Extracted from:

Pragmatic Version Control

Using Git

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Version Control

Using Git

The Pragmatic Starter Kit—Volume 1

Travis Swicegood

Edited by Susannah Davidson Pfalzer

Pragmatic Version Control

Using Git

Travis Swicegood

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

4.2 Committing Changes

Committing is a relatively straightforward process that adds your changes to the history of your repository and assigns a commit name to them.

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 7.3, *Keeping Up-to-Date*, on page ? and Section 7.4, *Pushing Changes*, on page ?.

You can use `git commit` in multiple ways to commit changes to your repository, but every commit requires a log message. For simple messages, 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 this order:

1. `GIT_EDITOR` environment variable.
2. `core.editor` Git configuration value.
3. `VISUAL` environment variable.
4. `EDITOR` environment variable.
5. Git tries `vi` if nothing else is set.

When you use the editor to create your commit message, by adding the `-v` option you can tell Git to add a diff in the editor showing the changes you are about to commit. There will be a lot of lines that begin with `#`. All of those are ignored when Git reads the commit message.

Like nearly every command in Git, there are a few different ways to handle a commit. Before we create a new commit, let's look at the three ways to generate a commit.

First, you can call `git add` in some form for the files—or changes if you're using `git add -p`—that you want to commit. This stages those changes for commit, and calling `git commit` closes the loop. The process looks like this:

```
prompt> git add some-file
prompt> git commit -m "changes to some-file"
```

That's an example of a bad commit log message, but it does fit nicely on a printed page, so I'll use it here as an example. In a real commit message,

don't just state the obvious—make sure to explain why you made the changes too. Remember from *What Should My Commit Log Message Contain?*, on page ? that your commit message should explain the commit like you would to a developer sitting right next to you.

Another way to handle commits is to pass `git commit` the `-a` parameter on the command line. 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 previous example was to `some-file`, you could perform the same commit by executing the following:

```
prompt> git commit -m "changes to some-file" -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 the options you want to pass Git.

Like the `-a` parameter, this takes the latest version from your working tree. But it takes just the file or files you specify. Here's an example from the two previous commits:

```
prompt> git commit -m "changes to some-file" some-file
```

All three examples have their uses. Staged commits are useful when you want to commit a portion of a file using the `git add -p` command. If you need to pull 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 and committing all changes or a particular file's changes. The last two methods commit the file as it exists the moment you execute the commit. The first method commits the change you staged.

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

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

Speaking of staged commits, you still have that change staged from the previous section. Let's leave it there for a little bit longer while we explore various ways to view what has changed in your repository.

Using Git Aliases Like SVN Shortcuts

If you're coming from Subversion, you are probably used to all 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.

Reading through this book and trying the examples, you might have tried those same aliases with Git and got an error that you weren't using a Git command. Git doesn't ship with all those aliases like Subversion, but it does give you a better option. You can add your own aliases via git config.

You can add git ci as a shortcut to git commit with the following:

```
prompt> git config --global alias.ci "commit"
```

That works for any Git command, so you can customize your environment just the way you want it. Substitute the portion after alias. for the alias you want to create, and you're set.

4.3 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. Someone walks into your office or you get caught up in preparation for an impending deadline right after you stage a change.

You need to find out what has changed in your working tree and how it has changed. You can use two of Git's commands, git status and git diff, to do that.

Viewing the Current Status

You can use git status to see all 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.

There's still that change you staged earlier. You can see it right now with git status:

```
prompt> git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
```

Note that it has the header Changes to be committed. That tells you that it's waiting to be committed. Let's add another change. This time add a Contact link after the About link. Here's the new line I added:

```
<li><a href="contact.html">Contact</a></li>
```

After you've saved your change, run git status again:

```
prompt> git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#
```

Now there are two index.html files listed. The first one is the change you staged earlier. You can tell this because of that header—Changes to be committed. The second one is the change you just made. It hasn't been staged yet, so it has the Changed but not updated header above it. The first will be green and the second red if you turned on color.ui back in Section 2.2, *Configuring Git*, on page ?.

You just made the changes, so I'm sure you still remember what they are. Sometimes you need to see, though. To do that, you can check the diffs—the differences—between the files.

Viewing Difference

Git can show you the differences between what's in your working tree, what's staged and ready to be committed, and what's in your repository. You use the git diff command to do this.

Calling git diff with no parameters shows you the changes in your working tree that you haven't staged or committed yet.

```
prompt> git diff
diff --git a/index.html b/index.html
index ca86894..5fdc539 100644
--- a/index.html
+++ b/index.html
@@ -7,6 +7,7 @@
     <h1>Hello World!</h1>
     <ul>
         <li><a href="about.html">About</a></li>
```

```
+        <li><a href="contact.html">Contact</a></li>
     </ul>
 </body>
 </html>
```

That shows you the Contact link you added—the + at the beginning of the line shows that it's an addition—but something is missing. The About link doesn't show any change.

Running git diff without any parameters compares the changes in your working tree against the staging area. You know that there are changes there, though. git status shows that something is staged. View the differences in the staging area and the repository by adding --cached to the call:

```
prompt> git diff --cached
diff --git a/index.html b/index.html
index e812d0a..ca86894 100644
--- a/index.html
+++ b/index.html
@@ -6,7 +6,7 @@
 <body>
     <h1>Hello World!</h1>
     <ul>
-        <li><a href="bio.html">Biography</a></li>
+        <li><a href="about.html">About</a></li>
     </ul>
 </body>
 </html>
```

Now you see that change you made back in Section 4.1, *Adding Files*, on page ?. Notice that there's a - in front of the Biography line. This shows you that the line is being removed. If colors are on, Git also marks the deleted content as red, and added content is green.

That diff doesn't show the changes that aren't staged yet, though. You can compare everything that's in your working tree—including your staged changes—against what's in your repository. To do that, execute git diff, and add HEAD to the end:

```
prompt> git diff HEAD
diff --git a/index.html b/index.html
index e812d0a..5fdc539 100644
--- a/index.html
+++ b/index.html
@@ -6,7 +6,8 @@
 <body>
     <h1>Hello World!</h1>
     <ul>
-        <li><a href="bio.html">Biography</a></li>
```

```
+        <li><a href="about.html">About</a></li>
+        <li><a href="contact.html">Contact</a></li>
      </ul>
 </body>
 </html>
```

HEAD is a keyword that refers to the most recent commit to the branch you're in. You don't need to worry about that branch part just yet, but it does come back up.

Let's create a commit so our changes are tracked:

```
prompt> git commit -a -m "Change biography link and add contact link" \
    -m "About is shorter, so easier to process" \
    -m "We need to provide contact info"
Created commit 6f1bf6f: Change biography link and add contact link
 1 files changed, 2 insertions(+), 1 deletions(-)
```

Now you know all 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. Let's get into some housecleaning commands now such as moving, copying, and even ignoring some files.