Gitting started with git for designers

A practical guide for learning the git version control system for people without version control experience.

Version control—also known as *source control* or *revision control*— is the best thing since sliced bread, but **why?**

- Communicate your work to other colleagues
- Share code amoungst your team
- Maintain "production" versions of code that are always deployable
- Simultaneously develop new features on the same codebase
- Keep track of changes in files over time

WHAT IS VERSION CONTROL?

Version control is essentially a communication tool, just like email or IM — but it works with code rather than human conversation.

At its simplest, a version control system can look like a backup device. Every file in the system has a full history of changes and can easily be restored to any version in its history.

There are many different programs for version control. This document is based on *git*, but you may be aware of Subversion (svn), CVS, darcs, Mercurial or others. Each has a slightly different strategy for operation.

Repository Structure

Every version control system starts with a *repository* where the files and their versions live. A repository works like a database; it can return any version of any file within, a history of changes for a given file, or a history of changes across the entire project.

```
#25 Joe  Adjust user profile information
#24 Fred Add login box
#23 Mary Allow user photo uploads
#22 Joe  Change the color of the header to yellow
#21 Mary Change the header to blue
```

The repository's users can *check out* a *working copy* — a copy of the latest files to which users can make changes. After making some changes, they can then *check in* (or *commit*) their changes back to the repository, which creates a new version with metadata about the files that were changed and the person who changed them.

With git, each user has a full copy of the repository on their local machine. Generally, you will commit many changes to your local repository and when you are ready to share your work, *push* your work to a shared repository for your team. You can also *pull* changes from other repositories.

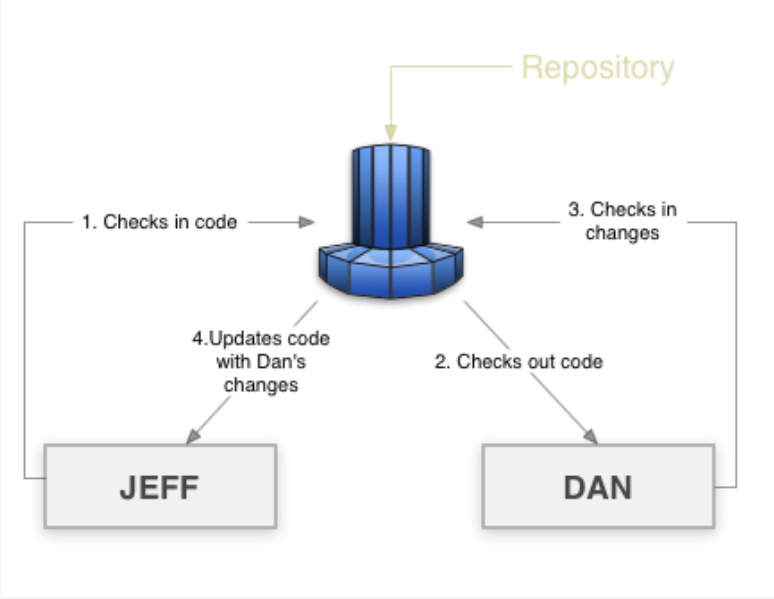

Figure 1: A basic source control system

Workflow

ATTACK OF THE CLONES

To get a working copy of your codebase, you need to *clone* a remote repository to your local machine. Cloning creates the repository and checks out the latest version, which is referred to as `HEAD` .

Let's clone an open-source project. Run this in your terminal:

```
$ git clone git://github.com/wycats/jspec.git
Initialized empty Git repository
```

Congratulations, you just cloned your first repository! The clone command sets up a few convenience items for you; it keeps the address of the original repository, and aliases it as *origin*, so you can easily send back changes (if you have authorization) to the remote repository.

You will now have a folder `jspec` in the current directory. If you `cd` into that directory, you should see the contents of the JSpec source code (it's just a few files).

Git can run over many protocols, including "git://" as above (most public projects will use git://). By default, git uses the ssh protocol, which requires you have secure access to the remote repository.

```
$ git clone user@yourserver.com:thing.git
```

MAKING CHANGES

Now that you have a working copy, you can start making changes. There is nothing magic about editing the files, so all you need to do is edit whatever file you're working with and then save it. Once the file is saved, you'll need to *add* the change to the current revision (or, more typically, you'll make changes to several files and add them to the current revision all at once). To do so, you need to `git add` the changed file. This is also known as "staging". You can add single files:

```
$ git add index.html
```

Or, you can add an entire directory at once:

```
$ git add public/
```

This will add any files in the `public/` directory to the staging area.

Or, add the current directory:

```
git add .
```

If you make any changes to the file after staging (before committing), you'll need to `git add` the file again.

The `git status` command shows you the current status of the repository.

```
$ git status
# On branch master
# Changes to be committed:
#  (use "git reset HEAD <file>..." to unstage)
#
# modified:  public/index.html
```

The `git diff` command shows you a view of what's changed. By default, it shows changes that haven't been staged. Adding the flag "--cached" will show you the staged changes only.

```
  $ git diff --cached
  diff --git a/public/index.html. b/public/index.html
  index a04759f..754492a 100644
  --- a/public/index.html
  +++ b/public/index.html
  @@ -8,7 +8,6 @@ revision control or source code management, is a system
that maintains versions
```

```
+ This a line that I added to the file
- This is a line I removed from the file
```

This output is called a `diff` or `patch` and can be emailed to co-workers so they can apply your changes to their local codebase. It's also human-readable: it shows you the filenames, the line numbers within the file, and changes with + and `-` symbols. You can *pipe* the diff into a file.

```
  $ git diff --cached > line_modify.patch
```

Aside: `.gitignore`

The `.gitignore` file lets you tell Git to ignore certain files or directories. This setting is useful for things like generated binaries, log files, or files with local passwords in them.

COMMIT TO SOMETHING IN YOUR LIFE

When you get your changes just the way you want them added to the current revision, then you need to commit that revision to your local repository. To do this, you'll need to run `git commit`. When you execute this command, a text editor will appear, with a list of files that have changed and some blank space at the top.

In that blank space, you need to describe what you've changed so your co-workers can tell at a glance what you've done. You'll need to enter something better than "stuff", but there's no need to go overboard and do something like:

```
Changed line 434 in index.html to use spaces rather than tabs.
Changed line 800 in products.html.erb to have two spaces between
the tags.
Changed line 343, 133, 203, 59, and 121 to have two spaces at the
start rather than 9.
```

A short description of what you changed will suffice. Concise commit messages are an art form, much like haiku.

```
Minor formatting changes in the code.
```

It is accepted custom to write one line of summary (less than 80 characters), a blank line, then a third line describing in more detail. The second and third lines are entirely optional.

Once your commit message is to your liking, then save the file and exit the text editor. It will commit to your local repository, and you can continue to make further changes.

PUSH BACK

Once your changes are committed to your local repository, you should push them to a shared repository so others can pull your changes. `git push` will push all the commits from your local repository up to the remote repository.

Git push takes several arguments: `git push <repository> <branch>` In this case, we want to push changes back to the original repository, which is aliased as `origin`, to the master branch.

```
$ git push origin master
```

Fortunately for our fingers, `git push` (and `git pull`) will default to pushing, and pulling, all branches common to the origin and the local repository.

When you execute `push`, you should see output similar to the following:

```
$ git push
updating 'refs/heads/master'
 from fdbdfe28397738d0d42eaca59c6866a87a0336e2
 to  1c9ec11f757c099680336875b825f817a992333e
 Also local refs/remotes/origin/master
Generating pack...
Done counting 2 objects.
Deltifying 2 objects...
 100% (2/2) done
Writing 2 objects...
 100% (2/2) done
Total 2 (delta 3), reused 0 (delta 0)
refs/heads/master: fdbdfe28397738d0d42eaca59c6866a87a0336e2 ->
1c9ec11f757c099680336875b825f817a992333e
```

This output says that you've got your files ready to be pushed (`Generating pack`) and the remote repository has received your files (`Writing 2 objects`). The remote repository has updated its `head/master` (the "main" branch of the repository) to point to the revision you just committed so that it knows it's the latest set of changes committed. Now others can update their local copies to be in sync with the changes that you've made. But how do you do that?

GET UPDATES FROM AFAR

To update your local repository and working copy to the latest revision committed to the remote repository, you need to execute `git pull`. This pulls all of the changesets down from the remote repository and merges them with your current changes (if there are any).

When you execute a `git pull,` the output should look something like the following:

```
  remote: Generating pack...
  remote: Done counting 12 objects.
  remote: Result has 8 objects.
  remote: Deltifying 8 objects...
  remote: 100% (8/8) done
  Unpacking 8 objects...
  remote: Total 8 (delta 4), reused 0 (delta 0)
   100% (8/8) done
   refs/remotes/origin/master: fast forward to branch 'master' of git@
yourco.com:git_project
   old..new: 0c793fd..fdbdfe2
  Auto-merged file.cpp
  Merge made by recursive.
   .gitignore               |  2 ++
   file.cpp                 |  8 ++++++--
   src/things.html          |  5 +++--
   your_file.txt            |  18 ++++++++++++++++++
   4 files changed, 19 insertions(+), 4 deletions(-)
   create mode 100644 .gitignore
   create mode 100644 your_file.txt
```

What's happened is more or less a `push` in reverse. The remote repository has prepared (`Generating pack`) and transferred the changes (`Unpacking 8 objects`) to your local repository. Your local repository then takes the changes and implements them in the same order as they were committed (e.g., merging them as the example shows for `file.cpp` or creating them like `.gitignore` or `your_file.txt`).

Branches

Branches fulfill the same role as drafts when writing an email. You work on the draft, saving it frequently until it is complete. then when it's done, you send the email and the draft is deleted. In this case, the outbox is not polluted by your frequent changes, until you hit "send".

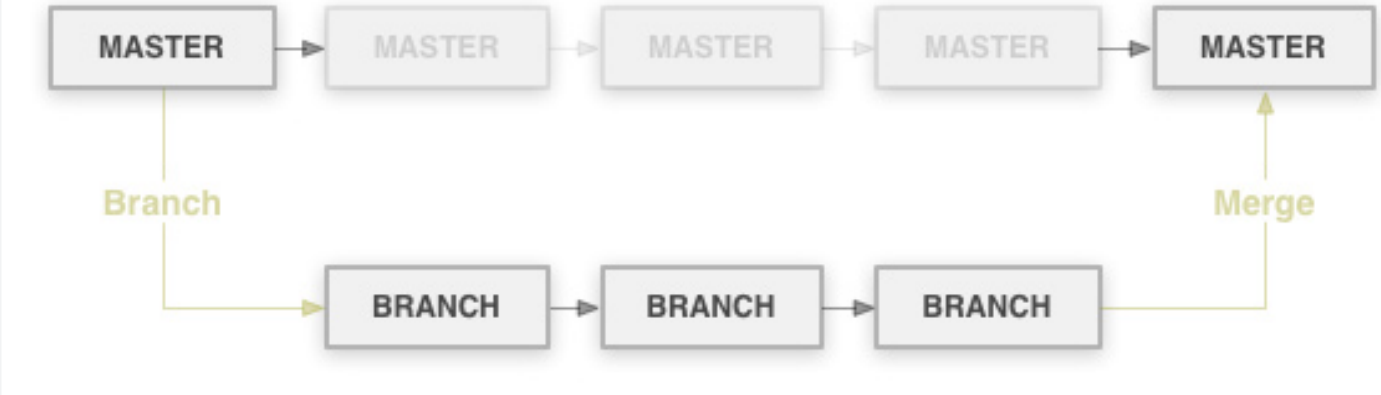

Figure 2: Branching and merging

WHY YOU SHOULD BRANCH

Branching is useful when developing new features, because it allows the *master* branch—the outbox—to be always working and deployable. There may be any number of drafts—experimental branches—in active development. Branches are easy to create and switch between.

You should always create a branch before starting work on a feature. This way, the master will always be in a working state, and you'll be able to work in isolation of other's changes. Creating a branch allows you to take the `master` branch, *"clone"* it, and make commits to that clone. Then when you're ready, you can merge

the branch back into `master` ; or, if there are changes made to master while you're working, you can merge those changes. It's just like pushing and pulling, but it all happens in the same directory.

Once the code in a branch is finished, the changes are *merged* into the master branch and the branch is removed, just like the email draft. If someone commits code to the master branch, it's easy to update the branch to the latest master code.

Branching is a great way for two people to work together on something that requires isolation from the main code base. This could be anything from code that will have permanent results, like a really big code refactoring or site redesign, to things that are only temporary, like performance testing.

Aside: What branch am I in?

To see your current branch, and a list of all local branches, execute `git branch`.

CREATING A BRANCH

To create a branch in Git, you execute `git checkout -b <branch name>.` Any modified files will be listed.

```
$ git checkout -b redesign
M public/index.html
Switched to a new branch "redesign"
```

You've now checked out the redesign branch. To switch back to master:

```
$ git checkout master
M public/index.html
Switched to a new branch "master"
```

You'll find it useful to push the branch on the remote repository so others can pull your changes:

```
$ git push origin redesign
```

This tells git to push all changes in the redesign branch in the local repository to the redesign branch in the remote repository.

If you need to update changes made in `master` to your branch (`redesign`) (e.g., important code changes, security updates, and so on), then you can do so using *git pull* and *git merge* like so:

```
git pull origin
git merge master
```

The first command tells git to *pull* all the changes from the `origin` repository, including all branches. The second command tells git to *merge* the master branch into your branch.

This updates `redesign` with the changes from `master`.

When you are ready to merge your branch into `master`, you need to checkout `master` and then merge the branch into it like so:

```
git checkout master
git merge redesign
```

This updates `master` with the changes from `redesign`.

If you're finished with the `redesign` branch you created, then you can delete it using the `-d` parameter.

```
git branch -d redesign
```

To delete the branch on the remote repository, you hijack the push command (remember you can push a local branch to a different remote branch with `git push <remote> <local branch>:<remote branch>`) and send an empty local branch to the remote branch.

```
git push origin :redesign
```

This strange syntax can be explained by thinking about pushing an empty set to the remote repository. It will overwrite your changes with nothing — erasing the branch on the remote repository.

Useful git commands

UNDOING YOUR CHANGES

If you want to revert a file that has changes that are staged, use `git reset HEAD <filename>` .

If you want to revert a file that does not have changes that are staged back to the copy in the repository, just check it out again with `git checkout <filename>`. If you want to revert to a specific version of a file, use `git checkout` as well. You will need to know the revision ID, which you can find with `git log`

```
$ git log index.html
commit 86429cd28708e22b643593b7081229017b7f0f8d
Author: joe <joe@example.com>
Date:  Sun Feb 17 22:19:21 2008 -0800
build new html files
```

```
commit 3607253d20c7a295965f798109f9d4af0fbeedd8
Author: fred <fred@example.com>
Date:  Sun Feb 17 21:32:00 2008 -0500
Oops.
```

To revert the file back to the older version (360725...) you execute checkout. Git will stage the older version for you, ready for review and commit.

```
$ git checkout 3607253d20c7a295965f798109f9d4af0fbeedd8 index.html
```

If you no longer want to restore this older version, you can unstage the file and checkout again

```
$ git reset HEAD index.html
$ git checkout index.html
```

Or in one command

```
$ git checkout HEAD index.html
```

Have you noticed that HEAD is interchangeable with the revision number? That's because with git, revisions and branches are effectively the same thing.

WHO WROTE THAT LINE?

Run `git blame <file>` to see who changed a file last, and when.

Some useful tools

VIEW THE COMPLETE TREE

There are several applications you can use to see a detailed history of your working copy.

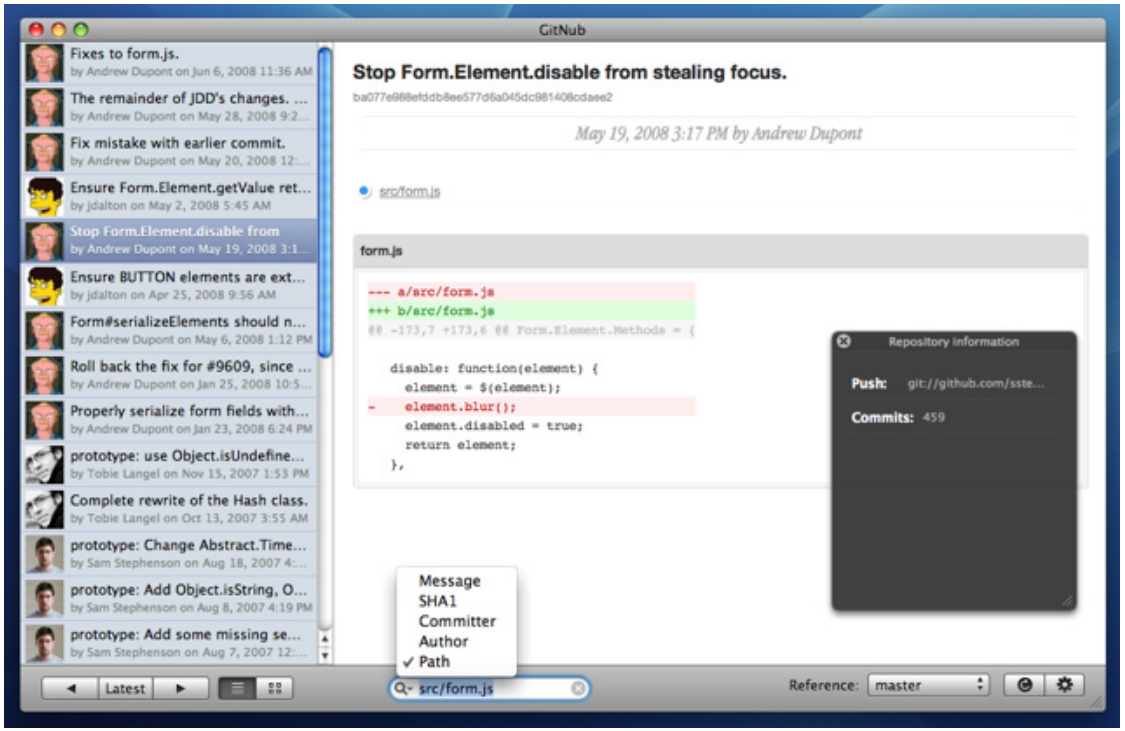

Figure 4: Sample GitNub screenshot

Gitnub is a Gitk-like application written in RubyCocoa by Justin Palmer that looks like it belongs on a Mac. Available for download at http://wiki.github.com/Caged/gitnub/home

GitX is a git GUI specifically for Mac OS X. It currently features a history viewer much like gitk and a commit GUI like git gui. But then in silky smooth OS X style! Available for download at http://wiki.github.com/pieter/gitx.

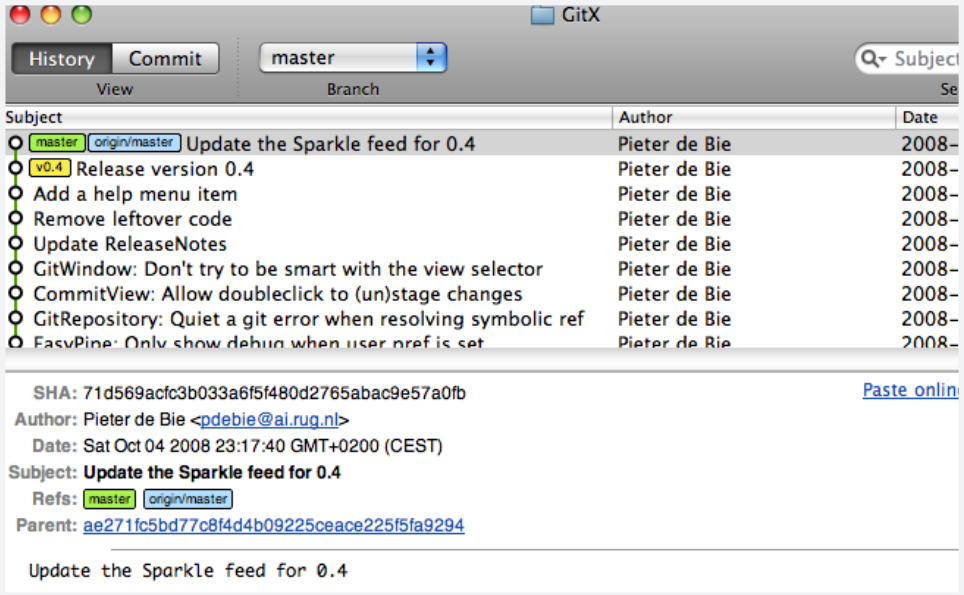

Figure 5: Sample GitX screenshot

The **gitk** application allows you to navigate through the tree of changes, view diffs, search old revisions, and more. This application comes bundled with the git repo.

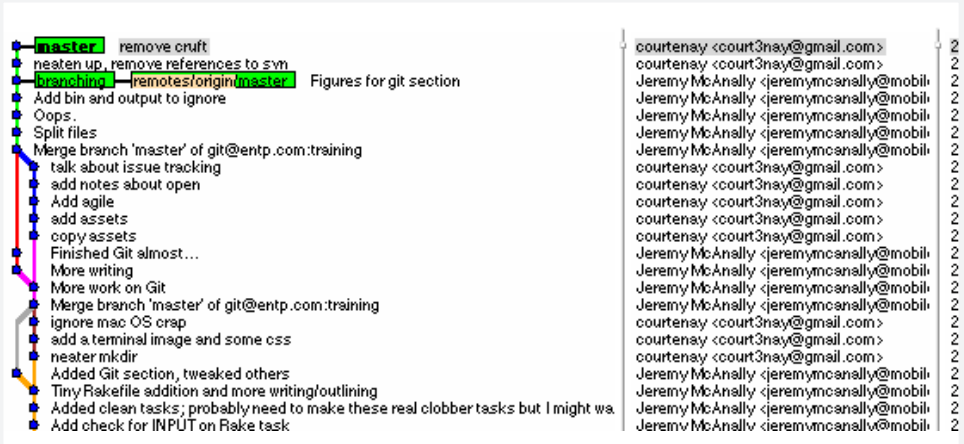

Figure 6: Sample gitk screenshot

Best practices

We thought we'd wrap up this section with just a few little hints and tips that can go a long way when working with version control systems.

COMMIT OFTEN

Just like people always say "Save often or you'll regret it" when working with word processors, you should commit to your local repository as often as possible. Not only does it keep you from possibly losing work (which you shouldn't if you follow the first piece of advice!), it will give you the security that you can step back at any time if you should need to. Of course, committing commit after commit every commit wocommitrd or lcommitecommittcommittcommitecommitrcommit could be a little excessive, at every "major" (for any or all definitions of major) step you take in your work, you should commit.

PULL OFTEN

Conversely, you should also pull often. Pulling often keeps your code up to date and, hopefully, cuts down on duplicated work. It's always frustrating when you spend hours working on a feature when your co-worker has already implemented it and pushed it to the repository, but you didn't know anything about it because you pull every 3 weeks.

USE `CHECKOUT` AND `RESET` WITH CAUTION

To revert any local changes you've made to a specific file since your last commit, you can use `git checkout <filename>`, or you can use `git reset` to kill all changes since your last commit. Having the ability to step back is a great tool (especially if you realize you're going down the totally wrong path), but it's definitely a double edged sword. Once your changes are gone, they're gone, so be careful! It's terrible when you realize you've just wasted a few hour's work by a wreckless reset .

CREATE YOUR OWN REPOSITORY ANYWHERE

If you want to get some version control on a simple local project (i.e., it doesn't have a big remote repository or anything), then you can simply use git init to create your own standalone local repository. For example, if you're working on some design concepts for a new application, then you could do something like the following:

```
mkdir design_concepts
git init
```

Now you can add files, commit, branch, and so on, just like a "real" remote Git repository. If you want to push and pull, you'll need to set up a remote repository.

```
git remote add <alias> <url>
git pull <alias> master
```