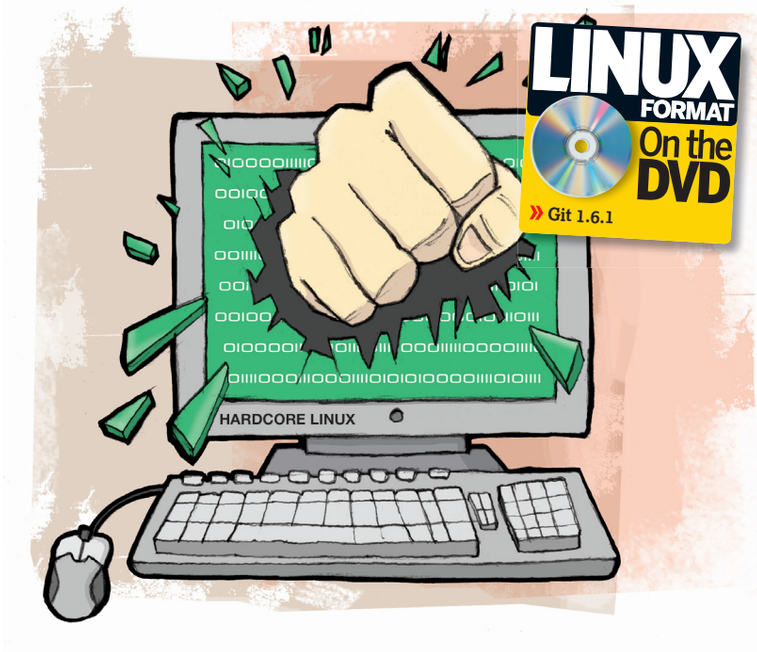
Git: Versioning

What can the creator of Linux do to software versioning? He can make it simple and at the same time completely confusing, says **Dan Frost**.



Our expert

Dan Frost is technical director of 3ev, a Brighton-based web development agency. He has developed for the *TYPO3* CMS project and is currently working on the 'Involve' CMS application for 3ev.

Anyone who's used *SVN*, *CVS* or one of the other major versioning tools for backing up, reverting and (trying to) collaborate will understand what drove Linus Torvalds to give in and write his own. Versioning software often drives coders mad – and *Git* has been called the versioning system that you need a PhD to understand. The complexity comes from its simplicity and its aim to make it easier to work in large groups of distributed developers, but if you approach *Git* calmly you'll get a lot from it.

Versioning software traditionally has a central server with contributors pushing content from their own working environments, dubbed 'sandboxes'. *CVS* and *SVN*, two of the biggest open source versioning systems, use this model and most of the software you've downloaded from SourceForge will have used one or other of these systems.

It works very well. Each user takes a copy of the project from the central repository and does some work, fixes some bugs or adds a feature. When the user has finished, they commit their changes back to the central repository, together with a message telling everyone else what they've done.

This model is easy to get your head around, but it doesn't scale in one important way. Open source projects are maintained by many geographically disparate developers working on different features at different times and often in an order that doesn't make sense. 'Branching' enables the code to split into several concurrent versions – one for the user interface updates, one for the file storage optimisation and perhaps one for the latest stable version. However, it gets very annoying if the head developer who manages the central repository won't give you a branch in the repository for your current project... Enter *Git*.

SVN made branching cheaper – you can very quickly create and work on a new version of the source. *Git* makes it cheap and cheerful – you can create a branch that's available only in your local repository, so you can keep track of dozens of ideas each in their own branch before pushing them back to the central repository.

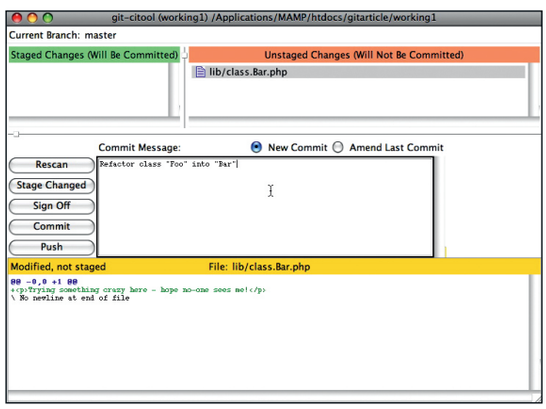
Getting started

If *Git* isn't already running in your installation then a simple **apt-get install git** or **yum install git** should do it. Once you have *Git* installed, open up a terminal, type **git** and see what you have:

```
~ $ git
usage: git [--version] [--exec-path[=GIT_EXEC_PATH]] [-pl--paginatel--no-pager] [--bare] [--git-dir=GIT_DIR] [--work-tree=GIT_WORK_TREE] [--help] COMMAND [ARGS]
The most commonly used git commands are:
add         Add file contents to the index
apply       Apply a patch on a git index file and a working tree
archive     Create an archive of files from a named tree
bisect      Find the change that introduced a bug by binary search
...etc
```

Let's start by creating a very simple project and initialising *Git* on it. Type the following:

```
cd ~
mkdir myproject && cd myproject
```



› ***Git*** **comes with a few simple graphical tools.** ***Citool*** **is a GUI for committing changes.**

» **Last month** We used webcams to build a home security system.

for the masses

Now create a simple directory structure and some files:

```
mkdir public && mkdir doc
touch public/index.php && touch doc/README
```

What we have now could be any project, written in any language. The next step is **git**:

```
git init
git add .
git commit -m "Setup repository"
```

You've created a new repository in your working directory – note that the repository is here in your current directory, rather than on a central repository somewhere. You can now keep track of your code from within this directory, so start by editing one of the files. Edit **public/index.php** to add:

```
<?php
echo "This is our very empty project... but at least we've started.";
?>
```

... and then **git status** to see a message telling you that the file has been modified but will not be committed. Unlike with *SVN* you actively have to ask *Git* to include the changes you made in the commit, although the **-a** option overrides this. Next, we add the file to the commit:

```
git add public/index.php
git commit -m "Added simple message to index file"
```

To see the results, we investigate the logs using **git log**:

```
$ git log
commit f5737c51c4b645617fa3017389e873628eec0edc
Author: Dan Frost <user@example.com>
Date:   Thu Nov 20 19:51:00 2008 +0000
Added simple message to index file

commit 03f5c27f3bf3ef72c5c92618a323a814eb76767b
Author: Dan Frost <dan@3ev.com>
Date:   Thu Nov 20 19:41:28 2008 +0000
Started repo
```

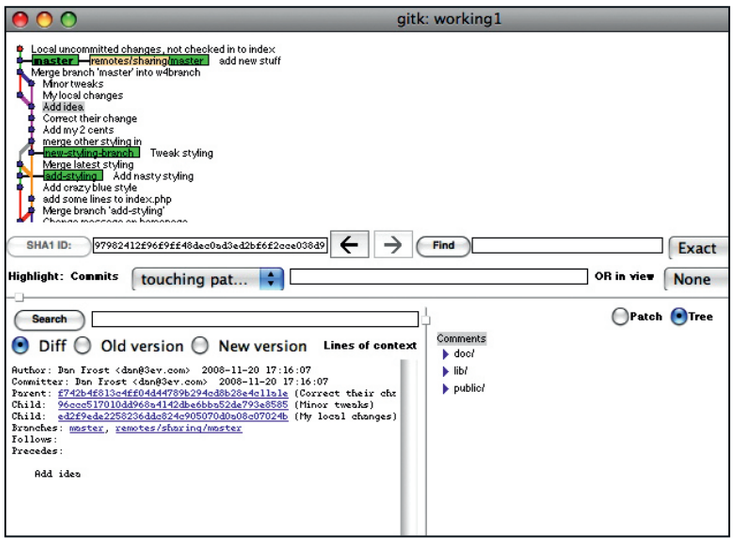
Show me by SHA1

So, what on Earth are those stupidly long strings? They're version numbers. Wouldn't version *numbers* be easier – 1, 2, 3 etc? Actually, this is only easier if you work alone: numbering version 1, 2, 3 and so on simply doesn't work if you have more than one repository. My version 2535 would be your version 102, and someone else's version 8745.

Git dispenses with the niceties of simple, integer version numbers in place of the mighty SHA1. This cryptographic hash enables *Git* to identify every part of every repository (not just yours) uniquely – the files, commits, trees and so on – and store them in its object database. The **git show** command enables you to inspect the object database by doing:

```
git show 03f5c27f3bf3ef72c5c92618a323a814eb76767b
```

This will show you exactly what corresponds to the SHA1. Almost every report makes use of the SHA1 – for example, **git log** as we saw above:



› The *Gitk* GUI gives a graphical representation of branches and merges, and the commits that happened between them.

Pick any of the SHA1s from this message and use them with the **git show** command to see what they correspond to. A full SHA1 is quite often an effort to type, so *Git* enables you to use enough of it that it isn't ambiguous:

```
$ git show c3e59317
```

Now you should be able to edit the files some more, create new files and add them all to *Git*. The basic commands of **git add**, **git commit** and **git log** are enough for even the most versioning-averse coder to get some use out of *Git*.

But what changed?

Git's logs can be viewed using the **git log** command, producing a simple log of all changes that have been made. But there's more to be seen, such as producing patches:

```
git log –p
git log --pretty=raw
```

Once you've found the version that used to work before someone else got their hands on your code, you can use **git diff** to inspect the changes:

```
$ git diff f5737c51c4b645617fa3017389e873628eec0edc
diff --git a/public/index.php b/public/index.php
index da4d4c6..95fdaa6 100644
--- a/public/index.php
+++ b/public/index.php
@@ -1,3 +1,4 @@
<?php
echo "This is our very empty project... but at least we've started.";
?>
+Added this
```

There's also a shorthand for diffing to the previous version, the one before that and so on: »

» `git diff HEAD`

Shows differences between the current working directory and the HEAD, while **git diff HEAD^** Shows them between the current directory and the version prior to the last commit.

You can compare a specific revision with the version before last by doing:

```
git diff 06c831d0a30833e0a037f0f1a4b11fd7e1ef226f HEAD^
```

Often, you have a pretty good idea of when the offending revisions occurred, so *Git* offers two ways of referring to time, both of which are nicely human-readable:

```
git diff "@{90 minutes ago}"
git diff "@{2008-10-01 17:30:00}"
```

Finally, a really useful command is **whatchanged**, which unsurprisingly shows what changed in a particular file:

```
git whatchanged public/index.php
```

Sharing your work

The ego of the average programmer means that their code won't stay private for very long. What happens when you want to share your beautifully crafted and versioned code?

There are several methods of sharing *Git* repositories. *Git* has its own protocol, which is highly efficient, and it can also run over HTTP or SSH. We'll share over SSH, as it's probably easier to set up if you have a couple of Linux machines.

On a machine that you're happy for others to access, create a directory to contain the share repository:

```
mkdir /var/git/project.git
cd /var/git/project.git
git init --bare
```

You'll notice that we add the **--bare** option here. If you return to the repository you created earlier and do **ls -a**, you'll notice a **.git** directory. This directory is the *Git* repository – similar to the one that was created in our working directory earlier. All of the commits and changes are stored in this.

When we make a bare repository we're actually making a *Git* repository without the working directory – ie, we're making the **.git** directory. If you have a look at the contents of our newly created bare repo and the **.git** directory created earlier you'll see very few differences.

Once you've created the repository for sharing, go back to your local machine and into your working directory. Start by adding the newly created bare repository as a **remote**:

```
git remote add sharing-server ssh://www.example.com/var/
git/project.git
git push sharing-server master
```

The first command adds the server's URL under the name **sharing-server**, while the second pushes your master branch to that server. If at any time you want to see what remote servers you've configured, do:

```
git remote
```

Once you've pushed to the remote repo, you'll want people to grab your brilliant new app. The simplest way of doing this is for users to clone:

```
mkdir ~/my-sandbox/ && cd ~/my-sandbox/
git clone ssh://myserver.com/var/git/myapp.git
```

This gives you a complete copy of the app to edit or deploy. Like **svn checkout**, anyone working with you or just using your software can grab the code with this method.

Unlike *SVN*, if you commit changes in your working version they won't show up in the 'remote' version until you push them up again using **git push sharing-server master**. Even though you and the collaborator can keep track of detailed changes to the code, your individual changes aren't automatically shared – this is distributed versioning.

Your collaborator can push changes to this repository with **git push**, after which you can pull them back down using:

```
git pull sharing-server master
```

This is similar to *SVN* commit and update, except that you can track changes locally before sharing them. And you can have your own branches...

Branches

Just before you start creating branches, you should add this useful little tweak (thanks to **simplisticcomplexity.com**) to your **.bash_profile** so that you can always see which branch you're working on:

```
parse_git_branch() {
  git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/(\1)/'
}
PS1="\w\$(parse_git_branch) $ "
```

This will show which branch you're on at any given time. Creating a branch in *Git* is very easy – the following creates the new branch based on the last commit you made. *Git* moves you into the branch to begin work:

```
git checkout -b my-new-branch
```

Once in the branch you can make any changes without worrying about breaking the 'master' branch. To switch between branches, use the **checkout** command again:

```
git checkout master
git checkout my-new-branch
git checkout -b my-new-idea
```

Your own private stash

Got an idea again? Just tried it out, but now need to get on with some real work? Stash it.

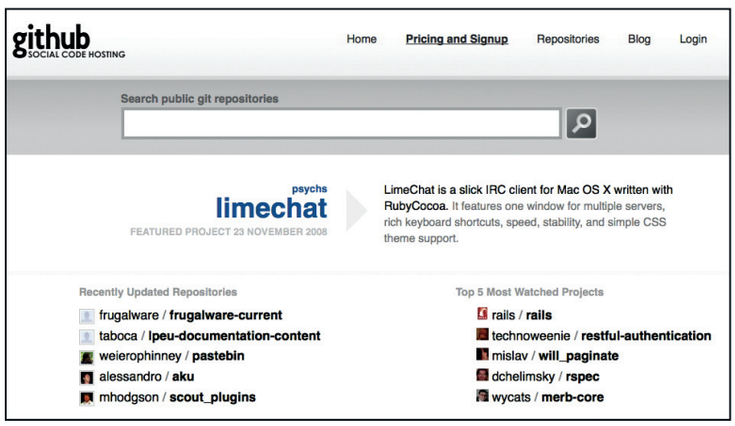
```
/www/gitarticle/working1(master) $ git stash
Saved "WIP on master: a6d6f31... merge other styling in"
HEAD is now at a6d6f31... merge other styling in
```

Now you're back to your previous commit but your recent changes have been 'stashed'. To get them back, just:

```
git stash apply stash@{1}
```

Or create a branch from the stash with:

```
git stash branch stash@{1}
```



› ***Git*** **hosting is available for your open source project from a few sources. GitHub provides *Git* hosting plus some cool features.**

Try this in the *Git* clone repository we created earlier – you'll find you can edit, move, create and remove files in the branch independently of the master branch that you cloned. And if you're not ready to commit your changes back to the remote repository, then switch back to the master to continue working on the main code with **git checkout master** and using **git pull** to update your source.

Of course, at some point you'll want to merge the changes you made in your branch back into the master and push them to the remote repository. You first switch back to the master, then merge the branch into the master and push your changes to the remote repository:

```
git checkout master
git merge my-branch
```

After merging, you might have conflicts. As with *CVS* and *SVN*, these are indicated in the source files:

```
<<<<<<< HEAD:public/index.php
    Hello!
    =======
    Hello, World!
>>>>>>> update-message:public/index.php
```

The first section shows the current branch's content, while the second shows the merged-in branch. When you've fixed the conflict use **git add** to add the file into the next commit. This indicates to *Git* that the conflict has been resolved.

When you've merged and mended everything, you're ready to push it all to the remote server:

```
git push
```

Then the original author can pull your changes and see them in his log. This way of working – by committing locally and then pushing to a remote server – shows how the distributed model differs from the central server model, and how *Git* is geared towards collaboration.

Multiple remotes

The example above creates a clone of the master repository, but you can also create your own working repository into which you pull from multiple remote repositories. This is where the magic of *Git* kicks in – you can work with 20 collaborators, each with their own repositories, and selectively pull in changes from each of them. To do this, create a repository and attach new remote repository to it:

```
git init
git remote add remote-server ssh://www.example.com/var/
git/project.git
git pull remote-server master
```

You can use any combination of remote servers and branches. Each time you add one, you're getting a copy of an author's changes copied into your local *Git* repository. Using this, you can easily perform diffs, merges and more on multiple authors' content. You can also take any branch of yours and push it to your own server:

```
git remote add my-server ssh://www.my-example.com/var/
git/my-fork.git
git push my-server my-branch
```

Your collaborators then add this server as a remote to their working repositories in order to get a copy of your changes. The idea of pushing and pulling from multiple repositories can be confusing, but makes a lot of sense as you start working with large groups. *Git* is designed to make it easier to work like this, so that you can simply grab someone else's source, make a change and push it to another person.

Watch the Gitters

Watch these videos to see why *Git* was created and what it can do:

» **Tech Talk – Linus Torvalds on *Git***
http://uk.youtube.com/watch?v=4XpnKHJAok8

» ***Git* – a Talk by Randal Schwartz**
http://video.google.com/videoplay?docid=-3999952944619245780&

For example, I might grab your recent changes:

```
git pull your-server master
```

... have a play with them, add some features I think you should have built and then:

```
git push my-server master
```

Too many branches

Branches are brilliant for working without people looking over your shoulder, but they're also an ideal way of losing what you were working on. As with any versioning or code management system, *Git* does require some discipline but also comes with tools to help you find out what's going on.

The **show-branch** function gives a somewhat graphical outline of the state of your branches.

```
$ git show-branch --all
! [mybranch] Add nasty styling
* [master] add some lines to index.php
--
+  [mybranch] Add nasty styling
+  [mybranch^] Add crazy blue style
* [master] add some lines to index.php
+* [mybranch~2] Change message on homepage
```

The first section is a list of branches where ***** denotes the current branch and **!** denotes the HEAD of all other branches. In this example, everything under the **--** is prefixed by two columns, one for each of the two branches (master and mybranch). A **+** means that the commit exists in the corresponding branch, while a blank space means that it does not. The names in brackets can be used in place of an SHA1 – for example, **git show mybranch~2**.

Going back in time

If you or someone else has broken something or you want to see how the app used to work, you can check out a specific revision. The revision numbers are pretty ugly – punching in a SHA1 every time you want to see what happened is cumbersome, so there are a few shortcuts. You can revert to a previous time using a human-readable statement:

```
git checkout "@{10 minutes ago}"
```

Now you've gone back in time, so if you find something good back here you might want to start a branch:

```
git checkout -b what-might-have-been
```

The new branch can be worked on just like any other, with commits, merges from the master branch of other branches and sharing with collaborators. When you want to merge it into some production code, merge just as you did previously.

```
git commit –m "Found some good changes here"
git checkout master
git merge my-new-branch
```

The world is flocking to *Git* – check out **github.com** to see the likes of *Rails*, *Scriptaculous* and *MooTools* moving there. It's easy to see why. The flexibility of the decentralised model and really cheap branching is too tempting to ignore. LXF

» **Next month** Get building with Google App Engine's big box o' code.