Git for the lazy

From Spheriki

git (http://git.or.cz/) is a distributed version control system (http://en.wikipedia.org/wiki/Distributed_revision_control) . No, you don't need to know what that means to use this guide. Think of it as a time machine: Subversion or CVS without the suck.

If you make a lot of changes, but decided you made a mistake, this will save your butt.

This guide is for people who want to jump to any point in time with their project/game/whatever, and want something to use for themselves.

Contents

Install git

Windows

1. Download Cygwin (http://www.cygwin.com/) .
2. Put `setup.exe` in a folder of its own in your documents.
3. Launch `setup.exe`.
4. While installing Cygwin, pick these packages:
   - **git** from the DEVEL category
   - **nano** (if you're wimpy) or **vim** (if you know it), both in the EDITORS category

You'll now have a shortcut to launch Cygwin, which brings up something like the Linux terminal.

Linux

Install the `git` package using your preferred method (package manager or from source).

Introduce yourself to git

Fire up your Cygwin/Linux terminal, and type:

```
git config --global user.name "Joey Joejoe"
git config --global user.email "joey@joejoe.com"
```

You only need to do this once.

Start your project

Start your project using the Sphere editor, or from a ZIP file, or just by making the directory and adding files yourself.

Now `cd` to your project directory:

```
cd myproject/
```

Tell git to start giving a damn about your project:

```
git init
```

... and your files in it:

```
git add .
```

Wrap it up:

```
git commit
```

Now type in a "commit message": a reminder to yourself of what you've just done, like:

```
Initial commit.
```

Save it and quit (type `Ctrl+o Ctrl+x` if you're in nano, `:x` if you're in vim) and you're done!

Work in bits

When dealing with git, it's best to work in small bits. Rule of thumb: if you can't summarise it in a sentence, you've gone too long without committing.

This section is your typical work cycle:

1. Work on your project.
2. Check which files you've changed:

```
git status
```

3. Check what the actual changes were:

```
git diff
```

4. Add any files/folders mentioned in step 2 (or new ones):

```
git add file1 newfile2 newfolder3
```

5. Commit your work:

```
git commit
```

6. Enter and save your commit message. If you want to back out, just quit the editor.

Repeat as much as you like. Just remember to always end with a commit.

Admire your work

To see what you've done so far, type:

```
git log
```

To just see the last few commits you've made:

```
git log -n3
```

Replace 3 with whatever you feel like.

For a complete overview, type:

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

Browse at your leisure.

View changes

To view changes you haven't committed yet:

```
git diff
```

If you want changes between versions of your project, first you'll need to know the commit ID for the changes:

```
git log --pretty=oneline
```

```
6c93a1960072710c6677682a7816ba9e48b7528f Remove persist.clearScriptCache() function.
c6e7f6e685edbb414c676df259aab989b617b018 Make git ignore logs directory.
8fefbce334d30466e3bb8f24d11202a8f535301c Initial commit.
```

The 40 characters at the front of each line is the commit ID. You'll also see them when you `git commit`. You can use it to show differences between commits.

To view the changes between the 1st and 2nd commits, type:

```
git diff 8fef..c6e7
```

Note how you didn't have to type the whole thing, just the first few unique characters are enough.

To view the last changes you made:

```
git diff HEAD^..HEAD
```

How to fix mistakes

Haven't committed yet, but don't want to save the changes? You can throw them away:

```
git reset --hard
```

You can also do it for individual files, but it's a bit different:

```
git checkout myfile.txt
```

Messed up the commit message? This will let you re-enter it:

```
git commit --amend
```

Forgot something in your last commit? That's easy to fix.

```
git reset --soft HEAD^
```

Add that stuff you forgot:

```
git add forgot.txt these.txt
```

Then write over the last commit:

```
git commit
```

Don't make a habit of overwriting/changing history if it's a public repo you're working with, though.

For the not so lazy

Just some extra reading here. Skip it if you're lazy.

Writing good commit messages

This part is all opinion, but worth reading.

Your first line should be a summary of the commit changes in a single sentence. It should be 50 characters or less. It should be in present tense: this matches up with git's merge commit messages, which you haven't met yet, but you'll eventually run into when you hit branching and merging.

The remaining body should go into more detail if needed. I use point form: a space, an asterisk (*), another space, followed by the point in detail.

e.g.

```
Add feature X to subsystem Y.

 * Feature X isn't working well with feature Z. Worth investigating.
 * Feature X still doesn't work for inputs A, B and C.
```

Ignoring files

When you check your project status, sometimes you'll get something like this:

```
git status
```

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       bleh.txt
#       module.c~
nothing added to commit but untracked files present (use "git add" to track)
```

If you don't want git to track these files, you can add entries to `.gitignore`:

```
nano .gitignore
```

And add the files you want ignored:

```
bleh.txt
*~
```

The first line ignores `bleh.txt` the second line ignores all files and directories ending with a tilde (~), i.e. backup files.

You can check if you got it right:

```
git status
```

```
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   .gitignore
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Don't forget to commit your changes to `.gitignore`!

```
git add .gitignore
git commit
```

With something like this for your commit message:

```
Make git ignore bleh.txt and backup files.
```

Use `.gitignore` to keep your messages clean, and stop git from bugging you about stuff you don't care about. It's a good idea to ignore things like executable binaries, object files, etc. Pretty much anything that can be regenerated from source.

Branching and merging

A branch is a separate line of development. If you're going to make a bunch of changes related to a single feature, it might be a good idea to make a "topic branch": a branch related to a topic/feature.

To make a new branch:

```
git branch feature_x
```

To view the current branches:

```
git branch
```

```
  feature_x
* master
```

The asterisk (*) shows your current branch. *master* is the default branch, like the trunk in CVS or Subversion.

To switch to your new branch, just type:

```
git checkout feature_x
```

If you check the branches again, you'll see the switch:

```
git branch
```

```
* feature_x
  master
```

Now go through the usual edit/commit cycle. Your changes will go onto the new branch.

When you want to put your branch changes back onto *master*, first switch to *master*:

```
git checkout master
```

Then merge the branch changes:

```
git merge feature_x
```

This will combine the changes of the *master* and *feature_x* branches. If you didn't change the *master* branch, git will just "fast-forward" the *feature_x* changes so *master* is up to date. Otherwise, the changes from *master* and *feature_x* will be combined.

You can see the commit in your project's log:

```
git log -n1
```

If you're happy with the result, and don't need the branch any more, you can delete it:

```
git branch -d feature_x
```

Now when you see the branches, you'll only see the *master* branch:

```
git branch
```

```
* master
```

You can make as many branches as you need at once.

Tags

If you hit a new version of your project, it may be a good idea to mark it with a tag. Tags can be used to easily refer to older commits.

To tag the current version of your project as "v1.4.2", for example:

```
git tag v1.4.2
```

You can use these tags in places where those 40-character IDs appear.

What now?

git can help with working with other people too. Of course, then you *do* have to learn about distributed version control. Until then, just enjoy this page.

But if you want to learn:

- Pro Git online book (http://progit.org/book/)
- gittutorial manual page online (http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html)
- Everyday git with 20 commands or so (http://www.kernel.org/pub/software/scm/git/docs/everyday.html)
- The official git web site (http://git.or.cz)

Main git selling points (ripped off the main site):

- Distributed development, i.e. working with other people.
- Strong support for non-linear development, i.e. working with other people *at the same time!*
- Efficient handling of large projects, i.e. fast!
- Cryptographic authentication of history, for the paranoid.
- Scriptable toolkit design, you can script pretty much any git task.

If something doesn't seem right or is confusing, contact me at my blog (http://tunginobi.spheredev.org/site/) . --tunginobi 10:14, 28 February 2009 (GMT)

Retrieved from "http://www.spheredev.org/wiki/Git_for_the_lazy"

Category: Tutorials
