# The Basics

**Initialize the Git Repository**

$ mkdir myrepo

$ cd myrepo

$ git init

Git is designed to be as unobtrusive as possible. Notice that there is now a .git directory in myrepo that stores all the tracking data for our repository. The .git folder is the only difference between a Git repository and an ordinary folder, so deleting it will turn your project back into an unversioned collection of files

git init can be run without any arguments to create the local Git repository in the current directory. Under the myrepo repository directory, a .git is created with various files and directories in the current directory.

$ find .git

.git/config //Contains the configuration of the local repository

.git/HEAD //Head pointer

.git/hooks

...

.git/objects // Object storage

.git/objects/info

.git/objects/pack

.git/refs

.git/refs/heads //Contains the branch pointers

.git/refs/tags //Contains the tag pointers

**Configure Git**

Git comes with a long list of configurations options from your name to your favorite merge tool. You can set options with the git config or by manually editing a file called .gitconfig in your home directory.

The first thing you do with any Git installation is set you name an email.

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

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

Once you’ve installed Git, the first thing you need to do is to tell Git your name and email (particularly before creating any commits). Rather than usernames, Git uses a name and an email address to identify the author of a commit.

We can do this with the git config command:

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

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

The --global option tells Git to use this configuration as a default for all of your repositories. Omitting it lets you specify different user information for individual repositories, which will come in handy later on.

First off, Mary needs to configure her repository so that we know who contributed what to the project.

git config user.name "Mary"

git config user.email mary.example@rypress.com

You may recall from the first module that we used a --global flag to set the configuration for the entire Git installation. But since Mary’s repository is on the local filesystem, she needs a local configuration.

Use a text editor to open up the file called config in the .git folder of Mary’s project (you may need to enable hidden files to see .git). This is where local configurations are stored, and we see Mary’s information at the bottom of the file. Note that this overrides the global configuration that we set in [The Basics](http://rypress.com/tutorials/git/the-basics.html).

Git’s command line relies on a text editor for most of its input. You can forece Git to use your editor of choice with the core.editor option

$ git config --global core.editor notepad

Git supports aliasing commands

$ git config --global alias.st status

## The fundamental Git workflow



The stage/commit process

The distinction between the working directory, the staged snapshot, and committed snapshots is at the core of Git version control. Nearly all other Git commands manipulate one of these components in some way, so understanding the interplay between them is a fantastic foundation for mastering Git.

Most Git commands operate on one of the three main components of a Git repository: the working directory, the staged snapshot, or the committed snapshots.

The working directory is where the developer edit files and compile code.

The staging area is an intermediary between the working directory and the local repository

A Git *repository* is the local collection of all the files related to a particular Git version control system and contains a .git subdirectory in its root. Git keeps track of the state of the files in the repository’s directory on disk.

Git repositories store all their data on your local machine. Making commits, viewing history, and requesting differences between commits are all local operations that don’t require a network connection. This makes all these operations much faster in Git than with centralized version control systems such as Subversion.

Under the new Git repository directory, a .git subdirectory is created with various files and directories under it.

**View the Repository Status**

It would be helpful to view the status of the new repository.

$ git status

$ git status -s

It outputs the state of the working directory and staging aea

It’s a good practice to run git status to see exactly what you’re committing before running git commit

**Stage a Snapshot**

$ echo “abcd” > file1.txt

$ git add file1.txt

$ git status

git add command tells Git to start tracking file1.txt. Git add file1.txt to the snapshot for the next commit. A snapshot represents the state of your project at a given point in time. Git’s term for creating a snapshot is called **staging**. We can add or remove multiple files before actually committing it to the project history.

Git’s staging area gives you a plcae to organize a commit before adding it to the project history. Staging is the process of moving changes from the working directory to the staged snapshot. It gives you the opportunity to pick and choose related changes from the working directory, instead of committing everything all at once

To delete a file from project, you need to add it to the staging area like a new or modified file.The next command will stage the deletion and stop tracking the file but it won’t delete the file from he working directory

$ git rm --cached file1.txt

If you need more detailed information about the changes in your working directory or staging area, you can generate a diff

$ git diff

This outputs a diff of every unstaged changes in your working directory. You can also generate a diff of all staged changes with the --cached flag

$ git diff --cached

**Commit the Snapshot**

Commits represent every saved version of a project which makes them the atomic unit of Git based version control.

The next command will open a text editor and prompt you to enter a message for the commit.

$ git commit

The -m option lets you specify a commit message on the command line instead of opening a text editor. This is just a convenient shortcut.

$ git commit -m "Add navigation links"

The --message flag for git commit can be abbreviated to -m (all abbreviations use a single -). If this flag is omitted, Git opens a text editor (specified by the EDITOR or GIT\_EDITOR environment variable) to prompt you for the commit message

It can also take the --all (or -a) flag to add all changes to files tracked in the repository into a new commit

Saving a version of your project is a two step process:

* **Staging.** Telling Git what files to include in the next commit.
* **Committing.** Recording the staged snapshot with a descriptive message.

Staging files with the git add command doesn’t actually affect the repository in any significant way—it just lets us get our files in order for the next commit. Only after executing git commit will our snapshot be recorded in the repository. Committed snapshots can be seen as “safe” versions of the project.

Our history can now be represented as the following. Note that the red circle, which represents the current commit, automatically moves forward every time we commit a new snapshot.



Current project history

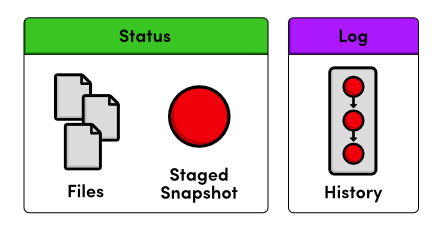
Notice that we skipped the staging step this time around. Instead of using git add, we passed the -a flag to git commit. This convenient parameter tells Git to automatically include *all* tracked files in the staged snapshot. Combined with the -m flag, we can stage and commit snapshots with a single command



Rather than require all changes in the working tree to build up new commits, git allows files to be added incrementally to the index.

**View the Repository History**

The git status command will *only* show us *staged* changes. To view our project history (*committed* changes), we need a new command git log



Status output vs. Log output

The git log command comes with a lot of formatting options. For now, we’ll just use the convenient --oneline flag. Git outputs only the first 7 characters of the checksum. These first few characters effectively serve as a unique ID for each commit

$ git log --oneline

Condensing output to a single line is a great way to get a high-level overview of a repository. Another useful configuration is to pass a filename to git log:

$ git log --oneline file1.txt

This displays only the file1.txt history.