Git/Print version

Git

Obtaining Git

Git is available for *nix operating systems, as well as MacOS and Windows.

Binary

Linux

Debian-based distributions (`.deb`)

Git is available on Debian, and derivatives like Ubuntu. It is currently packaged as `git`. More recent packages of git are available from the Ubuntu Git Maintainers' PPA (https://launchpad.net/~git-core/+archive/ppa). You may also want to install some extended git functionality, for example `git-svn`, which allows two-way interoperability with Subversion, or `git-email` which provides utility functions for sending and receiving git data (primarily patches) via email.

```
$ sudo apt-get install git git-svn git-email
```

RPM-based distributions (`.rpm`)

Linux distributions using the RPM package format can use yum to get git:

```
$ sudo yum install git-core
```

macOS

A graphical installer can be found at Google code (http://code.google.com/p/git-osx-installer). Alternative, if you have MacPorts (http://macports.org) installed, you can do

```
$ sudo port install git-core
```

git is also included in the *com.apple.pkg.Core* package:

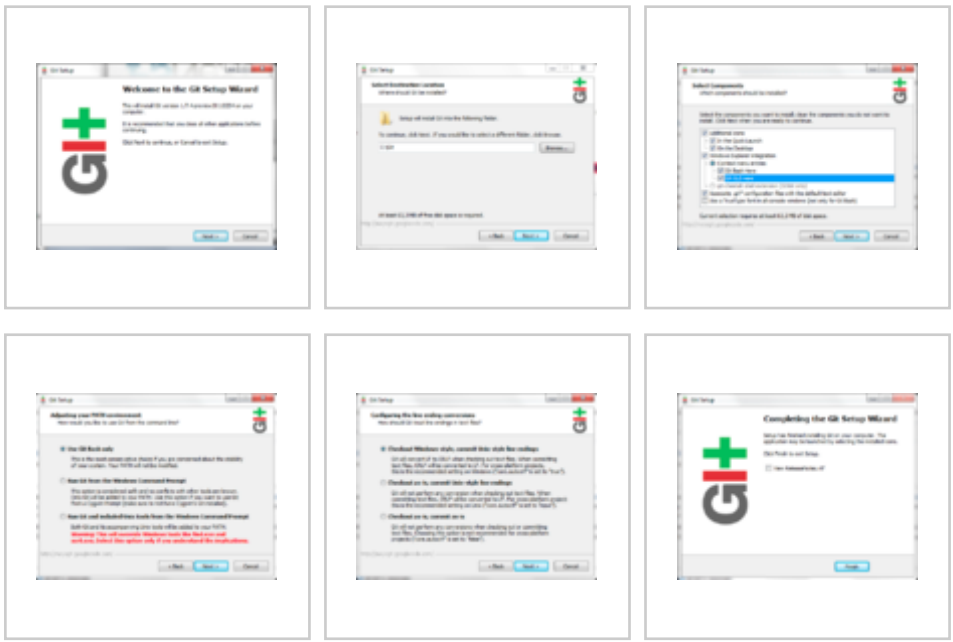
```
pkgutil --file-info `which git`
volume: /
path: /usr/bin/git

pkgid: com.apple.pkg.Core
pkg-version: 10.13.4.1.1.1522206870
install-time: 1526062261
uid: 0
gid: 0
mode: 755
```

Windows

Git for Windows is available as a precompiled binary msysGit (http://code.google.com/p/msysgit). This includes the command line utilities, a GUI, and an SSH client.

Additionally, those with Cygwin can use its setup to obtain Git.



Source

Tarball

You can obtain a copy of the newest stable git from the git homepage at: git.or.cz (http://git.or.cz/). In addition, daily snapshots (http://www.codemonkey.org.uk/projects/git-snapshots/git/) of git are provided by Dave Jones.

Below is an example of how to compile git from source, change "git-1.5.3.4.tar.gz" to the version you downloaded:

```
mkdir ~/src
cd ~/src
wget http://kernel.org/pub/software/scm/git/git-1.5.3.4.tar.gz
tar xzvf git-1.5.3.4.tar.gz
cd git-1.5.3.4
make configure
./configure --prefix=/usr/local
make
sudo make install
```

Without the added `--prefix` argument git will currently install to `~/bin`. This may or may not be what you want, in most distributions `~/bin` is not in the $PATH.[1] Without the --prefix, you might have to explicitly state the path to the component programs on invocation, i.e.: `~/bin/git-add foobar`. You can set `--prefix` to whatever better suits your particular setup.

Git

The source can additionally be acquired via git using:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Or in the event you have problems with the default Git port of 9418:

```
$ git clone http://www.kernel.org/pub/scm/git/git.git
```

Footnotes

1. ^ One effort to amend the lack of consistency through modern distributions and ~/bin, has been addressed by the Ubuntu developers which seeks to patch PAM, the authentication mechanism, to set up the environmental variable $PATH. You can find more information out about this at https://bugs.launchpad.net/ubuntu/+source/pam/+bug/64064 (https://bugs.launchpad.net/ubuntu/+source/pam/+bug/64064).

First configuration

To avoid to reenter one's credential during each sync, register the account:

```
git config --global user.email "michael.boudran@en.wikibooks.org"
git config --global user.name "Michael Boudran"
```

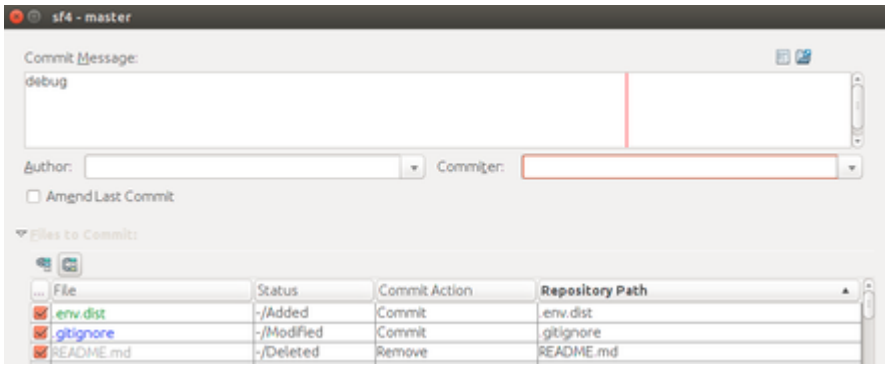
To avoid the password in Linux, it's necessary to store it in plain text into:

```
vim ~/.netrc
```

With (ex: for *github.com*):

```
machine github.com
      login <user>
      password <password>
```

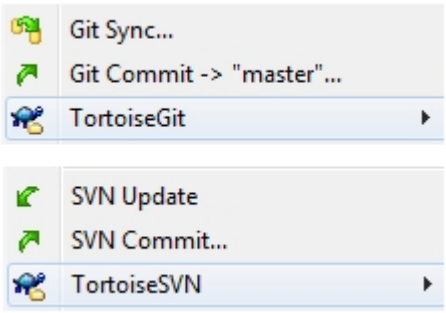
Other Git clients



NetBeans commit.

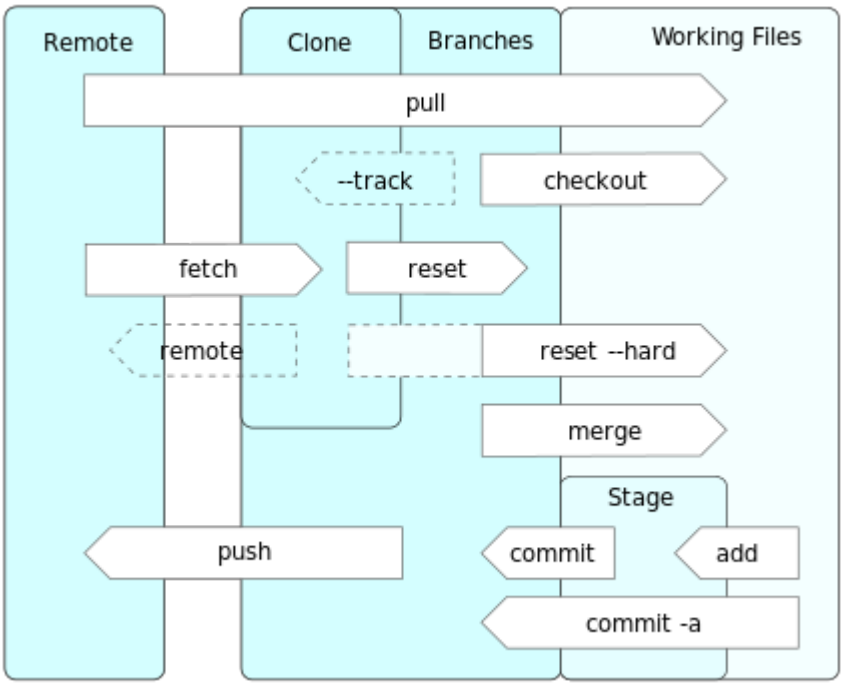
Some integrated development environments like NetBeans also provide or complete Git client.

TortoiseGit allows to access to its Git commands by right-clicking on the concerned files and folders.



TortoiseGit quick options.

Introduction



Git operations representation.

Here, we will introduce the simplest git commands: creating a new repository, adding and committing files, removing files, reverting bad commits, throwing away uncommitted changes, and viewing a file as of a certain commit.

Creating a git repository

Creating a new git repository is simple. There are two commands that cover this functionality: `git-init(1)` (http://git-scm.com/docs/git-init), and `git-clone(1)` (http://git-scm.com

/docs/git-clone). Cloning a pre-existing repository will be covered later. For now, let's create a new repository in a new directory:

```
$ git init myrepo
```

Initialized empty Git repository in:

1. /home/username/myrepo/.git/ on Linux.
2. C:/Users/username/myrepo/.git/ on Windows.

If you already have a directory you want to turn into a git repository:

```
$ cd $my_preexisting_repo
$ git init
```

Taking the first example, let's look what happened:

```
$ cd myrepo
$ ls -A
.git
```

The totality of your repository will be contained within the `.git` directory. Conversely, some SCMs leave droppings all over your working directory (ex, `.svn`, `.cvs`, `.acodeic`, etc.). Git refrains, and puts all things in a subdirectory of the repository root aptly named `.git`.

Remark: to set the default directory where Git will point at each opening, under Windows right click on the shortcut, and change the path of the field called "start in".

Checking Your Status

To check the status of your repo, use the `git-status(1) (http://git-scm.com/docs/git-status)` command. For example, a newly-created repo with no commits in it as yet should show this:

```
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

Get into the habit of frequent use of `git-status`, to be sure that you're doing what you think you're doing. :)

Adding and committing files

Unlike most other VCSs, git doesn't assume you want to commit every modified file. Instead, the user adds the files they wish to commit to the *staging area* (also known as the *index* or *cache*, depending on which part of the documentation you read). Whatever is in the staging area is what gets committed. You can check what will be committed with `git-status(1) (http://git-scm.com/docs/git-status)` or `git diff --staged`.

To stage files for the next commit, use the command `git-add(1) (http://git-scm.com/docs/git-add)`.

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

This shows us that we're using the branch called "master" and that there is a file which git is not *tracking* (does not already have a commit history). Git helpfully notes that the file can be included in our next commit by doing `git add file.txt`:

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

After adding the file, it is shown as ready to be committed. Let's do that now:

```
$ git commit -m 'My first commit'
[master (root-commit) be8bf6d] My first commit
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 file.txt
```

In most cases, you will not want to use the `-m 'Commit message'` form - instead, leave it off to have `$EDITOR` opened so you can write a proper commit message. We will describe that next, but in examples, the `-m 'Commit message'` form will be used so the reader can easily see what is going on.

You can use `git add -A` to automatically stage all changed and untracked files for inclusion in the next commit. Once a file is being tracked, `git add -u` will stage it if it has changed.

If you change your mind about staging a file, and you haven't committed yet, you can *un*stage it with the simplest form of the `git-reset(1) (http://git-scm.com/docs/git-reset)` command:

```
$ git reset file.txt
```

to unstage just the one file, or

```
$ git reset
```

to remove everything in the staging area.

`git-reset` has many more functions than this, for example:

- To cancel the two latest commits without touching the files: `git reset 'HEAD~2'`.
- To cancel the two latest commits and their modifications into the files: `git reset HEAD~2 --hard`.
- To cancel the two last operations on the branch: `git reset 'HEAD@{2}'` (which uses `git reflog`). This can be used to cancel an undesired reset.

To exclude certain untracked files from being seen by `git add -A`, read on...

Excluding files from Git

Often there are files in your workspace that you don't want to add to the repository. For example, `emacs` will write a backup copy of any file you edit with a tilde suffix, like `filename~`. Even though you can manually avoid adding them to the commit (which means never using `git add -A`), they clutter up the status list.

In order to tell Git to ignore certain files, you can create an ignore file, each line of which represents a specification (with wildcards) of the files to be ignored. Comments can be added to the file by starting the line with a blank or a `#` character.

For example:

```
# Ignore emacs backup files:
*~

# Ignore everything in the cache directory:
app/cache
```

Git looks for an ignore file under two names:

- `.git/info/exclude` — this is specific to your own personal copy of the repository, not a public part of the repository.
- `.gitignore` — since this is outside the `.git` directory, it will normally be tracked by Git just like any other file in the repository.

What you put in either (or both) of these files depends on your needs. `.gitignore` is a good place to mention things that everybody working on copies of this repository is likely to want to be ignored, like build products. If you are doing your own personal experiments that are not likely to concern other code contributors, then you can put the relevant ignore lines into `.git/info/exclude`.

Note that ignore file entries are only relevant to the `git status` and `git add -A` (add all new and changed files) commands. Any files you explicitly add with `git add` *`filename`* will always be added to the repository, regardless of whether their names match ignore entries or not. And once they are added to the repository, changes to them will henceforth be automatically tracked by `git add -u`.

Good commit messages

Tim Pope (http://tpo.pe/) writes (http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html) about what makes a model Git commit message:

```
Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the present tense: "Fix bug" and not "Fixed
bug."  This convention matches up with commit messages generated by
commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded by a
  single space, with blank lines in between, but conventions vary here

- Use a hanging indent
```

Let's start with a few of the reasons why wrapping your commit messages to 72 columns is a good thing.

- Git log doesn't do any special wrapping of the commit messages. With the default pager of `less -S`, this means your paragraphs flow far off the edge of the screen, making them difficult to read. On an 80 column terminal, if we subtract 4 columns for the indent on the left and 4 more for symmetry on the right, we're left with 72 columns.
- `git format-patch --stdout` converts a series of commits to a series of emails, using the messages for the message body. Good email netiquette dictates we wrap our plain text emails such that there's room for a few levels of nested reply indicators without overflow in an 80 column terminal.

Vim users can meet this requirement by installing my vim-git runtime files, or by simply setting the following option in your git commit message file:

```
:set textwidth=72
```

For Textmate, you can adjust the "Wrap Column" option under the view menu, then use ^Q to rewrap paragraphs (be sure there's a blank line afterwards to avoid mixing in the comments). Here's a shell command to add 72 to the menu so you don't have to drag to select each time:

```
$ defaults write com.macromates.textmate OakWrapColumns '( 40, 72, 78 )'
```

More important than the mechanics of formatting the body is the practice of having a subject line. As the example indicates, you should shoot for about 50 characters (though this isn't a hard maximum) and always, always follow it with a blank line. This first line should be a concise summary of the changes introduced by the commit; if there are any technical details that cannot be expressed in these strict size constraints, put them in the body instead. The subject line is used all over Git, oftentimes in truncated form if too long of a message was used. The following are just a

handful of examples of where it ends up:

- `git log --pretty=oneline` shows a terse history mapping containing the commit id and the summary
- `git rebase --interactive` provides the summary for each commit in the editor it invokes
- If the config option `merge.summary` is set, the summaries from all merged commits will make their way into the merge commit message
- `git shortlog` uses summary lines in the changelog-like output it produces
- `git-format-patch(1) (http://git-scm.com/docs/git-format-patch)`, `git-send-email(1) (http://git-scm.com/docs/git-send-email)`, and related tools use it as the subject for emails
- `git-reflog(1) (http://git-scm.com/docs/git-reflog)`, a local history accessible intended to help you recover from mistakes, get a copy of the summary
- `gitk`, a graphical interface which has a column for the summary
- Gitweb and other web interfaces like GitHub (http://github.com) use the summary in various places in their user interface.

The subject/body distinction may seem unimportant but it's one of many subtle factors that makes Git history so much more pleasant to work with than Subversion.

Removing files

Let's continue with some more commits, to show you how to remove files:

```
$ echo 'more stuff' >> file.txt
$ git status
# 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:   file.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Although git doesn't force the user to commit all modified files, this is a common scenario. As noted in the last line of `git status`, use `git commit -a` to commit all modified files without reading them first:

```
$ git commit -a -m 'My second commit'
[master e633787] My second commit
 1 files changed, 1 insertions(+), 0 deletions(-)
```

See the string of random characters in git's output after committing (bolded in the above example)? This is the abbreviation of the identifier git uses to track objects (in this case, a commit object). Each object is hashed using SHA-1, and is referred to by that string. In this case, the full string is `e6337879cbb42a2ddfc1a1602ee785b4bfbde518`, but you usually only need the first 8 characters or so to uniquely identify the object, so that's all git shows. We'll need to use these identifiers later to refer to specific commits.

To remove files, use the "rm" subcommand of git:

```
$ git rm file.txt
rm 'file.txt'
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    file.txt
#
$ ls -a
.  ..  .git
```

Note that this deletes the file from your disk. If you only want to remove the file from the git repository but want to leave the file in your working directory, use `git rm --cached`.

```
$ git commit -m 'Removed file.txt'
[master b7deafa] Removed file.txt
 1 files changed, 0 insertions(+), 2 deletions(-)
 delete mode 100644 file.txt
```

Reverting a commit

To revert a commit with another, use `git revert`:

```
$ git revert HEAD
Finished one revert.
[master 47e3b6c] Revert "My second commit"
 1 files changed, 0 insertions(+), 1 deletions(-)
$ ls -a
.  ..  file.txt  .git
```

You can specify any commit instead of HEAD. For example:

The commit before HEAD

```
git revert HEAD^
```

The commit five back

```
git revert HEAD~5
```

The commit identified by a given hash

```
git revert e6337879
```

Resetting a commit

`git reset` provides the same options as `git revert`, but instead of creating a revert commit, it just cancels the commit(s) and lets the file(s) uncommitted.

To cancel a reset, use: `git reset 'HEAD@{2}'`.

Throwing away local, uncommitted changes

To throw away your changes and get back to the most recently-committed state:

```
$ git reset --hard HEAD
```

As above, you can specify any other commit:

```
$ git reset --hard e6337879
```

If you only want to reset one file (where you have made some stupid mistake since the last commit), you can use

```
$ git checkout filename
```

This will delete all changes made to that file since the last commit, but leave the other files untouched.

Get a specific version of a file

To get a specific version of a file that was committed, you'll need the hash for that commit. You can find it with `git-log(1) (http://git-scm.com/docs/git-log)`:

```
$ git log
commit 47e3b6cb6427f8ce0818f5d3a4b2e762b72dbd89
Author: Mike.lifeguard <myemail@example.com>
Date:   Sat Mar 6 22:24:00 2010 -0400

    Revert "My second commit"

    This reverts commit e6337879cbb42a2ddfc1a1602ee785b4bfbde518.

commit e6337879cbb42a2ddfc1a1602ee785b4bfbde518
Author: Mike.lifeguard <myemail@example.com>
Date:   Sat Mar 6 22:17:20 2010 -0400

    My second commit

commit be8bf6da4db2ea32c10c74c7d6f366be114d18f0
Author: Mike.lifeguard <myemail@example.com>
Date:   Sat Mar 6 22:11:57 2010 -0400

    My first commit
```

Then, you can use `git show`:

```
$ git show e6337879cbb42a2ddfc1a1602ee785b4bfbde518:file.txt
hack hack hack...
more stuff
```

Git Checkout Is Not Subversion Checkout

If you are coming to Git after having used the Subversion (http://subversion.apache.org/) centralized version-control system, you may assume that the checkout operation in Git is similar to that in Subversion. It is not. While both Git and Subversion let you check out older versions of the source tree from the repository, only Subversion keeps track of which revision you have checked out. **Git does not**. `git-status(1) (http://git-scm.com/docs/git-status)` will only show you that the source tree does not correspond to the current branch HEAD; it will not check whether it corresponds to some prior commit in the history.

`diff` and `patch`: The Currency of Open-Source Collaboration

It is important to understand early on the use of the `diff(1)` and `patch(1)` utilities. `diff` is a tool for showing line-by-line differences between two text files. In particular, a *unified diff* shows added/deleted/changed lines next to each other, surrounded by *context* lines which are the same in both versions. Assume that the contents of `file1.txt` are this:

```
this is the first line.
this is the same line.
this is the last line.
```

while `file2.txt` contains this:

```
this is the first line.
this line has changed.
this is the last line.
```

Then a unified diff looks like this:

```
$ diff -u file1.txt file2.txt
--- file1.txt   2014-04-18 11:56:35.307111991 +1200
+++ file2.txt   2014-04-18 11:56:51.611010079 +1200
@@ -1,3 +1,3 @@
 this is the first line.
-this is the same line.
+this line has changed.
 this is the last line.
$
```

Notice the extra column at the start of each line, containing a "-" for each line that is in the first file but not in the second, a "+" for each line that is in the second file but not the first, or a space for an unchanged line. There are extra lines, in a special format, identifying the files being compared and the numbers of the lines where the differences were found; all this can be understood by the `patch` utility, in order to change a copy of `file1.txt` to become exactly like `file2.txt`:

```
$ diff -u file1.txt file2.txt >patch.txt
$ patch <patch.txt
patching file file1.txt
$ diff -u file1.txt file2.txt
$
```

Notice how the second `diff` command no longer produces any output: the files are now identical!

This is how collaborative software development got started: instead of exchanging entire source files, people would distribute just their changes, as a unified `diff` or in `patch` format (same thing). Others could simply apply the patches to their copies. And provided there were no overlaps in the changes, you could even apply a patch to a file that had already been patched with another `diff` from someone else! And so this way changes from multiple sources could be merged into a common version with all the new features and bug fixes contributed by the community.

Even today, with version-control systems in regular use, such diffs/patches are still the basis for distributing changes.

`git-diff(1)` (http://git-scm.com/docs/git-diff) and `git-format-patch(1)` (http://git-scm.com/docs/git-format-patch) both produce output which is compatible with `diff -u`, and can be correspondingly understood by `patch`. So even if the recipient of your patches isn't using Git, they can still accept your patches. Or you might receive a patch from someone who isn't using Git, and so didn't use `git-format-patch`, so you can't feed it to `git-am(1)` (http://git-scm.com/docs/git-am) to automatically apply it and save the commit; but that's OK, you can use `git-apply(1)` (http://git-scm.com/docs/git-apply), or even `patch` itself on your source tree, and then make a commit on their behalf.

Conclusion

You now know how to create a new repository, or turn your source tree into a git repository. You can add and commit files, and you can revert bad commits. You can remove files, view the state of a file in a certain commit, and you can throw away your uncommitted changes.

Next, we will look at the history of a git project on the command line and with some GUI tools, and learn about git's powerful branching model.

Branching & merging

Branching is supported in most VCSes. For example, Subversion (http://subversion.apache.org/) makes a virtue of "cheap copying"—namely, that creating a new branch does not mean making a copy of the whole source tree, so it is fast. Git's branching is just as fast. However, where Git really comes into its own is in *merging* between branches, in particular, reducing the pain of dealing with merge conflicts. This is what makes it so powerful in enabling collaborative software development.

Why Branch?

There are many reasons for creating multiple branches in a Git repo.

- You may have branches representing "stable" releases, which continue to get incremental bug fixes but no (major) new features. At the same time, you may have multiple "unstable" branches representing various new features being proposed for the next major release, and being worked on in parallel, perhaps by different groups. Those features which are accepted will need to be merged into the branch for the next stable release.
- You can create your own private branches for personal experiments. Later, if the code becomes sufficiently interesting to tell others about, you may make those branches public. Or you could send patches to the maintainer of the upstream public branch, and if they get accepted, you can pull them back down into your own copy of the public branch, and then you can retire or delete your private branch.

You may, in fact, want to add updates to different branches at different times. Switching between branches is easy.

Branching

View your branches

Use `git branch` with nothing else to see what branches your repository has:

```
$ git branch
* master
```

The branch called "master" is the default main line of development. You can rename it if you want, but it is customary to use the default. When you commit some changes, those changes are added to the branch you have checked out - in this case, master.

Create new branches

Let's create a new branch we can use for development - call it "dev":

```
$ git branch dev
$ git branch
  dev
* master
```

This only creates the new branch, it leaves your current HEAD where you remain. You can see from the * that the master branch is still what you have checked out. You can now use `git checkout dev` to switch to the new branch.

Alternatively, you can create a new branch and check it out all at once with

```
$ git checkout -b newbranch
```

Delete a branch

To delete the current branch, again use *git-branch*, but this time send the `-d` argument.

```
$ git branch -d <name>
```

If the branch hasn't been merged into master, then this will fail:

```
$ git branch -d foo
error: The branch 'foo' is not a strict subset of your current HEAD.
If you are sure you want to delete it, run 'git branch -D foo'.
```

Git's complaint saves you from possibly losing your work in the branch. If you are still sure you want to delete the branch, use `git branch -D <name>` instead.

Sometimes there are a lot of local branches which have been merged on the server, so have become useless. To avoid deleting them one by one, just use:

```
git branch -D `git branch --merged | grep -v \* | xargs`
```

Pushing a branch to a remote repository

When you create a local branch, it won't automatically be kept in sync with the server. Unlike branches obtained by pulling from the server, simply calling `git push` isn't enough to get your branch pushed to the server. Instead, you have to explicitly tell git to push the branch, and which server to push it to:

```
$ git push origin <branch_name>
```

Deleting a branch from the remote repository

To delete a branch that has been pushed to a remote server, use the following command:

```
$ git push origin :<branch_name>
```

This syntax isn't intuitive, but what's going on here is you're issuing a command of the form:

```
$ git push origin <local_branch>:<remote_branch>
```

and giving an empty branch in the `<local_branch>` position, meaning to overwrite the branch with nothing.

Merging

Branching is a core concept of a DVCS, but without good merging support, branches would be of little use.

```
git merge myBranch
```

This command merges the given branch into the current branch. If the current branch is a direct ancestor of the given branch, a *fast-forward* merge occurs, and the current branch head is redirected to point at the new branch. In other cases, a *merge commit* is recorded that has both the previous commit and the given branch tip as parents. If there are any conflicts during the merge, it will be necessary to resolve them by hand before the merge commit is recorded.

Handling a Merge Conflict

Sooner or later, if you're doing regular merges, you will hit a situation where the branches being merged will include conflicting changes to the same source lines. How you resolve this situation will be a matter of judgement (and some hand-editing), but Git provides tools you can use to try to get an insight into the nature of the conflict(s), and how best to resolve them.

Real-world examples of merge conflicts tend to be nontrivial. Here we will try to create a very simple, albeit artificial, example, to try to give you some flavour of what is involved.

Let us start with a repo containing a single Python source file, called `test.py`. Its initial contents are as follows:

```
#!/usr/bin/python3
#+
# This code doesn't really do anything at all.
#-

def func_common()
    pass
#end func_common

def child1()
    func_common()
#end child1

def child2()
    func_common()
#end child2

def some_other_func()
    pass
#end some_other_func
```

Commit this file to the repo, with a commit message saying something like “first version”.

Now create a new branch and switch to it, using the command

```
git checkout -b side-branch
```

(This second branch is to simulate work being done on the same project by another programmer.) Edit the file `test.py`, and simply swap the definitions of the functions `child1` and `child2` around, equivalent to applying the following patch:

```
diff --git a/test.py b/test.py
index 863611b..c9375b3 100644
--- a/test.py
+++ b/test.py
@@ -7,14 +7,14 @@ def func_common()
     pass
 #end func_common

-def child1()
-    func_common()
-#end child1
-
 def child2()
     func_common()
 #end child2

+def child1()
+    func_common()
+#end child1
+
 def some_other_func()
     pass
 #end some_other_func
```

Commit the update to the branch `side-branch` with a message like “swap a pair of functions around”.

Now switch back to the `master` branch:

```
git checkout master
```

This will also put you back to the previous version of `test.py`, since that was the last (in fact only) version committed to that branch.

On this branch, we now rename the function `func_common` to `common`, equivalent to the following patch:

```
diff --git a/test.py b/test.py
index 863611b..088c125 100644
--- a/test.py
+++ b/test.py
@@ -3,16 +3,16 @@
 # This code doesn't really do anything at all.
 #-

-def func_common()
+def common()
     pass
-#end func_common
+#end common

 def child1()
-    func_common()
+    common()
 #end child1

 def child2()
-    func_common()
+    common()
 #end child2

 def some_other_func()
```

Commit this change to the `master` branch, with a message like “rename func_common to common”.

Now, try to merge in the change you made on `side-branch`:

```
git merge side-branch
```

This should immediately fail, with a message like

```
Auto-merging test.py
CONFLICT (content): Merge conflict in test.py
Automatic merge failed; fix conflicts and then commit the result.
```

Just to check what `git-status(1) (http://git-scm.com/docs/git-status)` reports:

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:      test.py

no changes added to commit (use "git add" and/or "git commit -a")
```

If we look at `test.py` now, it should look like

```
#!/usr/bin/python3
#+
# This code doesn't really do anything at all.
#-

def common()
    pass
#end common

<<<<<<< HEAD
def child1()
    common()
#end child1

=======
>>>>>>> side-branch
def child2()
    common()
#end child2

def child1()
    func_common()
#end child1

def some_other_func()
    pass
#end some_other_func
```

Note those sections marked "<<<<<<< HEAD" ... "=======" ... ">>>>>>> *src-branch*": the part between the first two markers comes from the HEAD branch, the one we are merging *onto* (`master`, in this case), while the part between the last two markers comes from the branch named *src-branch*, which we are merging *from* (`side-branch`, in this case).

Assuming we know exactly what the code does, we can carefully fix up all the conflicting/duplicated parts, remove the markers, and continue the merge. But perhaps this is a large project, and no single person, not even the project leader, fully understands every corner of the code. In this case, it is helpful to at least narrow down the set of commits that lead directly to the conflict, in order to get a handle on what is going on. There is a command that you can use, `git log --merge`, which is designed specifically to be used during a merge conflict, for just this purpose. In this example, I get output something like this:

```
$ git log --merge
commit 9df4b11586b45a30bd1e090706e3ff09692fcfa7
Author: Lawrence D'Oliveiro <ldo@geek-central.gen.nz>
Date:   Thu Apr 17 10:44:15 2014 +0000

    rename func_common to common

commit 4e98aa4dbd74543d7035ea781313c1cfa5517804
Author: Lawrence D'Oliveiro <ldo@geek-central.gen.nz>
Date:   Thu Apr 17 10:43:48 2014 +0000

    swap a pair of functions around
$
```

Now, as project leader, I can look further at just those two commits, and figure out that nature of the conflict is really quite simple: one branch has swapped the order of two functions, while the other has changed the name of another function being referenced within the rearranged code.

Another useful command is `git diff --merge`, which shows a 3-way diff between the state of the source file in

the staging area, and the versions from the parent branches:

```
$ git diff --merge
diff --cc test.py
index c9375b3,863611b..088c125
--- a/test.py
+++ b/test.py
@@@ -3,18 -3,18 +3,18 @@@
  # This code doesn't really do anything at all.
  #-

--def func_common()
++def common()
      pass
--#end func_common
-
- def child2()
-     func_common()
- #end child2
++#end common

  def child1()
--    func_common()
++    common()
  #end child1

+ def child2()
 -    func_common()
++    common()
+ #end child2
+
  def some_other_func()
      pass
  #end some_other_func
$
```

Here you see, in the first two columns of each line, “+” and “-” characters indicating lines added/removed with respect to the two branches, or a space indicating no change.

Armed with this information, I can approach the problem of fixing up the conflicted file with a bit more confidence, creating the following merged version of `test.py`:

```
#!/usr/bin/python3
#+
# This code doesn't really do anything at all.
#-

def common()
    pass
#end common

def child2()
    common()
#end child2

def child1()
    common()
#end child1

def some_other_func()
    pass
#end some_other_func
```

Just to recheck, after doing `git add test.py` on the above fixed version, but before committing, do another `git`

`diff --merge`, which should produce output like:

```
diff --cc test.py
index c9375b3,863611b..088c125
--- a/test.py
+++ b/test.py
@@@ -3,18 -3,18 +3,18 @@@
  # This code doesn't really do anything at all.
  #-

--def func_common()
++def common()
      pass
--#end func_common
-
- def child2()
-     func_common()
- #end child2
++#end common

  def child1()
--    func_common()
++    common()
  #end child1

+ def child2()
 -    func_common()
++    common()
+ #end child2
+
  def some_other_func()
      pass
  #end some_other_func
```

And what does `git status` say?

```
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

       modified:   test.py
```

Now when you do like it says and enter `git commit`, Git automatically finishes the merge.

“The Stupid Content Tracker”

The git(1) (http://git-scm.com/docs/git) man page summarizes Git as “the stupid content tracker”. It is important to understand what “stupid” means in this case: it means that Git does not use elaborate algorithms to try to automatically handle merge conflicts, instead it concentrates on displaying just the relevant information to help human intelligence to resolve the conflict. Linus Torvalds has famously said that he wouldn’t trust his code to such elaborate merge conflict-resolution systems, which is why he deliberately designed Git to be “stupid”, and therefore, reliable.

Advanced

This tutorial covers some of the more advanced, multi-user features of git. For the single-user features, please go to the Single developer basics.

Checking out remote repositories

One way to check out a remote git repository is

```
$ git clone ssh://username@server.com:port/remote/path/to/repo
```

Now you have a local copy of that repository. You can use all the commands that were introduced in the Single developer basics. Once you are done, you might want to check in your changes to the central repository again.

First you want to do a `git pull` in case the repository has changed in the meantime and you might have to merge your branch with the repository. After merging, you can use `git push` to send your changes to the repository:

```
$ git pull /remote/path/to/repo
```

or

```
$ cd repo
$ git pull
```

then

```
$ git push
```

Checking out local repositories

git clone also works for local repositories:

```
$ git clone /local/path/to/repo
```

Checking out remote branches

You might also want to check out remote branches, work on them and check in your local branches. First you might want to know which branches are available:

```
$ git branch -r
$ git remote show origin
```

Get a remote branch (pull into a local branch):

```
$ git pull origin remoteBranchName:localBranchName
```

Update a remote branch (push a local branch into a remote branch):

```
$ git push origin localBranchName:remoteBranchName
```

This assumes that you have a remote repository called "origin". You can check this with `git remote`.

If you want to create a local branch from a remote branch, use:

```
$ git checkout -b mylocalbranch origin/maint
```

Deleting remote branches works like this

```
$ git push origin :remoteBranchNameToDelete
```

The following command synchronizes branches $ git fetch origin

Tags

Git allows you to specify some tags in order to focus on somethings in the history[1].

In order to add an annotated tag:

```
$ git tag -a mytag
```

or also:

```
$ git tag -a mytag my-branch
```

To add a lightweight tag:

```
$ git tag mytag
```

To force overwriting existing tag:

```
$ git tag -f mytag HEAD
```

To display previous tags:

```
$ git tag
```

Tags can be pushed to remote with

```
$ git push --tags
```

Tags vs Branches

Both tags and branches point to a commit, they are thus aliases for a specific hash and will save you time by not requiring to type in a hash.

The difference between tags and branches are that a branch always points to the top of a development line and will change when a new commit is pushed whereas a tag will not change. Thus tags are more useful to "tag" a specific version and the tag will then always stay on that version and usually not be changed.

Create and Apply a Patch

In order to create a plain text patch (series) for the changes between origin and master, use

```
$ git format-patch origin/master
```

In order to apply a submitted plain text patch (series), use

```
$ git apply --stat  P1.txt  #see the stats, how much will the path change?
$ git apply --check P1.txt  #check for problems
$ git am < P1.txt           #apply the patches in the correct order
```

References

1. http://git-scm.com/book/en/Git-Basics-Tagging

Internal structure

Understanding something of the internal structure of Git is crucial to understanding how Git works.

Naked Git Structure

The following is a freshly initialized git v1.9.0 repository.[2]

```
.
└── .git/
    ├── HEAD
    ├── branches/
    ├── config
    ├── description
    ├── hooks/
    │   ├── applypatch-msg.sample
    │   ├── commit-msg.sample
    │   ├── post-update.sample
    │   ├── pre-applypatch.sample
```

```
    │   ├── pre-commit.sample
    │   ├── prepare-commit-msg.sample
    │   ├── pre-push.sample
    │   ├── pre-rebase.sample
    │   └── update.sample
    ├── info/
    │   └── exclude
    ├── objects/
    │   ├── info/
    │   └── pack/
    └── refs/
        ├── heads/
        └── tags/
```

Additional files and folders may appear as activity happens on the repository.

Specific Files

COMMIT_EDITMSG

The message for a commit being made is saved here by a text editor.

FETCH_HEAD

Information is saved here from the last `git-fetch(1) (http://git-scm.com/docs/git-fetch)` operation, for use by a later `git-merge(1) (http://git-scm.com/docs/git-merge)`.

HEAD

HEAD indicates the currently checked out code. This will usually point to the branch you're currently working on.

You can also enter what git calls a "detached HEAD" state, where you are not on a local branch. In this state the HEAD points directly to a commit rather than a branch.

config

The configuration file for this git repository. It can contain settings for how to manage and store data in the local repository, remote repositories it knows about, information about the local user and other configuration data for git itself.

You can edit this file with a text editor, or you can manage it with the `git-config(1) (http://git-scm.com/docs/git-config)` command.

description

Used by repository browser tools - contains a description of what this project is. Not normally changed in non-shared repositories.

index

This is the staging area. It contains, in a compact form, all changes to files that have been staged for the next commit.

info/exclude

This is your own personal exclude file for your copy of the repo.

info/refs

If this file exists, it contains definitions, one to a line, of branches (both local and remote) and tags defined for the repository, in addition to ones that may be defined in individual files in `refs/heads` and `refs/tags`. This file seems to be used for large repositories with lots of branches or tags.

ORIG_HEAD

Operations that change commit history on the current branch save the previous value of HEAD here, to allow recovery from mistakes.

Folders Containing Other Files

branches

Never seems to be used.

hooks

Contains scripts to be run when particular events happen within the git repository. Git gives you a set of initial example scripts, with `.sample` on the ends of their names (see the tree listing above); if you take off the `.sample` suffix, Git will run the script at the appropriate time.

Hooks would be used, for example, to run tests before creating each commit, filter uploaded content, and implement other such custom requirements.

logs

The reflogs are kept here.

objects

This is where all the files, directory listings, commits and such are stored.

There are both unpacked objects in numbered directories under this, and "packs" containing many compressed objects within a pack directory. The uncompressed objects will be periodically collected together into packs by automatic "git gc" runs.

refs/heads

Can contain one file defining the head commit for each local branch (but see `info/refs` above).

refs/remotes

Can contain one subdirectory for each remote repository you have defined. Within each subdirectory, there is a file defining the tip commit for each branch on that remote.

refs/tags

Can contain one file defining the commit corresponding to each tag (but see `info/refs` above).

svn

This directory will appear if you use `git-svn(1) (http://git-scm.com/docs/git-svn)` to communicate with a Subversion server.

Basic Concepts

Object Types

A Git repository is made up of these object types:

- A *blob* holds the entire contents of a single file. It doesn't hold any information about the name of the file or any other metadata, just the contents.
- A *tree* represents the state of a directory tree. It contains the pathnames of all the component files and their modes, along with the IDs of the blobs holding their contents. Note that there is no representation for a directory on its own, so a Git repository cannot record the fact that subdirectories were created or deleted, only the files in them.
- A *commit* points to a tree representing the state of the source tree as of immediately after that commit. It also records the date/time of the commit, the author/committer information, and pointers to any parent(s) of that commit, representing the immediately-prior state of the source tree.
- A *tag* is a name pointing to a commit. These are useful, for example, to mark release milestones. Tags can optionally be digitally signed, to guarantee the authenticity of the commit.
- A *branch* is a name pointing to a commit. The difference between a branch and a tag is that, when a branch is the currently-checked-out branch, then adding a new commit will automatically update the branch pointer to point to the new commit.

Blobs, trees and commits all have IDs which are computed from SHA-1 hashes of their contents. These IDs allow different Git processes on different machines to tell whether they have identical copies of things, without having to transfer their entire contents over. Because SHA-1 is a cryptographically strong hash algorithm, it is practically impossible to make a change to the contents of any of these objects without changing its ID. Git doesn't prevent you

from rewriting history, but you cannot hide the fact that you have done so.

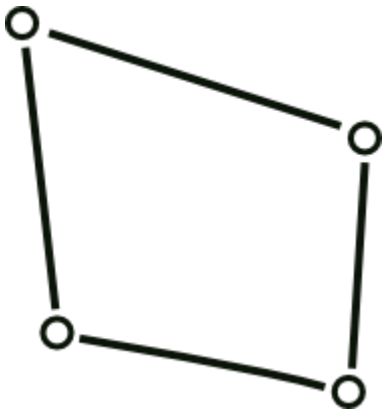
A commit may have 0, 1 or more parents. Typically there is only one commit with no parents—a *root commit*—and that is the first commit to the repository. A commit which makes some change to one branch will have a single parent, the previous commit on that branch. A commit which is a merge from two or more branches will have two or more parent commits.

Note that a branch points to a *single* commit; the chain of commits is implicit in the parent(s) of that commit, and their parents, and so on.

Topology Of Commits

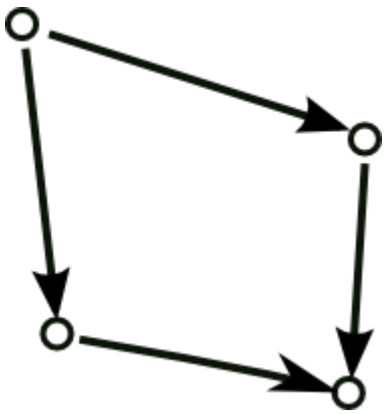
The commit history in Git is arranged as a *directed acyclic graph* (DAG). To understand what this means, let's take the terms step by step.

- In mathematical terms, a *graph* is a bunch of points (*nodes*) connected by lines (*edges*).



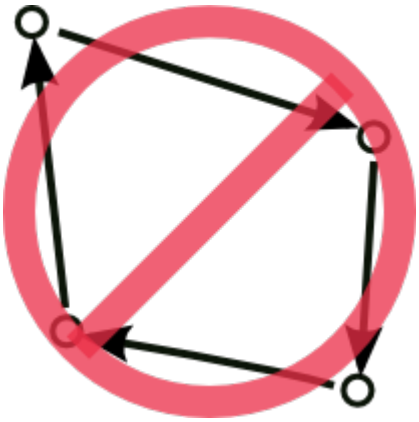
just a graph

- A *directed* graph is one where each edge has a direction, represented here by an arrowhead. Note that the arrow points from the child to the parent, not the other way round; it is the child that records who its parent(s) are, the parent does not record who its children are, because the set of children can change at any time, but the parent cannot change without invalidating its SHA-1 hash.



directed edges

- *Acyclic* means that, if you start from any point and traverse edges in the direction of the arrows, you can never get back to your starting point, no matter what choice you make at any branch. No child can ever be a (direct or indirect) parent of itself!



cycles not allowed

In Git terms, each node represents a commit, and the lines and arrows represent parent-child relationships. Banning cycles simply means that a commit cannot be a (direct or indirect) parent or a child of itself!

The Reflog

The *reflog*s record changes that are not saved as part of the commit history—things like rebases, fast-forward merges, resets and the like. There is one reflog per branch. The reflog is not a public part of the repository, it is strictly specific to your local copy, and information is only kept in it for a limited time (2 weeks by default). It provides a safety net, allowing you to recover from mistakes like deleting or overwriting things you didn't mean to.

Reachability And Garbage Collection

A commit is *reachable* if it is pointed to by a branch, tag or reflog entry, or is a parent of a commit which is reachable. A tree is correspondingly reachable if it is pointed to by a reachable commit, and a blob is reachable if it is pointed to by a reachable tree. Other commit/tree/blob objects are *unreachable*, and are not really serving any purpose beyond taking up space.

It is quite normal for your repositories to accumulate unreachable objects over time, perhaps as a result of aborted commits, deletion of unwanted branches, that kind of thing. Such objects will be deleted from the repository by a `git gc` command. This is also done automatically every now and then by some other commands, so it is rarely necessary to invoke `git gc` explicitly.

Footnotes

1. ^ Generated with tree v1.5.1.1 using `tree -AnaF`.

Rebasing

Rebasing

If you want to change the commit messages, the order or the number of commits, use:

```
$ git rebase -i HEAD~3
```

where HEAD can also be any other branch you are on and you can work on any number of commits, not just 3. You can delete commits, merge them together with "squash" or change their order. If something goes wrong, use

```
$ git rebase -i --abort
```

Note that this will change all the commit-ids in the moved commits. These are because the commit-id also takes the commit's history in to account and the same change in a different place is considered a different commit by git. Rebasing shared changes can make combining those changes down the track difficult - you will not normally want to rebase any changes which has been incorporated in to somebody else's or a shared repository.

Submodules and Superprojects

A **superproject** is a new aspect of git which has been in development for a long while. It addresses the need for better control over numerous git repositories. The porcelain for the superproject functionality is fairly new and was only recently released with Git v1.5.3.

The current terminology distinguishes a *git repository* outside of a superproject from one inside by calling the one inside a **submodule**. If consistency will help you grok this chapter, you can think of either a *submodule* as a *subproject,* or a *superproject* as a *supermodule.* It really doesn't make too much sense why either of these terms would have a *sub-* or *super-* prefix without their alternative; nonetheless, that's how the official Git documentation will refer to them.

The only git application specific to the submodule/superproject functionality is **git-submodule**.

Superprojects

A **Superproject**, is simply a git repository. To create a superproject, simply *git init* any directory, and *git submodule add* all of the git archives you wish to include. A quick aside, you can not currently *git submodule add* git repositories that are direct children within the same directory.[3]

The resulting structure will look similar to this:

```
|- superproject
  |- submodule (git archive) [a]
  |- submodule [b]
  |- submodule [c]
  |- submodule [d]
```

When someone pulls down the superproject, they will see a series of empty folders for each submodule. They can then *git submodule init* all of those that they wish to utilize.

Submodules

A *git archive* is said to become a *submodule* the second after you execute *git submodule add* in another git repository.

Work Flow

The work flow of superprojects, and submodules should generally adhere to the following:

1. Make change in submodule
2. *git commit* change in submodule
3. *git commit* change in superproject
4. *git submodule update* to push change to the individual repositories that predate the superproject.

Footnotes

1. ^ Well that isn't true at all, Git supports this as of v1.5.3, but the official porcelain doesn't. You can *git init* a parent directory, and create your own ".gitmodules", then follow it up with a *git submodule init*. Generally speaking though, what the porcelain doesn't cover is outside of the scope of this book.

Interoperation

This chapter is about inter-operation between Git and other systems.

Git is a great version control system, but it is not the only one. Many projects still use version control systems that predate git or alternate DSCMs. Thankfully, you can still use git to participate in many of these projects, thanks to programs like `git-svn`, `git-cvsimport`, etc.

You can also work with Git directly within many text editors, IDEs and other tools.

Customizing your command line prompt

This Bash customization script will modify your command line prompt to include some extra details whenever you're in a Git directory. It is based on Mike Stewart's work.[1]

The current branch name is shown in red (if it's got outstanding changes) or green (if there are no changes), followed by the current path. The hostname is also shown in a low-lighted colour at the beginning.

```
Color_Off="\[\033[0m\]"       # Text Reset
IBlack="\[\033[0;90m\]"       # High-intensity black
Green="\[\033[0;32m\]"        # Green
Yellow="\[\033[0;33m\]"       # Yellow
Red="\[\033[0;91m\]"          # Red
Hostname="\h"                 # Hostname (up to the first dot)
PathShort="\w"                # Current working directory (short version)
export PS1=$IBlack$Hostname$Color_Off'$(git branch &>/dev/null;\
if [ $? -eq 0 ]; then \
  echo "$(echo `git status` | grep "nothing \(added \)\?to commit" > /dev/null 2>&1; \
  if [ "$?" -eq "0" ]; then \
    # Clean repository - nothing to commit
    echo "'$Green'"$(__git_ps1 " (%s)"); \
  else \
    # Changes to working tree
    echo "'$Red'"$(__git_ps1 " {%s}"); \
  fi) '$BYellow$PathShort$Color_Off'\$ "; \
else \
  # Prompt when not in GIT repo
  echo " '$Yellow$PathShort$Color_Off'\$ "; \
fi)'
```

References

1. Stewart, Mike, *Ultimate GIT PS1 bash prompt*, http://mediadoneright.com/content/ultimate-git-ps1-bash-prompt

git-svn

Overview

Subversion is an extremely popular version control system, and there are many OSS and proprietary projects that use it. Git comes with an excellent utility, `git-svn`, that allows a user to both track a project maintained in a subversion repository, as well as participate. Users can generate local patches to send to a mailing list, or even commit changes directly back into the repository (given that they have commit access, of course).

Getting Started

To begin using git with a subversion-hosted project, you must create a local repository for your files, as well as configure `git-svn`. Typically you'll use commands much like the following:

```
mkdir project
cd project
git-svn init <url to repository root> -T/path/to/trunk
git-svn fetch -r <first rev>:HEAD
```

Usually when working with subversion repositories, you're given the full project URL. To determine the url to the repository root, you can issue the following command:

```
svn info <full project URL>
```

There will be a line in the output stating the repository root. The path to trunk is simply the rest of the URL that follows. It is possible to simply give `git-svn` the full project URL, but doing it this way gives you greater flexibility should you end up working on subversion branches down the line.

Also note the "first rev" argument. You could simply use "1", as that is guaranteed to work, but it would likely take a very long time (especially if the repository is over a slow network). Usually, for an entrenched project, the last 10-50 revs is sufficient. Again, `svn info` will tell you what the most recent revision is.

Interacting with the repository

Chances are if you're using git instead of svn to interact with a subversion repository, its because you want to make use of offline commits. This is very easy to do, if you keep a few caveats in mind. Most important is to never use "git pull" while in a branch from which you plan to eventually run `git-svn dcommit`. Merge commits have a tendency to confuse git-svn, and you're able to do most anything you'd need without git pull.

The most common task that I perform is to combine the change(s) I'm working on with the upstream subversion changes. This is the equivalent to an `svn update`. Here's how its done:

```
git stash  # stash any changes so you have a clean tree
git-svn fetch # bring down the latest changes
git rebase trunk
git stash apply
```

The first and last steps are unnecessary if your tree is clean to begin with. That leaves "git rebase trunk" as the primary operation. If you're unfamiliar with rebasing, you should go read the documentation for git-rebase (http://www.kernel.org/pub/software/scm/git/docs/git-rebase.html). The jist of it is that your local commits are now on top of svn HEAD.

Dealing with local changes

It often happens that there are some changes you want to have in a repository file that you do not want to propagate. Usually this happens with configuration files, but it could just as easily be some extra debugging statements or anything else. The danger of committing these changes is that you'll run "git-svn dcommit" in the branch without weeding out your changes. On the other hand, if you leave the changes uncommitted, you lose out on git's features for those changes, and you'll have to deal with other branches clashing with the changes. A dilemma!

There are a couple of solutions to this issue. Which one works better is more a matter of taste than anything. The first approach is to keep a "local" branch for each branch that you want to have local changes. For example, if you have local changes that you want in the branch "foo", you would create a branch "foo-local" containing the commit(s) with the changes you want to keep local. You can then use rebase to keep "foo" on top of "foo-local". e.g.:

```
git rebase trunk foo-local
git rebase foo-local foo
```

As the example code implies, you'll still spend most of your time with "foo" checked out, rather than "foo-local." If you decide on a new change that you want to keep locally, you're again faced with two choices. You can checkout "foo-local" and make the commit, or you can make the commit on "foo" and then cherry-pick the commit from foo-local. You would then want to use git-reset (http://www.kernel.org/pub/software/scm/git/docs/git-reset.html) to remove the commit from "foo".

As an alterative to the rebase-centric approach, there is a merge-based method. You still keep your local changes on a separate branch, as before. With this method, however, you don't have to keep "foo" on top of "foo-local" with rebase. This is an advantage, because 1) its more to type, and 2) historically, rebase has often asked you to resolve the same conflict twice if any conflicts occur during the first rebase.

So instead of using rebase, you create yet another branch. I call this the "build" branch. You start the build branch at whatever commit you want to test. You can then "git merge" the local branch, bringing all your changes into one tree. "But I thought you should avoid merge?" you ask. The reason I like to call this branch the "build" branch is to dissuade me from using "git-svn dcommit" from it. As long as its not your intention to run dcommit from the branch, the use of merge is acceptable.

This approach can actually be taken a step further, making it unnecessary to rebase your topic branch "foo" on top of trunk every day. If you have several topic branches, this frequent rebasing can become quite a chore. Instead:

```
git checkout build
git reset --hard trunk # Make sure you dont have any important changes
git merge foo foo-local # Octopus merges are fun
```

Now build contains the changes from trunk, foo, and foo-local! Often I'll keep several local branches. Perhaps one branch has your local configuration changes, and another has extra debugging statements. You can also use this approach to build with several topic branches in the tree at once:

```
git merge topic1 topic2 config debug...
```

Unfortunately, the octopus merge is rather dumb about resolving conflicts. If you get any conflicts, you'll have to perform the merges one at a time:

```
git merge topic1
git merge topic2
git merge local
...
```

Sending changes upstream

Eventually, you'll want your carefully crafted topic branches and patch series to be integrated upstream. If you're lucky enough to have commit access, you can run "git-svn dcommit". This will take each local commit in the current branch and commit it to subversion. If you had three local commits, after dcommit there would be three new commits in subversion.

For the less fortunate, your patches will probably have to be submitted to a mailing list or bug tracker. For that, you can use git-format-patch (http://www.kernel.org/pub/software/scm/git/docs/git-format-patch.html). For example, keeping with the three-local-commits scenario above:

```
git format-patch HEAD~3..
```

The result will be three files in $PWD, 0001-commit-name.patch, 0002-commit-name.patch, and 0003-commit-name.patch. You're then free to mail these patches of or attach them to a bug in Bugzilla. If you'll be mailing the patches, however, git can help you out even a little further. There is the git-send-email (http://www.kernel.org/pub/software/scm/git/docs/git-send-email.html) utility for just this situation:

```
git send-email *.patch
```

The program will ask you a few questions, most important where to send the patches, and then mail them off for you. Piece of cake!

Of course, this all assumes that you have your patch series in perfect working order. If this is not the case, you should read about "git rebase -i".

Examples

Get Pywikipedia:

```
$ git svn init http://svn.wikimedia.org/svnroot/pywikipedia/trunk/pywikipedia/
Initialized empty Git repository in .../.git/
$ git svn fetch -r 1:HEAD
...
r370 = 318fb412e5d1f1136a92d079f3607ac23bde2c34 (refs/remotes/git-svn)
        D       treelang_all.py
        D       treelang.py
W: -empty_dir: treelang.py
W: -empty_dir: treelang_all.py
r371 = e8477f292b077f023e4cebad843e0d36d3765db8 (refs/remotes/git-svn)
        D       parsepopular.py
W: -empty_dir: parsepopular.py
r372 = 8803111b0411243af419868388fc8c7398e8ab9d (refs/remotes/git-svn)
        D       getlang.py
W: -empty_dir: getlang.py
r373 = ad935dd0472db28379809f150fcf53678630076c (refs/remotes/git-svn)
        A       splitwarning.py
...
```

Get AWB (AutoWikiBrowser):

```
$ git svn init svn://svn.code.sf.net/p/autowikibrowser/code/
Initialized empty Git repository in .../.git/
$ git svn fetch -r 1:HEAD
```

```
...
r15 = 086d4ff454a9ddfac92edb4013ec845f65e14ace (refs/remotes/git-svn)
        M       AWB/AWB/Main.cs
        M       AWB/WikiFunctions/WebControl.cs
r16 = 14f49de6b3c984bb8a87900e8be42a6576902a06 (refs/remotes/git-svn)
        M       AWB/AWB/ExitQuestion.Designer.cs
        M       AWB/WikiFunctions/GetLists.cs
        M       AWB/WikiFunctions/Tools.cs
r17 = 8b58f6e5b21c91f0819bea9bc9a8110c2cab540d (refs/remotes/git-svn)
        M       AWB/AWB/Main.Designer.cs
        M       AWB/AWB/Main.cs
        M       AWB/WikiFunctions/GetLists.cs
r18 = 51683925cedb8effb274fadd2417cc9b1f860e3c (refs/remotes/git-svn)
        M       AWB/AWB/specialFilter.Designer.cs
        M       AWB/AWB/specialFilter.cs
r19 = 712edb32a20d6d2ab4066acf056f14daa67a9d4b (refs/remotes/git-svn)
        M       AWB/WikiFunctions/WPEditor.cs
r20 = 3116588b52a8e27e1dc72d25b1981d181d6ba203 (refs/remotes/git-svn)
...
```

Beware, this downloading operation can take one hour.

Hosting

Here is where you can find a list of public repositories. Feel free to pull down a sample project and play with it.

Free, public, and open source

This is a current list of repositories that will host anything (within reason) free under the condition that it is appropriately licensed and open-source.

1. NotABug (https://notabug.org/)
2. git.sv.gnu.org (http://git.sv.gnu.org/) Free Software Foundation's official GNU Savannah Git hosting
3. repo.or.cz (http://repo.or.cz) Git's primary free repository is maintained by Petr Baudis, who is also the maintainer of the git home page. Numerous notable projects are synced to that repository such as Postgres, and Linus's copy of the linux kernel tree. The repository and all of its projects can be inspected on the web thanks to the mod_perl gitweb interface.

The following sites offer free hosting and other services regardless of a project's license.

1. GitHub (https://github.com/)
2. GitLab (https://gitlab.com)

Birds of a Feather (BOF)

Please do not ask to create a project on these public repositories unless your project relates to that of the repository.

1. git.kernel.org (http://git.kernel.org/) Official Kernel git repository

2. gitweb.freedesktop.org (http://gitweb.freedesktop.org/) X Window System specific git repository
3. git.debian.org (http://git.debian.org/) Debian specific, hosted by Debian.

Public project specific repositories

Other

1. github.com (http://github.com/) Offers free and paid repository management with some custom tools.
2. Assembla (http://www.assembla.com/free_git_hosting) Offers free public or private repository(500Mb) with Trac (http://trac.edgewall.org/) tickets or integrated tickets (http://www.assembla.com/tour/tools_tickets). Commercial subscriptions costs 19$/month for 2Gb of disk space.

Setting up a Server

A Git server is not supposed to host any uncommitted file, so its repositories should be initialized with "bare":

```
git init --bare /repositories/repo1
```

Now its files are encrypted and can't be read as flat files from the server.

The distributed repositories can then be initialized with `git clone`, updated with `git pull`, and submitted to the server with `git push`.

To avoid any user to erase the server branches when pushing, the branches can be locked, forcing the users to create some pull requests with their changes to validate before merging.

git-daemon

`git-daemon` is a simple server for git repositories.

The daemon makes git repositories available over the `git://` protocol, which is very efficient, but insecure. Where security is a concern, pull with SSH instead. The `git://` protocol listens on port 9418 by default.

The daemon *can* be configured to allow pushing, however there is **no authentication whatsoever**, so that is almost always a very bad idea. It may be appropriate in a closed LAN environment, where everyone can be trusted, however

when in doubt, SSH should be used to push.

Gitosis

Gitosis is a tool to secure centralized Git repositories, permitting multiple maintainers to manage the same project at once, by restricting the access to only over a secure network protocol.

Installing Gitosis

Checkout the Gitosis Repository

To install Gitosis, you first must have the Git client installed. Once installed, checkout a copy of Gitosis from its repository:

```
git clone git://eagain.net/gitosis.git
```

Install:

```
cd gitosis
python setup.py install
```

Create a User to Manage the Repositories

Create a user to manage the repositories:

```
sudo adduser \
  --system \
  --shell /bin/sh \
  --gecos 'git version control' \
  --group \
  --disabled-password \
  --home /home/git \
  git
```

If you don't already have a public RSA key, create one on your local computer:

```
ssh-keygen -t rsa
```

Copying Your Public Key to the Gitosis Server

Copy this key to the Gitosis server. Assuming you are in your home directory:

```
scp .ssh/id_rsa.pub user@example.com:/tmp
```

Setting up Gitosis

Initializing Gitosis

Initialize Gitosis:

```
sudo -H -u git gitosis-init < /tmp/id_rsa.pub
```

Upon success, you will see:

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

Ensure the Git post-update hook has the correct permissions:

```
sudo chmod 755 /home/git/repositories/gitosis-admin.git/hooks/post-update
```

Configuring Gitosis

Clone the Gitosis Repository

Gitosis creates its own Git repository. To configure Gitosis, you will clone this repository, set your configuration options, then push your configuration back to the Gitosis server.

Cloning the Gitosis repository:

```
git clone git@example.com:gitosis-admin.git
cd gitosis-admin
```

Creating a Repository

Edit `gitosis.conf`

An example of a default gitosis.conf:

```
[gitosis]
```

```
[group gitosis-admin]
writable = gitosis-admin
members = jdoe
```

Defining Groups, Members, Permissions, and Repositories

You can define groups of members and what permissions they will have to repositories like so:

```
[group blue_team]
members = john martin stephen
writable = tea_timer coffee_maker
```

In this example, anyone in the group `blue_team`, in this case john, martin, and stephen, will be able to write to the Git repositories `tea_timer` and `coffee_maker`

Save, commit, and push this file.

```
git commit -am "Give john, martin, and stephen access to the repositories tea_timer and
coffee_maker."
git push
```

Creating a Repository

Next, create one of the repositories. You'll want to change to the directory where you you want to store your local copy of the Git repository first.

Create the repository:

```
mkdir tea_timer
cd tea_timer
git init
git remote add origin git@example.com:tea_timer.git
# Add some files and commit.
git push origin master:refs/heads/master
# The previous line links your local branch master to the remote branch master so you can
automatically fetch and merge with git pull.
```

Adding Users to a Repository

Users are identified by their public RSA keys. Gitosis keeps these keys inside the directory `keydir` within the gitosis-admin repository. Users are linked to their Git username by the name of the key file. For example, adding an RSA key to `keydir/john.pub` will link the user john to the machine defined by the RSA within `john.pub`. Keys *must* end in .pub!

Add a user:

```
cd gitosis-admin
cp /path/to/rsa/key/john.pub keydir/
git add keydir/*
git commit -am "Adding the john account."
git push
```

John can now clone the git repositories he has access to as defined by `gitosis.conf`. In this case, he can both read and write to the repository as he has `writable` permissions.

External Links

- Scie.nti.st (http://scie.nti.st/2007/11/14/hosting-git-repositories-the-easy-and-secure-way)

Gerrit Code Review

Gerrit (http://code.google.com/p/gerrit/) is a web-based code review tool for projects using the Git VCS. It allows both a streamlined code review process, and a highly-configurable hierarchy for project members. Typically, any user may submit patches ("changesets") to the server for review. Once someone has reviewed the changes to a sufficient degree, they are merged into the main line of development, which can then be pulled.

Overview

Implementation

Server

Gerrit uses a dedicated Jetty server, which is typically accessed via reverse proxy. Here's an example configuration for Apache:

```
<VirtualHost *>
    ProxyRequests Off
    ProxyVia Off
    ProxyPreserveHost On
    <Proxy *>
        Order deny,allow
        Allow from all
    </Proxy>

    # Reverse-proxy these requests to the Gerrit Jetty server
    RedirectMatch  ^/gerrit$               /gerrit/
    ProxyPass       /gerrit/               http://127.0.0.1:8081/gerrit/
</VirtualHost>
```

JGit

Gerrit uses JGit, a Java implementation of git. This has several limitations: a sacrifice in speed, and some unimplemented features[1]. For example, content-level merges are not supported in JGit - users must pull and merge changes, then re-upload them to the Gerrit server in cases where content-level merges are required.

Permissions model

Gerrit's permissions model allows a highly configurable hierarchy regarding who can submit patches, and who can review patches. This can be flattened out as desired, or ramped up, depending on the development model used for

each project.

Hooks

The server allows scripts to run in response to certain events. Hook scripts live in $GIT_INSTALL_BASE/hooks, and must be set executable on Unix systems. A hook could - for instance - allow a project to install an automated gatekeeper to vote +2 'submit approved' when sufficient +1 votes 'looks good to me' have been received:

```
#!/usr/bin/perl
#
# comment-added: hook for a +2 approval from a simple quorum of +1 votes.
#
# (c) 2012 Tim Baverstock
# Licence: Public domain. All risk is yours; if it breaks, you get to keep both pieces.

$QUORUM = 2; # Total number of +1 votes causing a +2
$PLEBIANS = 'abs(value) < 2'; # or 'value = 1' to ignore -1 unvotes
$AUTO_SUBMIT_ON_QUORACY = '--submit'; # or '' for none
$AND_IGNORE_UPLOADER = 'and uploader_account_id != account_id'; # or '' to let uploaders votes
count

$GERRIT_SSH_PORT = 29418;
$SSH_PRIVATE_KEY = '/home/gerrit2/.ssh/id_rsa';
$SSH_USER_IN_ADMIN_GROUP = 'devuser';

# Hopefully you shouldn't need to venture past here.

$SSH = "ssh -i $SSH_PRIVATE_KEY -p $GERRIT_SSH_PORT $SSH_USER_IN_ADMIN_GROUP\@localhost";

$LOG = "/home/gerrit2/hooks/log.comment-added";
open LOG, ">>$LOG" or die;

sub count_of_relevant_votes {
        # Total selected code review votes for this commit
        my $relevance = shift;
        $query = "
                select sum(value) from patch_sets, patch_set_approvals
                where patch_sets.change_id = patch_set_approvals.change_id
                and patch_sets.patch_set_id = patch_set_approvals.patch_set_id
                and revision = '$V{commit}'
                and category_id = 'CRVW'
                and $relevance
                $AND_IGNORE_UPLOADER
                ;";
        $command = "$SSH \"gerrit gsql -c \\\"$query\\\"\"";
        #print LOG "FOR... $command\n";
        @lines = qx($command);
        chomp @lines;
        #print LOG "GOT... ", join("//", @lines), "\n";
        # 0=headers 1=separators 2=data 3=count and timing.
        return $lines[2];
}

sub response {
        my $review = shift;
        return "$SSH 'gerrit review --project=\"$V{project}\" $review $V{commit}'";
}

# ######################
# Parse options

$key='';
while ( $_ = shift @ARGV ) {
        if (/^--(.*)/) {
                $key = $1;
```

```
        }
        else {
                $V{$key} .= " " if exists $V{$key};
                $V{$key} .= $_;
        }
}
#print LOG join("\n", map { "$_ = '$V{$_}'" } keys %V), "\n";

# ######################
# Ignore my own comments

$GATEKEEPER="::GATEKEEPER::";
if ($V{comment} =~ /$GATEKEEPER/) {
        print LOG localtime() . "$V{commit}: Ignore $GATEKEEPER comments\n";
        exit 0;
}

# ######################
# Forbear to analyse anything already +2'd

$submittable = count_of_relevant_votes('value = 2');
if ($submittable > 0) {
        print LOG localtime() . "$V{commit} Already +2'd by someone or something.\n";
        exit 0;
}

# ######################
# Look for a consensus amongst qualified voters.

$plebicite = count_of_relevant_votes($PLEBIANS);

#if ($V{comment} =~ /TEST:(\d)/) {
#        $plebicite=$1;
#}

# ######################
# If there's a quorum, approve and submit.

if ( $plebicite >= $QUORUM ) {
        $and_submitting = ($AUTO_SUBMIT_ON_QUORACY ? " and submitting" : "");
        $review = " --code-review=+2 --message=\"$GATEKEEPER approving$and_submitting due to
$plebicite total eligible votes\" $AUTO_SUBMIT_ON_QUORACY";
}
else {
        $review = " --code-review=0 --message=\"$GATEKEEPER ignoring $plebicite total eligible
votes\"";
        print LOG localtime() . "$V{commit}: $review\n";
        exit 0; # Perhaps don't exit here: allow a subsequent -1 to remove the +2.
}

$response = response($review);

print LOG localtime() . "RUNNING: $response\n";
$output = qx( $response 2>&1   );
if ($output =~ /\S/) {
        print LOG localtime() . "$V{commit}: output from commenting: $output";
        $response = response(" --message=\"During \Q$review\E: \Q$output\E\"");
        print LOG localtime() . "WARNING: $response\n";
        $output = qx( $response 2>&1   );
        print LOG localtime() . "ERROR: $output\n";
}

exit 0;
```

Setup

Importing project into Gerrit

Whether or not you're permitted to do this depends on which access group(s) you're in, and where you're trying to do this. It is most useful for pushing pre-existing repos to the server, but it can in theory be used whenever you want to push changes which are not to be reviewed.

Use:

```
$ git push gerrit:project HEAD:refs/heads/master
```

since you want to directly push into the branch, rather than create code reviews. Pushing to `refs/for/*` creates code reviews which must be approved and then submitted. Pushing to `refs/heads/*` bypasses review entirely, and just enters the commits directly into the branch. The latter does not check committer identity, making it appropriate for importing past project history.

The correct permission setup can be found here: http://stackoverflow.com/questions/8353988/how-to-upload-a-git-repo-to-gerrit. In addition, "Push Merge Commit" for "refs/*" may be needed for some repositories (for details see http://code.google.com/p/gerrit/issues/detail?id=1072).

Use

Registering

Submitting changes for review

Simply push into the project's magical `refs/for/$branch` (typically master) ref using any Git client tool:

```
$ git push ssh://user@host:29418/project HEAD:refs/for/master
```

Each new commit uploaded by the git push client will be converted into a change record on the server. The remote ref `refs/for/$branch` is not actually created by Gerrit, even though the client's status messages may say otherwise. Pushing to this magical branch submits changes for review. Once you have pushed changes, they must be reviewed and submitted for merging to whatever branch they apply to. You can clone/pull from gerrit as you would any other git repo (no refs/for/branch, just use the branch name):

```
$ git clone ssh://user@host:29418/project
$ git pull ssh://user@host:29418/project master:testbranch
```

Gerrit currently has no `git-daemon`, so pulling is via ssh, and therefore comparatively slow (but secure). You can run `git-daemon` for the repositories to make them available via `git://`, or configure Gerrit to replicate changes to another git repository, where you can pull from.

Since you will be frequently working with the same Gerrit server, add an SSH host block in ~/.ssh/config to remember your username, hostname and port number. This permits the use of shorter URLs on the command line as shown, such as:

```
$ tail -n 4 ~/.ssh/config
Host gerrit
    Hostname host.com
    Port 29418
    User john.doe
$ git push gerrit:project HEAD:refs/for/master
```

Alternatively, you can also configure your remotes in git's configuration file by issuing:

```
$ git config remote.remote_name.fetch +refs/heads/*:refs/remotes/origin/*
$ git config remote.remote_name.url ssh://user@host:29418/project_name.git
```

This should be done automatically for you if you've started your local repository off of Gerrit's project repository using

```
$ git clone ssh://user@host:29418/project_name.git
```

Note that the Gerrit server has its own sshd with different host keys. Some ssh clients will complain about this bitterly.

Re-submitting a changeset

This is useful when there are issues with a commit you pushed. Maybe you caught them, maybe the reviewer did – either way, you want to submit changes for review, replacing the bad changes you submitted previously. This keeps code review in one place, streamlining the process. First, squash your changes into one commit using git rebase -i – you will probably want to change the commit message.

This doesn't actually replace the previous push, it just adds your updated changeset as a newer version.

You can provide a Change-Id line in your commit message: it must be in the bottom portion (last paragraph) of a commit message, and may be mixed together with the Signed-off-by, Acked-by, or other such footers. The Change-Id is available in the metadata table for your initial pushed commit. In this case, Gerrit will automatically match this changeset to the previous one.

Alternatively, you can push to a special location: the `refs/changes/*` branch. To replace changeset 12345, you push to `refs/changes/12345`. This number can be found in the URL when looking at the changeset you wish to replace: `#change,12345`. You can also find it in the "download" section for the changeset: `...refs/changes/45/12345/1` (choose the middle number: ignore `/45/` and omit the trailing `/1`). In this case, your push becomes:

```
$ git rebase -i HEAD~2 # squash the relevant commits, probably altering the commit message
$ git push gerrit:project a95cc0dcd7a8fd3e70b1243aa466a96de08ae731:refs/changes/12345
```

Reviewing and merging changes

See also

- Current official documentation (https://gerrit-review.googlesource.com/Documentation/index.html)

- Download page (http://gerrit-releases.storage.googleapis.com/index.html)
- MW:Gerrit

References

1. Re: Failure to submit due to path conflict but no real conflict. (http://groups.google.com/group/repo-discuss/msg/2ee99ed99609bb4b) (Shawn Pearce)

GitHub

GitHub is a Git repository providing two formulae:

1. Free public.
2. Paying private.

GitHub logo.

Functionalities

Each repo can have its own wiki and issues tracking. The documentation is in markdown, and can be written directly on line.

The online review provides some dif.

Webhooks

It's possible to validate every pull request with webhooks for the continuous integration, like Jenkins[1].

References

1. https://developer.github.com/webhooks/

Bitbucket

Bitbucket is a web-based hosting service for projects that use either the Mercurial or Git revision control systems. Bitbucket offers both commercial plans and free accounts. Free accounts support up to five users and an unlimited number of private repositories.

Moreover, it provides some webhooks for the continuous integration[1].

Bitbucket logo.

References

1. https://confluence.atlassian.com/bitbucket/manage-webhooks-735643732.html

Repository on a USB stick

Instead of having to resort to a hosting company to store your central repository, or to rely on a central server or internet connection to contribute changes to a project, it's quite possible to use removable memory to exchange and update local repositories.

The basic steps are:

1. Mount the removable memory on a pre-determined path
2. Setup a bare repository in the removable memory
3. Add the repository in the removable memory as a remote

Throughout this article, it's assumed that the repository will be mounted in /media/repositories.

Starting from scratch

Let's assume a brand new project is being started, titled Foo. The people working on project Foo will use a removable memory device will be used as a central repository for the whole project. For this reason, let's create a fresh repository in a USB stick.

To start off, instead of relying on a path name generated automatically by the OS, the USB stick will be mounted on a custom path. Let's assume the USB pen drive is located at /dev/sdb1, and the intended mount point is /media /repositories. Another important aspect is that the USB pen drive needs to be mounted with proper permissions, so that it doesn't cause problems within the repository. One way to avoid these issues is to mount the USB drive with your own user ID and group ID, which can be achieved through the following commands:

```
$ sudo mkdir /media/repositories
$ my_uid=`id -u`
$ my_gid=`id -g`
$ mount -o "uid=$my_uid,gid=$my_gid" /dev/sdb1 /media/repositories
```

Having mounted the USB drive on the desired path, let's create a bare repository.

```
$ cd /media/repositories
$ mkdir /media/repositories/foo
$ git init --bare /media/repositories/foo
```

```
Initialized empty Git repository in /media/repositories/foo
```

With this step, a repository has been created in the USB memory drive. Now, all that's left is to mount the USB pen on any computer on a specific path, clone the repository, and work away.

```
$ git clone /media/repositories/foo
Cloning into 'foo'...
warning: You appear to have cloned an empty repository.
done.
```

Done.

In case the mount point changes to another path (in some cases, auto-mounting does that), the repository's URL can be set through the following command:

```
$ git remote set-url origin file://<path to new mount point>
```

Pushing local repository to a USB stick

Let's assume you've been working on a project, and you already have a working git repository that you've been working on your desktop. Now, you've decided that want to keep track and update several concurrent versions of that repository without having to use a network connection. One possible solution is to start a Git repository on a USB key, and use that USB key as a centralized repository where everyone can push their contributions and update their local version.

To start a Git repository on a USB stick, first let's mount the USB stick on a selected path. This is achieved through the process described in the previous section.

```
$ sudo mkdir /media/repositories
$ my_uid=`id -u`
$ my_gid=`id -g`
$ mount -o "uid=$my_uid,gid=$my_gid" /dev/sdb1 /media/repositories
```

Let's assume that the project tree is located in ~/devel/foo. To start a repository on /media/repositories/foo, run the following command:

```
git clone --bare ~/devel/foo foo
```

That's it.

Now, you can clone the Git repository stored in the USB drive and continue working on your project.

```
$ git clone /media/repositories/foo/
Cloning into 'foo'...
done.
```

If you wish to add to your local repository the newly created USB repository as a remote repository,

```
$ git remote add usb file:///media/repositories/foo
```

To establish the master branch of the USB repository as the upstream branch of your local master branch, the contents of the newly added remote branch must be fetched and the upstream branch must be specified. This step is performed by applying the following commands:

```
$ git fetch usb
From file:///media/repositories/foo
 * [new branch]      master     -> usb/master
$ git branch --set-upstream-to=usb/master
Branch master set up to track remote branch master from usb.
```

Git component programs

Git has many component program, the following is an unabridged list of the them as of `git version 1.5.2.5`.

git
git-add
git-add--interactive
git-am
git-annotate
git-apply
git-applymbox
git-applypatch
git-archive
git-bisect
git-blame
git-branch
git-bundle
git-cat-file
git-check-attr
git-checkout
git-checkout-index
git-check-ref-format
git-cherry
git-cherry-pick
git-clean
git-clone
git-commit
git-commit-tree
git-hash-object
git-http-fetch
git-http-push
git-imap-send
git-index-pack
git-init
git-init-db
git-instaweb
git-local-fetch
git-log
git-lost-found
git-ls-files
git-ls-remote
git-ls-tree
git-mailinfo
git-mailsplit
git-merge
git-merge-base
git-merge-file
git-merge-index
git-merge-octopus
git-merge-one-file
git-merge-ours
git-merge-recursive
git-rebase
git-receive-pack
git-reflog
git-relink
git-remote
git-repack
git-repo-config
git-request-pull
git-rerere
git-reset
git-revert
git-rev-list
git-rev-parse
git-rm
git-runstatus
git-send-pack
git-shell
git-shortlog
git-show
git-show-branch
git-show-index
git-show-ref
git-sh-setup
git-ssh-fetch

git-config
git-convert-objects
git-count-objects
git-daemon
git-describe
git-diff
git-diff-files
git-diff-index
git-diff-tree
git-fast-import
git-fetch
git-fetch-pack
git-fetch--tool
git-fmt-merge-msg
git-for-each-ref
git-format-patch
git-fsck
git-fsck-objects
git-gc
git-get-tar-commit-id
git-grep
git-merge-resolve
git-merge-stupid
git-merge-subtree
git-mergetool
git-merge-tree
git-mktag
git-mktree
git-mv
git-name-rev
git-pack-objects
git-pack-redundant
git-pack-refs
git-parse-remote
git-patch-id
git-peek-remote
git-prune
git-prune-packed
git-pull
git-push
git-quiltimport
git-read-tree
git-ssh-pull
git-ssh-push
git-ssh-upload
git-status
git-stripspace
git-symbolic-ref
git-tag
git-tar-tree
git-unpack-file
git-unpack-objects
git-update-index
git-update-ref
git-update-server-info
git-upload-archive
git-upload-pack
git-var
git-verify-pack
git-verify-tag
git-whatchanged
git-write-tree

Getting help

IRC

- #git on irc.freenode.net (irc://irc.freenode.net/git) (webchat (http://webchat.freenode.net/?channels=git))

Forums & mailing lists

- Mailing list: git@vger.kernel.org

Web pages

- FAQs
  - FAQ from kernel.org (http://git.wiki.kernel.org/index.php/GitFaq)
  - A FAQ with fast answers (http://www.sourcemage.org/Git_Guide)

- Tutorials
  - Official Git documentation (http://www.kernel.org/pub/software/scm/git/docs/index.html)
  - ProGit (http://progit.org/)
  - A very nice and concise tutorial (http://www-cs-students.stanford.edu/~blynn/gitmagic/) that also explains advanced features like branches, rebasing etc.
  - Git Community Book (http://book.git-scm.com/index.html)
- For users of other VCS
  - Crash course for SVN users (http://git-scm.org/course/svn.html)
- Misc
  - Seminars and presentations (http://git.wiki.kernel.org/index.php/GitLinks#Seminars_and_presentations) about Git
  - Lots of Git-related links (http://git.wiki.kernel.org/index.php/GitLinks)
  - The Git Parable (http://tom.preston-werner.com/2009/05/19/the-git-parable.html)
