Mike McQuaid
Foreword by Scott Chacon

Git in Practice

Includes 66 Techniques

SAMPLE CHAPTER

MANNING

Git in Practice
by Mike McQuaid

Chapter 3

brief contents

3 Filesystem interactions

This chapter covers

- Renaming, moving, and removing versioned files or directories
- Telling Git to ignore certain files or changes
- Deleting all untracked or ignored files or directories
- Resetting all files to their previously committed state
- Temporarily stashing and reapplying changes to files

When working with a project in Git, you'll sometimes want to move, delete, change, and/or ignore certain files in your working directory. You could mentally keep track of the state of important files and changes, but this isn't a sustainable approach. Instead, Git provides commands for performing filesystem operations for you.

Understanding the Git filesystem commands will allow you to quickly perform these operations rather than being slowed down by Git's interactions. Let's start with the most basic file operations: renaming or moving a file.

Technique 17 Renaming or moving a file: git mv

Git keeps track of changes to files in the working directory of a repository by their name. When you move or rename a file, Git doesn't see that a file was moved; it sees that there's a file with a new filename, and the file with the old filename was deleted (even if the contents remain the same). As a result, renaming or moving a file in Git is essentially the same operation; both tell Git to look for an existing file in a new location. This may happen if you're working with tools (such as IDEs) that move files for you and aren't aware of Git (and so don't give Git the correct move instruction).

Sometimes you'll still need to manually rename or move files in your Git repository, and want to preserve the history of the files after the rename or move operation. As you learned in technique 4, readable history is one of the key benefits of a version control system, so it's important to avoid losing it whenever possible. If a file has had 100 small changes made to it with good commit messages, it would be a shame to undo all that work just by renaming or moving a file.

Problem

In your Git working directory, you wish to rename a previously committed file named GitInPractice.asciidoc to 01-IntroducingGitInPractice.asciidoc and commit the newly renamed file.

Solution

1. Change to the directory containing your repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git mv GitInPractice.asciidoc 01-IntroducingGitInPractice.asciidoc`. There will be no output.
3. Run `git commit --message 'Rename book file to first part file.'`. The output should resemble the following.

Listing 3.1 Output: renamed commit

```
# git commit --message 'Rename book file to first part file.'
[master c6eed66] Rename book file to first part file.
1 file changed, 0 insertions(+), 0 deletions(-)
rename GitInPractice.asciidoc =>
  01-IntroducingGitInPractice.asciidoc (100%)
```

Commit message

No insertions/ deletions

Old filename changed to new filename

You've renamed GitInPractice.asciidoc to 01-IntroducingGitInPractice.asciidoc and committed it.

Discussion

Moving and renaming files in version control systems rather than deleting and re-creating them is done to preserve their history. For example, when a file has been moved into a new directory, you'll still be interested in the previous versions of the file before it was moved. In Git's case, it will try to auto-detect renames or moves on `git`

`add` or `git commit`; if a file is deleted and a new file is created, and those files have a majority of lines in common, Git will automatically detect that the file was moved and `git mv` isn't necessary. Despite this handy feature, it's good practice to use `git mv` so you don't need to wait for a `git add` or `git commit` for Git to be aware of the move and so you have consistent behavior across different versions of Git (which may have differing move auto-detection behavior).

After running `git mv`, the move or rename will be added to Git's index staging area, which, if you remember from technique 2, means the change has been staged for inclusion in the next commit.

It's also possible to rename files or directories and move files or directories into other directories in the same Git repository using the `git mv` command and the same syntax as earlier. If you want to move files into or out of a repository, you must use a different, non-Git command (such as a Unix `mv` command), because Git doesn't handle moving files between different repositories with `git mv`.

> **WHAT IF THE NEW FILENAME ALREADY EXISTS?** If the filename you move to already exists, you'll need to use the `git mv -f` (or `--force`) option to request that Git overwrite whatever file is at the destination. If the destination file hasn't already been added or committed to Git, then it won't be possible to retrieve the contents if you erroneously asked Git to overwrite it.

Technique 18 Removing a file: git rm

Like moving and renaming files, removing files from version control systems requires not just performing the filesystem operation as usual, but also notifying Git and committing the file. In almost any version-controlled project, you'll at some point want to remove some files, so it's essential to know how to do so. Removing version-controlled files is also safer than removing non-version-controlled files because even after removal, the files still exist in the history.

Sometimes tools that don't interact with Git may remove files for you and require you to manually indicate to Git that you wish these files to be removed. For testing purposes, let's create and commit a temporary file to be removed:

1. Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `echo Git Sandwich > GitInPracticeReviews.tmp`. This creates a new file named GitInPracticeReviews.tmp with the contents "Git Sandwich".
3. Run `git add GitInPracticeReviews.tmp`.
4. Run `git commit --message 'Add review temporary file.'`.

Note that if `git add` fails, you may have *.tmp in a .gitignore file somewhere (introduced in technique 21). In this case, add it using `git add --force GitInPracticeReviews.tmp`.

Problem

You wish to remove a previously committed file named GitInPracticeReviews.tmp in your Git working directory and commit the removed file.

Solution

1. Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git rm GitInPracticeReviews.tmp`.
3. Run `git commit --message 'Remove unfavourable review file.'`. The output should resemble the following.

Listing 3.2 Output: removed commit

```
# git rm GitInPracticeReviews.tmp

rm 'GitInPracticeReviews.tmp'

# git commit --message 'Remove unfavourable review file.'

[master 06b5eb5] Remove unfavourable review file.
1 file changed, 1 deletion(-)
delete mode 100644 GitInPracticeReviews.tmp
```

Commit message

1 line deleted

Deleted filename

You've removed GitInPracticeReviews.tmp and committed it.

Discussion

Git only interacts with the Git repository when you explicitly give it commands, which is why when you remove a file, Git doesn't automatically run a `git rm` command. The `git rm` command is indicating to Git not just that you wish for a file to be removed, but also (like `git mv`) that this removal should be part of the next commit.

If you want to see a simulated run of `git rm` without actually removing the requested file, you can use `git rm -n` (or `--dry-run`). This will print the output of the command as if it were running normally and indicate success or failure, but without removing the file.

To remove a directory and all the unignored files and subdirectories within it, you need to use `git rm -r` (where the `-r` stands for *recursive*). When run, this deletes the directory and all unignored files under it. This combines well with `--dry-run` if you want to see what would be removed before removing it.

> **WHAT IF A FILE HAS UNCOMMITTED CHANGES?** If a file has uncommitted changes but you still wish to remove it, you need to use the `git rm -f` (or `--force`) option to indicate to Git that you want to remove it before committing the changes.

Technique 19 Resetting files to the last commit: git reset

There are times when you've made changes to files in the working directory but you don't want to commit these changes. Perhaps you added debugging statements to files

and have now committed a fix, so you want to reset all the files that haven't been committed to their last committed state (on the current branch).

Problem

You wish to reset the state of all the files in your working directory to their last committed state.

Solution

1. Change to the directory containing your repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `echo EXTRA >> 01-IntroducingGitInPractice.asciidoc` to append "EXTRA" to the end of 01-IntroducingGitInPractice.asciidoc.
3. Run `git reset --hard`. The output should resemble the following.

Listing 3.3 Output: hard reset

```
# git reset --hard
HEAD is now at 06b5eb5 Remove unfavourable review file.
```

Reset commit

You've reset the Git working directory to the last committed state.

Discussion

The `--hard` argument signals to `git reset` that you want it to reset both the index staging area and the working directory to the state of the previous commit on this branch. If run without an argument, it defaults to `git reset --mixed`, which resets the index staging area but not the contents of the working directory. In short, `git reset --mixed` only undoes `git add`, but `git reset --hard` undoes `git add` and all file modifications.

`git reset` will be used to perform more operations (including those that alter history) later, in technique 42.

> **DANGERS OF USING GIT RESET --HARD** Take care when you use `git reset --hard`; it will immediately and without prompting remove all uncommitted changes to any file in your working directory. This is probably the command that has caused me more regret than any other; I've typed it accidentally and removed work I hadn't intended to. Remember that in section 1.1 you learned that it's very hard to lose work with Git? If you have uncommitted work, this is one of the easiest ways to lose it! A safer option may be to use Git's stash functionality instead.

Technique 20 Deleting untracked files: git clean

When working in a Git repository, some tools may output undesirable files into your working directory. Some text editors may use temporary files, operating systems may write thumbnail cache files, or programs may write crash dumps. Alternatively, sometimes there may be files that are desirable, but you don't wish to commit them

to your version control system; instead you want to remove them to build clean versions (although this is generally better handled by *ignoring* these files, as shown in technique 21).

When you wish to remove these files, you could remove them manually. But it's easier to ask Git to do so, because it already knows which files in the working directory are versioned and which are *untracked*.

You can view the files that are currently tracked by running `git ls-files`. This currently only shows 01-IntroducingGitInPractice.asciidoc, because that is the only file that has been added to the Git repository. You can run `git ls-files --others` (or `-o`) to show the currently untracked files (there should be none).

For testing purposes, let's create a temporary file to be removed:

1. Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `echo Needs more cowbell > GitInPracticeIdeas.tmp`. This creates a new file named GitInPracticeIdeas.tmp with the contents "Needs more cowbell".

Problem

You wish to remove an untracked file named GitInPracticeIdeas.tmp from a Git working directory.

Solution

1. Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git clean --force`. The output should resemble the following.

Listing 3.4 Output: force-cleaned files

```
# git clean --force
Removing GitInPracticeIdeas.tmp
```

Removed file

You've removed GitInPracticeIdeas.tmp from the Git working directory.

Discussion

`git clean` requires the `--force` argument because this command is potentially dangerous; with a single command, you can remove many, many files very quickly. Remember that in section 1.1, you learned that accidentally losing any file or change committed to the Git system is nearly impossible. This is the opposite situation; `git clean` will happily remove thousands of files very quickly, and they can't be easily recovered (unless you backed them up through another mechanism).

To make `git clean` a bit safer, you can preview what will be removed before doing so by using `git clean -n` (or `--dry-run`). This behaves like `git rm --dry-run` in that it prints the output of the removals that would be performed but doesn't actually do so.

To remove untracked directories as well as untracked files, you can use the `-d` ("directory") parameter.

Technique 21 Ignoring files: .gitignore

As discussed in technique 20, sometimes working directories contain files that are *untracked* by Git, and you don't want to add them to the repository. Sometimes these files are one-off occurrences; you accidentally copy a file to the wrong directory and want to delete it. More often, they're the product of software (such as the software stored in the version control system or some part of your operating system) putting files into the working directory of your version control system.

You could `git clean` these files each time, but that would rapidly become tedious. Instead, you can tell Git to ignore them so it never complains about these files being untracked and you don't accidentally add them to commits.

Problem

You wish to ignore all files with the extension .tmp in a Git repository.

Solution

1. Change to the directory containing your repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `echo \*.tmp > .gitignore`. This creates a new file named .gitignore with the contents "*.tmp".
3. Run `git add .gitignore` to add .gitignore to the index staging area for the next commit. There will be no output.
4. Run `git commit --message='Ignore .tmp files.'`. The output should resemble the following.

Listing 3.5 Output: ignore file commit

```
# git commit --message='Ignore .tmp files.'

[master 0b4087c] Ignore .tmp files.
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

Commit message

1 line deleted

Created filename

You've added a .gitignore file with instructions to ignore all .tmp files in the Git working directory.

Discussion

Each line of a .gitignore file matches files with a pattern. For example, you can add comments by starting a line with a `#` character or negate patterns by starting a line with a `!` character. Read more about the pattern syntax in `git help gitignore`.

A good and widely held principle for version control systems is to avoid committing *output files* to a version control repository. Output files are those that are created from input files that are stored in the version control repository.

For example, you may have a hello.c file that is compiled into a hello.o object file. The hello.c *input file* should be committed to the version control system, but the hello.o *output file* should not.

Committing .gitignore to the Git repository makes it easy to build up lists of expected output files so they can be shared between all the users of a repository and not accidentally committed. GitHub also provides a useful collection of gitignore files at https://github.com/github/gitignore.

Let's try to add an ignored file:

1. Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `touch GitInPracticeGoodIdeas.tmp`. This creates a new, empty file named GitInPracticeGoodIdeas.tmp.
3. Run `git add GitInPracticeGoodIdeas.tmp`. The output should resemble the following.

Listing 3.6 Output: trying to add an ignored file

```
# git add GitInPracticeGoodIdeas.tmp

The following paths are ignored by one of your .gitignore files:
GitInPracticeGoodIdeas.tmp                                  <-- 1 Ignored file
Use -f if you really want to add them.
fatal: no files added                                       <-- 2 Error message
```

1. GitInPracticeGoodIdeas.tmp wasn't added, because its addition would contradict your .gitignore rules.
2. was printed, because no files were added.

This interaction between .gitignore and `git add` is particularly useful when adding subdirectories of files and directories that may contain files that should to be ignored. `git add` won't add these files but will still successfully add all others that shouldn't be ignored.

Technique 22 Deleting ignored files

When files have been successfully ignored by the addition of a .gitignore file, you'll sometimes want to delete them all. For example, you may have a project in a Git repository that compiles input files (such as .c files) into output files (in this example, .o files) and wish to remove all these output files from the working directory to perform a new build from scratch.

Let's create some temporary files that can be removed:

1. Change to the directory containing your repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `touch GitInPracticeFunnyJokes.tmp GitInPracticeWittyBanter.tmp`.

Problem

You wish to delete all ignored files from a Git working directory.

Solution

1. Change to the directory containing your repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git clean --force -X`. The output should resemble the following.

Listing 3.7 Output: force-cleaning ignored files

```
# git clean --force -X

Removing GitInPracticeFunnyJokes.tmp          <-- Removed file
Removing GitInPracticeWittyBanter.tmp
```

You've removed all ignored files from the Git working directory.

Discussion

The `-X` argument specifies that `git clean` should remove *only* ignored files from the working directory. If you wish to remove ignored files *and* all the untracked files (as `git clean --force` would do), you can instead use `git clean -x` (note that the `-x` is lowercase rather than uppercase).

The specified arguments can be combined with the others discussed in technique 20. For example, `git clean -xdf` removes all untracked or ignored files (`-x`) and directories (`-d`) from a working directory. This removes all files and directories for a Git repository that weren't previously committed. Take care when running this; there will be no prompt, and all the files will be quickly deleted.

Often `git clean -xdf` is run after `git reset --hard`; this means you'll have to reset all files to their last-committed state and remove all uncommitted files. This gets you a clean working directory: no added files or changes to any of those files.

Technique 23 Temporarily stashing some changes: git stash

There are times when you may find yourself working on a new commit and want to temporarily undo your current changes but redo them at a later point. Perhaps there was an urgent issue that means you need to quickly write some code and commit a fix. In this case, you could make a temporary branch and merge it in later, but this would add a commit to the history that may not be necessary. Instead you can *stash* your uncommitted changes to store them temporarily and then be able to change branches, pull changes, and so on without needing to worry about these changes getting in the way.

Problem

You wish to stash all your uncommitted changes for later retrieval.

Solution

1. Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `echo EXTRA >> 01-IntroducingGitInPractice.asciidoc`.
3. Run `git stash save`. The output should resemble the following.

Listing 3.8 Output: stashing uncommitted changes

```
# git stash save

Saved working directory and index state WIP on master:
36640a5 Ignore .tmp files.
HEAD is now at 36640a5 Ignore .tmp files.          <-- Current commit
```

You've stashed your uncommitted changes.

Discussion

`git stash save` creates a temporary commit with a prepopulated commit message and then returns your current branch to the state before the temporary commit was made. It's possible to access this commit directly, but you should only do so through `git stash` to avoid confusion.

You can see all the stashes that have been made by running `git stash list`. The output will resemble the following.

Listing 3.9 List of stashes

```
stash@{0}: WIP on master: 36640a5 Ignore .tmp files.    <-- Stashed commit
```

This shows the single stash that you made. You can access it using `ref stash@{0}`; for example, `git diff stash@{0}` will show you the difference between the working directory and the contents of that stash.

If you save another stash, it will become `stash@{0}` and the previous stash will become `stash@{1}`. This is because the stashes are stored on a *stack* structure. A stack structure is best thought of as being like a stack of plates. You add new plates on the top of the existing plates; and if you remove a single plate, you take it from the top. Similarly, when you run `git stash`, the new stash is added to the top (it becomes `stash@{0}`) and the previous stash is no longer at the top (it becomes `stash@{1}`).

DO YOU NEED TO USE GIT ADD BEFORE GIT STASH? No, `git add` is not needed. `git stash` stashes your changes regardless of whether they've been added to the index staging area by `git add`.

DOES GIT STASH WORK WITHOUT THE SAVE ARGUMENT? If `git stash` is run with no `save` argument, it performs the same operation; the `save` argument isn't

needed. I've used it in the examples because it's more explicit and easier to remember.

Technique 24 Reapplying stashed changes: git stash pop

When you've stashed your temporary changes and performed whatever operations required a clean working directory (perhaps you fixed and committed the urgent issue), you'll want to reapply the changes (because otherwise you could've just run `git reset --hard`). When you've checked out the correct branch again (which may differ from the original branch), you can request that the changes be taken from the stash and applied onto the working directory.

Problem

You wish to pop the changes from the last `git stash save` into the current working directory.

Solution

1. Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git stash pop`. The output should resemble the following.

Listing 3.10 Output: reapplying stashed changes

```
# git stash pop

# 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:   01-IntroducingGitInPractice.asciidoc
#
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (f7e39e2590067510be1a540b073e74704395e881)
```

Current branch output

Begin status output

End status output

Stashed commit

You've reapplied the changes from the last `git stash save`.

Discussion

When running `git stash pop`, the top stash on the stack (`stash@{0}`) is applied to the working directory and removed from the stack. If there's a second stash in the stack (`stash@{1}`), it's now at the top (it becomes `stash@{0}`). This means if you run `git stash pop` multiple times, it will keep working down the stack until no more stashes are found, at which point it will output `No stash found`.

If you wish to apply an item from the stack multiple times (perhaps on multiple branches), you can instead use `git stash apply`. This applies the stash to the working tree as `git stash pop` does but keeps the top stack stash on the stack so it can be run again to reapply.

Technique 25 Clearing stashed changes: git stash clear

You may have stashed changes with the intent of popping them later, but then realize that you no longer wish to do so—the changes in the stack are now unnecessary, so you want to get rid of them all. You could do this by popping each change off the stack and then deleting it, but it would be handy to have a command that allows you to do this in a single step. Thankfully, `git stash clear` does just this.

Problem

You wish to clear all previously stashed changes.

Solution

1. Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git stash clear`. There will be no output.

You've cleared all the previously stashed changes.

Discussion

Clearing the stash is done without a prompt and removes every previous item from the stash, so be careful when doing so. Cleared stashes can't be easily recovered. For this reason, once you learn about history rewriting in technique 42, I'd recommend making commits and rewriting them later rather than relying too much on `git stash`.

Technique 26 Assuming files are unchanged

Sometimes you may wish to make changes to files but have Git ignore the specific changes you've made so that operations such as `git stash` and `git diff` ignore these changes. In these cases, you could ignore them yourself or stash them elsewhere, but it would be ideal to be able to tell Git to ignore these particular changes.

I've found myself in a situation in the past where I wanted to test a Rails configuration file change for a week or two while continuing to do my normal work. I didn't want to commit it because I didn't want it to apply to servers or my coworkers, but I did want to continue testing it while I made other commits rather than change to a particular branch each time.

Problem

You wish for Git to assume there have been no changes made to 01-IntroducingGitInPractice.asciidoc.

Solution

1. Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.

2 Run `git update-index --assume-unchanged 01-IntroducingGitInPractice.asciidoc`. There will be no output.

Git will ignore any changes made to 01-IntroducingGitInPractice.asciidoc.

Discussion

When you run `git update-index --assume-unchanged`, Git sets a special flag on the file to indicate that it shouldn't be checked for any changes. This can be useful to temporarily ignore changes made to a particular file when looking at `git status` or `git diff`, but also to tell Git to avoid checking a file that is particularly huge and/or slow to read. This isn't generally a problem on normal filesystems on which Git can quickly query whether a file has been modified by checking the File Modified timestamp (rather than having to read the entire file and compare it).

`git update-index --assume-unchanged` takes only files as arguments, rather than directories. If you assume multiple files are unchanged, you need to specify them as multiple arguments; for example, `git update-index --assume-unchanged 00-Preface.asciidoc 01-IntroducingGitInPractice.asciidoc`.

The `git update-index` command has other complex options, but we'll only cover those around the "assume" logic. The rest of the behavior is better accessed through the `git add` command; it's a higher-level and more user-friendly way of modifying the state of the index.

Technique 27 Listing assumed-unchanged files

When you've told Git to assume no changes were made to particular files, it can be hard to remember which files were updated. In this case, you may end up modifying a file and wondering why Git doesn't seem to want to show you the changes. Additionally, you could forget that you made the changes and be confused as to why the state in your text editor doesn't seem to match the state that Git is seeing.

Problem

You wish for Git to list all the files that it has been told to assume haven't changed.

Solution

1 Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.

2 Run `git ls-files -v`. The output should resemble the following.

Listing 3.11 Output: listing assumed-unchanged files

```
# git ls-files -v

H .gitignore
h 01-IntroducingGitInPractice.asciidoc
```

1 Committed file

2 Assumed-unchanged file

❶ shows that committed files are indicated by an uppercase `H` at the beginning of the line.

❷ shows that an assumed-unchanged file is indicated by a lowercase `h` tag.

Discussion

Like `git update-index`, `git ls-files -v` is a low-level command that you'll typically not run often. `git ls-files` without any arguments lists the files in the current directory that Git knows about, but the `-v` argument means it's followed by tags that indicate file state.

Rather than reading through the output for this command, you can instead run `git ls-files -v | grep '^[hsmrck?]' | cut -c 3-`. This uses Unix pipes, where the output of each command is passed into the next and modified.

`grep '^[hsmrck?]'` filters the output filenames to show only those that begin with any of the lowercase `hsmrck?` characters (the valid prefixes output by `git ls-files`). It's not important to understand the meanings of any prefixes other than `H` and `h`, but you can read more about them by running `git ls-files --help`.

`cut -c 3-` filters the first two characters of each of the output lines: `h` followed by a space, in the example.

With these combined, the output should resemble the following.

Listing 3.12 Output: assumed-unchanged files

```
# git ls-files -v | grep '^[hsmrck?]' | cut -c 3-
01-IntroducingGitInPractice.asciidoc      <— Assumed-unchanged file
```

HOW DO PIPES, GREP, AND CUT WORK? Don't worry if you don't understand quite how Unix pipes, `grep`, and `cut` work; this book is about Git rather than shell scripting, after all! Feel free to use the command as is, as a quick way of listing files that are assumed to be unchanged. To learn more about these, I recommend the Wikipedia page on Unix filters: http://en.wikipedia.org/wiki/Filter_(Unix).

Technique 28 Stopping assuming files are unchanged

Usually, telling Git to assume there have been no changes made to a particular file is a temporary option; if you have to keep files changed in the long term, they should probably be committed. At some point, you'll want to tell Git to once again monitor any changes made to these files.

With the example I gave previously in technique 26, eventually the Rails configuration file change I had been testing was deemed to be successful enough that I wanted to commit it so my coworkers and the servers could use it. If I merely used `git add` to make a new commit, then the change wouldn't show up, so I had to make Git stop ignoring this particular change before I could make a new commit.

Problem

You wish for Git to stop assuming there have been no changes made to 01-Introducing-GitInPractice.asciidoc.

Solution

1. Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git update-index --no-assume-unchanged 01-IntroducingGitIn-Practice.asciidoc`. There will be no output.

You can verify that Git has stopped assuming there were no changes made to 01-IntroducingGitInPractice.asciidoc by running `git ls-files -v | grep 01-Introducing-GitInPractice.asciidoc`. The output should resemble the following.

Listing 3.13 `--no-assume-unchanged` output

```
# git ls-files -v | grep 01-IntroducingGitInPractice.asciidoc

H 01-IntroducingGitInPractice.asciidoc
```

Git will notice any current or future changes made to 01-IntroducingGitInPractice .asciidoc.

Discussion

Once you tell Git to stop ignoring changes made to a particular file, all commands such as `git add` and `git diff` will start behaving normally on this file again.

3.1 Summary

In this chapter, you learned the following:

- How to use `git mv` to move or rename files
- How to use `git rm` to remove files or directories
- How to use `git clean` to remove untracked or ignored files or directories
- How and why to create a .gitignore file
- How to (carefully) use `git reset --hard` to reset the working directory to the previously committed state
- How to use `git stash` to temporarily store and retrieve changes
- How to use `git update-index` to tell Git to assume files are unchanged

SOFTWARE DEVELOPMENT

Git IN PRACTICE

Mike McQuaid

Git is a source control system, but it's a lot more than just that. For teams working in today's agile, continuous delivery environments, Git is a strategic advantage. Built with a decentralized structure that's perfect for a distributed team, Git manages branching, committing, complex merges, and task switching with minimal ceremony so you can concentrate on your code.

Git in Practice is a collection of battle-tested techniques designed to optimize the way you and your team manage development projects. After a brief overview of Git's core features, this practical guide moves quickly to high-value topics like history visualization, advanced branching and rewriting, optimized configuration, team workflows, submodules, and how to use GitHub pull requests. Written in an easy-to-follow Problem/Solution/Discussion format with numerous diagrams and examples, it skips the theory and gets right to the nitty-gritty tasks that will transform the way you work.

What's Inside

- Team interaction strategies and techniques
- Replacing bad habits with good practices
- Juggling complex configurations
- Rewriting history and disaster recovery

Written for developers familiar with version control and ready for the good stuff in Git.

Mike McQuaid is a software engineer at GitHub. He's contributed to Qt and the Linux kernel, and he maintains the Git-based Homebrew project.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/GitinPractice

"Shows how to make your team's workflows simpler and more effective."

From the Foreword by Scott Chacon, Author of *Pro Git*

"The best companion for your day-to-day journeys with Git."

—Gregor Zurowski, Sotheby's

"Ready to take off your Git training-wheels? Read this book!"

—Patrick Toohey
Mettler-Toledo Hi-Speed

"I learned more about how Git works in the first five chapters than I did in five years of using Git!"

—Alan Lenton, Arithmetica Ltd

ISBN 13: 978-1-617291-97-5
ISBN 10: 1-61729-197-8
53999
9 781617 291975

MANNING $39.99 / Can $41.99 [INCLUDING eBOOK]