



# GIT

**Distributed Version Control System**



CHANDRA SHEKHAR REDDY

## INTRODUCTION

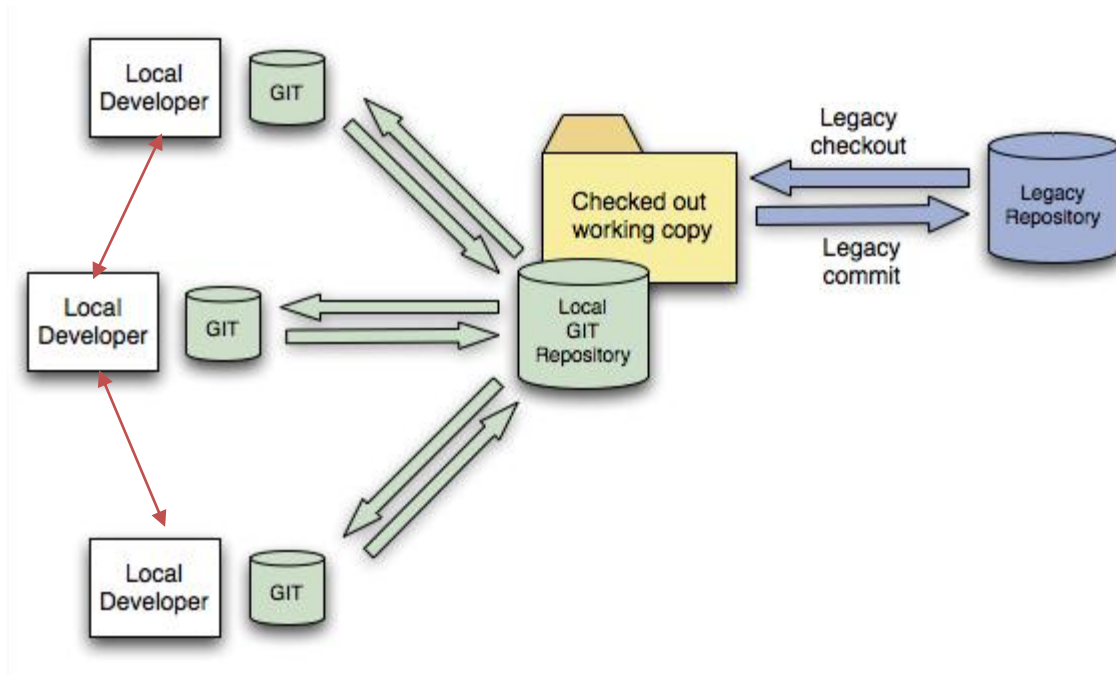
Git is a version control system that allows you to track the history of a collection of files and includes the functionality to revert the collection of files to another version. Each version captures a snapshot of the files at a certain point in time. The collection of files is usually *source code* for a programming language but a typical version control system can put any type of file under version control. The collection of files and their complete history are stored in a *repository*.

GIT stores the file content in BLOBs (binary large objects). The folders are represented as trees. Each tree contains other trees (subfolders) and BLOBs along with a simple text file which consists of the mode, type, name and SHA (Secure Hash Algorithm) of each blob and subtree entry. During repository transfers, even if there are several files with the same content and different names, the GIT software will transfer the BLOB once and then expand it to the different files.

The history of your project is stored in a commit object. Every time you make a modification you have to commit it. The commit file keeps the author, committer, comment and any parent commits that directly precede it.

### Distributed Version Control System:

Distributed revision control takes a peer-to-peer approach to version control, as opposed to the client-server approach of centralized systems. Rather than a single, central repository on which clients synchronize, each peer's working copy of the code base is a complete repository. Distributed revision control synchronizes repositories by exchanging patches (sets of changes) from peer to peer.



### **Important differences from a centralized system:**

- No canonical, reference copy of the code base exists by default; only working copies.
- Common operations (such as commits, viewing history, and reverting changes) are fast, because there is no need to communicate with a central server.
- Communication is only necessary when sharing changes among other peers.
- Each working copy effectively functions as a remote backup of the codebase and of its change-history, protecting against data loss.

Other differences include:

- Multiple "central" repositories.
- Code from disparate repositories are merged based on a web of trust, i.e., historical merit or quality of changes.
- Numerous different development models are possible, such as development / release branches or a Commander / Lieutenant model, allowing for efficient delegation of topical developments in very large projects. Lieutenants are project members who have the power to dynamically decide which branches to merge.
- Network is *not* involved for common operations.
- A separate set of "sync" operations are available for committing or receiving changes with remote repositories.

### **Advantages of Git:**

- Allows users to work productively when not connected to a network.
- Makes most operations much faster.
- Allows participation in projects without requiring permissions from project authorities, and thus arguably better fosters culture of meritocracy instead of requiring "committer" status.
- Allows private work, so users can use their changes even for early drafts they do not want to publish.
- Avoids relying on one physical machine as a single point of failure.
- Permits centralized control of the "release version" of the project
- On FLOSS software projects it is much easier to create a project fork from a project that is stalled because of leadership conflicts or design disagreements.

## GIT TERMINOLOGIES

### Repository:

Contains all the history... like commits, branches, tags and your source code. Repository can be remote which is a central storage for repository is. Local repository would be what the user/developer works with (make changes to). Remote repository is a centralized repository for sharing projects.

### Local repository

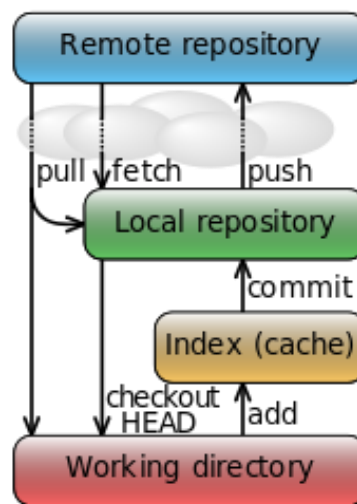
A local repo is a git-project on your computer. If you were to have a Remote repository (GitHub), then the local repo would be a clone (exact copy) of the remote git-project. When you make changes to your local repo like adding code, you will need to update (push) the git-project to remote repo.

### Index (cache) - Staging

When you make changes to your files, you can not commit (snapshot in time) until you "stage" the files. Once you stage the file, you can then commit the changes. What the purpose of this Index (staging area)? It allows you to control what you want to commit. So any code changes that are not complete can be left out of the commit.

### Working directory

A working directory is your git-project, which is your source code with git files. A working directory is a local repo which you can make changes (commit, branches) to it.



Git Data flow

### Remote repository

A centralized Git server to host all your git-projects. A central repository (repo) is not required but it is the best way to distribute (share) your git-projects to different user/developers. Most would call the remote repository (repo) the *Origin* and keeping it up-to-date is highly recommended. In our case, GitHub would be our remote repo. They all work together and it's all one thing. Ok, a couple of important things to know before talking about the rest of the diagram.

One, I said above you do not need a remote repo... that is because you will have the git software on your computer, which means you can use all of the git features like making your project a git-project, committing changes and using branching. The second thing is, the Local repo, Index, and Working directory are all contained in your git-project.

### **Blobs:**

Blob stands for **B**inary **L**arge **O**bject. Each version of file is represented by blob. A blob holds file data but doesn't contain any metadata about file. It is a binary file, in Git database it is named as SHA1 hash of that file. In Git, files are not addressed by name. Everything is content-addressed.

### **Trees:**

Tree is an object which represents a directory. It holds blobs as well as other sub-directories. A tree is a binary file that stores references to blobs and trees, which is also named as the **SHA1** hash of the tree object.

### **Commits:**

Commit holds the current state of the repository. A commit also named by **SHA1** hash. You can consider commit object as a node of the linked list. Every commit object has a pointer to the parent commit object. From given commit you can traverse back by looking at the parent pointer to view the history of the commit. If a commit has multiple parent commits, that means the particular commit is created by merging two branches.

### **Branches:**

Branches are used to create another line of development. By default Git has a master branch which is same as trunk in Subversion. Usually to work on new feature a branch is created. Once feature is completed; it is merged back with master branch and we delete the branch. Every branch is referenced by HEAD, which points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with latest commit.

### **Tags:**

Tag assigns a meaningful name with a specific version in the repository. Tags are very similar to branches, but the difference is tags are immutable. Means tag is a branch which nobody intends to modify. Once tag is created for particular commit, even you create a new commit, it will not be updated. Usually developer creates tags for product releases.

### **Clone:**

Clone operation creates the instance of the repository. Clone operation not only checkouts the working copy but it also mirrors the complete repository. User can perform many operations with this local repository. The only time networking gets involved is when the repository instances are being synchronized.

**Pull:**

Pull operation copy changes from a remote repository instance to local one. The pull operation is used for synchronization between two repository instances. This is same as update operation in Subversion.

**Push:**

Push operation copy changes from a local repository instance to a remote one. This is used to store changes permanently into the Git repository. This is same as commit operation in Subversion.

**Head:**

HEAD is pointer which always points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with latest commit. The heads of the branches are stored in **.git/refs/heads/** directory.

**Revision:**

Revision represents the version of the source code. Revisions in Git are represented by commits. These commits are identified by **SHA1** secure hashes.

**URL:**

Used to tell Git where the remote repository is. There is more than one type of URL

- HTTP: Should be a https (secure) `https://github.com/<user>/<git_project>`
- GIT: Uses git communication `git://github.com/<user>/<git_project>`
- SSH: Uses SSH `git@github.com:<user>/<git_project>`

**Collaborator:**

Collaborators are other GitHub users that you've given write access to one or more of your repositories. Collaborators may fork any private repository you've added them to without their own paid plan. Their forks do not count against your private repository quota.  
*This was taken from github.com*

**Fork:**

Fork is NOT a Git term, its a term born from the community. Forking is similar to branching in that you copy the origin (Master) to your account. The difference is that you are copying the source code from the author (owner) to your account. Branching is copying within the same account.

So when does this happen? Forking usually happens for 2 reasons:

1. You fork it to take the project in a new direction. Obviously you can only do this with open source projects
2. You fork the project to add to the project and request a merge back to the origin/master. You as a independent developer want to contribute to the code.

## START WORKING WITH GIT

### Creating the Git Repository:

The `git init` command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new empty repository. Most of the other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project. Executing `git init` creates a `.git` subdirectory in the project root, which contains all of the necessary metadata for the repository.

### Syntax:

```
[root@localhost opt]# git init  
  
[or]  
  
[root@localhost opt]# git init <directory_name>
```

### Example:

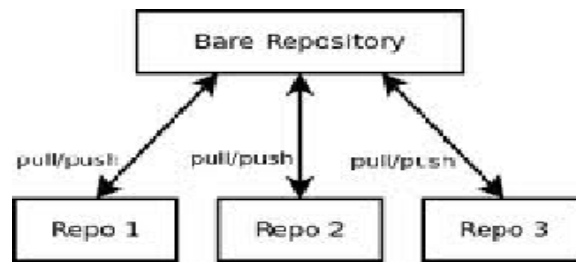
```
[root@localhost opt]# mkdir project_gitrepo  
[root@localhost opt]# cd project_gitrepo  
[root@localhost project_gitrepo]# git init  
  
[or]  
  
[root@localhost opt]# git init project_gitrepo
```

If you use '`git init`' inside any directory then it will create '`.git`' subdirectory in that, and '`git init directory_name`' creates a directory as a repository with '`.git`'.

Shared repositories should always be created with the `--bare` flag conventionally, repositories initialized with the `--bare` flag end in `.git`. For example, the bare version of a repository called `my-project` should be stored in a directory called `project_gitrepo`.

### Bare Repositories

The `--bare` flag creates a repository that doesn't have a working directory, making it impossible to edit files and commit changes in that repository. Central repositories should always be created as bare repositories because pushing branches to a non-bare repository has the potential to overwrite changes. Think of `--bare` as a way to mark a repository as a storage facility, opposed to a development environment. This means that for virtually all Git workflows, the central repository is bare, and developer's local repositories are non-bare.

**Syntax:**

```
[root@localhost opt]# git init --bare <directory_name>
```

**Example:**

```
[root@localhost opt]# git init --bare project_gitrepo
```

**Git clone:**

The git clone command copies an existing Git repository. This is sort of like svn checkout, except the “working copy” is a full-fledged Git repository—it has its own history, manages its own files, and is a completely isolated environment from the original repository.

As a convenience, cloning automatically creates a remote connection called origin pointing back to the original repository. This makes it very easy to interact with a central repository.

Clone the repository located at <repo> onto the local machine. The original repository can be located on the local filesystem or on a remote machine accessible via HTTP or SSH.

If a project has already been set up in a central repository, the git clone command is the most common way for users to obtain a development copy. Like git init, cloning is generally a one-time operation—once a developer has obtained a working copy, all version control operations and collaborations are managed through their local repository.

**Syntax:**

```
[root@localhost opt]# git clone <repository url>
```

**Example:**

```
[root@localhost opt]# git clone ssh://192.168.255.135/opt/repos/project_gitrepo
```

```
[root@localhost opt]# git clone https://192.168.255.135/opt/repos/project_gitrepo
```

```
[root@localhost opt]# cd project_gitrepo
```

```
root@localhost:opt/git_clones
[ root@localhost git_clones ]#
[ root@localhost git_clones ]# git clone ssh://root@192.168.255.135/opt/repos/project_gitrepo
Initialized empty Git repository in /opt/git_clones/project_gitrepo/.git/
root@192.168.255.135's password:
warning: You appear to have cloned an empty repository.
[ root@localhost git_clones ]#
```

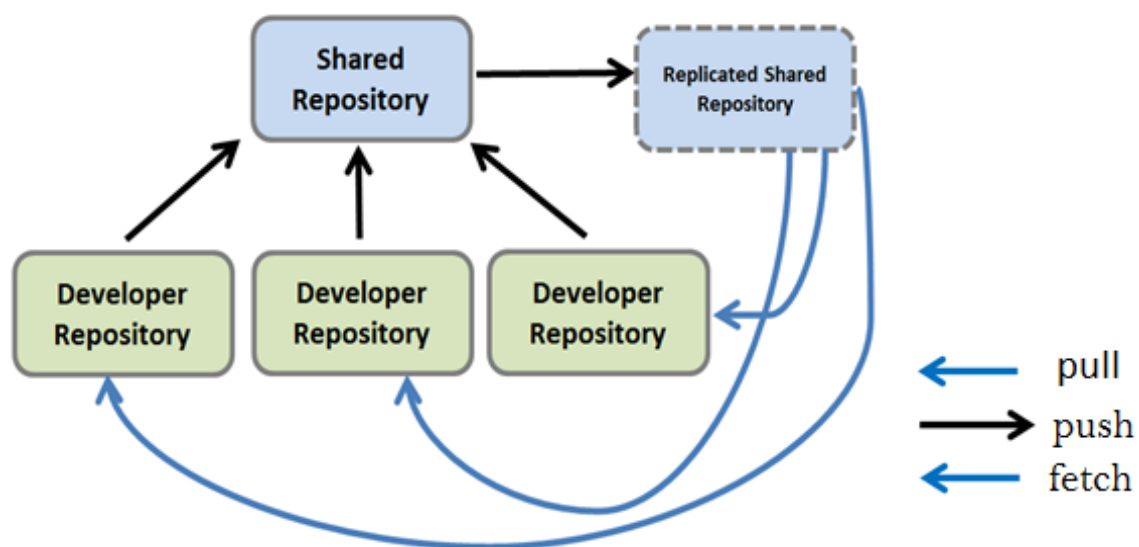


The first command initializes a new Git repository in the `project_gitrepo` folder on your local machine and populates it with the contents of the central repository. Then, you can `cd` into the project and start editing files, committing snapshots, and interacting with other repositories. Also note that the `.git` extension is omitted from the cloned repository. This reflects the non-bare status of the local copy.

### Repo-To-Repo Collaboration:

It's important to understand that Git's idea of a "working copy" is very different from the working copy you get by checking out code from an SVN repository. Unlike SVN, Git makes no distinction between the working copy and the central repository—they are all full-fledged Git repositories.

This makes collaborating with Git fundamentally different than with SVN. Whereas SVN depends on the relationship between the central repository and the working copy, Git's collaboration model is based on repository-to-repository interaction. Instead of checking a working copy into SVN's central repository, you push or pull commits from one repository to another.



### Git config:

The `git config` command lets you configure your Git installation (or an individual repository) from the command line. This command can define everything from user info to preferences to the behavior of a repository. Several common configuration options are listed below.

## Usage:

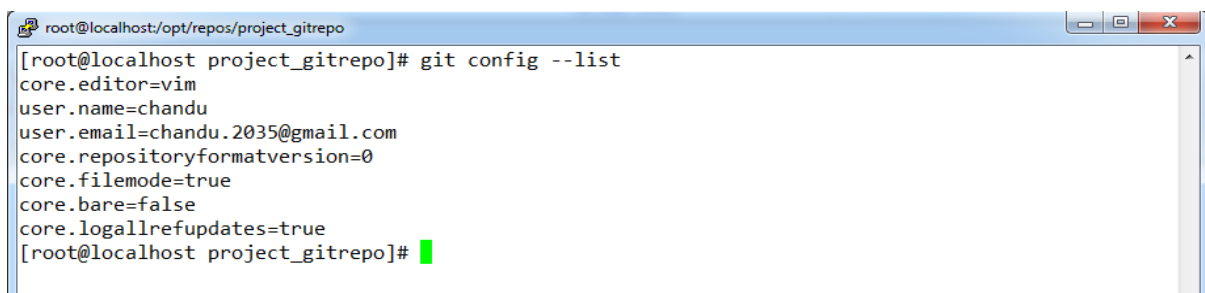
Define the author name to be used for all commits in the current repository. Typically, you want to use the `--global` flag to set configuration options for the current user.

## Syntax:

```
[root@localhost opt]# git config --global user.name <username>
```

```
[root@localhost opt]# git config --global user.email <email address>
```

## Example:

A terminal window titled 'root@localhost:opt/repos/project\_gitrepo' showing the output of the command 'git config --list'. The output lists several configuration options: core.editor=vim, user.name=chandu, user.email=chandu.2035@gmail.com, core.repositoryformatversion=0, core.filemode=true, core.bare=false, and core.logallrefupdates=true. The prompt is '[root@localhost project\_gitrepo]#'.

```
root@localhost:opt/repos/project_gitrepo
[root@localhost project_gitrepo]# git config --list
core.editor=vim
user.name=chandu
user.email=chandu.2035@gmail.com
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
[root@localhost project_gitrepo]#
```

In the similar way you can set all the configurations using the options like alias, core editor etc.....

```
[root@localhost opt]# git config --global --edit
```

The above command will open the global configuration file in a default text editor for manual editing.

A terminal window titled 'root@localhost:opt/git\_clones/project\_gitrepo' showing the contents of the global git configuration file. The file is displayed in a text editor with syntax highlighting. It shows sections for [core], [remote "origin"], and [branch "master"]. The status bar at the bottom indicates the file is '.git/config' with 11 lines and 277 characters.

```
root@localhost:opt/git_clones/project_gitrepo
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    fetch = +refs/heads/*:refs/remotes/origin/*
    url = ssh://root@192.168.255.135/opt/repos/project_gitrepo
[branch "master"]
    remote = origin
    merge = refs/heads/master
~
~
~
~
~
~
".git/config" 11L, 277C
```

All configuration options are stored in plaintext files, so the `git config` command is really just a convenient command-line interface. Typically, you'll only need to configure a Git installation the first time you start working on a new development machine, and for virtually all cases, you'll want to use the `--global` flag.

Git stores configuration options in three separate files, which lets you scope options to individual repositories, users, or the entire system:

<repo>/.git/config – Repository-specific settings.

~/.gitconfig – User-specific settings. This is where options set with the --global flag are stored.

\$(prefix)/etc/gitconfig – System-wide settings.

When options in these files conflict, local settings override user settings, which override system-wide. If you open any of these files, you'll see something like the following:

# Add some SVN-like aliases

```
git config --global alias.st status
```

```
git config --global alias.co checkout
```

```
git config --global alias.br branch
```

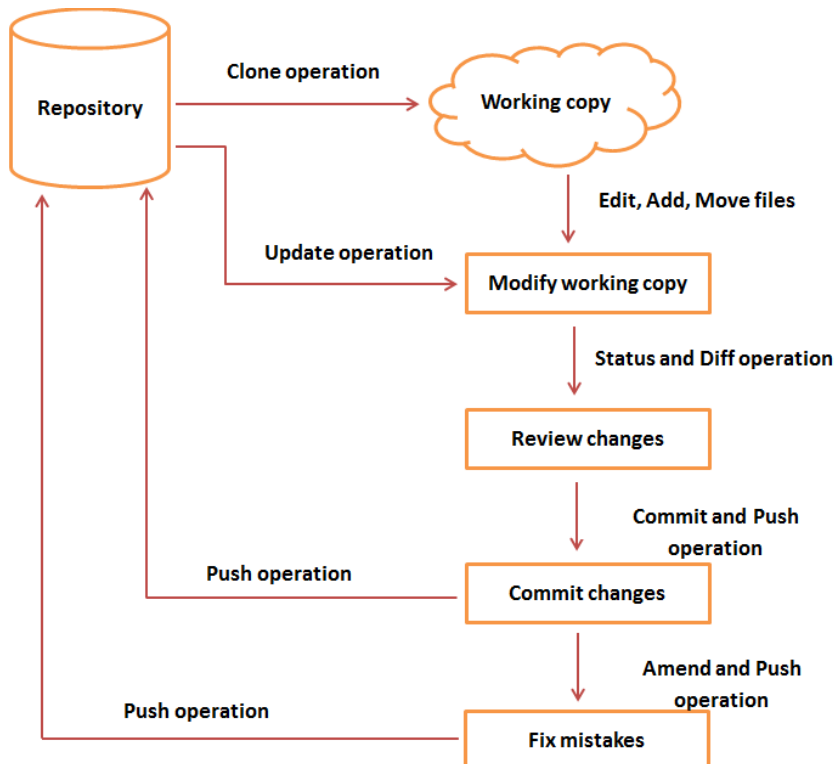
```
git config --global alias.up rebase
```

```
git config --global alias.ci commit
```

```
[user]
name = chandu
email = chandu.2035@gmail.com
[alias]
st = status
co = checkout
br = branch
up = rebase
ci = commit
[core]
editor = vim
```

## GIT LIFE CYCLE

### Basic Architecture with Lifecycle:



### The Staging Area:

The staging area is one of Git's more unique features, and it can take some time to wrap your head around it if you're coming from an SVN (or even a Mercurial) background. It helps to think of it as a buffer between the working directory and the project history.

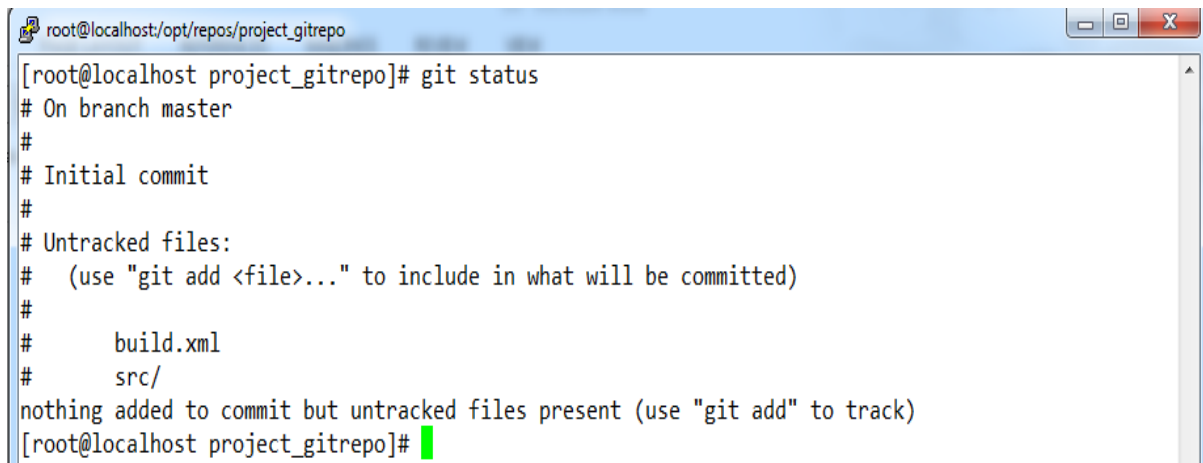
Instead of committing all of the changes you've made since the last commit, the stage lets you group related changes into highly focused snapshots before actually committing it to the project history. This means you can make all sorts of edits to unrelated files, then go back and split them up into logical commits by adding related changes to the stage and commit them piece-by-piece. It's important to create atomic commits so that it's easy to track down bugs and revert changes with minimal impact on the rest of the project.

### Git status:

Git status will tell you precisely that: the status of the files in your Git directory. To illustrate, let's move through an example. First, I make a new directory named `git_test`, and perform a `git init`. It's good practice to `git init` as soon as you begin a project, since, as we'll see, git will record all the changes we make. In the Terminal, I add a few more files and folders, and run `git status` to see what Git knows:

Git status gives us a number of details: We know what branch we're on (master), what's staged in our initial commit (nothing), and which files are untracked (everything I've added). Untracked files are those that Git knows exists, but weren't included in your last snapshot; Git won't begin including these files (and tracking the changes within them) until we stage them to be committed.

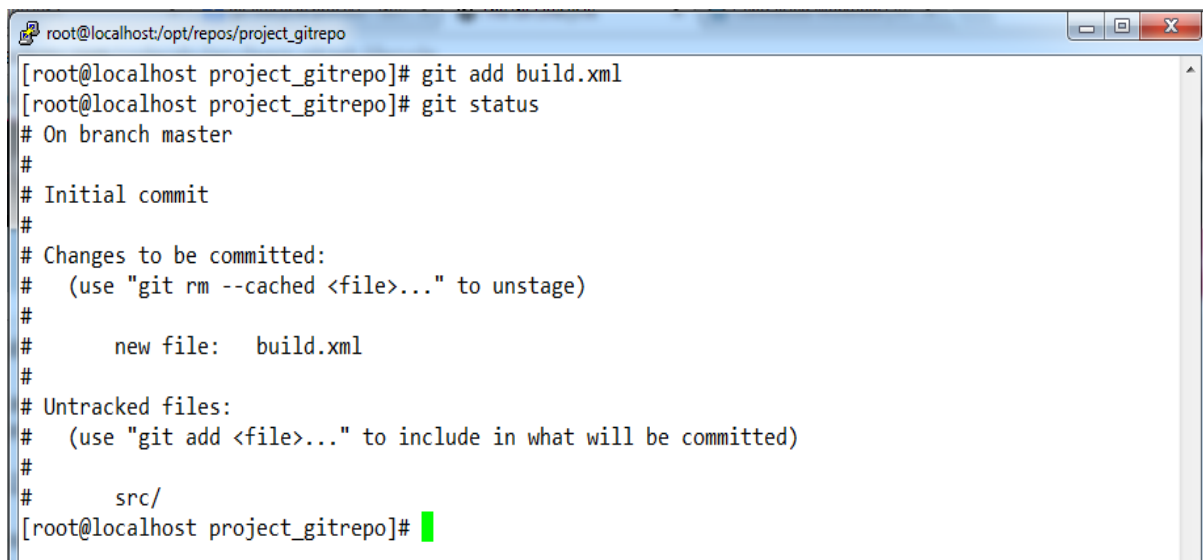
### Example:

A terminal window titled 'root@localhost/opt/repos/project\_gitrepo' showing the output of the 'git status' command. The output indicates the current branch is master, it's an initial commit, and there are untracked files: 'build.xml' and 'src/'. It suggests using 'git add' to track these files.

```
root@localhost/opt/repos/project_gitrepo
[root@localhost project_gitrepo]# git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       build.xml
#       src/
nothing added to commit but untracked files present (use "git add" to track)
[root@localhost project_gitrepo]#
```

### Staging Files:

Staged files have not yet been committed--their changes not yet recorded--but they are "on deck" so to speak for your next commit. To stage a file, you use the git add command:

A terminal window titled 'root@localhost/opt/repos/project\_gitrepo' showing the output of 'git add build.xml' followed by 'git status'. The output shows that 'build.xml' is now staged for commit, while 'src/' remains untracked.

```
root@localhost/opt/repos/project_gitrepo
[root@localhost project_gitrepo]# git add build.xml
[root@localhost project_gitrepo]# git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   build.xml
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       src/
[root@localhost project_gitrepo]#
```

Here build.xml file added but not src directory, so if we need to add all contents that are available in repository.

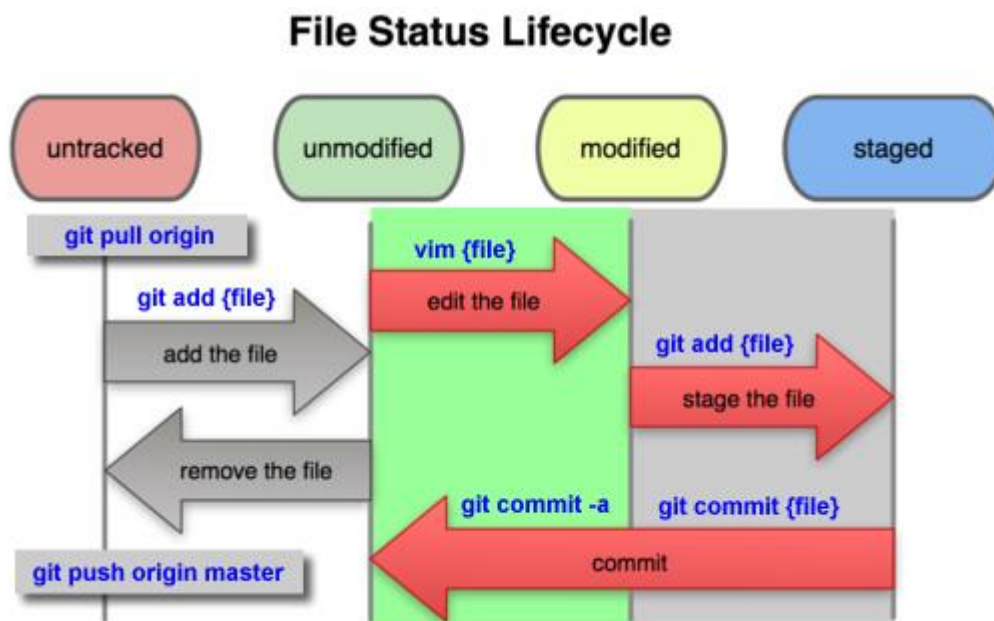
```
[root@localhost project_gitrepo]# git add *
```

```
root@localhost:/opt/repos/project_gitrepo
[root@localhost project_gitrepo]# git add *
[root@localhost project_gitrepo]# git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   build.xml
#       new file:   src/Hello.java
#       new file:   src/hello.jsp
#       new file:   src/index.html
#       new file:   src/web.xml
#
[root@localhost project_gitrepo]#
```

All contents of the repository added to the staging area.

```
[root@localhost project_gitrepo]# git add -p
```

Begin an interactive staging session that lets you choose portions of a file to add to the next commit. This will present you with a chunk of changes and prompt you for a command. Use `y` to stage the chunk, `n` to ignore the chunk, `s` to split it into smaller chunks, `e` to manually edit the chunk, and `q` to exit.



### Git commit:

The `git commit` command commits the staged snapshot to the project history. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to. Along with `git add`, this is one of the most important Git commands.

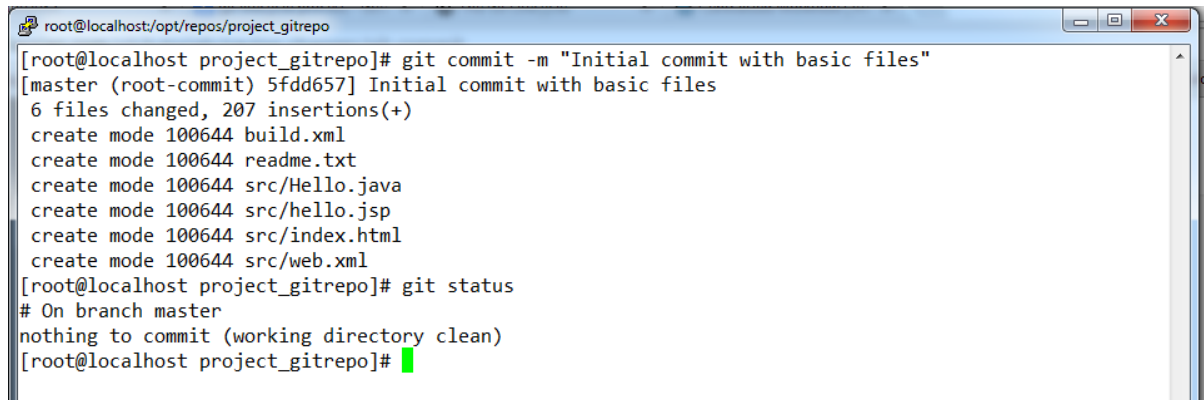
While they share the same name, this command is nothing like `svn commit`. Snapshots are committed to the local repository, and this requires absolutely no interaction with other Git repositories.

## Syntax:

```
[root@localhost project_gitrepo]# git commit -m "commit message"
```

## Example:

```
[root@localhost project_gitrepo]# git commit -m "Initial commit with basic files"
```

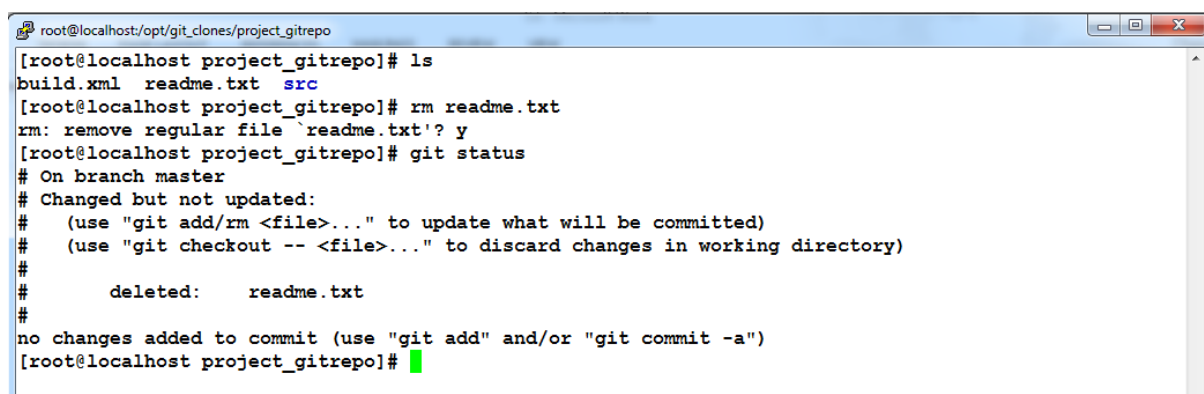
A terminal window titled 'root@localhost/opt/repos/project\_gitrepo' showing the execution of 'git commit -m "Initial commit with basic files"'. The output shows the commit was successful on the master branch, with 6 files changed and 207 insertions. The files listed are build.xml, readme.txt, src/Hello.java, src/hello.jsp, src/index.html, and src/web.xml. A subsequent 'git status' command shows the working directory is clean.

```
root@localhost/opt/repos/project_gitrepo
[root@localhost project_gitrepo]# git commit -m "Initial commit with basic files"
[master (root-commit) 5fdd657] Initial commit with basic files
6 files changed, 207 insertions(+)
create mode 100644 build.xml
create mode 100644 readme.txt
create mode 100644 src/Hello.java
create mode 100644 src/hello.jsp
create mode 100644 src/index.html
create mode 100644 src/web.xml
[root@localhost project_gitrepo]# git status
# On branch master
nothing to commit (working directory clean)
[root@localhost project_gitrepo]#
```

Commit a snapshot of all changes in the working directory. This only includes modifications to tracked files (those that have been added with git add at some point in their history).

Snapshots are always committed to the local repository. This is fundamentally different from SVN, wherein the working copy is committed to the central repository. In contrast, Git doesn't force you to interact with the central repository until you're ready. Just as the staging area is a buffer between the working directory and the project history, each developer's local repository is a buffer between their contributions and the central repository.

This changes the basic development model for Git users. Instead of making a change and committing it directly to the central repo, Git developers have the opportunity to accumulate commits in their local repo. This has many advantages over SVN-style collaboration: it makes it easier to split up a feature into atomic commits, keep related commits grouped together, and clean up local history before publishing it to the central repository. It also lets developers work in an isolated environment, deferring integration until they're at a convenient break point.

A terminal window titled 'root@localhost/opt/git\_clones/project\_gitrepo' showing the removal of 'readme.txt' and a subsequent 'git status' check. The status shows the file as deleted but not staged for commit.

```
root@localhost/opt/git_clones/project_gitrepo
[root@localhost project_gitrepo]# ls
build.xml  readme.txt  src
[root@localhost project_gitrepo]# rm readme.txt
rm: remove regular file `readme.txt'? y
[root@localhost project_gitrepo]# git status
# On branch master
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
[root@localhost project_gitrepo]#
```

## VIEWING THE COMMIT HISTORY

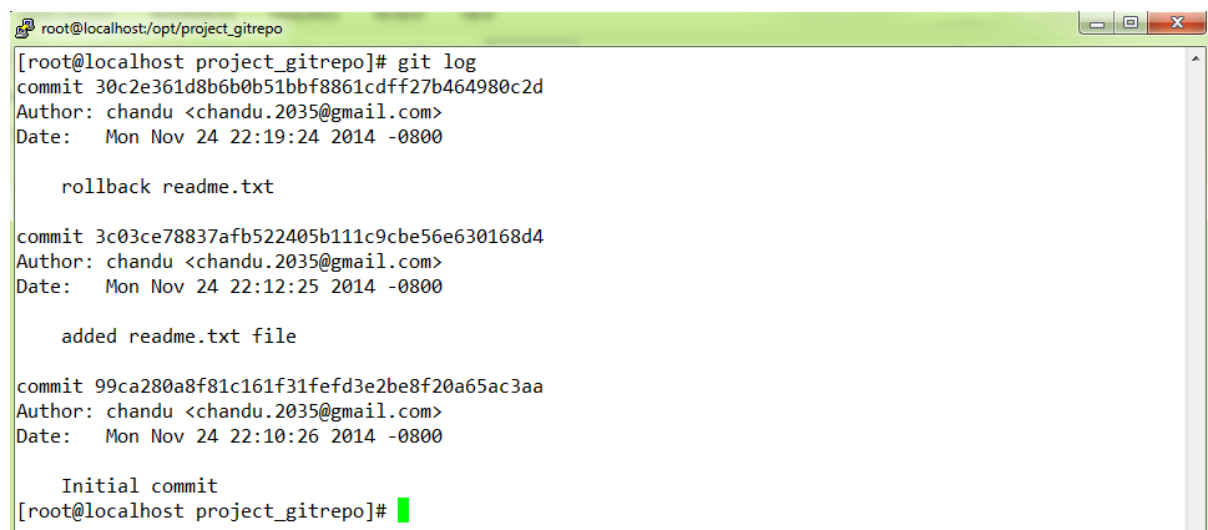
### Git log

Git log is used to show commit messages and also it will provide the other details like commit SHA1 id, author, and timestamp.

### Syntax:

```
[root@localhost project_gitrepo]# git log
```

### Example:



```
root@localhost:opt/project_gitrepo
[root@localhost project_gitrepo]# git log
commit 30c2e361d8b6b0b51bbf8861cdff27b464980c2d
Author: chandu <chandu.2035@gmail.com>
Date:   Mon Nov 24 22:19:24 2014 -0800

    rollback readme.txt

commit 3c03ce78837afb522405b111c9cbe56e630168d4
Author: chandu <chandu.2035@gmail.com>
Date:   Mon Nov 24 22:12:25 2014 -0800

    added readme.txt file

commit 99ca280a8f81c161f31fef3e2be8f20a65ac3aa
Author: chandu <chandu.2035@gmail.com>
Date:   Mon Nov 24 22:10:26 2014 -0800

    Initial commit
[root@localhost project_gitrepo]#
```

### Git log options:

**\$ git log -p:** which shows the difference introduced in each commit. Along with this option you can use numbers to limits the output depends on the number you have specified.

**Example: git log -p -2**

**\$ git log --oneline:** It will display only commit messages with abbreviated hash.

**\$ git log --stat:** It prints each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.

### Git log options:

Option	Description
<b>-p</b>	Show the patch introduced with each commit.
<b>--stat</b>	Show statistics for files modified in each commit.
<b>--shortstat</b>	Display only the changed/insertions/deletions line from the --stat command.
<b>--name-only</b>	Show the list of files modified after the commit information.
<b>--name-status</b>	Show the list of files affected with added/modified/deleted information as well.



<b>--abbrev-commit</b>	Show only the first few characters of the SHA-1 checksum instead of all 40.
<b>--relative-date</b>	Display the date in a relative format (for example, "2 weeks ago") instead of using the full date format.
<b>--graph</b>	Display an ASCII graph of the branch and merge history beside the log output.
<b>--pretty</b>	Show commits in an alternate format. Options include oneline, short, full, fuller, and format (where you specify your own format).

### Example:

```

root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git log --pretty=oneline
69ffcd0067c6eec81a7d6023becef3e1680c60d9 modified index.html
691f9bcb19c1509b7438fb81b01cdb55e7dc308 modified index.html
830e289b4b7d02dccb1aef46baf76e38588fd6ef Initial commit with source code
[root@localhost project_gitrepo]#

```

### Options to limit the output of git log:

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

### Example:

```

root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git log --pretty="%h - %s" --author=root
691f9bc - modified index.html
830e289 - Initial commit with source code
[root@localhost project_gitrepo]# git log --pretty="%h - %s" --author=chandu
69ffcd0 - modified index.html
[root@localhost project_gitrepo]#

```

### \$ git log --pretty=format:

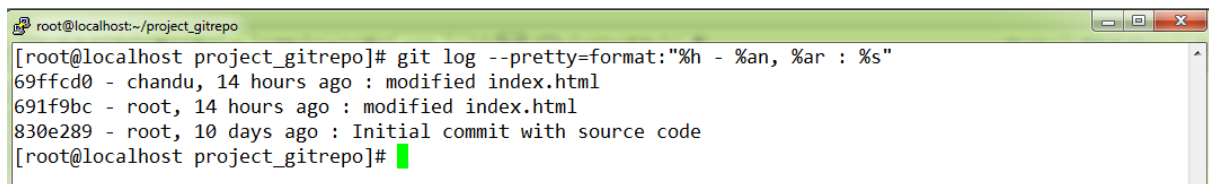
Which allows you to specify your own log output format. This is especially useful when you're generating output for machine parsing – because you specify the format explicitly, you know it won't change with updates to Git.

#### Option Description of Output:

- %H Commit hash
- %h Abbreviated commit hash

%T	Tree hash
%t	Abbreviated tree hash
%P	Parent hashes
%p	Abbreviated parent hashes
%an	Author name
%ae	Author e-mail
%ad	Author date (format respects the <code>--date=</code> option)
%ar	Author date, relative
%cn	Committer name
%ce	Committer email
%cd	Committer date
%cr	Committer date, relative
%s	Subject

### Example:



```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git log --pretty=format:"%h - %an, %ar : %s"
69ffcd0 - chandu, 14 hours ago : modified index.html
691f9bc - root, 14 hours ago : modified index.html
830e289 - root, 10 days ago : Initial commit with source code
[root@localhost project_gitrepo]#
```

## REVIEW CHANGES

### Git diff:

The **git diff** command is used to show the differences between the working directory and the index.

**Note:** So far we have used some example files, but from here I am going to use one sample web project.

**Example:** I have modified index.html file.

**File Path:** /project\_gitrepo/web/index.html

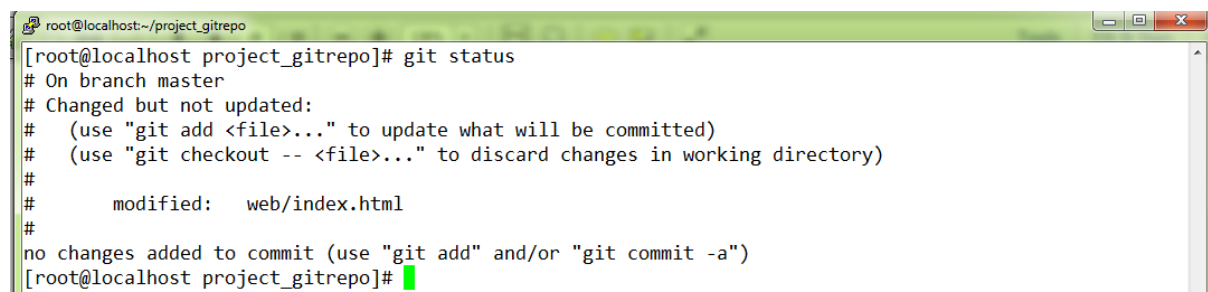


```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git diff
diff --git a/web/index.html b/web/index.html
index 0783283..4c002fb 100644
--- a/web/index.html
+++ b/web/index.html
@@ -21,6 +21,7 @@
     <li>To a <a href="hello.jsp">JSP page</a>.
     <li>To a <a href="hello">servlet</a>.
   </ul>
-
+   <p>
+   <h2> Sample web application </h2>
   </body>
-</html>
\ No newline at end of file
+</html>
[root@localhost project_gitrepo]#
```

In the above figure you will find the + symbol (2 lines) that was added to the index.html file.

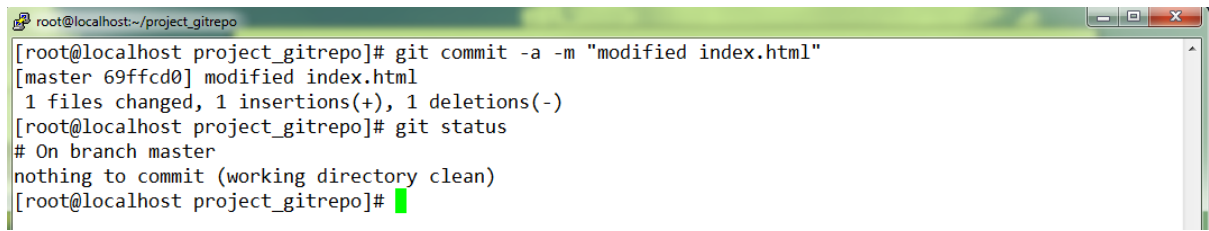
Here - symbols indicates the previous data, and the + symbol denotes the newly added data to that file.

Now **git status** command will gives the status of the working copy or we can use short status by using the command **git status -s**.



```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   web/index.html
#
no changes added to commit (use "git add" and/or "git commit -a")
[root@localhost project_gitrepo]#
```

Then commit your changes to the repository by using the command **git commit -a -m "log message"**. This command can be worked as combination of both **git add**, and **git commit -m "log message"** where -a is **add**.

A terminal window titled 'root@localhost:~/project\_gitrepo' showing the execution of git commands. The user runs 'git commit -a -m "modified index.html"', which outputs '[master 69ffcd0] modified index.html' and '1 files changed, 1 insertions(+), 1 deletions(-)'. Then, the user runs 'git status', which outputs '# On branch master' and 'nothing to commit (working directory clean)'. The prompt returns to 'root@localhost project\_gitrepo]#'.

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git commit -a -m "modified index.html"
[master 69ffcd0] modified index.html
1 files changed, 1 insertions(+), 1 deletions(-)
[root@localhost project_gitrepo]# git status
# On branch master
nothing to commit (working directory clean)
[root@localhost project_gitrepo]#
```

**\$ git diff --staged (or) git diff --cached:** This command will be used to display the differences in the staged area files.

## REVERT BACK CHANGES

### Undo changes:

At any stage, you may want to undo something. Here, we'll review a few basic tools for undoing changes that you've made. Be careful, because you can't always undo some of these undo. This is one of the few areas in Git where you may lose some work if you do it wrong.

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   web/index.html
#
no changes added to commit (use "git add" and/or "git commit -a")
[root@localhost project_gitrepo]#
```

I have modified a file and not yet added to the staging area at this time if you want to rollback or undo your changes to that modified file you need to use the command **git checkout**.

### Syntax:

`git checkout -- [file]`

### Example:

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git checkout -- web/index.html
[root@localhost project_gitrepo]# git status
# On branch master
nothing to commit (working directory clean)
[root@localhost project_gitrepo]#
```

In the above scenario I have used **git checkout -- web/index.html** to undo the changes that made to the file. It's important to understand that `git checkout -- [file]` is a dangerous command. Any changes you made to that file are gone – you just copied another file over it. Don't ever use this command unless you absolutely know that you don't want the file.

There is an advantage of using git at this point, if you use **git status** command every time it will suggest you to whatever operation that are available to perform.

If you add the modified file to the staging area it has to commit to the repository, at this point if you want revert back from the staging area you need to use the command **git reset**.

### Syntax:

`git reset HEAD <file>`

## Example:

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git status ←
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   web/index.html
#
no changes added to commit (use "git add" and/or "git commit -a")
You have new mail in /var/spool/mail/root
[root@localhost project_gitrepo]# git add web/index.html ←
[root@localhost project_gitrepo]# git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   web/index.html
#
[root@localhost project_gitrepo]# git reset HEAD web/index.html ←
Unstaged changes after reset:
M       web/index.html
[root@localhost project_gitrepo]# git status ←
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   web/index.html
#
no changes added to commit (use "git add" and/or "git commit -a")
[root@localhost project_gitrepo]#
```

Now here I want to commit changes to the repository. For that directly we can use the commit command.

## Syntax:

```
git commit -a -m "log message"
```

## Example:

```
git commit -a -m "Added another line of code to the index.html"
```

One of the common undo takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to try that commit again, you can run commit with the --amend option.

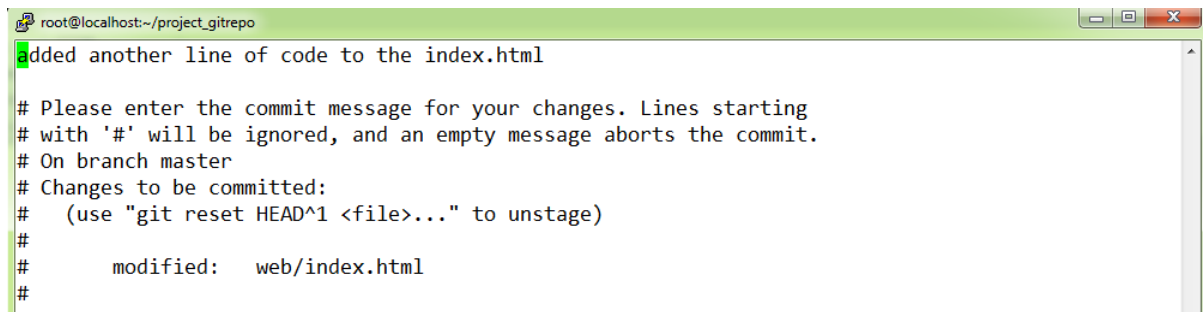
This command takes your staging area and uses it for the commit. If you've made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same, and all you'll change is your commit message. The same commit-message editor fires up, but it already contains the message of your previous commit. You can edit the message the same as always, but it overwrites your previous commit.

## Syntax:

```
git commit --amend
```

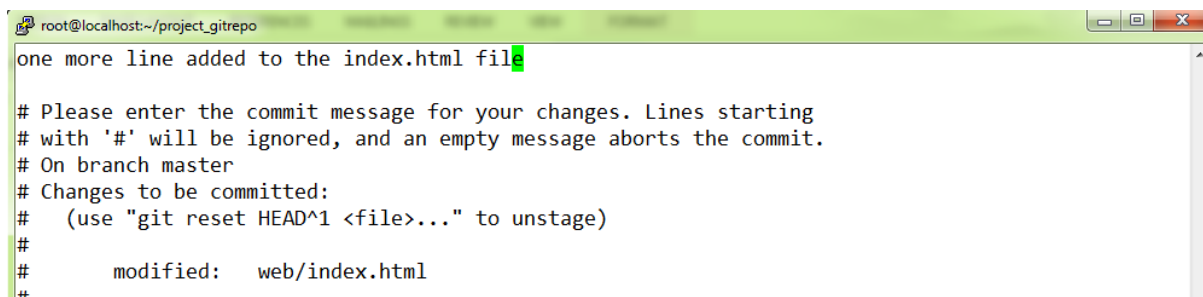
If you use this command it will take you to the core editor whatever editor you have configured for the git and it will give you the commit message here you can change the commit message.

Here I have configured **vim** editor for the git.



```
root@localhost:~/project_gitrepo
added another line of code to the index.html

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD^1 <file>..." to unstage)
#
#       modified:   web/index.html
#
```



```
root@localhost:~/project_gitrepo
one more line added to the index.html file

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD^1 <file>..." to unstage)
#
#       modified:   web/index.html
#
```

Here you can change the commit message and then save it then it will commit to the repository. If you use **vi** or **vim editor** to save the changes and exit you have to use **Esc :wq** option.

## WORKING WITH REMOTE REPOSITORIES

So far we have seen working with local repository and working copies of the git. But now here we go with remote repositories.

To connect the remote server you should have the intranet or internet. To access the remote machines throw local machine you should some network protocols.

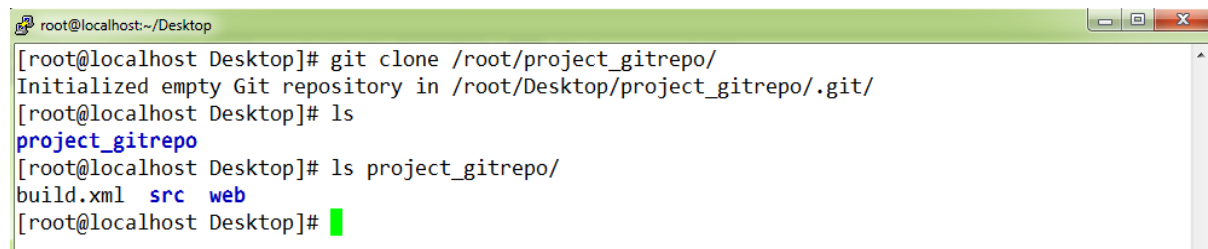
### The Protocols:

Git can use four major protocols to transfer data: Local, HTTP, Secure Shell (SSH) and Git. Here we'll discuss what they are and in what basic circumstances you would want (or not want) to use them.

### Local Protocol

The most basic is the *Local protocol*, in which the remote repository is in another directory on disk. This is often used if everyone on your team has access to a shared filesystem such as an NFS mount, or in the less likely case that everyone logs in to the same computer. The latter wouldn't be ideal, because all your code repository instances would reside on the same computer, making a catastrophic loss much more likely.

If you have a shared mounted filesystem, then you can clone, push to, and pull from a local file-based repository. To clone a repository like this or to add one as a remote to an existing project, use the path to the repository as the URL. For example, to clone a local repository, you can run something like this:



```
root@localhost:~/Desktop
[root@localhost Desktop]# git clone /root/project_gitrepo/
Initialized empty Git repository in /root/Desktop/project_gitrepo/.git/
[root@localhost Desktop]# ls
project_gitrepo
[root@localhost Desktop]# ls project_gitrepo/
build.xml  src  web
[root@localhost Desktop]#
```



```
root@localhost:~/Desktop
[root@localhost Desktop]# git clone file:///root/project_gitrepo
Initialized empty Git repository in /root/Desktop/project_gitrepo/.git/
remote: Counting objects: 25, done.
remote: Compressing objects: 100% (21/21), done.
Receiving objects: 100% (25/25), 13.27 KiB, done.
Resolving deltas: 100% (9/9), done.
remote: Total 25 (delta 9), reused 0 (delta 0)
[root@localhost Desktop]#
```

### The HTTP Protocols

Git can communicate over HTTP in two different modes. Prior to Git 1.6.6 there was only one way it could do this which was very simple and generally read-only. In version 1.6.6 a new, smarter protocol was introduced that involved Git being able to intelligently negotiate data transfer in a manner similar to how it does over SSH. In the last few years, this new HTTP



protocol has become very popular since it's simpler for the user and smarter about how it communicates. The newer version is often referred to as the "Smart" HTTP protocol and the Older way as "Dumb" HTTP. We'll cover the newer "smart" HTTP protocol first.

```
$ git clone https://example.com/gitproject.git
```

```
$ git clone http://example.com/gitproject.git
```

### **SMART HTTP:**

The "smart" HTTP protocol operates very similarly to the SSH or Git protocols but runs over standard HTTP/S ports and can use various HTTP authentication mechanisms, meaning it's often easier on the user than something like SSH, since you can use things like username/password basic authentication rather than having to set up SSH keys. It has probably become the most popular way to use Git now, since it can be set up to both serve anonymously like the git:// protocol, and can also be pushed over with authentication and encryption like the SSH protocol. Instead of having to set up different URLs for these things, you can now use a single URL for both. If you try to push and the repository requires authentication (which it normally should), the server can prompt for a username and password. The same goes for read access. In fact, for services like GitHub, the URL you use to view the repository online (for example, "<https://github.com/simplegit>") is the same URL you can use to clone and, if you have access, push over.

### **DUMB HTTP:**

If the server does not respond with a Git HTTP smart service, the Git client will try to fall back to the simpler "dumb" HTTP protocol. The Dumb protocol expects the bare Git repository to be served like normal files from the web server. The beauty of the Dumb HTTP protocol is the simplicity of setting it up. Basically, all you have to do is put a bare Git repository under your HTTP document root and set up a specific post-update hook, and you're done (See "**Git Hooks**"). At that point, anyone who can access the web server under which you put the repository can also clone your repository. To allow read access to your repository over HTTP, do something like this

```
$ cd /var/www/htdocs/  
$ git clone --bare /path/to/git_project gitproject.git  
$ cd gitproject.git  
$ mv hooks/post-update.sample hooks/post-update  
$ chmod a+x hooks/post-update
```

### **The SSH Protocol**

A common transport protocol for Git when self-hosting is over SSH. This is because SSH access to servers is already set up in most places – and if it isn't, it's easy to do. SSH is also an authenticated network protocol; and because it's ubiquitous, it's generally easy to set up and use. To clone a Git repository over SSH, you can specify ssh:// URL like this:

```
$ git clone ssh://user@server/project.git
```

Or you can use the shorter scp-like syntax for the SSH protocol:

```
$ git clone user@server:project.git
```

You can also not specify a user, and Git assumes the user you're currently logged in as.

```
chandu@localhost:~$ git clone ssh://root@192.168.89.128/root/project_gitrepo/
Initialized empty Git repository in /home/chandu/project_gitrepo/.git/
root@192.168.89.128's password:
remote: Counting objects: 25, done.
remote: Compressing objects: 100% (21/21), done.
Receiving objects: 100% (25/25), 13.27 KiB, done.
Resolving deltas: 100% (9/9), done.
remote: Total 25 (delta 9), reused 0 (delta 0)
[chandu@localhost ~]$
```

### Showing your Remotes:

To see which remote servers you have configured, you can run the `git remote` command. It lists the short names of each remote handle you've specified. If you've cloned your repository, you should at least see `origin` – that is the default name Git gives to the server you cloned from.

```
$ git remote
$ git remote -v
```

```
chandu@localhost:~/project_gitrepo$ git remote -v
origin  ssh://root@192.168.89.128/root/project_gitrepo/ (fetch)
origin  ssh://root@192.168.89.128/root/project_gitrepo/ (push)
[chandu@localhost project_gitrepo]$
```

```
$ git remote show origin
```

```
chandu@localhost:~/project_gitrepo$ git remote show origin
root@192.168.89.128's password:
* remote origin
  Fetch URL: ssh://root@192.168.89.128/root/project_gitrepo/
  Push URL:  ssh://root@192.168.89.128/root/project_gitrepo/
  HEAD branch: master
  Remote branch:
    master tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
[chandu@localhost project_gitrepo]$
```

This means we can pull contributions from any of these users pretty easily. We may additionally have permission to push to one or more of these, though we can't tell that here. Notice that these remotes use a variety of protocols

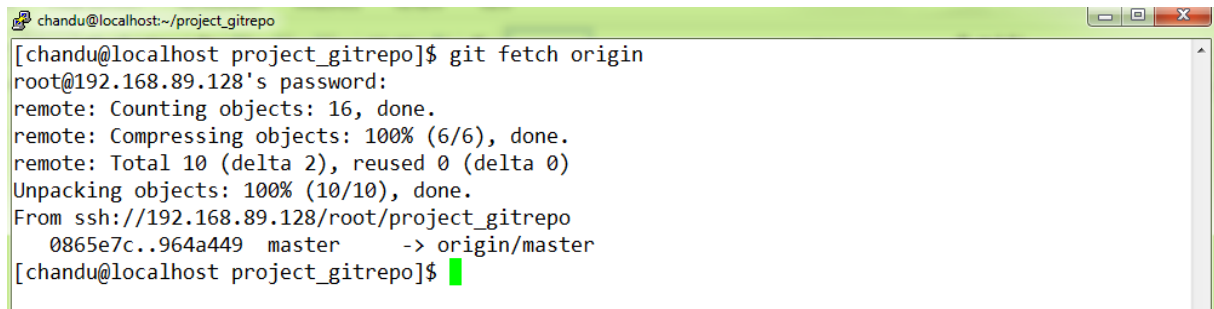
## The Git Protocol

This is a special daemon that comes packaged with Git; it listens on a dedicated port (9418) that provides a service similar to the SSH protocol, but with absolutely no authentication. In order for a repository to be served over the Git protocol, you must create the `git-daemon-export-ok` file – the daemon won't serve a repository without that file in it – but other than that there is no security. Either the Git repository is available for everyone to clone or it isn't. This means that there is generally no pushing over this protocol. You can enable push access; but given the lack of authentication, if you turn on push access, anyone on the internet who finds your project's URL could push to your project. Suffice it to say that this is rare.

## Fetching and Pulling from your Remotes:

As you just saw, to get data from your remote projects, you can run:

```
$ git fetch [remote-name]
```

A terminal window titled 'chandu@localhost:~/project\_gitrepo' showing the execution of the 'git fetch origin' command. The output shows the process of counting, compressing, and unpacking objects from the remote repository 'origin/master'.

```
chandu@localhost:~/project_gitrepo$ git fetch origin
root@192.168.89.128's password:
remote: Counting objects: 16, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 10 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (10/10), done.
From ssh://192.168.89.128/root/project_gitrepo
 0865e7c..964a449  master    -> origin/master
chandu@localhost:~/project_gitrepo$
```

The command goes out to that remote project and pulls down all the data from that remote project that you don't have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

If you clone a repository, the command automatically adds that remote repository under the name "origin". So, `git fetch origin` fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It's important to note that the `git fetch` command pulls the data to your local repository – it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to merge it manually into your work when you're ready.

If you have a branch set up to track a remote branch, you can use the `git pull` command to automatically fetch and then merge a remote branch into your current branch. This may be an easier or more comfortable workflow for you; and by default, the `git clone` command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is called) on the server you cloned from. Running `git pull` generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you're currently working on.

## Pushing to Your Remotes

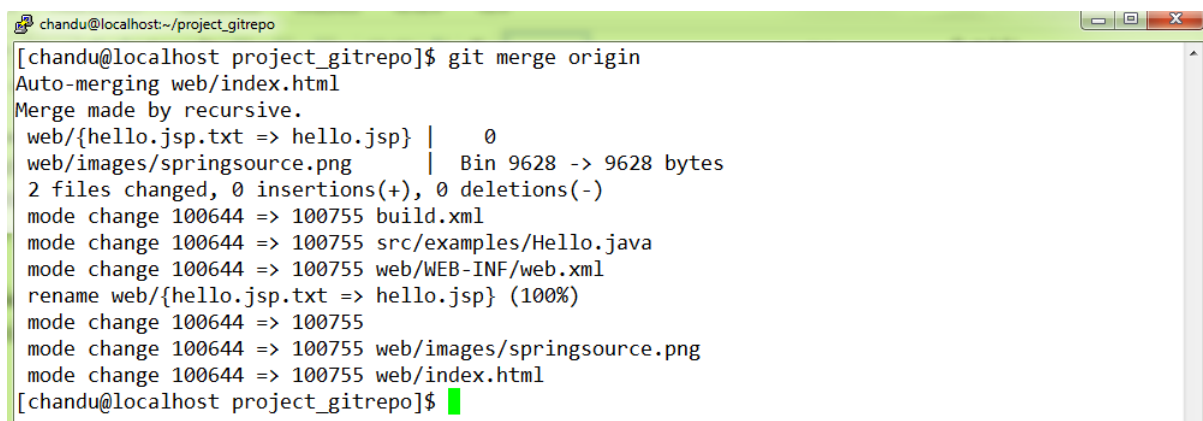
When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: `git push [remote-name] [branch-name]`. If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you've done back up to the server:

```
$ git push origin master
```

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push up - stream, your push will rightly be rejected. You'll have to pull down their work first and incorporate it into yours before you'll be allowed to push.

**Note:** Before using the push you need to merge the local repository with remote, otherwise it will be rejected the push.

```
$ git merge origin
```

A terminal window titled 'chandu@localhost:~/project\_gitrepo' showing the output of the 'git merge origin' command. The output indicates an auto-merge of 'web/index.html' and a recursive merge. It lists several file changes: 'web/{hello.jsp.txt => hello.jsp}' (0 changes), 'web/images/springsource.png' (Bin 9628 -> 9628 bytes), '2 files changed, 0 insertions(+), 0 deletions(-)', 'mode change 100644 => 100755 build.xml', 'mode change 100644 => 100755 src/examples/Hello.java', 'mode change 100644 => 100755 web/WEB-INF/web.xml', 'rename web/{hello.jsp.txt => hello.jsp} (100%)', 'mode change 100644 => 100755', 'mode change 100644 => 100755 web/images/springsource.png', and 'mode change 100644 => 100755 web/index.html'. The prompt returns to '[chandu@localhost project\_gitrepo]\$' with a green cursor.

```
[chandu@localhost project_gitrepo]$ git merge origin
Auto-merging web/index.html
Merge made by recursive.
 web/{hello.jsp.txt => hello.jsp} |      0
web/images/springsource.png      | Bin 9628 -> 9628 bytes
2 files changed, 0 insertions(+), 0 deletions(-)
mode change 100644 => 100755 build.xml
mode change 100644 => 100755 src/examples/Hello.java
mode change 100644 => 100755 web/WEB-INF/web.xml
rename web/{hello.jsp.txt => hello.jsp} (100%)
mode change 100644 => 100755
mode change 100644 => 100755 web/images/springsource.png
mode change 100644 => 100755 web/index.html
[chandu@localhost project_gitrepo]$
```

## BRANCHING IN GIT

Nearly every VCS has some form of branching support. Branching means you diverge from the main line of development and continue to do work without messing with that main line. In many VCS tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory, which can take a long time for large projects. Some people refer to Git's branching model as its "killer feature," and it certainly sets Git apart in the VCS community. Why is it so special? The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast. Unlike many other VCSs, Git encourages workflows that branch and merge often, even multiple times in a day. Understanding and mastering this feature gives you a powerful and unique tool and can entirely change the way that you develop.

The "master" branch in Git is not a special branch. It is exactly like any other branch. The only reason nearly every repository has one is that the `git init` command creates it by default and most people don't bother to change it.

### Creating a Branch:

In git very easy to create the branch by using the **branch** command. Let's assume here I have finished developing the project for initial release, so I want to create a new branch called test.

### Syntax:

```
$ git branch <branch name>
```

Instead of using two commands: **git branch**, **git checkout**. You can use below command to create and switch to that branch.

```
$ git checkout -b <branch name>
```

### Example:

```
$ git branch testing
```

How does Git know what branch you're currently on? It keeps a special pointer called HEAD. Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS. In Git, this is a pointer to the local branch you're currently on. In this case, you're still on master. The `git branch` command only *created* a new branch – it didn't switch to that branch.

You can easily see this by running a simple `git log` command that shows you where the branch pointers are pointing. This option is called `--decorate`.

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git log --oneline --decorate
964a449 (HEAD, testing, master) File changed
ea27675 file name changed in web directory
0865e7c one more line added to the index.html file
ccaa222 added one line of code to index.html
691f9bc modified index.html
830e289 Initial commit with source code
[root@localhost project_gitrepo]#
```

You can see the “master” and “testing” branches that are right there next to the **964a449** commit.

You will come to know how many branches are there in your current git project and what are they by using the command **git branch**.

*\$ git branch*

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git branch
* master
  testing
[root@localhost project_gitrepo]#
```

In the above figure \* (symbol) indicates that the current branch you are in, and remaining are the other branches that you have created.

### Switching Branches:

To switch to an existing branch, you run the **git checkout** command. Let's switch to the new testing branch. This moves HEAD to point to the testing branch.

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git checkout testing
Switched to branch 'testing'
[root@localhost project_gitrepo]# git branch
  master
* testing
[root@localhost project_gitrepo]#
```

Now create a new file and commit changes to the repository.

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# touch readme.txt
[root@localhost project_gitrepo]# git add readme.txt
[root@localhost project_gitrepo]# git commit -m "created readme.txt file"
[testing 920f379] created readme.txt file
0 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 readme.txt
[root@localhost project_gitrepo]#
```

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git branch
master
* testing
[root@localhost project_gitrepo]# ls
build.xml  readme.txt  src  web
[root@localhost project_gitrepo]# git checkout master
Switched to branch 'master'
[root@localhost project_gitrepo]# ls
build.xml  src  web
[root@localhost project_gitrepo]#
```

Now list out your working copy contents by using **ls** command, you will find the newly created file called **readme.txt** in **testing** branch.

Then switch to **master branch** using the **git checkout** command, and list out the directory in the repository, there you cannot find the file **readme.txt**.

You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work. Both of those changes are isolated in separate branches, you can switch back and forth between the branches and merge them together when you're ready. And you did all that with simple branch, checkout, and commit commands.

You can also see this easily with the **git log** command. If you run **git log --oneline --decorate --graph --all** it will print out the history of your commits, showing where your branch pointers are and how your history has diverged.

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git log --oneline --decorate --graph --all
* 920f379 (testing) created readme.txt file
* 964a449 (HEAD, master) File changed
* ea27675 file name changed in web directory
* 0865e7c one more line added to the index.html file
* ccaa222 added one line of code to index.html
* 691f9bc modified index.html
* 830e289 Initial commit with source code
[root@localhost project_gitrepo]#
```

Because a branch in Git is in actuality a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).

The project got tested and it was succeed, then we want to package the code and deploy it into the production.

For that we have created a branch called **v1.0**, and it gets deployed to the production server, now branch **v1.0** will be merged with the master branch.

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git branch v1.0
[root@localhost project_gitrepo]# git branch
* master
  testing
  v1.0
[root@localhost project_gitrepo]#
```

From **v1.0** branch I have created another branch called **v2.0** and these all branches will have the same contents. Now developers started working on **v2.0** and suddenly I got the bug in my project. So quickly I need to fix that bug, for that some of the developers need to work on it.

At this situation some of the developers are working on next release **v2.0** and some of the developers are working on the bug fix of **v1.0**. Before started working on the bug fix you need create the branch for it from the master branch (hotfix).

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git branch
hotfix
* master
  testing
  v1.0
  v2.0
[root@localhost project_gitrepo]# git checkout hotfix
Switched to branch 'hotfix'
[root@localhost project_gitrepo]# git branch
* hotfix
  master
  testing
  v1.0
  v2.0
[root@localhost project_gitrepo]#
```

Now developers fixed the bug and it is working fine now, so this branch gets merged with the master branch and then deletes the hotfix branch.

**Note:** Merging concept will be covered in the next sessions.

Then all the developers should merge with the master branch to get the latest changes. Then the developers will work on the future release. This will reduce the strain and it will not mess up the old contents.

### Deleting the Branch:

If we consider the above scenario, after merging the hotfix branch with the master branch you need to delete the branch. Because after merging with the all branches that are available in repository gets same contents. So for that we need to use hotfix at any point of time.



```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git checkout master
Switched to branch 'master'
[root@localhost project_gitrepo]# git branch -d hotfix
Deleted branch hotfix (was 964a449).
[root@localhost project_gitrepo]# git branch
* master
  testing
  v1.0
  v2.0
[root@localhost project_gitrepo]#
```

**Note:** If you want to delete the branch make sure that is not your current working branch.

### Renaming the Branch:

Already some branches were created, and one of the branches does not have the appropriate branch name. So we need to change the branch name, at this situation you have to use rename operation.

#### Syntax:

*\$ git branch -m <old branch name> <new branch name>*

#### Example:

*\$ git branch -m test testing*

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git branch
* master
  test
  v1.0
  v2.0
[root@localhost project_gitrepo]# git branch -m test testing
[root@localhost project_gitrepo]# git branch
* master
  testing
  v1.0
  v2.0
[root@localhost project_gitrepo]#
```

#### Git branch options:

Option	Description
<b>-m, -M</b>	Rename or move the branch
<b>--merged</b>	It shows the merged branch
<b>--no-merged</b>	Prints only not merged branches
<b>-d</b>	Deletes the branch
<b>-a</b>	List both remote - tracking and local branches
<b>-D</b>	Deletes the branch (even if not merged)
<b>-f or --force</b>	Force creation (when already exists)

## MERGING IN GIT

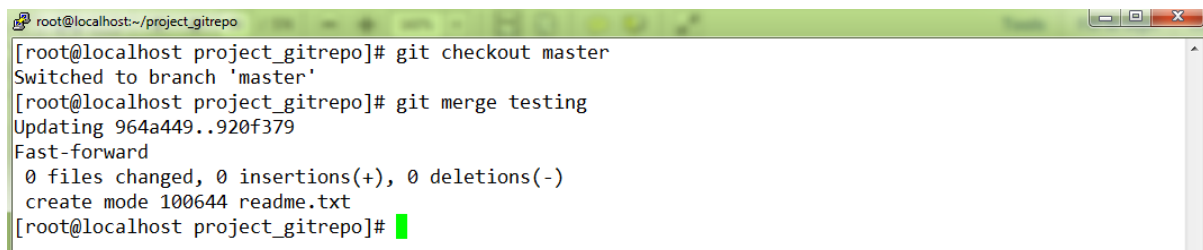
Suppose we have done some changes in testing branch in the above scenario, now that changes are not updated to the master branch. To update the changes to master branch to testing branch we should merge with that.

For that all we have to do is checkout the branch you wish to merge into and then run the **git merge** command.

### Syntax:

```
$ git merge <branch name>
```

### Example:



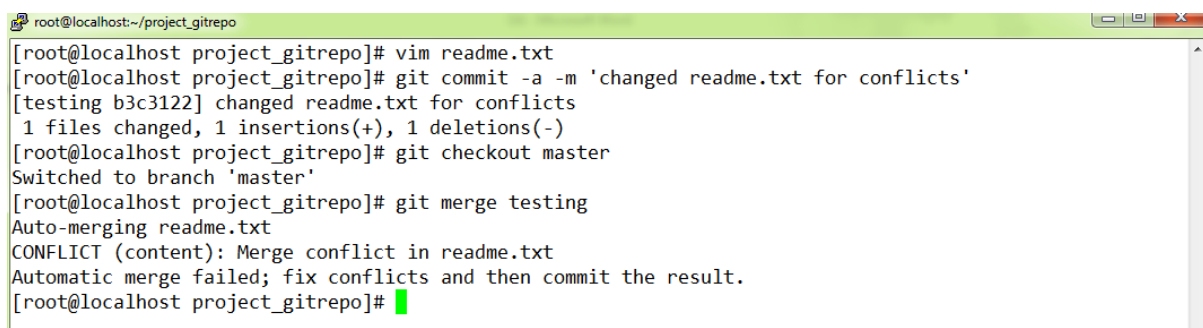
```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git checkout master
Switched to branch 'master'
[root@localhost project_gitrepo]# git merge testing
Updating 964a449..920f379
Fast-forward
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 readme.txt
[root@localhost project_gitrepo]#
```

We have created the **readme.txt** file in testing branch, to update that latest changes into master branch we will need to merge with the master branch. For that checkout the branch **master** then use the git merge command.

### Merge Conflicts:

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging together, Git won't be able to merge them cleanly.

For example we have a file **readme.txt** in testing branch and we have merged with master branch now both branches having the same content. Here if you done some changes in **readme.txt** file in testing branch and if we want merge that to master branch, the conflicts will be occurred.



```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# vim readme.txt
[root@localhost project_gitrepo]# git commit -a -m 'changed readme.txt for conflicts'
[testing b3c3122] changed readme.txt for conflicts
 1 files changed, 1 insertions(+), 1 deletions(-)
[root@localhost project_gitrepo]# git checkout master
Switched to branch 'master'
[root@localhost project_gitrepo]# git merge testing
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
[root@localhost project_gitrepo]#
```

## RESOLVING MERGE CONFLICTS

Whenever merge conflicts occurred in the git, to resolve the merge conflicts you should know why and where that conflicts was there.

Use the git status to display the where and why that conflicts occurred.

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git status
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
[root@localhost project_gitrepo]#
```

Now use the git commit command to resolve conflicts/merge conflicts to the master branch. Before commit changes to the branch you can also use **git diff** command to know the differences between them.

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# git commit -a -m "resolve conflicts"
[master 92140bb] resolve conflicts
[root@localhost project_gitrepo]# git status
# On branch master
nothing to commit (working directory clean)
[root@localhost project_gitrepo]# git merge testing
Already up-to-date.
[root@localhost project_gitrepo]#
```

In another case

```
root@localhost:~/project_gitrepo
[root@localhost project_gitrepo]# vim readme.txt
[root@localhost project_gitrepo]# cat readme.txt
from testing branch to resolve conflicts
[root@localhost project_gitrepo]# git commit -a -m "changes in readme.txt"
[testing e014d49] changes in readme.txt
1 files changed, 1 insertions(+), 0 deletions(-)
[root@localhost project_gitrepo]# git status
# On branch testing
nothing to commit (working directory clean)
[root@localhost project_gitrepo]# git checkout master
Switched to branch 'master'
[root@localhost project_gitrepo]# cat readme.txt
[root@localhost project_gitrepo]# git merge testing
Updating b56b2e9..e014d49
Fast-forward
 readme.txt | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
[root@localhost project_gitrepo]# cat readme.txt
from testing branch to resolve conflicts
[root@localhost project_gitrepo]#
```

## GIT REBASING

In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase. In this section you'll learn what rebasing is, how to do it, why it's a pretty amazing tool, and in what cases you won't want to use it.

If you go back to an earlier example from “**Merging**”, you can see that you diverged your work and made commits on two different branches.

With the rebase command, you can take all the changes that were committed on one branch and replay them on another one.

### Syntax:

```
$ git rebase <branch name>
```

### Example:

```
$ git checkout testing  
$ git rebase master
```

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

At this point, you can go back to the master branch and do a fast-forward merge.

```
$ git checkout master  
$ git merge testing
```

In other scenario, suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on the server-side changes until it's tested further. You can take the changes on client that aren't on server and replay them on your master branch by using the --onto option of git rebase:

```
$ git rebase --onto master server client
```

This basically says, “Check out the client branch, and then replay them onto master.” It's a bit complex, but the result is pretty cool.

Now you can fast-forward your master branch.

```
$ git checkout master  
$ git merge client
```

Let's say you decide to pull in your server branch as well. You can rebase the server branch onto the master branch without having to check it out first by running `git rebase [base branch] [topic branch]` – which checks out the topic branch (in this case, server) for you and replays it onto the base branch (master).

```
$ git rebase master server  
$ git merge testing
```

Then you can forward the base branch (master), similar to the above example.

## TAGGING IN GIT

Like most VCSs, Git has the ability to tag specific points in history as being important. Typically we use this functionality to mark release points (v1.0, and so on).

### Listing Your Tags:

Listing the available tags in Git is straightforward. Just type `git tag`:

#### Syntax:

```
$ git tag
```

This command lists the tags in alphabetical order; the order in which they appear has no real importance. You can also search for tags with a particular pattern. The Git source repo, for instance, contains more than 500 tags. If you're only interested in looking at the 1.8.5 series, you can run this.

```
$ git tag -l 'v1.8.5*'
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

### Creating Tags:

Git uses two main types of tags: lightweight and annotated.

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

Annotated tags, however, are stored as full objects in the Git database. They're checksummed; contain the tagger name, e-mail, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It's generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don't want to keep the other information, lightweight tags are available too.

### Annotated Tags

Creating an annotated tag in Git is simple. The easiest way is to specify `-a` when you run the tag command:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

The -m specifies a tagging message, which is stored with the tag. If you don't specify a message for an annotated tag, Git launches your editor so you can type it in.

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

```
$ git show v1.4
tag v1.4
Tagger: Chandu <chandu.2035@gmail.com>
Date: Sat May 3 20:19:12 2014 -0700
my version 1.4
commit ca82a6dff817ec66f44342007202690a93763949
Author: Shekhar <chandu.2035@yahoo.com>
Date: Mon Mar 17 21:52:11 2008 -0700
Changed the version number
```

That shows the tagger information, the date the commit was tagged, and the annotation message before showing the commit information.

### Lightweight Tags

Another way to tag commits is with a lightweight tag. This is basically the commit checksum stored in a file – no other information is kept. To create a lightweight tag, don't supply the -a, -s, or -m option:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

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

### Tagging Later:

You can also tag commits after you've moved past them. Suppose your commit history looks like this:

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: shekhar <chandu.2035@yahoo.com>
Date: Mon Mar 17 21:52:11 2008 -0700
changed the version number

$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
```

```
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

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

```
$ git tag -a v1.2 9fceb02
```

You can see that you’ve tagged the commit:

```
$ git tag -a v1.2 9fceb02
```

You can see that you’ve tagged the commit:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Chandu <chandu.2035@gmail.com>
Date: Mon Feb 9 15:32:16 2009 -0800
version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: shekhar <chandu.2035@yahoo.com>
Date: Sun Apr 27 20:43:35 2008 -0700
updated rakefile
...
```

### Sharing Tags:

By default, the git push command doesn’t transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches – you can run git push origin [tag name].

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:shekhar/simplegit.git
```

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

If you have a lot of tags that you want to push up at once, you can also use the `--tags` option to the `git push` command. This will transfer all of your tags to the remote server that are not already there.

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:shekhar/simplegit.git
* [new tag] v1.4 -> v1.4
* [new tag] v1.4-lw -> v1.4-lw
```

Now, when someone else clones or pulls from your repository, they will get all your tags as well.

### **Checking out Tags:**

You can't really check out a tag in Git, since they can't be moved around. If you want to put a version of your repository in your working directory that looks like a specific tag, you can create a new branch at a specific tag:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

Of course if you do this and do a commit, your `version2` branch will be slightly different than your `v2.0.0` tag since it will move forward with your new changes, so do be careful.



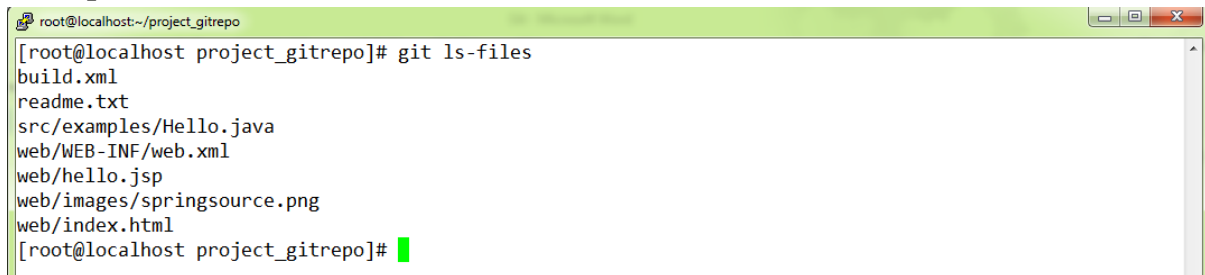
## LISTING THE GIT CONTENTS WITH STATUS

To list the all of the files that are available in the repository you can use the command called **git ls-files**.

### Syntax:

\$ git ls-files |options|

### Example:

A terminal window titled 'root@localhost:~/project\_gitrepo' shows the command '[root@localhost project\_gitrepo]# git ls-files' being executed. The output lists the following files: build.xml, readme.txt, src/examples/Hello.java, web/WEB-INF/web.xml, web/hello.jsp, web/images/springsource.png, and web/index.html. The prompt returns to '[root@localhost project\_gitrepo]#'.

### Options:

Option	Description
<b>-c, --cached</b>	Show cached files in the output (default)
<b>-d, --deleted</b>	Show deleted files in the output
<b>-m, --modified</b>	Show modified files in the output
<b>-o, --others</b>	Show other (i.e. untracked) files in the output
<b>-i, --ignored</b>	Show only ignored files in the output. When showing files in the index, print only those matched by an exclude pattern. When showing "other" files, show only those matched by an exclude pattern.
<b>-s, --stage</b>	Show staged contents' object name, mode bits and stage number in the output.
<b>--directory</b>	If a whole directory is classified as "other", show just its name (with a trailing slash) and not its whole contents.
<b>--no-empty-directory</b>	Do not list empty directories. Has no effect without --directory.
<b>-u, --unmerged</b>	Show unmerged files in the output (forces --stage)
<b>-k, --killed</b>	Show files on the filesystem that need to be removed due to file/directory conflicts for checkout-index to succeed.
<b>-z</b>	\0 line termination on output.
<b>-x &lt;pattern&gt;, --exclude=&lt;pattern&gt;</b>	Skips files matching pattern. Note that pattern is a shell wildcard pattern.
<b>-X &lt;file&gt;, --exclude-from=&lt;file&gt;</b>	Exclude patterns are read from <file>; 1 per line.

## GIT ALIASES

### Git Aliases:

Before we finish this chapter on basic Git, there's just one little tip that can make your Git experience simpler, easier, and more familiar: aliases. We won't refer to them or assume you've used them later in the book, but you should probably know how to use them. Git doesn't automatically infer your command if you type it in partially. If you don't want to type the entire text of each of the Git commands, you can easily set up an alias for each command using git config.

Here are a couple of examples you may want to set up:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

This means that, for example, instead of typing git commit, you just need to type git ci. As you go on using Git, you'll probably use other commands frequently as well; don't hesitate to create new aliases.

This technique can also be very useful in creating commands that you think should exist. For example, to correct the usability problem you encountered with unstaging a file, you can add your own unstage alias to Git:

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

This makes the following two commands equivalent:

```
$ git unstage fileA
$ git reset HEAD fileA
```

This seems a bit clearer. It's also common to add a last command, like this:

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

This way, you can see the last commit easily:

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Chandu <chandu.2035@gmail.com>
Date: Tue Aug 26 19:48:51 2008 +0800
test for current head
Signed-off-by: Shekhar <chandu.2035@yahoo.com>
```

As you can tell, Git simply replaces the new command with whatever you alias it for. However, maybe you want to run an external command, rather than a Git subcommand. In that case, you start the command with a ! character. This is useful if you write your own tools that work with a Git repository. We can demonstrate by aliasing git visual to run gitk:

```
$ git config --global alias.visual '!gitk'
```

## GIT WITH OTHER VERSION CONTROL SYSTEMS

At some point, you may want to convert your existing project to Git. Git provides such a nice experience for developers that many people have figured out how to use it on their workstation, even if the rest of their team is using an entirely different VCS. There are a number of these adapters, called “bridges,” available.

### **Git and Subversion:**

A large fraction of open source development projects and a good number of corporate projects use Subversion to manage their source code. It’s been around for more than a decade, and for most of that time was the *de facto* VCS choice for open-source projects. It’s also very similar in many ways to CVS, which was the big boy of the source-control world before that. One of Git’s great features is a bidirectional bridge to Subversion called `git svn`. This tool allows you to use Git as a valid client to a Subversion server, so you can use all the local features of Git and then push to a Subversion server as if you were using Subversion locally. This means you can do local branching and merging, use the staging area, use rebasing and cherry-picking, and so on while your collaborators continue to work in their dark and ancient ways. It’s a good way to sneak Git into the corporate environment and help your fellow developers become more efficient while you lobby to get the infrastructure changed to support Git fully. The Subversion bridge is the gateway drug to the DVCS world.

### **GIT SVN:**

The base command in Git for all the Subversion bridging commands is `git svn`. It takes quite a few commands, so we’ll show the most common while going through a few simple workflows. It’s important to note that when you’re using `git svn`, you’re interacting with Subversion, which is a system that works very differently from Git. Although you **can** do local branching and merging, it’s generally best to keep your history as linear as possible by rebasing your work, and avoiding doing things like simultaneously interacting with a Git remote repository. Don’t rewrite your history and try to push again, and don’t push to a parallel Git repository to collaborate with fellow Git developers at the same time. Subversion can have only a single linear history, and confusing it is very easy. If you’re working with a team, and some are using SVN and others are using Git, make sure everyone is using the SVN server to collaborate – doing so will make your life easier.

### **SETTING UP:**

To demonstrate this functionality, you need a typical SVN repository that you have write access to. If you want to copy these examples, you’ll have to make a writeable copy of my test repository. In order to do that easily, you can use a tool called `svnsync` that comes with Subversion. For these tests, we created a new Subversion repository on Google Code that was a partial copy of the `protobuf` project, which is a tool that encodes structured data for network transmission.

To follow along, you first need to create a new local Subversion repository:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Then, enable all users to change revprops – the easy way is to add a `prerevprop-change` script that always exits 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
```

```
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

You can now sync this project to your local machine by calling `svnsync init` with the to and from repositories.

```
$ svnsync init file:///tmp/test-svn \
http://progit-example.git.com/svn/
```

This sets up the properties to run the sync. You can then clone the code by running.

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .....[...]
Committed revision 2.
Copied properties for revision 2.
[...]
```

Although this operation may take only a few minutes, if you try to copy the original repository to another remote repository instead of a local one, the process will take nearly an hour, even though there are fewer than 100 commits. Subversion has to clone one revision at a time and then push it back into another repository – it’s ridiculously inefficient, but it’s the only easy way to do this.

### Getting Started:

Now that you have a Subversion repository to which you have write access, you can go through a typical workflow. You’ll start with the `git svn clone` command, which imports an entire Subversion repository into a local Git repository. Remember that if you’re importing from a real hosted Subversion repository, you should replace the `file:///tmp/test-svn` here with the URL of your Subversion repository:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcbfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
A m4/acx_pthread.m4
A m4/stl_hash.m4
A java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
A java/src/test/java/com/google/protobuf/WireFormatTest.java
...
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn/branches/Found branch
parent: (refs/remotes/origin/my-calc-branch) 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
Checked out HEAD:
file:///tmp/test-svn/trunk r75
```

This runs the equivalent of two commands – `git svn init` followed by `git svn fetch` – on the URL you provide. This can take a while. The test project has only about 75 commits and the code

base isn't that big, but Git has to check out each version, one at a time, and commit it individually. For a project with hundreds or thousands of commits, this can literally take hours or even days to finish.

The `-T trunk -b branches -t tags` part tells Git that this Subversion repository follows the basic branching and tagging conventions. If you name your trunk, branches, or tags differently, you can change these options. Because this is so common, you can replace this entire part with `-s`, which means standard layout and implies all those options. The following command is equivalent:

```
$ git svn clone file:///tmp/test-svn -s
```

At this point, you should have a valid Git repository that has imported your branches and tags:

```
$ git branch -a
* master
remotes/origin/my-calc-branch
remotes/origin/tags/2.0.2
remotes/origin/tags/release-2.0.1
remotes/origin/tags/release-2.0.2
remotes/origin/tags/release-2.0.2rc1
remotes/origin/trunk
```

Note how this tool manages Subversion tags as remote refs. Let's take a closer look with the Git plumbing command `show-ref`.

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git doesn't do this when it clones from a Git server; here's what a repository with tags looks like after a fresh clone.

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebcd695e2e4193db5e refs/tags/v1.0.0
```

Git fetches the tags directly into `refs/tags`, rather than treating them remote branches.

### **Committing back to Subversion:**

Now that you have a working repository, you can do some work on the project and push your commits back upstream, using Git effectively as a SVN client. If you edit one of the files and commit it, you have a commit that exists in Git locally that doesn't exist on the Subversion server.

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
1 file changed, 5 insertions(+)
```

Next, you need to push your change upstream. Notice how this changes the way you work with Subversion – you can do several commits offline and then push them all at once to the Subversion server. To push to a Subversion server, you run the `git svn dcommit` command.

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M README.txt
Committed r77
M README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and
refs/remotes/origin/Resetting to the latest refs/remotes/origin/trunk
```

This takes all the commits you've made on top of the Subversion server code, does a Subversion commit for each, and then rewrites your local Git commit to include a unique identifier. This is important because it means that all the SHA-1 checksums for your commits change. Partly for this reason, working with Git-based remote versions of your projects concurrently with a Subversion server isn't a good idea. If you look at the last commit, you can see the new `git-svn-id` that was added.

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date: Thu Jul 24 03:08:36 2014 +0000
Adding git-svn instructions to the README
git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
Notice that the SHA checksum that originally started with 4af61fd when
you committed now begins with 95e0222. If you want to push to both a Git
server and a Subversion server, you have to push (dcommit) to the Subversion
server first, because that action changes your commit data.
```

## GIT COMMANDS

### **Git config:**

Git has a default way of doing hundreds of things. For a lot of these things, you can tell Git to default to doing them a different way, or set your preferences. This involves everything from telling Git what your name is to specific terminal color preferences or what editor you use. There are several files this command will read from and write to so you can set values globally or down to specific repositories. The git config command has been used in nearly every chapter of the book.

In “**First-Time Git Setup**” we used it to specify our name, email address and editor preference before we even got started using Git.

In “**Git Aliases**” we showed how you could use it to create shorthand commands that expand to long option sequences so you don’t have to type them every time.

In “**Rebasing**” we used it to make --rebase the default when you run git pull.

In “**Credential Storage**” we used it to set up a default store for your HTTP passwords.

In “**Keyword Expansion**” we showed how to set up smudge and clean filters on content coming in and out of Git.

Finally, basically the entirety of “**Git Configuration**” is dedicated to the command.

### **Git help:**

The git help command is used to show you all the documentation shipped with Git about any command. While we’re giving a rough overview of most of the more popular ones in this appendix, for a full listing of all of the possible options and flags for every command, you can always run git help <command>. We introduced the git help command in “**Getting Help**” and showed you how to use it to find more information about the git shell in “**Setting Up the Server**”.

### **Getting and Creating Projects**

There are two ways to get a Git repository. One is to copy it from an existing repository on the network or elsewhere and the other is to create a new one in an existing directory.

#### **git init:**

To take a directory and turn it into a new Git repository so you can start version controlling it, you can simply run git init. We first introduce this in “**Getting a Git Repository**”, where we show creating a brand new repository to start working with. We talk briefly about how you can change the default branch from “master” in “**Remote Branches**”. We use this command to create an empty bare repository for a server in “**Putting the Bare Repository on a Server**”. Finally, we go through some of the details of what it actually does behind the scenes in “**Plumbing and Porcelain**”.

#### **git clone:**

The git clone command is actually something of a wrapper around several other commands. It creates a new directory, goes into it and runs git init to make it an empty Git repository, adds a



remote (git remote add) to the URL that you pass it (by default named origin), runs a git fetch from that remote repository and then checks out the latest commit into your working directory with git checkout. The git clone command is used in dozens of places throughout the book, but we'll just list a few interesting places. It's basically introduced and explained in **"Cloning an Existing Repository"**, where we go through a few examples.

In **"Getting Git on a Server"** we look at using the --bare option to create a copy of a Git repository with no working directory. In **"Bundling"** we use it to unbundle a bundled Git repository. Finally, in **"Cloning a Project with Sub modules"** we learn the --recursive option to make cloning a repository with sub modules a little simpler. Though it's used in many other places through the book, these are the ones that are somewhat unique or where it is used in ways that are a little different.

### **Basic Snapshotting:**

For the basic workflow of staging content and committing it to your history, there are only a few basic commands.

#### **git add:**

The git add command adds content from the working directory into the staging area (or "index") for the next commit. When the git commit command is run, by default it only looks at this staging area, so git add is used to craft what exactly you would like your next commit snapshot to look like. This command is an incredibly important command in Git and is mentioned or used dozens of times in this book. We'll quickly cover some of the unique uses that can be found. We first introduce and explain git add in detail in **"Tracking New Files"**. We mention how to use it to resolve merge conflicts in **"Basic Merge Conflicts"**. We go over using it to interactively stage only specific parts of a modified file in **"Interactive Staging"**.

Finally, we emulate it at a low level in **"Tree Objects"**, so you can get an idea of what it's doing behind the scenes.

#### **git status:**

The git status command will show you the different states of files in your working directory and staging area. Which files are modified and unstaged and which are staged but not yet committed. In it's normal form, it also will show you some basic hints on how to move files between these stages. We first cover status in **"Checking the Status of Your Files"**, both in it's basic and simplified forms. While we use it throughout the book, pretty much everything you can do with the git status command is covered there.

#### **git diff:**

The git diff command is used when you want to see differences between any two trees. This could be the difference between your working environment and your staging area (git diff by itself), between your staging area and your last commit (git diff --staged), or between two commits (git diff master branch B). We first look at the basic uses of git diff in **"Viewing Your Staged and Unstaged Changes"**, where we show how to see what changes are staged and which are not yet staged. We use it to look for possible whitespace issues before committing with the --check option in **"Commit Guidelines"**. We see how to check the differences between branches more effectively with the git diff A...B syntax in **"Determining What Is Introduced"**. We use it to filter out whitespace differences with -w and how to compare different stages of conflicted files with --theirs, --ours and --base in **"Advanced Merging"**.



Finally, we use it to effectively compare sub module changes with --submodule in **“Starting with Submodules”**.

#### **git difftool:**

The git difftool command simply launches an external tool to show you the difference between two trees in case you want to use something other than the built in git diff command. We only briefly mention this in ???.

#### **git commit:**

The git commit command takes all the file contents that have been staged with git add and records a new permanent snapshot in the database and then moves the branch pointer on the current branch up to it. We first cover the basics of committing in **“Committing Your Changes”**. There we also demonstrate how to use the -a flag to skip the git add step in daily workflows and how to use the -m flag to pass a commit message in on the command line instead of firing up an editor. In **“Undoing Things”** we cover using the --amend option to redo the most recent commit. In **“Branches in a Nutshell”**, we go into much more detail about what git commit does and why it does it like that. We looked at how to sign commits cryptographically with the -S flag in **“Signing Commits”**. Finally, we take a look at what the git commit command does in the background and how it’s actually implemented in **“Commit Objects”**.

#### **git reset:**

The git reset command is primarily used to undo things, as you can possibly tell by the verb. It moves around the HEAD pointer and optionally changes the index or staging area and can also optionally change the working directory if you use --hard. This final option makes it possible for this command to lose your work if used incorrectly, so make sure you understand it before using it. We first effectively cover the simplest use of git reset in **“Unstaging a Staged File”**, where we use it to unstage a file we had run git add on. We then cover it in quite some detail in **“Reset Demystified”**, which is entirely devoted to explaining this command. We use git reset --hard to abort a merge in **“Aborting a Merge”**, where we also use git merge --abort, which is a bit of a wrapper for the git reset command.

#### **git rm:**

The git rm command is used to remove files from the staging area and working directory for Git. It is similar to git add in that it stages a removal of a file for the next commit.

We cover the git rm command in some detail in **“Removing Files”**, including recursively removing files and only removing files from the staging area but leaving them in the working directory with --cached. The only other differing use of git rm in the book is in **“Removing Objects”** where we briefly use and explain the --ignore-unmatch when running git filter-branch, which simply makes it not error out when the file we are trying to remove doesn’t exist. This can be useful for scripting purposes.

#### **git mv:**

The git mv command is a thin convenience command to move a file and then run git add on the new file and git rm on the old file. We only briefly mention this command in **“Moving Files”**.

### **git clean:**

The git clean command is used to remove unwanted files from your working directory. This could include removing temporary build artifacts or merge conflict files. We cover many of the options and scenarios in which you might use the clean command in **“Cleaning your Working Directory”**.

### **Branching and Merging**

There are just a handful of commands that implement most of the branching and merging functionality in Git.

### **git branch:**

The git branch command is actually something of a branch management tool. It can list the branches you have, create a new branch, delete branches and rename branches.

### **git checkout:**

The git checkout command is used to switch branches and check content out into your working directory. We first encounter the command in **“Switching Branches”** along with the git branch command. We see how to use it to start tracking branches with the --track flag in **“Tracking Branches”**. We use it to reintroduce file conflicts with --conflict=diff3 in **“Checking Out Conflicts”**. We go into closer detail on it’s relationship with git reset in **“Reset Demystified”**. Finally, we go into some implementation detail in **“The HEAD”**.

### **git merge:**

The git merge tool is used to merge one or more branches into the branch you have checked out. It will then advance the current branch to the result of the merge. The git merge command was first introduced in **“Basic Branching”**. Though it is used in various places in the book, there are very few variations of the merge command — generally just git merge <branch> with the name of the single branch you want to merge in. We covered how to do a squashed merge (where Git merges the work but pretends like it’s just a new commit without recording the history of the branch you’re merging in) at the very end of **“Forked Public Project”**. We went over a lot about the merge process and command, including the -Xignore-all-whitespace command and the --abort flag to abort a problem merge in **“Advanced Merging”**. We learned how to verify signatures before merging if your project is using GPG signing in **“Signing Commits”**. Finally, we learned about Subtree merging in **“Subtree Merging”**.

### **git mergetool:**

The git mergetool command simply launches an external merge helper in case you have issues with a merge in Git. We mention it quickly in **“Basic Merge Conflicts”** and go into detail on how to implement your own external merge tool in **“External Merge and Diff Tools”**.

### **git log:**

The git log command is used to show the reachable recorded history of a project from the most recent commit snapshot backwards. By default it will only show the history of the branch you’re currently on, but can be given different or even multiple heads or branches from which to traverse. It is also often used to show differences between two or more branches at the commit level. This command is used in nearly every chapter of the book to demonstrate the history of a project.

### **git stash:**

The git stash command is used to temporarily store uncommitted work in order to clean out your working directory without having to commit unfinished work on a branch. This is basically entirely covered in **“Stashing and Cleaning”**.

### **git tag:**

The git tag command is used to give a permanent bookmark to a specific point in the code history. Generally this is used for things like releases. This command is introduced and covered in detail in **“Tagging”** and we use it in practice in **“Tagging Your Releases”**. We also cover how to create a GPG signed tag with the -s flag and verify one with the -v flag in **“Signing Your Work”**.

### **Sharing and Updating Projects:**

There are not very many commands in Git that access the network, nearly all of the commands operate on the local database. When you are ready to share your work or pull changes from elsewhere, there are a handful of commands that deal with remote repositories.

### **git fetch:**

The git fetch command communicates with a remote repository and fetches down all the information that is in that repository that is not in your current one and stores it in your local database. We first look at this command in **“Fetching and Pulling from Your Remotes”** and we continue to see examples of it use in **“Remote Branches”**. We also use it in several of the examples in **“Contributing to a Project”**. We use it to fetch a single specific reference that is outside of the default space in **“Pull Request Refs”** and we see how to fetch from a bundle in **“Bundling”**. We set up highly custom refsspecs in order to make git fetch do something a little different than the default in **“The Refspec”**.

### **git pull:**

The git pull command is basically a combination of the git fetch and git merge commands, where Git will fetch from the remote you specify and then immediately try to merge it into the branch you’re on. We introduce it quickly in **“Fetching and Pulling from Your Remotes”** and show how to see what it will merge if you run it in **“Inspecting a Remote”**. We also see how to use it to help with rebasing difficulties in **“Rebase When You Rebase”**.

We show how to use it with a URL to pull in changes in a one-off fashion in **“Checking Out Remote Branches”**. Finally, we very quickly mention that you can use the --verifysignatures option to it in order to verify that commits you are pulling have been GPG signed in **“Signing Commits”**.

### **git push:**

The git push command is used to communicate with another repository, calculate what your local database has that the remote one does not, and then pushes the difference into the other repository. It requires write access to the other repository and so normally is authenticated somehow. We first look at the git push command in **“Pushing to Your Remotes”**. Here we cover the basics of pushing a branch to a remote repository. In **“Pushing”** we go a little deeper into pushing specific branches and in **“Tracking Branches”** we see how to set up tracking branches to automatically push to. In **“Deleting Remote Branches”** we use the --delete flag to delete a branch on the server with git push. Throughout **“Contributing to a Project”** we see several examples of using git push to share work on branches through multiple remotes. We see how to use it to share tags that you have made with the --tags option in **“Sharing Tags”**.

In **“Publishing Submodule Changes”** we use the `--recurse-submodules` option to check that all of our submodules work has been published before pushing the superproject, which can be really helpful when using submodules. In **“Other Client Hooks”** we talk briefly about the pre-push hook, which is a script we can setup to run before a push completes to verify that it should be allowed to push. Finally, in **“Pushing Refspecs”** we look at pushing with a full refspec instead of the general shortcuts that are normally used. This can help you be very specific about what work you wish to share.

#### **git remote:**

The git remote command is a management tool for your record of remote repositories. It allows you to save long URLs as short handles, such as “origin” so you don’t have to type them out all the time. You can have several of these and the git remote command is used to add, change and delete them. This command is covered in detail in **“Working with Remotes”**, including listing, adding, removing and renaming them.

#### **git archive:**

The git archive command is used to create an archive file of a specific snapshot of the project. We use git archive to create a tarball of a project for sharing in **“Preparing a Release”**.

#### **git submodule:**

The git submodule command is used to manage external repositories within a normal repositories. This could be for libraries or other types of shared resources. The submodule command has several sub-commands (add, update, sync, etc) for managing these resources. This command is only mentioned and entirely covered in **“Submodules”**.

### **Inspection and Comparison**

#### **git show:**

The git show command can show a Git object in a simple and human readable way. Normally you would use this to show the information about a tag or a commit. We first use it to show annotated tag information in **“Annotated Tags”**. Later we use it quite a bit in **“Revision Selection”** to show the commits that our various revision selections resolve to. One of the more interesting things we do with git show is in **“Manual File Re-merging”** to extract specific file contents of various stages during a merge conflict.

#### **git shortlog:**

The git shortlog command is used to summarize the output of git log. It will take many of the same options that the git log command will but instead of listing out all of the commits it will present a summary of the commits grouped by author.

#### **git describe:**

The git describe command is used to take anything that resolves to a commit and produces a string that is somewhat human-readable and will not change. It’s a way to get a description of a commit that is as unambiguous as a commit SHA but more understandable. We use git describe in **“Generating a Build Number”** and **“Preparing a Release”** to get a string to name our release file after.

#### **Debugging:**

Git has a couple of commands that are used to help debug an issue in your code. This ranges from figuring out where something was introduced to figuring out who introduced it.

**git bisect:**

The git bisect tool is an incredibly helpful debugging tool used to find which specific commit was the first one to introduce a bug or problem by doing an automatic binary search. It is fully covered in “**Binary Search**” and is only mentioned in that section.

**git blame:**

The git blame command annotates the lines of any file with which commit was the last one to introduce a change to each line of the file and what person authored that commit. This is helpful in order to find the person to ask for more information about a specific section of your code. It is covered in “**File Annotation**” and is only mentioned in that section.

**git grep:**

The git grep command can help you find any string or regular expression in any of the files in your source code, even older versions of your project. It is covered in “**Git Grep**” and is only mentioned in that section.

**Patching:**

A few commands in Git are centered around the concept of thinking of commits in terms of the changes they introduce, as though the commit series is a series of patches. These commands help you manage your branches in this manner.

**git cherry-pick:**

The git cherry-pick command is used to take the change introduced in a single Git commit and try to re-introduce it as a new commit on the branch you’re currently on. This can be useful to only take one or two commits from a branch individually rather than merging in the branch which takes all the changes.

**git rebase:**

The git rebase command is basically an automated cherry-pick. It determines a series of commits and then cherry-picks them one by one in the same order somewhere else.

**git revert:**

The git revert command is essentially a reverse git cherry-pick. It creates a new commit that applies the exact opposite of the change introduced in the commit you’re targeting, essentially undoing or reverting it. We use this in “**Reverse the commit**” to undo a merge commit.

**Email:**

Many Git projects, including Git itself, are entirely maintained over mailing lists. Git has a number of tools built into it that help make this process easier, from generating patches you can easily email to applying those patches from an email box.

**git apply:**

The git apply command applies a patch created with the git diff or even GNU diff command. It is similar to what the patch command might do with a few small differences. We demonstrate using it and the circumstances in which you might do so in “**Applying Patches from E-mail**”.

**git am:**

The git am command is used to apply patches from an email inbox, specifically one that is mbox formatted. This is useful for receiving patches over email and applying them to your

project easily. We covered usage and workflow around git am in “**Applying a Patch with am**” including using the --resolved, -i and -3 options.

There are also a number of hooks you can use to help with the workflow around git am and they are all covered in “**E-mail Workflow Hooks**”. We also use it to apply patch formatted GitHub Pull Request changes in “**Email Notifications**”.

#### **git format-patch:**

The git format-patch command is used to generate a series of patches in mbox format that you can use to send to a mailing list properly formatted. We go through an example of contributing to a project using the git format-patch tool in “**Public Project over E-Mail**”.

#### **git send-email:**

The git send-email command is used to send patches that are generated with git format-patch over email. We go through an example of contributing to a project by sending patches with the git send-email tool in “**Public Project over E-Mail**”.

#### **git request-pull:**

The git request-pull command is simply used to generate an example message body to email to someone. If you have a branch on a public server and want to let someone know how to integrate those changes without sending the patches over email, you can run this command and send the output to the person you want to pull the changes in. We demonstrate how to use git request-pull to generate a pull message in “**Forked Public Project**”.

#### **External Systems:**

Git comes with a few commands to integrate with other version control systems.

#### **git svn:**

The git svn command is used to communicate with the Subversion version control system as a client. This means you can use Git to checkout from and commit to a Subversion server. This command is covered in depth in “**Git and Subversion**”.

#### **git fast-import:**

For other version control systems or importing from nearly any format, you can use git fast-import to quickly map the other format to something Git can easily record. This command is covered in depth in “**A Custom Importer**”.

#### **Administration**

If you’re administering a Git repository or need to fix something in a big way, Git provides a number of administrative commands to help you out.

#### **git gc:**

The git gc command runs “garbage collection” on your repository, removing unnecessary files in your database and packing up the remaining files into a more efficient format.

This command normally runs in the background for you, though you can manually run it if you wish. We go over some examples of this in “**Maintenance**”.

#### **Git fsck:**

The git fsck command is used to check the internal database for problems or inconsistencies. We only quickly use this once in “**Data Recovery**” to search for dangling objects.

**git reflog:**

The git reflog command goes through a log of where all the heads of your branches have been as you work to find commits you may have lost through rewriting histories. We cover this command mainly in **“RefLog Shortnames”**, where we show normal usage to and how to use git log -g to view the same information with git log output. We also go through a practical example of recovering such a lost branch in **“Data Recovery”**.

**git filter-branch:**

The git filter-branch command is used to rewrite loads of commits according to certain patterns, like removing a file everywhere or filtering the entire repository down to a single subdirectory for extracting a project. In **“Removing a File from Every Commit”** we explain the command and explore several different options such as --commit-filter, --subdirectoryfilter and --tree-filter.

**Plumbing Commands**

There were also quite a number of lower level plumbing commands that we encountered in the book. The first one we encounter is ls-remote in **“Pull Request Refs”** which we use to look at the raw references on the server. We use ls-files in **“Manual File Re-merging”**, **“Rerere”** and **“The Index”** to take a more raw look at what your staging area looks like.



## FAQ

1. What are the advantages of using GIT?

- Data redundancy and replication
- High availability
- Only one.git directory per repository
- Superior disk utilization and network performance
- Collaboration friendly
- Any sort of projects can use GIT

2. What language is used in GIT?

GIT is fast, and 'C' language makes this possible by reducing the overhead of runtimes associated with higher languages.

3. What is the function of 'GIT PUSH' in GIT?

'GIT PUSH' updates remote refs along with associated objects.

4. Why GIT better than Subversion?

GIT is an open source version control system; it will allow you to run 'versions' of a project, which show the changes that were made to the code overtime also it allows you keep the backtrack if necessary and undo those changes. Multiple developers can checkout, and upload changes and each change can then be attributed to a specific developer.

5. What is "Staging Area" or "Index" in GIT?

Before completing the commits, it can be formatted and reviewed in an intermediate area known as 'Staging Area' or 'Index'.

6. What is GIT stash?

GIT stash takes the current state of the working directory and index and puts in on the stack for later and gives you back a clean working directory. So in case if you are in the middle of something and need to jump over to the other job, and at the same time you don't want to lose your current edits then you can use GIT stash.

7. What is GIT stash drop?

When you are done with the stashed item or want to remove it from the list, run the git 'stash drop' command. It will remove the last added stash item by default, and it can also remove a specific item if you include as an argument.

8. How will you know in GIT if a branch has been already merged into master?

Git branch—merged lists the branches that have been merged into the current branch

Git branch—no merged lists the branches that have not been merged



9. What is the function of git clone?

The git clone command creates a copy of an existing Git repository. To get the copy of a central repository, 'cloning' is the most common way used by programmers.

10. What is the function of 'git config'?

The 'git config' command is a convenient way to set configuration options for your Git installation. Behaviour of a repository, user info, preferences etc. can be defined through this command.

11. What does commit object contain?

- A set of files, representing the state of a project at a given point of time
- Reference to parent commit objects
- An SHAI name, a 40 character string that uniquely identifies the commit object.

12. How can you create a repository in Git?

In Git, to create a repository, create a directory for the project if it does not exist, and then run command "git init". By running this command .git directory will be created in the project directory, the directory does not need to be empty.

13. What is 'head' in git and how many heads can be created in a repository?

A 'head' is simply a reference to a commit object. In every repository, there is a default head referred as "Master". A repository can contain any number of heads.

14. What is the purpose of branching in GIT?

The purpose of branching in GIT is that you can create your own branch and jump between those branches. It will allow you to go to your previous work keeping your recent work intact.

15. What is the common branching pattern in GIT?

The common way of creating branch in GIT is to maintain one as "Main" branch and create another branch to implement new features. This pattern is particularly useful when there are multiple developers working on a single project.

16. How can you bring a new feature in the main branch?

To bring a new feature in the main branch, you can use a command "git merge" or "git pull command".

17. What is a 'conflict' in git?

A 'conflict' arises when the commit that has to be merged has some change in one place, and the current commit also has a change at the same place. Git will not be able to predict which change should take precedence.

18. How can conflict in git resolved?

To resolve the conflict in git, edit the files to fix the conflicting changes and then add the resolved files by running “git add” after that to commit the repaired merge, run “git commit”. Git remembers that you are in the middle of a merger, so it sets the parents of the commit correctly.

19. To delete a branch what is the command that is used?

Once your development branch is merged into the main branch, you don't need development branch. To delete a branch use, the command “git branch -d [head]”.

20. What is another option for merging in git?

“Rebasing” is an alternative to merging in git.

21. What is the syntax for “Rebasing” in Git?

The syntax used for rebase is “git rebase [new-commit] “

22. What is the difference between ‘git remote’ and ‘git clone’?

‘git remote add’ just creates an entry in your git config that specifies a name for a particular URL. While, ‘git clone’ creates a new git repository by copying and existing one located at the URI.

23. What is GIT version control?

With the help of GIT version control, you can track the history of a collection of files and includes the functionality to revert the collection of files to another version. Each version captures a snapshot of the file system at a certain point of time. A collection of files and their complete history are stored in a repository.

24. Mention some of the best graphical GIT client for LINUX?

Some of the best GIT client for LINUX is

- Git Cola
- Git-g
- Smart git
- Gigggle
- Git GUI
- qGit

25. What is Subgit? Why to use Subgit?

‘Subgit’ is a tool for a smooth, stress-free SVN to Git migration. Subgit is a solution for a company - wide migration from SVN to Git that is:

- It is much better than git-svn
- No requirement to change the infrastructure that is already placed
- Allows to use all git and all sub-version features
- Provides genuine stress –free migration experience.

26. What is the function of 'git diff' in git?  
'git diff' shows the changes between commits, commit and working tree etc.
27. What is 'git status' is used for?  
As 'Git Status' shows you the difference between the working directory and the index, it is helpful in understanding a git more comprehensively.
28. What is the difference between the 'git diff' and 'git status'?  
'git diff' is similar to 'git status', but it shows the differences between various commits and also between the working directory and index.
29. What is the function of 'git checkout' in git?  
A 'git checkout' command is used to update directories or specific files in your working tree with those from another branch without merging it in the whole branch.
30. What is the function of 'git rm'?  
To remove the file from the staging area and also off your disk 'git rm' is used.
31. What is the function of 'git stash apply'?  
When you want to continue working where you have left your work, 'git stash apply' command is used to bring back the saved changes onto the working directory.
32. What is the use of 'git log'?  
To find specific commits in your project history- by author, date, content or history 'git log' is used.
33. What is 'git add' is used for?  
'git add' adds file changes in your existing directory to your index.
34. What is the function of 'git reset'?  
The function of 'Git Reset' is to reset your index as well as the working directory to the state of your last commit.
35. What is git ls-tree?  
'git ls-tree' represents a tree object including the mode and the name of each item and the SHA-1 value of the blob or the tree.
36. How git instaweb is used?  
'Git Instaweb' automatically directs a web browser and runs webserver with an interface into your local repository.

37. What does 'hooks' consist of in git?

This directory consists of Shell scripts which are activated after running the corresponding Git commands. For example, git will try to execute the post-commit script after you run a commit.

38. Explain what is commit message?

Commit message is a feature of git which appears when you commit a change. Git provides you a text editor where you can enter the modifications made in commits.

39. How can you fix a broken commit?

To fix any broken commit, you will use the command "git commit—amend". By running this command, you can fix the broken commit message in the editor.

40. Why is it advisable to create an additional commit rather than amending an existing commit?

There are couple of reasons.

- The amend operation will destroy the state that was previously saved in a commit. If it's just the commit message being changed then that's not an issue. But if the contents are being amended then chances of eliminating something important remains more.
- Abusing "git commit --amend" can cause a small commit to grow and acquire unrelated changes.

41. What is 'bare repository' in GIT?

To co-ordinate with the distributed development and developer's team, especially when you are working on a project from multiple computers 'Bare Repository' is used. A bare repository comprises of a version history of your code.

42. How do you start Git and what is the process called?

\$ git init

The process is called initializing a git repository

43. How do you determine the current state of the project?

\$ git status

44. How Can I Move all your changes since your last commit to the staging area.

\$ git add .

45. How to Store the saved changes in the repository and add a message "first commit".

\$ git commit -m "first commit"

46. Show a list of all the commits that have been made.

\$ git log or \$ git g

47. How do you put your local repository (repo) on the GitHub server?

- create a new repository in github
- get the SSH key (SSH stands for secure shell)

- `$ git remote add origin [SSH key - something like git@github.com...]`
  - `$ git push -u origin master`
48. How to take a git project from GitHub to your local machine to work on it.  
`$ git pull origin master`  
Or, if you are taking a project that is not already on your local machine, use `$ git clone`.
49. What is the purpose of working off multiple branches?  
You can create a separate branch when you are developing a feature, etc. and can commit to this branch separately. When the feature branch is working properly, you can merge it with the master branch.
50. Write the command to create a branch called working  
`$ git checkout -b working`
51. How to list all of the branches.  
`$ git branch`
52. How to switch to the working branch  
`$ git checkout working`
53. How to merge the changes that were made in the working branch with the master branch.  
`$ git checkout master`  
`$ git merge working`
54. How to delete the working branch  
`$ git branch -d working`
55. What is the command to Put everything up on the remote repository.  
`$ git push`
56. Pull a branch from a remote repository and create the branch locally.  
`git checkout -b fix_stuff origin/fix_stuff`
57. How Can I check my Branch Status?  
`$ git branch` or `$ git b`
58. Add a line to the some.txt file and then throw out all changes since the last commit.  
`$ git reset --hard`

---

# THE END

---