**Git Essentilas second Edition**

This is because in Git, every modification you make in a repository has to be signed with the name and email of the author. So, before doing anything else, we have to tell Git this information.

A **repository** is a container for your entire project; every file or subfolder within it belongs to that repository, in a consistent manner. Physically, a repository is nothing other than a folder that contains a special .git folder, the folder where the magic happens

You can configure Git to use your own preferred editor, but if you don't do it, this is what you have to deal with. Vim is powerful, but for newcomers, it can be a pain to use. It has a strange way of dealing with text. To start typing, you have to press *I* for inserting text, as shown in the following

Once you have typed your commit message, you can press *Esc* to get out of editing mode. Then, you can type the :w command to write changes and the :q command to quit. You can also type the command in pairs as :wq,

### Digging into Git internals

The result of the git init command is the creation of a .git folder, where Git stores all the files it needs to manage our repository

**$ git add .**

With this trick (the dot after the git add command), you can add all the new or modified files in one shot.

So, we can move this grocery folder wherever we want, and no data will be lost. Another important thing to highlight is that we don't need any server: we can create a repository locally and work with it whenever we want, even with no LAN or internet Connection

So, I prefer setting up usernames and emails per repository; in Git, you can set up your config variables at three levels: *repository* (with the --local option, the default one), *user* (with the --global option), and *system-wide* (with the --system option).

**$ git config user.name "Ferdinando Santacroce"**

**$ git config user.email** [**ferdinando.santacroce@gmail.com**](mailto:ferdinando.santacroce@gmail.com)

**$ git log --format=fuller**

**Other than the author, a commit preserves even the committer, and the committing date;**

**99% of commits in your repository will have the same values for the author and committer, and the same dates.** **In some situations, such as the cherry-pick, you carry an existing commit on top of another branch, making a brand-new commit that applies the same changes of the**

**previous. In this case, the author and author date will remain the same, while the**

**committer and the committing date will be related to the person who performed this**

**operation and the date they did it.**

**Using the git log command again, we can enable x-ray vision using the --format=raw option:**

**$ git log --format=raw**

**$ git cat-file -p a57d7**

**the output is the same of git log --format=raw**

Git, as we know, has a myriad of commands, some of which are practically never used by the average user; as by example, the previous git cat-file. These commands are called plumbing commands, while those we have already learned about, such as

git add, git commit, and so on, are among the so-called porcelain commands.

The first line contains the commit's **SHA-1** (https://en.wikipedia.org/wiki/SHA-1), an alphanumeric sequence of 40 characters representing a hexadecimal number. This *code*, or **hash**, as it is usually called, uniquely identifies the commit within the repository

Git uses four different types of **objects**, and *commit* is one of these. Then there are *tree*, *blob*, and *annotated tag*.

**[15] ~/grocery (master)**

**$ git cat-file -p a57d7**

**tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40**

this plumbing command lets you peek into the Git objects; with the -p option (which means *pretty-print* here), we ask Git to show an easier way to read what the contents of the object are.

* The **tree** is a **container** for blobs and other trees.
* Git blobs represent the files. Blobs are binary files, nothing more.

**$ git cat-file -p 637a0**

**banana**

Wow! Its content is exactly the content of our shoppingFile.txt file

Any file is compressed and transformed into a blob before archiving it into a Git repository. Each file is marked with a *hash*; this hash uniquely identifies the file within our repository, and it is thanks to this ID that Git can then retrieve it when needed, and detect any changes when the same file is altered (files with different content will have different hashes).

SHA-1 hashes are unique

**$ echo "banana" | git hash-object --stdin**

**637a09b86af61897fb72f26bfb874f2ae726db82**

The git hash-object command is the plumbing command to calculate the hash of any object; in this example, we used the --stdin option to pass as a command argument the result of the preceding command, echo "banana"; in a few words, we calculated the hash of the string "banana",

**an object**, whatever it is, **will always have the same hash in any repository**, in any computer on the face of the Earth.

**Git calculates the hash on the content of the file, not in the file itself.** This teaches us an important lesson: if you have two different files with the same content, even if they have different names and paths, in Git you will end up having only one blob.

**the Git storage object model**

**$ ls –al**

**$ ls -al .git/**

**$ ls -al .git/objects**

**$ ls -al .git/objects/63**

Git is amazingly smart and simple: to be quicker while searching through the filesystem, Git creates a set of folders where the name is two characters long, and those two characters represent the first two characters of a hash code; inside those folders, Git writes all the objects using as a name the other 38 characters of the hash, regardless of the kind of Git object.

Git compresses them using the zlib library to reserve space on your disk. This is why we use the git cat-file –p command, which decompresses them on the fly for us

This highlights once again the simplicity of Git: no metadata, no internal databases, or useless complexity, but simple files and folders are enough to make it possible to manage any repository.

So, every commit has a parent, and following these relations between commits, we can always navigate from a random one down to the first one, the already mentioned **root commit**

Now it's time to investigate another well-known difference between Git and other versioning systems. Take Subversion as an example: when you do a new commit, Subversion creates a new numbered revision that only contains deltas between the previous one; this is a smart way to archive changes to files, especially among big text files, because if only a line of text changes, the size of the new commit will be much smaller.

Instead, in Git even if you change only a char in a big text file, it always stores a new version of the file: **Git doesn't do deltas** (at least not in this case), and **every commit is actually a snapshot of the entire repository**.

Git repository can be imagined as a tree that, starting from a root (the root-commit), grows upward through one or more branches. These branches are generally distinguished by a name. *Master* is precisely the name of the *default branch* of a Git repository, somewhat like trunk is for Subversion

In Git, **a branch is nothing more than a label**, a *mobile label* placed on a commit

In fact, every leaf on a Git branch has to be labeled with a meaningful name to allow us to reach it and then move around, go back, merge, rebase, or discard some commits when needed.

I made a commit without first making git add; the *trick* is in the -a (--add) option added to the git commit command, which means *add to this commit all the modified files that I have already committed at least one time before*. In our case, this option allowed us to go faster and skip the git add command.

## Branches are movable labels

the commits are linked to each other by a parent-and-son relationship: each commit contains a reference to the previous commit.

So, branches are nothing but labels that are on the tip commit, the last one. This commit, our leaf, must always be identified by a label

Every time we make a commit to a branch, the **reference** that identifies that branch will move accordingly to always stay associated with the tip commit

**$ ls -al .git/**

**$ ls -al .git/refs**

**$ ls -al .git/refs/heads**

**$ cat .git/refs/heads/master**

**0e8b5cf1c1b44110dd36dea5ce0ae29ce22ad4b8**

Git manages all this articulated reference system... with a trivial text file! It contains the hash of the last commit made on the branch

As branches are, HEAD is a **reference**. It represents a pointer to the place on where we are right now, nothing more, nothing less. In practice instead, it is just another plain text file:

**$ cat .git/HEAD**

**ref: refs/heads/berries**

The difference between the HEAD file and branches text file is that the HEAD file usually refers to a branch, and not directly to a commit as branches do. The ref: part is the convention Git uses internally to declare a pointer to another branch, while refs/heads/berries is of course the relative path to the berries branch text file

*In Subversion, we usually have different folders for each different branch.* When you switch a branch, Git goes to the commit the branch is pointing to, and following the parent relationship and analyzing trees and blobs, rebuilds the content on the **working directory** accordingly, getting hold of that files and folders

**$ git checkout -**

**Switched to branch 'berries**

New trick: using the dash (-), you actually are saying to Git: "*Move me to the branch I was before switching*"; and Git obeys, moving us to the berries branch

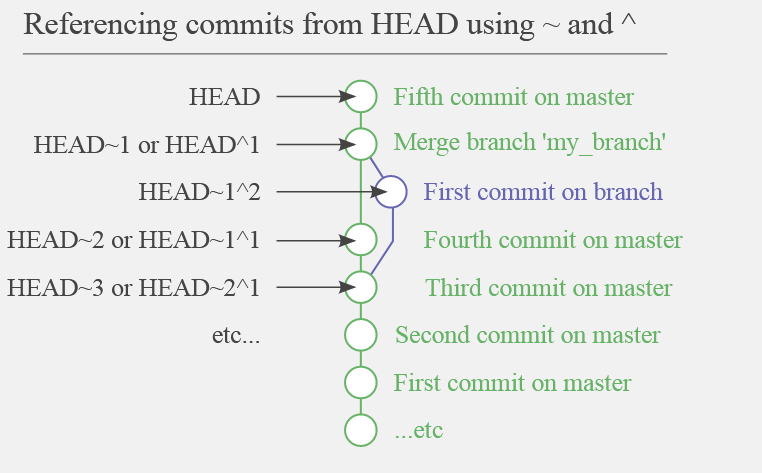
## Reachability and undoing commits

## $ git reset --hard master HEAD~1

In Git, you often have the need to point to a preceding commit, like in this case, the one before; for this scope, we can use HEAD reference, followed by one of two different special characters, the *tilde*~ and the *caret*^. A **caret** basically means *a back step*, while two carets means two steps back, and so on. As you probably don't want to type dozens of carets, when you need to step back a lot, you can use **tilde**: similarly, ~1 means *a back step*, while ~25 means 25 steps back, and so on

it simply won't delete unreachable commits, at least not immediately. It makes some housekeeping automatically at a given time, as it has some powerful **garbage collection** features

## [Git caret and tilde](http://www.paulboxley.com/blog/2011/06/git-caret-and-tilde)



In Git, you often have the need to point to a preceding commit, like in this case, the one before; for this scope, we can use HEAD reference, followed by one of two different special characters, the *tilde*~ and the *caret*

**$ git reset --hard HEAD^**

## Detached HEAD

Git is very kind and often tells us loads of useful information in its output messages. Don't under evaluate this behavior: especially at the beginning, reading Git messages allows you to learn a lot, so read them carefully

Git says we are in a detached HEAD state. Being in this state basically means that HEAD does not reference a branch, but directly a commit

## The reflogs

Git never forgets you. It has another powerful tool in its wrench box, and that is called the **reference log**, or reflog for short. Basically, the reflog (or better the reflogs, as there is one for every reference) records what happens in the repository while you commit, reset, check out, and so on. To be more precise, every reflog records all the times that tips of the branches and other references (such as HEAD) where updated

**Tags are fixed labels**

**Tags** are labels you can pin to a commit, but unlike branches, they will stay there.

**$ git tag nickname**

Tags are useful to give a particular meaning to some particular commits

Even tags are references, and they are stored, as branches, as simple text files in the tags subfolder within the .git folder; take a look under the .git/refs/tags folder, you will see a bugTag file; look at the content:

**$ cat .git/refs/tags/nickname**

To delete a tag, you have to simply append the -d option: git tag -d <tag name>.

you can create a tag that points a commit wherever you want, appending the hash of the commit as an argument, for example, git tag myTag 07b1858.

**Annotated tags**

Git has two kinds of tags; this is because in some situations you may want to add a message to the tag. Tags containing this extra information load belong to the second type, the **annotated tag**

**$ git tag -a annotatedTag 07b1858**

An annotated tag is both a *reference* and a *git object* such as commits, trees, and Blobs

**$ cat .git/refs/tags/annotatedTag**

**17c289ddf23798de6eee8fe6c2e908cf0c3a6747**

**$ git cat-file -p 17c289**

## Staging area, working tree, and HEAD commit

A common thing is to use the dot . as a wildcard, and this by default means, *add all the files in this folder and subfolders to the staging area*. This is the same as git add -A (or --all), and by "all" I mean:

* **Files in this folder and sub-folders I added in the past at least one time**:This set of files is also known as the **tracked files**
* **New files**: These are called **untracked files**
* Files marked for deletion

git add –A Stage all (new, modified, deleted) files

git add . Stage all (new, modified, deleted) files

git add --ignore-removal Stage new and modified files only

git add –u Stage modified and deleted files only

the -u option is the equivalent of --update.

The effect of git commit is to create a new commit with the content of the staging area, and then empty it.

To see the difference between the working tree version and the staging area one, try to input only the git diff command without any option or argument:

**$ git diff**

To see the differences between the last committed version of the

shoppingList.txt file and the one added into the staging area use the git diff --cached HEAD command

**$ git diff --cached HEAD**

We have to dissect this command to better understand what's the purpose; appending the HEAD argument, we are asking to use the last commit we did as a subject of the compare. To be true, in this case, the HEAD reference is optional, as it is the default:

git diff --cached would return the same result. Instead, the --cached option says, *compare the argument (HEAD in this case) with the* *version in the staging area*. Yes, dear friends: the staging area, also known as an index, sometimes is called cache, hence the --cached option.

**$ git diff --cached HEAD**

The last experiment that we can do is compare the HEAD version with the working tree one; let's do it with a git diff HEAD

**$ git diff HEAD**

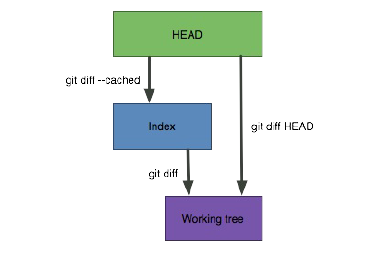
In Git, we work at three different levels:

* The **working tree** (or working directory)
* The **staging area** (or index, or cache)
* The **HEAD commit** (or the last commit or tip commit on the current branch)

Removing changes from the staging area.

To remove those changes to one or more files from the staging area, you can use the git reset HEAD <file> command

git reset HEAD won’t destroy your modification.it only moves away them from the staging area.



be careful: git checkout -- is a destructive command, use it carefully.

Other than this, we need to remember that git checkout overwrites even the staging area; as per the preceding figure, working tree and HEAD commit are in direct relationship: changes always go through the staging area. Later we will grasp this concept better while delving into git reset options

*git checkout is for branch switching* (or for commit inspection, going to a detaching HEAD state), as it can be used even to discard changes in a working tree,

This is true even for the git reset command; in fact, do a git reset -- <file> is

actually the same as doing a git reset HEAD <file>.

To be true, the double-dash -- notation is not mandatory; if you do a git checkout <file> or git reset <file> without --, in 99% of cases Git does what you expect. The double-dash is needed when, due to a coincidence, there is a file and a branch with the same name: in that case, Git has to know if you want to deal with branches, for example switching to another one with git checkout, or if you want to manage files. In this situation, the double-dash is the way to tell Git *I want to handle files, not branches*.

The following figure summarizes the commands to move changes between those three areas:



so, use the capital -D option to force the deletion

**$ git branch -D bug**

This mixed-reset technique can be useful, for example, to clean up all the staged changes in one shot, with a simple git reset HEAD.

This hard-reset technique is used to completely discard all the changes we did, with a git reset --hard HEAD command, as we did in our previous experiments

With the git rebase command you can also modify the story of branches; one of the things you do more often inside a repository is to change - or better to say - move the point where a branch started, bringing it to another point of the tree. This operation makes it possible to keep low the level of ramifications that would instead be generated using the command git merge, which we will see later

This is called **fast-forwarding**.

This time there's no merge commit, as it is not necessary; someone will argue that in this manner you lose the information that tells you when two branches have been merged. If you want to force Git always create a new merge commit, you can use the --no-ff (no fast-forward) option.

**Cherry picking**

If you want to track what was the commit you cherry-picked, you can append the –x option to the git cherry-pick command; then, while committing, don't append the message in the git commit command using the -m option, but type git commit and then press ENTER to allow Git to open the editor: it will suggest you a message that contains

the hash of the cherry-picked commit, as you can see in the following screenshot

**Clone a local repository**

Create a new folder on your disk to clone our grocery repository:

**$ mkdir grocery-cloned**

Then clone the grocery repository using the git clone command

**$ cd grocery-cloned**

**$ git clone ~/grocery .**

The dot . argument at the end of the command means *clone the repository in the current folder*, while the ~/grocery argument is actually the path where Git has to look for the repository

What is the *origin*?

Git uses origin as the default name of a remote. Like with master for branches, origin is just a convention: you can call remotes whatever you want.

The command to deal with remotes is git remote; you can add, remove, rename, list, and do a lot of other things with them; there's no room here to see all the options.

**Pushing a new branch to the remote**

**$ git push -u origin Risotti**

With the git push -u origin Risotti command, we told Git to upload our Risotti

branch (and the commits within it) to the origin; with the -u option, we set the local branches to track the remote one

Using the -u option, we told Git to **track** the remote branch. Tracking a remote branch is *the way to tie your local branch with the remote one*; please note that this behavior is not automatic, you have to set it if you want it. When a local branch tracks a remote branch, you actually have a local and a remote branch that can be kept easily in sync (please note that a local branch can track only one remote

branch). This is very useful when you need to collaborate with some remote coworkers at the same branch, allowing all of them to keep their work in sync with other people's changes.

**Adding a remote to a local Repository**

**$ git remote add origin https://github.com/fsantacroce/HelloWorld.git**

In Git, we have *three configuration levels*:

System

Global (user-wide)

Repository



The system level contains **system-wide configurations**; if you edit the configuration at this level, *every user and every user's repository will be affected*.This configuration is stored in the gitconfig file usually located in: C:\Program Files\Git\etc\gitconfig

To edit parameters at this level, you have to use the --system option

The global level contains **user-wide configurations**; if you edit the configuration at this level, *every user's repository will be affected*. This configuration is stored in the .gitconfig file usually located in:

C:\Users\<UserName>\.gitconfig

To edit parameters at this level, you have to use the --global option.

The repository level contains **repository only configurations**; if you edit the configuration at this level, *only the repository in use will be affected*.This configuration is stored in the config file located in the .git repository subfolder:

C:\<MyRepoFolder>\.git\config

To edit parameters at this level, you can use the --local option or simply avoid using any option, as this is the default one.

To get a list of all the configurations currently in use, you can run the git config --list command; if you are inside a repository, it will show all the configurations, from repository to system level. To filter the list, append optionally --system, --global or --local options to obtain only the desired level configurations:

Even if it is generally discouraged, you can modify Git configurations directly by editing the files. Git configuration files are quite easy to understand

In previous chapters, we have seen that we can change a Git variable value using the git config with the <variable.name> <value> syntax.

**Push default**

We already talked about the git push command and its default behavior. To avoid annoying issues, it is good practice to set a more convenient default behavior for this command.

There are two ways we can do this. First one: set Git to ask us the name of the branch we want to push every time, so a simple git push will have no effect. To obtain this, set push.default to nothing

**$ git config --global push.default nothing**

Another way to save yourself from this kind of mistake is to set the push.default parameter to simple, allowing Git to push only when there is a remote branch with the same name as the local one:

**$ git config --global push.default simple**

**Defining the default editor**

Some people really don't like vim, even only for writing commit messages; if you are one of them, there is good news: you can change it by setting the core.default config parameter:

**$ git config --global core.editor notepad**

The classic example is the git unstage alias:

**$ git config --global alias.unstage 'reset HEAD --'**

With this alias, you can remove a file from the index in a more meaningful way, compared to the equivalent git reset HEAD -- <file> syntax

**$ git unstage myFile.txt**

Want a fast way to revert the last ongoing commit? Create a git undo alias:

**$ git config --global alias.undo 'reset --soft HEAD~1'**

You can obviously use --hard instead of --soft, or go with the default --mixed option

A git last alias is useful to read about your last commit:

**$ git config --global alias.last 'log -1 HEAD'**

With the git difflast alias, you can see a diff against your last commit:

**$ git config --global alias.difflast 'diff --cached HEAD^'**

If you want the alias to run external shell commands, instead of a Git sub-command, you have to prefix the alias with a !:

**$ git config --global alias.echo !echo**

Suppose you are annoyed by the canonical git add <file> plus git commit <file> sequence of commands, and you want to do it in a single shot; you can call the git command twice in sequence by creating this alias:

**$ git config --global alias.cm '!git add -A && git commit -m'**

<https://github.com/GitAlias/gitalias>

**Removing an alias**

**$ git config --global --unset alias.cm**

How does git stash work? Actually, git stash is a fairly complex command. It basically saves from two up to three different commits

* A new *WIP commit* containing the actual state of the working copy; it contains all the tracked files, and their modifications.
* An *index commit*, as a parent of the WIP commit. This contains stuff added to the staging area.
* An optional third commit, let's call it an *untracked files commit*, which contains untracked files (using the --include-untracked option) or untracked plus previously ignored files (using the --all option).

The *WIP commit* contains the unstaged changes made to tracked files; as you can see, the *WIP commit* has two parents: one is the *index commit*, containing staged changes, the other is the last commit on the master branch, where HEAD was and where we run the git stash command.

To retrieve a stash, the command is git stash apply <stash>; it applies changes within the two commits, eventually modifying your working copy and staging area. The stash will not be deleted after the apply; you can do it manually using the git stash drop <stash> subcommand. Another way implies the git stash pop <stash> subcommand: it applies the stash and then deletes it.

create a new stash using -u (the --include-untracked option).

Summarizing, you basically use the git stash save command (with the -u or –all option if needed) to shelve your modification and then git stash apply to retrieve them; I suggest using git stash apply and then git stash drop instead of git pop to have a chance to redo your stash application when needed, or when your stash is not as trivial as usual

Working on source code in a team, it is not uncommon to have the need to look at the last modifications made to a particular file to better understand how it evolved over time. To achieve this result, we can use the git blame <filename> command.

The last tip I want to suggest is to use the Git GUI:

**[3] ~/Spoon-Knife (master)**

**$ git gui blame README.md**

*your repository isn’t only a dump*,

**Don't do partial commits**

Remember that *Git is not a backup tool*: backup your stash on another disk

The following tips can help you turn your VCS from a backup system into *a valuable tool for communication and documentation*.

**Bare repositories**

Bare repositories are repositories that do not contain working copy files, but only the .git folder. A bare repository is essentially *for sharing*: if you use Git in a centralized way, pushing and pulling to a common remote (a local server, a GitHub repository, and so on), you will agree that the remote has no interest in checking out files you work on; the scope of that remote is only to be a central point of contact for the team, so having working copy files in it is only a waste of space as no one will edit them directly on the remote.

If you want to set up a bare repository, you only have to use the --bare option:

**$ git init --bare NewRepository.git**

using a .git extension; this is not mandatory, but is a common way to identify bare repositories.

You can easily convert a regular repository to a bare one using the git clone command with the same --bare option:

**$ git clone --bare my\_project my\_project.git**

**Archiving the repository**

To archive the repository without including versioning information, you can use the

git archive command; there are many output formats but the classic one is the .zip one:

**$ git archive master --format=zip --output=../repoBackup.zip**

Please note that using this command is not the same as backing up folders in a filesystem; as you will have noticed, the git archive command can produce archives in a smarter way, including only files in a branch or even in a single commit; for example, by doing this you are archiving only the last commit:

**$ git archive HEAD --format=zip --output=../headBackup.zip**

Archiving files in this way can be useful if you have to share your code with people that don't have Git installed.

Another interesting command is the git bundle command. With git bundle, you can export a snapshot from your repository and then restore it wherever you want. Suppose you want to clone your repository on another computer, and the network is down or absent; with this command, you can create a repo.bundle file of the master

branch:

**$ git bundle create ../repo.bundle master**

With this other command, we can restore the bundle in the other computer using the git clone command:

**$ cd /OtherComputer/Folder**

**$ git clone repo.bundle repo -b master**

*your repository isn’t only a dump*,

**Don't do partial commits**

Remember that *Git is not a backup tool*: backup your stash on another disk

The following tips can help you turn your VCS from a backup system into *a valuable tool for communication and documentation*.

Resources:

Cmder is not really a Git GUI,

Linux server, Great web interfaces like *Gitosis*, *Gitlab,* and so on.

**SCM Manager** (https://www.scm-manager.org/) is a very easy solution to share your Git repositories in a local Windows network;

Resources

<https://onlywei.github.io/explain-git-with-d3/#fetchrebase>

<https://learngitbranching.js.org/>

<http://ndpsoftware.com/git-cheatsheet.html#loc=remote_repo>;

<http://www.ndpsoftware.com/>

Take a look at www.tfnico.com and www.gitminutes.com.

<http://news.humancoders.com/t/git/page/8>

<https://kolosek.com/git-reset-revert-and-checkout/?utm_source=gg>

https://plus.google.com/communities/112688280189071733518

Git Essentials 1 er version

With git difflast alias, you can indeed see a difference from your last commit, as shown here:

**$ git config --global alias.difflast 'diff --cached HEAD^'**

**Advanced aliases with external commands**

If you want the alias to run external shell commands instead of a Git subcommand, you have to prefix the alias with a !:

**$ git config --global alias.echo !echo**

Suppose you are annoyed by the canonical git add <file> plus git commit <file> sequence of commands, and you want to do it in a single shot; here you can call the git command twice in sequence creating this alias:

**$ git config --global alias.cm '!git add -A && git commit -m'**

There are also aliases that define and use complex functions or scripts, but I'll leave it to the curiosity of the reader to explore these aliases. If you are looking for inspiration, please take a look at mine at https://github.com/jesuswasrasta/GitEnvironment.

**Removing an alias**

Removing an alias is quite easy; you have to use the --unset option, specifying the alias to remove. For example, if you want to remove the cm alias, you have to run:

**$ git config --global --unset alias.cm**

Note that you have to specify the configuration level with the appropriate option; in this case, we are removing the alias from the user (--global) level.

**Aliasing the git command itself**

I already said I'm a bad typewriter; if you are too, you can alias the git command

itself (using the default alias command in Bash):

**$ alias gti='git'**

In this manner, you will save some other keyboard strokes. Note that this is not a Git

alias but a Bash shell alias.

**Git references**

Git manages these references as files in the .git/refs repository folder:

$ find .git/refs

**Cherry picking**

We can even pick commit sets if we want to by using the <startingcommit>..<ending-commit> syntax:

**$ git cherry-pick feat1~2..feat1~0**

With this syntax, you are basically picking the last two commits from the feat1 branch.

**Bare repositories**

Bare repositories are repositories that do not contain working copy files but contain

only the .git folder. A bare repository is essentially for sharing; if you use Git in a

centralized way, pushing and pulling to a common remote (a local server, a GitHub

repository, or so on), you will agree that the remote has no interest in checking out

files you work on; the scope of that remote is only to be a central point of contact for

the team, so having working copy files in it is a waste of space, and no one will edit

them directly on the remote.

If you want to set up a bare repository, you have to use only the --bare option:

**$ git init --bare NewRepository.git**

As you may have noticed, I called it NewRepository.git, using a .git extension;this is not mandatory but is a common way to identify bare repositories. If you pay attention, you will note that even in GitHub every repository ends with the .git extension.

**Converting a regular repository to a bare one**

You can easily convert a regular repository to a bare one using the git clone

command with the same --bare option:

**$ git clone --bare my\_project my\_project.git**

**Archiving the repository**

To archive the repository without including the versioning information, you can use the git archive command; there are many output formats of which ZIP is the classic one:

**$ git archive master --format=zip --output=../repbck.zip**

**$ git archive HEAD --format=zip --output=../headbck.zip**

**Cmder**

Cmder is not really a Git GUI, but a nicer portable console emulator you can use

instead of the classic Bash shell: