Git Basics

The simple approach

Git Basics

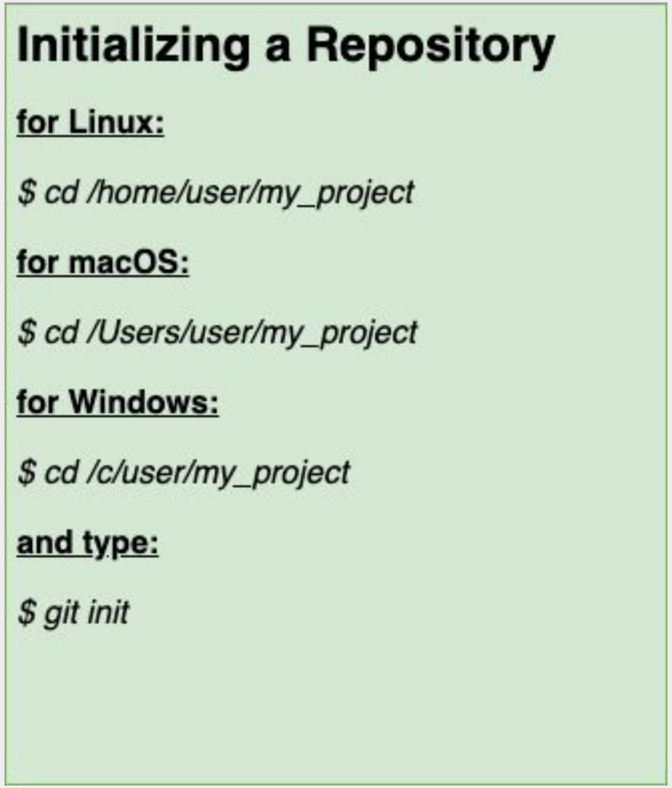
- Overview
- Getting a Git Repository
- Recording Changes to the Repository
- Viewing the Commit History
- Undoing Things

Overview

- This lesson covers every basic command you need to do the vast majority of the things you'll eventually spend your time doing with Git.
- By the end of the lesson, you should be able to configure and initialize a repository, begin and stop tracking files, and stage and commit changes.
- We'll cover how to set up Git to ignore certain files and file patterns, how to undo mistakes quickly and easily, how to browse the history of your project and view changes between commits, and how to push and pull from remote repositories.

Getting a Git Repository

- You typically can obtain a git repository in two ways:
  - You can make a local directory that is not under version control, and turn it into a Git repository, or
  - You can clone an existing Git repository from elsewhere.
- Both ways works. You will end up with a git repo on your local machine ready for work.



Initializing a Repository

for Linux:

```
$ cd /home/user/my_project
```

for macOS:

```
$ cd /Users/user/my_project
```

for Windows:

```
$ cd /c/user/my_project
```

and type:

```
$ git init
```

- Initializing a Git repository in a directory you already have on your local machine can be as simple as typing a very short command.
- First things first we need to navigate to the directory, which could look a little different depending on the system you are running.

Understanding

- In the last step we utilized a command of git. It initialized our directory as a repository.
  - It created a new subdirectory named .git with all the necessary files - the Git structure.
  - At this stage , nothing is being tracked.
- If you would like to start version controlling those files it will take a few Git commands:
  - $ git add *.c
  - $ git add LICENSE
  - $ git commit -m 'initial project version'
- These commands will be fully explained later but for now you have a Git repository with tracked files and an initial commit.

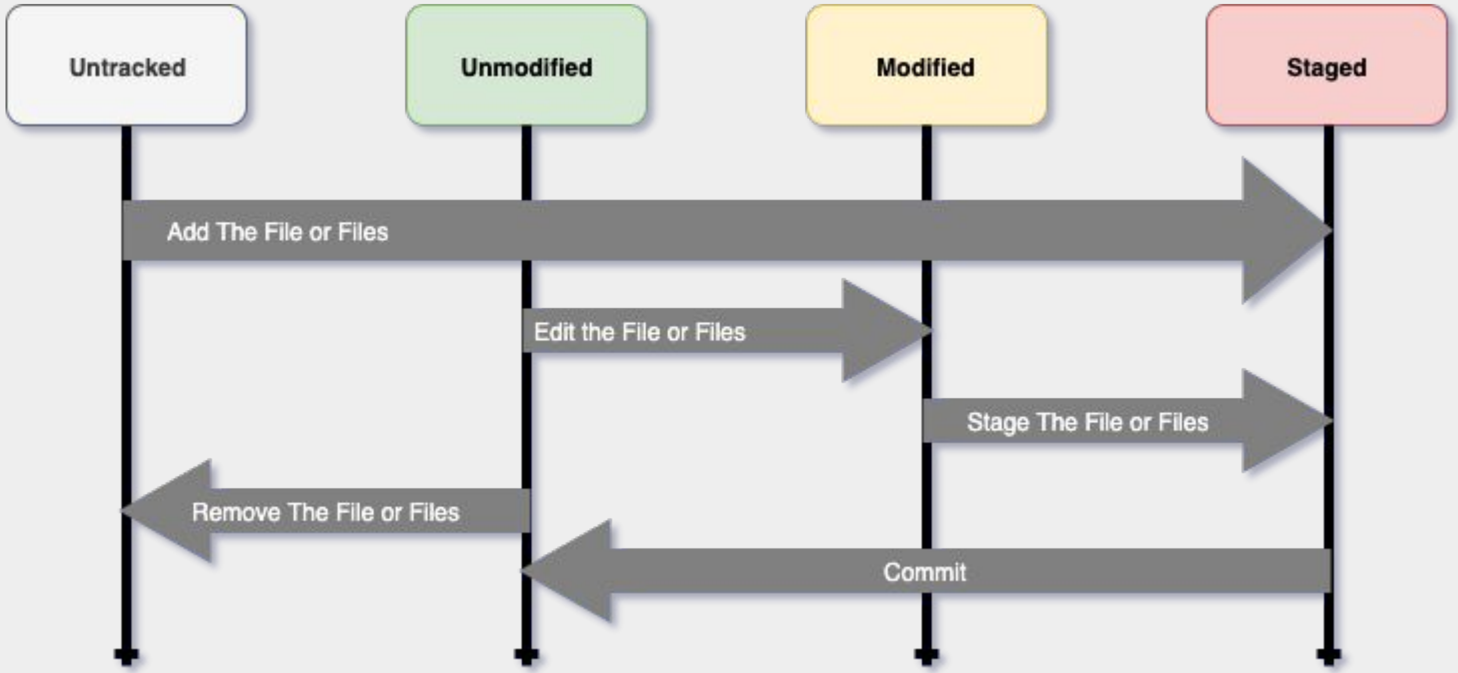
What about Cloning ?

- Cloning a repository is very different from a checkout.
- It’s a Copy of an existing repository. You are not checking it out
- This is an important distinction — instead of getting just a working copy, Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default when you run git clone.
- You clone a repository with git clone <url to any repo>.
- If you want to clone your repo into a specific folder you will have to use the same syntax with the folder name included.
- You clone a repository with git clone <url to any repo> yourFolder.

Recording changes.

- By now you should have a Git repository on your local machine, as well as a checkout or all the files copied and right in front of you.
- You are now able to work on files and commit them periodically
- There are two states of the files that are inside your directory:
    - Tracked & Untracked
- Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.
- Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.
- As you edit files, Git sees them as modified, because you’ve changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.

Recording Changes



Checking the Status of Your Files

- The main tool you use to determine which files are in which state is the git status command. If you run this command directly after a clone, you should see something like this:
  - $ git status
  - On branch master
  - Your branch is up-to-date with 'origin/master'.
  - nothing to commit, working directory clean
- This means you have a clean working directory; in other words, none of your tracked files are modified. Git also doesn’t see any untracked files or they would be listed here.

Tracking New Files

- In order for us to track files we have to use the command “git add”. The best way to do this is to run the simple command “git add .”(including the period). If you run this command you will add all untracked files to the staged area ready to be committed.
- Running git status again will show you something very different with a new heading attached to the output.
  - $ git status
  - On branch master
  - Your branch is up-to-date with 'origin/master'.
  - Changes to be committed:
  - (use "git reset HEAD <file>..." to unstage)
  - new file: README

Staging Modified Files

- Change a file that you are already tracking, then run the git status command again. You will see something like this.
  - $ git status
  - On branch master
  - Your branch is up-to-date with 'origin/master'.
  - Changes to be committed:
  -   (use "git reset HEAD <file>..." to unstage)
  -     new file: README
  -
  - Changes not staged for commit:
  -   (use "git add <file>..." to update what will be committed)
  -   (use "git checkout -- <file>..." to discard changes in working directory)
  -     modified: YourFileNamd.file

Staging Modified Files

- What it’s saying is you have files that are staged, as well as files that are not staged.
- Once you see this and see any file that you want staged ready for commit you should run your “git add .” command
- This will ensure you have all of your files staged ready for commiting.
-

Can I make it Simple

Short Status

- While the git status output is pretty comprehensive, it’s also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run git status -s or git status --short you get a far more simplified output from the command:
    -

- $ git status -s
- M README
- MM Rakefile
- A lib/git.rb
- M lib/simplegit.rb
- ?? LICENSE.txt

Short Status Explained

- New files that aren't tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on.
- There are two columns to the output — the left-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree.
- So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged.
- The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

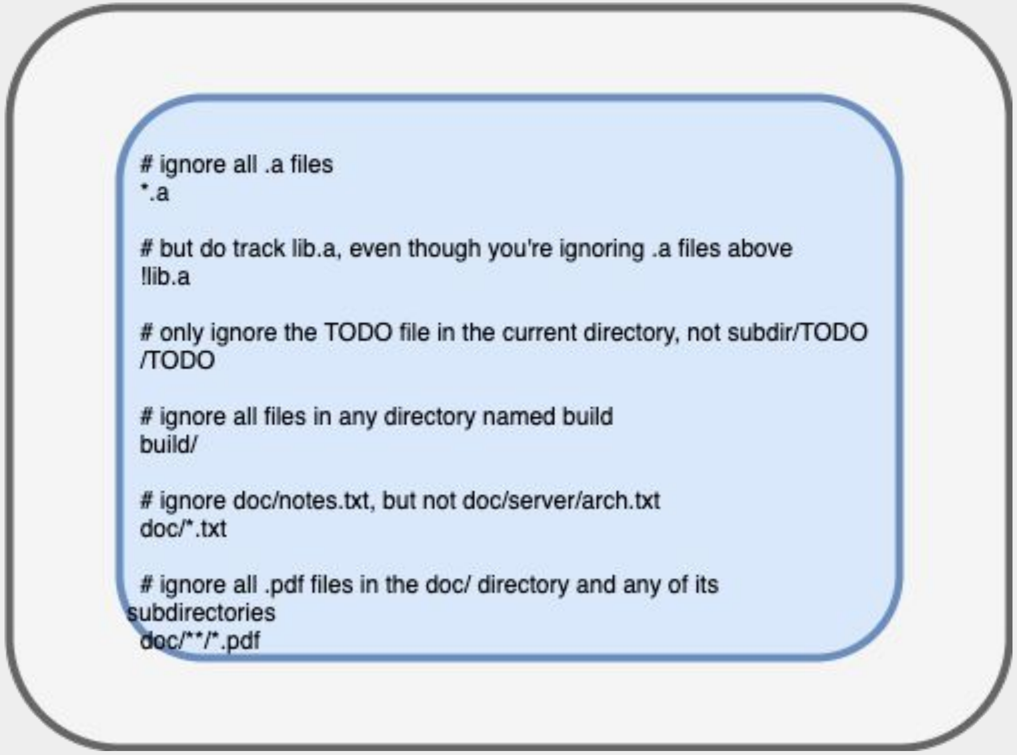
Ignoring Files

- You will often have a file class that you don't want Git to automatically add or even render you untracked. These are generally files created automatically by your build system, such as log files or records.
- In such cases, you can create a file listing patterns to match them named .gitignore. Here is an example .gitignore file:
  - $ cat .gitignore
  - *.[oa]
  - *~
-

Rules of the .gitignore file

- The rules for the patterns you can put in the .gitignore file are as follows:
    - Blank lines or lines starting with # are ignored.
    - Standard glob patterns work, and will be applied recursively throughout the entire working tree.
    - You can start patterns with a forward slash (/) to avoid recursivity.
    - You can end patterns with a forward slash (/) to specify a directory.
    - You can negate a pattern by starting it with an exclamation point (!).

Another .gitignore Example

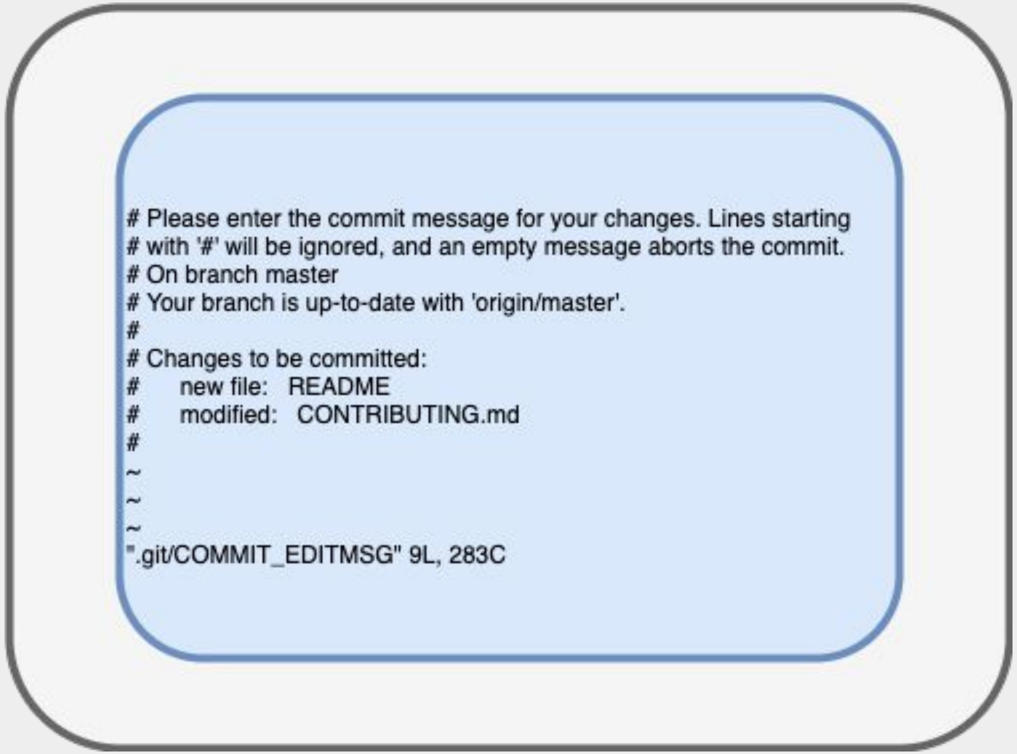


```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its
subdirectories
doc/**/*.pdf
```

Committing Your Changes

- You can commit your changes now that your staging area is set up the way you want it.
- Note that anything that is still unstaged— any files you've developed or updated that you haven't added git since you've edited them — will not go into this commit.
- They will remain on your disk as modified files.
- Let's assume in this case that the last time you ran git status, you saw that it was all on track.
- The simplest way to commit is to type:
  - $ git commit
- Doing so launches your editor of choice. (This is set by your shell's EDITOR environment variable — usually vim or emacs, although you can configure it with whatever you want using the git config --global core.editor command as you saw in Getting Started).

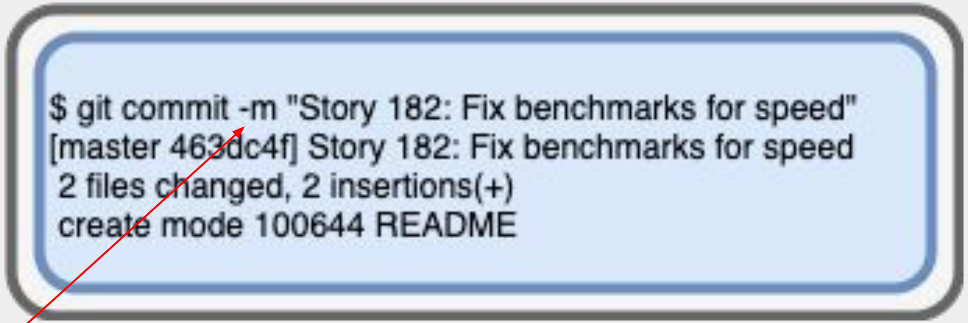
Committing Your Changes



```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#	new file:   README
#	modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Committing Your Changes

- You can see that the default commit message contains the latest output of the git status command commented out and one empty line on top. You can remove these comments and type your commit message, or you can leave them there to help you remember what you're committing.
- Alternatively, you can type your commit message inline with the commit command by specifying it after a -m flag, like this:

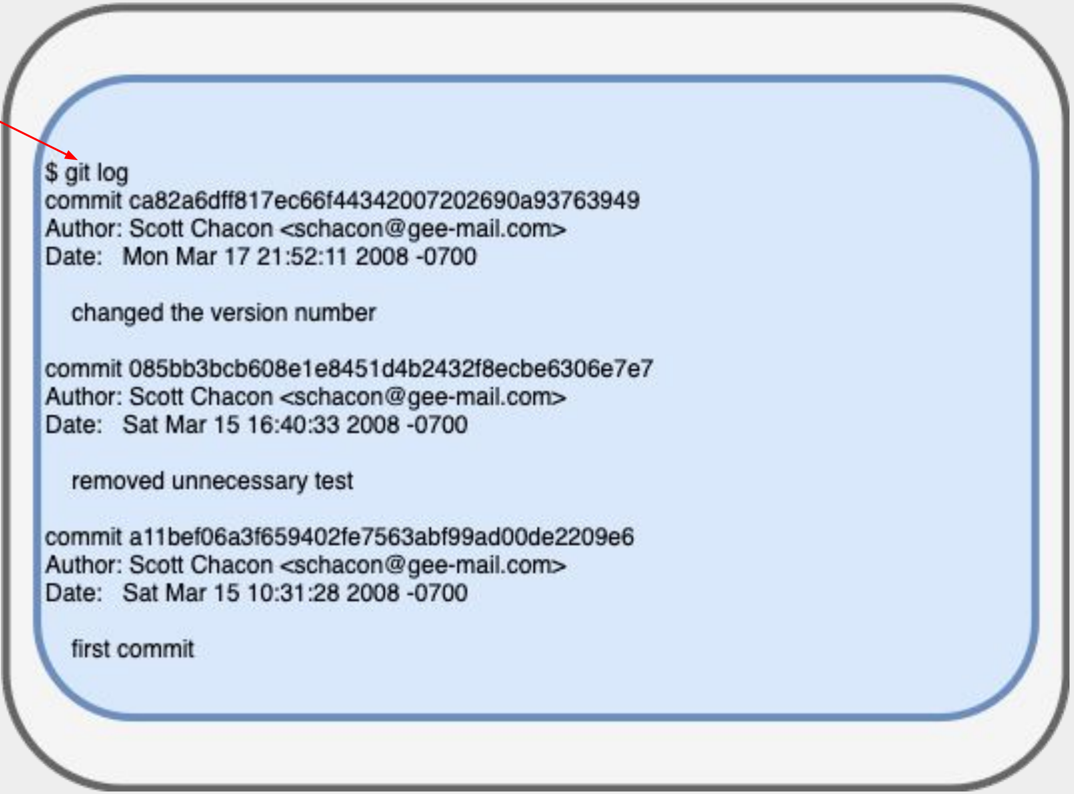


```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

Viewing the Commit History

- After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the git log command.
- These examples use a very simple project called "simplegit". To get the project, run
  - $ git clone https://github.com/schacon/simplegit-progit
- When you run git log in this project, you should get an output.

Log Example

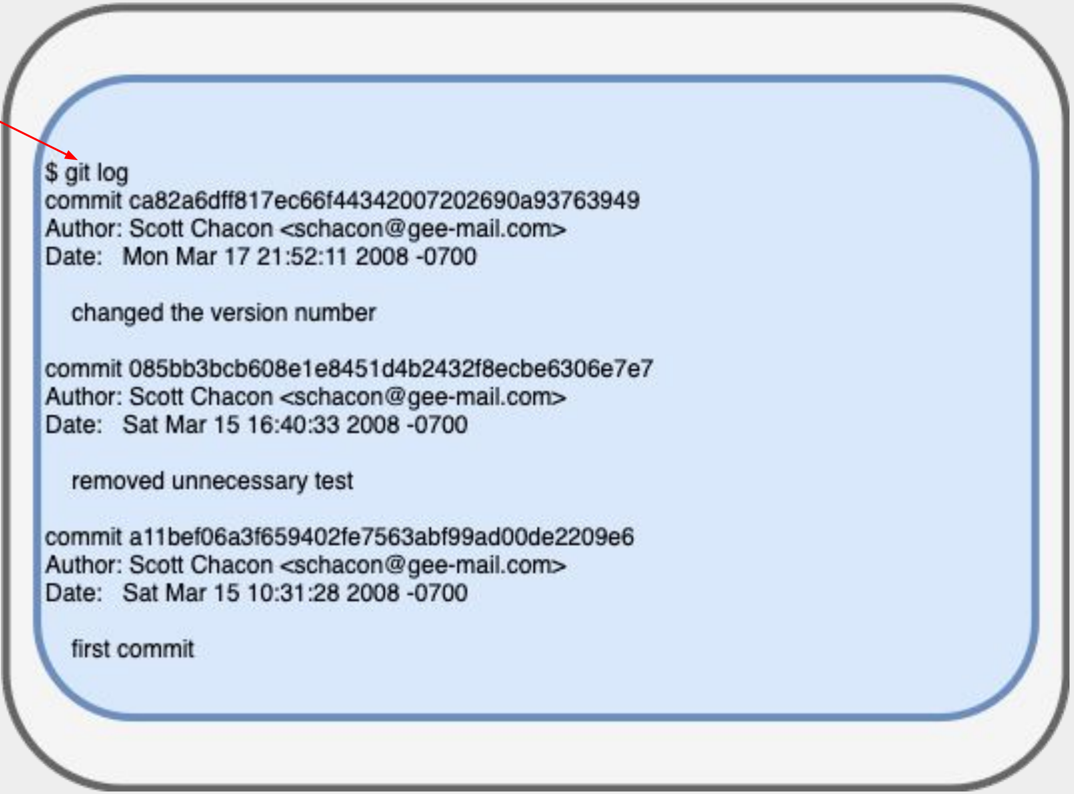


```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

Viewing the Commit History

- After you have created several commits, or if you have cloned a repository with an existing commit history, you’ll probably want to look back to see what has happened. The most basic and powerful tool to do this is the git log command.
- These examples use a very simple project called “simplegit”. To get the project, run
  - $ git clone https://github.com/schacon/simplegit-progit
- When you run git log in this project, you should get an output.

Log Example



```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

Undoing Things

- One of the common undos 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 redo that commit, make the additional changes you forgot, stage them, and commit again using the --amend option:
  - $ git commit --amend
- As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:
  - $ git commit -m 'initial commit'
  - $ git add forgotten_file
  - $ git commit --amend
- You end up with a single commit — the second commit replaces the results of the first.