Git

http://git-scm.com/docs/gittutorial

http://git-scm.com/docs/user-manual.html

https://www.atlassian.com/git/tutorial

Interactive Git Tutorial

For a quick interactive git tutorial I recommend:

http://try.github.io/levels/1/challenges/1

What is Git?

- Distributed version control software.

What is version control software?

Revision control, also known as version control and source control (and an aspect of software configuration management), is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision". For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

The need for a logical way to organize and control revisions has existed for almost as long as writing has existed, but revision control became much more important, and complicated, when the era of computing began. The numbering of book editions and of specification revisions are examples that date back to the print-only era. Today, the most capable (as well as complex) revision control systems are those used in software development, where a team of people may change the same files.

http://en.wikipedia.org/wiki/Revision_control

Overview

1. Getting started
2. Importing a new project
3. Making changes
4. Viewing project history
5. Managing branches
6. Using Git for collaboration
7. Exploring history
8. Other resources
9. Summary

Getting Started

Getting Started: Installation

Install git, preferably via linux:

$[1] sudo apt-get install git

Or homebrew (IMO better than macports) on OSX:

$ brew install git

Or by a GUI and/or the OS that shall not be named:

http://git-scm.com/downloads/guis

[1]$ <something> denotes running it in the command line in a terminal or command prompt.

Getting started: RTFM

RTFM:

$ git help

$ man git

This works for individual commands as well:

$ git help log

$ man git-help

$ git help add

$ man git-add

Getting Started: Basic Settings

After installation, it's good to let git know who you are:

```
$ git config --global user.name "Your Name Goes Here"
$ git config --global user.email you@yourdomain.example.com
```

Importing a new project

Importing a New Project

Say you have source code for a project in a tarball called “myproject.tar.gz”:

```
$ tar xvf myproject.tar.gz
```

Will extract it into a directory called “myproject” (assuming it was compressed properly. We can then do:

```
$ cd myproject
$ git init
```

Which moves us into the “myproject" directory and creates an initial empty repostiory. Git should respond:

```
Initialized empty Git repository in .git/
```

All files related to git will be saved in that hidden directory. This is an improvement over SVN and CVS, for example, which created a hidden file in *every* directory in the project (making them a major mess to clean up).

Importing a New Project

Now that the project has been initialized, we need to make git start tracking things. We can take a snapshot of all files in the current directory with the add command:

```
$ git add .
```

This takes a snapshot of the contents of all files in the "." directory (which is the current working directory) in a temporary staging area called git calls the *index*.

Importing a New Project

You can permanently store the contents of the index with the commit command:

```
$ git commit
```

Which will open up an editor (probably vim) and prompt you for a message. Every commit *requires* a message, in part so you know what you committed and can view your changes later.

Importing a New Project

You can specify the editor git uses for commit messages (and other things) with (see man git-commit[2]):

ENVIRONMENT AND CONFIGURATION VARIABLES

The editor used to edit the commit log message will be chosen from the GIT_EDITOR environment variable, the core.editor configuration variable, the VISUAL environment variable, or the EDITOR environment variable (in that order).

You can also specify the message via command line:

$ git commit -m “YOUR COMMIT MESSAGE HERE.”

[2] or stackoverflow: http://stackoverflow.com/questions/2596805/how-do-i-make-git-use-the-editor-of-my-choice-for-commits

Making Changes

Making changes

After modifying some files (lets call them file1, file2, and file3), you can add their updated context to the index:

```
$ git add file1, file2, file3
```

You are now ready to make another commit.

Making changes

However, before doing the commit we can see what is about to be committed with the diff command:

```
$ git diff --cached
```

If you don’t add the “--cached” argument, git will show you changes you’ve made but not yet added to the index.

Making changes

You can also get a summary of your current git “situation” with:

```
$ git status
```

For our current situation, it will return something like:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    modified:   file1
#    modified:   file2
#    modified:   file3
#
```

Making changes

You can also get a summary of your current git “situation” with:

```
$ git status
```

For our current situation, it will return something like:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    modified:   file1
#    modified:   file2
#    modified:   file3
#
```

Making changes

Alternately, the -a (or --all) argument to the commit command will "Tell the command to automatically stage files that have been modified and deleted, but new files you have not told Git about are not affected."[3]

```
$ git commit -a
```

This merges the above two commands, however it could possibly also commit other tracked files you have modified, but did not want to include.

Note that doing "git diff" will show you the changes that would be committed by "git commit -a".

[3] $ man git-commit

Making changes

Generally, it is good practice to format your git commit messages as follows:

<a short 50 character description>
<a blank line>
<a detailed description>

The first line is treated as the title for the commit. In general this format is very useful for other git commands, like “git format-patch”[4] which converts a commit into an email containing a patch. Github uses this title in it’s web interface.

[4] http://git-scm.com/docs/git-format-patch.html

A note on “git add”

For most version control systems, the add command simply tracks files, and is used to add a new file to the repository.

Git’s add command is more powerful. It adds new content (either a new file or changes to a file) into the index; making them ready for the next commit.

[4] http://git-scm.com/docs/git-format-patch.html

Viewing project history

Viewing project history

You can view the history of your project with the “git log” command:

```
$ git log
```

If you want to see the complete diffs for every commit you can use the “-p” argument:

```
$ git log -p
```

You can also get an overview of each of the changes:

```
$ git log --stat --summary
```

[4] http://git-scm.com/docs/git-format-patch.html

Managing branches

Managing Branches

Git allows you to have multiple *branches* of you code, which allow you to test different changes to the code. You can freely swap between the different branches with some simple commands.

You can create a new branch as follows:

```
$ git branch experimental
```

This will create a new branch called “experimental”.

Managing Branches

You can then use the branch command to get a list of existing branches:

```
$ git branch
```

In our example this would respond:

```
  experimental
* master
```

The * denotes which branch is currently active. The master branch is the default branch created automatically when you make a new git repository.

Managing Branches

You can change branches with the git checkout command:

```
$ git checkout experimental
```

When working on another branch, all commits apply only to that branch. So we can edit a file and then do:

```
$ git commit -a “this is a commit to the experimental branch”
$ git checkout master
```

After this, any changes made to the experimental branch will be gone, as all the files will have changed back to the master branch.

Managing Branches

So now while back on the master branch, we can make some more edits and do another commit:

```
$ (edit files)
$ git commit -a
```

At this point, the two branches have diverged from each other (as they both have commits with different content).

You can merge the changes with the following:

```
$ git merge experimental
```

This will merge the current branch (master) with the experimental branch.

Managing Branches

It's possible that the two branches could have conflicting changes (if the same part of the same file was modified in both). In this case, git will list the conflicting files and tell you to fix the conflicts. You can view the conflicts with:

```
$ git diff
```

After you've edited the files to fix the conflicts, you can commit again (git will generate a message for you automatically):

```
$ git commit -a
```

Managing Branches

Then it is possible to generate a nice graphical display of the resulting history (showing the diverging and merged branches) with:

```
$ gitk
```

Managing Branches

If you are done with a branch and want to delete it, this is possible with:

```
$ git branch -d experimental
```

This will only work if the changes in the specified branch to be deleted have been merged and are present in the current branch.

You can force a delete and trash changes (without requiring them to be merged into the current branch) with:

```
$ git branch -D crazy-idea
```

Branches are easy (and cheap), so they can be a good way to test ideas out without breaking anything.

Using git for collaboration

Using git for collaboration

So the big benefit of git is that it allows *distributed* version control. These examples assume that the different repositories are on the same system, however remote repositories can also be specified with https:// or ssh://

Using git for collaboration

So suppose that alice has a git repository in:

/home/alice/project

And bob (a user on the same machine) wants to contribute to it. Bob can clone alice's project with the "git clone" command:

bob$[5] git clone /home/alice project myrepo

This will create a copy of alice's repository in Bob's "./myrepo" directory.

[5] bob$ assumes a terminal in bob's home directory.

Using git for collaboration

After Bob makes some changes to his local directory, he can commit them as normal, and this will only effect his local repository:

```
(edit files)
bob$ git commit -a
(rinse, repeat)
```

When Bob has a version of his code that Alice wants to use, Alice can then pull it into her repository (hence the *distributed* version control):

```
alice$ cd /home/alice/project
alice$ git pull /home/bob/myrepo master
```

This will merge the changes Bob made in his master branch into Alice's current branch (similar to the merge command). Alice will need to have her own changes committed before the pull (similar to doing a merge).

Using git for collaboration

The pull command essentially does two commands, a fetch (to get a copy of the other data) and then a merge. It is possible to do a fetch on its own to take a look at the changes in the remote repository.

```
alice$ git fetch /home/bob/myrepo master
alice$ git log -p HEAD..FETCH_HEAD
```

This can be done even if Alice has uncommitted local changes. The FETCH_HEAD index is a special index created by the fetch command. HEAD refers to the current state in the current branch.

“HEAD..FETCH_HEAD” means show “me everything that is reachable from the FETCH_HEAD but not reachable from HEAD”. Essentially, it shows everything that happens between HEAD and FETCH_HEAD.

Using git for collaboration

After doing a fetch (or the pull) these changes can also be visualized with gitk, with the same notation:

```
gitk HEAD..FETCH_HEAD
```

Using git for collaboration

If you're using with a common remote repository, you can give it a name to use as shorthand:

```
alice$ git remote add bob /home/bob/myrepo
alice$ git fetch bob
```

Would be the same as:

```
alice$ git fetch /home/bob/myrepo
```

Using git for collaboration

When a name is given to a remote repository, fetch will store it locally (in a remotes branch) so it can be used for comparisons:

```
alice$ git fetch bob
alice$ git log -p master..bob/master
```

This would show the changes between the current repositories master branch, and the bob repositories master branch.

Merges can then be done similarly, using the remote repositories name:

```
alice$ git merge bob/master
```

Using git for collaboration

So in this case, this merge can also be done from the branch stored in remotes:

```
alice$ git pull . remotes/bob/master
```

Note that git pull always merges into the current branch, regardless of what is given on the command line.

Using git for collaboration

Bob can later update his repo with Alice's changes using:

```
bob$ git pull
```

The repository doesn't need to be specified, as the pull command will use what the repository was initially cloned from by default. If he forgets where this is, he can find the location used for pulls with:

```
bob$ git config --get remote.location.url
/home/alice/project
```

Using git for collaboration

If Bob later wants to use a different host for the source for pulls, this can be changed with:

```
bob$ git clone alice.org:/home/alice/project myrepo
```

Which will use the ssh protocol to grab the data from the remote location.

Git also has its own protocol, or can use rsync or http/https for pulls.[6]

[6] http://git-scm.com/docs/git-pull.html

Exploring history

Exploring History

The history of a git repository is made up of its interrelated commits. This is shown with the git log command:

```
deselt$ git log
commit 9c13563d2c6e36084689dbef37eb8edcd3d171ab
Author: Travis Desell <travis.desell@gmail.com>
Date:   Fri Sep 6 14:32:45 2013 -0500

    Local changes to OSX app

commit de6af505e6f97797e5413300e9ddef92f54c41e3
Merge: a1e0488 fed6c3b
Author: Kyle Goehner <KyleA.Goehner@gmail.com>
Date:   Thu Apr 25 21:28:43 2013 -0500

    Added two more feature sets.

commit a1e0488d8ed2823985cec6e4d19e9feda44c26ab
Author: Kyle Goehner <KyleA.Goehner@gmail.com>
Date:   Thu Apr 25 21:19:31 2013 -0500

    Updated surf file.
```

Note that the first line of each of those commits is the name for the commit.

Exploring History

If we know the name of a commit, we can use “git show” to see details about the commit:

```
$ git show 9c13563d2c6e36084689dbef37eb8edcd3d171ab
```

Exploring History

There are a number of other ways to show commits with the show command:

```
$ git show c82a22c39c    # the first few characters of the
                         # name are usually enough
$ git show HEAD          # the tip of the current branch
$ git show experimental  # the tip of the "experimental" branch
```

Exploring History

Commits have *parents* which refer to the previous state(s) or previous commit(s) of the project:

```
$ git show HEAD^        # to see the parent of HEAD
$ git show HEAD^^       # to see the grandparent of HEAD
$ git show HEAD~4       # to see the great-great grandparent
                        # of HEAD
```

Exploring History

A commit after a merge will have multiple parents (the current branch and the merged branch):

```
$ git show HEAD^1    # show the first parent of HEAD
                     # (same as HEAD^)
$ git show HEAD^2    # show the second parent of HEAD
```

Exploring History

You can tag commits with names of your own:

```
$ git tag v2.5 1b2e1d63f
```

After this you can refer to the commit 1b2… as v2.5, which makes life a lot easier. If you want other people to be able to refer to this tag, you can create a tag object for it[7].

[7] http://git-scm.com/docs/git-tag.html

Exploring History

Any git command that uses git commit names can use the tag instead:

```
$ git diff v2.5 HEAD         # compare the current HEAD to v2.5
$ git branch stable v2.5     # start a new branch named “stable"
                             # based at v2.5
$ git reset --hard HEAD^ # reset your current branch and
                         # working directory to its state at
                         # HEAD^
```

Be *very* careful with that last command: in addition to losing any changes in the working directory, it will also remove all later commits from this branch. If this branch is the only branch containing those commits, they will be lost.

“git reset” should not be used on a publicly-visible branch with other developers, as it will force them to do needless merges to clean up the history. If you need to undo changes that you have pushed, use git revert[8] instead.

[8] http://git-scm.com/docs/git-revert.html

Exploring History

You can use the “git grep” command just like the UNIX grep command. It will search through the files specified for the given text:

```
$ git grep “hello” v2.5      # search for all occurrences of
                             # hello in the v2.5 tag
$ git grep “hello”           # search through all files managed
                             # in the repository.
```

Exploring History

Many git commands use commit names (or tags). This can be very handy, for example:

```
$ git log v2.5..v2.6                # commits between v2.5 and
                                    # v2.6
$ git log v2.5..                    # commits since v2.5
$ git log --since="2 weeks ago"     # commits from the last 2 weeks
$ git log v2.5.. Makefile           # commits since v2.5 which
                                    # modify Makefile
```

Exploring History

"git log" can also work on ranges of commits (something1..something2) where something1 is not necessarily an ancestor of something2. If branches *master* and *stable* diverged from a commit some long time ago, then:

```
$ git log stable..master
```

will list the commits made in the master branch but not in the stable branch; while:

```
$ git log master..stable
```

will list the commits made in the stable branch but not in the master branch.

Exploring History

git log only shows commits in a list, however gitk (like shown before) will display the commits graphically in a graph, which can be much more useful; both take the same command line arguments.

Exploring History

git log only shows commits in a list, however gitk (like shown before) will display the commits graphically in a graph, which can be much more useful:

```
$ gitk --since="2 weeks ago" drivers/
```

Shows the last two weeks of commits that modified files in the “drivers” directory.

Exploring History

Most of these commands also will take filenames, so you can view changes to a specific file:

```
$ git diff v2.5:Makefile HEAD:Makefile.in
```

Will show the differences between Makefile in v2.5 and HEAD.

```
$ git show v2.5:Makefile
```

Will show Makefile from the commit with tag v2.5.

Summary

Summary

You can think of git as peer-to-peer or distributed version control.

Unlike other version control systems (like SVN and CVS) which have a central repository that other people commit changes to (and get changes from); with git, everyone keeps their own repository.

With everyone having their own repository, git provides the tools to push, pull and merge these remote repositories together.

Other resources

Other resources

I'd recommend walking through this quick 15 minute interactive tutorial:

http://try.github.io/

There's a very extensive git tutorial here:

https://www.atlassian.com/git/tutorial

Github is great for saving a remote and web viewable git repository (note github is ***not*** git, just a nice interface and place to store your repositories).

http://github.com