# Git Pro

## Chapter 1: Getting Started

### 1.1 About Version Control

Version control is a system that records changes to a file or a set of files over time so that you can recall specific versions later. It allows you to revert selected files back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. If you lose files, you can easily recover.

#### Local Version Control System

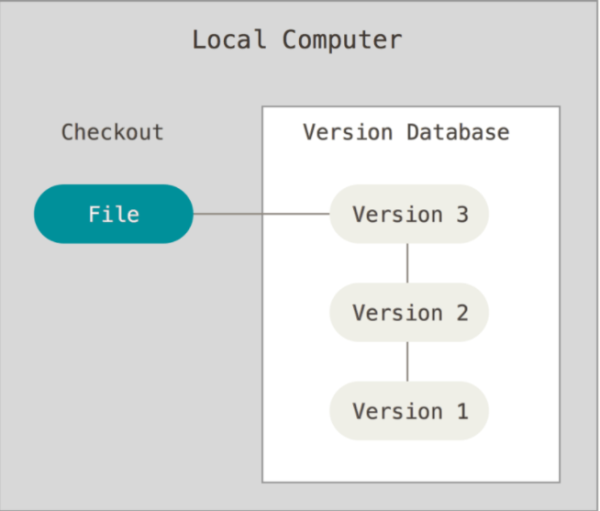


Figure : Local Version Control

Local version control system is a simple database that kept all the changes to files under version control. It keeps the differences between file in a special format on disk. It can then re-create what any file looked like at any point in time by adding up all the differences.

For example: RCS

Disadvantages: A developer can’t collaborate with developers on other systems.

#### Centralized Version Control Systems

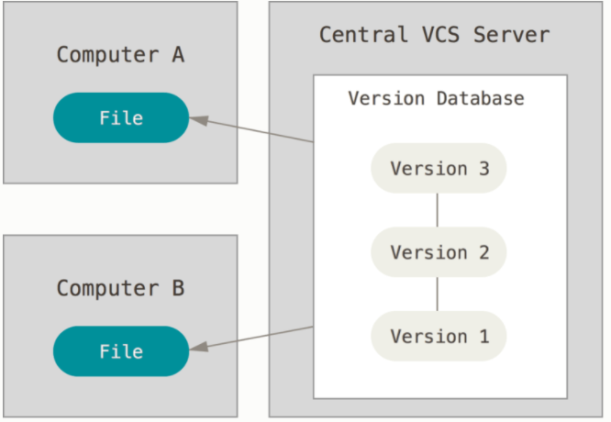


Figure : Centralized Version Control System

Centralized version control systems have a single server that contains all the versioned files, and a number of clients that check out files from that central place.

Example: CVS, Subversion, and Perforce

Disadvantages: If server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they’re working on.

If centralized database is on becomes corrupted, and proper backups haven’t been kept, you lose absolutely everything.

#### Distributed Version Control Systems

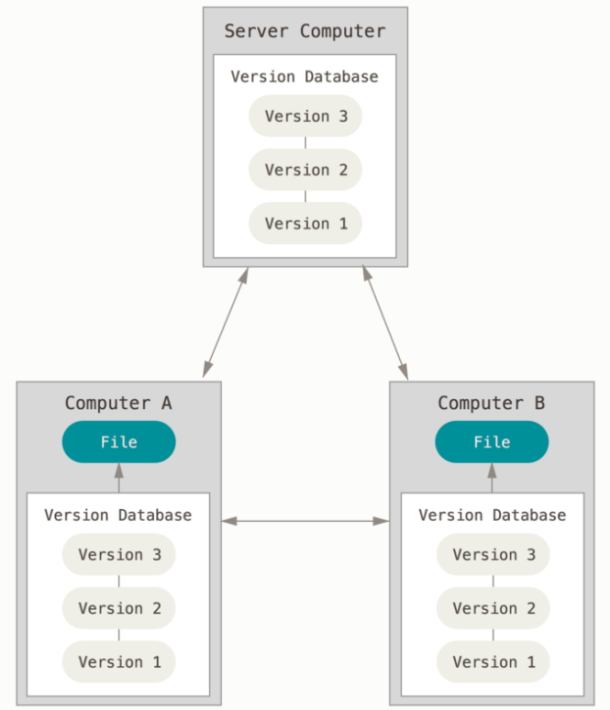


Figure : Distributed Version Control Systems

Clients don’t just check out the latest snapshot of the files; rather, they check out full repository and history from the server. If the server dies, any the client repositories can be copied back up to the server to restore it.

For example: Git, Mercurial, Bazaar or Darcs

### 1.2 A Short History of Git

In 2002, the Linux kernel project using a commercial DVCS called BitKeeper.

In 2005, the relationship was broken. The Linux community develops their own tool Git. Some goals of the new system:

* Speed
* Simple design
* Strong support for non-linear development
* Fully distributed
* Able to handle large projects

### 1.3 What is Git?

#### Snapshots, not differences

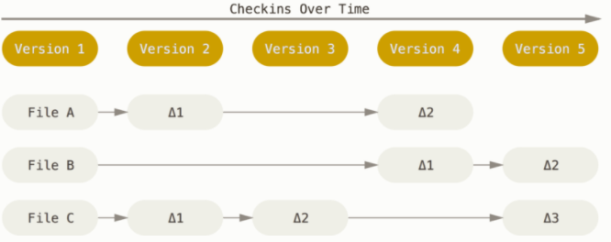


Figure : Storing data as changes to a base version of each file

Most of VCSs store information as a list of changes on each file.

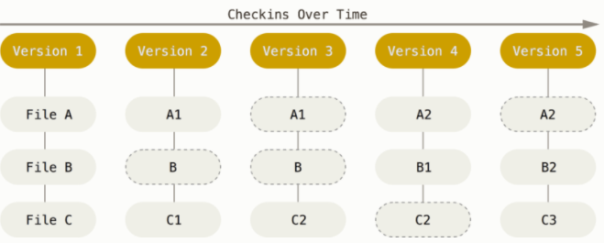


Figure : Storing data as snapshots of the project over time

Git stores information as a series of snapshots whole system. In Git, every time you commit, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. If file doesn’t change anything, Git link to the previous reference.

#### Nearly every operation is local

Because you have the entire history of the project right on your local disk, most of operations seem almost instantaneous. If you can’t connect to network, you can commit to your local copy and upload your work when you can connect to network.

#### Git has integrity

Git use SHA-1 hash that is calculated based on the contents of a file or directory structure in Git. Git stores everything in its database not by file name but by the hash value of its contents.

For example: 24b9da6552252987aa493b52f8696cd6d3b00373

#### The three states

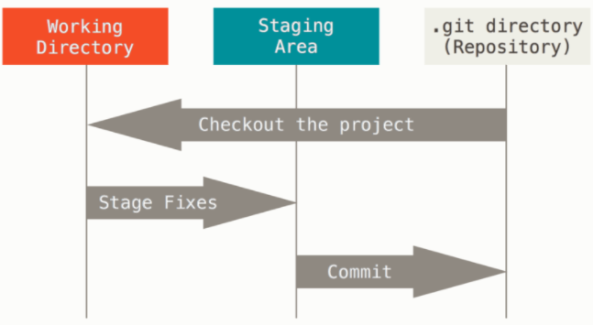


Figure : Working tree, staging area, and Git directory

Git has three main states: modified, staged, and committed

* Modified means that you have changed the file but have not committed it to your database yet.
* Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
* Committed mean that the data is safety stored in your local database.

### 1.4 The Command Line

For example: Terminal in macOS, command prompt or powershell in Windows.

### 1.5 Installing Git

#### Installing on Linux

$ sudo apt install git-all

#### Installing on macOS

$ git --version

#### Installing on Windows

Just go to <https://git-scm.com/download/win> and the download will start automatically.

#### Installing from Source

### 1.6 First-time Git Setup

You can view all of your settings and where they are coming from using:

$ git config --list --show-origin

#### Your identity

You can set up your user name and email address:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

#### Your editor

You can configure the default text editor:

git config --global core.editor emacs

In the case of Notepad++:

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

#### Your default branch name

To set “main” as the default branch name do:

$ git config --global init.defaultBranch main

#### Checking your settings

To check you configuration settings, you can use command:

$ git config --list

user.name=John Doe

user.email=johndoe@example.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

### 1.7 Getting Help

## Chapter 2: Git Basics

### 2.1 Getting a Git Repository

#### Initializing a Repository in an Existing Directory

Linux: $ cd /home/user/my\_project

macOS: $ cd /Users/user/my\_project

Windows: $ cd C:/Users/user/my\_project

$ git init

#### Cloning an Existing Repository

Same name with origin “libgit2”:

$ git clone https://github.com/libgit2/libgit2

Change name “libgit2” into “mylibgit”:

$ git clone https://github.com/libgit2/libgit2 mylibgit

### 2.2 Recording Changes to the Repository

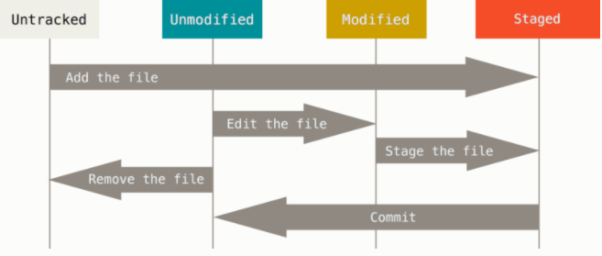


Figure : The lifecycle of the status of your files

#### Checking the Status of Your Files

Nothing changes or new:

git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean

File README didn’t exist before:

$ echo 'My Project' > README

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

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

README

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

#### Tracking New Files

To be tracking the README file:

$ git add README

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: README

#### Staging Modified Files

Let’s change a file CONTRIBUTING that was already tracked:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

To staged it or add this file to the next commit, you run git add command:

$ git add CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

Open CONTRIBUTING.md again and make change:

$ vim CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

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: CONTRIBUTING.md

The CONTRIBUTING.md now is listed as both staged and unstaged. If you commit now, the version of CONTRIBUTING.md as it was when you last ran the git add command is how it go into the commit, not the version of the file as it looks in your working directory when you run git commit. If you modify a file after you run git add, you have to run git add again to stage the lastest version of the file:

$ git add CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

#### Short Status

$ git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

?? - untracked file, M - modified or unstaged file, A - staged file, MM - unstaged and staged file

#### Ignoring Files

You can create a file listing patterns to match them named .gitignore:

$ cat .gitignore

\*.[oa]

\*~

The rules for the patterns you can put in the .gitignore file are as follows:

* Blank lines or lines starting with # are ignored.
* Standard glob patterns work, and will be applied recursively throughout the entire working tree.
* You can start patterns with a forward slash (/) to avoid recursivity.
* You can end patterns with a forward slash (/) to specify a directory.
* You can negate a pattern by starting it with an exclamation point (!).

Glob patterns are like simplified regular expressions that shells use:

* (\*) matches zero or more characters
* [abc] matches any character inside the brackets
* (?) matches a single character
* [0-9] matches any character between them (in this case 0 through 9)
* (\*\*) matches nested directories (a/\*\*/z would match a/z, a/b/z, so on)

Here is another example .gitignore file:

# ignore all .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# ignore all files in any directory named build

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories

doc/\*\*/\*.pdf

#### Viewing Your Staged and Unstaged Changes

Let’s say you edit and stage the README file again and the edit the CONTRIBUTING.md file without staging it. If you run git status command, you once again see:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README

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: CONTRIBUTING.md

The git diff command without argument will show you the changes in the files that are still unstaged.

$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

The git diff --staged or git diff --cached commands show you the changes in the files that are staged:

$ git diff --staged

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README

@@ -0,0 +1 @@

+My Project

$ git diff --cached

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

#### Committing Your Changes

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

[master 463dc4f] Story 182: fix benchmarks for speed

2 files changed, 2 insertions(+)

create mode 100644 README

#### Skipping the Staging Area

$ git status

On 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: CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")

$ git commit -a -m 'Add new benchmarks'

[master 83e38c7] Add new benchmarks

1 file changed, 5 insertions(+), 0 deletions(-)

#### Removing Files

Remove a file from working directory:

$ rm PROJECTS.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")

Then, you run git rm, it stages the file’s removal:

$ git rm PROJECTS.md

rm 'PROJECTS.md'

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

deleted: PROJECTS.md

Keep the file in working directory, but remove it from staging area:

git rm --cached README

#### Moving Files

$ git mv README.md README

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

### 2.3 Viewing the Commit History

Table : Common options to git log

| **Option** | **Description** |
| --- | --- |
| -p | Show the patch introduced with each commit. |
| --stat | Show statistics for files modified in each commit. |
| --shortstat | Display only the changed/insertions/deletions line from the --stat command. |
| --name-only | Show the list of files modified after the commit information. |
| --name-status | Show the list of files affected with added/modified/deleted information as well. |
| --abbrev-commit | Show only the first few characters of the SHA-1 checksum instead of all 40. |
| --relative-date | Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format. |
| --graph | Display an ASCII graph of the branch and merge history beside the log output. |
| --pretty | Show commits in an alternate format. Option values include oneline, short, full, fuller, and format (where you specify your own format). |
| --oneline | Shorthand for --pretty=oneline --abbrev-commit used together. |

For example:

$ git log -p -2

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

diff --git a/Rakefile b/Rakefile

index a874b73..8f94139 100644

--- a/Rakefile

+++ b/Rakefile

@@ -5,7 +5,7 @@ require 'rake/gempackagetask'

spec = Gem::Specification.new do |s|

s.platform = Gem::Platform::RUBY

s.name = "simplegit"

- s.version = "0.1.0"

+ s.version = "0.1.1"

s.author = "Scott Chacon"

s.email = "schacon@gee-mail.com"

s.summary = "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

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

Remove unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index a0a60ae..47c6340 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -18,8 +18,3 @@ class SimpleGit

end

end

-

-if $0 == \_\_FILE\_\_

- git = SimpleGit.new

- puts git.show

-end

$ git log --stat

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

Rakefile | 2 +-

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

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

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

Remove unnecessary test

lib/simplegit.rb | 5 -----

1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

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

Initial commit

README | 6 ++++++

Rakefile | 23 +++++++++++++++++++++++

lib/simplegit.rb | 25 +++++++++++++++++++++++++

3 files changed, 54 insertions(+)

Table : Useful specifiers for git log --pretty=format

| **Specifier** | **Description of Output** |
| --- | --- |
| %H | Commit hash |
| %h | Abbreviated commit hash |
| %T | Tree hash |
| %t | Abbreviated tree hash |
| %P | Parent hashes |
| %p | Abbreviated parent hashes |
| %an | Author name |
| %ae | Author email |
| %ad | Author date (format respects the --date=option) |
| %ar | Author date, relative |
| %cn | Committer name |
| %ce | Committer email |
| %cd | Committer date |
| %cr | Committer date, relative |
| %s | Subject |

For example:

$ git log --pretty=oneline

ca82a6dff817ec66f44342007202690a93763949 Change version number

085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Remove unnecessary test

a11bef06a3f659402fe7563abf99ad00de2209e6 Initial commit

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

ca82a6d - Scott Chacon, 6 years ago : Change version number

085bb3b - Scott Chacon, 6 years ago : Remove unnecessary test

a11bef0 - Scott Chacon, 6 years ago : Initial commit

$ git log --pretty=format:"%h %s" --graph

\* 2d3acf9 Ignore errors from SIGCHLD on trap

\* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit

|\

| \* 420eac9 Add method for getting the current branch

\* | 30e367c Timeout code and tests

\* | 5a09431 Add timeout protection to grit

\* | e1193f8 Support for heads with slashes in them

|/

\* d6016bc Require time for xmlschema

\* 11d191e Merge branch 'defunkt' into local

#### Limiting log output

Table : Options to limit the output of git log

| **Option** | **Description** |
| --- | --- |
| -<n> | Show only the last n commits |
| --since, --after | Limit the commits to those made after the specified date. |
| --until, --before | Limit the commits to those made before the specified date. |
| --author | Only show commits in which the author entry matches the specified string. |
| --committer | Only show commits in which the committer entry matches the specified string. |
| --grep | Only show commits with a commit message containing the string |
| -S | Only show commits adding or removing code matching the string |

For example:

$ git log --pretty="%h - %s" --author='Junio C Hamano' --since="2008-10-01" \

--before="2008-11-01" --no-merges -- t/

5610e3b - Fix testcase failure when extended attributes are in use

acd3b9e - Enhance hold\_lock\_file\_for\_{update,append}() API

f563754 - demonstrate breakage of detached checkout with symbolic link HEAD

d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths

51a94af - Fix "checkout --track -b newbranch" on detached HEAD

b0ad11e - pull: allow "git pull origin $something:$current\_branch" into an unborn branch

### 2.4 Undoing Things

#### Replacing old commit

$ git commit -m 'Initial commit'

$ git add forgotten\_file

$ git commit --amend

#### Unstaging a Staged File

$ git add \*

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

modified: CONTRIBUTING.md

Unstaged a staged file with git reset HEAD:

$ git reset HEAD CONTRIBUTING.md

Unstaged changes after reset:

M CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

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: CONTRIBUTING.md

Unstaged a staged file with git restore --staged :

$ git restore --staged CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

renamed: README.md -> README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

#### Unmodiyfing a Modified File

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: CONTRIBUTING.md

Unmodifying a modified file with git checkout --:

$ git checkout -- CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

Unmodifying a modified file with git restore:

$ git restore CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

renamed: README.md -> README

### 2.5 Working with Remotes

#### Showing Your Remotes

Showing the short name:

$ git clone https://github.com/schacon/ticgit

Cloning into 'ticgit'...

remote: Reusing existing pack: 1857, done.

remote: Total 1857 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.

Resolving deltas: 100% (772/772), done.

Checking connectivity... done.

$ cd ticgit

$ git remote

origin

Showing the URLs:

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

#### Adding Remote Repositories

Add a new remote Git repository with a shortname:

$ git remote

origin

$ git remote add pb https://github.com/paulboone/ticgit

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

pb https://github.com/paulboone/ticgit (fetch)

pb https://github.com/paulboone/ticgit (push)

#### Fetching and Pulling from Your Remotes

To get data from your remote repositories but not merge:

$ git fetch <remote>

To get data from your remote repositories and try to merge:

$ git pull <remote>

#### Pushing to Remotes

$ git push origin master

#### Inspecting a Remote

$ git remote show origin

\* remote origin

Fetch URL: https://github.com/schacon/ticgit

Push URL: https://github.com/schacon/ticgit

HEAD branch: master

Remote branches:

master tracked

dev-branch tracked

Local branch configured for 'git pull':

master merges with remote master

Local ref configured for 'git push':

master pushes to master (up to date)

#### Renaming and Removing Remotes

$ git remote rename pb paul

$ git remote

origin

paul

$ git remote remove paul

$ git remote

origin

### 2.6 Tagging

#### Listing Your Tags

$ git tag

v1.0

v2.0

$ git tag -l "v1.8.5\*"

v1.8.5

v1.8.5-rc0

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

#### Creating Tags

Git supports two types of tags: **Lightweight and annotated**.

A lightweight tag is very much like a branch that doesn’t change - it’s just a pointer to a specific commit.

Annotated tags are stored as full objects in the Git database. They’re checksummed; contain the tagger name, email, date; have a tagging message; and can be signed and verified with GNU. It’s generally recommended that you create annotated tags so you can have all this information.

#### Annotated Tags

Creating an annotated tag in Git:

$ git tag -a v1.4 -m "my version 1.4"

$ git tag

v0.1

v1.3

v1.4

You can see the tag data along with the commit that was tagged by using git show command:

$ git show v1.4

tag v1.4

Tagger: Ben Straub <ben@straub.cc>

Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

#### Lightweight Tags

git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5

This time, if you run git show on the tag, you don’t see the extra tag information. The command just shows the commit:

$ git show v1.4-lw

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

#### Tagging Later

Suppose your commit history looks like this:

$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 Create write support

0d52aaab4479697da7686c15f77a3d64d9165190 One more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc Add commit function

4682c3261057305bdd616e23b64b0857d832627b Add todo file

166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support

9fceb02d0ae598e95dc970b74767f19372d61af8 Update rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc Commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a Update readme

Now, suppose you forgot to tag the project at v1.2, which was at the “Update rakefile” commit. You can add it after the fact. To tag that commit, you specify the commit checksum (or apart of it) at the end of the command:

$ git tag -a v1.2 9fceb02

You can see that you’ve tagged the commit:

$ git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5

$ git show v1.2

tag v1.2

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date: Sun Apr 27 20:43:35 2008 -0700

Update rakefile

...

#### Sharing Tags

By default, the git push command doesn’t transfer tags to remote server. To share tag with remote server you can run git push origin <tag\_name>:

$ git push origin v1.5

Counting objects: 14, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (12/12), done.

Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.

Total 14 (delta 3), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.5 -> v1.5

If you have a lot of tags that you want to push up at once, you can also use --tags option to the git push command:

$ git push origin --tags

Counting objects: 1, done.

Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.

Total 1 (delta 0), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.4 -> v1.4

\* [new tag] v1.4-lw -> v1.4-lw

#### Deleting Tags

To delete a tag on your local repository, you can run git tag -d <tag\_name>:

$ git tag -d v1.4-lw

Deleted tag 'v1.4-lw' (was e7d5add)

Note that this doesn’t remove the tag from any remote server. To delete the tag from remote server, we have two common ways. The first way is git push <remote> :refs/tags/<tag\_name>

$ git push origin :refs/tags/v1.4-lw

To /git@github.com:schacon/simplegit.git

- [deleted] v1.4-lw

The second way to delete a remote tag:

$ git push origin --delete <tagname>

#### Checking out Tags

If you want to view the versions of files a tag is pointing to, you can do a git checkout of the tag:

$ git checkout v2.0.0

Note: switching to 'v2.0.0'.

You are in 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this

state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -c with the switch command. Example:

git switch -c <new-branch-name>

Or undo this operation with:

git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 99ada87... Merge pull request #89 from schacon/appendix-final

$ git checkout v2.0-beta-0.1

Previous HEAD position was 99ada87... Merge pull request #89 from schacon/appendix-final

HEAD is now at df3f601... Add atlas.json and cover image

In “detached HEAD” state, if you make changes and then commit it, the tag will stay the same. Thus, if you need to make changes - say you’re fixing a bug on an older version, for instance - you will generally want to create a branch:

$ git checkout -b version2 v2.0.0

Switched to a new branch 'version2'

Now, you make a commit, your version2 branch will be different than you v2.0.0 tag.

### 2.7 Git Aliases

You may setup alias for each command using git config:

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

### 2.8 Summary

## Chapter 3: Git Branching

### 3.1 Branches in a Nutshell

Git doesn’t store data as a series of changes sets or differences, but a series of snapshots. When you make a commit, git stores a commit object that contains a pointer to the snapshot of content that you staged. This object contains the author’s name and email address, the message that you typed, and pointers to the previous commit (it’s parent commit): zero parents for the initial commit, one parent for normal commit, and multiple parents for a commit that results from a merge of two or more branches.

$ git add README test.rb LICENSE

$ git commit -m 'Initial commit'

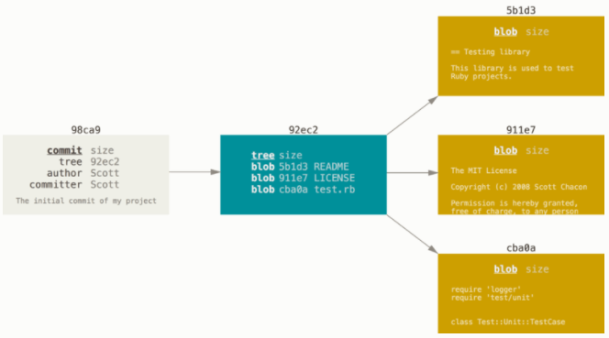


Figure : A commit and its tree

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it:

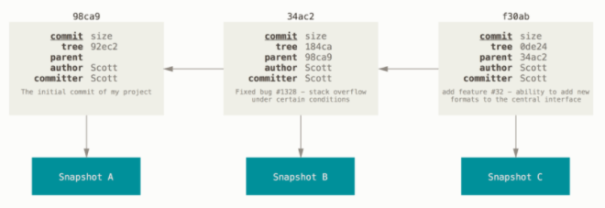


Figure : Commits and their parents

A branch in Git is simply a lightweight movable pointer to one of these commits.

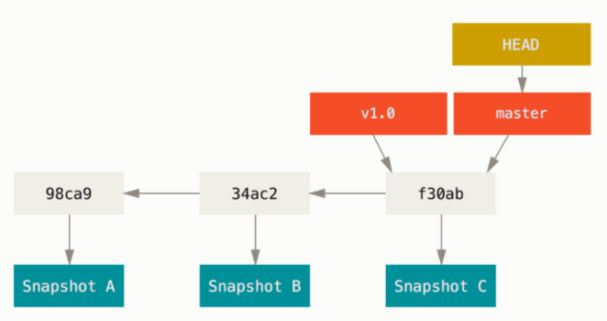


Figure : A branch and its commit history

#### Create a New Branch

$ git branch testing

This creates a new pointer to the same commit you’re currently on.



Figure : Two branches pointing into the same series of commits.

Git has a special pointer called HEAD to let you know what branch you’re currently on.

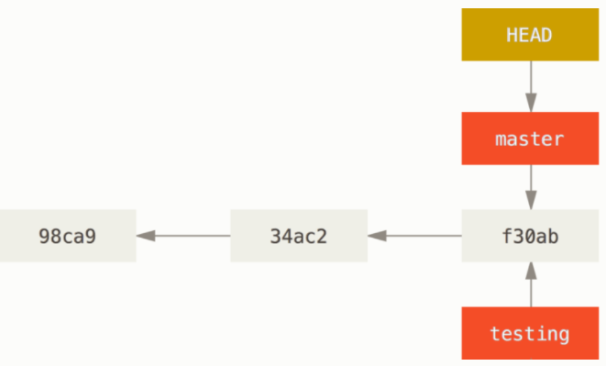


Figure : HEAD pointing to a branch.

You can see where the branch pointer is pointing by using git log with option called --decorate.

$ git log --oneline --decorate

f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to the central interface

34ac2 Fix bug #1328 - stack overflow under certain conditions

98ca9 Initial commit

#### Switching Branches

To switch to an existing branch, you run the git checkout command.

$ git checkout testing

This will move the HEAD pointer to point the testing branch.



Figure : HEAD points to the current branch.

HEAD pointer will move forward with the current branch when a commit is made.

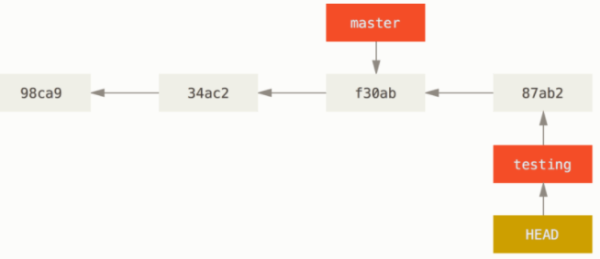


Figure : The HEAD branch moves forward when a commit is made.

Let’s switch back to the master branch.

$ git checkout master

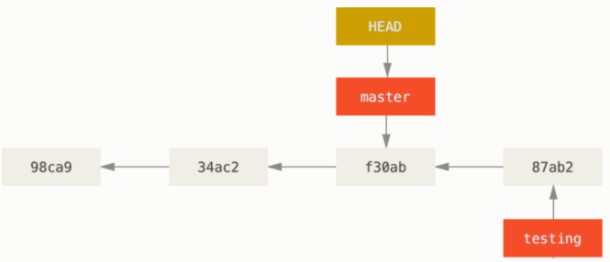


Figure : HEAD moves when you checkout.

Let’s make a change and commit again.

$ vim test.rb

$ git commit -a -m 'made other changes'

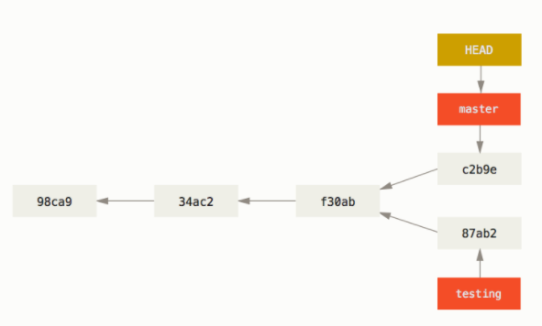


Figure : Divergent history.

To see the history of your commits, show where your branch pointers are and how your history look like, you run git log --oneline --decorate --graph --all.

$ git log --oneline --decorate --graph --all

\* c2b9e (HEAD, master) Made other changes

| \* 87ab2 (testing) Made a change

|/

\* f30ab Add feature #32 - ability to add new formats to the central interface

\* 34ac2 Fix bug #1328 - stack overflow under certain conditions

\* 98ca9 initial commit of my project

### 3.2 Basic Branching and Merging

#### Basic Branching

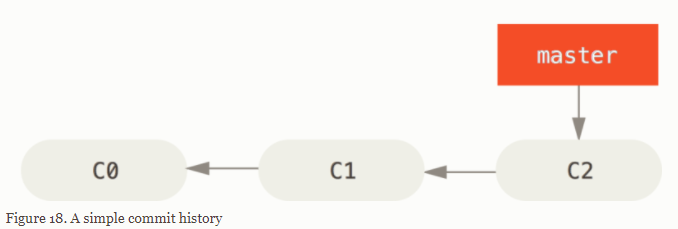


Figure : A simple commit history.

Create a new branch called iss53, and switch to iss53 branch.

$ git checkout -b iss53

Switched to a new branch "iss53"

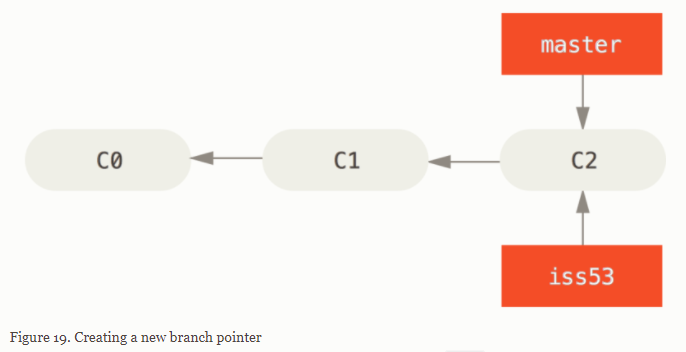


Figure : Creating a new branch pointer.

Doing some commit and iss53 moves forward since the HEAD pointer is pointing to it.

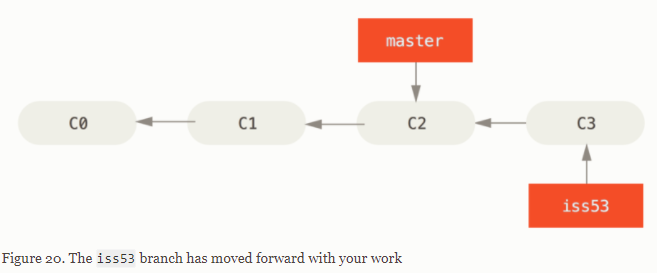


Figure : The iss53 has moved forward with your work.

You done on the iss53 and you switch back to the master branch.

$ git checkout master

Switched to branch 'master'

Let’s assume you have to concentrate on you hotfix problem. Then you create a hotfix branch and switch to hotfix.

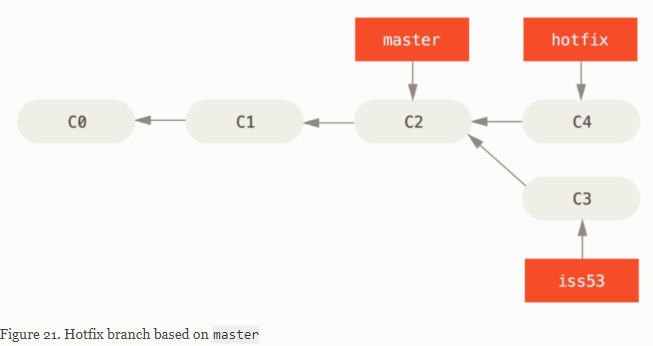


Figure : Hotfix branch based on master.

After you done on hotfix you want to merge the hotfix branch back into your master.

$ git checkout master

$ git merge hotfix

Updating f42c576..3a0874c

Fast-forward

index.html | 2 ++

1 file changed, 2 insertions(+)

The phase “fast-forward” means git simply moves the pointer forward.

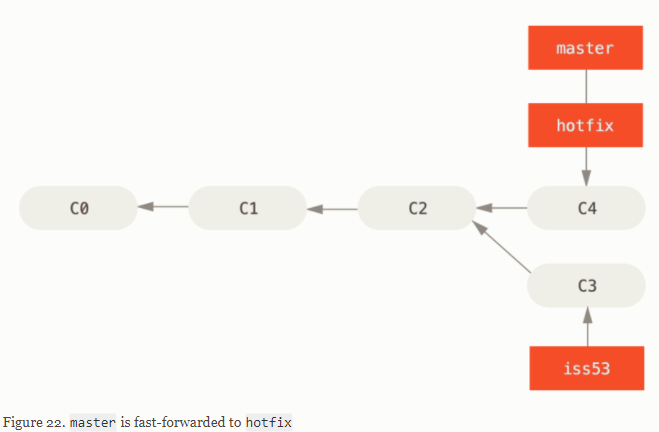


Figure : Master branch is fast-forward to hotfix.

When you done with hotfix branch, you can delete it with -d option to git branch.

$ git branch -d hotfix

Deleted branch hotfix (3a0874c).

Now, you come back to your work-in-process branch on issue53.

$ git checkout iss53

Switched to branch "iss53"

$ vim index.html

$ git commit -a -m 'Finish the new footer [issue 53]'

[iss53 ad82d7a] Finish the new footer [issue 53]

1 file changed, 1 insertion(+)

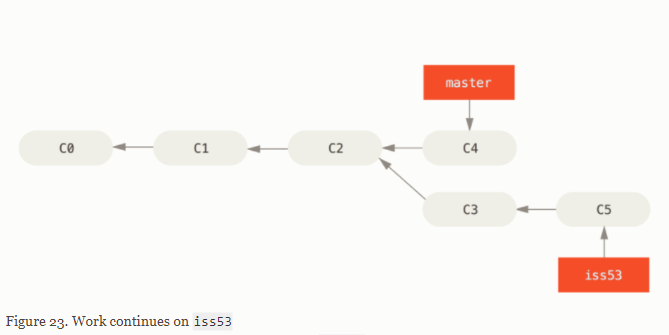


Figure : Work continues on iss53.

#### Basic Merging

Suppose you have done on issue #53 and ready to be merged into your master branch.

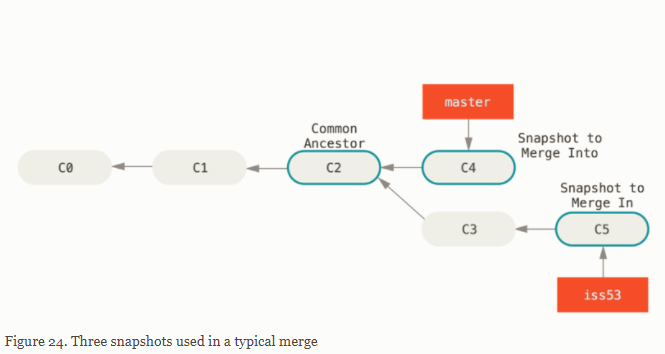


Figure : Three snapshots used in a typical merge.

Instead of move the branch forward, git creates a new snapshot that results from this three-way merge and automatically create a new commit that point to it.

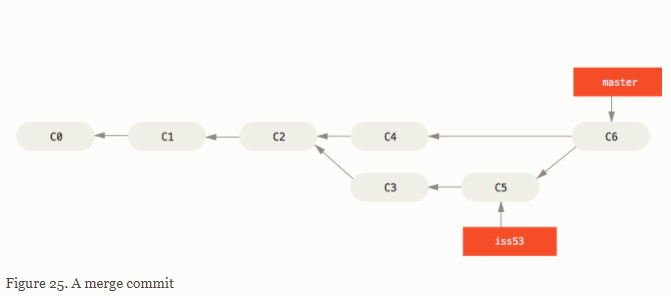


Figure : A merge commit.

If you no further need iss53 branch, you can close the issue in your issue-tracking system, and delete the branch.

$ git branch -d iss53

#### Basic Merge Conflicts

If you changes same part of same file in two different branch, Git won’t be able to merge them cleanly.

$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

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

Git hasn’t made a new merge commit. It has paused process while you resolve the conflict.

$ git status

On 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")

Run git status to show you the file that has conflict. Git adds markers to the file that has conflict.

<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

please contact us at support@github.com

</div>

>>>>>>> iss53:index.html

After you’ve resolved each of sections in each conflicted file, run git add and git commit to finish merging process.

### 3.3 Branch Management

The git branch without option will show you a list of branches.

$ git branch

iss53

\* master

testing

The git branch command with -v option will show you a list of branches and the last commit on each branch.

$ git branch -v

iss53 93b412c Fix javascript issue

\* master 7a98805 Merge branch 'iss53'

testing 782fd34 Add scott to the author list in the readme

Run git branch with --merge option to show the braches have been merged already. Then, you can run git branch -d to delete them.

$ git branch --merged

iss53

\* master

Run git branch with --no-merge option to show the branches haven’t been merged yet.

$ git branch --no-merged

testing

Git shows you an error when you try to delete a branch that hasn’t been merged yet.

$ git branch -d testing

error: The branch 'testing' is not fully merged.

If you are sure you want to delete it, run 'git branch -D testing'.

If you don’t want to keep it anyway, you can use -D to delete it.

#### Changing a branch name

Suppose that you have a branch named bad-branch-name and you want to change it to corrected-branch-name, while keep all history.

$ git branch --move bad-branch-name corrected-branch-name

The command renames the branch’s name on your local. To let others see the corrected branch on the remote push it.

$ git push --set-upstream origin corrected-branch-name

You can check all branches by using git branch --all.

$ git branch --all

\* corrected-branch-name

main

remotes/origin/bad-branch-name

remotes/origin/corrected-branch-name

remotes/origin/main

Now, you can see the corrected-branch-name branch on the remote. However, the bad-branch-name is still on the remote. You can delete the remote branch by using.

$ git push origin --delete bad-branch-name

#### Changing the master branch name

$ git branch --move master main

$ git push --set-upstream origin main

$ git branch --all

\* main

remotes/origin/HEAD -> origin/master

remotes/origin/main

remotes/origin/master

Before you’re able to delete the master branch on the remote, you need to do a few tasks:

* Any projects that depend on this one will need to update their code and/or configuration.
* Update any test-runner configuration files.
* Adjust build and release scripts.
* Redirect settings on your repo host for things like the repo’s default branch, merge rules, and other things that match branch names.
* Update references to the old branch in documentation.
* Close or merge any pull requests that target the old branch.

After you done all tasks, and you can delete the master branch.

$ git push origin --delete master

### 3.4 Branching Workflows

#### Long-Running Branches

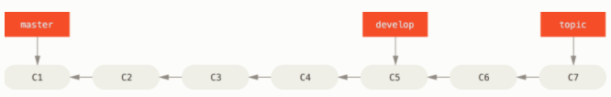


Figure : A linear view of progressive-stability branching.

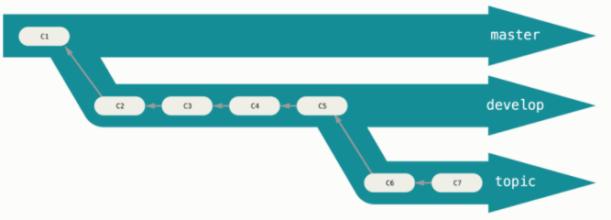


Figure : A ‘silo’ view of progressive-stability branching.

#### Topic Branches

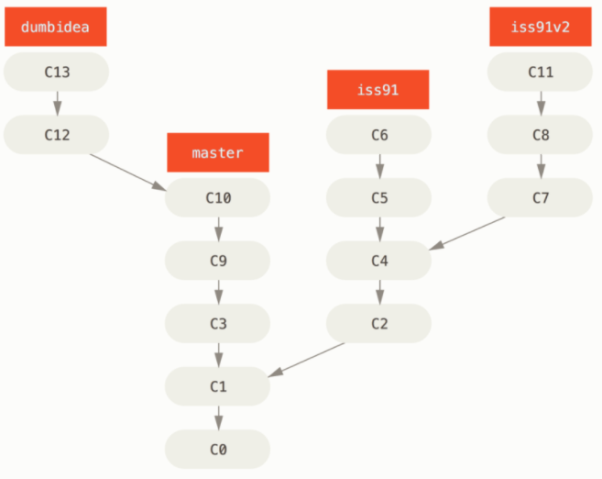


Figure : Multiple topic branches.

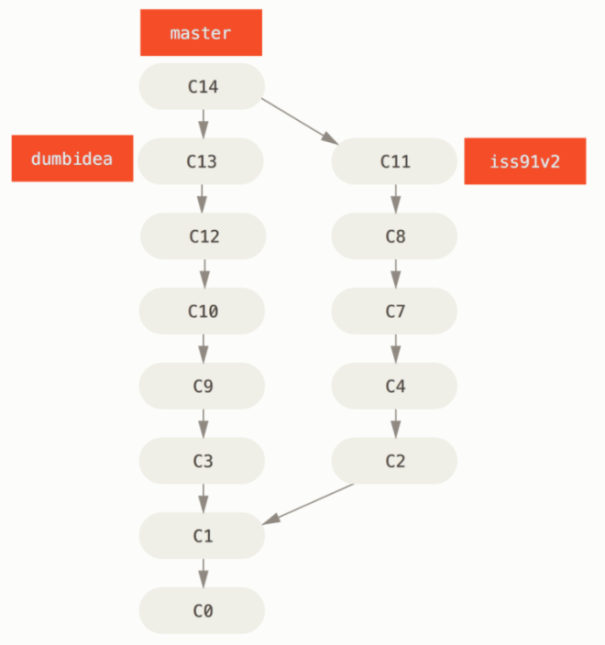


Figure : History after merging dumbidea and iss91v2.

### 3.5 Remote Branches

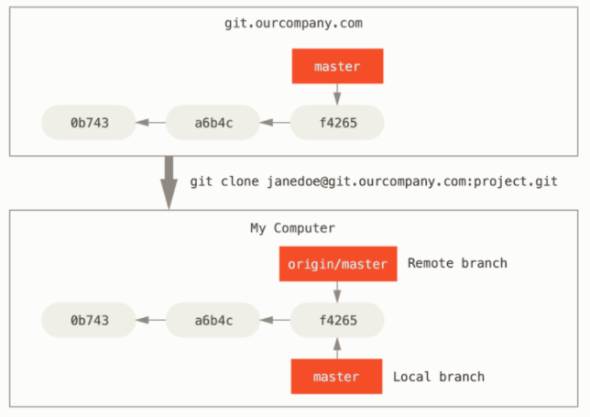


Figure : Server and local repositories after cloning.

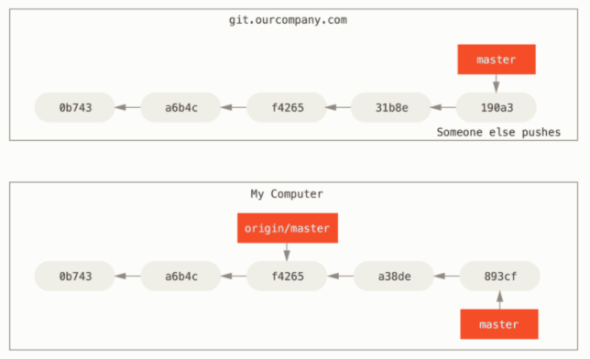


Figure : Local and remote work can diverge.

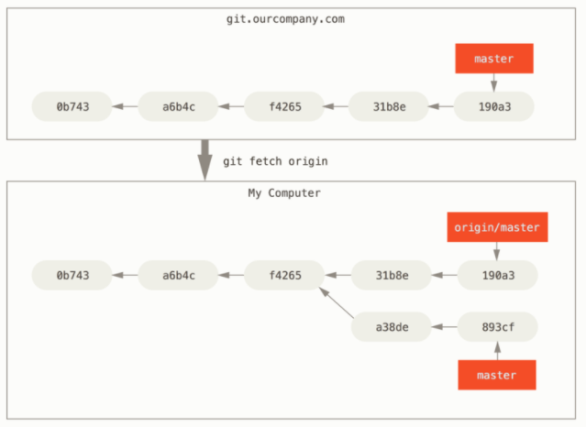


Figure : git fetch updates your remote-tracking branches.

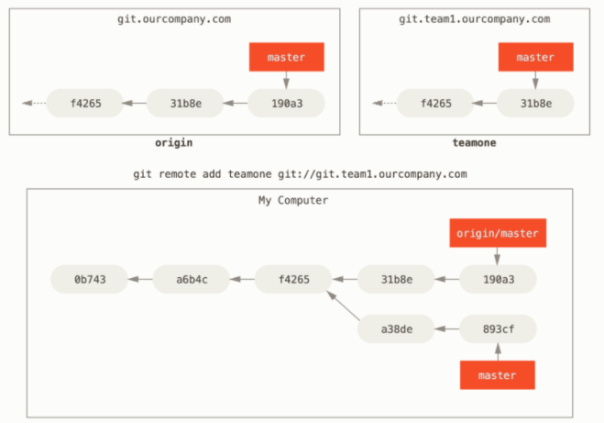


Figure : Adding another server as a remote.

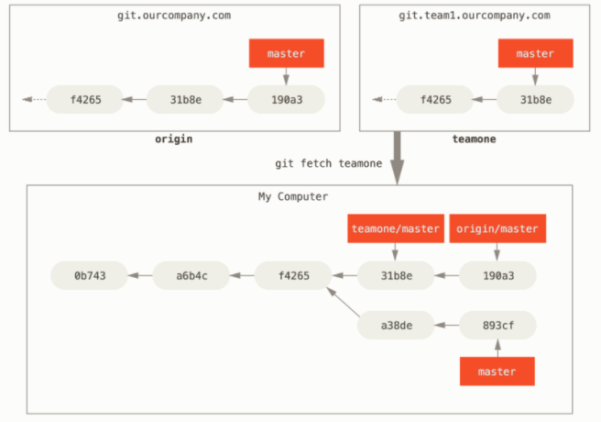


Figure : Remote-tracking branch for teamone/master.

#### Pushing

If you have a branch named serverfix that you want to share with others, you can push it up.

$ git push origin serverfix

Counting objects: 24, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (15/15), done.

Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.

Total 24 (delta 2), reused 0 (delta 0)

To https://github.com/schacon/simplegit

\* [new branch] serverfix -> serverfix

The next time one of collaborators fetches from the server, they will get a reference to where the server’s version of serverfix is under the remote branch origin/serverfix.

$ git fetch origin

remote: Counting objects: 7, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0)

Unpacking objects: 100% (3/3), done.

From https://github.com/schacon/simplegit

\* [new branch] serverfix -> origin/serverfix

When you do a fetch that brings down new remote-tracking branches, you don’t automatically have local. To merge this work into your current working branch, you can run git merge origin/serverfix.

#### Tracking Branches

To create a tracking branch for a upstream branch, running git checkout -b <branch> <remote>/<branch> or using --track shorthand:

$ git checkout --track origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'

If the branch name you’re trying to checkout doesn’t exist and exactly matches a name on only one remote, Git will create a tracking branch for you:

$ git checkout serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'

To set up a local branch with a different name than the remote branch:

$ git checkout -b sf origin/serverfix

Branch sf set up to track remote branch serverfix from origin.

Switched to a new branch 'sf'

If you already have a local branch and want to set it to a remote branch you just pulled down or want to change the upstream branch you’re tracking:

$ git branch -u origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

If you want to see what tracking branches you have set up, you can see the -vv option:

$ git branch -vv

iss53 7e424c3 [origin/iss53: ahead 2] Add forgotten brackets

master 1ae2a45 [origin/master] Deploy index fix

\* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] This should do it

testing 5ea463a Try something new

#### Pulling

The command git pull does fetch and merge commands.

#### Deleting Remote Branches

$ git push origin --delete serverfix

To https://github.com/schacon/simplegit

- [deleted] serverfix

### 3.6 Rebasing

### 3.7 Summary

## Chapter 4: Git on the Server

### 4.1 The Protocols

### 4.2 Getting Git on a Server

### 4.3 Generating Your SSH Public Key

### 4.4 Setting Up the Server

### 4.5 Git Deamon

### 4.6 Smart HTTP

### 4.7 GitWeb

### 4.8 GitLab

### 4.9 Third Party Hosted Options

### 4.10 Summary

## Chapter 5: Distributed Git

### 5.1 Distributed Workflow

#### Centralized Workflow

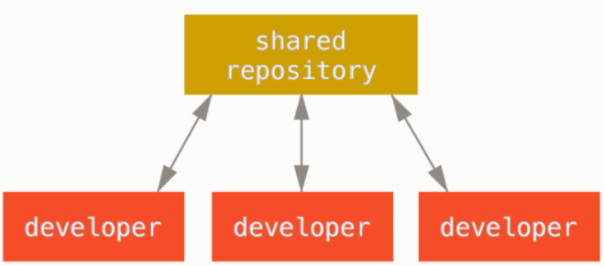


Figure : Centralized workflow.

#### Integration-Manager Workflow

The process works as follows:

1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer an email asking them to pull changes.
5. The maintainer adds the contributor’s repository as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.

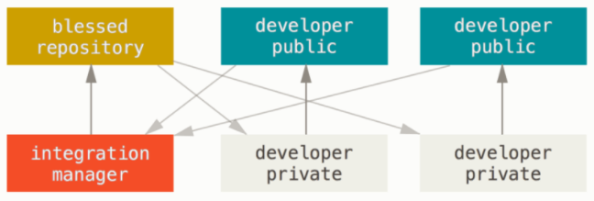


Figure : Integration-manager workflow.

#### Dictator and Lieutenants Workflow

1. Regular developers work on their topic branch and rebase their work on top of master. The master branch is that of the reference repository to which the dictator pushes.
2. Lieutenants merge the developers' topic branches into their master branch.
3. The dictator merges the lieutenants' master branches into the dictator’s master branch.
4. Finally, the dictator pushes that master branch to the reference repository so the other developers can rebase on it.

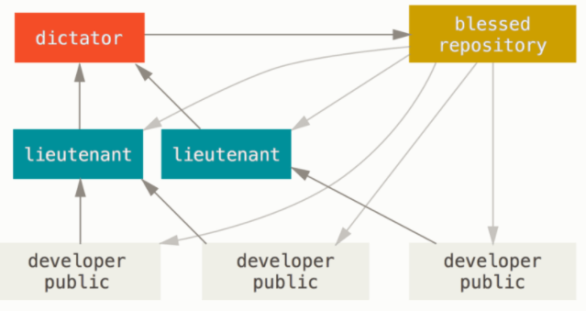


Figure : Benevolent dictator workflow.

#### Patterns for Managing Source Code Branches

https://martinfowler.com/articles/branching-patterns.html

### 5.2 Contributing to a Project

#### Commit Guidelines

First, your submissions should not contain any whitespace errors. Run git diff --check to list whitespace for you.

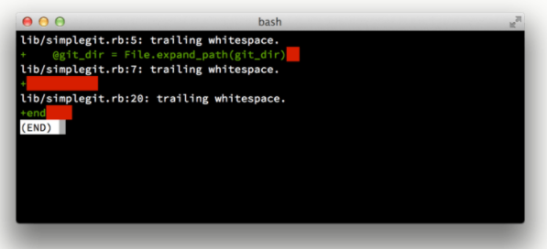


Figure : Output of git diff --check

Next, try to split your work into at least one commit per issue, with an useful message per commit.

Last, your messages should start with a single line that’s no more than about 50 characters and that describes the changeset concisely.

#### Private Small Team

The first developer, John, clones the repository

# John's Machine

$ git clone john@githost:simplegit.git

Cloning into 'simplegit'...

...

$ cd simplegit/

$ vim lib/simplegit.rb

$ git commit -am 'Remove invalid default value'

[master 738ee87] Remove invalid default value

1 files changed, 1 insertions(+), 1 deletions(-)

The second developer, Jessica, does the same thing.

# Jessica's Machine

$ git clone jessica@githost:simplegit.git

Cloning into 'simplegit'...

...

$ cd simplegit/

$ vim TODO

$ git commit -am 'Add reset task'

[master fbff5bc] Add reset task

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

Now, Jessica pushes her work to the server, which works just fine.

# Jessica's Machine

$ git push origin master

...

To jessica@githost:simplegit.git

1edee6b..fbff5bc master -> master

John makes some changes, commits them to his local repository, and tries to push them to the same server.

# John's Machine

$ git push origin master

To john@githost:simplegit.git

! [rejected] master -> master (non-fast forward)

error: failed to push some refs to 'john@githost:simplegit.git'

With Git, John must first merge the commits locally. John fetch Jessica’s upstream work.

$ git fetch origin

...

From john@githost:simplegit

+ 049d078...fbff5bc master -> origin/master

At this point, John’s local repository looks something like this.

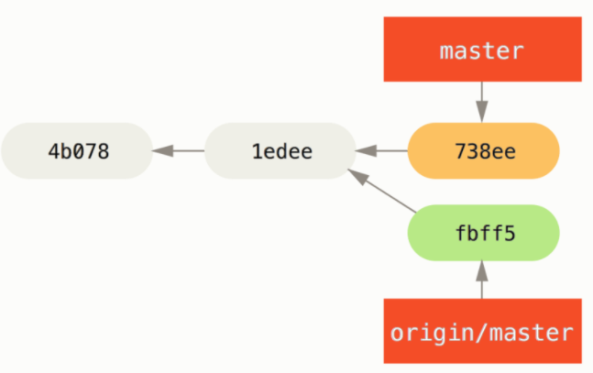


Figure : John’s divergent history

Now, John can merge Jessica’s work that he fetched into his own local work.

$ git merge origin/master

Merge made by the 'recursive' strategy.

TODO | 1 +

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

As long as that local merge goes smoothly, John’s updated history will now look like this.

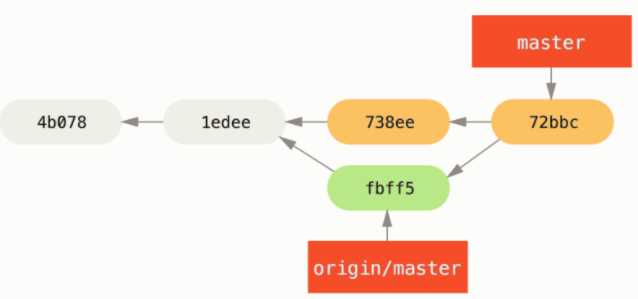


Figure : John’s repository after merging origin/master

John might want to test this new code to make sure none of Jessica’s work affects any of his and he can finally push the new merged work up to the server.

$ git push origin master

...

To john@githost:simplegit.git

fbff5bc..72bbc59 master -> master

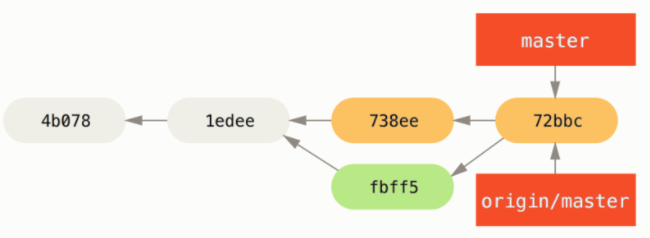


Figure : John’s history after pushing to the origin server

In the meantime, Jessica has created a new topic branch called issue54, and made three commits to that branch.

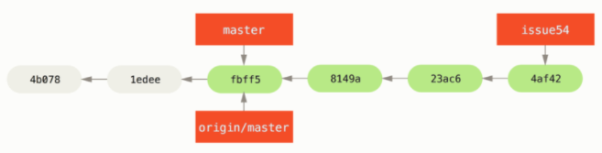


Figure : Jessica’s topic branch

Suddenly, Jessica knows that John has pushed some works on the server. She fetch all new content from the server that she does not yet have with.

# Jessica's Machine

$ git fetch origin

...

From jessica@githost:simplegit

fbff5bc..72bbc59 master -> origin/master

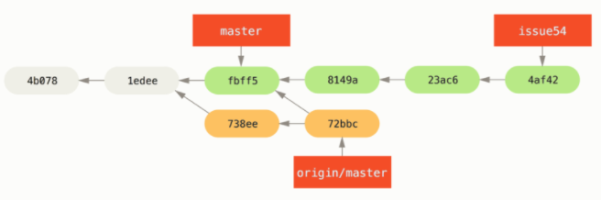


Figure : Jessica’s history after fetching John’s changes

Jessica thinks her topic branch is ready, but she want to know what part of John’s fetched work she has to merge into her work so that she can push. She run git log to find out.

$ git log --no-merges issue54..origin/master

commit 738ee872852dfaa9d6634e0dea7a324040193016

Author: John Smith <jsmith@example.com>

Date: Fri May 29 16:01:27 2009 -0700

Remove invalid default value

The issue54…origin/master is a log filter that asks Git to display only those commits that are on the origin/master branch that are not in the issue54 branch.

Now, Jessica can merge her topic work into her master branch, and then merge John’s work into her master branch, and finally push back to the server again.

$ git checkout master

Switched to branch 'master'

Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.

$ git merge issue54

Updating fbff5bc..4af4298

Fast forward

README | 1 +

lib/simplegit.rb | 6 +++++-

2 files changed, 6 insertions(+), 1 deletions(-

Everything merges cleanly, and Jessica’s history now looks like this.

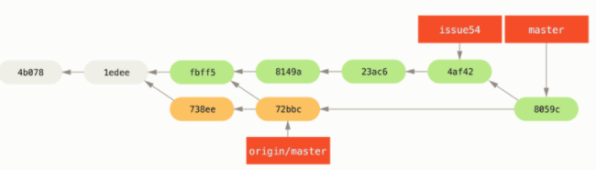


Figure : Jessica’s history after merging John’s changes

Assuming John hasn’t pushed even more changes in the meantime, so she should be able to successfully push.

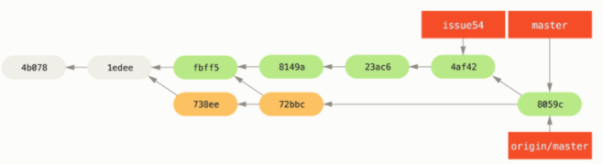


Figure : Jessica’s history after pushing all changes back to the server

This is the simplest workflows. You work for a while in a topic branch, and merge that work into your master branch when it’s ready to be integrated. When you want to share your work, you fetch and merge your master from origin/master if it has change, and finally push to the master branch on the server.

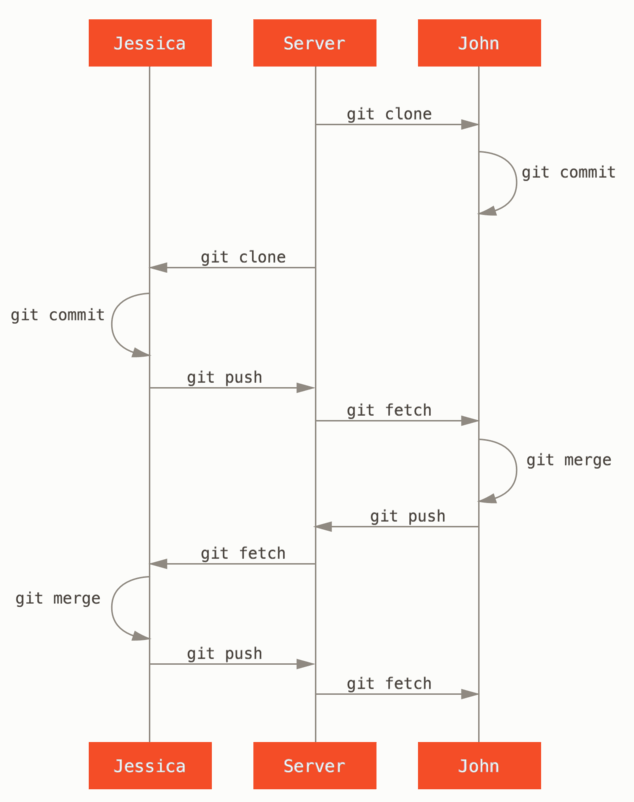


Figure : General sequence of events for a simple multiple-developer Git workflow

#### Private Managed Team

Let’s say that Jessica is working together on one feature A with John, while Jessica and a third developer, Josie, are working on a second feature B. Let’s follow Jessica’s workflow as she works on her two features, collaborating in parallel with two different developers in this environment.

Assuming Jessica already has cloned her repository. She decided to work on featureA first. She create a new branch for the feature and does some work on it.

# Jessica's Machine

$ git checkout -b featureA

Switched to a new branch 'featureA'

$ vim lib/simplegit.rb

$ git commit -am 'Add limit to log function'

[featureA 3300904] Add limit to log function

1 files changed, 1 insertions(+), 1 deletions(-)

She needs to share her work with John, so she pushed her featureA branch commit up to server. She doesn’t have push access to the master branch. She has to push to another branch in order to collaborate with John.

$ git push -u origin featureA

...

To jessica@githost:simplegit.git

\* [new branch] featureA -> featureA

Jessica emails John to tell him that she’s pushed some work into a branch named featureA and he can look at it now. While she’s waiting for John’s feedbacks, Jessica decides to start working on feature with Josie. To begin, she starts a new feature branch.

# Jessica's Machine

$ git fetch origin

$ git checkout -b featureB origin/master

Switched to a new branch 'featureB'

Now, Jessica makes a couple of commits on the feature branch.

$ vim lib/simplegit.rb

$ git commit -am 'Make ls-tree function recursive'

[featureB e5b0fdc] Make ls-tree function recursive

1 files changed, 1 insertions(+), 1 deletions(-)

$ vim lib/simplegit.rb

$ git commit -am 'Add ls-files'

[featureB 8512791] Add ls-files

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

Jessica’s repository now looks like this.

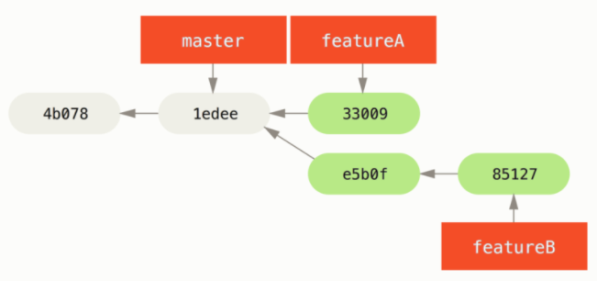


Figure : Jessica’s initial commit history

She has already to push her work, but gets an email from Josie that a branch with some initial “featureB” work on it was already pushed to the server as the featureBee branch. Jessica first fetches Josie’s changes with git fetch.

$ git fetch origin

...

From jessica@githost:simplegit

\* [new branch] featureBee -> origin/featureBee

Assuming Jessica is still on her checkout feature branch, she can now merge Josie’s work into that branch with git merge.

$ git merge origin/featureBee

Auto-merging lib/simplegit.rb

Merge made by the 'recursive' strategy.

lib/simplegit.rb | 4 ++++

1 files changed, 4 insertions(+), 0 deletions(-)

At this point, Jessica wants to push all of this merged “featuredB” work back to the server, but she doesn’t want to simply push her own feature branch. Jessica wants to push to featureBee branch.

$ git push -u origin featureB:featureBee

...

To jessica@githost:simplegit.git

fba9af8..cd685d1 featureB -> featureBee

Suddenly, Jessica gets email from John, who tells her he’s pushed some changes to featureA branch on which they are collaborating, and he asks Jessica to take a look at them. Again, Jessica runs a simple git fetch to fetch all new content from server.

$ git fetch origin

...

From jessica@githost:simplegit

3300904..aad881d featureA -> origin/featureA

Jessica can display the log of John’s new work by comparing the content of the newly-fetched featured branch with her local copy of same branch.

$ git log featureA..origin/featureA

commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6

Author: John Smith <jsmith@example.com>

Date: Fri May 29 19:57:33 2009 -0700

Increase log output to 30 from 25

If Jessica likes what she sees, she can merge John’s new work into her local featureA branch with:

$ git checkout featureA

Switched to branch 'featureA'

$ git merge origin/featureA

Updating 3300904..aad881d

Fast forward

lib/simplegit.rb | 10 +++++++++-

1 files changed, 9 insertions(+), 1 deletions(-)

Finally, Jessica might want to make a couple minor changes to all that merged content, so she is free to make those changes, commit them to her local featureA branch, and push the end result back to the server.

$ git commit -am 'Add small tweak to merged content'

[featureA 774b3ed] Add small tweak to merged content

1 files changed, 1 insertions(+), 1 deletions(-)

$ git push

...

To jessica@githost:simplegit.git

3300904..774b3ed featureA -> featureA

Jessica’s commit history now looks something like this.

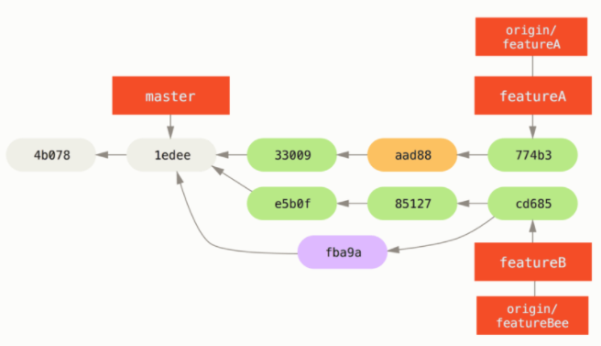


Figure : Jessica’s history after committing on a feature branch

At some point, Jessica, Josie, and John inform the integrators that the featureA and featureBee branches on the server are ready for integration into the mainline.

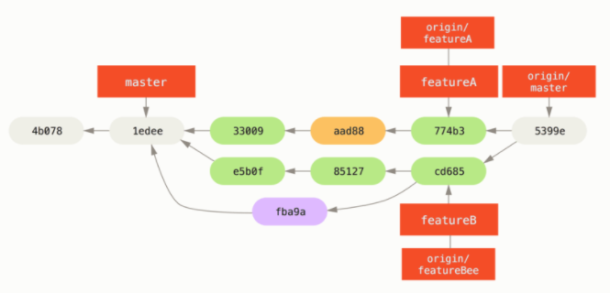


Figure : Jessica’s history after merging both her topic branches

The sequence for the workflow you saw here is something like this.

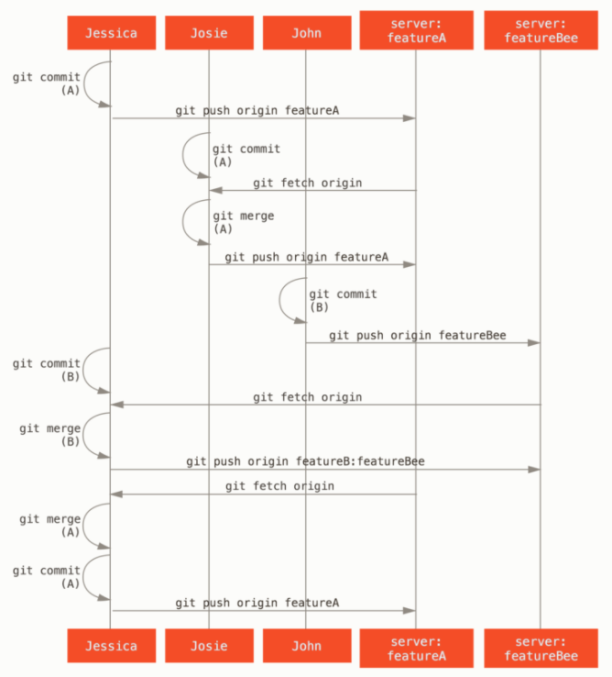


Figure : Basic sequence of this managed-team workflow

#### Forked Public Project

First, you’ll clone the main repository, create a topic branch, and do your work there.

$ git clone <url>

$ cd project

$ git checkout -b featureA

... work ...

$ git commit

... work ...

$ git commit

When your branch work is finished and you already to contribute it back to the maintainers, go to the original project page and click the “Fork” button, creating your own fork of the project. You then need to add this repository URL as a new remote of your local repository.

$ git remote add myfork <url>

You then need to push your work to this repository.

$ git push -u myfork featureA

Once you work has been pushed to your fork of the repository, you need to notify the maintainers of the original project that you have work you’d like them to merge.

$ git request-pull origin/master myfork

The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:

Jessica Smith (1):

Create new function

are available in the git repository at:

git://githost/simplegit.git featureA

Jessica Smith (2):

Add limit to log function

Increase log output to 30 from 25

lib/simplegit.rb | 10 +++++++++-

1 files changed, 9 insertions(+), 1 deletions(-)

This output can be sent to the maintainer.

$ git checkout -b featureB origin/master

... work ...

$ git commit

$ git push myfork featureB

$ git request-pull origin/master myfork

... email generated request pull to maintainer ...

$ git fetch origin

Now, each of your topics is contained with a silo.

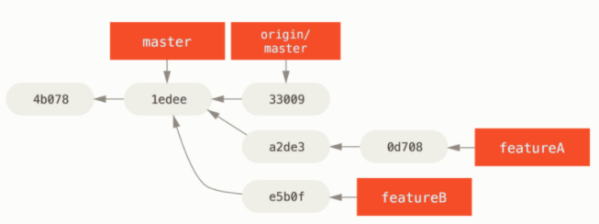


Figure : Initial commit history with featureB work

$ git checkout featureA

$ git rebase origin/master

$ git push -f myfork featureA

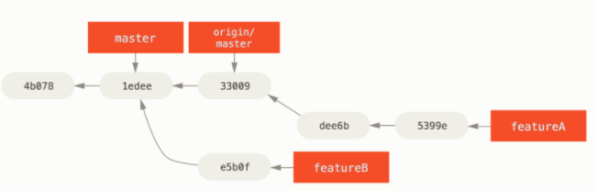


Figure : Commit history after featureA work

$ git checkout -b featureBv2 origin/master

$ git merge --squash featureB

... change implementation ...

$ git commit

$ git push myfork featureBv2

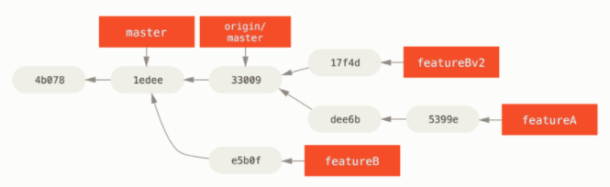


Figure : Commit history after featureBv2 work

#### Public Project over Email

### 5.3 Maintaining a Project

### 5.4 Summary

## Chapter 6: GitHub

### 6.1 Account Setup and Configuration

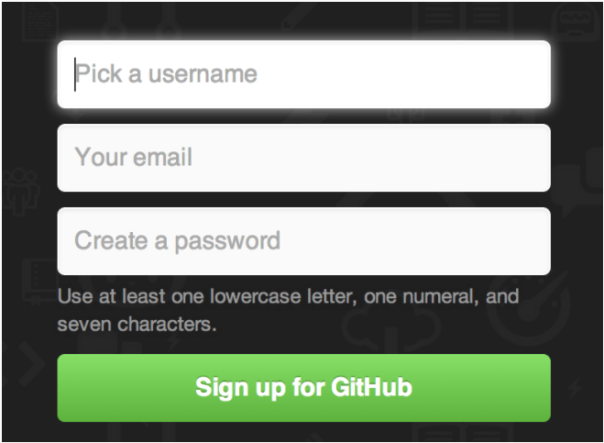


Figure : The GitHub sign-up form

#### SSH Access

Generating a new SSH key

* Open Git Bash
* Paste the text below, substituting in your GitHub email address.
  + $ ssh-keygen -t ed25519 -C "*your\_email@example.com*"
  + **Note:** If you are using a legacy system that doesn't support the Ed25519 algorithm, use:
  + $ ssh-keygen -t rsa -b 4096 -C *your\_email@example.com*
* This creates a new SSH key, using the provided email as a label.
  + Generating public/private *algorithm* key pair.
* When you’re prompted to “Enter a file in which to save the key”, press Enter. This accepts the default file location.
  + Enter a file in which to save the key (/c/Users/*you*/.ssh/):*[Press enter]*
* At the prompt, type a secure passphrase. For more information, see ["Working with SSH key passphrases](https://docs.github.com/en/articles/working-with-ssh-key-passphrases)."
  + Enter passphrase (empty for no passphrase): *[Type a passphrase]*
  + Enter same passphrase again: *[Type passphrase again]*

Add SSH key

* Click the “Add an SSH key” button,
* Give your key a name,
* Paste the contents of your ~/.ssh/id\_rsa.pub public-key file into the text area, and click “Add key”

#### Your Avatar

#### Your Email Address

#### Two Factor Authentication

### 6.2 Contributing to a Project

#### Forking Projects

When you ‘folk’ a project, GitHub will make a copy of the project that is entirely yours; it lives in your namespace, and you can push to it. People can fork a project, push to it, and contribute their changes back to the original repository by creating what’s call Pull Request.

To fork a project, visit the project and click the “Fork” button at the top-right of the page. After a few seconds, you’ll be taken to your page, with your own writeable copy of the code.



Figure : The “Fork” button

#### The GitHub Flow

Here’s how it generally works:

1. Fork the project.
2. Create a topic branch from master.
3. Make some commits to improve the project.
4. Push this branch to your GitHub project.
5. Open a Pull Request on GitHub.
6. Discuss, and optionally continue committing.
7. The project owner merges or closes the Pull Request.
8. Sync the updated master back to your fork.

#### Creating a Pull Request

### 6.3 Maintaining a Project

### 6.4 Managing an organization

### 6.5 Scripting GitHub