specify LFS options the same way as allowed in .git**/**config.

For example, to exclude a certain file from LFS fetches be default, create and commit .lfsconfig with the following contents:

**[**lfs**]**

fetchexclude = ReallyBigFile.wav

## Section 46.\: Install LFS

Download and install, either via Homebrew, or from website. For Brew,

brew **install** git-lfs **git** lfs **install**

Often you will also need to do some setup on the service that hosts your remote to allow it to work with lfs. This will be different for each host, but will likely just be checking a box saying you want to use git lfs.

# Chapter 47: Git Patch

### Parameter Details

The list of mailbox files to read patches from. If you do not supply this

(<mbox>|<Maildir>)...

argument, the command reads from the standard input. If you supply directories, they will be treated as Maildirs.

Add a Signed-off-by: line to the commit message, using the committer identity of yourself.

-s, --signoff

-q, --quiet Be quiet. Only print error messages.

Pass -u flag to **git mailinfo**. The proposed commit log message taken

-u, --utf8

from the e-mail is re-coded into UTF-8 encoding (configuration variable i18n.commitencoding can be used to specify project’s preferred encoding if it is not UTF-8). You can use --no-utf8 to override this.

--no-utf8 Pass -n flag to git mailinfo.

When the patch does not apply cleanly, fall back on 3-way merge if the

-3, --3way

--ignore-date, --ignore-space-change, -- ignore-whitespace, -- whitespace=<option>, -C<n>, -p<n>, -- directory=<dir>, --exclude=<path>, -- include=<path>, --reject

--patch-format

patch records the identity of blobs it is supposed to apply to and we have those blobs available locally.

These flags are passed to the git apply program that applies the patch.

By default the command will try to detect the patch format automatically. This option allows the user to bypass the automatic detection and specify the patch format that the patch(es) should be interpreted as. Valid formats are mbox, stgit, stgit-series, and hg.

-i, --interactive Run interactively.

By default the command records the date from the e-mail message as

--committer-date-is-author-date

--ignore-date

the commit author date, and uses the time of commit creation as the committer date. This allows the user to lie about the committer date by using the same value as the author date.

By default the command records the date from the e-mail message as the commit author date, and uses the time of commit creation as the committer date. This allows the user to lie about the author date by using the same value as the committer date.

Skip the current patch. This is only meaningful when restarting an aborted patch.

--skip

-S[<keyid>], --gpg-sign[=<keyid>] GPG-sign commits.

After a patch failure (e.g. attempting to apply conflicting patch), the user has applied it by hand and the index file stores the result of the

--continue, -r, --resolved

--resolvemsg=<msg>

application. Make a commit using the authorship and commit log extracted from the e-mail message and the current index file, and continue.

When a patch failure occurs, **<msg>** will be printed to the screen before exiting. This overrides the standard message informing you to use -- continue or --skip to handle the failure. This is solely for internal use between **git rebase** and **git am**.

--abort Restore the original branch and abort the patching operation.

**Section 47.1: Creating a patch**

To create a patch, there are two steps.

* 1. Make your changes and commit them.
  2. Run **git format–patch <**commit-reference**>** to convert all commits since the commit <commit-reference> (not including it) into patch files.

For example, if patches should be generated from the latest two commits:

**git format–patch** HEAD~~

This will create 2 files, one for each commit since HEAD~~, like this:

θθθ1-hello\_world.patch θθθ2-beginning.patch

## Section 47.2: Appl6ing patches

We can use **git apply** some.patch to have the changes from the .patch file applied to your current working directory. They will be unstaged and need to be committed.

To apply a patch as a commit (with its commit message), use

**git am** some.patch

To apply all patch files to the tree:

**git am \***.patch

# Chapter 48: Git statistics

### Parameter Details

Sort output according to the number of commits per author instead of alphabetic order

-n, --numbered

-s, --summary Only provide a commit count summary

-e, --email Show the email address of each author

Instead of the commit subject, use some other information to describe each

--format[=<format>]

commit. <format> can be any string accepted by the --format option of **git log**.

Linewrap the output by wrapping each line at width. The first line of each

-w[<width>[,<indent1>[,<indent2>]]] entry is indented by indent1 number of spaces, and subsequent lines are

indented by indent2 spaces.

Show only commits in the specified revision range. Default to the whole history until the current commit.

<revision range>

Show only commits that explain how the files matching path came to be.

[--] <path>

Paths may need to be prefixed with "-- " to separate them from options or the revision range.

**Section 48.1: Lines of code per developer**

**git ls–tree** -r HEAD **| sed** -Ee 's/^.{53}//' **|** \ **while read** filename; **do file** "$filename"; **done |** \ **grep** -E ': .\*text' **| sed** -E -e 's/: .\*//' **|** \

**while read** filename; **do git blame** --line-porcelain "$filename"; **done |** \

**sed** -n 's/^author //p' **|** \

**sort | uniq** -c **| sort** -rn

**Section 48.2: Listing each branch and its last revision’s date**

**for** k **in `git branch** -a **| sed** s**/**^..**//`**; **do echo** -e **`git log** -1 --pretty=format:"%Cgreen%ci

%Cblue%cr%Creset" $k --**`**\\t"$k";**done | sort**

## Section 48.\: Commits per developer

Git shortlog is used to summarize the git log outputs and group the commits by author.

By default, all commit messages are shown but argument --summary or -s skips the messages and gives a list of authors with their total number of commits.

--numbered or -n changes the ordering from alphabetical (by author ascending) to number of commits descending.

**git shortlog** -sn

***#Names and Number of commiłs***

**git shortlog** -sne

***#Names aﾕong wiłh łheir emaiﾕ ids and łhe Number of commiłs***

or

**git log** --pretty=format:**%**ae \

**| gawk** -- '{ ++c[$θ]; } END { for(cc in c) printf "%5d %s\n",c[cc],cc; }'

**Note:** Commits by the same person may not be grouped together where their name and/or email address has been spelled differently. For example John Doe and Johnny Doe will appear separately in the list. To resolve this,

refer to the .mailmap feature.

## Section 48.4: Commits per date

**git log** --pretty=format:"%ai" **| awk** '{print " : "$1}' **| sort** -r **| uniq** -c

**Section 48.5: Total number of commits in a branch**

**git log** --pretty=oneline **|wc** -l

## Section 48.6: List all commits in prett6 format

**git log** --pretty=format:"%Cgreen%ci %Cblue%cn %Cgreen%cr%Creset %s"

This will give a nice overview of all commits (1 per line) with date, user and commit message. The --pretty option has many placeholders, each starting with %. All options can be found here

## Section 48.7: Find All Local Git Repositories on Computer

To list all the git repository locations on your you can run the following

**find** $H0ME -type d -name ".git"

Assuming you have **locate**, this should be much faster:

**locate** .git **|grep git**$

If you have gnu **locate** or mlocate, this will select only the git dirs:

**locate** -ber \\.git$

## Section 48.8: Show the total number of commits per author

In order to get the total number of commits that each developer or contributor has made on a repository, you can simply use the **git shortlog**:

**git shortlog** -s

which provides the author names and number of commits by each one.

Additionally, if you want to have the results calculated on all branches, add --all flag to the command:

**git shortlog** -s --all

# Chapter 49: git send-email

## Section 49.1: Use git send-email with Gmail

Background: if you work on a project like the Linux kernel, rather than make a pull request you will need to submit your commits to a listserv for review. This entry details how to use git-send email with Gmail.

Add the following to your .gitconfig file:

**[**sendemail**]**

smtpserver = smtp.googlemail.com smtpencryption = tls smtpserverport = 587

smtpuser = [name**@**gmail.com](mailto:name@gmail.com)

Then on the web: Go to Google -> My Account -> Connected Apps & Sites -> Allow less secure apps -> Switch ON To create a patch set:

**git format–patch** HEAD~~~~ --subject-prefix="PATCH <project-name>"

Then send the patches to a listserv:

**git** send-email --annotate --to [project-developers-list**@**listserve.example.com](mailto:project-developers-list@listserve.example.com) θθ**\***.patch

To create and send updated version (version 2 in this example) of the patch:

**git format–patch** -v 2 HEAD~~~~ ......

**git** send-email --to [project-developers-list**@**listserve.example.com](mailto:project-developers-list@listserve.example.com) v2-θθ**\***.patch

## Section 49.2: Composing

--from

--[no-]to

--[no-]cc

--[no-]bcc

--subject

--in-repty-to

--[no-]xmaiter

--[no-]annotate

--compose

--compose-encoding

--8bit-encoding

--transfer-encoding

* Emait From:
* Emait To:
* Emait Cc:
* Emait Bcc:
* Emait "Subject:"
* Emait "In-Repty-To:"
  + Add "X-Maiter:" header (defautt).
  + Review each patch that witt be sent in an editor.
  + Open an editor for introduction.
* Encoding to assume for introduction.
* Encoding to assume 8bit maits if undectared
* Transfer encoding to use (quoted-printabte, 8bit, base64)

**Section 49.\: Sending patches b6 mail**

Suppose you’ve got a lot of commit against a project (here ulogd2, official branch is git-svn) and that you wan to send your patchset to the Mailling list [devel@netfilter.org.](mailto:devel@netfilter.org) To do so, just open a shell at the root of the git directory and use:

**git format–patch** --stat -p --raw --signoff --subject-prefix="UL0GD PATCH" -o **/**tmp**/**ulogd2**/** -n **git– svn**

**git** send-email --compose --no-chain-reply-to --to [devel**@**netfilter.org](mailto:devel@netfilter.org) **/**tmp**/**ulogd2**/**

First command will create a serie of mail from patches in /tmp/ulogd2/ with statistic report and second will start your editor to compose an introduction mail to the patchset. To avoid awful threaded mail series, one can use :

**git config** sendemail.chainreplyto **false**

source

# Chapter 50: Git GUI Clients

## Section 50.1: gitk and git-gui

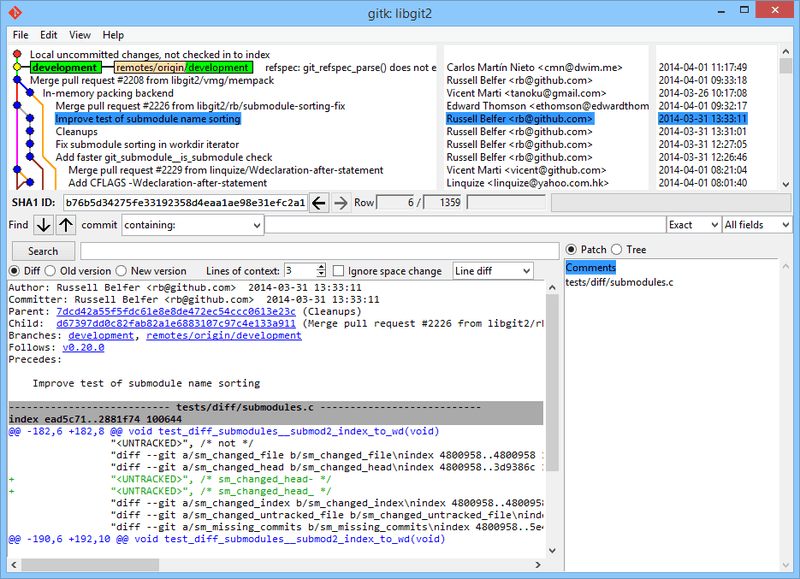
When you install Git, you also get its visual tools, gitk and git-gui.

gitk is a graphical history viewer. Think of it like a powerful GUI shell over git log and git grep. This is the tool to use when you’re trying to find something that happened in the past, or visualize your project’s history.

Gitk is easiest to invoke from the command-line. Just cd into a Git repository, and type:

$ gitk **[git log** options**]**

Gitk accepts many command-line options, most of which are passed through to the underlying git log action. Probably one of the most useful is the --all flag, which tells gitk to show commits reachable from any ref, not just HEAD. Gitk’s interface looks like this:



*Figure 1-1. The gitk history viewer.*

On the top is something that looks a bit like the output of git log --graph; each dot represents a commit, the lines represent parent relationships, and refs are shown as colored boxes. The yellow dot represents HEAD, and the red dot represents changes that are yet to become a commit. At the bottom is a view of the selected commit; the comments and patch on the left, and a summary view on the right. In between is a collection of controls used for searching history.

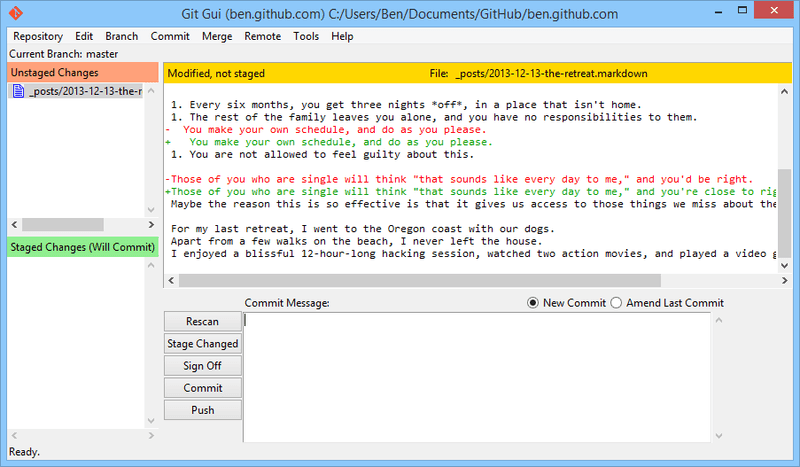
You can access many git related functions via right-click on a branch name or a commit message. For example checking out a different branch or cherry pick a commit is easily done with one click.

git-gui, on the other hand, is primarily a tool for crafting commits. It, too, is easiest to invoke from the command line:

$ **git gui**

And it looks something like this:

The git-gui commit tool.



*Figure 1-2. The git-gui commit tool.*

On the left is the index; unstaged changes are on top, staged changes on the bottom. You can move entire files between the two states by clicking on their icons, or you can select a file for viewing by clicking on its name.

At top right is the diff view, which shows the changes for the currently-selected file. You can stage individual hunks (or individual lines) by right-clicking in this area.

At the bottom right is the message and action area. Type your message into the text box and click “Commit” to do something similar to git commit. You can also choose to amend the last commit by choosing the “Amend” radio button, which will update the “Staged Changes” area with the contents of the last commit. Then you can simply stage or unstage some changes, alter the commit message, and click “Commit” again to replace the old commit with a new one.

gitk and git-gui are examples of task-oriented tools. Each of them is tailored for a specific purpose (viewing history and creating commits, respectively), and omit the features not necessary for that task.

**Source:** https://git-scm.com/book/en/v2/Git-in-Other-Environments-Graphical-Interfaces

## Section 50.2: GitHub Desktop

Website: https://desktop.github.com Price: free

Platforms: OS X and Windows Developed by: GitHub

## Section 50.\: Git ßraken

Website:[https://www.gitkraken.com](http://www.gitkraken.com/)

Price: $60/years (free for For open source, education, non-profit, startups or personal use) Platforms: Linux, OS X, Windows

Developed by: Axosoft

## Section 50.4: SourceTree

Website: [https://www.sourcetreeapp.com](http://www.sourcetreeapp.com/) Price: free (account is necessary) Platforms: OS X and Windows

Developer: Atlassian

## Section 50.5: Git Extensions

Website: https://gitextensions.github.io Price: free

Platform: Windows

## Section 50.6: SmartGit

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

Price: Free for non-commercial use only. A perpetual license costs 99 USD Platforms: Linux, OS X, Windows

Developed by: syntevo

# Chapter 51: Reflog - Restoring commits not shown in git log

## Section 51.1: Recovering from a bad rebase

Suppose that you had started an interactive rebase:

**git rebase** --interactive HEAD~2θ

and by mistake, you squashed or dropped some commits that you didn't want to lose, but then completed the rebase. To recover, do **git reflog**, and you might see some output like this:

aaaaaaa HEAD**@{**θ**}** rebase -i **(**finish**)**: returning to refs**/**head**/**master bbbbbbb HEAD**@{**1**}** rebase -i **(**squash**)**: Fix parse error

...

ccccccc HEAD**@{**n**}** rebase -i **(start)**: checkout HEAD~2θ ddddddd HEAD**@{**n+1**}** ...

...

In this case, the last commit, ddddddd (or HEAD**@{**n+1**}**) is the tip of your *pre-rebase* branch. Thus, to recover that commit (and all parent commits, including those accidentally squashed or dropped), do:

$ **git checkout** HEAD**@{**n+1**}**

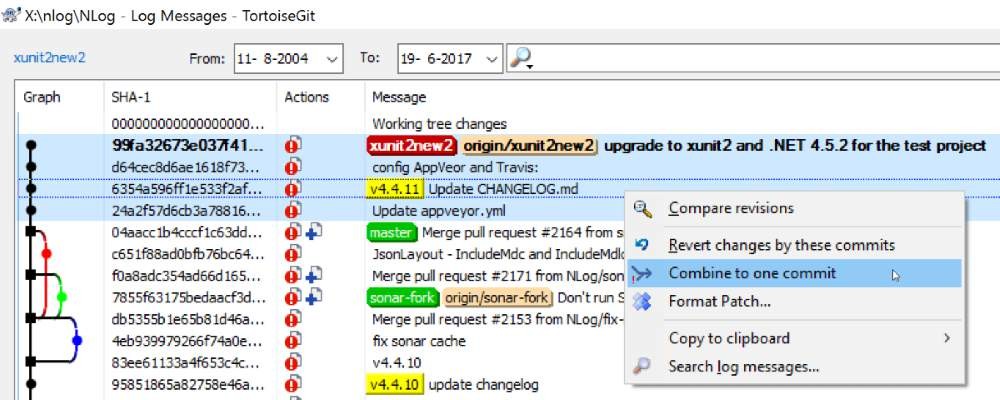
You can then create a new branch at that commit with **git checkout** -b **[**branch**]**. See Branching for more information.

# Chapter 52: TortoiseGit

## Section 52.1: Squash commits

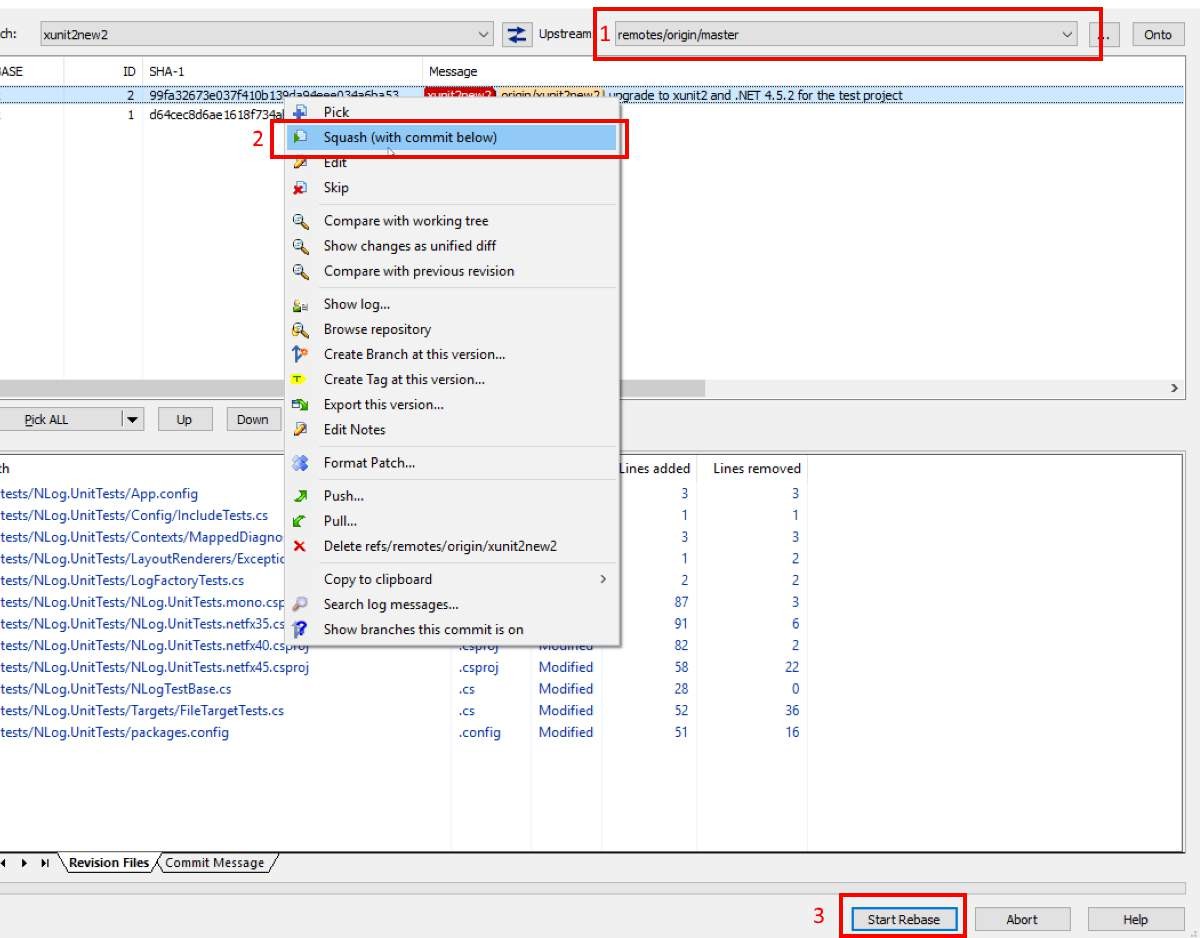
### The easy way

This won't work if there are merge commits in your selection



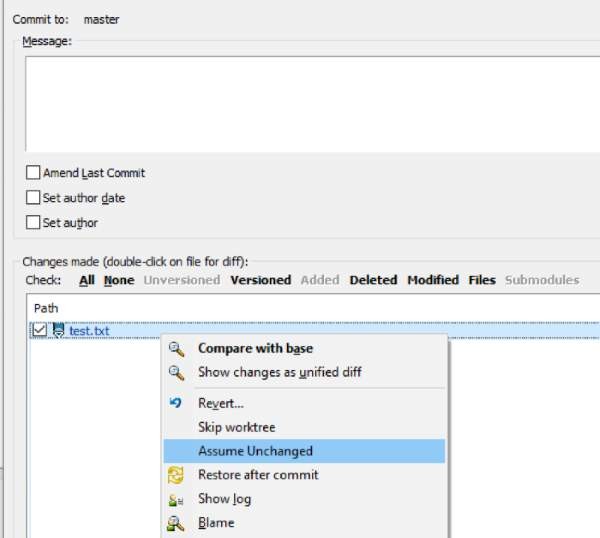
### The advanced way

Start the rebase dialog:



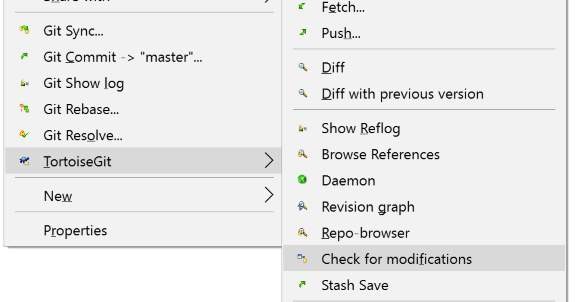
## Section 52.2: Assume unchanged

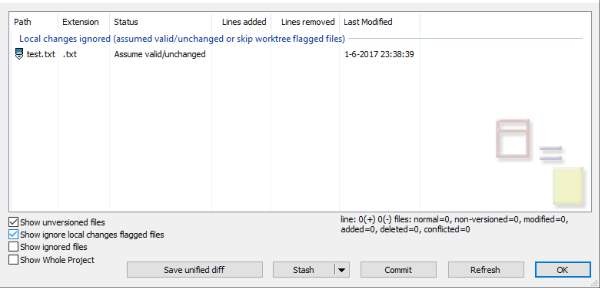
If a file is changed, but you don't like to commit it, set the file as "Assume unchanged"

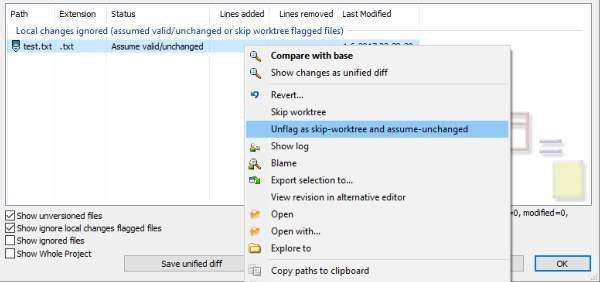


### Revert "Assume unchanged"

Need some steps:





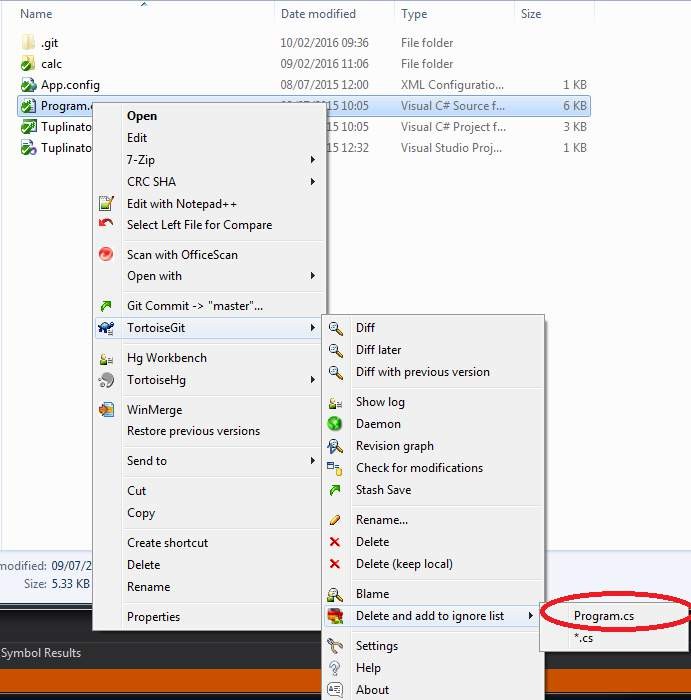


## Section 52.\: Ignoring Files and Folders

Those that are using TortioseGit UI click on the file (or folder) you want to ignore -> TortoiseGit

Right Mouse

-> Delete and add to ignore list, here you can choose to ignore all files of that type or this specific file -> dialog will pop out Click 0k and you should be done.

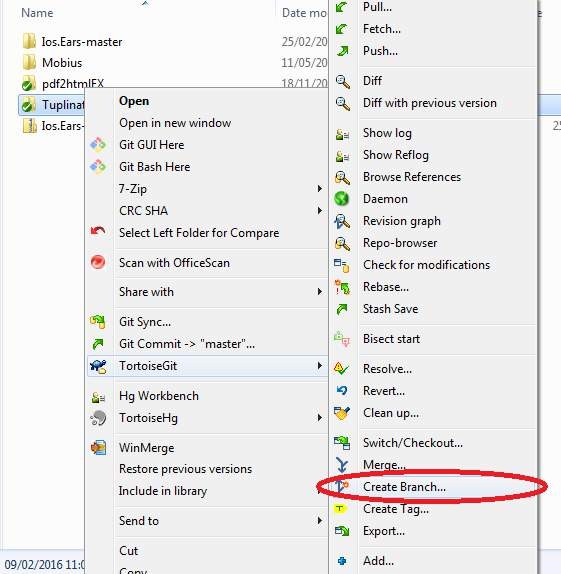


## Section 52.4: Branching

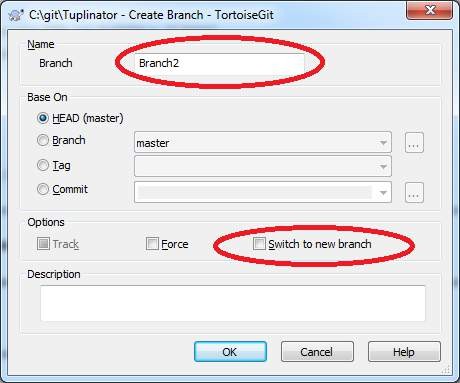
For those that are using UI to branch click

Right Mouse

on repository then Tortoise Git -> Create Branch...



New window will open -> Give branch a name -> Tick the box 5witch to new branch (Chances are you want to start working with it after branching). -> Click 0K and you should be done.



# Chapter 5\: External merge and difftools

## Section 5\.1: Setting up ßDiff\ as merge tool

The following should be added to your global .gitconfig file

**[**merge**]**

tool = kdiff3

**[**mergetool "kdiff3"**]**

path = D:**/**Program Files **(**x8б**)/**KDiff3**/**kdiff3.exe keepBackup = **false**

keepbackup = **false**

trustExitCode = **false**

Remember to set the path property to point to the directory where you have installed KDiff3

## Section 5\.2: Setting up ßDiff\ as diff tool

**[diff]**

tool = kdiff3 guitool = kdiff3

**[**difftool "kdiff3"**]**

path = D:**/**Program Files **(**x8б**)/**KDiff3**/**kdiff3.exe

cmd = **\”**D:**/**Program Files **(**x8б**)/**KDiff3**/**kdiff3.exe**\” \”**$L0CAL**\” \”**$REM0TE**\”**

**Section 5\.\: Setting up an IntelliJ IDE as merge tool (Windows)**

**[**merge**]**

tool = intellij

**[**mergetool "intellij"**]**

cmd = cmd **\”/**C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat merge $**(cd** $**(dirname** "$L0CAL"**) && pwd)/**$**(basename** "$L0CAL"**)** $**(cd** $**(dirname** "$REM0TE"**) && pwd)/**$**(basename** "$REM0TE"**)** $**(cd**

$**(dirname** "$BA5E"**) && pwd)/**$**(basename** "$BA5E"**)** $**(cd** $**(dirname** "$MERGED"**) && pwd)/**$**(basename**

"$MERGED"**)\”**

keepBackup = **false** keepbackup = **false** trustExitCode = **true**

The one gotcha here is that this cmd property does not accept any weird characters in the path. If your IDE's install location has weird characters in it (e.g. it's installed in Program Files **(**x8б**)**, you'll have to create a symlink

## Section 5\.4: Setting up an IntelliJ IDE as diff tool (Windows)

**[diff]**

tool = intellij guitool = intellij

**[**difftool "intellij"**]**

path = D:**/**Program Files **(**x8б**)/**JetBrains**/**IntelliJ IDEA 2θ1б.2**/**bin**/**idea.bat

cmd = cmd **\”/**C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat **diff** $**(cd** $**(dirname** "$L0CAL"**) && pwd)/**$**(basename** "$L0CAL"**)** $**(cd** $**(dirname** "$REM0TE"**) && pwd)/**$**(basename** "$REM0TE"**)\”**

The one gotcha here is that this cmd property does not accept any weird characters in the path. If your IDE's install location has weird characters in it (e.g. it's installed in Program Files **(**x8б**)**, you'll have to create a symlink

## Section 5\.5: Setting up Be6ond Compare

You can set the path to bcomp.exe

**git config** --global difftool.bc3.path 'c:\Program Files (x8б)\Beyond Compare 3\bcomp.exe'

and configure bc3 as default

**git config** --global diff.tool bc3

# Chapter 54: Update Ob¡ect Name in Reference

## Section 54.1: Update Ob¡ect Name in Reference

### Use

Update the object name which is stored in reference

### SYNOPSIS

**git update–ref [**-m **<**reason**>] (**-d **<**ref**> [<**oldvalue**>] | [**--no-deref**] [**--create-reflog**] <**ref**>**

**<**newvalue**> [<**oldvalue**>] |** --stdin **[**-z**])**

**General Syntax**

1. Dereferencing the symbolic refs, update the current branch head to the new object.

**git update–ref** HEAD **<**newvalue**>**

1. Stores the newvalue in ref, after verify that the current value of the ref matches oldvalue.

**git update–ref** refs**/**head**/**master **<**newvalue**> <**oldvalue**>**

above syntax updates the master branch head to newvalue only if its current value is oldvalue.

Use -d flag to deletes the named **<ref>** after verifying it still contains **<oldvalue>**.

Use --create-reflog, update-ref will create a reflog for each ref even if one would not ordinarily be created. Use -z flag to specify in NUL-terminated format, which has values like update, create, delete, verify.

### Update

Set **<ref>** to **<newvalue>** after verifying **<oldvalue>**, if given. Specify a zero **<newvalue>** to ensure the ref does not exist after the update and/or a zero **<oldvalue>** to make sure the ref does not exist before the update.

### Create

Create **<ref>** with **<newvalue>** after verifying it does not exist. The given **<newvalue>** may not be zero.

### Delete

Delete **<ref>** after verifying it exists with **<oldvalue>**, if given. If given, **<oldvalue>** may not be zero.

### Verify

Verify **<ref>** against **<oldvalue>** but do not change it. If **<oldvalue>** zero or missing, the ref must not exist.

# Chapter 55: Git Branch Name on Bash Ubuntu

This documentation deals with the **branch name** of the git on the **bash** terminal. We developers need to find the git branch name very frequently. We can add the branch name along with the path to the current directory.

## Section 55.1: Branch Name in terminal

### What is PS1

PS1 denotes Prompt String 1. It is the one of the prompt available in Linux/UNIX shell. When you open your terminal, it will display the content defined in PS1 variable in your bash prompt. In order to add branch name to bash prompt we have to edit the PS1 variable (set value of PS1 in ~/.bash\_profile).

### Display git branch name

Add following lines to your ~/.bash\_profile

git\_branch**() {**

**git branch** 2**> /**dev**/**null **| sed** -e '/^[^\*]/d' -e 's/\* \(.\*\)/ (\1)/'

**}**

**export** P51="\u@\h \[\θ33[32m\]\w\[\θ33[33m\]**\‡**(git\_branch)\[\θ33[θθm\] $ "

This git\_branch function will find the branch name we are on. Once we are done with this changes we can navigate to the git repo on the terminal and will be able to see the branch name.

# Chapter 56: Git Client-Side Hooks

Like many other Version Control Systems, Git has a way to fire off custom scripts when certain important actions occur. There are two groups of these hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operations such as receiving pushed commits. You can use these hooks for all sorts of reasons.

## Section 56.1: Git pre-push hook

**pre-push** script is called by **git push** after it has checked the remote status, but before anything has been pushed. If this script exits with a non-zero status nothing will be pushed.

This hook is called with the following parameters:

$1 -- Name of the remote to which the push is being done (Ex: origin)

$2 -- URL to which the push is being done (Ex: https://://.git)

Information about the commits which are being pushed is supplied as lines to the standard input in the form:

**<local\_ref> <local\_sha1> <remote\_ref> <remote\_sha1>**

Sample values:

local\_ref = refs**/**heads**/**master

local\_sha1 = б8aθ7ee4fбaf8271dc4θcaaeбcc23f283122ed11 remote\_ref = refs**/**heads**/**master

remote\_sha1 = efd4d512f34b11e3cf5c12433bbedd4b153271бf

Below example pre-push script was taken from default pre-push.sample which was automatically created when a new repository is initialized with **git init**

***# This sampﾕe shows how ło prevenł push of commiłs where łhe ﾕog message słarłs # wiłh "WIP" (work in progress).***

remote="$1" url="$2"

z4θ=θθθθθθθθθθθθθθθθθθθθθθθθθθθθθθθθθθθθθθθθ

**while read** local\_ref local\_sha remote\_ref remote\_sha

**do**

**if [** "$local\_sha" = $z4θ **] then**

***# Handﾕe deﾕełe***

:

**else**

**if [** "$remote\_sha" = $z4θ **] then**

***# New branch, examine aﾕﾕ commiłs***

range="$local\_sha"

**else**

***# Updałe ło exisłing branch, examine new commiłs***

range="$remote\_sha..$local\_sha"

**fi**

***# Check for WIP commił***

commit=**`git rev–list** -n 1 --grep '^WIP' "$range"**` if [** -n "$commit" **]**

**then**

**echo >&**2 "Found WIP commit in $local\_ref, not pushing"

**exit** 1

**fi**

**fi done**

**exit** θ

## Section 56.2: Installing a Hook

The hooks are all stored in the hooks sub directory of the Git directory. In most projects, that’s .git**/**hooks.

To enable a hook script, put a file in the hooks subdirectory of your .git directory that is named appropriately (without any extension) and is executable.

# Chapter 57: Git rerere

rerere (reuse recorded resolution) allows you to tell git to remember how you resolved a hunk conflict. This allows it to be automatically resolved the next time that git encounters the same conflict.

## Section 57.1: Enabling rerere

To enable rerere run the following command:

$ **git config** --global rerere.enabled **true**

This can be done in a specific repository as well as globally.

# Chapter 58: Change git repositor6 name

If you change repository name on the remote side, such as your github or bitbucket, when you push your exisiting code, you will see error: Fatal error, repository not found\*\*.

## Section 58.1: Change local setting

Go to terminal,

**cd** projectFolder

**git remote** -v **(**it will show previous **git** url**)**

**git remote** set-url origin [https:**//**username**@**bitbucket.org**/**username**/**newName.git](https://username@bitbucket.org/username/newName.git)

**git remote** -v **(**double check, it will show new **git** url**) git push (do** whatever you want.**)**

# Chapter 59: Git Tagging

Like most Version Control Systems (VCSs), Git has the ability to tag specific points in history as being important. Typically people use this functionality to mark release points (v1.θ, and so on).

## Section 59.1: Listing all available tags

Using the command **git tag** lists out all available tags:

$ **git tag**

**<**output follows**>** vθ.1

v1.3

**Note**: the tags are output in an **alphabetical** order.

One may also search for available tags:

$ **git tag** -l "v1.8.5\*"

**<**output follows**>** v1.8.5

v1.8.5-rcθ v1.8.5-rc1 v1.8.5-rc2 v1.8.5-rc3 v1.8.5.1

v1.8.5.2

v1.8.5.3

v1.8.5.4

v1.8.5.5

## Section 59.2: Create and push tag(s) in GIT

### Create a tag:

To create a tag on your current branch:

**git tag <** tagname **>**

This will create a local tag with the current state of the branch you are on. To create a tag with some commit:

**git tag** tag-name commit-identifier

This will create a local tag with the commit-identifier of the branch you are on.

### Push a commit in GIT:

Push an individual tag:

**git push** origin tag-name

Push all the tags at once

**git push** origin --tags

# Chapter 60: Tid6ing up 6our local and remote repositor6

## Section 60.1: Delete local branches that have been deleted on the remote

To remote tracking between local and deleted remote branches use

**git fetch** -p

you can then use

**git branch** -vv

to see which branches are no longer being tracked.

Branches that are no longer being tracked will be in the form below, containing 'gone'

branch

12345eб **[**origin**/**branch: gone**]** Fixed bug

you can then use a combination of the above commands, looking for where 'git branch -vv' returns 'gone' then using '-d' to delete the branches

**git fetch** -p **&& git branch** -vv **| awk** '/: gone]/{print $1}' **| xargs git branch** -d

# Chapter 61: diff-tree

Compares the content and mode of blobs found via two tree objects.

## Section 61.1: See the files changed in a specific commit

git diff-tree --no-commit-id --name-only -r COMMIT\_ID

## Section 61.2: Usage

**git diff–tree [**--stdin**] [**-m**] [**-c**] [**--cc**] [**-s**] [**-v**] [**--pretty**] [**-t**] [**-r**] [**--root**] [<**common-diff- options**>] <**tree-ish**> [<**tree-ish**>] [<**path**>**...**]**

### Option Explanation

-r diff recursively

--root include the initial commit as diff against /dev/null

## Section 61.\: Common diff options

### Option Explanation

-z output diff-raw with lines terminated with NUL.

-p output patch format.

-u synonym for -p.

--patch-with-raw output both a patch and the diff-raw format.

--stat show diffstat instead of patch.

--numstat show numeric diffstat instead of patch.

--patch-with-stat output a patch and prepend its diffstat.

--name-only show only names of changed files.

--name-status show names and status of changed files.

--full-index show full object name on index lines.

--abbrev=<n> abbreviate object names in diff-tree header and diff-raw.

-R swap input file pairs.

-B detect complete rewrites.

-M detect renames.

-C detect copies.

--find-copies-harder try unchanged files as candidate for copy detection.

-l<n> limit rename attempts up to paths.

-O reorder diffs according to the .

-S find filepair whose only one side contains the string.

--pickaxe-all show all files diff when -S is used and hit is found.

-a --text treat all files as text.

# Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to [web@petercv.com](mailto:web@petercv.com) for new content to be published or updated

Aaron Critchley Chapter 6

Aaron Skomra Chapter 49

aavrug Chapter 26

Abdullah Chapter 2

Abhijeet Kasurde Chapter 6

adarsh Chapter 16

Adi Lester Chapter 6

AER Chapters 12, 25 and 29

AesSedai101 Chapters 4, 11 and 53

Ahmed Metwally Chapter 2

Ajedi32 Chapter 11

Ala Eddine JEBALI Chapter 1

Alan Chapter 10

Alex Stuckey Chapter 46

Alexander Bird Chapter 12

Allan Burleson Chapter 1

Alu Chapter 50

ambes Chapter 45

Amitay Stern Chapters 1 and 10

anderas Chapters 6 and 12

AndiDog Chapter 16

andipla Chapter 44

Andrea Romagnoli Chapter 25 Andrew Sklyarevsky Chapter 10

Andy Hayden Chapters 1, 4, 7, 10 and 25

AnimiVulpis Chapters 1, 5 and 14

AnoE Chapter 24

Anthony Staunton Chapter 11

APerson Chapters 10 and 13

apidae Chapter 6

Aratz Chapter 2

Asaph Chapters 4 and 26

Asenar Chapters 11 and 13

Ates Goral Chapter 32

Atul Khanduri Chapters 17 and 59

Bad Chapter 14

bandi Chapters 10 and 16

Ben Chapter 5

Blundering Philosopher Chapter 25 BobTuckerman Chapter 14

Boggin Chapters 1, 7, 31 and 36

Božo Stojković Chapter 5

bpoiss Chapter 5

Braiam Chapter 5

brentonstrine Chapters 7 and 8

Brett Chapter 2

Brian Chapters 1 and 7

Brian Hinchey Chapter 26

bstpierre Chapter 11

bud Chapters 17, 26 and 28

Cache Staheli Chapters 5, 10 and 13

Caleb Brinkman Chapters 3 and 16

Charlie Egan Chapter 6

Chin Huang Chapter 20

Christiaan Maks Chapter 24

Cody Guldner Chapters 10 and 29

Collin M Chapter 5

ComicSansMS Chapter 9

Confiqure Chapters 4, 22, 24, 36 and 44

cormacrelf Chapter 10

Craig Brett Chapter 1

Creative John Chapter 18

cringe Chapter 29

Dániel Kis Chapter 45

dahlbyk Chapter 20

dan Chapter 14

Dan Hulme Chapter 1

Daniel Käfer Chapters 12, 14 and 50

Daniel Stradowski Chapter 14

Dartmouth Chapters 5, 25, 34, 43, 45, 47 and 48

David Ben Knoble Chapter 38

davidcondrey Chapters 2 and 10

Deep Chapters 10 and 26

Deepak Bansal Chapter 14

Devesh Saini Chapter 5

Dheeraj vats Chapter 5

Dimitrios Mistriotis Chapter 22

Dong Thang Chapter 49

dubek Chapter 17

Duncan X Simpson Chapter 14

e.doroskevic Chapters 12 and 13

Ed Cottrell Chapter 5

Eidolon Chapter 24

Elizabeth Chapter 45

enrico.bacis Chapter 5

ericdwang Chapter 10

eush77 Chapters 6, 11 and 16

eykanal Chapter 1

Ezra Free Chapter 25

Fabio Chapter 2

Farhad Faghihi Chapter 48

FeedTheWeb Chapter 48

Flows Chapters 2 and 24

forevergenin Chapters 3, 14 and 34

forresthopkinsa Chapter 22

fracz Chapters 5, 14 and 26

Fred Barclay Chapters 1, 10 and 14

frlan Chapter 29

Functino Chapter 5

fybw id Chapters 49 and 61

ganesshkumar Chapter 25

gavv Chapters 14 and 35

George Brighton Chapter 10

georgebrock Chapter 16

GingerPlusPlus Chapter 26

Glenn Smith Chapter 35

gnis Chapter 19

Greg Bray Chapter 50

Guillaume Chapter 29

Guillaume Pascal Chapters 5 and 36

guleria Chapter 2

hardmooth Chapter 22

heitortsergent Chapter 3

Henrique Barcelos Chapter 1

Horen Chapter 22

Hugo Buff Chapter 48

Hugo Ferreira Chapter 12

Igor Ivancha Chapter 10

Indregaard Chapter 36

intboolstring Chapters 2, 4, 5, 6, 9, 10, 12, 17 and 29

Isak Combrinck Chapter 57

J F Chapter 9

Jack Ryan Chapter 6

JakeD Chapter 6

Jakub Narębski Chapters 4, 5, 6, 26 and 43

james large Chapter 14

James Taylor Chapter 10

janos Chapters 1, 2 and 10

Jarede Chapter 26

Jason Chapter 14

Jav\_Rock Chapters 1, 45 and 49

Jeff Puckett Chapter 27

jeffdill2 Chapters 1, 3, 6 and 26

Jens Chapter 5

jkdev Chapter 4

joaquinlpereyra Chapter 5

Joel Cornett Chapter 14

joeytwiddle Chapter 10

JonasCz Chapter 5

Jonathan Chapter 14

Jonathan Lam Chapter 1

Jordan Knott Chapter 10 Joseph Dasenbrock Chapters 1 and 14 Joseph K. Strauss Chapters 6 and 12

joshng Chapter 5

jpkrohling Chapter 16

jready Chapter 2

jtbandes Chapters 11 and 12

Julian Chapters 13 and 52

Julie David Chapters 3, 18 and 26

jwd630 Chapter 39

Kačer Chapter 5

Kageetai Chapter 1

Kalpit Chapters 3 and 45

Kamiccolo Chapter 2

Kapep Chapter 5

Kara Chapter 26

Karan Desai Chapter 28

kartik Chapters 14 and 25

KartikKannapur Chapters 1, 10, 25 and 48

Kay V Chapters 1 and 4

Kelum Senanayake Chapter 56

Keyur Ramoliya Chapter 54

khanmizan Chapters 6 and 14

kirrmann Chapter 14

kisanme Chapters 14 and 17

Kissaki Chapters 23 and 31

knut Chapter 5

kofemann Chapter 49

Koraktor Chapter 26

kowsky Chapter 9

KraigH Chapter 2

LeftRight92 Chapter 5

LeGEC Chapters 2 and 12

Liam Ferris Chapter 8

Libin Varghese Chapter 12

Liju Thomas Chapter 47

Liyan Chang Chapter 13

Lochlan Chapter 17

lostphilosopher Chapter 24

Luca Putzu Chapter 12

lucash Chapter 12

Mário Meyrelles Chapter 29

maccard Chapter 1

Mackattack Chapter 5

madhead Chapter 11

Majid Chapters 6, 10, 12, 14, 22 and 26

manasouza Chapters 2 and 26

Manishh Chapter 55

Mario Chapter 21

Martin Chapter 14

Martin Pecka Chapter 5

Marvin Chapters 5 and 29

Matas Vaitkevicius Chapter 52

Mateusz Piotrowski Chapter 16

Matt Clark Chapters 2 and 10

Matt S Chapter 29

Matthew Hallatt Chapters 2, 7, 10, 42 and 46

MayeulC Chapters 5, 10, 14, 19, 23 and 29

MByD Chapter 3

Micah Smith Chapter 10 Micha Wiedenmann Chapter 53 Michael Mrozek Chapter 12

Michael Plotke Chapter 21

Mitch Talmadge Chapters 5 and 14

mkasberg Chapter 15

mpromonet Chapters 2, 9, 17 and 23

MrTux Chapter 41

mwarsco Chapter 24

mystarrocks Chapter 3

n0shadow Chapter 19

Narayan Acharya Chapter 5

Nathan Arthur Chapter 4

Nathaniel Ford Chapters 6 and 7

Nemanja Boric Chapter 12

Nemanja Trifunovic Chapter 50

nepda Chapter 14

Neui Chapters 1 and 5

nighthawk454 Chapters 30 and 42

Nithin K Anil Chapter 7

Noah Chapters 2, 8 and 14

Noushad PP Chapter 14

Nuri Tasdemir Chapter 5

nus Chapters 11 and 38

ob1 Chapter 1

Ogre Psalm33 Chapter 6

Oleander Chapter 2

olegtaranenko Chapter 14

orkoden Chapters 6 and 40

Ortomala Lokni Chapters 6, 12, 13 and 26

Ozair Kafray Chapter 14

P.J.Meisch Chapter 28

Pace Chapter 7

PaladiN Chapters 5, 9 and 14

Patrick Chapter 26

pcm Chapter 29

Pedro Pinheiro Chapters 2 and 50

penguincoder Chapters 6, 8 and 11

Peter Amidon Chapter 51

Peter Mitrano Chapters 12, 25 and 26

PhotometricStereo Chapter 28

pkowalczyk Chapter 25

pktangyue Chapter 5

Pod Chapters 1 and 10

pogosama Chapter 29

poke Chapter 5

Priyanshu Shekhar Chapters 13, 14, 19 and 42

pylang Chapters 5 and 12

Raghav Chapter 3

Ralf Rafael Frix Chapters 3, 14, 19 and 26

RedGreenCode Chapter 17

RhysO Chapter 5

Ricardo Amores Chapter 33

Richard Chapter 12

Richard Dally Chapter 4

Richard Hamilton Chapter 14

Rick Chapters 5, 7, 10, 25 and 36

riyadhalnur Chapter 11

Roald Nefs Chapter 1

Robin Chapter 14

rokonoid Chapter 5

ronnyfm Chapter 1

Salah Eddine Lahniche Chapter 3 saml Chapter 3

Sardathrion Chapter 22

Sascha Chapter 5

Sascha Wolf Chapter 5

SashaZd Chapter 48 Sazzad Hissain Khan Chapter 1

Scott Weldon Chapters 3, 5, 22, 23, 41 and 51

Sebastianb Chapters 5 and 26

SeeuD1 Chapter 5

shoelzer Chapter 46

Shog9 Chapter 23

Simone Carletti Chapters 14 and 41

sjas Chapter 5

SommerEngineering Chapter 10

sonali Chapter 45

Sonny Kim Chapter 10

spikeheap Chapter 5

Stony Chapter 23

strangeqargo Chapter 18

SurDin Chapter 6

Tall Sam Chapter 16

textshell Chapter 7

Thamilan Chapter 23

thanksd Chapter 11

the12 Chapter 14

TheDarkKnight Chapters 36 and 59

theJollySin Chapter 5

Thomas Crowley Chapter 60

tinlyx Chapter 9

Toby Chapters 5 and 20

Toby Allen Chapter 2

Tom Gijselinck Chapter 5

Tom Hale Chapter 11

Tomás Cañibano Chapters 26 and 29

Tomasz Bąk Chapter 5

Travis Chapter 12

Tyler Zika Chapter 1

tymspy Chapter 1

Undo Chapters 8, 9, 10 and 25

Uwe Chapter 14

Vi. Chapter 5

Victor Schröder Chapters 5, 12 and 44

Vivin George Chapter 3

Vlad Chapter 14

Vladimir F Chapter 10

Vogel612 Chapter 8

VonC Chapters 1, 3, 5, 9, 12 and 13

Wasabi Fan Chapter 12

Wilfred Hughes Chapter 5

Will Chapter 6

Wojciech Kazior Chapters 11, 25 and 26

Wolfgang Chapters 4, 5, 8, 13 and 14

WPrecht Chapter 42

xiaoyaoworm Chapter 58

ydaetskcoR Chapter 5

Yerko Palma Chapter 14

Yury Fedorov Chapters 5 and 14

Zaz Chapters 6, 7, 10, 18, 23 and 37

zebediah49 Chapter 41

zygimantus Chapter 14

□□ɴ□ʏɪ□□ɪs Chapter 18

□□□□□ Chapters 6 and 12

# You ma6 also like

