**Creating, Adding, and Committing: Git Basics**

<http://media.pragprog.com/titles/tsgit/chap-005-extract.html>

**Committing Changes**

Committing is a relatively straight forward process that adds your changes to the history of your repository and assigns a commit name to it.

The change is not sent to a central repository, though. Other people can pull the change from you, or you can push the change to some other repository, but there's no automatic updating. We'll talk about these in Section *Keeping Up-To-Date* and Section *Pushing Changes*.

You can use commit in multiple ways to commit changes to your repository, but every commit requires a log message. You can add a message by adding -m "your message". The message can be any valid string. You can also specify multiple paragraphs by passing multiple -m options to git commit.

For more complex messages that require an editor, you can execute git commit without the -m and Git launches your editor to create your log message. When Git tries to launch an editor, it looks through the following values in order:

* GIT\_EDITOR environment variable
* core.editor Git configuration value
* VISUAL environment variable
* EDITOR environment variable
* Git tries vi if nothing else is set

When you use the editor to create your commit message, you can tell Git to add a diff in the editor showing the changes you are about to commit by adding the -v.

The log message is only as good as the content that you commit, though. Like nearly every command in Git, there are a few different ways to handle a commit.

The first is to call git add in some form for every file---or change if you're using git add –p --- that you want to commit. This stages those changes for commit, calling git commit closes the loop. This process looks like this:

|  |  |
| --- | --- |
|  | prompt> **git add some-file** |
|  | prompt> **git commit -m "Refactor to simplify"** |
|  |  |

Another way to handle commits is to pass it the -a parameter on the command line. Remember, we talked about this as the shortcut around staging changes in the last section. It tells Git to take the most current version of your working tree and commit it to the repository. It won't add new, untracked files, however, only files that are already being tracked.

If the only change you had made to your working tree in the example above was to the some-file, you could perform the same commit by executing the following:

|  |  |
| --- | --- |
|  | prompt> **git commit -m "Refactor to simplify" -a** |

The last method of committing changes is to specify the file or files you want to commit. Just add each file you want to commit after you specify all of the options you want to pass Git. Using the example from the two previous commits:

|  |  |
| --- | --- |
|  | prompt> **git commit -m "Refactor to simplify" some-file** |
|  |  |

All three examples have their use. Staged commits are useful when you want to commit just a portion of a file using the git add -p command. If you need to pull just one file out of several that have changed and commit that, you can commit using the explicit file.

There is an important difference to remember between the first method of committing staged commits versus committing all changes or a particular file's changes. The last two commit the file as it exists the moment you execute the commit. The first commits the change you staged.

This means that you can stage a change, make a change to the file, then commit the change you had staged while still having a file that is changed in your working tree.

Think of a staged commit as a buffer. You add to the buffer with git add. That buffer stays there until you save it by executing git commit.

We've covered the basics now. We have a repository, we've added some files to it, and committed some changes. When you're working with the files in your repository, you'll need to see what changes you have made. That's up next.

**Missing Subversion Shortcuts**

If you're coming from Subversion, you are probably used to all of those shortcuts to commonly used commands. You never have to type svn checkout, or svn commit because a simple svn co or svn ci does the trick for you.

As you've worked with Git, you probably tried those same aliases and been given an error that it was not a git command. Its true that Git doesn't ship with all of those aliases, but it does give you a better option. You can add your own aliases via the git config.

To add git ci as a shortcut to git commit, just type git config --global alias.ci "commit" the command prompt. This works for any Git command so you can customize your environment just the way you want it.

**Seeing What Has Changed**

It's easy to remember that you added a new file or made a change to one file when it's fresh in your mind. Sometimes you don't have that luxury though. You need to find out what has changed in your working tree and how it's changed. You can use two of Git's provided commands, git status and git diff to do that.

**Viewing the Current Status**

You can use git status to see all of the changes that have occurred in your repository. The output it generates is based on the status of any staged commits and how your current working copy compares to what is tracked by the repository.

To start, change the line monday to read MONDAY. Now run git status:

|  |  |
| --- | --- |
|  | prompt> **git status** |
|  | *# On branch test* |
|  | *# Changed but not updated:* |
|  | *# (use "git add <file>..." to update what will be committed)* |
|  | *#* |
|  | *# modified: days.txt* |
|  | *#* |
|  | no changes added to commit (use "git add" and/or "git commit -a") |
|  |  |

This tells you that Git hasn't staged the change yet. A quick git add changes that:

|  |  |
| --- | --- |
|  | prompt> **git add days.txt** |
|  | prompt> **git status** |
|  | *# On branch test* |
|  | *# Changes to be committed:* |
|  | *# (use "git reset HEAD <file>..." to unstage)* |
|  | *#* |
|  | *# modified: days.txt* |
|  | *#* |

Now you've staged the commit. The header before the days.txt file has changed. Make another change to days.txt now and re-run git status.

|  |  |
| --- | --- |
|  | ... make change to days.txt ... |
|  | prompt> **git status** |
|  | *# On branch test* |
|  | *# Changes to be committed:* |
|  | *# (use "git reset HEAD <file>..." to unstage)* |
|  | *#* |
|  | *# modified: days.txt* |
|  | *#* |
|  | *# Changed but not updated:* |
|  | *# (use "git add <file>..." to update what will be committed)* |
|  | *#* |
|  | *# modified: days.txt* |
|  |  |

Now git status tells you there's two changes. The first one is the staged commit you already added; the second is the change you just made. If you were to make a commit right now, the first change that you staged is all that would be committed. This is because of the two-step commit process we talked about in Chapter 3.

**Viewing Difference**

Another useful command for looking at changes is git diff. It can show you differences between your working copy and the repository as well as your staged commit and the working copy.

You can view the changes you just made to your repository, but haven't committed by executing git diff from the command line.

Let's change a file first. Edit the days.txt file and change the line with Friday to be all upper case.

|  |  |
| --- | --- |
|  | ... change days.txt file and save it ... |
|  | prompt> **git diff** |
|  | index 6d612ab..c684aa5 100644 |
|  | --- a/days.txt |
|  | +++ b/days.txt |
|  | @@ -2,6 +2,6 @@ MONDAY |
|  | tuesday |
|  | wednesday |
|  | thursday |
|  | -friday |
|  | +FRIDAY |
|  | saturday |
|  | Sunday |

We can see from the output of diff that we changed the case of friday. The - marks the line we removed; the + marks the line we added. This format, with all of the extra information at the top, is called a *unified diff*. It is one of the most common ways for developers to communicate changes to one another when they are not using a common version control system.

git diff only shows changes in your working tree that you have not staged or committed yet. You can view the difference between what you have staged and what is in the repository by adding --cached to the git diff command.

|  |  |
| --- | --- |
|  | prompt> **git diff --cached** |
|  | diff --git a/days.txt b/days.txt |
|  | index f647364..6d612ab 100644 |
|  | --- a/days.txt |
|  | +++ b/days.txt |
|  | @@ -1,4 +1,4 @@ |
|  | -monday |
|  | +MONDAY |
|  | tuesday |
|  | wednesday |
|  | Thursday |

When you execute git diff without any parameters, it considers anything you have staged as part of the repository's content so it generates the diff as if that was already committed. To see the difference between what is in your working tree and the latest commit to the repository, add HEAD after git diff.

|  |  |
| --- | --- |
|  | prompt> **git diff HEAD** |
|  | diff --git a/days.txt b/days.txt |
|  | index f647364..c684aa5 100644 |
|  | --- a/days.txt |
|  | +++ b/days.txt |
|  | @@ -1,7 +1,7 @@ |
|  | -monday |
|  | +MONDAY |
|  | tuesday |
|  | wednesday |
|  | thursday |
|  | -friday |
|  | +FRIDAY |
|  | saturday |
|  | Sunday |

HEAD is just a keyword that refers to the most recent commit made to the repository. Whenever you see someone refer to the HEAD of the repository, they just mean the last commit.

Now you know all of the normal commands to get you going. You can add files, commit changes to the files you're tracking, and even compare the changes you've made against what's in your repository. Now you might want to do some housekeeping in it. Things such as moving, copying, and even ignoring some files. We'll cover these next.