Git

David Parker
CSCI 5828 - Presentation

Outline

- What is Git?
- History
- Features/Benefits
- Understanding Git
- How to's
- Git Internals
- Other Useful Related Tools
- What projects use Git?
- Other Open Source VCS' and DVCS'
- References

What is Git? (I)

1. Version Control System (VCS)
   - Keep different versions of files over time
   - Keep history of who changed what in a file
   - Generally maintained in a database or repository
   - Commonly centralized, though distributed VCSs have been in growing usage

2. Open Source
   - The source code is available so that anyone can see it, and modify it as needed

3. Fast

What is Git? (II)

4. Distributed VCS (DVCS)
   - No centralized server required
   - Every client mirrors the entire repository, not just the latest snapshots
   - Able to have several remote repositories
   - Allows for different workflows not able to be used with centralized VCSs
5. Designed to handle very large projects with speed and efficiency, as well as small repositories
6. Distributed Source Control Management tool (DSCM)

History (I)

Originally written by Linus Torvalds, creator of Linux

Maintained by Junio Hamano

The name:

- Quoting Linus: "I'm an egotistical bastard, and I name all my projects after myself."

History (II)

Linux originally used BitKeeper, which was proprietary, but had a falling out in 2005

- Linus wanted a distributed system similar to BitKeeper, but none of the free systems did what he wanted
- Linus needed something that was fast
- Linus was merging as many as 250 patches at a time, which at 30 seconds, takes nearly 2 hours

Linus thinks CVS is terrible: "I hate it with a passion"

Similarly, if Subversion is "CVS done right," then it is also bad: "There is no way to do CVS right"

Features / Benefits (I)

Cheap Local Branching

Git's most compelling feature that makes it stand apart from nearly every other SCM out there is its branching model. Git will allow you to have multiple branches that can be entirely independent of each other and the creation, merging, and deletion of those lines of development takes seconds. When you push a remote repository, you do not have to push all of your branches.

This means you can do things like:

- Create a branch to test out an idea, commit a few times, switch back to where you branched from, apply a patch, and switch back to where experimenting and merge it in.
- Have a branch that always contains only what goes into production, another that you merge work into for testing and several smaller ones for day to day work.
- Create new branches for each new feature you're working on, then delete each branch when that feature gets merged into your main line.
- Create a branch to experiment in, realize it's not going to work and just delete it, with nobody else seeing it (even if you've pushed other branches)

Features / Benefits (II)

<u>Everything is Local</u>

There is very little outside of 'fetch', 'pull' and 'push' that communicates in any way with anything other than your hard disk.

This make most operations much faster than most SCMs.

This also allows you to work on stuff offline.

You can work on a train or a plane!

You can do a fetch before going offline and later do comparisons, merges, and logs of that data but not yet in local branches.

This means it is super easy to have copies of everyone's branches that are working with you in your Git repository without messing up your own stuff.

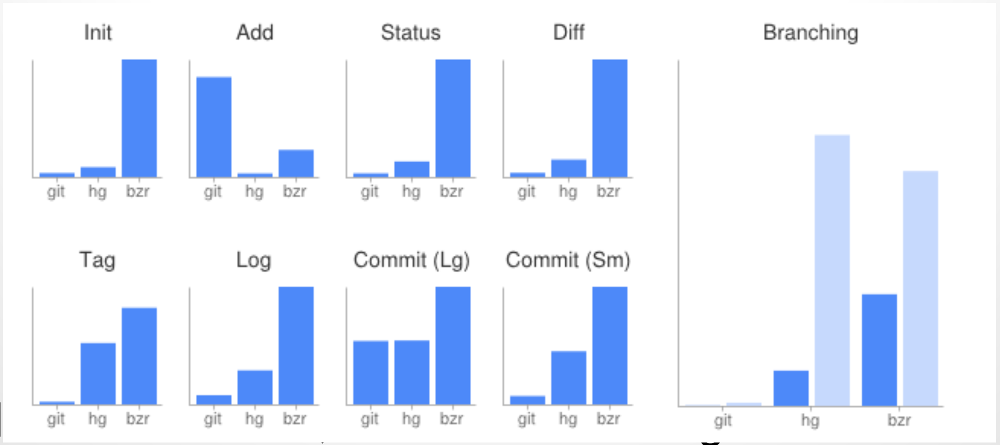
Features / Benefits (III)

Git is Fast

The fact that all operations are performed locally makes Git incredibly fast compared to other SCMs like Subversion and Perforce, both of which require network access for certain operations.

Another reason Git is fast is due to the fact that the primary developers made this a design goal of the application.

Here is a comparison of Git with Mercurial and Bazaar:



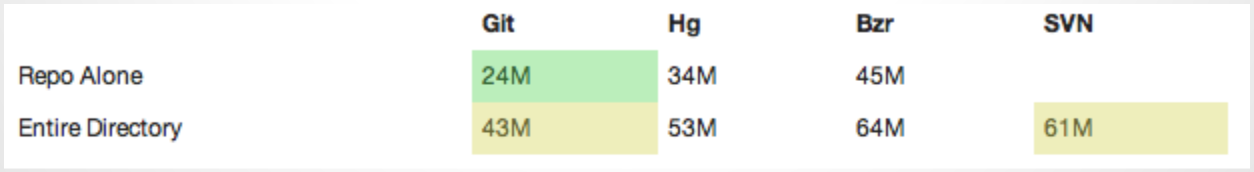
Note that the 'ad es, something most people don't do daily.

Features / Benefits (IV)

Git is Small

Git is really good at conserving disk space.

Here's a comparison using the Django project:



	Git	Hg	Bzr	SVN
Repo Alone	24M	34M	45M	
Entire Directory	43M	53M	64M	61M

Features / Benefits (V)

The Staging Area

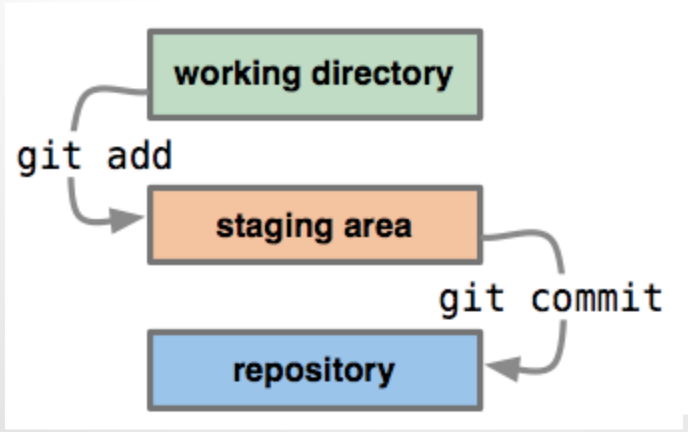
Git has something called the "staging area" or "index".

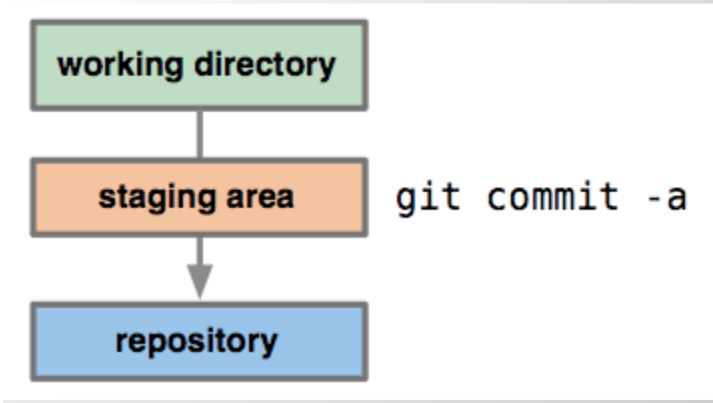
This is an intermediate area that you use to setup what you want your commit to look like before you commit.

You can easily stage some files as they're finished and commit just those files and not all the modified files.

You can also stage only portions of a modified file.

You can also skip the staging area if you don't need it.





Features / Benefits (VI)

Distributed

One of the best features of any Distributed SCM is that they are distributed by their nature .

This means that you "clone" an entire repository rather than "checkout" the current tip of some source code.

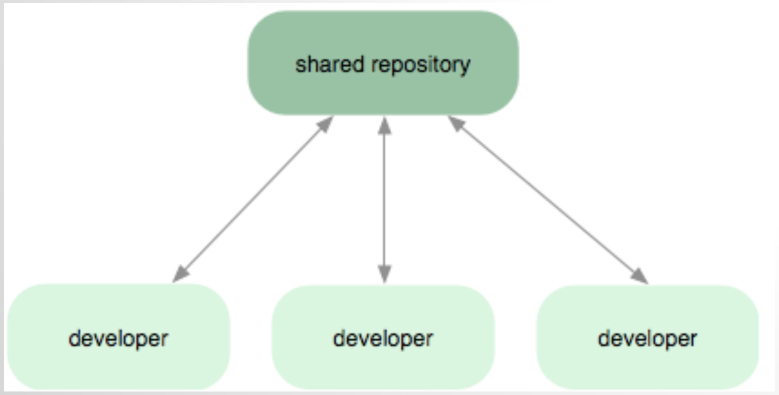
Even if you use a centralized workflow, every user essentially has a full backup of the main server, which means there is no single point of failure with Git.

Features / Benefits (VII)

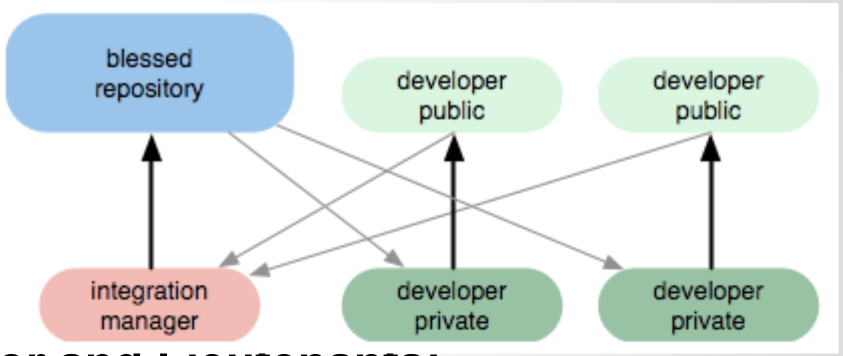
Any Workflow

Due to Git's distributed nature and branching system, you can implement any workflow you want.

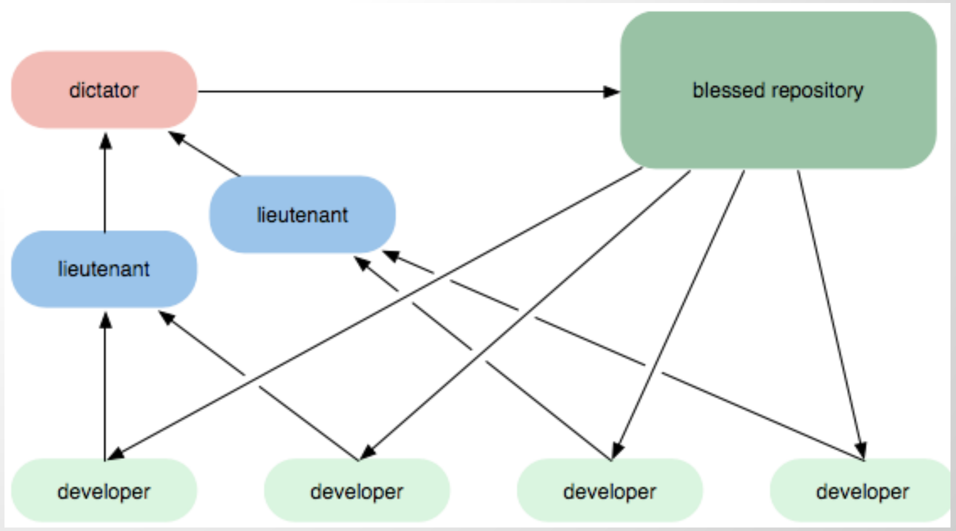
Subversion-Style



Integration Manager



Dictator and Lieutenants:

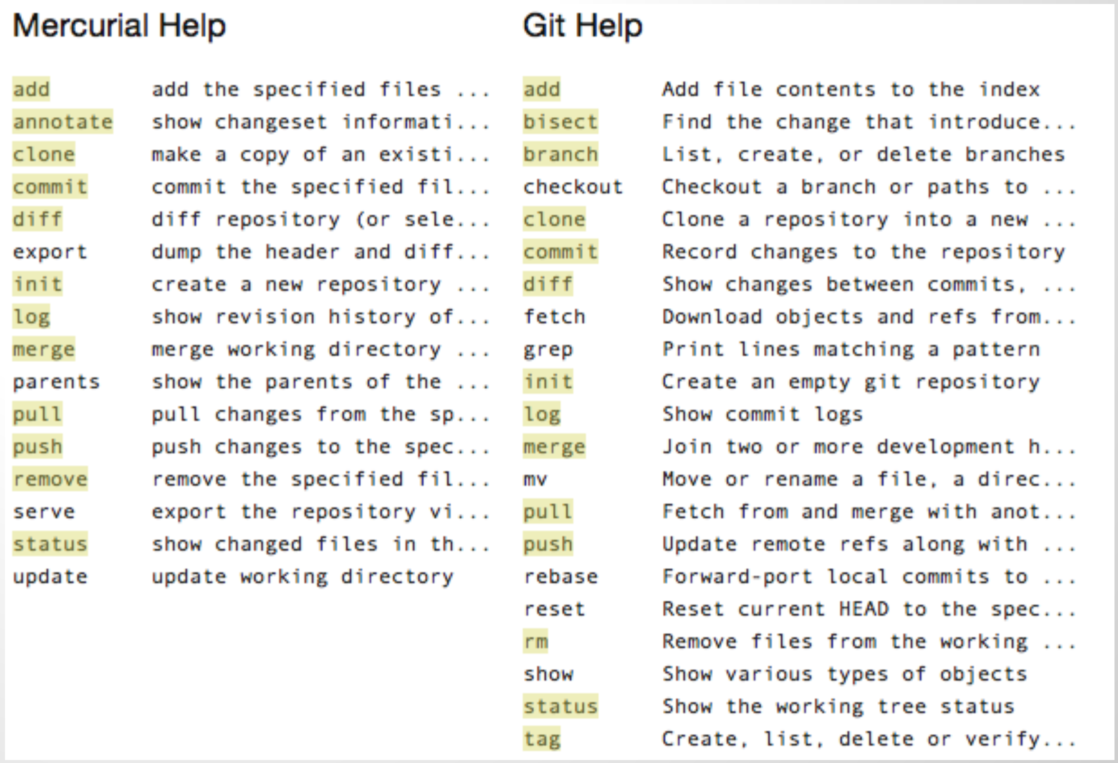


Features / Benefits (VIII)

Easy to Learn

In Git's early life, it wasn't a true SCM, but a bunch of tools that allowed someone to do versioned filesystems in a distributed manner.

Here's a quick difference between Git and Mercurial help (highlighted ones are near identical):



Mercurial Help

add	add the specified files ...
annotate	show changeset informati...
clone	make a copy of an existi...
commit	commit the specified fil...
diff	diff repository (or sele...
export	dump the header and diff...
init	create a new repository ...
log	show revision history of...
merge	merge working directory ...
parents	show the parents of the ...
pull	pull changes from the sp...
push	push changes to the spec...
remove	remove the specified fil...
serve	export the repository vi...
status	show changed files in th...
update	update working directory

Git Help

add	Add file contents to the index
bisect	Find the change that introduce...
branch	List, create, or delete branches
checkout	Checkout a branch or paths to ...
clone	Clone a repository into a new ...
commit	Record changes to the repository
diff	Show changes between commits, ...
fetch	Download objects and refs from...
grep	Print lines matching a pattern
init	Create an empty git repository
log	Show commit logs
merge	Join two or more development h...
mv	Move or rename a file, a direc...
pull	Fetch from and merge with anot...
push	Update remote refs along with ...
rebase	Forward-port local commits to ...
reset	Reset current HEAD to the spec...
rm	Remove files from the working ...
show	Show various types of objects
status	Show the working tree status
tag	Create, list, delete or verify...

Understanding Git

Unlike other VCSs that typically think about data as changes, Git thinks of changes as snapshots

- Snapshot of a mini filesystem
- To be efficient, if a file hasn't changed, then Git links to the previous identical file it has already stored

Integrity

Everything in Git is check-summed before it is stored and then referred to by that checksum.

- Impossible to change contents of file or directory without Git knowing about it
- Can't lose information in transit or get file corruption without Git knowing about it

Checksumming via SHA-1 hash

- 40-character string composed of hexadecimal characters (0-9 and a-f)

Three States

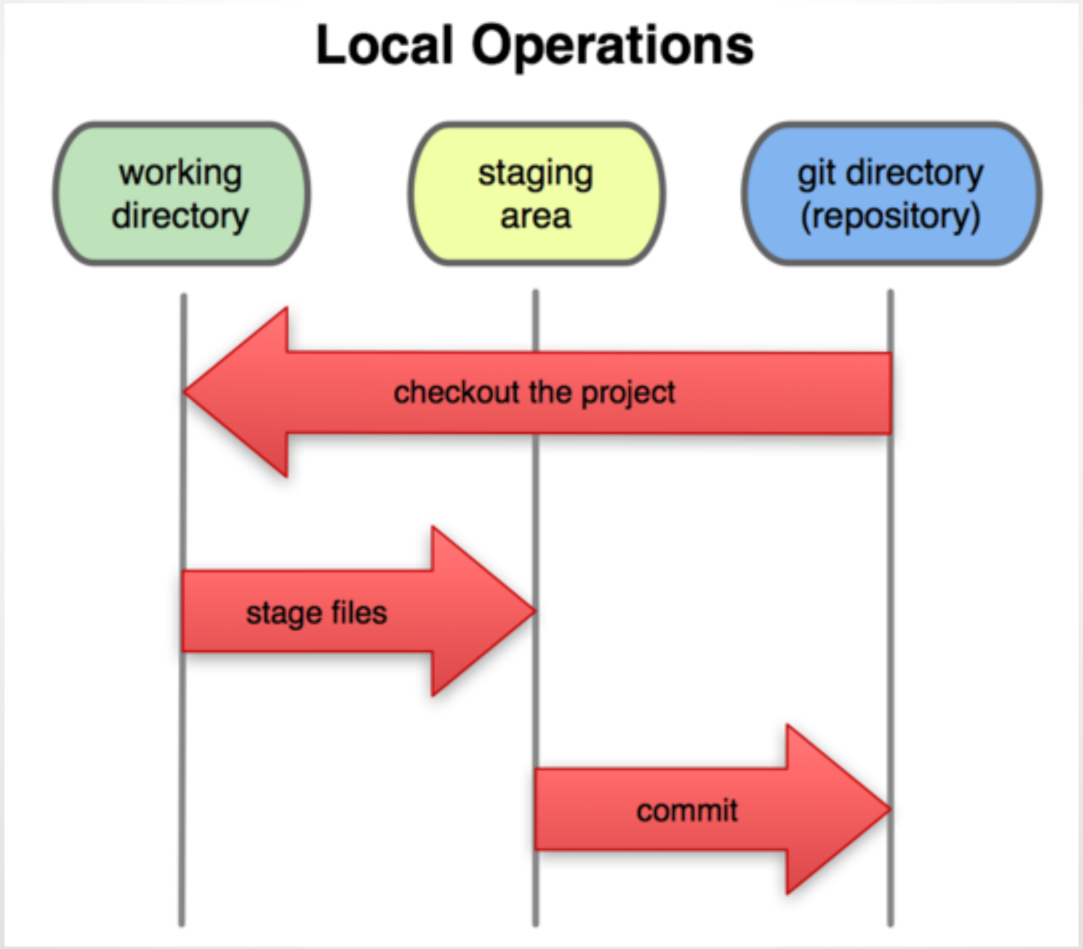
Three main states that a file can reside in:

1. Committed
   - Data safely stored in local database
2. Modified
   - Changed a file but not yet committed it to database
3. Staged
   - Marked file in current version to go into next commit snapshot

Three Main Sections

1. Git Directory (repository)
    - Where Git stores metadata and object database for the project.
    - What is copied when you clone a repository.
2. Working Directory
    - Single checkout of one version of the project.
3. Staging Area
    - A file that stores information about what will go into your next commit.
    - Also referred to as the index.

Three Main Sections (Picture)



How to's

Next, we'll take a look at doing a ton of different stuff in Git.

- Install/Set up/Help
- Creating a repo
- Cloning a repo
- Status of files
- Adding files
- Committing files
- Staged Files
- Ignoring Files
- (Re)moving files
- Logging
- Undoing Changes
- Working w/remote
- Tagging
- Branching
- Merging
- Rebasing
- Git on the Server
- And more...

How to: Install Git (I)

Git is available on Linux, OSX, and Windows

Install from Source:

- http://git-scm.com/download
  - Follow instructions (compile and install)

On Linux via package managers:

- (yum|apt-get) install git-core

How to: Install Git (II)

On Mac:

- via Graphical Installer:
    - http://code.google.com/p/git-osx-installer
- via MacPorts:
    - sudo port install git-core +svn +doc +bash_completion +gitweb

Windows:

- via msysgit
    - http://code.google.com/p/msysgit

How to: Setting up Git

Modify configuration file:

- ~/.gitconfig OR .gitconfig in $HOME on Windows

Identity:

- git config --global user.name "David Parker"
- git config --global user.email davidwparker@gmail.com

Editor:

- git config --global core.editor emacs

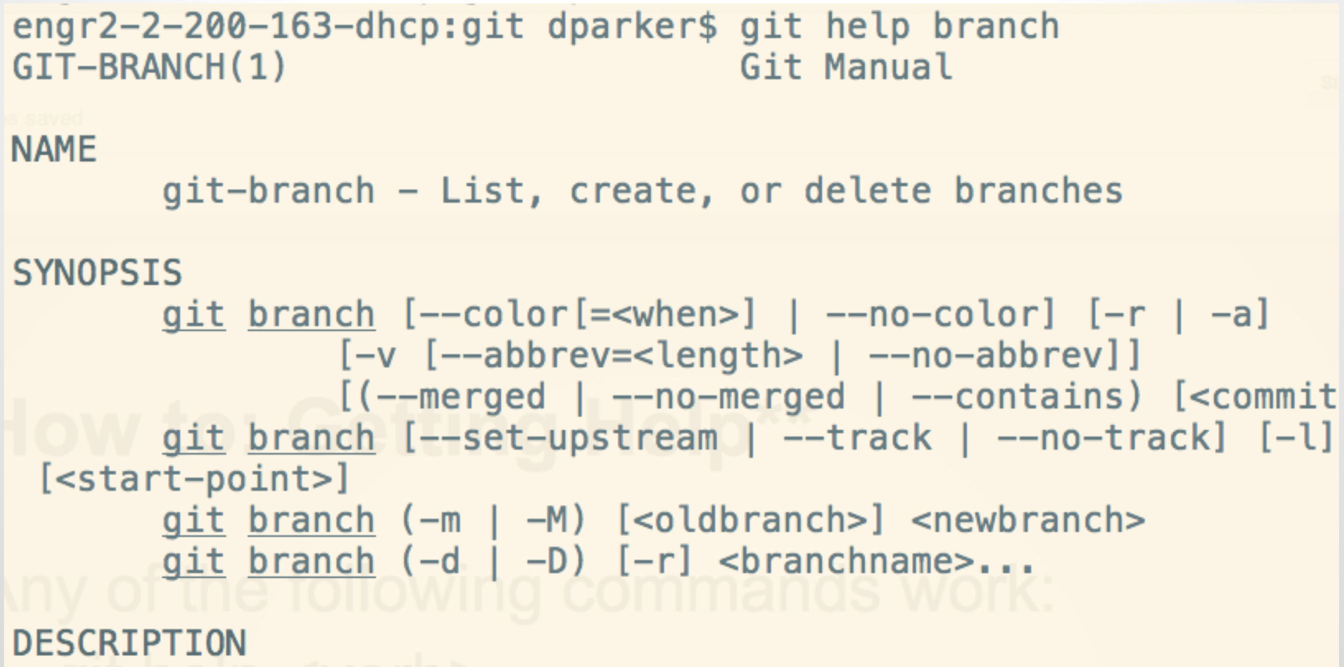
Check Settings

- git config --list

How to: Getting Help

Any of the following commands work:

- git help <verb>
- git <verb> --help
- man git-<verb>

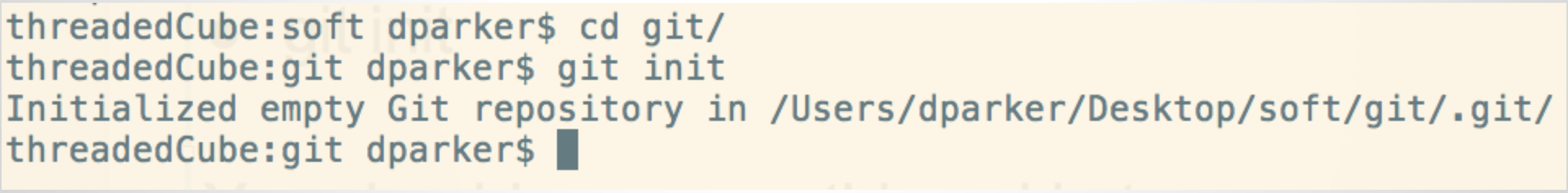


```
engr2-2-200-163-dhcp:git dparker$ git help branch
GIT-BRANCH(1)                     Git Manual

NAME
       git-branch - List, create, or delete branches

SYNOPSIS
       git branch [--color[=<when>] | --no-color] [-r | -a]
               [-v [--abbrev=<length> | --no-abbrev]]
               [(--merged | --no-merged | --contains) [<commit
       git branch [--set-upstream | --track | --no-track] [-l]
 [<start-point>]
       git branch (-m | -M) [<oldbranch>] <newbranch>
       git branch (-d | -D) [-r] <branchname>...

DESCRIPTION
```

How to: Create a Repository

In order to create a git repository, cd into the directory you would like to create the repository and type the command:

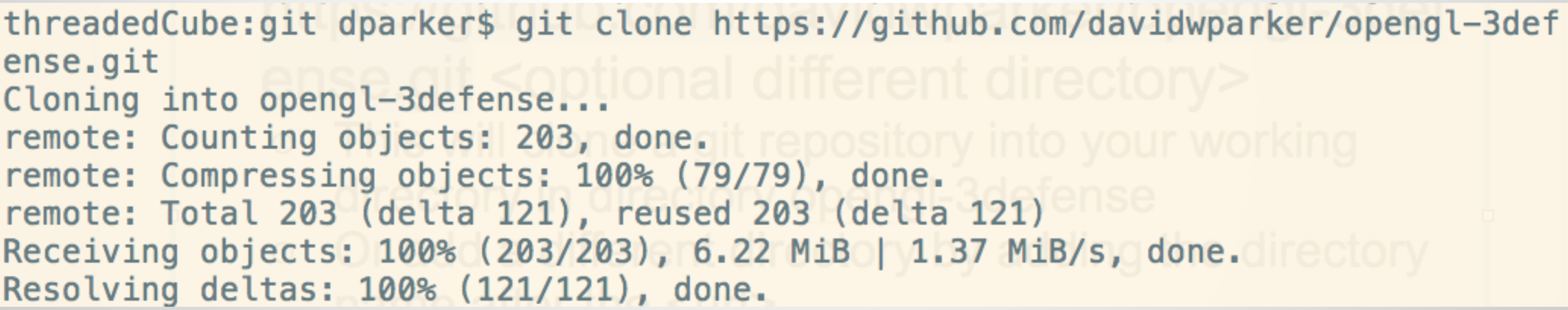
- git init



```
threadedCube:soft dparker$ cd git/
threadedCube:git dparker$ git init
Initialized empty Git repository in /Users/dparker/Desktop/soft/git/.git/
threadedCube:git dparker$ 
```

How to: Clone a Repository

- git clone <url> <optional different directory>
- git clone https://github.com/davidwparker/opengl-3defense.git <optional different directory>
  - This will clone a git repository into your working directory in directory opengl-3defense
  - Or add a different directory by adding the directory name after the <url>

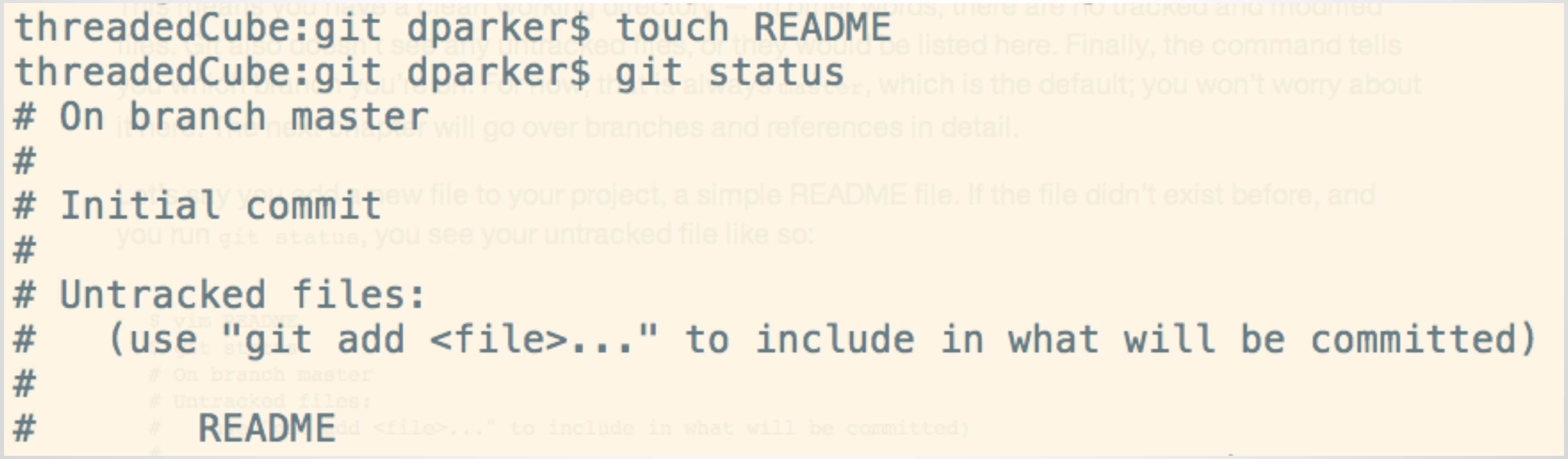


```
threadedCube:git dparker$ git clone https://github.com/davidwparker/opengl-3def
ense.git
Cloning into opengl-3defense...
remote: Counting objects: 203, done.
remote: Compressing objects: 100% (79/79), done.
remote: Total 203 (delta 121), reused 203 (delta 121)
Receiving objects: 100% (203/203), 6.22 MiB | 1.37 MiB/s, done.
Resolving deltas: 100% (121/121), done.
```

How to: Checking the Status of Files

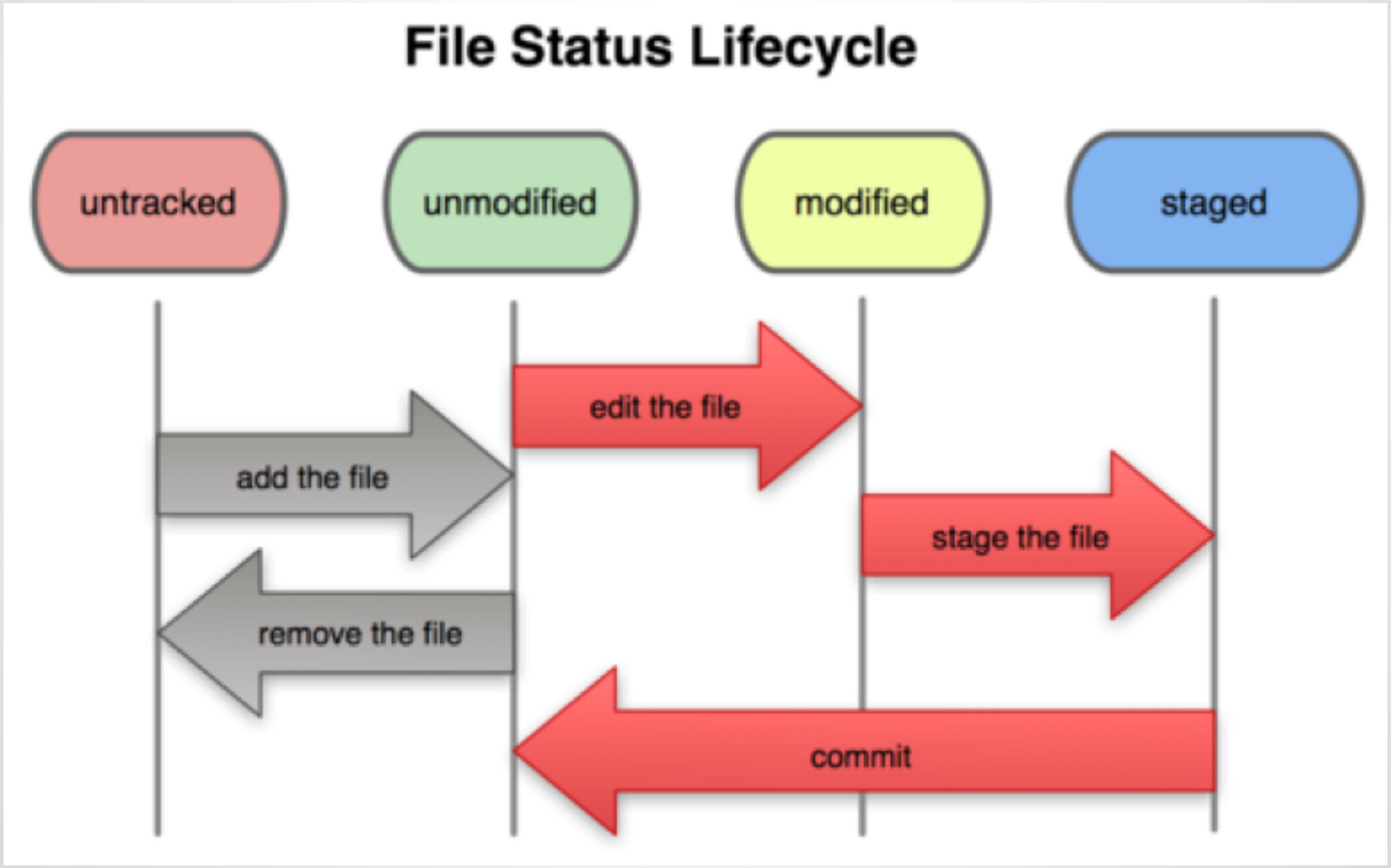
You can check the status of files in your Git repository very easily:

- git status



```
threadedCube:git dparker$ touch README
threadedCube:git dparker$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       README
```

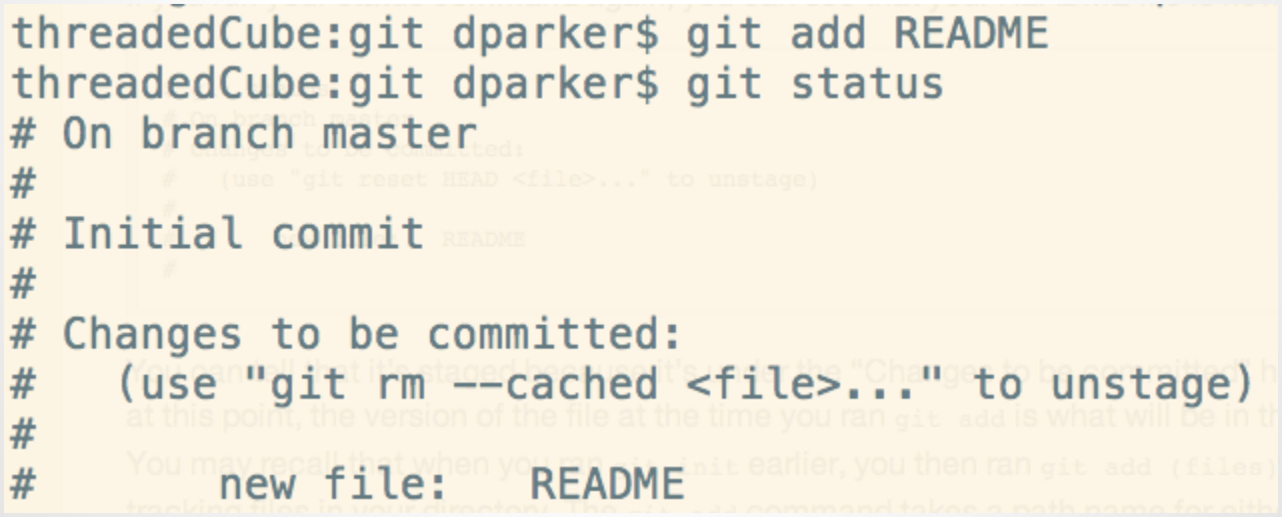
Lifecycle (status) of Files



How to: Adding Files

Add a file easily:

- git add <filename> OR git add *.<type>



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

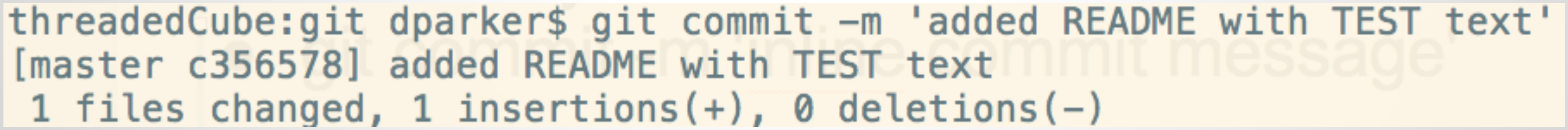
How to: Commit New Files

Easily commit new files:

- git commit
  - Launches editor of choice for git commit message

Alternatively:

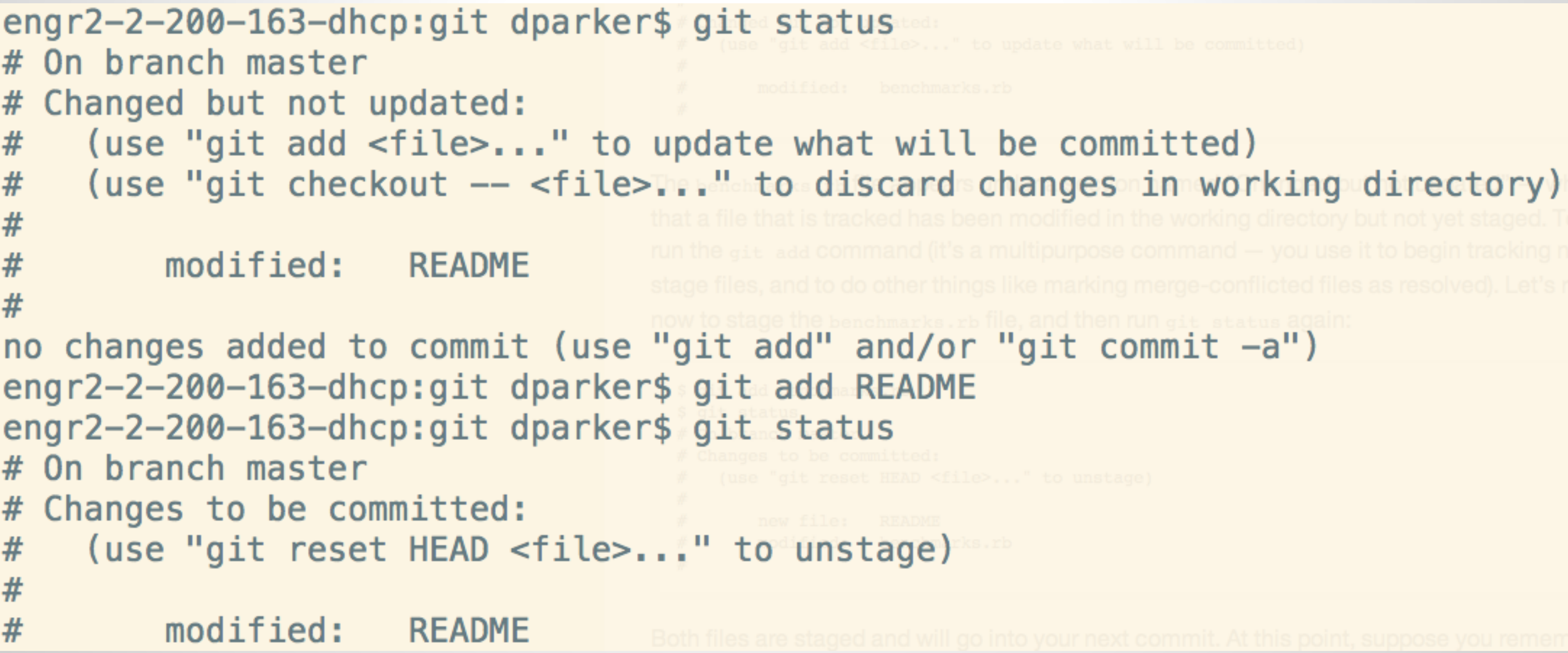
- git commit -m 'inline commit message'



```
threadedCube:git dparker$ git commit -m 'added README with TEST text'
[master c356578] added README with TEST text
 1 files changed, 1 insertions(+), 0 deletions(-)
```

How to: Staged Files

A staged file is a file that has previously been committed and has since been changed.



```
engr2-2-200-163-dhcp:git dparker$ 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:   README
#
no changes added to commit (use "git add" and/or "git commit -a")
engr2-2-200-163-dhcp:git dparker$ git add README
engr2-2-200-163-dhcp:git dparker$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README
```

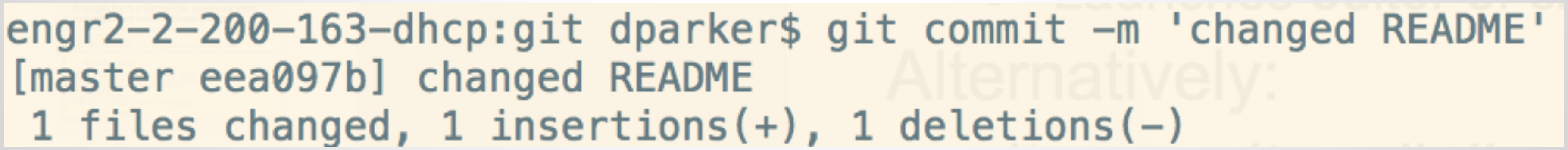
How to: Commit Staged Files

Committing staged files is the same as committing new files:

- git commit
  - Launches editor of choice for git commit message

Alternatively:

- git commit -m 'inline commit message'

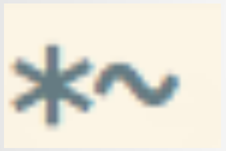


```
engr2-2-200-163-dhcp:git dparker$ git commit -m 'changed README'
[master eea097b] changed README
 1 files changed, 1 insertions(+), 1 deletions(-)
```

How to: Ignore Files (I)

You can ignore files and filetypes with .gitignore

- touch .gitignore
- emacs .gitignore



```
*~
```

- will ignore temporary files that are marked with a ~, which is common with editors such as Emacs.
- You can also add directories

How to: Ignore Files (II)

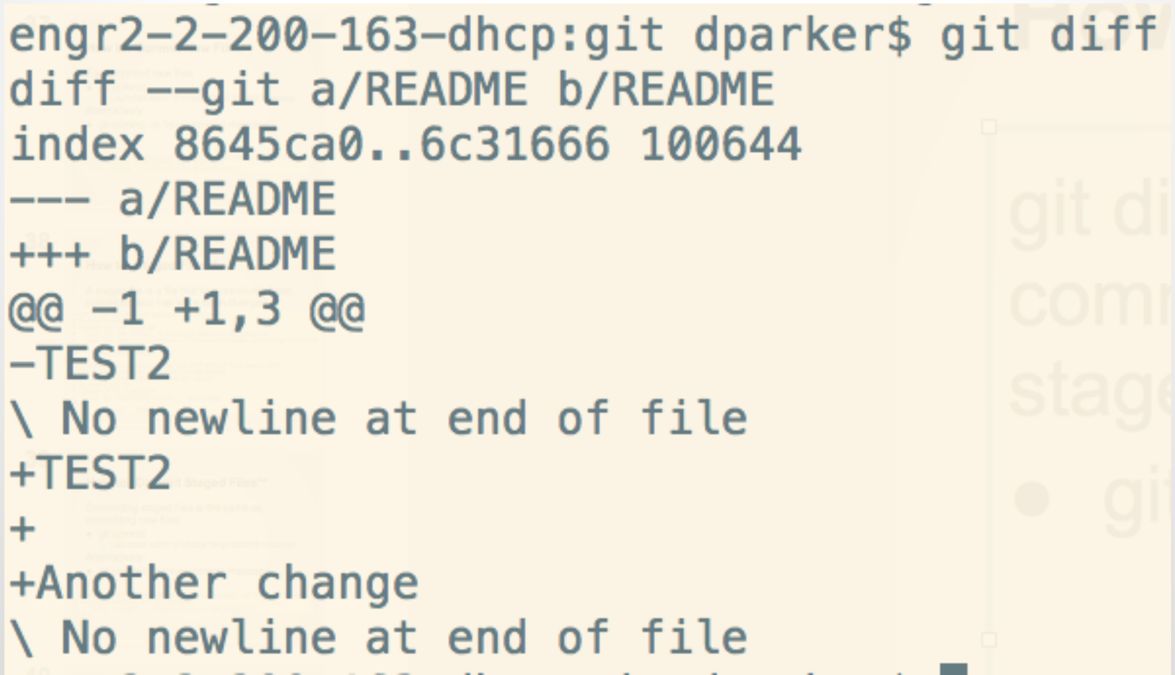
The rules for the patterns of what can be in the .gitignore file:

- Blank lines or lines starting with # are ignored
- Standard glob patterns work
- You can end patterns with a forward slash (/) to specify a directory
- You can negate a pattern by starting with an exclamation point (!)

How to: Diff (unstaged changes)

git diff is used for multiple reasons, but the most common is to see what has changed but not yet staged.

- git diff

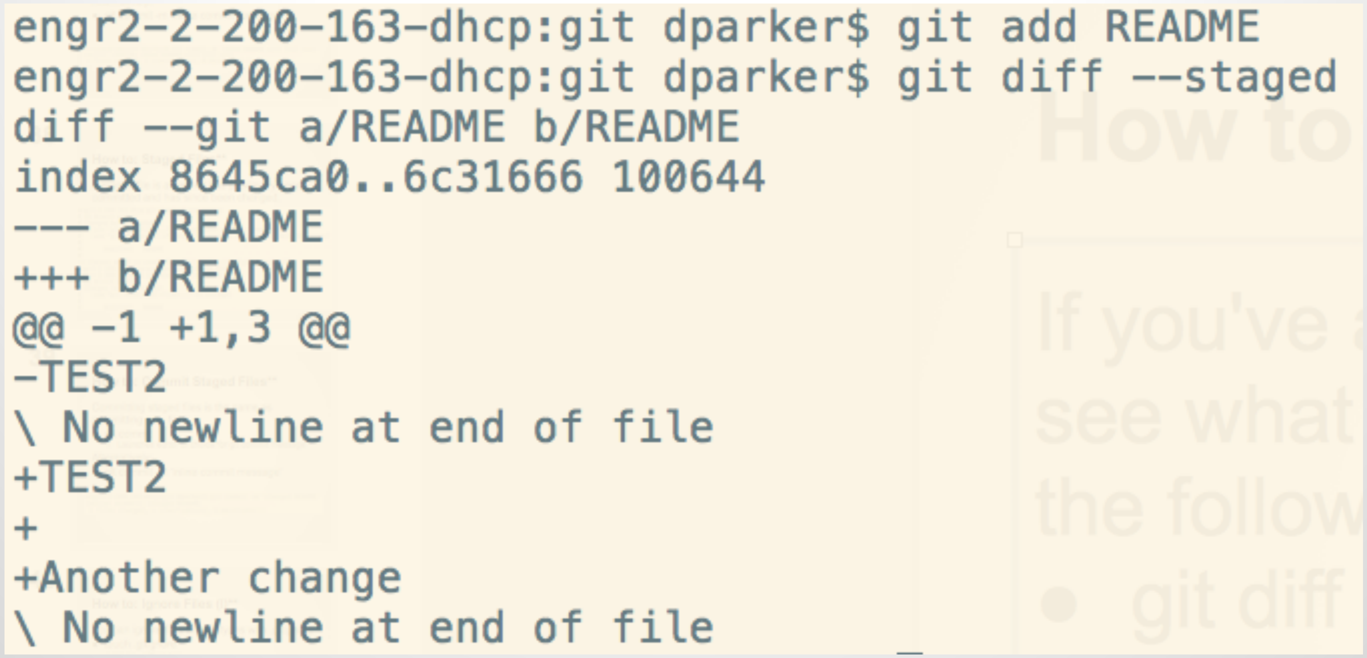


```
engr2-2-200-163-dhcp:git dparker$ git diff
diff --git a/README b/README
index 8645ca0..6c31666 100644
--- a/README
+++ b/README
@@ -1 +1,3 @@
-TEST2
\ No newline at end of file
+TEST2
+
+Another change
\ No newline at end of file
```

How to: Diff (staged changes)

If you've added files to staging, and you'd like to see what the diff of those changes, simply use the following:

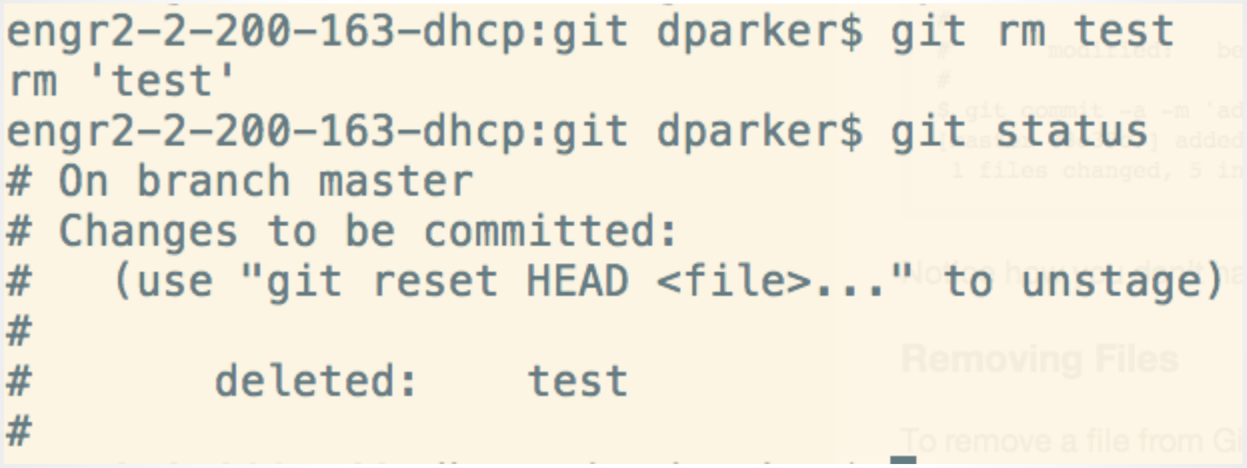
- git diff --staged



```
engr2-2-200-163-dhcp:git dparker$ git add README
engr2-2-200-163-dhcp:git dparker$ git diff --staged
diff --git a/README b/README
index 8645ca0..6c31666 100644
--- a/README
+++ b/README
@@ -1 +1,3 @@
-TEST2
\ No newline at end of file
+TEST2
+
+Another change
\ No newline at end of file
```

How to: Remove Files

- git rm <file>



```
engr2-2-200-163-dhcp:git dparker$ git rm test
rm 'test'
engr2-2-200-163-dhcp:git dparker$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    test
#
```

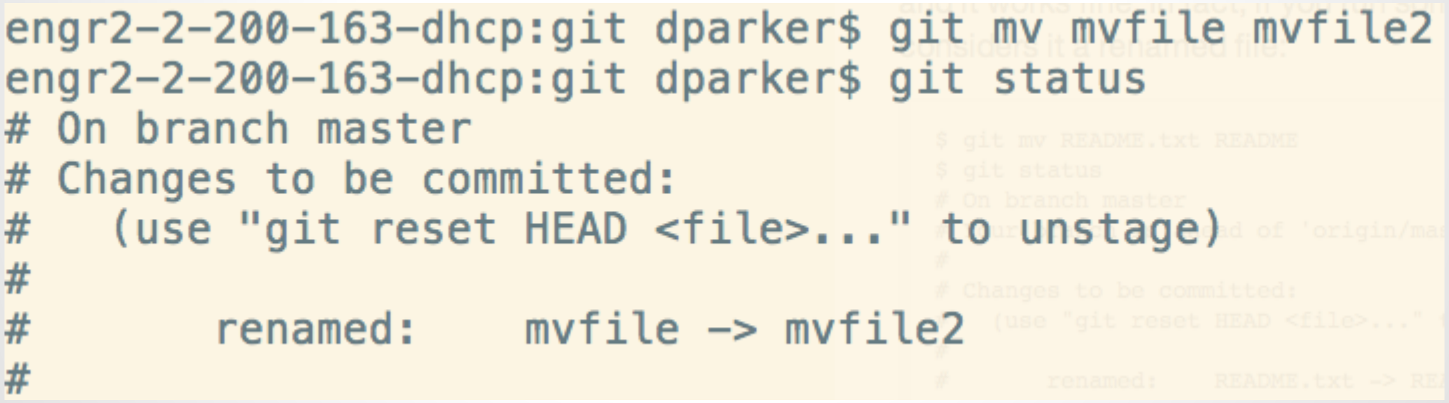
Now the removal of the file is ready to be committed.

Note the file is removed from the file system as well (it can be kept with the --cached flag)

How to: Move Files

Git technically doesn't keep track of file movement, but does offer a way to move files.

- git mv <file> <newfile>



```
engr2-2-200-163-dhcp:git dparker$ git mv mvfile mvfile2
engr2-2-200-163-dhcp:git dparker$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    mvfile -> mvfile2
#
```

This is the same as running the commands: git rm --cached orig; mv orig new; git add new

How to: Log (I)

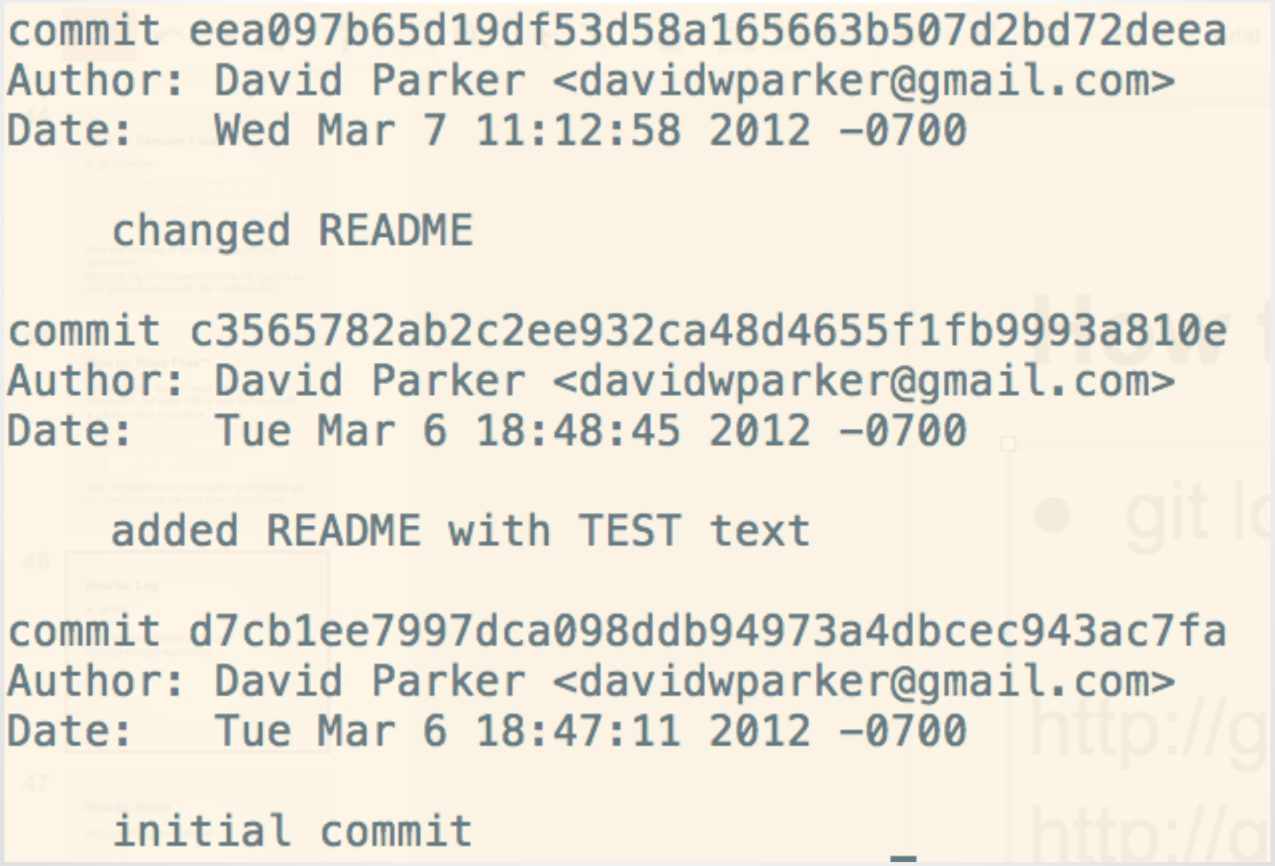
By default, git log lists commits in a repository in reverse chronological order.

It lists commit with SHA-1 checksum, author's name and email, date written, and commit message.

See the next slide for an example.

How to: Log (II)

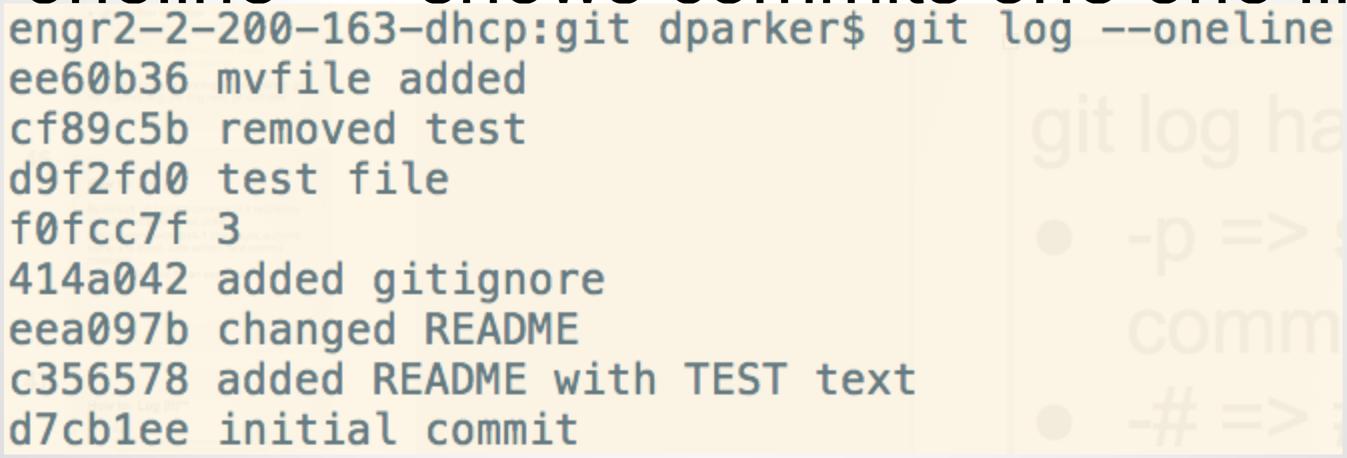
- git log



```
commit eea097b65d19df53d58a165663b507d2bd72deea
Author: David Parker <davidwparker@gmail.com>
Date:   Wed Mar 7 11:12:58 2012 -0700

    changed README

commit c3565782ab2c2ee932ca48d4655f1fb9993a810e
Author: David Parker <davidwparker@gmail.com>
Date:   Tue Mar 6 18:48:45 2012 -0700

    added README with TEST text

commit d7cb1ee7997dca098ddb94973a4dbcec943ac7fa
Author: David Parker <davidwparker@gmail.com>
Date:   Tue Mar 6 18:47:11 2012 -0700

    initial commit
```

How to: Log (III) - Options

- --pretty=format:"YOUR FORMAT"
  - Very powerful way to specify own log output format
- -p => shows diff introduced in each commit
- -# => shows only the last # commits.
- --oneline => shows commits one one line



```
engr2-2-200-163-dhcp:git dparker$ git log --oneline
ee60b36 mvfile added
cf89c5b removed test
d9f2fd0 test file
f0fcc7f 3
414a042 added gitignore
eea097b changed README
c356578 added README with TEST text
d7cb1ee initial commit
```

- many, many more!

How to: Undoing Changes

Changing last commit:

- git commit --amend

Unstaging a staged file:

- git reset HEAD <filename>

Unmodify a modified file:

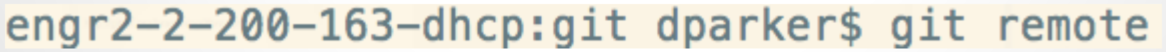
- git checkout -- <filename>
    - Warning: this overwrites the file, so you will lose any changes that you made. You sparingly.

How to: Working with Remote (I)

Remote repositories are versions of the project on the Internet or network.

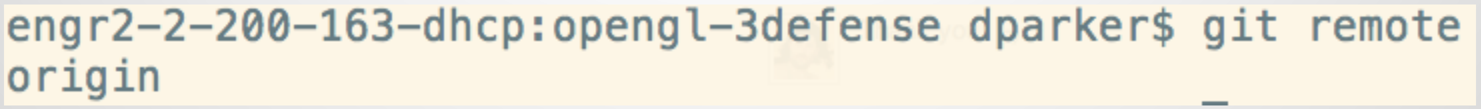
If this is a locally created git repository, then you won't see any git remotes:

- git remote



```
engr2-2-200-163-dhcp:git dparker$ git remote
```

If it isn't local, you will see origin:

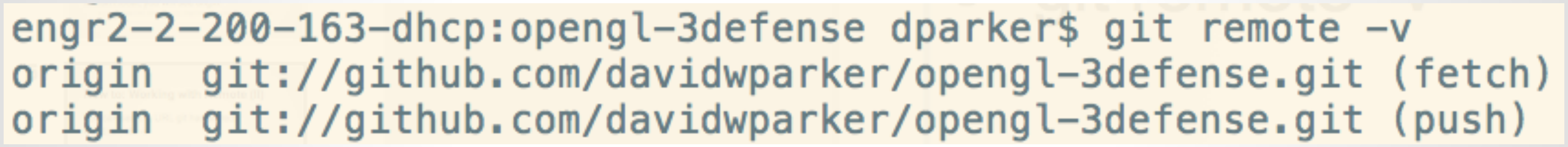


```
engr2-2-200-163-dhcp:opengl-3defense dparker$ git remote
origin
```

How to: Working with Remote (II)

You can also see the URL git has stored:

- git remote -v

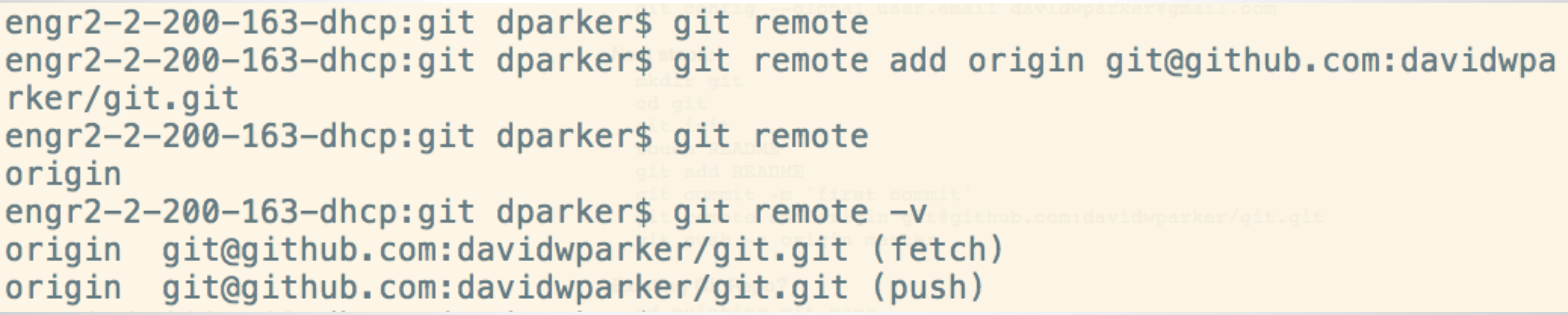


```
engr2-2-200-163-dhcp:opengl-3defense dparker$ git remote -v
origin  git://github.com/davidwparker/opengl-3defense.git (fetch)
origin  git://github.com/davidwparker/opengl-3defense.git (push)
```

How to: Adding Remote

You can easily add a remote repository as well:

- git remote add <shortname> <url>
- git remote add origin git@github.com:davidwparker/git.git

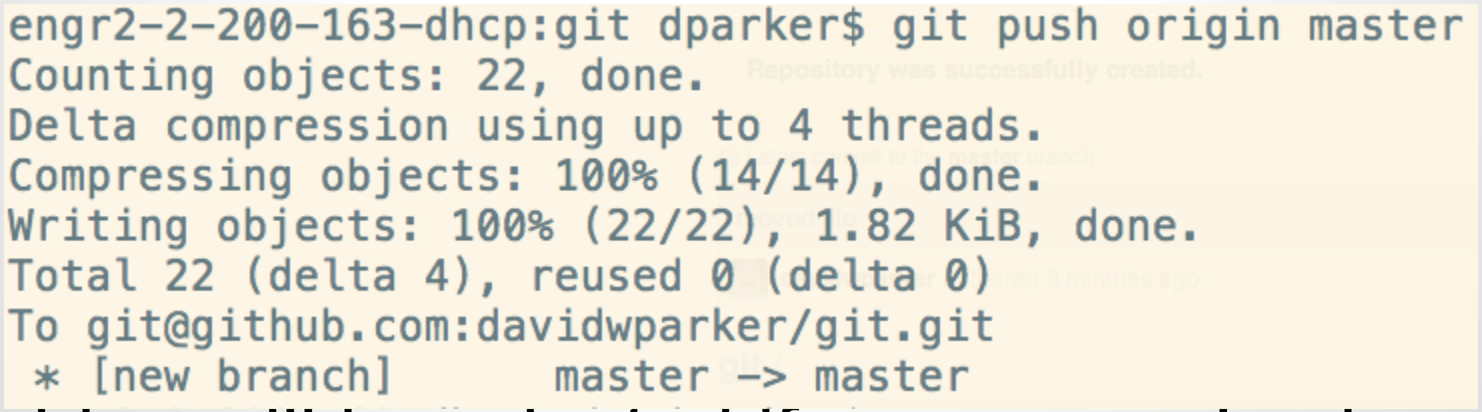


```
engr2-2-200-163-dhcp:git dparker$ git remote
engr2-2-200-163-dhcp:git dparker$ git remote add origin git@github.com:davidwpa
rker/git.git
engr2-2-200-163-dhcp:git dparker$ git remote
origin
engr2-2-200-163-dhcp:git dparker$ git remote -v
origin	git@github.com:davidwparker/git.git (fetch)
origin	git@github.com:davidwparker/git.git (push)
```

How to: Push Remote

Pushing to remote allows us to push our repository to the remote repository:

- git push <remote name> <branch name>
- git push origin master



```
engr2-2-200-163-dhcp:git dparker$ git push origin master
Counting objects: 22, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (22/22), 1.82 KiB, done.
Total 22 (delta 4), reused 0 (delta 0)
To git@github.com:davidwparker/git.git
 * [new branch]      master -> master
```

Pushing will be rejected if someone else has since pushed upstream

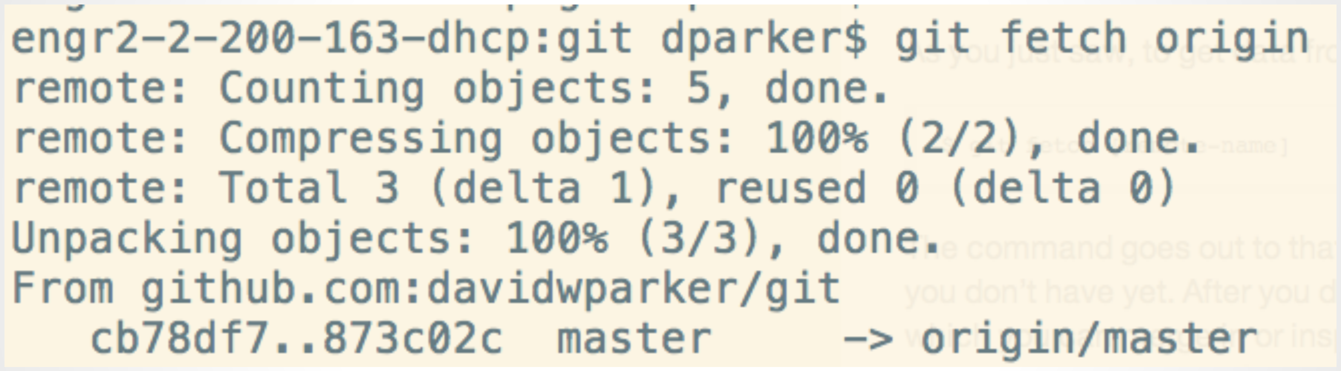
How to: Fetch Remote (I)

Fetching from a remote will pull down data you don't have yet.

It pulls the data into your local repository, but it doesn't automatically merge it with any of your work, or modify what you're currently working on.

How to: Fetch Remote (II)

- git fetch origin



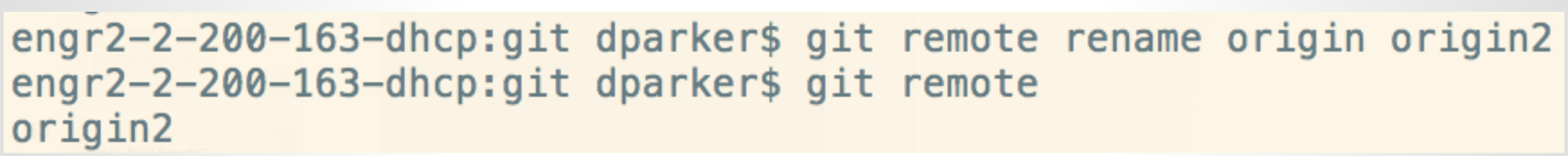
```
engr2-2-200-163-dhcp:git dparker$ git fetch origin
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From github.com:davidwparker/git
   cb78df7..873c02c  master     -> origin/master
```

In this example, I made changes on github.com and then fetched them into my repository.

How to: Changing Remotes

You can easily rename a remote

- git remote rename <old> <new>



```
engr2-2-200-163-dhcp:git dparker$ git remote rename origin origin2
engr2-2-200-163-dhcp:git dparker$ git remote
origin2
```

Or remove a remote

- git remote rm <name>

How to: Tagging

Tagging allows Git to forever remember a snapshot of a repository.

There are two types of tags in Git:

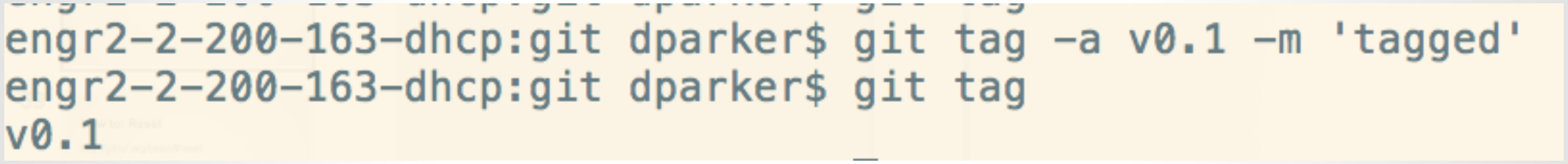
- Lightweight: a pointer to a specific commit
- Annotated: full objects in the Git database

It is recommended to use annotated tags.

How to: Creating an Annotated Tag

Annotated tagging is extremely easy:

- git tag -a <tagname> -m 'a message'



```
engr2-2-200-163-dhcp:git dparker$ git tag -a v0.1 -m 'tagged'
engr2-2-200-163-dhcp:git dparker$ git tag
v0.1
```

As you can see, you can also list tags with the command:

- git tag

How to: Creating a Signed Tag

Signed tagging is extremely easy:

- git tag -s <tagname> -m 'a message'

This uses GPG (GNU Privacy Guard)

The GPG signature can be seen using:

- git show <tagname>

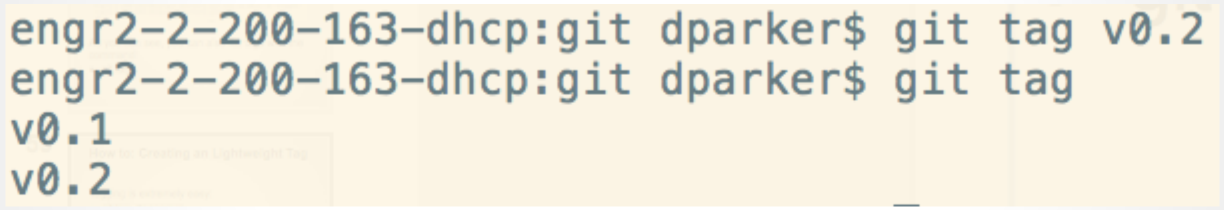
You can verify a signed tag as long as you have the signer's public key:

- git tag -v <tagname>

How to: Creating an Lightweight Tag

Lightweight tagging is extremely easy:

- git tag <tagname>



```
engr2-2-200-163-dhcp:git dparker$ git tag v0.2
engr2-2-200-163-dhcp:git dparker$ git tag
v0.1
v0.2
```

This will create a lightweight tag. Lightweight tags cannot use the -a, -s, or -m flags.

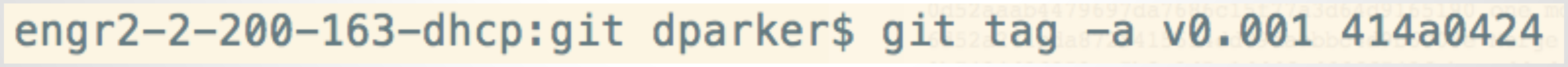
How to: Tagging later

If you forgot to tag, you can check your commits with:

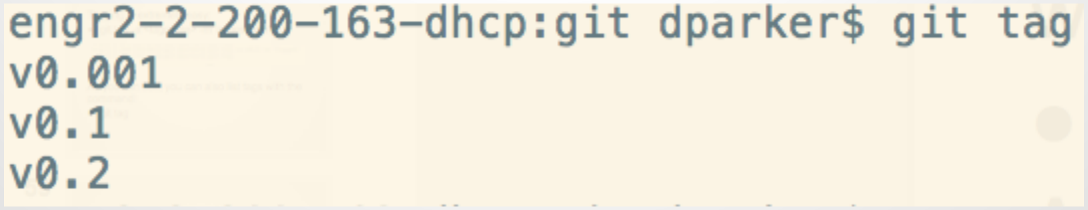
- git log --pretty=oneline

And then tag using the checksum:

- git tag -a <tagname> <checksum>



```
engr2-2-200-163-dhcp:git dparker$ git tag -a v0.001 414a0424
```



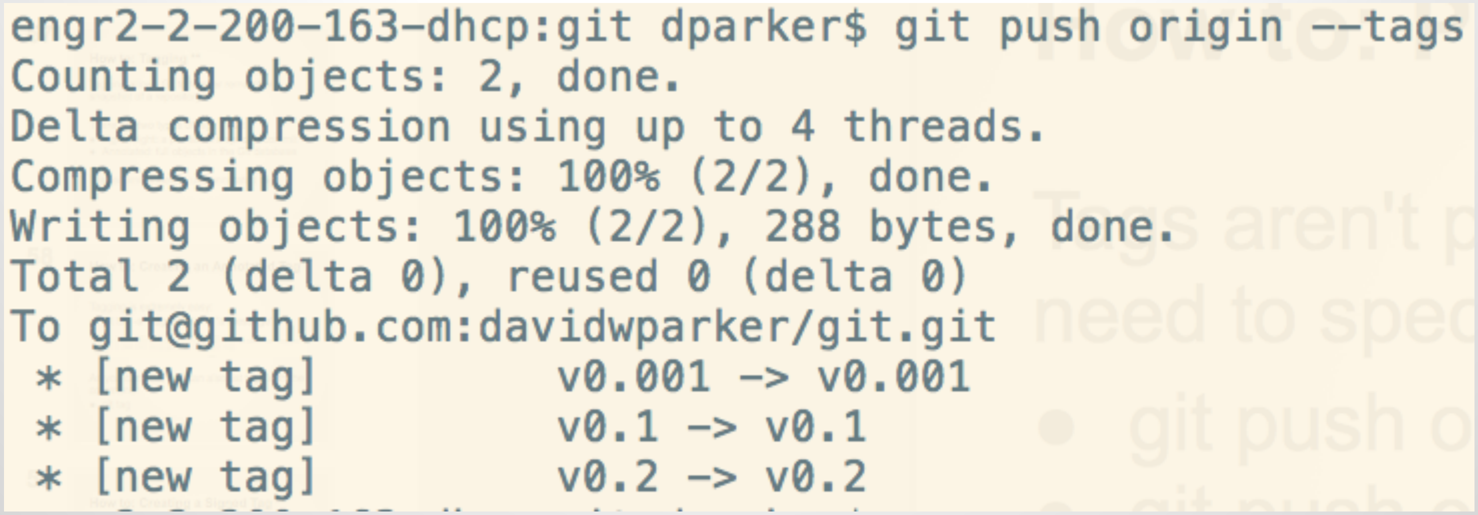
```
engr2-2-200-163-dhcp:git dparker$ git tag
v0.001
v0.1
v0.2
```

How to: Pushing Tags

Tags aren't pushed when doing a push, you need to specify them

- git push origin <tagname>
- git push origin --tags

Use the latter to push all tags



```
engr2-2-200-163-dhcp:git dparker$ git push origin --tags
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 288 bytes, done.
Total 2 (delta 0), reused 0 (delta 0)
To git@github.com:davidwparker/git.git
 * [new tag]         v0.001 -> v0.001
 * [new tag]         v0.1 -> v0.1
 * [new tag]         v0.2 -> v0.2
```

How to: Branching

One of Git's most powerful features is its branches.

Git's branches are incredibly lightweight, and it is nearly instantaneous to switch back and forth between branches.

Git encourages a workflow that branches and merges often, even multiple times a day.

How to: Why Branch?

A realistic workflow may be as follows:

1. Working on an app
2. Create a branch for a story you're working on
3. Do some work

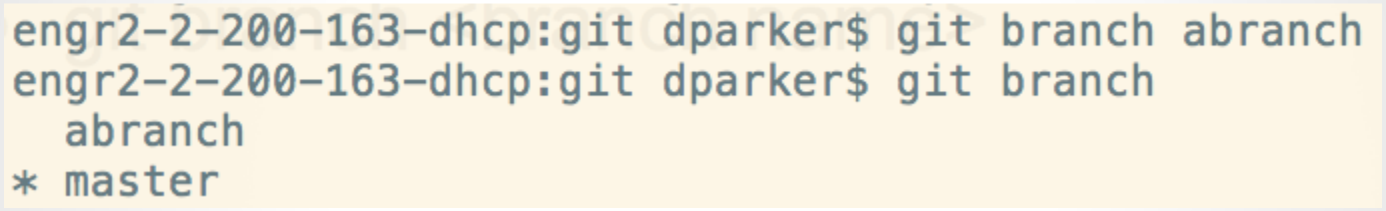
Then, you get a call for critical hotfix needed:

1. Revert back to production branch
2. Create branch for hotfix
3. Test hotfix and merge the branch, push to production
4. Switch back to original story and continue

How to: Creating a branch

Creating a branch is incredibly easy:

- git branch <branch name>



```
engr2-2-200-163-dhcp:git dparker$ git branch abranch
engr2-2-200-163-dhcp:git dparker$ git branch
  abranch
* master
```

This creates a pointer to the same commit you're currently on.

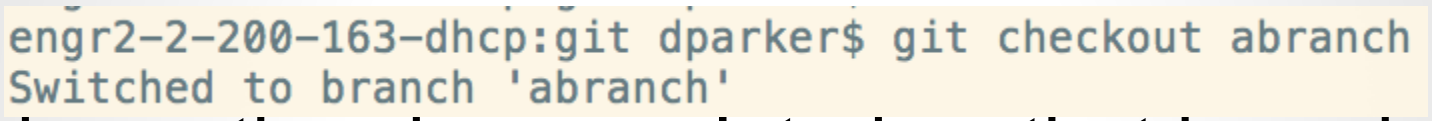
As you can see above, you can easily list what branches there are, as well as see your current branch (marked with *)

- git branch

How to: Branching

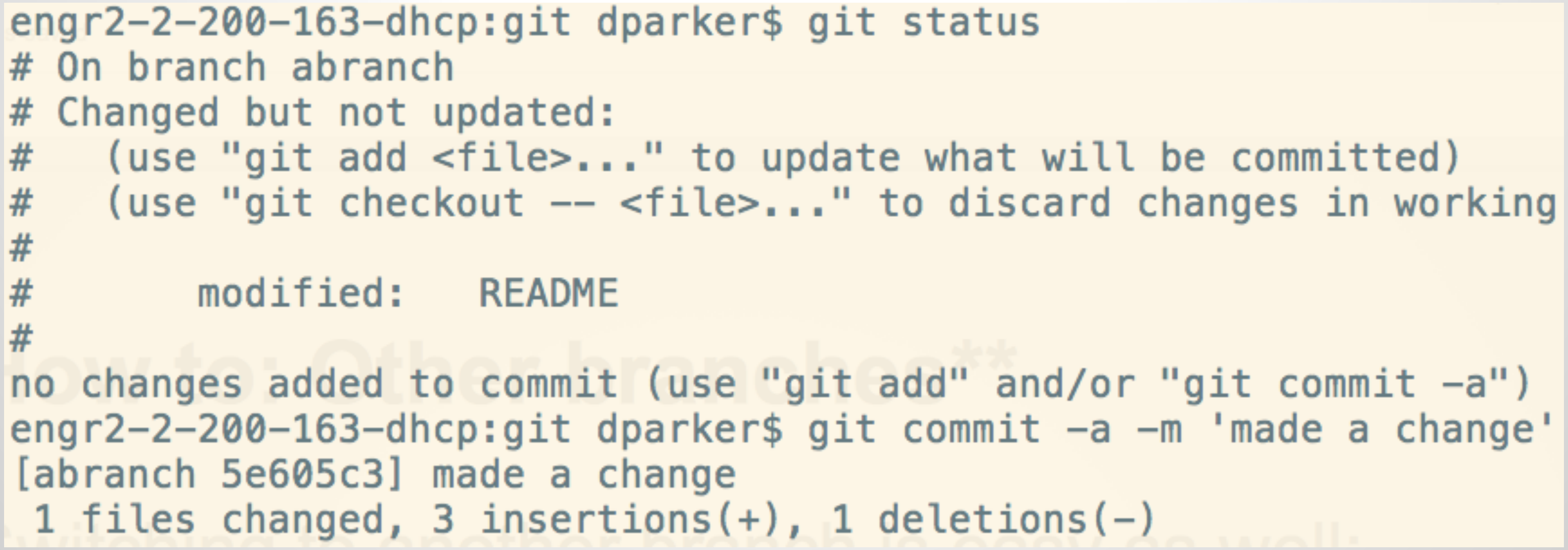
Switching to another branch is easy as well:

- git checkout <branch name>



```
engr2-2-200-163-dhcp:git dparker$ git checkout abranch
Switched to branch 'abranch'
```

Work can then be completed on that branch:

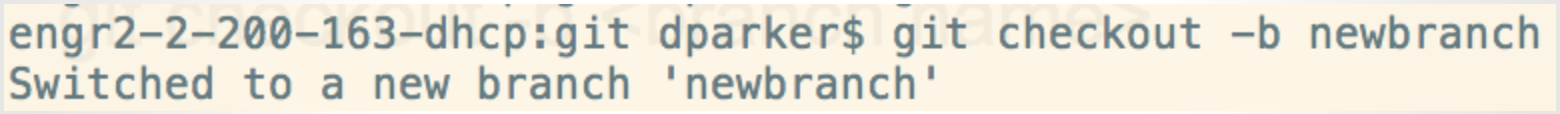


```
engr2-2-200-163-dhcp:git dparker$ git status
# On branch abranch
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#
#       modified:   README
#
no changes added to commit (use "git add" and/or "git commit -a")
engr2-2-200-163-dhcp:git dparker$ git commit -a -m 'made a change'
[abranch 5e605c3] made a change
 1 files changed, 3 insertions(+), 1 deletions(-)
```

How to: Branching

You can also easily checkout a branch when you create it:

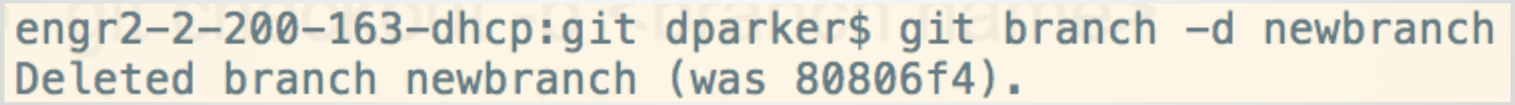
- git checkout -b <branch name>



```
engr2-2-200-163-dhcp:git dparker$ git checkout -b newbranch
Switched to a new branch 'newbranch'
```

When you are completely done with a branch, you can easily delete it:

- git branch -d <branch name>

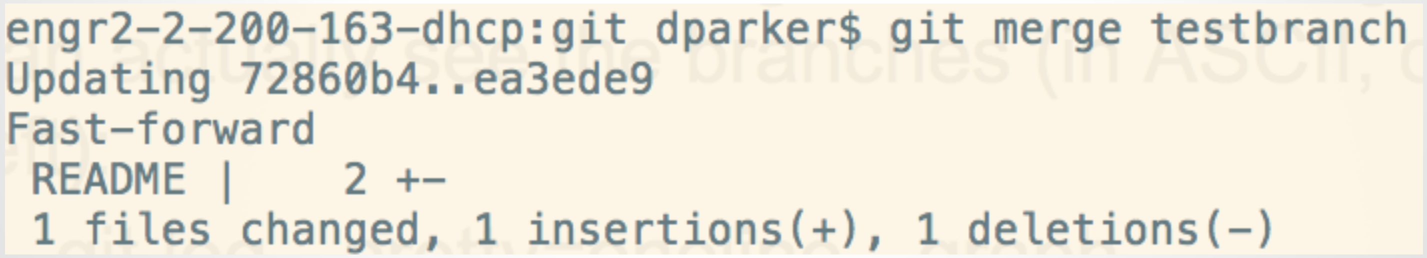


```
engr2-2-200-163-dhcp:git dparker$ git branch -d newbranch
Deleted branch newbranch (was 80806f4).
```

How to: Merging (I)

If you don't edit a branch, and then merge another branch where you have changed things, then Git performs a fast forward.

- git merge <branch name>

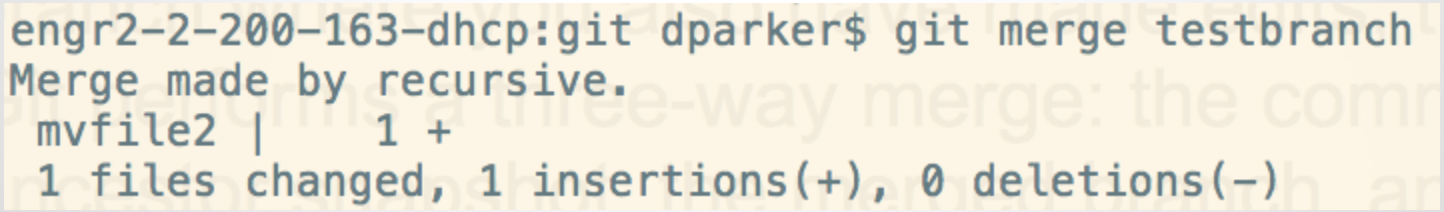


```
engr2-2-200-163-dhcp:git dparker$ git merge testbranch
Updating 72860b4..ea3ede9
Fast-forward
 README |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

How to: Merging (II)

If you do edit a branch, and then merge another branch where you also have made edits, then Git performs a three-way merge: the common ancestor snapshot, the merged branch, and the merging branch.

- git merge <branch name>

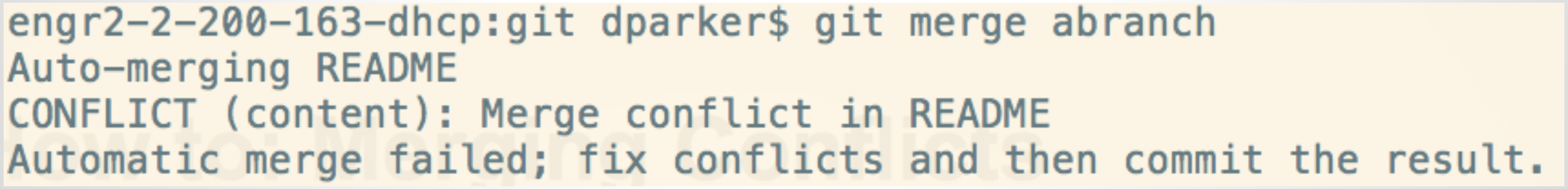


```
engr2-2-200-163-dhcp:git dparker$ git merge testbranch
Merge made by recursive.
 mvfile2 |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

How to: Merging Conflicts (I)

If you edit a branch, and attempt to merge another branch where you have edited the same part of the same file, you may end up with a conflict.

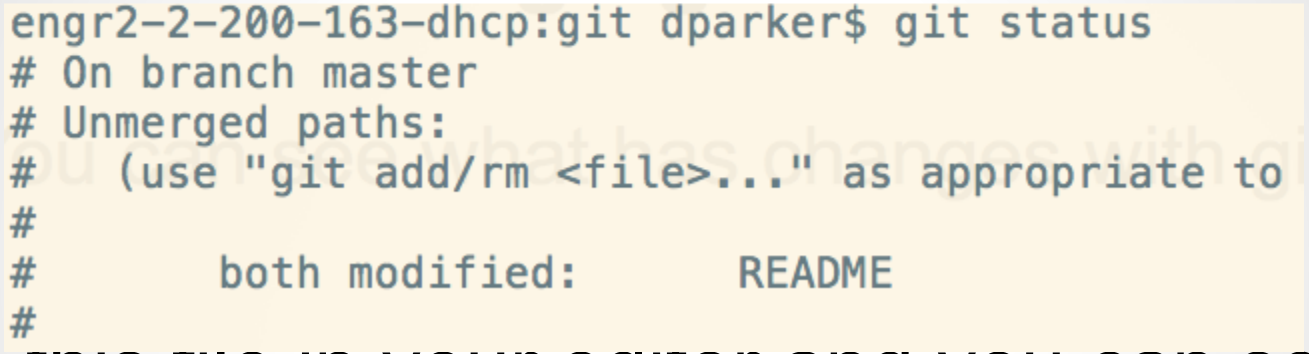
- git merge abranch



```
engr2-2-200-163-dhcp:git dparker$ git merge abranch
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
```

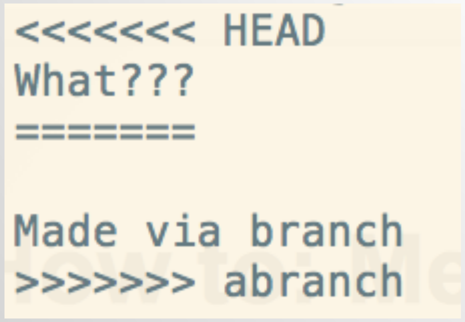
How to: Merging Conflicts (II)

You can see what has changes with git status



```
engr2-2-200-163-dhcp:git dparker$ git status
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to
#
#       both modified:      README
#
```

Open this file in your editor and you can see where the conflict is:

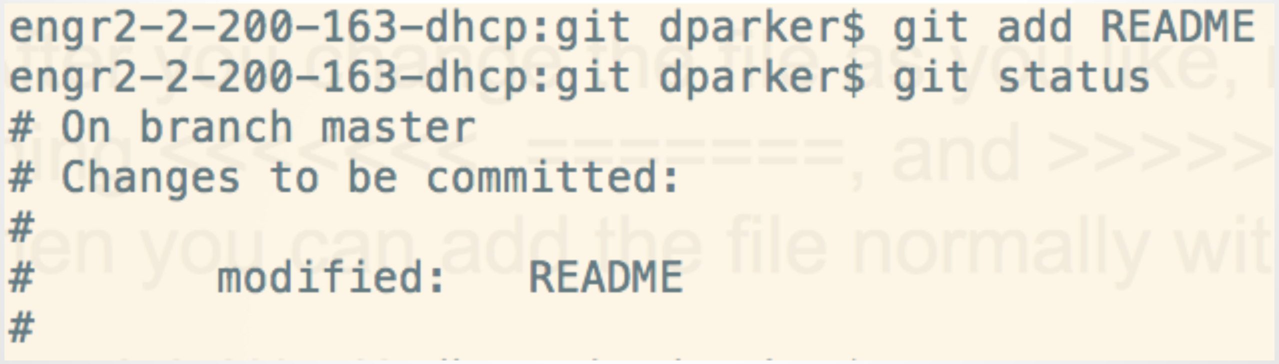


```
<<<<<<< HEAD
What???
=======

Made via branch
>>>>>>> abranch
```

Changes made in HEAD are above ======= and changes made in branch are below.

How to: Merging Conflicts (III)

After you change the file as you like, remove thing <<<<<<<, =======, and >>>>>>> lines, then you can add the file normally with git add.

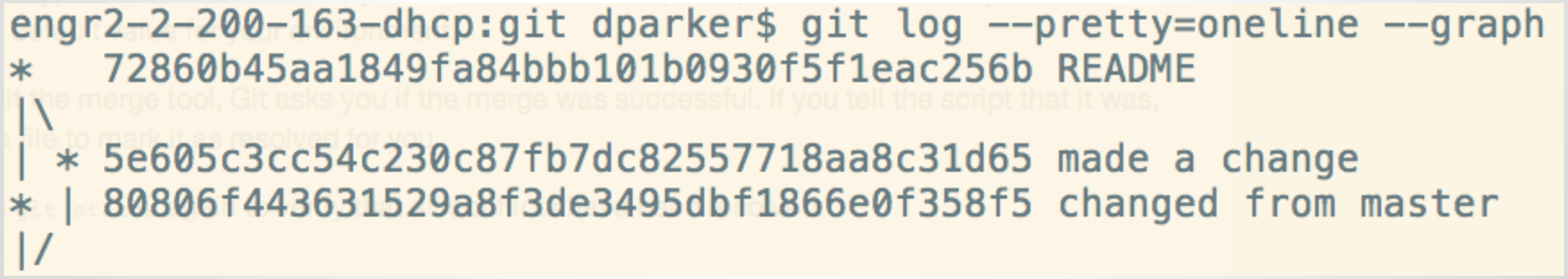


```
engr2-2-200-163-dhcp:git dparker$ git add README
engr2-2-200-163-dhcp:git dparker$ git status
# On branch master
# Changes to be committed:
#
#       modified:   README
#
```

How to: Branching (log)

Now that we have merged, if we do a log, we can actually see the branches (in ASCII, on the left):

- git log --pretty=oneline --graph

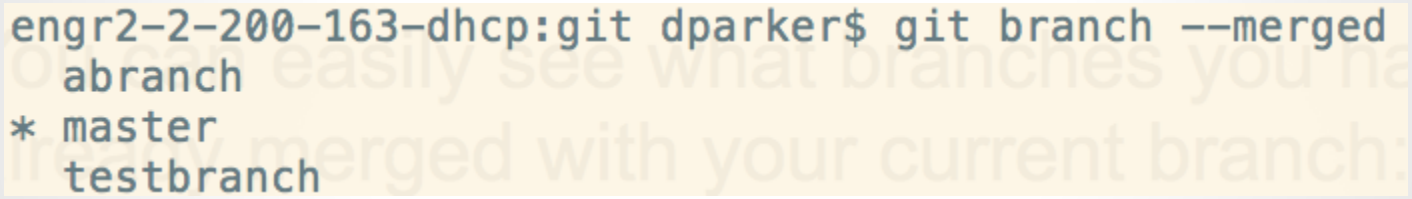


```
engr2-2-200-163-dhcp:git dparker$ git log --pretty=oneline --graph
*   72860b45aa1849fa84bbb101b0930f5f1eac256b README
|\
| * 5e605c3cc54c230c87fb7dc82557718aa8c31d65 made a change
* | 80806f443631529a8f3de3495dbf1866e0f358f5 changed from master
|/
```

How to: Branching tips

You can easily see what branches you have already merged with your current branch:

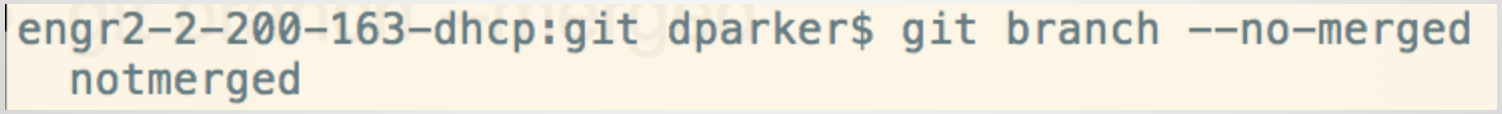
- git branch --merged



```
engr2-2-200-163-dhcp:git dparker$ git branch --merged
  abranch
* master
  testbranch
```

Or not merged:

- git branch --no-merged



```
engr2-2-200-163-dhcp:git dparker$ git branch --no-merged
  notmerged
```

How to: Remote branches

Remote branches work similarly to local branches, except that they take the form <origin>/<branch>.

In general, you must remember to "git fetch" from remote to get the latest.

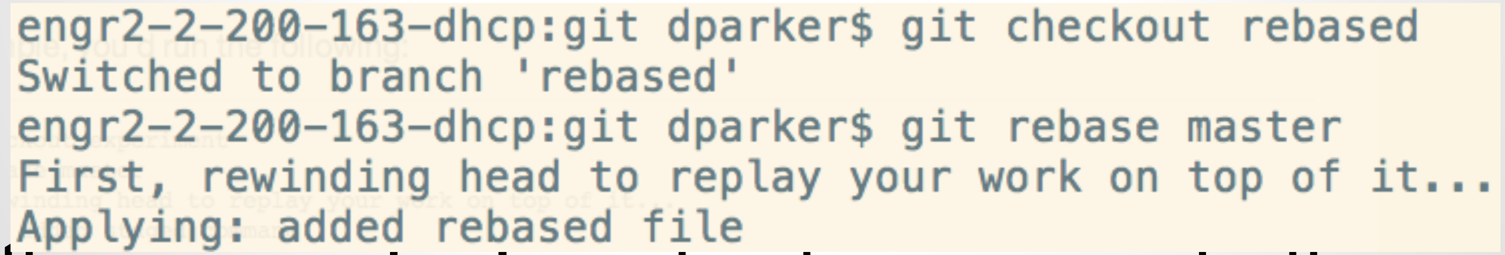
From there, you don't get that work in your working directory, but you can merge it with "git merge origin/<new branch>"

And you must "git push" to push the latest to the remote repository.

How to: Rebasing (I)

Rebasing is another tool that allows you to integrate changes from one branch to another.

Rebasing allows you to take all the changes that were committed on one branch and replay them on another branch.

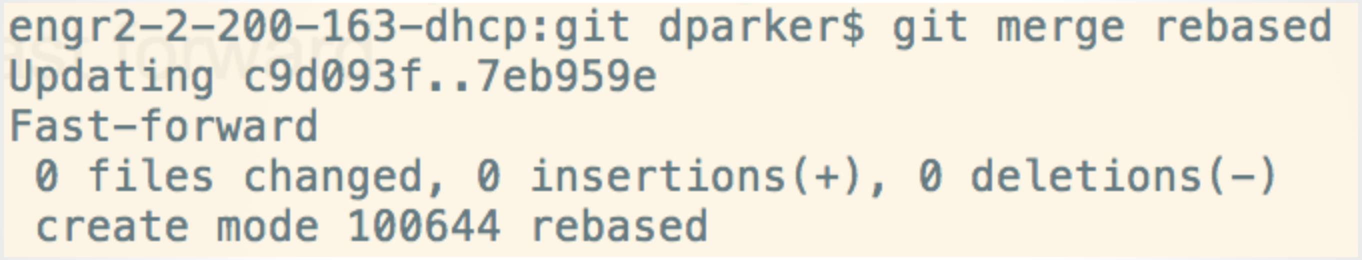


```
engr2-2-200-163-dhcp:git dparker$ git checkout rebased
Switched to branch 'rebased'
engr2-2-200-163-dhcp:git dparker$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added rebased file
```

In this example, I made changes on both rebased and master, then replayed the master changes on rebased.

How to: Rebasing (II)

At this point, you can go back to master and fast forward.



```
engr2-2-200-163-dhcp:git dparker$ git merge rebased
Updating c9d093f..7eb959e
Fast-forward
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 rebased
```

The most often usecase for rebase is to make sure your commits apply cleanly to a remote branch.

Rebasing is great for cleaning up when you have made a ton of 'unnecessary' commits.

How to: Rebasing (III)

If you follow the previous workflow, you'll be ok.
Otherwise, a warning: do not rebase commits that you have pushed to a public repository.

When you rebase, you're abandoning existing commits and creating new ones that are similar but different.

Only rebase commits that you haven't push publicly.

How to: Git on the Server (I)

So far, everything has been related to using Git on the client. However, in order to work with others, then someone is going to need to set up a server.

You can push/pull from individual repos, but that's typically frowned upon, as it may confuse who has what files.

Personal note: I didn't set up a personal server, but choose to use the excellent web app GitHub

How to: Git Server Protocols (I)

To run a Git server, all you need is to choose protocols you want your server to communicate with:

- Local
- SSH
- Git
- HTTP(S)

How to: Git Server Protocols (II)

Local

- git clone /opt/git/project.git

- Pros
  - Useful with shared filesystem
  - Quick to grab others' work
- Cons
  - May be harder than just sharing over network

How to: Git Server Protocols (III)

SSH

- git clone ssh://user@server:project.git

- Pros
  - Generally already set up (or easy to set up)
  - Authenticated network protocol
  - Used for writes anyway
  - Efficient
- Cons
  - Can't serve anonymous access to the repositories

How to: Git Server Protocols (IV)

Git

- git clone git://user@server:project.git

- Pros
    - Fastest protocol available
- Cons
    - Read only
    - Lacks authentication

How to: Git Server Protocols (V)

HTTP(S)

- git clone http://user@server:project.git

- Pros
  - Easy to set up
  - Commonly used protocols (so corporate firewalls are still generally okay)
- Cons
  - Read only
  - Inefficient for the client

How to: Setting up the Git Server

Get a bare repository

- git clone --bare my_project my_project.git

A bare repository is a repository without a working directory

Put the bare repository on the server

- scp -r my_project.git user@git.example.com:/opt/git

Puts my_project under /opt/git

Now just set up your protocols

How to: More on the Protocols

At this point you would set up SSH, git, or another protocol.

Setting up the protocols exactly is beyond the nature of this presentation, but know it's not too difficult to set up.

How to: All the Rest (I)

Git's capabilities seem to go on and on. At this point, I'm going to name drop a few of the other things that Git can do, but without going into much detail (I have too many slides already)

- Revision Selection
  - Git allows you to specify specific commits or a range of commits in several ways
- Interactive Staging
  - Command line scripts to make some issues with staging easier.

How to: All the Rest (II)

- Stashing
  - Used when you need to do work on another branch suddenly, but your current work is half-completed and you don't want to commit it yet.
- Rewriting History
  - Useful for when you need to rewrite a commit so that it looks a certain way
- Debugging
  - Git uses bisect (binary search) to help determine where code may have become bad

How to: All the Rest (III)

- Submodules
  - Great for using a project within a project

- Git Hooks
  - Program the system to automatically do something after you perform a commit, or another git action.
  - Used on client or server

- Much, much more.

Git Internals (I)

When you create a Git repository, Git creates a .git folder, which is where almost everything Git stores and manipulates is located.

Inside the .git folder, there are four core parts:

- HEAD file
- index file
- objects directory
- refs directory

Git Internals (II)

Two data structures:

- mutable index that caches information about the working directory and the next revision to be committed
- immutable, append-only object database

Object database objects:

- blob = content of file
- tree = directory
- commit = links trees together
- tag = container that contains reference to another object and can hold metadata

Git Internals (III)

Index connects object database and working tree

Each object is identified by a SHA-1 hash of its contents

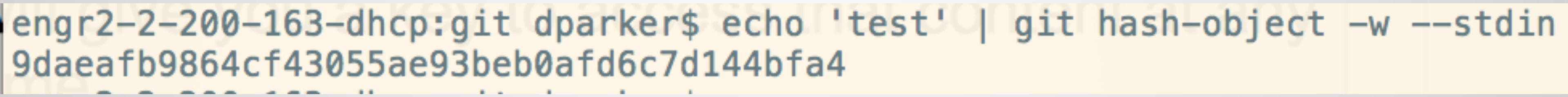
Objects are stored in entirety using zlib compression

Git Objects (I)

At its core, Git is a key-value object store.

You can insert any kind of content into it, and it will give you a key to access that content at any time. All Git objects are stored as blobs.

- echo 'test' | git hash-object -w --stdin
  - hash-object stores data in the .git directory.
  - -w says to store the object
  - --stdin tells command to read from stdin, otherwise it expects a file

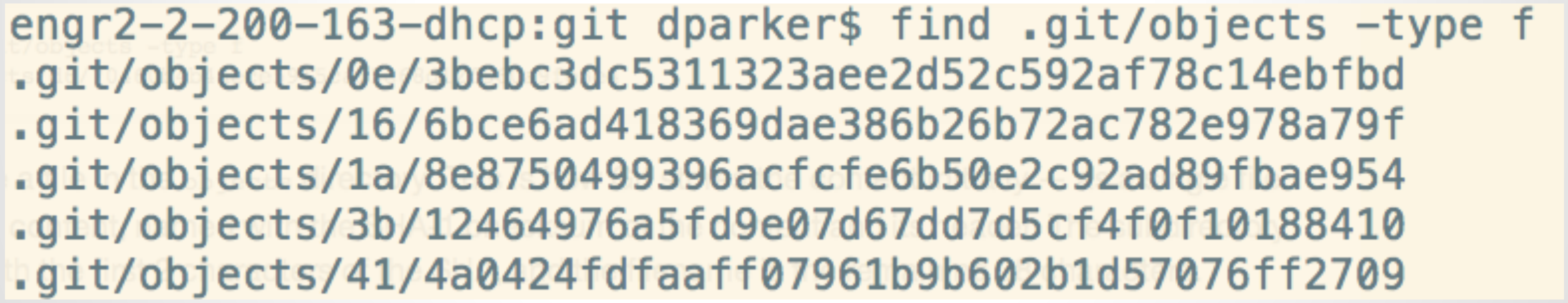


```
engr2-2-200-163-dhcp:git dparker$ echo 'test' | git hash-object -w --stdin
9daeafb9864cf43055ae93beb0afd6c7d144bfa4
```

Git Objects (II)

You can view the objects at any time with:

- find .git/objects -type f

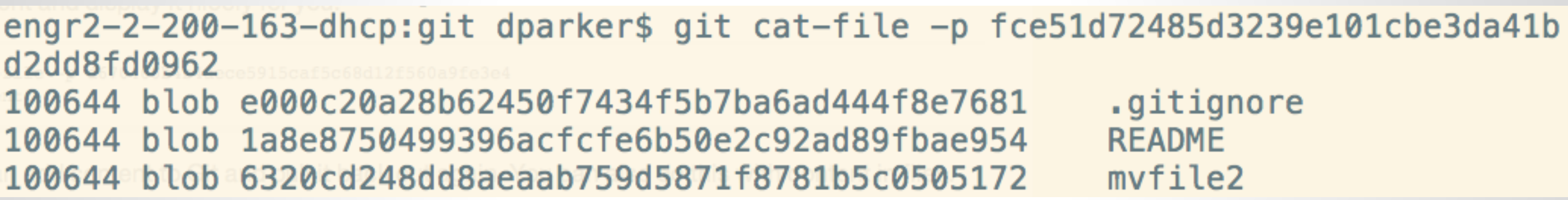


```
engr2-2-200-163-dhcp:git dparker$ find .git/objects -type f
.git/objects/0e/3bebc3dc5311323aee2d52c592af78c14ebfbd
.git/objects/16/6bce6ad418369dae386b26b72ac782e978a79f
.git/objects/1a/8e8750499396acfcfe6b50e2c92ad89fbae954
.git/objects/3b/12464976a5fd9e07d67dd7d5cf4f0f10188410
.git/objects/41/4a0424fdfaaff07961b9b602b1d57076ff2709
```

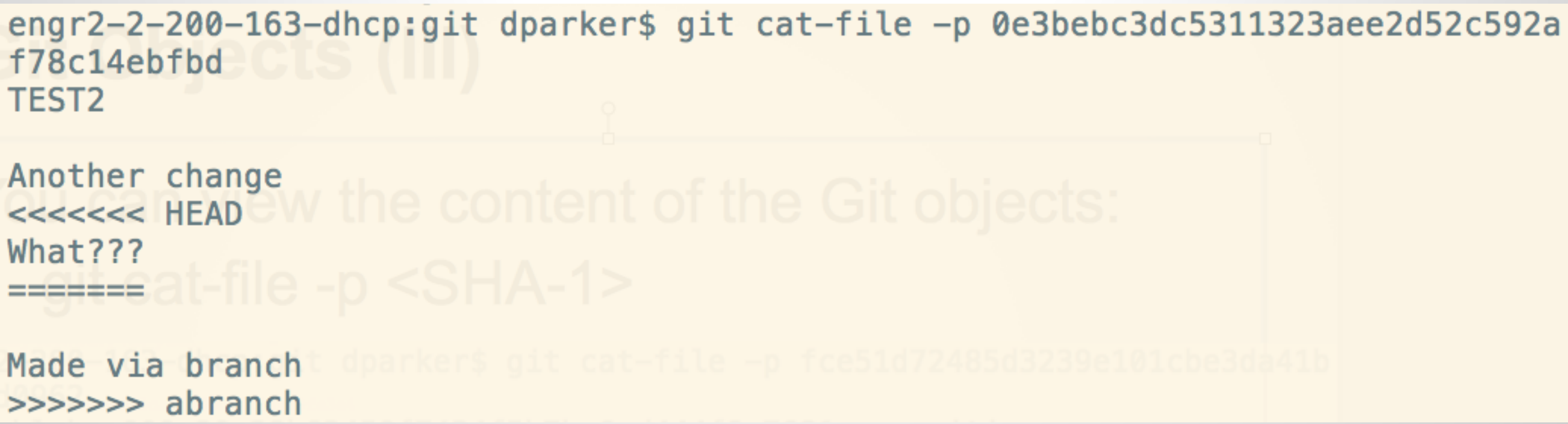
Git Objects (III)

You can view the content of the Git objects:

- git cat-file -p <SHA-1>



```
engr2-2-200-163-dhcp:git dparker$ git cat-file -p fce51d72485d3239e101cbe3da41b
d2dd8fd0962
100644 blob e000c20a28b62450f7434f5b7ba6ad444f8e7681    .gitignore
100644 blob 1a8e8750499396acfcfe6b50e2c92ad89fbae954    README
100644 blob 6320cd248dd8aeaab759d5871f8781b5c0505172    mvfile2
```

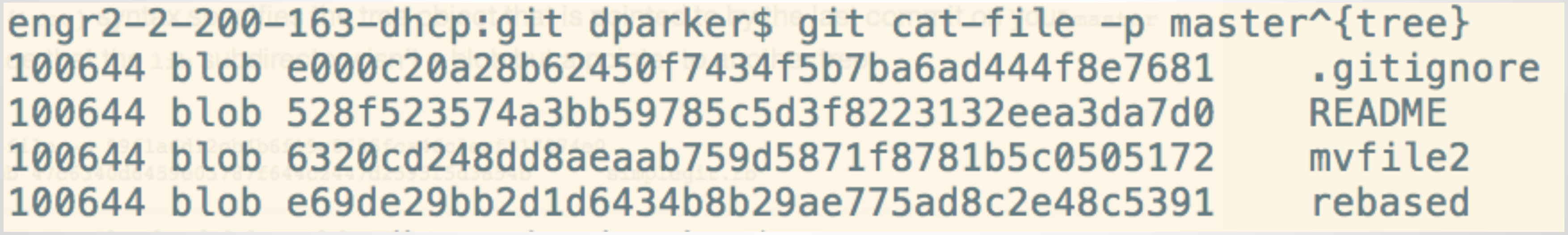


```
engr2-2-200-163-dhcp:git dparker$ git cat-file -p 0e3bebc3dc5311323aee2d52c592a
f78c14ebfbd
TEST2

Another change
<<<<<<< HEAD
What???
=======

Made via branch
>>>>>>> abranch
```

Tree Objects (I)

Tree objects solve the problem of storing the filename and also allows you to store a group of files together.

- git cat-file -p master^{tree}

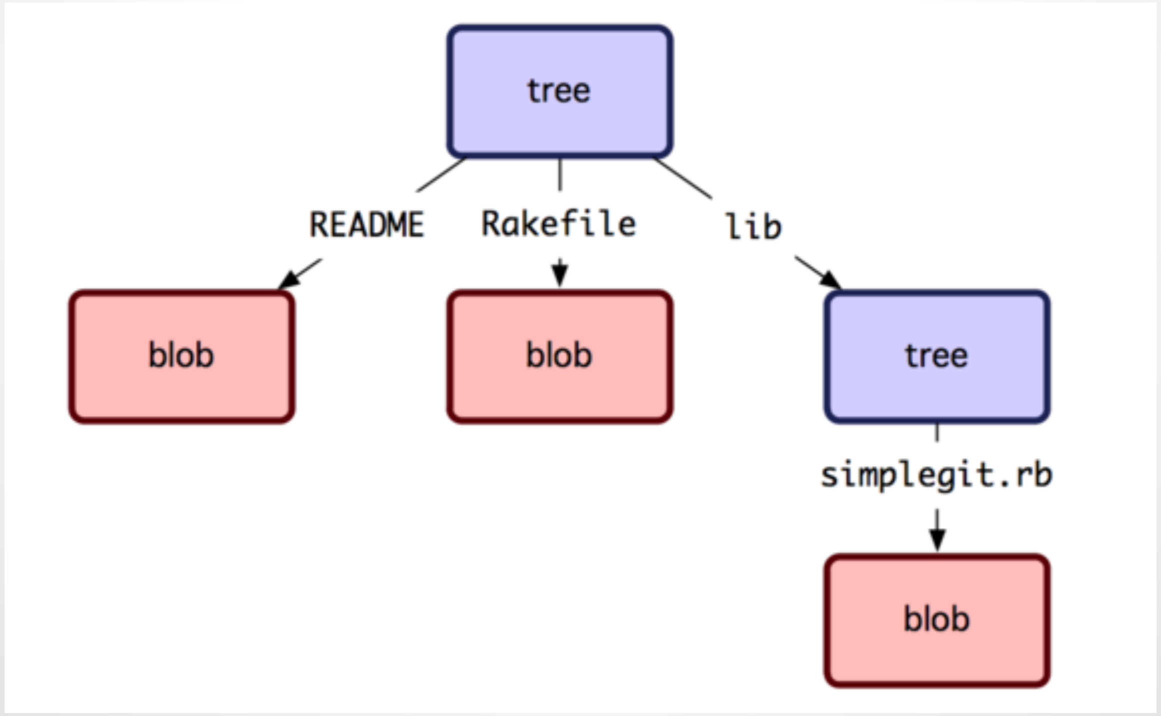


```
engr2-2-200-163-dhcp:git dparker$ git cat-file -p master^{tree}
100644 blob e000c20a28b62450f7434f5b7ba6ad444f8e7681    .gitignore
100644 blob 528f523574a3bb59785c5d3f8223132eea3da7d0    README
100644 blob 6320cd248dd8aeaab759d5871f8781b5c0505172    mvfile2
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    rebased
```

master^{tree} specifies tree object pointed to by last commit on the master branch.

Tree Objects (II)

Conceptually, Git stores something like this:

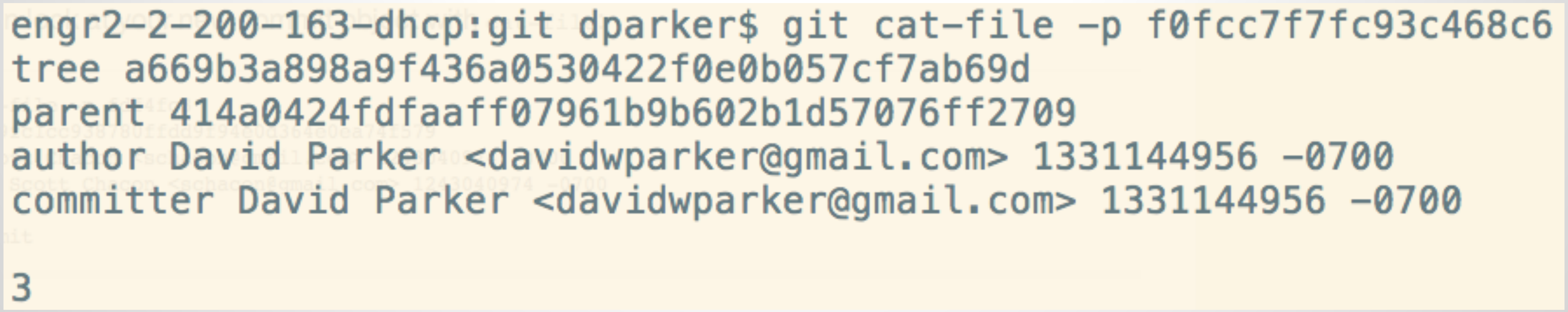


Commit Objects (I)

Commit objects store who saved snapshots, when they were saved, and/or why they were saved.

On a commit object, you can run the command:

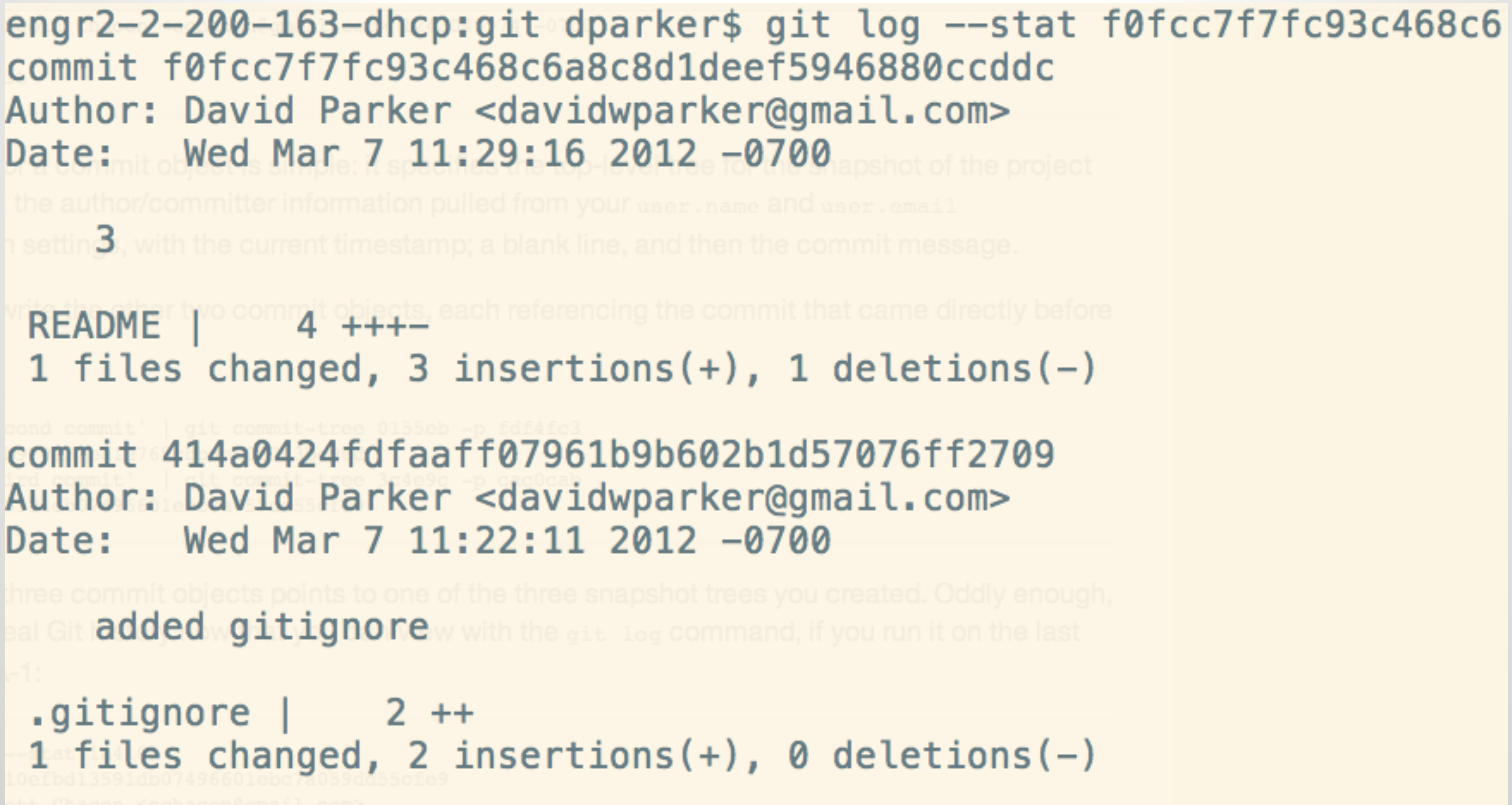
- git cat-file -p <SHA-1>



```
engr2-2-200-163-dhcp:git dparker$ git cat-file -p f0fcc7f7fc93c468c6
tree a669b3a898a9f436a0530422f0e0b057cf7ab69d
parent 414a0424fdfaaff07961b9b602b1d57076ff2709
author David Parker <davidwparker@gmail.com> 1331144956 -0700
committer David Parker <davidwparker@gmail.com> 1331144956 -0700

3
```

Commit Objects (II)

You can also run git log on a SHA-1 of a commit object to see the real Git history:



```
engr2-2-200-163-dhcp:git dparker$ git log --stat f0fcc7f7fc93c468c6
commit f0fcc7f7fc93c468c6a8c8d1deef5946880ccddc
Author: David Parker <davidwparker@gmail.com>
Date:   Wed Mar 7 11:29:16 2012 -0700

    3

 README |    4 +++-
 1 files changed, 3 insertions(+), 1 deletions(-)

commit 414a0424fdfaaff07961b9b602b1d57076ff2709
Author: David Parker <davidwparker@gmail.com>
Date:   Wed Mar 7 11:22:11 2012 -0700

    added gitignore

 .gitignore |    2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)
```

Git References

You can find the files that contain the SHA-1 values in the .git/refs directory.

If you are constantly referencing a specific file by its SHA-1, then refs helps you to remember those more easily.

Rather than: git log 1a40ae3...

You can do

- git update-ref refs/head/master 1a40ae3...

Then you get: git log master

The HEAD file

The HEAD file is a symbolic reference to the branch you're currently on.

Unlike a normal reference, it doesn't contain a SHA-1 value, but a pointer to another pointer

When you run a git commit, it creates a commit object, specifying the parent of that object to be whatever SHA-1 value the reference in HEAD points to. Read and write with the commands:

- git symbolic-ref HEAD
- git symbolic-ref HEAD refs/heads/different

Tag Objects

A tag object is like a commit object, except it points to a commit rather than a tree.

It's like a branch reference, but it never moves forward - it always points to same commit.

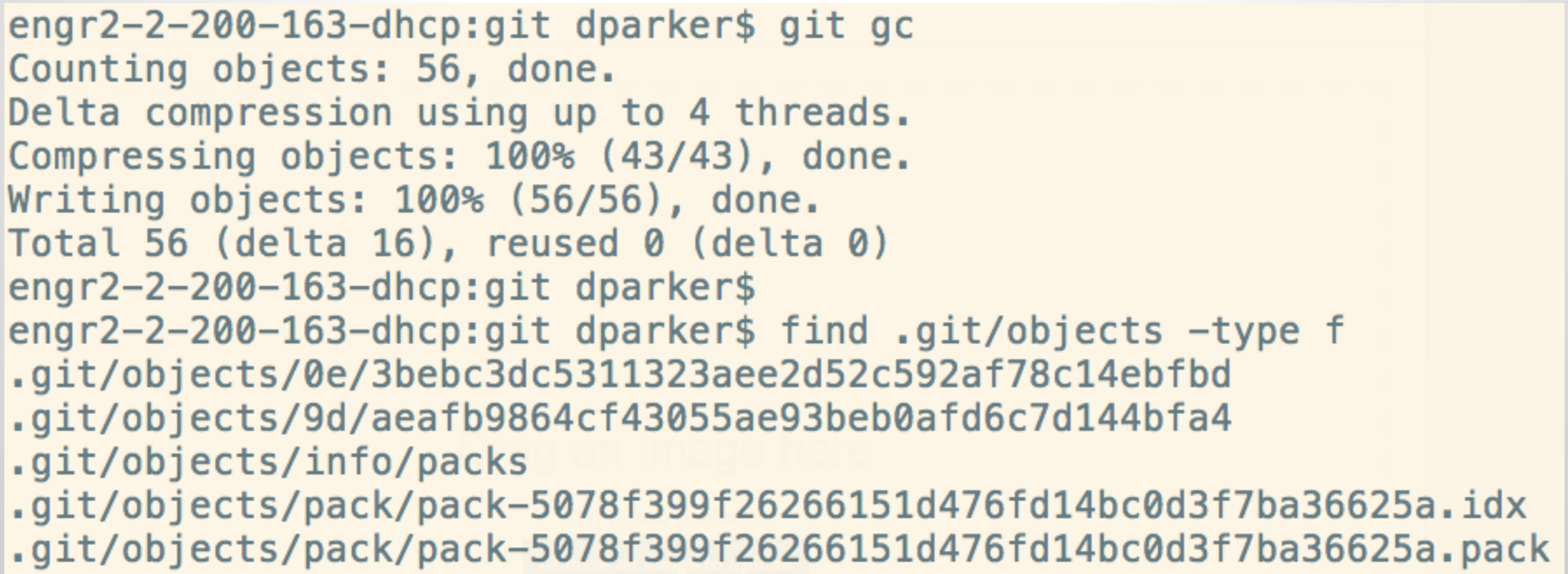
That's all a lightweight tag is (to be discussed later) - a branch that never moves

With an annotated tag, Git creates a tag object and then writes a reference to point to it rather than the commit directly.

Packfiles (I)

Git has the ability to only store files and their respective deltas, which is great when big files get modified so you don't store the file twice.

- git gc | find .git/objects -type f



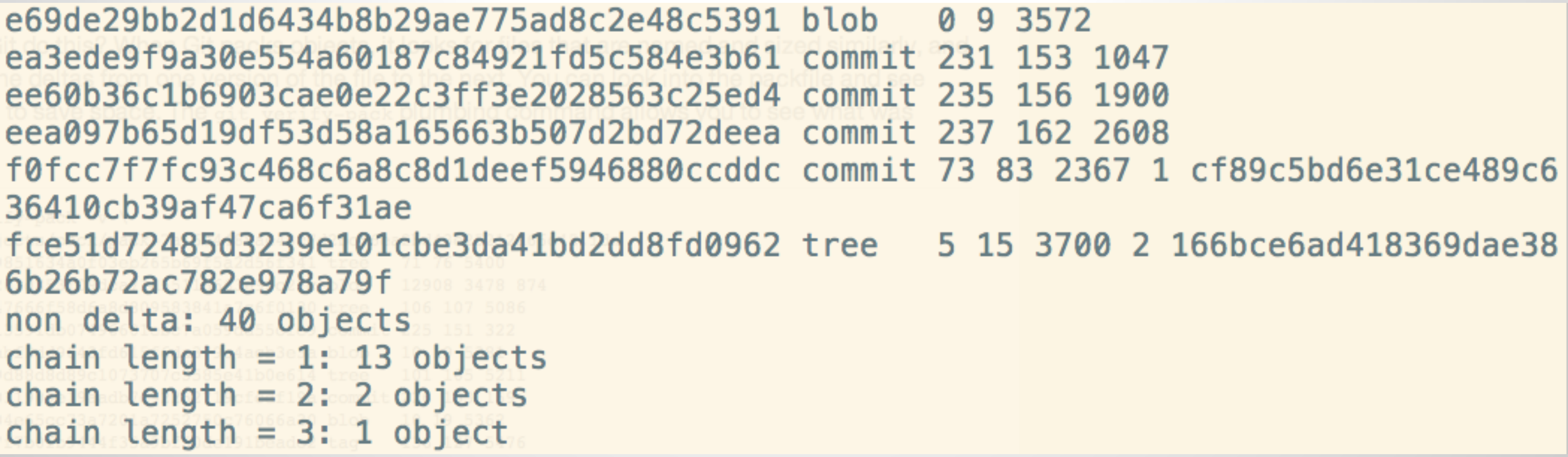
```
engr2-2-200-163-dhcp:git dparker$ git gc
Counting objects: 56, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (43/43), done.
Writing objects: 100% (56/56), done.
Total 56 (delta 16), reused 0 (delta 0)
engr2-2-200-163-dhcp:git dparker$
engr2-2-200-163-dhcp:git dparker$ find .git/objects -type f
.git/objects/0e/3bebc3dc5311323aee2d52c592af78c14ebfbd
.git/objects/9d/aeafb9864cf43055ae93beb0afd6c7d144bfa4
.git/objects/info/packs
.git/objects/pack/pack-5078f399f26266151d476fd14bc0d3f7ba36625a.idx
.git/objects/pack/pack-5078f399f26266151d476fd14bc0d3f7ba36625a.pack
```

Packfiles (II)

As you can see, a majority of the Git objects are gone, and now we have a .pack file.

Use verify-pack to see what was packed up:

- git verify-pack -v .git/objects/pack/<pack.idx>



```
e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 blob   0 9 3572
ea3ede9f9a30e554a60187c84921fd5c584e3b61 commit 231 153 1047
ee60b36c1b6903cae0e22c3ff3e2028563c25ed4 commit 235 156 1900
eea097b65d19df53d58a165663b507d2bd72deea commit 237 162 2608
f0fcc7f7fc93c468c6a8c8d1deef5946880ccddc commit 73 83 2367 1 cf89c5bd6e31ce489c6
36410cb39af47ca6f31ae
fce51d72485d3239e101cbe3da41bd2dd8fd0962 tree   5 15 3700 2 166bce6ad418369dae38
6b26b72ac782e978a79f
non delta: 40 objects
chain length = 1: 13 objects
chain length = 2: 2 objects
chain length = 3: 1 object
```

Other Useful Related Tools (I)

User Interface Tools:

- qgit: http://sourceforge.net/projects/qgit/
- Tig: http://jonas.nitro.dk/tig/

Tool Shipped with Git:

- gitk: Original TCL/TK GUI for browsing Git repos history
- Git-gui: Simple TK based graphical interface for common Git Operations
- gitweb: Full-fledged web interface for Git repositories

Other Useful Related Tools (II)

Version Control Interface Layers:

- StGit: http://www.procode.org/stgit/
- Cogito: http://git.or.cz/cogito/

Public Hosting:

- repo.or.cz: http://repo.or.cz/
- GitHub: https://github.com/
- Gitorious: http://gitorious.org/

What projects use Git?

- Linux Kernel
- Ruby on Rails
- Android
- Drupal
- WINE
- X.org
- Eclipse
- GCC
- KDE
- Qt
- GNOME
- jQuery
- Perl5
- Debian
- VLC
- Rubinius
- PostgreSQL
- Puppet
- phpMyAdmin
- GNU Scientific Library (GSL)
- Many more...

More Open Source VCSs and DVCSs

VCSs

- Subversion
- Concurrent Versions System (CVS)
- OpenCVS

DVCSs

- Bazaar
- Mercurial
- Darcs
- Fossil
- GNU arch
- Aegis
- Many more...

References

http://git.or.cz/index.html
http://git-scm.com/
http://progit.org/book/
http://gitref.org/
http://gitready.com/
http://whygitisbetterthanx.com/
http://en.wikipedia.org/wiki/Git_(software)
http://en.wikipedia.org/wiki/List_of_revision_control_software
http://en.wikipedia.org/wiki/Comparison_of_revision_control_software
https://git.wiki.kernel.org/articles/g/i/t/GitLinks_efb4.html
https://git.wiki.kernel.org/articles/g/i/t/GitProjects_8074.html
http://hoth.entp.com/output/git_for_designers.html
http://eagain.net/articles/git-for-computer-scientists/
http://schacon.github.com/git/everyday.html