This next section presents some of the basic commands that you will need to know in order to use Git effectively.

# CHAPTER 1 Understanding Git

## What Is Git?

Git is a tool for tracking changes made to a set of files over time, a task traditionally known as “version control.” Although it is most often used by programmers to coordinate changes to software source code, and it is especially good at that, you can use Git to track any kind of content at all. Any body of related files evolving over time, which we’ll call a “project,” is a candidate for using Git. With Git, you can:

* Examine the state of your project at earlier points in time
* Show the differences among various states of the project
* Split the project development into multiple independent lines, called “branches,” which can evolve separately
* Periodically recombine branches in a process called “merging,”

reconciling the changes made in two or more branches

* Allow many people to work on a project simultaneously,

sharing and combining their work as needed

Git is the technology behind the enormously popular “social coding” website GitHub, which includes many wellknown open source projects.

# Internals

**Chapter 30: Internals**

**Section 30.1: Repo**

A **git** repository is an on-disk data structure which stores metadata for a set of files and directories.

It lives in your project's .git**/** folder. Every time you commit data to git, it gets stored here.

Inversely, .git**/** contains every single commit.

It's basic structure is like this:

.git**/**

objects**/**

refs**/**

**Section 30.2: Objects**

**git** is fundamentally a key-value store. When you add data to **git**, it builds an object and uses the SHA-1 hash of the object's contents as a key.

Therefore, any content in **git** can be looked up by it's hash:

**git cat-file** -p 4bb6f98

There are 4 types of Object:

* Blob
* **Tree**
* Commit
* Tag

HEAD is a special ref. It always points to the current object. You can see where it's currently pointing by checking the .git**/**HEAD file. Normally, HEAD points to another ref:

$cat .git**/**HEAD

ref: refs**/**heads**/**mainline

This is what's known as a "detached head" - because HEAD is not attached to (pointing at) any ref, but rather points directly to an object.

A ref is essentially a pointer. It's a name that points to an object. For example,

"master" --**>** 1a410e...

They are stored in `.git/refs/heads/ in plain text files.

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

4bb6f98a223abc9345a0cef9200562333

Now, it's possible to navigate **git** purely by jumping around to different objects directly by their hashes. But this would be terribly inconvenient. A ref gives you a convenient name to refer to objects by. It's much easier to ask **git** to go to a specific place by name rather than by hash.

A commit is probably the object type most familiar to **git** users, as it's what they are used to creating with the **git commit** commands.

However, the commit does not directly contain any changed files or data. Rather, it contains mostly metadata and pointers to other objects which contain the actual contents of the commit.

A commit contains a few things:

* hash of a **tree**
* hash of a parent commit
* author name/email, commiter name/email
* commit message

You can see the contents of any commit like this:

$ **git cat-file** commit 5bac93

**Tree**

A very important note is that the **tree** objects stores EVERY file in your project, and it stores whole files not diffs. This means that each commit contains a snapshot of the entire project\*.

\**Technically, only changed files are stored. But this is more an implementation detail for efficiency. From a design perspective, a commit should be considered as containing a complete copy of the project*.

**Parent**

The parent line contains a hash of another commit object, and can be thought of as a "parent pointer" that points to the "previous commit". This implicitly forms a graph of commits known as the **commit graph**. Specifically, it's a directed acyclic graph (or DAG).

A **tree** basically represents a folder in a traditional filesystem: nested containers for files or other folders.

A **tree** contains:

* 0 or more blob objects
* 0 or more **tree** objects

A blob contains arbitrary binary file contents. Commonly, it will be raw text such as source code or a blog article. But it could just as easily be the bytes of a PNG file or anything else.

If you have the hash of a blob, you can look at it's contents.

The **git commit** command does a few things:

1. Create blobs and trees to represent your project directory - stored in .git**/**objects

2. Creates a new commit object with your author information, commit message, and the root **tree** from step 1 - also stored in .git**/**objects

3. Updates the HEAD ref in .git**/**HEAD to the hash of the newly-created commit

This results in a new snapshot of your project being added to **git** that is connected to the previous state.

Git uses objects to track changes throughout the history of a repository. To achieve this tracking, Git uses four types of objects. The objects are **commits**, **trees**, **blobs,** and **tags**. These objects are stored in **.git/objects**.

>ls -al

A commit object stores the hash of the directory tree object that the commit corresponds to, the parent commit hash, the author, the committer date and time, and the commit message:

**git cat-file -t 11b8b15**

**git cat-file -p 11b8b15**

**BLOBs**

Git uses blobs to store the contents of a file at a given point in time. A blob is a **Binary Large OBject (BLOB)**. It's Git's methodology of storing the contents of a file at a given point in its lifetime. A blob is created when we commence the tracking of a file by using the **git add** command:

# Configuration

1.4 Settings

You need to **set who** you are **\***before**\*** creating any commit

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

This will store the setting in your user's .gitconfig file: **%**USERPROFILE**%**\.gitconfig for Windows

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

**git config** --global user.email mail**@**example.com

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

This will store the setting inside the individual repository, in the file **/**path**/**to**/**your**/**repo**/**.git**/**config

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

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

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

**Chapter 13: Configuration**

Change the core.editor configuration setting.

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

**Section 13.2: Auto correct typos**

**git config** --global help.autocorrect 17

This enables autocorrect in git and will forgive you for your minor mistakes (e.g. **git** stats instead of **git status**).

The parameter you supply to help.autocorrect determines how long the system should wait, in tenths of a second, before automatically applying the autocorrected command. In the command above, 17 means that git should wait 1.7 seconds before applying the autocorrected command.

Git config allows you to customize how git works. It is commonly used to set your name and email or favorite editor or how merges should be done.

To see the current configuration.

$ **git config** --list

...

core.editor=**vim**

credential.helper=osxkeychain

...

To edit the config:

$ **git config <**key**> <**value**>**

$ **git config** core.ignorecase **true**

If you intend the change to be true for all your repositories, use --global

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

$ **git config** --global user.email "Your Email"

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

You can list again to see your changes.

**Section 13.4: Username and email address**

Right after you install Git, the first thing you should do is set your username and email address. From a shell, type:

**git config** --global user.name "Mr. Bean"

**git config** --global user.email mrbean**@**example.com

You have 3 sources for git configuration:

* (system) **<git>/etc/gitconfig**, with **<git>** being the git installation path.(on Windows, it is **<git>\mingw64\etc\gitconfig**)
* (global) ~**/**.gitconfig (Windows: **%**USERPROFILE**%**\.gitconfig)
* (local) .git**/**config (within a git repo $GIT\_DIR)

The order is important: any config set in one source can be overridden by a source listed below it. **git config** --system**/**global**/local** is the command to list 3 of those sources, but only git config -l would list *all resolved* configs.

Since git 2.8, if you want to see which config comes from which file, you type:

**git config** --list --show-origin

**Section 13.8: configuration for one command only**

you can use -c **<**name**>**=**<**value**>** to add a configuration only for one command.

To commit as an other user without having to change your settings in .gitconfig :

**git** -c user.email = mail**@**example commit -m "some message"

Note: for that example you don't need to precise both user.name and user.email, git will complete the missing information from the previous commits.

**Chapter 5: Ignoring Files and Folders**

This topic illustrates how to avoid adding unwanted files (or file changes) in a Git repo. There are several ways

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

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

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

1. temporary resources e.g. caches, log files, compiled code, etc.

2. local configuration files that should not be shared with other developers

3. files containing secret information, such as login passwords, keys and credentials

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

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

1. tracked by Git

2. reported by commands such as **git status** or **git diff**

3. staged with commands such as **git add** -A

*# Ignore files called 'file.ext'*

file.ext

*# Ignoring files with full path.*

*# This matches files in the root directory and subdirectories too.*

*# i.e. otherfile.ext will be ignored anywhere on the tree.*

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

otherfile.ext

*# Ignoring directories*

*# Both the directory itself and its contents will be ignored.*

bin**/**

gen**/**

*# Glob pattern can also be used here to ignore paths with certain characters.*

*# For example, the below rule will match both build/ and Build/*

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

*# Without the trailing slash, the rule will match a file and/or*

*# a directory, so the following would ignore both a file named `gen`*

*# and a directory named `gen`, as well as any contents of that directory*

bin

gen

*# Ignoring files by extension*

*# All files with these extensions will be ignored in this directory and all its sub-directories.*

**\***.apk

**\***.class

*# It's possible to combine both forms to ignore files with certain*

*# extensions in certain directories. The following rules would be*

*# redundant with generic rules defined above.*

java**/\***.apk

gen**/\***.class

*# To ignore files only at the top level directory, but not in its*

*# subdirectories, prefix the rule with a `/`*

**/\***.apk

**/\***.class

*# To ignore any directories named DirectoryA*

*# in any depth use \*\* before DirectoryA*

*# Do not forget the last /,*

*# Otherwise it will ignore all files named DirectoryA, rather than directories*

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

*# This would ignore*

*# DirectoryA/*

*# DirectoryB/DirectoryA/*

*# DirectoryC/DirectoryB/DirectoryA/*

*# It would not ignore a file named DirectoryA, at any level*

*# To ignore any directory named DirectoryB within a*

*# directory named DirectoryA with any number of*

*# directories in between, use \*\* between the directories*

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

*# This would ignore*

*# DirectoryA/DirectoryB/*

*# DirectoryA/DirectoryQ/DirectoryB/*

*# DirectoryA/DirectoryQ/DirectoryW/DirectoryB/*

*# To ignore a set of files, wildcards can be used, as can be seen above.*

*# A sole '\*' will ignore everything in your folder, including your .gitignore file.*

*# To exclude specific files when using wildcards, negate them.*

*# So they are excluded from the ignore list:*

**!**.gitignore

*# Use the backslash as escape character to ignore files with a hash (#)*

*# (supported since 1.6.2.1)*

\*#\*#*

**Cleaning up ignored files**

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

**git clean** -Xn *#display a list of ignored files*

**git clean** -Xf *#remove the previously displayed files*

Note: -X (caps) cleans up *only* ignored files. Use -x (no caps) to also remove untracked files.

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

$ **cat** .gitignore

**\***.o

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

example.o

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

**\***.txt

**!**important.txt

The above example instructs Git to ignore all files with the .txt extension except for files named important.txt.

===========================🡺 A completer avec doc maison

.gitignore ignores files locally, but it is intended to be committed to the repository and shared with other

contributors and users.

**Section 5.6: Ignore files locally without committing ignore rules**

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

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

The above command instructs Git to assume my-file.txt hasn't been changed, and not to check or report changes. The file is still present in the repository. This can be useful for providing defaults and allowing local environment overrides,

**Section 5.8: Ignoring a file in any directory**

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

foo.txt *# matches all files 'foo.txt' in any directory*

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

bar**/\*\*/**foo.txt *# matches all files 'foo.txt' in 'bar' and all subdirectories*

If you are unsure which rules to list in your .gitignore file, or you just want to add generally accepted exceptions to your project, you can choose or generate a .gitignore file:

https://www.gitignore.io/

https://github.com/github/gitignore

**Section 5.11: Create an Empty Folder**

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

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

$ touch .gitkeep

This command just makes a blank .gitkeep file in the current directory

**Section 5.12: Finding files ignored by .gitignore**

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

**git status** --ignored

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

$ **git status** --ignored --untracked-files=all

## Setting Up Your Profile

For every commit you do, Git will try to associate a name and email address. One of the first things you’ll want to do in Git is to set these values. You can set them as global configuration values with the git config command:

$ git config --global user.name “Scott Chacon”

$ git config --global user.email [schacon@gmail.com](mailto:schacon@gmail.com)

If you want to set different values for a specific project, just leave out the —global and it will write the same snippet into your .git/config file in that repository, which will overwrite your global values

You can change those variables at any time either by editing that file, or running the git config commands again

## Normal Workflow Examples

### Ignoring

First off, we will often want Git to automatically ignore certain files – often ones that are automatically generated during our development. To do this, we can add patterns into the .gitignore file to tell Git that we don’t want it to track them.

Basic Configuration

Git configuration is in *~/.gitconfig*; this is a plain-text file, which you can edit directly as well, if you like. Its format is called *INI style* (after a file extension commonly used for it,

though not by Git), and is divided into sections, like so:

The parameters have full names qualified by the section in which they appear using a dot;

for example, the parameters mentioned in this example are:

* user.name

You use these names when reading or setting parameters with git config, rather than editing the file yourself. To set a parameter with git config:

**$ git config --{local,global,system}** *parameter value*

If you give this command when your current directory is inside a Git repository, it implies --local, and it will change the configuration for that repository only, in the file *.git/config*. Otherwise, the default is --global, which applies to your overall personal Git configuration in *~/.gitconfig*

Git reads these three configurations, each if available, in the order system, global, then local. Settings made in a later configuration override those from an earlier one so that, for example, you can set your normal email address with --global but change it for commits made in a specific repository if you use a different address when corresponding about that work.

Personal Identification

**$ git config --global user.name "Richard E. Silverman"**

**$ git config --global user.email** [**res@oreilly.com**](mailto:res@oreilly.com)

Text Editor

The default editor varies by platform; on Unix, it is the ubiquitous *vi*. You can customize this with the environment variables GIT\_EDITOR, EDITOR, or VISUAL (the latter two are respected

by many other Unix programs as well), or by setting core.editor. For example (reflecting the author’s predilections):

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

Command Aliases

Git has its own internal alias system as well, which may be more convenient. This command:

**$ git config --global alias.cp cherry-pick**

defines git cp as an alias for git cherry-pick. An exclamation point means to pass the alias definition to the shell, letting you use more complex aliases; for example, this definition in *~/.gitconfig*:

setup = ! "git init; git add .; git commit"

defines an alias git setup, which sets up a new repository using the contents of the current directory.

Ignoring Files

While you’re working on a project, you may have files in your working directory that you want Git to simply ignore.

Generally speaking, anything that is automatically generated you probably don’t want tracked by Git, and you don’t want Git constantly including them in listings or complaining about them either.

Syntax of “Ignore Patterns”

# Ignore this specific file in a subdirectory.

conf/config.h

# Ignore this specific file in the current directory.

# (not “./”)

/super-cool-program

## Patterns without slashes apply everywhere in this

## directory and below.

# Ignore individual objects and object archives

# (\*.o and \*.a).

\*.[oa]

# Ignore shared objects...

\*.so

# ... but don't ignore this file, or my boyfriend

# will complain.

!my.so

# Ignore any directories named “temp,” but still

# notice regular files and symbolic links with

# that name.

temp/

# CHAPTER 2 Getting Started

Creating a New, Empty Repository

The command:

**$ git init *directory***

creates the argument directory if needed, and a directory named *.git* inside it holding a new, empty Git repository. Aside from the repository itself in *.git*, that directory will hold the *working tree*: copies of the files and directories under version control that you will edit. The *.git* directory holds the files and data structures that form the repository itself, including the database of all historical revisions of all project files. Unlike CVS and (until recently) Subversion, there is no control directory in each directory of the working tree (*CVS* and *.svn*); there is just

the one *.git* directory at the top of the project tree.

The default with no argument is the current directory; that is, a simple git init creates a new *.git* in the current directory. git init is a safe command. It will not remove any existing files in the target directory, the usual pattern being that you are about to add those files to the new repository. It will also not damage an existing repository, even though it gives a somewhat heartstopping message about “reinitializing” if you do it; all this actually does is make some administrative updates, such as picking up new templates for “hook” scripts made available by the system

administrator (see “Git Hooks” on page 196).

Selected Options

--bare

Creates a “bare” repository; that is, one without an associated working tree. The internal repository files that would otherwise be inside *.git* are instead created in the target directory

Importing an Existing Project

In more detail: git add . adds the current directory to the (initially empty) index; this includes files as well as directories and their contents, and so on, recursively. git commit then creates a new tree object capturing the current state of the index, as well as a commit object with your comment text, personal identification, the current time, and so on, pointing to that tree. It records these in the object database, and then finally sets the *master* branch to the new commit; that is, makes the ref refs/heads/ master point to the new commit ID:

# Committing

## Getting a Git Repository

There are two major ways you will get a Git repository – you will either clone an existing project, or you will initialize a new one

To create a new Git repository somewhere, simply go to the directory you want to add version control to and type:

$ git init

$ git add .

$ git commit -m ‘my first commit’

This will by default create a new directory called simplegit and do an initial checkout of the master branch. If you want to put it in a different directory than the name of the project, you can specify that on the command line, too

$ git clone git://github.com/schacon/simplegit.git my\_directory

## Adding and Committing



If we want to commit all our changes, we can use this shorthand, which will automatically run a git add on every modified file to our index, then commit the whole thing:

$ git commit -a -m ‘committing all changes’



there is another way of adding files that makes for a more controlled and thematic set of commits. This is called *interactive* adding, and it is a very powerful tool to controlling what goes into each commit.

The interactive shell is pretty simple and very powerful – playing with it instead of running git add commands directly may help in under­standing what’s happening, since you can see the status of your files in the index versus the working directory more clearly. It helps visualize that what is in your index (the staged column) is what will be committed when you run git commit.

For removing files from your tree, you can simply run:

git rm <filename>

This will remove that file from the index (and thus from the next com­mit) as well as from your working directory. On your next commit, the tree that commit points to will simply not contain that file anymore.

**Section 4.1 Staging all changes to files**

**git add** .

In version 2.x, **git add** . will stage all changes to files in the current directory and all its subdirectories. However, in 1.x it will only stage new and modified files, not deleted files.

**Section 4.2 Unstage a file that contains changes**

**git reset <**filePath>

**Section 4.3: Add changes by hunk**

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

**git add** -p

This opens an interactive prompt that allows you to look at the diffs and let you decide whether you want to include them or not. You can also open this via **git add** --interactive and selecting p.

**git add** -i (or --interactive) will give you an interactive interface where you can edit the index, to prepare what you want to have in the next commit. You can add and remove changes to whole files, add untracked files and remove files from being tracked, but also select subsection of changes to put in the index, by selecting chunks of changes to be added, splitting those chunks, or even editing the diff.

**Section 4.7: Stage deleted files**

**git rm** filename

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

**git rm** --cached filename

# Versioning Commits

In Git, files can have the following statuses:

**Untracked**: This a file that exists in the working tree whose changes are not being monitored by Git and aren't listed in the **gitignore** file.

**Unstaged**: This a file whose changes are being tracked by Git; the file has been changed since the last commit and has yet to be moved to the **index**.

**Staged**: This is a file whose changes are being tracked by Git; the file has been changed since the last commit and has been moved to the index. This file is a file that is ready for the Git commit. Staged files are the files in the index that are different from their corresponding files in the **HEAD**, that is, the version of the file in the most recent commit.

**git status** is a utility that is used by Git. It's used to retrieve the details of files that are untracked, unstaged, or staged. **git status** lists files in order of their statuses.

The **git status** output is lengthy in nature. To view a brief list and status, use the **-s** or **--short** option with the **git status** command.

you should be able to use **git status** to view untracked, unstaged, and staged files.

To ensure flexibility, atomic commits are preferred and emphasized in version control. The term atomic commits refers to units of change that can be treated as a single unit. This normally implies a single file or a small set of files. This enables the addition and removal of changes without affecting a large set of files.

The **git add** command uses the following syntax:

**git add [options] [path\_to\_files]**

The **[options]** used with **git add** include the following:

**-n or --dry-run**

This option simulates the behavior of **git add** for the specified file:

**-f or --force**

This option adds ignored files to the index:

**-i or --interactive**

This option creates an interactive prompt that can be used to add files from the working tree to the index:

**-p or --patch**

The **--patch** option caters to adding portions of a file to the index

The **git commit** command saves the files in the index. This commit operation stores a message along with the commit. The message describes the additions or alterations associated with the created snapshot.

The syntax of this command is as follows: **git commit [options].**

The **git commit** command requires that a message be provided for each commit operation.

The options supported by this command include **-m [text] or --message [text].**

This message is used to associate the index file with the commit action:

**-a or -all**

This option instructs the **git commit** utility to stage tracked files that are unstaged, that is, the tracked files have been added to the index if the files are yet to be staged. Untracked files are not added to the index.

**-p or --patch**

This launches the interactive patch tool. The options are akin to those that are available through the **git add** command. See the preceding table for more information on this.

**-C [commit hash]** or **--reuse-message=[commit hash]**

This instructs **git commit** to reuse a commit message and the author information of the specified commit hash.

**-F [file]** or **--file=[file]**

This command specifies a file from which a commit message should be obtained.

**-t [file]** or **--template [file]**

This command specifies the commit message template file.

**-e** or **--edit**

This command edits the provided commit message. This refers to the message provided by the **-F**, **-t**, and **-m** options.

**--no-edit**

This command uses the specified message as is. Do not launch an editor to edit the message.

**--author=[author]**

This command overrides the details of a commit author, and takes the following form:

**git commit --author="Kifeh Polyswarm <kifeh@poly-swarm.com>"**

**--date=[date]**

As you can see, this overrides the date details used in a commit.

**-q** or **--quiet**

This command suppresses the summary message that's returned after running the **git commit** command

**git commit -C 474b5caaf480f7a367c1c456a53868c7fe32b9df --no-edit**

### ***git rm***

The **git rm** command performs two roles. These roles are used to remove files from the working directory and the index.

**-r**

This option is applicable when using the **git rm** command in a directory. It removes the directory's contents recursively. This means that the directory and its contents are removed.

**--cached**

This option removes the specified files from the index only.

**-f** or **--force**

**git rm** checks the files marked for removal for matches, with the files in HEAD, at the tip of the current branch. This check is conducted before the file(s) are removed. The **-f** option overrides this check.

The ***rm*** command removes the specified file from the working tree only. The ***git rm*** command, on the other hand, removes the file from the index and the working tree. This provides a shorter process for deleting files, since with ***rm***, you need to run ***git add*** to impact the deletion process in the index.

### **git mv**

In the event that you need to update the index for both, old and new paths automatically, the **git mv** command serves that purpose.

This command has two forms of implementation:

1. git mv [options] [source] [destination]
2. git mv [options] [source] … [destination]

(1) is used to rename a file.

(2) is used to move a file.

### Amending a Single Most Recent Commit

The most recent commit can be edited by using **--amend** in the **git commit** command.

**Amending Multiple Commits**

The **git rebase** command provides the **reword** and **edit** options to edit the commits. The **reword** option allows you to edit a message, while the **edit** option supports editing a commit message, as well as the contents of a commit.

# Log

2.2 Prettier log

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

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

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

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

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

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

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

**Parameter Details**

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

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

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

%d ref names

%s subject [commit message]

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

**%**an author name

Searches for **addition** or **removal** of specific string or the string **matching** provided

git log -S"#define SAMPLES"

Searches for **changes** in **lines containing** specific string or the string **matching**

git log -G"#define SAMPLES"

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

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

$ **git shortlog** –s

Searching commit string

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

Log for a range of lines whithin a file.

$ **git log** -L 1,20:index.html

Filter logs

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

**git log** --after 2016-05-01

An alias to --after is --since.

Flags exist for the converse too: --before and --until.

You can also filter logs by author. e.g.

**git log** --author=author

log showing committed files

**git log** –stat

2.13 Git log between tow branches

**git log** master..foo

**git log**  --decorate --source --pretty=format:'%h %ar %an %d %s"' --all --graph

**Section 11.5: Show pretty log with branch graph**

**[alias]**

logp=log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

lg = log --graph --date-order --first-parent \

--pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold

cyan)<%an>%Creset'

lgb = log --graph --date-order --branches --first-parent \

--pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold

cyan)<%an>%Creset'

lga = log --graph --date-order --all \

--pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold

cyan)<%an>%Creset'

Here an explanation of the options and placeholder used in the --pretty format (exhaustive list are available with **git help** log )

--graph - draw the commit tree

--date-order - use commit timestamp order when possible

--first-parent - follow only the first parent on merge node.

--branches - show all local branches (by default, only current branch is shown)

--all - show all local and remotes branches

%h - hash value for commit (abbreviated)

%ad - Date stamp (author)

%an - Author username

%an - Commit username

%C(auto) - to use colors defined in [color] section

%Creset - to reset color

%d - --decorate (branch & tag names)

%s - commit message

%ad - author date (will follow --date directive) (and not commiter date)

%an - author name (can be %cn for commiter name

**Chapter 48: Git statistics**

Section 48.3 Git statistics

$ git shortlog –sn

$ git shortlog -sne

Git shortlog is used to summarize the git log ouput and group the commits by author

-s skips the summary

-n changes the ordering from alphabetical to number of commits descending

-e shows the email

**git shortlog** -sne *#Names along with their email ids and the Number of commits*

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

In order to get the total number of commits that each developer or contributor has made on a repository, you can simply use the **git shortlog**:

**git shortlog** -s

which provides the author names and number of commits by each one.

Additionally, if you want to have the results calculated on all branches, add --all flag to the command:

**git shortlog** -s --all

**Section 48.4: Commits per date**

**git log** --pretty=format:"%ai" **| awk** '{print " : "$1}' **| sort** -r **| uniq** -c

**Section 48.4: Commits per date**

**git log** --pretty=format:"%ai" **| awk** '{print " : "$1}' **| sort** -r **| uniq** -c

**Section 48.5: Total number of commits in a branch**

**git log** --pretty=oneline **|wc** -l

**Section 48.6: List all commits in pretty format**

**git log** --pretty=format:"%Cgreen%ci %Cblue%cn %Cgreen%cr%Creset %s"

This will give a nice overview of all commits (1 per line) with date, user and commit message.

Section50.1 gitk git gui

Gitk is a graphical history viewer

$ gitk [git log options]

$gitk –all

$ git gui is a primarly a tool for crafting commits

Suppose that you had started an interactive rebase:

**git rebase** --interactive HEAD~20

and by mistake, you squashed or dropped some commits that you didn't want to lose, but then completed the rebase. To recover, do **git reflog**

## History and Logs

The **git log** command lists the history of a branch and the repository, by extension. It uses options and a range to define the duration for which the logs should be retrieved:

**git log [options] [version range] [path\_to\_file\_or\_directory] [version range]**

The **git log** command can display the history of a branch when given a range of version hashes:

**git log [hash\_1]..[hash\_2]**

### **Options**

### The **--follow** command retrieves and displays the history of a file, beyond rename events:

The **--decorate[=short** or **full** or **no]** command displays the **ref** name of the listed commits as seen in the following screenshots:

**git log --decorate=short**

### *The short option omits the* ***ref/heads/,ref/remotes/,*** *and* ***ref/tags/*** *prefixes from the* ***ref*** *name that is displayed.*

### *The full option displays the full ref name. It includes the* ***ref/heads/, ref/remotes/,*** *and* ***ref/tags/*** *prefixes in the ref name that is displayed.*

The **-L [start],[end]:[path\_to\_file]** command views the changes that have been made to a section of a file, from line number X to line number Y:

**git log -L 6,12:./src/lib/compute.py**

The **--[number]** , -**n [number]** ,and **--max-count=[number]** the specified number of commits only

### **git log -3 or git log -n 3**

The **--skip=[number]** command skips the specified commits and displays the rest:

**git log --skip=4**

The **--since=[date]** or **--after=[date]** commits that have been created after a given date:

**git log --after=25/08/2018**

The **--until=[date]** or **--before=[date]** commits that precede a given date:

**git log --before=24/08/2018**

The **--pretty=[format]** command displays the history of a branch using a prescribed format:

**git log --pretty=oneline**

### ***git log --pretty=medium***

**git log --pretty=format:[format string]**

The format is a string that's in the form of **%placeholder1 %placeholder2 %placeholderN**.

The supported placeholders include the following and are shown in the following screenshots:

* **%H**: The commit hash
* **%h**: The abbreviated commit hash
* **%T**: A tree hash
* **%t**: An abbreviated tree hash
* **%P**: The parent hash
* **%p**: The abbreviated parent hash
* **%an**: Author name
* **%ae**: Author email
* **%ad**: Author date
* **%ar**: A relative author date
* **%at**: The Unix timestamp version of the author date
* **%s**: The subject of the commit
* **%b**: The body of the commit
* **%n**: A newline

**git log --pretty=format:"%H %an"**

### **git log --pretty=format:"%H %an %ae"**

### **git log --pretty=format:"%H %an %ae %n %s %n %b"**

## Log – the Commit History

The log com­mand can show you nearly anything you want to know about your commit history. Also, since the entire history is stored locally, it’s really fast compared with most other SCM systems,

$ git log

This will show you the SHA-1 of each commit, the committer and date of the commit, and the full message, starting from the last com­mit on your current branch and going backward in reverse chrono­logical order (so if there are multiple parents, it just squishes them together, interleaving the commits ordered by date)

### Formatting Log Output

The default format takes up a lot of space though, so there are ways to limit and format this output differently. —pretty is a useful option for formatting the output in different ways.

For example, we can list the commit SHA-1s and the first line of the message with —pretty=oneline:

With —pretty, you can choose between *oneline*, *short*, *medium*, *full*, *fuller*, *email*, *raw* and *format:(string)*, where (string) is a format you specify with variables (ex: —format:”%an added %h on %ar” will give you a bunch of lines like “Scott Chacon added f1cc9df 4 days ago”).

### Filtering Log Output

There are also a number of options for filtering the log output. You can specify the maximum number of commits you want to see with -n, you can limit the range of dates you want to see commits for with —since and —until, you can filter it on the author or committer, text in the commit message and more. Here is an example showing at most 30 commits between yesterday and a month ago by me :

### git log -n 30 -- since=”1 month ago” --until=yesterday --author=”schacon”

2.12 Show the contents of a single commit

**git show** 48c83b3

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

**git diff**

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

To add (stage) these changes, you can use **git add**. If a file is staged, but was modified after it was staged, **git diff** will show the differences between the current file and the staged version.

**Section 6.2: Show changes between two commits**

**git diff** 1234abc..6789def *# old new*

Show the changes made in the last 3 commits:

**git diff @**~3..**@** *# HEAD -3 HEAD*

Note: the two dots (..) is optional, but adds clarity. This will show the textual difference between the commits, regardless of where they are in the tree.

**Section 4.5: Show Staged Changes**

To display the hunks that are staged for commit:

**git diff** --cached

**Section 6.3: Show differences for staged files**

**git diff** --staged

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

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

**git diff** --cached

Which is just a synonym for --staged or

**git status** -v

Which will trigger the verbose settings of the status command.

**Section 6.4: Comparing branches**

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

**git diff** original new *# equivalent to original..new*

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

**git diff** original...new *# equivalent to $(git merge-base original new)..new*

Using only one parameter such as

git diff original

is equivalent to

git diff original..HEAD

**Section 6.5: Show both staged and unstaged changes**

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

**git diff** HEAD

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

**git status** -vv

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

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

**Section 6.6: Show dierences for a specific file or directory**

**git diff** myfile.txt

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

This also works for directories:

**git diff** documentation

The above shows the changes between the previous commit of all files in the specified directory (documentation**/**) and the locally-modified versions of these files, that have not yet been staged. To show the difference between some version of a file in a given commit and the local HEAD version you can specify the commit you want to compare against:

**git diff** 27fa75e myfile.txt

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

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

To show the difference between the version specified by the hash ada9b57 and the latest commit on the branch my\_branchname for only the relative directory called my\_changed\_directory**/** you can do this:

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

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

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

-Hello world

+Hello world**!**

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

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

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

git diff @~1 @ --word-diff=color

**git diff** HEAD^ HEAD

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

To view difference between two branch or commit

**git diff <**branch1**/**commitId1**>..<**branch2**/**commitId2**>**

To view diff with current branch

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

To view summary of changes

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

To view files that changed after a certain commit

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

To view files that are different than a branch

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

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

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

**Section 6.11: Using meld to see all modifications in the working directory**

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

will show the working directory changes. Alternatively,

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

will show the differences between 2 specific commits.

**Section 53.5: Setting up Beyond Compare**

You can set the path to bcomp.exe

**git config** --global difftool.bc3.path 'c:\Program Files (x86)\Beyond Compare 3\bcomp.exe'

and configure bc3 as default

**git config** --global diff.tool bc3

The **git diff** command is used to compare one snapshot of changes to another. As the name suggests, this utility supports evaluating the differences between two snapshots of a repository.

### Comparing the Working Tree to the Index

To compare the entire working tree to the index, run the git diff command without specifying a path:

$ git diff

This command supports examining the differences of a specific file or directory by accepting a path:

git diff -- [path\_to\_a\_file\_or\_directory]

1. git diff -- src/lib/
2. git diff -- src/lib/compute.py

(1) and (2) compare the version in the working directory of the specified paths to the version that's present in the index.

### Comparing the Working Tree to an Arbitrary Commit or Branch

A comparison can be made between the working tree and a specific commit on the same branch, or even the tip of a given branch. To compare the working tree to a given commit, use the following syntax:

$ git diff [commit\_hash] -- [path\_to\_a\_file\_or\_directory]

1. $ git diff HEAD -- src/
2. $ git diff f4e4e8d5b292dc94468b6f88223cac4f55c03713 -- src/lib/
3. $ git diff master

(1) Compares the version in the working directory of the **src** directory, to the version of the most recent commit on the current branch.

(2) Compares the version in the working directory of the src directory to the version in the snapshot represented by the hash f4e4e8d5b292dc94468b6f88223cac4f55c03713.

(3) Compares the version in the working directory of the **src** directory to the version at the tip of the branch **master**.

### Comparing the Index to an Arbitrary Commit

To compare the files in the index to a specific commit, (for example, the most recent commit, also referred to as the tip of the branch), you can use the **--staged** or **--cached** option with the **git diff** command. A commit hash is required for this scenario. The **git diff** command defaults to **HEAD** in the absence of a specific commit hash:

 **git diff --cached [commit\_hash]** or

**git diff --cached [commit\_hash] -- [path\_to\_a\_file\_or\_directory]**

1 **git diff --cached HEAD -- /src/lib/compute.py**

(1) compares the version in the index of the **compute.py** file to the version at the tip of the current branch.

### Comparing Commits and Branches

**git diff** provides a variant of the command's usage that supports comparing commits and branches.

To compare two commits or the tips of two branches, use the following syntax:

**git diff [commit\_hash or branch\_name] [commit\_hash or branch\_name]** or

**git diff [commit\_hash or branch\_name] [commit\_hash or branch\_name] -- [path\_to\_a\_file\_or\_directory]** or

**git diff [commit\_hash or branch\_name]..[commit\_hash or branch\_name]**

1. **git diff ft-add-encapsulating-class master**
2. **git diff ft-add-encapsulating-class..master**
3. **git diff da39a3ee5e6b4b0d3255bfef95601890afd80709 6f7e437faa5a7fce15d1ddcb9eaeaea377667b**

(1) and (2) compare the differences between the tips of the specified branches.

(3) compares the differences between the files at the point referenced by the specified hashes

Using the **...** notation, **git diff** is capable of comparing the changes that have been made on branch A to branch B. This occurs from the point where the two branches share an ancestor to the most recent commit of branch B.

To achieve this, use the following syntax:

**git diff [branch\_A]...[branch\_B] -- [path\_to\_a\_file\_or\_directory]**

1. **git diff ft-add-encapsulating-class...master**

(1) lists the changes that have occurred in the branch **master**, since the **ft-add-encapsulating-class** branch was created from the branch **master**.

Note

When .. or … are used with the **git diff** command, it implies the comparison of two points in history, and not a range.

**git diff** defaults to HEAD when a commit hash or branch name is not specified.

# Branches

**Section 14.1: Creating and checking out new branches**

To create a new branch, while staying on the current branch, use:

**git branch <**name**>**

Generally, the branch name must not contain spaces and is subject to other specifications listed here. To switch to an existing branch :

**git checkout <**name**>**

To create a new branch and switch to it:

**git checkout** -b **<**name**>**

To create a branch at a point other than the last commit of the current branch (also known as HEAD), use either of these commands:

**git branch <**name**> [<**start-point**>]**

**git checkout** -b **<**name**> [<**start-point**>]**

The **<start-point>** can be any revision known to git (e.g. another branch name, commit SHA, or a symbolic reference such as HEAD or a tag name):

**git checkout** -b **<**name**>** some\_other\_branch

**git checkout** -b **<**name**>** af295

**git checkout** -b **<**name**>** HEAD~5

**git checkout** -b **<**name**>** v1.0.5

To create a branch from a remote branch (the default **<remote\_name>** is origin):

**git branch <**name**> <**remote\_name**>/<**branch\_name**>**

**git checkout** -b **<**name**> <**remote\_name**>/<**branch\_name**>**

If a given branch name is only found on one remote, you can simply use

**git checkout** -b **<**branch\_name**>**

which is equivalent to

**git checkout** -b **<**branch\_name**> <**remote\_name**>/<**branch\_name**>**

Sometimes you may need to move several of your recent commits to a new branch. This can be achieved by branching and "rolling back", like so:

**git branch <**new\_name**>**

**git reset** --hard HEAD~2 *# Go back 2 commits, you will lose uncommitted work.*

**git checkout <**new\_name**>**



You can quickly switch to the previous branch using

**git checkout** -

**Section 14.2: Listing branches**

Git provides multiple commands for listing branches. All commands use the function of **git branch**, which will provide a list of a certain branches, depending on which options are put on the command line.

List local branches **git branch**

List local branches verbose **git branch** -v

List remote and local branches **git branch** -a OR **git branch** --all

List remote and local branches (verbose) **git branch** -av

List remote branches **git branch** -r

List remote branches with latest commit **git branch** -rv

List merged branches **git branch** --merged

List unmerged branches **git branch** --no-merged

List branches containing commit **git branch** --contains **[<**commit**>]**

* Adding an additional v to -v e.g. $ **git branch** -avv or $ **git branch** -vv will print the name of the upstream branch as well.
* Branches shown in red color are remote branches

**Section 14.3: Delete a remote branch**

To delete a branch on the origin remote repository, you can use for Git version 1.5.0 and newer

**git push** origin :**<**branchName**>**

and as of Git version 1.7.0, you can delete a remote branch using

**git push** origin --delete **<**branchName**>**

To delete a local remote-tracking branch:

**git branch** --delete --remotes **<**remote**>/<**branch**>**

**git branch** -dr **<**remote**>/<**branch**>** *# Shorter*

**git fetch <**remote**>** --prune *# Delete multiple obsolete tracking branches*

**git fetch <**remote**>** -p *# Shorter*

To delete a branch locally. Note that this will not delete the branch if it has any unmerged changes:

**git branch** -d **<**branchName**>**

To delete a branch, even if it has unmerged changes:

**git branch** -D **<**branchName**>**

$ **git branch** -d dev

Deletes the branch named dev *if* its changes are merged with another branch and will not be lost. If the dev branch does contain changes that have not yet been merged that would be lost, **git branch** -d will fail:

$ **git branch** -d dev

Per the warning message, you can force delete the branch (and lose any unmerged changes in that branch) by using the -D flag:

$ **git branch** -D dev

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

There are three ways of creating a new branch feature which tracks the remote branch origin**/**feature:

**git checkout** --track -b feature origin**/**feature,

**git checkout** -t origin**/**feature,

**git checkout** feature - assuming that there is no local feature branch and there is only one remote with

the feature branch.

To set upstream to track the remote branch - type:

**git branch** --set-upstream-to=**<**remote**>/<**branch**> <**branch**>**

**git branch** -u **<**remote**>/<**branch**> <**branch**>**

where:

**<remote>** can be: origin, develop or the one created by user,

**<branch>** is user's branch to track on remote.

To verify which remote branches your local branches are tracking:

**git branch** –vv

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

**git checkout** --orphan new-orphan-branch

**Section 14.8: Rename a branch**

Rename the branch you have checked out:

**git branch** -m new\_branch\_name

Rename another branch:

**git branch** -m branch\_you\_want\_to\_rename new\_branch\_name

**Section 14.9: Searching in branches**

To list local branches that contain a specific commit or tag

**git branch** --contains **<**commit**>**

To list local and remote branches that contain a specific commit or tag

**git branch** -a --contains **<**commit**>**

**Section 14.10: Push branch to remote**

Use to push commits made on your local branch to a remote repository.

The **git push** command takes two arguments:

A remote name, for example, origin

A branch name, for example, master

For example:

**git push <**REMOTENAME**> <**BRANCHNAME**>**

As an example, you usually run **git push** origin master to push your local changes to your online repository.

Using -u (short for --set-upstream) will set up the tracking information during the push.

**git push** -u **<**REMOTENAME**> <**BRANCHNAME**>**

By default, **git** pushes the local branch to a remote branch with the same name. For example, if you have a local called new-feature, if you push the local branch it will create a remote branch new-feature as well. If you want to use a different name for the remote branch, append the remote name after the local branch name, separated by ::

**git push <**REMOTENAME**> <**LOCALBRANCHNAME**>**:**<**REMOTEBRANCHNAME**>**

A branch is just a pointer to a commit, so you can freely move it around. To make it so that the branch is referring to the commit aabbcc, issue the command

**git reset** --hard aabbcc

Please note that this will overwrite your branch's current commit, and as so, its entire history. You might loose some work by issuing this command. If that's the case, you can use the reflog to recover the lost commits. It can be advised to perform this command on a new branch instead of your current one. However, this command can be particularly useful when rebasing or doing such other large history modifications.

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

**git rev-list** --oneline master ^origin**/**master

Git rev-list will list commits in one branch that are not in another branch. It is a great tool when you're trying to

figure out if code has been merged into a branch or not.

Using the --oneline option will display the title of each commit.

The ^ operator excludes commits in the specified branch from the list.

You can pass more than two branches if you want. For example, **git rev-list** foo bar ^baz lists commits

in foo and bar, but not baz.

**Section 60.1: Delete local branches that have been deleted on**

**the remote**

To remote tracking between local and deleted remote branches use

**git fetch** -p

you can then use

**git branch** -vv

to see which branches are no longer being tracked.

Branches that are no longer being tracked will be in the form below, containing 'gone'

branch 12345e6 **[**origin**/**branch: gone**]** Fixed bug

you can then use a combination of the above commands, looking for where 'git branch -vv' returns 'gone' then using '-d' to delete the branches

**git fetch** -p **&& git branch** -vv **| awk** '/: gone]/{print $1}' **| xargs git branch** –d

### Creating:

**git branch [branch\_name]**

**git branch --set-upstream-to [remote\_branch\_name]**

e.g. **git branch --set-upstream-to origin/ft-support-exponents**

**git branch --unset-upstream [branch\_name]**

**git branch [branch\_name] [start\_point]**

**Renaming:**

**git branch -m [old\_branch\_name] [new\_branch\_name]**

**git branch -M [old\_branch\_name] [new\_branch\_name]**

This is similar to invoking **git branch** with the **--move** and **--force** options.

**git branch -c [old\_branch\_name] [new\_branch\_name]**

**Copy:**

**git branch -C [old\_branch\_name] [new\_branch\_name]**

This is similar to invoking **git branch** with the **--copy** and **--force options.**

**Deleting:**

**git branch -d [branch\_name]**

Delete a branch, granted that it's fully merged into its upstream branch or the **HEAD**, in the event that the upstream branch is not specified.

**git branch -D [branch\_name]**

This forces the deletion of a branch.

It's similar to using **--delete --force.**

**Listing:**

**git branch --list**

**git branch --list [pattern]**

For example, you can use **git branch --list 'ft\*'.**

**git branch --contains [commit]**

For example, you can use **git branch --contains 8354043.**

**git branch --no-contains [commit]**

**git branch --merged [commit]**

For example, you can use **git git branch --merged 8354043.**

This lists branches that have been merged into a given commit, that is, commits whose tip is reachable from the given commit.

**git branch --no-merged [commit]**

This is used for branches that are not merged into the given commit.

**git branch -a**

This is used to get all branches:

**git branch -r**

This is used for the remote tracking of branches.

**Switching to New and Existing Branches**

The process of moving from one branch to another is done by switching to **[branch]** and then setting the files in the index and working tree to reflect [branch]'s latest commit. Lastly, you must set the **HEAD** to branch:

**git checkout [branch\_name]**

**git checkout -b [branch\_name]**

**git branch -B [branch\_name] [start\_point]**

Create a new branch and set its tip to **[start\_point]**. If the branch exists, then reset it to **[start\_point]**.

**git checkout -b sample**

**git branch sample**

**git tag v1.0**

**git checkout [commit\_hash]**

**git checkout [tag]**

**git checkout --detach [branch]**

**Switching to a Specific Version of a File**

When switching, the git check out command takes the following syntax:

**git checkout [commit] -- [path]**

Other uses of **git checkout are as follows:**

**git checkout -b --orphan [new\_branch] [start\_point]**

This creates a branch whereby the first commit has no parent. This is necessary when certain information contained in the repository's history needs to remain unexposed for privacy reasons.

# CHAPTER 5 Branching

Switching Branches

**$ git symbolic-ref HEAD**

refs/heads/theodore

**$ git checkout simon**

Switched to branch 'simon'

**$ git symbolic-ref HEAD**

refs/heads/simon

Here, git symbolic-ref HEAD shows the ref (branch name) to which HEAD points:

This attempts to do three things:

1. Change the HEAD symref to point to the *commander* branch

2. Reset the index to match the tip of the new branch

3. Update the working tree index to match the index (this is called “checking out” the index, which gives the commandits name)

If these succeed, then you are now on the *commander* branch, with an index and working tree that match the tip of that branch. The following are some possible complications.

Uncommitted Changes

Suppose you have uncommitted changes to a tracked file when you try to switch branches. There are now four versions of the file in play: the two in the tip commits of the *master* and *commander* branches, and the two in your working tree and index (one or both of which have been altered, depending on whether you have staged the changes with git add). If the committed versions in the current and destination branches are the same, then Git will preserve your altered versions when switching branches, since they represent the same sets of changes in the new branch as in the old. It reminds you of a modified file *foo* thus:

**$ git checkout commander**

M foo

If the committed versions differ, however, or if the file does not exist at all in the destination branch, then Git warns you and refuses to switch:

Check Out with Merge

git checkout has a --merge (-m) option to help with this case. It performs a three-way merge between your working tree and the new branch, with the current branch as the base; it leaves you on the new branch, with the merge result in the working tree. As with any merge, you may have conflicts to resolve

If *simon* is fully merged in the current branch, then Git deletes it with no complaint. If it is not, but it is fully merged in its upstream branch, then Git proceeds with a warning:

**$ git branch -d simon**

Since Git doesn’t check other branches, it may be safe to delete a

branch because you know it is fully merged into another one; you can do this with the -D option as indicated, or switch to that branch first and let Git confirm the fully merged status for you.

Deleting the branch from the origin repository is not so obvious:

**$ git push origin :simon**

This is the general syntax for directly updating a remote ref. In this case, the local object name to the left of the colon is blank, meaning to just delete the remote ref.

Renaming a Branch

Renaming a local branch is simple:

**$ git branch -m *old new***

There is no direct way to rename the corresponding branch in a remote repository, however; you must separately push the new branch and delete the old one:

**$ git push -u origin *new***

**$ git push origin :*old***

You will need to tell others that you’ve done this, since when they pull they will get the new branch, but they will have to manually delete the old name with git branch -d. “Renaming” a branch is not actually a Git operation per se; git branch -m is just a shortcut for the create/delete routine.

# Remote

1.3 Sharing Code:

To share your code you create a repository on a remote server to which you will copy your local repository. To minimize the use of space on the remote server you create a bare repository:

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

Now copy your local repository to the remote:

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

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

Section3.1 Deleting a remote branch

To delete a remote branch in Git:

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

**or**

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

Section3.2 Changing git remote url

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

Section3.4 Removing local copies of deleted remote branches

To prune deleted branches from a specific remote:

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

To prune deleted branches from all remotes

**$ git remote** --all --prune

The pull command combines a fetch and a merge

The pull with –rebase flag command combines a fetch and a rebase

Section 3.6 ls-remote

**$ git ls-remote**

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

**$ git remote** --refs

**$ git remote** --tags

Section 3.8 Set Upstream on a new branch

You can create a new branch and switch to it using

**$ git checkout** –b mybranch

**$ git push** –u origin mybranch

Section 3.10 Renaming a remote

**$ git remote** –rename origin destination

Section 3.10 Show information about a specific remote

**$ git remote show**  origin

Section 3.12

**Section 3.10 Renaming a remote**

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

**$ git remote** –rename origin destination

Get existing remote name

**git remote**

Check existing remote with URL

**git remote** –v

**Section 3.11 Show information about a specific remote**

Output some information about a known remote: origin

**$ git remote show**  origin

Print just the remote's URL:

**git remote** get-url origin

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

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

**Section 3.12 Set the url for a specific remote**

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

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

## Fetching and Delivering Code

the **git remote** utility, to explore how we can manage the connection between the local and upstream repositories.

The **git remote** utility includes commands that help with managing the remote/upstream repositories that are associated with a local repository.

add the remote tracked repository by using the **git remote add origin git@github. com:kifeh-polyswarm/remote-demo.git**

View the remote configuration by using the **git remote -v command**

Rename the remote configuration from **origin** to **source-truth** by using the **git remote rename origin source-truth command**

View the specifics of the **source-truth remote** by using the **git remote show source-truth** command:

*The* ***prune*** *command removes local branches that correspond to branches that have been deleted from the remote repository:*

**$ git remote prune source-truth**

The configured remote can be removed by using the **remove** command. For example, to remove the **source-truth** remote, run **git remote remove source-truth:**

### Fetching, Pushing, and Pulling Changes

To enable collaboration in a distributed version control system, Git provides the means to retrieve and publish your contributions to the shared repository. To demonstrate this, we will explore **git fetch, git push,** and **git pull**.

**git fetch**

To navigate changes that were made to a repository, you need to utilize references to the changes made to the repository, and consequently a branch. This command allows you to explore the changes before integrating them into your work.

The **git fetch** command downloads remote-tracking branches and tags from the remote repository. These branches and tags indicate changes that have been made to the remote repository.

The content retrieved by this command is isolated from the content in the local repository, and, when you do this, the work that is being undertaken locally isn't affected by the downloaded content.

*Remote-tracking branches keep track of changes occurring on the branches in the remote repository. Remote-tracking branches can be viewed by using the* ***git branch -r*** *command. The* ***refs*** *for remote-tracking branches are stored in* ***/.git/refs/remotes/[remote\_name]****; for example,* ***/.git/refs/remotes/origin.***

you can retrieve or download all of the branches of the repository specified by **[remote]** with **git fetch [remote]** e.g. **git fetch origin:**

**git fetch [remote] [branch]**

You can fetch the **[branch]** from the upstream repository specified by **[remote]** with **git fetch origin master:**

**git fetch --all**

You can fetch branches from all of the remote connections defined for a repository. For example, if you have a remote **origin** and **upstream**, as you will see in the Chapter 4: Branches, this command will fetch branches from the two repositories identified by **origin** and **upstream**:

**git fetch --prune**

You can remove remote-tracking references that have ceased to exist in the remote repository, and then proceed to fetch the branches and their corresponding commits, files, and **refs** with: **+refs/heads/\*:refs/remotes/origin/\***.

#### Note

The **git fetch** command uses the **refspec** defined in the repository-level config. This is defined in the **remote.[remote\_name].fetch** config value. The **refspec** can be retrieved by running the **git config --local --list** command.

The **refspec** dictates that **refs** stored in the remote repository in **refs/heads/** are tracked locally, under **refs/remotes/origin/.**

The **+** indicates that references should be updated, including in scenarios where the commit is not a **fast-forward.**

The **git push** command picks local commits and updates the remote branch with the local commits. By default, the **git push** command only supports pushing commits in a fast-forward mode. If the commits being pushed are non-fast-forward, you're required to either push changes by force, or update the local repository by merging the commits from the remote branch

This command takes the form of **git push [remote\_name]**

You can push commits from the current branch to the remote branch configured on the repository with **git push origin**

**git push**

This is similar to **git push [remote\_name]**

**git push [remote\_name] [branch\_name]**

This variant will push commits from the local branch to the specified branch on the specified remote repository. You can use **git push origin develop** for this

**git push origin [local\_branch]:[remote\_branch]**

This command creates a new remote branch, bearing the commits in the **[local\_branch]**. This command is useful when the local branch and the remote branch do not have matching names. You can use **git push origin feature-video:video-experiment:**

**git push origin :[branch\_name]**

This variant deletes the specified branch. You can use **git push origin :bug-logout-mobile** for this

There are several options that are commonly used with the **git push** command. These include the following options:

The **--all** local branches. These branches are the branches that correspond to the **heads** stored in **.git/refs/heads/.** You can use **git push --all** for this:

**--force or -f**

This forces **git push** to update the remote branch in the event non-fast-forward changes are encountered. You can use **git push -f origin master** for this**:**

**git push --force origin maste**

You can set a tracking reference to the upstream branch for the current branch by using **git push -u origin master.**

### Dealing with Non-Fast-Forward Commits

As we explained earlier, **git push** utilizes a fast-forward mode to merge commits. Non-fast-forward commits are therefore rejected, in order to avoid the loss of source code. To deal with this, it's advised that you use **git fetch** to retrieve the updated remote branch, and then run **git merge [remote\_name]/[branch\_name]** to merge the changes from the remote branch. Finally, run **git push** to upload your changes to the remote branch.

The **git rebase** command can also be used to integrate changes from the remote branch.

Git pull

The **git pull** command is an alternative to using the **git fetch** mechanism to retrieve and integrate changes in the remote repository.

The **git pull** command, in its default mode, runs a combination of **git fetch** and **git merge**. The **git pull** utility may use a **rebase** mechanism, if it's specified as the synchronization mechanism in the place of a **merge**

The **git pull** syntax is as follows:

**git pull [options] [remote\_name] [branch\_name or refspec]**

**git pull [remote]**

The preceding code retrieves changes from the remote branch that's been configured as the remote-tracking branch for the current branch and merges the changes into the local branch. This can be done with **git pull origin.**

**git pull**

This is similar to **git pull origin.**

You can fetch the remote branch **develop** and merge it into the current branch with **git pull origin develop.**

--no-commit

This option instructs **git merge** to integrate the changes from the remote repository, and to not perform a commit; for example, **git merge --no-commit origin master**

#### Note

In the event that the commits being merged are resolved to be fast-forward, the **--no-ff** option should be used to override the default behavior, which is to update the branch pointer without creating a merge commit; for example, **git merge --no-commit --no-ff origin master.**

**--no-ff**

This option creates a merge commit, including in a scenario where commits resolve as being fast-forward; for example, **git merge --no-ff origin master.**

**--ff**

This is the opposite of **--no-ff**. It updates the branch pointer of the current branch to the tip of the branch with the incoming changes commits resolved as being fast-forward; for example, **git merge --ff origin master.**

**--edit**

The **git pull** command fetches and merges the changes, and launches an editor to allow for the editing of the automatically generated commit message; for example, **git merge --edit --no-ff origin master**

**--no-edit**

This instructs the fetch and merge process of **git pull** to accept the auto-generated commit message; for example: **git merge --no-edit --no-ff origin master--strategy=[strategy].**

This option specifies the merge strategy to be used for the **git merge** step of **git pull**.

The strategies used by git are **resolve**, **recursive**, **octopus**, **ours**, and **subtree**.

**--strategy-option=[option]**

This sets a strategy option that's specific to the strategy specified by the **--strategy** command option. The options for the **recursive** strategy include **ours**, **theirs**, **no-renames**, **ignore-all-space**, **ignore-space-at-eol**, and **ignore-cr-at-eol**.

# CHAPTER 6 Tracking Other Repositories

When you check out a branch that doesn’t yet exist, but there is a remote-tracking branch by that name, Git automatically creates it and sets its upstream to be that tracking branch, so that subsequent push/pull operations will synchronize your local version of this branch with the remote’s version. For example, when you first clone a repository, Git checks out the remote’s HEAD branch, so this happens right away for one branch

Pushing

If you have added a local branch of your own and want to start sharing it with others, use the -u option to have Git add your branch to the remote, and set up tracking for your local branch in the usual way, for example:

**$ git push -u origin new-branch**

After this initial setup you can use just git push on this branch, with no options or arguments, to push to the same remote.

Push Defaults

There are several approaches Git can use when given no specific remote and ref to push (just plain git push, as opposed to git push *remote branch*):

* matching

Push all branches with matching local and remote names

* upstream

Push the current branch to its upstream (making push and pull symmetric operations)

* simple

Like upstream, but check that the branch names are the same (to guard against mistaken upstream settings)

* current

Push the current branch to a remote one with the same name (creating it if necessary)

* nothing

Push nothing (require explicit arguments)

The default as of this writing is matching, but with Git 2.0, this will change to simple, which is more conservative and avoids easy accidental pushing of changes on other branches that are not yet ready to be published.

The command git remote show *remote* gives a useful summary of the status of your repository in relation to a remote

**$ git remote show origin**

Note that unlike most informational commands, this actually examines the remote repository, so it will run *ssh* or otherwise use the network if necessary. You can use the –n switch to avoid this; Git will skip those operations that require contacting the remote and note them as such in the output.

git branch -vv gives a more compact summary without contacting the remote (and thus reflects the state as of the last fetch or pull; remember that the remote might have changed in the meantime

# Working

### Reversing Commits

Git provides multiple approaches to reversing changes that have been introduced to a file.

The **git revert** command uses the following syntax:

**git revert [options] [commit(s)]**

The options supported by **git revert** are as follows:

**--edit**

This option provides support for editing the commit message for a given reversal; for example, **git revert --edit [commit hash].**

**--no-edit**

The **--no-edit** option overrides the **--edit** option. It performs the revert with the default auto-generated commit message.

**--no-commit**

With this option, the **git revert** command makes changes to the index and the working tree, in order to reverse the modifications made by the given commit. It does this without creating a commit for the reversal.

**--mainline [parent-number]**

Two sets of changes exist in a merge commit, referenced by each of the two parents that constitute a merge commit. This option dictates which of the two sets should be used in the reversal process. We will explore this further in the cherry-pick section.

### Other Possible Uses

**git revert [least\_recent\_commit\_hash]..[most\_recent\_commit\_hash]**

**git reset**

The preceding code shows how the **git revert** command is a forward-moving **undo** operation that provides an effective means of undoing changes.

The **git reset** command is used to roll back a file, directory, or repository at large to a given point in its history. This entails updating the **index**, **working tree,** and **commit history** of a repository where applicable, based on the options passed in the command.

**Tags**

A **tag** is a pointer to a specific commit. The tag object contains the hash of the tagged object, the type of the object that the tag was created for, the tag name, and the the author details, such as the author name, date, and message:

**git tag -a v1.4 ab30a24 -m "This is demo commit"**

The **git reset** syntax follows the following form:

**git reset [tree-ish] [path(s)]**

**git reset [mode] [commit]**

To demonstrate how **git reset works**, we will explore the aforementioned variants and how each works to examine and establish how each affects a repository:

**git reset [tree-ish] [path(s)]**

In Git, **tree-ish** refers to identifiers that reference a tree object, that is, a directory or subdirectory at a given point in a repository's history.

This variant of **git reset** resets the **[path(s)]** index to its state at **[tree-ish]**. The files and directories in the working tree are not affected:

For example. **git reset master:./test.txt test.txt**

**git reset 5d2045c test.txt**

**git reset HEAD:README README**

**git reset :/Adding the word 'boat' -- test.txt**

The text **Adding the word 'boat'** represents a commit message.

**--patch**

The **--patch** option allows you to choose portions of the differences between the index and **[tree-ish]** to reset to their respective states at **[tree-ish]**; for example, **git reset --patch 5d2045c test.txt.**

**git reset [mode] [commit]**

This variant resets the head of the current branch to the state at **[commit]**. Additionally, the index and working tree are reset, depending on the mode used to conduct the reset.

The supported modes are as follows:

**--soft**

This resets the head to **[commit]**. The index and the working tree are not altered by this option.

**--hard**

The **--hard** mode resets the head to **[commit]**. The index and working tree are reset to **[commit]**. Changes introduced to the working tree are discarded.

**--mixed**

This mode resets the head to **[commit]**. Additionally, it resets, the index to **[commit]**. The changes made to the working tree are not altered or discarded.

The commit object stores a snapshot of the directories and files that constitute a repository at a given point in time. In addition, the commit specifies auxiliary information, which includes the parent commit of the created commit, the author, the committer date and time, and the commit message:

A branch is therefore a pointer to a snapshot of the repository. This pointer refers to the commit at the tip of the branch. These tips are the commit hashes stored in **.git/refs/heads/. HEAD** is the pointer that references the commit at the tip of the current branch. This commit is imperative because it's based on the fact that git is able to navigate the history of a repository with the help of the parent-child association between commits. The creation of a branch, in turn, creates a pointer and the head, which bears a branch's name. Navigating from one branch to another updates the **HEAD** to refer to the tip of the branch you switch to – or in Git terms, check out to.

**Incorporating Changes with Stashing**

In the book of development work, emerging requests are a typical occurrence, including in scenarios where you are attending a planned work stream over a specific period of time. The book of action in this scenario, normally, is to put aside what you're working on and attend to this request, be it an emergency or not.

How does Git enable you to "put away" what you're working on without losing the progress you'd achieved on a certain task? Ask no more.

In comes **git stash**. The **git stash** command temporarily moves staged, unstaged, or untracked modifications made to a repository, to and from the index and working directory.

To achieve this with **git stash**, use the following subcommands:

**git stash push -m [message] or git stash push —message [message]**

This saves modifications to a stash list and reverts the index and the working tree to the state reflected by HEAD.

The **—keep-index** option retains changes made to the index. This means that the modifications in the index are not reverted.

The **—include-untracked** option includes untracked files in the stash entry made to the stash list.

The **—all** option stashes ignored files in the stash entry made to the stash list.

#### Note

**git stash save** was deprecated in favor of this command.

**git stash list**

This command lists the entries in the stash list. These are all of the created stashes:

**Git stash show [stash\_id]**

This command displays the changes introduced by the stash identified by **[stash\_id].**

**git stash apply [stash\_id]**

This updates the working directory with the changes stored in **[stash\_id].**

**git stash pop [stash\_id]**

This updates the working directory with the stash **[stash\_id]** and removes it from the stash list.

**git stash drop [stash\_id]**

This removes the specified stash from the stash list.

**git stash clear**

This removes all stashes from the stash list.

**git stash branch [branchname] [stash\_id]**

#### Note

The stash list is available from every branch

**Merging**

A merge takes one of two modes, namely:

1. **Fast-forward merge**
2. **Three-way merge**

Merging is achieved by using the **git merge** command. This command uses the following syntax:

**git merge [options] [branch\_name]**

**--no-commit**

This option merges changes into the current branch. However, the command does not create a merge commit in order to leave room for evaluating the result of the merge.

**--edit**

This option launches the editor to allow for editing of the generated commit message.

**--no-edit**

This conducts the merge using the message generated by the command.

**--no-ff**

This option creates a merge commit in all merge scenarios, including when the merge resolves to a fast-forward merge.

**--squash**

This option instructs **git merge** to update the index and working directory to reflect the incoming changes without creating a commit. Using this option enables you to create a commit as part of the current branch, thus referencing the incoming changes. With this option, the **HEAD** is not changed and the **MERGE\_HEAD** ref is not recorded. As a result, the subsequent commit is not a merge commit.

**--strategy=[strategy]**

This specifies the merge strategy to be used for the merge.

The supported [strategy] includes

**--strategy-option=[strategy\_option]**

This sets the option that's specific to the provided strategy.

### Cherry-Pick

The **git cherry-pick** command takes a commit from one branch and applies the specified commit to another branch.

**git cherry-pick** is useful when you wish to check the effects of certain changes that have been introduced to a branch you're working on.

The syntax of the **git cherry-pick** command is a follows:

**git cherry-pick [options] [commit].**

**git cherry-pick** supports options that dictate how the introduced commits are handled. This includes the following:

**-x:** This option adds a standardized message to the commit of the form **"cherry picked from commit ..."** to specify the commit that introduces the incoming change.

**--edit:** This allows you to edit the commit message for the incoming changes.

**--no-commit:** You may wish to integrate changes from a specific commit without creating a corresponding commit. The **--no-commit** command integrates changes from a commit without creating a commit.

**--mainline [parent\_number]:** Since a parent commit possesses two parents, running a cherry-pick against a merge commit requires that a parent is specified. The given parent is compared to the merge tree of the merge commit and the resulting difference is introduced into the branch where **git cherry-pick** is invoked from.

**git cherry-pick -m 1 merge\_commit**

By specifying -m 1, we choose parent-1.

**Merge Conflicts**

A **merge conflict** is a term that depicts an issue whereby modifications made in separate branches can't be amalgamated into one unit of change or modification.

A merge conflict will occur when:

1. Modifications are made on the same line of a file.
2. Changes are made to a file on one branch and the same file is deleted on another branch.

Merge conflict resolution encompasses picking which of the differing sets of changes should be used in a merge process.

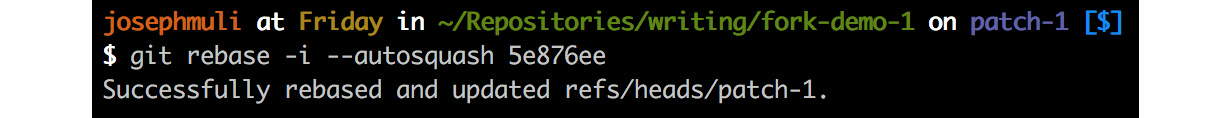
**Fixup and Squash Commits**

### Utilizing the Autosquash Feature

To **fixup** the previous commits using the autosquash facility in GitHub.

Instead of a normal commit, include a **--fixup** argument and a commit hash to the command as follows: **git commit --fixup dd4b896**

1. To clean this up, rebase and autosquash the commit, as shown in the following screenshot:



###### Figure 5.42: Rebasing and autosquashing the commit

#### Note

Be sure to select the commit hash that's just before the commit you want to autosquas

If you are running VIM as I am, select **I** on your keyboard to use the **Insert** option and edit the command. To drop and escape and select **:wq** without the quotes to write and quit, as shown in the following screenshot:

**Chapter 9: Submodules**

**Section 9.1: Cloning a Git repository having submodules**

When you clone a repository that uses submodules, you'll need to initialize and update them.

$ **git clone** --recursive https:**//**github.com**/**username**/**repo.git

This will clone the referenced submodules and place them in the appropriate folders (including submodules within submodules). This is equivalent to running **git submodule** update --init --recursive immediately after the clone is finished.

A submodule references a specific commit in another repository. To check out the exact state that is referenced for all submodules, run

**git submodule** update –recursive

To checkout the latest state of a specific submodule, you can use :

**git submodule** update --remote **<**submodule\_directory**>**

**Section 9.3: Adding a submodule**

You can include another Git repository as a folder within your project, tracked by Git:

$ **git submodule** add [https:**//**github.com**/**jquery**/**jquery.git](https://github.com/jquery/jquery.git)

To update that submodule to the latest commit of a branch of the submodule remote repo.

**git submodule** update --remote –recursive

**Section 9.5: Moving a submodule**

Run:

$ **git mv** /path/to/module *new/path/to/module*

**Section 9.6: Removing a submodule**

You can remove a submodule (e.g. the\_submodule) by calling:

$ **git submodule** deinit the\_submodule

$ **git rm** the\_submodule

**git submodule** deinit the\_submodule deletes the\_submodules' entry from .git/config. This excludes

the\_submodule from **git submodule** update, **git submodule sync** and **git submodule** foreach calls and

deletes its local content (source). Also, this will not be shown as change in your parent repository. **git**

**submodule** init and **git submodule** update will restore the submodule, again without commitable changes

in your parent repository.

**git rm** the\_submodule will remove the submodule from the work tree. The files will be gone as well as the

submodules' entry in the .gitmodules file (source). If only **git rm** the\_submodule (without prior **git**

**submodule** deinit the\_submodule is run, however, the submodules' entry in your .git/config file will remain.

### Submodules

### Submodules require a **.gitmodules** file and dependencies to be installed through the **git submodule** command.

Git submodules, often referred to as gitmodules, enable the **separation of concern** toward project dependencies. In other words, they help organize code based on a **single responsibility principle**.

When handling submodules for the first time, be sure to do the following:

* Add submodules through the **git submodule add <repository url>** command.
* Initialize the submodules through the **git submodule init** command.

**git submodule update** will fetch submodules that are not present in your project path, as shown in the following screenshot:

**git clone --recurse-submodules: https:github.com/<username>/abacus-scripts** clones the repository and submodules defined in a **.gitmodules** file.

**git blame**

**git blame** helps identify and present occurrences from a revision that has modified a block of code. Usually, this is done line by line. This means that through the command, we can identify who made a change, the commit used, and what line(s) were affected. This also makes Git a very effective audit tool.

**git blame** includes several other options, such as showing the filename in the original commit, whereby the default filename is displayed if a change comes from a different file, with a different name, as shown in the following screenshot:

git blame –f totodo.py

**git bisect**, as described on Git's official documentation, is used to find a commit that has introduced an anomaly through a binary search. This is most helpful when trying to identify a commit to test a failing feature after previous success.

$ git bisect start

$ git bisect bad XXXX

$git bisect good XXXX

**git reflog**

**git reflog** is short for Git **reference logs**. Reflogs keep track of changes to **HEAD** over a defined period of time. These changes can be best defined as events, as they are saturated, which is basically all activities, that is, checking out branches, rebase events, and branch updates from remote URLs.

To get the reflog on a specific branch, pass the branch name as an argument, as shown in the following screenshot:

A computer screen with text on it

Description automatically generated

## Housekeeping

In this subtopic, we will highlight a number of best practices that can be applied toward maintaining a clean and operable repository through the following commands:

* **git clean**
* **git gc**
* **git prune**

**git clean**

**git clean** recursively removes untracked files from a working tree. This emphasizes that any file that is not staged to be tracked or reset is rid of, maintaining a versioned only directory. Normally, **git clean** purges files through a list defined from a **.gitignore** file, but in special cases, these rules can be ignored and any untracked file is cleared.

*The* ***-n option*** *will enable a dry run accompanying* ***git clean***

To remove the file, the **-f** and **-i** options can be utilized

*The* ***-i*** *option presents an interactive mode that presents an interactive session*

**git gc**

**git gc** is responsible for garbage collection on targeted repositories. This process handles the deletion of staged and committed objects from unreachable branches, particularly those holding a reflog. Depending on the repository activity, **git gc** helps optimize disk space and maintains a decent repository.

To verify whether any housekeeping is required, **git gc** appends the **--auto** option and if not necessary, no output shall be displayed, as demonstrated in the following screenshot:

To proceed without necessarily requiring housekeeping, run the following command, and optionally append **--force**, as shown in the following screenshot:

**git prune**

The **git prune** command, similar to **git gc**, gets rid of all unreachable objects, that is, basically objects without references, such as ones from deleted branches. According to the documentation, it's best to go for **git gc,** which calls **git prune**, killing two birds with one stone.

**Removing Merged Local and Remote Branches**

Manual deletion locally.

Before we start taking out branches, we need to be aware of merged and not unmerged branches. The following commands help us identify this:

A screen shot of a computer

Description automatically generated

To delete the remote branch reference, run the following command:

A screen shot of a computer program

Description automatically generated

To delete a branch that is not merged, run the command with a **-D** option, as shown in the following screenshot:

A black screen with blue text

Description automatically generated

**…or create a new repository on the command line**

echo "# test3" >> README.md

git init

git add README.md

git commit -m "first commit"

git remote add origin https://github.com/pvtreservoir/test3.git

git push -u origin master

**…or push an existing repository from the command line**

git remote add origin https://github.com/pvtreservoir/test3.git

git push -u origin master

**git log -5 --pretty=format:"%H %an %ae %n %s %n %b"**

**git rebase -i HEAD~4**

bgen/static void NumQuery()

{

var numbers = new int[] { 1, 4, 9, 16, 25, 36 };

var evenNumbers = from p in numbers

where (p % 2) == 0

select p;

}