Working with Git

Tim Sutton

2011

Contents

1 Working with Git

This document provides a simple introduction to working with git. Git is a distributed source code management (SCM) system. Although this is a source code management system, you can use it for **any** type of documents that you want to version control and / or work on collaboratively with others.

If you are familiar with SVN, Git is quite similar but adds some new concepts. In particular, Git is distributed, which means there doesnt have to be one single repository that you work against. Git also is ideal for offline work where you still want to do version control. We will explore this more as we go on.

1.1 Installation

Installing Git is easy. Under linux, install like this:

Listing

```
sudo apt-get install git meld gitg
```

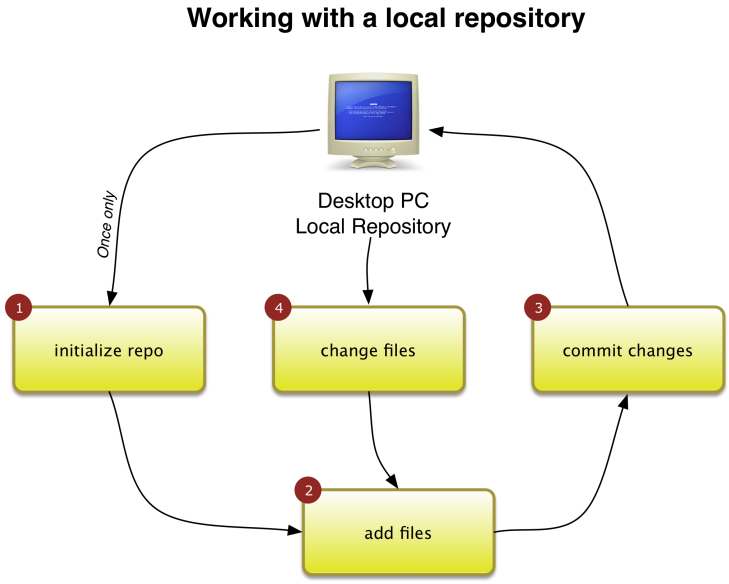
The latter two are not actually needed but will prove useful later.

Under windows you can use msysgit - notes on using msys git are provided further down in this document.

1.2 Quick Start

I would like to plunge in with a quick start to using git and then come back to a more systematic coverage of the tool. The idea is to get you familiar with the concepts in the following diagram:

linfiniti consulting * cc no: 2007/225838/23 * VAT Number: 4480250622 members Tim Sutton MA(GIS), Marcelle Sutton BA(HDE)
Buirski Plein,3, Jansen Straat, Swellendam, 6740, South Africa - 20.44409,-34.01962
tel : +27 (0)71 407 6101 fax: +27 (0)86 564 4056 web : www.linfiniti.com email : tim@linfiniti.com
open source consulting * web hosting * open source geospatial software development * linux and open source geospatial software support



1.2.1 Create a local repository

We start off by creating a local repository. A local repository resides on your own hard dist (its just a directory with some special hidden files in it), and allows you to do all normal SCM activities - check out code, make changes, commit, review your history and so on. We will just make a new directory and initialise it as a git repository:

Listing

```
cd dev
mkdir git-sandbox
cd git-sandbox
git init
```

Ok now we have a repository, lets do some work in it.

1.2.2 Adding a file to the repository

To version any files in the repository, you just need to copy them into your directory (in our example it is called git-sandbox) or create them. Let's create a file called README:

Listing

```
gedit README
```

or

Listing

```
vim README
```

Now we will write a little text to the file:

Listing

```
Hello world from my sandbox
```

Then save and close the file. We can use the git *status* command to see the status of our repository:

Listing

```
git status
```

Which will return something like this:

Listing

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

It tells us that we have one untracked file - the README file we just created. So how do we start tracking the file?

Listing

```
git add README
```

Now *git status* will show the file as tracked:

Listing

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   README
#
```

Ok, now you can commit your file:

Listing

```
git commit -m "First version of README"
```

```
[master (root-commit) 7ea07e1] First version of README
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 README
```

The *-m* option is used to specify the commit message. If you don't provide a *-m* option, git will prompt you for a message using a simple text editor. Once the file is committed, your repository will have a nice clean status:

Listing

```
git status
# On branch master
nothing to commit (working directory clean)
```

Let's make a small change to the README file:

Listing

```
Hello world from my sandbox - I'm using Git!
```

Git status will show that the file is now modified:

Listing

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

This change is **unstaged** - it wont be included in your next commit unless you add it.

Listing

```
git add README
```

Now its status is set to **staged** - it will be included in the next commit you make.

Listing

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README
#
```

Finally you can commit your change using the commit command again:

Listing

```
git commit -m "Improved the README"
[master 983f6fd] Improved the README
```

```
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Did you notice that odd looking number in the output? 983f6fd is shortened version of the unique SHA-1 hash assigned to that commit. Each commit you make will be assigned such an identifier. The commit numbers are not sequential numbers like SVN has. The reason for this is that the commits need to be globally unique in a distributed repository environment (which we will explore later).

We will finish off our quick start tour by running a few interesting commands on our repository:

Listing

```
git log README
commit 983f6fda163c09094ef6939b7d4db4af1bfa8c3c
Author: Tim Sutton <tim@linfiniti.com>
Date:   Mon May 9 23:04:08 2011 +0200

    Improved the README

commit 7ea07e1d1e029510a258efebc8cc170cb685803c
Author: Tim Sutton <tim@linfiniti.com>
Date:   Mon May 9 16:35:37 2011 +0200

    First version of README
```

The *git log* command shows you the history of commits made for a file (most recent changes are shown above older changes). You can see we have made two commits to this file, and the message associated with each commit.

Finally, lets look at the difference between these two commits:

Listing

```
git diff 7ea07e1 983f6 README
diff --git a/README b/README
index 2d564aa..d85ea56 100644
--- a/README
+++ b/README
@@ -1 +1 @@
-Hello from my sandbox
+Hello from my sandbox - I'm using Git!
```

You can see the two options I passed to the diff command are shortened versions of the SHA-1 hashes assigned to each commit. You can also see that the text in my README file was augmented with the phrase "I'm using Git!".

Hopefully this quick look at git has given you the basic concept. In the sections that follow we will explore with more detail some of the other things you can do with Git.

1.3 Git Workflows

In this section we are going to walk you through various scenarios to show you how git can be used effectively.

1.3.1 Repository Clones

You can create your repository in one of two ways:

1. Initialise a new one
2. Clone an existing one

We already created our own repository using the 'quick start' section above. Let us see how you can clone the repository we made earlier (we will assume that you are still in the **git-sandbox** directory at this time):

Listing

```
cd ..
git clone git-sandbox git-sandbox-clone
```

You should see a message like this:

Listing

```
Initialized empty Git repository in /tmp/git-sandbox-clone/.git/
```

Now if we enter the cloned repository, we can work in it just like we worked in the original directory. Lets run that *git log* command again:

Listing

```
[git-sandbox-clone] git log
commit 983f6fda163c09094ef6939b7d4db4af1bfa8c3c
Author: Tim Sutton <tim@linfiniti.com>
Date:   Mon May 9 23:04:08 2011 +0200

    Improved the README

commit 7ea07e1d1e029510a258efebc8cc170cb685803c
Author: Tim Sutton <tim@linfiniti.com>
Date:   Mon May 9 16:35:37 2011 +0200

    First version of README
```

You can see our clone has exactly the same commit history as the original repository has - it is an exact clone of the original. Let's make another change to our README file and commit it. I'm going to add this line:

Listing

```
This line was added in the cloned repository.
```

And then *git add* and *git commit* my changes:

Listing

```
git add README
git commit -m "Added a line while in my cloned repo"
```

Which produces output like this:

Listing

```
[master 218dfa8] Added a line while in my cloned repo
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Now run the *git log* command again and look at the output:

Listing

```
[git-sandbox-clone] git log README
commit 218dfa8474c3213b1a77973df8739ed75120bde5
Author: Tim Sutton <tim@linfiniti.com>
Date:   Mon May 9 23:31:21 2011 +0200

 Added a line while in my cloned repo

commit 983f6fda163c09094ef6939b7d4db4af1bfa8c3c
Author: Tim Sutton <tim@linfiniti.com>
Date:   Mon May 9 23:04:08 2011 +0200

 Improved the README

commit 7ea07e1d1e029510a258efebc8cc170cb685803c
Author: Tim Sutton <tim@linfiniti.com>
Date:   Mon May 9 16:35:37 2011 +0200

 First version of README
```

Well done - you cloned the repository and make a change in your local copy.

== Git conflicts =

Lets go back to our original repository and update the readme file with a new change:

Listing

```
cd ../git-sandbox
gedit README
```

And add this line:

Listing

```
This is a change in the original.
```

Now commit your changes and return to the clone directory:

Listing

```
[git-sandbox] git add README
git commit -m "A new update from the original repo"
cd ../git-sandbox-clone
```

Now we will merge any changes made in the original repo into our clone:

Listing

```
[git-sandbox-clone] git pull
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
```

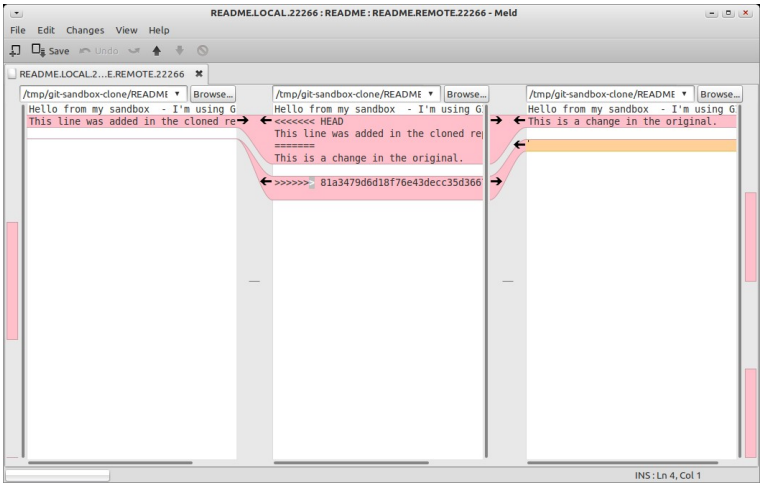
What happened? Git pulled in the changes made in the original repo, but detected that the same line had been changed in both the original repo and the clone, so it doesn't know how to combine them. We can help git to resolve the issues by running *git mergetool*.

Listing

```
[git-sandbox-clone] git mergetool
merge tool candidates: meld opendiff kdiff3 tkdiff xxdiff tortoisemerge
        gvimdiff diffuse ecmerge p4merge araxis vimdiff emerge
Merging the files: README

Normal merge conflict for 'README':
  {local}: modified
    {remote}: modified
    Hit return to start merge resolution tool (meld):
```

When you press enter a graphical user interface will appear (if you are on linux) which looks something like this.:



The merge tool shows a side by side view of the version or README in your local repository (clone) and the version of README. Alternatively you can simply open the README in a text editor and resolve the differences. Here is another example of where a merge conflict occurs (before resolving):

Listing

```
Hello from my sandbox  - I'm using Git!
This is a change in the original.
<<<<<<< HEAD
Local Change
=======
Yet another change
>>>>>>> 1386adb6b1b6131d64d18f2bffa877a6687b0073
```

And then after editing the conflicted file it might look like this:

Listing

```
Hello from my sandbox  - I'm using Git!
This is a change in the original.
Local Change
Yet another change
```

Once you have created the final version, you need to commit to indicate to Git that the conflict has been resolved. Doing a *git status* will show you what needs to be committed:

Listing

```
[git-sandbox-clone] git status
# On branch master
# Your branch and 'origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#   both modified:      README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   README.orig
no changes added to commit (use "git add" and/or "git commit -a")
```

So to finalise the conflict resolution, we can commit our change:

Listing

```
[git-sandbox-clone] git add README
[git-sandbox-clone] git commit -m "Resolved merge conflicts on README"
[master ed30a38] Resolved merge conflicts on README
```

Sidebar: We learnt something else new in this section: When we have a clone of repository, you can synchronise it with the changes in the original repository by doing *git pull*. In our simple sandbox example, the two repositories are just two directories in the same file system, though Git supports synchronising repositories over a network connection too, which is commonly how it is used.

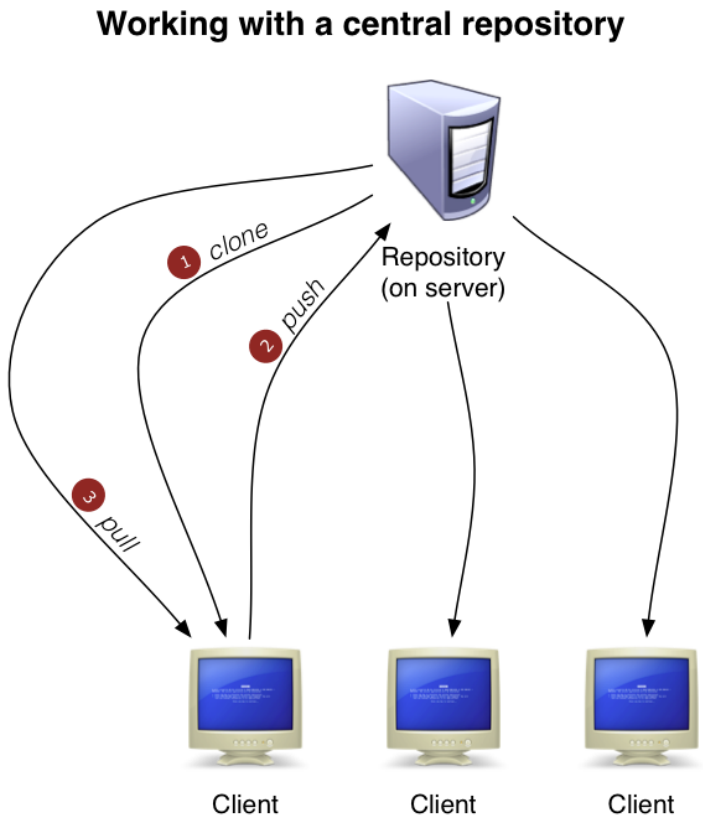
1.4 Git Topologies

Git is a distributed versioning system. That means there doesnt't have to be one single source for the repository. You can define the topology for your project in a way that is most convenient to you.

1.4.1 Central Repository

If you work in a small team, you may adopt a centralised repository model, similar to SVN. Actually it is a bit of a misnomer, since each client who checks out the repo obtains a copy (with all its version history) on their local machine. This is unlike SVN where you check out only a **snapshot** of the current state of the code. In SVN, any time you want to refer back to an older commit, you need to query the central repository over the network. In Git you have a full copy of the repository stored locally so that network connection is not required. Similarly, with SVN, if you want to commit a change, you need to have a network connection to the repository. Under Git however you commit to your local repository first and then push your changes up to the central repository when you have a connection available. This makes git great for working away from the office, on the plane etc. You just work as per normal and when you get connected to the internet again, you simply sync (push) your local changes too the *origin* repository.

Sidebar: One huge advantage of Git is that you inherantly have a full historised backup for each person who has cloned the repo. If the central repo should be lost, you can simply reinstate it from any of the clones on your developer's desktops.

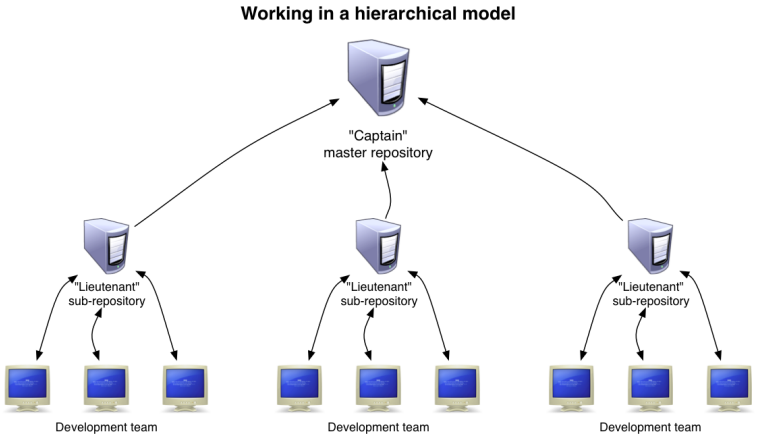


The workflow (as shown in the illustration above) when working with a central repository is simple:

1. Clone the repository initially
2. Make local changes and add them
3. Commit those changes locally
4. Push your changes up to the central repo

1.4.2 Heirarchical Topology

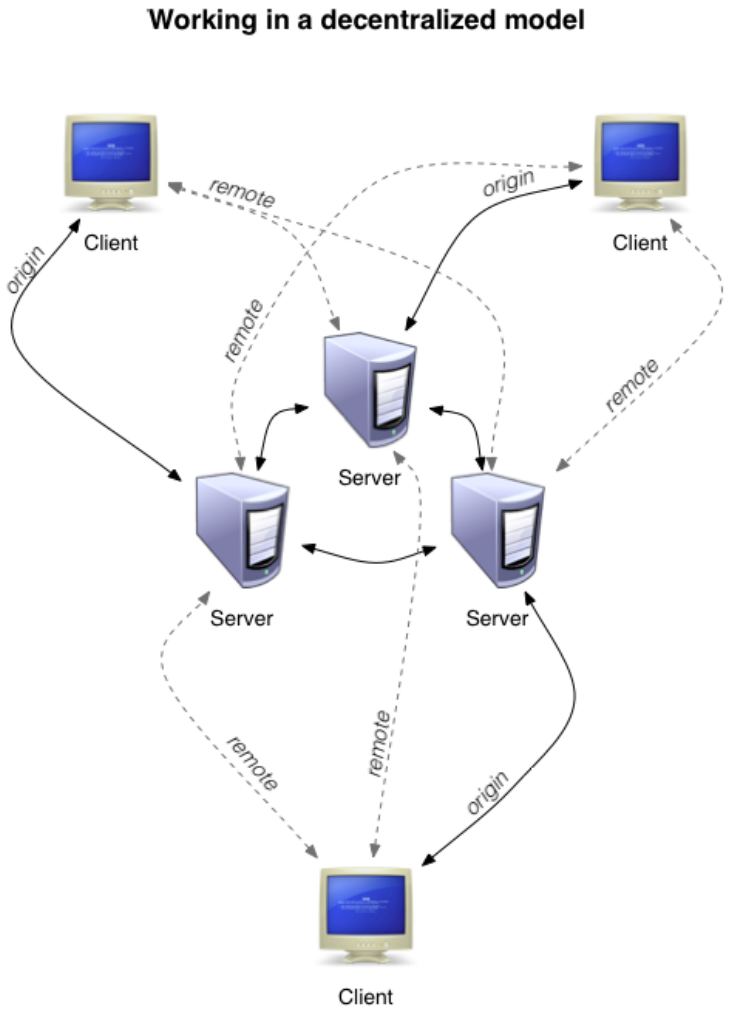
In larger teams, or where you have subteams working on different areas of your code base, you may wish to adopt a heirarchical model - also called the 'captain and luitenants model'.



In this arrangement, there is one person (the 'captain') who performs the final integration into the official repository. Under the 'captain', various 'luitenants' take responsibility for different aspects of the code base. Each 'luitenant' has a team of 1 or more developers who push to his repository (or request him to pull from theirs). The 'luitenant' collates all developer's work and when it is ready asks the captain to integrate the work into the official repository. This topology works well for organisations or projects with a well established heirarchy

1.4.3 Decentralised Model

In a decentralised model there is no real 'single point of truth'. Developer's share their work with each other on an *ad hoc* basis.



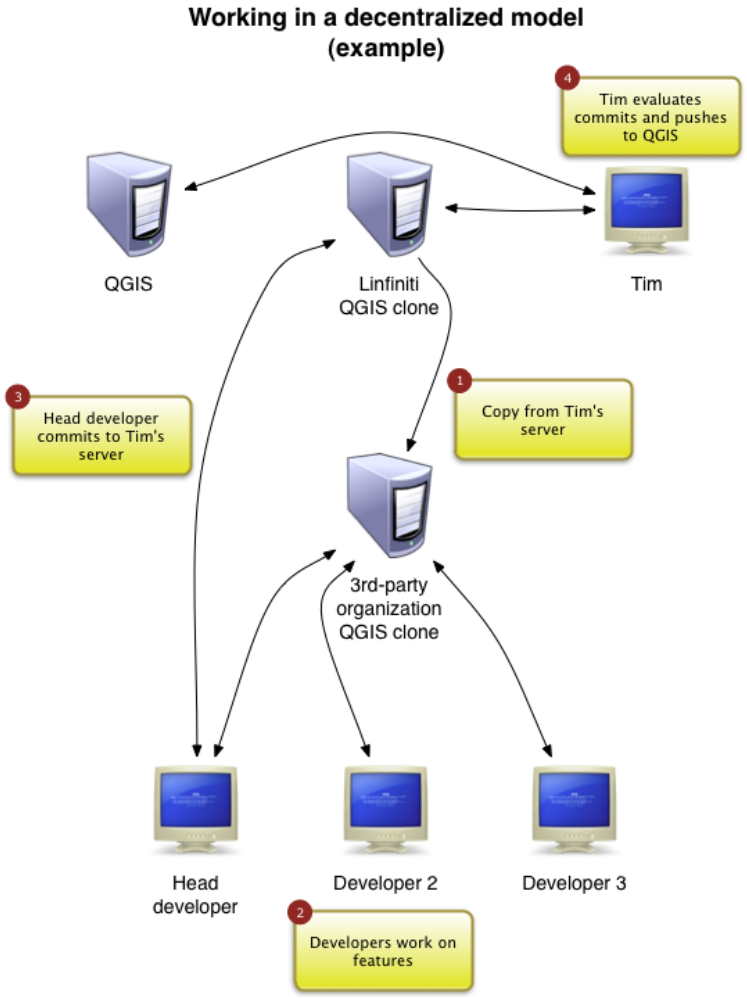
Version history is exchanged between repositories on an *ad hoc* basis. Repository owners add one or more *remotes* to their repository (a *remote* is a reference to another repository). They then *pull* or *push* their changes to these repos as needed. The choice of when to *push* or *pull* and to which *remote* is a social practice defined by the developers themselves and not enforced by any software.

1.4.4 Hybrid topology

The choice of topology is not fixed - you may choose to mix elements together in a way that suites your organisation of project the best. Here it is easiest to illustrate using Quantum GIS, my favourite open source project as an example.

Scenario: The QGIS project has a repository. SANSA is busy building extensions the QGIS project. Linfiniti is creating special software packages for them. Individual devel-

opers need to be able to work and collaborate and don't want their work to be puclically visible until they are ready. We can create an architecture like the one illustrated below:



In this architecture, any developer as SANSA can push their changes to the local SANSA GIS Clone. Periodically (e.g. nightly) SANSA will pull changes from the the QGIS official repo into their QGIS clone, so that they can be working against the most up to date QGIS code base. When they have an interesting new feature, they can request Linfiniti to pull their changes to the Linfiniti QGIS Clone, test and package the SANSA version of QGIS. When SANSA are ready, and the code complies with QGIS coding standards etc. Linfiniti can push any improvements created by SANSA into the official QGIS repository for the rest of the world to enjoy.

There are many variations and permutations that can be employed, but the key concept to realise is that SANSA can now independently work on the code base using all the coding best practices available and can share their work in an easy and flexible manner.

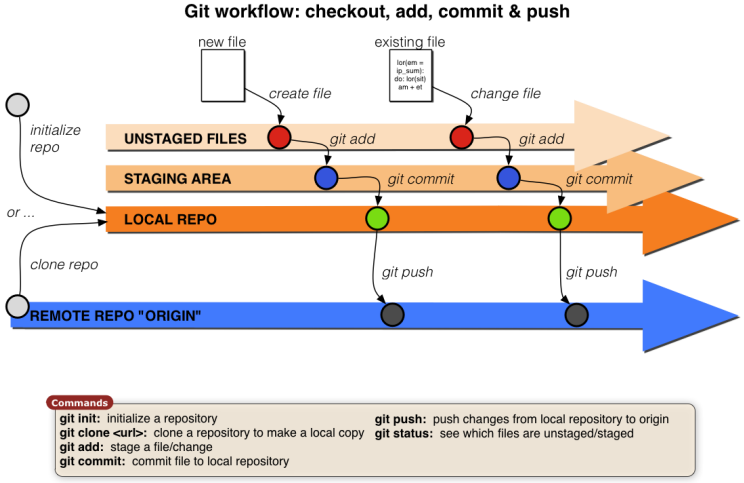
linfiniti consulting * cc no: 2007/225838/23 * VAT Number: 4480250622 members Tim Sutton MA(GIS), Marcelle Sutton BA(HDE)
Buirski Plein,3, Jansen Straat, Swellendam, 6740, South Africa - 20.44409,-34.01962
tel : +27 (0)71 407 6101 fax: +27 (0)86 564 4056 web : www.linfiniti.com email : tim@linfiniti.com
open source consulting * web hosting * open source geospatial software development * linux and open source geospatial software support

1.5 Git Working Practices

In this section we will take a look at Git working practices. There are many git commands and always 10 different ways of doing things, so we will try to cover the basic and commonly used activities.

1.6 Checkout, add, commit, push

We've shown you this in the preceeding sections, but it's worth repeating as it's the essence of working with GIT.



When you work with Git, you changes go through a routine:

1. Initially when you change a file or create a file, it changes are **unstaged**. Unstaged is just another way of saying 'will not form part of the next commit'.

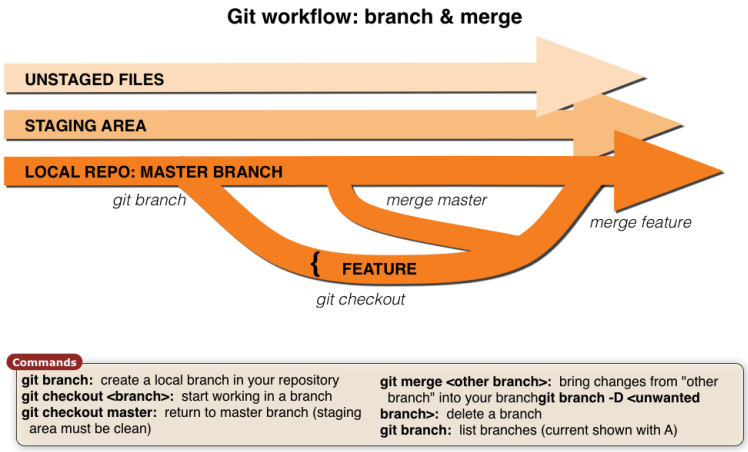
2. To stage a file (or group of files) you *git add* them, moving them from the unstaged area to the **staging area**.

3. One files are staged, you can *git commit* them to your local repository. A **commit** will be accompanied by a short log message and it also stores the name and email address of the committer.

4. You can pass your commits from your local repository over to a remote repository by doing a *git push*. All the commits in your local repository that don't yet exist in the upstream repo are **pushed** to it.

1.6.1 Working in branches

One of the principles of Git is to work in branches. SVN also support branches - the concept is the same in Git. However, Git's design (in particular the fact that it assigns SHA-1 globally unique id's to each commit) makes it very easy to create branches and merge their changes back and forth between other branches or the master branch.



If you create a branch your work in that branch is isolated from the main code store (the *master* branch) - allowing you to safely experiment without impacting anything else. In Git branches and features are often used synonymously - the idea being that a branch is a place where you develop a new feature. While you are working on the feature, changes are probably taking place in the *master* branch. To prevent your branch becoming too different to the *master branch* you will regularly *git merge* the changes from master in to your branch. Once your work on your feature is complete, you then merge your feature branch into the master branch.

You can obtain a list of branches in your local branch (and see which branch you are currently in) by using the *git branch* command. If you want to also see which branches are in a remote repository, you can use the //git branch -r// command.

After you have merged a branch into the *master* branch, usually you will throw away the branch - you can use //git branch -D <branch name> for this. Just make sure you are not currently in the branch that you are deleting or you will evaporate :-).

1.7 Git under windows

Here are some generic notes on using git under windows you should install [msys git app http://code.google.com/p/msysgit/]. In windows explorer go to c:\Documents and Settings\<your user>\

Make a directory called .ssh

In that directory create a text file called 'config' (note it has no extension) and put the following content into it:

Listing

```
Host <host name>
  User <user name>
  HostName <host name>
  Port <port>
```

Replace items in angle brackets above as appropriate.

Now copy your id_dsa into this directory (it should be a unix style one so you may need to convert from putty style private key, though just try with your existing one first).

Open the msys git shell then go to the directory where you want to check out your project to. For example to check it out to c:\dev\foo do

Listing

```
cd /c/
mkdir dev
cd dev
```

Now clone the directory:

Listing

```
git clone git@foo:bar.git bar
```

Make sure to type 'yes' in full when it asks you if you are sure you want to continue connecting.

Then enter your passphrase when prompted.

Wait a few minutes while it checks out.

Thereafter you use the git commands from the msys shell as normal. There is also a tortoisegit explorer integration for windows you can try but I havent used it and don't know how well it works.

1.8 Further Reading

http://gitref.org http://progit.org http://gitready.com