**A PARACTICAL INTRO TO GIT**

1. [Introduction](http://marc.helbling.fr/2014/09/practical-git-introduction#introduction)
   1. [Why a version control system?](http://marc.helbling.fr/2014/09/practical-git-introduction#why-a-version-control-system?)
   2. [A short history of vcs](http://marc.helbling.fr/2014/09/practical-git-introduction#a-short-history-of-vcs)
   3. [Why git?](http://marc.helbling.fr/2014/09/practical-git-introduction#why-git?)
   4. [About this document](http://marc.helbling.fr/2014/09/practical-git-introduction#about-this-document)
   5. [Setup](http://marc.helbling.fr/2014/09/practical-git-introduction#setup)
2. [git basics](http://marc.helbling.fr/2014/09/practical-git-introduction#git-basics)
   1. [First commit!](http://marc.helbling.fr/2014/09/practical-git-introduction#first-commit!)
   2. [Staging area](http://marc.helbling.fr/2014/09/practical-git-introduction#staging-area)
   3. [Data store](http://marc.helbling.fr/2014/09/practical-git-introduction#data-store)
      * [Objects](http://marc.helbling.fr/2014/09/practical-git-introduction#objects)
      * [Physical storage](http://marc.helbling.fr/2014/09/practical-git-introduction#physical-storage)
   4. [Immutability](http://marc.helbling.fr/2014/09/practical-git-introduction#immutability)
   5. [Reflog](http://marc.helbling.fr/2014/09/practical-git-introduction#reflog)
   6. [tl;dr](http://marc.helbling.fr/2014/09/practical-git-introduction#git-basics:-tl;dr)
3. [Branches and remotes](http://marc.helbling.fr/2014/09/practical-git-introduction#branches-and-remotes)
   1. [Branches](http://marc.helbling.fr/2014/09/practical-git-introduction#branches)
   2. [Comparing](http://marc.helbling.fr/2014/09/practical-git-introduction#comparing)
      * [Referencing parent commits](http://marc.helbling.fr/2014/09/practical-git-introduction#referencing-parent-commits)
      * [Comparing content](http://marc.helbling.fr/2014/09/practical-git-introduction#comparing-content)
      * [Comparing commits](http://marc.helbling.fr/2014/09/practical-git-introduction#comparing-commits)
      * [Comparing branches](http://marc.helbling.fr/2014/09/practical-git-introduction#comparing-branches)
   3. [Merging](http://marc.helbling.fr/2014/09/practical-git-introduction#merging)
      * [Three-way merge](http://marc.helbling.fr/2014/09/practical-git-introduction#three-way-merge)
      * [Fast forward](http://marc.helbling.fr/2014/09/practical-git-introduction#fast-forward)
      * [Resolving conflicts](http://marc.helbling.fr/2014/09/practical-git-introduction#resolving-conflicts)
   4. [Rebasing](http://marc.helbling.fr/2014/09/practical-git-introduction#rebasing)
      * [Rewriting history](http://marc.helbling.fr/2014/09/practical-git-introduction#rewriting-history)
   5. [Rebase caveats](http://marc.helbling.fr/2014/09/practical-git-introduction#rebase-caveats)
   6. [Remotes](http://marc.helbling.fr/2014/09/practical-git-introduction#remotes)
      * [Push](http://marc.helbling.fr/2014/09/practical-git-introduction#push)
      * [Fetch](http://marc.helbling.fr/2014/09/practical-git-introduction#fetch)
      * [Pull](http://marc.helbling.fr/2014/09/practical-git-introduction#pull)
      * [Force push](http://marc.helbling.fr/2014/09/practical-git-introduction#force-push)
   7. [tl;dr](http://marc.helbling.fr/2014/09/practical-git-introduction#branches-and-remotes:-tl;dr)
4. [Good practices](http://marc.helbling.fr/2014/09/practical-git-introduction#good-practices)
   1. [Workflow](http://marc.helbling.fr/2014/09/practical-git-introduction#workflow)
      * [Branching flow](http://marc.helbling.fr/2014/09/practical-git-introduction#branching-flow)
      * [Merge or rebase?](http://marc.helbling.fr/2014/09/practical-git-introduction#merge-or-rebase?)
   2. [What’s a good commit?](http://marc.helbling.fr/2014/09/practical-git-introduction#what’s-a-good-commit?)
      * [Atomicity](http://marc.helbling.fr/2014/09/practical-git-introduction#atomicity)
      * [Hiding sausage making](http://marc.helbling.fr/2014/09/practical-git-introduction#hiding-sausage-making)
      * [Commented hunks of code](http://marc.helbling.fr/2014/09/practical-git-introduction#commented-hunks-of-code)
   3. [What’s a good commit message?](http://marc.helbling.fr/2014/09/practical-git-introduction#what’s-a-good-commit-message?)
      * [Formatting](http://marc.helbling.fr/2014/09/practical-git-introduction#formatting)
      * [Content](http://marc.helbling.fr/2014/09/practical-git-introduction#content)
      * [Documentation](http://marc.helbling.fr/2014/09/practical-git-introduction#documentation)
5. [Going further](http://marc.helbling.fr/2014/09/practical-git-introduction)
   1. [Options](http://marc.helbling.fr/2014/09/practical-git-introduction#options)
   2. [Searching](http://marc.helbling.fr/2014/09/practical-git-introduction#searching)
   3. [Reset](http://marc.helbling.fr/2014/09/practical-git-introduction#reset)
   4. [Cherry pick](http://marc.helbling.fr/2014/09/practical-git-introduction#cherry-pick)
   5. [Stash](http://marc.helbling.fr/2014/09/practical-git-introduction#stash)
   6. [Bisect](http://marc.helbling.fr/2014/09/practical-git-introduction#bisect)
6. [Environment](http://marc.helbling.fr/2014/09/practical-git-introduction#environment)
   1. [Command-line completion](http://marc.helbling.fr/2014/09/practical-git-introduction#command-line-completion)
   2. [Configuration](http://marc.helbling.fr/2014/09/practical-git-introduction#configuration)
      * [Local/global/system](http://marc.helbling.fr/2014/09/practical-git-introduction#local/global/system)
      * [Basics](http://marc.helbling.fr/2014/09/practical-git-introduction#basics)
      * [Aliases](http://marc.helbling.fr/2014/09/practical-git-introduction#aliases)
   3. [Hooks](http://marc.helbling.fr/2014/09/practical-git-introduction#hooks)
   4. [Custom command-line prompt](http://marc.helbling.fr/2014/09/practical-git-introduction#custom-command-line-prompt)
   5. [Protocols](http://marc.helbling.fr/2014/09/practical-git-introduction#protocols)
   6. [GUIs and plugins](http://marc.helbling.fr/2014/09/practical-git-introduction#guis-and-plugins)
   7. [Jargon](http://marc.helbling.fr/2014/09/practical-git-introduction#jargon)
      * [upstream/downstream](http://marc.helbling.fr/2014/09/practical-git-introduction#upstream/downstream)
      * [bare and non-bare repository](http://marc.helbling.fr/2014/09/practical-git-introduction#bare-and-non-bare-repository)
      * [fork](http://marc.helbling.fr/2014/09/practical-git-introduction#fork)
      * [pull request](http://marc.helbling.fr/2014/09/practical-git-introduction#pull-request)
      * [porcelain/plumbing](http://marc.helbling.fr/2014/09/practical-git-introduction#porcelain/plumbing)
      * [gist](http://marc.helbling.fr/2014/09/practical-git-introduction#gist)
      * [hunk](http://marc.helbling.fr/2014/09/practical-git-introduction#hunk)
7. [References](http://marc.helbling.fr/2014/09/practical-git-introduction#references)

[**A practical git introduction**](http://marc.helbling.fr/2014/09/practical-git-introduction)

**Introduction**

In just a few years, [git](http://git-scm.com/) has become the dominant version control system in the software industry. Despite its widespread use, it often still appears as either magical or cumbersome when its core concepts are not fully grasped. This post is a walkthrough of practical git usage that will detail how git internally handles things.

**Why a Version Control System?**

When not relying on a dedicated tool to keep track of versions, one has to employ strategies such as using naming schemes like [[filename]\_v{0-9}+.doc](http://www.phdcomics.com/comics/archive/phd101212s.gif) or [timestamp]\_[filename]\_[comment].zip (where e.g. using the [ISO 8601](http://en.wikipedia.org/wiki/ISO_8601) format for dates will sort version) to keep track of things.

Simple things such as comparing versions, undoing some modifications or working on a same document in parallel quickly become very cumbersome.

This is where a [version control system](http://en.wikipedia.org/wiki/Revision_control) (vcs) becomes handy if not mandatory to use. A [vcs](http://mikemcquaid.com/2014/01/18/why-use-version-control/) is a tool that will store versions of a collection of documents. It does *not* change the document filenames, allows to undo/redo some modification and keeps a “context” for each modification set (e.g. an author, timestamp, comment for the modification etc.).

From [wikipedia](http://en.wikipedia.org/wiki/Revision_control):

*Version control is the management of changes to collections of documents. Changes are usually identified by a number or letter code termed the “revision”. Revisions can be compared, restored, and with some types of files, merged.*

Use cases for using a vcs are multiple when text is involved e.g.

* when writing a thesis requiring structure and content modifications;
* when maintaining multiple versions of one’s résumé, be it in distinct languages or when targeting multiple industries;
* when writing software code within a group of co-workers.

**A short history of vcs**

There has been [three](http://ericsink.com/vcbe/html/history_of_version_control.html) main [eras](http://codicesoftware.blogspot.com/2010/11/version-control-timeline.html) in the world of version control software:

1. local repositories operating on a single file (1970’s — 1980’s)
   * [Source Code Control System](http://en.wikipedia.org/wiki/Source_Code_Control_System)
   * [Revision Control System](http://www.gnu.org/software/rcs/)
2. centralized repositories handling full project (1990’s — )
   * [Concurrent Versions System](http://www.nongnu.org/cvs/)
   * [Perforce](http://www.perforce.com/)
   * [Subversion](https://subversion.apache.org/) (svn)
3. decentralized repository (2000’s — )
   * [git](http://git-scm.com/)
   * [mercurial](http://mercurial.selenic.com/)

Centralized versioning probably still represents the mostly used approach currently. However, decentralized repositories enable greater flexibility as no access to the central repository is required at all time and new solutions have improved branching a lot, making it very easy to have distinct histories living at the same time.

**Why git?**

Version control is not a new problem. Alternatives to git are numerous and wether git is better than those alternatives will not be discussed here. However let’s just list some of the attractive features provided by git:

* free/open source/binaries available for all major platforms
* decentralized
  + users keep a local clone, work locally at their own pace and deliver their changes when ready
  + every local clone is basically a (potentially partial) backup
* handle very large codebase and/or long history (git was built to handle the Linux kernel)
  + fast (most operations are performed locally)
  + reliable (data is mostly immutable)
* flexible regarding workflows
* with the help of [github](https://github.com/) (and also [bitbucket](https://bitbucket/) or [gitlab](https://gitlab.com/)) , it is now becoming *the*standard decentralized vcs
* [git](https://git.wiki.kernel.org/index.php/Git_FAQ#Why_the_.27Git.27_name.3F) philosophy is to perform simple operations and let complex ones — that actually occur not so frequently — to the user so it does no black magic

*One of the things that makes Git a pleasure to use for me is that I actually trust what Git does, because what Git does in the end is very, very stupid.*

[Linus Torvald](http://lwn.net/Articles/356626/)

**About this document**

This document intends to be a progressive introduction going from beginner’s usage (understanding what a commit is, handling branches), to intermediate usage (writing clean commits and being able to manipulate the commits history) and hopefully advanced usage. It takes a “learn the hard way” path: it only makes use of the command line and exposes some internals. Having to dive into git internals could be seen as an engineering failure, however those are pretty quick to cover and git is a very good example of how a very few low-level objects can offer powerful high-level user features.

Commands that should be typed are prefixed with the classical shell prompt $ and command output always follows. Seeing a block starting with #!EDITOR means we are editing from a text editor; if you are not familiar with a source code editor, please first check for [sublime text](http://www.sublimetext.com/), [vim](http://vim.org/), [emacs](http://www.gnu.org/software/emacs/) or whatever piece of software intended to edit raw text (which means *not* MS Word).

This document might be regularly updated; see the [history](https://github.com/marchelbling/marchelbling.github.io/commits/master/_posts/2014-09-22-practical-git-introduction.md) for the list of changes. Some [slides](http://marc.helbling.fr/talks/git.html) accompany this writing as well as some [exerices](https://gist.github.com/marchelbling/d103ef2ab0bbd89b2595).

**Setup**

Before running git commands, we need:

* git installed (preferably ≥ 2.0.0 as the default behavior for some commands has been greatly improved in late versions; see git 2.0.0 [changelog](https://git.kernel.org/cgit/git/git.git/tree/Documentation/RelNotes/2.0.0.txt))
* a git user i.e.:
  + a name git config --global user.name "My Name"
  + an email git config --global user.email "me@mail.org"

As mentioned previously, it is necessary to have a text editor installed (and presumably the [EDITOR](http://askubuntu.com/questions/432524/how-do-i-find-and-set-my-editor-environment-variable) environment variable). Setting up the command-line [tab completion](http://marc.helbling.fr/2014/09/practical-git-introduction#command-line-completion) script might also ease typing a lot.

Let’s start by initializing a new repository

$ git init $HOME/bonjour

Initialized empty Git repository in /Users/marc/bonjour/.git/

$ cd $HOME/bonjour

$ tree -a -I hooks

.

└── .git

├── HEAD

├── config

├── description

├── info

│   └── exclude

├── objects

│   ├── info

│   └── pack

└── refs

├── heads

└── tags

git init just created a hidden repository to contain internal data that we’ll describe later and that’s it, we’ve got a git repository!

**A note on revision control UX**

Newcomers to revision control systems are often confused about how the vcs changes the way they interact with the filesystem. The good news is: it basically changes nothing!

One should still edit and modify files and folders as if they were not under revision control and simply create a new revision whenever desired. Some other commands will usually allow to visualize a previous revision, compare some revisions, come back to a previous revision etc. In git, these commands will use the information stored in the .git folder that we will inspect.

**git basics**

**First commit!**

Before diving quickly into git internals, let’s create our first commit:

$ git status

On branch master

Initial commit

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

$ echo *"A dummy app listing ways to just say 'hello'"* > README.md

$ git status

On branch master

Initial commit

Untracked files:

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

README.md

$ git add README.md

$ git status

On branch master

Initial commit

Changes to be committed:

(use *"git rm --cached <file>..."* to unstage)

new file: README.md

$ git commit -m *"First commit"*

[master (root-commit) 45de2f7] First commit

1 file changed, 1 insertion(+)

create mode 100644 README.md

$ git status

On branch master

nothing to commit, working directory clean

Note that at each step, the git status command provides helpful overview of the current repository state and a *contextual* help.

**Staging area**

git uses a two-phase commit:

1. add desired changes to the staging area
2. commit changes.

After having changed some files, we may select the changes that should be committed using:

* git add to add a new file or some modification in an already tracked file; the --patch option allows to select hunks only
* git rm to remove a file from tracking
* git mv to rename a file

The staging area is an important aspect of git as it is what connects the local file system to the internal git storage. It is described by git status as *“changes to be committed”* and thus it is important to think of it as a commit draft.

$ mkdir {fr,en}

$ echo *'bonjour'* > fr/data

$ echo *'hello'* > en/data

$ git add fr/data

$ git status

On branch master

Changes to be committed:

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

new file: fr/data

Untracked files:

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

en/

Notice that we do not have to stage all local modifications. Once we are done selecting the changes that belong to a new revision, we may actually commit those changes.

$ git commit -m *"Create french data file"*

[master 13d1b4b] Create french data file

1 file changed, 1 insertion(+)

create mode 100644 fr/data

This example is really simple; a more realistic workflow would be:

1. make local changes
2. stage meaningful changes/unstage undesired ones (using git reset [--patch] HEAD file)
3. go to 1. until staged changes perform the desired goal
4. commit the changes.

Now that we have two commits, we may look at how git handles our data internally to better understand git mechanics.

**Data store**

**Objects**

After creating a commit, git displays a unique id for the newly created commit. We may use some git commands to inspect this [data](http://git-scm.com/book/en/Git-Internals-Git-Objects)

$ git cat-file -t 13d1b4b

commit

$ git cat-file commit 13d1b4b

tree 7efc3caa79efbab80f45335d4d5f8d2885daff29

parent 45de2f713305a9dcd3e82833653153c19081f36e

author Marc Helbling <marc.d.helbling@gmail.com> 1413120925 +0200

committer Marc Helbling <marc.d.helbling@gmail.com> 1413120925 +0200

Create french data file

From this, we see that a [commit](http://git-scm.com/docs/git-commit-tree) references:

* a “tree”: git internal description of the filesystem
* a parent: git is fundamentally a **direct acyclic graph** ([DAG](http://en.wikipedia.org/wiki/Directed_acyclic_graph)) in which nodes are commits that references their parent commit(s)
* an author: the person who originally wrote the current commit *content*
* an author date
* a committer: the person who created the git commit on behalf of the author (which, as in our case, may be the same person as the author)
* a committer date
* a message: the description for the changes contained in the commit.

This is the high level definition of a commit in git. We need to go one more step into internals and inspect the “tree” object to have a better picture of how git structures data.

$ git cat-file tree 7efc3caa79efbab80f45335d4d5f8d2885daff29

100644 README.mdF??Bd???*"?C-%D?J?x40000 fr\bbS?"*}nC??WJ!

$ git ls-tree 7efc3caa79efbab80f45335d4d5f8d2885daff29

100644 blob 4695a64264e4d7ea22d9432d25449f1e4aeb781e README.md

040000 tree 5c7f626253bb14227d6e074382ee91574a180f21 fr

$ git cat-file blob 4695a64264e4d7ea22d9432d25449f1e4aeb781e

A dummy app listing ways to just say *'hello'*

$ git ls-tree 5c7f626253bb14227d6e074382ee91574a180f21

100644 blob 1cd909e05d33f0f6bc4ea1caf19b5749b434ceb3 data

$ git cat-file -p 1cd909e05d33f0f6bc4ea1caf19b5749b434ceb3

bonjour

So we can see that:

* a “tree” contains pointers to “blobs” and other trees and a name for each pointer
* a “blob” is bunch of bytes representing user content (text, images etc.)
* both trees and blobs store a [file mode](http://en.wikipedia.org/wiki/Modes_%28Unix%29) (i.e. a [chmod](http://linux.die.net/man/1/chmod)); note however that the file ownership ([chown](http://linux.die.net/man/1/chown)) will depend on the user that performs the git commands and is up to the final user
* git performs deduplication based on content: if file foo is an exact copy of the file bar
  + they will be represented by the same blob
  + the tree will point to 2 blobs with the same id but referring different names and possibly different file modes

There are 4 git objects (listed from “low” to “high” level) that can be described as:

* blob: content
* tree: file tree description
* commit: project snapshot with contextual metadata
* [tag](http://git-scm.com/docs/git-tag): frozen commit name (we will not cover tags and refer to the [documentation](http://git-scm.com/book/en/v2/Git-Basics-Tagging)for more details)

**Physical storage**

As git handles history of files, we may ask ourselves how does git stores incremental differences for our data. To test this, let’s add some content in an existing file

$ echo *"salut"* >> fr/data

$ git add -u

$ git commit -m *"Add more frensh data"*

[master 456a082] Add more frensh data

1 file changed, 1 insertion(+)

and inspect the resulting blob object

$ git ls-tree 456a082

100644 blob 4695a64264e4d7ea22d9432d25449f1e4aeb781e README.md

040000 tree 3d9ba4d12442602bd81928438f80810622b9fd56 fr

$ git ls-tree 3d9ba4d12442602bd81928438f80810622b9fd56

100644 blob bd61b2ccb39197cc3a66b43f52a6fed66a237a29 data

$ git cat-file blob bd61b2c

bonjour

salut

The conclusion is that git stores pointers to “full” blobs which means that a blob is useful by itself, independently of the history file it represents. Practically, this means that even a shallow repository is usable (especially for [git≥1.9.0](https://raw.githubusercontent.com/git/git/master/Documentation/RelNotes/1.9.0.txt)). This could seem inefficient as for each file, git will keep a copy of the full content after each commit. However, git may also create “[packfiles](http://git-scm.com/book/en/Git-Internals-Packfiles)” that represent content ‘delta’s to optimize disk usage.

Time to examine how the data is actually stored.

$ cat .git/objects/bd/61b2ccb39197cc3a66b43f52a6fed66a237a29

xK??OR04aH????/-?\*N?)-?S?a

$ python -c *"""*

*import zlib, sys;*

*print repr(zlib.decompress(sys.stdin.read()))*

*"""* < .git/objects/bd/61b2ccb39197cc3a66b43f52a6fed66a237a29

*'blob 14\x00bonjour\nsalut\n'*

$ wc -c fr/data

14 fr/data

git physically stores content using zlib-compressed files containing:

1. the object type (blob, tree, commit or tag)
2. the uncompressed data size
3. a null byte
4. the actual content

The commit files follow the same construction. [Tree storage](http://stackoverflow.com/a/21599232/626278) is slightly different.

The zlib compression will optimize git disk usage for storing commits (especially if blobs represent large files).

The last important point for this quest is to understand how git names his internal files. Those names correspond to a cryptographic hash of the object. The hash function used is [SHA-1](http://en.wikipedia.org/wiki/SHA-1) and it may serve as a signature to assert data integrity (i.e. the decompressed object sha1 signature should match its filename and the content size should be the same as the size stored in the object). sha1 produces 160-bit hash value that git represents as a 40 digits long hexadecimal value. You may have noticed that when referencing git [objects](http://www.gitguys.com/topics/all-git-object-types-blob-tree-commit-and-tag/), we did not always used a 40 digits long value every time. git allows to use a shorter sub-sha1 *prefix*, provided that it is not ambiguous (i.e. that it enables to reference an object uniquely). It basically means that the bigger your repository (in terms of git objects), the longer the sha1 prefix you will have to use.

$ cat fr/data | git hash-object --stdin

bd61b2ccb39197cc3a66b43f52a6fed66a237a29

$ tree .git/objects/ --matchdirs -P bd

.git/objects/

├── 13

├── 1c

├── 3d

├── 45

├── 46

├── 5c

├── 77

├── 7e

├── bd

│   └── 61b2ccb39197cc3a66b43f52a6fed66a237a29

├── cf

├── info

└── pack

git shards objects into 16×16=256 subfolders to grant a [faster access](http://thread.gmane.org/gmane.comp.version-control.git/69322) to a given sha1. This matters as all manipulations in git involve a sha1 (sometimes through [aliases](http://marc.helbling.fr/2014/09/practical-git-introduction#branches)).

In the end, we have spent a bit of time looking at how git stores *loose* objects i.e. how git stores its objects in individual files. This helped us getting the big picture of the internal storage.

We will not dig the [packfile](http://schacon.github.io/gitbook/7_the_packfile.html) [format](https://www.kernel.org/pub/software/scm/git/docs/technical/pack-format.txt) which is an optimization to avoid cluttering the disk by regrouping objects together (typically by invoking git gc).

**Immutability**

When creating our last commit, we made an horrible typo. git allows to amend the last commit using git commit --amend. If the staging area contains any modification, they will be added to the commit.

In our case, we just want to fix our typo in the commit message, so we do not add anything to the staging area:

$ git commit --amend

opens our favorite editor (as defined by the [$EDITOR](https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration#Basic-Client-Configuration) environment variable) where we may edit the commit message

*#!EDITOR*

Add more french data

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

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

*#*

*# Date: Sun Oct 12 16:19:08 2014 +0200*

*#*

*# On branch master*

*#*

*# Changes to be committed:*

*# modified: fr/data*

*#*

and simply save our modification.

[master dd0f5d6] Add more french data

Date: Sun Oct 12 16:19:08 2014 +0200

1 file changed, 1 insertion(+)

We see that git creates a *new* commit object dd0f5d6 (remember that the sha1 involves the commit message and timestamps).

$ git cat-file commit dd0f5d6

tree 77a832b508bd5d2fb7c1eb8999e6e0a9f926434d

parent 13d1b4b062b7a7308553bc504dda2d43d32525ba

author Marc Helbling <marc.d.helbling@gmail.com> 1413123548 +0200

committer Marc Helbling <marc.d.helbling@gmail.com> 1413725760 +0200

Add more french data

This means that a commit is [immutable](http://www.brainshave.com/talks/immutable-data-trees); any modification creates a new commit instead of modifying the existing one thus making git a [functional](http://www.jayway.com/2013/03/03/git-is-a-purely-functional-data-structure/) DAG. We will later see that this is an important property to have in mind when multiple people are working on the same repository. The benefit is that every object is uniquely defined and when you manipulate a sha1 you do not have to worry about not having the correct content.

**Reflog**

If we look at our commit tree

$ git log --graph --oneline

\* dd0f5d6 Add more french data

\* 13d1b4b Create french data file

\* 45de2f7 First commit

we see that we created 3 commits until now. Amending our last commit did not add a new commit in the tree, it only replaced the last commit a new commit. We could fear that any git command have a direct and irreversible impact.

However git keeps a [reflog](https://git-scm.com/docs/git-reflog) which is a record of all commits that were referenced at some point.

$ git reflog

dd0f5d6 HEAD@{0}: commit (amend): Add more french data

456a082 HEAD@{1}: commit: Add more frensh data

13d1b4b HEAD@{2}: commit: Create french data file

45de2f7 HEAD@{3}: commit (initial): First commit

Any action that has been **committed** can be retrieved later, even if it is no longer referenced in the commit tree. This allows to undo bad commands very easily and may serve as a safety net.

Note however that git has [garbage](https://www.kernel.org/pub/software/scm/git/docs/git-gc.html) [collection](http://alblue.bandlem.com/2011/11/git-tip-of-week-gc-and-pruning-this.html) commands that will remove unreachable objects. By default those commands will not remove objects that were created in the last 2 weeks.

**git basics: tl;dr**

* git is a functional DAG where nodes represents filetrees with metadata and keep a link to their parents
* staging area is the bridge between local file tree and git data store
* git internal data is stored efficiently and safely in the .git folder
* git commands are performed by passing sha1 prefixes that identify objects uniquely
* when lost, run git status
* git reflog records a reference to all created commits even when no longer reachable
* don’t mess with the .git folder!

**Branches and remotes**

**Branches**

As we have seen in the previous section, git is a graph and you may have noticed that git status names branches:

$ git status

On branch master

nothing to commit, working directory clean

$ git branch

\* master

By default, git creates a branch called master. If we inspect this object

$ git cat-file -t master

commit

$ git show --oneline master

dd0f5d6 Add more french data

diff --git a/fr/data b/fr/data

index 1cd909e..bd61b2c 100644

--- a/fr/data

+++ b/fr/data

@@ -1 +1,2 @@

bonjour

+salut

we realize that a branch is simply a pointer to a “leaf” commit also called “tip” commit. The binding name/commit is stored in

$ tree .git/refs/heads/

.git/refs/heads/

└── master

$ cat .git/refs/heads/master

dd0f5d6500d72d54747dec1dc4139f13b5fdb8f2

git also keeps an alias for current branch last commit as [HEAD](http://git-scm.com/book/en/Git-Internals-Git-References) (with a special case for [detached HEAD](http://git-scm.com/docs/git-checkout#_detached_head)).

$ git show --oneline HEAD

dd0f5d6 Add more french data

diff --git a/fr/data b/fr/data

index 1cd909e..bd61b2c 100644

--- a/fr/data

+++ b/fr/data

@@ -1 +1,2 @@

bonjour

+salut

$ cat .git/HEAD

ref: refs/heads/master

Creating new branch is very easy with git:

$ git branch structure-data

$ git branch

structure-data

\* master

$ git checkout structure-data

Switched to branch *'structure-data'*

$ git branch

\* structure-data

master

$ git show --oneline HEAD

dd0f5d6 Add more french data

diff --git a/fr/data b/fr/data

index 1cd909e..bd61b2c 100644

--- a/fr/data

+++ b/fr/data

@@ -1 +1,2 @@

bonjour

+salut

We see that our brand new branch points exactly to the same commit as our masterbranch. This is the default behavior when creating a new branch; it is assumed that the new branch will start from HEAD and may be changed by passing the desired branching commit sha1 as a second argument i.e. git branch new\_branch new\_branch\_HEAD\_commit.

$ echo -e *"# old\n\nbonjour\nsalut"* > fr/data

$ git add fr/data && git commit -m *"add 'old' header to french data"*

[structure-data dd0109a] add *'old'* header to french data

1 file changed, 2 insertions(+)

$ echo -e *"\n# modern\n\nyo"* >> fr/data

$ git add -u && git commit -m *"add modern french data"*

[structure-data 5a40d5a] add modern french data

1 file changed, 4 insertions(+)

**Comparing**

Now that we have two distinct branches, we should make sure that the changes introduced by our new branch match the intended specification. We therefore need to see the differences in content and commits.

**Referencing parent commits**

* abc123^ parent commit of commit abc123
* abc123^^ grandparent commit of commit abc123
* more generally abc123~n
  + abc123~1 ⇔ abc123^
  + abc123~2 ⇔ abc123^^

**Comparing content**

* git diff --cached: changes that have been staged
* git diff A B: changes (computed using the [longest common subsequence algorithm](http://cbx33.github.io/gitt/afterhours3-1.html) algorithm)
* git diff A...B: changes from common ancestor of A and B to B

**Comparing commits**

* git branch --contains sha1: list all branches containing the commit sha1
* git log A --not B: list commits contained in branch A that are not in branch B

**Comparing branches**

* git branch --merged: list branches that are reachable in the current branch history
* git branch --no-merged: list branches that are not reachable in current branch history

If we compare our branch content with the master branch:

$ git diff master structure-data

diff --git a/fr/data b/fr/data

index bd61b2c..ea35d1f 100644

--- a/fr/data

+++ b/fr/data

@@ -1,2 +1,8 @@

+# old

+

bonjour

salut

+

+# modern

+

+yo

$ git log --oneline structure-data --not master

5a40d5a add modern french data

dd0109a add *'old'* header to french data

$ git checkout master && git branch --merged

\* master

we see that we have been structuring our data and adding new content in two commits that are ready to be merged.

**Merging**

As we are happy with the changes introduced in our new branch, we may now make our work available in the repository trunk. There are multiple strategies to perform this, the most basic one being a merge commit. In its simplest — yet most common — form, a merge will take 2 branches and create a commit having the branches respective HEADs as parents through a merge.

**Three-way merge**

[Three-way merge](http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging) is a central algorithm in git. As its name suggests, it involves 3 distinct commits:

* the MERGE\_HEAD commit i.e. the modification that we want to merge
* the HEAD commit i.e. the branch in which the MERGE\_HEAD will be merged i.e. the branch on which the git merge command is called
* the ORIG\_HEAD commit i.e. the *best common ancester* of MERGE\_HEAD and HEADthat will serve as the reference.

The result of a three-way merge will [look like](http://www.quora.com/How-does-Git-merge-work/answer/Anders-Kaseorg): HEAD + (MERGE\_HEAD - ORIG\_HEAD)

| **HEAD** | **+** | **MERGE\_HEAD** | **-** | **ORIG\_HEAD** | **→** | **result** |
| --- | --- | --- | --- | --- | --- | --- |
| foo |  | foo |  | foo |  | foo |
| foo |  |  |  |  |  | foo |
|  |  | foo |  |  |  | foo |
|  |  | foo |  | foo |  |  |
| foo |  |  |  | foo |  |  |
| foo |  | bar |  | baz |  | *conflict* |

This short overview of how the three-way merge works assume that ORIG\_HEAD is unique which can be wrong when merge commits are involved too. By default, git uses the merge-recursive strategy that is a three-way merge where the ORIG\_HEAD is a (virtual) merge commits for all common ancestors.

So, after running

$ git merge --no-ff structure-data

> Merge branch *'structure-data'*

>

> *# Please enter a commit message to explain why this merge is necessary,*

> *# especially if it merges an updated upstream into a topic branch.*

> *#*

> *# Lines starting with '#' will be ignored, and an empty message aborts*

> *# the commit.*

Merge made by the *'recursive'* strategy.

fr/data | 6 ++++++

1 file changed, 6 insertions(+)

$ cat fr/data

*# old*

bonjour

salut

*# modern*

yo

the commit tree now looks like

$ git log --graph --oneline

\* 62cbf27 Merge branch *'structure-data'*

|*\*

| \* 5a40d5a add modern french data

| \* dd0109a add *'old'* header to french data

|/

\* dd0f5d6 Add more french data

\* 13d1b4b Create french data file

\* 45de2f7 First commit

One thing to notice is that the merge commit (62cbf27) is linked to 2 (parent) commits: dd0f5d6 and 5a40d5a. [Referencing](http://marc.helbling.fr/2014/09/practical-git-introduction#referencing-parent-commits) a merge commit ancestors may therefore be ambiguous in some contexts.

**Fast forward**

When merging, we explicitly asked git to create a merge commit using the --no-ffflag. However, looking at the commit graph, we see that it is (almost) equivalent to the simplified one

\* yyyyyyy add modern french data

\* xxxxxxx add *'old'* header to french data

|

\* dd0f5d6 Add more french data

\* 13d1b4b Create french data file

\* 45de2f7 First commit

Indeed, in this case, HEAD and ORIG\_HEAD pointed to the same commits hence HEADmay just be updated to MERGE\_HEAD. Wether one should prevent fast-forward merges or not is a matter of workflow and we will discuss this point a bit later.

**Resolving conflicts**

We have seen that in some situations, the three-way merge results in a situation where the merge content may not be automatically deduced. This is called a merge conflict. Let’s create a conflict:

$ git checkout -b modern-french

Switched to a new branch *'modern-french'*

$ echo *"jourbon"* >> fr/data

$ git add -u && git commit -m *"add new modern data"*

[modern-french 81defc8] add new modern data

1 file changed, 1 insertion(+)

$ echo -e *"\n# slang\n\nchenu reluit"* >> fr/data

$ git add -u && git commit -m *"add french slang"*

[modern-french e6defd6] add french slang

1 file changed, 5 insertions(+)

$ git checkout master

$ echo *"wesh"* >> fr/data

$ git add -u && git commit -m *"add more modern data"*

[master 075ce36] add more modern data

1 file changed, 1 insertion(+)

Let’s see what happens when we try to merge:

$ git merge modern-french

Auto-merging fr/data

CONFLICT (content): Merge conflict in fr/data

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

$ git status

On branch master

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

You have unmerged paths.

(fix conflicts and run *"git commit"*)

Unmerged paths:

(use *"git add <file>..."* to mark resolution)

both modified: fr/data

$ git diff

diff --cc fr/data

index cd1a280,5a932d1..0000000

--- a/fr/data

+++ b/fr/data

@@@ -6,4 -6,9 +6,13 @@@ salu

*# modern*

yo

++<<<<<<< HEAD

+wesh

++=======

+ jourbon

+

+

+ *# slang*

+

+ chenu reluit

++>>>>>>> modern-french

We see the content from the master branch materialized in a block delimited by<<<<<<< and ======= and the content of modern-french is delimited by ======= and >>>>>>>. By default, git only shows HEAD (on the top of a conflict) and MERGE\_HEAD(on the bottom of a conflict). Visualizing the ORIG\_HEAD content is a matter of configuration and may be achieved by setting git config --local merge.conflictstyle diff3:

$ git merge --abort

$ git merge modern-french

Auto-merging fr/data

CONFLICT (content): Merge conflict in fr/data

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

$ git diff

diff --cc fr/data

index cd1a280,5a932d1..0000000

--- a/fr/data

+++ b/fr/data

@@@ -6,4 -6,9 +6,14 @@@ salu

*# modern*

yo

++<<<<<<< HEAD

+wesh

++||||||| merged common ancestors

++=======

+ jourbon

+

+

+ *# slang*

+

+ chenu reluit

++>>>>>>> modern-french

We now have the full picture:

* there was no content before attempting to merge;
* the master branch wants to add “wesh” in the modern section;
* the modern-french branch wants to add a “slang” section.

This conflicts is easy to [solve](https://help.github.com/articles/resolving-a-merge-conflict-from-the-command-line/) by editing the file and keeping both changes and thus having the following

$ git diff

+wesh

+ jourbon

+

+

+ *# slang*

+

+ chenu reluit

To finish the conflict resolution, we may now stage our changes and commit them

$ git add -u && git commit

> Merge branch *'modern-french'*

>

> Conflicts:

> fr/data

[master b20ab97] Merge branch *'modern-french'*

and look at our commit tree

$ git log --graph --oneline

\* b20ab97 Merge branch *'modern-french'*

|*\*

| \* e6defd6 add french slang

| \* 81defc8 add new modern data

\* | 075ce36 add more modern data

|/

\* 62cbf27 Merge branch *'structure-data'*

|*\*

| \* 5a40d5a add modern french data

| \* dd0109a add *'old'* header to french data

|/

\* dd0f5d6 Add more french data

\* 13d1b4b Create french data file

\* 45de2f7 First commit

It is important to note that when feeling unsure about the resolution of a conflict:

* reflog keeps previous states so it will always be possible to undo things
* the conflict resolution can be aborted using git merge --abort.

**Rebasing**

Once again, it feels like the graph

\* b20ab97 Merge branch *'modern-french'*

|*\*

| \* e6defd6 add french slang

| \* 81defc8 add new modern data

\* | 075ce36 add more modern data

|/

could make more sense as

|

\* yyyyyyy add french slang

\* xxxxxxx add new modern data

|

\* 075ce36 add more modern data

|

Indeed, the purpose of the modern-french branch was just to add new content; we could debate about the reason why we would create a new branch as we are continuing previous work. So let’s rewind our repository to where it was before merging

$ git reset --hard 075ce36

$ git checkout modern-french && git log --graph --oneline

\* e6defd6 add french slang

\* 81defc8 add new modern data

\* 62cbf27 Merge branch 'structure-data'

|\

| \* 5a40d5a add modern french data

| \* dd0109a add 'old' header to french data

|/

\* dd0f5d6 Add more french data

\* 13d1b4b Create french data file

\* 45de2f7 First commit

What we would like to achieve is to change the parent commit of 81defc8 to master’s HEAD (075ce36). git allows to move a series of commit (or branch) and replay them on another commit (or branch) with the [git rebase](https://www.kernel.org/pub/software/scm/git/docs/git-rebase.html) command.

git rebase [works](http://git-scm.com/book/ch3-6.html) by

1. going to the common ancestor of the two branches (the one you’re on and the one you’re rebasing onto),
2. getting the diff introduced by each commit of the branch you’re on, saving those diffs to temporary files,
3. resetting the current branch to the same commit as the branch you are rebasing onto,
4. and finally applying each change in turn.

There are 3 main differences with git merge:

1. git rebase does *not* create any additional commit object
2. when running git rebase other from the current branch, git will checkout the other branch before (re)applying the commits of the current branch. Practically this means that the other branch will stand for ORIG\_HEAD and current branch for MERGE\_HEAD
3. git will drop a commit for which the computed diff is now empty.

Let’s try to rebase modern-french onto master:

$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: add new modern data

Using index info to reconstruct a base tree...

M fr/data

Falling back to patching base and 3-way merge...

Auto-merging fr/data

CONFLICT (content): Merge conflict in fr/data

Failed to merge in the changes.

Patch failed at 0001 add new modern data

The copy of the patch that failed is found in:

/private/tmp/bonjour/.git/rebase-apply/patch

When you have resolved this problem, run *"git rebase --continue"*.

If you prefer to skip this patch, run *"git rebase --skip"* instead.

To check out the original branch and stop rebasing, run *"git rebase --abort"*.

$ git diff

diff --cc fr/data

index cd1a280,89bb164..0000000

--- a/fr/data

+++ b/fr/data

@@@ -6,4 -6,4 +6,9 @@@ salu

*# modern*

yo

++<<<<<<< HEAD

+wesh

++||||||| merged common ancestors

++=======

+ jourbon

++>>>>>>> add new modern data

When applying the commits, git uses the three-way merge algorithm which explains that we have a conflict similar to the one we had with a merge. We edit the conflict as previously:

$ git diff

diff --cc fr/data

index cd1a280,89bb164..0000000

--- a/fr/data

+++ b/fr/data

@@@ -6,4 -6,4 +6,5 @@@ salu

*# modern*

yo

+wesh

+ jourbon

and then stage our modification to continue the rebase process

$ git add -u && git rebase --continue

Applying: add new modern data

Applying: add french slang

Using index info to reconstruct a base tree...

M fr/data

Falling back to patching base and 3-way merge...

Auto-merging fr/data

$ git log --graph --oneline

\* b588260 add french slang

\* 7a48ea2 add new modern data

\* 075ce36 add more modern data

\* 62cbf27 Merge branch *'structure-data'*

|*\*

| \* 5a40d5a add modern french data

| \* dd0109a add *'old'* header to french data

|/

\* dd0f5d6 Add more french data

\* 13d1b4b Create french data file

\* 45de2f7 First commit

We have thus linearized our commit history by rebasing the modern-french branch onto the master branch.

Once again:

* git reflog would enable to undo a faulty rebase if needed
* at any conflict resolution, git rebase --abort would stop the rebase process and put the branch back at its original state (all rebase information is stored in the .git/rebase-merge/ folder)
* also, if a commit no longer makes sense due to changes in the upstream branch, git rebase --skip will skip the now obsolete commit.

**Rewriting history**

By rebasing our branch, we have avoided a merge commit. However, we now have two successive commits that seem to bring similar changes

\* 7a48ea2 add new modern data

\* 075ce36 add more modern data

Those changes probably deserve to belong to the same commit. git rebase --interactive enables to rewrite a set of commits interactively e.g.

* reword a commit message
* squash commits together
* edit commits
* remove commits
* reorder commits.

In our case we want to squash commits 075ce36 and 7a48ea2:

$ git rebase --interactive 075ce36^

will present all child commits that may be rewritten:

pick 075ce36 add more modern data

pick 7a48ea2 add new modern data

pick b588260 add french slang

*# Rebase 62cbf27..b588260 onto 62cbf27*

*#*

*# Commands:*

*# p, pick = use commit*

*# r, reword = use commit, but edit the commit message*

*# e, edit = use commit, but stop for amending*

*# s, squash = use commit, but meld into previous commit*

*# f, fixup = like "squash", but discard this commit's log message*

*# x, exec = run command (the rest of the line) using bash*

It is important to note that the commits order is reversed compared to the output of git log command; the first commits listed are the oldest one.

We just need to change the line

pick 7a48ea2 add new modern data

into (note that changing anything else that the verb at the beginning of a line will have no effect)

fixup 7a48ea2 add new modern data

and save the change

$ git rebase --interactive 075ce36^

[detached HEAD df55562] add more modern data

Date: Wed Oct 22 22:45:05 2014 +0200

1 file changed, 2 insertions(+)

Successfully rebased and updated refs/heads/modern-french.

We are done rewriting history and now have a clean commit tree:

$ git log --graph --oneline

\* cc4b70a add french slang

\* df55562 add more modern data

\* 62cbf27 Merge branch *'structure-data'*

|*\*

| \* 5a40d5a add modern french data

| \* dd0109a add *'old'* header to french data

|/

\* dd0f5d6 Add more french data

\* 13d1b4b Create french data file

\* 45de2f7 First commit

Now the modern-french branch should be renamed into master which can be done in multiple ways e.g.

$ git checkout master && git reset --hard modern-french

HEAD is now at cc4b70a add french slang

$ git branch --delete modern-french

Deleted branch modern-french (was cc4b70a).

or

$ git branch -D master

Deleted branch master (was 075ce36).

$ git branch --move modern-french master

**Rebase caveats**

Rewriting history can lead to predictable yet unexpected results.

Suppose we create a new file with some lines and a commit for each line.

$ git checkout -b rewriting-history

$ echo *"one"* > dummy && git add dummy && git commit -m *"first"*

[rewriting-history 9041c15] first

1 file changed, 1 insertion(+)

create mode 100644 dummy

$ echo *"two"* >> dummy && git add dummy && git commit -m *"second"*

[rewriting-history 50c1ff1] second

1 file changed, 1 insertion(+)

$ echo *"three"* >> dummy && git add dummy && git commit -m *"third"*

[rewriting-history 631a301] third

1 file changed, 1 insertion(+)

Now let’s say we want to swap commits

$ git rebase --interactive HEAD~3

and set

pick 9041c15 first

pick 631a301 third

pick 50c1ff1 second

This will give the following conflict diff

++<<<<<<< HEAD

++||||||| parent of 631a301... third

++two

++=======

+ two

+ three

++>>>>>>> 631a301... third

This is fully predictable: each commit stores full file snapshots however we tend to think in incremental delta, simply looking at the diff induced by changes from commit A to commit B (i.e. git diff A B). However when three-way merge is involved (be it for merge or rebase) is the diff with respect to the common ancestor (i.e. git diff A...B).

Let’s say that we resolved the conflict with the following

$ git diff

diff --cc dummy

index 5626abf,4cb29ea..0000000

--- a/dummy

+++ b/dummy

@@@ -1,1 -1,3 +1,2 @@@

one

-two

+ three

When we continue the rebase, we will again hit a conflict:

++<<<<<<< HEAD

+three

++||||||| parent of 50c1ff1... second

++=======

+ two

++>>>>>>> 50c1ff1... second

We see that we have moved the issue from the common ancestor to the MERGE\_HEAD, which in a rebase, is the parent commit and created a conflict cascade.

**Remotes**

Until now, we have been working locally. As the repository is now clean, we are now ready to publish our work to the world. git servers can be interacted with through the [git remote](https://www.kernel.org/pub/software/scm/git/docs/git-remote.html) command. By default no remote server is defined. We will use a repository declared on [GitHub](https://github.com/) as our remote called origin:

$ git remote add origin git@github.com:marchelbling/bonjour.git

$ git remote show origin

\* remote origin

Fetch URL: git@github.com:marchelbling/bonjour.git

Push URL: git@github.com:marchelbling/bonjour.git

HEAD branch: (unknown)

We see two new verbs, “Fetch” to retrieve modification *from* the remote repository and “Push” to publish our local modification *to* a remote repository. This local/remote binding is called the [refspec](http://git-scm.com/book/en/v2/Git-Internals-The-Refspec). By default, the “Fetch” URL is the same as the “Push” URL but this may be easily [configured](http://sleepycoders.blogspot.fr/2012/05/different-git-push-pullfetch-urls.html) if needed.

As we did not interact (to fetch or push) with the origin remote yet, the HEAD or upstream branch is unknown.

**Push**

git push origin local:remote will push the local branch as the remote branch onto the origin remote (where remote is equal to local by default).

git might be [configured](http://stackoverflow.com/a/948397/626278) to shorten the command line to push changes upstream. However care should be taken that this configuration depends a lot on the git version being used (and you might depend on multiple versions when working with distinct servers) so as a safe rule of thumb, always invoke git push origin local to push the local branch to the origin remote. Actually pushing changes is not the action that you perform most of the time and it will keep you out of any [embarrassing mistake](https://groups.google.com/forum/#!msg/jenkinsci-dev/-myjRIPcVwU/qOAqXGaRioIJ).

Note that if local is empty (i.e. git push origin :remote), the remote branch will be deleted from the origin remote.

We may push our master branch to our origin remote

$ git push origin master

Counting objects: 8, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (5/5), done.

Writing objects: 100% (8/8), 738 bytes | 0 bytes/s, done.

Total 8 (delta 0), reused 0 (delta 0)

To git@github.com:marchelbling/bonjour.git

\* [new branch] master -> master

and everyone with an access to the remote can now see our work.

**Fetch**

Until now, we have been working on our own on the repository. As we created a public repository, some changes may have been pushed to our remote. git does not automatically try to get modification from the remote so it is the user responsibility to make sure his repository is up to date. The command to retrieve remote changes is git fetch:

$ git fetch origin

remote: Counting objects: 4, done.

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

remote: Total 4 (delta 0), reused 0 (delta 0)

Unpacking objects: 100% (4/4), done.

From github.com:marchelbling/bonjour

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

We can see that a new branch english has been pushed. We see that locally, git refers to it as origin/english. Indeed, git keep remote object references in an eponym namespace

$ tree .git/refs/ --matchdirs -P remotes/origin

.git/refs/

├── heads

├── remotes

│   └── origin

│   ├── english

│   └── master

└── tags

It is important to understand that fetching a remote will *not* modify the working tree. Also, as a consequence of the remote namespacing, to reference a remote branch we should prefix it with the remote name

$ git checkout -b english origin/english

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

Switched to a new branch *'english'*

**Pull**

Often times, we fetch a remote to check for update on our working branch meaning that we want to update both git objects and our working tree. This means that we would actually like to fetch changes and apply them. This is what git pull does:

1. it fetches changes from the remote defined by the local branch refspec
2. it updates our working tree
   * by merging our local branch with the remote version
   * or by rebasing local branch on the remote branch when invoking git pull --rebase.

**Force push**

Let’s say we would like to fix the commit message from the branch we fetched

$ git commit --amend -m *"add english/american data"*

[english ea761bc] add english/american data

Author: Linux Tor <tor@linux.org>

Date: Sun Nov 2 20:19:35 2014 +0100

1 file changed, 20 insertions(+)

create mode 100644 en/data

$ git push origin english

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

error: failed to push some refs to *'git@github.com:marchelbling/bonjour.git'*

hint: Updates were rejected because the tip of your current branch is behind

hint: its remote counterpart. Integrate the remote changes (e.g.

hint: *'git pull ...'*) before pushing again.

hint: See the *'Note about fast-forwards'* in *'git push --help'* for details.

By amending, we have just replaced the branch tip commit and whenever a user tries to push a local branch to an existing remote branch, git will check that the remote branch HEAD is still reachable in the branch being pushed and will fail if it is unreachable. This may happen in 2 cases:

* local branch is out-of-sync due to changes pushed to the remote; the reflex should be immediately to [fetch](https://help.github.com/articles/dealing-with-non-fast-forward-errors/) the remote and update the branch
* history has been rewritten and the sha1 is no longer reachable; we need to force push our local branch to rewrite the remote history:

$ git push --force origin english

Counting objects: 1, done.

Writing objects: 100% (1/1), 192 bytes | 0 bytes/s, done.

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

To git@github.com:marchelbling/bonjour.git

+ 0ffd492...ea761bc english -> english (forced update)

We here see that pushing a rewritten history (git commit --amend or git rebase) should be done with care and we will discuss workflows in the next section.

**Branches and remotes: tl;dr**

* branches are simply references to commits
* HEAD references the tip commit for current branch
* git merge (and most “merging-like” commands) relies on three-way merge by default
* git rebase enables to rewrite history
  + git rebase foo will move current branch commits onto foo branch (note that in case of conflict ours and theirs can feel inverted as git rebase foo checkouts the foo branch under the hood)
  + git rebase --interactive sha1^ will allow to edit/squash/remove/reorder all commits from sha1 to HEAD
  + rebasing rewrites history and creates new commit objects
* comparing branches for merge/rebase should be done with git diff A...B to take into account the common ancestor
* retrieving remote updates is the user responsibility
* merge/rebase commands should usually reference branches from remote namespace
* **never** force push without
  + having checked if changes were pushed to the remote
  + fully specifying the remote/branch (to avoid force pushing wrong branches)

**Good practices**

**Workflow**

**Branching flow**

Creating branches with git is very cheap and should therefore be used without fear. The question is then: how should the branches relate to one another?

There is no single answer to that question. Mostly because the answer depends on type nature of the project: a [web app](http://nvie.com/posts/a-successful-git-branching-model/) runs a single ever up-to-date version when a [desktop app](https://www.kernel.org/pub/software/scm/git/docs/gitworkflows.html) may have multiple versions supported at a given time. The former may however be seen as a simple particular case of the latter.

*Don’t merge upstream code at random points.  
Don’t merge downstream code at random points either.*

[Linus Torvald](http://thread.gmane.org/gmane.comp.video.dri.devel/34739/focus=34744)

Most workflows maintain different contexts with careful synchronization points and the typical synchronization for a web app will look like:

* the master branch should always be stable and reflect the code running in production
* the develop branch hosts developments between 2 releases
* new features should branch off/get integrated in the develop branch
* hotfixes should branch off master and be integrated in both master and develop branches.