**Часть 1.**

**Goals**

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

**Setup Name and Email *01***

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 *02***

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

Часть 2

**Goals**

* Get the tutorial materials setup and ready to run.

**Get the Tutorial package. *01***

Git the tutorial package from either:

* The memory sticks provided in the tutorial
* The URL <http://gitimmersion.com/git_tutorial.zip>

**Unzip the tutorial *02***

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.

Часть 3

**Goals**

* Learn how to create a git repository from scratch.

**Create a “Hello, World” program *01***

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 *02***

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

Initialized empty Git repository in /Users/jim/working/git/git\_immersion/auto/hello/.git/

**Add the program to the repository *03***

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) 9416416] First Commit

1 files changed, 1 insertions(+), 0 deletions(-)

create mode 100644 hello.rb

Часть 4

**Goals**

* Learn how to check the status of the repository

**Check the status of the repository *01***

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.

Часть 5

**Goals**

* Learn how to monitor the state of the working directory

**Change the “Hello, World” program. *01***

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 *02***

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 *03***

Let’s stage the change.

Часть 6

**Goals**

* Learn how to stage changes for later commits

**Add Changes *01***

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.

Часть 7

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.

Часть 8

**Goals**

* Learn how to commit changes to the repository

**Commit the change *01***

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 *02***

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.

Часть 9

**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 *01***

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 *02***

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

**Execute:**

git add hello.rb

**Second change: Add a comment *03***

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 *04***

**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 *05***

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 582495a] Added a default value

1 files changed, 3 insertions(+), 1 deletions(-)

$ 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 *06***

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 *07***

**Execute:**

git commit -m "Added a comment"

Часть 10

**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 1f7ec5eaa8f37c2770dae3b984c55a1531fcc9e7

Author: Jim Weirich <jim (at) neo.com>

Date: Sat Apr 13 15:20:42 2013 -0400

Added a comment

commit 582495ae59ca91bca156a3372a72f88f6261698b

Author: Jim Weirich <jim (at) neo.com>

Date: Sat Apr 13 15:20:42 2013 -0400

Added a default value

commit 323e28d99a07d404c04f27eb6e415d4b8ab1d615

Author: Jim Weirich <jim (at) neo.com>

Date: Sat Apr 13 15:20:42 2013 -0400

Using ARGV

commit 94164160adf8faa3119b409fcfcd13d0a0eb8020

Author: Jim Weirich <jim (at) neo.com>

Date: Sat Apr 13 15:20:42 2013 -0400

First Commit

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

**One Line Histories *01***

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

1f7ec5eaa8f37c2770dae3b984c55a1531fcc9e7 Added a comment

582495ae59ca91bca156a3372a72f88f6261698b Added a default value

323e28d99a07d404c04f27eb6e415d4b8ab1d615 Using ARGV

94164160adf8faa3119b409fcfcd13d0a0eb8020 First Commit

**Controlling Which Entries are Displayed *02***

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 *03***

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 *04***

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

\* 1f7ec5e 2013-04-13 | Added a comment (HEAD, master) [Jim Weirich]

\* 582495a 2013-04-13 | Added a default value [Jim Weirich]

\* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]

\* 9416416 2013-04-13 | 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 *05***

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

Часть 11

**Goals**

* Learn how to setup aliases and shortcuts for git commands

**Common Aliases *01***

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 *02***

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 *03***

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) *04***

**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.

**Note:** Some of these shell aliases are a bit aggressive. In particular, gs will conflict with the Linux GhostScript program. Recently I started using the go programming language and had to disable the go alias above. So use these aliases with caution.

Часть 12

**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 *01***

**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

\* 1f7ec5e 2013-04-13 | Added a comment (HEAD, master) [Jim Weirich]

\* 582495a 2013-04-13 | Added a default value [Jim Weirich]

\* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]

\* 9416416 2013-04-13 | 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 9416416

Note: checking out '9416416'.

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 9416416... First Commit

$ cat hello.rb

puts "Hello, World"

A “detached HEAD” message in git just means that HEAD (the part of git that tracks what your current working directory should match) is pointing directly to a commit rather than a branch. Any changes that are committed in this state are only remembered as long as you don’t switch to a different branch. As soon as you checkout a new branch or tag, the detached commits will be “lost” (because HEAD has moved). If you want to save commits done in a detached state, you need to create a branch to remember the commits.

Older versions of git will complain about not being on a local branch rather than being in a detached HEAD state. 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 *02***

**Execute:**

git checkout master

cat hello.rb

You should see …

**Output:**

$ git checkout master

Previous HEAD position was 9416416... 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.

Часть 13

**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 *01***

**Execute:**

git tag v1

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

**Tagging Previous Versions *02***

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 582495a... 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 *03***

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 582495a... Added a default value

HEAD is now at 1f7ec5e... Added a comment

$ git checkout v1-beta

Previous HEAD position was 1f7ec5e... Added a comment

HEAD is now at 582495a... Added a default value

**Viewing Tags using the tag command *04***

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

\* 1f7ec5e 2013-04-13 | Added a comment (v1, master) [Jim Weirich]

\* 582495a 2013-04-13 | Added a default value (HEAD, v1-beta) [Jim Weirich]

\* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]

\* 9416416 2013-04-13 | 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).

Часть 14

**Goals**

* Learn how to revert changes in the working directory

**Checkout Master *01***

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

**Execute:**

git checkout master

**Change hello.rb *02***

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 *03***

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 *04***

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.

Часть 15

**Goals**

* Learn how to revert changes that have been staged

**Change the file and stage the change *01***

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 *02***

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 *03***

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 *04***

**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.

Часть 16

**Goals**

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

**Undoing Commits *01***

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. *02***

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 *03***

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 a10293f] Revert "Oops, we didn't want this commit"

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

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 *04***

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

**Execute:**

git hist

**Output:**

$ git hist

\* a10293f 2013-04-13 | Revert "Oops, we didn't want this commit" (HEAD, master) [Jim Weirich]

\* 838742c 2013-04-13 | Oops, we didn't want this commit [Jim Weirich]

\* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]

\* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]

\* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]

\* 9416416 2013-04-13 | 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 *05***

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

Часть 17

**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 *01***

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 *02***

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

**Execute:**

git hist

**Output:**

$ git hist

\* a10293f 2013-04-13 | Revert "Oops, we didn't want this commit" (HEAD, master) [Jim Weirich]

\* 838742c 2013-04-13 | Oops, we didn't want this commit [Jim Weirich]

\* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]

\* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]

\* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]

\* 9416416 2013-04-13 | 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 *03***

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 *04***

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 1f7ec5e Added a comment

$ git hist

\* 1f7ec5e 2013-04-13 | Added a comment (HEAD, v1, master) [Jim Weirich]

\* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]

\* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]

\* 9416416 2013-04-13 | 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 *05***

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

\* a10293f 2013-04-13 | Revert "Oops, we didn't want this commit" (oops) [Jim Weirich]

\* 838742c 2013-04-13 | Oops, we didn't want this commit [Jim Weirich]

\* 1f7ec5e 2013-04-13 | Added a comment (HEAD, v1, master) [Jim Weirich]

\* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]

\* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]

\* 9416416 2013-04-13 | 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.

Часть 18

**Goals**

* Remove the oops tag (housekeeping)

**Removing tag oops *01***

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 a10293f)

$ git hist --all

\* 1f7ec5e 2013-04-13 | Added a comment (HEAD, v1, master) [Jim Weirich]

\* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]

\* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]

\* 9416416 2013-04-13 | First Commit [Jim Weirich]

The oops tag is no longer listed in the repository.

Часть 19

**Goals**

* Learn how to amend an existing commit

**Change the program then commit *01***

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 *02***

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 *03***

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 eb30103] Add an author/email comment

1 files changed, 2 insertions(+), 1 deletions(-)

**Review the History *04***

**Execute:**

git hist

**Output:**

$ git hist

\* eb30103 2013-04-13 | Add an author/email comment (HEAD, master) [Jim Weirich]

\* 1f7ec5e 2013-04-13 | Added a comment (v1) [Jim Weirich]

\* 582495a 2013-04-13 | Added a default value (v1-beta) [Jim Weirich]

\* 323e28d 2013-04-13 | Using ARGV [Jim Weirich]

\* 9416416 2013-04-13 | 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.

Часть 20

**Goals**

* Learn how to move a file within a repository.

**Move the hello.rb file into a lib directory. *01***

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 *02***

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 *03***

Let’s commit this move.

**Execute:**

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

Часть 21

**Goals**

* Add another file to our repository

**Now add a Rakefile *01***

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!

Часть 22

**Goals**

* Learn about the structure of the .git directory

**The .git Directory *01***

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 hooks logs rr-cache

HEAD config index objects

MERGE\_RR description info refs

This is the magic directory where all the git “stuff” is stored. Let’s peek in the objects directory.

**The Object Store *02***

**Execute:**

ls -C .git/objects

**Output:**

$ ls -C .git/objects

09 1f 27 43 69 83 97 af e4 info

0f 22 28 58 6b 94 9c b5 e7 pack

11 24 32 59 78 96 a1 c4 eb

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 *03***

**Execute:**

ls -C .git/objects/<dir>

**Output:**

$ ls -C .git/objects/09

6b74c56bfc6b40e754fc0725b8c70b2038b91e 9fb6f9d3a104feb32fcac22354c4d0e8a182c1

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 *04***

**Execute:**

cat .git/config

**Output:**

$ cat .git/config

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

ignorecase = true

[user]

name = Jim Weirich

email = jim (at) neo.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

1f7ec5eaa8f37c2770dae3b984c55a1531fcc9e7

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 *06***

**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.

Часть 23

**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 *01***

**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

\* 96ee164 2013-04-13 | Added a Rakefile. (HEAD, master) [Jim Weirich]

**Dumping the Latest Commit *02***

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 96ee164

commit

$ git cat-file -p 96ee164

tree 096b74c56bfc6b40e754fc0725b8c70b2038b91e

parent 0f36766e05bc55d765ec8afe288430edc69fceea

author Jim Weirich <jim (at) neo.com> 1365880844 -0400

committer Jim Weirich <jim (at) neo.com> 1365880844 -0400

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 *03***

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

040000 tree e46f374f5b36c6f02fb3e9e922b79044f754d795 lib

Yep, I see the Rakefile and the lib directory.

**Dumping the lib directory *04***

**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 *05***

**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 *06***

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.