GIT / GITHUB LAB

Contents

Lab 1 Setup

Goals

- To setup git so that it is ready for work.

Setup Name and Email

If you have never used git before, you need to do some setup first. Run the following commands so that git knows your name and email. If you have git already setup, you can skip down to the line ending section.

Execute:

```
git config --global user.name "Your Name"
git config --global user.email "your_email@whatever.com"
```

Setup Line Ending Preferences

Also, for Unix/Mac users:

Execute:

```
git config --global core.autocrlf input
git config --global core.safecrlf true
```

And for Windows users:

Execute:

```
git config --global core.autocrlf true
git config --global core.safecrlf true
```

Text Editor

Please have your instructor help you with editor. If in Linux, emacs, nano, etc is good. If in Windows, Notepad++ and other Windows editors are fine

Lab 2 More Setup

Goals

- Get the tutorial materials setup and ready to run.

Get the Tutorial package

Git the tutorial package from download or desktop

Unzip the tutorial

The tutorial package should have a main directory “git_tutorial” with three sub-directories:

- html — These html files. Point your browser to html/index.html
- work — An empty working directory. Create your repos in here.
- repos — Prepackaged Git repositories so you can jump into the tutorial at any point. If you get stuck, just copy the desired Lab into your working directory.

Lab 3 Create a Project

Goals

- Learn how to create a git repository from scratch.

Create a "Hello, World" program

Starting in the empty working directory, create an empty directory named "hello", then create a file named `hello.rb` with the contents below.

Execute:

```
mkdir hello
cd hello
```

File: *hello.rb*

```
puts "Hello, World"
```

Create the Repository

You now have a directory with a single file. To create a git repository from that directory, run the git init command.

Execute:

```
git init
```

Output:

```
$ git init
warning: templates not found /Users/tonywok/.git_template
Initialized empty Git repository in
/Users/tonywok/src/git_immersion/auto/hello/.git/
```

Add the program to the repository

Now let's add the "Hello, World" program to the repository.

Execute:

```
git add hello.rb
git commit -m "First Commit"
```

You should see …

Output:

```
$ git add hello.rb
$ git commit -m "First Commit"
[master (root-commit) cf466b4] First Commit
 1 file changed, 1 insertion(+)
 create mode 100644 hello.rb
```

Lab 4 Checking Status

Goals

- Learn how to check the status of the repository

Check the status of the repository

Use the `git status` command to check the current status of the repository.

Execute:

```
git status
```

You should see

Output:

```
$ git status
On branch master
nothing to commit, working directory clean
```

The status command reports that there is nothing to commit. This means that the repository has all the current state of the working directory. There are no outstanding changes to record.

We will use the `git status` command to continue to monitor the state between the repository and the working directory.

Lab 5 Making Changes

Goals

- Learn how to monitor the state of the working directory

Change the "Hello, World" program.

It's time to change our hello program to take an argument from the command line. Change the file to be:

File: *hello.rb*

```
puts "Hello, #{ARGV.first}!"
```

Check the status

Now check the status of the working directory.

Execute:

```
git status
```

You should see …

Output:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   hello.rb

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

The first thing to notice is that git knows that the `hello.rb` file has been modified, but git has not yet been notified of these changes.

Also notice that the status message gives you hints about what you need to do next. If you want to add these changes to the repository, then use the `git add` command. Otherwise the `git checkout` command can be used to discard the changes.

Up Next

Let's stage the change.

Lab 6 Staging Changes

Goals

- Learn how to stage changes for later commits

Add Changes

Now tell git to stage the changes. Check the status

Execute:

```
git add hello.rb
git status
```

You should see …

Output:

```
$ git add hello.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   hello.rb
```

The change to the `hello.rb` file has been staged. This means that git now knows about the change, but the change hasn't been *permanently* recorded in the repository yet. The next commit operation will include the staged changes.

If you decide you *don't* want to commit that change after all, the status command reminds you that the `git reset` command can be used to unstage that change.

Lab 7 Staging and Committing

A separate staging step in git is in line with the philosophy of getting out of the way until you need to deal with source control. You can continue to make changes to your working directory, and then at the point you want to interact with source control, git allows you to record your changes in small commits that record exactly what you did.

For example, suppose you edited three files (`a.rb`, `b.rb`, and `c.rb`). Now you want to commit all the changes, but you want the changes in `a.rb` and `b.rb` to be a single commit, while the changes to `c.rb` are not logically related to the first two files and should be a separate commit.

You could do the following:

```
git add a.rb
git add b.rb
git commit -m "Changes for a and b"
git add c.rb
git commit -m "Unrelated change to c"
```

By separating staging and committing, you have the ability to easily fine tune what goes into each commit.

Lab 8 Committing Changes

Goals

- Learn how to commit changes to the repository

Commit the change

Ok, enough about staging. Let's commit what we have staged to the repository.

When you used `git commit` previously to commit the initial version of the `hello.rb` file to the repository, you included the `-m` flag that gave a comment on the command line. The commit command will allow you to interactively edit a comment for the commit. Let's try that now.

If you omit the `-m` flag from the command line, git will pop you into the editor of your choice. The editor is chosen from the following list (in priority order):

- GIT_EDITOR environment variable
- core.editor configuration setting
- VISUAL environment variable
- EDITOR environment variable

I have the EDITOR variable set to `emacsclient`.

So commit now and check the status.

Execute:

```
git commit
```

You should see the following in your editor:

Output:

```
|
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   hello.rb
#
```

On the first line, enter the comment: "Using ARGV". Save the file and exit the editor. You should see …

Output:

```
git commit
Waiting for Emacs...
[master 569aa96] Using ARGV
 1 files changed, 1 insertions(+), 1 deletions(-)
```

The "Waiting for Emacs…" line comes from the `emacsclient` program which sends the file to a running emacs program and waits for the file to be closed. The rest of the output is the standard commit messages.

Check the status

Finally let's check the status again.

Execute:

```
git status
```

You should see …

Output:

```
$ git status
On branch master
nothing to commit, working directory clean
```

The working directory is clean and ready for you to continue.

Lab 9 Changes, not Files

Goals

- Learn that git works with changes, not files.

Most source control systems work with files. You add a file to source control and the system will track changes to the file from that point on.

Git focuses on the changes to a file rather than the file itself. When you say `git add file`, you are not telling git to add the file to the repository. Rather you are saying that git should make note of the current state of that file to be committed later.

We will attempt to explore that difference in this Lab.

First Change: Allow a default name

Change the "Hello, World" program to have a default value if a command line argument is not supplied.

File: *hello.rb*

```
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

Add this Change

Now add this change to the git's staging area.

Execute:

```
git add hello.rb
```

Second change: Add a comment

Now add a comment to the "Hello, World" program.

File: *hello.rb*

```
# Default is "World"
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

Check the current status

Execute:

```
git status
```

You should see …

Output:

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

        modified:   hello.rb

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   hello.rb
```

Notice how `hello.rb` is listed twice in the status. The first change (adding a default) is staged and is ready to be committed. The second change (adding a comment) is unstaged. If you were to commit right now, the comment would not be saved in the repository.

Let's try that.

Committing

Commit the staged change (the default value), and then recheck the status.

Execute:

```
git commit -m "Added a default value"
git status
```

You should see …

Output:

```
$ git commit -m "Added a default value"
[master 6083cb8] Added a default value
 1 file changed, 3 insertions(+), 1 deletion(-)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
        modified:   hello.rb

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

The status command is telling you that `hello.rb` has unrecorded changes, but is no longer in the staging area.

Add the Second Change

Now add the second change to staging area, then run git status.

Execute:

```
git add .
git status
```

Note: We used the current directory (‘.’) as the file to add. This is a really convenient shortcut for adding in all the changes to the files in the current directory and below. But since it adds everything, it is a *really* good idea to check the status before doing an `add .`, just to make sure you don’t add any file that is not intended.

I wanted you to see the “add .” trick, but we will continue to add explicit files in the rest of this tutorial just to be safe.

You should see …

Output:

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

        modified:   hello.rb
```

Now the second change has been staged and is ready to commit.

Commit the Second Change

Execute:

```
git commit -m "Added a comment"
```

Lab 10 History

Goals

- Learn how to view the history of the project.

Getting a listing of what changes have been made is the function of the `git log` command.

Execute:

```
git log
```

You should see …

Output:

```
$ git log
commit a1189df6a2e597e7d31c2ecf07a9068cf820ce29
Author: Jim Weirich <jim (at) edgecase.com>
Date:   Sat Oct 25 16:25:15 2014 -00

    Added a comment

commit 6083cb81b8d92c360ec07cd6566ad895d15c5db6
Author: Jim Weirich <jim (at) edgecase.com>
Date:   Sat Oct 25 16:25:15 2014 -00

    Added a default value

commit b24f3ffefa338661aadc3861aa24013796ddde40
Author: Jim Weirich <jim (at) edgecase.com>
Date:   Sat Oct 25 16:25:15 2014 -00

    Using ARGV

commit cf466b4441efcc74b0a3db32ecbbb8988fbe39
Author: Jim Weirich <jim (at) edgecase.com>
Date:   Sat Oct 25 16:25:15 2014 -00

    First Commit
```

Here is a list of all four commits that we have made to the repository so far.

One Line Histories

You have a great deal of control over exactly what the `log` command displays. I like the one line format:

Execute:

```
git log --pretty=oneline
```

You should see …

Output:

```
$ git log --pretty=oneline
a1189df6a2e597e7d31c2ecf07a9068cf820ce29 Added a comment
6083cb81b8d92c360ec07cd6566ad895d15c5db6 Added a default value
b24f3ffefa338661aadc3861aa24013796ddde40 Using ARGV
cf466b4441efcc74b0a3db32ecbbb8988fbe39 First Commit
```

Controlling Which Entries are Displayed

There are a lot of options for selecting which entries are displayed in the log. Play around with the following options:

```
git log --pretty=oneline --max-count=2
git log --pretty=oneline --since='5 minutes ago'
git log --pretty=oneline --until='5 minutes ago'
git log --pretty=oneline --author=<your name>
git log --pretty=oneline --all
```

See man git-log for all the details.

Getting Fancy

Here's what I use to review the changes made in the last week. I'll add `--author=jim` if I only want to see changes I made.

```
git log --all --pretty=format:'%h %cd %s (%an)' --since='7 days ago'
```

The Ultimate Log Format

Over time, I've decided that I like the following log format for most of my work.

Execute:

```
git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
```

It looks like this:

Output:

```
$ git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
* a1189df 2014-10-25 | Added a comment (HEAD, master) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

Let's look at it in detail:

- `--pretty="..."` defines the format of the output.
- `%h` is the abbreviated hash of the commit
- `%d` are any decorations on that commit (e.g. branch heads or tags)
- `%ad` is the author date
- `%s` is the comment
- `%an` is the author name
- `--graph` informs git to display the commit tree in an ASCII graph layout
- `--date=short` keeps the date format nice and short

This is a lot to type every time you want to see the log. Fortunately we will learn about git aliases in the next Lab.

Other Tools

Both `gitx` (for Macs) and `gitk` (any platform) are useful in exploring log history.

Lab 11 Aliases

Goals

- Learn how to setup aliases and shortcuts for git commands

Common Aliases

`git status`, `git add`, `git commit`, and `git checkout` are such common commands that it is useful to have abbreviations for them.

Add the following to the .gitconfig file in your $HOME directory.

File: *.gitconfig*

```
[alias]
  co = checkout
  ci = commit
  st = status
  br = branch
  hist = log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
  type = cat-file -t
  dump = cat-file -p
```

We've covered the commit and status commands already. And we just covered the `log` command in the previous Lab. The `checkout` command will be coming up soon.

With these aliases defined in the `.gitconfig` file you can type `git co` wherever you used to have to type `git checkout`. Likewise with `git st` for `git status` and `git ci` for `git commit`. And best of all, `git hist` will allow you to avoid the really long `log` command.

Go ahead and give the new commands a try.

Define the `hist` alias in your `.gitconfig` file

For the most part, I will continue to type out the full command in these instructions. The only exception is that I will use the `hist` alias defined above anytime we need to see the git log output. Make sure you have a `hist` alias setup in your `.gitconfig` file before continuing if you wish to follow along.

`Type` and `Dump`

We've added a few aliases for commands we haven't covered yet. The `git branch` command will be coming up soon. And the `git cat-file` command is useful for exploring git, which we will see in a little while.

Shell Aliases (Optional)

Note: This section is for folks running a posix-like shell. Windows users and other non-posix shell users can feel free to skip to the next Lab.

If your shell supports aliases or shortcuts, then you can add aliases at that level too. Here are the ones I use:

File: *.profile*

```
alias gs='git status '
alias ga='git add '
alias gb='git branch '
alias gc='git commit'
alias gd='git diff'
alias go='git checkout '
alias gk='gitk --all&'
alias gx='gitx --all'

alias got='git '
alias get='git '
```

The `go` abbreviation for `git checkout` is particularly nice. It allows me to type:

```
go <branch>
```

to checkout a particular branch.

And yes, I do mistype `git` as `get` or `got` often enough to create aliases for them.

Lab 12 Getting Old Versions

Goals

- Learn how to checkout any previous snapshot into the working directory.

Going back in history is very easy. The checkout command will copy any snapshot from the repository to the working directory.

Get the hashes for previous versions

Execute:

```
git hist
```

Note: You did remember to define `hist` in your `.gitconfig` file, right? If not, review the Lab on aliases.

Output:

```
$ git hist
* a1189df 2014-10-25 | Added a comment (HEAD, master) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

Examine the log output and find the hash for the first commit. It should be the last line of the `git hist` output. Use that hash code (the first 7 characters are enough) in the command below. Then check the contents of the hello.rb file.

Execute:

```
git checkout <hash>
cat hello.rb
```

Note: The commands given here are Unix commands and work on both Mac and Linux boxes. Unfortunately, Windows users will have to translate to their native commands.

Note: Many commands depend on the hash values in the repository. Since your hash values will vary from mine, whenever you see something like <hash> or <treehash> in the command, substitute in the proper hash value for your repository.

You should see …

Output:

```
$ git checkout cf466b4
```

```
Note: checking out 'cf466b4'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at cf466b4... First Commit
$ cat hello.rb
puts "Hello, World"
```

The output of the `checkout` command explains the situation pretty well. Older versions of git will complain about not being on a local branch. In any case, don't worry about that for now.

Notice the contents of the hello.rb file are the original contents.

Return the latest version in the master branch

Execute:

```
git checkout master
cat hello.rb
```

You should see …

Output:

```
$ git checkout master
Previous HEAD position was cf466b4... First Commit
Switched to branch 'master'
$ cat hello.rb
# Default is "World"
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

'master' is the name of the default branch. By checking out a branch by name, you go to the latest version of that branch.

Lab 13 Tagging versions

Goals

- Learn how to tag commits with names for future reference

Let's call the current version of the hello program version 1 (v1).

Tagging version 1

Execute:

```
git tag v1
```

Now you can refer to the current version of the program as v1.

Tagging Previous Versions

Let's tag the version immediately prior to the current version v1-beta. First we need to checkout the previous version. Rather than lookup up the hash, we will use the `^` notation to indicate "the parent of v1".

If the `v1^` notation gives you any trouble, you can also try `v1~1`, which will reference the same version. This notation means "the first ancestor of v1".

Execute:

```
git checkout v1^
cat hello.rb
```

Output:

```
$ git checkout v1^
Note: checking out 'v1^'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 6083cb8... Added a default value
$ cat hello.rb
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

See, this is the version with the default value *before* we added the comment. Let's make this v1-beta.

Execute:

```
git tag v1-beta
```

Checking Out by Tag Name

Now try going back and forth between the two tagged versions.

Execute:

```
git checkout v1
git checkout v1-beta
```

Output:

```
$ git checkout v1
Previous HEAD position was 6083cb8... Added a default value
HEAD is now at a1189df... Added a comment
$ git checkout v1-beta
Previous HEAD position was a1189df... Added a comment
HEAD is now at 6083cb8... Added a default value
```

Viewing Tags using the `tag` command

You can see what tags are avaiLable using the `git tag` command.

Execute:

```
git tag
```

Output:

```
$ git tag
v1
v1-beta
```

Viewing Tags in the Logs *05*

You can also check for tags in the log.

Execute:

```
git hist master --all
```

Output:

```
$ git hist master --all
* a1189df 2014-10-25 | Added a comment (tag: v1, master) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (HEAD, tag: v1-beta) [Jim
Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

You can see both tags (`v1` and `v1-beta`) listed in the log output, along with the branch name (`master`). Also `HEAD` shows you the currently checked out commit (which is `v1-beta` at the moment).

Lab 14 Undoing Local Changes (before staging)

Goals

- Learn how to revert changes in the working directory

Checkout Master

Make sure you are on the latest commit in master before proceeding.

Execute:

```
git checkout master
```

Change hello.rb

Sometimes you have modified a file in your local working directory and you wish to just revert to what has already been committed. The checkout command will handle that.

Change hello.rb to have a bad comment.

File: *hello.rb*

```
# This is a bad comment.  We want to revert it.
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

Check the Status

First, check the status of the working directory.

Execute:

```
git status
```

Output:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   hello.rb

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

We see that the `hello.rb` file has been modified, but hasn't been staged yet.

Revert the changes in the working directory

Use the `checkout` command to checkout the repository's version of the `hello.rb` file.

Execute:

```
git checkout hello.rb
git status
cat hello.rb
```

Output:

```
$ git checkout hello.rb
$ git status
On branch master
nothing to commit, working directory clean
$ cat hello.rb
# Default is "World"
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

The status command shows us that there are no outstanding changes in the working directory. And the "bad comment" is no longer part of the file contents.

Lab 15 Undoing Staged Changes (before committing)

Goals

- Learn how to revert changes that have been staged

Change the file and stage the change

Modify the `hello.rb` file to have a bad comment

File: *hello.rb*

```
# This is an unwanted but staged comment
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

And then go ahead and stage it.

Execute:

```
git add hello.rb
```

Check the Status

Check the status of your unwanted change.

Execute:

```
git status
```

Output:

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

        modified:   hello.rb
```

The status output shows that the change has been staged and is ready to be committed.

Reset the Staging Area

Fortunately the status output tells us exactly what we need to do to unstage the change.

Execute:

```
git reset HEAD hello.rb
```

Output:

```
$ git reset HEAD hello.rb
Unstaged changes after reset:
M       hello.rb
```

The `reset` command resets the staging area to be whatever is in HEAD. This clears the staging area of the change we just staged.

The `reset` command (by default) doesn't change the working directory. So the working directory still has the unwanted comment in it. We can use the `checkout` command of the previous Lab to remove the unwanted change from the working directory.

Checkout the Committed Version

Execute:

```
git checkout hello.rb
git status
```

Output:

```
$ git status
On branch master
nothing to commit, working directory clean
```

And our working directory is clean once again.

Lab 16 Undoing Committed Changes

Goals

- Learn how to revert changes that have been committed to a local repository.

Undoing Commits

Sometimes you realized that a change that you have already committed was not correct and you wish to undo that commit. There are several ways of handling that issue, and the way we are going to use in this Lab is always safe.

Essentially we will undo the commit by creating a new commit that reverses the unwanted changes.

Change the file and commit it.

Change the `hello.rb` file to the following.

File: *hello.rb*

```
# This is an unwanted but committed change
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

Execute:

```
git add hello.rb
git commit -m "Oops, we didn't want this commit"
```

Create a Reverting Commit

To undo a committed change, we need to generate a commit that removes the changes introduced by our unwanted commit.

Execute:

```
git revert HEAD
```

This will pop you into the editor. You can edit the default commit message or leave it as is. Save and close the file. You should see …

Output:

```
$ git revert HEAD --no-edit
[master af5da8a] Revert "Oops, we didn't want this commit"
```

```
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Since we were undoing the very last commit we made, we were able to use `HEAD` as the argument to revert. We can revert any arbitrary commit earlier in history by simply specifying its hash value.

Note: The `--no-edit` in the output can be ignored. It was necessary to generate the output without opening the editor.

Check the log

Checking the log shows both the unwanted and the reverting commits in our repository.

Execute:

```
git hist
```

Output:

```
$ git hist
* af5da8a 2014-10-25 | Revert "Oops, we didn't want this commit" (HEAD,
master) [Jim Weirich]
* 47d54 2014-10-25 | Oops, we didn't want this commit [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

This technique will work with any commit (although you may have to resolve conflicts). It is safe to use even on branches that are publicly shared on remote repositories.

Up Next

Next, let's look at a technique that can be used to remove the most recent commits from the repository history.

Lab 17 Removing Commits from a Branch

Goals

- Learn how to remove the most recent commits from a branch

The `revert` command of the previous section is a powerful command that lets us undo the effects of any commit in the repository. However, both the original commit and the "undoing" commit are visible in the branch history (using the `git log` command).

Often we make a commit and immediately realize that it was a mistake. It would be nice to have a "take back" command that would allow us to pretend that the incorrect commit never happened. The "take back" command would even prevent the bad commit from showing up the `git log` history. It would be as if the bad commit never happened.

The `reset` command

We've already seen the `reset` command and have used it to set the staging area to be consistent with a given commit (we used the HEAD commit in our previous Lab).

When given a commit reference (i.e. a hash, branch or tag name), the `reset` command will …

1. Rewrite the current branch to point to the specified commit
2. Optionally reset the staging area to match the specified commit
3. Optionally reset the working directory to match the specified commit

Check Our History

Let's do a quick check of our commit history.

Execute:

```
git hist
```

Output:

```
$ git hist
* af5da8a 2014-10-25 | Revert "Oops, we didn't want this commit" (HEAD,
master) [Jim Weirich]
* 47d54 2014-10-25 | Oops, we didn't want this commit [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

We see that we have an "Oops" commit and a "Revert Oops" commit as the last two commits made in this branch. Let's remove them using reset.

First, Mark this Branch

But before we remove the commits, let's mark the latest commit with a tag so we can find it again.

Execute:

```
git tag oops
```

Reset to Before Oops

Looking at the log history (above), we see that the commit tagged 'v1' is the commit right before the bad commit. Let's reset the branch to that point. Since that branch is tagged, we can use the tag name in the reset command (if it wasn't tagged, we could just use the hash value).

Execute:

```
git reset --hard v1
git hist
```

Output:

```
$ git reset --hard v1
HEAD is now at a1189df Added a comment
$ git hist
* a1189df 2014-10-25 | Added a comment (HEAD, tag: v1, master) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

Our master branch now points to the v1 commit and the Oops commit and the Revert Oops commit are no longer in the branch. The `--hard` parameter indicates that the working directory should be updated to be consistent with the new branch head.

Nothing is Ever Lost

But what happened to the bad commits? It turns out that the commits are still in the repository. In fact, we can still reference them. Remember that at the beginning of this Lab we tagged the reverting commit with the tag "oops". Let's look at *all* the commits.

Execute:

```
git hist --all
```

Output:

```
$ git hist --all
* af5da8a 2014-10-25 | Revert "Oops, we didn't want this commit" (tag: oops)
[Jim Weirich]
```

```
* 47d54 2014-10-25 | Oops, we didn't want this commit [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (HEAD, tag: v1, master) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

Here we see that the bad commits haven't disappeared. They are still in the repository. It's just that they are no longer listed in the master branch. If we hadn't tagged them, they would still be in the repository, but there would be no way to reference them other than using their hash names. Commits that are unreferenced remain in the repository until the system runs the garbage collection software.

Dangers of Reset *06*

Resets on local branches are generally safe. Any "accidents" can usually be recovered from by just resetting again with the desired commit.

However, if the branch is shared on remote repositories, resetting can confuse other users sharing the branch.

Lab 18 Remove the oops tag

Goals

- Remove the oops tag (housekeeping)

Removing tag oops

The oops tag has served its purpose. Let's remove it and allow the commits it referenced to be garbage collected.

Execute:

```
git tag -d oops
git hist --all
```

Output:

```
$ git tag -d oops
Deleted tag 'oops' (was af5da8a)
$ git hist --all
* a1189df 2014-10-25 | Added a comment (HEAD, tag: v1, master) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

The oops tag is no longer listed in the repository.

Lab 19 Amending Commits

Goals

- Learn how to amend an existing commit

Change the program then commit

Add an author comment to the program.

File: *hello.rb*

```
# Default is World
# Author: Jim Weirich
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

Execute:

```
git add hello.rb
git commit -m "Add an author comment"
```

Oops, Should have an Email

After you make the commit, you realize that any good author comment should have an email included. Update the hello program to include an email.

File: *hello.rb*

```
# Default is World
# Author: Jim Weirich (jim@somewhere.com)
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

Amend the Previous Commit

We really don't want a separate commit for just the email. Let's amend the previous commit to include the email change.

Execute:

```
git add hello.rb
git commit --amend -m "Add an author/email comment"
```

Output:

```
$ git add hello.rb
$ git commit --amend -m "Add an author/email comment"
[master bfd1408] Add an author/email comment
 1 file changed, 2 insertions(+), 1 deletion(-)
```

Review the History

Execute:

```
git hist
```

Output:

```
$ git hist
* bfd1408 2014-10-25 | Add an author/email comment (HEAD, master) [Jim
Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

We can see the original “author” commit is now gone, and it is replaced by the “author/email” commit. You can achieve the same effect by resetting the branch back one commit and then recommitting the new changes.

Lab 20 Moving Files

Goals

- Learn how to move a file within a repository.

Move the hello.rb file into a lib directory.

We are now going to build up the structure of our little repository. Let's move the program into a lib directory.

Execute:

```
mkdir lib
git mv hello.rb lib
git status
```

Output:

```
$ mkdir lib
$ git mv hello.rb lib
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    hello.rb -> lib/hello.rb
```

By using git to do the move, we inform git of 2 things

1. That the file `hello.rb` has been deleted.
2. The file `lib/hello.rb` has been created.

Both of these bits of information are immediately staged and ready to be committed. The git status command reports that the file has been moved.

Another way of moving files

One of the nice things about git is that you can forget about source control until the point you are ready to start committing code. What would happen if we used the operating system command to move the file instead of the git command?

It turns out the following set of commands is identical to what we just did. It's a bit more work, but the result is the same.

We could have done:

```
mkdir lib
mv hello.rb lib
git add lib/hello.rb
git rm hello.rb
```

Commit the new directory

Let’s commit this move.

Execute:

```
git commit -m "Moved hello.rb to lib"
```

Lab 21 More Structure

Goals

- Add another file to our repository

Now add a Rakefile

This Lab assumes you have installed **rake**. Please do that before continuing. Check for your specific Operating System. Otherwise execute:

Execute:

```
gem install rake
```

Let's add a Rakefile to our repository. The following one will do nicely.

File: *Rakefile*

```
#!/usr/bin/ruby -wKU

task :default => :run

task :run do
  require './lib/hello'
end
```

Add and commit the change.

Execute:

```
git add Rakefile
git commit -m "Added a Rakefile."
```

You should be able to use Rake to run your hello program now.

Execute:

```
rake
```

Output:

```
$ rake
Hello, World!
```

Lab 22 Git Internals: The .git directory

Goals

- Learn about the structure of the `.git` directory

The `.git` Directory

Time to do some exploring. First, from the root of your project directory…

Execute:

```
ls -C .git
```

Output:

```
$ ls -C .git
COMMIT_EDITMSG  ORIG_HEAD       index           objects
HEAD            config          logs            refs
```

This is the magic directory where all the git "stuff" is stored. Let's peek in the objects directory.

The Object Store

Execute:

```
ls -C .git/objects
```

Output:

```
$ ls -C .git/objects
09      24      43      59      6b      9c      b0      bf      e4      pack
11      27      47      60      78      a1      b2      c4      e7
14      28      4b      69      97      af      b5      cf      info
```

You should see a bunch of directories with 2 letter names. The directory names are the first two letters of the sha1 hash of the object stored in git.

Deeper into the Object Store

Execute:

```
ls -C .git/objects/<dir>
```

Output:

```
$ ls -C .git/objects/09
```

```
6b74c56bfc6b40e754fc0725b8c70b28b91e  9fb6f9d3a1feb32fcac22354c4d0e8a182c1
```

Look in one of the two-letter directories. You should see some files with 38-character names. These are the files that contain the objects stored in git. These files are compressed and encoded, so looking at their contents directly won't be very helpful, but we will take a closer look in a bit.

Config File

Execute:

```
cat .git/config
```

Output:

```
$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[user]
        name = Jim Weirich
        email = jim (at) edgecase.com
```

This is a project-specific configuration file. Config entries in here will override the config entries in the `.gitconfig` file in your home directory, at least for this project.

Branches and Tags *05*

Execute:

```
ls .git/refs
ls .git/refs/heads
ls .git/refs/tags
cat .git/refs/tags/v1
```

Output:

```
$ ls .git/refs
heads
tags
$ ls .git/refs/heads
master
$ ls .git/refs/tags
v1
v1-beta
$ cat .git/refs/tags/v1
a1189df6a2e597e7d31c2ecf07a9068cf820ce29
```

You should recognize the files in the tags subdirectory. Each file corresponds to a tag you created with the `git tag` command earlier. Its content is just the hash of the commit tied to the tag.

The heads directory is similar, but is used for branches rather than tags. We only have one branch at the moment, so all you will see is master in this directory.

The HEAD File

Execute:

```
cat .git/HEAD
```

Output:

```
$ cat .git/HEAD
ref: refs/heads/master
```

The HEAD file contains a reference to the current branch. It should be a reference to master at this point.

Lab 23 Git Internals: Working directly with Git Objects

Goals

- Explore the structure of the object store
- Learn how to use the SHA1 hashes to find content in the repository

Now let's use some tools to probe git objects directly.

Finding the Latest Commit

Execute:

```
git hist --max-count=1
```

This should show the latest commit made in the repository. The SHA1 hash on your system is probably different that what is on mine, but you should see something like this.

Output:

```
$ git hist --max-count=1
* 14ba469 2014-10-25 | Added a Rakefile. (HEAD, master) [Jim Weirich]
```

Dumping the Latest Commit

Using the SHA1 hash from the commit listed above …

Execute:

```
git cat-file -t <hash>
git cat-file -p <hash>
```

Here's my output …

Output:

```
$ git cat-file -t 14ba469
commit
$ git cat-file -p 14ba469
tree 096b74c56bfc6b40e754fc0725b8c70b28b91e
parent 4b249b37a5bcc71d622344179aebfd7afc1ae5
author Jim Weirich <jim (at) edgecase.com> 1414268718 -00
committer Jim Weirich <jim (at) edgecase.com> 1414268718 -00

Added a Rakefile.
```

NOTE: If you defined the 'type' and 'dump' aliases from the aliases Lab, then you can type `git type` and `git dump` rather than the longer cat-file commands (which I never remember).

This is the dump of the commit object that is at the head of the master branch. It looks a lot like the commit object from the presentation earlier.

Finding the Tree

We can dump the directory tree referenced in the commit. This should be a description of the (top level) files in our project (for that commit). Use the SHA1 hash from the "tree" line listed above.

Execute:

```
git cat-file -p <treehash>
```

Here's what my tree looks like…

Output:

```
$ git cat-file -p 096b74c
100644 blob 28e0e9d6ea7e25f35ec64a43f569b550e8386f90  Rakefile
0000 tree e46f374f5b36c6ffb3e9e922b794f754d795        lib
```

Yep, I see the Rakefile and the lib directory.

Dumping the lib directory

Execute:

```
git cat-file -p <libhash>
```

Output:

```
$ git cat-file -p e46f374
100644 blob c45f26b6fdc7db6ba779fc4c385d9d24fc12cf72  hello.rb
```

There's the `hello.rb` file.

Dumping the `hello.rb` file

Execute:

```
git cat-file -p <rbhash>
```

Output:

```
$ git cat-file -p c45f26b
```

```
# Default is World
# Author: Jim Weirich (jim@somewhere.com)
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

There you have it. We've dumped commit objects, tree objects and blob objects directly from the git repository. That's all there is to it, blobs, trees and commits.

Explore On You Own

Explore the git repo manually on your own. See if you can find the original hello.rb file from the very first commit by manually following the SHA1 hash references starting in the latest commit.

Lab 24 Creating a Branch

Goals

- Learn how to create a local branch in a repository

It's time to do a major rewrite of the hello world functionality. Since this might take awhile, you'll want to put these changes into a separate branch to isolate them from changes in master.

Create a Branch

Let's call our new branch 'greet'.

Execute:

```
git checkout -b greet
git status
```

NOTE: `git checkout -b <branchname>` is a shortcut for `git branch <branchname>` followed by a `git checkout <branchname>`.

Notice that the git status command reports that you are on the 'greet' branch.

Changes for Greet: Add a Greeter class.

File: *lib/greeter.rb*

```
class Greeter
  def initialize(who)
    @who = who
  end
  def greet
    "Hello, #{@who}"
  end
end
```

Execute:

```
git add lib/greeter.rb
git commit -m "Added greeter class"
```

Changes for Greet: Modify the main program

Update the hello.rb file to use greeter

File: *lib/hello.rb*

```
require 'greeter'

# Default is World
name = ARGV.first || "World"

greeter = Greeter.new(name)
puts greeter.greet
```

Execute:

```
git add lib/hello.rb
git commit -m "Hello uses Greeter"
```

Changes for Greet: Update the Rakefile

Update the Rakefile to use an external ruby process

File: *Rakefile*

```
#!/usr/bin/ruby -wKU

task :default => :run

task :run do
  ruby '-Ilib', 'lib/hello.rb'
end
```

Execute:

```
git add Rakefile
git commit -m "Updated Rakefile"
```

Up Next *05*

We now have a new branch called **greet** with 3 new commits on it. Next we will learn how to navigate and switch between branches.

Lab 25 Navigating Branches

Goals

- Learn how to navigate between the branches of a repository

You now have two branches in your project:

Execute:

```
git hist --all
```

Output:

```
$ git hist --all
* 12414e9 2014-10-25 | Updated Rakefile (HEAD, greet) [Jim Weirich]
* 43ba308 2014-10-25 | Hello uses Greeter [Jim Weirich]
* acd8535 2014-10-25 | Added greeter class [Jim Weirich]
* 14ba469 2014-10-25 | Added a Rakefile. (master) [Jim Weirich]
* 4b249b3 2014-10-25 | Moved hello.rb to lib [Jim Weirich]
* bfd1408 2014-10-25 | Add an author/email comment [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

Switch to the Master Branch

Just use the `git checkout` command to switch between branches.

Execute:

```
git checkout master
cat lib/hello.rb
```

Output:

```
$ git checkout master
Switched to branch 'master'
$ cat lib/hello.rb
# Default is World
# Author: Jim Weirich (jim@somewhere.com)
name = ARGV.first || "World"

puts "Hello, #{name}!"
```

You are now on the master branch. You can tell because the hello.rb file doesn't use the `Greeter` class.

Switch Back to the Greet Branch.

Execute:

```
git checkout greet
cat lib/hello.rb
```

Output:

```
$ git checkout greet
Switched to branch 'greet'
$ cat lib/hello.rb
require 'greeter'

# Default is World
name = ARGV.first || "World"

greeter = Greeter.new(name)
puts greeter.greet
```

The contents of the `lib/hello.rb` confirms we are back on the **greet** branch.

Lab 26 Changes in Master

Goals

- Learning how to deal with multiple branches with different (and possibly conflicting) changes.

While you were changing the greet branch, someone else decided to update the master branch. They added a README.

Switch to the master branch.

Execute:

```
git checkout master
```

Create the README.

File: *README*

```
This is the Hello World example from the git tutorial.
```

Commit the README to master.

Execute:

```
git add README
git commit -m "Added README"
```

Lab 27 Viewing Diverging Branches

Goals

- Learn how to view diverging branches in a repository.

View the Current Branches

We now have two diverging branches in the repository. Use the following log command to view the branches and how they diverge.

Execute:

```
git hist --all
```

Output:

```
$ git hist --all
* 12414e9 2014-10-25 | Updated Rakefile (greet) [Jim Weirich]
* 43ba308 2014-10-25 | Hello uses Greeter [Jim Weirich]
* acd8535 2014-10-25 | Added greeter class [Jim Weirich]
| * 6fd9eff 2014-10-25 | Added README (HEAD, master) [Jim Weirich]
|/
* 14ba469 2014-10-25 | Added a Rakefile. [Jim Weirich]
* 4b249b3 2014-10-25 | Moved hello.rb to lib [Jim Weirich]
* bfd1408 2014-10-25 | Add an author/email comment [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

Here is our first chance to see the `--graph` option on `git hist` in action. Adding the `--graph` option to `git log` causes it to draw the commit tree using simple ASCII characters. We can see both branches (greet and master), and that the master branch is the current HEAD. The common ancestor to both branches is the "Added a Rakefile" branch.

The `--all` flag makes sure that we see all the branches. The default is to show only the current branch.

Lab 28 Merging

Goals

- Learn how to merge two diverging branches to bring the changes back into a single branch.

Merge the branches

Merging brings the changes in two branches together. Let's go back to the greet branch and merge master onto greet.

Execute:

```
git checkout greet
git merge master
git hist --all
```

Output:

```
$ git checkout greet
Switched to branch 'greet'
$ git merge master
Merge made by the 'recursive' strategy.
 README | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 README
$ git hist --all
*   9d0aceb 2014-10-25 | Merge branch 'master' into greet (HEAD, greet) [Jim
Weirich]
|\
| * 6fd9eff 2014-10-25 | Added README (master) [Jim Weirich]
* | 12414e9 2014-10-25 | Updated Rakefile [Jim Weirich]
* | 43ba308 2014-10-25 | Hello uses Greeter [Jim Weirich]
* | acd8535 2014-10-25 | Added greeter class [Jim Weirich]
|/
* 14ba469 2014-10-25 | Added a Rakefile. [Jim Weirich]
* 4b249b3 2014-10-25 | Moved hello.rb to lib [Jim Weirich]
* bfd1408 2014-10-25 | Add an author/email comment [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

By merging master into your greet branch periodically, you can pick up any changes to master and keep your changes in greet compatible with changes in the mainline.

However, it does produce ugly commit graphs. Later we will look at the option of rebasing rather than merging.

Up Next

But first, what if the changes in master conflict with the changes in greet?

Lab 29 Creating a Conflict

Goals

- Create a conflicting change in the master branch.

Switch back to master and create a conflict

Switch back to the master branch and make this change:

Execute:

```
git checkout master
```

File: *lib/hello.rb*

```
puts "What's your name"
my_name = gets.strip

puts "Hello, #{my_name}!"
```

Execute:

```
git add lib/hello.rb
git commit -m "Made interactive"
```

View the Branches

Execute:

```
git hist --all
```

Output:

```
$ git hist --all
*   9d0aceb 2014-10-25 | Merge branch 'master' into greet (greet) [Jim
Weirich]
|\
* | 12414e9 2014-10-25 | Updated Rakefile [Jim Weirich]
* | 43ba308 2014-10-25 | Hello uses Greeter [Jim Weirich]
* | acd8535 2014-10-25 | Added greeter class [Jim Weirich]
| | * 8697f14 2014-10-25 | Made interactive (HEAD, master) [Jim Weirich]
| |/
| * 6fd9eff 2014-10-25 | Added README [Jim Weirich]
|/
* 14ba469 2014-10-25 | Added a Rakefile. [Jim Weirich]
* 4b249b3 2014-10-25 | Moved hello.rb to lib [Jim Weirich]
* bfd1408 2014-10-25 | Add an author/email comment [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
```

```
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

Master at commit “Added README” has been merged to the greet branch, but there is now an additional commit on master that has not been merged back to greet.

Up Next

The latest change in master conflicts with some existing changes in greet. Next we will resolve those changes.

Lab 30 Resolving Conflicts

Goals

- Learn how to handle conflicts during a merge

Merge master to greet

Now go back to the greet branch and try to merge the new master.

Execute:

```
git checkout greet
git merge master
```

Output:

```
$ git checkout greet
Switched to branch 'greet'
$ git merge master
Auto-merging lib/hello.rb
CONFLICT (content): Merge conflict in lib/hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

If you open lib/hello.rb, you will see:

File: *lib/hello.rb*

```
<<<<<<< HEAD
require 'greeter'

# Default is World
name = ARGV.first || "World"

greeter = Greeter.new(name)
puts greeter.greet
=======
# Default is World

puts "What's your name"
my_name = gets.strip

puts "Hello, #{my_name}!"
>>>>>>> master
```

The first section is the version on the head of the current branch (greet). The second section is the version on the master branch.

Fix the Conflict

You need to manually resolve the conflict. Modify `lib/hello.rb` to be the following.

File: *lib/hello.rb*

```
require 'greeter'

puts "What's your name"
my_name = gets.strip

greeter = Greeter.new(my_name)
puts greeter.greet
```

Commit the Conflict Resolution

Execute:

```
git add lib/hello.rb
git commit -m "Merged master fixed conflict."
```

Output:

```
$ git add lib/hello.rb
$ git commit -m "Merged master fixed conflict."
[greet 885ca6f] Merged master fixed conflict.
```

Advanced Merging

git doesn't provide any graphical merge tools, but it will gladly work with any third party merge tool you wish to use. See http://onestepback.org/index.cgi/Tech/Git/UsingP4MergeWithGit.red for a description of using the Perforce merge tool with git.

Lab 31 Rebasing VS Merging

Goals

- Learn the differences between rebasing and merging.

Discussion

Let's explore the differences between merging and rebasing. In order to do so, we need to rewind the repository back in time before the first merge, and then redo the same steps, but using rebasing rather than merging.

We will make use the of the `reset` command to wind the branches back in time.

Lab 32 Resetting the Greet Branch

Goals

- Reset the greet branch to the point before the first merge.

Reset the greet branch

Let's go back in time on the greet branch to the point *before* we merged master onto it. We can **reset** a branch to any commit we want. Essentially this is modifying the branch pointer to point to anywhere in the commit tree.

In this case we want to back greet up to the point prior to the merge with master. We need to find the last commit before the merge.

Execute:

```
git checkout greet
git hist
```

Output:

```
$ git checkout greet
Already on 'greet'
$ git hist
*   885ca6f 2014-10-25 | Merged master fixed conflict. (HEAD, greet) [Jim
Weirich]
|\
| * 8697f14 2014-10-25 | Made interactive (master) [Jim Weirich]
* |   9d0aceb 2014-10-25 | Merge branch 'master' into greet [Jim Weirich]
|\ \
| |/
| * 6fd9eff 2014-10-25 | Added README [Jim Weirich]
* | 12414e9 2014-10-25 | Updated Rakefile [Jim Weirich]
* | 43ba308 2014-10-25 | Hello uses Greeter [Jim Weirich]
* | acd8535 2014-10-25 | Added greeter class [Jim Weirich]
|/
* 14ba469 2014-10-25 | Added a Rakefile. [Jim Weirich]
* 4b249b3 2014-10-25 | Moved hello.rb to lib [Jim Weirich]
* bfd1408 2014-10-25 | Add an author/email comment [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

That's a bit hard to read, but looking at the data we see that the "Updated Rakefile" commit was the last commit on the greet branch before merging. Let's reset the greet branch to that commit.

Execute:

```
git reset --hard <hash>
```

Output:

```
$ git reset --hard 12414e9
HEAD is now at 12414e9 Updated Rakefile
```

Check the branch.

Look at the log for the greet branch. We no longer have the merge commits in its history.

Execute:

```
git hist --all
```

Output:

```
$ git hist --all
* 12414e9 2014-10-25 | Updated Rakefile (HEAD, greet) [Jim Weirich]
* 43ba308 2014-10-25 | Hello uses Greeter [Jim Weirich]
* acd8535 2014-10-25 | Added greeter class [Jim Weirich]
| * 8697f14 2014-10-25 | Made interactive (master) [Jim Weirich]
| * 6fd9eff 2014-10-25 | Added README [Jim Weirich]
|/
* 14ba469 2014-10-25 | Added a Rakefile. [Jim Weirich]
* 4b249b3 2014-10-25 | Moved hello.rb to lib [Jim Weirich]
* bfd1408 2014-10-25 | Add an author/email comment [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

Lab 33 Resetting the Master Branch

Goals

- Reset the master branch to the point before the conflicting commit.

Reset the master branch

When we added the interactive mode to the master branch, we made a change that conflicted with changes in the greet branch. Let's rewind the master branch to a point before the conflicting change. This allows us to demonstrate the rebase command without worrying about conflicts.

Execute:

```
git checkout master
git hist
```

Output:

```
$ git hist
* 8697f14 2014-10-25 | Made interactive (HEAD, master) [Jim Weirich]
* 6fd9eff 2014-10-25 | Added README [Jim Weirich]
* 14ba469 2014-10-25 | Added a Rakefile. [Jim Weirich]
* 4b249b3 2014-10-25 | Moved hello.rb to lib [Jim Weirich]
* bfd1408 2014-10-25 | Add an author/email comment [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

The 'Added README' commit is the one directly before the conflicting interactive mode. We will reset the master branch to 'Added README' branch.

Execute:

```
git reset --hard <hash>
git hist --all
```

Review the log. It should look like the repository has been wound back in time to the point before we merged anything.

Output:

```
$ git hist --all
* 12414e9 2014-10-25 | Updated Rakefile (greet) [Jim Weirich]
* 43ba308 2014-10-25 | Hello uses Greeter [Jim Weirich]
* acd8535 2014-10-25 | Added greeter class [Jim Weirich]
| * 6fd9eff 2014-10-25 | Added README (HEAD, master) [Jim Weirich]
|/
```

```
* 14ba469 2014-10-25 | Added a Rakefile. [Jim Weirich]
* 4b249b3 2014-10-25 | Moved hello.rb to lib [Jim Weirich]
* bfd1408 2014-10-25 | Add an author/email comment [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

Lab 34 Rebasing

Goals

- Use the rebase command rather than the merge command.

Ok, we are back in time before the first merge and we want to get the changes in master into our greet branch.

This time we will use the rebase command instead of the merge command to bring in the changes from the master branch.

Execute:

```
git checkout greet
git rebase master
git hist
```

Output:

```
$ go greet
Switched to branch 'greet'
$
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added Greeter class
Applying: hello uses Greeter
Applying: updated Rakefile
$
$ git hist
* b8188 2014-10-25 | Updated Rakefile (HEAD, greet) [Jim Weirich]
* b7964 2014-10-25 | Hello uses Greeter [Jim Weirich]
* 0a576ce 2014-10-25 | Added greeter class [Jim Weirich]
* 6fd9eff 2014-10-25 | Added README (master) [Jim Weirich]
* 14ba469 2014-10-25 | Added a Rakefile. [Jim Weirich]
* 4b249b3 2014-10-25 | Moved hello.rb to lib [Jim Weirich]
* bfd1408 2014-10-25 | Add an author/email comment [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

Merge VS Rebase

The final result of the rebase is very similar to the merge. The greet branch now contains all of its changes, as well as all the changes from the master branch. However, the commit tree is quite different. The commit tree for the greet branch has been rewritten so that the master branch is a part of the commit history. This leaves the chain of commits linear and much easier to read.

When to Rebase, When to Merge?

Don't use rebase …

1. If the branch is public and shared with others. Rewriting publicly shared branches will tend to screw up other members of the team.
2. When the *exact* history of the commit branch is important (since rebase rewrites the commit history).

Given the above guidelines, I tend to use rebase for short-lived, local branches and merge for branches in the public repository.

Lab 35 Merging Back to Master

Goals

- We've kept our greet branch up to date with master (via rebase), now let's merge the greet changes back into the master branch.

Merge greet into master

Execute:

```
git checkout master
git merge greet
```

Output:

```
$ git checkout master
Switched to branch 'master'
$
$ git merge greet
Updating 6fd9eff..b8188
Fast-forward
 Rakefile       | 2 +-
 lib/greeter.rb | 8 ++++++++
 lib/hello.rb   | 6 ++++--
 3 files changed, 13 insertions(+), 3 deletions(-)
 create mode 100644 lib/greeter.rb
```

Because the head of master is a direct ancestor of the head of the greet branch, git is able to do a fast-forward merge. When fast-forwarding, the branch pointer is simply moved forward to point to the same commit as the greeter branch.

There will never be conflicts in a fast-forward merge.

Review the logs

Execute:

```
git hist
```

Output:

```
$ git hist
* b8188 2014-10-25 | Updated Rakefile (HEAD, master, greet) [Jim Weirich]
* b7964 2014-10-25 | Hello uses Greeter [Jim Weirich]
* 0a576ce 2014-10-25 | Added greeter class [Jim Weirich]
* 6fd9eff 2014-10-25 | Added README [Jim Weirich]
* 14ba469 2014-10-25 | Added a Rakefile. [Jim Weirich]
* 4b249b3 2014-10-25 | Moved hello.rb to lib [Jim Weirich]
```

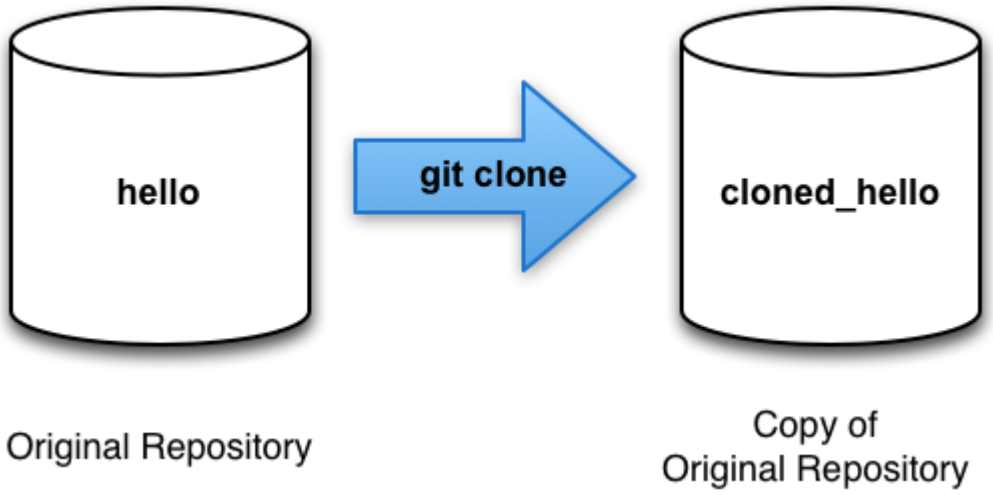
```
* bfd1408 2014-10-25 | Add an author/email comment [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

The greet and master branches are now identical.

Lab 36 Multiple Repositories

Up to this point we have been working with a single git repository. However, git excels at working with multiple repositories. These extra repositories may be stored locally, or may be accessed across a network connection.

In the next section we will create a new repository called “cloned_hello”. We will show how to move changes from one repository to another, and how to handle conflicts when they arise from between two repositories.



For now, we will be working with local repositories (i.e. repositories stored on your local hard disk), however most of the things learned in this section will apply to multiple repositories whether they are stored locally or remotely over a network.

NOTE: We are going be making changes to both copies of our repositories. Make sure you pay attention to which repository you are in at each step of the following Labs.

Lab 37 Cloning Repositories

Goals

- Learn how to make copies of repositories.

Go to the work directory

Go to the working directory and make a clone of your hello repository.

Execute:

```
cd ..
pwd
ls
```

NOTE: Now in the work directory.

Output:

```
$ cd ..
$ pwd
/Users/tonywok/src/git_immersion/auto
$ ls
hello
```

At this point you should be in your "work" directory. There should be a single repository here named "hello".

Create a clone of the hello repository

Let's make a clone of the repository.

Execute:

```
git clone hello cloned_hello
ls
```

Output:

```
$ git clone hello cloned_hello
Cloning into 'cloned_hello'...
warning: templates not found /Users/tonywok/.git_template
done.
$ ls
cloned_hello
hello
```

There should now be two repositories in your work directory: the original “hello” repository and the newly cloned “cloned_hello” repository.

Lab 38 Review the Cloned Repository

Goals

- Learn about branches on remote repositories.

Look at the cloned repository

Let's take a look at the cloned repository.

Execute:

```
cd cloned_hello
ls
```

Output:

```
$ cd cloned_hello
$ ls
README
Rakefile
lib
```

You should see a list of all the files in the top level of the original repository (`README`, `Rakefile` and `lib`).

Review the Repository History

Execute:

```
git hist --all
```

Output:

```
$ git hist --all
* b8188 2014-10-25 | Updated Rakefile (HEAD, origin/master, origin/greet,
origin/HEAD, master) [Jim Weirich]
* b7964 2014-10-25 | Hello uses Greeter [Jim Weirich]
* 0a576ce 2014-10-25 | Added greeter class [Jim Weirich]
* 6fd9eff 2014-10-25 | Added README [Jim Weirich]
* 14ba469 2014-10-25 | Added a Rakefile. [Jim Weirich]
* 4b249b3 2014-10-25 | Moved hello.rb to lib [Jim Weirich]
* bfd1408 2014-10-25 | Add an author/email comment [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

You should now see a list of the all the commits in the new repository, and it should (more or less) match the history of commits in the original repository. The only difference should be in the names of the branches.

Remote branches

You should see a **master** branch (along with **HEAD**) in the history list. But you will also have number of strangely named branches (**origin/master**, **origin/greet** and **origin/HEAD**). We'll talk about them in a bit.

Lab 39 What is Origin?

Goals

- Learn about naming remote repositories.

Execute:

```
git remote
```

Output:

```
$ git remote
origin
```

We see that the cloned repository knows about a remote repository named origin. Let's see if we can get more information about origin:

Execute:

```
git remote show origin
```

Output:

```
$ git remote show origin
warning: more than one branch.master.remote
* remote origin
  Fetch URL: /Users/tonywok/src/git_immersion/auto/hello
  Push  URL: /Users/tonywok/src/git_immersion/auto/hello
  HEAD branch: master
  Remote branches:
    greet  tracked
    master tracked
  Local branch configured for 'git pull':
    master merges with remote master
              and with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Now we see that the remote repository "origin" is simply the original **hello** repository. Remote repositories typically live on a separate machine, possibly a centralized server. As we can see here, however, they can just as well point to a repository on the same machine. There is nothing particularly special about the name "origin", however the convention is to use the name "origin" for the primary centralized repository (if there is one).

Lab 40 Remote Branches

Goals

- Learn about local VS remote branches

Let's look at the branches avaiLable in our cloned repository.

Execute:

```
git branch
```

Output:

```
$ git branch
* master
```

That's it, only the master branch is listed. Where is the greet branch? The **git branch** command only lists the local branches by default.

List Remote Branches

Try this to see all the branches:

Execute:

```
git branch -a
```

Output:

```
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/greet
  remotes/origin/master
```

Git has all the commits from the original repository, but branches in the remote repository are not treated as local branches here. If we want our own **greet** branch, we need to create it ourselves. We will see how to do that in a minute.

Lab 41 Change the Original Repository

Goals

- Make some changes to the original repository so we can try to pull the changes

Make a change in the original hello repository

Execute:

```
cd ../hello
# (You should be in the original hello repository now)
```

NOTE: Now in the *hello* repo

Make the following changes to README:

File: *README*

```
This is the Hello World example from the git tutorial.
(changed in original)
```

Now add and commit this change

Execute:

```
git add README
git commit -m "Changed README in original repo"
```

Up Next

The original repository now has later changes that are not in the cloned version. Next we will pull those changes across to the cloned repository.

Lab 42 Fetching Changes

Goals

- Learn how to pull changes from a remote repository.

Execute:

```
cd ../cloned_hello
git fetch
git hist --all
```

NOTE: Now in the *cloned_hello* repo

Output:

```
$ git fetch
From /Users/jim/src/git_immersion/auto/hello
   b8188..a5aa8e7  master     -> origin/master
$ git hist --all
* a5aa8e7 2014-10-25 | Changed README in original repo (origin/master,
origin/HEAD) [Jim Weirich]
* b8188 2014-10-25 | Updated Rakefile (HEAD, origin/greet, master) [Jim
Weirich]
* b7964 2014-10-25 | Hello uses Greeter [Jim Weirich]
* 0a576ce 2014-10-25 | Added greeter class [Jim Weirich]
* 6fd9eff 2014-10-25 | Added README [Jim Weirich]
* 14ba469 2014-10-25 | Added a Rakefile. [Jim Weirich]
* 4b249b3 2014-10-25 | Moved hello.rb to lib [Jim Weirich]
* bfd1408 2014-10-25 | Add an author/email comment [Jim Weirich]
* a1189df 2014-10-25 | Added a comment (tag: v1) [Jim Weirich]
* 6083cb8 2014-10-25 | Added a default value (tag: v1-beta) [Jim Weirich]
* b24f3ff 2014-10-25 | Using ARGV [Jim Weirich]
* cf466b4 2014-10-25 | First Commit [Jim Weirich]
```

At this point the repository has all the commits from the original repository, but they are not integrated into the the cloned repository's local branches.

Find the "Changed README in original repo" commit in the history above. Notice that the commit includes "origin/master" and "origin/HEAD".

Now look at the "Updated Rakefile" commit. You will see that it the local master branch points to this commit, not to the new commit that we just fetched.

The upshot of this is that the "git fetch" command will fetch new commits from the remote repository, but it will not merge these commits into the local branches.

Check the README

We can demonstrate that the cloned README is unchanged.

Execute:

```
cat README
```

Output:

```
$ cat README
This is the Hello World example from the git tutorial.
```

See, no changes.

Lab 43 Merging Pulled Changes

Goals

- Learn to get the pulled changes into the current branch and working directory.

Merge the fetched changes into local master

Execute:

```
git merge origin/master
```

Output:

```
$ git merge origin/master
Updating b8188..a5aa8e7
Fast-forward
 README | 1 +
 1 file changed, 1 insertion(+)
```

Check the README again

We should see the changes now.

Execute:

```
cat README
```

Output:

```
$ cat README
This is the Hello World example from the git tutorial.
(changed in original)
```

There are the changes. Even though "git fetch" does not merge the changes, we can still manually merge the changes from the remote repository.

Up Next

Next let's take a look at combining the fetch & merge process into a single command.

Lab 44 Pulling Changes

Goals

- Learn that `git pull` is equivalent to a `git fetch` followed by a `git merge`.

Discussion

We're not going to go through the process of creating another change and pulling it again, but we do want you to know that doing:

```
git pull
```

is indeed equivalent to the two steps:

```
git fetch
git merge origin/master
```

Lab 45 Adding a Tracking Branch

Goals

- Learn how to add a local branch that tracks a remote branch.

The branches starting with remotes/origin are branches from the original repo. Notice that you don't have a branch called greet anymore, but it knows that the original repo had a greet branch.

Add a local branch that tracks a remote branch.

Execute:

```
git branch --track greet origin/greet
git branch -a
git hist --max-count=2
```

Output:

```
$ git branch --track greet origin/greet
Branch greet set up to track remote branch greet from origin.
$ git branch -a
  greet
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/greet
  remotes/origin/master
$ git hist --max-count=2
* a5aa8e7 2014-10-25 | Changed README in original repo (HEAD, origin/master,
origin/HEAD, master) [Jim Weirich]
* b8188 2014-10-25 | Updated Rakefile (origin/greet, greet) [Jim Weirich]
```

We can now see the greet branch in the branch list and in the log.

Lab 46 Bare Repositories

Goals

- Learn how to create bare repositories.

Bare repositories (without working directories) are usually used for sharing.

Create a bare repository.

Execute:

```
cd ..
git clone --bare hello hello.git
ls hello.git
```

NOTE: Now in the work directory

Output:

```
$ git clone --bare hello hello.git
Cloning into bare repository 'hello.git'...
warning: templates not found /Users/tonywok/.git_template
done.
$ ls hello.git
HEAD
config
objects
packed-refs
refs
```

The convention is that repositories ending in '.git' are bare repositories. We can see that there is no working directory in the hello.git repo. Essentially it is nothing but the .git directory of a non-bare repo.

Lab 47 Adding a Remote Repository

Goals

- Add the bare repository as a remote to our original repository.

Let's add the hello.git repo to our original repo.

Execute:

```
cd hello
git remote add shared ../hello.git
```

NOTE: Now in the hello repository.

Lab 48 Pushing a Change

Goals

- Learn how out to push a change to a remote repository.

Since bare repositories are usually shared on some sort of network server, it is usually difficult to cd into the repo and pull changes. So we need to push our changes into other repositories.

Let's start by creating a change to be pushed. Edit the README and commit it

File: *README*

```
This is the Hello World example from the git tutorial.
(Changed in the original and pushed to shared)
```

Execute:

```
git checkout master
git add README
git commit -m "Added shared comment to readme"
```

Now push the change to the shared repo.

Execute:

```
git push shared master
```

shared is the name of the repository receiving the changes we are pushing. (Remember, we added it as a remote in the previous Lab.)

Output:

```
$ git push shared master
To ../hello.git
   a5aa8e7..7ed6bbe  master -> master
```

NOTE: We had to explicitly name the branch master that was receiving the push. It is possible to set it up automatically, but I *never* remember the commands to do that. Check out the "Git Remote Branch" gem for easy management of remote branches.

Lab 49 Pulling Shared Changes

Goals

- Learn how to pull changes from a shared repository.

Quick hop over to the clone repository and let's pull down the changes just pushed to the shared repo.

Execute:

```
cd ../cloned_hello
```

NOTE: Now in the *cloned_hello* repo.

Continue with…

Execute:

```
git remote add shared ../hello.git
git branch --track shared master
git pull shared master
cat README
```

Lab 50 Hosting your Git Repositories

Goals

- Learn how to setup git server for sharing repositories.

There are many ways to share git repositories over the network. Here is a quick and dirty way.

Start up the git server

Execute:

```
# (From the work directory)
git daemon --verbose --export-all --base-path=.
```

Now, in a separate terminal window, go to your work directory

Execute:

```
# (From the work directory)
git clone git://localhost/hello.git network_hello
cd network_hello
ls
```

You should see a copy of hello project.

Pushing to the Git Daemon

If you want to push to the git daemon repository, add `--enable=receive-pack` to the git daemon command. Be careful because there is no authentication on this server, anyone could push to your repository.